プログラミング言語Dartの基礎

（第36版）

㈱クレス

2012年01月10日（初版）
2016年11月01日（第36版）

この解説書は、JavaとJavaScriptの経験を持つ技術者向けにDart言語の基礎を解説している。

- 第1章はDart言語の概説で、その特徴等が記されている。
- 第2章から第14章即ち前半は、Dart言語の基礎をDart言語仕様書に沿って説明する為のものである。不明な箇所はDart言語仕様書を参照されたい。また構文の記述は仕様書の「2. 本仕様書の表記(Notation)」の節を参考にして頂きたい。
- 第15章はDartで書かれたプログラムの実行法を解説している。
- 第16章はDartによるアプリケーション開発に必要なパッケージ・マネージャを解説している。
- 第17章と第18章はDartにとって特徴的な非同期処理と並行処理が解説されている。
- 第19章以降の後半は主としてサーバ・サイドに関連した内容となっている。

本書ではAPIに関しては、サーバ・サイドに必要なコア、アイソレート、非同期及びIOのライブラリ以外特に説明していない。クライアント・サイドに関してはDart API ReferenceやA Tour of the Dart LibrariesなどのAPIに関する公式ドキュメントが参考になるが、現在既にクライアント・サイドに関する多くの解説書が存在している。

DartはJavaとJavaScriptの経験を持つ技術者には馴染みやすい言語である。オブジェクト指向の要素としてクラスとインターフェイスを使っている。デフォルトではJavaScriptと同じ動的な型づけを使っているが、静的な型づけも可能で、複雑化するアプリケーションをより構造化でき、またより厳格にチェックできるようになる。

Dart言語はその開発に沿って仕様書及びAPIも随時バージョン・アップされてきた。従ってそれに合わせこの資料も改版してきている。ここに示したサンプル・プログラムたちはこの資料の改版時点で動作が確認されたものである。なお標準化に関してはECMAは2014年6月末の第107回全体会議で正式に1.3版を承認している。

なお、本資料に添付されているサンプル・プログラムは、Windows（Windows 7以降）での動作が確認されている。これらのプログラムを商用に使う場合の結果に対しては当社はその責任を負わない。即ちMITライセンスである。これらのプログラムのダウンロードと実行に関しては、「本資料に含まれているプログラムのダウンロード」の章に示してある。

最近の改版内容

第20版：M4版の組み入れ。
第21版：HTTPSサーバの章を追加。
第22版：アイソレートの章をAPI変更に伴い書き換え。ファイル・アップロードの章を追加。
第23版：パッケージ・マネージャの章のWeb UIをPolymerに変更、およびAPI変更を組み入れ。
第24版：幾つかの部分的な追加、またDBアクセスの項をパッケージ・マネージャの章に追加。
第25版：概要の章の「標準化」の節を更新、HTTPサーバの章に「マルチアイソレート化」の節を追加。
第26版：Shelfパッケージの章を追加
第27版：HttpServerのAPI強化を組み入れ。後回しのロードの節を追加
第28版：RESTfulの章を追加。またpubに厳格に適合するよう一部のサンプル・アプリの構造を変更
第29版：第25章「Googleのウェブ・サービスの為のAPI (googleapis)」を追加
第30版：第26章「Google App EngineでDartを走らせる」を追加
第31版：10.18節「Dart Fletch」を追加。Dartiumに関する注意を追加
第32版：Dartチームの方針変更に伴いIDEをDart EditorからIntelliJに変更
第33版：第20章「HTTPSサーバ」をAPI変更に伴い変更。本文の一部見直し。コードをIDE表示に合わせる等
第34版：第18章「並行処理」の章の一部補足、およびその他の修正
第35版：第11章「文」の章にジェネレータ関数の記述を追加。第16.2節「Pubを試してみる」のなかのPolymerをAngularDartに入れ替え。IDEまわりの修正（Dart Editorの残存物をIntelliJに置き換え）、OpenWeatherMapのポリシー変更の対応、その他

目次

第1章　概要

Dartは大規模（Googleのアプリ規模）で構造化されたウェブ・アプリーケーション（サーバ・サイドとクライアント・サイドの双方において）に対応可能なプログラミング言語である。Dartの開発の中心になっているのがChromeブラウザのV8エンジンを担当したLars Bak（ラース・バーク）などであり、従ってDartはJavaScriptのV8エンジンの経験がベースになっている。またLars Bak, Kasper Lund, Gilad Bracha, Eric Claybergを含むDart言語の開発チームの多くはSmalltalkの開発に関わってきた経験を持っていることもDartを馴染み安い言語としている。Lars BakとJava言語仕様の作成者の一人であるGilad Bracha（ジラード・ブラーカ）が2011年10月にこの言語を最初に公式に発表したときのプレゼンテーションでは、この言語のことを「シンプルで意外性のないオブジェクト指向プログラミング言語」(A simple and unsurprising OO programming language)だと述べている。即ち：

- Dartでは総てがオブジェクトである
- クラス・ベース、単一継承、インターフェイスつき
- 静的な型づけはオプショナル
- 真の構文スコープ
- 単一スレッド。並行処理はアイソレートで実現可能
- なじみのある文法

が特徴だとしている。

シンプル性の追求はGoogle社の社風であり、これは言語及びAPIの仕様からも良く窺われる。DartはJavaに比べるとずっと簡潔(succinct)である。シンプルな言語にすることは次のような利点をもたらすとLars BakがこのチームのKasper Lundがともに行ったGoogle I/O 2013でのプレゼンテーションで述べている：

- 高速化が可能
- 生成されるコードが小さくなる
- 性能の予測可能性が高まる
- メモリ利用率があがる

速度に関して言えば、Dartで書かれたコードをDartのVMで実行させるとV8よりも約1.7倍高速（ベンチマークDeltaBlueで）となっている。また一時的ではあるが、2013年5月時点でベンチマークDeltaBlueでは既にDartで書かれたコードをdart2jsでJavaScriptに変換して実行させるほうがJavaScriptで書いたコードをV8で実行させるよりも早くまでなっている。しかしV8の性能も改善されており、現在dart2js生成のJavaScriptを5%ほど上回る状態で推移している。最近の詳細は「Dart Performance」を見て頂きたい。

Dartは開発当初はDashと呼ばれていた。しかしながら混乱を生じる可能性があった為、dashの「突進する」という意味を同じく持っている動詞としてdartが選択されたものである。

2014年6月時点でDart言語はECMA標準となっている。

2015年3月時点でDartはGoogle内ではGoogle Ads、Google Fiber、Google Express、及びGoogleの社内セールスなどのチームで採用されている（参照）。Google以外ではDGLogik、Easy Insure、Soundtrap他がある（ここをあるいはここを参照）。

Dartの型づけ

動的型づけにオプショナルな静的型づけを付加した言語は初めてでもあり、また議論を呼ぶところでもある。Dartのチームは静的な型づけを少し付加しているが動的型付けの利点をプログラマが十分活用できるという目標は崩していないという。

ユーザが静的型づけを使うことで:

- 作られたコードが理解しやすくなる
- IDE(統合開発環境)などのツールがそれを使って便利な機能を提供できる
- 静的チェッカ(static checker)が警告を出し、開発時には実行を止める
- dart2jsクロス・コンパイラがこれを使って性能をあげることができる

といった、より高度のアプリケーション作成の為のメリットが生じる。

実行時におけるDartの型チェックに対する取り扱いは次のようになる(コンパイル時の文法的チェックなどや実行時のオーバーフローなどの例外とエラーのチェックは他の言語と同じ):

実行時(checked mode: **チェックド・モード**、開発時に使う)	**静的型エラー**:静的型チェッカが**警告**を出すが、実行は停止しない **動的型エラー**:渡される値の動的な型をチェックをしてエラーがあれば**例外を発生**する
実行時(production mode: **運用モード**)	**静的型エラー**:型アノテーションは進行に影響を与えない(**無視**される) **動的型エラー**:渡される値の動的な型をチェックをしてエラーがあれば**例外を発生**する

即ち:

- プログラマはこれまでどおり型アノテーションを全く付さないでコードを書くことが可能である。

- 実行時には型アノテーションを付けたコードは静的チェッカがチェック・モード(checked mode)で代入(あるいは関数へのパラメタとしてのオブジェクトの渡し)に対しチェックしてくれ、一部の違反には例外を発生させる。一方運用モード(production mode / unchecked mode)では静的型アノテーションは処理に影響を与えないし、処理速度に影響しない。最高の処理速度が得られる。運用モードでは静的型づけを尊重はするが違反があっても極力処理を止めずに継続させる。

- しかもDartがコンパイル時に行う静的型チェックはJavaのそれよりはずっと「楽観的(optimistic)」である。従ってDartチームはいわゆる"type checker"ではなくて”static checker"だと強調している。例えば静的型がObjectの値を静的型Stringの変数に代入することを静的型チェッカは許している。正しいということは保障されないがもし実行中の値がたまたま文字列であったら、あるいはプログラマがそれを想定していたら構わないかもしれない。Dartのチームはこれを「型ヒューリスティックス(type heuristics)」と呼んでいる。

言語仕様書18.1節にはDart言語の実装に関し次のように書かれている:

- プログラムPに対し静的チェッカを走らせることは、Pのコンパイルと実行の為には要求されていない。
- プログラムPに対し静的チェッカを走らせることは、静的な警告が発生するかどうかに関わらず、Pの成功裏のコンパイルを阻止してはならないし、Pの実行を阻止してはならない。

詳細は仕様書及びDartの型処理と型チェックの節を見て頂きたい。

なお「チェックド・モード」という言葉はチェックされていないと安全でないという意味にとられるので変更すべきだ

という意見がGoogle内部である。

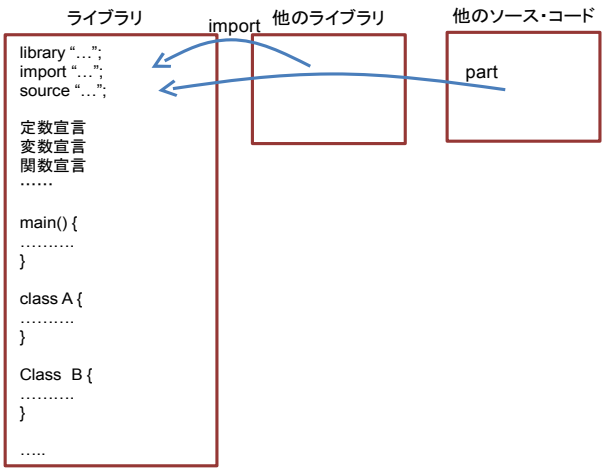
なおDartのVMのコンパイラはJIT（just in time：実行時）コンパイルで、例えば実行中ある関数を呼び出すときにその関数のコンパイルを行う。但しそのコードの構文解析は実行に先立ちアナライザ(dartanalyzer)が行う。従って型アノテーションとチェックド・モードを活用して開発段階に於いて完全にバグを取っておかないといけない。

中間コード

DartはJavaのように他の言語でも使えるバイト・コード仮想マシン(bytecode VM)の為の中間コードを生成しそれをインタープリタが実行するということはしない。プログラマが書いたコードはそのままDartに特化した言語仮想マシン(language VM)により一旦中間表現に変換されるものの、その都度コンパイルされ実行される。Dartのチームの説明によれば、VMをDartに特化させることで高い性能を得ることができる。また中間コードを持たないことでセキュリティが上がる。更に将来ブラウザにこれが実装されれば、Dartのプログラム開発はブラウザ上で簡単に出来てしまう。更にはそのプログラムを走らせたままで手を加えるライブ編集(live editing)も考えているとのことである。

Dartのコードの構成

Dartのコードは次のような構成になる：



必要なライブラリをまとめたものはパッケージと呼ばれる。

dart:で始まる基本ライブラリ以外のライブラリたちは第三者も登録できる大規模なもので、pubというパッケージマネージャの管理対象となる。

プログラムはあるライブラリ単位で管理される。ライブラリはトップ・レベルの定数、変数、関数、クラス、インターフェイスなどのコレクションである。他のライブラリの要素をアクセスしたいときはそのライブラリをインポートする。Dartには有用なAPIがdart:core、dart:html、あるいはdart:ioといった多くのライブラリとして用意されている。既に組み込まれているコア・ライブラリ（dart:core）はインポートの必要は無い。

ライブラリはまたカプセル化の単位であり、プライバシはライブラリ単位で守られる。Dartのプライベートな要素はその識別子の頭にアンダー・スコア('_')を付す。そのような要素はそのライブラリの中でのみ可視となる。

識別子は英数文字(大文字と小文字は識別される)およびアンダスコア('_')を使用する。その他の文字は使用してはいけない。

Dartでは変数、関数、及び型のためには単一の名前空間が使われていることに注意のこと。セッタ/ゲッタ以外で同じスコープ内に同じ名前のものが宣言されているとコンパイル時エラーとなる。

libraryはライブラリを定義し、指定したURIがアクセス可能であればそこに置く。importはURIで指定した他のライブラリをアクセス可能にする(その要素を現在のスコープ内に置く)。partはURIで指定したソース・ファイルを現在のライブラリに含める。例えば:

```
// 'dart:isolate'をインポートし、その要素には'isolate.'というプレフィックスを付けてアクセス
import 'dart:isolate' as isolate;
// 'dart:html'をインポートし、その要素は現在のライブラリ内と同じようにアクセス
import 'dart:html';
```

プレフィックスを付けることで、名前空間がライブラリの中で完結していることによる名前の衝突を解決できる。

ライブラリのトップ・レベルに置かれるのはクラスに含まれない定数宣言、変数宣言、関数宣言、クラス定義、インターフェイス定義などである。これらはこのライブラリ内のどこからでもアクセス可能である。例えばそこに含まれているクラスのオブジェクト内からもアクセスできる。Dartのコア・ライブラリには3つのトップ・レベル関数であるvoid **print**(Object obj)、bool **identical**(Object a, Object b)およびint **identityHashCode**(Object object)が存在している。従ってこれらの関数はどこからでも可視である。

トップ・レベル関数である**main**()はDartがこのコードを実行の際引数なし(またはコマンド行引数つき)で呼び出すものであり、これがないと実行時エラーになる。

具体的な例(初期のDartチームの解説書にあったもの)を示す:

code_01_1.dart

```
final String c = 'Time: '; // トップ・レベル定数
var v;                     // トップ・レベル変数
void f(String p) {         // トップ・レベル関数
  print(p);
}

main() {             // トップ・レベルのmain関数
  C myC = new C();
  v = new DateTime.now();
  f(myC.m());
}

class C {            // クラス宣言
  String m() {       // メソッド
    return '$c $v'; // トップ・レベルの要素へのアクセス
  }
}
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

このコードの詳細は説明を省くが（この概説書をひととおり読んでから見直して頂きたい）、次のような構成になっている：

- このプログラムのトップ・レベルにはcという定数、vという変数、fという関数、Cというクラスがある。
- Cというクラスにはmというメソッドのみがある。このメソッドはcとvから作った文字列を返す。
- main()の中では次の操作が行われている：
  - C型のインスタンスmyCを生成する。
  - 変数vにDate型の現在の時間を代入する。
  - myCオブジェクトのm()で取得したString型の値を引数にして関数fを呼ぶ。

トップ・レベルにあるc、v、fは関数main()とクラスCから可視である。

Dartがいう関数とは関数宣言に加えてクラス内のメソッド、セッタ、ゲッタ、コンストラクタ、あるいは関数リテラルをいう。トップ・レベルで宣言されている関数はそのライブラリのどこからも可視であり、これをライブラリ関数とも呼ぶ。メソッドは関数と同じ形式ではあるがクラス定義内で定義されているものをいう。

真の構文スコープ

なお可視性に関して付記すると、Dartの特徴のひとつに挙げられている「真の構文スコープ(true lexical scoping)」とは、Javaと同じように各識別子がそれが宣言されたブロック内でスコープ付けされているという意味である。すなわち、波括弧で括られたブロック{...}を見ればその変数のスコープを知ることができる。トップ・レベルの変数は従って何処からでも可視である。

例えば次のコードを考えてみよう：

```
var foo = 'top_level';
void bar() {
  if (true) {
    var foo = 'inside';
    print(foo);  // 内部のfooが可視
  }
  print(foo);  // 外部（トップ・レベル）のfooが可視
}
main() {
  bar();
}
```

この例ではトップ・レベルとif文の中の2か所でfooという変数が定義されている。従ってmainメソッドでこれを実行すると

```
inside
top_level
```

と出力されることになる。

Dartが真の構文スコープだという例として、繰り返しループの中で定義される変数が各繰り返しの中でその都度きちんとそのループ変数に対応したオブジェクトが生成されることがある。以下は関数が要素であるfunctionsという配列の例である：

```
main() {
 var functions = [];
 for (var i = 0; i < 3; i++) {
  functions.add(() => i);
 }
 functions.forEach((fn) => print(fn()));
}
/*
0
1
2
*/
```

ここではfunctions[0]には(){return 0}という関数がセットされる。ループ変数のiはそのループの外部からは不可視である。

一方JavaScriptの場合は:

```
var functions = [];
for (var i = 0; i < 3; i++) {
  functions[i] = function() { return i };
}
functions.forEach(function (fn) { console.log(fn())});
```

functions[0]にはfunction () {return i}という関数がセットされる。従ってその結果はi=3(可視)なので

```
3
3
3
```

と表示されてしまう。

クラスの組み立て

Dartではクラスとインターフェイスが分離しておらず、基本的にはクラス・ベースである。クラスを宣言することは暗示的なインターフェイスを宣言することにもなる。クラスたちをextends、implements及びwithキーワードを使って継承あるいは実装して構成することもできる。

詳細は本資料の各章(あるいは節)を読んで頂きたい。

下表はこれらをまとめたものである:

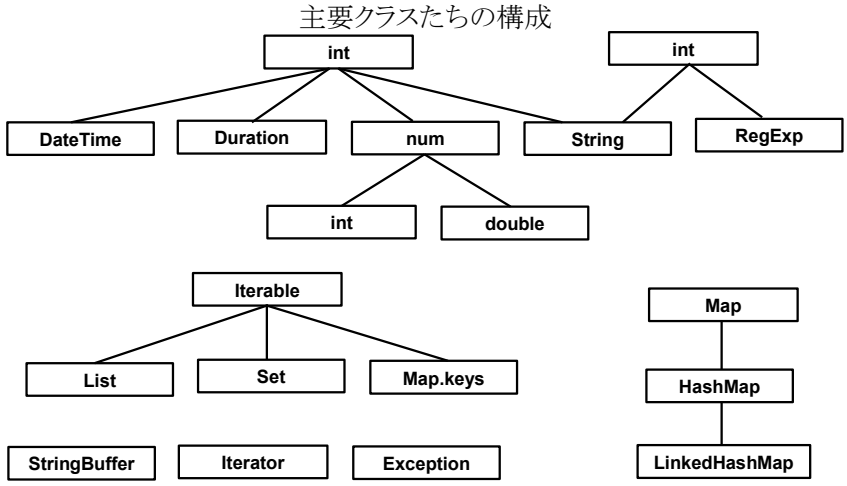
	クラス	インターフェイス	抽象クラス	ミクスイン (M3から実装された)
宣言	**class** Javaと同じ	**class** クラスを定義すると暗示的インターフェイスももたらされる	**abstract class**	**class** 暗示的に定義される
メンバ	未実装(即ち抽象メソッド)も許される	各メソッドは実装されていてもいなくても良い	少なくともひとつの未実装(抽象)メソッドを持つ	通常各メソッドは総て実装されている
使用時のキー	**extends**(継承するクラ	**implements**	**implements**(実装する	**with**

ワード	スの未実装メンバの総てを実装のこと） **new**（直接インスタンス化）		抽象クラスのメンバの総てを実装のこと） **extends**（継承する抽象クラスの未実装メソッドを実装のこと）	
注意		• 実装するインターフェイスのメンバの総てを実装のこと • 複数のインターフェイスを実装できる	• そのまま**new**でインスタンス化できない。インスタンス化出来るようにするにはファクトリ・コンストラクタが必要（APIで使われている） • 通常は暗示的なインターフェイスで足りる	• そのままインスタンス化されることを意図していない • ミクスインそのものはフィールドとコンストラクタを持たず、またそのスーパークラスは**Object**で、スーパークラス呼び出しを含まない

APIライブラリとそのクラス構成

Dartのコア APIライブラリ（即ちdart:core）は基本的な組込みAPIが集められているので、ひととおり知っておく必要があろう。DartではObjectクラスの説明に書かれているように、関数、変数など総てのものがオブジェクトとして扱われている。これはJavaのようにプリミティブな型を持っている言語とは異なる。このObjectがDartのクラス階層のルートとなっている。Objectは識別のためのハッシュコードと実行時の型を取得するruntymeTypeという属性を持っており、またtoString、型が同じかを調べるidentical、及びこのオブジェクトにそのようなメソッドが無いときの為のnoSuchMethodのメソッドが総てのオブジェクトに対し適用できる。本資料はAPIの詳細は扱わないが、最も良く使われる幾つかのクラス及び抽象クラスがどのような構成になっているかを下図に示す。

主要クラスたちの構成



なおString、MapやListなどが抽象クラスとして用意されているにもかかわらず、new Date()のようにあたかもクラスのようにインスタンス化できるのは、各々のファクトリ・コンストラクタが用意されているからである。

以下は現時点でGoogleが用意しているAPIライブラリのなかの主要なものである。

ライブラリ	概要

dart:core	コアとなるクラスと抽象クラスたち
dart:async	エイシンクと発音。非同期処理関連のクラスたち。2013年1月に追加された
dart:math	従来coreに含まれていたMathを2012年8月に独立させた（これまでのコードにimport('dart:math', prefix: 'Math')を付加するだけで良い）
dart:isolate	並行処理の為のアイソレートに関する関数及びクラス
~~dart:dom~~	2012年5月にdart:htmlに統合された
dart:html	HTML5ベースのDOM操作のクラスたち。クライアント用で**サーバ側（Dart VM）には実装されない。またクライアントのアイソレートではDOMアクセスは出来ない。**
dart:io	サーバ側に必要なネットワーク・インターフェイス。スタンドアロンのDart VM用で、**クライアント側（Dartium）には実装されない**ことに注意
dart:convert	文字コード、HTML、あるいはJSON処理等
~~dart:uri~~	IETFのRFC3986にもとづいたURIの処理だが2013年5月にdart:coreに移された
unittest	単体テストのライブラリ。2012年6月に追加された。パッケージ・マネージャ(pub)を使って使用される。

ライブラリの使い方はA Tour of the Dart Librariesという資料が参考になろう。

DartのAPIの特徴として次のものがあげられよう：

- HTML5対応：
  dart:htmlライブラリはChromeの蓄積を踏襲した巨大なもので、クライアント側で不自由を感じることは無いだろう。
- 非同期処理の為の多くの機能：
  Dartは単一スレッドではあるが、殆どの処理はコールバック関数で記述された非同期処理で行われる。従ってスループットは大きい。Futureインターフェイスがその基本になっている。更にStreamというインターフェイスは連続したデータやイベントを取り扱う為の基本的なツールを提供している。
- アイソレートによる並行処理：
  必要ならIsolateインターフェイスを使ってマルチ・スレッド動作をさせることができる。各アイソレートが自分自身のヒープとスタックを持っており、アイソレート間はリソースを共有しないので、スレッド安全の問題が無い。アイソレート間はメッセージ通信で情報が交換される。
- 開発のツールとしてパッケージ・マネージャ(Pub)や単体テスト（unittest）などが用意されている：
  Pubには多くの有用なパッケージが追加されてきている。特にAngularDart、ロガー、Googleアプリへのアクセス、DBドライバ等必要なものが揃ってきている。
- サーバ側の為のライブラリ（dart:io）が強化されてきている：
  サーバ・サイドにとってDartは魅力的な言語になりつつある。

DartコードとJavaScriptコードの比較

2012年2月1日にGoogleは比較のサイト(Dart Synonym)を立ち上げたので参考にして頂きたい。JavaScriptに馴染んだユーザたちには有用であろう。なお、Dartで書かれたプログラムはそのままDartのVM（DVM：Dart仮想マシン）が実行するが、dart2jsと称するクロス・コンパイラでJavaScriptのコードに変換し、通常のブラウザで実行させることも可能である。

Dartコードの記述

推奨されるDartのコードの書き方は、DartチームのBob Nystromが書いたDart Style Guideを見て頂きたいが、簡単にポイントを以下に示す。

- インデントはスペース2文字であり、タブは使用しない。
- 一行の長さを80文字に留める。
- 左鍵カッコはそれが続いている行に含める。例:
```
class Foo {
  method() {
    if (true) {
```
- 2項や3項演算子の前後、及びカンマの後にスペースを置くが、単項演算子の前後には置かない。例:
```
a = 1 + 2 / (3 * -b);
c = !condition == a > b;
d = a ? b : object.method(a, b, c);
```
- (, [, 及び {のあと、あるいは),], 及び }の前にはスペースを置かない

Dartにおいては以下の付名規約が一般的である:

- コンパイル時定数変数の名前は小文字を使わない。もしそれらが複数の単語で構成されているときは、それらの単語をアンダスコアで分離させる。例: `PI, I_AM_A_CONSTANT`
- 関数(ゲッタ、セッタ、メソッド、及びローカル及びライブラリ関数)と非定数変数の名前は小文字で始まる。もしそれらが複数の単語で構成されているときは、各単語(最初を除く)は大文字で始まる。それ以外には大文字は使わない。例:`camlCase, , dart4TheWorld`
- 型(クラス、型変数、及び型エイリアス)の名前は大文字で始まる。もしそれらが複数の単語で構成されているときは、各単語は大文字で始まる。それ以外には大文字は使わない。例:`CamlCase, Dart4TheWorld`
- 型変数の名前は短くする(一文字が好ましい)。例:`T, S, K, V, E`
- ライブラリ、またはライブラリ・プレフィックスの名前は決して大文字を使わない。もしそれらが複数の単語で構成されているときは、それらの単語をアンダスコアで分離させる。例: `my_favorite_library`

標準化(Standardization)

ECMA 第1版

Dart言語は下記のようにECMAで標準化が行われ、2014年6月末に承認された。以下はその経過である:

2013年12月13日にGoogleはECMAがDartの標準化のための新しい技術委員会TC52を設立したと発表した。

GoogleはこのTC52を介してウェブのコミュニティと協働しこの言語の発展を推進すると述べている。TC52の委員

長はGoogleデンマークのAnders Thorhauge Sandholmである。

2014年3月13日に開催されたTC52会合の報告として、言語仕様書担当のGilad Brachaは次のように報告している:

- 我々は1.2仕様書に少し手を加えたものをこの委員会に標準化のための言語仕様書案として提出する計画である。この委員会が承認すれば、6月末までに公式の標準ができる。
- 次回会合はデンマークのAarhusで5月1日に開催される。
- 彼としては年末までにenums及びdeferred loadingを付加した改定版をまとめたいと考えており、これらの機能提案を5月の会合で提出する。

2014年5月2日にGilad Brachaは次のように報告している:

- TC52は正式に現在の仕様(1.3版)を承認した。
- 6月末には批准されよう。
- 1.3版仕様はECMAのサイトで取得できるが、これはDartのサイトにある1.2版にECMAの書式を先頭に付加しただけのものである。
- TC52の別の会合ではenums、deferred loadingおよび非同期対応の付加に関して議論した。これらの機能追加の批准は時間がかかり12月になりそうである。
- 次回会合は7月1日および9月16日に仮設定(in Zurich)。

2014年7月2日にGilad Brachaは次のように報告している:

- ECMAは6月末の第107回全体会議で正式に1.3版を承認した。
- TC52の第3回会合ではenums, deferred loading, async, and minor bug fixesが討議された。
- 次回は9月16日にスイスで開催される。

Dartチームからの発表はここにある。
2014年8月現在提案されている変更事項案はAsyncWaitおよびEnumsである。

ECMA 第2版

2014年11月21日に1.6版のものがDraft: Dart Programming Language Specification, 2nd Editionとして公開されている。これはTC52で承認されたものだが、まだ全体会議では未承認のものである。第2版では以下のものが追加されている:

- 列挙型 (enum)　1.8版で実装済み
- 非同期関係(async, awaitほか)　部分的に1.8版で第1フェーズとして実装されている。
- 後回しのロード(import ... deferred as)　1.6版で実装済み

今後の会合予定:
第5回会合:　2015年1月14日(via Hangout)
第6回会合:　2015年3月23日(in Mountain View, CA, USA)

2014年12月11日にSapporoで開催された第108回全体会議でECMA-408 2nd editionが承認された。

ECMA 第3版

2015年1月30日にDartの広報役のSeth Laddが、1月14日に開催されたEcma TC52会合の報告をしている。

この会合では以下に記す幾つかの更なる改善事項が討議され、更に詳細を詰めることとした。今後数カ月かけて提案がなされる模様である。

- 設定可能インポート(Configurable Imports)

ライブラリによってはサーバでのみまたはクライアント（特にもし dart:htmlがインポートされてしまっているとサーバでの実行は出来なくなる）でのみ機能するものがある。条件付きコンパイルの類の機能が付加されると、どの環境に対しどのライブラリをインポートするかを選べるようになる。ただ条件付きコンパイルの機能は‘part’のメカニズムで実現されているとの指摘があった。

もうひとつの提案はパラメタ化されたライブラリ(parameterized libraries)というより複雑で強力なもので、ライブラリに型引数をもたせたり、他のライブラリを引数にしたり出来るようにする。

- 共用体(Union Types)

Dartの型システムは柔軟なので他の言語ほど共用体は重要でなくこれまで何度かこの提案は後回しにされて来ていた。しかしJavaScriptとの相互運用性が重要になってきており、JavaScriptと同じ機能を実現する為にDartにこの機能を待たせたほうが良い場合がある。また型TまたはFuture<T>を引数とする関数で、その関数がどちらかを判断させるケースが考えられる。

もうひとつの例として、異なった引数の数（アリティ）をもつ関数である。共用体の問題は構文解析で不一致を起こしがちなことで、これはtypedef宣言で制限できるが、総称体ではそうはいかない（ローカルなtypedefで解決できるかも）。静的型警告を出さない操作は共用体演算子の総てのオペランドに共通した副型にたいするものとなる。つまり総ての型で共有される操作に限られる。これにより一連の型関連コンパイルを避けることができる。

Dartの型システムにおける代入可能性規則（副型規則ではなく）ではある共用体を持った値がいろんなオペランド型で使えるので、Dartでは共用型は良く機能する。

- 総称メソッド(Generic Methods)

これは多相型メソッド(polymorphic methods)とも呼ばれる型引数をとるメソッドである。これには負荷がかかる型推論が必要で、Dartでは現在対応していない。Dartに導入する際は型推論を使わないよう制限される。即ち総ての型引数は明示的に指定するか、コンパイラが<dynamic..dynamic>として暗示的に定められるようになる。

- 一般化されたティアオフ(Generalized Tear-Offs)

属性抽出メカニズム（クロージャ化またはティアオフとも呼ばれる）では現在幾つかの問題がある：現在の文法では演算子、ゲッタ、セッタ、コンストラクタでは機能しないし、更に‘a.m’が属性抽出やゲッタ呼びだしとして実行されるべきかどうかが実行時に分からないので最適化ができない。提案されている文法では‘.’の代わりに‘#’を使ってレシーバとセレクタを切り分けるものである。互換性の為これまでの文法も共存させる。

- Null対応の演算子(Null-Aware Operators)

これは以前から要求されていた機能である。a?.b演算子(Elvis operator)ではaがnullのとき結果はnullになり、a??bではaがnullのときデフォルトとしてbの値をとり、また a ?= bではaがnullのときに限りaにbの値が入る。

- 型プロモーションの改善(Improvements to Type Promotion)

これは純粋に静的型システムの問題で、ある変数の型テストが先行した制御フローの中で存在しているときの型づけに関する。現在型プロモーションに対しては幾つかの規則をかけてきている。これは窮屈だという意見があるので緩和を検討している。

- その他のマイナーな変更

    - ‘await throw’を throw immediatelyに変更
    - ‘noSuchMethod’が宣言されているクラスのオブジェクトたちの型チェックの使用に関する規則は一般化が必要
    - 'main'という名前はトップ・レベルの関数以外にも使えるようにする
    - docコメントがインポートされたスコープにアクセスできるようにする

報告によれば、2015年6月17日に第109回ECMA総会で ECMA-408 3rd editionが承認された。

主たる追加事項は:

- Null対応の演算子(Null-Aware Operators)
- 一般化されたティアオフ(Generalized Tear-Offs)

である。

Null対応の演算子は実装作業中であり、ティアオフはその後実装される予定である。

参考リンク

公式サイト:
- Official Dart Homepage

言語仕様:
- 言語仕様書案
- 日本語翻訳版

API参照
- Dart API Reference

プログラマーズ・ガイド
- Programmer’s Guide
- A Tour of the Dart Language
- A Tour of the Dart Libraries
- 日本語の言語ガイド

ニュース:
- The Dartosphere

一般的なハウツー的な質問は:
- StackOverflow

ディスカッション・フォーラム(全般):
- Dart Misc

ディスカッション・フォーラム(サーバ・サイド):
- Dart Server-side and Cloud Development

バグ報告:

- https://github.com/dart-lang/sdk/issues

VMに興味がある人は:
- nothingcosmos wiki（日本語）

コード・サンプル集:
- Dart Code Samples

HTML5 UI関連サンプル（作業中）:
- Demo: http://yohanbeschi.github.com/pwt_proto.dart/
- Code: https://github.com/yohanbeschi/pwt_proto.dart

開発の経過

M1仕様

Dartの仕様書は頻繁に更新されており、Dartチームは何時1.0版としてリリースするかを発表することを拒んできていた。彼らはDartをより革新的で魅力的なものとすることに専念していた。しかしながら2012年7月にMilestone 1（M1）版における主要変更内容を発表している。スケジュール管理のM1という言葉を使い始めたことは、リリース版にむけ作業がかなり進行していることを示唆している。これらの内容は2012年9月から（仕様書0.11版から）実施されている（最終的にはこの0.11版に一部を追加した0.12版からで、この版にはM1版と明記されている）。そして2012年10月16日にこのチームのLars BakがM1バージョンが使用可能になったと発表した。エディタの対応バージョンはDart Editor build 13679 (M1)からである。今後は大規模な変更の導入よりも安定化と精錬化に集中すると彼は述べている。

なおここでのM1はDartのディスカッション・ルームでのバグ等の問題処理のスケジュールのM1と基本的には一致している。

2012年10月16日にDartチームが最初の安定版が出たと発表したと報じられた。これはM1版のことを意味し、前述のLars Bakの発表、及びこのチームの広報役のSeth Laddが最初の発表から1年経過したことを受けSDKがオープンな場のもとでより安定したバージョンが得られるようになったと発表したことがそのようにメディアに受け取られたものである。これらの発表は、10月1日のMicrosoftからのTypeScript発表に対抗する意味もあろう。

M2仕様

2012年12月13日に改訂されたDart言語仕様書は”Draft Version 0.20, M2 release”と記されている。仕様書上の主たる変更箇所は:

- クラス・メンバに対するabstract修飾子を削除
- ミクスインの導入（但し実装は未だされていない）と予約語withとwith句の追加
- NullPointerError(NPE)の廃止

なのである。詳細はSeth Laddが書いているメモを見られたい。APIでは簡素化がされているが、言語仕様ではミクスインの追加以外はM1のような大きな変更は存在しない。

ミクスインはサブクラスによって継承されることにより機能を提供し、単体で動作することを意図しないクラスである。ミクスインはメソッドが実装されているインターフェイスだともいえる。インターフェイスの章で説明するが、Dartでは各クラスがインターフェイスも持つようになった（従って明示的なインターフェイスは存在しなくなった）。従ってあるクラスをextendsキーワードにより継承、implementsキーワードにより実装、及びwithキーワードによりミクスインできるようになる。

M3仕様

2013年2月19日に改訂された言語仕様書は仕様書上のM3版である。仕様書上は文字列としてUnicode Normalization Form C（Unicode正規化形式C）に正規化することをあきらめ、Javaなどと同様UTF-16を採用したことが注目される。

M3ではミクスインが実装されているものの、現時点では未だ完全ではない。

APIでは連続したイベントあるいはデータの非同期処理の為のStreamというインターフェイスの導入がある。Dartチームは既存のAPIをこのコンセプトを組入れるべく大規模な改正作業を行った。その為幾つかの混乱を生じたものの、最近は集約してきており、Dartの特徴のひとつとしての機能となっている。その他Collectionの簡素化、Futureの簡素化などもこの改正に含まれる。

M4仕様

2013年4月16日にM4と称するエディタとSDKがリリースされた。但し仕様書は4月22日にリリースされた。

- Core、collection、及びasyncのライブラリが安定化された。今後はこれらのライブラリでは大規模変更はなされない。
- dart2js及びDart VMともに性能がアップした。
- Dart Editorは全く新しい解析エンジンが実装された。
- クラスがミクスインとして使用可能となった。

その他の大規模変更に関しては、List of Last Minute M4 Breaking Changesという記事を参考にされたい。

ベータ版 (M5)

2013年6月19日にGoogleはベータ版のリリースを発表している。主要な追加・変更は次のとおりである：

dart2js

- dart:typed_data対応を付加
- ユニオン型を使って型推論を改善するとともに副作用に常に注意している
- 複数のクラスにミクスインされたコードの共有を導入
- 総称型と取扱のカバーを拡大
- 性能改善
  Richardsで20%高速化、DeltaBlueで10% 高速化、Tracerで8%高速化
- ミラー対応で大きな進捗（作業中）

Dart VM

- M4に対しDeltaBlueで40%高速化
- M4に対しTracerで33%高速化
- 完全なSIMD加速
- 初期のスナップショットのサイズを縮小
  新規アイソレートの立ち上がり時間を改善
- デバッグ機能の安定化

エディタ

- エディタとSDKに20%高速の新しいアナライザを使用
  - これまでのアナライザを削除
- エディタ内でのクイック解決とリファクタリングの数を増やした
- コードたたみこみのはクラスとローカル関数も含めたかたちで復活
- 発生マークを復活
- Pub Deployコマンドを追加
- Code Completionで多くの強化

Dartium内でのWebGLが大きく性能改善（dart:typed_dataライブラリをtyped arraysに移した）

ライブラリ

- dart:async
  - Streamのバグ修正（大きな変更はキャンセルされたストリームへの再加入を認めなくしたこと）
  - dart:core
  - Pattern (StringとRegExpなど)が使える箇所を拡大
- dart:uri
  - dart:coreに移すとともにAPIの改善と安定化を実施

dart:crypto

- 現在はdart:とするほどではないのでpubパッケージに移した

dart:io

- IPv6にも対応
- HTTPとWebSocketの高速化と安定化。HTTPは50%以上高速化
- HTTPボディ部処理対応を改善
- HTTP認証を改善し、またプロキシに対応

pub:

- バージョン追跡を付加、SDKバージョン制約を付加
- オフライン・モードの付加
- pub配備の初期バージョン

html:

- dart:mdv_observeライブラリを新規追加
- Typed_data

dartium: devtoolsを改善

- コンソールからDartオブジェクトを新規生成出来るようにした
- Dartのフィールドとゲッタにマウスを置くとその値が表示される
- クラッシュを減らした

Web UI

- 新規のポリマ・ブランチ

1.0版

2013年11月14日に仕様書がついに1.0版に改版された。またSDKもDart SDK version 1.0.0.3_r30187となった。Dartチームはこれに関し正式な発表をしたが、これは11月11-15日にベルギーで開催されたDevoxx Conference

に合わせたもので、シンボリックなものでしかない。2.0版まではもう大きな変更を加えないという意味らしい。

1.0版の発表前にdart:isolateライブラリの大規模変更がなされている。これが唯一残っていた大規模改正だったと担当技術者のFlorian Loitschが述べている。

11月18日開催されたLars Bakが主催するこのチームの中心人物たちによる毎週の会合(Language Meeting)では次のような議論がなされている:

- 自分たちは今後の変更に対してはより慎重にならねばならないこと。
- さらなる機能追加項目の多くについてはECMAでの標準化の作業の中で進めることになる。
- それまでは安定化に集中する。
- したがってこの会合は年内は開催しない。

最新版の取得

Dartチームは2013年12月からエディタとSDKをdev(最新版)とstable(安定版)の二つのチャンネルでリリースすることにした。

- Devのリリースはhttp://gsdview.appspot.com/dart-archive/channels/dev/release/latest/
- Stableのリリースはhttp://gsdview.appspot.com/dart-archive/channels/stable/release/latest/

から最新の版をダウンロードすることができる。

Dart SDKとDartiumのダウンロードは「Dart SDKとDartiumのダウンロード」の節に記されている。

その後のStable版

2014年1月16日に安定版1.1が発表されている。高速化がはかられ1.0版に比べてRichards benchmarkで25%改善されている。サーバ・サイド(dart:io)では大サイズのファイル対応、ファイル・コピー、プロセス・シグナル・ハンドラ、および端末情報などに対応している。またストリーミング用にUDPプロトコルも扱えるようになった。

その後の改版は以下のようになっている:

版	発表日	主たる内容
1.1	2014年1月16日	高速化
1.2	2014年1月23日	Dart Editorの改善が主
1.3	2014年3月18日	Stringの一部、Dart Editorの改善
1.4	2014年5月22日	Dart Editorの改善、UTF-8をベースとする 新しいMapの追加:MapBase, UnmodifiableMapBase, MapView, UnmodifiableMapView JsonEncoderのコンストラクタを追加 dart:ioでの変更等 • New named argument for ByteBuilder constructor: 'copy'. • HttpResponse - new bufferOutput property

		• HttpResponse - detachSocket added named writeHeaders argument • HttpClient: a number of helper methods have been added • Experimental: ServerSocket reference - Makes it possible to share a ServerSocket across multiple isolates. • See the discussion on the Dart misc@ group for more information. • Added WebSocket.fromUpgradedSocket and deprecated the default constructor.
1.5	2014年6月24日	dart:collectionではMapMixin, SetMixin, SetBaseが追加された dart:ioではHttpClientでmaxConnectionsPerHost属性追加
1.6	2014年8月26日	後回しのロード: Dart VMとdart2jsともに試験的にこれに対応させた。 dartライブラリ: dart:core • Pattern.allMatchesにオプショナルなstart引数追加 • DurationにisNegative, abs(), 演算子 -追加 • FormatExceptionにsource とoffset 属性追加 • Uri: の性能改善 • Uriインスタンスにreplaceメソッド追加 dart:async • Futureにstatic doWhile メソッド追加 • Future.forEach はZonesに連携 • Zoneにインスタンス・ゲッタerrorZoneを追加 dart:io • HttpServerのヘッダでセキュリティ強化 • HttpClientにボディ部圧縮の制御を追加 dart:typed_data • ByteBufferに多くのasメソッドを追加 エディタ • メモリ使用を大幅削減 • デバッガでのcollection型の表示改善 • デバッガでの Activationビューとインスタンス変数表示を改善 • Pub Package Selectionダイヤログに Reloadボタン追加
1.7	2014年10月16日	dart:async: • ZoneにerrorCallbackを追加 dart:io: • HttpClientのシャットダウン処理の変更 • HttpServer.autoCompressを追加 dart:isolate: • Isolate.spawnUriにオプショナルな引数pausedとpackageRootを追加 Isolate.spawnUri added two optional parameters: paused and packageRoot. pub: • 実行物たちを自分のPATH上に置ける dart2js: • 同じページ上に複数のDartアプリがあっても後回しのロードが効く ソフトウエア配布: • Debianリポジトリを設置。(see https://www.dartlang.org/tools/debian.html) • Macユーザ向けにHomebrew 対応

1.8	2014年11月19日	dart:collection • Set()に SplayTreeが追加された dart:convert • JsonUtf8Encoderを追加 dart:core • String.fromCharCodesコンストラクタにオプショナルなstartとendの引数を追加 dart:io • ClientとServerのTLSプロトコルでALPN 拡張に対応する
1.9	2015年3月27日	**言語の変更** • async, await, sync *, async* yield, yield*, await for に対応（参照） • enumに完全対応（参照） **コア・ライブラリの変更** **ポイント:** • 共有サーバ・ソケットの新モデル：Socket参照が不要、全プラットホームで実装 • 新規の高速regexpエンジン • Isolate APIがVMとdart2jsで完全対応した **詳細:** dart:async • Future.waitにcleanUpという新しい名前付き引数を追加 • SynchronousStreamControllerという新規クラスを追加 dart:collection • SplayTreeSetにfrom(Iterable)というコンストラクタを追加 dart:convert • Utf8Encoder.convertとUtf8Decoder.convertにオプショナルなstartとendの引数を追加 dart:core • RangeErrorに以下の新しいstaticヘルパを追加: checkNotNegative, checkValidIndex, checkValidRange, checkValueInInterval • intにmodPow関数を追加 • StringにreplaceFirstMappedとreplaceRangeを追加 dart:io • 新規クラス: FileLockとProcessStartMode • File にlock, lockSync, unlock及びunlockSyncを追加 • HttpServerのstaticメソッドのbindとbindSecure に2つの名前付き引数を追加：v6Onlyとshared • Process.startにmode (ProcessStartMode)引数を追加 • Processにstaticメソッドのkill Pidを追加 • ServerSocket.bind, RawServerSocket.bind, SecureServerSocket.bind及びRawSecureServerSocket.bindのメソッドたちにsharedという引数を追加 • SocketReferenceのメソッドとクラスは廃止化対象に • SocketとRawSocketのstaticメソッドのconnectにsourceAddress引数を追加 • StdoutにnonBlockingインスタンス属性を追加 dart:isolate • Isolateにstaticゲッタのcurrentを追加

		• 以下のIsolate APIsはVM上で動作するようになった: addOnExitListener, removeOnExitListener, setErrorsFatal, addOnErrorListener, removeOnErrorListener • spawnFunctionはトップ・レベルとstatic関数をメッセージ引数として渡すことが可能になった
v1.10	4月29日	• dart:convert　HtmlEscapeの問題を修正 • dart:core　Uri.parseに位置的引数のstartとendを追加 • dart:html　CssClassSetメソッド引数は'tokens'でなければならなくした • dart:io　ProcessResultのコンストラクタを公開 • VMでimportとIsolate.spawnUriはData URIスキームに対応
v1.11	12月30日	• appendHtmlとinsertAdjacentHtmlに validatorと treeSanitizer引数を追加 • Listのイタレータは運用モードでは ConcurrentModificationErrorを積極的にはスローしない。チェックド・モードでは変更チェックを積極的に行う • 実験的なIsolateのAPIの更新 • Listにunmodifiableコンストラクタを追加
v1.12	2015-08-31	• null対応演算子が新しく導入された • dart:async　StreamControllerに onListen, onPause, onResume及びonCancelコールバックのためのセッタたちを追加 • dart:convert　LineSplitterにsplitというIterableをかえすクラス・メソッドを追加 • dart:core　Uriクラスはあるが生成されたときにパス正規化を行い、'.'や'..'を無くすようにした。また hasAbsolutePath, hasEmptyPath, 及びhasScheme属性を追加 • dart:developer　新しい log関数は Observatoryにログ・イベントを送信する • dart:html　NodeTreeSanitizerにtrusted属性を追加 • dart:io • 新しく WRITE_ONLYと WRITE_ONLY_APPENDのファイル・モードを追加。Windowsで stdout/stderrをバイナリ・モードに変更 • dart:isolate　Isolate.spawnUriにonError, onExit及びerrorsAreFatalというパラメタを追加 • dart:mirrors　InstanceMirror.delegateを ObjectMirrorにまで移す。セレクタが演算子の際 InstanceMirror.getFieldの最適化
v.1.13		• dart:async　StreamControllerにnew typedef void ControllerCallback()に対応して onListen, onPause及びonResumeのためのゲッタたちを追加。StreamControllerにnew typedef ControllerCancelCallback()に対応して onCancelのためのゲッタを追加。　no handleErrorコールバックなしで fromHandlersで生成されたStreamTransformerのインスタンスは結果としてのストリームにerrorとともに stack tracesを渡す • dart:convert　Base-64エンコード/デコード対応する。Base64Codec, Base64Encoder及びBase64Decoderを追加。　新たにトップ・レベルの const Base64Codec BASE64を追加 • dart:core　Uriにメソッド removeFragmentを追加。

		String.allMatches (implementing Pattern.allMatches)が後回しの生成とする。 Resourceは廃止、pub.dartlang.org/packages/resourceを推奨 • dart:developer Observatoryの timelineと関わりあう Timelineクラスを追加。 ServiceExtensionHandler, ServiceExtensionResponse及びregisterExtensionを追加してデベロッパたちが自分たちのVMサービス・プロトコル拡張ができるようにした • dart:io NSSライブラリからBoringSSLライブラリにきりかえ。 SecureSocket, SecureServerSocket, RawSecureSocket,RawSecureServerSocket, HttpClient及び HttpServerは SecurityContextを使うようにした。 HttpClientは要求の中でURIフラグメントを送信しないようになった（HTTPプロトコルを除く）。 アイソレートたちが同じポートで接続できるようになった。 • dart:isolate spawnUriの指名引数に environmentを追加
v.1.14	1月28日	dart:async • Future.anyというstaticメソッドを追加 • Stream.fromFuturesコンストラクタを追加 dart:convert • Base64Decoder.convertにオプショナルなstartとendパラメタを追加 dart:core • StackTraceクラスにcurrentゲッタを追加 • Uriクラスにdata URIs対応を付加 ◦ dataFromBytes及びdataFromStringの2つのコンストラクタを追加 ◦ dataにdataゲッタを追加 • List.filledコンストラクタにgrowableパラメタを追加 • DateTimeにmicrosecond対応性を持たす: DateTime.microsecond, DateTime.microsecondsSinceEpoch, 及び DateTime.fromMicrosecondsSinceEpoch dart:math • Randomにセキュアな乱数発生器を返すセキュア・コンストラクタを追加 dart:io • PlatformにisIOSというstaticゲッタを追加。Platform.operatingSystemはiosを返すことが可能 • PlatformにpackageConfigというstaticゲッタを追加。 • RFC 7692に準拠したWebSocket圧縮に対応 • 総てのWebSocketに対し圧縮をデフォルトとした。新しい CompressionOptionsクラスを使って圧縮を制御できる。 dart:isolate • Package Resolution Configurationに実験的に対応させた • Isolateに対しpackageConfig及びpackageRootインスタンス・ゲッタを追加。 • IsolateにresolvePackageUriメソッドを追加 • Isolate.spawnUriconstructorに対しpackageConfigと automaticPackageResolutionという指名引数を追加
v.1.15	3月9日	dart:async • StreamViewクラスをconstクラスとした dart:core

		• 同じ名前の複数のクエリ・パラメタを処理するためにUri.queryParametersAllを追加。 dart:io • SecurityContext.usePrivateKeyBytes, SecurityContext.useCertificateChainBytes, SecurityContext.setTrustedCertificatesBytes,及びSecurityContext.setClientAuthoritiesBytesを追加 • SecurityContext.setTrustedCertificatesの directory引数を削除 • PKCS12認証と鍵のコンテナのためのSecurityContext対応を付加 • 認証データを受け付けるSecurityContext内の総ての呼び出しにオプショナルなpasswordという指名パラメタを追加
v.1.16	3月10日	アナライザの強化のみ
		•

2.0版の提案

非ヌル制約型とそれのデフォルト化

2015年5月27日のDart言語幹部たちのDart言語強化提案（DEP）会合で議論された提案である。これはPatrice Chalin氏の提案によるものである。この提案は非ヌル制約型とそれのデフォルト化(Non-null Types and Non-null By Default (NNBD))である。非ヌル制約型によりヌルの変数を参照するエラー（NPE）を排除し、またJSに変換した際のオーバヘッドを低減できる。非ヌル制約型はTypeScriptでも検討されている。

現在非ヌル制約変数が導入されている言語としては以下のものがある：

- Ceylon (Red Hat)
- Fantom
- Flow (Facebook)
- Kotlin (JetBrains)
- Haste
- Swift (Apple)

これはかなり大きな変更であるが、Dart2JSの出力の高速化に大きく貢献する（とりわけDartのVM実装ブラウザの推進に重点が置かれた現在）。これは多分2.0版として導入される可能性がある。この提案に対する議論はここを見て頂きたい。但し2.0版が近い将来導入される可能性はない。

第2章　変数とその型(Variables and Types)

変数はメモリ内の蓄積場所である。

変数にはトップ・レベルに宣言された変数（トップ・レベル変数）、あるクラスに結び付けられた変数（static変数）、及びあるオブジェクトに結び付けられた変数（インスタンス変数）がある。インスタンス変数はあるオブジェクトのフィールド、あるいは属性（property）とも呼ばれる。

注意：非ヌル制約型とそれのデフォルト化の提案に関して
2015年5月27日のDart言語幹部たちのDart言語強化提案（DEP）会合で議論された提案である。これはPatrice Chalin氏の提案によるものである。この提案は非ヌル制約型とそれのデフォルト化(Non-null Types and Non-null By Default (NNBD))である。非ヌル制約型によりnullの変数を参照するエラー（NPE : null pointer error）を排除し、またJSに変換した際のオーバヘッドを低減できる。非ヌル制約型はTypeScriptでも検討されている。これはかなり大きな変更であるが、Dart2JSの出力の高速化に大きく貢献する（とりわけDartのVM実装ブラウザの推進に重点が置かれた現在）。その場合は従来nullがあり得るとして定義されてきた変数は何らかのアノテーション（例えば*?T*はTという非ヌル制約型をnull許容型にした型を意味する）を付す作業が必要となる。この提案に対する議論はここを見て頂きたい。これは2.0版として導入される可能性がある。但し2.0版が近い将来導入される可能性はない。

2.1節　　構文

variableDeclaration（変数の宣言）:
declaredIdentier（宣言された識別子） (', ' identier（識別子）)*
;

declaredIdentifier（宣言された識別子）:
　metadata（メタデータ） finalConstVarOrType（final、Const、又はVarOrType） identifier（識別子）
;
finalConstVarOrType（final、Const、又はVarOrType）:
　final type?
| **const** type?
| varOrType（varまたは型）
;
varOrType（varまたは型）:
var
| type
;

initializedVariableDeclaration（初期化された変数の宣言）:
declaredIdentier（宣言された識別子） ('=' expression（式）)? (', ' initializedIdentier（初期化された識別子）)*
;

initializedIdentierList（初期化された識別子のリスト）:
initializedIdentier（初期化された識別子） (', ' initializedIdentier（初期化された識別子）)*
;

変数宣言文(variable declaration statement)は新規ローカル変数を宣言する。

T id;または*T id* = *e*; の変数宣言文は静的型Tを持った新しい変数*id*を最も内側で包含するスコープ内にもたらす。**var** *id*;または**var** *id* = *e*;の変数宣言文は静的型**dynamic**を持った新しい変数*id*を最も内側で包含するスコープ内にもたらす。総ての場合で、もしその変数宣言にfinal修飾子が前に付いたときは、その変数はfinalとしてマークされる。*T id*;の形式の変数宣言文は*T id* = **null**;と等価である。即ち初期化されていない変数は例え型アノテーションが付されていても初期値**null**をもつ。Dartでは**null**もオブジェクトである。

識別子はASCII英数文字(大文字と小文字は識別される)およびアンダスコア('_')を使用する。その他の文字は使用してはいけない。変数の識別子の最初の文字は英数小文字かアンダスコア(該ライブラリ内のプライベート変数のとき)になる。なお、ファイル識別子は別のルールとなる。

なおアンダスコア('_')のみの識別子名は、アナライザ(構文チェッカ)が「その変数は使われていない」というエラーの生成を抑制するのによく使われる。例えば次のコードではfor-inループの中で'_'という名前の変数は使われてはいないが構文上必要である。'_'の代わりに別の識別子名を使うとアナライザは"local variable is not used"という警告を発生する。

```
main() {
 // using an underscore silences "local variable is not used" warnings when running
dartanalyzer
 for (var _ in new Iterable.generate(1)) {
  print('no warnings');
 }
}
```

'_'のみの変数名は以下のようにコールバック関数において引数に関係なくそのイベントに対応するときにもよく使われる:

```
Polymer.onReady.then((_) { ……
```

2.2節　　静的変数の初期化

初期化されていない変数は初期値**null**を持つ。Dartでは**null**もオブジェクトである。総てのオブジェクトは型Objectを継承している。

変数宣言に対し型アノテーションによりその静的型を明確化できる。

```
String name = 'Terry';
```

コンパイラやIDEはそれらの型アノテーションを使ってバグ検出と警告を行うことができる。しかし運用モード(production mode)での実行ではそれらのアノテーションは無視される。

当初のDart言語で注意しなければいけなかったことは静的変数(特定のインスタンスに結び付けられていないで、むしろあるライブラリまたはクラス全体に結び付けられている変数で、トップ・レベルの変数宣言とクラス宣言内のstatic変数がこれに相当する。)の宣言がコンパイル時に定数オブジェクトで初期化されないとコンパイル時エラーとなったことである。非定数オブジェクトで初期化出来るのはローカル変数とメンバ変数だけである。これはアプリケーションの立ち上がりを早くしたいとのGoogleの欲求からきている。しかしながらこの制約は厳しすぎる。例えば次のコードでは3行目のようなトップ・レベル変数定義は**new** c().dはコンパイル時定数ではないのでエ

ラーになる。同様に関数式もトップ・レベルに置けない。

code_02.2.dart

```
var a;
void test(){ print(a); } // これはok
//var b=new c().d;       // これはエラーだった
//var test = (){print('hi');}; // これもエラーだった
class c{
//  static var staticVariable = a; // これはエラー
  var d='hello';
}
main() {
  a='hi';
  print(a);
  test();
  var b=new c().d;
  print(b);
}
/*
hi
hi
hello
*/
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

しかしながら2012年3月23日に改正された仕様書0.08版ではこの制約が緩められた。即ち「静的変数(トップ・レベル変数を含む)の初期化に際しては、単に定数式でなくても、どの式も使うことが可能になる。但し初期化はその変数の最初の読み出し時点で初期化がなされる。その初期化で例外が発生したらその変数はnullにセットされ、例外が伝搬する」というもので、これをGoogleは後まわしの初期化(lazy initialization)と呼んでいる。

この変更はM1変更の一環として2012年9月に実装された。プログラムの実行中に初めて生じる初期化のために処理がそこで遅延を起こす可能性が出る。これは処理時間にクリチカルなアプリケーションでは注意が必要である。しかしGoogleにとってクライアント側(ブラウザ上)でのアプリケーションの起動が早いことはそれよりも優先される。Googleの最大の顧客は一般ユーザであって、プログラマではない。

2.3節　　クラス変数宣言(static変数)

クラスの項で説明するが、Javaと同じく、クラス宣言のなかの変数メンバに**static**をつけると、それはクラス変数となり、そのクラスのインスタンスに関係なくただひとつそのオブジェクトが用意される。トップ・レベル変数、即ちライブラリの真下に存在している(つまりクラスに属さない)変数は暗黙的にstatic変数である。トップ・レベル変数宣言に**static**を付けるとコンパイル時エラーになる。トップ・レベルの変数を不変と宣言するには**final**を付すだけで良い。

code_02.3.dart

```
import 'dart:math' as Math;
double piVal = Math.PI;
```

```dart
class Alarm{
  static final FIRE = 1;
  static final EARTHQUAKE = 2;
  static final TSUNAMI = 3;
}
main() {
  print(piVal);
  print(Alarm.TSUNAMI);
}
/*
3.141592653589793
3
*/
```

クラス変数はそのクラスをインスタンス化しなくてもAlarm.TSUNAMIという具合に呼び出すことができる。

2.4節　　変数宣言（final、var、及び型）

変数の宣言は**final**、**var**、または型、及びそれに続く識別子で構成される。初期化されていない変数は**null**の初期値を持つ。**var**と宣言したときは、それはJavaScriptと同じで動的な型であり、**dynamic**型を指定したと同じ効果を持つ。

code_02.4a.dart

```dart
String variable;
main() {
 print(variable);
}
/*
null
*/
```

String型の**null**は"または""（空の文字列）ではないことに注意されたい。

変数を**final**として宣言すると、一旦初期化されたらその値は変更できない。**final**は従って単一代入または初期化子（initializer）をもっているかでなければならない。

code_02.4b.dart

```dart
final variable = 1; // single assignment
class A {
 final variable;
 A(this.variable); // initializer
}
main() {
//  variable += 1; // Error: cannot assign value to final variable "variable"
//  new A(2).variable = 3;  // Error: field variable is final
}
```

この場合は最初の**final**変数宣言は単一代入であり、次の**final**変数宣言は初期化子で初期化される。いずれもそのような変数には代入はできなくなる。但し代入ではなくてその属性等は変更可能である（「代入」の節参照）。

code_02.4c.dart

```dart
final a = {'x': 1};
main() {
```

```
  print('final a = $a');
  a['y'] = 2;
  print('manipulated a = $a');
}
/*
a = {x: 1}
manipulated a = {x: 1, y: 2}
*/
```

2.5節　　不変定数宣言 (const)

constは値に対する修飾子である。使い方としては、例えばcollectionに対しconst [1, 2, 3]と記述したり、あるいはconst Point(2, 3)のように**new**を使わないでオブジェクトを構築したりする。Constの意味合いはコンパイル時にそのオブジェクトが決定され、そのオブジェクトは固定され、変化しない(immutable)ということである。従って**const**が付いたオブジェクトはコンパイル時定数(compile-time constants)とも呼ばれる。

1. constオブジェクトはコンパイル時に計算可能なデータから生成されねばならない：即ち実行時に計算しなければならないものへのアクセスは許されない。1 + 2は有効なconst式になるが、new DateTime.now()はconstオブジェクトにはなれない。例えば次のコードは正しい：

```
const bar = 1000000;       // 圧力単位 (in dynes/cm2)
const atm = 1.01325 * bar; // 標準大気圧
```

2. constオブジェクトは推移的に不変である。ListやMapなどのあるコレクションがconstであるとすると、そのコレクションの要素の総てがconstである。

3. あるconstの値に対して、ただひとつのconstオブジェクトが生成され、そのconst式が何回呼ばれようともそのオブジェクトが再使用される。すなわち；

code_02.5.dart

```
getConst() => const [1, 2];
main() {
  var a = getConst();
  var b = getConst();
  print(identical(a, b)); // true
}
```

2.6節　　名前空間(Namespaces)

Dartは構文的にスコープづけ(lexically scoped)されており、変数、関数、及び型の為には単一の名前空間(namespace)のみが使われる。セッタとゲッタ以外で同じ名前を持ったものが同じスコープ内にひとつ以上ある場合はコンパイル時エラーになることに注意が必要である。名前は、そのスコープ内で宣言することにより、あるいはインポートまたは継承といった他のメカニズムにより、あるスコープ内に導入される。**import**したものがもし同じ名前を持っている場合は、

```
import 'dart:html' as html;
```

のように、プレフィックスを付加する。

importディレクティブは全ての宣言の前に置かれていなければならない。

名前空間に関しては仕様書0.08版から仕様書の3.1節及び14章でより詳細に規定されているので、そちらを見て頂きたい。

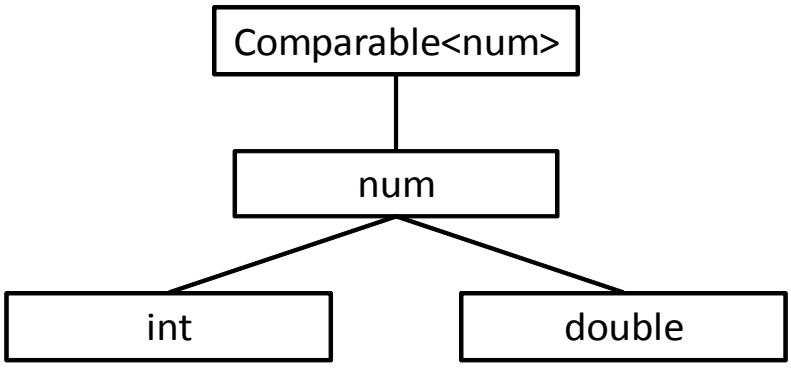
2.7節　　プライバシ

Dartはpublicとprivateの2つのプライバシ・レベルに対応している。但しpublicとかprivateとかいうキーワードが存在する訳ではない。

識別子がアンダースコア(_文字)で始まるときはその宣言はprivateであり、それ以外はpublicである。ある宣言mがライブラリL内で宣言されているとき、あるいはmがpublicとして宣言されているときは、mという宣言はライブラリLに対しアクセス可能である。このことはprivate宣言たちはそれらが宣言されているライブラリ内でのみアクセスされることを意味する。

2.8節　　数値変数の型

Dartには下図のように可変長整数(その最大長はメモリによってのみ制限される)である**int**(但しビット演算は32ビット)と、64ビット倍精度浮動小数点の**double**の2つが用意されており、ともに**num**という抽象クラスを実装している。**num**はまたComparableインターフェイスを実装している。



従って数値変数を宣言するときは**int**、**double**、あるいはそのどちらかでも良いときは**num**という型指定を行う。

Code_02.8a.dart

```
import 'dart:math';
num numval = PI;
main() {
```

```
  int intval = numval + 5; // Error in checked mode: double is not assignable
to int
  print(intval);
}
/*
8.141592653589793
*/
```

この例の3行目のintvalに対するint指定はチェック・モードで実行時の警告となる。従って**double**、**num**または**var**で宣言しなければならない。

以下は**num**抽象クラスのなかのメソッドから有用と思われるものである:

code_02.8b.dart

```
main() {
  // 絶対値
  print((-4).abs() == 4);

  // 切り上げときり下げ
  print(3.14.ceil() == 4.0);
  print(3.14.floor() == 3.0);

  // 四捨五入
  print(3.14.round() == 3);
  print(3.54.round() == 4);
  print(3.5.round() == 4);
  print(3.49.round() == 3);

  // 切り捨て
  print(3.141592.truncate() == 3.0);

  // double からint への変換
  print(3.14.toInt() == 3);
}

/*
  総てtrue
*/
```

なお**double**にはNAN (not a number) が定義されている(メソッドisNaNは**num**インターフェイスで定義されていることに注意)。これはそもそも機械語でのビット列での浮動小数点表現で使われる用語だったものであるが、Dartでは0/0演算がこれに相当する。また同じく**double**インターフェイスで定義されているINFINITYとNEGATIVE_INFINITYはゼロによる除算で発生する。NAN、INFINITY及びNEGATIVE_INFINITYにある値を加減乗除してもそれは変わらない(エラーにはならない)。

code_02.8c.dart

```
main() {
  double x = double.NAN;
  double y = 1.0;
  print("${x.isNaN}, ${y.isNaN}");
  double z = 0/0;
  print("$z, ${z.isNaN}");
  var p = 1/0;
  var q = -1/0;
  print("$p, $q");
```

```
}
/*
true, false
NaN, true
Infinity, -Infinity
*/
```

2.9節　　組込み済みの型

Dartのcoreライブラリにはいろんなインターフェイスが用意されているが、良く使われるのは次のようなものだろう:

- String
- bool
- num
  - int
  - double
- Collection
  - List
  - Set
- Map

これらの詳細はDartのAPI参照を見て頂きたい。

以前はDynamicという型が存在していた。これは静的型指定がないときに適用される。即ち型指定アノテーションの代役である。2012年9月13日にDartチームのメンバのLasse Nielsenが"Dynamic”を"dynamic”に変更するとともに、"Object"のdynamicゲッタを廃止する計画だと発表した。これは総ての組込み識別子とキーワードを小文字で統一する為だという。

第3章　関数(Functions)

関数は実行可能なアクションの抽象化である。Dartでは関数もオブジェクトでその型はFunctionである。即ちDartの関数は第1級関数(first-class function)である。従って後述のように、ある関数を別の関数にパラメタとして渡すことも可能である。

関数には関数宣言(function declarations)、メソッド(methods)、ゲッタ(getters)、セッタ(setters)、コンストラクタ(constructors)、及び関数リテラル(function literals)がある。

関数宣言にはライブラリのトップ・レベル（即ちクラスのメンバでない）にある関数（例えばmain()でトップ・レベル関数という）、あるクラスに結び付けられた関数（static関数とかクラス関数という）、あるオブジェクトに結び付けられた関数（インスタンス・メソッドという）、および関数の内部に宣言されたローカル関数がある。関数の中にネストした関数を持つことも可能である。

JavaScriptと違って定義の為のfunctionといったキーワードは存在しない。

総ての関数はひとつのシグネチュアとひとつのボディを持つ。シグネチュアはその関数の仮パラメタたち(formal parameters)、及びその名前と戻りの型を記述する。ボディはその関数によって実行される文たち(statements)が入っているひとつのブロック文(block statement)である。形式 => e の形式の関数ボディは{return e;}の形式のボディと等価である。

ある関数の最後の文が**return**文でないときは、その関数ボディに対してはreturn null;という文が暗示的に付加される。

また関数宣言が戻り値の型を明示的に指定していないときは、その型は**dynamic**型として扱われる。ここに**dynamic**は未知(unknown)という意味の型である。

3.1節　　構文

functionSignature（関数シグネチュア）:
returnType（戻りの型）? identifier（識別子） formalParameterList（仮パラメタ・リスト）
;

returnType（戻りの型）:
void |
type（型）
;

functionBody（関数ボディ）:
'=>' expression（式） ';' |
block（ブロック）
;

block（ブロック）:
'{' statements（文たち） '}'

;

関数宣言は関数からメソッド、ゲッタ、セッタ、及び関数リテラルを除いたものをいう。

仮パラメタの構文は次のようである:

formalParameterList(仮パラメタ・リスト):
'(' ')' |
'(' normalFormalParameters(通常の仮パラメタたち) (', ' namedFormalParameters(名前付き仮パラメタたち))? ')' |
(namedFormalParameters(名前付き仮パラメタたち))
;

normalFormalParameters(通常の仮パラメタたち):
normalFormalParameter(通常の仮パラメタ) (', ' normalFormalParameter(通常の仮パラメタ))*
;

optionalFormalParameters(オプショナルな仮パラメタたち):
optionalPositionalFormalParameters(オプショナルな位置的仮パラメタたち) |
namedFormalParameters(名前付き仮パラメタたち)
;
optionalPositionalFormalParameters(オプショナルな位置的仮パラメタたち):
'[' defaultFormalParameter(デフォルト仮パラメタ) (',' defaultFormalParameter)* ']'

namedFormalParameters(名前付き仮パラメタたち):
'[' defaultFormalParameter(デフォルトの仮パラメタ) (', ' defaultFormalParameter(デフォルトの仮パラメタ))* ']'
;

関数の呼び出しで実引数(actual argument)たちと仮パラメタたちとのバインドがされる:

arguments(引数):
'(' argumentList(引数リスト)? ')'
;

argumentList(引数リスト):
namedArgument(名前付き引数) (',' namedArgument(名前付き引数))*
| expressionList(式リスト) (',' namedArgument(名前付き引数))*
;

namedArgument(名前付き引数):
label(ラベル) expression(式)
;

ボディの実行は以下のことが最初に起きた時点で終了する;

- キャッチ(catch句による捕捉)されていない例外がスローされる
- ボディのなかでreturn行が実行される
- そのボディの最後の行が実行された

3.2節　　簡単な関数宣言例

JavaScriptと違ってfunctionというキーワードは存在しない。一行関数(one-line function)と呼ばれる最も簡単な関数宣言は次のようなものになる:

```
sayHello() => 'Hello World';
```

=>という記号はそれに続く式の値を返す(即ち{ return expr; })という意味で、次の記述と等価である:

```
sayHello() {
  return 'Hello World';
}
```

即ち一行関数はただ一つの式(条件式も含めて)からなる関数に適用できる。

Dartでは戻り値にたいしオプショナルに型指定ができる。型指定をしていないときはそれは**dynamic**(unknown: 未知)として扱われ、次の記述と等価である:

```
dynamic sayHello() {
  return 'Hello World';
}
```

sayGreeting関数はString型を返すので、正確に型指定をするときは次のように書く:

```
String sayHello() {
  return 'Hello World';
}
```

もし別の型(例えばintやdouble)を指定するとDartはコンパイル時に静的型警告を出すが、実行は正しく行われる。Dartは実行時(非チェック・モード)では型指定を無視する。

戻り値を持たない関数は型指定をする必要がないが、Javaと同じようにvoidを指定したり、またdynamicを指定することもできる:

```
  var n;
setVar(x) { n = x; }          // または
// void setVar(x) { n = x; } // または
// dynamic setVar(x) { n = x; }
  setVer(1.4142);
  print(n);
```

static変数に対応したstaticメソッドも存在する。この場合継承はできるが通常はあまり意味がないだろう。

```
var a = 'Hello', b = 'world';
class Human {
 static valueA() => '$a';
}
```

```
class Jim extends Human {
 static valueA() => '$b';
}
void main() {
 print(Human.valueA());
 print(Jim.valueA());
}
```

3.3節　　main()

Dartによるプログラムの実行はトップ・レベル関数（即ちクラスのメンバでない）の**main()**を引数なし（またはコマンド行引数つきで）呼ぶことで開始される。即ちトップ・レベルのmain関数はそのアプリケーションの入り口として機能する。もしそのコードがトップ・レベル関数main()を宣言またはインポートしていないときは実行時エラーである。

```
main() {
 var name = 'World';
 print('Hello, ${name}!');
}
```

このコードの解説：

- Dartプログラム内で変数を作りたいときは何時でもvarキーワード、finalキーワード、あるいは型の名前を使用してその変数を宣言しなければならない。
- print()による出力： Dartのprint()関数はテキストをコンソールに送信する。
- 文字列リテラル： 文字列を示すにはシングル・クオートまたはダブル・クオートのどれかを使用する。'World'と"World"は等価である。
- ${expression}による文字列の挿入： Dartでは文字列リテラルのなかに式を埋め込むことが出来る。これは文字列補完（または文字列インターポレーション）と呼ばれ、この例のようにその式が単に変数のときは中カッコ({})を省略できる。以下の2つの文は等価である：

```
print('Hello, ${name}!');
print('Hello, $name!');
```

main()関数は引数をとることができる。詳細は「Dart VM」の節の「引数付きでの実行」の項に記してある。同様に、main()関数は値を返すことも可能である。詳細は「Dart VM」の節の「プログラムからの終了」の項に示してある。

3.4節　　必須パラメタとオプショナルなパラメタ

関数のパラメタ・リストは丸括弧で囲ったパラメタのリストである。名前付きパラメタは名前付きで（**指名パラメタ(named parameters)**とも呼ぶ）順不同で引数とすることができる。一方**位置的パラメタ(positional parameters)はその順で引数としなければならず、また名前付きでの指定はできない。**

オプショナルな名前付きパラメタたちは中括弧('{'と'}')で囲ったリストである。またオプショナルな位置的パラメタたちも存在し、これは角括弧('['と']')で囲ったリストである。オプショナルなパラメタたちは呼び出し時は引数としなくても良い。通常そのようなパラメタはデフォルト値を要するので'='(位置的パラメタの時)または':'(名前付きパラメタのとき)を使って初期値を指定する。デフォルト値はコンパイル時に定数になっていなければならない。

以上を整理すると次のようになる:

パラメタの形式	必須パラメタ	オプショナルな位置的パラメタ	オプショナルな名前付きパラメタ
宣言時のパラメタ・リストの形式	パラメタ間をカンマで区切る	[]で囲ったカンマで区切られたパラメタたち	{}で囲ったカンマで区切られた名前付きのパラメタたち。名前付きのパラメタは名前:変数)の形式となる
宣言時のデフォルト値指定	不可	仮パラメタ:値で指定	名前:値で指定
呼び出し時の引数	指定された順にセット	指定された順にセットするが終わりのパラメタから連続してデフォルト値を使うときはそれらを省略可	名前:値で指定し、セット順は問わない。省略可

オプショナルなパラメタにはListやMapなどを使うこともできる。

次のコードはオプショナルな名前付きパラメタを含んだ例である:

code_03.4.dart

```
/* 挨拶*/
void sayGreeting(String name, {String salutation : 'Hello', String exclamation
: '!'}) {
  String greeting = '$salutation $name$exclamation';
  print(greeting);
}
main() {
  sayGreeting('Bill');
  sayGreeting('Tom', salutation: 'Hi');
  sayGreeting('Alia', exclamation: '!!', salutation: 'Good Morning');
}

/*
Hello Bill!
Hi Tom!
Good Morning Alia!!
*/
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の章の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

- このコードではsalutation(挨拶)とexclamation(感嘆符)2つの名前付きパラメタがオプションとなっている。
- 最初の呼び出し例では必須パラメタのみを指定している。オプショナルなパラメタにはデフォルト値が適用される

- 2番目の呼び出し例ではオプショナルなパラメタのsalutationのデフォルト値を'Hi'に変更している。
- 3番目の呼び出し例では更にオプショナルなパラメタを名前付きで順番を逆にして指定している。

上記のように**オプショナルな名前付きパラメタ**を鍵カッコで囲む。デフォルト値は**:**で指定する。

また**オプショナルな位置的パラメタ**を指定するときには角括弧で囲むようになる。デフォルト値は=で記述する。記述しないとnullがデフォルト値と見做されるが、**null**と明示的に書いたほうが好ましい。

```
main() {
  // The following are all valid.
  connectToServer('secret');
  connectToServer('secret', '1.2.3.4');
  connectToServer('secret', '1.2.3.4', 9999);
}
connectToServer(String authKey, [ip = '127.0.0.1', port = 8080]) {
  print('aunthKey = $authKey, ip = $ip, port = $port');
}

/*
aunthKey = secret, ip = 127.0.0.1, port = 8080
aunthKey = secret, ip = 1.2.3.4, port = 8080
aunthKey = secret, ip = 1.2.3.4, port = 9999
*/
```

この場合は呼び出しにあたっては位置的パラメタには名前が付すことができなず、名前付きでは指定できない。この例では:

- connectToServerという関数は3つの位置的パラメタが使われている
- 呼び出すときはこの3つのパラメタをその順にセットする
- 最初のauthKeyは必須パラメタだが、ipとportはオプショナルでデフォルト値がセットされている
- ipをデフォルト値以外の値にするときは2番目のパラメタとしてセットする
- portをデフォルト値以外の値にしたいときは、ipの値もセットしなければならない

3.5節　　関数内関数

無論関数内に関数を置くことは可能である。関数内関数はローカル関数とも呼ぶ。次の例ではimportantify（重要性強調）という内部関数が定義されている:

code_03.5.dart

```
String sayHello(String msg, String to) {
  String importantify(msg) => '!!! ${msg} !!!';
  return '${importantify(msg)} to ${to}';
}
main(){
  print(sayHello('Urgent', 'Bill'));
}
/*
!!! Urgent !!! to Bill
*/
```

但し内部関数は親の関数が呼ばれるごとに割り当てられるので、一般には処理速度が遅くなることに注意が必要である。

関数内関数はそれが呼ばれる前に定義されていなければならない。

関数を再帰的に使うことも可能であり、これは関数リテラルの所の階乗計算で説明する。

3.6節　　オブジェクトとしての関数

本章の最初に示したように、Dartの関数は第1級関数(first-class function)である。

関数あるいは関数リテラルは変数（あるいは関数のパラメタ）として扱える。

code_03.6a.dart

```
main() {
  var say = (s) => print(s); // varの代わりにFunctionと書くことも可
  say("Hello!"); // "Hello!"と出力
}
```

次の例は関数を関数あるいはメソッドの引数として使ったものである:

code_03.6b.dart

```
printElement(element) {
  print(element);
}

main() {
  var list = [1,2,3];
  list.forEach(printElement); // printElementをパラメタとして渡す
}
```

もう一つのシンプルな例を示そう:

code_03.6c.dart

```
Function adder(num addBy) {
  return (num i) => addBy + i; // 関数リテラルを返す関数定義
}

main() {
  var addByTwo = adder(2); // 2を加算する関数をaddByTwoという変数として扱う
  print(addByTwo(3));      // 5を出力
}
```

構文クロージャ(Lexical closures)

構文クロージャとはある関数（あるいは関数リテラル）に於いて、たとえその関数がオリジナルのスコープの外部で使われていたとしてもその構文スコープ内に存在している変数へのアクセスが可能な関数あるいは関数リテラル（関数式）のオブジェクトをいう。

関数はそれが囲むスコープ内で定義された変数たちを包含している。上記の例では、adder()という関数はaddByという変数を包含している。この関数が返している関数リテラルがどこで使われようとも、この関数リテラルはaddByという変数を覚えている。

print(addByTwo(3));という行では、addByTwo(3)という式のなかで実際は2という値を持ったaddByという変数が使われている。この値はもともとadderという関数に渡されたものである。

もし前記のコードに以下の行をmain関数のなかの最後に追加したらどうなるであろうか：

```
var addBy100 = adder(100);
print(addBy100(1)); // 101
print(addByTwo(1)); // 3を出力、即ち addByTwo でセットされた addBy の値が保持されている。
```

このようにその関数リテラルがどこで使われようとも該関数リテラルはaddByという変数を覚えている。

3.7節　　関数型エイリアス(typedef)

関数型エイリアス(function type alias)は型表現の為の名前を宣言する。Dartでは関数は文字列や数字と同じくオブジェクトである。関数を変数や戻りの型として使うときには**typedef**プレフィックスを使った関数型エイリアスを使用する。

functionTypeAlias（関数型エイリアス）:
　typedef functionPrefix（関数プレフィックス） typeParameters（型パラメタたち）? formalParameterList（仮パラメタ・リスト） ';'
　;

functionPrefix（関数プレフィックス）:
　returnType（戻りの型）? identifier（識別子）
　;

Dartでは関数もオブジェクトである。関数型エイリアスはある関数の型に名前を付し、フィールドや戻り値の型を宣言できるようにする。typedefはある関数型がある変数に代入されたときにその型情報を保持する。

typedefによる関数型エイリアスの動作を以下のコードで示す：

code_03.6.dart

```
typedef String TempToText(var something);
String fromCelsius(var temp)
  => 'Celsius: $temp degreesC, Fahrenheit: ${temp*9/5+32} degreesF';
String fromFahrenheit(var temp)
```

```
  => 'Celsius: ${(temp-32)*5/9} degreesC, Fahrenheit: $temp degreesF';
void printTemp(var someTemp, TempToText whichMethod){
  print(whichMethod(someTemp));
}
main() {
  bool celsius = true; // unit of the someTemp
  var someTemp = 0;    // input temperature
  TempToText whichMethod;
  if (celsius) whichMethod = fromCelsius;
    else whichMethod = fromFahrenheit;
  printTemp(someTemp, whichMethod);
}
/*
Celsius: 0 degreesC, Fahrenheit: 32 degreesF
*/
```

- typedefでTempToText（温度を入力して文字列を返す）という名前の関数型は、入力がvar即ちdynamic型で戻り値がStringである関数であると定義している。
- void printTemp(var someTemp, TempToText whichMethod)というメソッドは、someTempというvar型の温度入力と、whichMethodというTempToText型の関数を入力としている。そのボディ部はsomeTempを引数にしてwhichMethodを読んだ結果をプリントする。
- fromCelSiusとfromFahrenheitの2つのメソッドはTempToText型の関数で各々摂氏入力と華氏入力を受け付け、それを摂氏と華氏の温度で文字列に変換する。
- main()メソッドの中ではcelsiusというブール変数で摂氏入力かどうかを指定する。またsomeTempは摂氏または華氏の入力温度をセットする。
- 次にcelsiusが真かどうかでどのメソッドを使うかを決め、それをwhichMethodという変数にセットする。
- Mainメソッドの最後の文であらかじめ定義したprintTemp関数を呼び出している。

typedefは単に別名であるので、これは任意の関数の型をチェックする手段にもなる：

```
typedef int Compare(int a, int b);
int sort(int a, int b) => a - b;
main() {
  assert(sort is Compare);  // True!
}
```

Dartでは仕様書上は型エイリアスは型式の名前として宣言できるようにするものであるが、現在は型エイリアスは関数のみに制限されている。

第4章　関数リテラル(Function Literal)

関数リテラル（関数式とも書く）は関数のひとつであるが、関数宣言を持たない。つまり仮パラメタ・リストと関数式のボディのみからなる。関数リテラルはコードのある実行可能な単位をカプセル化するひとつのオブジェクトである。関数リテラルは組込みインターフェイスである**Function**を実装している。関数リテラルはJavaScriptにはあるが、Javaには導入されていない。

関数式は仕様書上は「式」の章に含まれている項目ではあるが、この資料では前章の関数との比較の為に抜き出して関数の章のあとに記した。関数は関数宣言をすることで定義されるが、関数リテラルは文の中の式として記述される。

関数リテラルはコールバック関数記述に良く使用される。例えばTimerというインターフェイスの参照ドキュメントを見ると、そのコンストラクタは次のようになっている：

```
factory Timer(Duration duration, void callback())
```

これはduration時間が経過したらcallback関数が呼び出されるというタイマを構成する宣言である。コールバック関数を外部で定義するのではなく、コンストラクタの中の式として記述するときは次のようになる：

```
new Timer(const Duration(seconds: 1), (Timer t){print('Elapsed 1
Second!');});
```

また後述するように関数リテラルを変数として定義し、パラメタとして使用したりすることもある。

4.1節　　構文

初期の仕様書では構文は下記のように関数のそれとほぼ同じだった。但し識別子がオプションであり、識別子が無いものを匿名関数式と呼ばれる。

functionExpression（関数式）:
　(returnType（戻りの型）? identifier（識別子）)? formalParameterList（仮パラメタ・リスト）
functionExpressionBody（関数式ボディ）
　;

しかしながら現在の仕様書では下記のように匿名関数式のみが許されている：

functionExpression（関数式）:
　formalParameterList（仮パラメタ・リスト） functionExpressionBody（関数式ボディ）
　;

functionExpressionBody（関数式ボディ）:
　'=>' expression（式）
　| block（ブロック）
　;

関数式呼び出し*i*は$e_f(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式をとり、ここに$e_f$は式である。$e_f$が識別子*id*のときは、

*id*は上記のとおり必然的にローカル関数、ライブラリ関数、ライブラリまたは静的ゲッタまたは変数を意味しなければならず、あるいは*i*は関数式呼び出しとは看做されない。もしe_fが属性アクセス式のときは、*i*は通常のメソッド呼び出しとして扱われる。そうでないときは:

関数式呼び出し$e_f(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の計算は通常のメソッド呼び出し$e_f.call(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$と等価である。

この定義、及びメソッドcall()がからむ他の定義たちの意味合いは、それらがcall()メソッドを定義しているときに限りユーザ定義の型たちが関数値として使えるということである。メソッドcall()はこの点に関し特別である。call()メソッドのシグネチュアが組み込み呼び出し文法を介してそのオブジェクトを使う際に適正なシグネチュアを決める。

e_fの静的な型がある関数型に割り当てられない可能性があるときは静的警告である。もし*F*が関数型でないときは、*i*の静的型は**dynamic**である。そうでないときは*i*の静的型は*F*の宣言された戻り型である。

以下は関数式の例である:

```
loudPrint(String msg) {
  print(msg.toUpperCase());
}
```

これは文字列msg大文字に変換してプリントする式であるが、単行式でこれを表現すると次のようになる:

```
(msg) => print(msg.toUpperCase())
```

4.2節　　関数と関数リテラル

自分の口座への預金と引き出しの2つの手続きを持った次のコードを見てみよう:

code_04.2.dart

```
main() {
  var balance = 0;

var deposit = (amount) { balance += amount; };  // 預金1：関数リテラル
//  deposit(amount) { balance += amount; }          // 預金2：関数定義
var withdraw = (amount) { balance -= amount; }; // 引き出し1：関数リテラル
//  withdraw(amount){balance -= amount; }           // 引き出し2：関数定義

  deposit(1000);  // 預金呼び出し
  withdraw(100);  // 引き出し呼び出し

  print(balance);
}
```

（注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと）

関数を使っても関数リテラルを使っても呼び出しは同じで、また結果も同じである。関数リテラルは名前は不要である。その代わり**Function**型のオブジェクトとしてdepositあるいはwithdrawという変数に代入されている。従って

関数リテラルのオブジェクトは実行時に生成される。関数及び関数リテラルともにオブジェクトであるから別の関数の引数に使ったり、また戻しの値として使うことが可能になる。

4.3節　　関数リテラルを式の関数の識別子と同じように使うことができる

関数式を()で括ったものは関数識別子と同じように扱える。

以下の例では2つの値の平均値を求める関数リテラルに10と8という2つの値を与えて実行し、その結果をプリントしている。つまりprintというメソッドの中の式として関数リテラルが組み入れられている:

code_04.3.dart

```
/* 平均値 */
main() {
  print(((var x, y){return (x + y)/2;})(10, 8));
}
/*
9.0
*/
```

printの行は単行関数式を使って次のようにも書ける:

```
print(((x, y)=>(x + y)/2)(10, 8));
```

4.4節　　関数のパラメタに関数リテラルとデフォルト値を含むことができる

consoleOutは結果の文字列を表示する為のメソッド(ラッパー)・オブジェクトだが、これは呼び出し側で実体化する。またtoFahrenheitはどちらに変換するかを指定するが、デフォルトでは摂氏から華氏への変換が実行される。呼び出し側でこれを変更するときはtoFahrenheit: falseという具合に名前つきで指定する。

code_04.4.dart

```
/* 摂氏と華氏の変換 */
tempUnitConvert(var temp, consoleOut, {bool toFahrenheit : true}) {
  var tempC, tempF;
  if (toFahrenheit){
    tempC = temp; tempF = tempC * 9 / 5 + 32;
  } else {
    tempF = temp; tempC = (tempF - 32) * 5 / 9;
  }
  String result = 'Celsius: $tempC,  Fahrenheit: $tempF';
  consoleOut(result);
}
main(){
```

```
  tempUnitConvert(32, (str){print(str);}, toFahrenheit: false);
  tempUnitConvert(32, (str){print('--- $str ---');});
}

/*
Celsius: 0,  Fahrenheit: 32
--- Celsius: 32,  Fahrenheit: 89.6 ---
*/
```

4.5節　　関数及び関数リテラルのネスト

関数及び関数リテラルは関数及び関数リテラルを含むことができる。

code_04.5.dart

```
/* 挨拶 */
greeting([String salutation = 'Hello']) => (String name) => "$salutation
$name!";
main() {
  final greeting1 = greeting();
  final greeting2 = greeting('Hi');
  print(greeting1('Tom'));
  print(greeting2('Jim'));
}
/*
Hello Tom!
Hi Jim!
*/
```

オプショナルなsalutionという引数を持つgreetingという関数は次の記述と等しい:

```
greeting([String salutation = 'Hello']) {
  return foo(String name) => "$salutation $name!";
}
```

即ちこの場合はgreetingという関数の戻り値にfooという関数呼び出しが使われている。

あるいは次のように関数リテラルを関数型オブジェクトとして定義しても良い:

```
Function greeting = ([String salutation = 'Hello']) => (String name) =>
"$salutation $name!";
```

4.6節　　ネストした関数及び関数リテラル

関数及び関数リテラルは再帰可能であるが、処理速度は一般に遅くなる。次のコードは関数の再帰可能性を示す為に良く使われるパタンだが、最初に呼ばれたときのnという変数とその関数の中で呼ばれた次の関数オブジェクトのnとは別のものであることを使っており、ややトリッキーである。

code_04.6a.dart

```
/* 階乗計算（関数） */
factorial(n){if (n<=1) {return 1;} else {return n * factorial(n-1);}}
main() {
  print(factorial(5));
}
/*
120
*/
```

code_4.6b.dart

```
/* 階乗計算（関数リテラル） */
var factorial;
main() {
  factorial = (n){if (n<=1) {return 1;} else {return n * factorial(n-1);}};
  print(factorial(5));
}
/*
120
*/
```

もうひとつ良く引用されるサンプルはFibonacci数列である。これはnが増えると計算時間が増えるので、時間がかかる処理のシミュレーションによく使われる。

code_4.6c.dart

```
main() {
  print(new Fibonacci().fib(10)); // 89
}

class Fibonacci {
  int fib( int value ) {
    if( value == 0 || value == 1 ) {
      return 1;
    }
    return fib( value - 1 ) + fib( value - 2 );
  }
}
```

第5章　クラス (Classes)

クラスはそのインスタンスたちであるオブジェクトたちのセットの形式と振る舞いを規定する。Dartのクラスとインターフェイスの文法は基本的にJavaに準拠している。従ってJavaの技術者たちには基本的なことは特に説明する必要が無かろう。但しDartでは新しく名前つきコンストラクタ、セッタ/ゲッタ、関数エミュレーションなどが付加されている。

5.1節　　構文

classDefinition（クラス定義）:
　class identifier（識別子） typeParameters（型パラメタたち）? superclass（スーパークラス）? interfaces（インターフェイスたち）?
　'{' classMemberDefinition（クラス・メンバ定義）* '}'
;
classMemberDefinition（クラスのメンバの定義）:
　declaration（宣言） ';'
| methodSignature（メソッド・シグネチュア） functionBody（関数ボディ）
;
methodSignature（メソッド・シグネチュア）:
　constructorSignature initializers（コンストラクタ・シグネチュア・イニシャライザたち）?
| factoryConstructorSignature（ファクトリ・コンストラクタ・シグネチュア）
| **static** functionSignature（関数シグネチュア）
| **static**? getterSignature（ゲッタ・シグネチュア）
| **static**? setterSignature（セッタ・シグネチュア）
| operatorSignature（演算子シグネチュア）
;
declaration（宣言）:
　constantConstructorSignature（定数コンストラクタ・シグネチュア） (redirection（リダイレクション） | initializers（イニシャライザたち）)?
| constructorSignature（コンストラクタ・シグネチュア） (redirection | initializers)?
| **external** constantConstructorSignature
| **external** constructorSignature
| **external** factoryConstructorSignature（ファクトリ・コンストラクチャ・シグネチュア）
| ((**external static**?) | **abstract**)? getterSignature（ゲッタ・シグネチュア）
| ((**external static**?) | **abstract**)? setterSignature（セッタ・シグネチュア）
| **external**? operatorSignature（演算子シグネチュア）
| ((**external static**?) | **abstract**)? functionSignature（関数シグネチュア）
| **static** (**final** | **const**) type? staticFinalDeclarationList（static finalな宣言リスト）
| **const** type? staticFinalDeclarationList（イニシャライザ識別子リスト）
| **final** type? initializedIdentifierList
| **static**? (**var** | type) initializedIdentifierList
;
staticFinalDeclarationList（static finalな宣言のリスト）:
: staticFinalDeclaration（static fainalの宣言） (',' staticFinalDeclaration（static fainalの宣言）)*

;

staticFinalDeclaration（static final宣言）:

identifier（識別子） '=' expression（式）

;

クラスはコンストラクタ(constructors)、インスタンス・メンバたち(instance methods)、及び静的メンバたち(static members)を持つ。あるクラスの静的メンバは、その静的メソッドたち(static methods)、ゲッタたち(getters)、セッタたち(setters)、及び静的変数(static variables)たちである。あるクラスのメンバたちはその静的及びインスタンス・メンバたちである。

コンストラクタは継承されない。Cに言えばサブクラスはそのスーパークラスのコンストラクタを継承しない。コンストラクタが宣言されていないサブクラスはデフォルト（引数も名前も持たない）のコンストラクタのみを持つ。

各クラスはスーパークラスがない**Object**クラスを除いて単一のスーパークラスを持つ。クラスはそのinplements節(implements clause)のなかで宣言することで幾つかのインターフェイスを実装できる。

*抽象クラス(abstract class)*は**abstract**修飾子で明示的に宣言されたクラスである。抽象クラスはインスタンス化できない。その抽象クラスがインスタンス化出来るよう見えるようにするにはファクトリ・コンストラクタが必要である。

*クラスCのインターフェイス*は暗示的なインターフェイス(implicit interface)であって、*C*によって宣言されたインスタンス・メンバたちに対応したインスタンス・メンバたちを宣言しており、その直接的なスーパーインターフェイスたちは*C*の直接のスーパーインターフェイスである。型あるいはインターフェイスとしてあるクラス名がある場合は、その名前はそのクラスのインターフェイスを意味する。つまりJavaと異なり、明示的なインターフェイスは存在しない。その代わり総てのクラスは暗示的なインターフェイスを持つ。必要なら抽象クラスで明示的なインターフェイスを表現しても良い。これに関しては次の「インターフェイス」の章で解説する。

なおM1変更で、実装を伴わないメソッドは抽象メソッドとして扱われるようになった。その為、メソッドに対する**abstract**キーワードは使用しないこととなったことに注意されたい。クラス宣言のなかでボディの無いメソッド宣言は抽象メソッドと扱われる。

またM2変更でミクスイン(mixin)が導入された。ミクスインはサブクラスによって継承されることにより機能を提供し、単体で動作することを意図しないクラスである。ミクスインはメソッドが実装されているインターフェイスだともいえる。インターフェイスの章で説明するが、Dartでは各クラスがインターフェイスも持つようになった（従って明示的なインターフェイスは存在しなくなった）。従ってあるクラスをextendsキーワードにより継承、implementsキーワードにより実装、そしてwithキーワードによりミクスインできるようになる。

5.2節　　シンプルな例

次のコードはGoogleのチュートリアルにあったサンプルである:

code_05.2a.dart

```
class Greeter {
  var prefix = 'Hello,';

  greet(name) {
    print('$prefix $name');
  }
```

```
}

main() {
  var greeter = new Greeter();
  greeter.greet("Class!");
}
/*
Hello, Class!
*/
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

Greeter(挨拶状)というクラスはprefix(前置文)というインスタンス変数と、それにname(名前)をつないだgreet(挨拶)というインスタンス・メソッドからなる。

- class文
  この例ではデフォルトのスーパークラスObjectを持つGreeterという名前のクラスを定義している。Dartでは、総てのクラスは直接あるいは間接的にObjectがおおもとになっている。非Objectスーパークラスを指定したいときには、extendsキーワードを使用する。
- インスタンス変数
  通常の変数と同じように、インスタンス変数を生成するときにはvar、final、または型キーワードが必要である。各Greeterオブジェクトは'Hello,'に初期化されたprefixという名前の変数のコピーを持つ。インスタンス変数に直接(例えばgreeter.prefix = 'Hi,')、あるいはコンストラクタで、あるいはセッタ・メソッドをつかって値をセットできる。(セッタとゲッタのメソッドは別途説明する)
- コンストラクタ
  あるクラスのインスタンスを生成するには、newキーワードとそれに続くそのクラスのコンストラクタを呼び出す--この場合はnew Greeter()。このコードではGreeterコンストラクタが定義されていないので、そのときはそのスーパークラスの引数なしのコンストラクタが呼び出される。GreeterのスーパークラスはObjectなので、new Greeter()というコードはObject()を呼び出す。
- インスタンス・メソッド
  greet()メソッドはGreeterオブジェクトに結び付けられたある関数を定義している。

以前はインスタンス変数の初期化変数宣言の式はコンパイル時定数でなければならなかった。非初期化された変数はnullという値を持つ。次の例のように、new Date.now()はコンパイル時定数で無いのでコンパイル時エラーとなっていたが、現在は実行時に初期化されるようになっている。

code_05.2b.dart

```
class FieldTest {
  int minimum, maximum;  // null
  String name = 'Cresc'; // String constant
  DateTime today = new DateTime.now(); // Error, expected constant expression
}
main() {
  var ft = new FieldTest();
  print('${ft.name}, ${ft.today}');
}
```

組込み識別子の**static**をフィールドやメソッドの前に付けると、それはクラス変数やクラス・メソッドになる。つまりそのクラスのインスタンス毎に用意されるのではなく、そのクラスに対しひとつ用意される。従ってそのような変数や

メソッドにアクセスするときはそのクラスをインスタンス化する必要は無い。次の例は前記のコードを変更したものである。クラス変数は一般に定数を定義する為に使われ、予約語の**final**も付される:

code_05.2c.dart

```
class Greeter {
static final prefix = 'Hello,';

static greet(name) {
    print('$prefix $name');
  }
}

main() {
  print(Greeter.prefix);
  Greeter.greet('Class!');
}
/*
Hello,
Hello, Class!
*/
```

5.3節　　メソッドのカスケード呼び出し

メソッドのカスケード呼び出し(cascaded method invocation)は、SmallTalkで最初に導入されているが、これは2012年3月28日の仕様書改定第0.08版からDartにも採用された。これは同じようなメソッド呼び出し文が多く含まれるようなコードをより滑らか(fluent)に且つ手短に記述できるようにする。例えば:

```
myTokenTable.add("aToken");
myTokenTable.add("anotherToken");
// many lines elided here
// and here
// and on and on
myTokenTable.add("theUmpteenthToken");
```

といったコードは、

```
myTokenTable
  ..add("aToken")
  ..add("anotherToken")
// many lines elided here
// and here
// and on and on
  ..add("theUmpteenthToken");
```

という具合に書けるようになる。ここに、"**..**"はカスケード化されたメソッド(あるいはゲッタまたはセッタ)呼び出し操作を意味する。

以下は正規式をより言葉に近い表現で得るためのツール(パッケージ・ライブラリ verbal_expressions)の使用例である。パッケージに関してはパッケージ・マネージャの章を参考のこと。

```
var regex = new VerbalExpression()
 ..startOfLine()
 ..then("http").maybe("s")
 ..then("://")
 ..maybe("www.").anythingBut(" ")
 ..endOfLine();
```

このように分りやすい記述が可能となる。この文は以下のような正規式(regex)を生成する:

```
^http(s)?\\:\\/\\/(www\\.)?([^\\ ]*)\$
```

もう少しややこしい例を示そう。

code_05.3.dart

```
main() {
 Map mimeMap = const {
  'rtf': 'application/rtf',
  '3g2': 'video/3gpp2',
  'exe': 'application/octet-stream',
  'abs': 'audio/x-mpeg',
 };
 List keys = mimeMap.keys.toList()..sort();
 for (String key in keys) {
  print('extension: $key, mime type:${mimeMap[key]}');
 };
}
/*
extension: 3g2, mime type:video/3gpp2
extension: abs, mime type:audio/x-mpeg
extension: exe, mime type:application/octet-stream
extension: rtf, mime type:application/rtf
 */
```

これはあるマップの内容をキーをアルファベット順にソーティングしてプリントするものだが、

```
List keys = mimeMap.keys.toList()..sort();
```

は

```
List keys = mimeMap.keys.toList().sort();
```

と書いてはいけない。sortメソッドはkeysに対するメソッド、即ちレシーバはkeysだからである。

このような場合は

```
List keys = mimeMap.keys.toList();
     keys.sort();
```

と書いたほうが好ましい。

5.4節　　コンストラクタによるフィールドの初期化

次のようなPointというクラスのコンストラクタPoint(num x, num y)を考えてみよう:

```
class Point {
  num x, y;
  Point(num x, num y) {
    this.x = x;
    this.y = y;  }
}
```

このコードはthisを使って次のように簡単化される:

```
class Point {
  num x, y;
  Point(this.x, this.y);
}
```

ボディ部が空のコンストラクタでは{}を使わないでその代りに;でその宣言を終端できることに注意されたい。

イニシャライザ・リスト(initializer list)

コンストラクタでは**イニシャライザ・リスト**(initializer list)とボディを使って多様な初期化が可能である。イニシャライザ・リストはコロン(':')で始まり、カンマ(',')で分離されたイニシャライザたちのリストで構成される。イニシャライズ・リストはボディ部の前に定義すべきフィールドたちの値をセットするのに使われる。これはfinalなフィールドには必要である。詳細は仕様書を見て頂きたいが、簡単な例を以下に示す:

code_05.4.dart

```
class Point {
  num x, y, z;
  List v;
  Point(this.x, this.y) : z = 10 {
    v = [];
    for(var i=1; i<=5; i++) {
      v.add(i);
    }
  }
}
main() {
  var point = new Point(1, 2);
  print('x:${point.x} y:${point.y} z:${point.z}');
  for (var i=0; i<point.v.length; i++){
    print(point.v[i]);
  }
}
/*
x:1 y:2 z:10
1
2
3
```

```
4
5
*/
```

この例では変数zがイニシャライザ・リストにより10にセットされている。またvというリストは1から5までの値が追加されている。

イニシャライザには2つの種類がある:

- スーパーイニシャライザはスーパーコンストラクタ、即ちスーパークラスの特定のコンストラクタを指定する。スーパーイニシャライザの実行によりスーパーコンストラクタのイニシャライザ・リストが実行される。
- インスタンス変数イニシャライザは個々のインスタンス変数にある値を代入する。

5.5節　　常数コンストラクタ

常数コンストラクタ(constant constructor)はコンパイル時の常数オブジェクトを生成するのに使われる。従ってコンパイル時にその値が実行時に得られることが確定できなければならない。常数コンストラクタは予約語である**const**が先行している。即ち定数コンストラクタに対応したクラスでは、**new**の代わりに**const**を付すことでコンパイル時定数を生成できる。常数コンストラクタがボディをもっているときはコンパイル時エラーとなる。

constantConstructorSignature（常数コンストラクチャ・シグネチュア）:
const qualified（修飾） formalParameterList（仮パラメタ・リスト）
;

定数オブジェクト式(constant object expression)は定数コンストラクタを呼び出す。

constObjectExpression（定数オブジェクト式）:
const type（型） ('.' identifier（識別子）)? arguments（引数たち）
;

次のコードは定数コンストラクタによる初期化とその呼び出しを示している:

code_05.5a.dart

```
class A {
 final a1;
 final a2;
 const A(var x): a1=6, a2 = x + 5;
 String toString(){return "class A object";}
}
main(){
 var x = const A(5);
 print('a: ${x.toString()}, a1: ${x.a1}, a2: ${x.a2}');
}
/*
a: class A object, a1: 6, a2: 10
*/
```

var x = const A(5);で定数コンストラクタを呼び出しているが、これの引数の5の代わりに何か変数名を入れるとコ

ンパイラは定数でないとしてエラーを出す。しかしながら**const**の代わりに**new**を使った次のコードではエラーにはならない：

code_05.5b.dart

```
class A {
 final a1;
 final a2;
 A(var x): a1=6, a2 = x + 5;
 String toString(){return "class A object";}
}
main(){
 var p = 5;
 var x = new A(p);
 print('a: ${x.toString()}, a1: ${x.a1}, a2: ${x.a2}');
}
```

同一の2つのコンパイル時定数を生成すると、それらの定数は単一のインスタンスを参照することになる。

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a,b)); // これらは同じインスタンスを参照している！
```

identicalというdart:coreの関数は2つのオブジェクトが同じオブジェクトであるかどうかを調べる。

5.6節　　名前付きコンストラクタ

Dartではクラス名のあとにドット'.'を付し識別子を付けた「名前付きコンストラクタ(Named Constructor)」により、複数のコンストラクタを持たせることができる。例えば

code_05.6.dart

```
class Greeter {
  var prefix = 'Hello,';
  Greeter();
  Greeter.withPrefix(this.prefix);
  greet(name) {
    print('$prefix $name');
  }
}

main() {
  var greeter1 = new Greeter();
  greeter1.greet('Class');
  var greeter2 = new Greeter.withPrefix('Howdy,');
  greeter2.greet('Class');
}
```

```
/*
Hello, Class
Howdy, Class
*/
```

Greeter.withPrefix(this.prefix);という名前付きコンストラクタを宣言するには、デフォルトのコンストラクタであるGreeter();を明文化しておく必要がある。

次の例は名前付きコンストラクタを使った初期化リストを使ったフィールドの初期化（原点設定）、及びボディ部を伴ったコンストラクタによるフィールドの初期化（極座標入力）を示したものである:

```
class Point {
  num x, y;
  Point(this.x, this.y);
  Point.zero() : x = 0, y = 0;
  Point.polar(num theta, num radius) {
    x = Math.cos(theta) * radius;
    y = Math.sin(theta) * radius;  }
}
```

5.7節　　ファクトリ・コンストラクタ (Factory Constructor)

Dartではファクトリ・コンストラクタ(Factory Constructor)と呼ばれる新しい種類のコンストラクタが導入されている。

ファクトリ・コンストラクタは、必ずしも常にそのクラスの新規インスタンスを生成するわけではないコンストラクタを実装する際にfactoryキーワードを付す。ファクトリ・コンストラクタは例えばあるキャッシュからあるインスタンスを返すとかある副型のインスタンスを返すとかに使われる。

ファクトリは一見staticなメソッドのように見えるが、**明示的に指定したクラスのインスタンスを返す**という点で異なっている。ファクトリはコンストラクタのように呼び出すことができ、そのメソッドのボディは返されるインスタンスを制御できる。ファクトリは他の言語でのコンストラクタに関わる弱点に対処している。ファクトリは新規に割り当てられたものではないインスタンスを生成できる:即ちこれらのインスタンスはキャッシュから得られる。同様にまた、ファクトリは異なったクラスのインスタンスを返すことが出来る。また、そのクラスのインスタンスを初期化するコードから分離して追加の処理コードが必要になるときにもこのコンストラクタは使用できる。

ファクトリは組み込み識別子である**factory**が先行したスタチックなメソッドである。

factoryConstructorSignature（ファクトリ・コンストラクチャ・シグネチュア）:
　factory qualified（修飾）('.' identifier（識別子）)? formalParameterList（仮パラメタ・リスト）
;

ファクトリの名前は次のものでなければならない:

- コンストラクタ名
- そのファクトリが宣言されている場所でスコープ内にあるインターフェイスのコンストラクタの名前

ファクトリ・コンストラクタを呼び出すときは予約語の**new**を付す。

Dartチームは M1 変更のなかで、明示的なインターフェイスを無くした（「インターフェイス」の章を参照のこと）。これに伴い、APIドキュメントからはインターフェイスとデフォルト実装クラスで構成されていたインターフェイスは総て抽象クラスに書き変え、デフォルト実装はファクトリ・コンストラクタで構成している。それでもDartチームの中ではAPIドキュメントにある抽象クラスを「インターフェイス」と呼んでいる人が多い。

この場合の一般的なパタンは次のようになる。理解しにくいかもしれないが、詳細はディスカッションを見て頂きたい:

code_05.7a.dart

```
abstract class A {
  A._();
  factory A() => new B();
  foo() => 'method of A';
  bar();
}

class B extends A {
  B() : super._();
  bar() => 'method of B';
}

main() {
  print(new A().bar()); // method of B
  print(new B().foo()); // method of A
}
```

次の例はGoogleの技術者のSeth Laddが示しているDartにおけるシングルトン・パタンである

code_05.7b.dart

```
class Singleton {
  static final Singleton _singleton = new Singleton._internal();

  factory Singleton() {
    return _singleton;
  }

  Singleton._internal();
}

//You can construct it with new

main() {
  var s1 = new Singleton();
  var s2 = new Singleton();
  print(identical(s1, s2));  // true
  print(s1 == s2);           // true
}
```

2行目の_singletonはstaticでかつfinalなのでクラス変数であり、これはこのクラスの最初のインスタンス化で設定される。factory Singleton()というファクトリ・コンストラクタは従ってこのクラスでただひとつのオブジェクトを返す。main()のなかでnew Singleton();を呼ぶとこのファクトリ・コンストラクタが実行される。従って、s1やs2といったオブジェクトは全く同じオブジェクトとなる。

次の例はオブジェクトのキャッシュを使うものである。Googleの技術者のBob Nystromの記事で示されているものである。インスタンスをキャッシュにストアすることで、newが呼ばれるごとに毎回同じインスタンスを生成することによる負荷をなくし、高速化が図れる。:

code_05.7c.dart

```
class Symbol {
  final String name;
  static Map<String, Symbol> _cache;

  factory Symbol(String name) {
    if (_cache == null) {
      _cache = {};
    }

    if (_cache.containsKey(name)) {
      return _cache[name];
    } else {
      final symbol = new Symbol._internal(name);
      _cache[name] = symbol;
      return symbol;
    }
  }

   Symbol._internal(this.name);
}
```

- nameというString型の変数はfinal宣言されているので、Symbolクラスのインスタンスに固有の名前が与えられることになる。
- インスタンスたちのキャッシュはMapで構成され、各インスタンスは名前付きで出し入れされる。このキャッシュの名前は_cacheとアンダスコアが先行しているのでprivateとなっている。
- ファクトリはfactory Symbol(String name)というシグネチュアになっている。デフォルト(名前付きでない)のコンストラクタにfactoryというプレフィックスが付いていることで、Dartはそれはファクトリ・コンストラクタであることを知る。このボディ部では以下のことが行われる:
  - キャッシュがまだ存在しない、即ちnullの状態なら、要素が空のマップとする。
  - キャッシュが存在しているなら、そのキャッシュに指定された名前のインスタンスが存在するかを調べる
    - もし存在するなら、そのインスタンスをとりだして返す。
    - 存在しないならこのクラスのインスタンスを内部コンストラクタにより生成し、それを名前付きでキャッシュにストアするとともに、そのインスタンスを返す。
- これにより、キャッシュには指定された名前のインスタンスが必ずひとつストアされる。

例えば呼び出し側で次のような2つのインスタンスを同じ名前で生成したとする:

```
var a = new Symbol('something');
var b = new Symbol('something');
```

最初のaというインスタンスは新しいインスタンスを生成したものであるが、bというインスタンスはキャッシュされていたものである。プログラム開発の最初の段階では通常のコンストラクタを使っていて、必要に応じファクトリを用意すれば良い。呼び出し側のコードはその為に変更しなくても良い。

5.8節　　セッタとゲッタ (Setters and Getters)

オブジェクトの属性へのアクセスはobject.someProperty形式で可能であるのはJavaと同じであるが、Dartでは更にあたかも属性アクセスのようにして何らかの処理をさせることができる。つまりゲッタ/セッタでアクセスできるものとフィールドとはおなじようにアクセスできる。

getterSignature（ゲッタ・シグネチュア）:
　　static? returnType（戻りの型）? **get** identifier（識別子）
;
setterSignature（セッタ・シグネチュア）:
　　static? returnType（戻りの型）? **set** identifier（識別子）
;

戻り値が指定されていないときは、そのゲッタの型は**dynamic**である。またセッタ/ゲッタはメソッドをオーバライド出来ず、メソッドはまたセッタ/ゲッタをオーバライドできない。

以下に示すクラスは絶対値absとラジアン角argをフィールドに持った2次元ベクトルである。しかし場合によってはX軸長とY軸長でこのクラスを操作したいこともあろう。その為にxValとyValがあたかもフィールドであるかのように操作出来るようにセッタとゲッタが用意されている:

code_05.8a.dart

```
import 'dart:math' as Math;
class D2Vector {
  // fields
  num abs, arg;
  // constructor
  D2Vector(this.abs, this.arg);
  // xVal setter / getter
  num get xVal => abs * Math.cos(arg);
  set xVal(num x) {
    num y = abs * Math.sin(arg);
    abs = Math.sqrt(x * x + y * y);
    arg = Math.atan(y / x);
  }
  // yVal setter / getter
  num get yVal => abs * Math.sin(arg);
  set yVal(num y) {
    num x = abs * Math.cos(arg);
    abs = Math.sqrt(x * x + y * y);
    arg = Math.atan(y / x);
  }
}
main(){
  D2Vector myVec = new D2Vector(1, 0);
  print('abs: ${myVec.abs} arg: ${myVec.arg} xVal: ${myVec.xVal} yVal: ${myVec.yVal}');
  myVec.yVal = 1;
  print('abs: ${myVec.abs} arg: ${myVec.arg} xVal: ${myVec.xVal} yVal: ${myVec.yVal}');
}
/*
abs: 1 arg: 0 xVal: 1 yVal: 0
abs: 1.4142135623730951 arg: 0.7853981633974483 xVal: 1.0000000000000002 yVal: 1
*/
```

main()のなかでは、最初にmyVecというD2Vectorのオブジェクトを絶対値1.0、角度0で生成している。次のprint行ではmyVec.xValとmyVec.yValで、フィールドではない値をあたかもフィールドのように読みだしている。セットもmyVec.yVal = 1;のように、yValをセットしているが、実際はこれによりabsとargを変更している。ここではxValもyVal

も1になったので、absは√2でargはπ/2である。

このコードは説明のためのもので、実際には絶対値が0とかX長が0のときの処置、例えば例外をスローすることなどが必要である。

ゲッタとセッタはAPIの中で多用されている。これらを使うことでコンパクトなコーディングが可能になる。「Dartの実行」の節の「HTML5対応を試してみる」の項のコードが参考になろう。

また、クラスのフィールドに対するアクセスも暗示的なゲッタとセッタだと考えることもできよう:

code_05.8b.dart

```
class Location {
  num lat, lng;
}

void main() {
  var waikiki = new Location();
  waikiki.lat = 21.271488; // 暗示的なセッタ
  waikiki.lng = -157.822806; // 暗示的なセッタ

  print(waikiki.lat); // 暗示的なゲッタ
  print(waikiki.lng); // 暗示的なゲッタ
}
```

なお2012年7月の0.11版からはセッタ宣言及びゲッタ宣言で文法に変更がなされている。

ゲッタでは空のパラメタ・リストは不要となった。つまり:

```
int get theAnswer() => 42;
```

ではなくて

```
int get theAnswer => 42;
```

のように記述する。これにより定義の記述が実際に呼び出すときの記述により接近させている。

セッタでは名前の後に'='が必要になった:

```
set theAnswer = (int value) { print('The answer is now $value.'); }
```

これは将来実装予定のリフレクションでセッタ名に'='が付く為だと説明されている。

5.9節　　リダイレクト・コンストラクタ (Redirecting Constructor)

生成的コンストラクタはリダイレクション(インスタンス生成処理の振り向け)ができ、その場合は別の生成的コンストラクタを呼び出すだけである。リダイレクション・コンストラクタはボディ部を持たず、その代りそのリダイレクトでどのコンストラクタを呼び出すか、そしてどんな引数で呼び出すのかを指定する次の構文を持つ:

redirection(リダイレクション):

':' **this** ('.' identifier（識別子）)? arguments（引数たち）
;

あるコンストラクタの唯一の目的が同じクラスの別のコンストラクタに振り向けるだけという場合がある。振り向ける側のコンストラクタのボディ部は空であり、その後にコロンと振り向けられるコンストラクタの呼び出しが続く。

```
class Point {
  num x;
  num y;

  Point(this.x, this.y);                  // このクラスの為のメインのコンストラクタ
  Point.alongXAxis(num x) : this(x, 0); // このメインのコンストラクタに委譲する
}
```

この例ではX軸上のPointなので、Y軸は0である。従ってPoint.alongXAxis(num x)というコンストラクタは引数yを0にしてメインのコンストラクタを呼び出している。

5.10節　　演算子

Dartでは演算子は特別な名前を持つインスタンス・メソッドとなっている。詳細はAPI参照を見て頂きたい。例えばAPI仕様のnumというインターフェイスには数値に関する演算子がoperatorを使って定義されている。同様にObject、List、int、double、Stringなどでも演算子が定義されている。

なお、代入演算子については「代入」の節を参照されたい。また演算子の使用に関しては「式」の章も参照のこと。

operatorSignature（演算子シグネチュア）:
returnType（戻りの型）? **operator** operator（演算子） formalParameterList（仮パラメタ・リスト）
;

operator（演算子）:
unaryOperator（単項演算子）
| binaryOperator（二項演算子）
| '[' ']'
| '[' ']' '='
（注意：0.10版でcall、negate、equalsを削除）
;

unaryOperator（単項演算子）:
'!'
| '~'
;

binaryOperator（2項演算子）:
multiplicativeOperator（積算演算子）
| additiveOperator（加算演算子）
| shiftOperator（シフト演算子）

| relationalOperator（関係演算子）
| equalityOperator（イコール性演算子）
| bitwiseOperator（ビット演算子）
;

prefixOperator（前置演算子）:
'-'
| unaryOperator（単項演算子）
;

（注意：0.11版で否定演算子が廃止された）

演算	演算子
関係演算子、等しいかどうか	==
関係演算子、違っているかどうか	!=
関係演算子、小さいかどうか	<
関係演算子、大きいかどうか	>
関係演算子、小さいかまたは等しいか	<=
関係演算子、大きいかまたは等しいか	>=
加算演算子、加算	+
加算演算子、減算	-
増分演算子、増分	++
増分演算子、減分	--
積算演算子、除算	/
積算演算子、除算、整数を返す	~/
積算演算子、乗算	*
積算演算子、剰余	%
ビット演算子、論理和	|
ビット演算子、排他的論理和	^
ビット演算子、論理積	&
ビット演算子、左シフト	<<
ビット演算子、右シフト（符号伝搬）	>>
ビット演算子、ビット反転	~
リスト要素追加	[]=
リスト要素読み出し	[]
代入演算子（「代入」の節を参照のこと）	=, ??=
複合代入演算子（「代入」の節を参照のこと）	=, –=, /=, %=, >>=, ^=, +=, *=, ~/=, <<=, &=, |=
型テスト	is, is!

型キャスト	as

注意しなければならないのは、Dartではビット演算はintで定義されていることである。従ってビット列の一番左側(MSB)といちばん右(LSB)を連結したシフト回転はできない。またMSBが符号を意味する訳ではない。

code_05.10a.dart

```
main() {
  int x = 0xffffffff;
  int y = 4;
  print(x);      // 4294967295
  print(x >> y); // 268435455
  print(x << y); // 68719476720
  x = -0x7ffffff;
  print(x);      // -134217727
  print(x >> y); // -8388607
  print(x << y); // -2147483632
}
```

演算子の優先順位

演算子の優先順位は高いほうから次の順になっている:

記述(Description)	演算子(Operator) eは式を意味する	結合則(Associativity)	優先度(Precedence)
単項後置 (Unary postfix)	., ?., e++, e–, e1[e2], e1() , ()	なし	16
単項前置 (Unary prefix)	-e, !e, ~e, ++e, --e	なし	15
乗除 (Multplicative)	*, /, ~/, %	左	14
加減 (Additive)	+, -	左	13
シフト (Shift)	<<, >>	左	12
関係 (Relational)	<, >, <=, >=, **as**, **is, is!**	なし	11
等価 (Equality)	==, !=	なし	10
ビットAND (Bitwise AND)	&	左	9
ビットXOR (Bitwsie XOR)	^	左	8
ビットOR (Bitwise Or)	|	左	7

論理AND (Logical And)	&&	左	6
論理OR (Logical Or)	||	左	5
If-null	??	左	4
条件式 (Conditional)	e1? e2 : e3	なし	3
カスケード (Cascade)	..	左	2
代入 (Assignment)	=, *=, /+, +=, =+, ~=, %=, <<=, >>>=, >>=, &=, ^=	右	1

例えば、単項後置が単項前置よりも優先度が高いので、

```
print(-1.abs())  // 1
```

は1が出力されるのに対し、

```
var a = -1;
print(-a.abs()); // -1
```

は-1を出力する('.' が '-' より優先する)。

また括弧が最優先されるので、

```
main() {
  int i = 40;
  int j = 80;
  int k = 40;
  print(i / k * 2 + j);   // 82.0
  print(i /( k * 2) + j); // 80.5
}
```

となる。Javaとは違って除算が入っているので結果はdouble型となる。

演算子定義

*演算子(operators)*たちは特別な名前を持つインスタンス・メソッドたちである。組込み識別子operatorを使うとユーザが演算子を定義できる。

operatorSignature(演算子シグネチュア):
returnType(戻りの型)? **operator** operator(演算子) formalParameterList(仮パラメタ・リスト)
;

以下の名前たちはユーザ定義の演算子として許される: ==, <, >, <=, >=, -, +, /, ~/, *, %, |, ^, &, <<, >>, []=, [], ~, **call**, **negate, equals**。

組み込み識別子である**negate**は単項マイナスを意味するために使われる。

次のコードはPoint型のオブジェクトの加算演算子の定義と使用法を示したものである:

code_05.10b.dart

```
import 'dart:math' as Math;

class Point {
  Point(this.x, this.y);        // コンストラクタ
  var x, y;                     // インスタンス変数
  distanceTo(Point other) {     // インスタンス・メソッド
    var dx = x - other.x;
    var dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
  operator +(other) => new Point(x+other.x, y+other.y); // 演算子定義
}

main() {
  Point p1, p2, p3;
  p1 = new Point(5, 10);
  p2 = new Point(3, 4);
  p3 = p1 + p2;         // 定義された演算子を使用
  print('Added result :  X=${p3.x}, Y=${p3.y}');
  print('Distance : ${p1.distanceTo(p2)}');
}

/*
Added result :  X=8, Y=14
Distance : 6.324555320336759
*/
```

5.11節　　関数エミュレーション

仕様書の0.07から関数エミュレーション(function emulation)が追加された(実装はM2変更から)。関数エミュレーションはPythonで既に導入されている。オブジェクト指向の観点に立てば、各関数はオブジェクトであり、変数に代入でき、関数の戻り値や引数にもなり得る。実際Scalaは関数としてのクラスというコンセプト(object functional styleともいう)に基づいている。

あるクラスを関数として使い、自分の関数型を実装させるには次のようなステップを踏む:

1. callという名前のメソッドを持ったクラスを定義する
2. ()シンタックスで関数として呼び出されたときにそのクラスのインスタンスが何をするかを定義する為にcall()メソッドを実装する
3. 良いスタイルはそのクラスにFunction抽象クラスを実装させることである

call()メソッド

Dartではcallという新しいメソッドを導入して、クラスを関数の代役として使えるようにしている。callはどんな種類の仮パラメタ・リストをも持つことができる。あるクラスの中でcallメソッドを定義することで、関数のエミュレーションが可能となる。例えば:

code_05.11a.dart

```
class WannabeFunction {
  call(String a, String b) => '$a $b';
}

main() {
  var wf = new WannabeFunction();
  wf("Hello", "World" );
  print(wf); // "Hello World"
}
```

ここではWannabeFunction(関数になりたい)というクラスの中で、callは引数の2つの文字列を連結して返す演算子として定義されている。この関数をエミュレートしたクラスを使うには、これをインスタンス化して所定の引数をセットして呼び出せば良い。

この例はあまり意味が無く、それなら直接関数を書いたほうが良い。しかしながらこうできることが非常に有用な場合がある。またこれはまたDart言語の設計哲学のコアにもなっている:

- オブジェクトで問題となるのはその振る舞いである。オブジェクトaが他のオブジェクトbのそれと互換性を持つ手続き的インターフェイスを持っているとしたら、aはbに置き換え可能である。
- どの種のオブジェクトのインターフェイスも、常にしかるべく定義された別のオブジェクトによってエミュレートできる。

Dartは$x(a_1, a_2, \ldots, a_n)$という式を計算するときに、それが通常の関数の場合はその関数を呼び出す。そうでないときはxがcallに対応していればそのcallを呼び出す。noSuchMethod()のデフォルト実装が演算子ではなくてメソッド名callの為に呼ばれたかどうかをチェックし、もしそうならクロージャ使用(f.callのような使いかた)の為に使われたのではないかとして警告を出す。

このクラスに条件設定などの為のコンストラクタやメソッドを用意してやれば、条件に応じてこの関数の動作を変えることも可能になる。

apply()メソッド

GoogleのDartチームのGilad Brachaはその記事の中で、これの使い方の良いスタイルとしてFunction抽象クラスの実装を示している。実際の関数オブジェクトはFunction抽象クラスを実装しており、それにはapply()というメソッドのみが用意されている:

```
interface Function {
  apply(ArgumentDescriptor args);
}
```

ここにDescriptorはその関数への引数(位置的引数たちと名前付き引数たち)の記述子である。applyメソッドは次のようなシグネチュアになっている:

```
external static apply(Function function,
  List positionalArguments,
  [Map<String, dynamic> namedArguments]);
```

apply()メソッドにより、関数を汎用的な形で呼び出すことができるようになる。関数エミュレーションの為のクラスの最も良い作り方はFunctionを実装して自分の為のapply()メソッドを定義することである。例えば整数を2倍する関数として:

```
class Int2Int implements Function {
  apply(ArgumentDescriptor args) => this(args.positionalArguments[0]);
  // この場合では引数は位置的パラメタひとつのみ
  operator call(int a) => a*2; // 別のシグネチュアを持った関数
}
```

もっと簡単にはapplyを次のように定義する:

```
apply(args) => this.call.apply(args);
```

しかしながらFunctionの実装は必須ということではない。そのcallメソッドをクロージャ化してそれにapplyメソッドを呼ぶことで、ArgumentDescriptorを使ってあるオブジェクトfを関数として呼び出すことは可能である:

```
f.call.apply(args);
```

関数の型

ところでエミュレートする関数の型をどう指定すれば良いのだろうか？以下はその例である:

```
 typedef StringFunction(a,b);
...
new WannabeFunction() is StringFunction; // true
```

従ってあるオブジェクトのクラスがcall()というメンバ・メソッドを持っているときは該オブジェクトはある関数型のメンバであり、そのメソッドはその関数型のメンバであることを定めている。

noSuchMethod()との関わり合い

Dartではオブジェクトたちが自分たちのクラスのチェインの中で明示的に定義されていないメソッドたちに対して如何に反応するかをObjectクラスの中で定義されているnoSuchMethod()メソッドをオーバライドすることでカスタム化できる。noSuchMethod()メソッドの内部で関数エミュレーションをどのように使用するのかの例を示すと:

```
noSuchMethod(InvocationMirror msg) =>
  msg.memberName == 'foo' ? msg.invokeOn(bar())
                          : Function.apply(baz,
                              msg.positionalArguments,
                              msg.namedArguments);
```

2行目にあるinvokeOn()はその呼び出しを特定のオブジェクト（ここではbar()の結果）に転送したいという一般的

な事例を取り扱っている。その後の3行は単にそのパラメタたちを別の関数に転送したいという事例を処理している。もしbazという関数が名前付き引数をとらないことが分かっていれば、そのコードはFunction.apply(baz, msg.positionalArguments)となる。

noSuchMethod()への引数はInvocationMirrorのみであり、それは現在次のように定義されている:

```
abstract class InvocationMirror {
  String get memberName;
  List get positionalArguments;
  Map<String, dynamic> get namedArguments;
  bool get isMethod;
  bool get isGetter;
  bool get isSetter;
  bool get isAccessor => isGetter || isSetter;

  invokeOn(Object receiver);
}
```

isXXXの名前を持ったInvocationMirrorのブール値属性たちは、以下に示す表に従ってそのメソッド呼び出しの文法的書式を特定している:

	メソッド呼び出しの書式		
	x.y	x.y = e	x.y(...)
isMethod	FALSE	FALSE	TRUE
isGetter	TRUE	FALSE	FALSE
isSetter	FALSE	TRUE	FALSE
isAccessor	TRUE	TRUE	FALSE

isMethodであることは非アクセサが検索されていることを意味していると考えてはいけないことは重要であり、これはDartのセマンティクスでは通常のメソッドもゲッタも見つからないときにのみnoSuchMethod()が呼ばれたということを意味しているからである。同様に、isGetterはゲッタが検索されていることを意味しない;もしあるメソッドが存在していればそれがクロージャ化され、返される。

5.12節　　リフレクション (Mirror Based Reflection)

リフレクション(Reflection)はJavaScriptとJavaにも存在するので、読者の中には馴染みの人もいよう。しかしながら、Dartでは言語仕様担当のBrachaが提案していたミラー(Mirrors)のコンセプトをベースとしている。

リフレクションはプログラムそのものをデータ(あるいはオブジェクト)として扱う、即ちあるプログラムの機能を調べたり(introspection)、あるオブジェクトの機能や構造を実行時に変更したり(Dynamic Evaluation)できるようにするもので、VMやインタープリタが実行するようなプログラミング言語でサポートされることが多い。但し処理時間がかかるので、テスト用に使われることが多い。

ミラーという用語はリフレクション(反射)のための鏡から来ている。即ちミラーは他のオブジェクトをリフレクトするオブジェクトである。ユーザはあるクラスやオブジェクトをリフレクトさせたいときは、それらに対するミラーと呼ばれ

るオブジェクトを取得しなければならない。ミラー・ベースにすることで各エンティティごとにミラーを用意するという面倒くささがあるものの、そのプログラムの、セキュリティ、配布および配備に関して有利なAPIとなっている。

dart:mirrorsというライブラリはDartのリフレクションのためのライブラリであり、以下の機能をサポートしている:

- Introspection:実行中のプログラム自身の構造を調べることが可能なリフレクションのサブセットである。例えば、任意のオブジェクトの総てのメンバたちの名前をプリントする関数など。
- Dynamic evaluation:例えば、「ある引数としてその名前が与えられるメソッドを呼び出す」といったコンパイル時に文字として指定されたことがない(その名前がデータベース検索から得られるとかユーザがインタラクティブに指定するとかといった理由で)コードを調べる機能を言う。

このライブラリのドキュメントを読むときは以下のことに注意する必用がある:

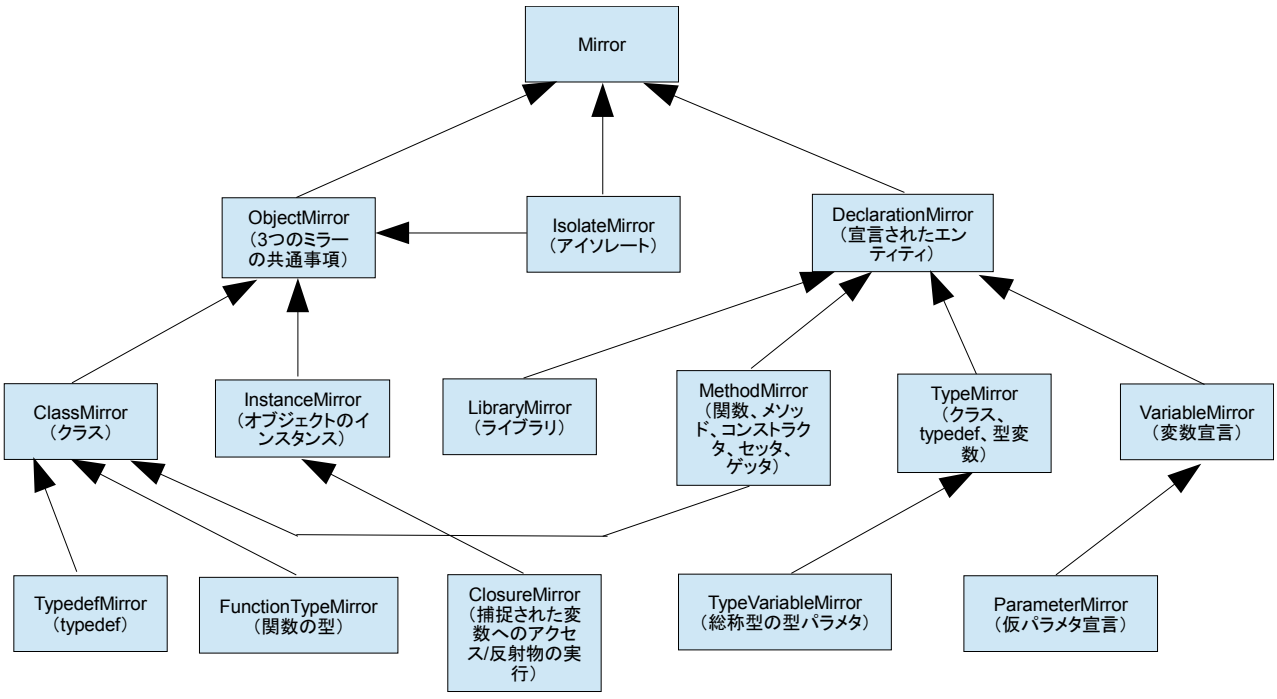
- このライブラリはSymbolクラスのインスタンスを使ってDartの宣言物たちの名前を表現している。従ってこのAPIドキュメントで「クラスSymbolのオブジェクトs」と記述しているときは、それはsを組み立てるときに使われるときに使われた文字列を意味する。

- またこのドキュメントではo.x(a)といった疑似コード(oとaはオブジェクト)といった記述をしばしば使っているが、これは実際はo'とa'がoとaにバインドされたDartの変数たちであるo'.x(a')のことを意味している。更にo'とa'は新鮮な変数(このプログラムの中の他の変数たちとは分離している事を意味する)だと仮定している。

- またこのドキュメントでは直列化可能オブジェクト(serializable objects)を参照していることがある。直列化可能オブジェクトは、num、bool、String、アイソレートたちに亘って直列化可能なオブジェクトたちのリストまたはキーをもったマップたち、およびアイソレートたちに亘って総てが直列化可能な値たちに限られる。

- 現時点(2014年4月)ではdart:mirrorsライブラリはまだ未完状態である。

ミラーには次のようなものがある(注意:今後追加・変更の可能性大):



あるクラスおよびその現在のインスタンスに対するミラーの使用法の簡単な例を以下に示す:

code_05.12a.dart

```
import 'dart:mirrors';

class MyClass {
  int i, j;
  void my_method() {  }

  int sum() => i + j;

  MyClass(this.i, this.j);

  static noise() => 42;

  static var s;
}

main() {
  MyClass myClass = new MyClass(3, 4);
  InstanceMirror myClassInstanceMirror = reflect(myClass);

  ClassMirror MyClassMirror = myClassInstanceMirror.type;

  InstanceMirror res = myClassInstanceMirror.invoke(#sum, []);
  print('sum = ${res.reflectee}');

  var f = MyClassMirror.invoke(#noise, []);
  print('noise = $f');

  print('\nMethods:');
  Iterable<DeclarationMirror> decls =
  MyClassMirror.declarations.values.where(
          (dm) => dm is MethodMirror && dm.isRegularMethod);
  decls.forEach((MethodMirror mm) {
    print(MirrorSystem.getName(mm.simpleName));
  });

  print('\nAll declarations:');
  for (var k in MyClassMirror.declarations.keys) {
    print(MirrorSystem.getName(k));
  }

  MyClassMirror.setField(#s, 91);
  print(MyClass.s);
}
```

このプログラムの実行結果を以下に示す:

```
sum = 7
noise = InstanceMirror on 42

Methods:
my_method
sum
noise

All declarations:
i
j
s
my_method
sum
noise
MyClass
91
```

- reflectはdart:mirrorsライブラリでトップ・レベルの関数として定義されており、引数で指定したオブジェクトのInstanceMirrorを返す。
- im.type属性はそのInstanceMirrorオブジェクトのClassMirrorである。即ち、インスタンス・ミラーからクラス。ミラーを取得している。
- InstanceMirrorのinvokeメソッドは指定した関数を呼び出して、その結果に関するmirrorを返す。invoke(#sum, [])はsum()という関数を引数なしで呼び出している。結果としてのmirrorがresという変数に代入される。
- InstanceMirrorのrefrectee属性は実インスタンスへのアクセスを提供している。ここでは実インスタンスのsum()の結果、すなわち7という数値が返される。
- これに対しクラス・メソッドの場合はClassMirrorのinvokeメソッドを使うことになる。MyClassMirror.invoke(#noise, []);はnoise()というクラス・メソッドを呼び出している。結果は42という数値にある。
- このクラスのメソッドの一覧は、ClassMirror.declarations.valuesを使う。where((dm) => dm is MethodMirror && dm.isRegularMethod)はMethodMirror型でRegularMethodであるもののみを指定している。その各々のMethodMirrorに対しMirrorSystem.getName()でその名前を取得している。
- そのクラスの宣言の総てはClassMirror.declarations.keysを使う。
- そのクラスのフィールドをセットすることもできる。ClassMirror.setField(#s, 91);はsという名前クラス変数に91という値をセットしている。

5.13節　　オブジェクトへの属性の付加

クラスではなくあるオブジェクトに対し動的に属性を付加したいこともあろう。DartではExpando<T>というクラスが用意されている。

例えば次の例を見てみよう:

code_05.13.dart

```
//Expando Sample
class Person {
  String name;
  Person(this.name);
}
main() {
  var me = new Person('Terry');
  var nationality = new Expando();
  nationality[me] = 'Japan';
  print('${me.name} : ${nationality[me]}');
  var age = new Expando();
  age[me] = {'age': 50};
  print('${me.name} : ${age[me]["age"]}');
}
/*
Terry : Japan
Terry : 50
*/
```

これはPersonというクラスに対してではなく、そのオブジェクトに対しnationalityとかageとかいう名前の属性を付加

した例である。

対象となるオブジェクトは以下のものであってはいけない:

- bool
- num
- String
- null

付加するオブジェクトはこの例にあるように、Stringなどだけでなく、Mapなども使用できる。

JavaScriptでは総てのオブジェクトexpandだともいえるが、DartではExpandoオブジェクトを生成しなければならない。

5.14節　　ティアオフ (Tear-offs)

ティアオフとはその名の通りあるオブジェクトのメソッドを引き剥がしてクロージャ化することである。

次の例を見てみよう:

code_05.14.dart

```
class Friendly {
 String name;
 Friendly(this.name);
 sayHi() {
  print("Hi, I'm $name!");
 }
}
main(){
 var friendly = new Friendly("Terry");
 var tearOff = friendly.sayHi; // メソッドの引き剥がし
 tearOff();
}
```

main()関数の中の2行目ではsayHiメソッドを引き剥がしている。そのメソッドを直接呼び出しているのではなく、それが呼ばれたときに該メソッドを呼び出すクロージャ(第1級関数)を返している。

Dartでは括弧で囲まれた引数リストが存在しないときに、それはそのメソッドを呼び出すのではなくてティアオフだと判断している。これはC#、Python、JSなどと同じである。しかしながらこれは引数を持たないゲッタ等では機能しない。

Dart言語仕様書第3版では、これが一般化され、'.'の代わりに'#'を使ってレシーバとセレクタを切り分けるようにしている(これまでどうりメソッドに対する'.'は残されている)。

第6章　インターフェイスと抽象クラス (Interfaces and abstract classes)

インターフェイス(interface)はユーザがあるオブジェクトとどのように関わり合えるかを定義する。はメソッドたち、ゲッタたち、セッタたち及びコンストラクタたち、及びスーパーインターフェイスたちのセットを持つ。クラス宣言においては、複数のクラスを継承(extends)することはできないが、複数のインターフェイスを実装(implements)することができる。

注意：Dart仕様のM1版ではシンプル化の一環として明示的なインターフェイスは削除された。従ってinterfaceというキーワードはなくなった。これは6.2節で示すように、総てのクラスに暗示的なインターフェイス(implicit interface)を持たせたからである。またクラスは抽象メソッドを持つことも可能になる。また明示的にインターフェイスを書かなくても抽象クラスを書くことで等価な結果が得られる。

従ってこれまでのインターフェイス定義は今後は抽象クラス及び抽象インスタンス・メンバで行うこととなる。DartのAPIはこれに伴いすべてのインターフェイスはクラスへ切り替えられた。これらの詳細は提案書に記されているので参照されたい。

抽象クラス(abstract class)は具体クラスによって継承（または実装）されるもので、内部を公開しないAPIなどでよく使用される。抽象クラスは空のメソッド（名前と引数だけからなる）を持つことができる。

6.1節　　シンプルな例

GoogleのチュートリアルにあったコアにあるComparable抽象クラスの実装例を次に示す：

code_06.1.dart

```
class Greeter implements Comparable {
  String prefix = 'Hello,';
  Greeter() {}  // default constructor
  Greeter.withPrefix(this.prefix); // named constructor
  greet(String name) => print('$prefix $name'); // print greet
  int compareTo(Greeter other) => prefix.compareTo(other.prefix); // compare prefixes
}

void main() {
  Greeter greeter = new Greeter();
  Greeter greeter2 = new Greeter.withPrefix('Hi,');

  num result = greeter2.compareTo(greeter);
  if (result == 0) {
    greeter2.greet('you are the same.');
  } else {
    greeter2.greet('you are different.');
  }
}
/*
Hi, you are different.
*/
```

（注意：これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと）

- インターフェイスの実装
  Greeter（挨拶するプログラム）クラスはコア・ライブラリのComparable抽象クラスを実装しており、ソーティング等の目的のために2つのGreeterオブジェクトを比較出来るようにしている。この抽象クラスにはint compare(Comparable a, Comparable b)というスタティック・メソッド及びabstract int compareTo(Comparable other)というメソッドが現在定義されていて、APIではDateTime、Duration、num、およびStringがこれを実装している。この抽象クラスを実装するには2つのステップを踏む：クラス・ステートメントのなかにComparable実装を付加（行1）、及びComparableが必要とするメソッドのみcompareTo()の定義の追加（行6）がなされている。Stringという抽象クラスは既にComparableを実装しているので、これのcompareTo(Comparable other)を利用すれば良い。
- main()のなかでは、デフォルトのコンストラクタでprefixが'Hello 'というGreeterのインスタンスgreeterを、名前付きコンストラクタでprefixが'Hello 'というGreeterのインスタンスgreeter2を生成している。
- 新しく定義したcompareToメソッドを使って、2つのインスタンスのprefixが同じかどうかを調べる。
- 同じであれば'you are the same.'を、そうでなければ'you are different.'をプレフィックスに連結させてプリントしている。

注意）
intとdoubleはプリミティブな型と読者は考えるかもしれないが、これらは実際は抽象クラスであってnum抽象クラスを継承したものである。このことは、intとdoubleの変数もnumであることを意味する。
重要：numは初期化をすること。これらはオブジェクトであるので、これらの初期値は0ではなくてnullである。
使用上はintとdoubleは読者が多分馴染んでいるプリミティブ型のように感じられる。例えば、これらの値をセットするのに文字列を使うことが出来る：

```
int height = 160;
double rad = 0.0;
```

6.2節　　総てのクラスはインターフェイスでもある

簡素化という目的のもとでなされたDartのM1変更では、各クラス定義はまたインターフェイスを暗示的に定義していることになった（但しGo言語の暗示的インターフェイスとは異なる）。クラス定義により、そのクラスが持っているメソッドたちのパブリックな構文（即ちインスタンス・メンバたちの宣言のボディ部分を除いたもの）を記述したあるインターフェイスを暗示的に定義していることになる。

従ってクラスを継承できるだけでなくそのクラスを実装できる。つまりBの実装を継承することなくBのAPIが使えるクラスAを作りたいときは、クラスAはBを（継承するのではなくて）実装すれば良い。だから明示的なインターフェイス定義なしでもDartが書ける：インターフェイスを定義したい場合は、抽象クラス(abstract class)を宣言することで済ませることができる。何らかのメソッドのデフォルト実装を用意したい場合は、そのボディ部を書くだけで良い。実際Dartチームでは自分たちのコードをインターフェイスではなくて抽象クラスで書くように切り替えを実施した。DartのAPIでは、その抽象クラスを実装したクラスを探さなくても、その抽象クラスから直接オブジェクトを生成できる。これは抽象クラスがファクトリ・コンストラクタを持っているからである。

次のクラス定義を考えてみよう：

```
class Person {
  final _name;
  Person(this._name);
```

```
  void greet(who) => 'Hello $who, I am $_name.';
}
```

ここではPersonという具体クラスを定義していて、greetというメソッドと_nameというフィールドを持っている。これを例えば次のように関数の中で使うことができる:

```
greetBob(Person person) => person.greet('bob');
```

このPersonというクラスとは別の新たなImposter(でしゃばり屋？)というクラスを定義したいとする。このクラスはPersonをそのまま継承するのではなく、greetメソッドの動作を変えたいとする。その場合は、extendsではなくてimplementsを用いてPersonをインターフェイスとして活用できる:

```
class Imposter implements Person {
  void greet(who) => 'Hello $who, it is a pleasure to meet you.';
}
```

以下はその具体的なコードである:

```
// person :この暗示的インターフェイスは greet()を含んでいる
class Person {
 // このインターフェイス内の属性、このライブラリ内でのみ可視
 final _name;
 // これはコンストラクタ故インターフェイスには含まれない
 Person(this._name);
 // このインターフェイス内のメソッド
 String greet(who) => 'Hello, $who. I am $_name.';
}

// Person インターフェイスの実装
class Imposter implements Person {
 // 定義する必要があるが、このコードでは使われない
 final _name = "";
 String greet(who) => 'Hi $who. Do you know who I am?';
}

greetBob(Person person) => person.greet('bob');

main() {
 print(greetBob(new Person('kathy'))); // Hello, bob. I am kathy.
 print(greetBob(new Imposter())); // Hi bob. Do you know who I am?
}
```

Dartはまたクラスの中に抽象メソッドをおくことを許している。ボディ部のないメソッドは抽象メソッドとして扱われる。従って抽象メソッドと暗示的なインターフェイスがあれば、明示的なインターフェイスを用意する必要性が無くなる。総てのメソッドを抽象メソッドとしたクラスをつくり、その暗示的なインターフェイスを実装することで同じ効果が得られる。

単に簡素化だけでなく、これによりJavaのインターフェイスのような静的メンバが持てないという制約が無くなる。またAPIの設定にあたって、インターフェイスとして定義するかあるいは抽象クラスとして定義するかを悩む必要も無くなる。

抽象クラスの例

前述のように抽象クラス(abstract class)は具体クラスによって継承（または実装）されるもので、内部を公開しないAPIなどでよく使用される。抽象クラスはabstractという前置詞（プレフィックス）を持つ。抽象クラスは抽象メソッド（名前と引数だけからなる）を持つことができる。

以下はその例である：

code_06.2.dart

```
abstract class Describable {
 void describe();
 void describeWithEmphasis() {
  print('========');
  describe();
  print('========');
 }
}

class MyTitle extends Describable {
 void describe() => print("General Manager");
}
class MySubtitle implements Describable {
 void describe() => print("Development Division");
 void describeWithEmphasis() {
  print('********');
  describe();
  print('********');
 }
}

main(){
 new MyTitle().describeWithEmphasis();
 new MySubtitle().describeWithEmphasis();
}
```

Describableという抽象クラスはdescribe()という抽象メソッドと、そのdescribe()を使ったdescribeWithEmphasis()というメソッドを有している。

具体クラスはこれを継承または実装する：

- implements（実装する抽象クラスのメンバの総てを実装のこと）
- extends（継承する抽象クラスの未実装メソッドを実装のこと）

MyTitleというクラスはDescribable抽象クラスを継承しているので、describe()というメソッドを具体化すればよい。しかしながらDescribable抽象クラスを実装したクラスMySubtitleではdescribeWithEmphasis()というメソッドも具体化しなければならない。

abstract修飾詞を外したらどうなるであろうか？アナライザは以下のような警告をする：

```
'describe' must have a method body because 'Describable' is not abstract
```

しかしながらこのまま実行しても正しい結果が得られる。

第7章　ミクスイン (Mixins)

ミクスインはM2仕様書から導入された。ミクスイン(Mixins)はあるクラスのコードを複数のクラス階層の中で再利用することであるクラスに対しある機能を付加する手段のひとつである。ミクスインを使用するにはwithキーワードを使いそのあとに複数のミクスインの名前が続く:

```
mixins（mixinたち）:
    with typeList（型リスト）
  ;
```

例えば以下のコードはミクスインを使った2つのクラスを定義している:

```
abstract class Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}

class Musician extends Performer with Musical {
  // ...
}

class Maestro extends Person with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

MusicianはMusicalを使用し、MaestroはMusical、Aggressive、及びDementedを使用している。

次節に示すように、ミクスインはObjectを継承し、コンストラクタを持たず、またsuper呼び出しを持ってはいけない。Musicalというミクスインは抽象クラスとして定義されている。

次のコードを見てみよう:

code_07.0.dart

```
main() {
  var myIcecream = new MyIcecream();
  print(myIcecream.onTop()); // Here is a list of toppings
}
```

```
class Icecream{
}

class Topping {
  List chocolateToppings, nutToppings, dropsToppings;
  String onTop() {
    return 'Here is a list of toppings';
  }
}

class MyIcecream extends Icecream with Topping {
  MyIcecream() : super();
}
```

自分が欲しいアイスクリーム(MyIceceam)はアイスクリーム(Icecream)を継承したものである。即ちMyIcecreamのスーパークラスはIcecreamである。自分のアイスクリームにはトッピング(Topping)を乗せたいとする。このトッピングがミクスインになる。即ちIcecreamにToppingを付加したものを継承したものがMyIcecreamである。このように、ミクスインはスーパークラスとの差分を構成していると考えることができる。

注意:

```
MyIcecream() : super();
```

は初期化リスト(initializer list)の記述で、ここではIcecreamのコンストラクタを呼び出している。

当初の仕様ではミクスインには以下の3つの制約が課されていた(7.2節を見て頂きたい):

1. 該クラスはコンストラクタが宣言されていない
2. 該クラスのスーパークラスはObjectである
3. 該クラスはスーパークラス呼び出しを含んでいない

しかしながら、これは1.13版からはそのうち2つは外されている:

- Object以外のクラスからの継承が可能になった
- super()呼び出しが可能となった

但しこれらの「スーパー・ミクスイン」はDart2JSではまだサポートされていないし、DartAnalizerでは--supermixinというフラグが必要である。

＊＊＊

この章の残りの部分は仕様担当のGilad Brachaが書いた解説(Mixins in Dart)の翻訳である。ミクスインの基本に関してはWikipediaの日本語版なども参考にされたい。

＊＊＊

7.1節　　基本的なコンセプト

Mixinに関する学術的文献に馴染んでいる読者の場合はこの章を飛ばして構わない。そうでない場合は、ここでは重要なコンセプトと表記が記されているので、この章を読んで頂きたい。もっと詳しく学術的にこれを調べたい場合はGilad BrachaによるMixins in Strongtalkという論文を読まれたい。

クラスと継承に対応している言語においては、クラスは暗示的にミクスインを定義している。ミクスインは通常は暗

示的である--これはそのクラスのボディ部によって定義され、クラスとそのクラスのスーパークラス間の差分を構成している。クラスは実際「ミクスインのアプリケーション」である--その暗示的に定義されたミクスインをそのスーパークラスに適用した結果である。

「ミクスインのアプリケーション」という表現は関数アプリケーションとの密なアナロジから来ている。数学的には、あるミクスインMはスーパークラスからサブクラスへの関数と見做せる：Mをスーパークラスsに渡し、Sの新しいサブクラスが返される。このことはしばしば学術的な記述では*M |> S*として書かれる。

関数アプリケーションという概念にたてば、関数構成を定義できる。このコンセプトはミクスインの構成にもあてはめられる；われわれはM1とM2の二つのミクスインたちの構成（*M1 * M2* と書く）を*(M1 * M2) |> S = M1 |> (M2 |> S)*と定義する。

関数は、異なった引数たちに適用できる為有用なものである。同じことがミクスインにも言える。あるクラスによって暗示的に定義されたミクスインは、通常ただ一度クラス宣言の中で与えられたそのスーパークラスに適用される。異なったスーパークラスたちへの適用を可能とするためには、他の個々のスーパークラス毎にミクスインを宣言するか、あるいはあるクラスの暗示的なミクスインを遊離させそれをそのオリジナルの宣言の外で再使用するかが出来なければならない。Dartでは以下に記すようにそのことが提案されている。

7.2節　　構文と意味

ミクスインは通常のクラス定義を介して暗示的に定義される。原則的には、各クラスはそこから抽出できるミクスインを定義する。しかしながら本提案では、ミクスインは以下の制約に従ったクラスからのみ抽出される：

4. 該クラスはコンストラクタが宣言されていない
5. 該クラスのスーパークラスはObjectである
6. 該クラスはスーパークラス呼び出しを含んでいない

制約1はコンストラクタのパラメタたちを継承チェインのわたって渡す必要があることに伴う複雑さを回避している。そのような状況に於いて、制約2はミクスインを明示的に宣言することを奨励している。制約3はミクスインのアプリケーションに応じ再構築するか、あるいは動的にそれらをバインドするかよりは、実装側が引き続き静的にスーパー呼び出しをバインドできることを意味する。

例1：

```
abstract class Collection<E> {
    Collection<E> newInstance();
    Collection<E> map(f) {
     var result = newInstance();
     forEach((E e){result.add(f(e));})
     return result;
  }
}

typedef DOMElementList<E> = abstract DOMList with Collection<E>;

typedef DOMElementSet<E> = abstract DOMSet with Collection<E>;

// ... 28 more variants
```

ここに、Collection<E>はミクスイン宣言に使われている通常のクラスである。DOMElementListとDOMElementSetのクラス双方ともミクスインのアプリケーションである。これらはtypedef宣言で定義されており、名前を与えているとともに、with句によってミクスインのスーパークラスへのアプリケーションと等しいことを宣言している。このクラスはCollectionのなかで定義されている抽象メソッドのnewInstance()を実装していないので抽象クラスである。

上記に於いて、DOMElementListは実効的に*Collection mixin |> DOMList*であり、一方DOMElementSetは*Collection mixin |> DOMSet*である。

ここでの利点はCollectionクラスのコードが多重クラス階層のなかで共有され得るということである。我々は上記に於いて2つのそのような階層をリストしている--ひとつはDOMListがルートとなっており、もうひとつはDOMSetをルートとしている。Collectionの中のコードを繰り返す/コピーする必要が無く、またCollectionになされた各変更は双方の下位層に伝搬され、そのコードの保守が非常に楽になる。この例は現実とは密に基づいたものではなく、Dartのライブラリの中では存在していないものである。

上記の例はミクスインのアプリケーションのひとつの形式を示したものであり、ここでは該ミクスイン・アプリケーションはそれに対し適用するミクスインとスーパークラスを指定しており、そのアプリケーションに名前を与えている。

代替的な形式では、識別子たちをカンマで区切ったクラス宣言のwith句のなかでミクスインのアプリケーションが出現する。総ての識別子たちはクラスを指定していなければならない。この形式の場合は、複数のミクスインが構成され、extends句のなかで指名されたスーパークラスたちに適用され、ひとつの匿名のスーパークラスを造る。同じ例を使うと、次のようになろう:

```
class DOMElementList<E> extends DOMList with Collection<E> {
   DOMElementList<E> newInstance() => new DOMElementList<E>();
}

class DOMElementSet<E> extends DOMSet with Collection<E> {
   DOMElementSet<E> newInstance() => new DOMElementSet<E>();
}
```

ここに、DOMElementListは*Collection mixin |> DOMList*アプリケーションではない。そうではなくてこれはそのスーパークラスがそのようなアプリケーションである新しいクラスである。DOMElementSetに関する状況も類似的である。各事例に於いて抽象メソッドnewInstance()は実装に於いてオーバライドされており、従ってこれらのクラスは直接インスタンス化出来ることに注意されたい。

もしDOMListが意味があるコンストラクタを持っていると何が起きるかを考えよう:

```
class DOMElementList<E> extends DOMList with Collection<E> {
   DOMElementList<E> newInstance() => new DOMElementList<E>(0);
   DOMElementList(size): super(size);
}
```

各ミクスインが独立して呼び出されるそれ自身のコンストラクタを持っており、スーパークラスもそうなっている。ミクスインのコンストラクタは宣言出来ない為、その呼び出しは文法により削除され得る; 下位実装のなかでは、この呼び出しは常に初期化リストな最初に置かれ得る。

該コンストラクタはフィールドたち及び総称型パラメタたちに値をセットし得る。

この規則により、これらの例がスムースに走り、また制約(1)が外されればきれいに一般化される。

第2の形式は、複数の仲介的な宣言を持ちこむことなくあるクラス内に複数のミクスインたちをミックス・イン出来るという利便性を持ったものである。例えば:

```
class Person {
  String name;
  Person(this.name);
}

class Maestro extends Person with Musical, Aggressive, Demented {
  Maestro(name):super(name);
}
```

ここでは、このスーパークラスはミクスイン・アプリケーションである:

Demented mixin |> *Aggressive mixin* |> *Musical mixin* |> *Person*

我々はPersonのみが引数を持ったコンストラクタを持っていると仮定している。*Musical mixin* |> *Person*がPersonのコンストラクタを継承しているので、ミクスインのアプリケーションたちのシリーズとして構成されているMaestroの実際のスーパークラスまでそうなっている。

実際にはこの例では我々はDemented、Aggressive、及びMusicalは実際に状態を必要としそうな面白い属性たちを持つことが期待される。

7.3節　　問題の可能性

予想される以下の問題に関する幾つかの領域を論じる:

- ·Privacy
- Statics
- Types

プライバシ (Privacy)

ミクスインのアプリケーションはオリジナルのクラスを宣言したライブラリの外部で宣言され得る。このことはミクスイン・アプリケーションのインスタンスのメンバたちを誰がアクセスできるかということに関して何ら効果を持つべきではない。メンバたちへのアクセスはそれらが最初に宣言されたライブラリに基づいて、まさしく通常の継承と同じく、決まる。厳密にいえば、これはミクスインのアプリケーションの語義に、即ち下位層言語のなかで継承の語義によって決まっていること、に基づいているので、自分はこれを取り上げることさえも必要無い。

Statics

ミクスイン・アプリケーションを介してオリジナルのクラスのstatic物を使うことが可能か否か？

ここでも、この答え（Noという）は継承の語義に従っている。Dartではstatic物は継承されない。

型

ミクスイン・アプリケーションのインスタンスの型は何だろうか？一般的にはこれはそのスーパークラスの副型であり、またそのミクスインで定義されているメソッドたちに対応している。しかしながらそのミクスインの名前それ自身はオリジナルのクラスの型を示しており、それはそれ自身のスーパークラスを持ち特定のミクスインのアプリケーションとは互換性を持たないかもしれない。

あるクラスが対応しているインターフェイスに関してはどうだろうか？そのミクスインはそれらに対応するのだろうか？一般にはインターフェイス対応は継承された機能に依存できるので、それは否である。このことはミクスイン・アプリケーションはそれがどのインターフェイスを実装しているかを明示的に宣言しなければならないことを意味する。

我々は当面は安全にこの問題を無視できよう。制約(2)があるので、あるミクスインの型は、該ミクスインによって宣言されているあるいは総てのオブジェクトたちによって共有されているものたちを超えて更なるメンバたちを含むことは無い。たとえ該ミクスインがインターフェイスたちを実装しているとしてもそのミクスイン自身はそのインターフェイスたちのメソッドたちを実装しなければならず、従って該ミクスインの中にミックスインするどれもがそのミクスインの名前で示された完全な型の副型であると考えても良い。

しかしながら制約(2)が外されたときには、この問題が生起しよう。

（以下省略）

第8章　列挙型と総称型 (Enums and Generics)

8.1節　　列挙型 (enums)

列挙型は2014年12月にECMA-408の第2版（Dart 1.8）から第13章としてGenericsの前に追加された。章番号を変更したくない為、本書ではこの章の先頭にenumの解説を追加したものである。

*列挙型(enumerated type)*または*enum*は固定数の定数値たちを表現するのに使われる。

構文

enumType（列挙型）:
metadata（メタデータ） **enum** id（識別子） ‘{’ id [‘, ’ id]* [‘, ’] ‘}’
;

metadata **enum** E { id_0, . . . id_{n-1}};の形式の宣言は次のクラス宣言と同じ効果を持つ:

```
metadata class E {
final int index;
const E(this.index);
static const E id0 = const E(0);
. . .
static const E id_{n-1} = const E(n - 1);
static const List<E> values = const <E>[id0 . . . idn−1];
String toString() => { 0: ‘E.id0’, . . ., n-1: ‘E.idn−1’}[index]
}
```

enumをサブクラス化する、ミクスインする、または実装する、またはあるenumを明示的にインスタンス化するのはコンパイル時エラーである。

基本的な使い方

列挙型の宣言は予約語のenumを付して次のように行う:

```
enum Color {
  red,
  green,
  blue
}
```

前項(構文)で示したように、各要素には0から始まるインデックスが付されている:

```
assert(Color.red.index == 0);
assert(Color.green.index == 1);
assert(Color.blue.index == 2);
```

Enumsはvaluesゲッタを使ってListに変換できる:

```
List<Color> colors = Color.values;
assert(colors[2] == Color.blue);
```

列挙型に対しSwitch文を使うときは、総ての要素を処理しないと警告が出る:

code_08.1a.dart

```
enum Color {
  red,
  green,
  blue
}

main() {
  Color aColor = Color.blue;
  switch (aColor) {
    case Color.red:
      print('Red as roses!');
      break;
    case Color.green:
      print('Green as grass!');
      break;
    default: // これがないと警告が出る
       print(aColor);
  // 'Color.blue'
  }
}
```

列挙型に対する注意点は:

- 列挙型をサブクラス、ミクスイン、あるいは実装することはできない
- 列挙型は明示的にインスタンス化できない

ことである。

インデックスは次のように使うことができる(出典:stackoverflow)。

code_08.1b.dart

```
enum Status {
  none,
  running,
  stopped,
  paused
}

void main() {
  print(Status.values);
  Status.values.forEach((v) => print('value: $v, index: ${v.index}'));
  print('running: ${Status.running}, ${Status.running.index}');
  print('running index: ${Status.values[1]}');
}
/*
[Status.none, Status.running, Status.stopped, Status.paused]
value: Status.none, index: 0
value: Status.running, index: 1
value: Status.stopped, index: 2
```

```
value: Status.paused, index: 3
running: Status.running, 1
running index: Status.running
*/
```

8.2節　　総称型 (Generics)

Genericsは筆者がその昔使っていた強い型づけのプログラミング言語Adaで最初に導入された（当時は「汎用体」と訳していた）ものだが、Javaにも2004年にJ2SE 5.0で導入されているので、Javaの技術者には特に説明の必要がなかろう。但しDartの総称型はJavaと違って具体化された総称型(reified generics)に対応している。即ち総称型のオブジェクトたちはそれらの型引数たちの情報を実行時に保持する。総称型のコンストラクタに型引数を渡すのはランタイムの操作である。しかしながらDartは型づけがオプションとなっている。従って<型>という記述は書かなくてもDartは走り、むしろ大規模あるいは高信頼性アプリで意図を明確化するために使用される。それをどう調和させるかが問題となる。この件は「Dartの型処理と型チェック」の節でも触れてある。

インスタンス生成

Dartでは型づけなしでのプログラム作成を可能としており、また実行時は型アノテーションを無視している。従ってDartでは型パラメタを与えなくても総称型クラスのインスタンスの生成が可能なようにしている。例えば

```
new List();
```

と書いても構わない。型指定をしたいときは正規の記述をすれば良い。例えばString型で使うときは

```
new List<String>();
```

と書く。総称型を型指定しないでインスタンス生成すると、Dartはそれを

```
new List<dynamic>();
```

だと解釈する。ここに**dynamic**は未知(unknown)という意味の型である。

コンストラクタ内では、型パラメタは実行時に必要であり、それらは実行時に渡されるので、型テストの為の演算子であるisが使える（Javaではreifiedでないので型情報が無くinstanceOfは使えないし、例えばList<String>の配列も作れない）:

```
new List<String>() is List<Object>  // true: 各文字列はオブジェクトである
new List<Object>() is List<String>  // false: 総てのオブジェクトが文字列ではない
```

更に幾つかの例を示すと:

```
new List<String>() is List<int>     // false
new List<String>() is List          // true
new List<String>() is List<dynamic> // 上と同じ
```

```
new List() is List<dynamic>          // true まさしく同じもの
new List() is List<String>           // true 未知の型はどの型にも対応できる
```

Dartでの総称型の取り扱い

Dartではどのように総称型を扱っているか調べてみよう。

code_8.2.dart

```
class A<T> {
  final T a;
  A(T this.a);
  String toString(){return 'class A object, a = $a';}
}

int main(){
  A x = new A<int>(3.14); // Warning: double is not assignable to int
  print(x.a is int);
  print(x.toString());
  x = new A<String>('Hi');
  print(x.toString());
}
/*
false
class A object, a = 3.14
class A object, a = Hi
*/
```

- Aという総称型のクラスはaというインスタンス化時に指定された型の値を保持する変数をもつ。またtoStringというaの値を含めた文字列を返すメソッドを持っている。
- main()のなかでは最初にint、次にString型を指定したAのインスタンスを生成してxという変数にしている。
- 注意すべきことは、最初のコンストラクタ呼び出しでdoubleの値を引数にするとチェック・モードで警告は出されるが処理は正しく処理される。しかもx.aの型はdoubleではなくてintだと報告している。
- この呼び出しをnew A<int>('3.14')としてもチェック・モードでは実行が止まるが非チェック・モードだとそのまま実行を続け、同じ結果が出力される。

第9章　メタデータ (MetaData)

メタデータ(MetaData)はプログラム・コードに付加的情報を与えるものである。メタデータは仕様書の0.11版から10章として新たに導入された。これによりDartはユーザが定義したアノテーションをプログラム構造に付加する為に使われるメタデータに対応するようになった。これはJavaのアノテーションに相当する。

metadata（メタデータ）:
('@' qualified（修飾された） ('.' identifier（識別子）)? (arguments（引数たち）)?)*
;

メタデータは一連のアノテーションたちで構成され、その各々は文字@で始まり以下コンパイル時定数（例えばdeprecated）への参照、または定数コンストラクタ呼び出しが続く。

メタデータはライブラリ、クラス、typedef、型パラメタ、コンストラクタ、ファクトリ、関数、フィールド、パラメタ、あるいは変数宣言の前に、及びインポートまたはエクスポート指令の後に置かれ得る。

2015年1月時点で@deprecated、@override及び@proxyという3つが属性としてdart:coreに登録されている。自分のライブラリを使用するには以下のように自分のライブラリをインポートする。

```
import 'todo.dart';
```

9.1節　　@deprecated

APIドキュメントには次のように記されている:

クラス、ゲッタ、セッタ、メソッド、トップレベル変数、あるいはトップレベル関数にたいし、もはや使えなくなっているものだということをマークする為に使われるアノテーション。ツールたちはこのアノテーションを使ってこのマークされた要素への参照に対する警告を提供できる。

具体的には自分のIDEがこれをどのように実装しているかを以下に説明する:

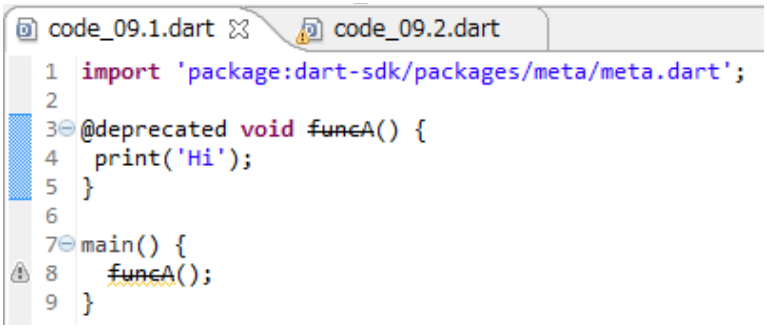
最初に次のようなコードがあったとする:

```
void funcA() {
 print('Hi');
}

main() {
  funcA();
}
```

プログラマがトップレベル関数のfuncAはもう不要になったと判断したときは、このアノテーションをこの関数定義に付加する。そうするとDart Editorでは次のようにその関数名をストライク・アウト表示してくれる:

また!のシンボルではTop-level function 'funcA' is deprecatedと表示してくれる。

code_09.1.dart



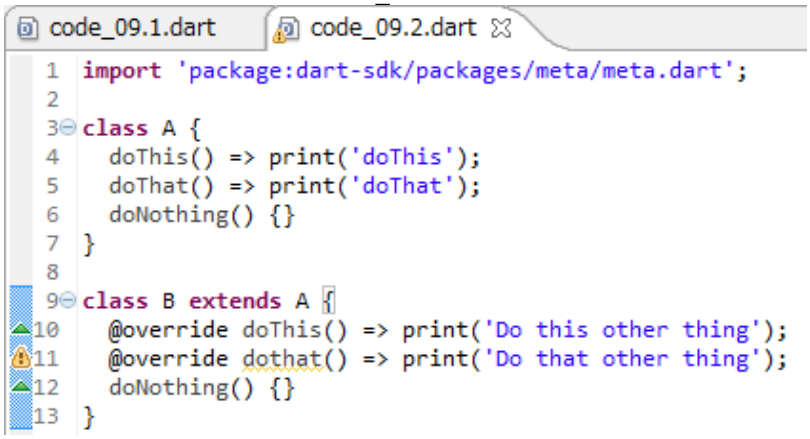
9.2節　　@override

APIドキュメントには次のように記されている:

インスタンス・メンバたち(メソッド、フィールド、ゲッタ、またはセッタ)が継承しているクラスのメンバをオーバライドしていることをマークするのに使われるアノテーション。ツールたちはこのアノテーションを使ってオーバライドされているメンバが存在しないときにそれを警告することができる。

具体的には開発ツールである自分のIDEがこれをどのように実装しているかを下図で説明する:

code_09.2.dart



ここではプログラマはAというクラスを継承したBというクラスはdoThis及びdoThatというメソッドをオーバライドしようとして間違ってdoThatをdothatと書いてしまった例である。@overrideアノテーションによりDart EditorではdothatというメソッドはオーバライドしていないとしてMethod marked with @override, but does not override any superclass elementという警告をする。なお10行目と12行目ではオーバライドしていることをoverrides A.doThisのように教えてくれる。

9.3節　　@proxy

@proxyアノテーションは、あるクラスに対しそのメンバたちをnoSuchMethodメソッドを介してダイナミックに実装させるようにする。このアノテーションはどのクラスに対しても適用させることができる。このアノテーションはスーパークラス及びインターフェイスからのサブクラスによって継承される。

あるクラスが@proxyでアノテートされているとき、あるいは@proxyでアノテートされたクラスを実装しているときは、該クラスは静的型解析に関してはどのメンバも実装していると見なされる。従って、該クラスによって実装されていない該オブジェクトのメンバにアクセスするとき、あるいは宣言されているとは異なったパラメタたちの数とは異なった数であるメソッドを読んだときには、静的型警告は出されない。

このアノテーションは、アノテートされたクラスがどのクラスたちを実装するかに関しては変更を与えないし、該オブジェクトによって実装されていない静的型でもってあるオブジェクトをある変数に代入することに対する静的警告を抑制しない。

警告の抑制はメンバへのアクセスに関する静的型警告にのみ影響を与える。該オブジェクトの実行時の型には影響を与えない。特別なインターフェイスを実装しているとは見なされないので、それをある型づけされた変数に代入するとチェックド・モードで引っかかり、演算子でそれをテストするとそれが実際に実装または拡張した型にのみtrueを返す。該クラスが実装していないメンバにアクセスすると通常noSuchMethodが呼ばれ、この@proxyアノテーションは単にこれらのnoSuchMethod呼び出しをスマートに処理する意図を単に示している。

@proxyとマークされたクラスはObjectで宣言されている noSuchMethodをオーバライドしなければならない。

@proxyの意図はコンパイル時には判っていないある型（あるいは複数の型たち）を実装したオブジェクトを生成することである。これらの型がコンパイル時に判ればこれらの型たちを実装したクラスを書くことができる。

一番シンプルな例は次のようなものである：

```
@proxy
class C { noSuchMethod(i) { print("GOTCHA!"); } }
main() {
 C c = new C();
 c.foo();
}
```

このプログラムは"GOTCHA!"と出力するが、@proxyアノテーションを外すとクラスCにはfoo()というメンバ・メソッドが定義されていないので、The method 'foo' is not defined for the class 'C'という静的警告が出される。

9.4節　　@observable

これはWeb UIの為のメタデータである。

Web UIは「Web UIのパッケージ」の項で簡単に紹介してある。Web UIはDart Web Compiler (dwc)と呼ばれるコンバータで構成されており、このコンバータがこのメタデータを検出し、そのフィールドや変数たちにその変更を記録する為のゲッタとセッタを生成する。

以下はこのメタデータを持つPersonというクラスの例である：

```
@observable
class Person {
  String name;

  Person(this.name);
}
```

なおWeb UIは2013年7月時点で新しいPolymerプロジェクトに切り替えられつつあるので注意されたい。

9.5節　　ユーザが自分のアノテーションを用意する

ユーザは自分のメタデータのアノテーションを定義できる。次の例は2つの引数を持った@todoアノテーションである。この類のアノテーションは開発の段階で追ってそこの個所を(この例では誰が何を)詰めることを管理するのに良く使われる。

code_09.5a.dart

```
library todo;

class todo {
 final String who;
 final String what;

 const todo(this.who, this.what);
}
```

以下はこの@todoアノテーションを使った例である:

code_09.5b.dart

```
import 'code_09.5a.dart';

@todo('seth', 'make this do something')
void doSomething() {
 print('do something');
}

main() {
 doSomething();
}
```

第10章 式 (Expressions)

式はDartコードの一部であり、ある値（*value*：常にオブジェクトである）を引き出す為に実行時に計算される。各式はそれに結び付けられたある静的な型を持つ。各値はそれに結び付けられたある動的な型を持つ。

expression（式）:
　assignableExpression（代入可能式） assignmentOperator（代入演算子） expression（式）
| conditionalExpression（条件式）
;

expressionList（式リスト）:
　expression（式） (',' expression（式）)*
;

primary（プライマリ）:
　thisExpression（this式）
| **super** assignableSelector（代入可能セレクタ）
| functionExpression（関数式）
| literal（リテラル）
| identifier（識別子）
| newExpression（new式）
| constantObjectExpression（定数オブジェクト式）
| '(' expression（式） ')'
;

10.1節　　定数、null、数、ブール値

予約語のtrueとfalseは各々ブール値の真と偽を表現するオブジェクトを意味する。これらはブール値リテラルである。

booleanLiteral（ブール値リテラル）:
　true
| **false**
;

trueとfalseの双方とも組込みインターフェイスboolを実装している。これらはboolのただ二つのインスタンスである。

あるオブジェクトはブール変換が可能である。これはフロー制御に使われる。

*ブール変換(boolean conversion)*は以下に定めるように何らかのオブジェクト*o*をブール値にマップする：

```
(bool v){
        assert(null != v);
        return true === v;
}(o)
```

この定式化がJavaScriptとは劇的に異なっていて、JavaScriptでは殆どの数とオブジェクトはtrueと解釈される。
Dartのアプローチではif (a-b) … ;といった使用法を許さない。

定数式

定数式(constant expression)はその値が変わらず、コンパイル時に完全に計算できる式である。

定数式は以下のもののどれかである:

- リテラル数値
- リテラルブール値
- 文字列リテラルで文字列内挿入（$による文字列インターポレーション）がコンパイル時定数のもの
- null
- staticでfinalな変数またはトップ・レベル変数またはクラスへの参照
- 定数コンストラクタ呼び出し
- 定数リスト・リテラル
- 定数マップ・リテラル
- トップ・レベル関数または静的メソッドを示す修飾識別子
- 定数式を括弧で括ったもの
- **identical**(e_1, e_2)の形式の式でe_1, e_2が定数式
- e1 ==e2またはe1 != e2の形式のひとつの式で、ここでe1とe2は数値、文字列、またはブール値を計算する定数式
- e1 && e2またはe1 || e2の形式のひとつの式で、ここにe1とe2はブール値を計算する定数式
- ~ e, e1 ~/ e2, e1 ^ e2, e1 & e2, e1 | e2, e1 >> e2またはe1 << e2の形式のひとつの式で、ここにe1とe2は整数値を計算する定数式
- e1 + e2, e1 - e2, e1 * e2, e1 / e2, e1 >e2, e1 < e2, e1 >= e2, e1 <= e2またはe1 % e2の形式のひとつの式で、ここにe1とe2は数値を計算する定数式

リテラルには次のものがある:

- nullリテラル
- ブール値リテラル
- 数値リテラル
- 文字列リテラル
- マップ・リテラル
- リスト・リテラル

これらについては別途説明する。

null

nullオブジェクトは組み込みクラスNullの唯一のインスタンスである。

code_10.1a.dart

```
main(){
  int a;
  print(a);
```

```
  a = null;
  print(a);
  a = 1;  // deleting this line will cause unhandled exception
  a++;
  print(a);
}
/*
null
null
2
*/
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

このコードで判るようにint a;と宣言しただけではaはnullの状態である。nullにnullを代入してもnullである。nullの状態でこれをインクリメントなどの操作をしようとするとunhandled exceptionが生じる。

null対応の演算子

1.12版からnull対応の演算子(Null-Aware Operators)が新たに導入されている。解説はブログに記載されている:

```
// y が null でなければ y を、そうでなければ z を x に代入する
x = y ?? z;

// x が null でなければ y を x に代入する
x ??= y;

// x が null でないときに限り foo を呼ぶ
x?.foo();
```

- ??
  もしexpの計算結果がnullでなければその結果を、そうでなければotherExpの計結果を返したいときは??を使用する。
  ```
  exp ?? otherExp
  ```
  は次の式と等価である:
  ```
  ((x) => x == null ? otherExp : x)(exp)
  ```

- ??=
  そのオブジェクトがnullであるときに限りvalueを代入し、そうでないときはそのオブジェクトを返したいときは??=を使う。
  ```
  obj ??= value
  ```
  は次の式と等価である:
  ```
  ((x) => x == null ? obj = value : x)(obj)
  ```

- ?.
  そのオブジェクトがnullでないときに限り、あるメソッドやゲッタを呼び、そうでないときはnullを返したいと

きに?.を使用する。

```
obj?.method()
```

は次の式と等価である:

```
((x) => x == null ? null : x.method())(obj)
```

?.はチェインにできる。例えば:

```
obj?.child?.child?.getter
```

ではobjまたはchild1またはchild2がnullならこの式はnullを返し、そうでないときはgetterが呼ばれその結果が返される。

簡単なコード例を示そう:

code_10.1aa.dart

```
main() {

 // The ?? operator returns the first expression IFF it is not null
 var monday = 'doctor';
 var tuesday;
 var next = tuesday ?? monday;
 print('next appointment: $next');

 // the ??= operator assigns a value IFF it is not null
 var wednesday;
 next ??= wednesday;
 print('next appointment: $next');

 // the ? operator calls a function IFF the object is not null
 String thursday;
 // print(thursday.length); // null object does not have a getter 'length'
 print('length: ${thursday?.length}'); // ok

}
```

3っ目のブロックではthursday.lengthをそのまま呼ぶとnull object does not have a getter 'length'というエラー・メッセージが出される。thursday?.lengthとすれば、thursdayがnullならnullとなり、そうでなければゲッタのlengthが呼び出だされ、この問題が解決される。従って、このコードは次のような結果を出力する:

```
next appointment: doctor
next appointment: doctor
length: null
```

数リテラル

数リテラル(numeric literals)はサイズが固定されていない(JavaScriptに変換される場合は-2^{53}から2^{53}の範囲)10進または16進の整数 : int、または倍精度の10進数(64ビット浮動小数点 : double)である。

numericLiteral(数リテラル):
NUMBER

```
    | HEX_NUMBER
    ;

NUMBER（数）:
    DIGIT+ ('.' DIGIT+)? EXPONENT?
    | '+'? '.' DIGIT+ EXPONENT?
    ;

EXPONENT（指数部）:
    ('e' | 'E') ('+' | '-')? DIGIT+
    ;


HEX_NUMBER（16進数）:
    '0x' HEX_DIGIT+
    | '0X' HEX_DIGIT+
    ;

HEX_DIGIT（16進桁）:
    'a'..'f'
    | 'A'..'F'
    | DIGIT
    ;
```

数リテラルがプレフィックス‘0x’または‘0X’で始まるときは、それは16進整数リテラルで、‘0x’（以下‘0X’もおなじ）に続くリテラルの部分により表現された16進整数を意味する。そうでないときは、もしその数リテラルが小数点を含んでいないときは10進整数リテラルであることを意味し、10進整数を意味する。そうでないときは、その数リテラルはIEEE 754標準で規定された64ビット倍精度浮動小数点数を意味する。

以前は整数リテラルまたは倍精度リテラルはオプショナルにプラス・サイン(+)を頭に付すことが出来た。これは意味的な効果は無い。Dartには単項プラス演算子はない。しかしながら明確性及びJavaScriptとの多少の互換性の為に10進数リテラルの先頭のプラスを許していた。2012年8月時点ではこれは認められていない。0.12版仕様書では正式に単項プラスは削除されている。

整数は固定した長さの域に制限されていない。Dartの整数は真の整数で、32ビットまたは64ビットあるいは他の固定域表現ではない。これらのサイズは実装物で利用できるメモリにのみ制限を受ける。但し-2^{53}と2^{53}の範囲外の整数ではDartのVMとJavaScript (Dart2JS)では取り扱いが異なってくるので注意が必要である。

code_10.1b.dart

```
class NumberSyntax {
  f() {
    1; 12; 123;
    1.0; 12.0; 123.0;
    .1; .12; .123;
    1.0; 12.12; 123.123;

    1e1; 12e12; 123e123;
    1e+1; 12e+12; 123e+123;
    1e-1; 12e-12; 123e-123;

    1.0e1; 12.0e12; 123.0e123;
    1.0e+1; 12.0e+12; 123.0e+123;
```

```
    1.0e-1; 12.0e-12; 123.0e-123;

    .1e1; .12e12; .123e123;
    .1e+1; .12e+12; .123e+123;
    .1e-1; .12e-12; .123e-123;

    1.0e1; 12.12e12; 123.123e123;
    1.0e+1; 12.12e+12; 123.123e+123;
    1.0e-1; 12.12e-12; 123.123e-123;

    1.1234e+444;
    -1.1234e+444;
    +1.1234e+444;

    0x0; 0x1; 0x12; 0x123; 0x12345;
    0X0; 0X1; 0X12; 0X123; 0X12345;
    0x9A; 0x9a; 0X9A; 0x9a;
    0x9abcde; 0X9ABCDE;
    0x0123456789abcdef;
    -0x0123456789abcdef;
//  +0xfedcba9876543210; // no unary plus operator in Dart

  }
}
main(){}
```

整数は10進数と16進数の表示のリテラルしか許されない。むろん出力は8進数なども使える。例えばnumインターフェイスのString toRadixString(int radix)を使うと、次のように10進数の255を16進数のFF、8進数の377と出力する。

code_10.1c.dart

```
main() {
print(255.toRadixString(16));
print(255.toRadixString(8));
}
/*
ff
377
*/
```

なおn進数表記文字列（数リテラルではない！）をintに変換するには次のようにparseメソッドを使用する:

code_10.1d.dart

```
var str = "654a1661a99aff";

main() {
  print(int.parse(str, radix:16));
}

// 28510432636017407
```

ブール値

予約語の**true**と**false**は各々ブール値の真と偽を表現するオブジェクトを意味する。これらはブール値リテラル(boolean literal)である。

booleanLiteral(ブール値リテラル):
true
| **false**
;

あるオブジェクトoのブール変換は次のように行われる。

(**bool** v){
assert(v != **null**);
return v === **true**;
}(o)

非nullのオブジェクトまたは非ゼロの数をtrueとして取り扱われるJavaScriptなどとは異なり、Dartではbool型のオブジェクトはtrue、false、またはnullの値しか持てない。bool値で無いものは総てfalseと判断される。またbool値を得る式はtrueまたはfalseの値しかもたらさない。bool型の変数にはbool型の値しか代入出来ない。

以下はそれらの例である:

code_10.1e.dart

```
main() {
  var a = true;
  bool b = true;
  bool c = false;
  bool d = null;
  String s = 'abc';
  if (a) print('a is $a');
  if (b) print('b is $b');
  if (c == false) print('c is $c');
  if (d == null) print('d is $d');
  if (d == false) print('d is $d');
  if (s != true) print ('s is $s');
  }

/*
a is true
b is true
c is false
d is null
s is abc
*/
```

JavaScriptのようにif(x - y) print(x - y);というような文は許されず、if(x-y == 0) print(x - y);と書かねばならない。

code_10.1f.dart

```
main() {
  try {
    var x = 1;
    var y = 1;
    if(x-y == 0) print(x - y);
    if(x - y) print(x - y);
  } on Exception catch(e){print(e);}
}
/*
0
Unhandled exception:
type 'int' is not a subtype of type 'bool' of 'boolean expression'.
*/
```

10.2節　　文字列

文字列(string)は有効なUTF-16のコード単位(code units)の並びである。以前の仕様書では文字列として Unicode Normalization Form C（Unicode正規化形式C）に正規化すると書かれていたが、残念ながら現在は Javaなどと同じUTF-16となっている。これはJavaScriptへのクロス・コンパイルの制約からそうなったのだろう。

stringLiteral（文字列リテラル）:
(multilineString（複行文字列）
| singleLineString（単行文字列）)+
;

文字列は単行文字列または複行文字列のどちらかになれる。

singleLineString（単行文字列）:
' "' stringContentDQ* '"'
| ''' singleContentSQ* '''
| 'r' ''' (~(''' | NEWLINE))* '''
| 'r' '"' (~('"' | NEWLINE))* '"'
;

単行文字列はソース・コードの1行以上にわたれない。単行文字列は相互に対応したシングル・クオート(''')またはダブル・クオート('"')で終端される。

複行文字列

複行文字列は次のような構文になる:

multilineString（複行文字列）:
'"""'stringContentTDQ*'"""' |

```
    “”””stringContentTDQ*“””|
        ’r’ """""  (~ """""")* """"" |
      ’r’ """ (~ """)* """
      ;
```

code_10.2a.dart

```
main() {
// 単行文字列（注意：+記号は不要となった）
  String a = "abcde" 'fghijk';
// 複行文字列（注意：+記号は不要となった）
  String b = """QWERTY
qwerty"""  '''AIUEO
aiueo''';
  print('a: $a');
  print('b:$b');
}
/*
a: abcdefghijk
b:QWERTY
qwertyAIUEO
aiueo
*/
```

エスケープ・シーケンス

文字列は特別な文字の為のエスケープ・シーケンスに対応している。これらのエスケープは次のようである（'\'は日本語のコンピュータでは通常円記号'¥'になる）：

- \n は改行(newline)を示し、\x0Aと等しい
- \r はキャリッジ・リターンを示し、\x0Dと等しい（通常の端末では\nと\rは同じとして扱われる）
- \f はフォーム・フィードを示し、\x0Cと等しい
- \b fはバックスペースを示し、\x08と等しい
- \t はタブを示し、\x09と等しい
- \v は垂直タブを示し、\x0Bと等しい
- \x*HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$、は\u{ *HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$}と等しい
- \u*HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$ *HEX_DIGIT*$_3$ *HEX_DIGIT4*、は \u{ *HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$ *HEX_DIGIT*$_3$ *HEX_DIGIT*$_4$}と等しい
- \u{*HEX_DIGIT_SEQUENCE*} は*HEX_DIGIT_SEQUENCE*で表現されるユニコードのスカラ値である。*HEX_DIGIT_SEQUENCE*の値が有効なユニコード・スカラ値でないときはランタイム時エラーである
- $ は補完（インターポレート）された式の始まりであることを意味する
- それ以外は、*k*はその文字が{*n, r, f, b, t, v, x, u*}のなかに無いどの文字*k*であることを示す

次のコードはそのサンプルである：

code_10.2b.dart

```
main() {
  print('\\ " ');
  print("\\ ' ");
```

```
  print('abc\ndef');              // エスケープ改行
  print(r'abc\ndef');             // 生文字列
  print('\a\c\d\e\g');            // 非エスケープ文字
  print('\x43\x72\x65\x73\x63'); // ASCII 1バイト文字
  print('\u{43}\u{72}\u{65}\u{73}\u{63}');
  print('\u30AF\u30EC\u30B9');   // Unicode 2バイト文字
  print('\u{30af}\u{30ec}\u{30b9}');
}
/*
\ "
\ '
abc
def
abc\ndef
acdeg
Cresc
Cresc
クレス
クレス
*/
```

文字列内挿 (String Interpolation)

それらの式が計算され、結果の値が文字列に変換され、それを包んでいる文字列に連結するように、式を文字列リテラルに埋め込むことが可能である。このプロセスは文字列内挿（または文字列インターポレーションあるいは文字列補間ともいう）として知られるものである。

STRING_INTERPOLATION（文字列内挿入）:
'$' IDENTIFIER_NO_DOLLAR（$を含まない識別子）
| '$' '{' Expression（式） '}'
;

文字列内のエスケープしない$文字が挿入された式の始まりの印となる。$印のあとに以下のどれかが続く:

- $文字を含んではいけない識別子*id*
- 中括弧{}で終端された式

$id**の形式は**${id}の形式と等価である。文字列内挿入された文字列's_1**${e}**$s_2$'は'$s_1$' + *e*. **toString**() + 's_2'と等価である。同様に+が文字列連結演算子だとすれば、文字列内挿された文字列"s_1**${e}**$s_2$"は"$s_1$" + *e*. **toString**() + "s_2"と等価である。

挿入内の式自身が文字列を含む、即ち再帰的に再度文字列内挿入することは可能である。

code_10.2c.dart

```
main() {
  print('Nested string ${'interpolation ${'example'}.'}');
}
```

```
/*
Nested string interpolation example.
*/
```

隣接文字列の連結

2012年2月中頃からStringインターフェイスで+演算子を使わなくても隣接文字列が連結出来るようになった。これは:

```
var msg = 'hello ' + 'world';
```

の代わりに:

```
var msg = 'hello ' 'world';
```

と書くことが許されるものである。これにより複数行になってしまうような長い文字列を書くときに、いちいち行ごとに+を付けなくて良くなるという利点がある。

オーバロードされたStringの+演算子は2012年3月26日（0.08版）の仕様書改定で廃止された（6月17日から対応されなくなった）。従ってDartのコードを書くときは、文字列の連結に+は使えなくなっていた。しかしながら2013年3月にDartチームはString.concatをString.operator+に戻すと発表した。これにより文字列の連結に+演算子も使えることとなった。これは文字列内挿入の促進効果があったのに残念だとの意見があったが、結局実施されている。

生文字列

どの文字列もrをその先頭にプレフィックスが付けられ、これは生の文字列(raw string)であることを示し、この場合はエスケープや文字列内挿入は認識されない。以前はその識別にrではなくて@が使われていたが、2012年9月末から仕様書0.11版にあわせ新しい仕様に切り替えられた。つまりこれまでの記法は次のように変更された:

```
@"..."  ->  r"..."
@'...' -> r'...'
@"""..."""  ->  r"""..."""
@'''...'''  ->  r'''...'''
```

生文字列はHTMLテキスト作成や次の例で示すようなファイル系の取り扱いで良く使用される:

```
new Path(r'c:\a\b').toString() == '/c:/a/b'
```

文字列バッファ (StringBuffer)

StringBufferを使うとプログラム的に文字列を生成するときに便利である。StringBufferはtoString()が呼ばれるまでは新しいStringオブジェクトを生成しない。

code_10.2d.dart

```
main() {
```

```
 StringBuffer sb = new StringBuffer();

 sb
 ..write("Use a StringBuffer")
 ..writeAll(["for ", "efficient ", "string ", "creation "])
 ..write("if you are ")
 ..write("building lots of strings");

 String fullString = sb.toString();
// 但しprintするだけならprint(sb);だけで可、print(sb)はsb.toString()を印刷する
 print(fullString);
// 結果：Use a StringBufferfor efficient string creation if you are building
lots of strings
}
```

経過： StringBufferは2013年2月時点でM3変更の一環として"StringSink"（この名前も変だという議論もなされているが）という抽象クラスを実装するようになっている。以下は主たる変更点である：

```
StringBuffer.add -> StringBuffer.write
StringBuffer.addCharCode -> StringBuffer.writeCharCode
StringBuffer.addAll(iterable) -> StringBuffer.writeAll(iterable);
StringBuffer.clear() -> often: new StringBuffer()
```

UTF-16

StringはUTF-16コード単位の並びである。従ってある文字は16ビットがひとつの場合と2つの場合が存在することに注意が必要である。例えば[]演算子で文字を取り出す場合は、16ビット以上の文字では正しく取り出せない。但し日本語を含む殆どの文字（基本多言語面（BMP））は1つの16ビットの単位で構成されるのであまり心配する必要が無い。特殊なシンボルを扱うときは注意のこと。

下図の音楽シンボルのひとつであるG CLEF ('\u{1D11E}') はUTF-16の16進表示では0xD834 0xDD1E (d834dd1e)になる。

Javaでは同じようにCharacter.charCount()は2となる。Dartもこのような文字に対応する為にrunes（ルーンズ：古代文字）という属性を使う必要がある：

```
var clef = "\u{1D11E}";
clef.length;  // => 2
clef.runes.first == 0x1D11E;  // => true
clef.runes.length;  // => 1
clef.codeUnitAt(0);  // => 0xD834
clef.codeUnitAt(1);  // => 0xDD1E
// 次の文字列はUTF-16サロゲート・ペアを2分割してしまい
```

```
// 従って無効なUTF-16文字列になる
clef[0];  // => a string of length 1 with code-unit value 0xD834.
clef[1];  // => a string of length 1 with code-unit value 0xDD1E.
```

文字列のなかの各文字を正しく順番に取り出して操作するには、処理時間がかかるが次のように書ける:

```
String s=....
for (var t in s.runes) {var c=new String.fromCharCode(t);};
```

という記述になる。もしルーンズを1文字のStringではなくて整数として扱うなら、処理時間は格段に改善される:

```
for (var t in s.runes) {var c=t;};
```

更に処理速度を上げるときは整数として扱い、やや長い記述になるが:

```
var codes=s.codeUnits;
var c;
for (int i=0; i<codes.length; i++)
   c=codes[i];
```

といった具合に書けばよい。

10.3節　　リスト (List<E>)

リスト・リテラル(list literal)は整数のインデックスが付けられたオブジェクトたちの集まりであるリスト(配列)を意味する。リスト・リテラルは角括弧即ち[]で要素の集まりを記述する。

listLiteral(リスト・リテラル):
const? typeArguments(型引数)? '[' (expressionList(式リスト) ','?)? ']'
;

リストはゼロまたはそれ以上のオブジェクトを持てる。あるリストの要素数はそのサイズ(size)である。リストはそれに結び付けられたインデックスたちのセットを持つ。空のリストは空のインデックスたちのセットを持つ。非空のリストはインデックスのセット{0 … n -1}をもち、ここにnはそのリストのサイズである。そのインデックスたちのセットのメンバでないインデックスを使ってあるリストをアクセスしようとするのは実行時エラーである。

あるリストが予約語であるconstで始まるときは、それは定数リスト・リテラルでそれはコンパイル時に計算される。そうでないときはそれは実行時リテラルで実行時に計算される。

[]はリスト・リテラルを示すことになるので、

```
var list = [] ;
```

とあるListを宣言したとすると、それは

```
List list = new List<dynamic>();
```

と宣言したと等価である。

リストの全要素の型が固定されているときは例えば次のように宣言できる:

```
var a = <int>[0,1,2];
List<int> b;
```

また、

```
List list;
```

という宣言はlistという名前の宣言だけで、オブジェクトは生成されずnullであることに注意。

code_10.3a.dart

```
List myList;
main() {
  print(myList);
//  myList.add("Hello"); // error
  myList = new List();
  print(myList.length);
  myList.add("Hello");
  print(myList);
  print(myList.length);
}
/*
null
0
[Hello]
1
*/
```

- 最初のList myList;は変数を定義しただけで、オブジェクトは生成されない。従ってこれはnullと出力される。
- nullのオブジェクトに対しては、myList.add("Hello");という操作はできない（NullPointerExceptionが生起される）。
- myList = new List();はコンストラクタを呼びオブジェクトが生成される。ここでは長さが0のオブジェクトである。これはvar myList[];あるいはList myList = [];という記述でも同じ効果を持つ。
- これに対してはmyList.add("Hello");という操作が可能になる。

code_10.3b.dart

```
import 'dart:math';
main() {
  List myList = ['pi is ' , PI, ', and cos pi is ', cos(PI)];
  String s = '';
  for(int i = 0; i < myList.length; i++) {
    s = "$s${myList[i].toString()}";
  }
  print(s);
}
/*
pi is 3.141592653589793, and cos pi is -1
```

```
*/
```

- この例ではmyListというリストに2つの文字列と2つのdoubleの値が最初に含められている。
- sという変数はなにもセットしないとnullのままなので、初期値として空の文字列''をセットする。空の文字列はnullではないので、他の文字列を連結させることができる。
- リストの中身のインデックスは0から始まるので最後の値はmyList.lengthになる。
- 総てのオブジェクトのもとになっているObjectにはtoStringメソッドが定義されているので、これを使って総ての要素を文字列に変換して連結させている。但しDartはtoStringを付けなくても自分で判断して文字列に変換する。各自確認されたい。

文字列が要素であるListのソーティングの例を次に示す。StringはComparable<String>を実装しているので、比較を使ったソートが簡単に実現できる:

code_10.3c.dart

```
main(){
  List<String> list = ['安部', '97', '01', 'abcd', 'ABcd', '安部晋一郎'];
  list.sort((a, b) => a.toLowerCase().compareTo(b.toLowerCase()));
  list.forEach((s){print(s);});
}
/*
01
97
abcd
ABcd
安部
安部晋一郎
*/
```

ListはIterableを実装しているのでこのようにforEachメソッドが使える。4行目のprintの箇所は次のように記述できる:

```
list.forEach(print);
```

比較関数としてcompareToが使われている。toLowerCaseで一旦小文字にして比較するのは、同じ英文字の大文字と小文字では大文字のコードが小さいので、'Zabc'が'aabc'よりも若くなってしまい、見づらくなる為である。

このような使い方はComparableを実装しているDate, Duration, Level, String, intx, numに適用できる。

ListはCollectionを実装しているので、このメソッドのreduceを使ってリストの中から最大または最小の要素をとりだすことも出来る。これまで存在していたこのインターフェイスのmaxとminのメソッドは廃止対象(Deprecated)になっている。

```
import 'dart:math' as Math;

main () {
  var a = [7, 2, 4, 1, 9];
  print(a.reduce(a.first, Math.min)); // 1
  print(a.reduce(a.first, Math.max)); // 9
}
```

マトリクス演算

マトリクスは次のように構成できる:

```
List list = new List.filled(3, new List.filled(3, 0));
```

これは2次元の3行3列マトリクスの例であるが、これはまた次のように生成できる:

```
var inner = new List.filled(3, 0);
var list = new List.filled(3, inner);
```

更にList.generateコンストラクタを使うと:

```
var list = new List.generate(3, (_) => new List.filled(3, 0));
```

とも書ける。各要素へのアクセスは次の例を見れば理解できよう:

code_10.3d.dart

```
main(){
  var list = new List.filled(3, new List.filled(3, 0));
  list[0][0] = 1;
  list[1][1] = 2;
  list[2][2] = 3;
  print(list);
}
// [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
```

バイト列

バイト列はネットワークを介したデータ交換やファイル・データの取り扱いに良く使われるが、リストはバイト列を表現するのにも良く使われる。

code_10.3e.dart

```
import 'dart:convert';
main() {
  var string = 'Dart言語';
  List<int> bytes = [];
  bytes.addAll(new Utf8Codec().encode(string));
  print(bytes);
}
// [68, 97, 114, 116, 232, 168, 128, 232, 170, 158]
```

これは'Dart言語'という文字列をUTF-8のバイト列に変換したものである。このようにbytesというバイト列はバイト(8ビット即ち0から255までの整数)の列になっている。

またdart:typed_dataというライブラリにはUint8Listという抽象クラスが存在する。このクラスはListとTypedDataを継承したもので、Listよりメモリ消費が少なくまた処理速度があがる。buffer属性を使うとByteBufferとして機能できるので、IOで良く使われる。

10.4節　　マップ (Map<K, V>)

マップ・リテラル(map literal)とは文字列たちからオブジェクトたちへのマップを示す。マップ・リテラルは波括弧即ち{}でエントリ即ち要素の集まりを記述する。

mapLiteral（マップ・リテラル）:
　const? typeArguments（型引数）? '{' (mapLiteralEntry（マップ・リテラル・エントリ） (',' mapLiteralEntry（マップ・リテラル・エントリ）)* ','?)? '}'
　;

mapLiteralEntry（マップ・リテラル・エントリ）:
　identifier（識別子） ':' expression（式）
　| stringLiteral（文字列リテラル） ':' expression（式）
　;

マップ・リテラルはゼロまたはそれ以上のエントリ(entries)で構成される。各エントリは文字列であるキーと、オブジェクトである値を持つ。nullを含むどのオブジェクトも値になれる。あるエントリのキーは識別子で、またはコンパイル時定数文字列で指定できる。もしそのキーが識別子idで指定されているときは、その指定はあたかもそれが文字列'id'であるかのごとく解釈される。

{}はマップ・リテラルを示すことになるので、

```
var map = {} ;
```

とあるMapを宣言したとすると、それは

```
Map map = new Map<String, dynamic>();
```

と宣言したと等価である。

また、

```
Map map;
```

という宣言はmapという名前の宣言だけで、オブジェクトは生成されずnullであることに注意されたい。

簡単な番号案内のプログラムを示す:

code_10.4a.dart

```
main() {
  Map directory = const{'fire': 119, 'cops': 110, 'emergency': 120, 'time':
117};
  String s = 'time';
  if (directory.containsKey(s))
  print('$s? dial ${directory[s]}');
  else print('sorry, no number available for $s');
```

```
}
/*
time? dial 117
*/
```

- Map型の電話帳であるdirectoryはStringの電話先とintの電話番号からなる。(この例では0から始まる番号は扱えない)
- containsKey(s)というメソッドは指定したキーが存在するかどうかをboolで返す。
- [s]という演算子は指定したキーの値を返す。

Map＜K, V＞の詳細はDartのAPI参照を見て頂きたいが、幾つかの使い方を示す:

code_10.4b.dart

```
main() {
  var directory = {'fire': 119, 'cops': 110, 'emergency': 120, 'time': 117};
  directory['weather'] = 177;    // 要素の追加
  print(directory['weather']);   // 177
  print(directory.length);       // 5, キーと値のペアの数
  directory.remove('weather');   // 要素の削除
  print(directory.length);       // 4
  print(directory['weather']);   // null
  directory.forEach((k,v) => print(k));  // 繰り返し操作
  directory.putIfAbsent('earthquake', () {
    // do something
    return 171;
  });
  print(directory['earthquake'] );       // 171
}

/*
 * 177
5
4
null
fire
cops
emergency
time
171
*/
```

なおM1変更で総てのオブジェクトがhashCode()メソッドを持つようになった。これはObjectクラスのなかで定義されている。従って、次のようにMapのキーとしてオブジェクトが使えるようになった。但しマップ・リテラルは別である。

code_10.4e.dart

```
class Person {
  String firstName, lastName;
  Person(this.firstName, this.lastName);
}
```

```
class Puppy {
  final bool cuddly = true;
}

main() {
  var spot = new Puppy();
  var alice = new Person("Alice", "Smith");
  var petOwners = new Map();
  petOwners[alice] = spot;
  print(petOwners[alice].cuddly); // true
}
```

Mapのキーが String で無いときは、たとえ宣言してもエラーが生じる。例えば:

```
Map<int, String> myMap = {};
```

は、Map<String, dynamic>で無いのでエラーになる。従って次のようにコンストラクタを使った宣言とインスタンス生成が必ず必要になる:

```
Map<int, String> myMap = new Map();
Map yourMap = new Map();

myMap[1] = 'Tarou';
yourMap[2] = 'Jirou';
```

MapとJSON

DartではMapで記述したオブジェクトはJSONテキストと同じになる:

code_10.4c.dart

```
import 'dart:convert';
var jsonObj = {"language":"DART",
               "targets":["dartium","javascript","Android?"],
               "website":{"homepage":"www.dartlang.org","api":"api.dartlang.org"}};
main() {
  String jsonStr = JSON.encode(jsonObj);
  print(jsonStr);
}
/**
{"language":"DART",
"targets":["dartium","javascript","Android?"],
"website":{"homepage":"www.dartlang.org","api":"api.dartlang.org"}}
*/
```

このマップは"language"(文字列)、"targets"(リスト)、及び"website"(マップ)の3つの要素からなっている。これをJSON.encodeメソッドを使ってJSONの文字列に変換しても全く同じ文字列となる。逆にJSON.decodeメソッドを使ってJSON文字列をあるクラスのオブジェクトに変換することができない。

従って各要素に対しては

```
jsonObj["website"]["homepage"]
```

というアクセスはできるが、通常のクラスのように

```
jsonObj.website.homepage
```

というアクセスはできない。これはユーザにとっては面倒であり議論されている。その解決策として幾つかのサードパーティのライブラリが用意されているがやはり手間がかかる。

Iterable

ListもMapも繰り返し操作の為のIterableを実装している。繰り返し操作に関してはfor-in文の節で説明するが、Iterableには他にも多くの有用なメソッドがある。

any	要素のどれかがある条件を満たしているかを調べる
contains	あるオブジェクトが含まれているかどうかを調べる
elementAt	指定した場所の要素を取り出す
every	全要素がある条件を満たしているかどうかを調べる
expand	Iterableを返すある関数を各要素に適用し、その結果を連結して各要素をゼロまたはそれ以上の数に拡張する。例えば: `var a = [[1, 2, 3], ['a', 'b', 'c'], [true, false, true]];` `var flat = a.expand((i) => i).toList();`
firstWhere	指定した条件を満たす最初の要素
fold	各要素にある関数を適用してひとつの要素とする。例えば各要素の集計をするときは: `iterable.fold(0, (prev, element) => prev + element);`
forEach	各要素に対しある関数を適用する
join	各要素をStringに変換した後で連結する
lastWhere	指定した条件を満たす最後の要素
map	各要素にたいしある関数を適用して作られる要素たちからあらたなIterableを作る
noSuchMethod	Objectから継承
reduce	指定した関数で各要素を組み合わせることで単一の値とする。例えば各要素の集計は: `iterable.reduce((value, element) => value + element);`
singleWhere	ある条件を満たす要素を一つ返す
skip	最初のn個の要素を飛ばしたIterableを返す
skipWhile	ある条件を満たした要素を飛ばしたIterableを返す
take	最大n個の要素からなるIterableを返す
takeWhile	ある条件を満たさなくなるまでの要素からなるIterable

toList	Listに変換
toSet	Setに変換
toString	Stringに変換
where	ある条件を満たす要素からのみなるIterableを返す

10.5節　　This

予約語の**this**は現在のインスタンス・メンバ呼び出しのターゲットであることを意味する。

thisExpression（this式）:
this
;

thisの静的な型は直前に包含しているクラスのインターフェイスである。

これはJavaの経験者には説明の必要がなかろう。

10.6節　　代入

代入(Assignment)は可変変数(mutable variable ：finalでない変数）または属性(property)に結び付けられた値を変更する。演算子は=または複合代入演算子である。

assignmentOperator（代入演算子）:
'='
| compoundAssignmentOperator（復号代入演算子）
;

Javaでは値渡しとなるプリミティブは存在せず、Dartでは総てがオブジェクトである。Java同様、Dartでは代入の右辺を計算した結果のオブジェクトが左辺にバインドされる。即ち参照渡し（正確にはpass-references-by-value）であることに注意されたい。2つの変数間のa=b;という代入はひとつのオブジェクト参照（バインド）していることになる。Stringやnumのような単純なオブジェクトであればその後aやbに代入を行えばそこで両者間の参照関係が切れてしまうので、通常は気にする必要はない。しかしオブジェクトaやbそのものに別のオブジェクトを代入をしない限り、バインド即ち参照は維持される。これはオブジェクトbをラップしたクラスなどに良く使われる。

そのような例として良く使われるのがList、Map、及びクラスのようなオブジェクトである。次の例を考えてみよう：

code_10.6a.dart

```
Map original = {};

main(){
  original["a"]=3.14;
```

```
  original["b"]=1.414;
  print("original : $original");
  Map copy = original;
  copy["c"] = "Hello copy!";
  print("copy : $copy");
  print("original : $original");
}
/*
original : {a: 3.14, b: 1.414}
copy : {a: 3.14, b: 1.414, c: Hello copy!}
original : {a: 3.14, b: 1.414, c: Hello copy!}
 */
```

この場合はコピーのオブジェクトはその要素を変更してもオリジナルとの参照が維持される。copy["c"] = "Hello copy!";という操作は代入ではなくて追加である。従ってコピーしたほうにそのような変更を加えれば、それはオリジナルに変更を加えたことになる。

同じことがクラスにおいても言える:

code_10.6b.dart

```
class Original {
  var a = 1;
  var b;
}

main() {
  var original = new Original();
  var copy = original;
  copy.b = 2;
  print('original.a = ${original.a}');
  print('original.b = ${original.b}');
}
/*
original.a = 1
original.b = 2
 */
```

copy.b = 2;という操作はcopyというオブジェクトに新たなオブジェクトを代入している訳ではないので、copyはoriginalとの参照を維持し続ける。

上記2つの例でcopyという変数がoriginalとの参照を必ず維持し続ける為にはvar copy やMap copyという記述ではなくて、**final** copy（あるいは**final** Map copy）と記述するよう習慣づけるべきである。**final**な変数は初期化時にそのバインディングが固定される変数である。従って**final**な変数は初期化後は常に同じオブジェクトを参照する。

もうひとつ勘違いしそうな例を示そう:

code_10.6c.dart

```
void reassignOne(List arg) {
  arg = new List.from([100, 200, 300]);
  print('In call: $arg');
}
```

```
void main() {
  List list = [1, 2, 3];
  print(list);
  reassignOne(list);
  print(list);
}
/*
[1, 2, 3]
In call: [100, 200, 300]
[1, 2, 3]
*/
```

この場合はreassignOneという関数の引数argにはlist即ち[1, 2, 3]オブジェクトへの参照が渡される。しかしながらこの関数の中ではargには**new** List.from([100, 200, 300])という新しいオブジェクトへの参照が渡される。すなわちここで外のlistとの参照関係が無くなってしまう。従ってreassignOneはlistの中身を変更することにはならない。

複合代入演算子

変数の値に何らかの演算を行った後結果を再度その変数に代入する複合代入演算子には以下のものがある:

代入	演算子
積算代入	*=
除算代入	/=
除算(整数部)代入	~/=
剰余代入	%=
加算代入	+=
減算代入	-=
左シフト代入	<<=
右シフト代入	>>=
論理積代入	&=
排他的論理和代入	^=
論理和代入	|=

各演算子に関しては「演算子」の節を見て頂きたい。

10.7節　　条件式

条件式(conditional expression)はJavaScriptでは条件演算子(conditional operator)と呼ばれており、ブール条件

に基づき2つの式のひとつを計算する。3項演算子(三項演算子)と呼ぶ人もいる。

conditionalExpression（条件式）:
logicalOrExpression（論理または式）('?' expression（式）':' expression（式）)?
;

e_1 ? e_2 : e_3の形式の条件式cの計算は以下のように進行する:

最初に、オブジェクトo_1としてe_1が計算される。チェック・モードでは、o_1が型**bool**でないときは動的型エラーである。そうでないときは、次にo_1はブール変換の対象であり、オブジェクトrをつくる。もしrがtrueなら、cの値は式e_2の計算結果である。そうでないときは、cの値は式e_3の計算結果である。

code_10.7.dart

```
main() {
  bool isMember = false;
  print('The fee is ${isMember ? @'$2.00' : @'$10.00'}');
}
/*
The fee is $10.00
*/
```

10.8節　　論理ブール式

論理ブール式(Logical Boolean Expressions)はブール積(&&)と和(||)の演算子を使ってブール値オブジェクトたちを組み合わせる。

また論理否定は!を使用する。

「ブール値」の項も参照されたい。

code_10.8.dart

```
main() {
var a1 = true && true;       // t && t returns true
var a2 = true && false;      // t && f returns false
var a3 = false && true;      // f && t returns false
var a4 = false && (3 == 4);  // f && f returns false
var a5 = "Cat" && "Dog";     // str && str returns false with warning
var a6 = false && "Cat";     // f && str returns false with warning
var a7 = "Cat" && false;     // str && f returns false with warning
var o1 = true || true;       // t || t returns true
var o2 = false || true;      // f || t returns true
var o3 = true || false;      // t || f returns true
var o4 = false || (3 == 4);  // f || f returns false
var o5 = "Cat" || "Dog";     // str || str returns false with warning
var o6 = false || "Cat";     // f || str returns false with warning
var o7 = "Cat" || false;     // str || f returns false with warning
var n1 = !true;              // !t returns false
var n2 = !false;             // !f returns true
var n3 = !"Cat";             // !str returns true with warning
```

```
print(a1); print(a2); print(a3); print(a4); print(a5);
print(a6); print(a7); print('');
print(o1); print(o2); print(o3); print(o4); print(o5);
print(o6); print(o7); print('');
print(n1); print(n2); print(n3);
}
```

Dartでは論理値のtrue以外はfalseとして扱われることに注意。

10.9節　　等価式

*等価式(equality expression)*はオブジェクトたちの同一性または等価性をテストする。M1変更で2つのオブジェクトが等しいかどうかの演算子は===では無くて==となった。これは総てのクラスのスーパークラスであるObjectのなかで定義されている。この定義により、**null**に対するテストとして===を使う必要が無くなり、また**null** == eあるいはe == **null**を書くべきかどうかを心配することもなくなった。これまでの===演算子はJavaScriptとPHPで==演算子を拡張したものだったもので存続されてはいたが、その後削除された。==は双方の型が違っていても値が同じならtrueになるが、===では型も一致しなければtrueにならなかった。

またbool **identical**(Object a, Object b)という関数は、printとおなじくdart:coreライブラリに新しく定義されたトップ・レベル関数である。これは2つの参照が同じオブジェクトを指しているときにtrueを返す。

equalityExpression（等価式）:
　relationalExpression（関係式） (equalityOperator（等価演算子） relationalExpression（関係式）)?
| **super** equalityOperator（等価演算子） relationalExpression（関係式）
;

equalityOperator（等価演算子）:
　'=='
| '!='
;

等価式は関係式、または**super**または式e_1に対し引数e_2での等価演算子の呼び出し、のいずれかである。

e_1 == e_2の形式の等価式*ee*は以下のように進行する:

- 式e_1が評価されオブジェクトo_1となる。
- 式e_2が評価されオブジェクトo_2となる。
- o_1とo_2のどちらかが**null**のときは、*ee*は**identical**(o_1, o_2)と計算される。そうでないときは、
- *ee*はメソッド呼び出しo_1.==(o_2)として計算される。

super == *e* の形式の等価式***ee***の計算は以下のように進行する:

- 式*e*が計算されオブジェクト*o*になる。
- もし**this** または *o* がnullのときは、*ee*は**true**と計算される。そうでないときは、
- **this**または*o*のどちらかが**null**のときは、*ee*は**identical(this**, *o*)と計算される。そうでないときは、
- *ee*はメソッド呼び出し**super**.==(*o*)と等価である。

e_1 != e_2の形式の等価式は式!(e_1 == e_2)と等価である。**super** != e の形式の等価式は式!(**super** == e)と等価である。

等価式の静的な型はboolである。

現時点での実装は以下のようになっていている。

code_10.9.dart

```
var x = 1;
int y = 1;
double z = 1.0;
String p;
var q = '';
main() {
  try{
    print(x == y);   // true
    print(identical(x, y));  // true
    print(x != y);   // false
    print(x == z);   // true
    print(identical(x, z));  // false
    print(x == y);   // true
    print(y == z);   // true
    print(identical(y, z));  // false
    print(x == '1'); // false
    print(p == q);   // false
    print(identical(p, q));  // false
    print(p == null);// true
    p = "";
    print(p == null);// false
    print(p == q);   // true
    print(identical(p, q));  // true
    print(identical(p, q)); // true
  } on Exception catch(e){
    print(e);
  }
}
```

10.10節　単項式

単項式(unary expression)は次のような形式になる:

- 式!*e*は式*e*? **false**: **true**と等価である。
- ++*e*の形式の式の計算は*e* += 1と等価である。
- --*e*の形式の式の計算は*e* -= 1と等価である。
- 式-*e*はメソッド呼び出し*e*.**negate**()と等価である。

- 式-**super**はメソッド呼び出し**super**.**negate**()と等価である。
- op *e*形式の単項式*u*はメソッド呼び出し式*e*.op()と等価である。
- op **super**形式の式はメソッド呼び出し**super**.op()と等価である。

- *v* ++の形式の後置式の計算は、(){**var** *r* = *v*; *v* = *r* + 1; **return** *r*}()と等価である。
- *C.v* ++の形式の後置式の計算は、(){**var** *r* = *C.v*; *C.v* = *r* + 1; **return** *r*}()と等価である。
- e_1.*v*++の形式の後置式の計算は、(*x*){**var** *r* = *x.v*; *x.v* = *r* + 1; **return** *r*}(e_1)と等価である。
- *e*--の形式の後置式の計算は*e* ++ (-1)と等価である。

Javaでもそうであるが、**後置演算結果の代入には注意が必要**である。たとえば:

```
main() {
  int i = 10;
  i = i++;
  print(i); // prints 10
  }
```

は10とプリントされる。これは定義により

```
i = (){var r=i; i=r+1; return r;}();
```

と等価なので、増分されていないrがiに代入されるからである。従って増分した結果を印刷したいときはこの行は:

```
i++;
```

と記さねばならない。**通常後置式はforループ以外には使用しないこと**が推奨される。

10.11節　型テスト

is-式(is-expression)はあるオブジェクトがある型のメンバであるかどうかをテストする。

IsOperator（is演算子）:
is '!'?
　;

is-式*e* **is** *T*の計算は以下のように進行する:

式*e*は値*v*として計算される。次に、もし*v*のクラスから誘導されたインターフェイスが*T*の副型のときは、そのis-式はtrueと計算される。そうでないときはfalseとして計算される。

現時点の実装では次のような結果になっている:

code_10.11.dart

```
var x = 1;
int i = 1;
double d = 1.23;
```

```
String s =  '1';
main() {
  print(x is int);    // true
  print(x is num);    // true
  print(x is double); // false
  print(i is num);    // true
  print(i is int);    // true
  print(i is double); // false
  print(i is num);    // true
  print(d is int);    // false
  print(s is num);    // false
  print(s is String); // true
}
```

なお、

```
x is! A
```

という式は

```
!(x is A)
```

という式と等価である。

10.12節　型キャスト (Type Cast)

M1変更で型キャストが追加されている。

キャスト式*(cast-expression)*はあるオブジェクトがある型のメンバであることを確保する。

typeCast（型キャスト）:
  asOperator type
;
asOperator（as演算子）:
as
  ;

e **as** *T* という形式のキャスト式の計算は以下のように進行する:

式*e*が計算され値*v*が得られる。次に、*v*のクラスのインターフェイスが*T*の副式であるならば、このキャスト式は*v*と計算される。そうでないときは、もし*v*が**null**なら、このキャスト式は*v*と計算される。それ以外の総てに対してはCastExceptionがスローされる。

型キャストが導入されたことで、型アノテートされたローカル変数を宣言しなくてもある式にたいし型をキャストできるようになる。例えば次のようなコードは:

```
ButtonElement button = query('button');
button.value = 1234;
```

次のような簡単な記述が出来るようになる:

```
(query('button') as ButtonElement).value = 1234;
```

あるいは、

```
if (person is Person) {               // Type check
  person.firstName = 'Bob';
}
```

といった型チェックの代わりに、

```
(person as Person).firstName = 'Bob';
```

と簡単な記述が可能になる。但しこの方法だとpersonがnullだったりPerson型で無かった場合は例外が発生する。

第11章 文 (Statements)

言語仕様書第17章に記されているように、文(Statements)はローカル関数宣言(local function declaration)、ローカル変数宣言(local variable declaration)、式文(expression statement)に加え、制御フロー(control flow statements)の記述に使われる。式を波括弧（{と}）で囲ったものはブロックである。ブロックによって新しい可視域(scope)が作られる。

本章はそのうち制御フロー関わるものを解説している。

11.1節 If

*if文(if statement)*により、文たちの条件つきでの実行が可能になる。

if(*b*) s_1 **else** s_2の形のif文の実行は以下のように進行する:

> 最初に式*b*が計算されオブジェクト*o*が得られる。チェック・モードでは、もし*o*が型**bool**でないときは動的型エラーである。そうでないとき、*o*は次にブール変換の対象となりオブジェクト*r*が得られる。もし*r*がtrueなら、次に文s_1が実行され、そうでないときは文s_2が実行される。
>
> もし式*b*の型が**bool**に代入出来ないかもしれないときは静的型エラーである。

if (*b*) s_1の形式のif文は**if**(*b*) s_1 **else** {}なるif文と等価である。

次のコードはこれらの形式を説明するものである:

code_11.1.dart

```
main(){
  var age = 20;
  if( age > 18 ){
    print('Qualifies for driving');
  }

  bool password = false;
  if(password){
   print('You are eligible');
  }else{
  print('Sorry, not eligible');
  }

  DateTime now = new DateTime.now();
  DateTime deadline = new DateTime(2013, 2, 25, 0, 0, 0, 0);
  if (now.difference(deadline).inDays > 0) print('Expired!');

  String myFriend = 'Tom';
```

```
  if (myFriend == 'Bll') print('Hello');
  else print('Who are you?');
}
/*
Qualifies for driving
Sorry, not eligible
Expired!
Who are you?
*/
```

(注意:これらのコードをシンプルなオンラインのDartコードの開発ツールであるDartPadまたは統合開発環境で試すには、第15章 Dartの実行 (Dart Execution)の**DartPadの節**または**IntelliJ IDEA CEの節**を参照のこと)

11.2節　　For

*For文(for statement)*は繰り返しをサポートする。

Forループ (For Loop)

for (**var** $v = e_0$; c; e) sの形式のfor文の実行は以下のように進行する:

もしcが空のときはc'を**true**とし、そうでなければc'はcとしよう。
最初に変数宣言文**var** $v = e_0$が実行される。次に;

1. もしこれがこのforループの最初の繰り返しのときは、vをvとし、そうでないときはv'はステップ4の以前の実行で作られた変数v''としよう。
2. 式$[v'/v]c$ が計算され、ブール変換の対象とする。もしこの結果が**false**のときは、このforループは終了する。そうでないときは、実行はステップ3で継続する。
3. 文$[v'/v]s$が実行される。
4. v''を新規の変数だとする。v''はv'にバインドされる。
5. $[v''/v]e$が計算され、このプロセスはステップ1に再帰呼び出しされる。

基本的な書き方は次のようになる:

code_11.2a.dart

```
main(){
  var j = 1;
  for (int i=0; i<8; i++)
  {
    j += j;
  }
print(j);  // 256
}
```

For-in

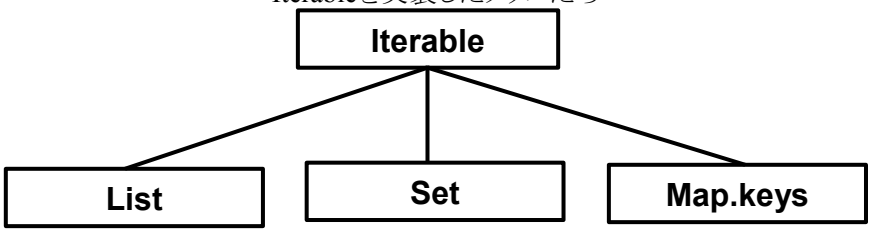
for (*finalVarOrType* id i**n** *e*) *s*の形式のfor文は以下のコードと等価である:

```
var n0 = e.iterator;
while (n0.moveNext()) {
  varOrType? id = n0.current;
  s
}
```

ここにn0はこのプログラムのどこにも生じない識別子である。

この形式はIterable抽象クラスを使用している。対象となるオブジェクトは次のインターフェイスを実装している必要がある。Iterable抽象クラスはこれを実装したオブジェクトからIteratorを取得出来るようにする為のものである。

Iterableを実装したクラスたち



次のサンプルは**for** (*finalVarOrType* id i**n** *e*) と等価なmoveNext及びforEachを使った場合を示している:

code_11.2b.dart

```
main() {
  List s = ['hello', 'world'];
  for(String x in s) print(x);
  // above statement is equivalent to:
  Iterator i = s.iterator;
  while (i.moveNext()) {
    String x = i.current;
    print(x);
  }
}
/*
hello
world
hello
world
*/
```

より具体的な使い方を次に示す:

code_11.2c.dart

```
main() {
  var callbacks = [];
  for (var i = 0; i < 2; i++) {
    var j = i;
    callbacks.add(() => print(j));
```

```
  }
  callbacks.forEach((c) => c());
}
/*
0
1
*/
```

これは典型的なクロージャで、関数リテラルのオブジェクト（即ちprint(j)、ここに各j値がセットされている）をListの用意していて、入力にあわせて処理を選択するもので、コールバックなどに使えよう。callbacksはコールバックの処理（ここではiという値を出力する）のオブジェクトのリストである。この例ではその処理を順番に呼び出している。Collectionインターフェイスにはvoid forEach(void f(E element))というメソッドがあり、これはこのコレクションの各要素にその関数（(c) => c()：つまりcをパラメタにした匿名の関数はそのcを実行する）を適用するものである。

なお2012年1月時点ではDartC (try.dartlang.org)を使った場合はjという変数経由でないとcallbacksには同じものが2つ入ってしまい、Forループの仕様を満たしていなかった。現時点ではではこの仕様は実装されていてprint(i)と書いて良い。

Mapの場合もMapのキーたちがIterableを実装しているので次のような使い方ができる：

code_11.2d.dart

```
main() {
  Map data = { '内閣総理大臣' : '野田佳彦', '総務大臣' : '川端達夫', '法務大臣' :
'平岡秀夫' };

// for-inループ
  for (var key in data.keys) {
    print('$key, ${data[key]}');
  }

// forEachループ

  data.forEach((key, value){
    print('${key}, ${value}');
  });
}
/*
内閣総理大臣, 野田佳彦
総務大臣, 川端達夫
法務大臣, 平岡秀夫
内閣総理大臣, 野田佳彦
総務大臣, 川端達夫
法務大臣, 平岡秀夫
*/
```

foeEachを使った箇所は次のようにも記述できる：

```
// forEachループ
 printSingle(key, value){print('${key}, ${value}');}
 data.forEach(printSingle);
```

Iteratorという抽象クラスはIterableなオブジェクトからIteratorを取得するに使われる。言い換えれば任意のオブジェクトをfor-in（あるいはforEach）ループで使えるようにする。たとえば：

code_11.2e.dart

```
main() {
  var iterable = new Iterable.generate(3, (int i){
    print('element: $i');
    });
  iterable.forEach((e){e;});
}
/*
element: 0
element: 1
element: 2
*/
```

は(int i){ print('element: $i'); })（ここにiは0から2まで）という3つの関数リテラルを要素として持つiterableというオブジェクトを生成している。これをforEachで呼ぶと各要素が順に実行される。

```
iterable.forEach((e){e;});
```

を、

```
for (var e in iterable) {e;}
```

と置き換えても同じ結果が得られる。

11.3節　　While

while文は、その条件がそのループの前に計算された条件による繰り返しをサポートする。

whileStatement（while文）：
　　while '(' expression（式） ')' statement（文）
;

while (*e*) *s*;の形式のwhile文の実行は以下のように進行する：

式*e*がオブジェクト*o*として計算される。チェック・モードでは、もし*o*が型**bool**でないときは動的型エラーである。そうでないときは*o*はブール変換の対象になり、あるオブジェクト*r*をつくる。もし*r*が**true**なら、次に文*s*が実行され、そして次にこのwhile文が繰り返し的に再実行される。*r*が**false**なら、そのwhile文の実行は終了する。

code_11.3.dart

```
main() {
  int i = 0;
  int j = 1;
```

```
  while (i < 8)
  {
    j += j;
    i++;
  }
print(j);  // 256
}
```

11.4節　　Do

do文は、その条件がそのループの後に計算された条件による繰り返しをサポートする。

doStatement（do文）:
do statement（文） **while** '(' expression（式） ')' ';'
;

do s while (e);の形式のdo文の実行は以下のとおり進行する:

文sが実行される。次に、式*e*がオブジェクト*o*として計算される。チェック・モードでは、もし*o*が型**bool**でないときは動的型エラーである。そうでないときは*o*はブール変換の対象になり、あるオブジェクト*r*をつくる。*r*が**true**なら、次にこのdo文が繰り返し的に再実行される。*r*が**false**なら、そのwhile文の実行は終了する。

code_11.4.dart

```
main() {
  int i = 0;
  int j = 1;
  do {
    j += j;
    i++;
  }
  while (i < 8);
  print(j);  // 256
}
```

11.5節　　Switch

スイッチ文(switch statement)は多数のケースたち間への制御を振り分けをサポートする。

switchStatement（switch文）:
switch '(' expression（式） ')' '{' switchCase（スイッチ・ケース）* defaultCase（デフォルト・ケース）? '}'
;

switchCase（スイッチ・ケース）:
label（ラベル）? (**case** expression（式） ':')+ statements（文たち）
;

defaultCase（デフォルト・ケース）:
label（ラベル）? (**case** expression（式） ':')* **default** ':' statements（文たち）
;

switch文**switch** (*e*) { **case** e_1: s_1 … **case** e_n: s_n **default** s_{n+1}}の実行は次のように進行する:

文**var** n = *e*が実行される。ここにnはその名前がこのプログラム内の他のどの変数とも異なる変数名である。次にもし存在すればcase節e_1: s_1が実行される。もしe_1: s_1が存在しないときは、default節がs_{n+1}を実行することで実行される。

switch文**switch** (*e*) { **case** e_1: s_1 … **case** e_n: s_n **default** s_{n+1}}のcase節 case e_k: s_kの実行は次のように進む:

式*n* == e_kが計算されオブジェクト*o*が得られ、次にそれがブール変換の対象になり値*v*を得る。もし*v*が**false**またはs_kが空のときは、もし存在すれば以下のケース **case** e_{k+1}: s_{k+1}がs_{n+1}を実行するすることで実行される。もし*v*が**true**なら、文シーケンスs_kが実行される。

switch文**switch** (*e*) { **case** e_1: s_1 … **case** e_n: s_n}は**switch** (*e*) { **case** e_1: s_1 … **case** e_n: s_n **default**}と等価である。

code_11.5.dart

```
main() {
  DateTime now = new DateTime.now();
  switch(now.weekday) {
    case (DateTime.SUNDAY): print('hobby'); break;
    case (DateTime.SATURDAY): print('sleep'); break;
    default: print('work!');
  }
}
```

各case節にbreakを入れて分離しないとFallThroughErrorのエラーが出るので注意。これはあるcase節が実行されてから次の節も実行されるいわゆるcaseフォールスルーはバグが起きやすいとしてDartでは採用していないからである。**breakを要求することにより、CやJavaのようにcaseフォールスルーが可能な言語から来たプログラマがDartのコードを書いたときにcaseフォールスルーが可能だと思って起こし得る見つけ難いエラーを防止することができる。**

但し例外として空のcase節だけはフォールスルーが許されている:

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':     // 空の場合はフォールスルーする
  case 'NOW_CLOSED':
    // CLOSEDとNOW_CLOSEDの双方で実行される
    executeClose();
    break;
}
```

11.6節　　Try

try文(try statement)は構造化されたやり方での例外処理コードの定義をサポートする。

TryStatement（try文）:
　　try block（ブロック） (catchPart（キャッチ部）+ finallyPart（fainally部）? | finallyPart（fainally部）)
　;

catchPart（キャッチ部）:
　　catch '(' simpleFormalParameter（シンプル仮パラメタ） (',' simpleFormalParameter（シンプル仮パラメタ）)? ')' block（ブロック）
　;

finallyPart（fainally部）:
　　finally block（ブロック）
　;

try文は少なくとも次のひとつが後に付いたブロック文で構成される:

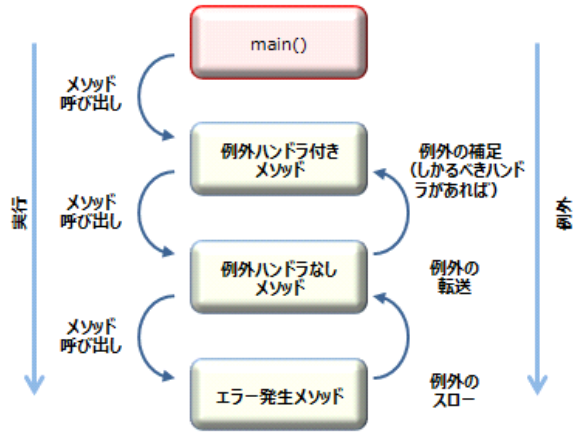
1. catch節たちのセットで、その各々がひとつまたは2つの例外パラメタたちとひとつのブロック文を規定している。
2. ブロック文で構成されるfinally節

code_11.6a.dart

```
main() {
  try {
    print( 5 % 0);
  }
  on Exception catch (e) {
    print(e.toString());
  }
  finally {
    print('finally statement');
  }
}
/*
IntegerDivisionByZeroException
finally statement
*/
```

より具体的な使い方はJavaと同じなので、説明する必要はなかろう。発生した例外は下図のようにcatchされるまで呼び出した側に伝搬してゆく。main()のなかで最終的に捕捉されたもの、またはそこでも捕捉されないエラーと例外(**uncaught**または**unhandled**なerrorまたはexceptionと呼ばれる)ではその時点でプログラムの進行が終了する。

例外の伝搬



finally句は例外が発生したかどうかにかかわらず実行される。catch句で捕捉されなかった例外またはエラーはfinally句が実行された後で伝搬する。

Try / Catch文法はM1変更の対象となっていて、2012年9月から(仕様書0.11版から)実施されている。その理由は、Dartの型アノテーションはオプショナルであって実行時には効果を持たないにもかかわらずこれまではcatch句のなかでは型が例外をスローする為のテストに使われているという問題があったからである。例えばcatch(foo)という句は、「どの型の例外でも捕捉しfooという名前で結び付ける」という意味なのか「fooという型の例外を捕捉してそれを変数として結び付けない」という意味なのかが判らない。

従って今後はcatch句のなかでは混乱を防止する為にcatch(var foo)またはcatch(foo someVar)という記述のみが許される。ここに最初のfooはどの型の例外に結び付けられた変数であり、次のsomeVarはfooの型の変数であることを意味する。

更にM1変更では新たな記述が追加された。

```
try {
  ...
} on SomeException catch(ex) { ... }
```

on SomeException部分は捕捉したい例外の型を規定し、総ての例外を捕捉するときはオミットできる。catch(ex)部でそれにバインドしたい変数を指定する。

具体的にはこれまでの記述:

```
  try { ... } catch (var e) { ... }
  try { ... } catch (final e) { ... }
  try { ... } catch (T e) { ... }
  try { ... } catch (final T e) { ... }
```

は次のような記述となった:

```
  try { ... } catch (e) { ... }  // まとめて捕捉（キャッチ・オール）
  try { ... } catch (e) { ... }
  try { ... } on T catch (e) { ... }  // 型ごとに個別に捕捉
  try { ... } on T catch (e) { ... }
```

DartにはErrorとExceptionがある。基本的にErrorとそのサブクラスはプログラム・エラーであり、そのプログラムは

修正が必要である。一方非エラーのExceptionは実行時エラーである。これは通常プログラムであらかじめスローされるのを防止出来ない。従ってこれらのグループはグループ毎に個別に捕捉しなければならないことに注意されたい。その他のエラーにはイベント・ベースで捕捉するエラーがあるが、これはtry catchとは異なり、イベントに対するコールバック関数の中で処理される。

Errorのサブクラス	Exceptionのサブクラス	その他
AbstractClassInstantiationError, ArgumentError, AssertionError, CastError, ConcurrentModificationError, FallThroughError, JsonUnsupportedObjectError, NoSuchMethodError, NullThrownError, OSError, OutOfMemoryError, RuntimeError, StackOverflowError, StateError, UnsupportedError	DirectoryIOException, ExpectException, FileIOException, FormatException, HttpException, HttpParserException, IntegerDivisionByZeroException, IsolateSpawnException, IsolateUnhandledException, LinkIOException, LocaleDataException, MimeParserException, MirrorException, MirroredError, ProcessException, SerializationException, SocketIOException, WebSocketException	dart:asyncやdart:ioの非同期操作でFutureオブジェクトが発生するエラー。これはFutureのcatchErrorメソッドで捕捉する （以前はAsyncErrorというクラスが存在していたが、2013年4月16日に廃止された。） 詳細は19.10節のZoneに関する記述、及び17.6節のFutureにおけるエラー処理を参照のこと

具体的な例を以下に示す:

code_11.6b.dart

```
import 'dart:async';
main() {
  var funcs = [funcA, funcB, funcC, funcD];
  for (var func in funcs) {
    try{
    func(0);
    }
    on Error catch (e, stackTrace)  {print('Error! $e \nStack Trace :\n$stackTrace');
    }
    on Exception catch (e, stackTrace) {print('Exception! $e \nStack Trace
:\n$stackTrace');
    }
  }
}

// ArgumentError
funcA(x) {
  if(x == 0) throw new ArgumentError('Generated ArgumentError');
}

// IntegerDivisionByZeroException
funcB(_) {
  int x = 1 % 0;
}

// AsyncError (Bad state: More than one element)
funcC(_) {
  final testData = ['a', 'b'];
  var stream = new Stream.fromIterable(testData);
  stream.single.then((value){})
    .catchError((e, stackTrace) => print('AsyncError! $e \nStack Trace :\n$stackTrace'));
}
```

```
//AsyncError (Same error but caught in a zone)
funcD(_) {
  final testData = ['a', 'b'];
  var stream = new Stream.fromIterable(testData);
  runZoned(() {
    stream.single.then((value){
    });
  },
  onError: (e, stackTrace) => print('AsyncError(zoned)! $e \nStack Trace :\n$stackTrace'));
}
```

このプログラムには各々異なった例外を起こす関数が存在している:

- funcA :ArgumentErrorをプログラム的に発生させる。
- funcB :IntegerDivisionByZeroExceptionを発生させる。
- funcC :FutureのErrorを発生させる。
- funcD :Zone内でFutureのErrorを発生させる。

これらの関数はforループにより順番に呼び出される。またmainはErrorとExceptionを個別に捕捉してその内容を出力している。

このプログラムを実行すると、次のように出力される:

```
Error! Invalid arguments(s): Generated ArgumentError
Stack Trace :
スタックトレース
```

最初の行がこのエラーの内容を表示している。次の行以降はスタック・トレースであり、funcAで発生したエラーがmainと伝搬してmainのループのなかでこれが捕捉されたことが確認されよう。funcBではExceptionのトラップが動作することが確認される:

```
Exception! IntegerDivisionByZeroException
Stack Trace :
スタック・トレース
```

funcCとfuncDはFutureインターフェイスのなかのエラーを表現したオブジェクトを発生させる。これは非同期処理のコールバック関数内で発生しており、catchErrorまたはonErrorというメソッドで捕捉され、**try / catchとは別の仕組みとなることに注意**されたい。うっかりFutureまたはStreamのオブジェクトを扱っていることを忘れて、エラーや例外を捕捉しないという失敗を起こしがちである。詳細は、「Futureにおけるエラー処理」の節及び19.10節のZoneに関する記述を参照のこと。

ユーザはExceptionを実装した自分の例外クラスが必要になることがあろう。そのような場合には次の例が参考になろう:

```
class MyException implements Exception{
    const MyException([String this.message = ""]);
    String toString() => "MyException: $message";
    final String message;
}
```

11.7節　　Break

break文(break statement)は予約語の**break**とオプショナルとしてのラベルで構成される。

breakStatement（break文）:
break identifier（識別子）? ';'
;

break文はdo、for、switchまたはwhile文のなかでループを終了させるのに使用される。

code_11.7a.dart

```
main() {
  for (int i = 0; i < 1000; i++) {
    print(i);
    if (i == 10) {
       break;
    }
  }
}
```

ラベルを使うと入れ子になったループのどこに戻るのかを指定できる:

code_11.7b.dart

```
main() {
  loopExit:
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 5; j++){
      print('$i, $j');
        if (i == 2 && j ==2) {
          break loopExit;
        }
     }
  }
}
```

11.8節　　Continue

*continue文(continue statement)*は予約語の**continue**とオプショナルとしてのラベルで構成される。

continueStatement（continue文）:
continue identifier（識別子）? ';'
;

continue文はdo、for、switchまたはwhile文のなかでループの文ブロックのなかの残りの処理をスキップさせるのに使用される。

次の例はfor(;;) {}という無限ループをベースにしているので、好ましいものではないが、breakとcontinueがどのよ

うに機能するかの理解の為のものである。

code_11.8.dart

```
main() {
  var x = 0;
  var y = 0;
  outerLoop:
    for(;;) {
    x++;
    innerLoop:
    for(y = 0; y < 10; y++) {
      if (x == 9 ) break outerLoop;  // quit outer loop
      if (y > 3) break;              // Quit the innermost loop
      if (x == 2) break innerLoop;   // Do the same thing
      if (x == 4) continue outerLoop;     // new outer loop test
      if ((x >= 7) && (x < 9)) continue;  // new inner loop test
      print('x = $x, y = $y');
    }
  }
  print('At end : x = $x, y = $y');
}
/*
x = 1, y = 0
x = 1, y = 1
x = 1, y = 2
x = 1, y = 3
x = 3, y = 0
x = 3, y = 1
x = 3, y = 2
x = 3, y = 3
x = 5, y = 0
x = 5, y = 1
x = 5, y = 2
x = 5, y = 3
x = 6, y = 0
x = 6, y = 1
x = 6, y = 2
x = 6, y = 3
At end : x = 9, y = 0
*/
```

11.9節　　Throwとrethrow

*throw文(throw statement)*はある例外を生起(raise)または再生起(re-raise)させるのに使用される。

throwStatement（throw文）:
throw expression（例外）? ';'

;

*現在の例外(current exception)*はスローされた最新の未処理の例外である。*現在のスタック・トレース(current stack trace)*は現在の例外がスローされた場所で実効が未だ終了していない現在のアイソレート内での総ての関数活性化の記録である。そのような関数活性化の各々に対し、現在のスタック・トレースには、その関数の名前、その総ての仮パラメタたちのバインディングたち、ローカル変数たちと**this**、及びその関数が実行していた位置(pisition)が含まれる。

rethrow文は同じ例外を再スローして、その上のtryブロックに伝搬させるものである。もしrethrow文がon-catch句内に含められていないときはコンパイル時エラーである。

M1変更ではnullをスローするのは動的エラー扱いとなった。これはnullがスローされる場合は通常はプログラム・エラーの場合であり、何らかの例外オブジェクトをスローするつもりだったものがエラーによりnullが返されたのが殆どだろう。従ってこれにより早期にエラー検出ができまた対策が可能となる。

実際の使い方は次のようになる:

code_11.9a.dart

```
class MyException implements Exception {
  String mes = '';
  MyException([this.mes]);
}

void ThrowDemo(int x) {
  try {
    try {
      // Throw an exception that depends on the argument
      if (x == 0)
        throw new MyException('200 : x equals zero');
      else
        throw new MyException('201 : x does not equal zero');
    }
    on MyException catch (e) {
    // Handle the exception
      switch (e.mes.substring(0, 3)) {
        case '200':
          print('$e - handled locally');
          break;
        default:
        // Throw the exception to a higher level
          print('not a "200" error .. ${e.mes}');
          throw new MyException('300 : threw from inner to higher');
      }
    }
  }
  on MyException catch (e) {
  // Handle the higher-level exception.
    print('$e - handled higher up');
  }
```

```
}
main() {
  ThrowDemo(0);
  ThrowDemo(1);
}

/*
200 : x equals zero - handled locally
not a "200" error .. 201 : x does not equal zero
300 : threw from inner to higher - handled higher up
*/
```

スタック・トレースは次のように使う:

code_11.9b.dart

```
main() {
  try {
    throw "Stack trace example";
  } catch (e, s) {
    print("Caught: $e");
    print("Stack: $s");
  }
}
/*
Caught: Stack trace example
Stack: #0      main
(file:///C:/dart_applications/LanguageGuideSampleCodes/test_stacktrace.dart:3
:5)
*/
```

catch句の2番目の識別子はスタック・トレースを意味する。オブジェクトsは通常のクラスを意味しておらず、唯一可能な操作はtoStringのみであることに注意。

次のコードはrethrowを使った例である:

code_11.9c.dart

```
main() {
  try {
    errorCode();
  }
  catch (e, st) { // catch any error and exception
    print('Caught an exception in the main() method : \n$e\n$st');
  }
}
errorCode() {
  try{
    var x = 5 % 0;
  }
  catch (e, st) {
    print('Caught an exception in the errorCode() method : \n$e\n$st');
    rethrow;
```

```
  }
}
```

rethrowにより、errorCode関数で発生した例外はその上のmainの関数の例外ハンドラに渡される。従って、次のようにコンソールに出力される：

```
Caught an exception in the errorCode() function :
IntegerDivisionByZeroException
#0      int.% (dart:core-patch/integers.dart:35)
#1      errorCode (file:///C:/Users/Name/Downloads/dart_code_samples-master/codes/code_11.9c.dart:11:15)
#2      main (file:///C:/Users/Name/Downloads/dart_code_samples-master/codes/code_11.9c.dart:3:14)
#3      _startIsolate.isolateStartHandler (dart:isolate-patch/isolate_patch.dart:190)
#4      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:93)

Caught an exception in the main() function :
IntegerDivisionByZeroException
#0      int.% (dart:core-patch/integers.dart:35)
#1      errorCode (file:///C:/Users/Name/Downloads/dart_code_samples-master/codes/code_11.9c.dart:11:15)
#2      errorCode (file:///C:/Users/Name/Downloads/dart_code_samples-master/codes/code_11.9c.dart:15:5)
#3      main (file:///C:/Users/Name/Downloads/dart_code_samples-master/codes/code_11.9c.dart:3:14)
#4      _startIsolate.isolateStartHandler (dart:isolate-patch/isolate_patch.dart:190)
#5      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:93)
```

11.10節　Assert

*assert文(assert statement)*は与えられたブール条件が成立しないときに通常の実行を中断する為に使われる。

> **AssertStatement（assert文）:**
> **assert** '(' conditionalExpression（条件式） ')' ';'
> ;

生産モードではassert文は効果を持たない。チェック・モードではassert文assert(*e*);の実行は以下のように進行する：

> 条件式*e*がオブジェクト*o*として計算される。もし*o*のクラスが**Function**の副型のときは、*r*を引数なしで*o*を呼び出した結果だとしよう。そうでないときは、*r*を*o*としよう。もし*o*が型**bool**または型**Function**でないときは動的エラーである。もし*r*が**false**なら、我々はその表明(assertion)が失敗したという。もし*r*が**true**なら、我々はその表明が成功したという。もしその表明が成功したら、そのassert文の実行は終了する。もしその表明が失敗したら、**AssertionError**がスローされる。

assert文をソースコードの中に置くことにより、プログラマがそのコードのテストとデバッグをし易くする。

例えば次のような単調なコードを書かねばならなくときは、えてして入力漏れが起きる。その際は本来起きない状態をassertで検出する。この場合は木曜日の入力を忘れるとチェック・モードでは例外が発生し、プログラマが入力忘れを知ることができる。

code_11.10.dart

```
// calculate day of the week
String getDayOfWeek(days) {
  if (days % 7 == 0) {
    return "Sunday";
  } else if (days % 7 == 1) {
```

```
      return "Monday";
  } else if (days % 7 == 2) {
      return "Tuesday";
  } else if (days % 7 == 3) {
      return "Wednesday";
//  } else if (days % 7 == 4) {
//      return "Thursday";
  } else if (days % 7 == 5) {
      return "Friday";
  } else {
      assert (days % 7 == 6);
    return "Saturday";
  };
}

main() {
  try{
    print(getDayOfWeek(5));
    print(getDayOfWeek(4));
  }on Exception catch(e){
    print(e.toString());
  }
}
/*
checked mode:
 Friday
 Failed assertion
production mode:
 Friday
 Saturday
*/
```

11.11節　YieldとYield-Each

これはDart 1.0版以降から導入されたジェネレータ関数(generator functions)に関するものである。これは関数のボディにsync*またはasync*キーワードが付加されたものである。Yield文とyield-each文はそれらに対応したものである。

ジェネレータ関数

ジェネレータ関数は一連の結果を後回しで計算する関数を言う。ジェネレータ関数は同期または非同期の2種がある。同期ジェネレータはオンデマンドで値を生成する:即ち受け取り側はそのジェネレータから値たちを引き出す。一方非同期ジェネレータ関数は自分のペースで(イベント・ドリブンで)値たちを生成し、受け取り側が受

け取り可能な時にそれらを受け取り側に押し出す。

同期ジェネレータ関数 (Synchronous generators: sync)*

関数ボディに**sync***修飾詞を付すと同期ジェネレータ関数であることを意味する。次のコードは最初のn個の自然数を得るものである。

Code_11.11a.dart

```
Iterable naturalsTo(n) sync* {
 int k = 0;
 while (k < n) yield k++;
}
main(){
 var it = naturalsTo(7).iterator;
 while (it.moveNext()) {
 print(it.current);
 }
}
```

naturalsTo関数は呼び出されると直ちにIterableオブジェクトを返す。これはasyncとマークされた関数が直ちにfutureを返すのと良く似ている。この関数のボディは該 iterator上で moveNextメソッドが呼ばれるまで実行されない。**yield**文は式を有し、その式が計算される。計算が終了したらこの関数が停止し、moveNext は**true**を返す。次回 moveNextが呼ばれたらこの関数は計算を再開する。ループが終了したらこのメソッドは暗示的にreturnを実行し、これでこの関数呼び出しは終了し、 moveNextは呼び出し側に**false**を返す。

非同期ジェネレータ関数 (Asynchronous generators: async)*

非同期にあるデータのシーケンスを作るにはStreamが使われる(「ストリーム(Streams)」の節を参照のこと)。非同期ジェネレータ関数を使うとそのようなストリームの取り扱いが容易になる。関数ボディにasync*修飾詞を付すとそれは非同期ジェネレータ関数であることを意味する。

先ほどと同じく自然数を発生させるジェネレータ関数の名前を asynchronousNaturalsToに変え、またそのボディにasync*を付してみよう。

Code_11.11b.dart

```
import 'dart:async';
Stream asynchronousNaturalsTo(n) async* {
 int k = 0;
 while (k < n) yield k++;
}
main(){
 asynchronousNaturalsTo(7).listen((i) => print(i));
}
```

asynchronousNaturalsToは呼び出されると直ちにStreamのオブジェクトを返す。これはsync*関数がIterableのオブジェクトを直ちに返すのと似ている。またasyncとマークされた関数が直ちにfutureを返すのとも良く似ている。

このストリームをlistenすることで、その関数ボディの実行が始まる。この関数は必ずしも保留状態になるわけではないが、通常の非同期関数は何らかのイベントによってその時点でデータの値を送信する。従って主導権は受けて（consumer）ではなく、該ストリームがリスナ関数にその値を渡す。

Code_11.11b.dartではリスナ関数は単に受理した値をプリントするだけである。通常はジェネレータ関数が終了したことをリスナ関数が知る必要があるので、以下のように終了やエラーのイベントにも対応する記述が使われよう。

Code_11.11c.dart

```
import 'dart:async';
Stream asynchronousNaturalsTo(n) async* {
 int k = 0;
 while (k < n) yield k++;
}
main(){
 asynchronousNaturalsTo(7)
  .listen((v) => print(v),
    onError: (err) => print("An error occured: $err"),
    onDone: () => print("The stream was closed"));
}
```

Yield-each

yieldは魅力的ではあるが、再帰的な関数の場合は時間が二乗的にかかってしまうという問題が生じる。次のようなnから1まで逆に計数する同期ジェネレータ関数を考えてみよう:

Code_11.11d.dart

```
Iterable naturalsDownFrom(n) sync* {
 if (n > 0) {
   yield n;
   for (int i in naturalsDownFrom(n-1)) { yield i; }
 }
}
main(){
 var it = naturalsDownFrom(3).iterator;
 while (it.moveNext()) {
 print(it.current);
 }
}
```

このコードは正しく動作するが二乗的に時間がかかってしまう。結果のシーケンスのn番目の要素を得るのにyield iを(n-1)回実行する。例えば最初の要素の3に対しては yield n; 文によってイールドされる2番目の要素2にはyield n;文によって1回とyield n; によって1回イールドされる。3番目の要素の1に対してはyield n;文によって1回とyield n; によって2回イールドされる。即ち全部合わせるとn(n − 1)回のyield i;によるイールドが行われる。

yield* (yield-eachと発音される) 文はこの問題の回避のために用意されている。yield*のあとの式は別のシーケンスであることを示す。yield*は副シーケンスの全要素を現在組み立てられているシーケンスに挿入するもので、あたかも各要素に対して個々のyieldがあるようにする。

Code_11.11e.dart

```
Iterable naturalsDownFrom(n) sync* {
 if ( n > 0) {
  yield n;
  yield* naturalsDownFrom(n-1);
 }
}
main(){
 var it = naturalsDownFrom(7).iterator;
 while (it.moveNext()) {
 print(it.current);
 }
}
```

この場合の実行時間はnに対してリニアにかかるだけである。

第12章 ライブラリ (Libraries)

概要の章で説明したとおり、ライブラリはインポートたちのセット（空のこともあり）、及びトップ・レベルの宣言たちのセットで構成される。トップ・レベルの宣言はクラス、インターフェイス、型宣言、関数あるいは変数宣言のことを言う。

ライブラリはプライバシの単位である。ライブラリ*L*内で宣言されたprivate宣言（その名前がアンダースコア('_')で始まる）はL内のコードによってのみアクセス可能である。プライベート・メンバ宣言を*L*の外部からアクセスしようとすると実行時エラーが発生する。

Dartの各アプリケーションは、たとえlibrary宣言がされていないにしてもそれ自身はライブラリである。組込みライブラリはdart:coreのようにdart:が付いている。

12.1節　　インポート (Imports)

インポート*(import)*指令はあるライブラリを別のライブラリのスコープ内で使うよう指定する。

import（インポート）:
　metadata **import**　stringLiteral（文字列リテラル）（**as** identifier)? combinator（組合せ子）* ("&" export)?";"
;

combinator（組合せ子）:
show identifierList（識別子リスト）
| **hide** identifierList（識別子リスト）
;

identifierList（識別子リスト）:
　identifier（識別子）(, identifier)*
;

Import以外に組み入れ指令(include directive)として**part**がある。これはソース・コードを組み入れる指令である。組み入れるファイルは指令を持つことはできない。URIにpackage:が付いていると、それはパッケージ・マネージャが用意したライブラリであることを意味する。asはそのライブラリの要素を使うときに付けるプレフィックスを指定する。

```
Library 'my_library'; // ライブラリ宣言
import 'dart:io'; // 組込みライブラリのインポート
import 'package:mylib/mylib.dart'; // pubのようなパッケージ・マネージャが用意したライブラリのインポート
import 'package:utils/utils.dart'; // 同上
import 'package:lib1/lib1.dart'; // プレフィックスなし
import 'package:lib2/lib2.dart' as lib2; // プレフィックスを使用
// ...
```

```
var element1 = new Element();      // lib1にあるElementを使用
var element2 = new lib2.Element(); // lib2にあるElementを使用
```

12.2節　　part

*パート指令(part directive)*は現在のライブラリに組み入れるべきDartのコンパイル単位が見つかるであろうURIを指定する。

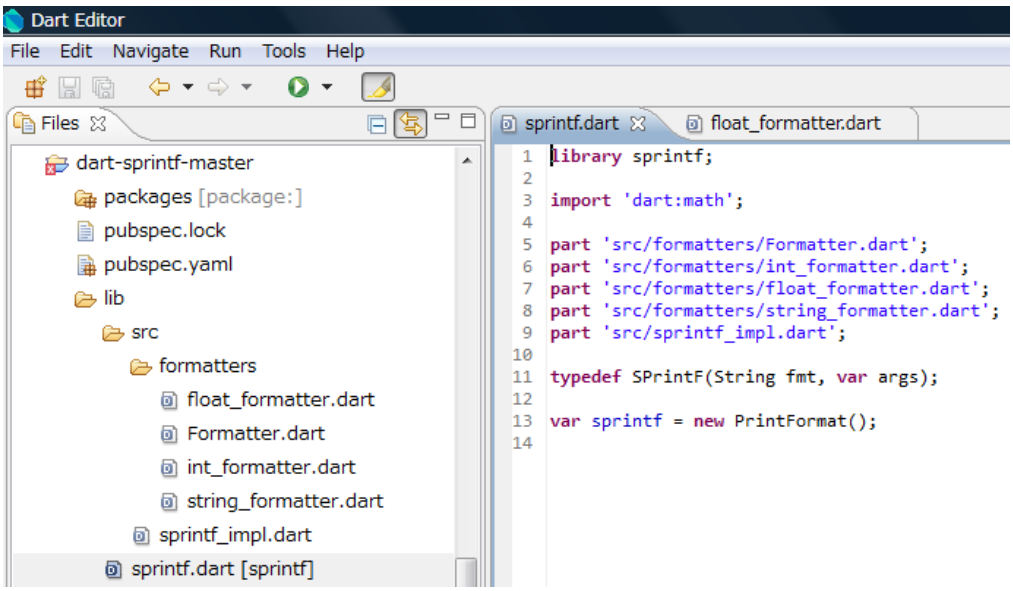
partDirective（part指令）:
  metadata（メタデータ）**part** uri（URI）";"
  ;

partHeader（partヘッダ）:
  metadata **part of** qualified（修飾された）
  ;

partDeclaration（part宣言）:
  partHeader topLevelDefinition（トップ・レベル定義）* EOF
  ;

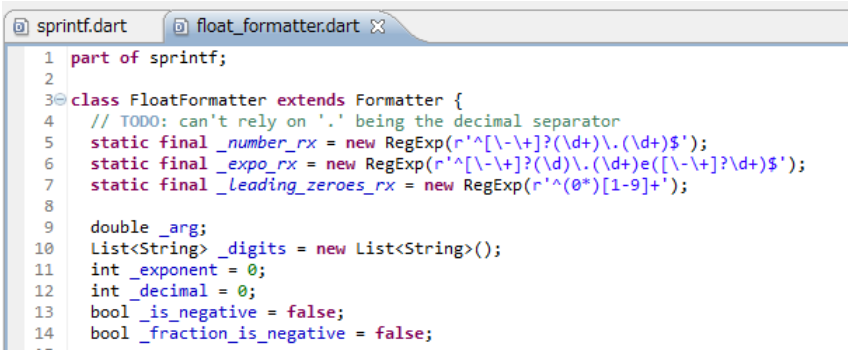
*part*ヘッダ*(part header)*は**part of**で始まり、そのパートが属するライブラリの名前が続く。part宣言はpartヘッダで始まり、トップ・レベルの宣言たちの並びが続く。

具体的な例で示したほうが理解が早い。例えばパッケージマネージャにあるsprintfというライブラリはC言語のsprintf関数をDartで実現するものであるが、これのlibというフォルダは次のような構成になっている:



srcのフォルダの中にはformattersというフォルダとsprintf.dartというファイルが存在している。このファイルが基となるファイルではあるが、これにはpart宣言が5行書かれている。即ち5つのファイルのソースコードで構成されることをコンパイラに対し宣言している。

その中のfloat_formatter.dartを見ると次のようになっている:



即ち最初の行のpart of sprintf;というpartヘッダでこのソースコードはsprintfの一部であることを示している。残りの4つのファイルの先頭にもこのヘッダが付されている。

12.3節　　経過

インポートに関して以下のような変更がなされてきている。

1. インポートしたライブラリに新しい名前が追加されて、その名前がインポートする側に既に存在する名前と衝突する場合は、既にあった名前が優先され、エラーにはしない。(仕様書0.09版で導入)

2. インポートしたライブラリたち間で名前がぶつかっている場合は、その名前をインポートする側で使用していなければエラーにはしない。(仕様書0.09版で導入)

3. 選択的インポート(Selective imports)(仕様書0.10版で導入)
   インポートするライブラリの名前たちのひとつがインポートする側の名前と衝突しているときは、これまではプレフィックスを付けて、そのライブラリにある名前たちを使うときは総てにプレフィックスつきで使用しなければならなかった。しかし必要な名前だけを指定できれば、プレフィックスを付す必要が無くなる可能性が出てくる:
   ```
   import('somelib.dart', show: ['foo', 'bar']);
   ```
   この記述はfooとbarという名前だけが必要な場合に使用する。逆に、
   ```
   import('somelib.dart', hide: ['foo', 'bar']);
   ```
   は、fooとbarという名前が不要である(例えば衝突している)ことを指定する。

4. 再エクスポート(Re-export)(仕様書0.10版で導入)
   あるアプリケーションの組み換えを行う際に、あるクラスをあるライブラリから別のライブラリに移す必要が出たが、しかしこれまでのアプリケーションはそのまま活かさねばならない状況を考えてみる。その為にDartでは再エクスポートを可能としている。あるライブラリをインポートする際に、インポートする名前たちを再エクスポート出来るかどうかをして出来る。
   ```
   // bar.dart
   someMethod() => print('hi!');
   anotherMethod() => print('hello!');

   // foo.dart
   ```

```
import('bar.dart', export: true);
```

この例では、bar.dartというライブラリは再エクスポートできることを指定している。つまりfoo.dartをインポートしたどのコードも、あたかも自分がbar.dartをインポートしたようにsomeMethod()及びanotherMethod()にアクセスできる。無論これに選択的インポートを組み合わせることも可能である:

```
import('bar.dart', show: ['someMethod'], export: true);
```

この場合はsomeMethodのみが再エクスポート可能となる。

更に2012年7月17日に仕様書担当のGilad Brachaからライブラリに関する仕様(第12章)の大幅変更の提案がなされた:

1. #libraryがlibraryに変更
2. #sourceがpartに変更
3. sourceされたファイルは'part of'を持つ
4. ライブラリ名が有用になった
5. インポートされたライブラリからの名前を再エクスポートできる
6. show及びhideを使ってインポートされる名前のセットを制限できる

更にM1変更では#は不要となった。

12.4節　後回しのロード(Deferred Loading)

Dartチームは2014年8月28日にDart 1.6から後回しのロード(Deferred LoadingまたはLazy Loadingともいう)が実験的に利用可能になったと発表した。これはECMAで改版予定になっている項目のひとつである。

後回しのロードは静的変数の後回しの初期化と似て、ブラウザでの立ち上がりを早くするのがその目的のひとつである。

これによりアプリケーションは必要なときにオンデマンドでライブラリをロードできる。この技術はアプリケーションの初期スタートアップ時間を短縮するのに使える。また次のような場合にこの技術が使える可能性がある:

- 他国言語化　－　別々のライブラリに置かれた言語変換
- A/Bテスト　－　異なったライブラリに置かれているアルゴリズムの代替的実装(ある試験者がライブラリAを使い、一方他者がライブラリBを使う)
- ユーザ調査　－　ランダムに選択されたユーザが動的にロードされた調査票を埋める
- オプションのスクリーンとダイアログ　－　たとえば設定パネルのようにユーザが滅多に使わない機能は動的にロードできる

下図はその例である:

後回しのロード

```
import 'package:deferred/hello.dart' deferred as hello;
greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

deferred asに続く識別子（ここではhello）が使う側のプレフィックスとなる。

この機能は公開されていないloadLibrary という関数が暗示的に使われている。その詳細はDartチームの解説に書かれている。この関数は非同期でFutureを返すので、ここではasync関数とawait文が使われている。

この例では、helloというライブラリがロードされたら、それを待ってそのライブラリの中のprintGreeting()という関数を呼び出している。

第13章 Dartの型処理と型チェック (Types)

Dartでは各オブジェクトの実行時の型は**Type**というクラスのインスタンスとして表現され、これはDartのクラス階層のルートにある**Object**クラスで定義されているゲッタの**runtimeType**を呼ぶことで取得できる。

Dartではユーザが定義した型や型アノテーションを次のように扱っている:

実行時(チェック・モード)	渡される値の動的な型をチェック
実行時(生産モード)	型アノテーションは進行に影響を与えない。進行できない問題が起きたときには例外をスローする

Dartの静的型チェッカはJavaのそれよりはずっと寛大である。

JavaScriptのような動的型づけ言語では、ある変数にある値を代入するときには、その変数はその値の型に設定される。その為に各値ごとにその型を識別する為のタグが付いている。Dartもこれを踏襲しており、型アノテーションは尊重はするが可能な限りその値の代入を行い処理を継続する。

静的型チェックが寛大な分、動的な型チェックがチェック・モードでなされる。チェック・モードはそのプログラムの開発段階で使われるモードである。チェック・モードで走らせた場合、パラメタ渡し、結果の戻し、及び値の代入の時点で何らかの型チェックがなされる。Dartは実行時のオブジェクトの型が宣言されている静的型の副型であるかどうかをチェックする。チェック・モードで引っ掛かったらDartはその時点で実行を停止し、その理由をメッセージで示す。

非チェック・モードでは、Dartはその実行を停止せず、その型のままでそのオブジェクトの代入や渡しを行う。

Dartのサイトにあったサンプルを示す:

```
main() {

  // 静的型警告のシンプルな例
  String s1 = '9';
  String s2 = '1';
//  ...
  int n = s1 + s2;
  print(n);

  /* 非チェック・モードではこのコードは走り、nには91がセットされプリントされる。
チェック・モードでは型チェックで失敗と表示され停止する */
}
```

もうひとつのサンプルを示すと:

```
main() {
  Object lookup(String key) {
  // 異質のものたちのテーブルからの検索をする関数
  }

  String s = lookup('Frankenstein');
```

```
}
```

DartはObject型のオブジェクトをString型の変数に代入する際に警告を出さない。これはlookup('Frankenstein')がString型のオブジェクトを返すことをプログラマが想定していると見做しているからである。

Dartの開発者はこれを「楽観的(optimistic)」な型チェックと称している。これはプログラマが動的型付けの利点を十分活用できるようにするという、基本的な目標をベースにしているからだとDartの開発者が述べている。

13.1節　　上位クラスのオブジェクトへの代入と下位クラスのオブジェクトへの代入

Dartは継承もと(スーパークラス)の変数への代入(up-assignment)だけでなく、その反対の継承先(サブクラス)の変数への代入(down-assignment)(ダウンキャストともいう)も許している。

最初に上位クラス・オブジェクトへの代入であるが、次のコードを見て頂きたい:

```
main() {
  String s = 'Hello';
  Object o = s; // up-assignment
  print(o);           // Hello
  print(o is Object); // true
  print(o is String); // true
  print(o.length);    //.5
  num n = 1.41;
  print(n);           // 1.41
  print(n is num);    // true
  print(n is int);    // true
  print(n is double); // true
  int i = n;  // down-assignment
  print(i);           // 1.41
  print(i is num);    // true
  print(i is int);    // true
  print(i is double); // true
}
```

このコードはチェック・モードにおいても警告されない。プログラマがiを整数としてプリントしているにもかかわらずDartは1.41とプリントする。

上位クラス・オブジェクトへの代入では、Object 型と宣言されたoは、o = sという代入により、Object型のoはString型にもなっている。従って代入されたoに対しStringのgetメソッドであるlengthを呼ぶことができる。Javaでは上位クラスへの代入は可能ではあるが、その場合は上位クラスのメソッドしか使えない。

またnum型と宣言されたnは、1.41という値が代入されたことによりint型とdouble型にもなっている(Javaでは下位クラスのオブジェクトへの代入はキャスト出来ないとしてエラーになる)。従ってo = s及びi = nという代入はそのまま代入されている。静的型チェッカはこれらの代入に対して警告を出すことをしない。

次のサブクラスへの代入例ではどうだろうか:

```
main() {
  try {
   String s;
// s = new Object();   // down-assignment, fails in checked mode
   Object foo(){return "Hi";}
   s = foo();           // down-assignment, works fine
   print(s);            // Hi
  } catch (var e){print(e);}
}
```

- この場合はs = new Object()という代入はチェック・モードで代入型エラーとなる。
- しかしこの文をコメント・アウトすると、次の文のs = foo()というObject型オブジェクトからString型オブジェクトへの代入に対しては型チェッカは警告しない。Object foo(){return "Hi";}という関数は、実行時は実際はString型を返すのでs = foo()という代入をチェック・モードでもエラーにしない。

DartチームのEli Brandtが書いている記事には哺乳類(Mammal)というクラスと、その副型のモー(moo)と鳴くメソッドを持つ牛(Cow)及びブーブー(oink)と鳴くメソッドを持つ豚(Pig)というクラスを使った例が示されている:

```
class Mammal {}
class Cow extends Mammal { String moo() => 'moo!'; }
class Pig extends Mammal { String oink() => 'oink!'; }

main() {
  Mammal mammal = new Mammal();
  Cow cow = new Cow();
  Pig pig = new Pig();

  try{
    mammal = cow;      // [1]
    pig = mammal;      // [2] 静的チェックは通過;
                      //      実行時はチェック・モードで動的チェックに引っ掛かる

    // 非チェック・モードではpigはCow 型となる
    print(pig.oink()); // [3] NoSuchMethodExceptionの例外が生起される
  } catch (e) { print(e); }
}
```

- [1]のスーパークラスのオブジェクトへの代入は問題を起こさない。この時点でmammalはCow型であるとともにCow型でもある。
- [2]のサブクラスへの代入は静的型チェッカではそれが正しい場合もあるので警告をださない。しかしmammalはPig型ではないのでチェック・モードで動的な型チェックに引っ掛かりエラーとなる。
- 非チェック・モードではこの代入が行われ、pigはなんとCow型になる。従って[3]のようにPigのメソッドを呼ぶとNoSuchMethodExceptionの例外が生起される。

一般の言語であればそのような副型への代入は許さないが、Dartでは「楽観的」取り扱いをしている。これはすごいポリモーフィズムであり、Javaのプログラマを当惑させよう。

13.2節　甘い静的型チェックは安全でないことを意味しない

しかしEli BrandtはDartの型システムは「甘い」チェックになっているがそれは「決して安全でない」ということを意味しないと主張している。上の例では実行時には非チェック・モードではきちんとエラーをNoSuchMethodExceptionの例外として検出している。

pigに代入されるmammalの値が確実にPig型であるようにしたいプログラマはその代入文の前に次の文を挿入したいと思うだろう:

```
assert (mammal is Pig);
```

こうすればチェック・モードでFailed assertionの例外でそれを知らせてくれる。しかしながら実はチェック・モードではまさしく同じことをやっており、次のような警告をだす:

```
Failed type check: type Cow is not assignable to type Pig
```

つまりチェック・モードではT x = oという代入に対してはassert(o === null || o is T)文が置かれたと等価のチェックを実施している。

無論このような静的型づけを導入したことにより、Javaのような厳格なものに比べれば多少劣るだろうが、JavaScriptよりはずっと安全なプログラムを実現できる。

13.3節　総称型の共変性

総称型の共変性(covariance)とは型パラメタが継承関係にあれば、総称型も継承関係を持つことをいう。Javaにおいても配列は共変性を持っているが総称型ではできない。しかしDartではListのような総称型も共変させることができる。しかし仕様書の13章「型」では次のように書いている:

Dartの型システムは総称型の共変性の為にしっかりしたものではない。これは慎重に考えるべき(そして間違いなく意見が割れる)選択である。経験は総称型のためのしっかりした規則はプログラマたちの直観とは真っ向から対立することを示している....

次の例を見てみよう:

```
main() {
  List<String> strings = new List<String>();
  strings.add("Time : ");
  List<Object> objects = strings; // これはJavaでは不可
  objects.add(new Date.now());    // チェック・モードではエラー
  // 非チェック・モードではこれでstringsがString以外のオブジェクト（Date型）を持つ
  print(strings[1] is Date);
  print(strings[0] + strings[1].toString());
}
```

Object型のオブジェクトたちのリストであるobjectsにString型のオブジェクトたちのリストであるstringsを代入している。共変性によりobjectsにたいしObjectのメソッドを適用すればstringsもそれに応じ操作されることになる。非チェック・モードではobjectsに対してはどんなオブジェクトもadd()メソッドを提供できる。ここではobjectsにDate型のオブジェクトを追加している。従ってstringsにDate型のオブジェクトが入ってしまっている。しかしobjectsにString型のオブジェクトを追加する限りこのコードは何の問題も起こさない。

Eli Brandtが示している例を見てみよう:

```
class Mammal {}
class Cow extends Mammal { String moo() => 'moo!'; }
class Pig extends Mammal { String oink() => 'oink!'; }

// Uses "list" covariantly.
Mammal peekMammalList(List<Mammal> list) {
  return list[2];
}

main() {
try {
  List<Cow> cowList = new List<Cow>(4);
  cowList[2] = new Cow();
  var o = peekMammalList(cowList);
  print(o.moo());
  // Covariant use:
  // * static type checking OK
  // * dynamic type checking OK
  // * runs happily
  } catch(var e) { print(e); }
}
```

ここでもMammal型のオブジェクトリストを引数とする関数peekMammalに、その副型であるCow型のオブジェクトのリストを引数として呼び出すとCow型のオブジェクトが返される。これは静的チェックも動的チェックも通過するし、その動作に問題は生じない。

Dartのチームは、これは型アノテーションをシンプル、軽量、且つオプショナルにするというこの言語のアプローチから来ているという。他の言語の共変性の為のアノテーションやワイルドカードといったものを使いやすいと思っているプログラマは少ない。従ってJavaScriptからDartに発展させ型アノテーションを付加しやすくするには、そのようなものを要求するのは不適切だという主張である。

第14章 組込み識別子、予約語およびコメント (Reserved Words and Comments)

14.1節　　組込み識別子 (Built in identifiers)

組込み識別子はDartにおけるキーワードとして使われる識別子たちであるが、JavaScriptの予約語ではない。JavaScriptコードをDartにインポートする際の非互換性を最小化する為に、これらは予約語とはされていない。

abstract
as
deferred
dynamic
export
external
factory
get
implements
import
library
operator
part
set
static
typedef

14.2節　　予約語 (Reserved words)

予約語は識別子としては使用できない。

Assert
async
async*
break
case
catch
class
const
continue
default
do
else
enum
extends
false

final
finally
for
if
in
is
new
null
rethrow
return
super
switch
sync*
this
throw
true
try
var
void
while
with
yield
yield*

async, async*, sync*,yield及びyield*は新たに追加された。async, async*またはsync*でマークされた関数の中では、async, awaitまたはyieldを識別子として使ってはならない。

14.3節　　コメント

Dartは単行と複行のコメントの双方に対応している。*単行コメント(single line comment)*はトーケン//で始まる。//とその行の終わりまでの総てはDartのコンパイラは無視する。

*複行コメント(multi-line comment)*はトーケン/*で始まりトーケン*/で終わる。/*と*/の間はそのコメントがドキュメンテーション・コメントでない限りなんでもDartのコンパイラは無視する。

コメントはネスト（入れ子に）できる。

*ドキュメンテーション・コメント(documentation comments)*はトーケン/**で始まる。ドキュメンテーション・コメントの内側では、Dartのコンパイラはそれが括弧で囲まれていない限り総てのテキストを無視する。

```
// 単行コメント

/* 複行コメント
   複行コメント
   複行コメント */

/** 複行ドキュメンテーション・コメント
    ドキュメンテーション・コメント
    ドキュメンテーション・コメント
```

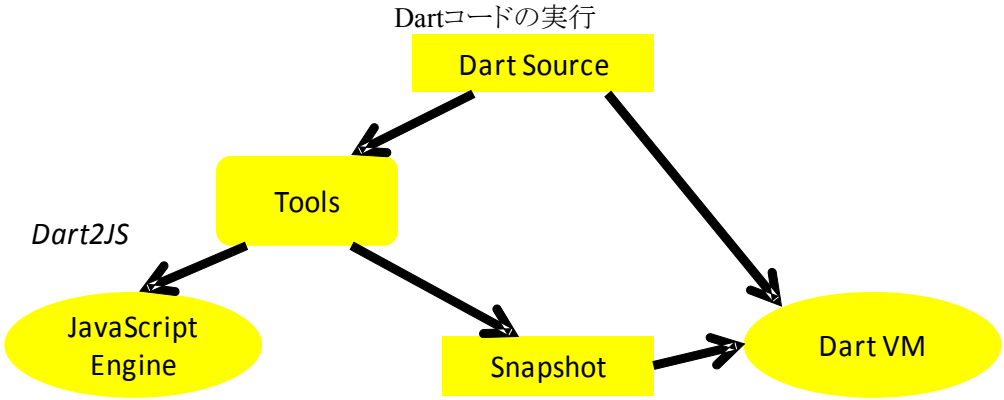
```
*/

/// 単行ドキュメンテーション・コメント
```

ドキュメンテーション・コメントに関しては"Guidelines for Dart Doc Comments"という指針を見て頂きたい。コード内でのコメントの書き方は"Dart Style Guide"のCommentの項を見て頂きたい。

第15章 Dartの実行 (Dart Execution)

Dartで書かれたプログラムの実行には2つの手段がある。ひとつはネーティブ（専用）なVM（仮想マシン）上での実行であり、もうひとつはDartのコードをJavaScriptに変換するDart2JSと呼ぶクロス・コンパイラを使ってJavaScriptエンジン上で実行させる方法である。後者の場合は、Dartでウェブ・アプリケーションを書いて、それをどのブラウザ上ででも実行させることが出来る。クロス・コンパイラはDartC、更にfrogと呼ばれているDart言語で書かれたものが当初開発された。2012年6月からこのクロス・コンパイラはfrogの後継で同じくDartで書かれているdart2jsに一本化されている。

2012年2月16日にGoogleのDartソフトウエア技術者たちがそのブログ(The Chromium Blog)にDart VM実装Chrome（Dartiumというニックネームが付いている）のテスト・バージョンが使えるようになったと発表した。3月末以降はDartのネーティブVMはGoogleのChromeブラウザに実装され、このブラウザはDartiumという名前が付けられている。ちなみにChromeのプレリリース版はChromiumと呼ばれる。DartiumはEclipseライクな開発ツールであるDart Editorに同梱されていた。



上図で注目すべきことはSnapshotと呼ばれる処理である。Dartのsnapshotは該Dartコードを構文解析して得られるトーケン・ストリームのバイナリのストリームである。つまりDartのアプリケーションをロードした後でアプリケーションのヒープには、すべてのオブジェクトがファイルに書き込まれる。また、そのヒープを直列化する処理が含まれていて、これらによりロード時間の大幅な短縮がなされている。例えば54173行からなるDartのコードをロードするには640msかかるが、同じアプリケーションをSnapshotからロードすると60msで済んでしまい、10倍以上の高速化が図れる。通常のJavaScriptのコードをJavaScriptエンジンがロードする時間もSnapshotなしのDartコードのロード時間並みだとGoogleが述べている。Snapshotはまたアイソレート間のオブジェクト渡しにも使用されている。

GoogleはまたDartで書かれたコードをブラウザ上で（Dart VMまたはJSエンジンで）実行させるだけでなく、サーバ上でのDart VMによる実行も可能にしている。これによりフロント・エンドとバック・エンドの双方が同じプログラミング言語で書かれた「Google規模」のウェブ・アプリケーションが可能になる（JavaScriptとNode.jsで書かれたアプリケーション専用サーバのように）。

Dartアプリケーションの開発には以下のようなツール（上図ではToolsと書かれている）が用意されている：

- DartPad： これは非常にシンプルなオンラインのDartコードの開発ツールである。簡単なコマンド行のコードやDart、HTML、およびCSSで構成される簡単なクライアント・コードを試すことができる
- Dart Editor： Eclipseライクな総合開発環境（但し2015年4月にIntelliJ/WebStormが推奨IDEとなり、1.11版からはDart Editorはサポートされなくなった。）
- pub: パッケージ・マネージャでDartコードのグローバルな利用と共有ができる
- analyzer: 静的アナライザで、チェック、エラー検出、解決ヒント生成などを行う（Analysis ServerとしてIDE

のプラグイン、DartPad、Dart Editorなどで共通的に使われてもいる）
- dart2js: 既に述べた一般的なブラウザが実行できるようにするためのDartからJavaScriptへのクロス・コンパイラ
- pub serve: 開発時点でウェブ・アプリを実行させるためのHTTPサーバで、pubコマンドのひとつ
- dart: Dart VM即ちDartコード実行マシンで、サーバでのDartコードの実行に使われる
- Dartium: Dart VM実装のChromiumで、ブラウザ内でDartコードを直接実行する

DartPadとIDE以外はDartのサイトからダウンロードできる。ダウンロードの詳細は「Dart SDKとDartiumのダウンロード」の節に記されている。

読者がDart言語で作成したコードを試すための現時点で最も簡易で便利な手段はDartPadというオンラインの簡単なツールか、あるいは標準的なIDEであるIntelliJ/WebStormあるいはEclipseのプラグインの使用である。なおDartで書かれたプログラムを商用稼働させるときは、読者はChrome等のブラウザが使用されるか（クライアント・コードの場合）、あるいはDart SDKに含まれているDartのVMが使用される（サーバ・コードの場合）ことになる。

Dartium (Dart VM実装Chromeブラウザ)に関する注意：

2015年3月25日にDartチームは、Dart VM実装ブラウザの開発を止め、Dart2JSに特化すると発表した。これは現在Dartでビジネス用アプリケーションを開発しているGoogle社内外のチーム（例えばGoogle Adsのチーム）たちからのよりJavaScriptとの統合を高めて欲しいとの意見によるものだという。但しVMそのものはサーバやツール用に開発を継続する。**Dartiumは将来Chromeに実装されることはないが、ツールとして残される。**これに関する議論はここまたはここを見て頂きたい。

15.1節　　DartPad

DartPadは2015年5月に公式に公開されたブラウザ上でDartで書かれたコードを試すためのオンラインのツールである。この種のツールは以前からTryDartなどとして存在していたが、より多くの機能を含んだものとなっている。DartPadはDart言語、コアとなっているライブラリたち、およびHTML/CSSに対応しているので、小規模なウェブ・アプリケーションにも適用できる。但しdart:ioライブラリを使用するコードは利用できない。また現時点では外部パッケージを使用するコードは利用できない。またプロキシを介したアクセスでは正しく動作しない場合があるので注意のこと。

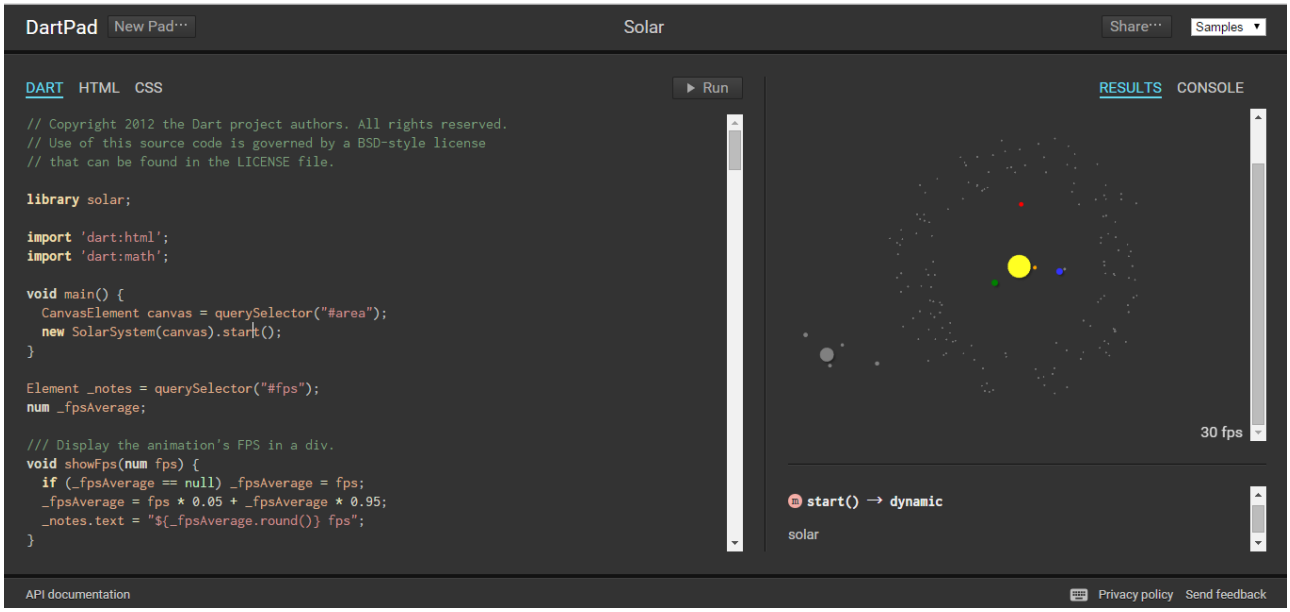
DartPadを開く

DartPadを使うときはなるべくChromeブラウザを使用することを推奨する。このページを開くと左側がコード（.dart、.html、および.css）部、右側が出力部となっている。RESULTSはブラウザ出力で、ウェブ・アプリケーションで使われる。CONSOLEはコンソール出力で、コマンド行アプリケーションで使われる。

ユーザは左側のコード領域で自分のコードをコピー/ペーストあるいは手入力して、その実行結果をRunボタンをクリックすることで知ることができる。エラーはリアルタイムに指摘してくれる。

サンプル（太陽系）を試す

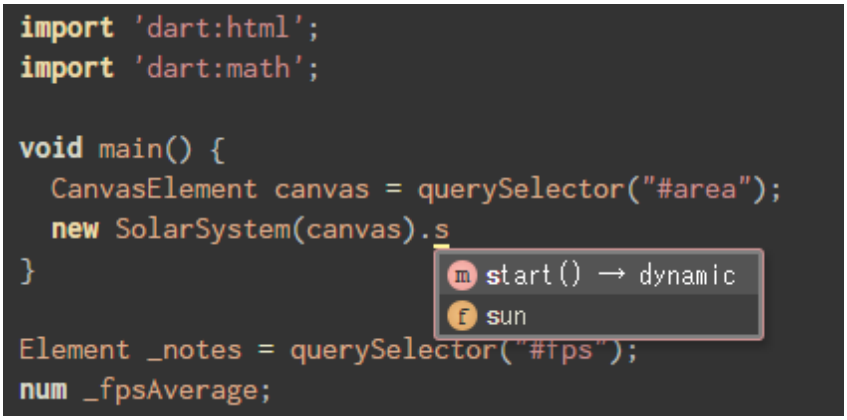
右上のSamples選択メニューからSolarを選択、Startボタンをクリックすると下図のような結果が得られる：



太陽系サンプル

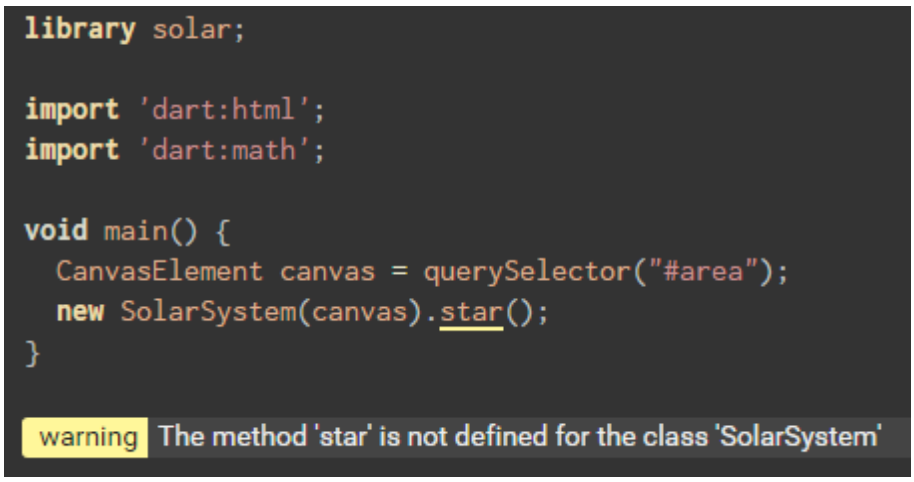
このサンプルは水星、金星、地球と月、火星、木星と4つの惑星、および火星と木星間の小惑星帯をシミュレートしたものである。

コード入力中に可能なAPIを以下のようにガイドしてくれる：



APIガイド

またエラーもリアルタイムで指摘してくれる：



エラーの指摘

使ったAPIのドキュメントは知りたい個所を右クリックすれば画面の右下に表示される。太陽系サンプルの画面では、startメソッドを右クリックした結果として、そのドキュメントが表示されている。

作成したコードをGistに永久的に保管し、他のユーザと共有することも可能である。

本資料のコードを試す

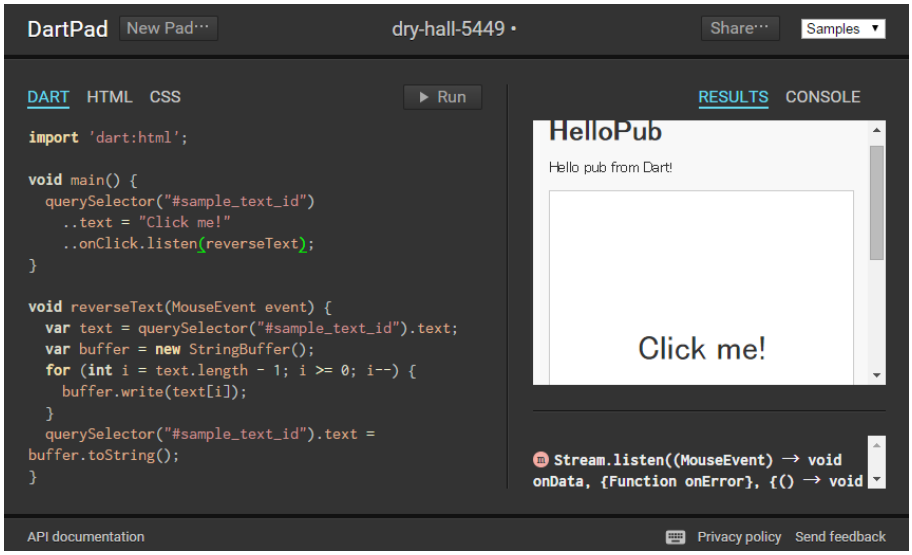
本資料には多くのサンプル・コードが教材として含まれている。Githubに登録されているdart_code_samplesはそれらのなかでも主として基本的なサンプル・コードを集めたものである。そのなかのcodesは各章にあるcode_xx.yy.dartの名前（xxは章番号、yyは節番号）で表示されたコード・サンプルが含まれており、それらはDartPadで簡単に試すことができる（dart:ioをインポートしているコードを除く）。

注意：HTML及びCSSの領域はクリアしておくこと。

下図は第1章にあるcode_01_1.dartをコピー/ペーストした例である：

code_01_1.dart

dart_code_samplesのなかのappsは基本的なアプリケーションのサンプルを集めたものである。そのなかでもDartPadでテストできるものがある。例えばHelloPubは完全なクライアントだけのサンプルなので、そのなかのwebディレクトリにあるhellopub.dart, hellopub.html, hellopub.cssの3つのファイルを貼り付け実行させると、下図のようになる:



HelloPub

Click me!をクリックすると文字列が反転する。

15.2節　IntelliJ IDEA CE

本解説書の第32版以降では、統合開発環境（IDE）としてIntelliJ IDEA CEを使用している。

2015年4月28日に開催されたDart Summitのなかで、1.11版から推奨開発環境（IDE）として従来のDart EditorからDart Plugin for IntelliJ/WebStormに切り替え、Dart Editorは廃止すると発表された。これはかなり衝撃的な動きで、Dart VM実装Chromeの開発停止の発表とともに大きな議論を起こした。Dart EditorはDart言語開発の初期からこのコミュニティのなかで馴染み親しまれてきたものだったからである。

これはGoogle Adsのような大規模ウェブ・アプリケーションのユーザの意見を受け、よりウェブ・アプリケーション対応にリソースを振り向け、またウェブ・アプリケーションの為のIDEとしてはむしろより高度な機能を持っているコミュニティ（Eclipseのような）あるいはサード・パーティ（IntelliJ/WebStormのような）のツールを活用したほうが良いとの判断による。

発表では代替IDEとして:

- 初歩者向けの小さなコード片のテストにはDartPad
- 専門家たちによるDart開発にはWebStormのDartプラグイン

が推奨されている。

JetBrains社が提供するIntelliJ IDEAとWebStormの双方にはDartプラグインが同梱されている。WebStorm（2015年5月時点で￥5,724）はIntelliJの軽量版であり、Java開発関係が省かれている。無料で使えるIntelliJ CE(IntelliJ Community edition)ではクライアント側DartおよびHTML、CSS、JavaScriptのデバッグができないことに注意。

Dart EditorはDart開発の初期からGoogleのEric Claybergが中心になって進められてきたものでJavaで書かれており、Eclipseライクでコンパクトではあるが非常に有用なツールである。特にこの資料のような教材のためには最適なものである(ここを参照のこと)。インストールもダウンロードして解凍するだけでよい。一方EclipseのDartプラグインはDart Editorと同じJavaのコード・ベースであり、引き続き使用可能でEclipseが普及している日本のユーザたちには好ましいものではあるが、残念ながらWindowsには対応していない。

以上の理由から2015年6月以降、本資料ではIDEとして無料で利用できるIntelliJ IDEA CEを採用することとした。多少使える機能は限定されるが、Dartを試してみたい読者には、ツールが無料であることが絶対条件である。IntelliJ CEとWebStormのDartコード開発の為の機能の相違はStack Overflow上で議論されているのでそれを見ていただきたい。

IntelliJ CEで出来る機能:

- サーバ・サイドのアプリのデバッグ
- Pubspec.yaml と Pub の統合 (Pub Get)

IntelliJ CEで出来ない機能:

- YAML サポート(YAMLファイルの編集)
- HTML、CSS サポート
- ブラウザ・プラグインを介したJavascriptのデバッグ
- ブラウザ・プラグインを介したDartコードのデバッグ

Dart言語にある程度馴染んだあとで、WebStormまたはIntelliJ IDEAを購入されることをお勧めする。

IntelliJ IDEA CEのダウンロードとインストール

IntelliJ IDEA CE(Community Edition)のダウンロードとインストールは従来のDart Editorに比べるとやや複雑なので、「推奨IDE(IntelliJ / Webstorm)のインストール」という章にして説明してあるので、まずこの章に従って各自インストールをお願いする。この章の最初の「Dart SDKとDartiumのダウンロード」の節にはこのIDEでDartのプログラムを開発するのに必要なDart SDKとDartiumのダウンロードの詳細が記されている。

サンプル・コード(dart_code_samples)をひらく

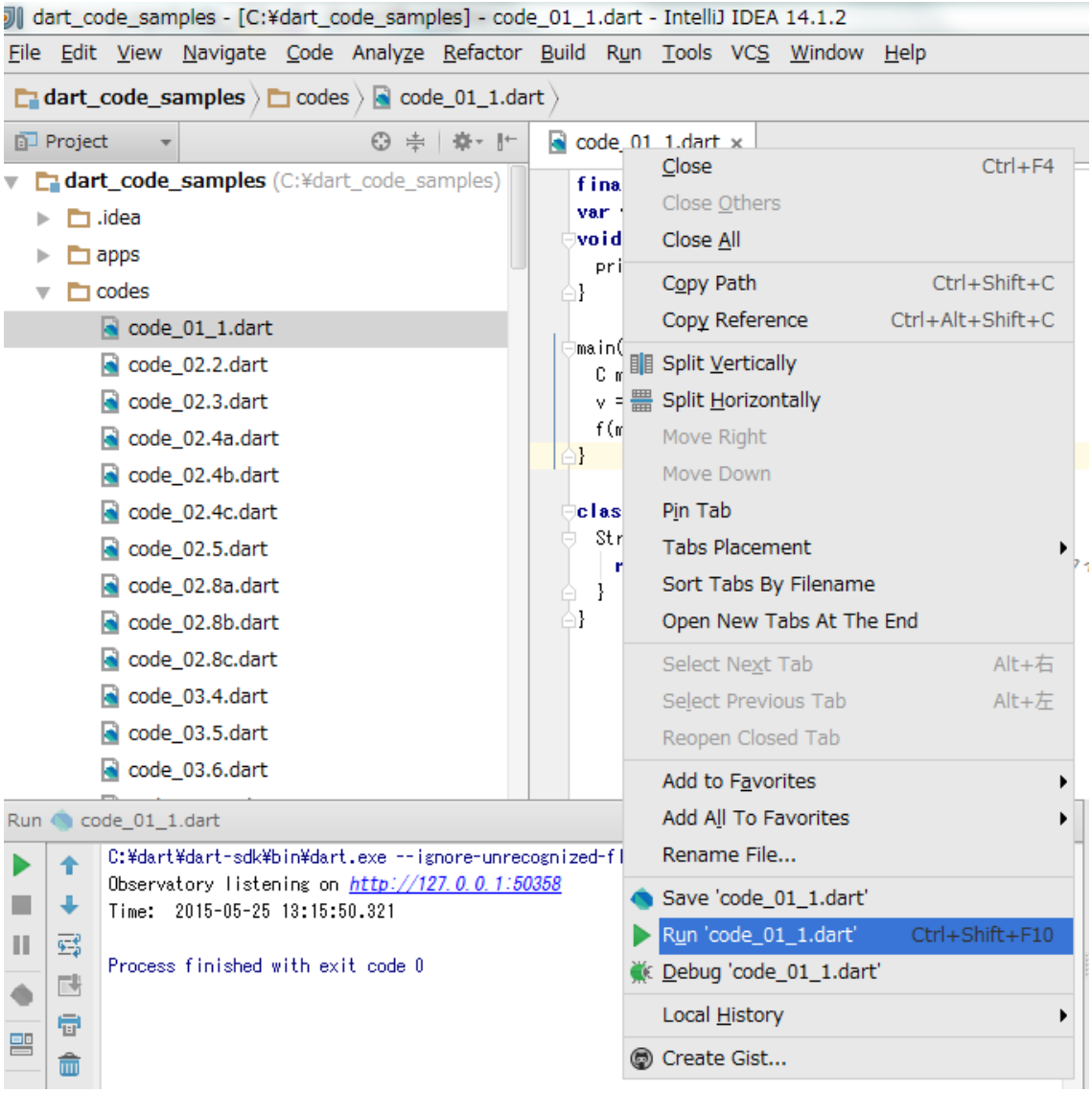
本資料で使われているサンプルはGithubに登録されているので、「本資料に含まれているプログラムのダウンロード」の章に従ってdart_code_samplesをIntelliJ IDEA CEで開く。その際「外部パッケージの取り込み」の節で記されている手順を忘れないようにする。

コマンド行プログラムのサンプルを試す

Dartチームはブラウザ上で走るアプリケーションをウェブ・アプリケーションと呼んでいる。これに対し、サーバ上でVMが直接実行するアプリケーションをサーバ・アプリケーションと呼ばれる。またDartではサーバ・アプリケー

ションのことをコマンド行アプリケーション(Command-Line Apps)とも呼ぶ。これはユーザがブラウザでHTMLファイルを呼ぶことで起動させるのではなく、DOSのコマンド行（WindowsではDOSプロンプト）を使って動させるプログラムであることによる。コマンド行アプリケーションはまたコンソール・アプリケーション(Console Application)と呼ぶ場合もある。

本資料ではDart言語解説のために多くのサンプル・コードが添付されているが、これらはコマンド行プログラムである。コマンド行プログラムは具体的には.dartファイルに収容されたDartコード（及び必要なライブラリのファイルたちのパッケージ）のことである。コマンド行のサンプルはdart_code_samplesの下のcodesというフォルダに収容されている。例えば最初のcode_01.1.dart（第1章の最初のサンプル・コード）はこれをダブル・クリックしてコード表示域に表示されるので、これを右クリックするとこのコードに対する処理の一覧が下図のように表示される:



コマンド行コードの実行

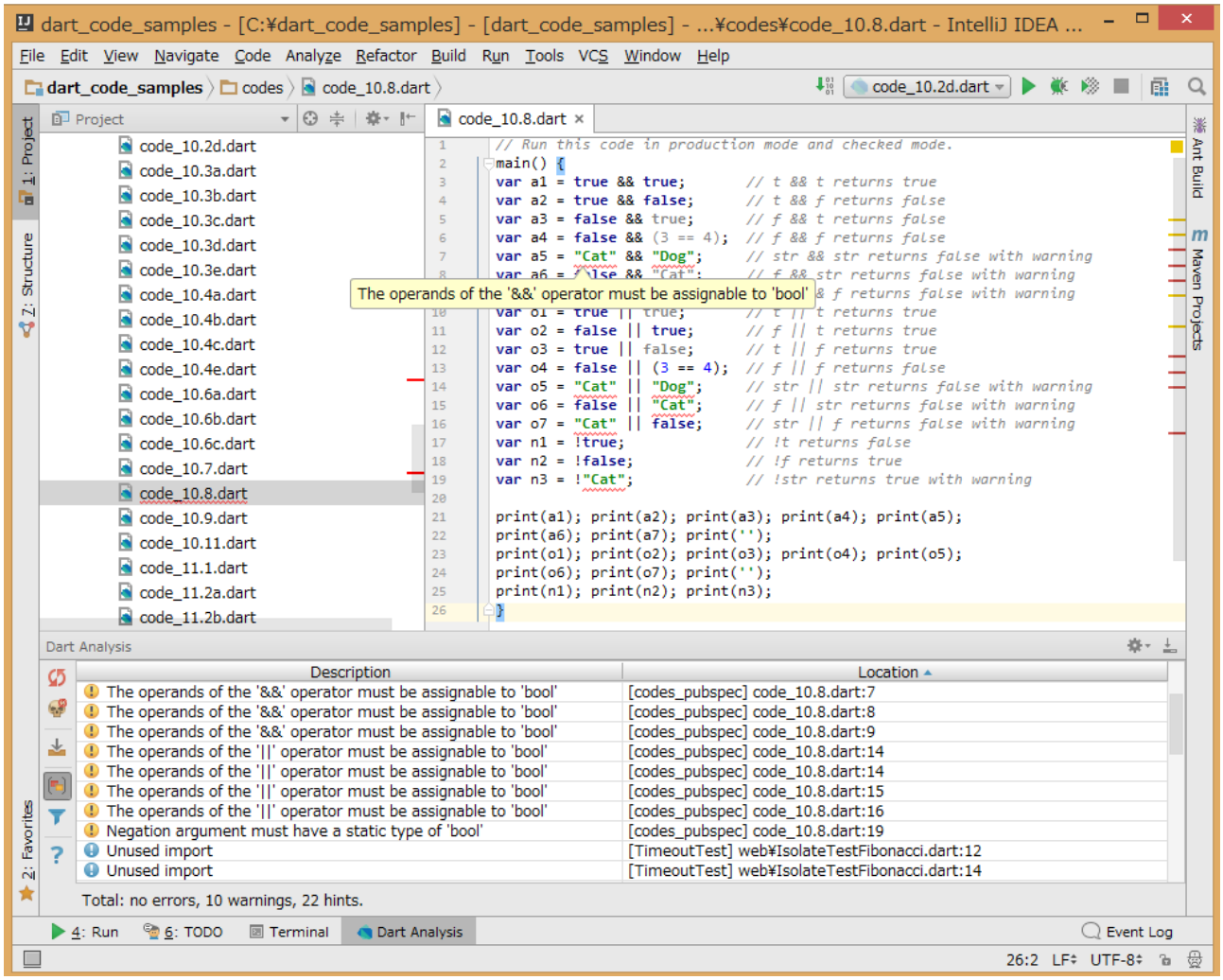
ここでRun 'code_01_1.dart'を選択すれば、このコードが実行され、その結果が左下のコンソール域に表示される。

```
C:\dart\dart-sdk\bin\dart.exe --ignore-unrecognized-flags --checked --enable-vm-
service:50358 --trace_service_pause_events C:\dart_code_samples\codes\code_01_1.dart
Observatory listening on http://127.0.0.1:50358
Time:  2015-05-25 13:15:50.321

Process finished with exit code 0
```

エラーの処理（チェックド・モードと運用モード）

「概要」の章で解説したように、Dartでは静的型チェックやエラーはチェックド・モード(checked mode)では厳格に検出されるが、運用モード(production mode)では極力実行を進めようとする。特に型アノテーションは運用モードでは無視される。これらはDartのアナライザ・エンジンによって行われる。それをcode_10.8.dartで確認してみよう。

このコードはIDEのエディタ上では次のようにリアルタイムで問題個所が表示される。その個所にカーソルを置くと問題点の記述が表示される。また下部にあるDart Analysisを選択すればそれらの一覧が表示される。

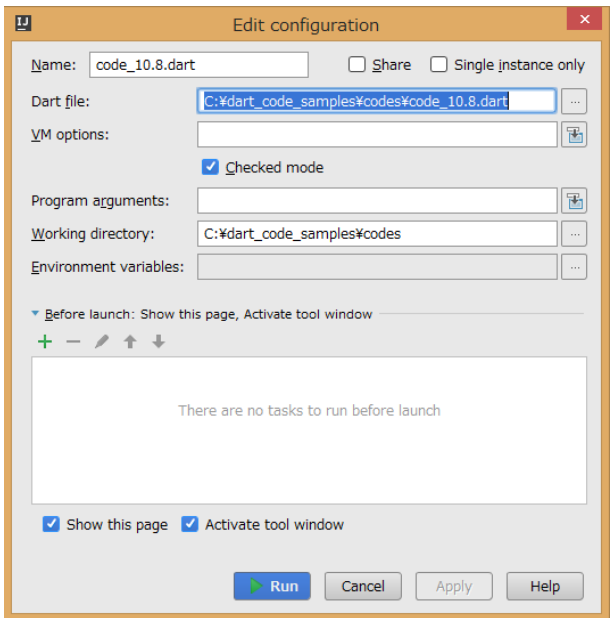


エラーの表示

この図では問題箇所がユーザに分かるように赤のギザギザのアンダラインで表示されている。

それではこのコードをチェックド・モードと運用モードで実行させてみよう：

1. Run → Edit configurationsで実行開始の為の設定のウィンドウを開き、下図のようにChecked modeをチェックしたのちRunボタンをクリックして実行させる：



チェックド・モードでの開始設定

そうするとコンソールには以下のように7行目のエラーを検出して実行を停止したことが表示される:

```
Unhandled exception:
type 'String' is not a subtype of type 'bool' of 'boolean expression'.
#0      main (file:///C:/dart_code_samples/codes/code_10.8.dart:7:10)
#1      _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:255)
#2      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:142)

Process finished with exit code 255
```

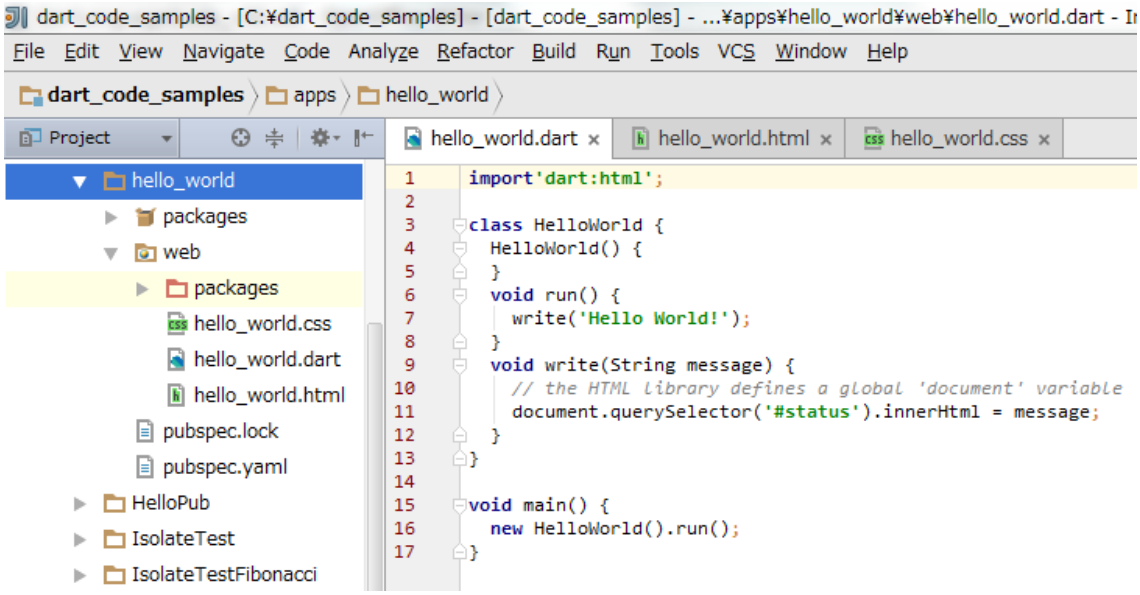
2. つぎにChecked modeのチェックを外して実行させると、次のように最後まで実行される:

```
true
false
false
false
false
false
false

true
true
true
false
false
false
false

false
true
true

Process finished with exit code 0
```

ウェブ・アプリケーションのサンプルを試す

ウェブ・アプリケーションというのはブラウザが開くアプリケーションであり、通常はHTML(.html)ファイル（及びCSSファイル）、Dartのコードのファイル(.dart)（及び必要なライブラリのファイルたちのパッケージ）またはそれをJSに変換したファイル(.dart.js)が最低限必要である。ウェブ・アプリケーションのサンプルはappsというフォルダに収容されている。

例えばhello_worldというサンプルは次のようになっている：



hello_world

これは最もシンプルなウェブ・アプリケーションである。このアプリケーションは以下の3つのコードが核となっている。

- hello_world.dart（dartで書かれたコード）
- hello_world.html（dartコードを含めてこのアプリケーションを実行させるhtmlファイル）
- hello_world.css（このページのスタイルを記述したcssファイル）

加えてこのアプリケーションは次章で説明する「パッケージ」の構成となっている。従ってこのパッケージを規定する

- pubspec.yaml

というファイルもユーザが作成する必要がある。

それ以外のファイル(pubspec.lock)とディレクトリ(packages)はパッケージ・マネージャがpubspec.yamlを読んで用意したものである。packagesにはこのアプリケーションに必要なライブラリたちが収容されている。

Dartではウェブ及びサーバのアプリケーションはパッケージの構成をとる。ユーザはアプリケーションを開発する際はパッケージの知識が必要になる。パッケージの詳細は次章で解説してある。

コードの概説：

このアプリケーションは単に”Hello World”を表示するだけである。

hello_world.dart

```
import'dart:html';

class HelloWorld {
  HelloWorld() {
  }
  void run() {
    write('Hello World!');
  }
  void write(String message) {
    // the HTML library defines a global 'document' variable
    document.querySelector('#status').innerHtml = message;
  }
}

void main() {
  new HelloWorld().run();
}
```

hello_world.html

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title>HelloWorld</title>
  </head>
  <body>
    <h1>HelloWorld</h1>

    <h2 id="status">dart is not running</h2>

    <script type="application/dart" src="hello_world.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

このコードの中で注意する点としては、document.querySelector('#status').innerHtml = message;という文であろう。documentはDocumentのスーパーインターフェイスであるElementのそのまたスーパーインターフェイスのNodeのなかのgetメソッドとして定義されている

```
Document get document();
```

というゲッタを呼んだものである。querySelectorというメソッドはElementインターフェイスの中で次のように定義されている:

```
Element querySelector(String selectors)
```

セレクタはstatusというIDを持ったエレメントを選択する。その要素の中を書き換えるInnerrHtmlというメソッドはElementインターフェイスの中で次のように定義されている:

```
void set innerHtml(String value);
```

ウェブ・アプリケーションの実行手段:

IDE上でウェブ・アプリケーションを実行させるには2つの手段がある:

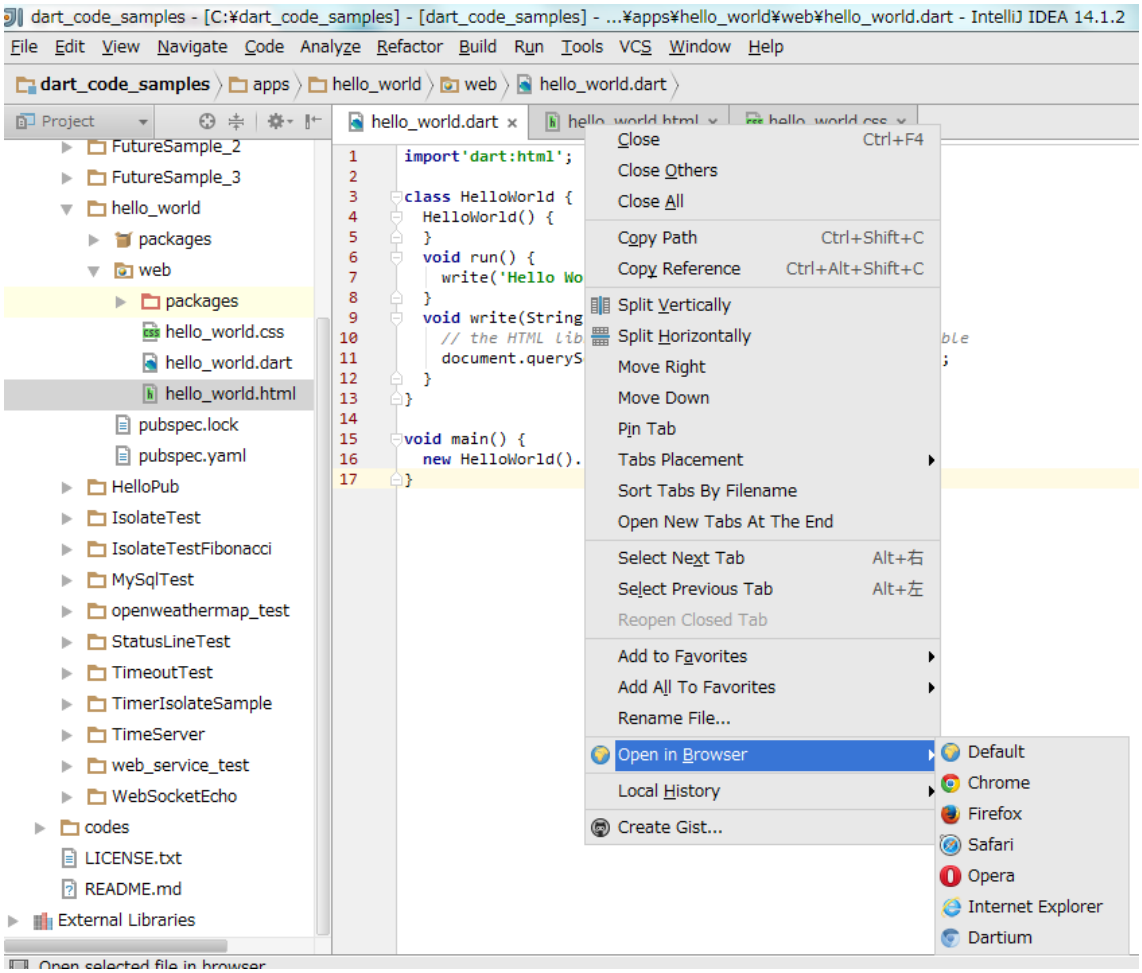
- Dartiumブラウザで直接実行させる

- dart2jsで.dartファイルのコードをJavaScriptに変換して通常のブラウザ（ここではChrome）で実行させる

hello_worldをDartiumで実行させる

実行に先立ち、hello_worldフォルダにはpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込む。そうするとIDE上には以下のように展開される

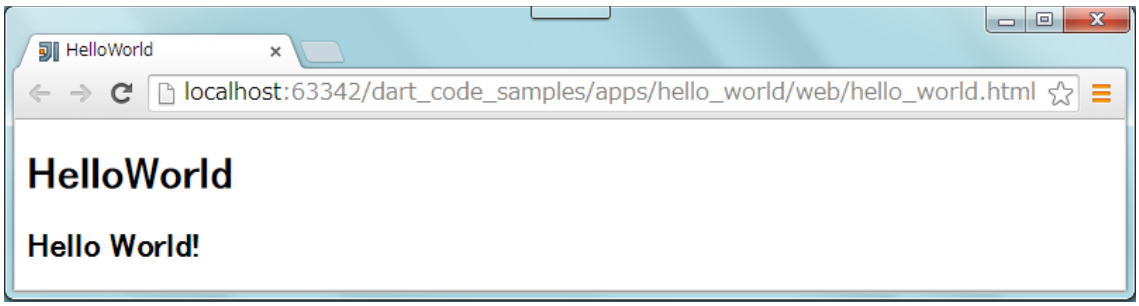
次に下図のようにJDE上でhello_world.htmlを選択し、右クリックするとこのファイルに対する処理の一覧が表示されるので、そこからOpen in Browserで選択メニューのDartiumを選択するとDartiumが起動し、このアプリケーションをサーバ経由で実行する。



実行ブラウザの選択

下図がその実行結果である。IDEが起動したDartiumは以下のようにIDEが用意したウェブ・サーバの代理としてのHTTPサーバに対してhello_world.htmlを呼び出している：

```
http://localhost:63342/dart_code_samples/apps/hello_world/web/hello_world.html
```



Dartiumによる実行

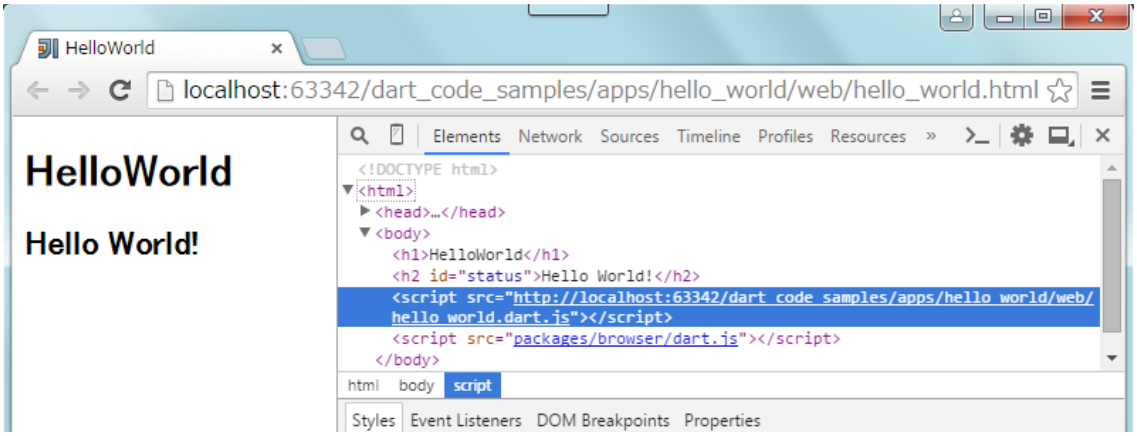
IDEのサーバ（アドレス：http://localhost）経由でこのアプリケーションを実行することで、IDEはそのアプリケーションの実行状況をより詳しく把握できる。

hello_worldを通常のブラウザで実行させる

Dart2JSを使って普通のブラウザで実行させる場合は、Chromeなどのブラウザを選択すれば良い。その場合は自動的に最初にDart2JSでJSコードにクロスコンパイルされ、それが当該ブラウザから実行される。Dart2JSが実行されるため、最初の場合はブラウザに表示されるのに多少時間がかかる。Dartiumとその他のブラウザとのきりわけの詳細は「dart.jsブートストラップ・コード」の項を参照のこと。この場合は実行されるHTMLコード（Chromeではその他のツール→デベロッパ　ツール、IEではF12 開発者ツール→DOM Explorerで調べることができる）は次のようになっている：

```
<html><head>
  <meta charset="utf-8">
  <title>HelloWorld</title>
 </head>
 <body>
  <h1>HelloWorld</h1>

  <h2 id="status">Hello World!</h2>

  <script
src="http://localhost:63342/dart_code_samples/apps/hello_world/web/hello_world.dart.js"></s
cript>
  <script src="packages/browser/dart.js"></script>

</body></html>
```

下図はChromeからの実行とそのときの実行HTMLコードを示す：



Chromeからのhello_worldの実行

hello_world.dart.jsというファイルはdart2JSが作成したものだが、このままではIntelliJ IDEA CE上でアクセスできない。IntelliJ上でJavaScript変換されたファイルを得るにはPub buildというコマンドを使用することになるが、詳細は次章パッケージ・マネージャの「HelloPubをデフォルトのブラウザで実行させる」で解説する。

dart.jsブートストラップ・コード

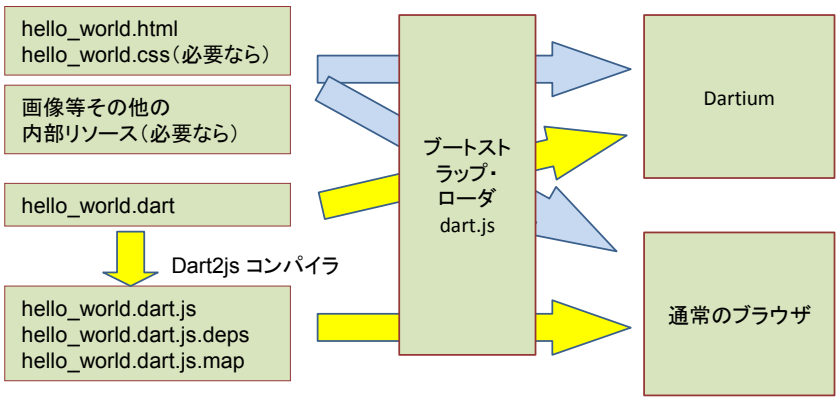
ウェブ・アプリケーションの場合は、サーバ・アプリケーションと違って、Dartで書かれたプログラムを直接実行できるブラウザは現在はDartiumと呼ばれるDartのVMを実装したChromeブラウザしか存在しない。従って通常のブラウザで実行させるには、JavaScriptにクロス・コンパイルされたコードが必要になり、これはIDEがdart2jsコンパイラを呼び出してコンパイルしてくれる。

あるウェブ・アプリケーションをDartiumと通常のブラウザとがともに実行できるようにするには次のものが収容されたパッケージが必要になる:

- そのドキュメントを構成するhtml、css、画像等の内部リソースなどのファイル
- DartiumのVMが実行するdartコードのファイル
- そのdartコードを通常のブラウザが実行する為にJavaScriptにクロス・コンパイルしたjsファイル
- ブラウザがDartiumかどうかでそのどちらかをブラウザに実行させるdart.jsというスクリプト

dart.jsというスクリプトはブートストラップ・スクリプトとも呼ばれるもので、Dart VMの起動、及び非Dart対応ブラウザとの互換性の面倒をみる。即ち下図のHelloWorldの例に示すように、ブラウザはdart.jsのなかで自分がDart対応(即ちDartium)かどうかで.dartのコードをロード・実行するかまたはコンパイルされた.dart.jsコードをロード・実行するか判断する。

ブートストラップ・スクリプト



HTMLファイルの中の<script src="packages/browser/dart.js"></script>というタグでdart.jsの場所が記されている。このタグでわかるように、標準的にはこのコードはpubというパッケージ・マネージャの書式に従いpackages/browser/というサブ・フォルダに置かれる。

エディタに対しそのようなライブラリを配置するよう指示するにはpubspec.yamlというファイルを用意しなければならない。"pubspec.yaml"には以下のテキストが収容されている：

```
name: sample_pubspec
description: A sample application
dependencies:
  browser: any
```

エディタはこのファイルを見てライブラリから必要なライブラリ（ここではbrowser）を取り込む。browserというライブラリにはdart.jsが含まれている。pubspec.yamlに関しては次章で詳しく解説する。

コマンド行からクロス・コンパイラでJS変換させてChromeで実行させる

IntelliJ上ではPub buildコマンドで.dartファイルを.jsファイルに変換したものを含むアプリケーション配布用フォルダを作成することができるが、DOSプロントからdart2jsを実行させることもできる。例えば前述のhello_worldというアプリケーションの場合は：

```
C:\>cd C:\dart_code_samples\apps\hello_world\web
C:\dart_code_samples\apps\hello_world\web>path C:\dart\dart-sdk\bin
C:\dart_code_samples\apps\hello_world\web>dart2js -out=hello_world.dart.js
hello_world.dart
```

- 最初に自分が作成したアプリケーション（ここではhello_world.dart）が入っているディレクトリに移る
- 次にDart VMがあるディレクトリへのパスを指定する。これにより直接dartコマンドが使えるようになる（環境変数で設定すれば、コマンド・プロンプトを開いたたびにこのステップを踏まなくてもよくなる）
- dart2jsを実行する
  - --out（または-o）オプションは出力ファイル名を指定する
  - その他のオプションはDartのサイトのdart2jsコマンドの解説に記されている

以下はそうしたときにdart2jsが作成したファイルたちである：

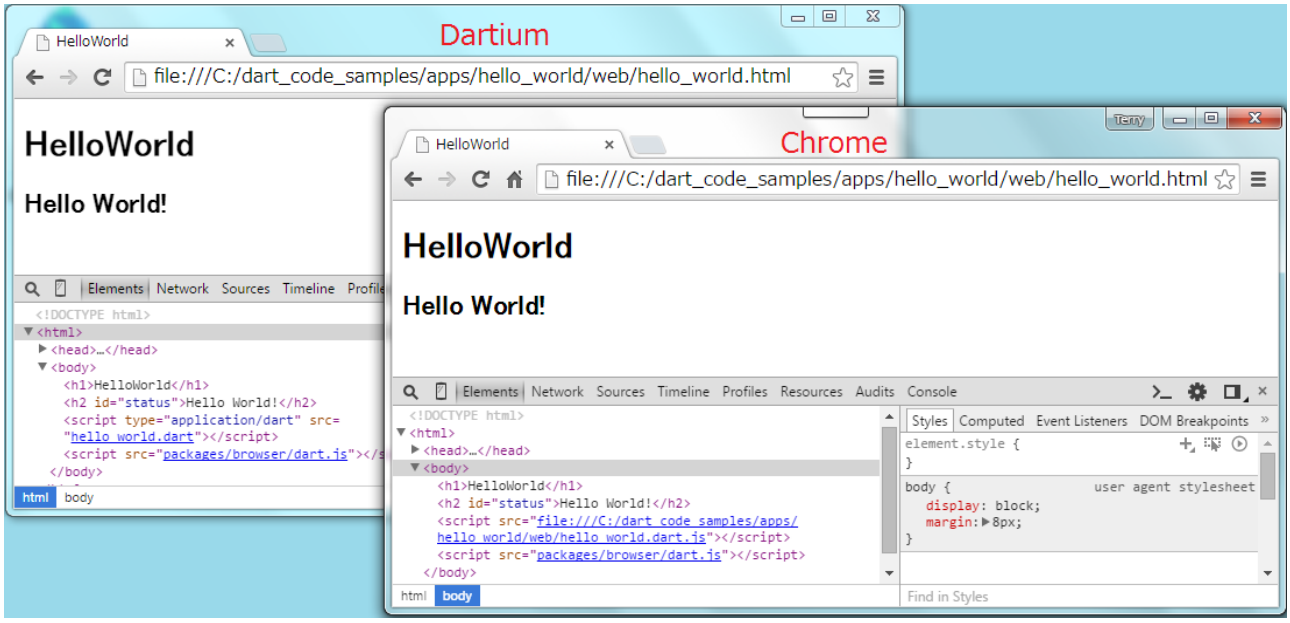
- hello_world.dart.js （クロスコンパイルされたJSスクリプト）

- hello_world.dart.js.deps （コンパイルに使われたすべての参照ファイルたちのリスト）
- hello_world.dart.js.map （デバッグ用のJSからDartへのコード行ソース・マップ）

生成されたこれらのファイルはIDE上で表示されるので、各自試してみると良い。hello_world.dart.jsは3500行にもなっている。

こうすれば通常のブラウザ及びDartiumからfile:///形式でこのアプリケーションを実行させることが可能となる。Dartium単体はこのアドレスから取得できる。

下図はこの形式でDartiumとChromeからhello_world.htmlを呼び出した例である：



file:///形式でDartium及びChromeからこのアプリケーションを実行

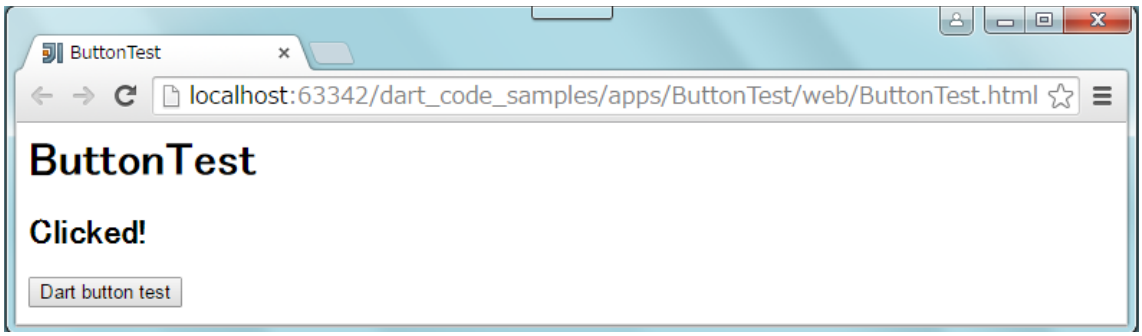
dart.jsブートストラップ・スクリプトにより、そのブラウザに対応してHTMLから呼び出すスクリプト（.dartか.dart.jsか）が異なっていることに注意しよう。

HTML5対応を試してみる

IntelliJ IDEA CEを使ってDartを使ったHTML5対応を試してみよう。dart:htmlライブラリにはその為のAPIが用意されている。より高度なDartによるHTML5利用サンプルはここなどを見られたい。

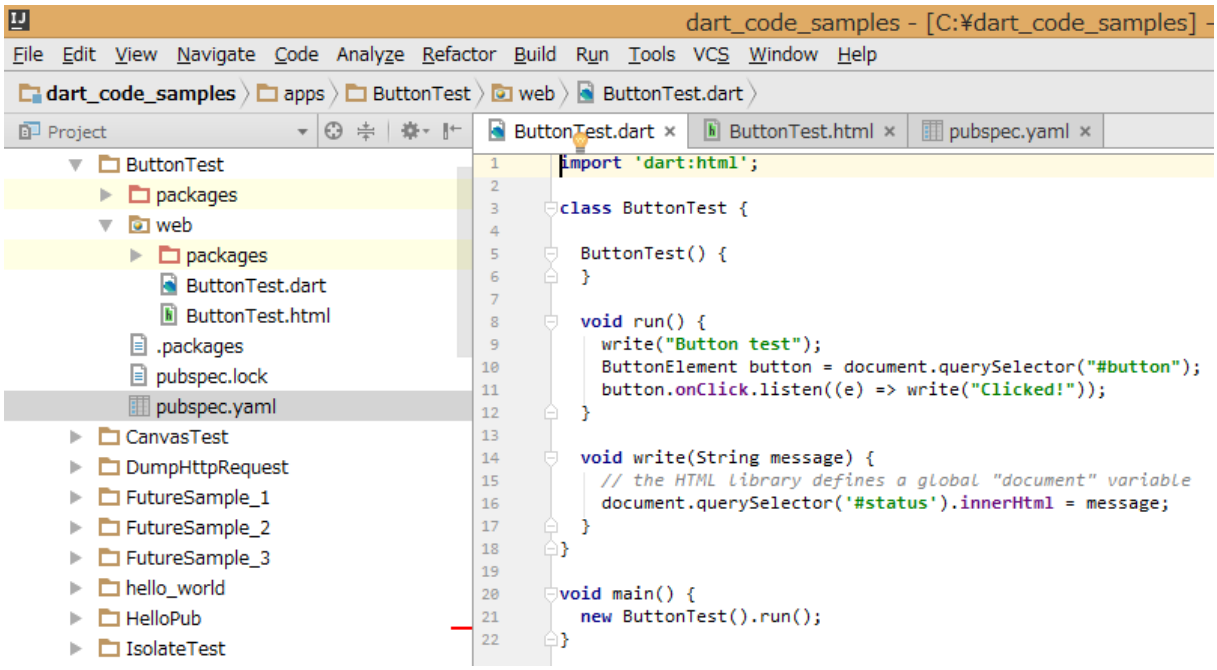
ButtonTest

ボタンをクリックするとそのことを表示するだけの簡単なコードを示す。



ButtonTest実行例

このプログラムはHelloWorldのコードを一部変更しただけのものである。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IDEから\dart_code_samples-master\apps\ButtonTestのフォルダを開くと下図のように展開される:



このアプリケーションにはpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込む。上図はpub get実行後の構成を示したものである。

htmlコードとdartコードは次のようなものである:

HTMLコード(ButtonTest.html)

```
<html>
  <head>
    <title>ButtonTest</title>
  </head>
  <body>
    <h1>ButtonTest</h1>
    <h2 id="status">Dart is not running</h2>
    <button id="button"> Dart button test </button>
    <script type="application/dart" src="ButtonTest.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
```

```
</html>
```

Dartコード（ButtonTest.dart）

```
import 'dart:html';

class ButtonTest {

  ButtonTest() {
  }

  void run() {
    write("Button test");
    ButtonElement button = document.querySelector("#button");
    button.onClick.listen((e) => write("Clicked!"));
  }

  void write(String message) {
    // the HTML library defines a global "document" variable
    document.querySelector('#status').innerHtml = message;
  }
}

void main() {
  new ButtonTest().run();
}
```

読者はIDEを使って実際にこのコードを試して見られたい。ここで注目すべき点はbutton.onClick.listen((e) => write("Clicked!"));というイベント・リスンの文であろう。以前はbutton.on.click.add((e) => write("Clicked!"));と記述するようになっていたが、2013年1月からAPIが変更された。ボタンそのものはButtonElementであるが、それが実装しているElement抽象クラスにはfinal Stream<MouseEvent> onClickというStream型のフィールドが用意されている。

```
final Stream<MouseEvent> onClick
```

Stream抽象クラスにはそのイベントを受信する為のlistenメソッドがある：

```
abstract StreamSubscription<T> listen(void onData(T event), {void
onError(AsyncError error), void onDone(), bool unsubscribeOnError})
```

ここではデータ受信のハンドラとしてwrite()を呼ぶだけの簡単な関数が定義されている。

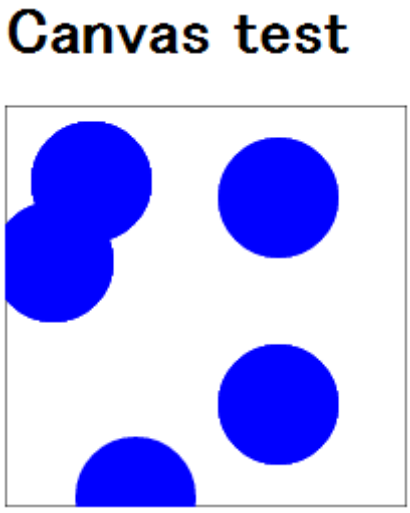
このようにDartではStreamを使って非常に短くその処理を書くことができる。

CanvasTest

リッチなグラフィックスを描画するにはHTML5のCanvas（dart:htmlライブラリを使用する）かSVG（dart:svgライブラリを使用する）を使うことになる。Dartはどちらにも対応するが、ここではCanvasを使用したサンプルを示すことに

する。

このサンプルでは、下図に示すようにキャンバス内でクリックした場所を中心にした円を描く。この場合はイベント・ハンドラをonMouseDown(event)という独立したメソッドにしている。このアプリケーションもGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IDE上で\dart_code_samples-master\apps\CanvasTestのフォルダを開くと良い。このアプリケーションもpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込んでいる。



CanvasTest.html

```
<html>
  <head>
    <title>CanvasTest</title>
  </head>
  <body>
    <h1>CanvasTest</h1>
    <h2 id="status">dart is not running</h2>
    <canvas id="canvas" width="200" height="200"></canvas>
    <script type="application/dart" src="CanvasTest.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

一方CanvasTest.dartは次のようになっている:

CanvasTest.dart　CanvasTest.html

```
1  import 'dart:html';
2  import 'dart:math';
3
4  class CanvasTest {
5
6    CanvasTest() {
7    }
8
9    int width, height;            // キャンバスの寸法
10   CanvasRenderingContext2D ctx; // 2D描画コンテキスト
11   static int radius = 30;       // 描画する円の半径
12
13   void run() {                  // このクラスのオブジェクトの実行メソッド
14     write("Canvas test");       // dart is not runningをCanvas testに書き換え
15     CanvasElement ce = document.querySelector("#canvas"); // 画面からCanvasElementオブジェクトを取得
16     ctx = ce.getContext("2d");  // 描画コンテキストを取得
17     width = ce.width; height = ce.height; // キャンバスの寸法を取得
18     ctx.strokeRect(0, 0, width, height);  // キャンバスの縁を描く
19     ce.onMouseDown.listen(onMouseDown);   // キャンバス内でのマウス・クリックのイベント待機と処理関数の登録
20   }
21
22   void write(String message) {  // #status要素にmessageを描くメソッド
23     document.querySelector('#status').innerHtml = message;
24   }
25
26   void onMouseDown(event) {     // マウス・クリックのイベント処理
27     int x = event.offset.x;     // クリックされた座標を取得
28     int y = event.offset.y;
29     ctx.moveTo(x + radius, y);  // 円の描画開始点
30     ctx.arc(x, y, radius, 0, PI * 2, false); // 円の描画
31     ctx.fillStyle = 'blue';     // その円を塗りつぶす色を指定
32     ctx.fill();                 // 塗りつぶし
33   }
34 }
35
36 void main() {                   // トップ・レベルのmain関数
37   new CanvasTest().run();
38 }
```

APIドキュメントを見ながら緑色で示したコメントを追っていけば、このHTML5ベースのプログラムの書き方が理解されよう。HTML5のJavaScriptの経験者であれば更に理解が早かろう。

パッケージ・マネージャとストリームの詳細に関しては、この後の「パッケージ・マネージャ」の章と「イベント処理」の章を読んで頂く必要がある。

15.3節　　Dartium（Dart VM実装Chromium）の経過

2012年2月16日にGoogleのDartソフトウエア技術者たちがそのブログ(The Chromium Blog)にDart VM実装Chrome（Dartiumというニックネームが付いている）のテスト・バージョンが使えるようになったと発表した。現在はMac OS X及びLinux用であるが、Windows用も間もなく出す予定だと述べている。なおChromiumはChromeブラウザが持っている殆どのソフトウエア要素の為のオープン・ソフトの組織である。

その後2012年3月16日にWindows上のDartiumも限定的に使えるようになった。また4月2日にGoogleはWindows版Dartiumが正式に限定的利用が可能になったと発表した。

2012年9月時点では、Dartiumはどの標準的なプラットホーム上でも使用可能になっている。またDart Editorに同梱されるようになった。

2015年3月25日にDartチームは、ブラウザ上のVM開発を止め、Dart2JSに特化すると発表した。これは現在Dartでビジネス用アプリケーションを開発しているGoogle社内外のチーム（例えばGoogle Adsのチーム）たちからのよりJavaScriptとの統合を高めて欲しいとの意見によるものだという。但しVMそのものはサーバやツール用に開発を継続する。また**Dartiumは将来Chromeに実装されることはないが、ツールとして残される。**これに関する議論はここまたはここを見て頂きたい。

Dartiumのダウンロードと実行は　本資料の「Dart SDKとDartiumのインストール」の節、および https://www.dartlang.org/dartium/を見て頂きたい。

15.4節　　Dart VM（サーバ・アプリケーションの実行）

サーバ・サイドのアプリケーションを実行するにはDart VMを直接走らせる必要がある。なおこの場合はサーバがブラウザに直接アクセスすることはないので、dart:htmlのライブラリは使用できない。代わりにdart:ioライブラリを使ってネットワークにアクセスする。

2012年4月以降はDart EditorにはDart VMとDartiumが同梱されていた。Dart Editorはもはやサポートされていないが、Dart VMとDartiumはダウンロードのサイトからダウンロードできる。この場合は：

- Dart VMは\dart-sdk\binのディレクトリにdart.exeとして存在する。
- Dartiumは\chromiumのディレクトリにchrome.exeとして存在する。

Dart SDKとDartiumのダウンロードと実行は　本資料の「Dart SDKとDartiumのインストール」の節を見て頂きたい。

なお、**dart-sdkとDartiumはWindows XPでは動作しない**ので注意されたい。

DOSレベルでの実行

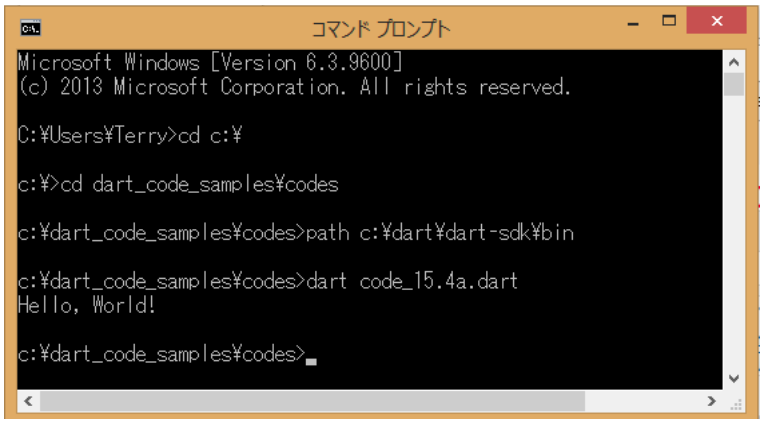
WindowsでDart VMを使ってプログラムを実行させるにはコマンド・プロンプトが使用できる。Dartチームはそのようなアプリケーションを「コマンド行アプリケーション(Command-line application)」とも呼んでいる。以下はコマンド・プロンプトからあるアプリケーションを実行させた例である（パスやディレクトリは自分の環境に合わせる）：

```
C:\dart_code_samples>cd codes

C:\dart_code_samples\codes>path C:\dart\dart-sdk\bin

C:\dart_code_samples\codes>dart code_15.4a.dart
Hello, World!

C:\dart_code_samples\codes>
```

- 最初に自分が作成したアプリケーション（ここではcode_15.4a.dart）が入っているディレクトリに移る
- 次にDart VMがあるディレクトリへのパスを指定する。これにより直接dartコマンドが使えるようになる（環

境変数で設定すれば、コマンド・プロンプトを開いたたびにこのステップを踏まなくてもよくなる）
- 次にdart code_15.4a.dartという具合にcode_15.4a.dartを実行させる
- Dart VMがprint文を実行した結果のHello Worldが出力されている

下図は実際のコマンド プロンプト画面である：



コマンド プロンプト画面

最も簡単なコードの実行

実験に使ったHelloWorld.dartは次のような極めて簡単なコードである：

code_15.4a.dart

```
void main() {
  print("Hello, World!");
}
```

引数付きでの実行

コマンド行から単純な文字列たちを引数として渡したいときには：

code_15.4b.dart

```
void main(List<String> args) {
  print(args);
}
```

のようにListとして受け取る。

以下はコマンド行として実行した例である。引数として渡した6つの文字列がListとして出力されている：

```
C:\dart_code_samples>cd codes

C:\dart_code_samples\codes>path c:\dart_editor\dart-sdk\bin

C:\dart_code_samples\codes>dart code_15.4b.dart 1 2 3 4 5 end
[1, 2, 3, 4, 5, end]

C:\dart_code_samples\codes>
```

引数を構文解析してmainに渡したいときはargsというパッケージ・ライブラリを使用する。このライブラリは生のコマンド行引数たちをオプションと値のセットとして構文解析するもので、GNU及びPOSIXスタイルのオプションに対応している。使い方はstackoverflowなどを参考にすると良い。

stdin、stdout、およびstderr

他の言語たち同様、Dartには標準入出力及びエラーのストリームがある。これらの標準I/Oストリームはdart:ioライブラリのトップ・レベルで定義されている:

ストリーム	記述
stdout	標準出力
stderr	標準エラー
stdin	標準入力

stdout:

例えばもし-nフラグがセットされておればstdoutに行番号とその行をファイルに書き込むプログラムの一部である:

```
if (showLineNumbers) {
 stdout.write('${lineNumber++} ');
}

stdout.writeln(line);
```

write()とwriteln()メソッドは任意の型のオブジェクトを受け付け、それを文字列に変換しプリントする。writeln()は改行コードもプリントする。この例では行番号にwrite()メソッドが使われてるので、同じ行に行番号とテキストが表示されることになる。

writeAll()メソッドを使うとオブジェクトたちのリストをプリントでき、またaddStream()を使うとあるストリームからの総ての要素たちを非同期でプリントできる。

stdoutはprint()関数よりも多くに機能を有している。例えば、stdoutを使ってあるストリームの中身を表示できる。但しJavaScriptに変換され実行されるプロブラムの場合はstdoutではなくてprint()を使わねばならない。

stderr:

コンソールにエラー・メッセージを書き込む場合にはstderrを使用する。標準エラーのストリームは stdoutと同じメソッドたちを有しており、stdoutと同じようにそれらのメソッドを使用すれば良い。stdoutとstderrともにコンソールにプリントするが、これらの出力は分離されており、そのコマンド行であるいはプログラム的に異なった先にリダイレクトまたはパイプできる。

次のコードはユーザがファイルではなくてディレクトリを指定したとき、または指定したファイルが見つからないときにエラー・メッセージをプリントする:

```
if (isDir) {
 stderr.writeln('error: $path is a directory');
} else {
 stderr.writeln('error: $path not found');
}
```

stdin:

標準入力ストリームは一般的にはキーボードから同期でデータを読み込むが、非同期読み込むことも可能であるし、別のプログラムの標準出力からパイプで入力することもできる。

以下のコードはstdinから一行を読み込む:

```
import 'dart:io';

void main() {
 stdout.writeln('Type something');
 String input = stdin.readLineSync();
 stdout.writeln('You typed: $input');
}
```

readLineSync()は標準入力ストリームから同期でテキストを読み込むので、この例ではユーザがテキストをタイプしreturnを押すまでは実行がブロックされる。

以下の行はユーザがコマンド行でファイル名を指定しなかった場合、stdin経由でpipe()メソッドを使って同期でそれを読み込む際に使われている:

```
return stdin.pipe(stdout);
```

この場合はユーザがテキスト行をタイプ・インすればこのプログラムはそれらの行をstdoutにコピーする。入力を終了したいときはユーザは<ctl-d>をタイプしてこれを知らせる。

以下はその実行例である:

```
$ dart dcat.dart
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
...
```

IntelliJ IDEAからのサーバ・アプリケーションの実行

以前dart-sdkにはサンプルとして時刻サーバのライブラリがあった。現在は存在しないので、これに相当するものをGithubからダウンロードできるようにした。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IDEから\dart_code_samples-master\apps\TimeServerのフォルダを開くと良い。

このサーバをDart VMで実行させよう。このプログラムは次のようになっており、これを参考にすれば簡単にHTTPサーバが開発できよう:

time_server.dart

```
001 // Copyright (c) 2012, the Dart project authors.  Please see the AUTHORS file
002 // for details. All rights reserved. Use of this source code is governed by a
003 // BSD-style license that can be found in the LICENSE file.
004 // December 2013, Modified by Terry.
005 // Access this server as 'http://localhost:8080/time'.
006
007 import "dart:io";
008 import "dart:convert";
009
010 const HOST = "localhost";
```

```
011 const PORT = 8080;
012 final REQUEST_PATH = "/time";
013
014 const LOG_REQUESTS = true;
015
016 void main() {
017   HttpServer.bind(HOST, PORT)
018   .then((HttpServer server) {
019     server.listen(
020         (HttpRequest request) {
021           if (LOG_REQUESTS) {
022             print("Request: ${request.method} ${request.uri} "
023             "from ${request.connectionInfo.remoteAddress}");
024           }
025           if (request.uri.path == REQUEST_PATH) {
026             service(request);
027           }
028           else request.response.close();
029         });
030     print("${new DateTime.now()} : Serving $REQUEST_PATH on http://${HOST}:${PORT}.\n");
031   });
032 }
033
034 // create and send response for the request here.
035 // if you are using Future or Stream objects inside of the try block,
036 // you have to catch and handle their error in the callback.
037 void service(HttpRequest request) {
038   try {
039 //  throw new Exception('exception raised');  // uncomment this line to test exception handling
040     String htmlResponse = createHtmlResponse();
041     sendResponse(request, htmlResponse);
042   }
043   catch (e, st){ // catch any error and exception
044     sendErrorResponse(request, e, st);
045   }
046 }
047
048 void sendResponse(HttpRequest request, String htmlResponse){
049   String htmlResponse = createHtmlResponse();
050   List<int> encodedHtmlResponse = UTF8.encode(htmlResponse);
051   request.response.headers
052     ..set(HttpHeaders.CONTENT_TYPE, "text/html; charset=UTF-8")
053     ..contentLength = encodedHtmlResponse.length;
054   request.response
055     ..add(encodedHtmlResponse)
056     ..close();
057 }
058
059 void sendErrorResponse(HttpRequest request, e, st){
060   request.response
061     ..statusCode = HttpStatus.INTERNAL_SERVER_ERROR
062     ..headers.set('Content-Type', 'text/html; charset=UTF-8')
063     ..write('<pre>internal server error:\n$e\n$st</pre>')
064     ..close();
065 }
066
067 String createHtmlResponse() {
068   return
069 '''
070 <html>
071   <style>
072     body { background-color: teal; }
073     p { background-color: white; border-radius: 8px; border:solid 1px #555;
074         text-align: center; padding: 0.5em;
075         font-family: "Lucida Grande", Tahoma; font-size: 18px; color: #555; }
076   </style>
077   <body>
078     <br/><br/>
079     <p>Current time: ${new DateTime.now()}</p>
080   </body>
081 </html>
082 ''';
083 }
084
```

- 037行目のserviceメソッドはサーバがリスン状態（019行目で設定）にあるときに、クライアントからのHTTP要求が到来したときにその要求を処理する関数になる。これはJavaにおけるHTTPServletのserviceに相当するもので、要求オブジェクトが引数になる（Servletと違って応答は要求の属性である）。Servletと違うのは単一スレッドであることでである。
- 067行目のcreateHtmlResponseはクライアントに返すHTTP応答を生成しており、040行及び041行目でHTTP応答のボディ部の為に出力ストリームにセットするとともにクライアント側に送信している。応答をクローズすると未送信の応答データのすべてが送信される。

HTTPサーバでのエラー処理は、サーブレットに相当するrequestReceivedHandlerのなかで処理するのが一般的であろう。この例ではtryブロックの中で発生した例外とエラーを捕捉し、それをエラー・ページとしてクライアントに返している。039行目のコメントを外してその動作を確認するとよい。但しこのブロックの中でFutureやStreamを使っている場合は、そこで発生するエラーはonErrorやcatchErrorで捕捉し、処理する必要がある。その詳細は「イベント処理」の章を読んで頂きたい。

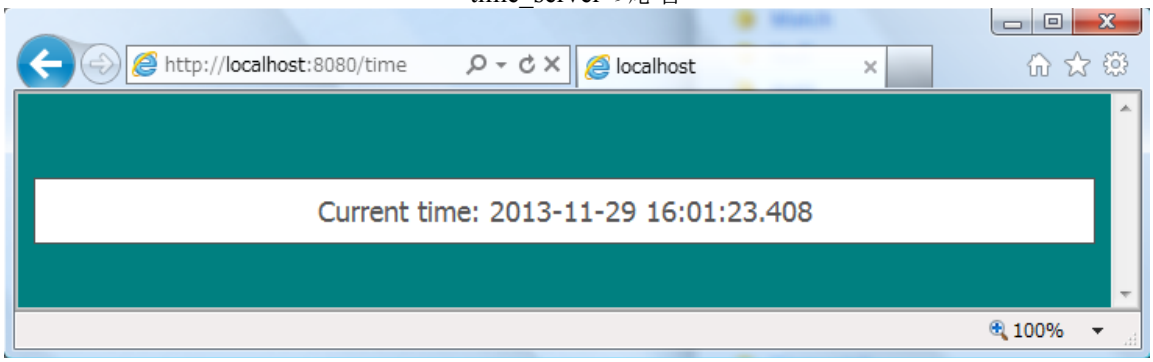
HTTPサーバの詳細については「HTTPサーバ」の章を読んで頂きたい。

このサーバはDOSレベルでは以下のように実行させる：

```
C:>cd C:\dart_code_samples\apps\TimeServer
（注意：これは例で、自分がtime_server.dartを置いたパスに移る）

C:\dart_code_samples\apps\TimeServer>path c:\dart\dart-sdk\bin

C:\dart_code_samples\apps\TimeServer>dart time_server.dart
Serving the current time on http://127.0.0.1:8080.
Request: GET /time from InternetAddress('', IP_V6)
```

- このサーバが起動すると、Serving /time on http://127.0.0.1:8080.とメッセージを出力し、サービスを開始したことを知らせる。
- 次に自分のブラウザからhttp://localhost:8080/timeとこのサーバを呼び出すと、下図のように応答することが確認できよう。ここに127.0.0.1はネットワークとの接続端で折り返す為のIPv4ループバック・アドレスで、localhostとして呼び出すことができる：

time_serverの応答

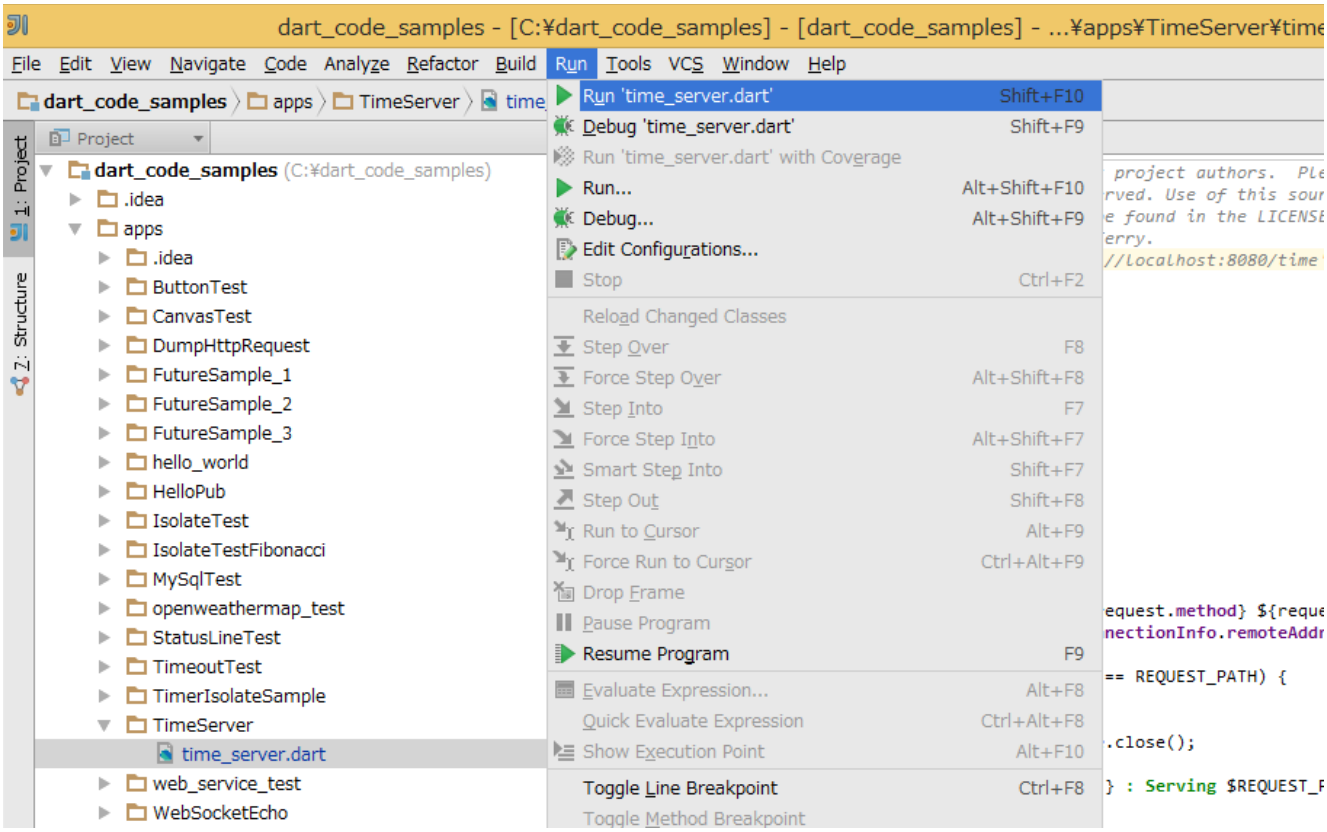


サーバ側のコンソールにはRequest: GET /timeとGET要求が到来したと報告している。

IntelliJ IDEAでは次のようにこのコードを実行させる：

1. File → Openでこのアプリケーションが入っているフォルダを指定する
2. TimeServer\time_server.dartを開く

3. Runで次のようにこのアプリケーションの実行を指定する。



time_serverの起動

4. そうするとVMがこのコードを実行し、コンソールには次のようなメッセージが出力される。

time_serverのコンソール出力

5. 実行を停止するにはこの画面にある赤い四角のアイコンをクリックする。

プログラムからの終了

VMによる実行をプログラム上で終了させるには、dart:ioライブラリのトップ・レベル関数のexitを使用する:

code_15.4c.dart

```
import 'dart:io';

main() {
  exit(0); // or non-zero for some error code
}
```

エラーで終了させるときはゼロ以外の値を引数（ステータス・コード）にしてこの関数を呼ぶ。この場合は非同期処理で待っているプロセスも強制終了させる。従ってexitを使うとデータが失われる可能性があるので注意しなければならない。

Linux及びOS Xでは通常の終了時の終了ステータス・コードは常に0..255の範囲になる。この範囲外のコードを指定したときは下ビットのみが符号なし整数として受け付けられる。例えば-1という整数を指定するとそれは255として扱われる。

Windowsでは32ビットの任意の整数を指定できる。しかし幾つかの値はクラッシュのようなシステム・エラー用に予約されている。

これらに加えDartのVMはコンパイル時エラーに254を、実行時エラー（処理不能の例外）の報告用に255が使われている。

従ってDartプログラムの結果を環境に渡すときには0..127の範囲の整数のみを使うべきである。

第16章 パッケージ・マネージャ(Pub)

Dartには開発を支援する為に以下のライブラリが用意されている:

- 共有ライブラリ・ベースの開発のためのパッケージ・マネージャ(pub)
- モジュール単位でのテストのための単体テスト(Unit Test)

この章ではパッケージ・マネージャ(pub)に関して解説する。

Dartコードの中には通常その最初のところに例えば次のようなimport文が置かれる:

```
import 'dart:io';
import 'package:args/args.dart';
import 'package:shelf/shelf.dart';
import 'package:shelf/shelf_io.dart' as io;
import 'package:shelf_static/shelf_static.dart';
```

これらの中にはDartのAPI参照に含まれていないdart:ではなくてpackage:というサフィックスが付いているライブラリが指定されている。これはpubライブラリ・マネージャを使ってインポートされることを想定している。このパッケージはhttps://pub.dartlang.org/というリポジトリの中に含まれている。このリポジトリは現在成長し続けており、Dartのユーザには貴重な財産になっている。https://pub.dartlang.org/にはDartのチームが開発中のライブラリも多く含まれている。

pubライブラリ・マネージャはアプリケーションの中に含まれるpubspec.yamlという特別なファイルを見て、そこに記載されているそのアプリケーションに必要なpubライブラリ(依存物:dependency)をhttps://pub.dartlang.org/から取り込み、packagesというフォルダに収容する。

PubはDartの為のパッケージ・マネージャのシステムである。これはおなじレポジトリ・システムのGithubの影響を強く受けており、互換性がある。これによりプログラマたちは既存のDartコードの再利用ができ、他の人たちとの共有と再利用の為にDartのアプリケーションたちとライブラリたちをバンドル出来るようになる。Pubはバージョン管理と依存物(dependancy)管理をするので、自分のアプリケーションが確実に他のマシン上でも自分のマシン上で走ると同じように走らせることができる。Pubはコマンド行ベースで使うことも出来るし、Dart Editorの中で使うこともできる。PubはSDKの要素であるので、Dart EditorあるいはSDKが更新されればpubも新しい版に更新される。詳細はpub.dartlang.orgを見て頂きたい。

＊＊＊

本章の次節の前半分及び16.3節以降は推奨IDEのひとつのIntelliJを用いたPubの使用法を解説書に基づいて解説する。コマンド行を用いた使用法は解説書を見ていただきたい。

＊＊＊

16.1節　pubの概要

Pubの使い方の手順をまず概説する。

1. Pubを使用するにはまずpubspec.yamlというヤムル・ファイルを作成し（未だ存在していない場合は）、使用するパッケージたちをそこに依存物(dependency)としてリストアップする。例えばあるアプリケーション(my_app)のなかでweb_uiパッケージを使用するときは、これをpubspec.yamlという名前のファイルのトップ・レベルのファイルにそれを記述する:

```
name: my_app
dependencies:
  web_ui: any
```

2. コマンド行（Windowsの場合はコマンド・プロンプト）から、あるいはIntelliJのpub getでpubインストールを実行する。

3. 次のようにそのパッケージからひとつまたはそれ以上のライブラリを次のようにインポートする:

```
import 'package:web_ui/web_ui.dart';
```

pubのインストールと設定

PubはDart SDKのなかにあり、これはダウンロードしたDart Editorの中に含まれているのでDart Editorを使っているユーザは特にPubをインストールする必要はない。pubはDart Editorを介して使うことができるし、コマンド行アプリケーション（これはDart SDKのbinディレクトリの中にある。Windowsの場合は\dart_editor\dart-sdk\bin\pub.batというバッチ・ファイル）を介しても使うことができる。

コマンド行でpubやその他のツールを使用する場合は、自分のシステム・パスにこのSDKのbinディレクトリを追加する必要があろう。例えばMacとLinuxの場合は:

```
export PATH=$PATH:<path to sdk>/bin
```

ここで<path to sdk>はこのSDKのメインのディレクトリへの絶対パスである。例えばDart Editorを/home/me/dartにインストールしている場合は、次のようにパスをPATHに追加する:

```
/home/me/dart/dart-sdk/bin
```

Windowsの場合は、コントロール・パネルでシステム環境変数のPATH変数に追加出来る。あるいはコマンド・プロンプトからたとえば次のようにpath設定する（c:\dartは自分のインストール・ディレクトリにする）:

```
>set PATH=%PATH%;c:\dart\dart-sdk\bin
```

最後に自分のディレクトリからdart --versionを実行して、binのなかのコマンドが使えることを確認する。

パッケージの作成

パッケージはDartのコード及びリソース、テスト、及びドキュメントといったファイルたちを含んだディレクトリである。

Dart Editor上ではプロジェクト(project)とも表現される。フレームワークとか再使用可能なライブラリたちは明白なパッケージであるが、アプリケーションもまたパッケージである。自分が作ったアプリケーションがpubパッケージたちを必要とするときは、そのアプリケーションもまたパッケージである必要がある。

pubではなんでもパッケージになるが、実際の使用上少し異なった2つのパッケージの種類が存在する。ライブラリ・パッケージ(library package)は他のパッケージによって再利用されることを意図したパッケージである。これは通常他のパッケージがインポートするコードを含んでおり、ユーザが取得できるどこかにホストされることになる。アプリケーション・パッケージはパッケージを消費するだけでそれ自身は再利用されない。言い換えると、ライブラリ・パッケージは依存物(dependencies)として使われるが、アプリケーション・パッケージはそうではない。

殆どの場合これらの2つの相違はなく、我々は単に「パッケージ」という。区別する必要があるときのみ「アプリケーション・パッケージ」または「ライブラリ・パッケージ」と区別する。

自分の作ったアプリケーションをアプリケーション・パッケージとするには、単にそれにpubspecファイルを付加すれば良い。このファイルはYAML（ヤムル）言語で書かれており、その名前はpubspec.yamlである。一番簡単なpubspecはそのパッケージの名前だけが含まれているものである。このpubspecファイル自分のアプリケーションのルート・ディレクトリの中にpubspec.yamlとして保管すれば良い。

以下は最もシンプルなpubspec.yamlである:

```
name: my_app
```

これでmy_appがpubパッケージとなる。

依存物(dependency)の付加

pubの主たる仕事のひとつは依存物管理である。依存物というのは読者のパッケージが依存している別のパッケージのことである。例えば読者のアプリケーションが“transmogrify（変形）”という名前の変換ライブラリを使っているとすると、読者のアプリケーション・パッケージはtransmogrifyパッケージに依存することになる。

読者はpubspecファイルのなかに自分のパッケージ名の直後に自分のパッケージの依存物たちを指定する。例えば:

```
name: my_app
dependencies:
  transmogrify:
```

ここでは、フィクションではあるがtransmogrifyパッケージに依存することを宣言している。

依存物たちをインストールする

依存物を宣言したらpubにたいしてそれをインストールするよう指示する。Dart Editorを使っているときは“Tools”メニューから“Pub Install”を選択する。Dart Editorが自動的にインストールするよう設定することも可能である。Tools → Preferences → Editor → PubでAutomatically run Pubをチェックする。

コマンド行を使いたいときは以下のようにする:

```
$ cd path/to/your_app
$ pub install
```

注意:現時点ではこのコマンドはpubspec.yamlが含まれているディレクトリから実行しなければならない。将来はそのパッケージのどのディレクトリからでも実行可能になろう。

これを実行すると、pubはpubspec.yamlとおなじディレクトリの中にパッケージ・ディレクトリを作成する。そこにpubは読者のパッケージが依存する各パッケージをダウンロードしてインストールする(これらは皆さんの即座の依存物たち(immediate dependencies)と呼ばれる)。pubはまたこれらのパッケージを調べて、それらが依存している総てをものを再帰的にインストールする(これらは他動的依存物たち(transitive dependencies)という)。

これが終了すると、読者のプログラムが実行に必要な各パッケージたちが含まれているパッケージたちのディレクトリ (packages) ができる。

依存物からコードをインポートする

これで依存物を結び付けたので、その依存物のコードが使えるようにする必要がある。別のパッケージにあるライブラリにアクセスするには、package:スキームを使って次のようにインポートする:

```
import 'package:transmogrify/transmogrify.dart';
```

これはtransmogrifyパッケージの中を調べてそのトップ・レベルにあるtransmogrify.dartを探す。殆どのパッケージはそのパッケージの名前と同じ名前の単一のエントリ点を定義している。そのパッケージのドキュメンテーションを調べてそのパッケージが自分がインポートしたいと思うものとなにか別のものを含んでいるか調べる。

これは生成されたパッケージのディレクトリの内部を調べれば良い。エラーが発生した場合はそのディレクトリが古いものになっている可能性がある。自分のpubspecを変更するときは何時もpubインストールを走らせることでこれを解決する。

自分自身のパッケージの内部のライブラリをインポートするには以下のようなスタイルが使える。例えば自分のパッケージの構成が次のようだったとする:

```
transmogrify/
  lib/
    transmogrify.dart
    parser.dart
  test/
    parser/
      parser_test.dart
```

parser_testファイルは次のようにparser.dartをインポートできよう:

```
import '../../lib/parser.dart';
```

しかしこれはかなり面倒な相対パス指定である。もしparser_test.dartがあるディレクトリから上下のディレクトリに

移ってしまったら、そのパスは不正なパスになってしまいそのたびにparser_test.dartのコード（import文）を修正しなければならなくなる。そうならないようにするには次のように指定する：

```
import 'package:transmogrify/parser.dart';
```

こうすれば、このimport文が含まれるファイルが何処にあろうとも常にparser.dartを得ることができる。

依存物の更新 (Updating a dependency)

読者のパッケージの為に新しい依存物を最初にインストールしたときは、pubはその最新のバージョンをダウンロードし、それは読者がインストールした他の依存物たちと互換性があるものである。pubは次に自分のパッケージをロックし、ロックファイルを生成して以後常にそのバージョンを使うようにする。pubは読者のパッケージが使う各依存物（即座の依存物たちと他動的依存物たち）のバージョンをリストアップする。

この読者のパッケージがアプリケーション・パッケージの場合は、読者はソースコード・コントロールでこのロックファイルをチェックするだろう。同じように読者のアプリケーションを利用する誰もがそれらのパッケージの総てが必ず同じ版のものである必要がある。これにより読者は自分のアプリケーションを実用の為に本稼働させるときに同じバージョンのものが常に使われているようにできる。

自分の依存物たちを最新のバージョンに更新しても良い場合は以下のようにする：

```
$ pub update
```

これはpubに対して読者のパッケージの為にロックファイルを最新の取得可能な版を使ってロックファイルを作り直すよう指示する。もし特定の依存物だけを更新したいときは以下のように指定する：

```
$ pub update transmogrify
```

これはtransmogrifyを最新の版に更新するが、それ以外の依存物たちは変更しない。

パッケージの公開 (Publishing a package)

pubは版に他の人たちのパッケージを使う為だけのものではない。pubを使って自分のパッケージを世界中が共有できるようにすることも可能である。ある有用なコードを作成し、誰もがそれを使えるようにしたいと思ったら、次を実行するだけで良い：

```
$ pub publish
```

pubは読者のパッケージがpubspecの書式とパッケージの組み立て規約に従っているかどうかをチェックし、そのパッケージをpub.dartlang.orgにアップロードする。そうするとpubのユーザの誰もがそれをダウンロードしたりpubspecsのなかでそれに依存したり出来るようになる。例えば、読者がtransmogrifyという名前のパッケージの1.0.0版を公開したときは、他のユーザは次のように書くことができる：

```
dependencies:
```

```
transmogrify: ">= 1.0.0 < 2.0.0"
```

この公開は永久的であることに注意されたい。読者が自分の優れたパッケージを公開したら即座にユーザたちはそれに依存できるようになる。一旦ユーザたちがそれを始めたら、そのパッケージを削除するとユーザたちのパッケージが走らなくなってしまう。その事態を回避する為に、pubはパッケージの削除を極力しないよう求めている。読者は何時も最新のバージョンをアップロードできるが、更新の準備が出来ていないユーザたちの為に古いバージョンはいつでも利用可能である。

筆者が公開したライブラリ mime_type

参考までに筆者がアップロードしたパッケージを紹介する。これはHTTPサーバがあるファイルを応答としてクライアントに送信する際にそのヘッダに付加するMIMEタイプの文字列を取得するものである。このライブラリは以下のアドレスに存在するので見て頂きたい：

```
https://pub.dartlang.org/packages/mime_type
```

読者がウェブ・サーバ・アプリケーションをDart言語で開発する際には有用である。

このパッケージの内容を知りたい場合は、Versionsのタグを開いて、最新のアーカイブのDownloadのアイコンをクリックすればこのパッケージ全体を圧縮形式で取得できる。

MIMEライブラリ

16.2節　　Pubを試してみる

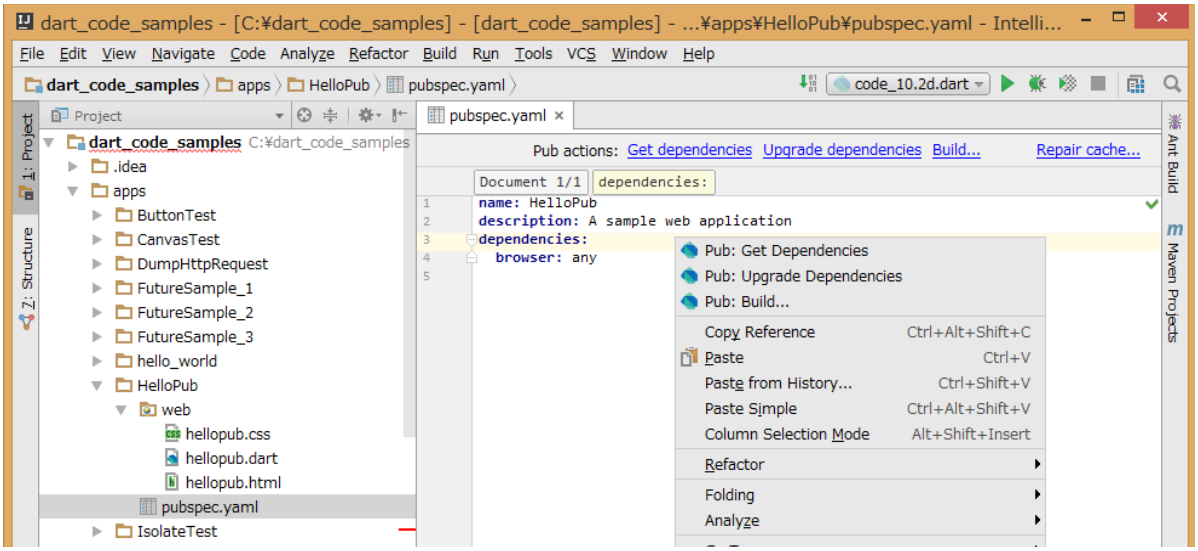
Pubの概要を知ったうえで、簡単なアプリケーションでそれを試してみることにする。これは簡単なアプリケーションを開発するテンプレートにもなる。読者が本格的なアプリケーション（IDE上ではあるひとつのProject）を開発する場合は、「Dartのサンプルを試してみる」の節で示されているDartのチームが用意したサンプルをテンプレートとして使用することが推奨される。

HelloPub

一番単純なのは筆者が用意したHelloPubというパッケージを試してみることであろう。このアプリケーションはGithubからダウンロードすることもできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJで\dart_code_samples-master\apps\HelloPubのフォルダを開くと良い。

HelloPubパッケージ

このアプリケーションをIntelliJ上で展開すると、下図のように4つのファイルで構成されていることが判る：



HelloPubアプリケーションの初期構成

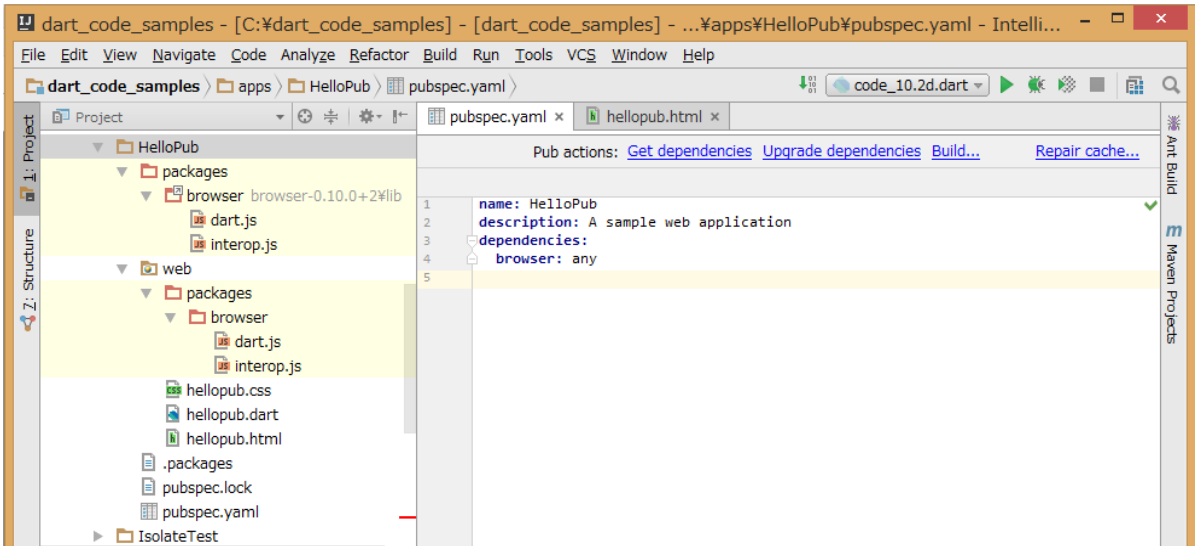
	パス	名前	内容
1	HelloPub\web\	hellopub.css	この場合はウェブ・アプリケーションであるので、そのページを構成する総ての内部リソースがwebというフォルダに収容される。
2	HelloPub\web\	hellopub.dart	
3	HelloPub\web\	hellopub.html	
4	HelloPub\	pubspec.yaml	使用する外部ライブラリを指定する

更に前節で説明したとおり、このパッケージの構成を指定するpubspec.yamlというファイルが必要である。

pubspec.yamlには上図のようにbrowserというライブラリが依存物として記されている。dart.jsブートストラップはこのページを開いたブラウザがDartiumかどうかを判定し、それに応じたファイルを該ブラウザに送信する。

Pub getの実行

このままではhellopub.htmlを実行させてもpackages/browser/dart.jsが見つからないのでエラーとなる。前図にはpub actionsが表示されているので、必要なpubのコマンドを実行させる必要がある。このpubアクションの提示はIntelliJのパッケージ・ビュー上のpubspec.yamlファイルを選択し右クリックして得ることもできる。殆どの場合はget dependenciesをクリックする（即ちpub getを実行させる）ことで済む。pub get実行後のこのアプリケーションは次のような構成になる：

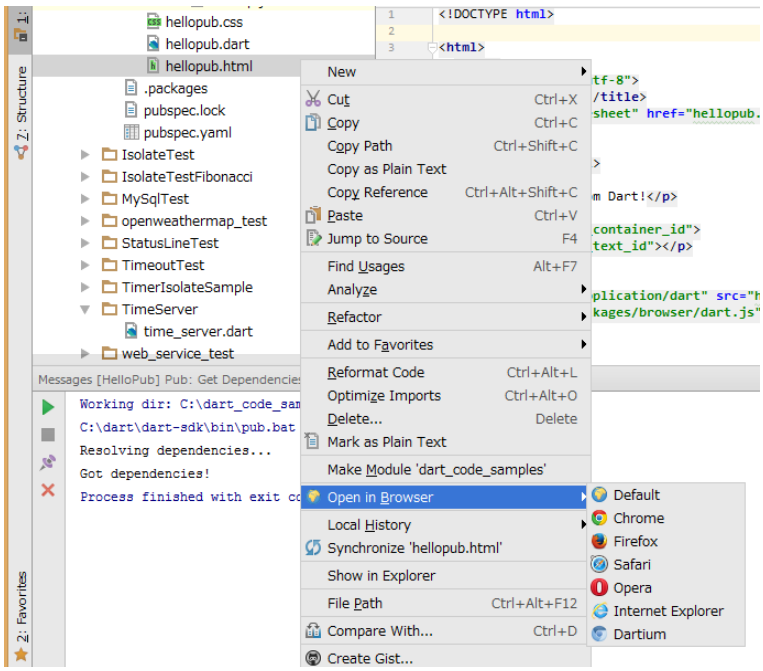


pub get実行後のHelloPubアプリケーションの構成

packagesというフォルダがHelloPub及びHelloPub/webの2つのパス上に追加される。このフォルダにはpubspec.yamlで指定されている外部ライブラリが置かれている。またpubが必要とするpubspec.lock及び.packagesという2つのファイルも配置される。

hellopub.htmlをDartiumで実行する

パッケージ・ビュー上のhellopub.htmlを左クリックしてOpen in Browserを選択、次にDartiumを選択・実行させる。DartiumにはClick me!というメッセージが表示される。これをクリックする度に文字順が反転する。



hellopub.htmlをブラウザで開く

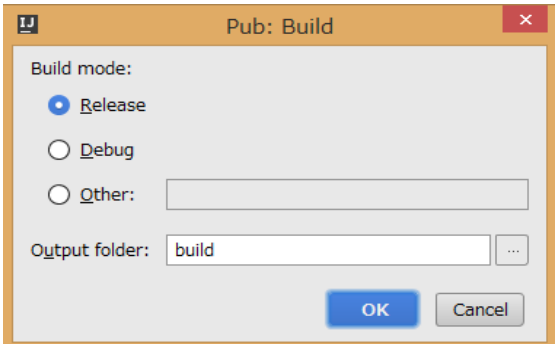
次のようにDartiumから直接このファイルをアクセスしても良い:

```
file:///C:/dart_code_samples/apps/HelloPub/web/hellopub.html
```

HelloPubをデフォルトのブラウザで実行させる

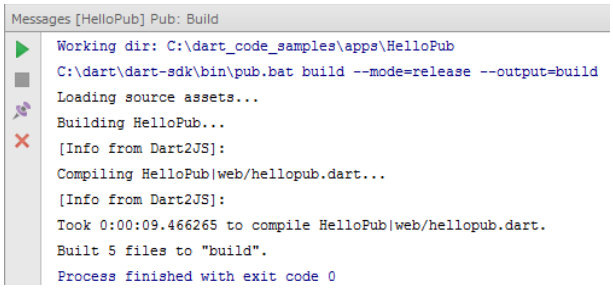
パッケージ・ビュー上のhellopub.htmlを選択・左クリックしてOpen in Browserを選択、次にDefaultを選択・実行させる。IntelliJはdart2jsを呼んでhellopub.dartをjsファイルにクロス・コンパイルし、次にdart.js経由でデフォルトのブラウザにこれを実行させる。

実際にはしかるべきウェブ・サーバに配備することになる。その場合はIntelliJ上でpubspec.yamlを選択し、右クリックしてコマンドを実行する。



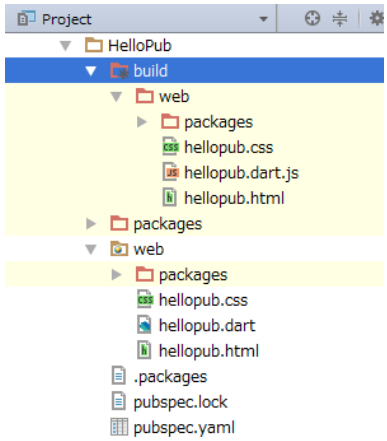
Pub buildの実行

OKをクリックすれば次のように変換で行われる:



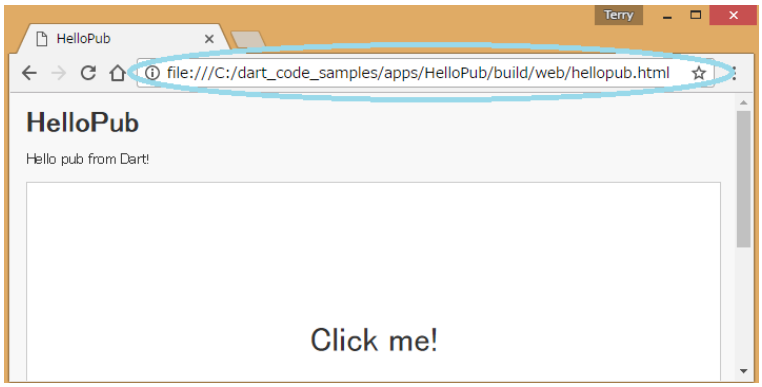
Pub buildの実行過程

このコマンド実行で新たにbuildというフォルダが作成される。このフォルダにはJavaScript変換されたhellopub.dart.js以外にライブラリを含む必要なファイル一式が入っているので、このフォルダを配備用に使用すればよい。



Pub buildで生成されたbuildフォルダ

実際このフォルダ内のhellopub.htmlファイルのパスをコピーし、通常のブラウザでアクセスすると以下のように表示される:



Pub buildで生成されたbuildフォルダ

Pubの更新

必要に応じ、ユーザはPubの更新版の取得(もしあれば)を行うことができる。その場合はpubspec.yamlファイルを左クリックで選択し、次に右クリックしてPub Updateを選択する。

AngularDartのパッケージ

AngularはHTMLとJavaScriptまたは他のJavaScriptに変換される言語（DartまたはTypescript）を使ってクライアント・アプリケーションを構築するためのオープンソースのJavaScriptフレームワークで、グーグルとユーザ・コミュニティによって開発されており、既に馴染みの読者も多いと思われる。従ってAngular DartはAngularJSのDart版である。AngularJSは動的ビューに重きが置かれたMVCモデルの実装であり、日本語の解説（例えばここまたはここ）も多いので、そちらをまず一読されたい。

Angular DartはAngular TypeScriptとともにAngularコアチームによって現在はAngular2として開発されている。AngularDartはangular2パッケージとして公開されており、他のDartのパッケージ同様Pubツールを使って利用できる。

この節ではAngularDartの解説は行わず、AngularDartのアプリケーション開発のためのライブラリについて略説する。この節を試される読者は、あらかじめ前節の「HelloPub」を学習済みであることを前提としている。

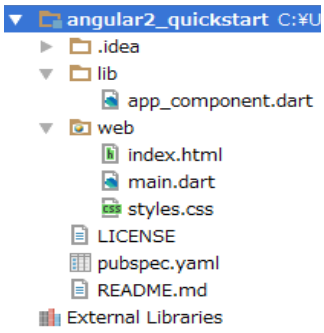
Polymerに関する注意：

- この解説書の以前の版（第34版）まで存在していたPolymerの節はこのAngularの節に置き換えた。これは何もPolymerのプロジェクトが止まったことを意味する訳でなく、単に解説していたDartチームが用意していた簡単なサンプル(stopwatch)が2016年時点で削除されてしまったからである。PolymerとAngularは共存するフレームワークである。
- Web UIは2013年7月時点で新しいPolymerプロジェクトに切り替えられつつある。但しWeb UIで作ったコードは簡単な変更でPolymer用に使用できる。

本節はAngularDartのドキュメントにあるQuickstartの箇所をベースにしている。この解説に使われているベーシックなアプリケーション（下図のようにMy First Angular Appと表示するだけ）はGithubからダウンロードできる。

angular2_quickstartの構成

angular2_quickstartというサンプルはAngularの技本的なコンセプトを初心者向けに解説するために用意されている。このサンプルはあらかじめこの資料の添付としてGithubに含めてあるので、この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJで\dart_code_samples\apps\angular2_quickstartのフォルダを開くと良い。そうするとこのサンプルは次のような構成になっていることが判る：



IDE上でのangular2_quickstartの構成

	パス	名前	内容
1	C:\dart_code_samples\apps\	index.html	

2	angular2_quickstart\web\	styles.css	この場合はウェブ・アプリケーションであるので、そのページを構成する総ての内部リソースがwebというフォルダに収容される。
3		main.dart	
4	C:\dart_code_samples\apps\angular2_quickstart\lib\	app_component.dart	このアプリのコンポネントをライブラリとして収容
5	C:\dart_code_samples\apps\angular2_quickstart\	pubspec.yaml	使用する外部ライブラリを指定する
6		README.md	このプログラムを解説したマークダウン形式のファイル
7		LICENSE	MITライセンスを記述したテキスト・ファイル

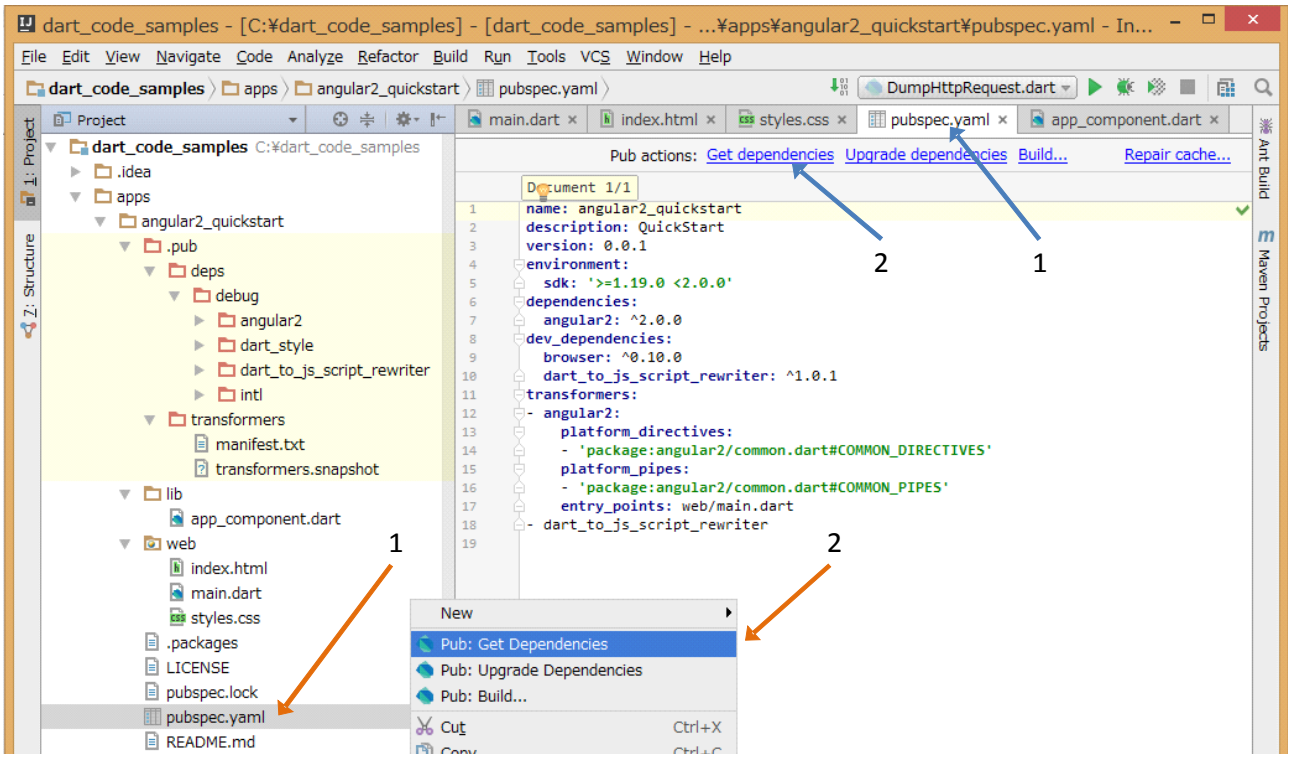
この構成はAngularDartを使ったウェブ・アプリケーションのひな型でもある。各ファイルのより詳細は後で説明する。

Pub getの実行

このままではindex.htmlを実行させても必要なライブラリが見つからないのでエラーとなる。従って必要なpubのコマンドを実行させ、pubspec.yamlファイルで指定されたライブラリを取り込まねばならない。このpubアクションの提示は下図のように:

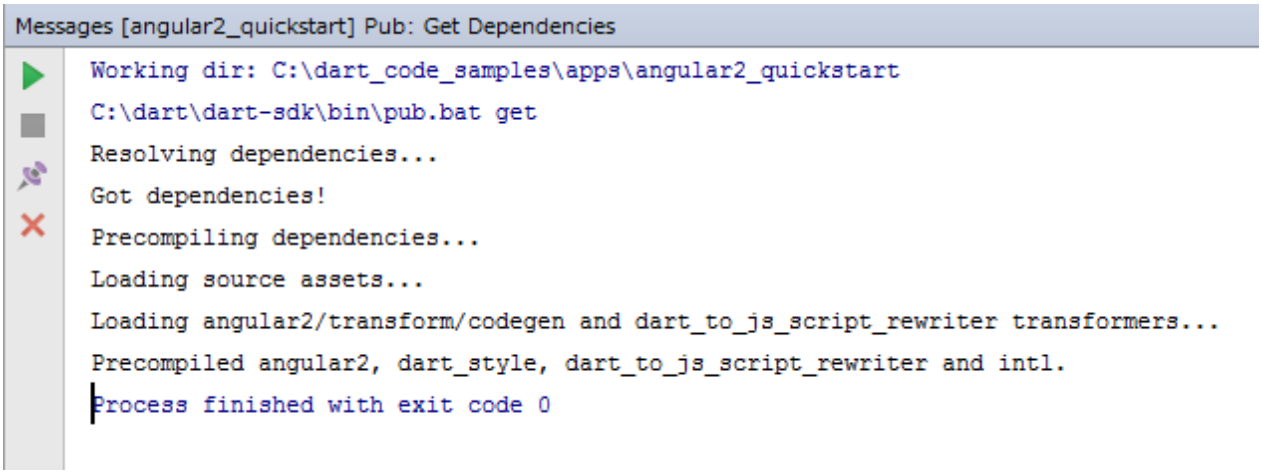
- IntelliJのコード・ビュー上でpubspec.yamlファイルを開くとPub actions:が表示される
- IntelliJのパッケージ・ビュー上のpubspec.yamlファイルを選択し右クリックするとPub:メニューが表示される

で得るこがもできる。殆どの場合はget dependenciesをクリックする(即ちpub getを実行させる)ことで済む。



Pub Getの実行法

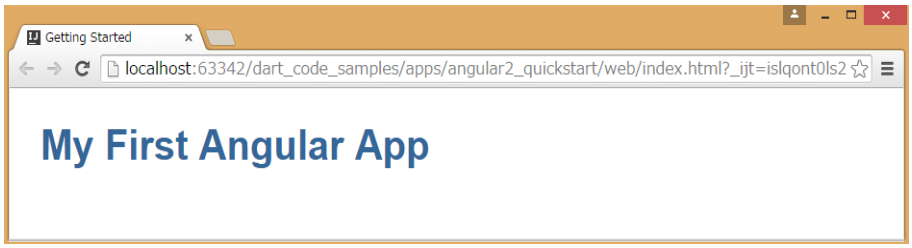
Pub Getを実行すると、コンソールには以下のようなメッセージが表示される:



もしここでエラーがでるときは、SDKバージョンが正しいかを確認する、あるいはPub Repair Cacheを実行してみる。

Index.htmlをブラウザで実行する

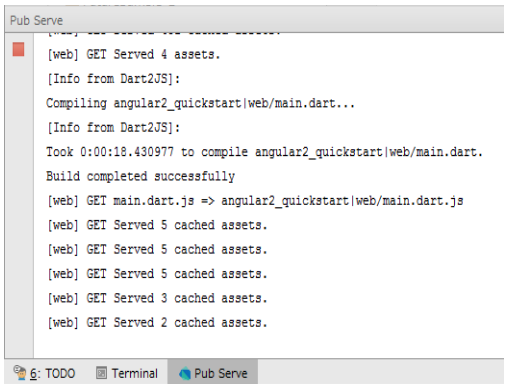
この状態でパッケージ・ビュー上のindex.htmlファイルを選択し、左クリックしてOpen in Browserを選択、次にDartiumを選択・実行させると下図のようにその結果が表示される。



Dartiumで開いたangular2_quickstart

また他のブラウザを選択すると、dart2jsで変換された.jsファイル（キャッシュされIDE上には表示されない）が呼び出れ、同様な表示が得られる。

最初はアクセスに時間がかかるが、これはIDEがHTTPサーバとして機能するためのPub Serveが起動し、必要な処理をするためである。IntelliJではPub Serveの動作は下図のようにそのコンソール出力で知ることができる。



Pub Serveの報告

Pub Serveを終了させるにはこの図の赤いボタンをクリックする。

各ファイルの説明

1. Pubspec.yaml

   このアプリケーションに必要なライブラリ・パッケージの取り込みを記述したもの。YAML（ヤムル）の書式ではその行の最初の'-'は配列であることを理解すればこの記述の理解が早い。pubspec.yamlの記述の詳細は「pubspecの書式」の節を参照のこと。Angular2アプリケーションではangular2とbrowserが依存物（dependencies）として記述されていなければならない。またそのアプリケーションが使う他のパッケージやdart_to_js_script_rewriter といった変換機（transformers）も記述する。

```
name: angular2_quickstart
description: QuickStart
version: 0.0.1
environment:
 sdk: '>=1.19.0 <2.0.0'
dependencies:
 angular2: ^2.0.0
dev_dependencies:
 browser: ^0.10.0
 dart_to_js_script_rewriter: ^1.0.1
transformers:
- angular2:
   platform_directives:
   - 'package:angular2/common.dart#COMMON_DIRECTIVES'
   platform_pipes:
   - 'package:angular2/common.dart#COMMON_PIPES'
   entry_points: web/main.dart
- dart_to_js_script_rewriter
```

Pubspec.yaml

   Angular2のアプリケーションをDartiumではなくて一般のブラウザでアクセスできるようにするには Angular2変換機(Angular transformer)を使用しなければならない。この変換機はDartのコードを分析し、リフレクションを使ったコードをstaticなコードに変換し、Dartのビルド・ツールがより高速でコンパクトなJavaScriptにコンパイルできるようにしている。- angular2:の箇所でAngular2変換機の設定を行っている。entry_points:の記述はこのアプリのエントリ点を指定することで bootstrap.dartがインポートされたときにdart:mirrorsを使わなくするようにしている。dart:mirrorsが使われるとJavaScriptにコンパイルされたときに性能上の問題をおこすからである。

   dart_to_js_script_rewriterは該アプリケーションの性能を改善させるためにHTMLファイルに生成したJavaScriptを直接含めてしまうものである。

2. lib/app_component.dart

```
      import 'package:angular2/core.dart';
@Component(
  selector: 'my-app',
  template: '<h1>My First Angular App</h1>')
```

```
class AppComponent {}
```

app_component.dart

各Angularアプリケーションは少なくともひとつの慣習的に AppComponent と名付けられたルート・コンポネント(root component)を持っていなければならない。このルート・コンポネントはクライアントのユーザ経験を収容するものである。コンポネントはAngularアプリケーションの基本的な構成要素である。コンポネントはそれに結び付けられたテンプレートを介してそのスクリーンのある部分（ビュー）を制御する。

このQuickStartというアプリケーションは極めて単純なコンポネントただ一つのみ持っているものだが、これは今後アプリケーションを開発する際に必ず記述することになる必須の構造になっている：

- 我々が必要とするものを参照するためのひとつ以上のimport文
- Angularに対してどのテンプレートを使用するか、及びどのようにそのコンポネントを生成するかを知らせる為の@Componentアノテーション
- そのテンプレートを介してあるビューの見た目と挙動(appearance and behavior)を記述したcomponentクラス

Import：

Angularアプリケーションはモジュール構成になっている。各アプリケーションは各々がある目的のために特化した多くなファイルで構成される。Angular自身もモジュール構成になっている。各々が我々が自分たちのアプリケーションのために使うことになる幾つかの関連した機能たちで構成されているライブラリ・モジュールたちの集積である。
あるモジュールあるいはライブラリから何かが必要になったときはそれをインポートすることになる。
このサンプルでは Angular 2のコアをインポートしており、これにより我々が記述しているコンポネントのコードが@Componentアノテーションにアクセスできるようになっている。

```
import 'package:angular2/core.dart';
```

@Componentアノテーション：

@Componentはコンポネント・クラスとメタデータを結び付けるできるようにするアノテーションである。このメタデータはAngularに対しこのコンポネントをどのように生成しまた使用するかを伝える。

```
@Component(
  selector: 'my-app',
  template: '<h1>My First Angular App</h1>')
```

@Componentコンストラクタ呼び出しにはセレクタ(selector)とテンプレート(template)の2つの名前付きのパラメタたちを有する。
selectorは該コンポネントを表現するHTML要素のためのシンプルなCSSセレクタを指定している。このコンポネントに対する要素は my-appと名付けてある。 Angularはホストしているの中でmy-appに出会ったときはAppComponentのインスタンスを生成し、表示する。
templateは該コンポネントのテンプレートを指定しており、Angularに対しこのコンポネントのビューをどのように表示するかを強化HTMLの書式で書かれている。このサンプルでは"My First Angular App"を表示する1行のテンプレートとなっている。
より進んだテンプレートではコンポネントの属性とバインドするデータや、それ自身のテンプレートを持っている他のアプリケーションコンポネントを特定するデータが含まれ得る。これらのテンプレートは更に他のコンポネントを特定することもある。

Componentクラス

このファイルの最後はAppComponentという名前の何もしないクラスが置かれている。

```
class AppComponent {}
```

より大きなアプリケーションが書けるようになれば、このクラスを属性たちとアプリケーション・ロジックで拡張すればよい。この QuickStartは何もしなくてよいので空のままになってい

3. main.dart

```
import 'package:angular2/platform/browser.dart';
import 'package:angular2_quickstart/app_component.dart';
void main() {
 bootstrap(AppComponent);
}
```

このアプリケーションの立ち上げに必要な2つのものをインポートしている:

- Angularのブラウザ・ブートストラップ機能
- このアプリケーションのルート・コンポネントである AppComponent

ブートストラップはプラットホーム依存:

angular2/core.dartからではなく angular2/platform/browser.dartからブートストラップ関数をインポートしていることに注意する。このアプリケーションのブートストラップは単一の手段ではないのでブートストラップはコアではない。実際ブラウザ内で走るほとんどのアプリケーションはこのライブラリからブートストラップ関数を呼び出している。
しかしながら他の環境でコンポネントをロードすることは可能である。 モバイル機器上では Apache Cordovaあるいは NativeScriptでロードする。立ち上げを高速化したりSEOを活用したりするのにサーバ上で自分たちのアプリケーションの最初のページを表示したい場合もあろう。これらの場合はそれに対応したブートストラップ関数が必要であり、これらの場合には別のライブラリからインポートすることになる。

どうして main.dartとアプリケーションコンポネントを別々のファイルとするのか？:

main.dartとコンポネントのファイルはともに小さなものである。これは QuickStartという初歩的な教材であるからである。これらは一つのファイルに合体でき、複雑化してくれば分離させてもよい。
ここではAngularのアプリケーションの適切な構成を示すためにこうなっている。合体すると今後困難を生じる可能性がある。我々は別々のブートストラップを使う幾つかの環境で AppComponentを使う場合もある。そのアプリケーション全体を走らせるよりそのコンポネントをテストしたほうがずっと簡単である。そのために多少手間をかけるほうが正しい手法といえる。

4. index.html

index.htmlファイルはそのアプリケーションを収容したウェブ・ページを定義している。 Angularが main.dartのなかでブートストラップ関数を呼びだすと、それが AppComponentメタデータを読みだし、my-appセレクタを見つけ、my-appという名前の要素を特定し、それらのタグ間のビューを表示する。

```
<!DOCTYPE html>
<html>
 <head>
  <title>Getting Started</title>
  <link rel="stylesheet" href="styles.css">
  <script defer src="main.dart" type="application/dart"></script>
```

```
  <script defer src="packages/browser/dart.js"></script>
 </head>
 <body>
  <my-app>Loading...</my-app>
 </body>
</html>
```

5. styles.css

最小限のスタイルは次のようなものである:

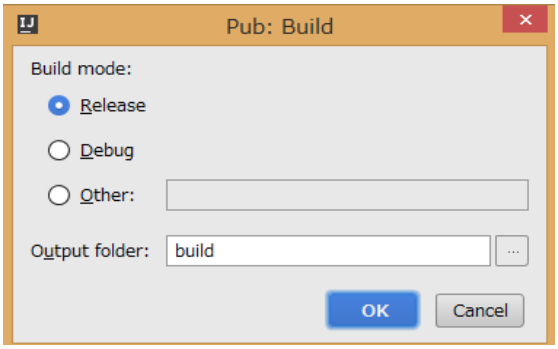
```
<!DOCTYPE html>
<html>
/* Master Styles */
h1 {
 color: #369;
 font-family: Arial, Helvetica, sans-serif;
 font-size: 250%;
}
h2, h3 {
 color: #444;
 font-family: Arial, Helvetica, sans-serif;
 font-weight: lighter;
}
body {
 margin: 2em;
}
```

サンプルにあるファイルはより高度なものになっている。

デプロイ用にJavaScript変換されたアプリケーションを作る

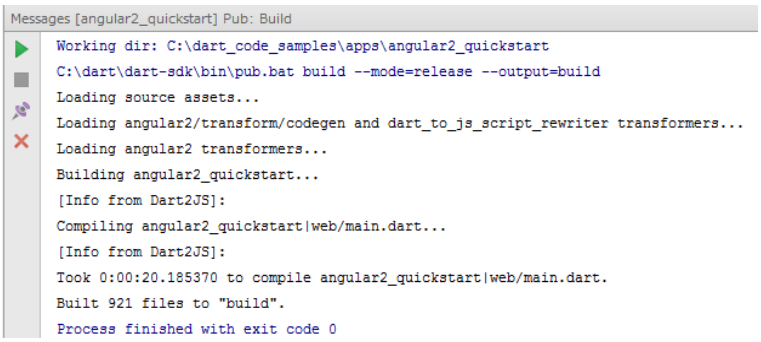
1. Pub build

このアプリケーションをウェブ・サーバ上に配備するにはどのブラウザでも使えるようにJavaScriptファイルを生成する必要がある。それには pub buildコマンドが便利である。



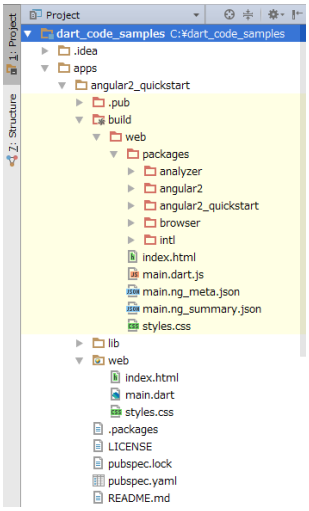
Pub buildの選択

OKをクリックすれば次のように変換で行われる:



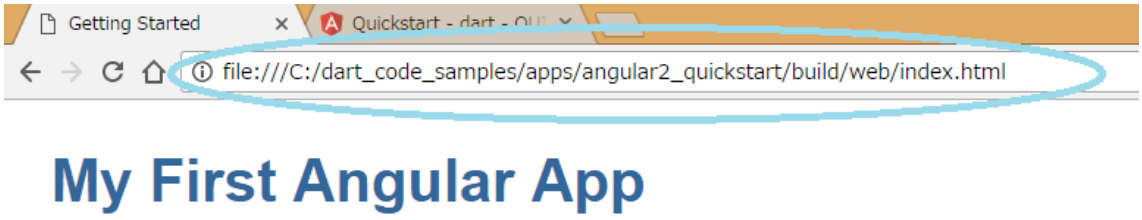
Pub buildの実行

そうすると下図のように新たにBuildというフォルダが作られる。これには必要なファイル一式が含められている。



Pub buildで生成されたBuildフォルダ

このフォルダを配備用に使えばよい。例えばこのフォルダ内のindex.htmlファイルのパスをコピーし、Chromeでこれを読みだすと次のように正しく表示される:



Pub buildで生成されたアプリの直接アクセス

2. Angular transformer

AngularアプリケーションのJavaScriptを生成するためにはAngularの変換機(Angular transformer)を使用すること。これはDartのコードを調べ、reflectionが使われているコードをstaticなコードに変換し、Dartのビルド・ツールがより高速でより小規模なJavaScriptにコンパイルできるようにしている。
entry_pointsという項目はmain()関数を有しているDartファイルを指定している。詳細はAngular

transformerのwikiページを参照のこと。

3. dart_to_js_script_rewriter

   アプリケーションの性能改善のためにはHTMLファイルが生成されたJavaScriptを直接含むようにすれば良い。そのひとつの手段は dart_to_js_script_rewriterを使うことである。この書き換えプログラムはpubspecのなかの dependenciesと transformersの2か所で指定される。

AngularDartをより詳しく知るには

以下の資料が参考になろう:

1. W3CのAngularJSのチュートリアル:
   これはまずAngularJSでAngularの基礎を知るのに良い。その他日本語の資料も多い。
2. AngularDartのチュートリアル
3. AngularDartアプリケーションのアーキテクチャ
4. AngularDartのAPI
5. AngularDartのgithub

DBアクセス (Database Drivers)

HTTPあるいはWebSocketサーバのアプリケーションでは、通常はデータ・ベースが使われる。2013年末時点では幾つかのデータ・ベース・ドライバが現在サード・パーティのソフトウエアとしてPubに登録されている:

名称	内容	Pub	Github
Mongodart	MongoDbドライバ	mongo_dart	https://github.com/vadimtsushko/mongodart/
SqlJocky	Mysqlコネクタ	sqljocky	https://github.com/jamesots/sqljocky
Redis_client	Redisクライアント	redis_client	https://github.com/dartist/redis_client
PostgreSQL	PostgreSQLドライバ	postgresql	https://github.com/xxgreg/postgresql
riak-dart	Riak分散データベース	riak-dart	https://code.google.com/p/riak-dart/
IndexedDB	ブラウザ用IndexedDBライブラリ	dart:indexed_db	https://api.dartlang.org/docs/channels/stable/latest/dart_indexed_db.html

注意:IndexedDBはブラウザのための簡易データ・ベースで、dart:htmlが必要(参考資料)

本節ではDartによるDBアクセスの例として、欧州では広く普及しているMySQLとそのコネクタを説明する。このコネクタの著者のJames Otsによれば、sqljockeyという名前は著名なダーツの選手のJocky Wilsonが由来だという。

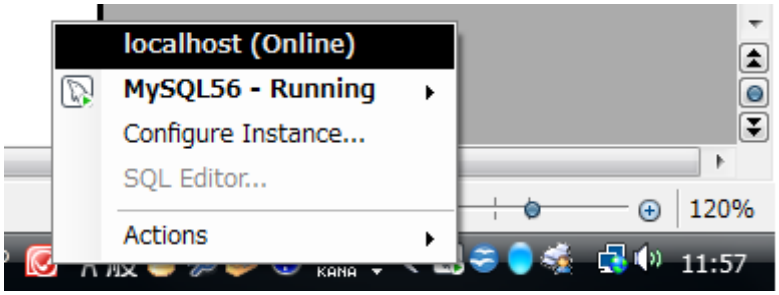
MySQLサーバのインストール

MySQLのインストールに関しては多くの解説が存在するので、そちらを見てインストールして頂きたい。

たとえば:

- http://awoni.net/personal-site/mysql
- http://www.kkaneko.com/rinkou/mysql/mysqlinstall.html あるいは
- http://dev.mysql.com/doc/refman/5.1/ja/installing.html

本章では、MySQL Community ServerのGenerally Availableリリースの Windows (x86, 64-bit), MSI InstallerとMySQL Utilitiesをダウンロードし、インストールしている。インストールが完了したら下図のように、デスクトップ画面右下のタスクバーに新しく作られたMySQL Notifierのアイコンを右クリックして、オンラインで稼働中であることを確認する。



Configure Instance...を右クリックするとMySQL Workbenchが開く。MySQL Workbenchは非常に便利なツールである。

ユーザの登録

最初にこのデータベースにrootユーザとしてアクセスし（root connectionのアイコンを右クリック、Start Command Line Clientを選択して、ルート・ユーザのパスワードを入力する）、mysqlコマンドを実行してテスト用のデータベースを用意する:

```
mysql> create database testDB character set utf8;
```

注意点としては、デフォルトの文字セットとしてUTF8を使用することである。DartではShift_JISなどの日本語文字セットに未だ対応していない。この場合はWindowsのDOSプロンプトで日本語を使うと文字化けが発生するので、DOSプロンプトでは日本語文字を含む2バイト文字は使用してはいけない。

次にユーザを登録する:

```
mysql> grant select on testdb.* to test@'localhost' identified by 'test';
```

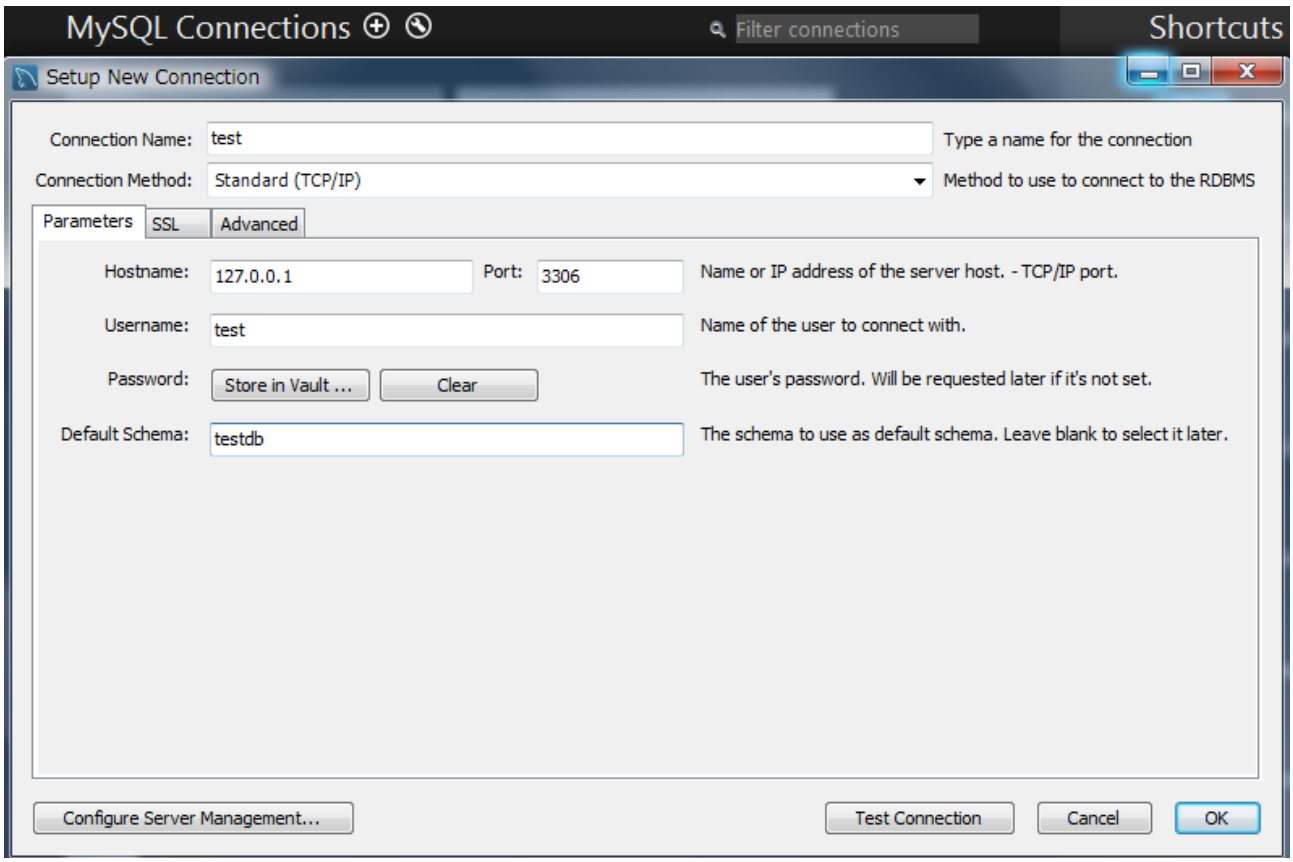
ここではtest@'localhost'というユーザに対しtestというパスワードでtestdbへのアクセスを許している。

そうすると次のようにtestというユーザとパスワードが登録されていることが確認される:

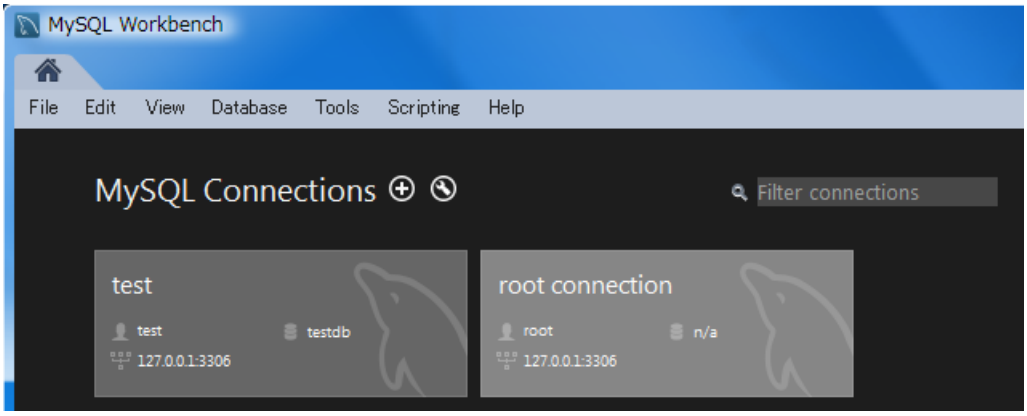
```
mysql> select host, user, password from mysql.user;
+-----------+------+-------------------------------------------+
| host      | user | password                                  |
+-----------+------+-------------------------------------------+
| localhost | root | *972652CE85ED41FA786FE7933EB883C909486572 |
| 127.0.0.1 | root | *972652CE85ED41FA786FE7933EB883C909486572 |
| ::1       | root | *972652CE85ED41FA786FE7933EB883C909486572 |
| localhost | test | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
+-----------+------+-------------------------------------------+
5 rows in set (0.04 sec)
```

新規接続の登録と確認

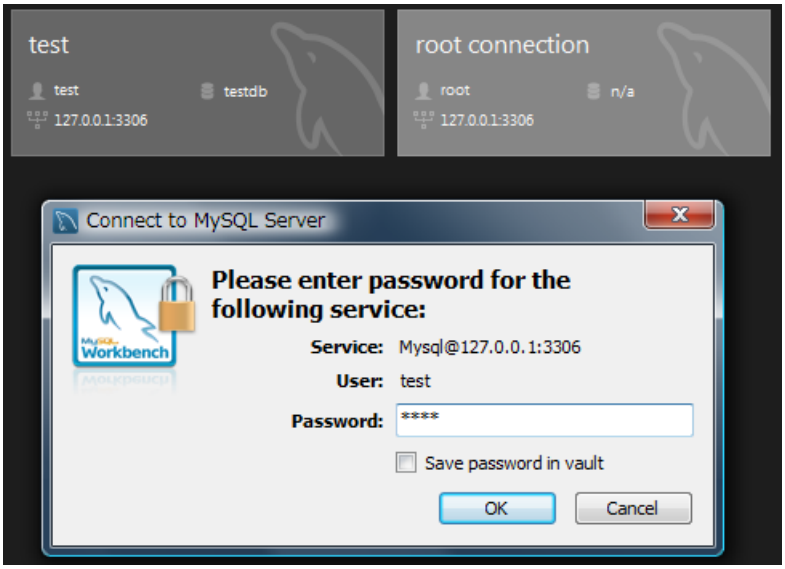
丸に+のシンボルをクリックすると、下図のような新規接続のウィザードが開く。ここに接続名(Connection Name)、ユーザ名(Username)、パスワード(Password)、およびデフォルトDB名(Default Schema)を指定する。パスワードはStore in Vaultボタンを押して入力する。入力が終了したらOKボタンを押す。Test Connectionボタンを押して正しく接続ができることを確認することも重要である。



こうするとMySQL Workbenchには次のように接続が表示される:

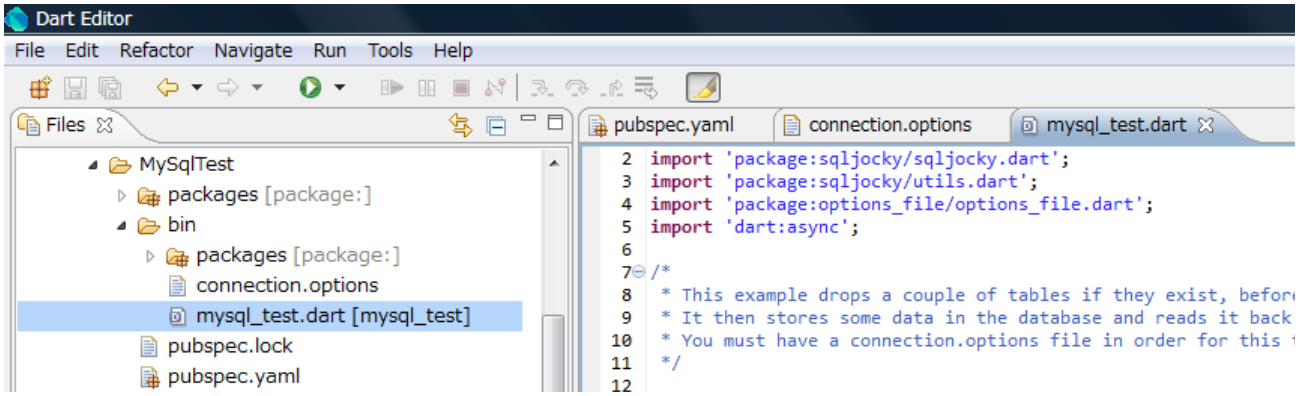


testという接続を右クリックして、Open Connectionを選択すると、下図のようにパスワードを聞いてくるので、testというパスワードを入力後OKボタンを押すと、正しく接続ができることをあらかじめ確認する。



テスト・プログラムの実行

この教材に添付されているdart_code_samplesにはappsというフォルダ内にMySqlTestというフォルダが存在している。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJから\dart_code_samples-master\apps\MySqlTestのフォルダを開くと良い。IntelliJ上ではこのアプリケーションは次のように表示される:



- pubspec.yamlには依存パッケージとしてsqljockyが登録されている。
- pubspec.lockファイルとpackagesフォルダはパッケージ・マネージャが自動的に配置する。
- 実行コードであるmysql_test.dartはもともとsqljockyに含まれているサンプルであるが、ここではbinのフォルダに収容されている。このコードはDB接続に必要な情報を収容したconnection.optionsというファイルを参照している。

mysql_test.dartを右選択して実行させると、次のようなコンソール出力が得られる筈である:

opening connection	接続開始
connection open	接続完了
running example	このサンプルの実行
dropping tables	指定したテーブルの削除開始
dropped tables	テーブルの削除完了

```
creating tables         新規テーブルたちの生成開始
executing queries       クエリ実行
created tables          テーブルたちの生成完了
prepared query 1        プリペアド・クエリ1
executed query 1        プリペアド・クエリ1実行完了
prepared query 2        プリペアド・クエリ2
executed query 2        プリペアド・クエリ2実行完了
querying                クエリ中
got results             クエリ結果取得
ID: 1, Name: Dave, Age: 15, Pet Name: Rover, Pet Species: Dog
ID: 2, Name: John, Age: 16, Pet Name: Daisy, Pet Species: Cow
ID: 2, Name: John, Age: 16, Pet Name: Spot, Pet Species: Dog
ID: 3, Name: Mavis, Age: 93, No Pets
K THNX BYE!
```

Dart言語をある程度理解したSQLの経験者であればmysql_test.dartを追っていけばその動作は容易に理解されよう。sqljockyの使い方はhttps://github.com/jamesots/sqljockyのREADME.mdのUsageを見ると良い。

16.3節　　パッケージの詳細

＊＊＊

本節は基本的にGoogleのパッケージ・レイアウト規約の翻訳である。

＊＊＊

Pubにおけるパッケージは、コードだけでなく画像などのリソース、ドキュメント、またテストに必要なファイルなどそのアプリケーションに必要ないろんな形式のファイルを含めたディレクトリといえる。再使用可能なライブラリなどもまたパッケージである。

- アプリケーション・パッケージ(application package)
  他のパッケージたちを使うものの、それ自身は再使用されないパッケージをいう。

- ライブラリ・パッケージ(library package)
  他のパッケージたちによって再使用されることを意図したパッケージ。他の人たちが使用できるようにどこかにホストされる。通常他のパッケージをインポートしたコードを含む。ライブラリ・パッケージは依存物(dependancy)として使われる。

殆どの場合これらの相違は無く、単に「パッケージ」と呼ばれる。相違が問題になる場合は「アプリケーション・パッケージ」または「ライブラリ・パッケージ」と区別している。

パッケージのレイアウト

ライブラリ・パッケージは多くの人たちが利用するものであるので、Googleはパッケージのレイアウトやファイルの名前付けに関し以下の規約に従うことを求めている。今後開発されるであろういろんなツールもこの規約にあっ

たパッケージを対象にすることになろう。

全体像を把握しやすいように規約に本準拠した完全なパッケージ（便宜的にメキシコ料理のenchiladaという名前がつけられている）の例を示す:

```
enchilada/
  pubspec.yaml
  pubspec.lock *
  README.md
  LICENSE
  bin/
    enchilada
    packages/ **
  doc/
    getting_started.md
  example/
    lunch.dart
    packages/ **
  lib/
    enchilada.dart
    tortilla.dart
    src/
      beans.dart
      queso.dart
  packages/ **
  test/
    enchilada_test.dart
    tortilla_test.dart
    packages/ **
  tool/
    generate_docs.dart
  web/
    index.html
    main.dart
    style.css
```

各要素を説明すると:

```
enchilada/
  pubspec.yaml
  pubspec.lock *
```

各パッケージはそのルート・ディレクトリにpubspec（すなわちpubspec.yamlというヤムル・ファイル）を持つ。このパッケージで一旦インストール(pub install)や更新(pub update)を実行するとロック・ファイル(pubspec.lock)が作られる。そのパッケージがアプリケーション・パッケージの場合は、ソース・コードの管理に使われる。

```
enchilada/
  packages/
```

```
  ...
```

Pubインストールを実行するとpackagesというディレクトリが作られる。これはユーザが気にする必要はない。

```
enchilada/
  README.md
```

オープン・ソースの世界では一般的にプロジェクトのトップ・レベルにREADME、LICENSE、AUTHORSなどのファイルを置く。そのなかでもREADMEは極力置くことが求められる。Gitでお馴染みのマークダウン記法によるREADME.mdファイルが一般的である。

パブリックなライブラリ・パッケージ

```
enchilada/
  lib/
    enchilada.dart
    tortilla.dart
```

多くのパッケージはライブラリ・パッケージで、これらは他のパッケージがインポートして使用するためのDartのライブラリを含む。これらのDartライブラリは上の例で示したようにlibというディレクトリに収容されている。

ほとんどのライブラリ・パッケージは単一のDartライブラリを指定するものなので、その場合はパッケージ名は通常そのDartのライブラリ名と同じものを使用する。別の名前が使われる場合は、ユーザはそのライブラリをパッケージ名とライブラリ・ファイルの名前を使ってインポートすることになる。以下はそれらの例である:

```
import "package:enchilada/enchilada.dart";
import "package:enchilada/tortilla.dart";
```

もし自分のパブリックなライブラリたちの編成を変えたい場合が出たときには、libの中にサブ・ディレクトリを作ることもできる。その場合は、ユーザはそれらをインポートするときはそのパスを指定することになる。例えば次のようなディレクトリ階層を作ったとしよう:

```
enchilada/
  lib/
    some/
      path/
        olives.dart
```

その場合は、ユーザがolives.dartをインポートするときは次のように指定する:

```
import "package:enchilada/some/path/olives.dart";
```

ライブラリたちのみがlibディレクトリに置かれねばならないことに注意のこと。エントリ点（即ちmain()関数を持ったDartコード）はlibには置けない。そのようなDartコードをlibに置くと、それが含んでいるpackage: インポートは解決できないことを読者は判るだろう。エントリ点はbin、example、test、tool、あるいはwebといったしかるべきディレクトリに置かれねばならない。

実装ファイル (Implementation files)

```
enchilada/
  lib/
    src/
      beans.dart
      queso.dart
```

libの内部にあるライブラリたちはパブリックに可視であり、他のパッケージたちはそれらを自由にインポートできる。しかしパッケージのコードの多くは内部実装ライブラリ(internal implementation libraries)たちであって、そのパッケージ自身によってインポートされ使われるべきものである。それらのファイルはsrcと呼ばれるlibのサブディレクトリ内部に置かれる。srcのなかにサブディレクトリを作ることもそれが有意義なら構わない。

同じパッケージ内の他のDartコードのなかから（libのなかの他のライブラリたち、binのなかのスクリプトたち、及びtestsのように）lib/src内にあるライブラリをインポートするのは自由であるが、他のパッケージのlib/srcディレクトリからは決してインポートしてはならない。これらのファイルたちはそのパッケージのパブリックなAPIの要素ではなく、これらは変更される可能性があり、そうなると読者のコードを動かなくしてしまう可能性がある。

読者自身のパッケージ内からライブラリたちを使うときは、たとえsrc内のものであっても、読者はそれらをインポートするのに"package:"を使わねばならない。次の例は全く正確な記述である：

```
import "package:enchilada/src/beans.dart";
```

ここで使う名前（この例ではenchilada）はそのpubspec内で読者のパッケージを指定するときの名前である。

ウェブ・ファイル (Web files)

```
enchilada/
  web/
    index.html
    main.dart
    style.css
```

Dartはウェブの世界を対象にした言語であるので、多くのpubパッケージはウェブの要素を扱う。即ちHTML、CSS、画像、及びおそらくはJavaScriptも扱うことを意味する。これらの総ては皆さんのパッケージのwebディレクトリに収容される。このディレクトリ中身をどのように構成するかは自由である。従ってそのほうが良ければサブ・ディレクトリたちを設けても構わない。

そして重要なことであるが、Dartのどのwebのエントリ点（言い換えれば<script>タグのなかで参照されているDartのスクリプトたち）はlibではなくてwebディレクトリに入る。これにより近くにpackagesディレクトリがあってそれにより確実にpackage:インポートたちが正しく解決されるように出来る。

（「自分のウェブ・ベースのサンプル・プログラムを何処におこうか？programsそれともweb？」と自問する事態に

なったときは、exampleディレクトリに収容すると良い）

コマンド行アプリケーション (Command-line apps)

```
enchilada/
  bin/
    enchilada
```

一部のパッケージはコマンド行から直接実行できるプログラムを定義している。そのようなプログラムはシェル・スクリプトまたはDartを含むその他のスクリプト言語であり得る。pubアプリケーションそれ自身もひとつの例であり、これはpub.dartを呼び出す簡単なスクリプトである。

もし読者のパッケージがそのようなものを定義しているときは、それをbinという名前のディレクトリに収容する。

何時かの時点でpubは読者のシステム・パスにそのディレクトリを自動的に付加して、これらのスクリプトが簡単に呼び出せるようになろう。

テスト

```
Tests#
enchilada/
  test/
    enchilada_test.dart
    tortilla_test.dart
```

まともなパッケージはテストを持っていなければならない。pubの場合は、その規約はそれらをtestディレクトリ（あるいはそのほうが良ければその中の何らかのディレクトリ）に置き、それらのファイル名の終わりに_testが付くということである。

一般的にこれらはunittestパッケージを使っているが、その他のテストのためのシステムを使うことも自由である。

ドキュメンテーション (Documentation)

```
enchilada/
  doc/
    getting_started.md
```

コードとテストを作ったら次に行うことは、良いドキュメンテーションで自分の開発物の影響力を最大化することである。ドキュメントはdocという名前のディレクトリ内に置かれる。現時点ではこのなかの構成や書式に関する指針は無い。好みのマークアップ書式を使ってドキュメントを作成するとよい。

このディレクトリは単にIntelliJ上でdartdocを使って自分のソース・コードから自動的に生成されたドキュメントを収容するだけではいけない。それはそのパッケージ内で既にそのコードから直接得られるものなので、単にそれをここに置くのは冗長である。そうではなくて、このディレクトリは生成されたAPI参照に加えてチュートリアル、ガイド、およびその他の作成者が自分で書いたドキュメンテーションのためのものである。

サンプル (Examples)

```
enchilada/
  example/
    lunch.dart
```

この時点で皆さんは恵まれた機会を持とうとしている。コード、テスト、ドキュメント、それ以外にユーザは何を望むだろうか？無論それは皆さんのパッケージを使ったスタンドアロンのサンプル・プログラムである。これらのプログラムはexampleディレクトリの中に置かれる。そのサンプルが複雑で複数のファイルが使われているときは、各サンプルのためにディレクトリを作ることを検討されたい。そうでないときは、各々をexapleの中に置くことができる。

自分自身のパッケージ内からファイルをインポートするのにpackage:を使う際に検討する場所としてここは重要である。package:を使うことで自分のパッケージ内のサンプル・コードは自分のパッケージ外で使われるコードとまさしく同じにすることができる。

内部ツールとスクリプト (Internal tools and scripts)

```
enchilada/
  tool/
    generate_docs.dart
```

枯れたパッケージではしばしば小規模なヘルパ・スクリプトとプログラムを有しており、これによりそのパッケージ自身を開発中にそれを実行できるようにしている。テスト・ランナ、ドキュメンテーション生成ツールその他のオートメーション・ツールの類である。

binのなかのスクリプトとは違って、これらはそのパッケージの外部ユーザの為のものではない。これらはtoolという名前のディレクトリに置かれる。

16.4節　Pubspecの書式 (Pubspec Format)

注意：この節はpub.dartlang.orgにあるPubspec Formatというドキュメントの翻訳をベースとしている。

各pubパッケージはその依存物たちを指定できるように何らかのメタデータが必要である。他の人たちと共有されるようなpubパッケージではまた他のユーザがそれらを発見できるように何らかのその他の情報が含まれている必要がある。pubはこれをpubspec.yamlという名前のファイルにストアしている。このファイルはしたがってYAML言

語で書かれている。

トップ・レベルに置かれるのは一連の以下のフィールドたちである。現在サポートされているフィールドは以下のものである:

Name（名前）	各パッケージに必要
Version（バージョン）	パッケージに必要で、これはpub.dartlang.org上でホストされる
Description（記述）	パッケージに必要で、これはpub.dartlang.org上でホストされる
Author/Authors（著者）	オプショナル
Homepage（ホームページ）	オプショナル
Dependencies（依存物）	そのパッケージに依存物がなければオミットできる

その他のフィールドは無視される。シンプルではあるが完全なpubspecは次のようなものになる:

```
name: newtify
version: 1.2.3
description: >
  Have you been turned into a newt? Would you like to be? This
  package can help: it has all of the newt-transmogrification
  functionality you've been looking for.
author: Nathan Weizenbaum <nweiz@google.com>
homepage: http://newtify.dartlang.org
dependencies:
  efts: '>=2.0.4 <3.0.0'
  transmogrify: '>=0.4.0'
```

名前 (Name)

各パッケージには名前が必要である。自分のプログラムが世界の舞台で評判を得れば素晴らしいことである。また公開することで他のパッケージがこの名前で皆さんのパッケージを参照する。

この名前は小文字のみでなり、単語のセパレータとしてアンダスコアが使われる（例:just_like_this）。基本ラテン文字とアルファベットの数字のみ[a-z0-9_]を使い、またDartの有効な識別子であるようにする（即ち最初が数字文字でなく、また予約語でないこと）。

はっきりしていて簡便でまた既に使われていない名前を選択する。あとになって悩むことがないように https://pub.dartlang.org/packages のクイック・サーチを使って自分の名前がまだ誰も使っていないことを確認する。

バージョン (Version)

各パッケージにはバージョンがある。皆さんのパッケージをホストするにはバージョン番号が必要であるが、ローカルだけのパッケージの場合は無くてもよい。バージョン番号がない場合は暗示的に0.0.0というバージョンだと

見做される。

バージョン管理は面倒ではあるが、バージョンアップが頻繁におきるなかでコードを再利用するには必要悪である。バージョン番号は0.2.43といったようにドットで区切られた3つの数字で構成される。これはまたオプショナルにはビルド(+hotfix.oopsie)またはプリ・リリース(-alpha.12)がつけられる。

公開(publish)する度に皆さんはある特定のバージョンとしてそれを公開することになる。一旦公開したら、それはしっかり封止するようにし、誰ももう変更できないようにする。更なる変更をするときは、新しいバージョンが必要になる。

バージョン番号を選択するには、他のユーザたちが理解できるよう意味的バージョン設定（大規模改版（プロジェクト管理のマイルストーンやリリースのアップ）、マイナな改版、パッチなど）に従うこと。

記述 (Description)

自分だけの個人的なパッケージであればこれはオプショナルだが、自分のパッケージを他の人たちに共有できるようにしたいときは、記述を用意すべきである。これは比較的身近な文章（数センテンス、一つのパラグラフだけ）で、カジュアルな読者がそのパッケージに関し知りたいと思われることを伝えるようにする。

記述を皆さんのパッケージの宣伝文句だと考えればよい。ユーザたちはパッケージをブラウズするときにこれを使う。これは単純なテキストであって、マークダウンまたはHTMLではない。これはREADMEとおなじものである。

著者 (Author/Authors)

これらのフィールドを使って自分のパッケージの著者（たち）を記述して連絡情報を提供することが推奨される。authorは著者が一人のときに使い、authorsのほうは一人以上の人たちがこのパッケージを書いたときにYAMLのリストを用いて使う。各著者は単一の名前(e.g. Nathan Weizenbaum)または電子メール・アドレスつき(e.g. Nathan Weizenbaum <nweiz@google.com>)のどちらでもよい。たとえば：

```
authors:
- Nathan Weizenbaum <nweiz@google.com>
- Bob Nystrom <rnystrom@google.com>
```

もし誰かが皆さんのパッケージをここにアップロードするときは、pub.dartlang.orgはこの電子メール・アドレスを示し、皆さんがそれをOKするようにする。

ホームページ (Homepage)

これは皆さんのパッケージのためのウェブサイトを指すURLでなければならない。ホストされているパッケージの場合は、このURLはそのパッケージのページからリンクされる。技術的にはこれはオプショナルであるが、これは付して欲しい。ユーザたちは皆さんのパッケージがどこから来ているか理解できるようにする。少なくともそのソー

ス・コードをホストしているURL（GitHub, code.google.comなど）のURLが使える。

依存物 (Dependencies)

最後にこのpubspecの存在意義であるdependenciesを説明する。このフィールドでは皆さんのパッケージが動作するのに必要な各パッケージをリストする。直接依存しているもの（即座の依存物たち(immediate dependencies)）のみをリストする。他動的依存物たち(transitive dependencies)はpubが自動的に処理する。

各依存物毎に皆さんがいぞんしているパッケージの名前を指定する。ライブラリ・パッケージの場合は、そのパッケージの皆さんが許容するバージョンの範囲を指定する。またpubに対しそのパッケージをどのように特定するか、およびそのパッケージを探すのに必要なそのソースの付加的情報を告げるためにそのソースを指定することもできる。

皆さんがどれだけ多くのデータを指定するかに依存して、依存物の指定方法が幾つかある。最も短い方法は単に名前を指定するだけである:

```
dependencies:
  transmogrify:
```

この場合transmogrifyの依存性をつくる。すべてのバージョンが適用され、デフォルトのソース（このサイト自身の中）を使ってそれをロックする。あるバージョンの範囲に依存性を制限するには、バージョン制約を指定できる:

```
dependencies:
  transmogrify: '>=1.0.0 <2.0.0'
```

この場合デフォルトのソースと1.0.0から2.0.0まで（2.0.0は含まず）のバージョンが許されるtransmogrifyの依存性をつくる。バージョン制約の文法の詳細は以下に記されている。

ソースを指定したいときは、この文法は少し違ってくる:

```
dependencies:
  transmogrify:
    hosted:
      url: http://some-package-server.com
```

これはホストされたソースを使ってtransmogrifyパッケージに依存する。このソース・キー（ここではurl:キーを使ったマップがひとつ）の下にあるすべてがこのソースに渡される記述である。各ソースは以下に詳述される記述書式を持つ。

これにもバージョン制約をかけることができる:

```
dependencies:
  transmogrify:
    hosted:
      name: transmogrify
      url: http://some-package-server.com
    version: '>=1.0.0 <2.0.0'
```

この長い書式はデフォルトのソースを使わない場合、あるいは指定するのに長い記述が必要な場合に使われる。しかし殆どの場合、単純な“name: version”を使うことになろう。

依存物のソース (Dependency sources)

以下はpubがパッケージを探すことが可能なソースと、それが可能なものの記述である:

ホストされたパッケージ (Hosted packages)

ホストされたパッケージとはpub.dartlang.orgのサイト(あるいは同じAPIに対応した別のHTTPサーバ)からダウンロードできるパッケージのことをいう。殆どの依存物たちはこの書式になろう。これらは以下のようになる:

```
dependencies:
  transmogrify: '>=0.4.0 <1.0.0'
```

ここで指定していることは、自分のパッケージが“transmogrify”という名前のホストされたパッケージに依存しており、0.4.0から1.0.0の範囲の版(1.0.0自身は含まない)に対応するということである。

自分自身のパッケージ・サーバを使いたいときは、そのURLを指定した記述が使える:

```
dependencies:
  transmogrify:
    hosted:
      name: transmogrify
      url: http://your-package-server.com
    version: '>=0.4.0 <1.0.0'
```

SDKパッケージ (SDK packages)

一部のパッケージはDart SDKの一部として組み込まれている。これはDartをインストールすれば自由に使えるバッテリ内蔵(batteries included)パッケージである。

```
dependencies:
  i18n:
    sdk: i18n
```

ここではsdkはこのパッケージはDart SDK組み込みのものであり、“i18n”はそのパッケージの名前であることを意味する。

Gitパッケージ (Git packages)

しばしば開発段階にあってまだ正式にリリースされていないものを使う必要がある場合がある。その場合は多分の皆さんのパッケージ自身も開発中であり、また同時に未だ開発中である他のパッケージを使っている。そのような作業をやり易くする為に、皆さんはGitレポジトリにストアされたパッケージに直接依存することができる。

```
dependencies:
  kittens:
    git: git://github.com/munificent/kittens.git
```

ここでgitはこのパッケージはGitを使って発見できることを示し、またそのあとのURLはそのパッケージのクローンの為に使えるGit URLであることを示す。Pubはこのパッケージはこのgitレポジトリのルートに存在するとみなす。

もし特定のコミット、ブランチ、またはタグに依存したいときは、ref引数を指定できる:

```
dependencies:
  kittens:
    git:
      url: git://github.com/munificent/kittens.git
      ref: some-branch
```

このrefはGitがあるコミットを特定するのに許しているものなら何でもよい。

バージョン制約 (Version constraints)

皆さんのパッケージがアプリケーションの場合は、一般には自分の依存物たちに対しバージョン制約を指定する必要は無い。皆さんが自分のアプリケーションを作成するときは通常はその依存物の最新のバージョンを使いたいはずである。次に皆さんはロックファイル(lockfile)をつくり、そこで自分の依存物たちをそれらのあるバージョンに固定化する。そこで自分のpubspecのなかでバージョン制約を指定するのは通常は冗長なことである(指定したならそれは可能ではあるが)。

しかしながら、皆さんがユーザたちに使ってもらいたいと思うライブラリ・パッケージの場合は、バージョン制約を指定することが重要である。そうすることで皆さんのパッケージを使っているユーザたちは依存物たちのどのバージョンが皆さんのライブラリと互換性があって依存できるかを知ることができる。皆さんの目的はなるべく広いバージョンの範囲にして皆さんのユーザたちたちに柔軟性を持たせることである。しかしながら一方では動作しないとわかっているバージョンまたはテストしていないバージョンを排除する為に十分狭くせねばならない。

Dartのコミュニティでは意味的バージョン設定を使っていて、これによりどのバージョンが動作するかを判り易くしている。もしある依存物の1.2.3で自分のパッケージが正しく動作することが判っているときは、意味的バージョン設定によって(少なくとも)2.0.0まで動作するはずだということが判る。

バージョン制約はスペースで分離されたバージョン要素のつながりである。ある要素は以下のもののひとつである:

any	文字列"any"はどのバージョンも許されることを意味する。これはバージョン制約を書かないことと等価ではあるが、より明示的指定である。

1.2.3	まさしくそのバージョンのみにその依存物を固定する具体的なバージョン番号。ユーザたちにバージョンをロックさせ得ること、及びユーザたちが皆さんのパッケージ及び一緒に使うパッケージたちを使いずらくするので、極力これは使用しないこと。
>=1.2.3	このバージョンまたはそれ以上のバージョンを可能とする。一般的にはこれが使用されることになる。
>1.2.3	このバージョン以上が許されるが、このバージョン自身は許されない。
<=1.2.3	このバージョンまたはそれ以下のバージョンが許される。これは一般には使われないだろう。
<1.2.3	このバージョン以下が許されるが、このバージョン自身は許されない。これはこれ以上のバージョンでは自分のパッケージでは動作しないことが判っているバージョン（何らかの大規模変更がされた最初のバージョンなので）が指定できるので、皆さんが通常使うものである。

これらの比較演算子の間及びその後のバージョン番号との間にスペースを入れ他はならない。>=1.2.3は可であるが>= 1.2.3は不可である。

バージョン要素は好きなだけ指定できるし、それらの範囲も指定できる。例えば、>=1.2.3 <2.0.0は 2.0.0を除いて1.2.3から2.0.0までの範囲が許される。

> はまた有効なYAML文法であるので、そのバージョン制約がこれで始まるときは引用符を付す（'<=1.2.3 >2.0.0'のように）ことに注意のこと。

16.5節　　Pubのコマンド（pub commands)

Pubのツールは単にDart Editorに組み込まれているだけではなく、コマンド行（Windowsではコマンド・プロンプト）でより幅広く使用できる。通常それらのコマンドは:

- dart-sdk\binにpathを設定する
- プロジェクトのフォルダにディレクトリを置く

ことで実行される。

以下はそれらのコマンドと、どのような目的で使われるかを示したものである。

コマンド	アプリケーションの作成と保守	アプリの開発		配備	出版	機能
		ウェブ・ベース	サーバ・ベース			
pub build				✔		buildディレクトリをつくり、dart2jsでJSスクリプトを生成し、それらのリソースをbuildディレクトリに収容する。
pub cache	✔					システム・キャッシュと共に機能する。そのキャッシュに新たなパッケージを付加する、または総てのパッケージを再インストールする。
pub deps	✔					あるパッケージで使われる総ての依存物をリストする。

pub get	✔					そのアプリケーションの為の依存物としてリストされているパッケージたちを取り込む。
pub global			✔			パッケージ外からグローバルに得られるパッケージ内または依存物内のDartコードを実行する。
pub publish					✔	そのパッケージをpub.dartlang.orgにアップロードする。
pub run			✔			パッケージ内または依存物内のDartコードを実行する。
pub serve		✔				開発サーバを開始させる。このサーバはブラウザからlocalhostとしてアクセスでき、ウェブ・ベースのアプリを確認できるようにする。
pub upgrade	✔					そのアプリケーションの為の依存物としてリストされているパッケージたちの最新版を取り込む。
pub uploader					✔	そのパッケージをpub.dartlang.orgにアップロードする。

共通オプション、及び各コマンドの詳細はPubコマンドのマニュアルを見るか、上記の表の中の各リンクを追って頂きたい。

第17章 イベント処理 (Asynchronous Processing)

イベント駆動型の系は、非同期処理のひとつである。同期処理系ではある処理が終わらないと次の処理に移れない。しかしながら多くの処理が存在していると、同期処理では効率が悪いし、全体の処理が遅れてしまうことになる。単一スレッド・ベースのDartでは、これを解決する為にブラウザのユーザ・インターフェイスなどでイベント処理の為の豊富なAPIが用意されている。加えてdart:async（エイシンクと発音）ライブラリにFuture、Completer及びStreamインターフェイスが用意されており、非同期処理間の連携が可能になっている。下表に示すとおりFutureは式を非同期処理するためのコンセプトであり、StreamはIterableを非同期化するためのコンセプトである。これらの詳細を以下の節で記述することにする。

関数の4つの型

	単一	複数
同期	T	Iterable<T>
非同期	Future<T>	Stream<T>

Dartではdart:html（クライアントのみに実装）やdart:io（サーバのみに実装）などでイベント・ベースのコードが書きやすくする多くのAPIが用意されている。特にブラウザ上でDartを走らせるアプリケーションでは、タイマやユーザによる画面操作やネットワークからの受信などがイベントの要因になり、殆どの場合アイソレートを使わなくても十分なスループットが得られるであろう。

JavaScriptでもそうだが、Dartではプログラムの殆どがコールバック関数で記述されることになり、クラスを使うなど書き方に注意しないと非常に見づらいコードになってしまうので注意が必要である。

イベント・ハンドラはVM実装にも依るが、通常は処理待ち状態のハンドラ処理たちを、ひとつのスレッド（トップ・レベルのmainを実行したスレッド）のみをスケジューリングして実行させている。従って、時間がかかる処理のために他のハンドラ処理が待たされてしまう可能性があることに注意しなければならない。また複数のハンドラが待ち状態になったときにどのハンドラから受け付けられるかは、プログラムからは関与できない。

Dartにおけるイベント・キューのスケージューリングに関しては"The Event Loop and Dart"という記事にわかりやすく説明されている。

このAPIのこれまでの経過：
2012年2月のAPI改訂以前はIsolate#spawnメソッドはFutureオブジェクトを返すようになっていた。従ってFutureとCompleterの説明を以前は並行処理の章に含めていたが、API改正によりアイソレートとFuture / Completerとがより分離されたので、この資料も「イベント処理」の章として分離した。なお2013年1月のM3変更により、Future及びCompleterは新たに用意されたライブラリdart:asyncに移されるとともに、Futuresの削除を含めて大幅な変更がなされた。また連続したデータやイベントを処理するためのStreamも導入された。

17.1節　FutureとCompleter

FutureとCompleterは共に総称型の抽象クラスである。これらの抽象クラスはイベント・ドリブンのスタイルのアプリケーションに使われる。

Futureは後で何時かの時点で値を取得する為に使われる抽象クラスである。Futureの受け手はFutureの**then**メソッドにコールバック関数を渡すことで値を取得・処理する。例えば：

```
Future<int> future = getFutureFromSomewhere(); future.then((value) {
print("I received the number " + value);
});
```

ここではコールバック関数は関数リテラルなので、あたかもthenメソッドを定義しているかのように見えるところが面白い。thenメソッドは新たにFutueオブジェクトを返すので、その後の一連の処理を記述するのが楽になる:

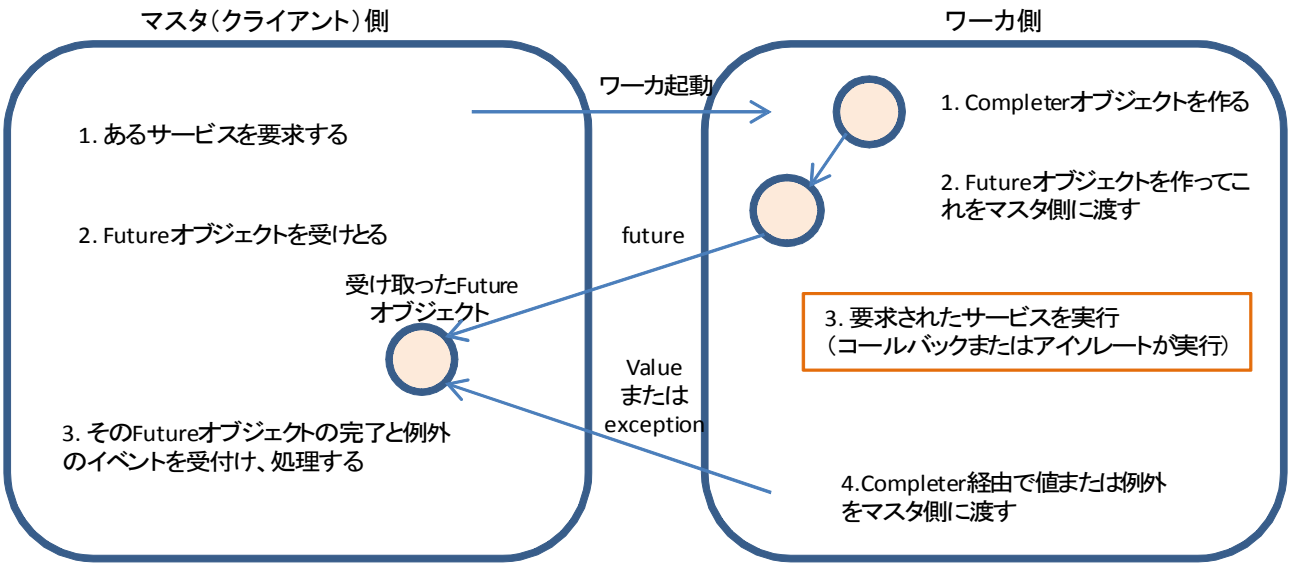
```
Future result = costlyQuery();

return result.then((value) => expensiveWork())  // costlyQueryが終わったら
            .then((value) => lengthyComputation())  // 次にexpensiveWorkが終わったら
            .then((value) => print('done!'))  // 最後にlengthyComputationが終わったら
            .catchError((exception) => print('DOH!'));  // この間エラーが発生したときは
```

これは後述の順序づけられたイベント処理の項でさらに解説する。

一方CompleterのほうはFutureオブジェクトをつくり、その値が取得できるようになったときにそのFutureオブジェクトに値を供給するのに使われる。その値を直ちに返すのではなくてFutureオブジェクトを返したいサービスの場合は、Completerを次のように使用する:

```
Completer completer = new Completer();
// futureオブジェクトをクライアントに送り返す...
return completer.future;
...

// あとで値が取得可能になったときに、completer.complete(value);を呼ぶ
completer.complete(value);

// そうではなくて、そのサービスが値を作り出せないときは
// エラーをクライアントに送り返すことができる
completer.completeError(error);
```

これらのオブジェクトの一般的な使い方は次のようなアプリケーションをイメージすれば良いだろう:



但し場合によっては既に値がセットされた(完了した)Futureオブジェクトを返すだけで済むアプリケーションもあろう。

CompleterはそのFutureたちを非同期で完了することに注意されたい(M4以前は同期していた)。

以下はこれらの抽象クラスの基本的なメソッドの使い方を示す簡単なコードである:

```
import 'dart:async';
main() {
  Completer<String> completer = new Completer<String>();

  Future<String> future = completer.future;
  future.then((message) {
    print("Futureはメッセージ$messageで完了");
  });
  future.catchError((exception){
    print("$exceptionを受付けた");
    return true;
  });
  Exception exception = new Exception("completerからの例外");
//  completer.completeError(exception);
  completer.complete("山");
  completer.complete("川");
}
/*
Futureはメッセージ山で完了
Exception: future already completed
またはコメントアウトを外すと--
Exception: completerからの例外を受付けた
Exception: future already completed
*/
```

17.2節　　非同期コールバック処理

より具体的な前図に沿った非同期処理の例を示そう。

ボタン・クリックの待機

典型的な非同期処理はユーザの画面操作の待機であろう。これらはイベントとして下位層から渡されるが、アプリケーションはそのイベントだけを何時も待っていると:

- その為に処理の進行がブロックされ、全体の処理が遅れてしまう
- 別の画面操作やその他のイベントがイベントが先に発生してもそれらは後回しになってしまい、複数のイベントを到来順に処理できなくなる

という問題があるので、非同期処理は欠かせない。

FutureSample_1というアプリケーションは、ユーザがボタンをクリックするのを待ってそれを報告するClickProcessWorkerというワーカと、そのワーカを生成して仕事を依頼するFutureSample_1という2つのクラスで構成されている。

FutureSample_1.dart

```
// Dart code sample descripting basic use of Future / Completer interfaces
// Tested on Dartium
// Source : www.cresc.co.jp/tech/java/Google_Dart/DartLanguageGuide.pdf
// March 2012, by Cresc corp.
// January 2013, Incorporated API and location of dart.js changes
// February 2013, Incorporated API changes

import 'dart:html';
import 'dart:async';

void write(String message) {
  String timestamp = new DateTime.now().toString();
  document.querySelector('#status').insertAdjacentHtml('beforeend', '<br>$timestamp :
$message');
}

class ClickProcessWorker {
  Future run() {
    var completer = new Completer();
    var isComplete = false;
    ButtonElement button = document.querySelector("#button");
    button.onClick.listen((e){
      if (!isComplete) {
        completer.complete('button');
        isComplete = true;
      }
    });
    write('Returned future');
  return completer.future;
  }
}

class FutureSample_1 {
  void run() {
    write("FutureSample_1.run() started");
    Future future = new ClickProcessWorker().run();
    future.then( (result) => write('Accepted $result result'));
    // do things here
    write("FutureSample_1.run() exited");
  }
}

void main() {
  new FutureSample_1().run();
}
```

ClickProcessWorkerクラスの実行メソッドrunはFuture型のオブジェクトを返すメソッドである:

1. Completer型のオブジェクトcompleterを用意する
2. HTMLドキュメントのボタン要素にたいし、クリックされたときにcompleter.complete('buttom')文で完了を通知するとともに'button'という値を渡すイベント処理のコールバック関数を付加する。なおif (!isComplete)という条件は、既に完了したFutureにたいしcompleter.completeメソッドを呼ぶ(複数回クリックしたとき)と例外が発生するからである。button.onClick.listenというメソッドはM3から導入されたStreamベースのイベント受付処理であり、これは「ストリーム」の節で詳しく説明する。
3. スクリーン上にタイムスタンプを付けてfutureオブジェクトを呼び出し側に返したことを知らせる
4. completer.futureでこのメソッドを呼び出した側にfutureオブジェクトを返す

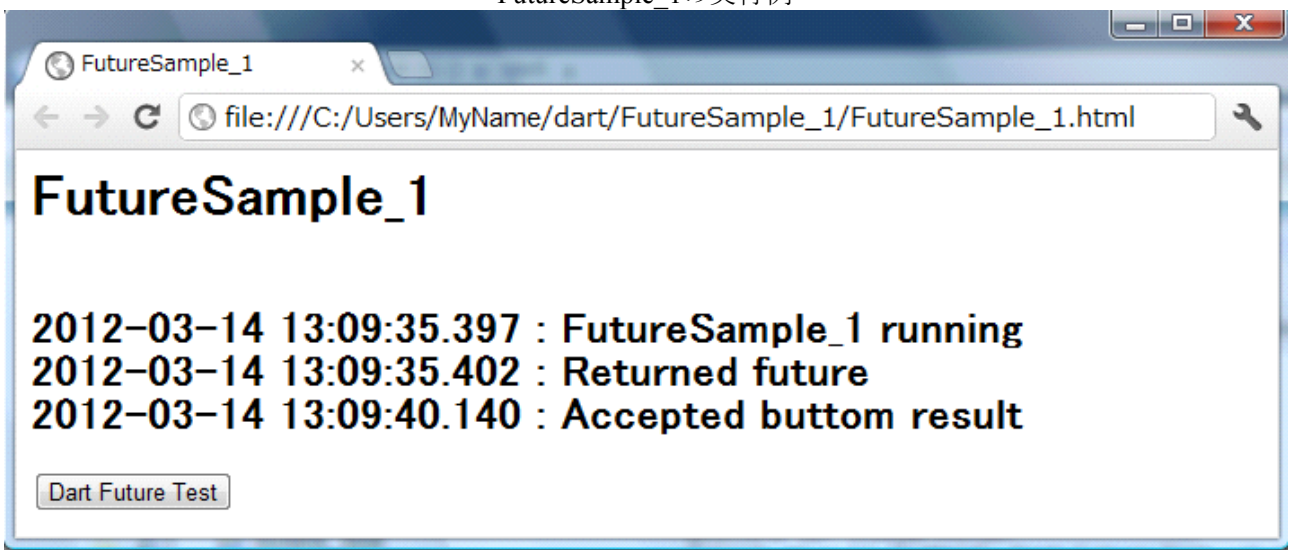
このメソッドは、ボタン・クリックのイベントを待つことなく帰るので、結局直ちに呼び出し側にFutureのオブジェクトが渡されることになる。その後ユーザがボタンをクリックすれば、イベント・ハンドラのスレッドがcompleter.completeで呼び出し側にその結果を知らせる。

FutureSample_1クラスの実行メソッドrunの動作は次のようである:

1. new ClickProcessWorker().run()でワーカのオブジェクトを生成してそのrunメソッドを呼ぶことで、ワーカを起動するとともにFutureのオブジェクトを受理する
2. 必要があればボタン・クリックを知る前にここで何か仕事をする
3. future.thenメソッドでワーカからの完了報告を待ち、報告の結果valueが渡されたらそれをタイムスタンプ付きでスクリーン上に表示する

下図はそのアプリケーションの実行例である:

FutureSample_1の実行例



1. 2012-03-14 13:09:35.397 : FutureSample_1 running　　このアプリケーションが起動した
2. 2012-03-14 13:09:35.402 : Returned future　　ワーカが起動し、futureオブジェクトを返した
3. 2012-03-14 13:09:40.140 : Accepted button result　　ユーザのボタン・クリックでbuttonという値を受け付けた

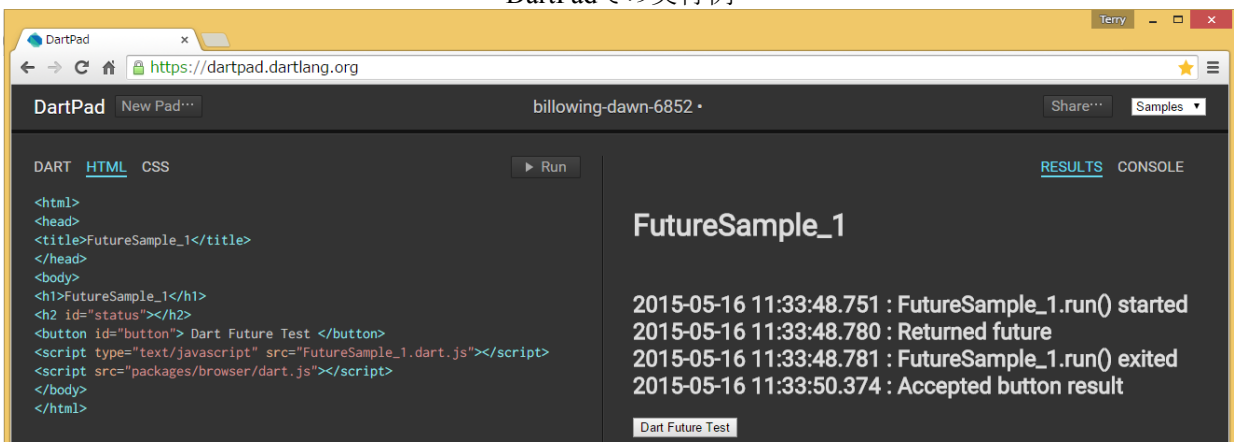
ちなみにこのアプリケーションのHTMLファイルは次のようである:

FutureSample_1.html

```
<html>
  <head>
    <title>FutureSample_1</title>
  </head>
  <body>
    <h1>FutureSample_1</h1>
    <button id="button"> Dart Future Test </button>
    <h2 id="status"></h2>
    <script type="application/dart" src="FutureSample_1.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJなどのIDEから\dart_code_samples-master\apps\FutureSample_1のフォルダを開くと良い。あるいはDartPadに上記2つのコードをコピー/ペーストして実行させてもよい。下図はDartPadでの実行例である:

DartPadでの実行例



ファイル操作における非同期処理

ファイル操作は通常ディスク・ドライブをアクセスするので時間がかかる。例えば:

code_17.2a.dart

```
import 'dart:io';

main() {
  print('ファイル読み出し開始');
  var file = new File('testData.txt');
  var contents = file.readAsStringSync(); // プログラムは総ての内容が読み出されるまでブロックされてしまう
  print(contents);
  print('読み出し完了');
}
```

dart:ioライブラリのFileという抽象クラスのreadAsStringSync([Encoding encoding = Encoding.UTF_8])というメソッドは、そのファイルのデータをUTF_8の文字列だとして同期して読みだし、Stringを返すメソッドである。従ってプログラムは下位層がそのファイルを読みだすまでそこで待たされてしまう(英語ではブロックするという)。

これに対しreadAsString([Encoding encoding = Encoding.UTF_8])というメソッドは非同期で読み出すもので、Future<String>オブジェクトを即座に返す。そのFutureオブジェクトが完了したときにプログラムが待機状態にあればthen(またはwhenComplete)で登録されたコールバック関数を実行する。

code_17.2b.dart

```
import 'dart:io';

main() {
```

```
  print('ファイル読み出し開始');
  var file = new File('testData.txt');
  file.readAsString().then((String contents) {
    print(contents); // ファイルが読みだされたときに出力
    print('読み出し完了');
  });
}
```

このコードは次節で述べる非同期関数を使うとより簡単にあたかも同期処理のごとく記述できる:

code_17.2c.dart

```
import 'dart:io';

main() async {
  print('ファイル読み出し開始');
  var file = new File('testData.txt');
  var contents = await file.readAsString(); // この間他のイベントに対応できる
  print(contents);
  print('読み出し完了');
}
```

データベース・アクセス

サーバ側で非同期動作が必要な典型的なものがデータベースとウェブ・サービスへのアクセスであろう。ともにクライアントからの要求処理の中で最も時間がかかるものである。

以下はその典型的な使い方である:

```
DatabaseConnection db;
Future future = openDatabase();  // DB接続開始
future
 .then((_db) {  // 接続が終わったときの処理を登録
  db = _db;
  db.query();  // DBアクセス
 })
 .catchError(print)
 // 終了したらそのDB接続を解放
 .whenComplete(() => if (db != null) db.close());
```

DBへの各操作に対してもFutureが返されるので、次のようなチェイン(次の節で説明する)の記述が可能になる:

```
db.open()
  .then((_) => db.nuke())
  .then((_) => db.save("world", "hello"))
  .then((_) => db.save("is fun", "dart"))
  .then((_) => db.getByKey("hello"))
  .then((value) => query('#text').text = value)
```

```
  .catchError((e) => print(e));
```

ここでのcatchError関数はFutureで定義されていて、このチェインのどこかで発生した例外を捕捉する。

データベース・アクセスに関しては、「パッケージ・マネージャ」の章の「DBアクセス」の節が参考になろう。

17.3節　　非同期関数（Async Functions）

Dart言語仕様担当のGilad Bracha氏は2014年10月に新規の機能として非同期関数を解説している。これは：

- await式
- asyncメソッド

で構成されている。2014年10月時点ではこれを試すにはdart:asyncのインポートが必要だが、将来はFutureとともにdart:coreに移される予定だという。

非同期関数

非同期関数とはそのボディ部にasync修飾子が付された関数のことをいう。

```
foo() async => 42;
```

非同期関数が呼ばれるとその関数は直ちにFutureを返す。非同期関数のボディ部の実行開始は非同期処理のスケジューラが行う。ボディ部の実行が終了したら、その結果が正常だったかあるいは例外が発生したかにかかわらずそのFutureは完了する。この例ではfooは呼ばれたら直ちにFutureを返し、そのFutureは最終的には42という数字で完了する。

これはasync修飾子を使わなくても次のように記述できる：

```
foo() => new Future(() => 42);
```

しかしasync修飾子は多少は簡素化にはなるが、一番のポイントは関数のなかでawait式が使えるようになることである。

await式

await式を使うと非同期のコードをあたかも同期コードであるかのごとき記述が可能になる。例えばmyFileというFile（詳細はdart:ioのFileクラスを参照のこと）のオブジェクトとした変数を考えてみよう。このファイルを新しい場所であるnewPathにコピーしたければ、

```
String newPath = '/some/where/out/there';
```

そうすると次の行でコピーができ、trueが得られそうである:

```
myFile.copy(newPath).path == newPath;
```

しかしながらDartのI/Oライブラリは非同期なので、copy操作はFutureを返すだけであり、それにはpathを呼ぶことはできない。従ってcopy()から返されたFutureに対するコールバック関数を用意しなければならない。そのコールバック関数が到来パラメタfで比較を行うことになる:

```
myFile.copy(newPath).then((f) => f.path == newPath);
```

これは七面倒くさい記述である。単に非同期のファイル・コピー操作が終了するのを待ってその結果を得たら実行を再開するだけである。await式を使うとその式のとおり事柄が進むことになる:

```
(await myFile.copy(newPath)).path == newPath;
```

このawait式が走るとmyFile.copy()が呼び出され、Futureが返される。実行はそこで止まりそのFutureが完了するのを待機する。そのFutureがfileで完了したら実行が再開される。await式の値はそのFutureの完了値で、即ち待っていたファイルである。そうすれば属性値pathをとりだせ、newPathとそれとを比較できるようになる。これは確かに記述がすっきりするので、重宝しそうである。

一般的にはawait式は次の書式をとる:

```
await e
```

ここでeは単項式である。一般的にeは非同期処理であり、その値はFutureとなろう。このawait式はeを計算し、次にその結果が「良し」となる(即ちそのFutureが完了する)まで現在の処理を停止する。このawait式の結果はこのFutureの完了である。

もしこのFutureが値ではなくてエラーで完了したときは、このawait式はこの実行が再開された時点でそのエラーをスローする。これは非同期コードにおける例外処理を大きく簡素化する。

もしeがFutureを返さなかったらどうなるか？この場合はひたすら待つだけである。

await式は非同期関数の中でのみ使用可能である。通常の関数の中で使おうとしたらコンパイル・エラーとなる。

以下はGoogleのdart:io担当のSøren Gjesse氏があるバグ報告の中で示しているサンプルである。このプログラムはWebSocketの知識が必要であるので、詳細は「WebSocketサーバ」の章の最初の部分を見ていただきたい。このプログラムを実行するとwebsocket.orgのエコー・サーバにWebSocket接続をし、次に'echo!'というメッセージを送信し、エコーバックされてきたメッセージをコンソールに表示している。このままだと動作を継続したままだが、実際には不要になった時に接続を終了させる必要がある。

code_17.3.dart

```
// sample of await expressions and async methods
// refer to https://github.com/dart-lang/sdk/issues/24278

import 'dart:async';
import 'dart:io';

main() async {
 // Connect to a web socket.
 WebSocket socket = await WebSocket.connect('ws://echo.websocket.org');
```

```
// Setup listening.
socket.listen((message) {
 print('message: $message');
}, onError: (error) {
 print('error: $error');
}, onDone: () {
 print('socket closed.');
}, cancelOnError: true);

// Add message
socket.add('echo!');

// Wait for the socket to close.
try {
 await socket.done;
 print('WebSocket down');
} catch (error) {
 print('WebScoket done with error $error');
 }
}
```

ここではmain()関数そのものをasyncを使って非同期関数にしているので、その中でawait式を使うことができる。WebSocket.connectメソッドはFuture<WebSocket>を返すので、thenを使わなくてもawaitを使うと記述が簡単になる。

17.4節　　Timer.run()

2013年2月のM3版からdart:asyncのTimerクラスにTimer run(void callback())というクラス・メソッドが導入されている。これは非同期で可能な限り即座にコールバック関数を実行するstaticメソッドである。これは前述のasyncより先に導入されており、asyncと同様に新規の非同期コールバック関数を作るのに有用である。

17.5節　　複数のイベント・ベースのアプリケーション

2012年2月にこのAPIに複数非同期コールバック・ベースのプログラム作成に便利なメソッドたち（chainとtransform）、及び名前付きコンストラクタ（Future.immediate）が追加されている。

順序づけられたイベント処理

Future抽象クラスのthenというメソッドが順序付けされたイベント処理の為に使われる（以前あったchainというメソッドは無くなった）。このメソッドの詳細はAPI和訳を見て頂きたい。thenはFutureオブジェクトを返すので、これを連結させると順序付けされたイベント処理が可能になる。そのための関数の一般的な形式は次のようである：

```
Future doStuff() {
   return someAsyncProcess().then((msg) => msg.result);
 }
```

ここではdoStaffというFutureを返す関数を示している。someAsyncProcess（ある非同期プロセス）が終了したら、thenでその結果に対する処理を行い、更なるチェインのためにFutureオブジェクトを返すとか、あるいはこの例のように単に値を返すとかができる。

たとえばデータ・ベースがFutureを使っていないときは、Futureオブジェクトを返すdoStaff関数はCompleterを使って次のように記述されよう：

```
 Future doStuff() {
   Completer completer = new Completer();
   runDatabaseQuery(sql, (results) {
     completer.complete(results);
   });
   return completer.future;
 }
```

画面上に3つのボタンがあって、順番にそれらのボタンのクリックを受付、その結果をもとにある処理を行うシナリオを考えてみよう。なおこのアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、Dart Editorから\dart_code_samples-master\apps\FutureSample_2のフォルダを開くと良い。

以下はこのアプリケーションのコードである：

FutureSample_2.dart

```
// Dart code sample of Future chain
// Accept button 1, button 2, button 3 and TimeConsumingWork sequentially
// Tested on Dartium
// Source : www.cresc.co.jp/tech/java/Google_Dart/DartLanguageGuide.pdf
// March 2012, by Cresc corp.
// October 2012, incorporated M1 changes
// January 2013, incorporated API changes

import 'dart:html';
import 'dart:async';

void write(String message) {
  String timestamp = new Date.now().toString();
  document.querySelector('#status').insertAdjacentHtml('beforeend', '$timestamp :
$message<br>');
}

int timeConsumingWork(int durationInMs){
  var watch = new Stopwatch();
  watch.start();
  while (watch.elapsedMilliseconds < durationInMs){}
  watch.stop();
  return watch.elapsedMilliseconds;
}

class ClickProcessWorker {
  Future<String> run(String buttonID) {
    var completer = new Completer();
    var isComplete = false;
```

```
    ButtonElement button = document.querySelector('#$buttonID');
    button.on.click.add((e){
      if (!isComplete) {
        completer.complete(buttonID);
        isComplete = true;
      }
    });
    write('$buttonID : Returned future');
    return completer.future;
  }
}

class FutureSample_2 {
  void run() {
    try {
      write("FutureSample_2 running");
      Future future = new ClickProcessWorker().run('button1');
      future.then((value){
        write('Accepted "$value" result');
        return new ClickProcessWorker().run('button2');
      })
      .then((value){
        write('Accepted "$value" result');
        return new ClickProcessWorker().run('button3');
      })
      .then((value){
        write('Accepted "$value" result');
        var completer = new Completer();
        timeConsumingWork(1000);
        completer.complete('timeConsumingWork');
        return completer.future;
      })
      .then((value){
         write('Accepted "$value" result');
         write('Done!');
         }
      );
    }
    catch (e) {
      write('Exception occured');
    }
  }
}

void main() {
  new FutureSample_2().run();
  write('End of "main"');
}
```

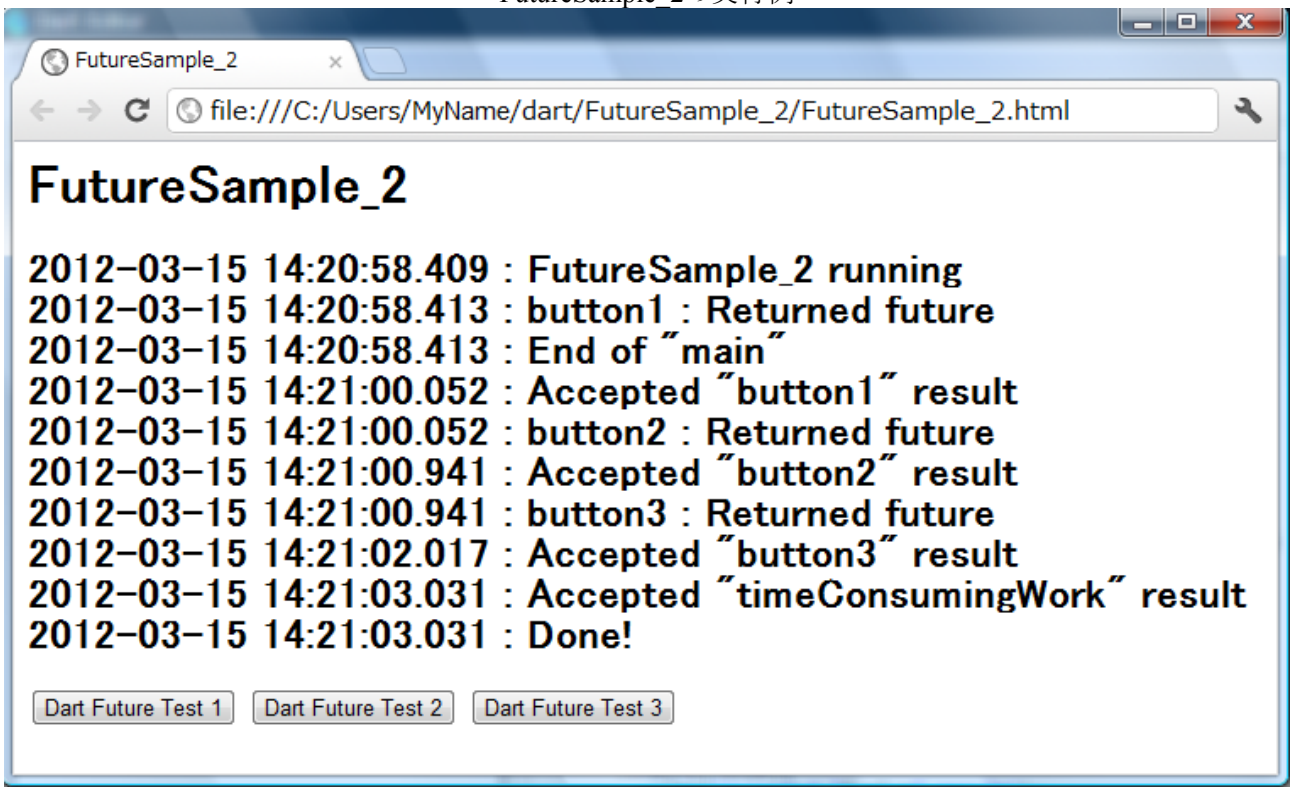
ここでは3つのボタンのクリックを受け付けるハンドラと、それを基に行う1秒間（1000ミリ秒）の処理がチェインを構成している。FutureSample_2.runメソッドがその動作を記述している。ボタン1、ボタン2、ボタン3、そして1秒間の処理が順番に終了したらDone!を表示する。

なおエラー処理に関しては、ここではFutureが用意しているエラー処理ではなく、グローバルな例外処理即ちtry-catchのメカニズムを使用している。興味のある読者は、新しいAPIが用意しているエラー処理にチャレンジし

て見られたい。

下図はその実行例である。

FutureSample_2の実行例



1. このプログラムが開始すると
2. 即座にボタン1のハンドラがセットされ、またfutureオブジェクトが返される
3. 同時にmainのメソッドは終了する
4. ユーザがボタン1をクリックするとハンドラが起動し、そのオブジェクトにbutton1という値をセットするので、処理はチェインのメソッドに移り、その値を処理する(ここではその値を受けつけたことを表示して、次のボタン2の受付を開始させる)
5. ボタン2のハンドラがセットされ、またその為のfutureオブジェクトが返される
6. ユーザがボタン2をクリックするとハンドラが起動し、そのオブジェクトにbutton2という値をセットするので、処理は次のチェインのメソッドに移り、その値を処理する(ここではその値を受けつけたことを表示して、次のボタン3の受付を開始させる)
7. ボタン3のハンドラがセットされ、またその為のfutureオブジェクトが返される
8. ユーザがボタン3をクリックするとハンドラが起動し、そのオブジェクトにbutton3という値をセットするので、処理は次のチェインのメソッドに移り、その値を処理する(ここではその値を受けつけたことを表示して、次の1秒間の時間待ちを開始させる)
9. 1秒後にその処理が終了すると、その処理は完了済み(値がセットされた)Futureオブジェクトを返す
10. thenメソッドはこのチェインの終了を受け、チェインの最後のFutureオブジェクトの値を表示するとともに、Done!を表示する

以下はHTMLのコードである:

FutureSample_2.html

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <title>FutureSample_2</title>
  </head>
  <body>
    <h1>FutureSample_2</h1>
    <button id="button1"> Dart Future Test 1 </button>
    <button id="button2"> Dart Future Test 2 </button>
    <button id="button3"> Dart Future Test 3 </button>
    <h2 id="status"></h2>
    <script type="application/dart" src="FutureSample_2.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

注意:このHTMLファイルをブラウザで読み出しこのアプリケーションを実行するには、dart.jsファイルが2013年1月からpubに移されているので、FutureSample_1のときと同じようにpackages/browser/dart.jsファイルを同じディレクトリに用意する必要がある。

イベントの並行受付

今度は3つのボタンのクリックの受付と、時間がかかる(1秒間)処理の4つを順序なしで受け付けるシナリオを考えてみよう。その為にwait(Iterable<Future> futures)というstaticなメソッド(即ちクラス・メソッド)が用意されている。futuresというのは各々の処理のFutureオブジェクトのiterableなコレクションである。

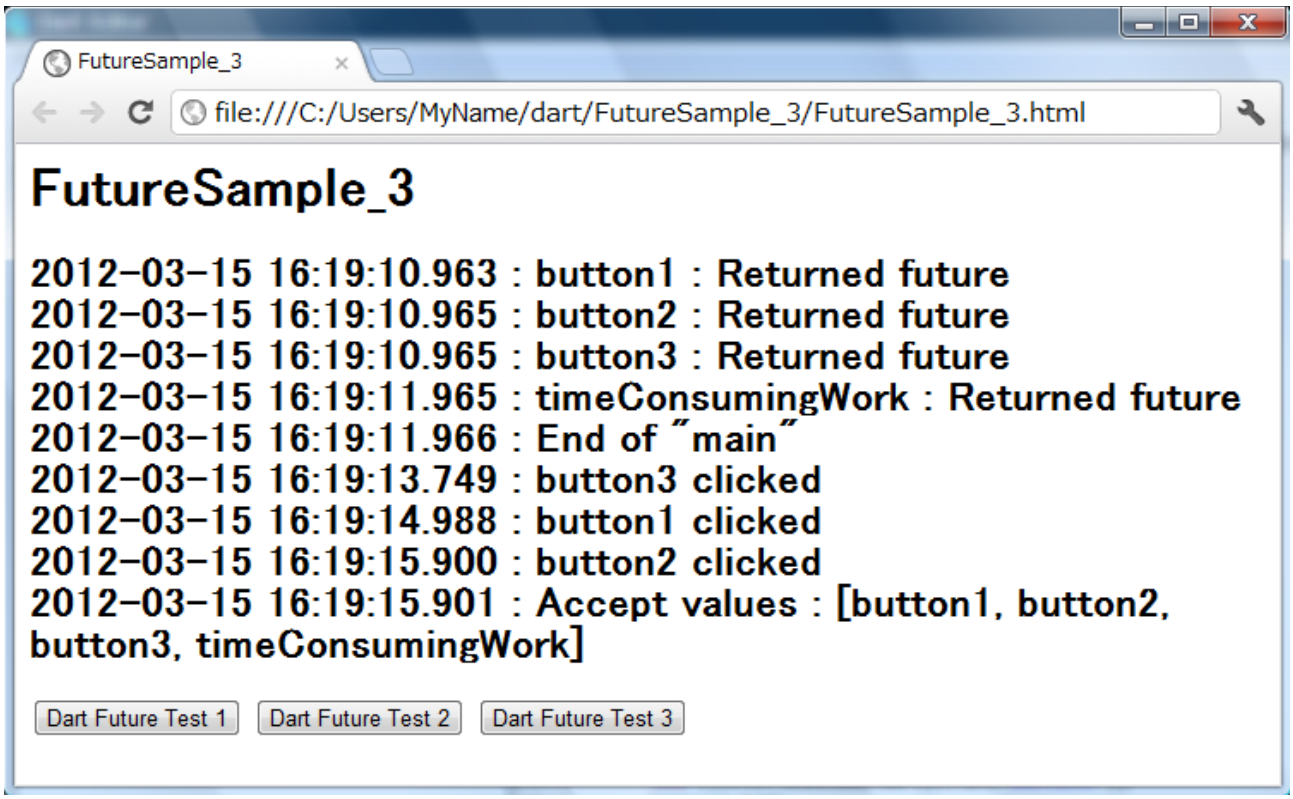
このメソッドの使い方は次のようになる:

```
  List<Future> futures = [
    new ClickProcessWorker().run('button1'),
    new ClickProcessWorker().run('button2'),
    new ClickProcessWorker().run('button3'),
    heavyWork()
  ];
  Future.wait(futures).then((values){
    write('Accept values : $values');
  });
  futures.forEach((future){
    future.catchError((exception) => write('Exception occured'));
  });
```

- futuresは各ハンドラからのFutureオブジェクトのコレクションである
- Future.wait(futures).thenはこれらのオブジェクト総てに値がセットされるのを待つ

下図はその実行例である:

FutureSample_3の実行例



1. プログラムが開始するとbutton1、button2、button3の3つのハンドラが即座にFutureオブジェクトを返してくる
2. 1秒後にtimeConsumingWorkがFutureオブジェクトを返してくる
3. この時点でmainのメソッドは終了する
4. 次にユーザがボタン3、ボタン1、及びボタン2の順でボタンをクリックしている
5. ボタン2がクリックされたことで総てのFutureオブジェクトが完了状態になって値がセットされたので、それらの値が表示されている

以下はこのサンプルのコードである。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、Dart Editorから\dart_code_samples-master\apps\FutureSample_3のフォルダを開くと良い。:

FutureSample_3.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>FutureSample_3</title>
  </head>
  <body>
    <h1>FutureSample_3</h1>
    <button id="button1"> Dart Future Test 1 </button>
    <button id="button2"> Dart Future Test 2 </button>
    <button id="button3"> Dart Future Test 3 </button>
    <h2 id="status"></h2>
    <script type="application/dart" src="FutureSample_3.dart"></script>
        <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

注意:このHTMLファイルをブラウザで読み出しこのアプリケーションを実行するには、dart.jsファイルが2013年1

月からpubに移されているので、FutureSample_1のときと同じようにpackages/browser/dart.jsファイルを同じディレクトリに用意する必要がある。

FutureSample_3.dart

```
// Dart code sample to explain Futures.wait method
// Wait for button 1, button 2, button 3 and TimeConsumingWork to complete
// Tested on Dartium
// Source : www.cresc.co.jp/tech/java/Google_Dart/DartLanguageGuide.pdf
// March 2012, by Cresc corp.
// October 2012, incorporated M1 changes
// January 2013, incorporated API changes

import 'dart:html';
import 'dart:async';

void write(String message) {
  String timestamp = new Date.now().toString();
  document.querySelector('#status').insertAdjacentHtml('beforeend', '$timestamp :
$message<br>');
}

int timeConsumingWork(int durationInMs){
  var watch = new Stopwatch();
  watch.start();
  while (watch.elapsedMilliseconds < durationInMs){}
  watch.stop();
  return watch.elapsedMilliseconds;
}

class ClickProcessWorker {
  Future<String> run(String buttonID) {
    var completer = new Completer();
    var isComplete = false;
    ButtonElement button = document.querySelector('#$buttonID');
    button.on.click.add((e){
      if (!isComplete) {
        write('$buttonID clicked');
        isComplete = true;
        completer.complete(buttonID);
      }
    });
    write('$buttonID : Returned future');
  return completer.future;
  }
}

class FutureSample_3 {
  void run() {
    List<Future> futures = [
      new ClickProcessWorker().run('button1'),
      new ClickProcessWorker().run('button2'),
      new ClickProcessWorker().run('button3'),
      heavyWork()
    ];
    Future.wait(futures).then((values){
      write('Accept values : $values');
      write('Done!');
    });
```

```
    futures.forEach((future){
      future.catchError((exception) => write('Exception occured'));
    });
  }

  Future heavyWork(){
    var completer = new Completer();
    timeConsumingWork(1000);
    completer.complete('timeConsumingWork');
    write('timeConsumingWork : Returned future');
    return completer.future;
  }
}

void main() {
  new FutureSample_3().run();
  write('End of "main"');
}
```

17.6節　　Futureにおけるエラー処理

＊＊＊

この節はDartチームに最近加わったShailen Tuli氏が書いた"Futures and Error Handling"という資料の翻訳である。FutureにはこれまでのerrorやExceptionのtry / catchの方式とは別の独自のエラー処理が組み込まれている。この資料にはその使い方が詳細に記されているので一読をお勧めする。また2014年からはZoneが利用できるようになった。これを使うとあるゾーン内での非同期エラーはそのゾーンで確実に捕捉できるようになる。詳細は「サーバの動作継続の為のZone」の節を参照されたい。

＊＊＊

Futureを受理するレシーバ関数myFuncがあったとする。そのようなレシーバはそのFutureを完了させる値またはエラーを処理するコールバック関数を次のように登録できる:

```
myFunc().then(processValue)
        .catchError(handleError);
```

登録されたこれらのコールバック関数は次の規則にしたがって呼び出される:

- then()のコールバックは値で完了したFutureで呼び出される。
- catchError()のコールバックはエラーで完了したFutureで呼び出される。

上の例ではmyFunc()のFutureがある値で完了すればthen()のコールバックが呼び出される。もしthen()の中で新しいエラーが起きなければcatchError()のコールバックは呼び出されない。一方myFunc()のFutureがあるエラーで完了すれば、then()のコールバックが呼びだされず、catchError()のコールバックが呼び出される。

then()でcatchError()を使う

then()とcatchError()をチェインで呼び出すのはFutureを扱う場合の一般的なパタンで、try-catchブロックとほぼ等価なものと考えることができる。このパタンでの幾つかの事例を次に示す。

catchError()を包括的なエラー・ハンドラとして使う

次の例はthen()内のコールバック関数内から例外をスローするもので、catchError()の持つエラー・ハンドラとしての多様性のデモでもある:

```
myFunc()
  .then((value) {
    doSomethingWith(value);
    ...
    throw("some arbitrary error");
  })
  .catchError(handleError);
```

もしmyFunc()のFutureがある値で完了したらthen()のコールバック関数が呼び出される。もしこのthen()のコールバック関数内のコードがスローすれば（上記の例のように）、thenはFutureを返す関数なのでこのFutureがエラーで完了することになる。そのエラーがcatchError()によって処理される。

無論myFunc()のFutureがあるエラーで完了すればthenのFutureはそのエラーで完了する。そのエラーもcatchError()で処理される。

そのエラーがmyFunc()のなかからかそれともthen()のなかから始まったかに関わらず、catchError()はそれをきちんと受理する。

then()内でのエラー処理

より細かなエラー処理のためには、then()内に2番目の(onError)コールバックを登録して、エラーで完了したFutureたちを処理できる。APIドキュメントには次のようにこのシグネチュアが記されている:

```
abstract Future then(onValue(T value), {onError(AsyncError asyncError)})
```

then()に渡されるエラーとthen()内で発生したエラーを区別したいときにのみオプショナルなonErrorコールバック関数を次のように登録する:

```
funcThatThrows()
  .then(successCallback, onError: (e) {
    handleError(e);          // オリジナルのエラー
    anotherFuncThatThrows(); // 新しいエラー！
  })
  .catchError(handleError);  // then()内からのエラーが処理される
```

上の例ではfuncThatThrows()のFutureのエラーがonErrorコールバック関数内で処理される。anotherFuncThatThrows()はthen()のFutureをエラーで完了させ、このエラーのほうはcatchError()で処理される。

一般的に2つの異なったエラー処理を実装するやり方は勧められない：then()内でそのエラーを捕捉しなければいけない特別な理由がある場合に限り第2のコールバック関数を登録すべきである。

長いチェインの途中で発生するエラー

then()呼び出しを続けさせ、catchError()を使ってそのチェインのなかで発生したエラーを捕捉するのは一般的である。

```
Future<String> one()   => new Future.immediate("from one");
Future<String> two()   => new Future.immediateError("error from two");
Future<String> three() => new Future.immediate("from three");
Future<String> four()  => new Future.immediate("from four");

void main() {
  one()                                    // Futureは"from one"で完了
    .then((_) => two())                    // Futureはtwo()のエラーで完了
    .then((_) => three())                  // Futureはtwo()のエラーで完了
    .then((_) => four())                   // Futureはtwo()のエラーで完了
    .then((value) => processValue(value)) // Futureはtwo()のエラーで完了
    .catchError((e) {
      print("Got error: ${e.error}");     // 最終的にコールバックが呼ばれる
      return 42;                           // Futureは42で完了
    })
    .then((value) {
      print("The value is $value");
    });
}

// Output of this program:
//   Got error: error from two
//   The value is 42
```

この場合はthen()のチェインの中のtwo()関数が即エラーで完了したFutureを返している。エラーで完了したFutureでthen()が呼び出されると、then()のおコールバック関数は呼び出されない。その代わりthen()のFutureはそのレシーバのエラーで完了する。上の例では、このことはtwo()がよばれた以降は、その後のthen()はtwo()のエラーで完了したFutureを返す。そのエラーは最終的にcatchError()で処理される。

特定のエラーの処理

特定のエラーを捕捉したいときはどうすればよいだろうか？あるいは一つ以上のエラーをどうやって捕捉したらよいだろうか？

catchError()はオプショナルなtestという名前付き引数を持っており、スローされたエラーの種類をしらべることができる。

```
abstract Future catchError(onError(AsyncError asyncError), {bool test(Object error)})
```

与えられたパラメタをもとにユーザを認証し、そのユーザを然るべきURLに振り向けるhandleAuthResponse(params)という関数を考えてみよう。その複雑な作業を考えると、handleAuthResponse()はいろんなエラーと例外を発生させ得るので、それらを個別に処理すべきである。次の例はそのためにtestが使えることを示している：

```
void main() {
```

```
  handleAuthResponse({'username': 'johncage', 'age': 92})
    .then((_) => ...)
    .catchError(handleFormatException,
                test: (e) => e is FormatException)
    .catchError(handleAuthorizationException,
                test: (e) => e is AuthorizationException);
}
```

whenComplete()を使った非同期のtry-catch-finally

then().catchError()がtry-catchと同じものと見做せば、whenComplete()はfinallyと等価と考えられる。whenComplete()内で登録されたコールバック関数は、値で完了してもエラーで完了してもwhenComplete()のレシーバが完了したときに呼び出される。

```
var server = connectToServer();
server.post(myUrl, fields: {"name": "john", "profession": "juggler"})
      .then(handleResponse)
      .catchError(handleError)
      .whenComplete(server.close);
```

上記はクライアントがあるサーバに接続してPOST要求を行うものを想定している。このserver.post()が有効な応答を取得したかあるいはエラーになったかに関わらずserver.closeを呼びたいとしよう。これはwhenComplete()のなかにそれを置くことでこれが達成される。

whenComplete()で返されたFutureを完了させる

whenComplete()内でエラーが発生しなかったときは、そのFutureはwhenComplete()が呼ばれたFutureと同じように自分のFutureを返す。これはサンプルを見ればよく理解できる。

次のコードに於いて、then()のFutureはエラーで完了し、したがってwhenComplete()のFutureもまたそのエラーで完了する:

```
void main() {
  funcThatThrows()
    .then((_) => print("Won't reach here..."))   // Futureはエラーで完了する
    .whenComplete(() => print("... or here..."))) // Futureは同じエラーで完了する
    .then((_) => print("... nor here."))         // Futureは同じエラーで完了する
    .catchError(handleError)                     // エラーはここで処理される
}
```

次のコードでは、then()のFutureがあるエラーで完了し、それが現在catchError()で処理されている。catchError()のFutureがsomeObjectで完了しているので、whenComplete()のFutureも同じオブジェクトで完了する。

```
void main() {
  funcThatThrows()
    .then((_) => ...)         // Futureがエラーで完了
    .catchError((e) {
```

```
    handleError(e);
    printErrorMessage();
    return someObject;
  })                                  // Futureはは someObjectで完了
  .whenComplete(() => print("Done!")); // Futureはは someObjectで完了
}
```

whenComplete()内で起きたエラー

もしwhenComplete()のコールバックがエラーをスローすると、whenComplete()のFutureはそのエラーで完了する。

```
void main() {
  funcThatThrows()
    .catchError(handleError)               // Futureはある値で完了する
    .whenComplete(() => throw "new error") // Futureはあるエラーで完了する
    .catchError(handleError);              // エラーが処理される
}
```

潜在的問題:エラー・ハンドラを早期登録しなかった場合

Futureが完了するよりも前にエラー・ハンドラたちがインストールされていることが必須である:これによりFutureがエラーで完了し、そのエラー・ハンドラがまだ付加されておらず、そしてそのエラーがたまたま伝搬するというシナリオが回避される。次のようなコードを考えてみよう:

```
void main() {
  Future future = funcThatThrows();

  // まずい。例外をハンドルするには遅すぎる
  new Future.delayed(const Duration(milliseconds: 500), () {
    future.then(...)
          .catchError(...);
  });
}
```

上のコードでは、スローを起こすfuncThatThrows()が呼ばれた0.5秒以降で無いとcatchError()が登録されず、そのエラーは処理されないことになってしまう。

もしfuncThatThrows()がFuture.delayed()コールバック内で呼ばれれば、この問題は無くなる。

```
void main() {
  new Future.delayed(const Duration(milliseconds: 500), () {
    funcThatThrows().then(processValue)
                    .catchError(handleError)); // We get here.
  });
}
```

潜在的問題：同期と非同期のエラーの偶発的混在

Futureを返す関数は大抵自分たちのエラーを将来発生する。そのような関数を呼び出す側が複数のエラー処理のシナリオを組み入れることを我々は望まないので、同期エラーが外部にリークするのを防止しなければならない。次のコードを考えてみよう：

```
Future<int> parseAndRead(data) {
  var filename = obtainFileName(data);          // スローする可能性あり
  File file = new File(filename);
  return file.readAsString().then((contents) {
    return parseFileData(contents);             // スローする可能性あり
  });
}
```

ファイル名を取得するobtainFileName()とファイル・データを構文解析するparseFileData()という2つの関数が同期的にスローする可能性がある。parseFileData()が非同期のthen()コールバック内で実行されるので、そのエラーはこの関数からリークするしない。その代りthen()のFutureはparseFileData()のエラーで完了し、そのエラーは最終的にparseAndRead()のFutureを完了させ、そのエラーはcatchError()によってきちんと処理されることになる。

しかしながらobtainFileName()はthen()コールバック内で呼ばれていないので、もしこの関数がスローしたら、同期エラーが次のように伝搬してしまう：

```
void main() {
  parseAndRead(data).catchError((e) {
    print("inside catchError");
    print(e.error);
  });
}

// プログラムの出力:
//   Unhandled exception:
//   <error from obtainFileName>
//   ...
```

catchError()を使ってもこのエラーは捕捉されないので、parseAndRead()のクライアントはこのエラーのための別のエラー処理を組み込むことになる。

解決策：自分のコードをFuture.of()を使ってラップする

ある関数から偶発的に同期エラーがスローされないようにする一般的なパタンはその関数ボディをnew Future.of()コールバック内にラップすることである：

```
Future<int> parseAndRead(data) {
  return new Future.of(() {
    var filename = obtainFileName(data);          // スローする可能性あり
    File file = new File(filename);
    return file.readAsString().then((contents) {
      return parseFileData(contents);             // スローする可能性あり
    });
  });
}
```

もしこのコールバックは非Futureの値を返すと、Future.of()のFutureはその値で完了する。もしこのコールバックがスローすると（上の例で示したように）、このFutureはエラーで完了する。もしこのコールバック自身がFutureを返せば、そのFutureの値またはエラーがFuture.of()のFutureを完了させる。

コードをFuture.of()内にラップすることで、catchError()は総てのエラーを捕捉できる：

```
void main() {
  parseAndRead(data).catchError((e) {
    print("inside catchError");
    print(e.error);
  });
}

// プログラム出力
//   inside catchError
//   <error from obtainFileName>
```

Future.of()により捕捉されない例外(uncaught exceptions)に対する耐性があがる。多くのコードがある関数に入っているような場合、気が付くこと無く危険を冒している可能性がある：

```
Future myFunc() {
  return new Future.of(() {
    var x = someFunc();     // 極めてまれに予期せぬスローが起きる
    var y = 10 / x;         // xはゼロであってはいけない
    ...
  });
}
```

Future.of()は自分が起きるかもしれないと判っているエラーを処理できるようにするだけでなく、その関数からエラーが偶発的にリークしてしまうのを防止できる。

タイムアウト監視

Futureベースのアプリケーションでは、タイムアウト監視が必要な場合が多い。Dartのディスカッション・ルームでそのようなサンプルに関する質問にDartチームのBob Nystromが自分がPubで使っているタイマを紹介している。

```
/// Wraps [input] to provide a timeout. If [input] completes before
/// [milliseconds] have passed, then the return value completes in the same way.
/// However, if [milliseconds] pass before [input] has completed, it completes
/// with a [TimeoutException] with [description] (which should be a fragment
/// describing the action that timed out).
///
/// Note that timing out will not cancel the asynchronous operation behind
/// [input].
Future timeout(Future input, int milliseconds, String description) {
  var completer = new Completer();
  var timer = new Timer(new Duration(milliseconds: milliseconds), () {
    completer.completeError(new TimeoutException(
        'Timed out while $description.'));
  });
  input.then((value) {
```

```
    if (completer.isCompleted) return;
    timer.cancel();
    completer.complete(value);
  }).catchError((e) {
    if (completer.isCompleted) return;
    timer.cancel();
    completer.completeError(e);
  });
  return completer.future;
}
```

これは[input]というFutureオブジェクトにタイムアウトを付加するラッパ関数である。[input]が[milliseconds]経過前に完了すれば、返される値はこれまでと同じように完了する。しかしながら[input]が完了する前に[milliseconds]経過したときは、[description]つきの[TimeoutException]で完了する。このタイムアウトでは[input]の背後で行われている非同期操作をキャンセルしていないことに注意が必要である。TimeoutExceptionはユーザがExceptionを実装して用意する。

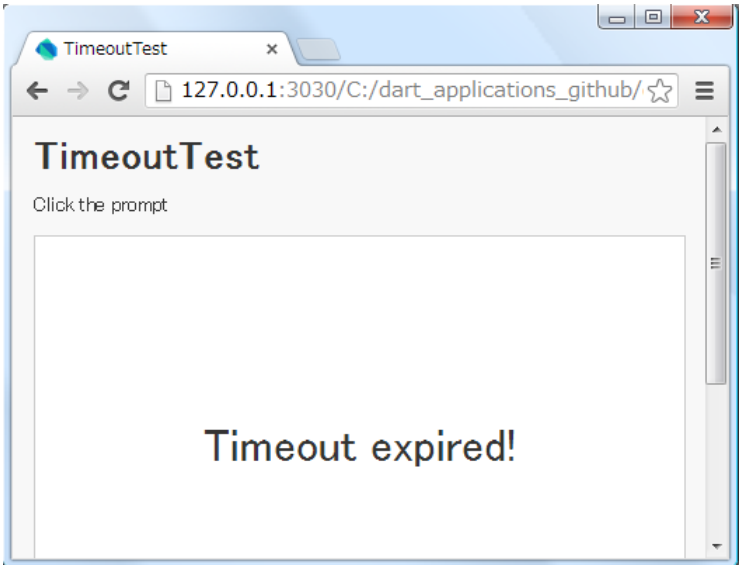
具体的な例はTimeoutTestというアプリケーションを見て頂きたい。のアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、Dart Editorから\dart_code_samples-master\apps\TimeoutTestのフォルダを開くと良い。

このアプリケーションをDartiumで実行すると次のような画面が表示される:

初期画面

Click me within 5 seconds!のテキストを5秒以内にクリックすれば、この文字が反転表示される。5秒以上クリックしないでおくと、下図のようにタイムアウトが発生したことを知らせる:

タイムアウト時の画面



このプログラム主要な箇所を以下に示す:

```
doRepeat() {
  Future future = new ClickProcessWorker().run();
  timeout(future, timerValue, "prompting click")
  .then( (result) {
    reverseText();
    doRepeat();
    },
    onError: (err){
    query("#prompt_text_id").text = "Timeout expired!";
 });
}

Future timeout(Future input, int milliseconds, String description) {
  var completer = new Completer();
  var timer = new Timer(new Duration(milliseconds: milliseconds), () {
    completer.completeError(new TimeoutException(
        'Timed out while $description.'));
  });
  input.then((value) {
    if (completer.isCompleted) return;
    timer.cancel();
    completer.complete(value);
  }).catchError((e) {
    if (completer.isCompleted) return;
    timer.cancel();
    completer.completeError(e);
  });
  return completer.future;
}

class TimeoutException implements Exception{
    const TimeoutException([String this.message = ""]);
    String toString() => "TimeoutException: $message";
    final String message;
}
```

doRepeatという関数はクリックされたらこのテキストを反転し再度クリックを受け付けるという関数である。timeoutという関数は既に紹介した。TimeoutExceptionは新しく用意した例外のクラスで、Exceptionを実装している。これは例外のクラスを自分で用意するときの基本的なパタンである。

なお2013年12月にFuture.timeout及びStream.timeoutというメソッドが追加されている。これを使えば上記のコードがよりシンプルになる。

17.7節　　ストリーム (Streams)

Streamは2012年11月末にRFC(コメント募集)としてその構想が発表された。Streamはシーケンシャルに発生するイベント(データまたはエラー)たちを処理するものである。そのストリームが空になったときには更に「完了(done)」という単発のイベントを送信できる。

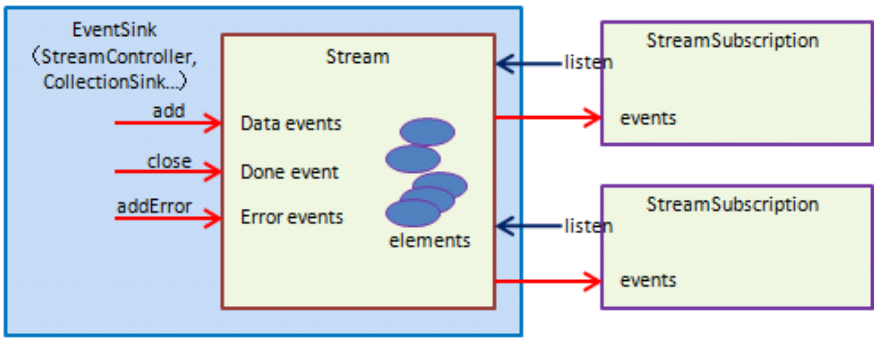
対象となるイベントの例は:

- ファイルからのバイトたち
- WebSocketからのメッセージたち
- ボタン・クリックたち
- DOMからのイベントたち
- HTTPサーバからの到来要求のチャンクたち

などである。

そして2013年1月からdart:asyncという新しいライブラリにStreamという抽象クラス及びそれに関連した多くの抽象クラスやクラスが追加された。更にHTMLやIO関係の多くのインターフェイスが現在Streamを採用している。追加された基本となる抽象クラスたちの中でも、StreamControllerは他の受信者たちにそのオブジェクトが内蔵しているstreamオブジェクト上でデータ、エラー、完了のイベントを送信できる便利なクラスである。単発イベントを扱うFutureの場合は完了という概念は無く、そのような情報はデータに含めて知らせるしかできなかったが、Streamの場合は、データ(data)、エラー(error)に加えて完了(done)のイベントを別々に発生する。

基本的な概念を下図に示す:



Streamの基本的コンセプト

- シーケンシャルに発生するしたイベント(データ、完了、またはエラー)を取り扱う。
- Streamのなかではイベントは要素(element)の集まりとして待ち行列に置かれている。
- Streamからのイベントの受信者はStreamSubscriptionを実装する。

- 受信者は単一の場合と複数受信者（放送：broadcast）の場合がある。
- 受信者はlistenメソッドで該ストリームからのイベントを受理するコールバック関数を用意し、またStreamSubscriptionのオブジェクトを取得する。このコールバック関数で受理したデータ、エラー、および終了のイベントを非同期で処理する。

ここでは簡単にその概要を説明するが、詳細は次の「Stream抽象クラス」の節で解説する。

Streamは連続したイベント源である。Streamのなかではイベントは要素(element)の集まりとして待ち行列に置かれている。そのイベントたちを受け取る側（受信者）はStreamSubscriptionであり、このオブジェクトはStreamのlistenメソッドを呼ぶことで取得される。受信者は単一の場合と複数受信者（放送：broadcast）の場合がある。通常はStreamにある要素たちは順番に受信者に送信されるが、各種の加工やフィルタリングも可能である。このようにIterableがプルで要素たちを受け取るのではなくて、プッシュ形式で要素たちを受信者たちに渡すメカニズムがStreamだともいえよう。現にIterableとStreamを比較すると、読者は同じ属性たちが存在することに気がつくだろう。

一方Streamにデータ、エラー、あるいは完了のイベントを渡す為にEventSink（StreamSinkは廃止対象化された）という抽象クラスがあり、それを継承したクラスたち（CollectionSink<T>, EventSinkView, IsolateSink, StreamController<T>, StreamSinkView<T>）のひとつとしてStreamControllerというクラスが用意されている。このクラスはStreamのオブジェクトをフィールドとして持っている。EventSinkのadd、close、及びaddErrorメソッドを呼ぶことで、内蔵しているStreamにこれらのイベントを渡すことができる。

最初にstackoverflowでKai Sellgren氏が示した最も簡単なサンプルを紹介する：

```
import 'dart:async';
import 'dart:io';

class Application {
  Stream onExit;

  Application() {
    // ストリーム・コントローラを生成し、そのストリームを"onExit"に代入する
    var controller = new StreamController();
    onExit = controller.stream;

    // 我々のストリームを使用する何らかのクラスを生成する
    new UserOfStream(this);

    // 我々がこのアプリケーションを終わるときは何時もそれを聴いている誰もに最初にそれを通知する
    controller.add('we are shutting down!');
    exit(0);
  }
}

class UserOfStream {
  UserOfStream(app) {
    app.onExit.listen((String message) => print(message));
  }
}

main() => new Application();
```

これはあるアプリケーション(Application)が終了したときに、そのアプリケーションのストリームを使っているユーザ(UserOfStream)にそれを通知するという例である。そのユーザはapp.onExitというストリームを聴いて（即ち受信者となって）、データを受信したらそれをコンソールに出力する。

このようにユーザのアプリケーションが簡単にイベントを発生できる。

より一般化して、あるクラスの属性の変更と終了をイベントとして扱うサンプルがDartのディスカッション及びpubにあるので、興味のある読者は試して頂きたい。

Stream抽象クラス

前記のようにこのクラスは非同期のイベントたち（メッセージ、ファイル・データ、マウス・クリックなど）の源を表現する。これはmx.events.EventDispatcherなどの影響を受けたものだろう。

Streamはイベントたちのシーケンスを発生させる。即ちStreamは繰り返し発生するイベントを取り扱うオブジェクトである。各イベントはデータ・イベントまたはエラー・イベントのどちらかであり、ある単一の計算の結果を表現する。IOライブラリを使うアプリケーションではそれらのイベントは主としてデータ・イベントでバイト配列として表現されるデータとなる。アイソレートを使うアプリケーションではList、Map、String、数値、及びブール値などがデータとなる。このStreamが空になったときは、このStreamは単一の"done”即ち完了のイベントを送信できる。

あるストリームが送信するイベントを受信するにはそのストリームをlistenメソッドを使ってリスンする（聴く）ことができる。リスンするときはそのストリームからのイベントを聴くのを止め(stop)たり一時的に止める(pause)するのに使えるStreamSubscriptionオブジェクトを受け取る。

あるイベントが起きたら（fireしたら）、その時点に存在するリスナたちがその通知を受ける。あるイベントが発生している最中にあるリスナが追加または外されたときは、その変更はそのイベントが完全に発生し終わってからのみ有効となる。

Streamたちは常に"pause"要求を尊重する。もし必要なら、これらのStreamたちはその入力をバッファリングする必要があるが、しばしば、そしてより好ましい手段としては、これらのStreamたちは単に自分たちの入力を同じくポーズするよう要求できる。

Streamたちは通常の「単一受信("single-subscription")」ストリームと「放送("broadcast")」の2種類がある。

単一受信ストリームではある時点でにはただひとつのリスナが許される。このストリームはリスナを取得するまでイベントを抑止し、そのリスナが受信取り消ししたときはたとえそのストリームが完了していなくてもそれ自身を空にすることができる。単一受信ストリームは一般的にはファイルI/Oのような連結したデータの部分たちをストリームするのに使われる。

放送ストリームでは任意の数のリスナたちが許され、放送ストリームたちはリスナたちが存在するかどうかに関わらずそに準備が出来ていればそのイベントたちを通知(fire)する。放送ストリームは独立したイベント/オブザーバたちの為に使われる。

isBroadcastのデフォルト実装はfalseを返す。Streamを継承している放送ストリームはtrueを返すようisBroadcastをオーバライドしなければならない。

Streamはまた、部分的にデータを取り出す(first, take...)、条件に一致するかどうかをチェックする(contains, any,

every)、及び変換する(ListからStringへ)といった一連のメソッドを取りそろえている。

dart:htmlにおけるストリーム

この新しいStream/Subscriptionのデザイン・パタンに積極的なDartチームは、2013年1月時点において、HTMLページにおけるイベント(所謂DOMイベント)を、これまでのaddEventListener及びremoveEventListenerのアプローチから、ストリーム(DOM Event Streams)として扱うようAPIの切り替えの為の作業を実施した。

Dart:htmlライブラリには現在EventStreamProvider<T extends Event>クラスが用意されている。イベントのリスンはこれまでの

```
element.on.click.add((e) {
});
```

から

```
element.onClick.listen((e) {
});
```

へ変更になった。onClickはElementのStream<MouseEvent>型のフィールドである。Elementには現在Stream型のonKeyPressのようなonXxxxイベントが多数定義されている。

またイベントの捕捉は

```
element.on.click.add((e) {
}, true);
```

から

```
Element.clickEvent.forTarget(element, useCapture:true).listen((e) {
});
```

へと変更になった。

更にメディアのElementのようにイベントが継続的に発生するような場合は、

```
document.body.$dom_addEventListener('canPlay', (e) {}, false);
```

から

```
MediaElement.canPlayEvent.forTarget(document.body).listen((e) {
});
```

と変更になった。

Element.clickEventやMediaElement.canPlayEventはEventStreamProvider<T>の型である。EventStreamProviderはDOMイベントをストリームとして提供する為のクラスであり、forTargetというメソッドのみを含んでいる。

Stream<MouseEvent>.listenなどのlistenメソッドはStreamSubscription型のオブジェクトを返す。このオブジェクト

はlistenでイベントを待機しているのをキャンセルしたり、イベント待機を何かの条件が揃うまで待つ為に使用できる。例えば:

```
StreamSubscription subscription = query('#button').onClick.listen((e) =>
handleTheClick());
// ...
// ボタンのクリックを待つ必要が無くなったら
subscription.cancel();
```

のような使い方になった。

dart:ioにおけるストリーム

2013年2月にFutureとStreamを積極的に取り込むことを目的にしたこのライブラリの大規模な見直しが行われた。Dart:io開発担当者たちは「dart:io v2」とこれを呼んでいる。IOの場合はイベントよりはデータ列の受け渡しが中心になる。
IO関係でStreamを実装しているのは以下のクラスたちである:

クラス/抽象クラス	実装
HttpClientResponse	Stream<List<int>>
HttpRequest	Stream<List<int>>
HttpServer	Stream<HttpRequest>
RawSecureServerSocket	Stream<RawSecureSocket>
RawServerSocket	Stream<RawSocket>
RawSocket	Stream<RawSocketEvent>
SecureServerSocket	Stream<SecureSocket>
ServerSocket	Stream<Socket>
Socket	IOSink<Socket> Stream<List<int>>
WebSocket	Stream<Event>

たとえばデータを読み出すときには:

```
Stream<List<int>> stream = ...
stream.listen(
  (data) { /* Process data. */ },
  onDone: () { /* All data received. */ },
  onError: (e) { /* Error on input. */ });
```

といった記述になる。

またデータを書き込む為にはStreamConsumer<List<int>, T>を実装したIOSinkが用意されている。

クラス/抽象クラス	実装

HttpClientRequest	IOSink<HttpClientRequest>
HttpResponse	IOSink<HttpResponse>
Socket	IOSink<Socket> Stream<List<int>>
WebSocket	Stream<Event> IOSink<List<int>>

IOSinkにデータを書き込むときには:

```
IOSink sink = …
sink.add([72, 101, 108, 108, 111]);  // Hello
sink.addString(", world!");
sink.close();
```

といった記述になる。

これらの使い方はHTTPサーバ関連は「HTTPサーバ」の章を、WebSocket関連は「WebSocketサーバ」の章を見て頂きたい。

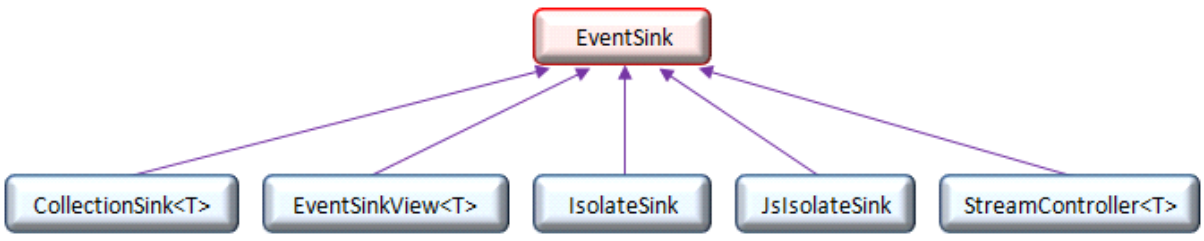
17.8節　　ストリームへのイベントやデータの書き込み

ストリームへのイベントやデータの書き込みを抽象化したものがSinkである。この名前に対しては、ネーティブな技術者たちの間でもやはり異論がが多かった。いずれにしてもデータの書き込みにはIOSinkが、イベント中心の書き込みにはEventSinkというインターフェイスが用意されている。それ以外にも下記のようにIterableなオブジェクトをもとに直接Streamを作ることも可能である:

```
var data = [1,2,3,4,5]; // サンプル・データ
var stream = new Stream.fromIterable(data);  // ストリームの生成
```

イベントとデータの為のインターフェイスの構成を下図に示す。

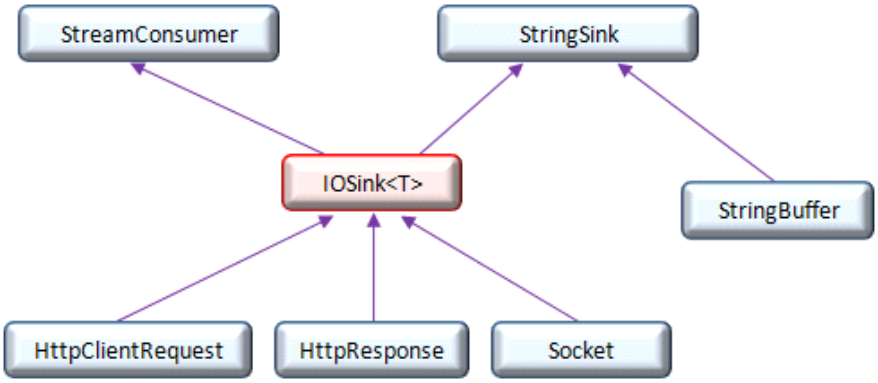
イベント中心の書き込みに対しては既に「ストリーム(Streams)」の節で説明してあるが、最初にイベント中心の書き込みの為のSinkの構成を示す。



- CollectionSink<T> ： データ・イベントをCollectionとしてストアし、Streamに送るクラス
- EventSinkView ： EventSink抽象クラスのラッパ・クラス
- IsolateSink ： MesssageBoxでストアした他のアイソレートからのメッセージをIsolateStreamに送るクラス
- StreamController ： Streamを制御するとともにイベントをそのストリームに送るクラス

- StreamSinkView ： StreamSink抽象クラスのラッパ・クラス

一方データ中心のSinkに関してはdart:ioライブラリのグループがまとめているので、メソッド等で差が出てくる。EventSinkではaddというメソッドが使われるのに対し、IOSinkではwriteというメソッドが使われる。



この図で見るとHTTPやソケットなどネットワークが中心になっているように見えるが、FileにはIOSink<File>のオブジェクトを作る

```
abstract IOSink<File> openWrite({FileMode mode: FileMode.WRITE, Encoding
encoding: Encoding.UTF_8})
```

というメソッドが存在する。これは当該ファイルの為の新しい独立したIOSinkを作るもので、文字列を対象としている。このIOSinkはシステムのリソースを解放する為にもう使わなくなったときに必ずクローズしなければならない。

ファイルの為のIOSinkには2つのモードで開くことができる：

- FileMode.WRITE：このファイルを長さゼロまで切り詰める
- FileMode.APPEND：初期書き込み位置をこのファイルの最後にセットする

返されたIOSinkを介して文字列を書き込むときは、encodingを使って指定されたエンコーディングが使われる。返されたIOSinkはこのIOSinkが生成された後で変更可能なencodingという属性を持つ。

なおStremConsumerはStream.pipeのターゲットとなるオブジェクトなので、このIOSinkに別のストリームからパイプで書き込むことも可能ではあるが、文字列を扱う限りその必要性はなかろう。

以下はopenWriteを使ったファイルへのデータ書き込みのサンプルである。このコードではある文字列をIOSinkに書き込むことで、文字列をUTF-8エンコードされたバイト列としてファイルに書き込んでいる。

Code_17.8.dart

```
import 'dart:io';
import 'dart:async';
final fileName = 'testData.txt';
final String testData = '''安倍晋三首相は２６日午前、政権発足から３カ月を迎えたことについて、
「今までと同じように結果を出していくことに全力を尽くしていきたい」と述べた。''';

main() {
  // create a file to write
  File file = new File(fileName);
  // get IOSink to write data
  IOSink fileSink = file.openWrite(mode: FileMode.WRITE, encoding: Encoding.UTF_8);
  // and write the data using StringSink\write
```

```
  fileSink.write(testData);
  print('Done writing $file');
  fileSink.close();
  // read back the file
  file = new File(fileName);
  file.open(mode: FileMode.READ)
    .then((f){
      file.readAsString()
        .then((data) {
           print('Readbacked data : $data');
           f.close();
        })
        .catchError((err, stackTrace){
          print('Read error : $err');
          print('Stack Trace : $stackTrace');
          f.close();
        });
    })
    .catchError((err) {
      print('Open error : $err');
    });
}
```

手順は:

- file.openWrite()メソッドでIOSinkオブジェクトを取得する。デフォルトでは書き込みモードで、UTF-8エンコードによるバイト列で当該ファイルに書き込む。
- fileSink.write(testData)でこのFileSinkオブジェクトにStringオブジェクトのtestDataを書くことで、このファイルにデータが書き込まれる。

と簡単である。

このプログラムの後半で書き込んだファイルを読み戻し出力している。この部分はテキスト・ファイルの読み出しの標準的なコードといえよう。この場合にはFutureのcatchErrorメソッドを使って、ファイル読み出し中のエラーと、ファイルを開いたときに発生するエラーを分けて処理していることに注意のこと。

注意： IntelliJのコンソール出力が日本語は文字化けを起こすことがある。そのときは設定を変更する必要がある。正しくファイルに書かれているかは、IntelliJでtestData.txtを開くと良い。

17.9節　　ストリームのデータの操作

ストリームからデータまたはイベントを受信するにはlistenというメソッドを呼び出してStreamSubscriptionオブジェクトを取得することは「ストリーム(Streams)」の節で総て述べた。

しかしながらStreamには単にStreamSubscriptionオブジェクトにデータ(onData)、エラー(onError)、及び完了(onDone)のイベントを渡すだけでなく、多様な操作ができるよう設計されているので、ここではやや詳しくこれを紹介する。

属性としてのイベントの取得

Streamにはfirst、last、及びsingle（ただひとつの要素があるときのみ、それ以外はエラーになる）というこのストリームの中の要素を返す属性が用意されている。これらの属性を読みだすということは暗黙的に受信が付加されたということになる。従ってこれらの属性を続けて読みだそうとするとStateError (Stream already has subscriber.)が発生する。従ってあらかじめBroadcastStreamに変換しておく必要がある。

次の例では4つのStreamSubscriptionが付加したと思えば良い:

code_17.7a.dart

```
import 'dart:async';
final testData = ['a', 'b', 'c', 'd', 'e'];

main() {
  var stream = new Stream.fromIterable(testData);
  var broadcastStream = stream.asBroadcastStream();
  broadcastStream.single.then((value) => print('stream.single: $value'))
    .catchError((Error err){print('Async error : $err');});  // Bad state
  broadcastStream.first.then((value) => print('stream.first: $value'));     // a
  broadcastStream.last.then((value) => print('stream.last: $value'));       // e
  broadcastStream.isEmpty.then((value) => print('stream.isEmpty: $value')); // false
  broadcastStream.length.then((value) => print('stream.length: $value'));   // 5
  broadcastStream.listen( (value) {
    print('Received: ${value}'); // onData handler
  });
}
```

注意しなければならないのは、**これらのprint出力はプログラム・コードの順番になるとは保証されない**ことである。イベントに対するコールバックが複数存在しているときは、それはVM内のスケジューラに依存する。

ストリーム変換機能

Streamにはストリームからのデータに何らかの加工をして新たなストリームとして受信側に渡す機能（transformメソッド）が用意されている。その典型的なのがHTTPサーバの受信した要求のストリームをWebSocketの要求に変換するもWebSocketTransformerである。これはWebSocketプロトコルがHTTPプロトコルにインフラストラクチャに乗り易いよう設定されている為でもある。詳細は「基本的なWebSocketサーバ」の節を見て頂きたい。またこのインターフェイスのサブクラスにはネットワーク上のバイト列とStringとの変換器であるStringEndoderとStringDecoderなども存在している。

変換をするには変換器(StreamTransformer)を用意する。これのコンストラクタのシグネチュアは次のようになっている:

```
factory StreamTransformer({void handleData(S data, EventSink<T> sink), void
handleError(AsyncError error, EventSink<T> sink), void handleDone(EventSink<T> sink)})
```

データのハンドラであるhandleDataはストリームからのデータとEventSinkが渡される。それ以外にもエラーのハンドラ(handleError)と完了のハンドラ(handleDone)も付加できる。

具体的な使い方を次に示す:

code_17.7b.dart

```
import 'dart:async';
import 'dart:math';
main() {
  var data = [1, 'a', 3, 4, 5, 'end', 6];
  var stream = new Stream.fromIterable(data);

  // define a stream transformer
  var transformer = new StreamTransformer.fromHandlers(handleData: (value, sink) {
    if (value is num)
      // create new values from the original value
      sink.add('√$value =  ${sqrt(value)}');
    // complete the stream if value is 'end'
    else if (value == 'end') sink.close();
    // trigger error for the illegal value
    else sink.addError(new Error('$value is not a number'));
    },
  handleDone: (sink) => sink.close(),
  handleError: (err, stackTrace, sink) => sink.addError(err)
  );

  // transform the stream and listen to its output
  stream.transform(transformer).listen((value) => print("$value"),
  onError:(err) => print('$err${err.stackTrace}'),
  onDone:() => print('Done!')
  );
}
```

このプログラムはListにあるオブジェクトたちをイベントとして送出するStreamに対し、その要素に応じて

- numのオブジェクトであれば√に変換する
- 'end'という文字列の場合はその変換器を終了させる
- それ以外のオブジェクトに対してはエラーのイベントを発生する

というものである。

このコードを見れば、このプログラムが容易に理解できよう。このまま実行すると次のような出力が得られる:

```
√1 =  1.0
AsyncError: 'a is not a number'
null
√3 =  1.7320508075688772
√4 =  2.0
√5 =  2.23606797749979
Done!
```

Streamの中身を調べる(any, every and contains)

Streamにはその要素たちがある条件を満たしているかどうかを検査する為のメソッド(いずれもbool値を返す)が用意されている。

- Future<bool> **any**(bool test(T element)) ：条件を満たすものがひとつでも存在するか
- Future<bool> **contains**(T match) ：一致するものが含まれているかどうか
- Future<bool> **every**(bool test(T element)) ：総てが条件を満たしているか

Streamの要素たちの一部を取り出す

Streamにはその要素たちの中からある条件を満たすものだけを受信する為の多くのメソッドが用意されている。例えば

- Stream<T> **skip**(int count) ：最初のcount分のイベントをスキップする
- Stream<T> **skipWhile**(bool test(T value)) ：testに一致するイベントをスキップする
- Stream<T> **take**(int count) ：最初のcount分のイベントを送出する
- Stream<T> **takeWhile**(bool test(T value)) ：testを満たす限りそのイベントを送出する。満たさない要素が見つかった時点で完了する
- Stream<T> **where**(bool test(T event)) ：testを満たすデータ・イベントのみをデータ・イベントとして送出する

これらはStreamTransformerでも実現できる。

17.10節　関連APIの和訳

dart:async

Dart:async 非同期処理関係のライブラリ	
関数	
dynamic **runZonedExperimental**(body(), {void onError(error), void onDone()})	それ自身のゾーン内でbodyを実行させる。 もしonErrorが非nullのときは、このゾーンはエラー・ゾーンだと見做される。同期または非同期のすべてのこのゾーン内の捕捉されていないエラーたちは捕捉され、このコールバックによって処理される。 onDoneは（非nullのとき）はこのゾーンでこれ以上対応するコールバックがなくなったときに呼び出される。 例: `runZonedExperimental(() {` `  new Future(() { throw "asynchronous error"; });` `}, onError: print); // これは"asynchronous error"と出力する`

	以下の例では"1", "2", "3", "4"をこの順番でプリントする。 `runZonedExperimental(() {` `  print(1);` `  new Future.value(3).then(print);` `}, onDone: () { print(4); });` `print(2);` エラーたちはエラー・ゾーンの境界をまたぐことはない。このことはあるゾーンから離れるときには直観的ではあるが、これはまたあるエラー・ゾーンに入るようなエラーにも適用される。エラー・ゾーンの境界をまたごうとするエラーたちは捕捉されていないと見做される。 `var future = new Future.value(499);` `runZonedExperimental(() {` `  future = future.then((_) { throw "error in first error-zone"; });` `  runZonedExperimental(() {` `  future = future.catchError((e) { print("Never reached!"); });` `  }, onError: (e) { print("unused error handler"); });` `}, onError: (e) { print("catches error of first error-zone."); });`
dynamic **deprecatedFutureValue**(_FutureImpl future)	
void **runAsync**(void callback())	与えられたコールバックを非同期で実行する。 この関数を介して登録されたコールバック関数たちは常にその順で実行され、他の非同期のイベント(TimerまたはDOMイベントたちのような)たちよりも前に実行されることが保証される。 警告:このメソッドを介して非同期のコールバックたちを登録することでDOMを飢餓状態に置くこともありうる。たとえば、次のプログラムはTimerコールバックに実行する機会を与えないままコールバック関数たちを実行させてしまう。 `Timer.run(() { print("executed"); });  // 決して実行されない;` `foo() {` `  asyncRun(foo);  // 他のイベントたちの頭に[foo]をスケジュールする` `}` `main() {` `  foo();` `}`
dynamic **getAttachedStackTrace**(o)	これは実験的なAPIである。 oにStackTraceを付加する。 もしオブジェクトoがスローされ、あるdart:asyncメソッドで捕捉されない場合に、StackTraceオブジェクトがそれに付加される。このオブジェクトを取得するには、getAttachedStackTraceを使用する。 もし StackTraceが添付されないときはnullを返す。

Future<T>

Future<T> abstract class

Futureは遅れを持ったある計算を表現している。これは現在未だ得られないが将来何時かの時点で得られる値またはエラーを取得するのに使われる。あるFutureの受け手は、一旦それが取得可能になったらその値またはエラーを処理する為のコールバック関数を登録できる。例えば:

```
Future<int> future = getFuture();
future.then((value) => handleValue(value))
      .catchError((error) => handleError(error));
```

あるFutureは2つの手段で終了する:即ちある値("the future succeeds":そのfutureが成功した)で、またはあるエラー("the future fails":そのfutureが失敗した)で終了する。ユーザは各ケースに対してコールバック関数をインストールできる。ペアとなるこれらのコールバック関数を登録した結果は新しいFuture("successor":後継者)となり、これは対応するコールバック関数を呼び出した結果で完了するものである。この後継者はもし呼び出されたコールバック関数がスローすればエラーで完了する。例えば:

```
Future<int> successor = future.then((int value) {
    // 該futureがある値で完了したときに呼び出される
    return 42;  // この後継者は42という値で完了する
  },
  onError: (AsyncError e) {
    // 該futureがあるエラーで完了したときに呼び出される
    if (canHandle(e)) {
      return 499;  // この後継者は499という値で完了する
    } else {
      throw e;  // この後継者はエラーeで完了する
    }
  });
```

もしあるfutureが後継者を持たないもののあるエラーで完了したときは、このfutureはこのエラー・メッセージをグローバル・エラー・ハンドラに渡す。この特別なケースを持たすことで、何もなしに廃棄されるエラーが無いようにしている。しかしながらこのことは、エラー・ハンドラが先にインストールされていて、あるfutureがエラーで完了したらすぐにそのハンドラが提示されねばならないことを意味する。以下の例はこの潜在的なバグを示している:

```
var future = getFuture();
new Timer(5, (_) {
  // このエラー・ハンドラはこのfutureが受信されて5ms後にのみ登録される
  // もしこのfutureが途中で失敗すれば、たとえこのエラーを処理するコードが以下のように
  // あったとしてもこれはこのエラーをグローバルなエラー・ハンドラに渡してしまう
  future.then((value) { useValue(value); },
              onError: (e) { handleError(e); });
});
```

一般的には2つのコールバック関数を同時に登録することは我々は推奨しておらず、代わりにひとつの引数(値のハンドラ)でthenを使い、エラーの処理にはcatchErrorを使うことを勧めている。抜けているコールバック関数(thenの為のエラー・ハンドラとcatchErrorの為の値のハンドラ)はその値/エラーを転送する("forward")よう自動的に設定される。値とエラーのハンドリングを分離した登録呼び出したちに分離すると通常より推理しやすいコードが作られる。実際これにより非同期のコードが同期コードと非常に似たものになる:

```
// 同期コード.
try {
  int value = foo();
  return bar(value);
} catch (e) {
  return 499;
}
```

等価な非同期コード、futureベース

```
Future<int> future = foo();  // foo がこれでfutureを返す
future.then((int value) => bar(value))
      .catchError((e) => 499);
```

同期コードと似て、エラー・ハンドラ（catchErrorで登録されている）が'foo'の呼び出しと'bar'の呼び出しから来た例外の為のエラーを処理している。もしエラー・ハンドラが値のハンドラと同時に登録されているときはこうはならない。

Futureたちにはひとつ以上のコールバックのペアが登録できる。各後継者たちが独立して取り扱われ、あたかもそれが唯一の後継者であるかのごとく扱われる。

サブクラス	
SubstituteFuture<T>	
staticメソッド	
Future<List> **wait**(Iterable<Future> futures)	与えられたfutureたちの総てが完了するのを待ち、それらの値を収集する。 リストの中の総てのfutureたちが一旦完了したら完了するfutureを返す。もしこのリストの中のfutureたちのどれかがエラーで完了したら、結果となるfutureもまたエラーで完了する。そうでないときは、返されるfutureの値は作られた値たちの総ての値のリストになる。
Future **forEach**(Iterable input, Future f(element))	iterableなinputオブジェクトの各要素に対し非同期操作を順番に実行する。 inputのなかの各要素に対し順番にfを実行しfが完了したことでFutureが返されたときに限り次の要素に移る。総ての要素が処理されたとき完了するFutureを返す。 返される総てのFuitureたちの値は廃棄される。エラーが発生するとこの繰り返し操作は停止し、そのエラーは返されたFutureを介してパイプされる。
コンストラクタ	
factory **Future.delayed**(int milliseconds, T value())	ある遅延の後で完了するfutureを生成する。 このfutureはmillisecondsが経過した後で完了し呼び出しているvalueの結果を持つ。もし millisecondsが0のときは、早くとも次のイベント・ループの繰り返しのなかで完了する。 もし呼び出しているvalueがスローすれば、生成されたfutureはエラーで完了する。 非同期で計算された値たちを持ったfutureたちに関してはCompleterを参照のこと。
factory **Future.immediate**(T value)	その値が次のイベント・ループのなかで取得可能なfuture。 非同期で計算された値たちを持ったfutureたちに関してはCompleterを参照のこと。

factory **Future.immediateError**(error, [Object stackTrace])	次のイベント・ループのなかでerrorで完了するfuture。 非同期で計算された値たちを持ったfutureたちに関してはCompleterを参照のこと。
factory **Future.of**(function())	呼び出し関数の結果を含むfutureを生成する。 function()の計算の結果は戻される値かスローかのいずれかである。 もし値が返されたときは、それが生成されたfutureの結果となる。 もし呼び出す関数がスローしたときは、生成されたFutureは、スローされた値と捕捉されたスタックトレースを含む非同期エラーで完了する。 然しながら、関数呼び出しの結果が既に非同期の結果であるときは、我々はそれを特別に取り扱う。 もし返された値がFutureそれ自身のときは、この生成されたfutureの完了は、返されたfutureが完了するまで待たされ、次に同じ結果で完了する。 もしスローされた値がAsyncErrorのときは、それは直接生成されたfutureの結果として使われる。
メソッド	
abstract Stream<T> **asStream**()	thisの完了の値、データまたはエラーをその受信者たちに送信するStreamを生成する。このStreamはこの完了の値の後でクローズする。
abstract Future **catchError**(onError(AsyncError asyncError), {bool test(Object error)})	このFutureが出すエラーを処理する。 新しい Future fを返す。thisが値でもって完了するときは、その値は加工されること無くfに渡される。即ち、fは同じ値でもって完了する。 thisがエラーでもって完了するときは、そのエラーの値でtestが呼び出される。もしその呼び出しがtrueを返せば、 AsyncErrorのなかにラップされたエラーでonErrorが呼び出される。onErrorの結果はthenのonErrorとまさしく同じく処理される。 もしtest呼び出しがfalseを返せば、その例外はonErrorでは取り扱われないで、加工されないでスローされ、従ってそれはfに転送される。 testがオミットされているときは、デフォルトとして常にtrueを返す関数になる。 例: `foo` `  .catchError(..., test: (e) => e is ArgumentError)` `  .catchError(..., test: (e) => e is NoSuchMethodError)` `  .then((v) { ... });` このメソッドは以下と等価である: `Future catchError(onError(AsyncError asyncError),` `                  {bool test(Object error)}) {` `  this.then((v) => v,  // Forward the value.`

	// But handle errors, if the [test] succeeds. onError: (AsyncError e) { if (test == null || test(e.error)) { return onError(e); } throw e; }); }
abstract Future **then**(onValue(T value), {onError(AsyncError asyncError)})	もしこのfutureがある値でもって完了するときは、次にこの値でonValue が呼び出される。もしこのfutureが既に完了しているときは、次にonValueの呼び出しは次のイベント・ループの繰り返しまで待たされる。 新しいFuture fを返し、そのfはonValue（thisがある値で完了するとき）またはonError（thisがあるエラーで完了するとき）呼び出しの結果でもって完了する。 もしこの呼び出されたコールバック関数が例外をスローするときは、このメソッドが返すfはそのエラーでもって完了する。スローされた値がAsyncErrorであるときは、エラーの結果としてそれは直接使われる。そうでないときは、それは最初にAsyncErrorのなかにラップされる。 呼び出されたコールバック関数が Future f2を返すときは、fとf2はチェインとなる。即ち、fはf2の完了値でもって完了する。 onErrorが指定されていないときは、それは(e) { throw e; }と等価である。即ち、これはそのエラーをfに渡す。 殆どの場合、値とエラーを単一のthen呼び出しのなかで処理するよりは、catchErrorを分離して使用する（あるテスト・パラメタで）ほうがより読みやすいコードとなる。
abstract Future<T> **whenComplete**(action())	このfutureが完了したときに呼び出される関数を登録する。 action関数はそれが値またはエラーであろうと完了ようともこのfutureが完了したときに呼び出される。 これは"finally"ブロックの非同期の等価版といえる。 この呼び出しで返されるfutureのfは、action呼び出しの中で、またはこのaction呼び出しであるFuture返されたFutureのなかでエラーが発生しない限り、thisのfutureと同じやり方で完了する。もしactionへの呼び出しがfutureを返さないときは、その戻り値は無視される。 もしaction呼び出しがスローすると、fはこのスローされたエラーで完了する。 もしaction呼び出しがFutureのf2を返すときは、fの完了はf2が完了するまで待たされる。もしf2がエラーで完了すれば、それはfの結果にもなる。 このメソッドは以下のコードと等価である: Future<T> whenComplete(action()) { this.then((v) {

	`` action(); return v }, onError: (AsyncError e) { action(); throw e; }); } ```
Future<T> **timeout**(Duration timeLimit, {dynamic onTimeout()})	timeLimitが経過したらfuture計算をタイムアウトさせる。 このfutureがその時間内で完了したらこのfutureと同じ値で完了する新規のfutureを返す。 このfutureが timeLimit経過する前に完了しないときは、代わりにonTimeoutアクションが実行され、その結果（返すかあるいはスローするかにかかわらず）が返されたfutureの結果として使われる結果として使用される。 onTimeout関数はTまたはFuture<T>を返さねばならない。 onTimeoutが指定されていないときは、タイムアウトにより返されたfutureはTimeoutExceptionで完了する。 （注：このメソッドは2013年12月に追加された）

Completer<T>

Abstract Class **Completer<T>**	
CompleterはFutureたちを生成し、それが取得可能になったときにそれらFutureたちの値を提供する。 呼び出し側たちに値たちを渡し、Futuireたちを返したいようなサービスでは、以下のようにCompleterが使える： ``` Completer completer = new Completer(); // futureオブジェクトをクライアントに送り返す... return completer.future; ... // あとで値が取得可能になったときに、completer.complete(value);を呼ぶ completer.complete(value); // そうではなくて、そのサービスが値を作り出せないときは // エラーをクライアントに送り返すことができる completer.completeError(error); ```	
コンストラクタ	
factory **Completer**()	completerを生成する。
属性	
final Future **future**	このcompleterに提供される結果を含めるfuture。
メソッド	
abstract void **complete**([T value])	指定された値たちでfutureを完了させる。 そのfutureのリスナたちには直ちにその値が知らされる。

abstract void **completeError**(Object exception, [Object stackTrace])	futureをエラーで完了させる。 あるfutureをエラーで完了させることは、ある値を作り出そうとしている際に例外がスローされたということを示す。 引数のexceptionはnullであってはならない。 exceptionがAsyncErrorのときは、これはそのfutureのリスナたちにそのエラーメッセージを直接送るのに使われ、stackTraceは無視される。 そうでない場合は、この exceptionとオプショナルである stackTraceはあるAsyncErrorのなかに組み入れられ、このfutureのリスナたちに送信される。

Stream<T>

（2013年12月時点）

Abstract Class **Stream<T>**

非同期のデータ・イベントたちの源を表現する。

Streamはイベントたちのシーケンスを提供する。各イベントはデータ・イベントまたはエラー・イベントのどちらかであり、ある単一の計算の結果を表現する。このStreamが空になったときは、このStreamは単一の"done”イベントを送信できる。

あるストリームが送信するイベントを受信するにはそのストリームをlistenメソッドでリスン（聴く）ことができる。リスンするときはそのストリームからのイベントを聴くのを止め(stop)たり一時的に止める(pause)するのに使えるStreamSubscriptionオブジェクトを受け取る。

あるイベントが起きたら（fireしたら）、その時点に存在するリスナたちがその通知を受ける。あるイベントが発生している最中にあるリスナが追加または外されたときは、その変更はそのイベントが完全に発生し終わってからのみ有効となる。

終了(done)イベントが起きたら（fireしたら）、そのイベントを受信する前に加入者(subscribers)たちは非加入化(unsubscribed)される。このイベントが送信されたあとではこのストリームは加入者を持たなくなる。この時点で新たン加入者の追加は許されるが、その加入者は単になるべく早く新たな”done”イベントを受信するだけである。

Streamたちは常に"pause"要求を尊重する。もし必要なら、これらのStreamたちはその入力をバッファリングする必要があるが、しばしば、そしてより好ましい手段としては、これらのStreamたちは単に自分たちの入力を同じくポーズするよう要求できる。

Streamたちは通常の「単一加入("single-subscription")」ストリームと「放送("broadcast")」の2種類がある。

単一加入ストリームではある時点でにはただひとつのリスナが許される。このストリームはリスナを取得するまでイベントを抑止し、そのリスナが受信取り消したときはたとえそのストリームが完了していなくてもそれ自身を空にすることができる。

単一加入ストリームは一般的にはファイルI/Oのような連結したデータの部分たちをストリームするのに使われる。

放送ストリームでは任意の数のリスナたちが許され、放送ストリームたちはリスナたちが存在するかどうかに関わ

らずその準備が出来ていればそのイベントたちを通知(fire)する。

放送ストリームは独立したイベント/オブザーバたちの為に使われる。

もし単一加入のストリームを幾つかのリスナたちがリスンしたいときは、asBroadcastStreamを使って非放送ストリームの上にある放送ストリームを生成する。

どちらの形式のストリームでもwhereとskipのようなストリーム変換は、但し書きされていない限りこのメソッドが呼ばれたと同じ形式のストリームを返す。

あるイベントが起きたら（fireしたら）、その時点におけるリスナ（たち）がそのイベントを受信する。あるイベントが発生されている間にある放送ストリームにあるリスナが付加されたら、そのリスナは現在発生中のイベントを受信しない。あるリスナがキャンセルされたら、それは直ちにイベントの受信を停止する。

終了(done)イベントが起きたら（fireしたら）、加入者たちはそのイベントを受信する前に加入解除される(unsubscribed)。そのイベントが送信された後は、そのストリームは加入者を持たないことになる。この時点以降での放送ストリームへの新規の加入者たちの追加は許されるが、それらの加入者は単に極力早く新たな終了(done)イベントを受信するだけになる。

ストリームたちは常に"pause"要求を尊重する。もし必要なら、これらのストリームたちはその入力をバッファリングする必要があるが、しばしば、そしてより好ましい手段としては、これらのストリームたちは単に自分たちの入力を同じくポーズするよう要求できる。

デフォルト実装であるisBroadcastと asBroadcastStreamはこれが単一受信ストリームであると想定しており、Streamを継承している放送ストリームはこれらがtrueとthisを返す為にはこれらをオーバライドしなければならない。

サブクラス	
CustomStream<T>	
ElementStream<T>	
HttpClientResponse	
HttpMultipartFormData	
HttpRequest,	
HttpServer	
MimeMultipart	
RawSecureServerSocket	
RawServerSocket	
RawSocket	
ReceivePort	
ReceivePortImpl	
SecureServerSocket	
ServerSocket	
Socket	
Stdin	
StreamView<T>	
StreamZip	

WebSocket	
コンストラクタ	
new **Stream**()	新規のObjectインスタンスを生成する。 Objectインスタンスは意味ある状態を持たなく、単にその識別を介してのみ有用なものである。Objectインスタンスはそれ自身と等しい。
factory **Stream.eventTransformed**(Stream source, EventSink mapSink(EventSink<T> sink))	ある既存のストリームからの総てのイベントがあるシンク変換を介してパイプされるストリームを生成する。 指定する mapSinkクロージャはこの返されたストリームがリスンされるときに呼び出される。このsourceからの総てのイベントはこの呼び出しで返されるイベント・シンクに付加される。この変換は総ての返還されたイベントたちをmapSinkクロージャがその呼び出し中に受信したシンクに置く。コンセプト的にはこのmapSinkは、入力シンクが返されたEventSinkで出力シンクがそれが受信しているシンクであるような変換パイプを生成する。 このコンストラクタは変換器を作るのに頻繁に使われる。 データを重複させる変換器の例を示す: `class DuplicationSink implements EventSink<String> {` `  final EventSink<String> _outputSink;` `  DuplicationSink(this._outputSink);` `  void add(String data) {` `    _outputSink.add(data);` `    _outputSink.add(data);` `  }` `  void addError(e, [st]) => _outputSink(e, st);` `  void close() => _outputSink.close();` `}` `class DuplicationTransformer implements StreamTransformer<String, String> {` `  // Some generic types ommitted for brevety.` `  Stream bind(Stream stream) => new Stream<String>.eventTransform(` `      stream,` `      (EventSink sink) => new DuplicationSink(sink));` `}` `stringStream.transform(new DuplicationTransformer());`
factory **Stream.fromFuture**(Future<T> future)	該futureからの単一受信ストリームを新しく生成する。 該futureが完了したときは、それがデータであろうとエラーであろうともこのストリームはひとつのイベントを発生(fire)し、次にdoneイベントでクローズする。
factory **Stream.fromIterable**(Iterable<T> data)	dataからそのデータを取得する単一受信のストリームを生成する。 もし繰り返し中のデータがエラーをスローしたときは、このストリームはそのエラーで即座に終了する。doneイベントは送信されない（繰り返しが完了していない）が、その繰り返しは継続できなくなるのでデータ・イベントは作られれない。
factory **Stream.periodic**(Duration period, [T computation(int computationCount)])	一定時間間隔で繰り返しイベントを出すストリームを生成する。 そのイベントの値はcomputationを呼び出すことで計算される。このコールバック関数の引数は0から始まり各イベントごとに増分される整数である。

	もしcomputationが指定されていないときは、そのイベントの値はnullとなる
属性	
final Future<T> **first**	このストリームの最初の要素を返す。 最初の要素が受信されたあとはこのストリームへのリスニングを停止する。 最初のデータが受信される前にエラーが発生すれば、結果としてのfutureはそのエラーで完了する。 もしこのストリームが空のときは（最初のデータ・イベントよりも前に終了イベントが発生した）、結果としてのfutureはStateErrorで完了する。 このエラーのタイプ以外に関しては、このメソッドはthis.elementAt(0)と等価である。
final bool **isBroadcast**	該ストリームが放送ストリームであるかどうか。
final Future<bool> **isEmpty**	このストリームが要素を含んでいるかどうかを報告する。
final Future<T> **last**	このストリームの最後の要素を返す。 最初のデータが受信される前にエラーが発生すれば、結果としてのfutureはそのエラーで完了する。 もしこのストリームが空のときは（最初のデータ・イベントよりも前に終了イベントが発生した）、結果としてのfutureはStateErrorで完了する。
final Future<int> **length**	このストリームの中の要素の数を数える。
final Future<T> **single**	単一の要素を返す。 もしthisが空または1つよりも多い要素を持っているときはStateErrorをスローする。
メソッド	
Future<bool> **any**(bool test(T element))	このストリームが用意している要素のどれかをtestが受け付けるかどうかをチェックする。 答えが判ったときにこのFutureを完了させる。このストリームがエラーを報告したときは、このFutureはそのエラーで完了する。
Stream<T> **asBroadcastStream**({void onListen(StreamSubscription<T> subscription), void onCancel(StreamSubscription<T> subscription)})	thisと同じイベントたちを作り出す複数加入のストリームを返す。 もしこのストリームが既に放送ストリームのときは、それは加工されること無く返される。 もしこのストリームが単一加入のものだったときは複数の加入を許す新たなストリームを返す。その最初の加入者が付加されたときにこのストリームに加入し、このストリームが終了した、あるいはコールバックがこの加入をキャンセルするまでは加入された状態を維持する。 もしonListenが指定されているときは、それはこのストリームへのもととなっている加入を表現する加入ライクなオブジェクトで呼び出される。onListen呼び出し中にこの加入を保留、再開、あるいは取り消しをすることは可能である。 StreamSubscription.asFuture使用を含むイベント・ハンドラの変更はできない。 もしonCancelが指定されているときは、返されたストリームがリスナを持つことを止

	めたときにonListenと似たように呼び出される。もしそれが新たなリスナを得たときは、onListen関数が再度呼び出される。 たとえば、加入者がいないときにイベント損失を防ぐためにもととなっている加入を保留（ポーズ）するとき、あるいは加入者がいないときにこの加入をキャンセルするためにこのコールバックを使用する。
Stream **asyncExpand**(Function Stream convert(T event))	オリジナルのイベントあたりあるストリームのイベントたちを持った新しいストリームを生成する。 これはexpandのように機能するが、convertがIterableの代わりにStreamを返すことが異なる。返されたストリームのイベントたちが作られた順番に返されるストリームのイベントたちとなる。 convertがnullを返す時は、あたかも空のストリームが返されたごとく、出力のストリームには値がセットされない。
Stream **asyncMap**(Function convert(T event))	このストリームの各データ・イベントが新しいイベントに非同期でマップされた新しいストリームを生成する。 このメソッドはmapのように畿央駿河、convertがFutureを返せること、そしてこの場合はこのストリームはその結果で継続する前にfutureが完了するのを待つ点で異なる。 このストリームがそうであれば返されるストリームはブロードキャスト・ストリームである。
Future<bool> **contains**(T match)	このストリームが用意している要素たちのなかでmatchが発生するかどうかをチェックする。 この答えが判っているときはこのFutureを返す。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream<T> **distinct**([bool equals(T previous, T next)])	これらが以前の(previous)データ・イベントと等しいときはそのデータ・イベントをスキップする。 返されたストリームは、2つのつながった同じデータ・イベントたちのを用意しないことを除いては、このストリームと同じイベントたちを用意する。 対等性は用意されたequalsメソッドによって判断される。もしこれがオミットされているときは、最後に提供されたデータ要素に対しては'=='が適用される。
Future **drain**([futureValue])	このストリーム上の総てのデータを廃棄するが、その完了またはエラーが発生したときにはそれを通知する。 drainを使って加入すると cancelOnErrorはtrueとなる。このことはこのfutureは最初のエラーで完了し次にこの加入を取り消す事を意味する。 doneイベントの場合は、このfutureは指定したfutureValueで完了する。
Future<T> **elementAt**(int index)	このストリームの index番目のデータ・イベントの値を返す。 もしエラー・イベントが発生したときは、このfutureはこのエラーで終了する。 もしこのストリームが閉じる前にindex少ない要素よりもしか用意していないときは、

	エラーが報告される。 もしこの値が見つかる前にdoneイベントが生じたときは、このfutureはRangeErrorで完了する。
Future<bool> **every**(bool test(T element))	Testがこのストリームが用意している総ての要素を受け付けるかどうかをテストする。 この答えが判っているときはこのFutureを完了させる。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream **expand**(Iterable convert(T value))	各要素をゼロまたはそれ以上のイベントたちに変換する新しいストリームをこのストリームから生成する。 到来する各イベントは新しいイベントたちの Iterableに変換され、これらの新しいイベントたちの各々が次に返されるストリームによって順番に送信される。
Future<T> **firstWhere**(bool test(T value), {T defaultValue()})	Testにマッチするこのストリームの最初の要素を探す。 testがそれに対しtrueを返したこのストリームの最初の要素で満たされたfutureを返す。 このストリームが終了する前にそのような要素が見つからず、また defaultValue関数が与えられているときは、 defaultValue呼び出しの結果がそのfutureの値となる。 エラーが発生したとき、あるいはこのストリームがmatchを見出すことなく終了し、また defaultValue関数が指定されていないときは、このfutureはエラーを受信する。
Future **fold**(initialValue, combine(previous, T element))	combineを繰り返し適用することで値たちのシーケンスを減らす。
Future **forEach**(void action(T element))	このストリームの各データ・イベントでactionを実行する。 このストリームの総てのイベントが処理された時点で返されたfutureが完了する。もしこのストリームがエラー・イベントを有しているとき、あるいはactionがスローしたときはこのfutureはエラーで完了する。
Stream<T> **handleError**(Function onError, {bool test(error)})	このストリームからの何らかのエラーを取り上げ処理するラッパ・ストリームを生成する。 このストリームがtestをマッチするエラーを送信するときは、次にそれはhandle関数によって取り上げられる（インターセプトされる）。 onErrorコールバックはvoid onError(error)またはvoid onError(error, StackTrace stackTrace)の型式でなければならない。この関数の形式によってこのストリームはスタック・トレースの有りか無しかでonErrorを呼び出す。このスタック・トレース引数は、もしこのストリームがそれ自身スタック・トレースなしでエラーを受信したときはnullになる可能性がある。 もしtest(e)がtrueを返すときは[AsyncError] [:e:]は test関数によってマッチがとられる。もしtestがオミットされているときは、各エラーはマッチしていると見做される。 もしそのエラーが取り上げられたときは、このhandle関数はそれに対してどうするかを判断できる。新しい（または同じ）エラーを生起させたいときはスローできるし、

	あるいは単に戻ることでこのストリームがそのエラーを忘れさせることができる。 あるエラーをデータ・イベントに変換する必要があるときは、データ・イベントを出力シンクに書き込むことでそのイベントを処理するためにより一般的なStream.transformを使用のこと。
Future<String> **join**([String separator = ""])	データ・イベントたちの文字列表現の文字列を収集する。 もしseparatorが指定されているときは、それは2つの要素間に挿入される。 このストリーム内の何らかのエラーはこのfutureをエラーで完了させる。そうでないときは、"done"イベントが到着したときに収集した文字列で完了する。
Future<T> **lastWhere**(bool test(T value), {T defaultValue()})	Testにマッチするこのストリームの中の最後の要素を探す。 最後にマッチする要素が見つかることを除いてFirstMatchingと似ている。このことはその結果はこのストリームが終了するまでその結果は得られないことを意味する。
abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})	このストリームに受信を追加する。 このストリームからのデータ・イベントの各々で、受信者たちのonDataハンドラが呼び出される。もしonDataがnullのときは、何も起きない。 このストリームからのエラーに対しては、onErrorハンドラにはそのエラーを記述したオブジェクトが与えられる。 onErrorコールバックはvoid onError(error)またはvoid onError(error, StackTrace stackTrace)の型式でなければならない。この関数の形式によってこのストリームはスタック・トレースの有りか無しかでonErrorを呼び出す。このスタック・トレース引数は、もしこのストリームがそれ自身スタック・トレースなしでエラーを受信したときはnullになる可能性がある。そうでないときは単なるエラーのオブジェクトで呼び出される。 もしこのストリームが閉じたときは、onDoneハンドラが呼び出される。 もしcancelErrorがtrueのときは、この受信は最初のエラーが報告されたときに終了する。デフォルトはfalseである。
Stream **map**(convert(T event))	このストリームの各要素をconvert関数を使って新しい値に変換する新しいストリームを生成する。
Future **pipe**(StreamConsumer<dynamic, T> streamConsumer)	指定された StreamConsumerの入力としてこのストリームをバインドする。
Future **reduce**(initialValue, combine(previous, T element))	繰り返しcombineを適用することで値たちのシーケンスを減らす。
Future<T> **singleWhere**(bool test(T value))	testにマッチするこのストリーム中の単一の要素を探す。 このストリームのなかで一つ以上のマッチした要素があるときはエラーであることを除いてlastMatchと似ている。

Stream<T> **skip**(int count)	このストリームから最初のcount数のデータ・イベントをスキップする。
Stream<T> **skipWhile**(bool test(T value))	それらがtestにマッチしている間はこのストリームからのデータ・イベントをスキップする。 返されるストリームではエラーと完了のイベントに対しては何も加工されない。 testがそのイベント・データに対してtrueを返した最初のデータ・イベントから、返されるストリームはこのストリームと同じイベントたちを持つようになる。
Stream<T> **take**(int count)	このストリームの最大n個の最初の値たちを提供する。 このストリームの最初のn個のデータ・イベントと総てのエラー・イベントたちを返されるストリームに転送し、完了イベントで終了する。 もしこのストリームが終了前にcountの値よりも少ない数をつくるときは、返されるストリームもそうする。
Stream<T> **takeWhile**(bool test(T value))	testたちが成功している間はデータ・イベントたちを転送する。 返されるストリームはそのイベント・データに対してtestがtrueを返す限り同じデータ・イベントを提供する。このストリームはこのストリームが完了した、またはこのストリームがtestが受け付けない値を最初に提供したときに完了する。
Stream **timeout**(Duration timeLimit, {Function void onTimeout(EventSink sink)})	このストリームと同じイベントからなる新しいストリームを生成する。 このストリームからの2つのイベント間にtimeLimit以上が経過したときは、onTimeout関数が呼ばれる。 返されたストリームがリスンされるまではカウントダウンは開始しない。イベントがこのストリームから渡される度、あるいはこのストリームがポーズし再開される度にこのカウントダウンはリセットされる。 onTimeout関数が一つの引数（EventSink）で呼ばれる: EventSinkによりイベントたちを返されたイベントたちのなかに置くことができる。 onTimeoutが指定されていないときはタイムアウトは返されるストリームのエラー・チャンネルの中に TimeoutExceptionとして置かれる。 このストリームが放送ストリームの時は返されるストリームも放送ストリームとなる。ある放送ストリームが一回以上リスンされているときは、リスンごとにカウントを開始タイマを個々に持ち、加入たちのタイマーたちはここにポーズされ得る。
Future<List<T>> **toList**()	このストリームのデータをListに集める。
Future<Set<T>> **toSet**()	このストリームのデータをSetに集める。
Stream **transform**(StreamTransformer<T, dynamic> streamTransformer)	指定された StreamTransformerの入力としてこのストリームを連結する。 streamTransformer.bind自身の結果を返す。
Stream<T> **where**(bool test(T event))	何らかのデータ・イベントを廃棄する新しいストリームをこのストリームから生成する。 新しいストリームはこのストリームと同じエラーと完了のイベントを送信するが、test

	を満たすデータ・イベントたちのみを送信する。

StreamController<T>

Class **StreamController<T>**	
それが制御するストリームを持ったコントローラ。 このコントローラによりそのストリーム上でデータ、エラー、及び完了のイベントを送信できるようになる。このクラスは他のストリームが聴取できるシンプルなストリームを生成し、イベントをそのストリームにプッシュするのに使われる。 このストリームがポーズしているかどうか、及びそれが受信者を持っているかどうかをチェックできるとともに、それらが変更になった時にコールバック関数を取得できる。	
実装	
StreamSink<T>	
コンストラクタ	
new **StreamController**({void onPauseStateChange(), void onSubscriptionStateChange()})	単一の受信者のみに対応するストリームを持ったコントローラ。 このコントローラはその受信者が再登録されるまでは総ての到来イベントたちをバッファリングする。 onPauseStateChange関数はこのストリームがポーズ状態になったとき、またはポーズから再開するときに呼び出される。現在のポーズ状態は isPausedで読み出せる。nullのときは無視される。 onSubscriptionStateChange関数はこのストリームがその最初のリスナを受けたときまたはその最後のリスナを失ったときに呼び出される。現在の受信状態は hasSubscribersで読み出せる。nullのときは無視される。
new **StreamController.multiSubscription**()	現在このコンストラクタは未だ公開されていない。
new **StreamController.broadcast**({void onPauseStateChange(), void onSubscriptionStateChange()})	放送ストリームを持ったコントローラ。 onPauseStateChange関数はこのストリームがポーズ状態になったとき、またはポーズから再開するときに呼び出される。現在のポーズ状態は isPausedで読み出せる。nullのときは無視される。 onSubscriptionStateChange関数はこのストリームがその最初のリスナを受けたときまたはその最後のリスナを失ったときに呼び出される。現在の受信状態は hasSubscribersで読み出せる。nullのときは無視される。
属性	
final bool **hasSubscribers**	このストリーム上に現在受信者がいるかどうか。
final bool **isPaused**	ひとつまたはそれ以上のアクティブな受信者がポーズを要求したかどうか。
final StreamSink<T> **sink**	StreamSinkインターフェイスのみを曝すこのオブジェクトのビューを返す。

final _StreamImpl<T> **stream**	
メソッド	
void **add**(T value)	あるデータ・イベントを送信するかまたは待ち行列に入れる。
void **close**()	"done"（完了）メッセージを送信するかまたは待ち行列に入れる。 この完了メッセージはあるストリームによって最大一回送信されねばならず、それはまた送信される最後のメッセージでなければならない。
void **signalError**(Object error, [Object stackTrace])	エラー・イベントの並びを送信するかまたは待ち行列に入れる。 もしerrがAsyncErrorでないときは、errorとオプショナルな stackTraceがAsyncErrorに組み入れられ、このストリームのリスナたちに送信される。 そうでない場合には、もしエラーがAsyncErrorのときは、それはリスナたちに報告されるerrorオブジェクトとして直接使われ、stackTraceは無視される。 もしある受信がエラーにより受信解除であるよう要求しているときは、それはこのイベントを受信した後で受信解除される。

StreamSubscription<T>

Abstract Class **StreamSubscription<T>** あるStreamに受信する為の制御オブジェクト。 Stream.listenメソッドを使ってあるStreamに受信するとStreamSubscriptionオブジェクトが返される。このオブジェクトはあとで再度受信加除したり、このストリームのイベントたちを一時的に保留したりするのに使われる。	
メソッド	
abstract void **cancel**()	この受信をキャンセルする。その後のイベントは受信しなくなる。 あるイベントが送信中の場合は、現行のイベントを総ての受信者たちが受信し終わってからでないとこの受信解除は有効とならない。
abstract void **onData**(void handleData(T data))	この受信のデータ・イベント・ハンドラを設定またはオーバライドする。
abstract void **onDone**(void handleDone())	この受信の完了イベント・ハンドラを設定またはオーバライドする。
abstract void **onError**(void handleError(AsyncError error))	この受信のエラー・イベント・ハンドラを設定またはオーバライドする。
abstract void **pause**([Future resumeSignal])	このストリームに対してfutureで通知するまでイベントたちを保留するよう要求する。 resumeSignalのfutureが完了した時点でこのポーズを解除する。このfutureがエラーで終了したときは、これは扱われ無いことに注意。 resume呼び出しでもポーズを解除する。

	もしこの受信が1回以上ポーズしているときは、そのストリームを再開する為には同じ数のresumeが呼ばれねばならない。
abstract void **resume**()	ポーズ後に再開する。

StreamTransformer<S, T>

Abstract Class **StreamTransformer<S, T>** Stream.transform呼び出しのターゲット。 Stream.transform呼び出しは、それ自身をこのオブジェクトに渡し、次に結果としてのストリームを返す。	
サブクラス	
StreamEventTransformer <S, T>	
StringDecoder	
StringEncoder	
WebSocketTransformer	
コンストラクタ	
factory **StreamTransformer**({void handleData(S data, EventSink<T> sink), void handleError(AsyncError error, EventSink<T> sink), void handleDone(EventSink<T> sink)})	イベントたちを指定された関数たちに委譲するStreamTransformerを生成する。 これは実際のところStreamEventTransformerであり、イベント処理はfunction引数によって実行される。もし引数が指定されていなければ、これはこれに対応したStreamEventTransformerのデフォルトのメソッドたちとして動作する。 使用例: stringStream.transform(new StreamTransformer<String, String>(handleData: (Strung value, EventSink<String> sink) { sink.add(value); sink.add(value); // 到来イベントを重ねる }));
メソッド	
abstract Stream<T> **bind**(Stream<S> stream)	

EventSink<T>

Abstract Class **EventSink<T>** Streamへのイベント送信を抽象化したインターフェイス。	
サブクラス	
CollectionSink<T>	
IsolateSink	

JsIsolateSink	
StreamController<T>	
EventSinkView<T>	
メソッド	
abstract void **add**(T event)	データ・イベントを生成する
abstract void **addError**(AsyncError errorEvent)	非同期のエラーを生成する。
abstract void **close**()	ストリームに対しクローズを要求する

IOSink<T>

Abstract Class **IOSink<T>** [StreamConsumer, T>]をラップする為のヘルパー・クラスであり、StreamConsumerに直接書き込む為のユーティリティ関数たちを用意している。このIOSinkはwrite, writeAll, writeln, writeCharCode及びwriteBytesで与えられた入力をバッファリングし、このバッファがフラッシュされるまでconsumeまたはwriteStreamを遅らせる。 このIOSinkがストリーム向けのもののときは（consumeまたはwriteStreamを介して）、このIOSinkへの呼び出しはStateErrorをスローする。	
スーパークラス	
HttpClientRequest	
HttpResponse	
Socket	
実装	
StringSink	
StreamConsumer<List<int>, T>	
コンストラクタ	
factory **IOSink**(StreamConsumer<List<int>, T> target, {Encoding encoding: Encoding.UTF_8})	
属性	
final Future<T> **done**	総てのデータがこのIOSinkに書き込まれ、クローズされたときに完了するfutureを取得する。
Encoding **encoding**	
メソッド	
abstract void **close**()	該ターゲットをクローズする。
abstract Future<T> **consume**(Stream<List<int>> stream)	このIOSinkにパイプする為の機能を用意する。

abstract void **write**(Object obj)	StringSinkから継承 toStringを呼ぶことでobjをStringに変換し、その結果をthisに付加する。
abstract void **writeAll**(Iterable objects)	StringSinkから継承 与えられたobjectsで繰り返し操作をしそれらを順番に書き込む。
abstract void **writeBytes**(List<int> data)	バイト列をそのままconsumerに書き込む。
abstract void **writeCharCode**(int charCode)	StringSinkから継承 charCodeをthisに書き込む。 このメソッドはwrite(new String.fromCharCode(charCode))と等価である。
abstract void **writeln**([Object obj = ""])	StringSinkから継承 objをtoStringを呼ぶことでStringに変換し、その結果をthisに加える。次に改行を追加する。
abstract Future<T> **writeStream**(Stream<List<int>> stream)	consumeと似ているが、終了してもターゲットをクローズしない。

第18章 並行処理 (Concurrent Processing)

DartはJavaと違って単一スレッドである。しかしながら並行処理はアイソレートを使って実現できる。これは通信の世界から生まれたEarlang言語の影響を強く受けたものである。アイソレートはマルチコアやマルチCPU環境にも拡大できる。アイソレートは他のアイソレートとは資源（オブジェクト）を共有しないので、マルチ・スレッドに関わる複雑な競合問題（スレッド安全性問題）は存在しない。アイソレート間はメッセージ通信を介して通信しあい、アイソレートはその為のポートを有している。アイソレート間でオブジェクトを送信する為に、「Dartの実行」の節で述べたスナップショット機能がシリアライズの為に使われている。

JavaにおいてはJSR-121が2005年7月にjavax.isolateの最終仕様を決めているが、SE6のAPIドキュメントなどには含まれていないので一般のJavaのユーザには馴染みがないものであろう。JavaではJVMのマルチタスク化（Multi-Tasking Virtual Machine (MVM)）でこれを実現しているが、Dartでは現時点ではクライアント用ではWeb Workersが使われている。従ってDOMアクセスは出来ない。

2015年からDartチームはFletch（Dartに羽をつけるという意味か？）アイソレートとは別の並行性メカニズムの開発を開始している。これに関しては「Dart Fletch」の節でその都度その内容を紹介する予定である。

なお2012年2月末にGoogleのDartチームはIsolateに関わるAPIの変更を発表している。従って、この資料の第5版以降はこの章の内容が変更されていることに注意されたい。その改良内容は：

1. Isolateクラスをコア・ライブラリからdart:isolateライブラリに移し、Isolateというクラスは廃止対象とする
2. どのトップ・レベルの静的関数もアイソレートの開始点として有効にし、アイソレートの開始点の為のクラスを用意しなくても良くする
3. このAPIの他の部分も簡素化を考えている。例えばアイソレートを産み付けた後のポート取得の簡素化
4. URLが付与されたアイソレートの産み付けも可能

特に、アイソレートの産み付けに際しては従来はFutureオブジェクトを返していたが、この改正で送信ポートを返すようになっており、アイソレートとFuture / Completerとは切り分けがされていて、よりすっきりしたものとなっている。またアイソレートの重量/軽量といった区分もなくなっている。

新しいAPIにおける基本的なコンセプトは以下のとおりである（詳細は後述）：

アイソレートのコンセプト

1. アイソレートを使っていないプログラム・コードも一つのアイソレートである。アイソレートは子供のアイソレートを産み付け(spawn)、それを走らせる。
2. ある時点で2つのアイソレートが同じスレッドを共有することは無い。アイソレート内ではある時点ではただひとつのコールバック関数が実行される。
3. 各アイソレートは自分がアクセスできるグローバルな値を含む値をメモリにもつ。他のアイソレートが所有している値は他のアイソレートからはアクセスできない。
4. アイソレートたちが相互に通信出来る唯一の手段はメッセージ渡しのみである。
5. アイソレートはSendPortたちを使ってメッセージを送信し、それに対応したReceivePortたちを使ってメッセージを受信する。
6. メッセージの内容としては以下のものがあり得る：
    - プリミティブな値(null, num, bool, double, String)
    - SendPortのインスタンス
    - その要素が上記のもの（他のListとMapを含む）であるListまたはMap
    - 特別な状況においては任意の型のオブジェクト

7. 各アイソレートはReceivePortをひとつ持つ。
8. あるウェブ・アプリケーションがdart2jsによりJavaScriptにコンパイルされたときは、そのアイソレートはWeb Workersとして実装される。Dartium上で走る場合は、アイソレートはVMの中で走る。
9. サーバのようなスタンドアロンのVMの場合には、main()関数は最初のアイソレートの中で走る(これはルート・アイソレートとも呼ばれる)。このルート・アイソレートが終了したら、他のアイソレートがまだ走っていたとしてもVM総てを終了させてしまうことに注意。

アイソレートのAPIは2013年10月にさらに大規模変更がなされた。この内容は本資料の第22版から組み入れられている。

1. ストリーム・ベースのクラスたち(IsolateSink, IsolateStream, and MessageBox)が無くなった。その代りReceivePortがStreamを実装し、その結果"receive"メソッドは"listen"メソッドによって置き換えられた。Steamを実装したことにより、メッセージ受信とその処理のコードはイベント・ドリブンの記述が求められる。
2. SendPort.callが無くなった。またSendPort.sendは引数が一つだけとなり、replyToポートを指定することができなくなった。どうしても返送先をメッセージに付加したいときは、リストあるいはマップの形で付加すれば良い。
3. ReceivePort.toSendPort()が無くなり、ゲッタReceivePort.sendPortを使用することになった。
4. spawnFunctionとspawnUriがIsolate.spawnと Isolate.spawnUriに変更になった。各々のメソッドはエントリ・ポイントに渡される初期メッセージを引数にする。'spawnUri'はまた"main"のためのList<String>引数をとる。特に注意しなければならないことは、DOM対応のストリーム(すなわちdart:htmlをインポートしたブラウザ用のコード)から子供のストリームを産み付けるには'spawnUri'を使わねばならなくなったことである。そのようなアプリケーション(HTMLファイル)を自分のブラウザから直接アクセスすると、Chromeではセキュリティ・エラーが起きる(Firefoxでは起きない)。実際の運用ではウェブ・サーバにアクセスするので問題はないが、デバッグやテストでは支障をきたす。
5. またStreamを実装せず、Zoneを尊重しないRawReceivePortが導入されている(現時点ではVMのみ)。
6. Isolateクラスが復活し、グローバルなportとストリーム変数たちも廃止となった。

18.1節　　ブラウザ上でのアイソレート

dart2jsで変換された通常のブラウザでのアイソレートは前述のようにWeb Workersである。実験によればメインのスレッドとアイソレートのスレッドとは分離されている。従ってアイソレート内で受信のコールバック関数の中にdo{}while(true);のような無限ループを置くと、アイソレートとの通信はそれ以降出来なくなるが、メインのその他のイベント処理は継続できる。

WHATWGで作業中のWeb Workersは、ユーザ・インターフェイスのスクリプトと独立してバックグラウンドで走るスクリプトの為のAPIである。バックグラウンドの処理としては例えばつぎのものを挙げている:

- 時間がかかる計算、処理
- 定期的に株価情報を取り込むような入出力処理

ワーカとはメッセージ渡しの通信であることもアイソレートと同じである。GoogleのDartチームはこれを想定して(実際はWeb Workersを呼び出す)クライアント側のアイソレートを開発している。従って現在はDOMアクセスが出来なくなっている。

現にGoogleの技術者は「新しいAPIはWeb Workerのコンテキストで走るアイソレートのみをサポートしており、

従ってUIアクセスを持っていない。将来spawnDOMIsolateに似たものを追加する計画はあるが、しばらく時間がかかる」と書いている。

また新しいAPIドキュメントのdart:isolateの説明には次のように書かれている:

「そのアイソレートが同じスレッドであるいは別のスレッドで走るかを示す手段は現在このAPIには存在していない。下位層のシステムがそのアイソレートを適正にスケジューリングする。近い将来我々はDOMアイソレートたちを生成する為のAPIを追加する予定である。これらはDOMアクセスを共有するアイソレートたちである。総てのDOMアイソレートはUIスレッド上で走ることになる。」

ということは、将来再度登場するかもしれないDOMアクセスできるアイソレートはやはり軽量アイソレートのままということになりそうである。これはそもそもDOMがマルチ・スレッド対応で無いことによると思われる。

ブラウザ上でのアイソレート(即ちdart:htmlをインポートしたアイソレート)はdart:ioがサポートされず、またDOMにも対応しないことに注意。且つこれまではdart:ioとdart:htmlにあるタイマ機能がdart:coreにないので、ブラウザ上のアイソレートはタイマ・イベントが使用できなかった。一方Web WorkerではsetTimeout()/clearTimeout()及びsetInterval()/clearInterval()がサポートされている。2012年8月からはTimerは暫定的ではあるがdart:isolateに移されている。その後2013年1月に新しく出来たdart:asyncライブラリに移されている。そして2013年2月にM3変更の一環としてwindow.setTimeoutとwindow.setIntervalが廃止され、Timerに集約された。

18.2節　　Isolateライブラリ

Dartチームはアイソレートに関わる2012年2月のAPIの再編で、core.Isolateクラスの機能はIsolateライブラリのトップ・レベル関数に持たすようにした。更には2013年10月の変更でこれらのトップ・レベル関数はIsolateクラスのstatic関数となった。またsendメソッドはメッセージという単一の引数しか持たなくなっている。

新しいAPIによるシンプルなエコーのサンプル・コードは、特にグローバルなportが無くなりまたReceivePortがStreamを実装したことにより、従来よりも長くなる:

code_18.2.dart

```
import 'dart:isolate';

void remote(SendPort replyTo) {
  var receivePort = new ReceivePort();
  replyTo.send(receivePort.sendPort);
  receivePort.listen((msg) {
    print('remote received : $msg');
    if (msg == 'bar') receivePort.close();
    replyTo.send('Echo : $msg');
  });
}

main() {
  var receivePort = new ReceivePort();
  var sendPort = receivePort.sendPort;
  Isolate.spawn(remote, sendPort).then((_) {
    receivePort.listen((msg) {
      if(msg is SendPort) {
```

```
      sendPort = msg;
      sendPort.send('foo');}
    else {
      print ('received : $msg');
      sendPort.send('bar');
    }
  });
 });
}
```

この件は、変更発表に対する多くの意見にも反映されている。しかしながら現時点ではDartチームはこのAPIにユーザまたはサード・パーティが用途に応じたライブラリを乗せることを想定している。

このプログラムの意味に関しては追って説明することとする。

このライブラリは次のようなもので構成されている:

抽象クラス	RawReceivePort	アイソレート間でのストリームを実装していない低レベルメッセージ受信ポート(現時点ではVMのみが実装)
	ReceivePort	アイソレート間でのメッセージ受信ポート
	SendPort	アイソレート間でのメッセージ送信ポート
クラス	Isolate	新規アイソレート生成のためのstaticメソッドたち(これまではライブラリのトップ・レベルの関数だった)
例外	IsolateSpawnException	アイソレート産み付けの際の例外

Isolateクラスは新規アイソレートを生成する為のテンプレートである。即ちこのクラスのサブクラスのオブジェクトのspawn(またはspawnUri)メソッドを呼ぶことで新しいアイソレートが産み付け(spawn)られる。アイソレートはspawn(またはspawnUri)メソッドの引数で指定された関数から開始する。

spawn(またはspawnUri)メソッドの引数には最初に親から産み付けた子に渡すメッセージが含まれている。通常このメッセージには親に送信するための送信ポートを含めることになる。そうしないと子は親に送信するための手段が無くなる。

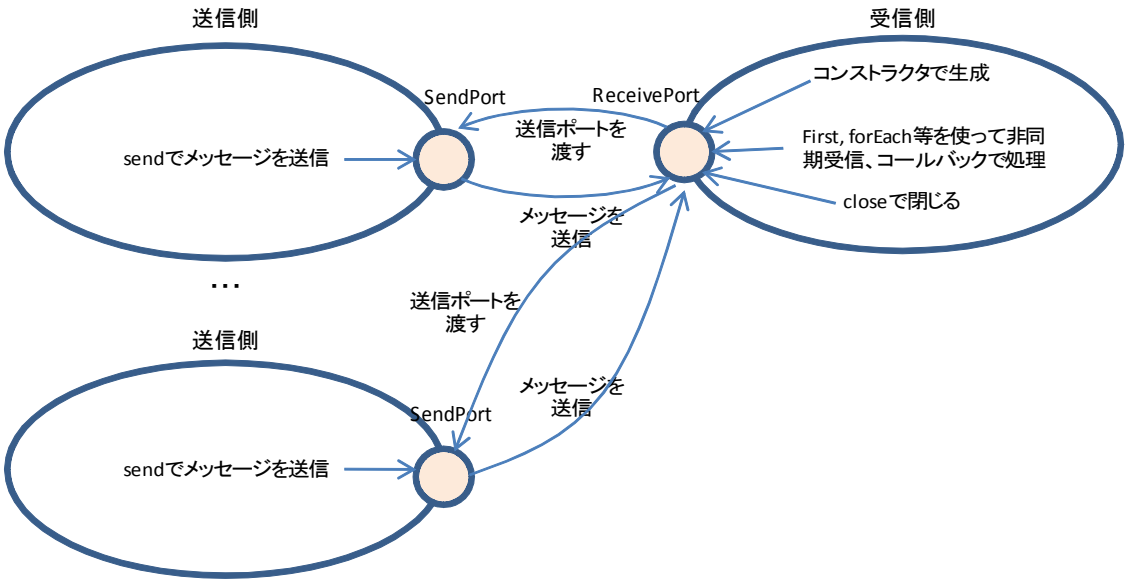
反対に子から親への最初の送信メッセージには親が子に送信するための送信ポートの情報を含めねばならない。そうすることで両者間の通信の手段が確立する。従来のAPIでは各送信メッセージに返信のためのポートを含めることが可能だったが、そのようなオプションは無くなっている。

18.3節　　ポート

これまでの説明で送信及び受信ポートのイメージが掴めたかと思う。ポートには受信ポートであるReceivePortとそのポートに送信する為の送信ポートであるSendPortとがあり、共に抽象クラスである。これらはともにアイソレート間の通信の為の唯一の手段である。ReceivePortはsendPortというゲッタを持ち、これがSendPortを返す。このSendPortを通過して送信されるメッセージは、その送信ポートを生成したReceivePortに渡される。そこでは、それらはその受信ポートに登録されているコールバック関数に渡される。

ReceivePortは複数のSendPortを持つことができる。

下図はそのイメージである:



この図では送信側はメッセージの受信をしていないが、これは送受ポートの概念を示すためのもので、送信側が受信が出来ない訳ではない。メッセージ通信により送信側のReceivePortに送信するためのSendPortオブジェクトをもらい、受信側でその送信ポートを使って送信側にメッセージを送信することも無論可能である。

ポートだけを使うことは無いだろうが、以下はポートの動作を知るための簡単なコードである:

code_18.3.dart

```
import 'dart:isolate';

class Sender {
  SendPort sendPort;
  String senderId;
  Sender(SendPort sendPort, String senderId) {
    this.sendPort = sendPort;
    this.senderId = senderId;
  }
  run() {
      sendPort.send('received messaage from $senderId');
  }
}

main() {
  var receivePort = new ReceivePort();
  receivePort.forEach((msg) {
    print(msg);
    if(msg.endsWith('#2')) {
      receivePort.close();
    }
  });
  new Sender(receivePort.sendPort, 'sender #1').run();
  new Sender(receivePort.sendPort, 'sender #2').run();
  new Sender(receivePort.sendPort, 'sender #3').run();
}
```

ここでは送信側をSenderというクラスで表現している。このクラスのコンストラクタには受信側への送信ポート(sendPort)と、そのオブジェクトの識別のための文字列(senderId)を引数として渡している。このオブジェクトのrunメソッドはその送信ポートから識別文字列を含んだメッセージを送信するだけである。

受信側はmain()メソッドであり、

1. 受信ポートを用意する。
2. その受信ポートを使って、メッセージが受かった時の処理のコールバック関数を登録している。
3. コールバックの中では、受信したメッセージ毎にそれを取り出してコンソールに表示する。
4. もし#2の送信側からのメッセージが受かっているときは、受信ポートをクローズする。
5. 準備ができたら、送信側を3個生成し、実行させる。

forEachというメソッドは、StreamであるReceivePortに受信されたメッセージを順番に取り出すもので、未だ受信メッセージが存在しないときは、それが到来するまで待つ。したがって、このメソッドはFutureオブジェクトを返す。メッセージを受信したら、コールバックでそのメッセージを処理する。

このコードを実行させると、

received messaage from sender #1
received messaage from sender #2

とのみ表示される。これは2番目の送信側からのメッセージを受けたことで受信ポートが閉じたためである。

受信ポートを閉じると、その後の受信は廃棄される。受信ポートを閉じないと、このプログラムはいつまでも受信待ち状態を継続する。

今回の場合は問題ではないが、子供のアイソレートでは注意が必要である。ReceivePortは誰もそれにそれ上送信しなくなったときに自動的にガベージ・コレクトされる訳ではない(Dartではアイソレートにまたがるガベージ・コレクションの機能はない)。従ってReceivePortはリソースの類だとして取り扱い、**以後使用されなくなったときには必ずクローズしなければならない。**

18.4節　　アイソレートの産み付け (Spawning Isolates)

dart:isolateライブラリはアイソレートの産み付け及び通信にかかわるAPIを定めている。

Dartの総てのコードはアイソレートのコンテキスト(環境)内で走る。各アイソレートは各々のヒープを持つ、即ちこれはグローバルのものも含めてメモリ内の総ての値はそのアイソレートのみがアクセスできることを意味する。アイソレートたち間の通信に使える唯一のメカニズムはメッセージ渡しである。メッセージはポートを介して送信される。本ライブラリでは通信チャンネルの受信端を表現するReceivePort、及び送信端を表現するSendPortを規定している。

新規のアイソレートを産み付けるにはIsolateクラスのspawnFunction及びspawnUriの2つのメソッドが使える。spawnFunctionは現在のアイソレートと同じソース・コードを使う新規アイソレートを生成し、spawnUri は独立して書かれたアイソレートを産み付けることができる。

Future<Isolate> **spawn**(void entryPoint(message), message)	現在走っているのアイソレートと同じコードを共有するアイソレートを生成し産み付ける。

	引数のentryPointは産み付けられたアイソレートの開始点を指定する。これはstaticなトップ・レベルの関数かあるいは引数を持たないstaticメソッドかでなければならない。関数クロージャを渡すことは許されていない。 このエントリ・ポイント関数は初期メッセージで呼び出される。産み付け側と産み付けられた側間での相互通信が可能になるよう、通常この初期メッセージにはSendPortのオブジェクトが含められる。 このメソッドはIsolateのインスタンスを持ったFutureオブジェクトを返す。このIsolateのインスタンスは産み付けられたアイソレートをコントロールするのに使うことができる。
Future<Isolate> **spawnUri**(Uri uri, List<String> args, message)	指定したURIのライブラリからのコードで実行するアイソレートを生成し産み付ける。 DOM対応の（すなわちdart:htmlをインポートした）コードからのアイソレートの産み付けにはこのメソッドを使用しなければならない。 このアイソレートは指定されたURIのライブラリのトップ・レベルにあるmain関数の実行を開始する。 ターゲットとなるmainは次の3つのシグネチュアの一つをとる： • main() • main(args) • main(args, message) messageの引数が存在するときはそれは初期メッセージにセットされる。argsが存在するときは、それらはargsリストにセットして渡される。 このメソッドはIsolateのインスタンスを持ったFutureオブジェクトを返す。このIsolateのインスタンスは産み付けられたアイソレートをコントロールするのに使うことができる。

spawn

spawnを使った具体的な例を示す：

code_18.4a.dart

```
import 'dart:isolate';
import 'dart:async';

void echo(SendPort initialReplyTo) {
  var port = new ReceivePort();
  initialReplyTo.send(port.sendPort);
  port.listen((msg) {
    var data = msg[0];
    SendPort replyTo = msg[1];
    replyTo.send(data);
    if (data == "bar") port.close();
  });
}

Future sendReceive(SendPort port, msg) {
```

```
  ReceivePort response = new ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}

main() {
  var response = new ReceivePort();
  Future<Isolate> remote = Isolate.spawn(echo, response.sendPort);
  remote.then((_) => response.first)
    .whenComplete(response.close)
    .then((sendPort) {
    sendReceive(sendPort, "foo").then((msg) {
      print("received: $msg");
      return sendReceive(sendPort, "bar");
    }).then((msg) {
      print("received another: $msg");
    });
  });
}
```

このコードはcode_18.2.dartと似ているが、Futureを返すsendReceveという関数を介しているところが異なる。

```
Isolate.spawn(echo, response.sendPort);
```

という行では、echoというトップ・レベルの関数（子供のアイソレート）を指定し、また親の受信ポート(response)にメッセージを送信するための送信ポートを渡している。

sendReceiveという関数はその都度受信ポートを生成して子供のアイソレートからのメッセージを受信している。この関数はresponse.firstというインスタンスのFutureオブジェクトを返しているので、thenメソッドでこれを処理している。

spawnUri

spawnUriは別のファイルに収容されているアイソレートのコードからアイソレートを産み付けるのに使われる。そのコードにはmain関数が存在しなければならない。

また、未だ公式に公開されていないが、改正されたdart:isolateライブラリでは、DOM対応アイソレート（すなわちdart:htmlを実装した）からはspawnは使用出来なくなっている。その代りspawnUriを使用しなければならない。サーバ・サイドではspawnが使用できる。

簡単な例を示す:

code_18.4b.dart

```
// spawning an isolate using spawnUri
import 'dart:isolate';
import 'dart:async';
main() {
  var response = new ReceivePort();
  Future<Isolate> remote =
      Isolate.spawnUri(Uri.parse('code_18.4c.dart'), ['foo'], response.sendPort);
  remote.then((_) => response.first)
    .then((msg) { print('received: $msg'); });
}
```

code_18.4c.dart

```
// echo isolate
import 'dart:isolate';
void main(List<String> args, SendPort replyTo) {
  replyTo.send(args[0]);
}
```

code_18.4b.dartが親のアイソレートのコードであり、Isolate.spawnUriメソッドを使ってcode_18.4c.dartファイルに記述されている子のアイソレートを産み付けている。子のコードのURIを取得するにはUri.parseというメソッドを使用する。引数であるargsはStringのリストであり、例えば最初に渡す文字メッセージ等をセットする。messageはポート間で送受されるメッセージと同じでプリミティブなオブジェクトも含み、ここでは送信ポートのオブジェクトを渡している。

code_18.4c.dartは子供のアイソレートを記述したファイルであり、ここではmainの引数は(List<String> args, SendPort replyTo)としている。

mainは:

- main()
- main(args)
- main(args, message)

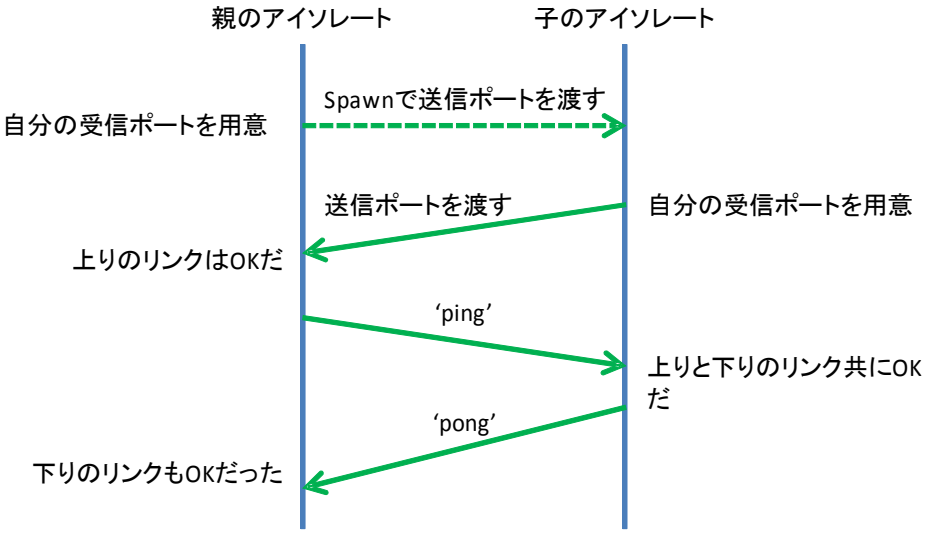
の形式をとることができる。

18.5節　　アイソレート間の通信リンクの確立

Dartではメインのアプリケーション自身もアイソレートである。これまでに説明したように、各アイソレートは自分の受信ポートを用意するとともに、その受信ポートに送信する為の送信ポートを相手に知らせることで相互の通信のリンクを確立する。しかしながらその手順には最初の通信リンクが必要になる。新しいdart:isolateのライブラリでは、その為の送信ポートは産み付け関数のspawnFunctionまたはspawnUriで産み付けた子供のアイソレートに渡す。子供は自分の受信ポートを生成して、そのための送信ポートを親のアイソレートにメッセージとして送信する。子供はそれに対する返信を受信したことで通信のリンクが確立したことを知る。

従って、非同期でメッセージを交信する為のリンクは次のようにして確立することになる:

アイソレート間のリンクの確立



子のアイソレートはpingが帰ってきたことで上りと下りのリンクが確立していることを知る。一方親のほうはpongが帰ってきて初めて上りと下りのリンクが確立していることを知る。

なおIsolateクラスには新しくpingメソッドが追加されているので、それも参考にされたい。pingメソッドは相手のアイソレートのコードが感知することなく実行されるので、相手のアイソレートが活きているかどうかの確認に適している。周期的に相手にpingをかけ、所定の時間内にその応答が帰ってこなければ相手のアイソレートが停止していると判断する。イベント処理の章のタイムアウト監視の項を参考にすればよい。

以下はアイソレート間のリンク確立のプログラム例である。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJなどのIDEから\dart_code_samples-master\codes\code_18.5.dartのフォルダを開くと良い。

code_18.5.dart

```
001 // Dart code sample for establishing communications link between isolates
002
003 import 'dart:async';
004 import 'dart:isolate';
005 import 'dart:math';
006 //import 'dart:html'; // enable this to run on  Dartium or Dert2JS
007
008 // isolate status
009 final CONNECTING = 1;
010 final CONNECTED = 2;
011 final STOPPED = 3;
012
013 // top level child isolate function
014 childIsolate(SendPort port) {
015   // status
016   var status = CONNECTING;
017   // long-lived ports
018   var receivePort = new ReceivePort();
019   var sendPort = port;
020   log('child started');
021
022   // after link establishment, do things here using receivePort and sendPort like:
023   run([msg = null]){
024     if (msg != null) log('child received : $msg');
025     if (msg == null) { // active transmission
026       sendPort.send({'one way message from child':[1,2,3,5]}); // send Map with List
027     }
028     else if (msg is List) {  // echo back the List with added elements
029       msg.addAll([', and cos pi is ', cos(PI)]);
030       sendPort.send(msg);
031     }
032     else if (msg == 'quit') { // close command
```

```
033       status = STOPPED;
034       receivePort.close();
035       log('child closed it\'s receive port');
036     }
037     else {
038       sendPort.send('child echoed : $msg');  // simple echo
039     }
040   }
041
042   // establish communication link
043   sendPort.send(receivePort.sendPort);
044   linkEstablish(msg){
045     if (msg == 'ping') {
046       log('child received : $msg');
047       sendPort.send('pong');
048       status = CONNECTED;
049       run();
050     }
051   }
052
053   // receive messages and dispatch them
054   receivePort.listen((msg){
055     if (status == CONNECTING) linkEstablish(msg);
056     else if (status == CONNECTED) run(msg);
057     });
058 }
059
060
061 // parent isolate class
062 class ParentIsolate {
063   // status
064   var status = CONNECTING;
065   // long-lived ports
066   var receivePort = new ReceivePort();
067   SendPort sendPort;
068
069   // establish communication link
070   linkEstablish(msg){
071     if (msg is SendPort) {
072       sendPort = msg;
073       sendPort.send('ping');
074     }
075     else if (msg == 'pong') {
076       log('main received : $msg');
077       log('link established');
078       status = CONNECTED;
079       run();
080     }
081   }
082
083   // link established, then do things here
084   run([msg = null]) {
085     if (msg == null) { // active transmission
086       sendPort.send('one way message from parent');
087       var myList = ['pi is ' , PI];
088       sendPort.send(myList);  // you can send List, Map and other object also
089       sendPort.send('quit');  // send 'quit' to close the child receive port
090     }
091     else log('main received : $msg');  // response transmission
092   }
093
094   // main process
095   void main() {
096     // communication link establishment
097     Isolate.spawn(childIsolate, receivePort.sendPort).then((iso){
098       log('spawned child isolate #${iso.hashCode}');
099     });  // receive messages and dispatch them
100     receivePort.listen((msg){
101       if (status == CONNECTING) linkEstablish(msg);
102       else if (status == CONNECTED) run(msg);
103     });
104   }
105 }
106
```

```
107
108 main() {
109   new ParentIsolate().main();
110 }
111
112
113 void log(String msg) {
114   String timestamp = new DateTime.now().toString().substring(11);
115   print('$timestamp : $msg');
116   //enable next line to run on Dartium or Dart2JS
117 //  document.body.nodes.add(new Element.html('<div>$msg</div>'));
118 }
```

このプログラムは次のような構成となっている:

- 子供のアイソレートを記述したトップ・レベルの関数(childIsolate(SendPort port))
  - リンク確立後のメッセージ交換の関数(run([msg = null]))
  - リンク確立の手続きの関数(linkEstablish(msg))
  - メッセージ受信とそのメッセージの振り向けのコールバック(receivePort.listen((msg){....})
- 親のアイソレートを記述したクラス(ParentIsolate)
  - リンク確立後のメッセージ交換の関数(run([msg = null]))
  - リンク確立の手続きの関数(linkEstablish(msg))
  - メッセージ受信とそのメッセージの振り向けの関数(main())
- トップ・レベルのmain関数
- ログを記録するトップ・レベルの関数(log(String msg))

2013年10月のAPIの大規模変更でReceivePortがStreamを実装したこと受け、アイソレートのコードはよりイベント・ドリブンのスタイルにすることが求められる。子供アイソレートの場合はイベントの源は受信ポートである。したがって、このイベントをもとに状態遷移が発生し、またその状態に基づいてイベントの振り向け(ディスパッチ)が行われる。

このサンプルでは3つの状態が用意されている:

- CONNECTING 通信リンクを確立中
- CONNECTED 通信リンクが確立しており、双方間での通信が可能
- CLOSED 受信ポートが閉じており、その後の通信はできない

このプログラムを実行するとコンソール上で次のようなログが出力されるはずである:

```
15:24:49.274 : child started
15:24:49.276 : spawned child isolate #220590462
15:24:49.285 : child received : ping
15:24:49.285 : main received : pong
15:24:49.285 : link established
15:24:49.287 : child received : one way message from parent
15:24:49.289 : child received : [pi is , 3.141592653589793]
15:24:49.289 : child received : quit
15:24:49.290 : main received : {one way message from child: [1, 2, 3, 5]}
15:24:49.290 : main received : child echoed : one way message from parent
15:24:49.291 : main received : [pi is , 3.141592653589793, , and cos pi is , -1.0]
15:24:49.297 : child closed it's receive port
```

これを見ると次のことがわかる:

1. 子供がpingを受信し、親がpongを受信したことでリンクが確立したと報告している。
2. その後子供が3つのメッセージを受信している。
3. 子供が受信した最後のquitというメッセージは、子供に受信ポートを閉じるためのコマンドである。このように何らかの形で受信ポートを閉じないと、これはガーベージ・コレクションの対象になっていない為で

ある。

4. また親は3つのメッセージを受信している。ひとつは子供が自発的に送信したMapであり、残りは親からのメッセージに対する応答である。
5. 最後に子供はその受信ポートを閉じている。

子供のアイソレートのコード(childIsolate)はトップ・レベルの関数として次のように定義されている:

```
013 // top level child isolate function
014 childIsolate(SendPort port) {
015   // status
016   var status = CONNECTING;
017   // long-lived ports
018   var receivePort = new ReceivePort();
019   var sendPort = port;
020   log('child started');
021
022   // after link establishment, do things here using receivePort and sendPort like:
023   run([msg = null]){
024     if (msg != null) log('child received : $msg');
025     if (msg == null) { // active transmission
026       sendPort.send({'one way message from child':[1,2,3,5]}); // send Map with List
027     }
028     else if (msg is List) {  // echo back the List with added elements
029       msg.addAll([', and cos pi is ', cos(PI)]);
030       sendPort.send(msg);
031     }
032     else if (msg == 'quit') { // close command
033       status = STOPPED;
034       receivePort.close();
035       log('child closed it\'s receive port');
036     }
037     else {
038       sendPort.send('child echoed : $msg');  // simple echo
039     }
040   }
041
042   // establish communication link
043   sendPort.send(receivePort.sendPort);
044   linkEstablish(msg){
045     if (msg == 'ping') {
046       log('child received : $msg');
047       sendPort.send('pong');
048       status = CONNECTED;
049       run();
050     }
051   }
052
053   // receive messages and dispatch them
054   receivePort.listen((msg){
055     if (status == CONNECTING) linkEstablish(msg);
056     else if (status == CONNECTED) run(msg);
057     });
058 }
```

1. この関数はトップ・レベルに置かれねばならないことに注意。
2. 018行目で受信ポート(receivePort)を取得している。
3. 019行目に示すように、相手に送信するための送信ポート(sendPort)はこの関数の引数となっている。
4. 043行目にあるように、最初に相手に自分の受信ポートに送信するための送信ポートのオブジェクトを親に送信することで、リンク確立作業が開始される。
5. 行054-057では受信ポートで受信したメッセージをどう振り向けるかを記述している。接続中であればlinkEstablish関数に、確立済みであればrun関数にそのメッセージを渡している。
6. 行044-051がlinkEstablish関数である。pingが到来したらリンクが確立したと判断して状態をCONNECTEDに切り替え、引数なしでrun関数を読んでいる。この関数は引数がnullのときは自発的に行うなんかの作業を記述する。ここでは自発的なメッセージ送信を行っている。
7. 行023-040がrun関数である。この関数は引数がnullのときは自発的に行うなんかの作業を記述する。こ

こでは自発的なメッセージ送信を行っている。引数が存在するときはそのメッセージをもとに何らかの作業を行う。exitというメッセージは親からの終了コマンドであり、これを受けたら受信ポートをクローズする。

一方親のアイソレートのプロセスはParentIsolateというクラスで表現されている:

```
061 // parent isolate class
062 class ParentIsolate {
063   // status
064   var status = CONNECTING;
065   // long-lived ports
066   var receivePort = new ReceivePort();
067   SendPort sendPort;
068
069   // establish communication link
070   linkEstablish(msg){
071     if (msg is SendPort) {
072       sendPort = msg;
073       sendPort.send('ping');
074     }
075     else if (msg == 'pong') {
076       log('main received : $msg');
077       log('link established');
078       status = CONNECTED;
079       run();
080     }
081   }
082
083   // link established, then do things here
084   run([msg = null]) {
085     if (msg == null) { // active transmission
086       sendPort.send('one way message from parent');
087       var myList = ['pi is ' , PI];
088       sendPort.send(myList);  // you can send List, Map and other object also
089       sendPort.send('quit');  // send 'quit' to close the child receive port
090     }
091     else log('main received : $msg');  // response transmission
092   }
093
094   // main process
095   void main() {
096     // communication link establishment
097     Isolate.spawn(childIsolate, receivePort.sendPort).then((iso){
098       log('spawned child isolate #${iso.hashCode}');
099     });  // receive messages and dispatch them
100     receivePort.listen((msg){
101       if (status == CONNECTING) linkEstablish(msg);
102       else if (status == CONNECTED) run(msg);
103     });
104   }
105 }
```

1. 066行目で受信ポートを用意している。
2. 行095-104がこのクラスのメインのメソッドである。
3. 行100-103では受信ポートに到来したメッセージを状態に応じ所定のメソッドに振り向けている。
4. 行084-092がrun関数である。この関数は引数がnullのときは自発的に行うなんかの作業を記述する。ここでは自発的なメッセージ送信を行っている。引数が存在するときはそのメッセージをもとに何らかの作業を行う。
5. 行071-081がlinkEstablishメソッドである。pongが到来したらリンクが確立したと判断して状態をCONNECTEDに切り替え、引数なしでrun関数を読んでいる。この関数は引数がnullのときは自発的に行うなんかの作業を記述する。ここでは自発的なメッセージ送信を行っている。

なおこのプログラムは時間的な状態の推移を知ることができるように幾つかの個所でlog()メソッドを呼び出している。実際に読者がこのプログラムを流用する場合は、それらを削除すれば良い。

18.6節　　リストやマップの送受信

上記のように通信リンクが確立したら、そのリンク上では単にStringのようなプリミティブな値だけでなく、ListとMapのオブジェクトも交信できる（現在はnum, String, bool, null, ListおよびMapの交信が可能）。更に2つのアイソレートたちが同じコードを共有しまた同じプロセス内で走っている状況（例えばこのプログラムのようなIsolate.spawnでアイソレートたちが生成されているとき）内では、オブジェクトのインスタンスを送信する（そのプロセス内でコピーされることになる）ことも可能である。

以下はこのプログラムの終わりの部分でリストを送受信した例である。

親のアイソレート側のオブジェクト送受信

```
// link established, then do things here
run([msg = null]) {
  if (msg == null) { // active transmission
    sendPort.send('one way message from parent');
    var myList = ['pi is ' , PI];
    sendPort.send(myList);  // you can send List, Map and other object also
    sendPort.send('quit');  // send 'quit' to close the child receive port
  }
  else log('main received : $msg');  // response transmission
}
```

親は子供に対し更にmyListというListオブジェクトを送信している。

子供のアイソレート側のオブジェクト送受信

```
// after link establishment, do things here using receivePort and sendPort like:
run([msg = null]){
  if (msg != null) log('child received : $msg');
  if (msg == null) { // active transmission
    sendPort.send({'one way message from child':[1,2,3,5]}); // send Map with List
  }
  else if (msg is List) {  // echo back the List with added elements
    msg.addAll([', and cos pi is ', cos(PI)]);
    sendPort.send(msg);
  }
  else if (msg == 'quit') { // close command
    status = STOPPED;
    receivePort.close();
    log('child closed it\'s receive port');
  }
  else {
    sendPort.send('child echoed : $msg');  // simple echo
  }
}
```

一方子供のほうは、

- 一方的な送信メッセージは'one way message from child'というキーを持った[1,2,3,5]というリストからなるマップのオブジェクトとなっている。
- 受信したメッセージがList型オブジェクトかどうかを判定し、もしそうならそのリストに2つの要素を追加し、親に返信している。そうでなければそれに”child echoed : ”という文字列を頭に付けて親に返信している。

$msgという変換はListやMapオブジェクトを見やすい形で出力してくれる。ここでは子供が一方的に送信した

マップを{one way message from child: [1, 2, 3, 5]}と、また子供のアイソレートが追加をした2つを含めて4つの要素からなるリストを[pi is , 3.141592653589793, , and cos pi is , -1.0]のように要素ごとにカンマで区切って表示してくれている。

下図はDartiumでこのプログラムに相当したIsolateTestを走らせた例である:

この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJなどのIDEからdart_code_samples / apps / IsolateTest フォルダを開き、IsolateTest.htmlを開き、「Dartiumで実行」を選択する。この場合はspawnUriで子供のアイソレートを産み付けねばならない。したがってIsolateTest_child.dartというファイルが用意されている。

DOM対応アイソレートではprint()メソッドも使用出来なくなるので、親にそのログを送信している。

18.7節　　並行処理の確認

前節では2つのアイソレート間の通信について説明した。本節では2つのアイソレートが本当に2つのスレッドで並行処理を行っているのかを確認する。

そのような動作確認のために良く使用されるのがFibonacci(フィボナッチ)関数である。これは次数が上がると処理時間がかかるので、重い処理をシミュレートするのに都合が良いからである。この関数はWikipediaなどを見て頂きたいが、Dartで書くと次のような典型的な処理時間がかかる再帰型の関数となる(「ネストした関数及び関数リテラル」の項を参照のこと):

```
int fib(int i) {
  if (i < 2) return i;
  return fib(i - 2) + fib(i - 1);
}
```

この関数を親と子供の双方で実行させ、同時処理が実行されていることを確認しよう。

なお2013年11月のdart:isolateライブラリの大規模変更に伴い、DOM対応のアイソレートでは、産み付ける子供

のアイソレートは別ファイルとしなければならなくなった。この場合は関数の共有の実験ができなくなる。従ってここではVM単体で動作するcode_18.7.dartのほうをIDE上で実験することとする。内容はIsolateTestFibonacci.dartと同じである。

1. この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、Githubからダウンロードからダウンロードし、IntelliJなどのIDEから\dart_code_samples-master\codes\code_18.7.dartのフォルダを開く。
2. あるいはDart Editorからcode_18.7.dartを実行させる（例えば右クリック→Runを選択）。

このプログラムは次のようになっている：

code_18.7.dart

```
001 /*
002  * Dart code sample for concurrency tests
003  */
004
005 //import 'dart:html';
006 import 'dart:async';
007 import 'dart:isolate';
008
009 // isolate status
010 final CONNECTING = 1;
011 final CONNECTED = 2;
012 final STOPPED = 3;
013
014 // top level child isolate function
015 childIsolate(SendPort port) {
016   // status
017   var status = CONNECTING;
018   // long-lived ports
019   var receivePort = new ReceivePort();
020   var sendPort = port;
021
022   // after link establishment, do things here using receivePort and sendPort like:
023   run([msg = null]){
024     if (msg == null) { // active transmission
025       sendPort.send('${new DateTime.now().toString().substring(11)}'
026                     ' : child started Fib computing');
027       int i = 40;
028       int y;
029       y = fibo(i);
030 //  y = fib(i);
031       sendPort.send('${new DateTime.now().toString().substring(11)}'
032                     ' : child finished Fib(${i}) = ${y}');
033     }
034     else {
035       sendPort.send('child echoed : $msg');  // simple echo
036     }
037   }
038
039   // establish communication link
040   sendPort.send(receivePort.sendPort);
041   linkEstablish(msg){
042     if (msg == 'ping') {
043       log('child received : $msg');
044       sendPort.send('pong');
045       status = CONNECTED;
046       run();
047     }
048   }
049
050   // receive messages and dispatch them
051   receivePort.listen((msg){
052     if (status == CONNECTING) linkEstablish(msg);
053     else if (status == CONNECTED) run(msg);
054     });
055 }
056
057 // parent isolate class
```

```
058 class ParentIsolate {
059   // status
060   var status = CONNECTING;
061   // long-lived ports
062   var receivePort = new ReceivePort();
063   SendPort sendPort;
064
065   // establish communication link
066   linkEstablish(msg){
067     if (msg is SendPort) {
068       sendPort = msg;
069       sendPort.send('ping');
070     }
071     else if (msg == 'pong') {
072       log('main received : $msg');
073       log('link established');
074       status = CONNECTED;
075       run();
076     }
077   }
078
079   // link established, then do things here
080   run([msg = null]) {
081     if (msg == null) { // active transmission
082       log('parent started Fibonacci computing');
083       int i = 40;
084       int y = fib(i);
085       log('parent finished Fib(${i}) = ${y}');
086     }
087     else log('main received : $msg');  // response transmission
088   }
089
090   // main process
091   void main() {
092     // communication link establishment
093     Isolate.spawn(childIsolate, receivePort.sendPort).then((iso){
094       log('spawned child isolate #${iso.hashCode}');
095     });  // receive messages and dispatch them
096     receivePort.listen((msg){
097       if (status == CONNECTING) linkEstablish(msg);
098       else if (status == CONNECTED) run(msg);
099     });
100   }
101 }
102
103
104 int fib(int i) {
105   if (i < 2) return i;
106   return fib(i - 2) + fib(i - 1);
107 }
108
109 int fibo(int i) {
110   if (i < 2) return i;
111   return fibo(i - 2) + fibo(i - 1);
112 }
113
114 main(){
115   new ParentIsolate().main();
116 }
117
118 void log(String message) {
119   String timestamp = new DateTime.now().toString().substring(11);
120   print('$timestamp : $message');
121 //  document.body.nodes.add(new Element.html('<div>$timestamp : $message</div>'));
122 }
```

このプログラムは、前前節で示したプログラムを加工しただけなので、読者は容易に理解できよう。主たる変更箇所は赤で示されている。

- 子供のアイソレートは025-032行でFibonacci関数を計算しその前後の時間を親に報告している。
- 親のアイソレートは082-085行でFibonacci関数を計算しその前後の時間をログ出力している。

- Fibonacci関数は104-107および109-112行に2つfib及びfiboという名前で用意されている。

このプログラムでは引数が40になっているが、読者は自分のコンピュータ能力にあわせてこの値を変更されたい。

親のアイソレートの実行時間への子供のアイソレートの影響

最初に029及び030行をコメントアウトして、親だけにfib関数を実行させる次のような結果が得られる:

```
21:32:33.536 : spawned child isolate #520680098
21:32:33.544 : child received : ping
21:32:33.548 : main received : pong
21:32:33.548 : link established
21:32:33.549 : parent started Fibonacci computing
21:32:34.366 : parent finished Fib(40) = 102334155
21:32:34.366 : main received : 21:32:33.548 : child started Fibonacci computing
21:32:34.367 : main received : 21:32:33.549 : child finished Fib(40) = null
```

これから親は817mSの時間がかかっていることが判る。

次に030行目のコメントアウトを外して、子供のアイソレートにfibo関数を実行させるようにすると、次のような結果が得られる:

```
21:37:04.518 : spawned child isolate #471862191
21:37:04.525 : child received : ping
21:37:04.531 : main received : pong
21:37:04.531 : link established
21:37:04.531 : parent started Fibonacci computing
21:37:05.354 : parent finished Fib(40) = 102334155
21:37:05.354 : main received : 21:37:04.531 : child started Fibonacci computing
21:37:05.366 : main received : 21:37:05.366 : child finished Fib(40) = 102334155
```

この場合は親は04.531秒目に開始し、823mS後の05.354秒目に計算を終了している。また子供は同じ04.531秒目に計算を開始して835mS後の05.366秒目に終了している。即ち**親のアイソレート側は、子供が実行していることの処理時間への影響を受けていない**。親も子供もほぼ2.1秒で計算を終了している。これは親と子供のアイソレートに対しOSが2つのスレッドを割り当てているからである。

アイソレート間での関数の共有

逆に031行目のコメントアウトのほうを外して、親と子供が同じfibという関数を使用するようにしたらどうなるだろうか？その結果は次のようになる:

```
21:41:41.076 : spawned child isolate #358623313
21:41:41.084 : child received : ping
21:41:41.089 : main received : pong
21:41:41.089 : link established
21:41:41.090 : parent started Fibonacci computing
21:41:41.939 : parent finished Fib(40) = 102334155
21:41:41.939 : main received : 21:41:41.089 : child started Fibonacci computing
```

21:41:41.939 : main received : 21:41:41.927 : child finished Fib(40) = 102334155

これで判るように、**2つのアイソレートが同時に同じ関数をアクセスしても、双方が干渉することなく計算が実行されている**。親は41.090秒目から849mS後の41.939秒目に計算を終了し、子供は41.089秒目から838mS後の41.927秒目に計算を終了している。これは子供がfibo関数を使った場合と殆ど処理時間に相違がない。

このようにDartのグローバル関数はリエントラントになっており、スレッド安全になっているように見える。但し**グローバル変数はそうはいかないので、共有してはいけない**。アイソレート間でリソースを共有するコードは作成可能であるが、その場合はスレッド安全に留意しなければならない。

Dartiumでの実行またはdart2jsによるChrome上での実行

1. この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、Githubからダウンロードからダウンロードし、IntelliJなどのIDEから\dart_code_samples-master\apps\IsolateTestFibonacciのフォルダを開くと良い。3つのファイル
   1. IsolateTestFibonacci.dart
   2. IsolateTestFibonacci_child.dart
   3. IsolateTestFibonacci.html

   の3つのファイルが\dart_code_samples\apps\IsolateTestFibonacci\webのディレクトリに含まれていることを確認する。

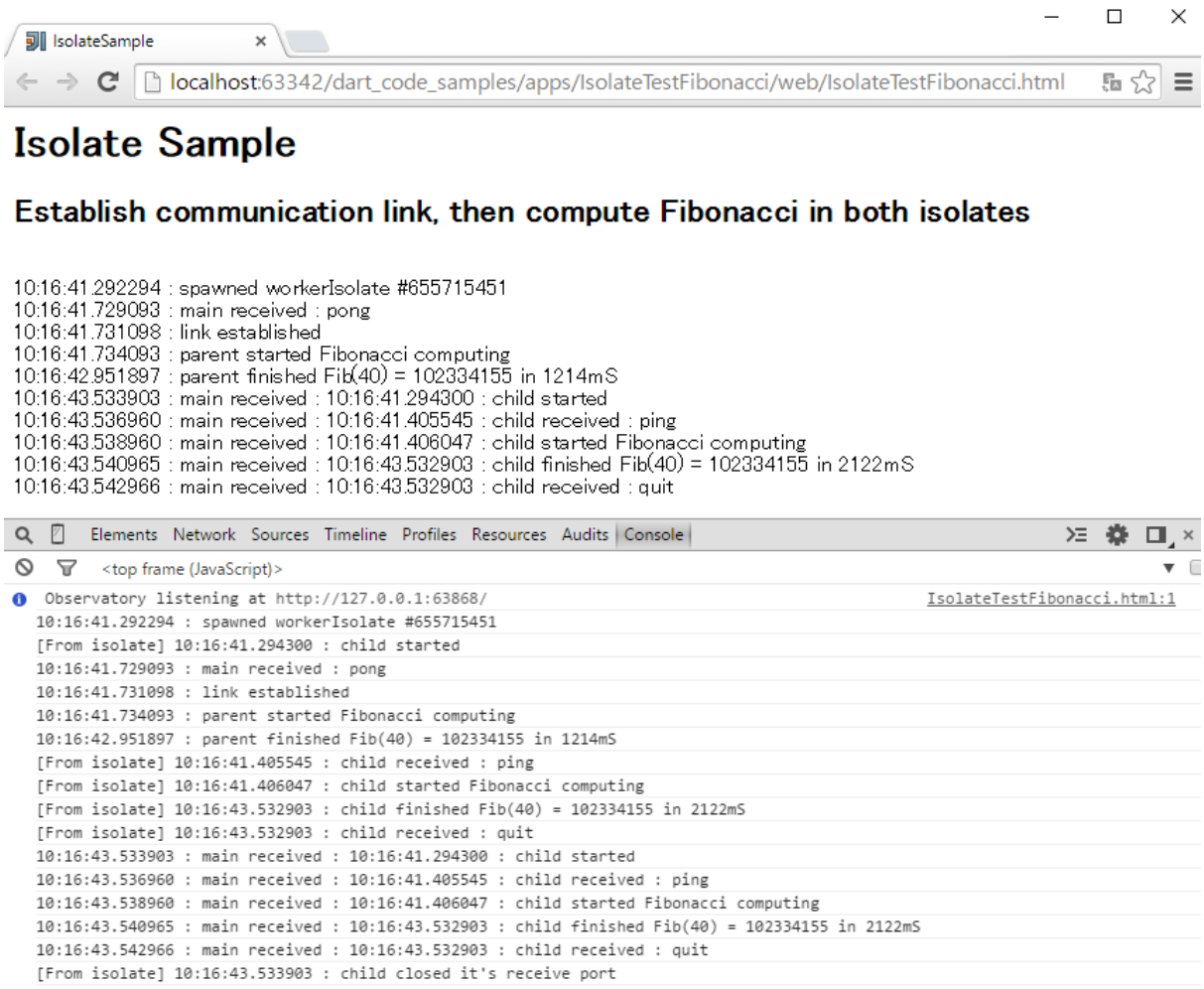
2. 未だPubを組み入れていないときは\dart_code_samples\apps\IsolateTestFibonacci\pubspec.yamlを右クリックして、Pub: Get Dependenciesを実行しすると、必要なパッケージ（ブートストラップ・コードの為のpackageフォルダ）が\dart_code_samples\apps\IsolateTestFibonacci\packages\browserとして組み込まれる。

3. IsolateTestFibonacci.htmlを右クリックして、Open in Browser → Dartiumをクリックする。

4. あるいはDartiumをあらかじめ開いておいて、Dartiumから直接IsolateTestFibonacci.htmlを開く（例えばfile:///C:/...../IsolateTestFibonacci/IsolateTestFibonacci.html）。

下図はDartiumでの実行結果である。ここではJavaScriptコンソールも画面下部に表示してある：

Dartiumでの実行



VM実行と比較してFibonacci関数の処理時間が長くなり、また親と子供のアイソレートでは計算時間に大きな開きがある（但し別のマシンではともに1.3秒台で開きはなかった）。

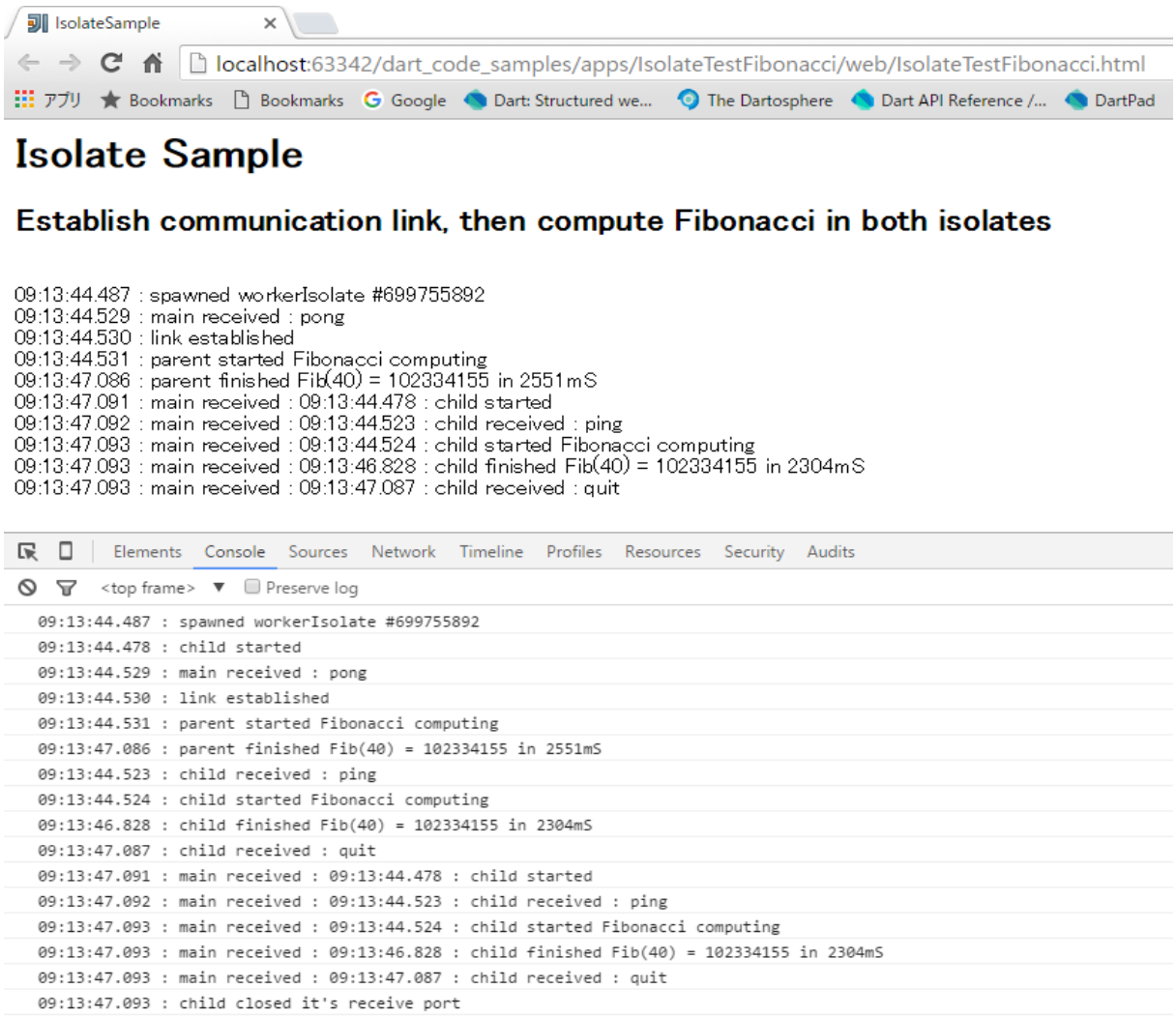
IDE上でChromeでJavaScriptに変換しこれを実行させてみよう。

1. IsolateTestFibonacci.htmlを右クリックして、Open in Browser → Chromeをクリックする。

そうするとIDEはJavaScriptに変換しChromeを呼び出しこのアプリケーションを実行させる。この場合は変換されたJSファイルはIDE上には表示されない。

下図はその実行例である。この場合は親と子の処理速度の開きは少なくなっている。

Chromeでの実行



Dart2JSで.JSファイルたちを用意してChromeから直接IsolateTestFibonacci.htmlを実行させようとするとエラー（Uncaught SecurityError: Failed to create a worker:）になる問題がある。Firefoxではサーバ経由ではなくて直接ブラウザからアクセスできる。これはDartの問題ではなくてブラウザの問題である。しかしながら通常のアプリケーションではサーバ経由でファイルを取得するので、問題はなかろう。

18.8節　　タイマ・アイソレート (Timer Isolate)

アイソレートのサンプルとして、タイマ機能をアイソレートとして分離させたタイマ・アイソレートを紹介する。このライブラリを使用するとミリ秒といった高精度のタイマが実現できる。しかもアイソレートとすることで、メインの処理のオーバヘッドを最小化できる。現にFirefoxではJavaのThreadがタイマのメカニズムとして使われている。タイマがカウントしている時間が設定された設定時間（複数）を超えたら親のアイソレートに即座にそれを通知する。このタイマはまた停止だけでなく、一時停止と再開も可能である。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IntelliJなどのIDEから\dart_code_samples-master\apps\TimerIsolateSampleのフォルダを開くと良い。

このタイマ・アイソレートはタイマ関係のAPIが未だ貧弱だった2012年4月に作成されたので、当時はかなり有用なアイソレートだった。しかしながら現在はStopwatchなどのAPIは整備されており、このプログラムは単にアイソ

レートを使ったアプリケーション開発のための参考用のものでしかない。現在使用しているStopwatchクラスはMHzといった周波数が使われており（いわゆる高分解能モード）、またそのAPIも強化されているのでこのようなアプリケーションに適している。現にChromeではWindowsを高分解能モードにしてDateを実現している。以前使用していたClockインターフェイスは2012年8月に廃止されている。

一方マウス・イベントなどの外部イベント以外に、監視やポーリングなどの目的で一定時間毎に何かを行うアプリケーションが多い。そのような用途の為にDartでは以下のものが用意されている：

- dart:htmlのWindowインターフェイス（HTML DOMのWindowオブジェクトに対応）の
  - int **setInterval**(void handler(), int timeout)
  - int **setTimeout**(void handler(), int timeout)
  - void **clearInterval**(int handle)
  - void **clearTimeout**(int handle)
- dart:ioの**Timer**インターフェイス（注意：このライブラリはいろんな推移があったが、最終的には2013年1月に新しいdart:asyncというライブラリに移された）

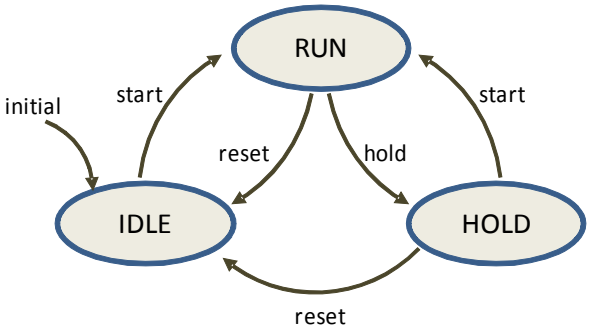
これらは一定時間がきたらハンドラ（あるいはコールバック）関数が呼び出される。これらのAPIは重複しているので、近い将来Timerひとつにしてコア・ライブラリに含められる議論がされているが、まだ決着していない。2013年2月13日に、window.setTimeout及びwindow.setIntervalを廃止して、Timerに一本化するという計画が発表されている。

このアイソレート・コードの使用上の注意点としては：

- なるべく処理能力の高いコンピュータを使用する。OSとしてはWindows Vista、Windows Server 2008、Windows 7あるいはそれ以降でないと高分解能クロックは利用できない。
- クライアント・サイドでこれを使うときは、高精度なDateを実現しているDartiumまたはChromeブラウザ（JS変換して）を使用する。サーバ・サイドで使うときはDateがどのようにそのOS上で実装されているかを確認しなければならない。
- クライアント・サイドで使うときは、前述のように他の処理がタイマ・アイソレートへの定期的なメッセージの送信を遅らせてしまうと、タイマの精度が落ちる。
- タイマ・アイソレート自体は高精度であっても、それを使う側がその精度を活かせないと意味がない。メインの処理はそれに応じた短時間で処理できるようにする。つまりこのアイソレートからのメッセージを受信のコールバック関数が即座に処理できるよう、他の処理を極力その前に済ませておく。
- 実際の使用の前にログを調べて、いろんな状態でも所定の精度が得られることを確認する。

タイマの状態遷移

タイマは次のような状態遷移をとる：



- IDLE: 初期状態であり、タイマの値はゼロにセットされる。設定されているトリップ時間たち(tripTimes)に対応した達時フラグ(expiredFlags)はリセットされる。他のRUN及びHOLDの状態からはresetコマンドによりこの状態に遷移する
- RUN: タイマが1ミリ秒ステップで動作している状態であり、設定されているトリップ時間に達したかどうかを常にチェックする。達したら直ちにそれに対応した達時フラグをセットし、これをポート経由で相手に通報する。他のIDLE及びHOLDの状態からはstartコマンドによりこの状態に遷移する
- HOLD: タイマが動作を停止している状態であり、タイマの値は停止したときの値を保持する。また設定されているトリップ時間たち(tripTimes)に対応した達時フラグ(expiredFlags)も保持される。RUN状態からholdコマンドによりこの状態に遷移する

アイソレート間で通信するメッセージ

親からタイマ・アイソレートへの下り方向メッセージは次のようである:

1. "tick" : これはアイソレートに対するティックである。前述のように現時点ではDartium上のアイソレートには周期的イベントを与えるAPIが存在しないので、暫定的に親からメッセージ受信という形でそのイベントを与えている
2. "reset" : resetコマンドに対応するStringオブジェクトのコマンド。HOLD及びRUN状態で受け付けられる
3. "start" : startコマンドに対応するStringオブジェクトのコマンド。IDLE及びHOLD状態で受け付けられる
4. "hold" : holdコマンドに対応するStringオブジェクトのコマンド。RUN状態で受け付けられる
5. "?state" : カウンタの現在の状態を問い合わせるStringオブジェクトのコマンド
6. "?elapsed" : カウンタの現在の値を問い合わせるStringオブジェクトのコマンド
7. "?expiredFlags" : 現在の達時フラグのリストを問い合わせるStringオブジェクトのコマンド
8. "?tripTimes" : 現在設定されているトリップ時間のリストを問い合わせるStringオブジェクトのコマンド
9. {'tripTimes' : [カンマで区切ったmS単位のトリップ時間たち]} : トリップ時間のリストを送信するMapオブジェクト。IDLEの状態でのみ受け付けられる。受け付けたら暗示的にresetコマンドが実行される
10. "quit" : タイマ・アイソレートの受信ポートをクローズさせる

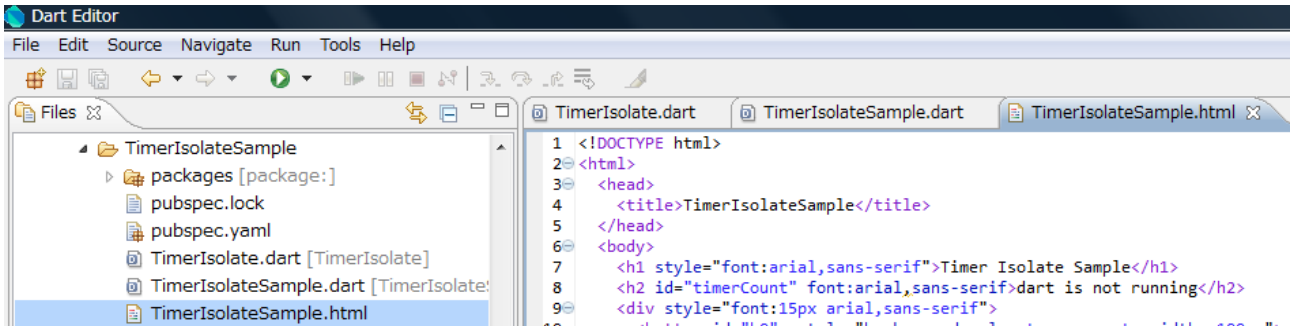
タイマ・アイソレートからの上り方向メッセージは次のようである:

1. "resetOk" : resetコマンドを受け付けたことを通知するStringオブジェクト
2. "startOk" : startコマンドを受け付けたことを通知するStringオブジェクト

3. "holdOk" : holdコマンドを受け付けたことを通知するStringオブジェクト
4. {'state' : 状態を示す整数} : "?state"に対応したMapオブジェクトの応答
5. {'elapsed' : タイマ値を示す整数} : "?elapsed"に対応したMapオブジェクトの応答
6. {'expiredFlags' : ブールたちのリスト} : "?expiredFlags"に対応したMapオブジェクトの応答
7. {'tripTimes' : mS単位のトリップ時間たちのリスト} : "?tripTimes"に対応したMapオブジェクトの応答
8. {'expired' : 達時したタイマ値を示す整数} : 検出されたら直ちに通知されるMapオブジェクト

Dartiumでの実行例

1. 最初にダウンロードしたファイルをIntelliJなどのIDE上に下図のように配置する:



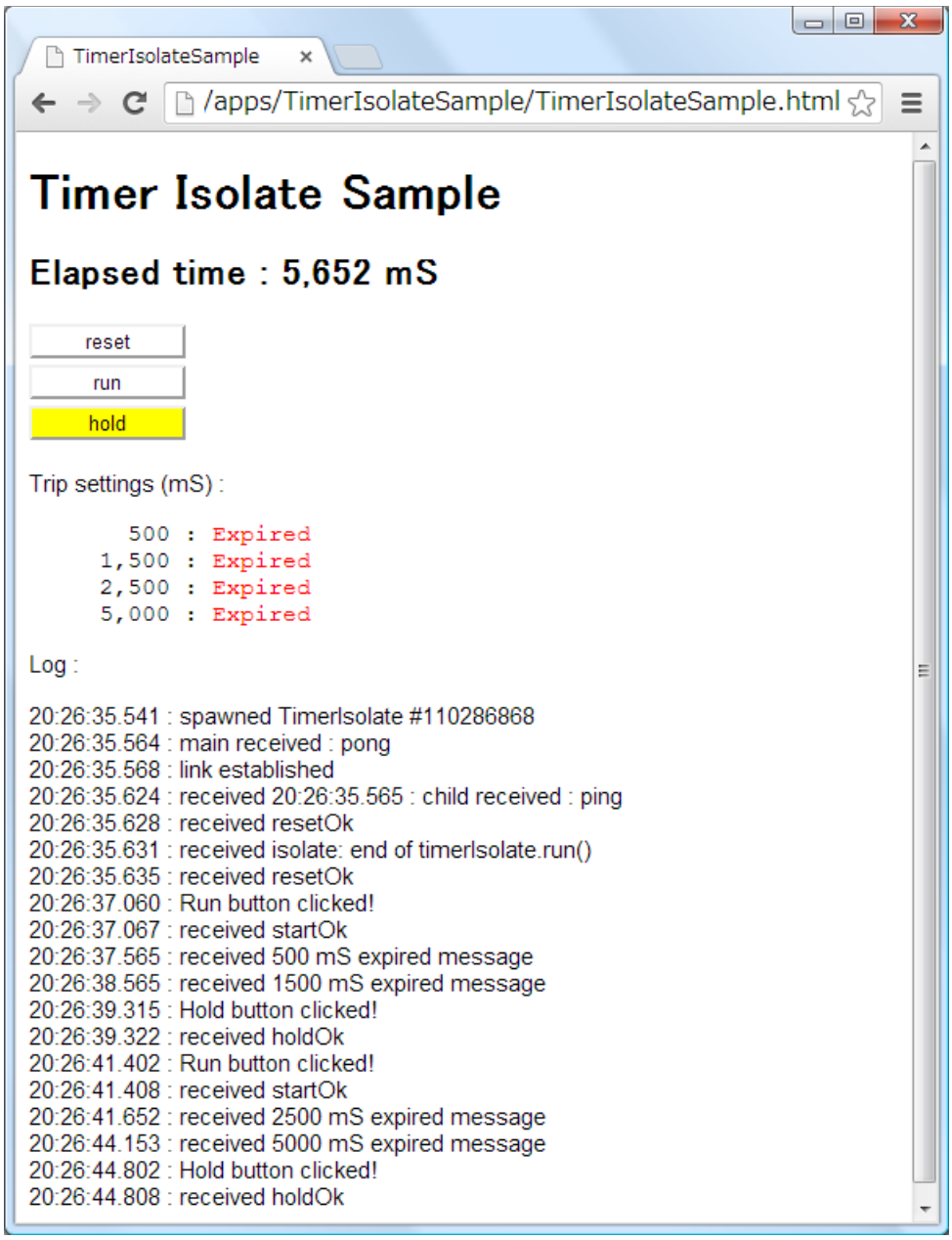
Dart.jsは「dat.jsブートストラップ・コード」の節で説明したように、pubを使わないときはbrowserというフォルダに収容することが推奨される。

2. 次にTimerIsolateSample.htmlを右クリックして、Run in Dartiumを選択する。

代替的な手段は直接Dartiumブラウザからこのアプリケーションを起動することである。

1. TimerIsolateSample.htmlを左クリックしてCopy File Pathを選択する。
2. 次にDartiumのアドレス・バーにfile:///を入力し更にコピーしたパスを貼り付ける(例えば file:///C:/dart_applications/TimerIsolateSample/TimerIsolateSample.html)。
3. ブラウザにこのアドレスをアクセスさせる。

下図はその実行例(Dartiumで直接実行)である:



ログの部分を抜き出すと次のようになっている:

001 20:26:35.541 : spawned TimerIsolate #110286868
002 20:26:35.564 : main received : pong
003 20:26:35.568 : link established
004 20:26:35.624 : received 20:26:35.565 : child received : ping
005 20:26:35.628 : received resetOk
006 20:26:35.631 : received isolate: end of timerIsolate.run()
007 20:26:35.635 : received resetOk
008 20:26:37.060 : Run button clicked!
009 20:26:37.067 : received startOk
010 20:26:37.565 : received 500 mS expired message
011 20:26:38.565 : received 1500 mS expired message
012 20:26:39.315 : Hold button clicked!
013 20:26:39.322 : received holdOk
014 20:26:41.402 : Run button clicked!
015 20:26:41.408 : received startOk

016 20:26:41.652 : received 2500 mS expired message
017 20:26:44.153 : received 5000 mS expired message
018 20:26:44.802 : Hold button clicked!
019 20:26:44.808 : received holdOk

9行目のスタート・コマンドを受け付けたというメッセージを受けた時刻、10行目の0.5秒のトリップを検出したというメッセージを受けた時刻から、11行目の1.5秒のトリップ報告メッセージを受けた時刻、17行目の2.5秒のトリップ報告メッセージを受けた時刻、そして最後に18行目の5秒のトリップ報告メッセージを受けた時刻との時間差は、所定の時間差に対して2ミリ秒の精度で納まっていることが確認できよう。また9行間から13行目までの時間差2,255と15行目から17行目の時間差2,745を加えると、ちょうど5,000（5秒間）になる。

これらのコードの詳細は省略する。興味ある読者は本資料の概要のページの下のほうにある表からこれらのコードをアクセスして開いて追って頂きたい。

18.9節　　複数のアイソレートの管理

まず複数のアイソレートを走らせるサンプルを紹介する。

Code_18.8.dart

```
import 'dart:isolate';
import 'dart:async';

costlyIsolate(reply){
    new Timer(new Duration(seconds: 1), () => reply.send('costly'));
}

expensiveIsolate(reply){
    new Timer(new Duration(seconds: 2), () => reply.send('expensive'));
}

lengthyIsolate(reply){
    new Timer(new Duration(seconds: 3), () => reply.send('lengthy'));
}

costlyQuery() {
  var reply = new ReceivePort();
  var completer = new Completer();
  Isolate.spawn(costlyIsolate, reply.sendPort)
    .then((_) => reply.first)
    .then((msg) {completer.complete(msg);});
  return completer.future;
}

expensiveWork() {
  var reply = new ReceivePort();
  var completer = new Completer();
  Isolate.spawn(expensiveIsolate, reply.sendPort)
    .then((_) => reply.first)
    .then((msg) {completer.complete(msg);});
  return completer.future;
}
```

```
lengthyComputation() {
  var reply = new ReceivePort();
  var completer = new Completer();
  Isolate.spawn(lengthyIsolate, reply.sendPort)
    .then((_) => reply.first)
    .then((msg) {completer.complete(msg);});
  return completer.future;
}

void main() {
/*
  // process each time it completes
  costlyQuery().then((value){print(value);});
  expensiveWork().then((value){print(value);});
  lengthyComputation().then((value){print(value);});
*/

  // wait for all complete
  Future.wait([costlyQuery(), expensiveWork(), lengthyComputation()])
  .then((List values) {
    print(values);
  });

}
```

この場合はcostlyQuery()、expensiveWork()、及びlengthyComputation()の3つの仕事を各々costlyIsolate()、expensiveIsolate()、及びlengthIsolate()の3つのアイソレートのトップ・レベル関数に分担させている。各アイソレートは親からのメッセージ（ここでは空の文字列）を受信してから1秒後、2秒後、及び3秒後にそれを完了し文字列を返信している。これらのアイソレートは親が終了しない限り存続して親からのメッセージを待つ。costlyQuery()はCompleterオブジェクトを用意し、costlyIsolateアイソレートを生成し、port.callというメソッドでそのアイソレートに対し単発のメッセージを送信し、返事が返ってきたらそれを値としてそのFutureを完了させる。port.callはFutureオブジェクトを返すので、呼び出し側はthenでそのFutureオブジェクトの完了を知ることができる。

3つの仕事が完了するまで待つにはFuture.waitというメソッドが利用できる。これらは既に「イベント処理」の章で説明してある。

18.10節　Dart Fletch

Dartチームは2015年1月からアイソレートとは別の並行性メカニズムであるFletchと呼ばれるプロジェクトを開始している。Fletchは「〈矢に〉羽根をつける」という意味であり、Dartの羽に絡めて付した名前だと思われる。

この件に関しては進捗に応じて報告する予定である。

Fletchの設計の核となっているのがcoroutine / thread / processである。これらに関してはhttps://github.com/dart-lang/fletch/wiki/Coroutines-and-Threads及びhttps://github.com/dart-lang/fletch/wiki/Processes-and-Isolatesに次のように記されている：

- Process

プロセスはこれまでのアイソレートにほぼ相当したものと考えることができる。Fletchは複数のプロセスの並行実行

に対応している。各プロセスは各々のヒープを持っており、プロセスたちは他のプロセスとはメッセージ渡しでのみ関わりあう。Fletchはネーティブなスレッドたちの遥かに小さなのセットで非常に大量のプロセスたちをスケジュールできる。あるプロセスをブロックすることは、現在実行しているプロセスのネーティブなスレッドをブロックすることにはならない。Fletchは現在32ビットのマシンで 450,000以上のプロセスを並行して実行させ、それらをそのシステムのコア数に合った幾つかのネーティブなスレッドで走るようスケジュールできる（例えば4コアのマシンなら4つのネーティブなスレッドで実現する）。

次の例はFletchでどのようにプロセスたちを産み付けるかを示している:

```
main() {
 for (int i = 0; i < 50000; i++) {
  Process.spawn(fib, 12);
 }
}

int fib(n) {
 if (n <= 1) return 1;
 return fib(n - 1) + fib(n - 2);
}
```

- Isolate

アイソレートはプロセスより若干高いレベルで動作する。アイソレートのAPIを使うと外部からアイソレートたちを停止、再開、終了させることが可能であり、現在低レベルのプロセスの上にこれを乗せるように検討中とのことである。現時点（2015年2月15日）ではFletchは dart:isolateライブラリに対応する様にはなっていない。

- Coroutines

Fletchはcoroutineに組み込みで対応している。Coroutinesは関数に似たオブジェクトであり、複数の値たちを作り出すことができるので、発生器たち(generators)と密に関連している。あるcoroutineを作りだしたら、それはそのコードの現在の場所で保留に置かれ、それがあとで再開されるまでその実行スタックは保持される。あるcoroutineが実行・完了しその最終値を返すと、それは官僚と見做され、再度再開することはない。以下はcoroutineの使用法を示す簡単な例である:

```
var co = new Coroutine((x) {
 Expect.equals(1, x);
 Expect.equals(3, Coroutine.yield(2));
 return 4;
});
Expect.isTrue(co.isSuspended);
Expect.equals(2, co(1));
Expect.isTrue(co.isSuspended);
Expect.equals(4, co(3));
Expect.isTrue(co.isDone);
```

- スレッド

Fletchでは同期的プログラミングができ、各プロセスがあるイベントが起きるまで待っているときは停止する様にできる。あるプロセスの最初から最後まで停止してしまうことを避けるため、Fletchはあるプロセス内で相互協調動作するようスケジュールされたスレッドたちが使え、それらが互いに独立して停止されることができる。このことは、複数の分離した制御フローたちが、これまでの非同期のコーディングでなされていた暗示的な方法ではなくて分離したスレッドたちとして明示的にモデル化できることを意味する。

以下はある共有チャンネルを介して2つのスレッドが通信しあう例を示す。channel.receiveを呼ぶときはブロック動作なので、コンシューマ側のスレッドはプロジューサ側がある値を作りだすまでは実行を継続できない。

```
main() {
 var channel = new Channel();
 Thread.fork(() => producer(channel));
 for (int i = 0; i < 10; i++) {
  print(channel.receive());
 }
}

producer(Channel channel) {
 for (int i = 0; i < 10; i++) {
  channel.send(i);
 }
}
```

18.11節　関連APIの和訳

dart:isolateライブラリ

Library **dart:isolate**

dart:isolateライブラリはアイソレートの産み付け及び通信にかかわるAPIを定めている。

Dartの総てのコードはアイソレートのコンテキスト（環境）内で走る。各アイソレートは各々のヒープを持つ、即ちこれはグローバルのものも含めてメモリ内の総ての値はそのアイソレートのみがアクセスできることを意味する。アイソレートたち間の通信に使える唯一のメカニズムはメッセージ渡しである。メッセージはポートを介して送信される。本ライブラリでは通信チャンネルの受信端を表現するReceivePort、及び送信端を表現するSendPortを規定している。

新規のアイソレートを産み付けるにはIsolateクラスのspawnFunction及びspawnUriの2つのメソッドが使える。spawnFunctionは現在のアイソレートと同じソース・コードを使う新規アイソレートを生成し、spawnUri は独立して書かれたアイソレートを産み付けることができる。

そのアイソレートが同じスレッドであるいは別のスレッドで走るかを示す手段は現在このAPIには存在していない。下位層のシステムがそのアイソレートを適正にスケジューリングする。近い将来我々はDOMアイソレートたちを生成する為のAPIを追加する予定である。これらはDOMアクセスを共有するアイソレートたちである。総てのDOMアイソレートはUIスレッド上で走ることになる。

このライブラリは以下のもので構成されている：

抽象クラスたち：

- RawReceivePort
- ReceivePort
- SendPort

クラス:
- Isolate

例外:
- IsolateSpawnException

Dart:isolate.Isolate

Isolateクラス
隔離された(isolated)Dartの実行環境。総てのDartコードはあるアイソレート上で走り、各コードは同じアイソレートからのみのクラスと値たちにアクセスできる。個々のアイソレートたちはポート(ReceivePort, SendPort参照)を介して値をやり取りできる。 あるIsolateオブジェクトはあるアイソレートへの参照であり、通常は現行のアイソレートとは別のものになる。これは他のアイソレートを表現し、また他のアイソレートをコントロールできる。 新たなアイソレートを産みつける(spawn)ときは、産みつけるアイソレートはその産み付け(spawn)操作が成功すればその新規のアイソレートを表現するあるIsolateオブジェクトを受理する。 アイソレートたちはそれ自身のイベント・ループ内で走り、各イベントはより小さなタスクたちをネストされたマイクロスタック・キューのなかで走らせることができる。 Isolateオブジェクトにより、それが表現しているアイソレートのイベント・ループを他のアイソレートが制御し、また該アイソレートを検査できるようにしている。例えば該アイソレートを止めたり、該アイソレートが補足されなかったエラーを起こした際にイベントを取得したりできる。 controlPortはそのアイソレートの制御にアクセスでき、pauseCapabilityとterminateCapabilityは何らかの制御操作へのアクセスをガードする。spawn操作により用意されたIsolateオブジェクトは該アイソレートの制御に必要な制御ポートと制御機能を持つことになる。必要に応じそれらの機能の幾つかを有しない新規のアイソレートのオブジェクトを生成することができる。 IsolateオブジェクトはSendPort上で送信することはできないが、制御ポートと制御機能は送信可能であり、受信しているポート側のアイソレートの中で新たに機能するIsolateオブジェクトを生成するのに使える。

常数	
int BEFORE_NEXT_EVENT = 1	pingとkillの引数:次のイベントの前でのアクションを要求する
int IMMEDIATE = 0	pingとkillの引数:即座のアクションを要求する
static属性(すべてリード・オンリー)	
Isolate **current**	現行Isolateを返す。 該アイソレート・オブジェクトにより、そのアイソレートの検査・停止あるいは止めるのに必要な機能へのアクセスができるし、他のアイソレートたちにこれらの機能を許すことができる。
Future<Uri> **packageConfig**	現行アイソレートのパッケージ・ルートを返す(もしあれば)。もし該アイソレートがpackageRootを使っているか、該アイソレートがパッケージ解決の設定がまだされていないときはnullを返し、そうでないときはパッケージ設定URIを返す。

Future<Uri> **packageRoot**	現行アイソレートのパッケージ・ルートを返す(もしあれば)。もし該アイソレートがpackageRootを使っているか、該アイソレートがパッケージ解決の設定がまだされていないときはこのゲッタはnullを返し、そうでないときはパッケージ・ルート(パッケージURIたちが解決されたディレクトリ)を返す。
staticメソッド	
Future<Isolate> **spawn**(void entryPoint(message), message)	現在走っているのアイソレートと同じコードを共有するアイソレートを生成し産み付ける。 引数のentryPointは産み付けられたアイソレートの開始点を指定する。これはstaticなトップ・レベルの関数かあるいは引数を持たないstaticメソッドかでなければならない。関数クロージャを渡すことは許されていない。 このエントリ・ポイント関数は初期メッセージで呼び出される。産み付け側と産み付けられた側間での相互通信が可能になるよう、通常この初期メッセージにはSendPortのオブジェクトが含められる。 このメソッドはIsolateのインスタンスを持ったFutureオブジェクトを返す。このIsolateのインスタンスは産み付けられたアイソレートをコントロールするのに使うことができる。
Future<Isolate> **spawnUri**(Uri uri, List<String> args, message)	指定したURIのライブラリからのコードで実行するアイソレートを生成し産み付ける。 このアイソレートは指定されたURIのライブラリのトップ・レベルにあるmain関数の実行を開始する。 ターゲットとなるmainは次の3つのシグネチュアの一つをとる: • main() • main(args) • main(args, message) messageの引数が存在するときはそれは初期メッセージにセットされる。argsが存在するときは、それらはargsリストにセットして渡される。 このメソッドはIsolateのインスタンスを持ったFutureオブジェクトを返す。このIsolateのインスタンスは産み付けられたアイソレートをコントロールするのに使うことができる。
Future<Uri> **resolvePackageUri**(Uri packageUri)	あるpackage: URIを非パッケージ Uriにマップする。 現行アイソレート内に該パッケージからの有効なマッピングが存在しないときは、この呼び出しはnullを返す。非package: URIはなにもそのまま返される。
コンストラクタ	
Isolate(SendPort controlPort, {Capability pauseCapability, Capability terminateCapability})	制限された機能のセットを有する新規のIsolateオブジェクトを生成する。 このポートは別のIsolateオブジェクトから持ってきているので、あるアイソレートの制御ポートでなければならない。 機能たちはオリジナルのアイソレートで使える機能たちのサブセットでなければならない。あるアイソレートの機能たちはそのアイソレートにロックされ、それ以外には効果を持たないいので、機能たちはその制御ポートと同じアイソレートからきたものでなければならない。 総ての利用可能な機能たちが含まれている場合は、その振る舞いは制御ポートと機能たちによって定義されているので、新規のオブジェクトを生成する理由がな

	い。
属性	
SendPort **controlPort**	リード・オンリー 該アイソレートに制御メッセージを送信するのに使われる制御ポート。このクラスは制御ポートに制御メッセージを送信するヘルパー機能を用意している。 制御ポートは該アイソレートを識別する。
Stream **errors**	該アイソレートからの補足されなかったエラーたちの放送ストリームを返す。各エラーはこのストリーム上のエラー・イベントとして提供される。 実際のエラー・オブジェクトとstackTraceたちは必ずしも実際のアイソレートのなかと同じオブジェクト型にはならないが、Object.toStringは常に同じ結果となる。 このストリームはaddErrorListenerとremoveErrorListenerをもとにしている。
int **hashCode**	このオブジェクトのためのハッシュ・コード
Capability **pauseCapability**	リードオンリー 該アイソレートをポーズさせる能力を与える能力。 この機能(capability)はpauseによって使われる。この機能が該アイソレートにとって正しいpause機能でないとき（もしこのcapabilityがnullの場合を含む）は、このポーズ呼び出しは効果を持たない。 該アイソレートがポーズ状態から開始したときは該アイソレートを再開させるのに引数としてこのcapabilityを使用する。
Type **runtimeType**	このオブジェクトの実行時の型を表現している。
Capability **terminateCapability**	リードオンリー 該アイソレートを終了させる能力を与える能力。 この機能(capability)はkillまたはsetErrorsFatalによって使われる。この機能が該アイソレートにとって正しい終了機能でないとき（もしこのcapabilityがnullの場合を含む）は、このポーズ呼び出しは効果を持たない。
メソッド	
void **addErrorListener**(SendPort port)	該アイソレートの補足されなかったエラーたちをportに送り返すよう要求する。 エラーは2つの要素のリストとして送り返される。最初の要素は該エラーを表現するStringで、通常該エラーのtoStringを呼ぶことで生成される。2番目の要素はそれに伴うスタック・トレースを表現したString、あるいはスタック・トレースが得られないときはnullとなる。これをStackTraceオブジェクトに戻すには StackTrace.fromStringを使用すること。 1回以上同じポートを使ってリスンしても何も起きない。これは各エラーを1回取得する。 アイソレートたちは並行して走るので、エラー・リスナが確立される前にそれが終了してしまう可能性がある。それを避けるにはアイソレートのポーズを開始させ、リスナを付加し、次に該アイソレートを再開させる。
void **addOnExitListener(**SendPort responsePort, {Object response})	該アイソレートにたいしそれが終了したときにresponsePort上で応答を送信するよう要求する。 該アイソレートはそれが終了する前の最後のこととしてresponsePort上で応答メッセージを送信する。このメッセージが送信された後はさらなるコードを走ることはない。 1回以上同じポートを付加しても付加された最後の値を使って一つのメッセージを受信するだけである。 該アイソレートが既に死んでいるときはメッセージは送信されない。該アイソレート

	に応答を送れないときは、この要求は無視される。総てのアイソレートに送信できるシンプルな値たち（null、ブール値、数値、あるいは文字列のような）のみを使うことを推奨する。 アイソレートたちは並行して走るので、エラー・リスナが確立される前にそれが終了してしまう可能性がある。それを避けるには、spawn関数うに対応したパラメタを使うか、またはアイソレートのポーズを開始させ、リスナを付加し、次に該アイソレートを再開させる。
void **kill**({int priority: BEFORE_NEXT_EVENT})	該アイソレートを終了させることを要求する。 該アイソレートはそれ自身を終了することが要求される。priority引数はいつこれが起きねばならないかを指定する。priorityはIMMEDIATEまたはBEFORE_NEXT_EVENTのどれかのこと。終了はpriorityに基づいて複数回行われる。 • IMMEDIATE: 該アイソレートは極力すぐに終了する。制御メッセージたちは順に処理されるので、このアイソレートからこれまで送信されたすべてのメッセージは処理される。 BEFORE_NEXT_EVENTで送信されたときは終了はあその前に起きる。別のイベントの実行中であってもその前にクリーンに終了する手段を有しているときはその前に終了する可能性がある。 • BEFORE_NEXT_EVENT: 終了は受信アイソレートが現行イベントの後で、そして既にスケジュールされている制御イベントが終了した後で、イベント・ループに次回戻る際に起きるようにスケジュールされる。
dynamic **noSuchMethod**(Invocation invocation)	存在しないメソッドまたは属性がアクセスされたときに呼び出だされる。各クラスはカスタムのふるまいを提供するためにnoSuchMethodをオーバライドできる。 値が返されるときはそれはオリジナルの呼び出しの結果となる。デフォルトの振舞いは NoSuchMethodErrorのスローである。
Capability **pause**([Capability resumeCapability])	該アイソレートにポーズすることを要求する。 該アイソレートはそのイベント・キューをポーズさせることでイベント処理を停止させねばならない。この要求は最終的に該アイソレートを何もしなくしてしまう。現行アイソレートからこのアイソレートに他のメッセージたちがその後送信される前に処理されるが、保証はされない。。 送信されたものの未だ実行されていないメッセージたちが到達する前でのイベント・ループはポーズされ得る。 もしresumeCapabilityが指定されているときは、それはポーズを識別するのに使われ、resumeを使ってポーズを再度終了するのに使ってはならない。そうでないときは新規の再開機能(resume capability)が生成され、返される。 同じ機能(capability)を使ってあるアイソレートが一度以上ポーズされたときは、該ポーズを終了させるにはその機能でのただ一つの再開(resume)が必要である。 一つ以上の機能を使ってあるアイソレートがポーズされたときは、そのアイソレートが再開する前にそれらの総てが個々に終了されていなければならない。 該ポーズを再開するのに使われねばならない機能(capability)が返される。
void **ping**(SendPort responsePort, {Object response, int priority:	該アイソレートに対しresponsePort上で応答を送信するよう要求する。 もし該アイソレートが活きておれば、そのアイソレートは最終的に応答ポート上で

IMMEDIATE})	応答(デフォルトはnull)を返す。 優先度(priority)は IMMEDIATEまたはBEFORE_NEXT_EVENTのどれかである。pingのタイプに依存して応答は複数回送信されうる。 • IMMEDIATE: この制御メッセージを受けたら該アイソレートは極力すぐに応答する。同じアイソレートからのこれまでの制御メッセージがあればそれらを受信してからとなるが、別のイベントの実行中においては応答し得る。 • BEFORE_NEXT_EVENT: 応答は該アイソレートが現行イベントの後で、そして既にスケジュールされている制御イベントが終了した後で、イベント・ループに次回戻る際に起きるようにスケジュールされる。 もし応答がそのアイソレートに送信できなかったときは該要求は無視される。シンプルな値たち(null、ブール値、数値、あるいは文字列のような)のみを使うことを推奨する。
void **removeErrorListener**(SendPort port)	ポートを介して補足されなかったエラーたちをリスンすることを停止する。 該ポートはaddErrorListenerでエラーをリスンしているポートでなければならない。この呼び出しは該アイソレートに対し該ポートを使ってエラーを送信することを停止することを要求する。 addErrorListenerを介して一回以上同じポートが渡されているときは、エラー受信を停止させるには唯一度の removeErrorListener呼び出しで良い。 送信ポートの最後で受信ポートを閉じても該アイソレートのエラー送信を停止することにならず、それらは捨てられるだけである。
void **removeOnExitListener**(SendPort responsePort)	該アイソレートからのexitメッセージをリスンすることを停止する。 ある呼び出しが同じ送信ポートで以前addOnExitListenerのために使われていたときは、これは該ポートの登録解除(unregister)となり、該アイソレートが終了したときにメッセージを受信しなくなる。該アイソレートによってこの操作が完全に処理されるまでは応答は送信され得る。
void **resume**(Capability resumeCapability)	ポーズ状態のアイソレートを再開させる。 resumeCapabilityを使って要求されたポーズを終了させるメッセージをあるアイソレートに送信する。 総てのアクティブなポーズ要求たちがキャンセルされたときは、該アイソレートは通常のメッセージ処理を継続する。 機能(capability)はこのアイソレートをポーズさせる呼び出しで返されたものでなければあならない。そうでないときはこの再開呼び出しは無視される。
void **setErrorsFatal**(bool errorsAreFatal)	補足されなかったエラーが該アイソレートを終了させるかどうかをセットする。 致命的(fatal)なエラーの場合は、補足されないエラーは該アイソレートのイベント・ループを終了させ、該アイソレートを終了させる。

	この呼び出しには該アイソレートのためのterminateCapabilityが必要である。該機能(capability)が正しいものでないときは、何も変更されない。 アイソレートたちは並行動作しているので、エラーが非致命的(non-fatal)とセットされる前にあるエラーのために終了してしまう可能性がある。これを回避するには、spawn関数にたいし対応したパラメタを使うか、あるいは該アイソレートをポーズ状態で開始させ、エラーを非致命的とセットし、次にこのアイソレートを再開させるかのどれかを行う。
String **toString**()	このオブジェクトの文字列表現を返す。

dart:isolate.ReceivePort

ReceivePort abstract class SendPortと組みになってアイソレート間の通信の唯一の手段を構成する。 ReceivePortはsendportというゲッタを有しており、これはSendPortオブジェクトを返す。このSendPortを通して送信されたどのメッセージもそこから生成されたReceivePortに渡される。そこでは、そのメッセージはそのリスナに渡される。 ReceivePortは非ブロードキャッストのストリームである。このことはこれはリスナが登録されるまで到来メッセージたちをバッファリングすることを意味する。ただ一つのリスナのみがメッセージを受信できる。このポートをブロードキャスト・ストリームに変換するにはStream.asBroadcastStreamを参照のこと。 ReceivePortは多くのSendPortを持つこともできる。	
サブクラス	
ReceivePortImpl	
実装	
Stream	
コンストラクタ	
factory **ReceivePort**()	メッセージ受信のための長期に存続するポートを開設する。 ReceivePortは非ブロードキャッストのストリームである。このことはこれはリスナが登録されるまで到来メッセージたちをバッファリングすることを意味する。ただ一つのリスナのみがメッセージを受信できる。このポートをブロードキャスト・ストリームに変換するにはStream.asBroadcastStreamを参照のこと。 受信ポートはその加入をキャンセルすることで閉じられる。
factory **ReceivePort.fromRawReceivePort**(RawReceivePort rawPort)	RawReceivePortからReceivePortを生成する。 与えられたrawPortのハンドラはこの結果の生成中にオーバライトされる。
属性	
final Future<T> **first**	（Streamから継承）

	このストリームの最初の要素を返す。 最初の要素が受信されたあとはこのストリームへのリスニングを停止する。 最初のデータが受信される前にエラーが発生すれば、結果としてのfutureはそのエラーで完了する。 もしこのストリームが空のときは（最初のデータ・イベントよりも前に終了イベントが発生した）、結果としてのfutureはStateErrorで完了する。 このエラーのタイプ以外に関しては、このメソッドはthis.elementAt(0)と等価である。
final bool **isBroadcast**	（Streamから継承） このストリームが放送ストリームかどうかを報告する。
final Future<bool> **isEmpty**	（Streamから継承） このストリームが何らかの要素を含んでいるかどうかを報告する。
final Future<T> **last**	（Streamから継承） このストリームの最後の要素を返す。 最初のデータが受信される前にエラーが発生すれば、結果としてのfutureはそのエラーで完了する。 もしこのストリームが空のときは（最初のデータ・イベントよりも前に終了イベントが発生した）、結果としてのfutureはStateErrorで完了する。
final Future<int> **length**	（Streamから継承） このストリームの中の要素数を計数する。
final SendPort **sendPort**	この受信ポートに送信する送信ポートを返す。
final Future<T> **single**	（Streamから継承） 単一の要素を返す。 もしこのストリームが空または一つ以上の要素を有しているときはStateErrorをスローする。
メソッド	
Future<bool> **any**(bool test(T element))	（Streamから継承） このストリームが用意している要素のどれかをtestが受け付けるかどうかをチェックする。 答えが判ったときにこのFutureを完了させる。このストリームがエラーを報告したときは、このFutureはそのエラーで完了する。
Stream<T> **asBroadcastStream**({void onListen(StreamSubscription<T> subscription), void onCancel(StreamSubscription<T> subscription)})	（Streamから継承） thisと同じイベントたちを作り出す複数加入のストリームを返す。 もしこのストリームが既に放送ストリームのときは、それは加工されること無く返される。

	もしこのストリームが単一加入のものだったときは複数の加入を許す新たなストリームを返す。その最初の加入者が付加されたときにこのストリームに加入し、このストリームが終了した、あるいはコールバックがこの加入をキャンセルするまでは加入された状態を維持する。 もしonListenが指定されているときは、それはこのストリームへのもととなっている加入を表現する加入ライクなオブジェクトで呼び出される。onListen呼び出し中にこの加入を保留、再開、あるいは取り消しをすることは可能である。StreamSubscription.asFuture使用を含むイベント・ハンドラの変更はできない。 もしonCancelが指定されているときは、返されたストリームがリスナを持つことを止めたときにonListenと似たように呼び出される。もしそれが新たなリスナを得たときは、onListen関数が再度呼び出される。 たとえば、加入者がいないときにイベント損失を防ぐためにもととなっている加入を保留（ポーズ）するとき、あるいは加入者がいないときにこの加入をキャンセルするためにこのコールバックを使用する。
Future<bool> **contains**(Object needle)	（Streamから継承） このストリームが用意している要素たちのなかでneedleが発生するかどうかをチェックする。 この答えが判っているときはこのFutureを返す。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
abstract void **close**()	thisを閉じる。 このストリームが未だキャンセルされたことがないときは、このイベント・キュー（待ち行列）にクローズ・イベントを追加し、その後の到来メッセージは廃棄する。 もしこのストリームが既にキャンセルされているときは、このメソッドは何の効果も持たない。
Stream<T> **distinct**([bool equals(T previous, T next)])	（Streamから継承） これらが以前の(previous)データ・イベントと等しいときはそのデータ・イベントをスキップする。 返されたストリームは、2つのつながった同じデータ・イベントたちのを用意しないことを除いては、このストリームと同じイベントたちを用意する。 対等性は用意されたequalsメソッドによって判断される。もしこれがオミットされているときは、最後に提供されたデータ要素に対しては'=='が適用される。
Future **drain**([futureValue])	（Streamから継承） このストリーム上の総てのデータを廃棄するが、その完了またはエラーが発生したときにはそれを通知する。 drainを使って加入すると cancelOnErrorはtrueとなる。このことはこのfutureは最初のエラーで完了し次にこの加入を取り消す事を意味する。 doneイベントの場合は、このfutureは指定したfutureValueで完了する。
Future<T> **elementAt**(int index)	（Streamから継承） このストリームの index番目のデータ・イベントの値を返す。

	もしエラー・イベントが発生したときは、このfutureはこのエラーで終了する。 もしこのストリームが閉じる前にindex少ない要素よりもしか用意していないときは、エラーが報告される。 もしこの値が見つかる前にdoneイベントが生じたときは、このfutureはRangeErrorで完了する。
Future<bool> **every**(bool test(T element))	（Streamから継承） Testがこのストリームが用意している総ての要素を受け付けるかどうかをテストする。 この答えが判っているときはこのFutureを完了させる。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream **expand**(Iterable convert(T value))	（Streamから継承） 各要素をゼロまたはそれ以上のイベントたちに変換する新しいストリームをこのストリームから生成する。 到来する各イベントは新しいイベントたちの Iterableに変換され、これらの新しいイベントたちの各々が次に返されるストリームによって順番に送信される。
Future<T> **firstWhere**(bool test(T value), {T defaultValue()})	（Streamから継承） Testにマッチするこのストリームの最初の要素を探す。 testがそれに対しtrueを返したこのストリームの最初の要素で満たされたfutureを返す。 このストリームが終了する前にそのような要素が見つからず、また defaultValue関数が与えられているときは、 defaultValue呼び出しの結果がそのfutureの値となる。 エラーが発生したとき、あるいはこのストリームがmatchを見出すことなく終了し、また defaultValue関数が指定されていないときは、このfutureはエラーを受信する。
Future **fold**(initialValue, combine(previous, T element))	（Streamから継承） combineを繰り返し適用することで値たちのシーケンスを減らす。
Future **forEach**(void action(T element))	（Streamから継承） このストリームの各データ・イベントでactionを実行する。 このストリームの総てのイベントが処理された時点で返されたfutureが完了する。もしこのストリームがエラー・イベントを有しているとき、あるいはactionがスローしたときはこのfutureはエラーで完了する。
Stream<T> **handleError**(Function onError, {bool test(error)})	（Streamから継承） このストリームからの何らかのエラーを取り上げ処理するラッパ・ストリームを生成する。 このストリームがtestをマッチするエラーを送信するときは、次にそれはhandle関数によって取り上げられる（インターセプトされる）。 onErrorコールバックはvoid onError(error)またはvoid onError(error, StackTrace stackTrace)の型式でなければならない。この関数の形式によってこのストリームは

	スタック・トレースの有りか無しかでonErrorを呼び出す。このスタック・トレース引数は、もしこのストリームがそれ自身スタック・トレースなしでエラーを受信したときはnullになる可能性がある。 もしtest(e)がtrueを返すときは[AsyncError] [:e:]は test関数によってマッチがとられる。もしtestがオミットされているときは、各エラーはマッチしていると見做される。 もしそのエラーが取り上げられたときは、このhandle関数はそれに対してどうするかを判断できる。新しい（または同じ）エラーを生起させたいときはスローできるし、あるいは単に戻ることでこのストリームがそのエラーを忘れさせることができる。 あるエラーをデータ・イベントに変換する必要があるときは、データ・イベントを出力シンクに書き込むことでそのイベントを処理するためにより一般的なStream.transformを使用のこと。
Future<String> **join**([String separator = ""])	（Streamから継承） データ・イベントたちの文字列表現の文字列を収集する。 もしseparatorが指定されているときは、それは2つの要素間に挿入される。 このストリーム内の何らかのエラーはこのfutureをエラーで完了させる。そうでないときは、"done"イベントが到着したときに収集した文字列で完了する。
Future<T> **lastWhere**(bool test(T value), {T defaultValue()})	（Streamから継承） Testにマッチするこのストリームの中の最後の要素を探す。 最後にマッチする要素が見つかることを除いてFirstMatchingと似ている。このことはその結果はこのストリームが終了するまでその結果は得られないことを意味する。
abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})	（Streamから継承） このストリームに受信を追加する。 このストリームからのデータ・イベントの各々で、受信者たちのonDataハンドラが呼び出される。もしonDataがnullのときは、何も起きない。 このストリームからのエラーに対しては、onErrorハンドラにはそのエラーを記述したオブジェクトが与えられる。 onErrorコールバックはvoid onError(error)またはvoid onError(error, StackTrace stackTrace)の型式でなければならない。この関数の形式によってこのストリームはスタック・トレースの有りか無しかでonErrorを呼び出す。このスタック・トレース引数は、もしこのストリームがそれ自身スタック・トレースなしでエラーを受信したときはnullになる可能性がある。そうでないときは単なるエラーのオブジェクトで呼び出される。 もしこのストリームが閉じたときは、 onDoneハンドラが呼び出される。 もしcancelErrorがtrueのときは、この受信は最初のエラーが報告されたときに終了する。デフォルトはfalseである。
Stream **map**(convert(T event))	（Streamから継承） このストリームの各要素をconvert関数を使って新しい値に変換する新しいストリー

	ムを生成する。
Future **pipe**(StreamConsumer<dynamic, T> streamConsumer)	(Streamから継承) 指定された StreamConsumerの入力としてこのストリームをバインドする。
Future **reduce**(initialValue, combine(previous, T element))	(Streamから継承) 繰り返しcombineを適用することで値たちのシーケンスを減らす。
Future<T> **singleWhere**(bool test(T value))	(Streamから継承) testにマッチするこのストリーム中の単一の要素を探す。 このストリームのなかで一つ以上のマッチした要素があるときはエラーであることを除いてlastMatchと似ている。
Stream<T> **skip**(int count)	(Streamから継承) このストリームから最初のcount数のデータ・イベントをスキップする。
Stream<T> **skipWhile**(bool test(T value))	(Streamから継承) それらがtestにマッチしている間はこのストリームからのデータ・イベントをスキップする。 返されるストリームではエラーと完了のイベントに対しては何も加工されない。 testがそのイベント・データに対してtrueを返した最初のデータ・イベントから、返されるストリームはこのストリームと同じイベントたちを持つようになる。
Stream<T> **take**(int count)	(Streamから継承) このストリームの最大n個の最初の値たちを提供する。 このストリームの最初のn個のデータ・イベントと総てのエラー・イベントたちを返されるストリームに転送し、完了イベントで終了する。 もしこのストリームが終了前にcountの値よりも少ない数をつくるときは、返されるストリームもそうする。
Stream<T> **takeWhile**(bool test(T value))	(Streamから継承) testたちが成功している間はデータ・イベントたちを転送する。 返されるストリームはそのイベント・データに対してtestがtrueを返す限り同じデータ・イベントを提供する。このストリームはこのストリームが完了した、またはこのストリームがtestが受け付けない値を最初に提供したときに完了する。
Stream **timeout**(Duration timeLimit, {Function void onTimeout(EventSink sink)})	(Streamから継承) このストリームと同じイベントからなる新しいストリームを生成する。 このストリームからの2つのイベント間にtimeLimit以上が経過したときは、onTimeout関数が呼ばれる。 返されたストリームがリスンされるまではカウントダウンは開始しない。イベントがこのストリームから渡される度、あるいはこのストリームがポーズし再開される度にこのカウントダウンはリセットされる。 onTimeout関数が一つの引数(EventSink)で呼ばれる: EventSinkによりイベント

	たちを返されたイベントたちのなかに置くことができる。 onTimeoutが指定されていないときはタイムアウトは返されるストリームのエラー・チャンネルの中に TimeoutExceptionとして置かれる。 このストリームが放送ストリームの時は返されるストリームも放送ストリームとなる。ある放送ストリームが一回以上リスンされているときは、リスンごとにカウントを開始タイマを個々に持ち、加入たちのタイマーたちはここにポーズされ得る。
Future<List<T>> **toList**()	（Streamから継承） このストリームのデータをListに集める。
Future<Set<T>> **toSet**()	（Streamから継承） このストリームのデータをSetに集める。
Stream **transform**(StreamTransformer<T, dynamic> streamTransformer)	（Streamから継承） 指定された StreamTransformerの入力としてこのストリームを連結する。 streamTransformer.bind自身の結果を返す。
Stream<T> **where**(bool test(T event))	何らかのデータ・イベントを廃棄する新しいストリームをこのストリームから生成する。 新しいストリームはこのストリームと同じエラーと完了のイベントを送信するが、testを満たすデータ・イベントたちのみを送信する。

Dart:isolate.RawReceivePort

RawReceivePort abstract class	
コンストラクタ	
factory **RawReceivePort**([void handler(event)])	メッセージ受信のための長期に存続するポートを開設する。 RawReceivePortは低レベルのものでZoneと共には機能しない。これにはポーズをかけられない。このデータ・ハンドらは最初のイベントを受信するよりも前にセットされねばならない。
属性	
abstract void set **handler**(Function newHandler)	各到来メッセージの為のハンドらをセットする。 このハンドらはルート・ゾーン(Zone.ROOT)のなかで呼び出される。
メソッド	
abstract void **close**()	このポートを閉じる。 このメソッドの呼び出し後は、どの到来メッセージもそのまま廃棄される。

dart:isolate.SendPort

Interface **SendPort** SendPortたちはReceivePortたちから生成される。SendPortを通して送信されるメッセージは、それに対応するReceivePortに渡される。同じReceivePortあたり多くのSendPortを持つことがあり得る。 SendPortは他のアイソレートに送信され得る。	
実装	
Capability	
属性	
final int **hashCode**	この送信ポートの==演算子と一貫した不変ハッシュ・コード(immutable hash code)を返す。
演算子	
bool operator ==(other)	otherがこれと同じReceivePortに対応しているSendPortかどうかをテストする。
メソッド	
void **send**(message, ~~[SendPort replyTo]~~)	この送信ポートを介して非同期メッセージを対応するReceivePort に送信する。 メッセージの中身はプリミティブな値(null, num, bool, double, String)、SendPortのインスタンス、そしてその要素が上記のものであるlistとmapであり得る。ListとMapはサイクリックなものも許される。 2つのアイソレートたちが同じコードを共有しまた同じプロセス内で走っているような特別な状況（例えばspawnFunctionでアイソレートたちが生成されているとき）内では、オブジェクトのインスタンスを送信する（そのプロセス内でコピーされることになる）ことも可能である。現在これはdartVMのみが対応している。当面はfrogコンパイラのみが上記の制限されたメッセージに対応している。 送信は即座に起き、ブロックしない（非ブロッキング操作）。対応する受信ポートはそのアイソレートのイベント・ループがそれを処理できるようになれば送信アイソレートがしていることとは独立して即座にそのメッセージを受信できる。 ＊　2012年3月に変更、2013年10月にオプションのreplyToは廃止

dart:async.Timer

Interface **Timer** 2012年6月にdart:ioからこのdart:isolateに移された。更に2013年1月にdart:asyncに移された。	
コンストラクタ	
new **Timer(**Duration duration, void callback(Timer timer))	新規のタイマを生成。 callbackというコールバック関数は与えられたduration後に呼び出される。マイナスのdurationは0のdurationと同じように扱われる。 このdurationが静的に0だと判っているときはrunの使用を検討されたい。 しばしば、このdurationは定数または以下の例（Durationクラスの乗算演算子を活

	かしている）で示したように計算されるかのいずれかである： `const TIMEOUT = const Duration(seconds: 3);` `const ms = const Duration(milliseconds: 1);` `startTimeout([int milliseconds]) {` `  var duration = milliseconds == null ? TIMEOUT : ms *` `milliseconds;` `  return new Timer(duration, handleTimeout);` `}` 注意：もしTimerを使ったDartコードがJavaScriptにコンパイルされているときは、ブラウザで得られる最も細かい精度は4ミリ秒である。
new **Timer.periodic**(Duration duration, void callback(Timer timer))	新規の繰り返しタイマを生成する。callbackというコールバック関数はキャンセルされるまで各duration間隔毎に呼び出される。 マイナスのdurationは0のdurationと同じように扱われる。
staticメソッド	
void **run**(void callback())	指定されたcallbackを出来るだけ早く非同期で走らせる。 この関数はnew Timer(Duration.ZERO, callback)と等価である。
メソッド	
void **cancel**()	このタイマをキャンセルする。

第19章 HTTPサーバ (HttpServer)

サーバ・サイドでDartを使うには、dart:ioライブラリを使用する。特にHTTPサーバが良く使用されると思われるので、この章ではこのライブラリに含まれているHttpServerクラスについて手短に説明する。

HTTPサーバ開発にはHTTPプロトコルを理解しておく必要がある。筆者の改訂サーブレット・チュートリアルの第2章の一読をお勧めする。

DartのHttpServerクラスは、HTTPプロトコルによるサーバの為の基本的な手段を提供しているだけで、いろんなツールはいずれ誰かが用意してくれるという考え方であるので、Dartの基本思想に準じシンプルである。セッション管理は筆者の指摘で追加されたが、当面はユーザはその他の機能を自分で用意するかPubを探す必要がある。例えばPubには以下のものが登録されている:

- http_server ： HttpServerと組み合わせてより高度なツールを提供
- shelf ： ミドルウエアで、ウェブ・サーバの開発をより簡単化する

これらは「ファイル・アップロード」及び「ミドルウエア・フレームワーク」の章で解説する。

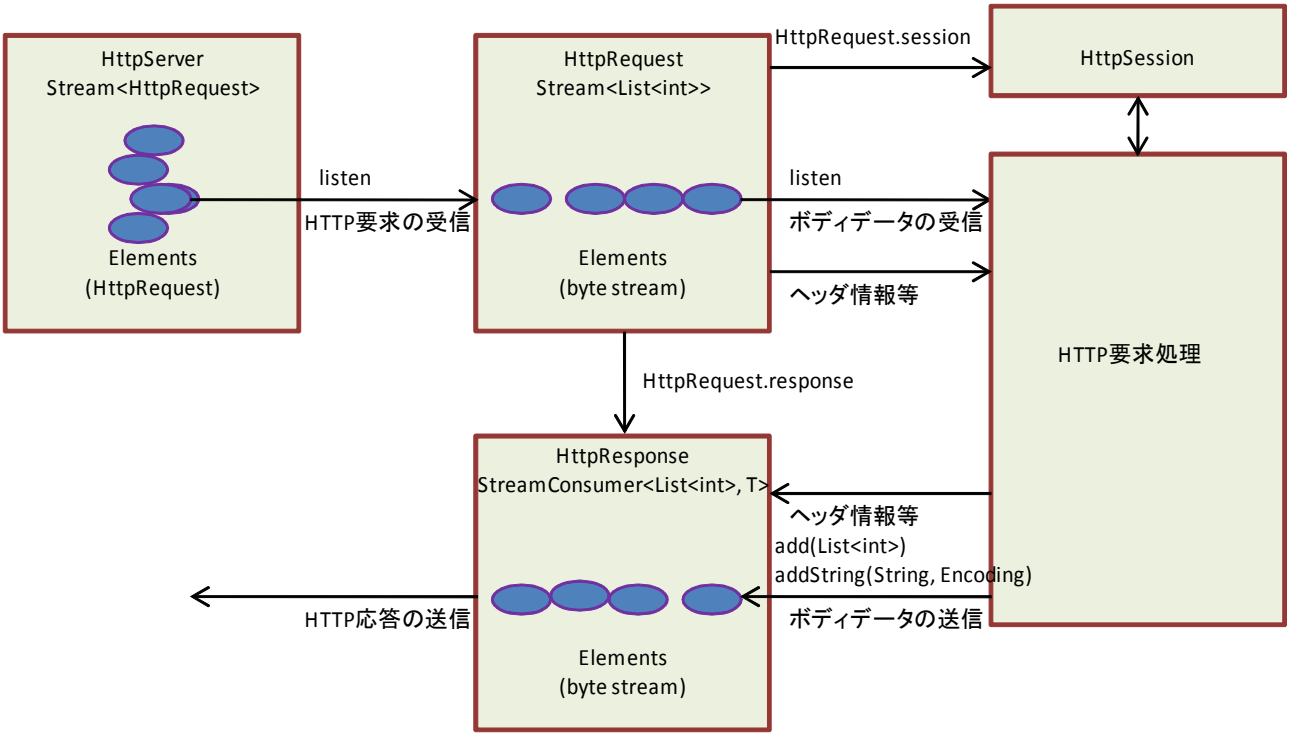
この章は2013年2月に大幅変更したAPIをベースにしている。それ以前には2012年4月18日に大幅な変更がされ、要求ハンドラ登録形式に切り替えられている。また筆者等の指摘によりバグ修正もなされている。2014年8月にはより安全化のためのデフォルト設定が追加されている。

19.1節　新しいAPIの概要

更に2013年2月のM3版でdart:ioに更なる大規模変更がなされた。これは非同期処理の為のFutureとStreamというDart固有の新しいコンセプトの積極的な導入である。

- これまでdart:io固有のインターフェイスだったInputStreamとOutputStreamが無くなり、IOSinkとStream<List<int>> を実装したクラスたちに置き換えられた。
- HttpRequestとHttpClientResponseのオブジェクトはStream<List<int>>を実装する。つまりバイト列のデータ・イベントとしてボディ・データを渡す。
- HttpClientRequestとHttpResponseオブジェクトはIOSinkを実装する。つまりボディ・データを受け取りネットワークに送信する。
- クライアントからのHTTP要求を受け付けるHttpServerはstaticなメソッドのbindで生成される。
- HttpServerはHttpRequestのストリームとして機能する。つまりHttpRequestのオブジェクトをデータ・イベント・ベースで渡す。

HTTPサーバの場合は次のようなイメージとなる:



HttpServerはその要素がHttpRequestであるストリームでもある。HttpServerのあるオブジェクトに対しlistenメソッドによりその要素（即ちHTTP要求）を受理する受信(StreamSubscription)を付加する。そうするとその受信に対し到来したHTTP要求に対応したオブジェクトがイベントとして渡される。

HttpRequestもまたその要素がHTTP要求のボディ部のバイト列であるストリームでもある。どうしてボディ・データがストリームとして供給されるのか不思議に思われるかも知れないが、長いボディ・データはチャンクとして送信されてくるからである。このストリームに対しlistenメソッドによりバイト列を取り込む。全部のデータが受理されたかどうかはlistenメソッドのonDoneで知ることができる。必要ならStringに変換するコンバータをtransformメソッドで付加することもできる。HTTP要求のヘッダ部やセッション、そしてHttpResponseなどはHttpRequestオブジェクトの属性として取得できる。

HTTP応答を返す為のHttpResponseはHttpRequestオブジェクトの属性として取得される。これは(request, response)のパラダイムのJava Servletに馴染んだ読者には新鮮に映るかもしれない。

HttpResponseはIOSinkを実装しているが、そのIOSinkはバイト列を取り込むStreamConsumer<List<int>, T>を実装している。従ってaddやaddStringメソッドによりバイト列としてHTTP応答のボディ部を受け付ける。ヘッダ部は別途HttpResponseオブジェクトの属性として設定できる。

HttpSessionもまたHttpRequestオブジェクトの属性として取得できる。HTTP要求処理はこのオブジェクトを使ってクライアント間のセッションを管理できる。

基本的なHttpServerの作り方は次のようである：

```
HttpServer.bind(“127.0.0.1”, 8080)
    .then((HttpServer server) {
      server.listen(
          (HttpRequest request) {
            // 到来要求処理
          });
```

```
}
```

bindというクラス・メソッドを使って指定したIPアドレスとポートを持ったHttpServerのオブジェクトを非同期で取得する。HttpServerのオブジェクトのlistenメソッドで非同期で到来要求を処理する。

POSTメソッドによるHTTP要求のボディ部は次のように取り込むことができる:

```
  HttpResponse response = request.response;
  String bodyString = "";      // request body byte data
  var completer = new Completer();
  if (request.method == "GET") { completer.complete("query string data
received");
  } else if (request.method == "POST") {
    request
      .transform(UTF8.decode)
      .listen(
          (String str){bodyString += str;},
          onDone: (){
            completer.complete("body data received");},
          onError: (e){
            print('exeption occured : ${e.toString()}');}
        );
  }
```

ここではtransformメソッドでStringにUTF-8デコードするコンバータを付加したストリームにし、これをlistenで受信している。これは完了（onDone）のイベントが到来するまで継続してチャンク形式のデータ受信に対応している。

一方HTTP応答をクライアントに返す為にはHttpResponseのオブジェクトに対し次のような操作を行えばよい:

```
List<int> htmlPageBytes = htmlPageString.charCodes;
    response
      ..headers.set('Content-Type', 'text/html; charset=UTF-8')
      ..contentLength = htmlPageBytes.length
      ..add(htmlPageBytes)
      ..close();
```

ここではStringであるクライアントに送信するHTMLテキストをString.charCodesでバイト列に変換し、その長さをcontentLengthでヘッダにセットし、ボディ部にはaddメソッドでこのバイト列を書き込んでいる。

2014年8月のDart v1.6 ではDartで開発されたHTTPサーバがよりセキュアのものとするためにヘッダおよびクッキー設定にデフォルトを用意し、ベスト・プラクティスに従うようにした。

- HttpServerに defaultResponseHeadersという新しいフィールドが追加された。これは推奨ヘッダたちを集めたものである。各要求に対応したHttpResponseはdefaultResponseHeadersからのヘッダが使われる。この値を変更したければ HttpResponse.headersでこれらの値を追加または変更する。

- またHttpHeadersに明確なメソッドたちが付加された。これにより defaultResponseHeadersのすべての値をクリアしたり、このヘッダのオブジェクトの他のインスタンスをクリアしたりできる。

- 極力デフォルト値を常に使い、必要に応じ特定の応答だけに個々のヘッダを変更・追加するようにすることが好ましい。
- HttpServerまたはHttpHeadersを実装したクラスをユーザが作成している場合は、これらの付加されたメンバたちも実装する必要がある。
- Cookiesのデフォルト値には‘httponly’がセットされるようにした（筆者からのコメント参照）。

19.2節　Java Servletとの比較

Java Servletはスレッド・ベースであるので、プログラマはスレッド安全性に対しかなり神経質にならねばならない。しかしながらDartでは非同期ベースであるので、記述が楽になる（ServletではServlet 3.0からは非同期処理が導入されているがかなり複雑である）。例えば次のコードはHTTP要求が来たらあるファイルの内容(data)をクライアントに返すというシナリオ（現実的ではないが）である：

```
import "dart:io";

main() {
  HttpServer.bind("127.0.0.1", 1337).then((server) {
    print("listening on ${server.port}");
    server.listen((request) {
      var response = request.response;
      new File("/path/to/some/huge.file")
          .openRead()
          .listen((data) { response.writeBytes(data); },
              onDone: () { response.close(); });
      response.done
          .then((_) { print("done"); })
          .catchError((error) { print(error); });
    });
  });
}
```

プログラムの詳細はこの章のなかで説明されているが、listenあるいはthenというメソッドは非同期処理のメソッドであり、クライアントからの要求を受け付けたメインの処理は極めて早く終了する。従って次のクライアント要求処理にすぐに取りかかれる。listen及びthenのコールバック関数たちは、イベントが発生した時点でVMのスケジューラにより呼び出され実行される。各コールバック関数のオブジェクトは各イベントのオブジェクトごとに用意されるので、**各コールバックが共通のオブジェクトを変更しない限り**、後述のように混乱することはない。

従って非同期ベースであるDartの場合はアイソレートを使わなくても大きなスループットが達成できる。更に必要ならIsolateのプールを使って要求処理を振り分けることも可能である。Isolateの場合はリソースがIsolate間で共有されず、Isolate間はメッセージ通信を介して行われるので、スレッドのようなリソース共有にかかわる競合問題は回避される。マルチアイソレート化に関しては、2014年に新たな実験的APIが用意されている。

但し、Java Servletの場合はserviceメソッド（あるいはそれを展開したdoGetなど）を終了したら明示的に書かなくて

もその応答バッファをクローズ・フラッシュさせる。しかしながらDartのHttpServerの場合は、所定のアドレスとポート番号への総ての到来HTTP要求に対し、プログラマがその応答の送信の面倒を見なければならない。

19.3節　　HttpServerの経過

当初のAPIでDart VM（サーバ・アプリケーションの実行）の節で説明したtime_serverが、2012年4月時点で実際にどのようなHTTP応答を返しているのだろうかをMINAプロキシを使って調べた。結果を見る限りこのサーバは未だ問題が多い状態だった：

1. 最初のGET要求と次のGET /favicon.ico要求に対し同じ応答を返しており、これは本来のGET /favicon.ico要求の目的とは違っている。多分このプログラムを作成した技術者はfabicon.ico要求が来ることを知らない。この要求に対してはアイコンまたは404応答を返さねばならない。
2. HTTP応答のヘッダ部分がばらばらのTCPパケットで送信されており、非常に見苦しい。
3. チャンクのパケットとTCPパケットが一致していない、即ちひとつのチャンクが複数のTCPパケットで送信されているので、これも見苦しい。

この件は転送効率と応答時間に影響を与えるので、Ugly Dart HTTP Server responseとしてDart issue2012年4月に指摘してある。これに対し担当者(Søren Gjesse)からは5月2日に”The current implementation of the Dart HTTP library has not yet been tuned for performance.”とのつれない回答が来ていた。

バグ・レポートから1年以上経過した2013年5月14日になってDartチームはこの件に関して作業を開始した。

HTTPに関してはヘッダ部はひとつのバッファとして送信し、ボディ部は小さなデータ用に16kのバッファを用意し蓄積するが、到来データが4k以上のときはチャンクとして送信するとのこと。またWebSocketはヘッダ部はひとつのチャンク、ペイロード部はもうひとつのチャンク即ち2つのチャンクで構成するとしている。

2013年6月26日にこのチームのAnders Johnsenは作業を完了させたと報告してきた。

HttpServerのAPIは未だ改善がなされよう：

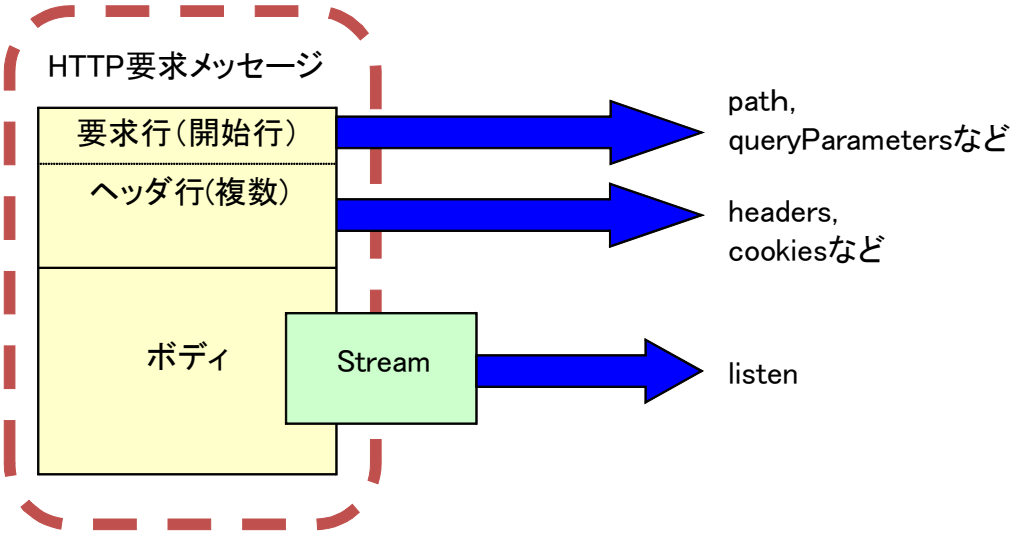
1. ヘッダやクエリ文字列処理用のURLエンコードとデコード、及びセッション管理の為のAPIが未だ公開されていなかった。（これはその後解決された）
2. 日本ではShift-JIS (Windows-31J)ほかの2バイト・コードも一般的に使用されており、その為のAPIが不可欠である。
3. StreamやIOSinkを実装したことでメソッドが多くなってしまい、初めてのユーザにはどれをどう使えば良いのか迷ってしまう。いずれフレームワークなどが開発され、ユーザがより簡素に開発できるようになろう。
4. Java Servletと違って、多数のクライアントからの要求処理の為にスレッドが使用できない。現在のAPIでどの程度のスループットが期待されるのか、あるいはアイソレートでこの問題が解決出来るのかが議論されている。
5. サーバとしてはクラッシュしないことが必須条件である。現在その為のエラー捕捉の強化が議論されている。（これもその後対策が取られてきている）

2014年4月14日にAnders JohnsenはDart 1.3を機にHTTPサーバの要求処理能力（スループット）を大幅に改善させたと発表した。その技術的詳細は彼のブログを読むとよい。細かなチューニングにより、例えばJSON処理を含む要求処理では毎秒約20,000要求とこれまでの約2倍の高速化が得られている。

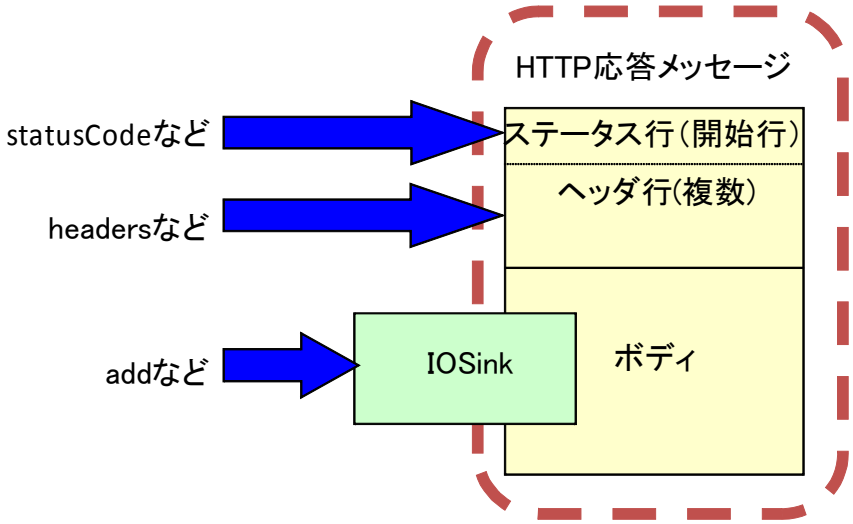
19.4節　HttpServerクラス

HttpServerクラスはサーブレット・コンテナのDart版ともいえる。従ってクライアントからの要求が到来するごとにHttpRequestとHttpResponse（HttpRequestの属性として）のオブジェクトが渡される。サーブレットと違う点はその要求に対する処理と応答がイベント・ベースになっていることである。

HttpRequestオブジェクトを基に、そのHTTP要求メッセージの情報を大まかには下図のように取り出すことが出来る。



反対に、HTTP応答メッセージはHttpResponseオブジェクトを基に、大まかには下図のように作成することができる。



サーブレットでもそうだが注意しなければならないのは、ボディ部にデータを書き込んだ時点でヘッダ部がネットワークに送信され、その後でヘッダ部分を変更しようとすると例外を発生することである。また総てのデータを書き込んだ後は、この出力ストリームをクローズしフラッシュさせねばならない。サーブレットではスレッドがサーブレットを抜けた時点で自動的にクローズされる。addで出力ストリームにデータを書き込むと、デフォルトではそれ

はUTF-8に変換されて送信される。それ以外に現時点で用意されているエンコーディングはISO_8859_1とASCIIである。我々としてはShift-JISも早く追加してもらいたいものである。

このインターフェイスのメソッドや属性たちは「関連APIの和訳」のところにあるので、見て頂きたい。

このインターフェイスの基本的な使い方は、次の節で示すDumpHttpRequest.dartや、Dartの実行の章で紹介したtime_server.dartなどを参考にすると良い。

実用に耐え得るHTTPサーバの標準的な記述は次のようになる:

標準的なHTTPサーバの記述

```
001 void listenHttpsRequest() {
002   HttpServer.bind(HOST_NAME, SERVER_PORT)
003   .then((HttpServer server) {
004     server.sessionTimeout = SESSION_MAX_INACTIVE_INTERVAL; // set session timeout
005     server.listen(
006       (HttpRequest req) {
007         req.response.done.then((d){
008           if (LOG_REQUESTS) log('sent response to the client for request : $
{req.uri}');
009         }).catchError((err) {
010           log('Error occured while sending response.. $err');
011         });
012         if (req.uri.path.contains(REQ_PATH)) processRequest(req);
013         else if (req.uri.toString().contains('favicon.ico'))
014           fhandler.doService(req, '../resources/favicon.ico');
015         else {
016           req.response.statusCode = HttpStatus.BAD_REQUEST;
017           req.response.close();
018         }
019       },
020       onError: (err) {
021         log('Listen request error.. $err');
022       },
023       onDone: () {
024         log('Done request listening');
025       },
026       cancelOnError: false
027       );
028     log('Server started. Serving $REQ_PATH on http://$HOST_NAME:${SERVER_PORT}$
{REQ_PATH}');
029   });
030 }
```

- 002行目： ホスト名がHOST_NAME、サーバのポートがSERVER_PORT（通常は本番用は80、実験用は8080）のHttpServerのFutureを取得する。
- 003行目： HttpServerが起動したら、以下のことを行う:
  - 004行目： サーバが持っているセッション管理のセッションごとの有効期限をここで設定する。
  - 005行目： ここでクライアントからのHTTP要求の到来を待つ。
    - 006行目： 要求が到来したら006-025行で指定したコールバック関数が実行される。
      - 007行目： 該要求に対する応答がクライアントへ返された時点（または途中でエラーが発生した時点）で実行するコールバック関数（007-011行）を設定する。
        - 008行目： 応答の送信が終了したときの処理。ここではログを記録する。
        - 009-011行目： 要求受付中にエラーが発生したときの処理。必要ならエラー・コード付きのエラー・ページをクライアントに送信する。
      - 012-015行目： 到来したHTTP要求のURIを調べ処理を振り分ける。即ち:
        - 012行目： 所定の要求パスのときはその要求は正規の要求なので、

processRequest(req)メソッドを呼んで該要求を処理する。

- 013-014行目： favicon.ico（ブラウザに表示するそのサーバの為のアイコン）要求のときは、必要なら自分のアイコンを返す。ここではfhandler.doService(req, '../resources/favicon.ico')メソッドが呼ばれている。favicon.icoを用意しないときは、req.response.close();としてボディ部が空の応答を返せばよい。
- 015-018行目： それ以外の要求URIは無視してBAD_REQUESTエラー・コードを付けたエラー・ページを返す。Java Servletの場合はserviceメソッド（あるいはそれを展開したdoGetなど）を終了したら明示的に書かなくてもその応答バッファをクローズ・フラッシュさせる。しかしながらDartのHttpServerの場合はプログラマがその面倒を見なければならない。

- 020-021行目： その要求処理中にエラーが発生したときは通常はログを取る。サーバの動作を止めてはいけない。
- 023-025行目： これはリスンが終了したときのコードで、通常サーバは終了を起こすことはないので、ログをとるだけである。必要ならここでサーバを再起動させる。

- 028行目： これでサーバが要求待機状態になったので、そのことのログを取る。

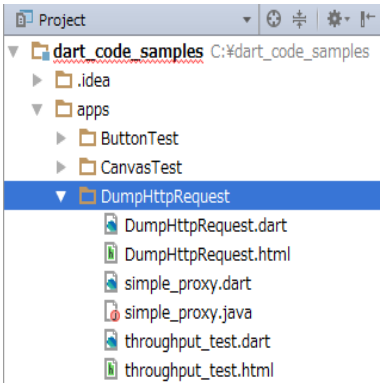
19.5節　　要求オブジェクト (DumpHttpRequest)

まずこのHttpServerがどのようにクライアントからの要求をユーザに渡しているのかを調べることとする。

サーブレットを使ったアプリケーション開発の場合と同様、クライアントからどのような要求が来ており、それをどのように取り出すかを調べる簡単なツールとしてのDumpHttpRequest.dartというサーバのコードを紹介する。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IDE上で\dart_code_samples-master\apps\DumpHttpRequestのフォルダを開くと良い。

このプログラムは次のように使用する：

1. 自分のIntelliJ IDEAからこのアプリケーションが入っているフォルダをひらく：



2. Run > Run 'DumpHttpRequest.dart'でサーバのコードを実行させる。サーバはServing the dump out on http://127.0.0.1:8080.とコンソールに表示する。

3. Chromeブラウザから直接DumpHttpRequest.htmlをアクセスする（IDE上でこのファイルを右クリックしてpathをコピーすると便利である。これをブラウザ上でたとえば

file:///C:\dart_code_samples\apps\DumpHttpRequest\DumpHttpRequest.htmlと指定する)と、次のような画面が表示される:

4. この画面のテキスト・エリアに適当な文字列を書き込み、2つのボタンのどれかをクリックする。Submit using POSTはPOST要求をし、Submit using GETはGET要求をする為のボタンである。
5. 例えばPOSTで送信すると、サーバは次のような応答を返す:

POST要求ではそのボディ部の内容をcontent-typeヘッダ行に従い:

- text/plainの場合はUTF-8でコードして表示
- application/x-www-form-urlencodedの場合はさらにURLでコードして表示
- それ以外の場合は単に8ビットASCIIの並びとして表示(表示できない文字は?で表示)している。

GET要求では、下図のように要求パラメタがクエリ文字列としてアドレス・バーに表示される:

6. このようにして、GETまたはPOST要求で送られたデータがサーバ側でどのように受け取られるのかを知ることができる。これらの内容は「改訂サーブレット・チュートリアル」の「第7章　要求オブジェクト」を参照されたい。

このプログラムは、読者がHTTPサーバを開発する際のテンプレートにもなる。読者はcreateHtmlResponse(HttpRequest request, String bodyString)という関数を自分のアプリケーションに置き換えるだけで良い。LOG_REQUESTSをtrueにしておけば、自分のアプリケーションに対しクライアントがどのような要求をしているのかをコンソールで知ることができ、デバッグには至極便利である。但しこの関数の中ではtryブロックを使って発生する例外とエラーを捕捉・処理し、サーバが停止することの無いようにしなければならない。またこのtryブロックの中にStreamやFutureを使ったコールバック関数があるときは、その関数の中でさらに別途例外とエラーの捕捉・処理が必要になる。

localhostに関する注意

localhostはいわゆる「ループバック・アドレス」であり、単一のコンピュータ上でクライアント(ブラウザ)とサーバを構成してテストするには便利なアドレスである。しかしながら'localhost'という名前はIPv4でもIPv6でも使われている。通常はOSに応じてどちらかが選択されるので、このことはあまり気にする必要はないが、間にプロキシをかませたり、IPv4またはIPv6で確実に動作させたいときには、注意が必要である。

明示的に確実にIPv4またはIPv6で動作させたいときは次のような記述を使う必要がある:

	IPv4	IPv6
サーバのアドレス	InternetAddress.LOOPBACK_IP_V4 または"127.0.0.1"	InternetAddress.LOOPBACK_IP_V6 または"[::1]"

ブラウザからのアクセス例	http://127.0.0.1:8080/....	http:[::1]:8080/....

プロキシを介在させてTCPレベルでの交信データを知る

プロキシ・サーバを介在させてHTTPの要求と応答の交信をTCPレベルで知りたいときは、「プロキシ・サーバに関して」の項の手順に従う。この際DumpHttpRequest.htmlは次のようにポート番号を8080から12345に変更する:

DumpHttpRequest.html

```
<!--
 Send HTTP test request to the DumpHttpRequest server
-->
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>

 <head>
  <title>DumpHttpRequest</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
 <!--
 replace upper line with :
 <meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
 and compare the result
 -->
 </head>

 <body>
  <H1>Send text area value to the server</H1>
  <form method="post" action="http://127.0.0.1:12345/DumpHttpRequest"
enctype="text/plain">
 <!--
 replace upper line with :
  <form method="post" action="http://localhost:12345/DumpHttpRequest">
 and compare the result (default enctype is "application/x-www-form-urlencoded")
 -->
   <textarea rows="5" cols="80" name="submitPost"></textarea><br>
   <input type="submit" value="Submit using POST">
  </form>
  <br>
  <form method="get" action="http://localhost:12345/DumpHttpRequest">
   <textarea rows="5" cols="80" name="submitGet"></textarea><br>
   <input type="submit" value="Submit using GET">
  </form>
 </body>

</html>
```

プロキシ・サーバに関して

本資料の添付であるdart_code_samplesのapps\DumpHttpRequestにsimple_proxy.java及びsimple_proxy.dartを追加したので、これらのサーバの使用をお勧めする。javaで書かれたsimple_proxy.javaのプログラムはA simple

proxy serverとして公開されているものを少し追加を加えたものである。Dartで書かれたsimple_proxy.dartのほうは、筆者が新たにSocketおよびServerSocketインターフェイスを使って作成したものである。

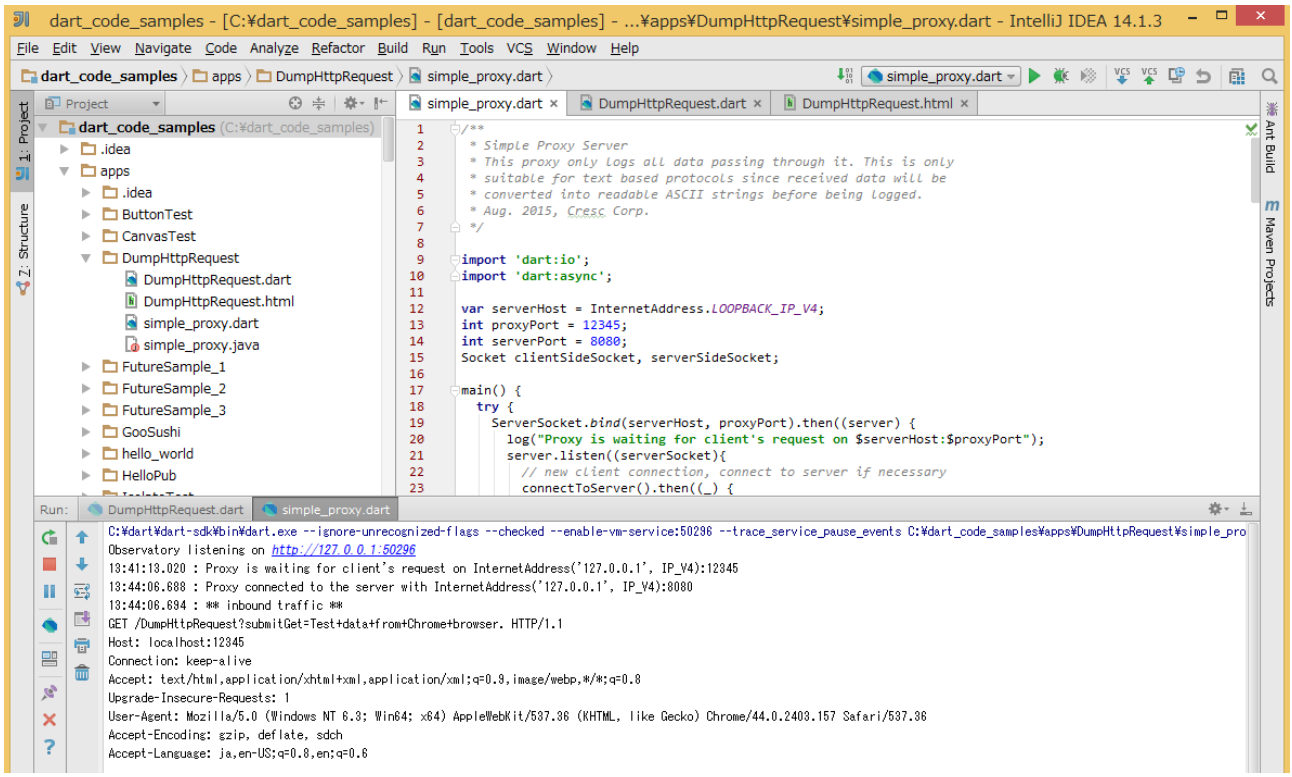
対象とするサーバのポート番号を12345と指定することでこのプロキシが介入し、そのソケット・レベルの交換メッセージをコンソールに表示する。例えばlocalhost:8080のサーバを起動させ、ブラウザからlocalhost:12345を開くとこのプロキシが介在し、交換される要求と応答のHTTPメッセージ(バイト列)をASCIIとして表示する。

Javaで書かれているので、多少経験ある読者は自分のIDE上でこのプロキシを稼働させることができよう。

手順は次のようになる:

1. サーバを起動させる
2. プロキシ・サーバを起動させる
3. ブラウザから"http://localhost:8080/サーバ・アプリへのパス"ではなくて"http://localhost:12345/サーバ・アプリへのパス"でこのサーバ・アプリをアクセスする
4. そうすればプロキシ・サーバのコンソールにはブラウザからの要求メッセージとサーバからの応答メッセージが8ビットASCIIとして表示される

以下はIntelliJ CE上で「要求オブジェクト」の節で紹介するDumpHttpRequest.dartをサーバとして、simple_proxy.dartをプロキシ・サーバとして、同時に動作させている例である:



プロキシを介した要求と応答の確認

以下は実際にブラウザからこのプロキシを介してDumpHttpRequestを呼び出したときのsimple_proxy.dartのコンソール出力である:

```
10:56:36.503 : ** connection (972487806, 633606771) inbound traffic **
POST /DumpHttpRequest HTTP/1.1
```

```
Accept: text/html, application/xhtml+xml, */*
Accept-Language: ja-JP
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: 127.0.0.1:12345
Content-Length: 231
DNT: 1
Connection: Keep-Alive
Cache-Control: no-cache

submitPost=IE%E3%81%8B%E3%82%89%E3%81%AE%E3%83%87%E3%83%95%E3%82%A9%E3%83%AB%E3%83%88%E3%81%AEURL
%E3%82%A8%E3%83%B3%E3%82%B3%E3%83%BC%E3%83%89%E3%81%AB%E3%82%88%E3%82%8BFORM%E3%83%87%E3%83%BC
%E3%82%BF%E3%81%AEPOST%E8%A6%81%E6%B1%82
10:56:36.712 : ** connection (972487806, 633606771) outbound traffic **
HTTP/1.1 200 OK
set-cookie: DARTSESSID=b473e389a4d1f68e340913cad311f112; Path=/; HttpOnly
transfer-encoding: chunked
content-encoding: gzip
x-frame-options: SAMEORIGIN
content-type: text/html; charset=UTF-8
x-xss-protection: 1; mode=block
x-content-type-options: nosniff

A
??????????
2EC
?T?N?A???)&??@v???m!!(???`?K3??vGfw??YZ?c{!?k_?hHL??$>?&????Nik???????????U?7??i?@=}??b????$?v????]??
j??l$?(J|d-d??????????^D?H\????`?
:F??>?1??D???b?????}*?????g?)E???2NO9????Q>???
j?8??1Fm??dl?"V?I?R?.??,??"!??\?NHL9?W1???=as?R?1?F?Q??NB ?lz??8F?.?.CPcd???a????b~W?n??qm??N????F?
>Qb?}??!?9|??????b????'?(??T#?????Z???Lo?7??a_????S.?????vk?U?j???!????F4w??s<??y???PI?pL?4g2?g?f????
a??fH?p>O??!I???}??U???qNM?U?H?NX??AL???f?[????Vi??d???!?h???????e???????????^?K??????4C?G?n??\'(??
B??.D:?=3&???UB??3?jw??j??Ps+?Y??[????+CGo???3e t???>2?K????z?*????????9???????j??????=
??`)??????,??]}?F????,???n??5"h?????]??e??l???,??????????>K??TO??OW????f!??F?w?Yi????????USi???
0
```

最初のブロックがブラウザからサーバに送信されたTCPレベルでのHTTP要求であり、次のブロックがサーバからのHTTP応答である。HTTP要求のメッセージでは、FORMテキスト「IEからのデフォルトのURLエンコードによるFORMデータのPOST要求」がsubmitPost=....としてボディ部にURLエンコードされて送信されている。コンテント長は231バイトである。HTTP応答のメッセージでは、HTMLテキストがGZIP圧縮されてチャンク形式で送信されている(ここでは3つのチャンク)。データの最後は長さ0のチャンクが送信されたことで知る。

これに対応したIE画面は次のようになっている:

```
Data available from the request
request.headers.host : 127.0.0.1
request.headers.port : 12345
request.connectionInfo.localPort : 8080
request.connectionInfo.remoteAddress : InternetAddress('127.0.0.1', IP_V6)
request.connectionInfo.remotePort : 50857
request.method : POST
request.persistentConnection : true
request.protocolVersion : 1.1
request.contentLength : 231
request.uri : /DumpHttpRequest
request.uri.scheme :
request.uri.path : /DumpHttpRequest
request.uri.query :
request.uri.queryParameters :
request.cookies :
request.headers.expires : null
request.headers :
 user-agent : Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
 connection : Keep-Alive
 cache-control : no-cache
 accept : text/html, application/xhtml+xml, */*
 accept-language : ja-JP
 accept-encoding : gzip, deflate
```

```
 dnt : 1
 content-length : 231
 host : 127.0.0.1:12345
 content-type : application/x-www-form-urlencoded
request.session.id : b473e389a4d1f68e340913cad311f112
requset.session.isNew : true
request body string (URL decoded): submitPost=IE からのデフォルトの URL エンコードによる FORM データの POST 要求
```

これらを詳しく調べれば、TCPレベルでクライアントから送信されてきたバイト列のHTTP要求メッセージをHttpServerがどのようにこれをHttpRequestのオブジェクトとしてアプリケーションに渡しているかが理解できよう。

simple_proxy.dartのプログラムはまた、ソケット・レベルでの通信のプログラム作成の良いサンプルにもなるので、興味のある読者はこのプログラムのウオークスルーをお勧めする。一般にブラウザは複数のTCP接続を使ってウェブ・サーバとやり取りしている。したがって接続ごとにデータ送受信のためのコールバック関数を用意してやらねばならない。また現在ハンドルしている接続たちを管理することが必須である。

要求オブジェクトから取得できるデータ

これらのツールを使うと、ブラウザからUTF-8で送信させた場合とShift-JISで送信させるとどうなるか、あるいは漢字のような多バイト文字を含んだデータを送信するとどうなるかなどの実験が可能となる。更にはこのツールのコードを見れば、どのようにしてクライアントからの要求から必要なデータを取得するかが理解されよう。

例えばPOST要求の場合は次のようになっている:

```
request.headers.host : 127.0.0.1
request.headers.port : 8080
request.connectionInfo.localPort : 8080
request.connectionInfo.remoteHost : 127.0.0.1
request.connectionInfo.remotePort : 50082
request.method : POST
request.persistentConnection : true
request.protocolVersion : 1.1
request.contentLength : 23
request.uri : /DumpHttpRequest
request.uri.path : /DumpHttpRequest
request.uri.query :
request.uri.queryParameters :
request.cookies :
  dartsessid=aae5f13dc65751cfa787abcb76fef2ba
request.headers.expires : null
request.headers :
  user-agent: Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.22 (KHTML, like Gecko) Chrome/25.0.1364.97
Safari/537.22
  connection: keep-alive
  cache-control: max-age=0
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  origin: null
  content-length: 23
  accept-language: ja,en-US;q=0.8,en;q=0.6
  host: 127.0.0.1:8080
  accept-charset: Shift_JIS,utf-8;q=0.7,*;q=0.3
  content-type: text/plain
  accept-encoding: gzip,deflate,sdch
  cookie: DARTSESSID=aae5f13dc65751cfa787abcb76fef2ba
request.session.id : c14dfb66743f5c9efdd9482609c5c919
requset.session.isNew : true
request body string : submitPost=Hello world!
```

要求データはボディ部にUTF-8の23バイトのデータとして送信されている。これはクライアントのHTMLコードに<form method="post" action="http://localhost:8080/DumpHttpRequest" enctype="text/plain">としてエンコードをテキストと指定しているからである。デフォルトはURLエンコードであるので正しく読めなくなるので注意しなければならない。このデータをStringDecoderは内部のUTF16ユニコードに正しく変換してくれている。

一方GET要求の場合は次のようになる:

```
request.headers.host : localhost
request.headers.port : 8080
request.connectionInfo.localPort : 8080
request.connectionInfo.remoteHost : 127.0.0.1
request.connectionInfo.remotePort : 50100
request.method : GET
request.persistentConnection : true
request.protocolVersion : 1.1
request.contentLength : -1
request.uri : /DumpHttpRequest?submitGet=Hello+world%21
request.uri.path : /DumpHttpRequest
request.uri.query : submitGet=Hello+world%21
request.uri.queryParameters :
  submitGet : Hello world!
request.cookies :
  dartsessid=cb3a89d9cf9243aa9f49435b1d1b4b02
request.headers.expires : null
request.headers :
  user-agent: Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.22 (KHTML, like Gecko) Chrome/25.0.1364.97
Safari/537.22
  connection: keep-alive
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  accept-language: ja,en-US;q=0.8,en;q=0.6
  accept-encoding: gzip,deflate,sdch
  cookie: DARTSESSID=cb3a89d9cf9243aa9f49435b1d1b4b02
  host: localhost:8080
  accept-charset: Shift_JIS,utf-8;q=0.7,*;q=0.3
request.session.id : cb3a89d9cf9243aa9f49435b1d1b4b02
requset.session.isNew : false
```

この場合request.queryParametersというメソッドは一応正しくデコードしている。また「野田 佳彦」と1バイトのスペース交じりの3バイト文字からなる文字列をいれてGET要求すると次のようになる:

```
request.headers.host : localhost
request.headers.port : 8080
request.connectionInfo.localPort : 8080
request.connectionInfo.remoteHost : 127.0.0.1
request.connectionInfo.remotePort : 50100
request.method : GET
request.persistentConnection : true
request.protocolVersion : 1.1
request.contentLength : -1
request.uri : /DumpHttpRequest?submitGet=%E9%87%8E%E7%94%B0+%E4%BD%B3%E5%BD%A6
request.uri.path : /DumpHttpRequest
request.uri.query : submitGet=%E9%87%8E%E7%94%B0+%E4%BD%B3%E5%BD%A6
request.uri.queryParameters :
  submitGet : 野田 佳彦
request.cookies :
  dartsessid=cb3a89d9cf9243aa9f49435b1d1b4b02
request.headers.expires : null
request.headers :
  user-agent: Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.22 (KHTML, like Gecko) Chrome/25.0.1364.97
Safari/537.22
  connection: keep-alive
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  accept-language: ja,en-US;q=0.8,en;q=0.6
  accept-encoding: gzip,deflate,sdch
  cookie: DARTSESSID=cb3a89d9cf9243aa9f49435b1d1b4b02
```

```
  host: localhost:8080
  accept-charset: Shift_JIS,utf-8;q=0.7,*;q=0.3
request.session.id : cb3a89d9cf9243aa9f49435b1d1b4b02
requset.session.isNew : false
```

すなわち1バイト文字のスペースは+に変換され、3バイト文字は16進のエスケープ変換されたまま(これはURL変換の対象にならない為)となってブラウザから送信されている。しかしながら6行目に見られるように、request.queryParametersはこれを正しくデコードしている。

?submitGet=%E9%87%8E%E7%94%B0+%E4%BD%B3%E5%BD%A6はutf-8の場合であるが、日本で良く使用されているShift-JIS(DumpHttpRequest.htmlの9行目をcharset=windows-31jとして試されたい)の場合は?submitGet=%96%EC%93c+%89%C0%95Fという形式で送信されてくる。これは漢字は2バイト文字だからである。

これに絡んだ日本語文字セット対応に関しては別のバグを開いてくれとのことだったので、それも見て頂きたい。読者の中でこれに関しご意見があればこのバグにコメントを入れて頂きたい。

URLデコーダ (URL Decoder)

2012年5月22日にDartチームはdart:uriというライブラリを導入している。同時に日本語を含む多バイト文字からなるGET要求データは正しくデコードされるよう修正がされている。

当初Dart:uriライブラリはJavaScriptのdecodeURIComponentとencodeURIComponentに対応したもので、これはHTML Formのデフォルト・エンコーディングのapplication/x-www-form-urlencodedに対応していなかった。しかしその後このクラスは強化されている。詳細は「関連APIの和訳」のdart.core.Uriを参照のこと。

2013年5月末にdart:uriはdart:coreに移された。これはこのライブラリが比較的小さなものなので、単独のライブラリにするまでもないと判断したためだという。

HTTPのボディとして受信されたURLエンコードされたバイト列(ここではbytesとしている)をStringに戻すには次のようにする:

```
Uri.decodeQueryComponent(UTF8.decode(bodyBytes))
```

日本ではUTF-8以外にShift-JISも広く使用されている。従って前述のようにコード変換機能を持っているJavaのURLEncoder / URLDecoderに相当するクラスが必要になる。

URLエンコードはまた、cookiesに何らかの文字列をセットする場合に必要になる。何故ならcookieのセットは応答ヘッダ行として行われ、ヘッダ部分はURLエンコードされていることが必須条件だからである。これにはdart:coreのUriクラスのencodeQueryComponentとdecodeQueryComponentが利用できる。

要求オブジェクトの中身のログを出力するメソッド(createLogMessage)

皆さんがあるウェブ・アプリケーションを開発する際は、クライアントからどのような要求が来ているかを確認する必要が出てくる。その為に、createLogMessageという関数を紹介する。この関数はDumpHttpRequest.dartに含まれているものを取り出したものである。引数はHttpRequestオブジェクトと、入力ストリームで取得したボディ部の文

字列である。GET要求のみの場合は、これを指定しなければよい。開発に際しては、要求処理の関数の最初の所でこの関数を呼ぶようにする。戻りはStringBufferなので、コンソール出力するときは print(createLogMessage(request, bodyString));のように記述する。また、コードの頭（即ちトップ・レベル）に次の行を追加する。開発中はクライアントからの要求が到来するごとに、その内容がコンソールに出力される。デバッグが終了したらLOG_REQUESTSの値をfalseにする。

```
final LOG_REQUESTS = true;
```

CreateLogMessage

```
// create log message
StringBuffer createLogMessage(HttpRequest request, [List<int> bodyBytes]) {
 var sb = new StringBuffer( '''request.headers.host : ${request.headers.host}
request.headers.port : ${request.headers.port}
request.connectionInfo.localPort : ${request.connectionInfo.localPort}
request.connectionInfo.remoteAddress : ${request.connectionInfo.remoteAddress}
request.connectionInfo.remotePort : ${request.connectionInfo.remotePort}
request.method : ${request.method}
request.persistentConnection : ${request.persistentConnection}
request.protocolVersion : ${request.protocolVersion}
request.contentLength : ${request.contentLength}
request.uri : ${request.uri}
request.uri.scheme : ${request.uri.scheme}
request.uri.path : ${request.uri.path}
request.uri.query : ${request.uri.query}
request.uri.queryParameters :
''');
 request.uri.queryParameters.forEach((key, value){
  sb.write("  ${key} : ${value}\n");
 });
 sb.write('''request.cookies :
''');
 request.cookies.forEach((value){
  sb.write("  ${value.toString()}\n");
 });
 sb.write('''request.headers.expires : ${request.headers.expires}
request.headers :''');
 request.headers.forEach((name,values){
  sb.write('\n  $name :');
  values.forEach((value) {
   sb.write(' $value');
  });
 });
 sb.write('''\n
request.session.id : ${request.session.id}
requset.session.isNew : ${request.session.isNew}
''');
 if (request.method == "POST") {
  var enctype = request.headers["content-type"][0];
  if (enctype.contains("text")) { // UTF8 encoded text/plain
   sb.write("request body string (text/plain) : ${UTF8.decode(bodyBytes)}");
  } else if (enctype.contains("urlencoded")) { // URL encoded
   sb.write("request body string (URL decoded): "
    + Uri.decodeQueryComponent(UTF8.decode(bodyBytes)));
  }
 }
 sb.write("\n");
 return sb;
}
```

このコードで注意して頂きたいのは、POSTでボディ部に置かれたデータの取り出しである。ボディ部に置かれたデータはcontent-typeヘッダでどのようなエンコーディングが使われているのかをブラウザが知らせている。エンコーディングは<form enctype="value">のようにHTMLで記述される。enctypeを指定しないとデフォルトの

application/x-www-form-urlencodedが適用される。エンコードしない場合はvalueとしてtext/plainを指定する。但しtext/plainの場合はスペースは"+"に変換されるので、これをbodyString.replaceAll('+', ' ')と変換しなければならないことに注意のこと。URLエンコーディングされている場合は、新しいUriクラスのdecodeUriComponentを使ってデコードする。

ボディ・データ入力の為のストリーム

入力ストリームはクライアントのテキスト・エリアからの比較的長いテキストやファイル・アップロードなどのデータを受理する為に使われる。転送形式はテキストであればURLエンコードされた各種エンコーディングの文字列データ、あるいは単純なUTF-8形式のバイト列であるが、画像などの場合はバイナリが使われる。

それらのデータはHTTPメッセージのボディ部に置かれ、その転送形式や長さはヘッダ行で知らされる。

HttpRequestはStream<List<int>>を継承している。つまりこのオブジェクトからボディ部のデータをバイト列として受理できるようになっている。このオブジェクトに対し:

1. Stream transform(StreamTransformer<T, dynamic> streamTransformer)でバイト列をユニコードのリストに変換する。StreamTransformerのサブクラスにはStringDecoderがあり、これのコンストラクタはデフォルトでUTF-8だとしてデコードしてくれる。UTF-8は最も一般的なエンコーディングであるが、それ以外にもASCII、ISO_8859_1（これはサーブレットのデフォルト）、及びSYSTEMが現在用意されている。しかしながら日本でよく使われるWindows-31J（Shift-JIS）は含まれていない。
2. このコンバータが付加されたStreamに対し、listenメソッドでsubscription（受信受信）が用意され、データが読みとられる。受信ハンドラは:

abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})

HttpServerはこの入力ストリームを介して次のようなイベントを渡す:

- 入力ストリームに受信デーが存在し、読み出し可能になった。(onData)
- エラーが発生した。(onError)
- データを総て受信し、データの総てが読みだされた。(onDone)

従ってこれらの3つのハンドラを使って正しくデータを読みだす必要がある。

なお、データが総て受信したことは、次の状態でこのサーバが知ることができる:

- 所定の長さのバイトを読みだした。
- チャンク形式の場合は長さ0のチャンクを受信した。
- TCP接続が切れた。

このインターフェイスの詳細は「関連APIの和訳」のところにあるので、見て頂きたい。

一般的な使い方はDumpHttpRequest.dartを見て頂ければ良い。

```
001 void requestReceivedHandler(HttpRequest request) {
002   HttpResponse response = request.response;
003   String bodyString = "";      // request body byte data
004   var completer = new Completer();
```

```
005   if (request.method == "GET") { completer.complete("query string data received");
006   } else if (request.method == "POST") {
007     request
008       .transform(new StringDecoder())
009       .listen(
010           (String str){bodyString = bodyString.concat(str);},
011           onDone: (){
012             completer.complete("body data received");},
013           onError: (e){
014             print('exeption occured : ${e.toString()}');}
015         );
016   }
017   else {
018     response.statusCode = HttpStatus.METHOD_NOT_ALLOWED;
019     response.close();
020     return;
021   }
```

1. 003行目： bodyStringは読みだした総ての文字列である。
2. 004行目： completerはボディ部からのデータ読み出しが完了したことを、次の処理のトリガとする為のCompleterオブジェクトである。
3. 005行目： 該要求がGETのときは入力ストリームは使わないので、直ちにcompleter.completeを呼ぶ。
4. 007-021行目： この部分がPOST要求でのボディ部読み出し部分である。
5. 008行目： transform(new StringDecoder())でストリームにコード・コンバータをバインドし、UTF-8エンコーディングで文字列を受け付けるStringの形式の入力ストリームを生成する。
6. 010行目： onDataのハンドラである。読みだしたデータ（この場合は文字列）をこれまでに読みだされたデータと連結する。
7. 011-012行目： onDoneのハンドラである。この場合は読み出しが完了したことになるので、completer.completeでこれで待機している処理を起動させる。
8. 013-014行目： onErrorのハンドラである。このときは例外を出力する。必要ならクライアントに500 (Internal Server Error)応答を返すようにもできよう。
9. 017-021行目： GETまたはPOST以外の要求が来たときは中身が空の応答を返しているが、これも何らかの応答（例えば405 Method Not Allowedというメッセージ）を返すようにも出来よう。

なお2013年夏からボディ部の処理が簡単になるhttp-serverというPubパッケージが使用可能となっている。POST要求でボディ部を使ったアプリケーションでは、そちらを使用することをお勧めする。このパッケージの詳細は「ファイル・アップロード」の章で示してある。

複数の要求を非同期で同時に処理する

簡単なサーバで複数の要求が非同期で同時処理されることを確認しよう。但し各々の要求が共通のリソースに変更を加えることが生じないことが条件である。要求間のリソース・アクセスの競合問題は、Javaのスレッド間の競合と似ている。その場合は排他メカニズムが必要になる。

同じDumpHttpRequestのフォルダの中にあるthroughput_test.dartはある要求が到来したら、指定された秒数だけスリープして、受け付けた時刻と応答を返した時刻をクライアントに返す。

throughput_test.dart

```
import "dart:io";
import "dart:async";

final HOST = "127.0.0.1";
final PORT = 8080;
final REQUEST_PATH = "/throughput";
```

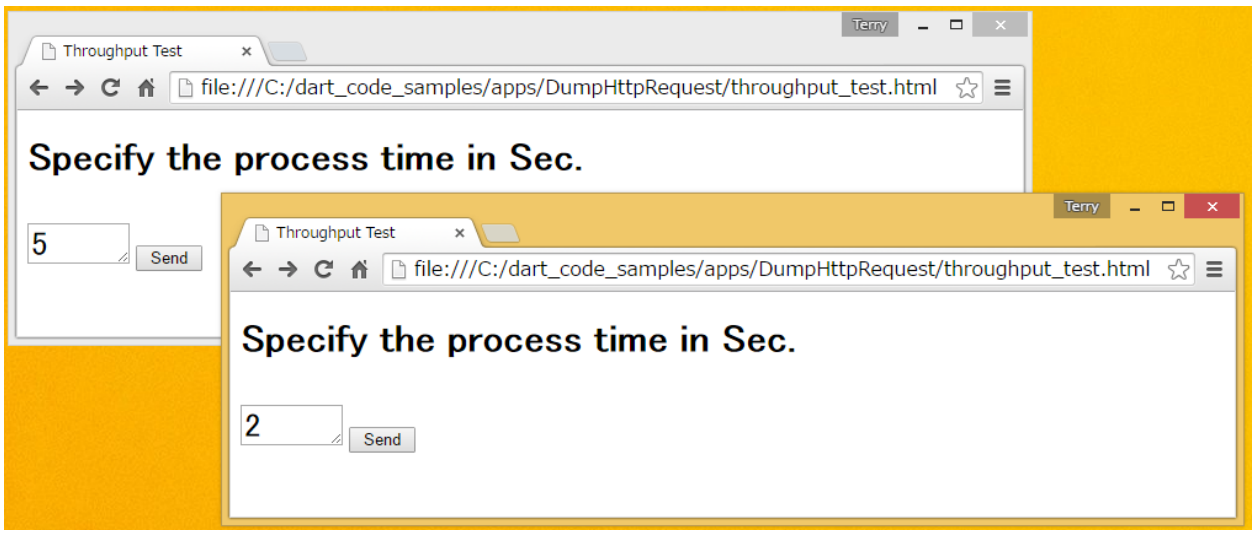
```
void main() {
  HttpServer.bind(HOST, PORT)
  .then((HttpServer server) {
    server.listen((HttpRequest request) async {
      var response = request.response;
      print("sent response to the client ${request.hashCode} for request : ${request.uri}");
      response.done.then((d){
      }).catchError((e) {
        print("Error occured while sending response: $e");
      });
      if (request.uri.path == REQUEST_PATH) {
        try {
          var pauseTime = num.parse(request.uri.queryParameters["pause"]);
          response.write("\n" + log("Request for $pauseTime sec. pause accepted"));
          await sleep(new Duration(seconds: pauseTime));
          response.write("\n" + log("Sent response for $pauseTime sec. pause time"));
        } catch(e,st){
          response.write("\n" +  log("Input error, try again \n$st"));
        }
        response.close();
      }
      else response.close();
    });
    print("${new DateTime.now()} : Serving $REQUEST_PATH on http://${HOST}:${PORT}.\n");
  });
}

// Sleeps during the specified duration
Future sleep(Duration duration) {
  return new Future.delayed(duration, () => "awaked");
}

// Create log message with timestamp
String log(String msg) {
  String timestamp = new DateTime.now().toString().substring(11);
  return '$timestamp : $msg';
}
```

このサーバは

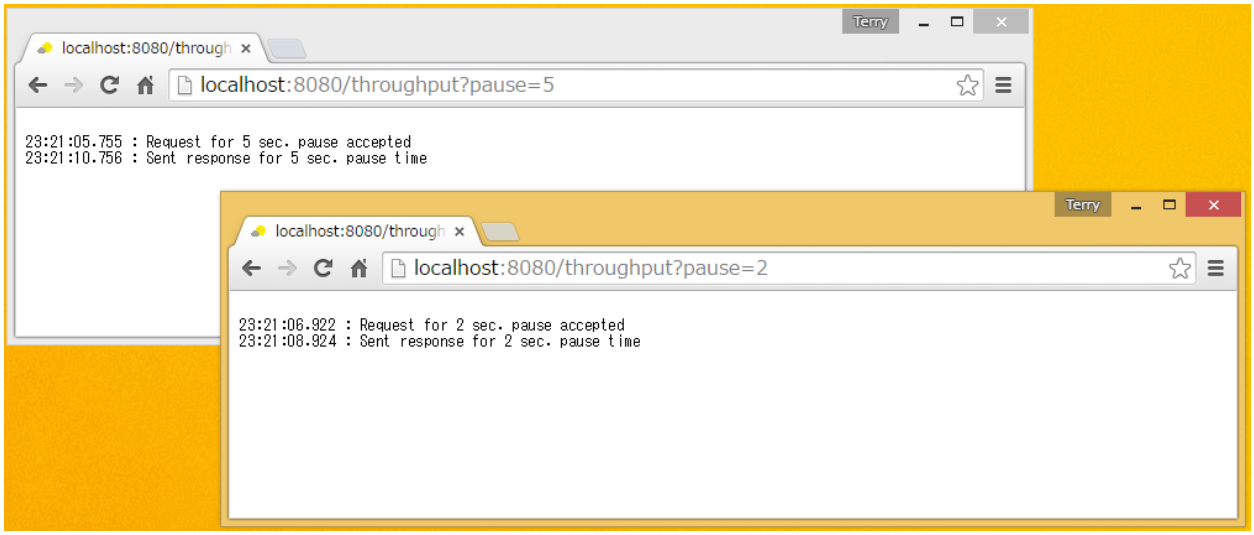
```
await sleep(new Duration(seconds: pauseTime));
```

の箇所で指定した秒だけスリープしており、この関数はFutureを返しているので、awaitでその終了を待っている。この行を含む関数ブロックはserver.listenのコールバック関数であり、各要求オブジェクト毎に用意される。従って複数の要求が到来しても、あたかも平行して処理されているかのように振る舞う。

このサーバをブラウザから2つのタブまたは2つのブラウザ画面からthroughput_test.htmlを介してアクセスすると、例えば次のような結果が得られる:



2つのブラウザ画面からの入力画面



2つのブラウザ・ウィンドウ上での応答例

ブラウザ1(5秒を指定):

```
23:21:05.755 : Request for 5 sec. pause accepted
23:21:10.756 : Sent response for 5 sec. pause time
```

ブラウザ2(2秒を指定):

```
23:21:06.922 : Request for 2 sec. pause accepted
23:21:08.924 : Sent response for 2 sec. pause time
```

つまりタブ1の要求処理が05.775秒目から10.756秒目の5.001秒間に行われている最中に、タブ2の要求処理が06.992秒目から08.924秒目の2.002秒間で行われており、2つのブラウザ間での干渉は起きていない。これは要求処理のコールバック関数のオブジェクトが各要求オブジェクト毎に用意されているからである。

前述のプロキシを介在させれば、より詳細にネットワーク上でその動作時刻を確認することができる。

19.6節　応答オブジェクト

HttpResponseインターフェイスはクライアントに返すHTTP応答を抽象化したものである。HTTP応答のヘッダ部はこのインターフェイスのフィールドたちを使って、オブジェクトを使ってヘッダ部とボディ部を書き込む。ボディ部はこのオブジェクトが出力ストリームそのものでもあるので便利になった。

このオブジェクトは該応答のHTTPヘッダ設定の為の一連の属性を持っている。該ヘッダが設定されたら、該HTTP応答の実際のボディ部に書き込むのにIOSinkからのメソッドたちが使えるようになる。このIOSinkのメソッドのどれかが初めて使われると、応答ヘッダ部はネットワークに送信される。それが送信された後でのヘッダ部を変更するメソッド呼び出しには例外がスローされる。

IOSinkを介して文字列データを書き込む際は、Java Servletと同じように、エンコーディングは"Content-Type"ヘッダの"charset"パラメタによって決まる。

またサーブレットと違ってHttpResponseオブジェクトはHttpRequestオブジェクトの属性として渡されるように2013年2月から変更された:

```
final HttpResponse response = request.response;
```

応答オブジェクトを使ってクライアントにHTTP応答を送信する基本的な使い方は、DumpHttpRequest.dartのコードを読めば理解できよう。

1. 必要に応じHttpResponse.headersを使ってHttpHeadersオブジェにクトに値をセットすることで、HTTP応答のヘッダ部をセットする。
2. HttpResponse.addStringボディ部にデータを書き込みネットワークに送出させる。この際ヘッダ部分が直ちにネットワーク側に送信され、その後のヘッダ部分の変更は出来なくなるので注意しなけらばならない。
3. 一般的にはHttpResponse.writeを使って文字列を所定の文字セットでデータを書き込む。writeのデフォルトのエンコーディングはJava Servletと同じくISO-8859-1 (Latin 1) である。必要なら"Content-Typeヘッダでこれを変更して別のエンコーディングを適用することもできる。
4. 総てのデータの書き込みが終了したら、その出力ストリームを閉じることで、HTTP応答の送信処理が終了する。
5. ネットワークへのヘッダ部とボディ部の送信が終了したことは、OutputStream.doneで知ることができる。

ここではやや詳しくその仕組みを説明する。

応答ヘッダ部への値のセット

2014年8月のDart v1.6 ではDartで開発されたHTTPサーバがよりセキュアのものとするためにヘッダおよびクッキー設定にデフォルトを用意し、ベスト・プラクティスに従うようにした。

- HttpServerにdefaultResponseHeadersという新しいフィールドが追加された。これは推奨ヘッダたちを集めたものである。各要求に対応したHttpResponseはdefaultResponseHeadersからのヘッダが使われる。この値を変更したければHttpResponse.headersでこれらの値を追加または変更する。

- またHttpHeadersに明確なメソッドたちが付加された。これによりdefaultResponseHeadersのすべての値をクリアしたり、このヘッダのオブジェクトの他のインスタンスをクリアしたりできる。

- 極力デフォルト値を常に使い、必要に応じ特定の応答だけに個々のヘッダを変更・追加するようにすることが好ましい。

応答ヘッダのステータス行は、HttpHeadresで特に指定しない限り"200 OK”がセットされる。それではそれ以外のステータスを設定したら、HttpServerインターフェイスはどのような応答をするのだろうか？

次のようなサーバ（StatusLineTest.dart）を実行させてみよう。このサーブレットはいろんなステータスをセットして応答を返すと、どのような応答が返されるかを調べるためのものである。すなわちブラウザ側のHTTPステータス選択ラジオボタンのひとつを選択してサブミット・ボタンをクリックすることで、いろんなステータス行のHTTP応答を返させることができる。

このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、IDEから例えば\dart_code_samples\apps\StatusLineTestのフォルダを開くと良い。

使い方は前節とおなじで、:

1. StatusLineTest.dartをサーバとして実行させる。
2. 自分のブラウザからおなじホルダにあるStatusLineTest.htmlファイルを例えば次のように選択する:
   C:\.....\StatusLineTest\StatusLineTest.html
   このアドレス（ファイル・パス）は、IDE上でこのHTMLファイルを選択し、右クリックでCopy File Pathを選択し、これをブラウザのアドレス・バーに貼り付ける。例えば:
   file:///C:\.....\StatusLineTest\StatusLineTest.html
3. ブラウザにはステータス・コードの総てが選択できる画面が表示されるので、それからひとつ、例えば401を選択し、サブミット・ボタンをクリックする。
4. そうするとサーバは次のような画面を返す:
   401 Unauthorized
   The request requires user authentication.

このとき実際にネットワークを通過するHTTPのヘッダ部はプロキシで調べると次のようになっている:

```
HTTP/1.1 401 Unauthorized
set-cookie: DARTSESSID=554da0cb65cd7797e801a3dabf6816f7; Path=/; HttpOnly
transfer-encoding: chunked
content-encoding: gzip
x-frame-options: SAMEORIGIN
content-type: text/html; charset=UTF-8
x-xss-protection: 1; mode=block
x-content-type-options: nosniff
```

すなわち仕様（RFC)では最初のステータス行は次のようになっているが:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

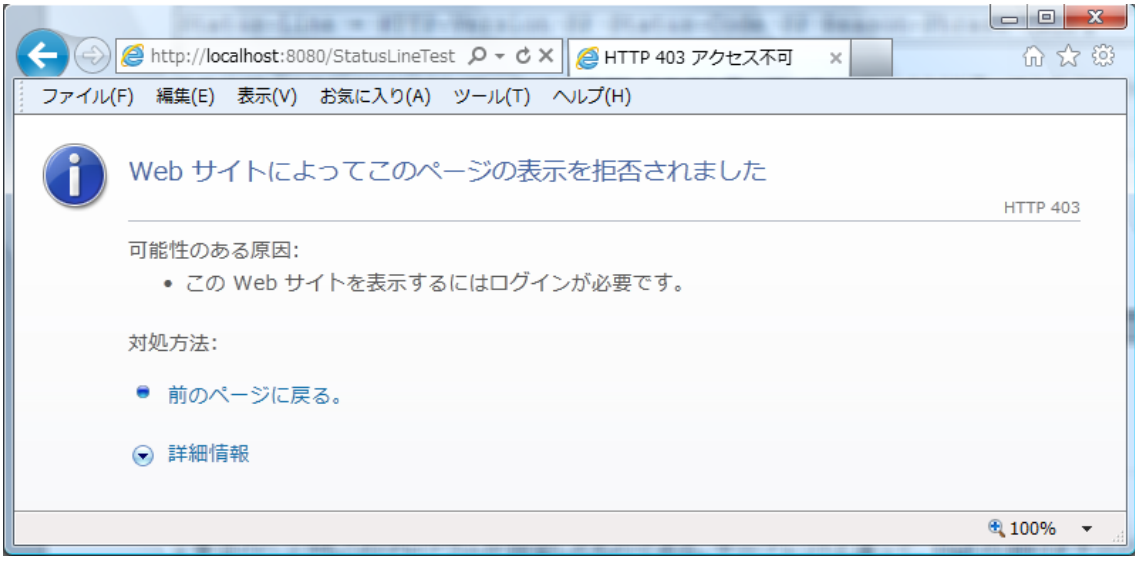
その仕様通りの行が送信されている。こう設定したからと言って、サーブレットとは違ってクライアント認証を開始することは無い。それはプログラマの仕事になる。

6行目のヘッダもこのプログラムが設定したものである。サーブレットと違って、DartのHTTPサーバはこの設定によって文字セットを選択したりすることは無い。これもプログラマの責任である。

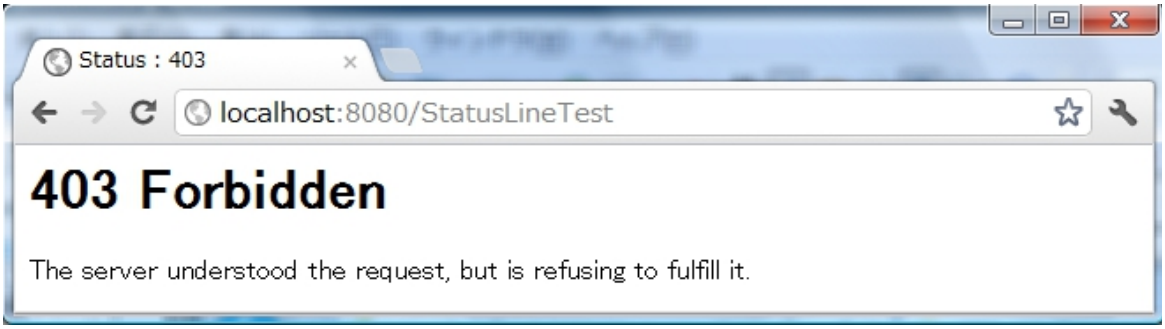
- 2行目はクライアントにセッション・クッキーをセットする
- 3行目はボディ部をチャンク送信を行うことを示す

- 4行目はボディ部はGZIP圧縮されていることを示す（Dartでは総てGZIP圧縮される）
- 5、7、8行目はセキュリティ強化のためdefaultResponseHeadersとして設定されているヘッダ行である

ブラウザがこの応答に対してどのような画面を出すかは、ブラウザによって異なる。例えばInternet Explorerでは403応答に対しては次のような画面を表示する：



一方Chromeでは次のように単純な画面となる：



逆に406応答に対してはChromeは次のような画面を表示する：

以下はサーバStatusLineTest.dartのコードの一部である:

StatusLineTest.dart

```
001 // Dart code sample : Status line setting and HTTP response
002 // Returns a response with required status line
003 // 1. Create a folder named StatusLineTest
004 // 2. Put StatusLineTest.dart and StatusLineTest.html into the folder
005 // 3. From Dart editor, File -> Open Folder, and select the StatusLineTest folder
006 // 4. Run StatusLineTest.dart as server
007 // 5. Access file:///c:/..../StatusLineTest/StatusLineTest.html from your browser
008 // 6. Select a status code and test response of available browsers
009 // Ref: www.cresc.co.jp/tech/java/Google_Dart/DartLanguageGuide.pdf (in Japanese)
010 // May 2012, by Cresc Corp.
011 // September 2012, incorporated API change (dart:math)
012 // October 2012, incorporated M1 changes
013 // February 2013, incorporated M3 changes
014
015 import "dart:io";
016 import 'dart:math' as Math;
017
018 final HOST = "127.0.0.1";
019 final PORT = 8080;
020 final REQUEST_PATH = "/StatusLineTest";
021 final LOG_REQUESTS = true;
022
023 void main() {
024   HttpServer.bind(HOST, PORT)
025   .then((HttpServer server) {
026     server.listen(
027         (HttpRequest request) {
028           if (request.uri.path == REQUEST_PATH) {
029             requestReceivedHandler(request);
030           }
031         });
032     print("Serving $REQUEST_PATH on http://${HOST}:${PORT}.");
033   });
034 }
035
036 void requestReceivedHandler(HttpRequest request) {
037   HttpResponse response = request.response;
038   logRequest(request);
039   int status = int.parse(request.queryParameters['raioButton']); // get status code from the query
040
041   List statusCode;
042   if (status == 100) { statusCode = [HttpStatus.CONTINUE, '100 Continue', 'The client SHOULD
continue with its request.'];
043   } else if (status == 101) { statusCode = [HttpStatus.SWITCHING_PROTOCOLS, '101 Switching
Protocols', 'The server understands and is willing to comply with the client\'s request, via the
Upgrade message header field (section 14.42), for a change in the application protocol being used on
this connection.'];
```

```
044   } else if (status == 200) { statusCode = [HttpStatus.OK, '200 OK', 'The request has
succeeded.'];
途中省略
058   } else if (status == 400) { statusCode = [HttpStatus.BAD_REQUEST, '400 Bad Request', 'The
request could not be understood by the server due to malformed syntax.'];
059   } else if (status == 401) { statusCode = [HttpStatus.UNAUTHORIZED,'401 Unauthorized', 'The
request requires user authentication.'];
060   } else if (status == 403) { statusCode = [HttpStatus.FORBIDDEN,'403 Forbidden', 'The server
understood the request, but is refusing to fulfill it.'];
061   } else if (status == 404) { statusCode = [HttpStatus.NOT_FOUND, '404 Not Found', 'The server has
not found anything matching the Request-URI.'];
062   } else if (status == 405) { statusCode = [HttpStatus.METHOD_NOT_ALLOWED, '405 Method Not
Allowed', 'The method specified in the Request-Line is not allowed for the resource identified by the
Request-URI.'];
063   } else if (status == 406) { statusCode = [HttpStatus.NOT_ACCEPTABLE, '406 Not Acceptable', 'The
resource identified by the request is only capable of generating response entities which have content
characteristics not acceptable according to the accept headers sent in the request.'];
途中省略
075   } else if (status == 500) { statusCode = [HttpStatus.INTERNAL_SERVER_ERROR, '500 Internal Server
Error', 'The server encountered an unexpected condition which prevented it from fulfilling the
request.'];
076   } else if (status == 501) { statusCode = [HttpStatus.NOT_IMPLEMENTED, '501 Not Implemented',
'The server does not support the functionality required to fulfill the request.'];
077   } else if (status == 502) { statusCode = [HttpStatus.BAD_GATEWAY, '502 Bad Gateway', 'The
server, while acting as a gateway or proxy, received an invalid response from the upstream server it
accessed in attempting to fulfill the request.'];
078   } else if (status == 503) { statusCode = [HttpStatus.SERVICE_UNAVAILABLE, '503 Service
Unavailable', 'The server is currently unable to handle the request due to a temporary overloading or
maintenance of the server.'];
079   } else if (status == 504) { statusCode = [HttpStatus.GATEWAY_TIMEOUT, '504 Gateway Timeout',
'The server, while acting as a gateway or proxy, did not receive a timely response from the upstream
server specified by the URI.'];
080   } else if (status == 505) statusCode = [HttpStatus.HTTP_VERSION_NOT_SUPPORTED, '505 HTTP Version
Not Supported', 'The server does not support, or refuses to support, the HTTP protocol version that
was used in the request message.'];
081
082   String statusPageHtml = """
083 <html><head>
084 <title>Status : ${statusCode[0]}</title>
085 </head><body>
086 <h1>${statusCode[1]}</h1>
087 <p>${statusCode[2]}</p>
088 </body></html>""";
089
090   response.statusCode = statusCode[0];
091   response.headers.add("Content-Type", "text/html; charset=UTF-8");
092   response.addString(statusPageHtml);
093   response.close();
094 }
095
096 // log out contents of the request
097 void logRequest(HttpRequest request, [String bodyString = '']) {
098   print(createLogMessage(request).toString());
099 }
100
101 // create log message
102 StringBuffer createLogMessage(HttpRequest request, [String bodyString]) {
中身省略
147 }
148
```

応答送信の為の出力ストリーム

HTTP応答メッセージのボディ部へのデータの書き込みの為にHttpResponseはIOSink<HttpResponse>を実装しており、そのIOSinkはStreamConsumer<List<int>, T>を実装している。即ちバイト列としてボディ部をネットワークに送信できる。このAPIは「関連APIの和訳」を見て頂きたい。

なお2013年3月にIOSinkインターフェイスがStringSinkを実装するよう変更された。従ってwrite、writeln、writeAll及びwriteCharCodeメソッドを使って文字列をIOSinkに書き込むにはencoding属性を使ってエンコーディングを指定する。HttpResponse及びHttpClientRequestのIOSinkでは、Java Servletと同じようにContent-Typeヘッダのcharsetパラメタをもとにエンコーディングが選択される。デフォルトのエンコーディングはこれもJava Servletと同じで ISO_8859_1となった。

送信するデータには画像のようなバイナリ・ファイルとHTMLテキストのような文字列の形式がある。文字列データの送信にはHttpResponseのwriteを、バイト・データの送信には通常pipeメソッドを使用する。どちらも送信形式はバイト数指定、即ちcontent-length:ヘッダで長さを相手に知らせる方式と、HTTP/1.1で導入されたtransfer-encoding : chunkedヘッダでそれを相手に知らせるチャンク形式が利用できる。

テキストの送信は、読者は既にこれまでのサンプルを見て理解されている筈である。従ってここではバイナリ・データの送信についてその例(ファイル・サーバ)を次節で示すことにする。

圧縮伝送

HTTP応答にはデフォルトの応答ヘッダが通常セットされる。それ以外の場合は必要に応じて応答ヘッダを変更する。

よく使われるのが応答ボディ部をGZIP圧縮してクライアントに渡す手段である。2014年8月のDart v1.6からはHttpServer.autoCompressをtrueにすると、該応答のボディ部はGZIP圧縮され、それに応じて応答ヘッダ部のcontentLengthヘッダもセットされるようになった(それ以前はすべての応答がGZIP圧縮されていた)。但し条件としては:

- クライアントがGZIP圧縮を受けるという要求ヘッダで要求してきた(通常殆どのブラウザがそうなっている)
- クライアントがHTTP/1.1で要求してきた(これも通常殆どのブラウザがそうなっている)
- HttpHeaders.chunkedTransferEncodingがtrueにセットされていること(クライアントがHTTP/1.1で要求してきたときはそうなっている)

例えば以下のサーバを実行させてみよう:

code_19.6a.dart

```
// Most simple GZIP echo server
import "dart:io";

void main() {
 HttpServer
 .bind(InternetAddress.LOOPBACK_IP_V4, 8080)
 .then((server) {
  server.autoCompress = true;
  server.listen((HttpRequest request) {
   request.response..write('Hello, world!')
    ..close();
  });
 });
}
```

このサーバをブラウザからプロキシ経由でhttp://localhost:12345/でアクセスするとその応答は次のようになっている:

```
HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block
transfer-encoding: chunked
x-content-type-options: nosniff
content-encoding: gzip

A
#
....
0
```

即ちcontent-encoding: gzipのヘッダが追加され、そのボディ部はGZIP圧縮され10バイトのチャンクになっている。

一方server.autoCompress = false;にセットすると、以下のように平文で送信されチャンクのサイズは13バイトである。

```
HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block
transfer-encoding: chunked
x-content-type-options: nosniff

D
Hello, world!
0
```

ブラウザにはともにHello, world!と表示される。

ブラウザやプロキシ・サーバを使わないで簡単に圧縮伝送のテストをしたいときには、次のようなコードを使えばよい:

code_19.6b.dart

```
/*
 GZIP compression test
 */
import 'dart:io';
import 'dart:convert';

void testServer() {
 HttpServer
 .bind(InternetAddress.LOOPBACK_IP_V4, 8080)
 .then((server) {
  server.autoCompress = true; // change here to disable compression
  print('Server started with: '
    '${server.address}:${server.port}');
  server.listen((HttpRequest request) {
   print('Server received a request with: ${request.uri.path}');
   request.response..write('Hello, world!')
            ..close();
  });
 });
}
```

```
void main() {
 var httpRequest = 'GET / HTTP/1.1\nAccept-Encoding: GZIP\n\n';
 // start the test server
 testServer();
 // connect, send a request, and receive its response
 Socket.connect(InternetAddress.LOOPBACK_IP_V4, 8080).then((socket) {
  print('Connected to: '
    '${socket.remoteAddress.address}:${socket.remotePort}');
  socket.listen((data) {
   print('Response from the server:\n'
    + new String.fromCharCodes(data).trim());
   });
  socket.add(UTF8.encode(httpRequest));
  print('Request to the server:\n$httpRequest');
 });
}
```

このプログラムはcode_19.6a.dartと同様なサーバ（testServer）とSocketを使ったブラウザを擬したsocketというクライアントからなる。クライアントからはHTTP要求メッセージ(httpRequest)をサーバに送信し、その応答メッセージをASCIIテキストとして表示する。要求メッセージにはAccept-Encoding: GZIPというヘッダが存在しているので、サーバはHello, world!というテキストを圧縮してクライアントに返す。

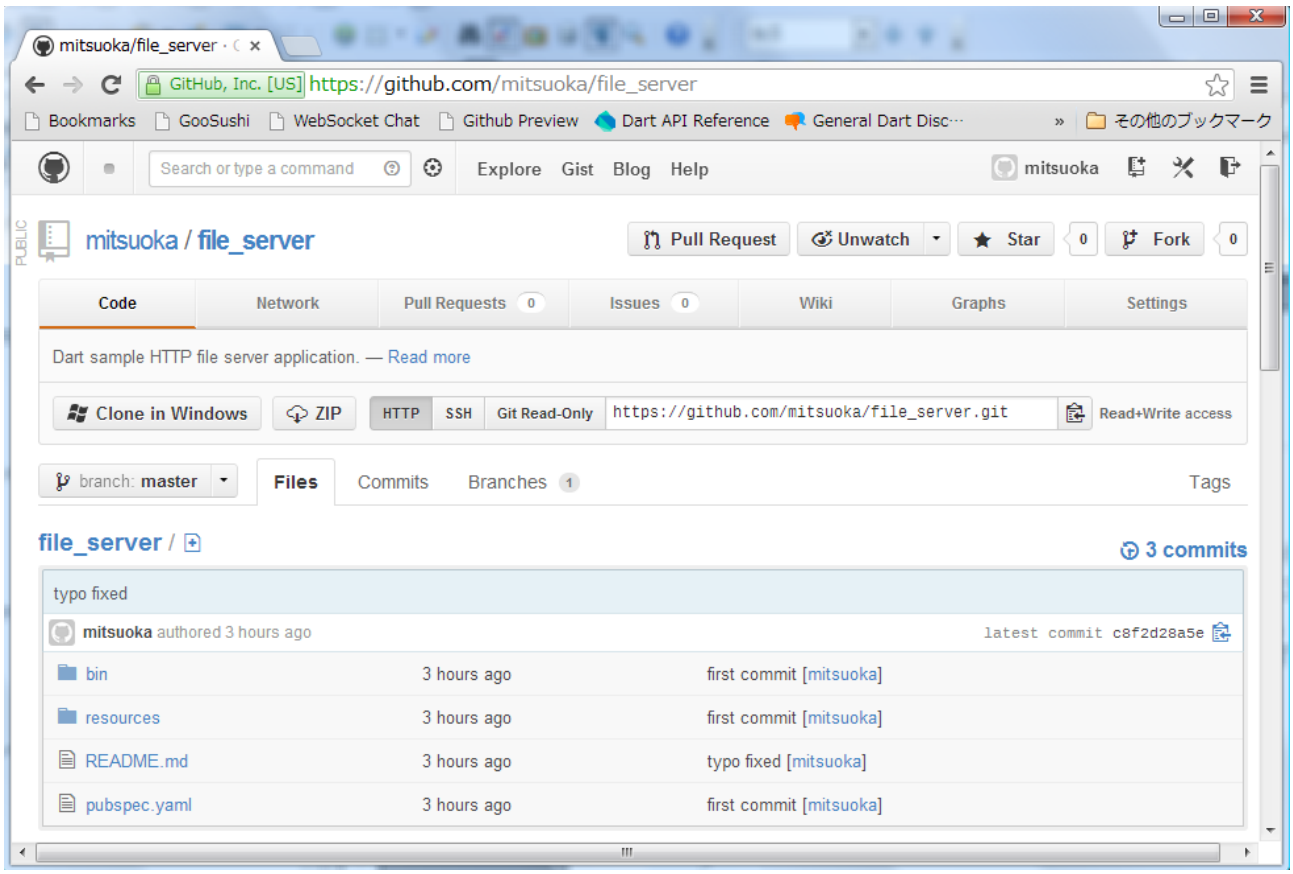
19.7節　　ファイル・サーバ

FileServer.dartは簡単なファイル・サーバである。このコードはGithubのリポジトリからダウンロードできる。このコードは基本的にGistで公開されているサンプルをもとにしている。このサーバはあくまでもDartコードのサンプルであり、実際のアプリケーションに使ってはいけない。何故なら、このコードはセキュリティに配慮されていないからである。一般にはこの種のアプリケーションではHTTPSなどのセキュアな接続が必要である。

このアプリケーションのダウンロードと配備

「本資料に含まれているプログラムのダウンロード」の章を参考にするとよい。

1. Githubのfile_serverのリポジトリをブラウザで開くと次のような画面が表示される：



2. この画面の左中央にあるZIPと表示した個所をクリックしてfile_server-master.zipという圧縮ファイルをダウンロードする。masterというのはブランチの識別のために付されている。

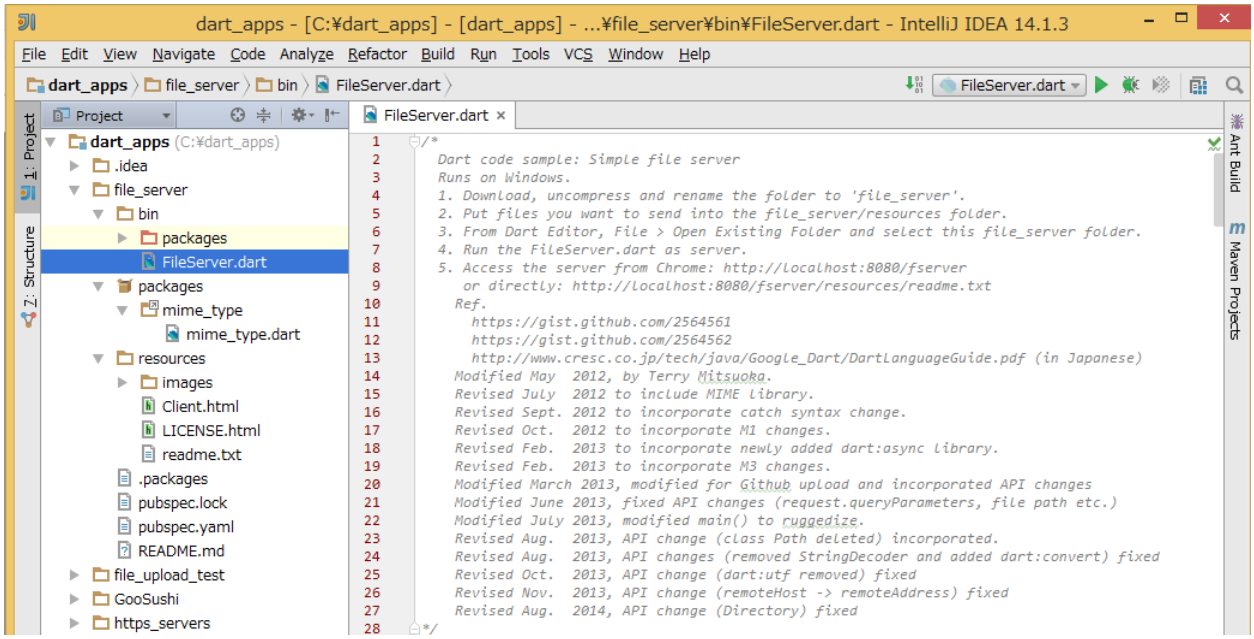
3. このファイルを適当な解凍ツールを使って解凍するとfile_server-masterというフォルダが作成される。このフォルダの名前をfile_serverと変更する。

4. IDE上でプロジェクトを設定する。IntelliJのユーザの場合は、例えばC:\にdart_appsというプロジェクトを作る。あるいはこのフォルダをすでにプロジェクトとして用意してあるdart_code_samples / appsのディレクトリにコピーするのが最も簡便な方法である。

5. 自分のIDE上でFile > Open ..を選択し、Browseボタンを押してこのファイルを選択する。

6. このアプリケーションのpubspec.yamlを開く、あるいは右クリックして、Pub : Get Dependenciesを実行して、必要なライブラリを取り込む。

7. しばらくすると自分のIDE上のファイル・ビューには次のようなファイル構成が表示される:



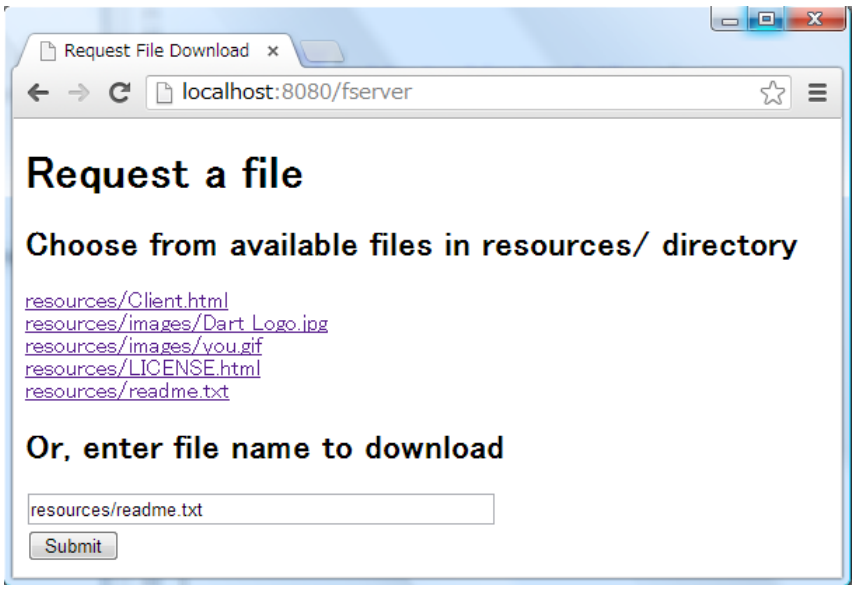
file_serverの展開

- packages ： これはPubがpubspec.yamlを見て追加したフォルダ
- mime_type ： これはPubがpubspec.yamlを見てダウンロードしたpubライブラリのフォルダ
- pubspec.lock ： これもPubがpubspec.yamlを見て追加した管理用のファイル
- pubspec.yaml ： このアプリケーションの仕様
- bin ： サーバの実行に必要なファイルを収容する
- FileServer.dart ： これがファイル・サーバのプログラム
- resources ： サーバがクライアントに開放するファイルを置くフォルダ
- README.md ： Githubには必要なマークダウン形式のファイルで、このリポジトリの内容を説明したもの

FileServer.dartの使い方

1. FileServer.dartを実行させる。IDE上でこのファイルを選択し、Run 'FileServer.dart'を実行する。コンソールには
   Serving /fserver on http://127.0.0.1:8080.
   と表示される。

2. 次に自分のブラウザからこのサーバを次のようにアクセスする:
   http://localhost:8080/fserver
   サーバは次のような初期画面を生成してブラウザ側に返す:



3. このサーバはresources/のディレクトリに置かれたファイルのみしかダウンロードを許していない。ここではまずこのディレクトリ内のファイルの一覧を表示してその中からユーザが選択できるようにしている。各ファイルの表示にはそのファイルを呼び出すリンクが張られている。例えばresources/Client.htmlの表示にはhttp://localhost:8080/fserver/resources/Client.htmlというリンクが張られている。

4. 従ってHTMLページを介さなくてもブラウザのアドレス・バーに直接http://localhost:8080/fserver/resources/Client.htmlを入力しても同じ結果が得られる。

5. もうひとつはテキスト・エリア経由でファイルの名前を入力してSubmitボタンでこれをサーバにクエリ文字列としてサーバに送信できるようにしてある。
つまりサーバは直接http://localhost:8080/fserver/resources/Client.htmlといった要求パスによる要求とhttp://localhost:8080/fserver?fileName=resources%2FClient.htmlといったクエリによる要求の双方に対応している。

6. テキスト・エリアからファイルを指定するときは、相対指定でも絶対指定でも可能である。前記のようにresources/readme.textと入力する代わりにC:/dart_apps_server/file_seerver/resources/readme.textといったように絶対パスを入力することも可能である(ここではこのアプリケーションがC:/dart_apps_serverというフォルダに含まれているとしている)。
絶対パスはIDE上でそのファイルを選択し、右クリック > Copy Pathでそのパスをコピーし、テキスト・エリアに張り付ければ良い。各自試して見られたい。

FileServer.dartのポイント

最初にクライアントからの要求処理を示す。ここではクライアントからの要求が到来した時点でその要求パスとクエリを調べ、どのように処置するかを判断している。

```
001   // process the request
002   completer.future.then((data){
003     try {
004       if (LOG_REQUESTS) {
005         print(createLogMessage(request, bodyString));
006       }
```

```
007       // selsect requests with 'fileName' query
008       if (request.queryParameters['fileName'] != null) {
009         new FileHandler().onRequest(request);
010       } else
011       // selest request without 'fileName' query
012       {
013         // select direct designation of the file
014         String fName = request.uri.path.replaceFirst(REQUEST_PATH, '');
015         if (fName.length > 2) {
016           fName = fName.substring(1);
017           if (fName.contains('resources/')) {
018             new FileHandler().onRequest(request, fName);
019           }
020           else new InitialPageHandler().onRequest(request,
021               'you can access files in resouces/ only!');
022         }
023         // new client, send initial page
024         else {
025 //        new FileHandler().onRequest(request, 'resources/Client.html');
026           new InitialPageHandler().onRequest(request);
027         }
028       }
029     }catch(err) {
030       print('File Handler error : $err.toString()');
031     }
032   });
```

1. この処理の本体は、002行目はクライアントからのHTTP要求を完全に受理したfuture.thenメソッドの関数リテラルとして記述されている。

2. 008行目でfileNameという名前のクエリが存在するかどうか、即ちテキスト・ボックスにファイル名が入力されSubmitボタンが押された結果の要求かどうかを調べている。その場合は直ちにFileHandlerのonRequestメソッドを呼び出している。

3. 014行目以降は正しい直接指定かどうかを調べている。該要求の要求パスの/server以降にresources/という文字列があるかどうかを調べ、存在するときは018行目で020行目でFileHandlerのonRequestメソッドを呼び出している。

4. そうでないときは20行目で正しくない要求だとしてyou can access files in resouces/ only!という表示を付けて初期画面をクライアントに返す。

5. そうでないときは初期画面をクライアントに返している。

このサーバの中心になっているのがFileHandlerというクラスであるので、次にこのクラスを説明する。

```
001 class FileHandler {
002
003   void onRequest(HttpRequest request, [String fileName = null]) {
004     try {
005       HttpResponse response = request.response;
006       if (fileName == null) {
007         fileName = request.queryParameters['fileName'];
008       }
009       if (LOG_REQUESTS) {
010         print('Requested file name : $fileName');
011       }
012       File file = new File(fileName);
013       String mimeType;
014       if (file.existsSync()) {
```

```
015         mimeType = mime.mime(fileName);
016         if (mimeType == null) mimeType = 'text/plain; charset=UTF-8'; // dafault
017         response.headers.set('Content-Type', mimeType);
018 //      response.headers.set('Content-Disposition', 'attachment; filename=\'$
{fileName}\'');
019         // Get length of the file for Content-Length header.
020         RandomAccessFile openedFile = file.openSync();
021         response.contentLength = openedFile.lengthSync();
022         openedFile.closeSync();
023         // Pipe the file content into the response.
024         file.openRead().pipe(response);
025       } else {
026         if (LOG_REQUESTS) {
027           print('File not found: $fileName');
028         }
029         new NotFoundHandler().onRequest(request);
030       }
031     } catch (err) {
032       print('File Handler error : $err.toString()');
033     }
034   }
035 }
```

1. クライアントから要求されたファイル名は007行目に示されるように"fileName”という名前でクエリ文字列から取得される。

2. 012行目に示すようにFileインターフェイスのオブジェクトはその名前を引数にして取得される。

3. 013に示すように、当該ファイルが存在するかどうかを知るのにexists()またはexistsSync()メソッドを使用する。両者の相違は、そのメソッドが実行完了まで留まる(ブロックされる)か、または非ブロッキングでFutureのイベントを使うかである。

4. ファイル名の拡張子(extension)によって"Content-Type"ヘッダでクライアントに知らせるMIME形式が異なるので、015行で示されているようにMIMEタイプもライブラリのメソッドを使用する。MIME.dartというライブラリは筆者が用意しpub.dartlangに登録したものであるが、これらの形式以外のファイル形式の場合はそれを追加する必要がある。IANAのサイトなどで確認して追加されたい。mime_typeライブラリのmime(String FileName)というメソッドは、該当するファイル・タイプが存在しないときはnullを返すので、そのときは016行目のようにブラウザのデフォルトのMIMEタイプを設定する。

5. コメントアウトされている"Content-Disposition", "attachment; filename=\"${fileName}\""というヘッダを追加するとブラウザの対応が異なってくるので、試されると良い。その値である”inline”はエンティティがユーザにすぐに表示されるべきであるのに対し、”attachment”はユーザがエンティティを閲覧するためには追加的行動をとる必要があることを示す。ここでは”attachment”と指定しているので、Chromeではブラウザに表示するのではなく、ダウンロード物として取り扱う。IEでは次のような問合せをしてくる:

6. ファイルを開くには022行に示すように同期して開くopenSyncと非同期(非ブロック)で開くopenのふたつのメソッドがある。スループットが問題になる場合を除いて、通常はコーディングが楽なopenSyncが使われそうである。次の行のlengthSyncも同様である。

7. 応答ヘッダ部分の設定が終わったので、020-024で該ファイルの中身を出力ストリームに書き込む。この際file.openRead()のpipeメソッドを使うと、024行目のように非常に簡単な記述で済んでしまう。デフォルトでは入力ストリームからのデータを総て書き込むと、出力ストリームは自動的に閉じるので、closeを実行する必要がない。

なおファイルの存在チェックに関しては、これまでは:

```
var f = new File('myfile');
f.onError = (e) => print(e);
f.exists((b) => print('exists: $b'));
```

という書き方だったが、2012年5月から次のように書くようAPIが変更されている:

```
var f = new File('myfile');
var existsFuture = f.exists();
existsFuture.then((b) => print('exists: $b'));
existsFuture.handleException((e) {
 print(e);
 return true;
});
```

この変更の利点は:

- 他のAPIのFuture使用方法と一貫させる。
- FileとDirectoryのオブジェクトがイミュータブル(不変:一度生成したら状態が決して変更されることがない)となる。逆にいえばその名前のファイルが付加されたときには、新たにFileオブジェクトを生成してその存在を確認する必要がある。
- エラー処理がローカルにできるし、エラー処理メカニズムをfutureに組み込ませるやりかたを活用できる。

と述べている。

なお、ファイルを読みだしてHttpResponseオブジェクトに書き込む最も簡単なFutureベースのメソッドは次のようになろう:

```
Future _streamFile( File file, HttpRequest request) {
   return request.response.consume(file.openRead());
}
```

または

```
Future _streamFile( File file, HttpRequest request) {
   return file.openRead().bind(request.response);
}
```

更には次のようにreduceを使う手段もある:

```
Future _streamFile( File file, HttpRequest request) {
   return file.openRead().reduce(null, (_, data) {
      request.response.writeBytes(data);
   });
}
```

但しこの場合はFutureが完了したら自分でresponse.close()を実行させねばならない。

MIMEライブラリ

例えばInternet Explorerでは、Internet Explorerが受け付けるContent-TypeをContent-Typeヘッダにセットしてやらないと、たとえWordの文書であってもInternet Explorerはこれを正しく表示しないことがあるので注意が必要である。ブラウザが自分のウィンドウ内で開けないドキュメントはダウンロード扱いとなる。

しかしながら特にAcceptヘッダでブラウザから提示されていないタイプであっても、setContentTypeで正しいMIMEタイプをブラウザに教えてやるのは良い習慣である。

mime_typeはその為のライブラリである。このライブラリのメソッドは：

- String **mime**(String fileName)
- String **mimeFromExtension**(String extension)

の2つである。**mime**(String fileName)はファイル名（即ち拡張子を含む）を指定するとそれに対応したMIMEタイプを戻し、**mimeFromExtension**(String extension)は、拡張子を指定するとそれに対応したMIMEタイプを戻す。該当するMIMEタイプが無いときはともにnullを返す。

19.8節　　セッション管理

セッション管理はウェブ・サーバには欠かせない機能であり、その概要は筆者の「改訂サーブレット・チュートリアル」の第9章を見て頂きたい。

当初HttpServerはセッション管理に関するAPIを用意していなかった。これは多分：

- サード・パーティが開発することを想定している、及び
- HTTPSを視野にいれており、HTTPSではクライアント確認がなされるので、クッキーなどを使わなくても済む。
- HTTPよりはSPDYやWebSocketに集中している。

ということが理由になっていたと推定される。

しかしながら筆者からの要請を受け、DartチームのIO担当グループは2012年10月22日にr13870としてIOライブラリに追加した。これはHttpSession.dataという単なるオブジェクトがバインドできるというシンプルなものだったが、それでも簡単なラッパをかぶせることでJava Servlet並みの機能を提供することが出来た。これはシンプル性を追求するというDartの目標に沿ったものであろう。

その後2013年2月11日にdart:ioライブラリにという抽象クラスが追加され、単なるオブジェクトではなくてMapを実装させ、サーブレットと同じように名前と値のペアでオブジェクトをバインド出来るようにした。これに加え同時にisNewも付加されたこともあり、ラッパを用意しなくても基本的なアプリケーションには十分な機能が得られるようになっている。

DartのHttpSessionクラス

Dartが用意しているAPIはdart.io.HttpSession、dart.io.HttpServer.sessionTimeout()及び dart.io.HttpRequest.sessionである。詳細はこの章の終りにある関連ライブラリの邦訳を見て頂きたい。

HttpSessionオブジェクトはHttpRequestオブジェクトの属性として取得する。

セッションのタイムアウト管理はHttpServerの属性としてセットできる。

```
set sessionTimeout(int timeout)
```

従ってセッションごとのタイムアウト設定はできないことに注意されたい。

またセッションに対するオブジェクトのバインドはHttpSessionそのものがMapであるので簡単である。

```
void operator []=(K key, V value)
```

どのような型でも値としてバインド可能である。

セッション管理のメカニズム

HttpServerにおけるセッション管理はクッキー・ベースで行われる。つまり当該クライアントとのセッションが維持されるということは、サーバが渡したこのセッションに対応したクッキー（そのセッション固有のID文字列）をそのクライアントがHTTP要求ヘッダの中に含めて送信しているということになる。HttpSession session([init(HttpSession session)])というメソッド呼び出しにより、サーバは該要求に対するセッション・オブジェクトが存在するかどうかを調べる。存在しない場合は新規のセッションIDを持ったセッション・オブジェクトを渡し、これを返す。このオブジェクトはHttpServerが一括管理する。つまり:

- クライアントに返すHTTP応答メッセージのなかにそのIDを含めたクッキー・ヘッダ(set-cookieヘッダ行)を含める。例えば:
  set-cookie: DARTSESSID=6d2afc1b0bdadb50eae546f568ec0c90 HttpOnly
- アプリケーションからのrequest.**session**ゲッタ呼び出しにより、当該要求に対するセッション・オブジェクトを返される。
- このオブジェクトには当該セッション固有のデータがキーと値のペアとしてセットできる。具体的にはショッピング・カートのようなこのセッション（クライアント）固有のオブジェクトのセット/ゲットである。
- 管理しているセッション・オブジェクトたちの各々にたいし、当該オブジェクトに対応したHTTP要求が次に到来するまでの期間がタイムアウト時間を経過したかどうかを調べる。タイムアウトしたものはこれを必要があればアプリケーションに通知する（void set **onTimeout**(void callback())セッタを使用する）。これはServletには無い機能であるが、この関数を使わなくてもそのアプリケーションの通常のセッション処理（画面遷移処理など）あるいは例外発生処理で済ませても構わない。あるクライアントがそのアプリケーションをアクセス中に何らかの理由でタイムアウト時間以上長く席を外してしまい、その後引き続きそのアプリケーションにアクセスするということは頻繁であり、アプリケーションは画面遷移シーケンス・チェック等で必ずそれに対応できるように作成される。この関数はその為のひとつの手段を提供している。
- HttpServerは唯一つのタイムアウト時間しかもてない。Servletのようにセッションごとにタイムアウト時間を設定できないことに注意しなければならない。
- **destroy**()メソッド呼び出しで当該セッションを破棄できる。たとえばショッピング・カートのアプリケーションでクライアントが商品発注の手続きを完了したときに当該セッションを破棄する。廃棄してもその要求処理の期間はrequest.session()メソッドは現在の当該セッションを返す。次の要求からは新規のセッションが割り当てられることに注意。

なおクッキーに対しては、2012年5月にdart:ioの中にCookieというインターフェイスが追加されている。これは筆者が指摘した時点で既に担当のGjesseが追加作業を行っていた。

ブラウザのクッキー保持

クッキーはブラウザのインスタンスが保持している。一般的にはExpires属性による期限指定がないクッキーに対しては、最近のブラウザたちはそれをセッション・クッキーだとみなして、そのブラウザのセッションの終了時点（そのブラウザのインスタンスが閉じる時点）でそれらのクッキーを削除している。従って、PC上に同じブラウザの複数のタブやウィンドウを立ち上げても、クッキーの値は共有されることに注意のこと。つまり同じPC上では例えば複数のChromeのタブやウィンドウを開いても、それらは同じクッキー、つまり同じセッションが適用される。

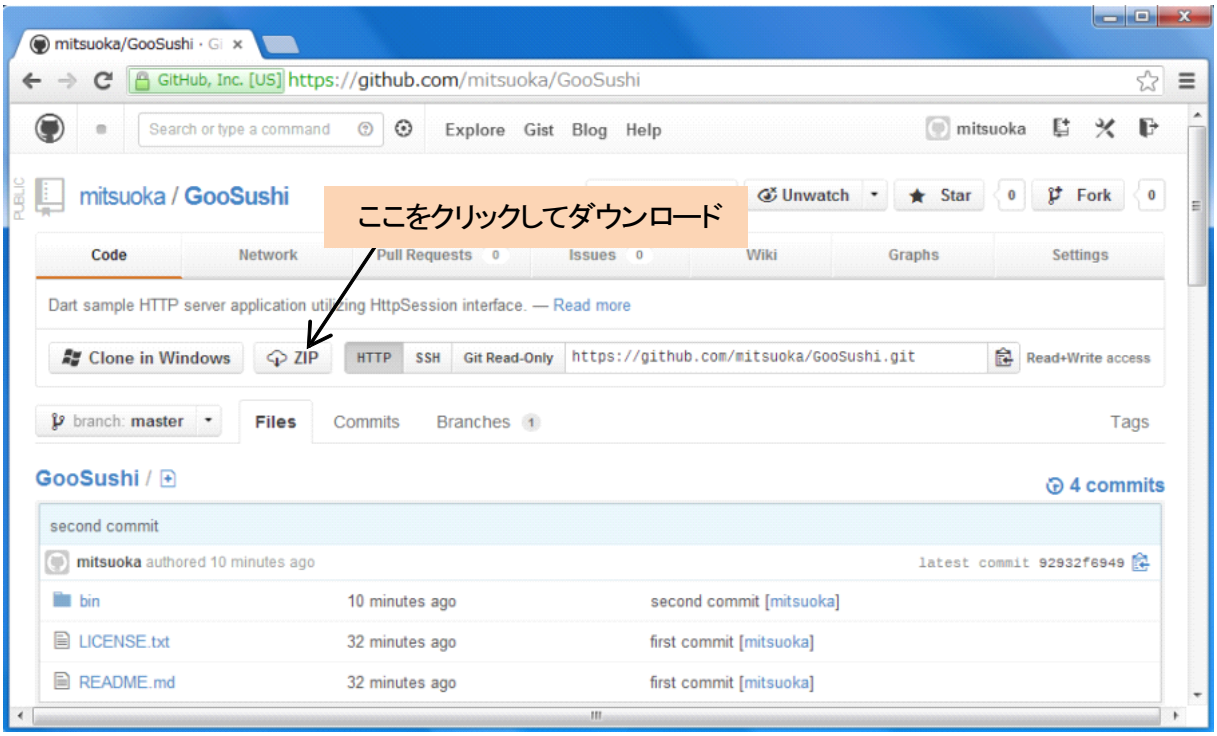
Internet Explorer、 Opera、Safari、及びChromeの総てがそのような動作になっている。しかしながら、Firefox（3.0.9時点）ではそのような規則に準じておらず、そのブラウザを閉じたときだけでなくOSを再スタートしたときでもそのクッキーは存続するという報告がある。これはFirefoxの設計によるもので、Firefoxを閉じるときにタブを保存するかどうかを聞いてくるが、「保存して終了」を選択すると再立ち上げしたときにこれまでの状態に復旧する。これを「セッション復旧(session restore)」と呼んでいる。もしそれを望まないときは、タブの総てを閉じてからブラウザを閉じれば良い。

簡単なテスト・サーバによる確認

このサーバを動作させるには：

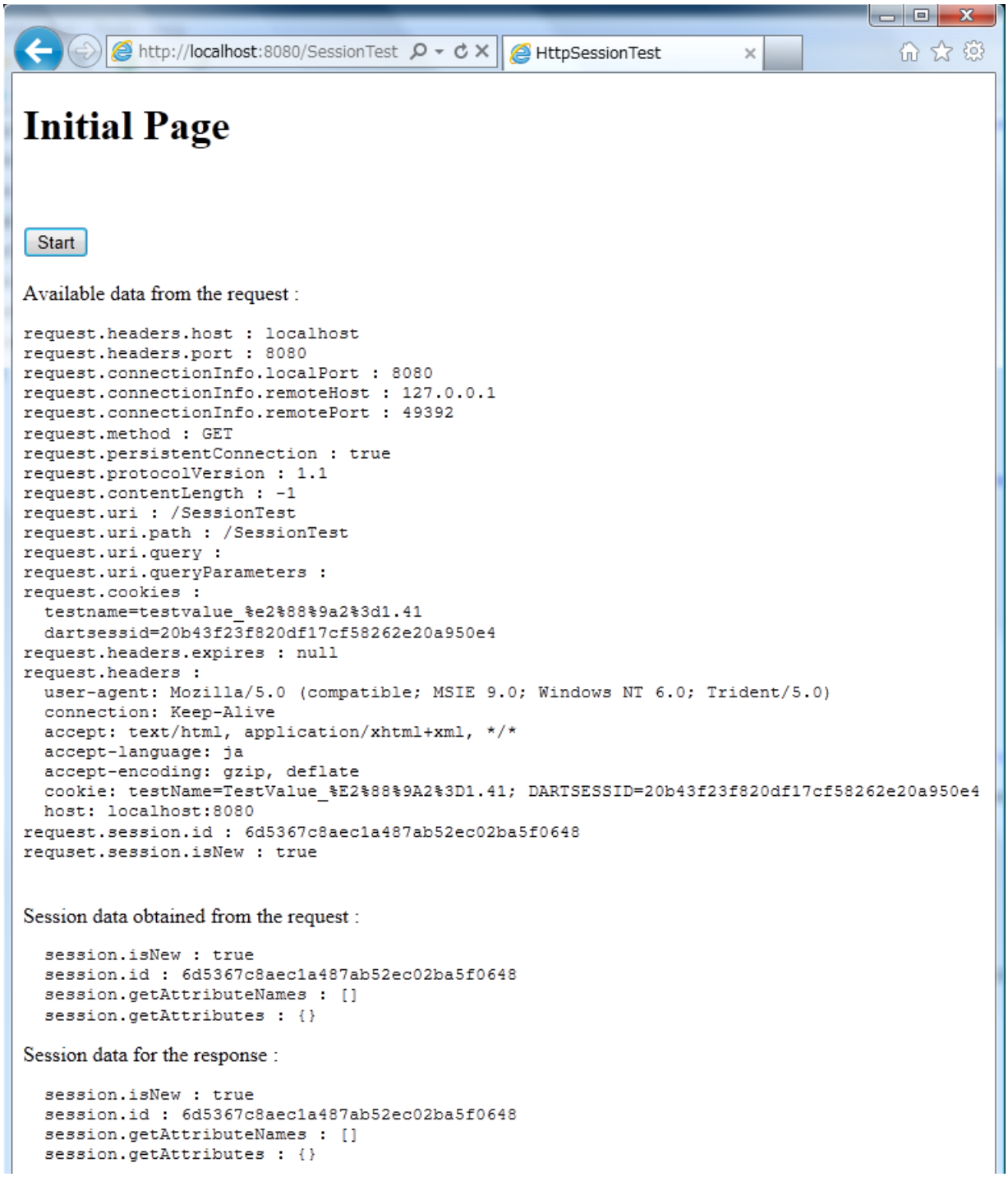
1. 最初にGithubにあるGooSushiのリポジトリをアクセスしてダウンロード/解凍して、適当な名前のホルダに収納する。：
    1. HttpSessionTestServer.dart
    2. SimpleShoppingCartServer.dart

    ダウンロードは下図のようにZIP圧縮ファイルのダウンロードのボタンをクリックすれば良い。



2. 自分のIDE上でこのアプリケーションのためのプロジェクトを用意する。
3. そのプロジェクト上で、File → Openで解凍したGooSushiのホルダを選択しOkをクリックする
4. ファイル・ビュー上でそのホルダにあるHttpSessionTest.dartを選択する
5. run → Run 'HttpSessionTest.dart'でこのサーバ・アプリケーションの実行を指定する。
6. サーバが起動するとServing the SessionTest on http://127.0.0.1:8080.とコンソールに表示する。
7. Chromeまたはその他のブラウザからhttp://localhost:8080/SessionTestをアクセスする。

最初に次のような初期画面が表示されよう:



この画面では:

- クライアントからのHTTP要求に対応したオブジェクトから得られるデータ
- その要求から得られるセッション関連データ
- クライアントに応答を返す時点でのセッション関連データ

が表示されている。要求オブジェクトに関するデータは既に解説済みであるので省略する。セッションに関する情報はHttpSessionクラスのものではなく、それをラップしたSessionというクラスからの情報である。このようなラッパ

を使用したのは、Java Servletに馴染んだユーザには（名前、値）ペアによる属性のバインドがより使い勝手が良いだろうとの判断による。このクラスは後で説明するが、ここでは

- isNewはこのセッションが新規に生成されたものかどうか
- sessionIdはHttpServletが用意したこのセッションの為のID
- getAttributesはこのセッションにバインドされた属性の一覧（Map）
- getAttributeNamesはこのセッションにバインドされた属性たちの名前のリスト（List）

が表示されている。

この初期画面でstartボタンをクリックすると次のような表示が出よう：

```
Page 1

 Session will be expired after 20 seconds.



 Available data from the request : request.headers.host : localhost
request.headers.port : 8080
request.connectionInfo.localPort : 8080
request.connectionInfo.remoteHost : 127.0.0.1
request.connectionInfo.remotePort : 49398
request.method : GET
request.persistentConnection : true
request.protocolVersion : 1.1
request.contentLength : -1
request.uri : /SessionTest?command=Start
request.uri.path : /SessionTest
request.uri.query : command=Start
request.uri.queryParameters :
  command : Start
request.cookies :
  testname=testvalue_%e2%88%9a2%3d1.41
  dartsessid=6d5367c8aec1a487ab52ec02ba5f0648
request.headers.expires : null
request.headers :
  user-agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; Trident/5.0)
  connection: Keep-Alive
  accept: text/html, application/xhtml+xml, */*
  accept-language: ja
  accept-encoding: gzip, deflate
  cookie: testName=TestValue_%E2%88%9A2%3D1.41; DARTSESSID=6d5367c8aec1a487ab52ec02ba5f0648
  host: localhost:8080
  referer: http://localhost:8080/SessionTest
request.session.id : 979b1fd290234e91493eee54050f7cab
requset.session.isNew : true

 Session data obtained from the request :   session.isNew : true
  session.id : 979b1fd290234e91493eee54050f7cab
  session.getAttributeNames : []
  session.getAttributes : {}
 Session data for the response :   session.isNew : true
  session.id : 979b1fd290234e91493eee54050f7cab
  session.getAttributeNames : [pageNumber]
  session.getAttributes : {pageNumber: 1}
```

このデータから次のことが理解されよう：

- 初期画面からStartボタンを押したことでクライアント（即ちブラウザ）から送信されるHTTP要求には名前がDARTSESSIDで値が6d5367c8aec1a487ab52ec02ba5f0648であるクッキーが含まれている。これは以

前の使われていたセッションIDをサーバに送り返した為である。しかしながらサーバではこのIDは無効化されているのでこの要求が到来した時点で新しいセッションを用意する必要があると判断している。即ちrequset.session.isNew : trueとなっている。

- 実はこの画面をクライアントに返した時点でサーバがHTTP応答の中にset-cookie: DARTSESSID=979b1fd290234e91493eee54050f7cabというヘッダ行を含めている。クライアントはそのクッキーを保管し、以後このサーバに送信する要求にはこのクッキーを返すことで、サーバはこの要求がどのセッションに対応したものかを知ることが出来る。
- このセッションにはこの要求が来た時点では属性がバインドされていない。つまりsession.attributesは空の状態である。しかしながらこの画面をクライアントにHTTP応答として渡す時点では、名前がpageNumber、値が1というデータが属性としてセットされている。これはこのクライアントには1ページめの画面を送ったという情報である。次回同じセッションIDを持った要求が到来したときには、サーバはそのクライアントは1ページめの画面から要求を送ってきたということが判る。この属性は画面遷移に良く使用される。

クッキーはブラウザ単位で保持される。従って同じPC上で複数のタブからこのサーバをアクセスしてもそれは同じクライアントと見做される。同じPC上でChromeとIEを立ち上げ各々からこのサーバをアクセスすれば、別のクライアントと見做されるので、各自確認されたい。

オブジェクトのセッションへのバインド

オブジェクトのセッションへのバインドは、上記のような画面遷移決定の目的だけではなく、いわゆるショッピング・カートで良く使用される。クライアントがあるショッピングのアプリケーションにアクセスしているとき、そのセッション（即ちそのクライアント）に対する固有の買物情報をショッピング・カートのオブジェクトとしてバインドする。後の節ではその具体的なプログラムを説明する。

Dartは動的な型づけの言語であるので、Javaと違ってセッションに値としてバインドしたオブジェクトを取得するのにキャストする必要は無い。また単一スレッドであるのでスレッドに対する配慮の必要がない。

セッションのタイムアウト

この状態で20秒間以上放置すると、IDEのコンソールには次のようなonTimeout(void callback())でセットしたコールバック関数の出力が表示される：

```
2013-02-23 20:40:38 : timeout occurred for session 979b1fd290234e91493eee54050f7cab
```

これは、このHttpSessionオブジェクトがタイムアウトを起こしたことを通知するものである。

HTTPプロトコルにはそのクライアント間がアクティブでなくなった（このサイトから別のサイトに移ってしまった）ことを示す手段は存在しないので、サーバがそうだと判断する為の手段としてタイムアウトが使われる。つまり当該セッション・オブジェクトが所定時間を経過してもアクセスされないことでタイムアウトだと判断している。DartのHttpServerはセッションごとにタイムアウトを持つことができず。総てのセッションに対して適用される。HttpServerの持っているデフォルトのタイムアウト時間は20分間である。set sessionTimeout(int timeout)というセッタでこのタイムアウト時間を秒単位で指定できる。このHttpSessionTest.dartアプリケーションでは20秒がセットされている。

HttpServerがあるHttpSessionオブジェクトがタイムアウトを起こしたことを検出したら：

- onTimeout(void callback())でセットしたコールバック関数が実行される。
- 当該HttpSessionオブジェクトを廃棄する。即ちdestroy()メソッドが呼ばれる。
- 次回のそのクライアント（当該セッションID）からのHTTP要求のセッション・クッキーは無視され当該HttpSessionオブジェクトは削除される。またHttpRequest.session呼び出しに対しては新規のHttpSessionオブジェクトが返される。

onTimeoutコールバック関数は、当該HttpSessionオブジェクトが存続していれば呼び出される。この関数を使えば、顧客がそのサイトでのセッションの途中で昼食等で席を離れてしまったりしたことが判り、その時点で何らかの対策（例えばその顧客とのセッションの放棄、DBセッションの開放、携帯メール経由での顧客への通知など）をとることができる。このコールバックはサーブレットには無かったもので、顧客管理のひとつの情報源となり得る。

セッションの廃棄

セッションの廃棄はdestroyとかdiscardとかという言葉が良く使われる。Servletではinvalidateと呼んでいる。

例えばオンライン・ショッピングのアプリケーションでは、あるクライアントが幾つかの画面を遷移しながら在庫確認から発注、最終確認までの一連の手順が終了した時点で、そのセッションが終了したことになる。そのクライアントが次回このアプリケーションをアクセスしたときは、このクライアントは再度このアプリケーションの最初からショッピングの手順を踏めば良い。そのような場合には当該セッションの廃棄が良く使用される。

アプリケーションが明示的にセッションを廃棄しなくても：

- アプリケーションで再度そのクライアントがアクセスしたときでも、これまでのショッピングの手順が完了していることを知り、そのクライアントには最初の手順を踏ませることができる。
- タイムアウトのメカニズムが使われなくなったセッション・オブジェクトを整理してくれるので、そのようなオブジェクトが蓄積されてしまい、サーバの負荷となることが防止される。

DartのHttpSession抽象クラスでは、destroy()というメソッドが用意されている。このメソッドは：

- このメソッドが呼ばれても、このオブジェクトそのものが消えるわけではない。従って、このメソッドを呼んだあとでもこのオブジェクトにはアクセスが可能である。
- 次回のそのクライアント（当該セッションID）からのHTTP要求のセッション・クッキーは無視され当該HttpSessionオブジェクトは削除される。またHttpRequest.session呼び出しに対しては新規のHttpSessionオブジェクトが返される。

例えばあるページからNew Sessionというボタンをクリックして初期画面に戻ったときの画面には次のようなセッション・データが表示される：

```
Session data obtained from the request :
  session.isNew : false
  session.sessionId : 97df64cbfad26bcbeca5dfb9381298ab
  session.attributes : {pageNumber: 4}
  session.getAttributeNames : [pageNumber]

Session data for the response :
  session.isNew : false
  session.sessionId : 97df64cbfad26bcbeca5dfb9381298ab
  session.attributes : {pageNumber: 4}
  session.getAttributeNames : [pageNumber]
```

New Sessionというボタンをクリックしたときの HTTP要求に対しては、このサーバは実は当該HttpSessionオブジェクトのdestroy()メソッドを呼び出している。それにもかかわらずHTTP応答を返す時点ではこのオブジェクトへのアクセスが可能であり、その内容も変化していない。

Http only クッキー・パラメタ

通常クライアントに送るセッションIDの為のクッキー・ヘッダ行にはHttpOnlyが付加されているが、Dartでは当初これが付加されていなかった。しかし筆者からの指摘が即座に受け入れられ、これが付加された。Java ServletではHttpOnlyの付加がデフォルトとなっている。Java Servlet仕様書の新しい3.0版に書かれているように、HttpOnlyクッキーはクライアントに対しこれらのクッキーがクライアント・サイドのJavascriptのコードからは見えなくするべきであることを示している（これはそのクライアントがこの属性を探すことを知っていないかぎりフィルタされない）。HttpOnlyクッキー使用はある種のサイトをまたぐスクリプティング攻撃を最小化するのにも寄与する。

HttpSessionTest.dartの概要

このプログラムのコードはGithubから取得できる。

Sessionクラス

SessionクラスはHttpSessionをラップして、これまでの（名前、値）ペアで属性をバインドすることに馴染んでいるServletユーザのために用意したものである。出来ればこのクラスはライブラリにしたほうが好ましい。このクラスにより、Javaで書かれたアプリケーションをDartに移植するのが容易になる。このクラスをHttpRequestのオブジェクトを引数にしてインスタンス化することで、HttpSessionのHttpRequest.session()メソッドが呼び出される。ある要求処理プロセスの中で何回もこのクラスのオブジェクトを生成してもSessionのオブジェクトの内容はisNewを除いて変化せず、問題は起きない。

このクラスのフィールドは：

String **id**	当該セッションのID
bool **isNew**	このセッションが新規のもの、つまり当該要求に該当するセッションが存在せず、このオブジェクト生成時に初めてこれに対するセッションがHttpServerの中に作成されたことを示す
HttpSession **session**	HttpSessionのオブジェクトで、アプリケーションが直接扱うことは無い

メソッドは：

Session(HttpRequest request)	コンストラクタで、要求オブジェクトを引数とする
getAttribute(String name)	ある名前を持った値（オブジェクト）を取得
void **setAttribute**(String name, dynamic value)	（名前、値）のペアであるオブジェクトを属性としてこのセッションにバインドする
List **getAttributeNames**()	バインドされている属性たちの名前のリストを取得

Map **getAttributes**()	総ての属性を含むMapを取得
void **removeAttribute**(String name)	指定した名前の属性を削除
void **invalidate**()	このセッションを無効化する。これは次の同じセッションIDを持った要求から効果を持つ

以下はそのコードである。

```
/*
 * Session class is a wrapper of the HttpSession
 * Makes it easier to transport Java server code to Dart server
 */
class Session{
  HttpSession _session;
  String _id;
  bool _isNew;
  Session(HttpRequest request){
    _session = request.session;
    _id = request.session.id;
    _isNew = request.session.isNew;
    request.session.onTimeout = (){
    print("${new DateTime.now().toString().slice(0, 19)} : "
        "timeout occured for session ${request.session.id}");
    };
  }

  // getters
  HttpSession get session => _session;
  String get id => _id;
  bool get isNew => _isNew;

  // getAttribute(String name)
  dynamic getAttribute(String name) {
    if (_session.containsKey(name)) {
      return _session[name];
    }
    else {
      return null;
    }
  }

  // setAttribute(String name, dynamic value)
  void setAttribute(String name, dynamic value) {
    _session.remove(name);
    _session[name] = value;
  }

  // getAttributes()
  Map getAttributes() {
    Map attributes = {};
    for(String x in _session.keys) attributes[x] = _session[x];
    return attributes;
  }

  // getAttributeNames()
  List getAttributeNames() {
    List names = [];
    for(String x in _session.keys) names.add(x);
    return names;
  }
```

```
  // removeAttribute()
  void removeAttribute(String name) {
    _session.remove(name);
  }

  // invalidate()
  void invalidate() {
    _session.destroy();
  }
}
```

タイムアウト処理

このコードではコンストラクタの中でHttpSessionオブジェクトを取得したあとで次のようなタイムアウト処理を付加している:

```
    request.session.onTimeout = (){
    print("${new DateTime.now().toString().slice(0, 19)} : timeout occured for session $
{request.session.id}");
    };
```

ここではこのオブジェクトがタイムアウトを起こしたときにそれをタイムスタンプつきでコンソールに出力するだけである。タイムアウトは既に説明したようにアプリケーションの中でいろんなやり方で処理でされるが、このコールバック関数はその為のひとつの手段になり得る。

要求処理ハンドラ

要求処理ハンドラは到来要求を処理し応答をクライアントに返すベースとなる関数である。この関数の中で004行から015行の範囲でこの間に生じた何らかの例外をトラップし、そのことをエラー・パージとしてクライアントに返す。これによりサーバはサービスを継続できる。

008行目ではNew Sessionというボタンのクリックで到来した要求に対してはsession.invalidate()つまりHttpSessionのdestroyメソッドを呼んでいる。これは既に述べたように現在の処理中のsessionオブジェクトには影響を与えないので、010行目の新たなSessionオブジェクトの取得は意味がない。この行は読者の確認実験の為に置かれている。

020行目にクッキー設定の例を示している。このクッキーはセッション・クッキーの為のset-cookieヘッダとは別のset-cookieヘッダで送信される。最近の仕様書(RFC 6265)の第3章では「set-cookieフォールディング」は使用すべきではないと書かれているので、これに準拠している。クッキーを含めたHTTPヘッダ行で使われる文字コードはASCIIに限定されるので、URLエンコードしなければならない。setCookieParameterという関数はその為にあり、引数の名前と値は多バイトのユニコードであっても構わないよう配慮されている。ここではtestName=TestValue_√2=1.41はtestName=TestValue_%E2%88%9A2%3D1.4と安全な文字列に変換されクッキーとして保持される。

```
001 void requestReceivedHandler(HttpRequest request, HttpResponse response) {
002   String responseBody;
003   Session session;
004   try {
005     reqLog = createLogMessage(request);
006     if (LOG_REQUESTS) print(reqLog.toString());
007     session = new Session(request); // get session for the request
008     if (request.queryParameters["command"] == "New Session") {
009       session.invalidate(); // note: HttpSession.destroy() is effective from the next request
010       session = new Session(request);  // this call has no effect
011     }
```

```
012     sesLog = createSessionLog(request, session);
013     if (LOG_REQUESTS) print(sesLog.toString());
014     responseBody = createHtmlResponse(request, session);
015   } catch (err) {
016     responseBody = createErrorPage(err.toString());
017   }
018   response.headers.add("Content-Type", "text/html; charset=UTF-8");
019   // cookie setting example (accepts multi-byte characters)
020   setCookieParameter(response, "testName", "TestValue_v2=1.41", request.path);
021   response.outputStream.writeString(responseBody);
022   response.outputStream.close();
023 }
```

画面遷移

画面遷移はcreateHtmlResponse(request, session)のなかで行われている。画面遷移の処理は遷移表を使う、セッションにバインドされたデータ(どのページにある筈だなどの)をもとに判断する、あるいはHTMLのformで指定されたデータ(hiddenやボタンの値)を使うなどの手段があろう。ここでは簡単にボタン・クリックの値で判断している。

19.9節　　ショッピング・カートのアプリケーション・サーバ

オブジェクトのセッションへのバインドは、いわゆるショッピング・カートで良く使用される。クライアントがあるショッピングのアプリケーションにアクセスしているとき、そのセッション(即ちそのクライアント)に対する固有の買物情報をショッピング・カートのオブジェクトとしてバインドする。以下の節ではその具体的なサーバ・プログラムを説明する。このサーバはDartが提供しているネーティブなHttpSessionクラスを使用したものと、筆者が用意したServlet等価のHttpSessionライブラリを使用したものの2つが存在する。どちらも主要な箇所は殆ど相違がないことを、読者は比較・確認されることをお勧めする。

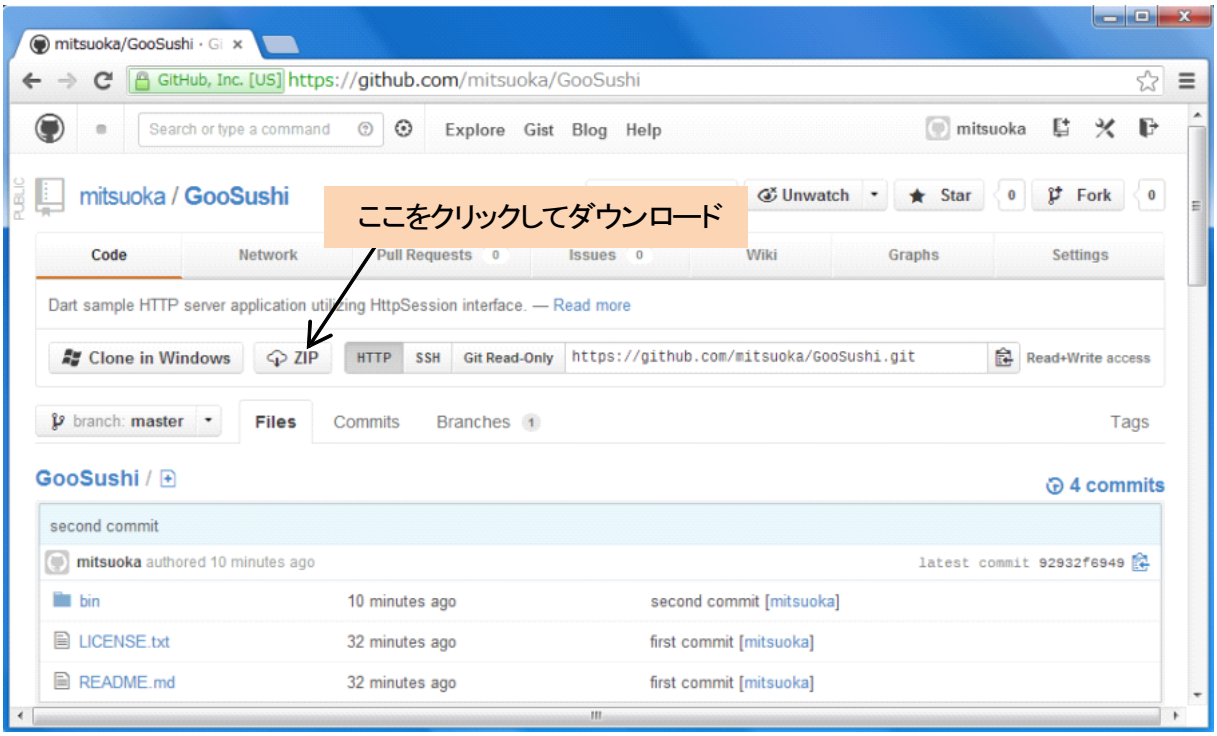
これらのプログラムで使っているSessionというラッパ・クラスあるいはHttpSessionライブラリを用いることで、従来のJavaで書かれたサーバ・アプリケーションを比較的容易にDartサーバに移植できる。

このサーバ・アプリケーションはGooSushiという寿司屋の注文から、注文の確認、代金清算までを含むサービスを表現したものである。

GooSushiプログラムの実行手順

1. 最初にGithubにあるGooSushiのリポジトリをアクセスしてダウンロード/解凍して、適当な名前のホルダに収納する。前節でHttpSessionTest.dartをIDE上で開いている場合は、この手順は不要である:
    1. HttpSessionTestServer.dart
    2. SimpleShoppingCartServer.dart

    ダウンロードは下図のようにZIP圧縮ファイルのダウンロードのボタンをクリックすれば良い。



2. 自分のIDE上でこのアプリケーションのためのプロジェクトを用意する。

3. そのプロジェクト上で、File → Openで解凍したGooSushiのホルダを選択しOkをクリックする

4. ファイル・ビュー上でそのホルダにあるSimpleShoppingCartServer.dartを選択する

5. run → Run 'SimpleShoppingCartServer.dart'でこのサーバ・アプリケーションの実行を指定する。

6. サーバが起動するとコンソールには次のような表示がされる:

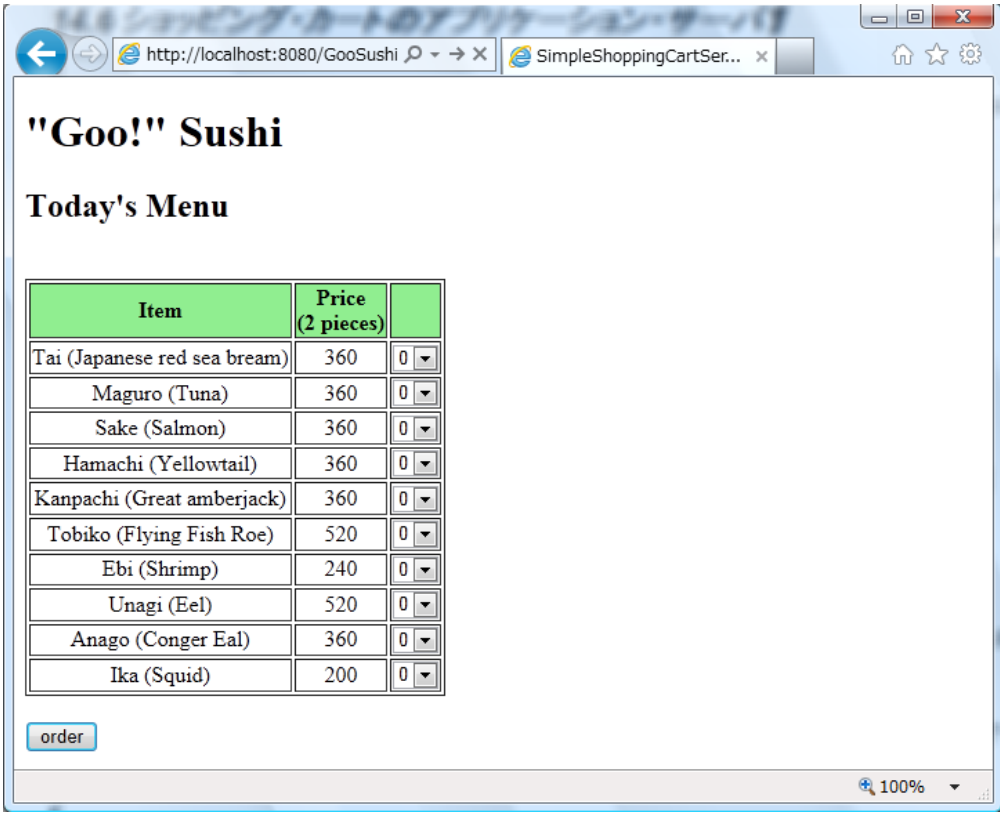
```
today's menu
itemCode : 150, itemName : Tobiko (Flying Fish Roe), perItemCost : 520.0
itemCode : 160, itemName : Ebi (Shrimp), perItemCost : 240.0
itemCode : 170, itemName : Unagi (Eel), perItemCost : 520.0
itemCode : 180, itemName : Anago (Conger Eal), perItemCost : 360.0
itemCode : 190, itemName : Ika (Squid), perItemCost : 200.0
itemCode : 260, itemName : Kanpachi (Great amberjack), perItemCost : 360.0
itemCode : 270, itemName : Hamachi (Yellowtail), perItemCost : 360.0
itemCode : 280, itemName : Sake (Salmon), perItemCost : 360.0
itemCode : 290, itemName : Maguro (Tuna), perItemCost : 360.0
itemCode : 300, itemName : Tai (Japanese red sea bream), perItemCost : 360.0
Serving /GooSushi on http://127.0.0.1:8080.
```

最初に用意されている本日のメニューをアイテム・コード(整数)の若い順にアイテムの名前、及び単価のリストを表示している。その後このサーバはIPアドレスがローカル・アドレス(ループバック・アドレス)でTCPポート番号が8080でクライアントからの要求の受付開始をしたことを表示している。

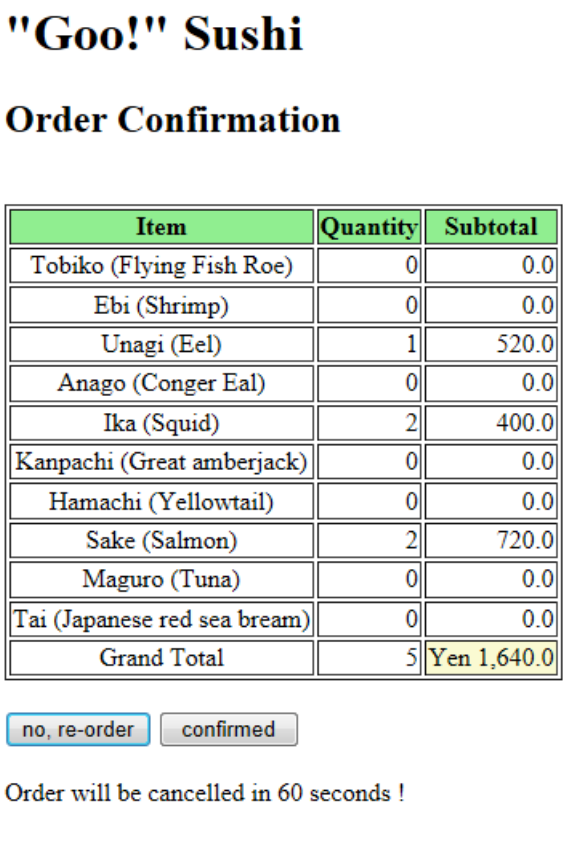
実行例

このプログラムを以下のように実行する:

1. 自分のブラウザのアドレス・バーに、http://localhost:8080/GooSushiと入力してこのサーバをアクセスする。
そうすると下図のようなメニュー画面が表示される。

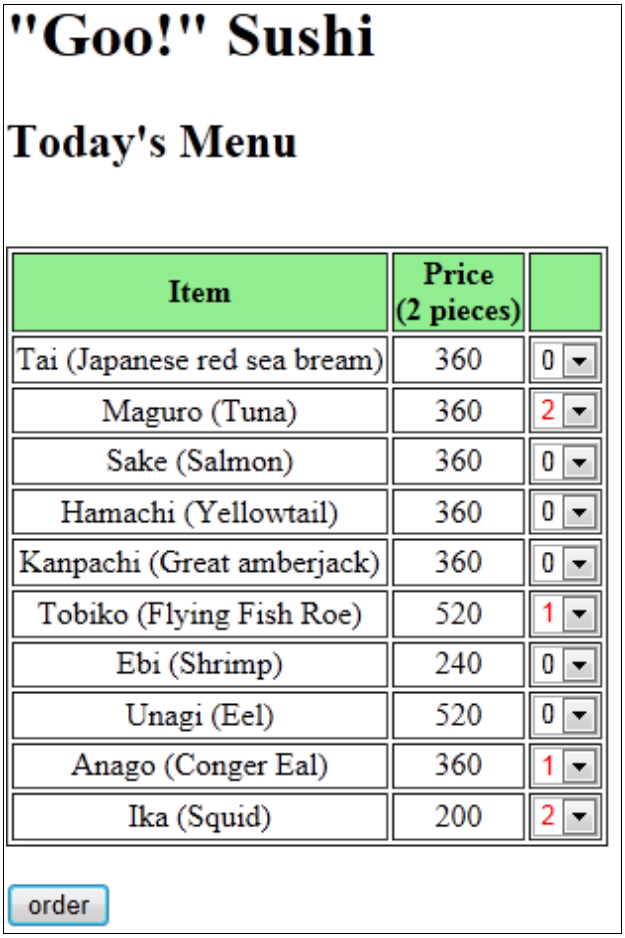


2. ここで適当な数量を選択メニューから選択し、orderボタンをクリックすると、確認画面に遷移する。



3. Confirmedボタンをクリックすると最終画面に遷移する。

4. 確認画面で注文しなおしの為にno, re-orderボタンをクリックすると、最初のメニュー画面に遷移するが、既に選択したアイテムの個数が赤で表示されているところが相違している。



ショッピング・カート

ショッピング・カートはそのカートに入れる為の商品アイテムのMap(_items)がその主たる構成要素である。各アイテムは商品コード（int _itemCode）をキーとしてアクセスできる。それ以外のこのカートに関する情報、例えばこの例では総額_amountと_orderedAtがフィールドとして存在し、これらの要素に対するセッタとゲッタが用意されている。

```
class ShoppingCart {
  Map<int, ShoppingCartItem> _items;
  double _grandTotal;
  DateTime _orderedAt;

  // constructor
  ShoppingCart() {
    _items = new TodaysMenu().items;
    _grandTotal = 0.0;
  }
....
```

それ以外にこのカート内の各アイテムの追加と削除などのメソッドも存在する。

各アイテムは商品コード(_itemCode)、商品名(_itemName)、購入数量(_qty)、単価(_perItemCost)、小計(_subtotal)などがその要素になり、これらの各要素はセッタ及びゲッタでアクセスする。これはJavaのBeansと似た構成である。

```
class ShoppingCartItem {
    String _itemCode;
    String _itemName;
    int _qty;
    double _perItemCost;
    double _subTotal;
....
```

HTTPサーバにおける例外とエラーの捕捉とエラー・ページ

Dartでは発生した例外は、そのコードを呼び出した側に伝搬する。最終的にどの呼び出し側でもその例外を捕捉しないと、サーバは停止する。一般にサーバ・アプリケーションでは、サーバの停止は極力避けねばならない。従って、しかるべき場所で例外を捕捉し、エラー・ページでそれをクライアントに通知するのが一般的な対処法である。エラー・ページはJSPではデフォルトのページが用意されているが、Dartでは自分で用意することになる。

このプログラムでは、基となる要求処理のための関数requestReceivedHandlerで発生した例外を捕捉している。この関数の役割は、到来した要求に対しセッション・オブジェクトを取得し、その要求とセッションをHTML応答を作成する関数に渡し、得られたHTMLテキストを出力ストリームに書き込み、クライアントにその応答を送信することである。加えて、このアプリケーションのパラメタであるLOG_REQUESTSがtrueになっていれば、到来要求及びそれに対するセッションの情報をコンソールに出力する。

この関数の中で発生した総ての例外をtry { ~ } catchで捕捉し、エラー・ページとしてクライアントに送信している。

```
void requestReceivedHandler(HttpRequest request, HttpResponse response) {
```

```
  var htmlResponse;
  try {
    if (LOG_REQUESTS) print(createLogMessage(request).toString());
    var session = getSession(request, response);
    if (session != null){
      if (session.isNew()) session.setMaxInactiveInterval(MaxInactiveInterval);
    }
    if (LOG_REQUESTS) print(createSessionLog(session, request).toString());
    htmlResponse = createHtmlResponse(request, session).toString();
  } catch (err) {
    htmlResponse = createErrorPage(err).toString();
  }
  response.headers.add("Content-Type", "text/html; charset=UTF-8");
  response.outputStream.writeString(htmlResponse);
  response.outputStream.close();
}
```

なお、アプリケーションによってはrequestReceivedHandlerのなかでFutureやStreamのオブジェクトを使用する場合もあろう。**この場合はそれらのオブジェクトで発生したエラーはonErrorやcatchErrorで捕捉し、これをExceptionのオブジェクトとしてスローする必要がある**:

```
throw new Exception('exception raised');
```

そうすればこれはrequestReceivedHandlerのtryブロックのcatch文で捕捉される。

HttpServerのエラー

それではサーブレット・コンテナに相当する部分であるHttpServerではエラーはどう処理されているのだろうか？Googleの技術者のAnders Johnsen は次のように説明している:

- HTTP要求のリスン中はエラーは発生しない。不完全なHTTPヘッダがあればそれは無視され、当該ソケットはクローズされる。
- 完全なHTTP要求が受信されたら、HttpRequestオブジェクトが生成される。ボディ部分のリスン中（例えばrequest.listen(...)とり出し中）に、もしそのボディ全部が受信される前に当該ソケットが閉じてしまったときはエラーが生成される。ここではソケットが閉じられるが、このエラーはユーザに対してこの要求データが終了していないことを警告するために生成されている。GET要求のようなボディ部を持たない要求の場合はエラーは発生し得ない。
- HttpResponseにデータを送信中はソケット関連のエラーは発生しない。応答にデータを送信中にエラーが発生するただ2つの例は:

1. HTTP規約の違反。ひとつのシナリオは Content-Lengthで設定した値と実際のコンテント長が異なった場合。このときは HttpExceptionが投げられる。
2. 応答にエラーを送信した場合。例えば:

```
var response = ...;
var future = new File("non_existing_file").openRead().pipe(response);
```

このコードは HttpResponseにFileSystemExceptionを送信し、pipe'から返されるfutureはerrorで完了する。

- HTTP応答にデータを送信中に当該ソケットがクローズされていると、そのデータは単に無視される。

したがって上記の1と2を除いて、HttpServer抽象クラスで発生する例外を除いて、ユーザはサーブレットの場合と同様に気にする必要はない。

1と2の例外はこのアプリケーションでは次のように捕捉・処理されている:

```
        request.response.done.then((d){
          if (LOG_REQUESTS) print ("${new DateTime.now()} : "
              "sent response to the client for request ${request.uri}");
        }).catchError((e) {
          print ("${new DateTime.now()} : Error occured while sending response: $e");
        });
```

つまりエラーが生じたことをコンソールに表示するだけである。これをつけなくてもサーバは停止することはなく、サービスを継続する。

なお2014年からはZoneが利用できるようになった。これを使うとあるゾーン内での非同期エラーはそのゾーンで確実に捕捉できるようになる。詳細は次節の「サーバの動作継続の為のZone」を参照されたい。

ショッピング・カートのセッションへのバインド

このアプリケーションでは、ショッピング・カートのオブジェクトは顧客がメニュー・ページ上で選択した鮨の商品コードと個数のリストをもとに、注文確認のページを作成する際に生成される。以下は注文確認ページ作成のための関数(createConfirmPage)の一部である:

```
// create a shopping cart
var cart = new ShoppingCart();
request.queryParameters.forEach((String name, String value) {
  int quantity;
  if (name.startsWith("pieces_")) {
    quantity = Math.parseInt(value);
    if (quantity != 0) {
      var cartItem = new ShoppingCartItem();
      cartItem.itemCode = name.substring(7);
      cartItem.qty = quantity;
      cartItem.itemName = menu[cartItem.itemCode].itemName;
      cartItem.perItemCost = menu[cartItem.itemCode].perItemCost;
      cartItem.subTotal = cartItem.perItemCost * quantity;
      cart.addItem(cartItem);
    }
  }
}); // cart completed
cart = sortCart(cart); // sort
session.setAttribute("cart", cart); // and save it to the session
```

1. 最初にショッピング・カートのオブジェクトを用意する。
2. 要求パラメタを調べる。
3. 要求パラメタの中で、注文個数が0で無いものに対しカート・アイテムのオブジェクトを用意する。
4. そのアイテムに対し、メニュー・テーブルをもとにそのカート・アイテムに必要な情報をセットする。
5. 出来たアイテムをショッピング・カートに追加する。
6. 最後にそのカートのオブジェクトをcartという名前でセッション・オブジェクトにバインドする。

バインドされたショッピング・カートのオブジェクトは、サンキュー・ページの作成及び再注文のページで使用される。

ページ(画面)遷移

到来した要求に対し、次にどのページを表示するかの判断にどのメカニズムを使うかは、各アプリケーションの内容とプログラマの流儀によって異なってこよう。

- セッションに現在どのページかなどの情報をバインドする。
- HTMLの"hidden"属性を使って、現在どのページなのか等の情報を伝える。
- 入力フォームからのパラメタたちの名前と値から判断する。
- クッキーを利用する。

いずれにしても、ユーザは予期せぬ要求を発生させる可能性がある。例えば:

- ブラウザ上で別のタブのアドレス・バーに、現在のタブのアドレス・バーをコピーしたものを貼り付け、更にそのクエリの部分を加工してこのサーバをアクセスする。
- 現在のアドレス・バーのクエリ部分の内容を変更して、このサーバをアクセスする。

そのような不正なアクセスに対し、間違った処理をすることなく、初期ページに遷移させる、あるいはクライアントにその要求は不正だと通知することが必要である。その為に、ページ遷移の条件は極力厳格にするのが推奨される。

このアプリケーションでは以下に示すように、ボタンの名前と値からどのページに遷移させるかを判断している。

```
StringBuffer createHtmlResponse(HttpRequest request, HttpSession session) {
  if (session.isNew() || request.queryString == null) {
    return createMenuPage();
  }
  else if (request.queryParameters.containsKey("menuPage")) {
    return createConfirmPage(request,session);
  }
  else if (request.queryParameters["confirmPage"].trim() == "confirmed") {
    StringBuffer sb = createThankYouPage(session);
    session.invalidate();
    return sb;
  }
  else if (request.queryParameters["confirmPage"].trim() == "no, re-order") {
    return createMenuPage(cart : session.getAttribute("cart"));
  }
  else {
    session.invalidate();
    return createErrorPage("Invalid request received.");
  }
}
```

サンキュー・ページを生成したあと、及びエラー・ページ生成時に、当該セッションを無効化していることに注意されたい。

19.10節　サーバの動作継続の為のZone

前節で例外とエラーに対する対処を説明したが、サーバ動作をより堅牢化するために2014年からdart:asyncに抽象クラスのZoneとstaticメソッドのrunZonedが追加された。この節はFlorian Loitsch及びKathy Walrathの両名によるZone解説書の前半に準拠しているので、残りの後半はこの解説書を見ていただきたい。

ZoneというのはLispで言う動的エクステント(dynamic extent)の非同期版ともいえる。非同期コールバック関数たちはそれが待ち行列に入っていたと同じゾーンで実行される。例えばあるfuture.thenコールバックはそれらが呼び出されたと同じゾーン内で実行される。

Zonesの最も一般的な使いみちは非同期で実行されるコードのなかで生起されたエラーの取り扱いである。例えばシンプルなHTTPサーバでの使い方は以下のようなコードとなる:

```
runZoned(() {
  HttpServer.bind('0.0.0.0', port).then((server) {
    server.listen(staticFiles.serveRequest);
  });
},
onError: (e, stackTrace) => print('Oh noes! $e $stackTrace'));
```

このHTTPサーバをあるゾーン内で走らせることで、このサーバの非同期コード内で捕捉されないエラー(uncaught errors:但し致命的でないエラー)が起きてもこのアプリケーションを停止させないようにできる。

注意:この使用例は必ずしもzoneを必要とするものではない。dart:isolateが将来捕捉されないエラーに対するAPIを有するようになると考えられる。

Zonesを使うと以下のタスクが可能になる:

- 前例で示したように非同期コード内でスローされた処理されない例外の為にアプリケーションが止まってしまうのを防ぐ。
- データ(ゾーン・レベルの値たちとして知られる)を個々のゾーンたちに結びつける。
- そのコード内の一部またはすべてのなかでのprint()あるいはscheduleTask()といった限定されたメソッドのセットのオーバライド。
- そのコードがあるゾーンに入るあるいは出る度にある操作(タイマの開始や停止、あるいはスタック・トレースの保管など)を実行する。

読者は他の言語でzoneに似たものに出くわしたことがあるかもしれない。Node.jsにおけるDomainがDartのZoneのもとになっている。Javaのthread-localストレージも似たものである。最も近いのはDartのzonesをBrian Ford氏がJavaScriptにポートした zone.jsで、彼のビデオが参考になる。

Zoneの基礎

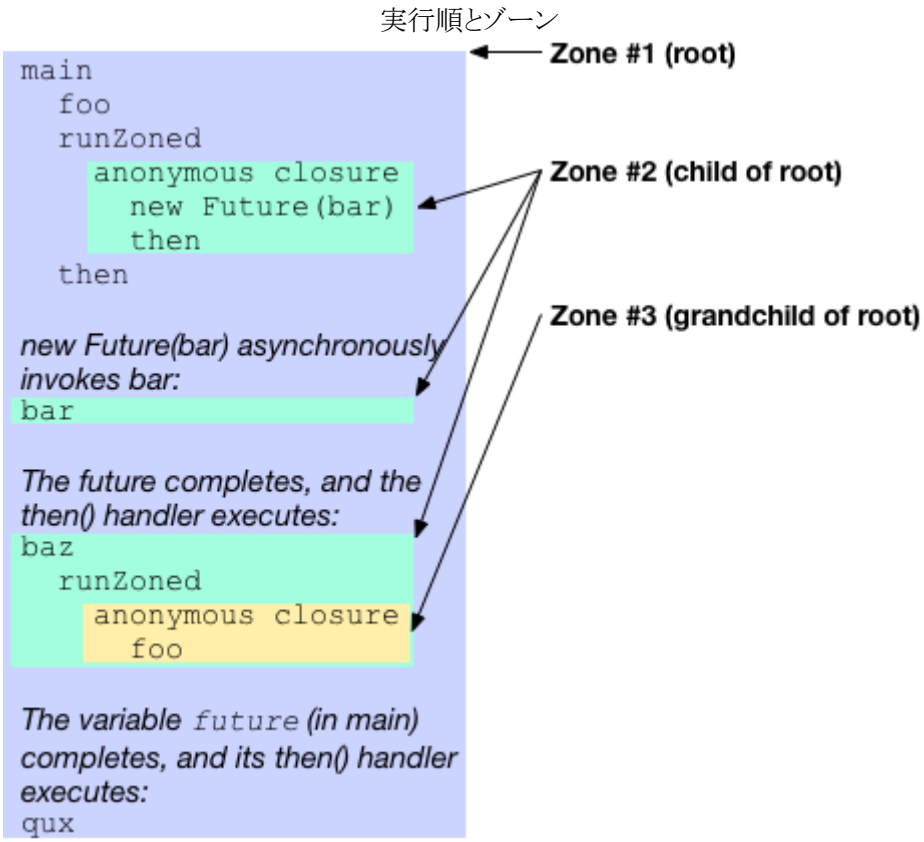
Zoneはある呼び出しの非同期動的エクステントを表現している。これはそのコードによって登録されている呼び出し及び(推移的なものとしての)非同期コールバックたちの一部として実行される計算である。前述のHTTP

サーバの例では、bind()、then()、およびthen()のコールバックの総てが同じゾーン、即ちrunZoned()で生成されたzone内で実行される。

次の例を考えてみよう。このコードはzone #1（ルート・ゾーン）、zone #2、およびzone #3の3つのゾーンが走る。

```
import 'dart:async';

main() {
  foo();
  var future;
  runZoned(() {            // 新規の子供のゾーンを開始させる(zone #2).
    future = new Future(bar).then(baz);
  });
  future.then(qux);
}

foo() => ...foo-body...   // 2回実行 (2つのzone内で各々).
bar() => ...bar-body...
baz(x) => runZoned(() => foo()); // 新規の子供のzone(zone #3).
qux(x) => ...qux-body...
```

下図はこのコードの実行順とこのコードがどのゾーン内で走るかを示している：



runZoned()が呼び出される度に新しいzoneが生成され、そのzone内でコードが実行される。そのコードがあるタスク（例えばbaz()呼び出し）のスケジューリングするときは、そのタスクはそれがスケジュールされたzone内で走る。例えば、qux()（main()の最後の行）呼び出しは、例えそれ自身が zone #2内で走るfutureが付加されていても、

zone #1（ルート・ゾーン）内で走る。

子のzoneは親のzoneに完全に置き換わる訳ではない。そうではなくて、新しいzoneは自分たちを取り巻いているzoneの内部にネストされる。例えば、zone #2は zone #3を含んでおり、zone #1（ルート・ゾーン）はzone #2とzone #3の双方を含んでいる。

総てのDartコードはルート・ゾーン内で走る。コードは他のネストした子供のzone内で走る場合もあるが、少なくとも常にルート・ゾーン内で走る。

非同期エラーの取り扱い

最も使われるZonesの機能のひとつは、非同期で実行しているコード内での捕捉されないエラーが取り扱えられることである。このコンセプトは同期コードにおけるtry-catchと似ている。

「処理されないエラー」はしばしばthrowを使ってcatch文で捕捉されない例外を生起するコードが原因となる。これはhttp_serverのpubライブラリを使用したときなどで発生することがある。捕捉されないエラーを起こすもうひとつの手段はnew Future.error()あるいはCompleterのcompleteError()メソッドを呼び出すことである。

Zone化したエラー・ハンドラ（そのzone内の捕捉されないエラー発生ごとに呼び出される非同期エラー・ハンドラ）をインストールするには、runZoned()のonError 引数を使う。例えば：

```
runZoned(() {
  Timer.run(() { throw 'Would normally kill the program'; });
}, onError: (error, stackTrace) {
  print('Uncaught error: $error');
});
```

このコードでは例外を発生する非同期コールバック（Timer.run()を介した）がある。通常この例外は処理されないエラーになり、トップ・レベルまで伝搬してしまう（即ちスタンド・アロンのDart実行コードではその実行プロセスを止めてしまう）。然しこのゾーン化されたエラー・ハンドラではこのエラーはエラー・ハンドラに渡され、このプログラムを停止させない。

try-catchとゾーン化されたエラー・ハンドラの顕著な相違点は捕捉されないエラーが発生したとしてもそのゾーンは実行を継続することである。そのゾーン内で他の非同期コールバックがスケジュールされていたときは、それらのコールバックも実行を継続する。その結果ゾーン化されたエラー・ハンドラは複数回呼び出される可能性がある。

またあるエラー・ゾーン（エラー・ハンドラを持ったゾーン）はそのゾーンの子供（あるいは孫など）内で発生したエラーも取り扱う。future変換（then()またはcatchError()を使った）のシーケンスのなかでエラーたちはどこで取り扱われるかはシンプルなルールに基づく：

- Futureチェイン上のエラーはエラー・ゾーンのバウンダリを決して跨がない

あるエラーがエラー・ゾーンのバウンダリに達したら、その時点でそれは処理されないエラーとして取り扱われる。

エラー・ゾーンに入れないエラーの例

次の例では、最初の行で生起されたエラーはエラー・ゾーンに入れない。

```
import 'dart:async';
main() {
  var f = new Future.error(499); //次のイベント・ループでエラーを発生させるFuture
  f = f.whenComplete(() { print('Outside runZoned'); });
  runZoned(() {
    f = f.whenComplete(() { print('Inside non-error zone'); });
  });
  runZoned(() {
    f = f.whenComplete(() { print('Inside error zone (not called)'); });
  }, onError: print);
}
```

この例を実行させると次のような出力となる:

```
Outside runZoned
Inside non-error zone
Uncaught Error: 499
Unhandled exception:
499
...stack trace...
```

runZoned()呼び出しを削除するまたはonError引数を削除すれば、次のような出力となる:

```
Outside runZoned
Inside non-error zone
Inside error zone (not called)
Uncaught Error: 499
Unhandled exception:
499
...stack trace...
```

ゾーンたちのどれか、またはエラー・ゾーンを削除するとそのエラーはさらに伝搬することに注意のこと。

このエラーはあるエラー・ゾーンの外部で発生しているので、スタック・トレースが出力される。このコード全体を囲むエラー・ゾーンを付加すれば、このスタック・トレースを発生させなくできる。

エラー・ゾーンを抜けられないエラーの例

前の例で示されているように、エラーはゾーンを跨げない。同じように、エラーはエラー・ゾーンの外に伝搬できない。次の例を考えてみよう:

```
import 'dart:async';
main() {
  var completer = new Completer();
  var future = completer.future.then((x) => x + 1);
  var zoneFuture;
  runZoned(() {
```

```
    zoneFuture = future.then((y) => throw 'Inside zone');
  }, onError: (error) {
    print('Caught: $error');
  });
  zoneFuture.catchError((e) { print('Never reached'); });
  completer.complete(499);
}
```

例えfutureチェインがcatchError()で終了していたとしても、この非同期エラーはこのエラー・ゾーンから出られない。このエラーはゾーンのエラー・ハンドラ（onErrorで指定されている）が取り扱う。その結果、zoneFutureは決して値やエラーで完了することはない。

ストリームでZoneを使う

ストリームとゾーンのルールはfutureよりもシンプルである：

- 変換およびその他のコールバック関数たちはそのストリームがリスンされているゾーン内で実行される

このルールはストリームはリスンされるまでは何の副作用も持ってはいけないという指針に沿ったものである。同期コードにおける似たような状況はIterableたちの振る舞いで、この場合は値たちを取りに行くまでは計算されない。

例：runZoned()でStreamを使う

runZoned()でStreamを使った例を以下に示す：

```
var stream = new File('stream.dart').openRead()
    .map((x) => throw 'Callback throws');

runZoned(() { stream.listen(print); },
        onError: (e) { print('Caught error: $e'); });
```

このコールバックでスローされた例外はrunZoned()のエラー・ハンドラで捕捉される。出力は次のようになる：

```
Caught error: Callback throws
```

この出力が示すように、このコールバックはリスンしているゾーンに結び付けられていて、map() が呼び出されているゾーンではない。

19.11節　マルチアイソレート化

ウェブ・サーバでは多数のクライアント要求をマルチスレッドで処理している。dart:ioライブラリではこれまでそのよ

うな並行処理のためのアイソレート・ベースのAPIが用意されていなかった。

アイソレートでウェブ・サーバを構成するには次のような障壁がある:

- アイソレート生成の為のオーバヘッド及びメモリ使用量が大きい
- メッセージ交換のためのスループット(直列化が必要)及び渡すオブジェクトの制約(Socketオブジェクトは渡せない)

反面、dart:ioはDart2JSにおけるweb workersからの束縛から解放されている(VMのみが対象)ので、専用APIの発展の自由度が大きい。

これらの問題に関しては以前から議論されていた(例えばServer-side performanceと題した討議)が、その実装は遅れていた。しかしながら2014年5月21日にGoogleのdart:io担当のAnders JohnsenはDart 1.4版で実験的なServerSocketReference(ServerSocket参照)を用意し、これを取得する属性をServerSocket抽象クラスに組み込んだことを発表した。

この機能はあくまでも実験的なもので、Linuxに限定されているが、将来の展開が期待される。

ServerSocket参照はServerSocketの属性として取得され、他のアイソレートたちに渡すことができる。この参照をもとにオリジナルのServerSocketのクローンを生成できる。このクローンはネーティブのソケットをオリジナルのServerSocketと共有する。このことにより到来TCP接続を複数のアイソレート上で受け付けることができる。

例えば次のような最も単純なHTTPサーバを考えよう:

```
void listen(HttpServer server) {
  server.listen((HttpRequest request) {
    request.response.write("Hello, world");
    request.response.close();
  });
}

void main() {
  HttpServer.bind('127.0.0.1', 8080).then(listen);
}
```

このサーバはメインのアイソレートのみで実行される。このサーバをHTTPベンチマーク・ツールのwrkで次のようにアクセスする(256接続、15秒間、4スレッド):

```
$ wrk -H 'Host: localhost' -d 15 -c 256 -t 4 http://localhost:8080/
```

彼の実験ではこの場合毎秒25,000要求のスループットが得られている。

次にこれ(listenメソッドが要求処理)をServerSocket参照を使ったマルチ・アイソレートのサーバにすることを考えてみよう:

```
void handle(reference) {
  // アイソレートのワーカ、WebSocket参照からWebSocketを生成する
  reference.create().then((serverSocket) {
    listen(new HttpServer.listenOn(serverSocket));
```

```
  });
}

void main() {
  ServerSocket.bind('127.0.0.1', 8080).then((server) {
    // サーバ・ソケットへの参照を取得
    var ref = server.reference;
    for (int i = 1; i < 8; i++) {
      // 'ref'を引数としてアイソレートを産み付け
      Isolate.spawn(handle, ref);
    }
    listen(new HttpServer.listenOn(server));
  });
}
```

この場合、7個の新規アイソレートたちに参照を送信しているので、メインを含めて8個のアイソレートでこのサーバ・ソケットをlistenでリスンすることになる。

これを同じ'wrk'コマンドでアクセスするとこのサーバのスループットは8倍の毎秒約200,000要求に向上することになる。

この機能を使うとポートを使ったやり取りが不要になるので、スループットが向上する。

19.12節　関連APIの和訳

dart:io.HttpServer

2013年2月の改訂でHttpServerはStreamを実装した。2014年8月にはより安全性を強化するためにデフォルトが追加された。

Factory **HttpServer抽象クラス**	
実装	
Stream<HttpRequest>	
コンストラクタ	
HttpServer.**listenOn**(ServerSocket serverSocket)	既存のServerSocketにこのHTTPサーバを付加する。この HttpServerが閉じたときはこのHttpServerは単に自らを切り離し、現行の接続を閉じるが、serverSocketは閉じない。
staticメソッド	
Future<HttpServer> **bind**(address, int port,	指定されたaddressとport上でのHTTP要求の受付を開始する。

{int backlog: 0, bool v6Only: false, bool shared: false})	addressはString またはInternetAddressのいずれかであり得る。もしaddressがStringのときは、bindはInternetAddress.lookupを実行し、そのリストの最初の値を採用する。ローカル・ホストからの到来接続のみを受け付けるループバック・アダプタ上でリスンするときは、この値としてInternetAddress.LOOPBACK_IP_V4またはInternetAddress.LOOPBACK_IP_V6 を使用すること。 ネットワークからの到来接続を受け付けるときは、総てのインターフェイスにバインドするためにInternetAddress.ANY_IP_V4またはInternetAddress.ANY_IP_V6という値のどれかを使用するか、特定のインターフェイスのIPアドレスを使用する。 もしIPバージョン6(IPv6)アドレスが使われているときは、IPバージョン6(IPv6)とIPバージョン4(IPv4)の接続の双方とも受け付けられる。IPバージョン6(IPv6)アドレスのみに制限したいときは、IPバージョン6のみとセットするために v6Onlyを使用する。 もし0のポートが指定されたときは、このサーバはエフェメナル・ポートを選択する。オプショナルな引数のbacklogは下位層のOSのリスンのセットアップの為にバックログのリスンを指定する為に使える。 オプショナルな引数である backlogは下位に存在しているOSのリスン設定に対しリスン・バックログを指定するのに使われる。backlogが0（デフォルト値）という値を持っているときは、このシステムで妥当な値が選択される。 オプショナルな引数であるsharedは同じaddress, port 及びv6Onlyの組み合わせにたいし、同じDartのプロセスからの追加的なバインドを可能とするかどうかを指定する。もしsharedがtrueのときは、付加的なバインドが実行され、到来接続はHttpServerたちのセット間で分配される。これを使う一つの手段としては、複数のアイソレートたちに到来接続たちを配分させることである。
Future<HttpServer> **bindSecure**(address, int port, SecurityContext context, {int backlog: 0, bool v6Only: false, String certificateName, bool requestClientCertificate: false, bool shared: false})	addressはString またはInternetAddressのいずれかであり得る。もしaddressがStringのときは、bindはInternetAddress.lookupを実行し、そのリストの最初の値を採用する。ローカル・ホストからの到来接続のみを受け付けるループバック・アダプタ上でリスンするときは、この値としてInternetAddress.LOOPBACK_IP_V4またはInternetAddress.LOOPBACK_IP_V6 を使用すること。 ネットワークからの到来接続を受け付けるときは、総てのインターフェイスにバインドするためにInternetAddress.ANY_IP_V4またはInternetAddress.ANY_IP_V6という値のどれかを使用するか、特定のインターフェイスのIPアドレスを使用する。 もしIPバージョン6(IPv6)アドレスが使われているときは、IPバージョン6(IPv6)とIPバージョン4(IPv4)の接続の双方とも受け付けられる。IPバージョン6(IPv6)アドレスのみに制限したいときは、IPバージョン6のみとセットするために v6Onlyを使用する。 もしportが0という値であるときは、このシステムではエフェメラル・ポートがこのシステムで選択される。実際に使われているポート番号はportゲッタで取得できる。 オプショナルな引数である backlogは下位に存在しているOSのリスン設定に対しリスン・バックログを指定するのに使われる。backlogが0（デフォルト値）という値を持っているときは、このシステムで妥当な値が選択される。

	ニックネームまたは識別名（DN: Distinguished Name）をもった証明書のcertificateNameは証明書データベース内で検索され、サーバ認証用に使用される。もし requestClientCertificateがtrueのときは、そのサーバはクライアントたちにクライアント証明書で認証するよう要求する。 オプショナルな引数であるsharedは同じaddress, port 及びv6Onlyの組み合わせにたいし、同じDartのプロセスからの追加的なバインドを可能とするかどうかを指定する。もしsharedがtrueのときは、付加的なバインドが実行され、到来接続はHttpServerたちのセット間で分配される。これを使う一つの手段としては、複数のアイソレートたちに到来接続たちを配分させることである。
属性	
InternetAddress get **address**	このサーバがリスンしているアドレスを返す。このアドレスがhostnameからの検索で捕まえられているときに、これは使われている実アドレスを取得するのに使える。
HttpHeaders get **defaultResponseHeaders**	総ての応答オブジェクトに付加されるヘッダたちのデフォルトのセット。デフォルトでは、以下の次のセットがヘッダたちとなる: Content-Type: text/plain; charset=utf-8 X-Frame-Options: SAMEORIGIN X-Content-Type-Options: nosniff X-XSS-Protection: 1; mode=block もしServerのヘッダがここで付加され、serverHeaderもまたセットされているときは、serverHeaderの値が優越する。
final Future<T> **first**	（Streamから継承） 最初の要素を返す。これが空のときはStateErrorを返す。そうでないときはこのメソッドはthis.elementAt(0)と等価である。
final bool **isBroadcast**	（Streamから継承） このストリームが放送ストリームかどうか。
Duration **idleTimeout**	アイドルのキープ・アライブ接続に使われるタイムアウトを取得または設定する。直前の要求が完了したあとのidleTimeout内に更なる要求が到来しないときは、該接続は落とされる。 デフォルトの値は120秒である。 アイドル接続が放棄されるには最大2 * idleTimeoutの時間がかかることに注意。 このタイムアウトを無効とするにはidleTimeoutにnullをセットする。
final Future<bool> **isEmpty**	（Streamから継承） このストリームがなにか要素を含んでいるかどうかを報告する。
final Future<T> **last**	（Streamから継承） 最後の要素を返す。もし空のときはStateError.をスローする。
final Future<int> **length**	（Streamから継承） このストリームの中の要素の数を数える。
final int **port**	このサーバが要求を待っているポート番号を返す。これはlisten呼び出しで指定されたportの値が0のときに実際のポートを取得するのに使える。
abstract set **sessionTimeout**(int	このHTTPサーバでのセッションたちのタイムアウトを秒単位でセットする。デフォ

timeout)	ルトは20分間である。
String **serverHeader**	このHttpServerで生成された総ての応答のための本サーバのヘッダのデフォルト値を取得またはセットする。 serverHeaderがnullのときは、各応答にはサーバのヘッダは付加されない。 デフォルト値はnullである。
final Future<T> **single**	（Streamから継承） 単一の要素を返す。もし空のとき、あるいはひとつ以上の要素があるときはStateErrorをスローする。
メソッド	
Future<bool> **any**(bool test(T element))	（Streamから継承） このストリームが用意している要素のどれかをtestが受け付けるかどうかをチェックする。 その答えが判ったときにFutureを完了させる。もしこのストリームがエラーを報告したときは、このFutureはエラーを報告する。
Stream<T> **asBroadcastStream**()	（Streamから継承） これと同じイベントたちを作り出す複数受信のストリームを返す。もしこのストリームが単一受信のときは、複数の受信者が許される新しいストリームを返す。その最初の受信者が付加されたときにこれはこのストリームに受信し、最後の受信者がキャンセルされたときに再度受信解除される。 もしこのストリームが既に放送ストリームであるときは、これは手を加えられないで返される。
Stream **asyncExpand**(Function Stream convert(T event))	（Streamから継承） オリジナルのイベントあたりあるストリームのイベントたちを持った新しいストリームを生成する。 これはexpandのように機能するが、convertがIterableの代わりにStreamを返すことが異なる。返されたストリームのイベントたちが作られた順番に返されるストリームのイベントたちとなる。 convertがnullを返す時は、あたかも空のストリームが返されたごとく、出力のストリームには値がセットされない。
Stream **asyncMap**(Function convert(T event))	（Streamから継承） このストリームの各データ・イベントが新しいイベントに非同期でマップされた新しいストリームを生成する。 このメソッドはmapのように畿央駿河、convertがFutureを返せること、そしてこの場合はこのストリームはその結果で継続する前にfutureが完了するのを待つ点で異なる。 このストリームがそうであれば返されるストリームはブロードキャスト・ストリームである。

abstract void **close**()	クライアントからの要求待ち状態を停止する。これによりこのストリームは完了イベントでクローズする。
abstract HttpConnectionsInfo **connectionsInfo**()	このサーバが取り扱っている現在のTCPソケット接続の概要をHttpConnectionsInfoオブジェクトとして返す。
Future<bool> **contains**(T match)	（Streamから継承） このストリームが用意した要素たちのなかでmatchが生じるかどうかをチェックする。 この答えが判ったときにこのFutureを完了させる。このストリームがエラーを報告したときは、該Futureはそのエラーを報告する。
Stream<T> **distinct**([bool equals(T previous, T next)])	（Streamから継承） もし以前のデータ・イベントと同じのときはこれらのデータ・イベントをスキップする。 返されるストリームは、ふたつの連続したデータが決して同じで無いということを除いて、このストリームと同じイベントを作る。 用意されるequalsメソッドによって等しいかどうかが判断される。もしこれがオミットされているときは、最後に用意されたデータ要素には'=='演算子が使われる。
Future<T> **elementAt**(int index)	（Streamから継承） このストリームのindex番目のデータ・イベントの値を返す。 もしエラーが起きれば、このfutureはエラーで終了する。 もしこのストリームが閉じる前にindex要素たちより少ない数しか用意していないときは、エラーが報告される。
Future<bool> **every**(bool test(T element))	（Streamから継承） このストリームが用意している総ての要素をtestが受け付けるかどうかをチェックする。 この答えが判った時点でこのFutureを完了させる。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream **expand**(Iterable convert(T value))	（Streamから継承） 各要素をゼロまたはそれ以上のイベントたちに変換する新しいストリームをこのストリームから生成する。 各到来イベントは新しいイベントたちのIterableに変換され、これらの新しいイベントたちの各々が次に順番に返されたストリームによって送信される。
Future<T> **firstWhere**(bool test(T value), {T defaultValue()})	（Streamから継承） testにマッチするこのストリームの最初の要素を見つける。 testがtrueを返すこのストリームの最初の要素で満たされたfutureを返す。 このストリームが完了する前にそのような要素が見つからず、またdefaultValue関数が用意されているときは、defaultValue呼び出しの結果がそのfutureの値となる。 エラーが発生したとき、あるいはこのストリームが一致する要素が見つからないで終了し、またdefaultValue関数が用意されていないときは、このfutureはエラーを受信する。

Future **fold**(initialValue, Function combine(previous, T element))	（Streamから継承） 繰り返しcombineを適用することで値たちのシーケンスを減らす。
Future **forEach**(Function void action(T element))	（Streamから継承） このストリームの各データ・イベントでactionを実行する。 このストリームの総てのイベントが処理されたら返されたFutureを完了する。もしこのストリームがエラー・イベントを有していたりactionがスローしたときはこのfutureはエラーだ完了する。
Stream<T> **handleError**(void handle(AsyncError error), {bool test(error)})	（Streamから継承） このストリームからの何らかのエラーを横取りするラッパーのストリームを生成する。 もしこのラッパ・ストリームがtestにマッチするエラーを送信するときは、それはhandle関数により横取りされる。 test(e)がtrueを返すと[AsyncError] [:e:]はtest関数によりマッチがとられる。testがオミットされているときは各エラーはマッチしていると見做される。 もしそのエラーが横取りされたときは、handle関数はそれに対してどうするかを判断できる。この関数は新しい（あるいは同じ）エラーを生起させたいときはスローできるし、あるいはこのストリームにこのエラーを忘れさせる為に単に戻ることができる。 あるエラーをデータ・イベントに変換したいときは、より一般的なStream.transformEvenを使って出力sinkへのデータ・イベントを書いて、このイベントを処理させる。
Future<String> **join**([String separator=""])	（Streamから継承） データ・イベントたちの文字列表現たちの文字列を集める。 separatorが指定されているときはそれは二つのイベント間に挿入される。 このストリーム内にエラーがあればそのfutureはエラーで完了する。そうでないときは"done"イベントが到来したときに収集した文字列で完了する。
Future<T> **lastMatching**(bool test(T value), {T defaultValue()})	（Streamから継承） このストリームの中でtestにマッチする最後の要素を探す。 最後のマッチする要素を見つけることを除いてfirstMatchingとおなじ。このことはこのストリームが完了するまでこの結果は得られないことを意味する。
abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})	（Streamから継承） このストリームにひとつの受信を付加する。 このストリームからの各テータ・イベントにたいし、受信者たちのonDataハンドラが呼び出される。もしonDataがnullのときは何も起きない。 このストリームからのエラーにたいし、onErrorハンドラにはそのエラーを記述したAsyncErrorが与えられる。 このストリームがクローズしたときは、onDone ハンドラが呼び出される。

	unsubscribeOnErrorがtrueのときは、最初のエラーが報告されたときにこの受信は終了する。デフォルト値はfalseである。
Stream **map**(convert(T event))	（Streamから継承） このストリームの各要素をconvert関数を使って新しい値に変換する新しいストリームを生成する。
Future **pipe**(StreamConsumer<T, dynamic> streamConsumer)	（Streamから継承） このストリームを用意されているStreamConsumerの入力に結び付ける。
Future **reduce**(initialValue, combine(previous, T element))	（Streamから継承） combineを繰り返し適用することで値たちの並びを減らす。
Future<T> **singleMatching**(bool test(T value))	（Streamから継承） testにマッチするこのストリームのなかの最初の要素を探す。 このストリームの中でひとつ以上のマッチする要素が生じないときにエラーとなることを除いてlastMatchと似ている。
Stream<T> **skip**(int count)	（Streamから継承） このストリームからの最初のcount個数のデータ・イベントをスキップする。
Stream<T> **skipWhile**(bool test(T value))	（Streamから継承） testにマッチする間はこのストリームからのデータ・イベントをスキップする。 返されたストリームはエラーと完了のイベントを加工しないで渡す。 そのイベント・データに対しtestがtrueを返す最初のデータ・イベントから始まり、返されるこのストリームはこのストリームと同じイベントを持つことになる。
Stream<T> **take**(int count)	（Streamから継承） このストリームの最大n個の値を渡す。 このストリームの最初のn個のデータ・イベント、及び総てのエラー・イベントを返されるストリームに転送し、完了イベントで終了する。 このストリームがその完了前にcount値より少ない場合は、返されるストリームもそうする。
Stream<T> **takeWhile**(bool test(T value))	（Streamから継承） testが成功している間はデータ・イベントを転送する。 返されたストリームはそのイベントデータに対しtestがtrueを返す限りこのストリームと同じイベントを渡す。このストリームはthisストリームが完了した、あるいはthisストリームが最初にtestを受け付けない値を最初に渡したときに完了する。
Stream **timeout**(Duration timeLimit, {Function void onTimeout(EventSink sink)})	（Streamから継承） このストリームと同じイベントからなる新しいストリームを生成する。 このストリームからの2つのイベント間にtimeLimit以上が経過したときは、onTimeout関数が呼ばれる。 返されたストリームがリスンされるまではカウントダウンは開始しない。イベントがこのストリームから渡される度、あるいはこのストリームがポーズし再開される度にこのカウントダウンはリセットされる。

	onTimeout関数が一つの引数（EventSink）で呼ばれる： EventSinkによりイベントたちを返されたイベントたちのなかに置くことができる。 onTimeoutが指定されていないときはタイムアウトは返されるストリームのエラー・チャンネルの中に TimeoutExceptionとして置かれる。 このストリームが放送ストリームの時は返されるストリームも放送ストリームとなる。ある放送ストリームが一回以上リスンされているときは、リスンごとにカウントを開始タイマを個々に持ち、加入たちのタイマーたちはここにポーズされ得る。
Future<List<T>> **toList**()	（Streamから継承） このストリームのデータをListとして収集する。
Future<Set<T>> **toSet**()	（Streamから継承） このストリームのデータをSetとして収集する。
Stream **transform**(StreamTransformer<T, dynamic> streamTransformer)	（Streamから継承） このストリームを指定されたStreamTransformerの入力として連結する。 streamTransformer.bind自身の結果を返す。
Stream<T> **where**(bool test(T event))	（Streamから継承） 何らかのデータ・イベントたちを破棄する新しいストリームをこのストリームから生成する。 新しいストリームはこのストリームと同じエラーと完了のイベントを送信するが、testを満足させるデータ・イベントのみを送信する。

dart.io.HttpRequest

Abstract class **HttpRequest** HTTPサーバのコールバックに渡されるHTTP要求。HttpRequestは該要求のボディ部のコンテントのStreamである。このデータを処理するにはこのボディ部をリスンし、ボディ部の総てが受信されたときには通知される。	
実装	
Stream<List<int>>	
属性	
final X509Certificate **certificate**	この要求をしているクライアントのクライアント認証を返す。この接続がセキュアTLSまたはSSL接続で無いとき、あるいはもしこのサーバがクライアント認証を要求していないとき、あるいは該クライアントがそれを提供していないときはnullを返す。
final HttpConnectionInfo **connectionInfo**	クライアント接続に関する情報を取得する。該ソケットが取得できないときはnullを返す。
final int **contentLength**	要求ボディ部のコンテント長を返す。あらかじめその要求ボディのサイズが判っていないときは-1を返す。
final List<Cookie> **cookies**	該要求内の（cookieヘッダたちからの）クッキーたちのリストを返す

final Future<T> **first**	（Streamから継承） もしthisが空のときはStateErrorを返す。そうでないときは、このメソッドはthis.elementAt(0)と等価である。
final HttpHeaders **headers**	要求ヘッダたちを返す。
final bool **isBroadcast**	（Streamから継承） このストリームが放送ストリームかどうか。
final Future<bool> **isEmpty**	（Streamから継承） このストリームが何らかの要素を含んでいるかどうかを報告する。
final Future<T> **last**	（Streamから継承） 最後の要素を返す。 もし空のときはStateErrorをスローする。
final Future<int> **length**	（Streamから継承） inherited from Stream Counts the elements in the stream.
final String **method**	該要求のメソッドを返す。
final bool **persistentConnection**	クライアントから示されている継続接続状態を返す。
final String **protocolVersion**	該要求で使われているHTTPプロトコルのバージョンを返す。これは"1.0"または"1.1"になる。
final Map<String, String> **queryParameters**	解析されたクエリ・パラメタたちを返す。
final HttpResponse **response**	HttpResponseオブジェクトを取得する。応答をクライアントに送り返すのに使われる。object, used for sending back the response to the client.
final HttpSession **session**	与えられた要求に対するセッションを取得する。この呼び出しでセッションが初期化されているときは返されるセッションのisNewがtrueとなる。デフォルトのタイムアウトを変更する手段はHttpServer.sessionTimeoutを参照のこと。
final Future<T> **single**	（Streamから継承） 単一の要素を返す。 もしこれが空のとき、あるいはひとつ以上の要素を持っているときはStateErrorをスローする。
final Uri **uri**	該要求のURIを返す。
メソッド	
Future<bool> **any**(bool test(T element))	（Streamから継承） このストリームが用意している要素のどれかをtestが受け付けるかどうかをチェックする。 その答えが判ったときにFutureを完了させる。もしこのストリームがエラーを報告したときは、このFutureはエラーを報告する。
Stream<T> **asBroadcastStream**()	（Streamから継承） これと同じイベントたちを作り出す複数受信のストリームを返す。もしこのストリームが単一受信のときは、複数の受信者が許される新しいストリームを返す。その最初の受信者が付加されたときにこれはこのストリームに受信し、最後の受信者が

	キャンセルされたときに再度受信解除される。 もしこのストリームが既に放送ストリームであるときは、これは手を加えられないで返される。
Future<bool> **contains**(T match)	(Streamから継承) このストリームが用意した要素たちのなかでmatchが生じるかどうかをチェックする。 この答えが判ったときにこのFutureを完了させる。このストリームがエラーを報告したときは、該Futureはそのエラーを報告する。
Stream<T> **distinct**([bool equals(T previous, T next)])	(Streamから継承) もし以前のデータ・イベントと同じのときはこれらのデータ・イベントをスキップする。 返されるストリームは、ふたつの連続したデータが決して同じで無いということを除いて、このストリームと同じイベントを作る。 用意されるequalsメソッドによって等しいかどうかが判断される。もしこれがオミットされているときは、最後に用意されたデータ要素には'=='演算子が使われる。
Future<T> **elementAt(**int index)	(Streamから継承) このストリームのindex番目のデータ・イベントの値を返す。 もしエラーが起きれば、このfutureはエラーで終了する。 もしこのストリームが閉じる前にindex要素たちより少ない数しか用意していないときは、エラーが報告される。
Future<bool> **every**(bool test(T element))	(Streamから継承) このストリームが用意している総ての要素をtestが受け付けるかどうかをチェックする。 この答えが判った時点でこのFutureを完了させる。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream **expand**(Iterable convert(T value))	(Streamから継承) 各要素をゼロまたはそれ以上のイベントたちに変換する新しいストリームをこのストリームから生成する。 各到来イベントは新しいイベントたちのIterableに変換され、これらの新しいイベントたちの各々が次に順番に返されたストリームによって送信される。
Future<T> **firstMatching**(bool test(T value), {T defaultValue()})	(Streamから継承) testにマッチするこのストリームの最初の要素を見つける。 testがtrueを返すこのストリームの最初の要素で満たされたfutureを返す。 このストリームが完了する前にそのような要素が見つからず、またdefaultValue関数が用意されているときは、defaultValue呼び出しの結果がそのfutureの値となる。 エラーが発生したとき、あるいはこのストリームが一致する要素が見つからないで終了し、またdefaultValue関数が用意されていないときは、このfutureはエラーを受信する。

Stream<T> **handleError**(void handle(AsyncError error), {bool test(error)})	（Streamから継承） このストリームからの何らかのエラーを横取りするラッパーのストリームを生成する。 もしこのラッパ・ストリームがtestにマッチするエラーを送信するときは、それはhandle関数により横取りされる。 test(e)がtrueを返すと[AsyncError] [:e:]はtest関数によりマッチがとられる。testがオミットされているときは各エラーはマッチしていると見做される。 もしそのエラーが横取りされたときは、handle関数はそれに対してどうするかを判断できる。この関数は新しい（あるいは同じ）エラーを生起させたいときはスローできるし、あるいはこのストリームにこのエラーを忘れさせる為に単に戻ることができる。 あるエラーをデータ・イベントに変換したいときは、より一般的なStream.transformEvenを使って出力sinkへのデータ・イベントを書いて、このイベントを処理させる。
Future<T> **lastMatching**(bool test(T value), {T defaultValue()})	（Streamから継承） このストリームの中でtestにマッチする最後の要素を探す。 最後のマッチする要素を見つけることを除いてfirstMatchingとおなじ。このことはこのストリームが完了するまでこの結果は得られないことを意味する。
abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})	（Streamから継承） このストリームにひとつの受信を付加する。 このストリームからの各テータ・イベントにたいし、受信者たちのonDataハンドラが呼び出される。もしonDataがnullのときは何も起きない。 このストリームからのエラーにたいし、onErrorハンドラにはそのエラーを記述したAsyncErrorが与えられる。 このストリームがクローズしたときは、onDone ハンドラが呼び出される。 unsubscribeOnErrorがtrueのときは、最初のエラーが報告されたときにこの受信は終了する。デフォルト値はfalseである。
Stream **map**(convert(T event))	（Streamから継承） このストリームの各要素をconvert関数を使って新しい値に変換する新しいストリームを生成する。
Future<T> **max**([int compare(T a, T b)])	（Streamから継承） このストリームのなかの最大の要素を探す。 もしこのストリームが空のときはその結果はnullである。そうでないときは、その結果はこのストリームからの他のどの値よりも小さくないこのストリームからの値となる（Comparatorでなければならないcompareに従って）。 もしcompareがオミットされているときは、そのデフォルトはComparable.compareである。
Future<T> **min**([int compare(T a, T b)])	（Streamから継承） このストリームのなかの最小の要素を探す。

	もしこのストリームが空のときはその結果はnullである。そうでないときは、その結果はこのストリームからの他のどの値よりも大さくないこのストリームからの値となる（Comparatorでなければならないcompareに従って）。 もしcompareがオミットされているときは、そのデフォルトはComparable.compareである。
Future **pipe**(StreamConsumer<T, dynamic> streamConsumer)	（Streamから継承） このストリームを用意されているStreamConsumerの入力に結び付ける。
Future **pipeInto**(StreamSink<T> sink, {void onError(AsyncError error), bool unsubscribeOnError})	（Streamから継承）
Future **reduce**(initialValue, combine(previous, T element))	（Streamから継承） 繰り返しcombineを適用することで値たちの並びを減らす。
Future<T> **singleMatching**(bool test(T value))	（Streamから継承） testにマッチするこのストリームのなかの最初の要素を探す。 このストリームの中でひとつ以上のマッチする要素が生じないときにエラーとなることを除いてlastMatchと似ている。
Stream<T> **skip**(int count)	（Streamから継承） このストリームからの最初のcount個数のデータ・イベントをスキップする。
Stream<T> **skipWhile**(bool test(T value))	（Streamから継承） testにマッチする間はこのストリームからのデータ・イベントをスキップする。 返されたストリームはエラーと完了のイベントを加工しないで渡す。 そのイベント・データに対しtestがtrueを返す最初のデータ・イベントから始まり、返されるこのストリームはこのストリームと同じイベントを持つことになる。
Stream<T> **take**(int count)	（Streamから継承） このストリームの最大n個の値を渡す。 このストリームの最初のn個のデータ・イベント、及び総てのエラー・イベントを返されるストリームに転送し、完了イベントで終了する。 このストリームがその完了前にcount値より少ない場合は、返されるストリームもそうする。
Stream<T> **takeWhile**(bool test(T value))	（Streamから継承） testが成功している間はデータ・イベントを転送する。 返されたストリームはそのイベントデータに対しtestがtrueを返す限りこのストリームと同じイベントを渡す。このストリームはthisストリームが完了した、あるいはthisストリームが最初にtestを受け付けない値を最初に渡したときに完了する。
Future<List<T>> **toList**()	（Streamから継承） このストリームのデータをListとして収集する。
Future<Set<T>> **toSet**()	（Streamから継承）

	このストリームのデータをSetとして収集する。
CodeStream **transform**(StreamTransformer<T, dynamic> streamTransformer)	（Streamから継承） このストリームを指定されたStreamTransformerの入力として連結する。 streamTransformer.bind自身の結果を返す。
Stream<T> **where**(bool test(T event))	（Streamから継承） 何らかのデータ・イベントたちを破棄する新しいストリームをこのストリームから生成する。 新しいストリームはこのストリームと同じエラーと完了のイベントを送信するが、testを満足させるデータ・イベントのみを送信する。

dart.io.HttpResponse

注意2013年3月の変更でStringSinkを実装したIOSinkを実装することになった。

abstract class **HttpResponse**

そのクライアントに返送されるHTTP応答。

このオブジェクトは該応答のHTTPヘッダ設定の為の一連の属性を持っている。該ヘッダが設定されたら、該HTTP応答の実際のボディ部に書き込むのにIOSinkからのメソッドたちが使えるようになる。このIOSinkのメソッドのどれかが初めて使われると、応答ヘッダ部はネットワークに送信される。それが送信された後でのヘッダ部を変更するメソッド呼び出しには例外がスローされる。

IOSinkを介して文字列データを書き込む際は、エンコーディングは"Content-Type"ヘッダの"charset"パラメタによって決まる。

```
HttpResponse response = ... response.headers.contentType

  = new ContentType("application", "json", charset: "utf-8");
response.write(...); // 書き込まれる文字列はUTF-8でエンコードされる
```

charsetが設定されていないときは、ISO-8859-1 (Latin 1)が使用される。

```
HttpResponse response = ... response.headers.add(HttpHeaders.CONTENT_TYPE,
"text/plain"); response.write(...); // 文字列はISO-8859-1でエンコードされる
```

対応していないエンコーディングが指定されているときは、文字列をとるwriteメソッドたちのひとつが使われていると例外がスローされる。

実装	
IOSink<HttpResponse>	
属性	
final HttpConnectionInfo **connectionInfo**	クライアント接続に関する情報を取得する。ソケットが取得できないときはnullを返す。
int **contentLength**	該応答のコンテント長を取得及び設定する。あらかじめ該応答のサイズが判って

	いないときは、デフォルト値でもある-1がセットされる。
final List<Cookie> **cookies**	このクライアント内で（Set-Cookieヘッダ内で）セットされたクッキーたち。
final Future<T> **done**	（IOSinkから継承） 総てのデータがIOSinkに書き込まれ、それがクローズしたときに完了するfutureを返す。
Encoding **encoding**	（IOSinkから継承）
final HttpHeaders **headers**	応答ヘッダたちを返す。
bool **persistentConnection**	継続接続状態の取得及び設定を行う。この属性の初期値は該要求からの継続接続状態である。
String **reasonPhrase**	理由句の取得と設定を行う。理由句を明示的に設定しないときはデフォルトの理由句が用意される。
int **statusCode**	ステータス・コードの取得と設定を行う。どの整数値も受け付けるが、正式なHTTPステータス・コードとする為には、HttpStatusのフィールドを使用すること。ステータス・コードが明示的にセットされていないときは、デフォルト値のHttpStatus.OK が使用される。
メソッド	
void **close**()	（IOSinkから継承） このターゲットをクローズする。
Future<T> **consume**(Stream<List<int>> stream)	（IOSinkから継承） このIOSinkにパイプする為の機能を持つ。
abstract Future<Socket> **detachSocket**()	下位層のソケットをこのHTTPサーバから切り離す。ソケットが切り離されると、そのHTTPサーバはそのソケット上での操作をしなくなる。これは通常HTTPアップグレード要求を受信し、該通信を異なったプロトコルで継続しなければならないときに使用される。
abstract void **write**(Object obj)	（StringSinkから継承） objをtoStringを呼ぶことでStringに変換しその結果をこれに追加する。 Converts obj to a String by invoking toString and adds the result to this.
abstract void **writeAll**(Iterable objects)	（StringSinkから継承） 与えられたobjectsで繰り返し操作を行いそれらを順番に書き込む。
abstract void **writeBytes**(List<int> data)	（IOSinkから継承） バイト列を変換することなくこのコンシューマに書き込む。
abstract void **writeCharCode**(int charCode)	（StringSinkから継承） charCodeをこれに書き込む。 このメソッドは write(new String.fromCharCode(charCode)) と等価である。
abstract void **writeln**([Object obj = ""])	（StringSinkから継承） objをStringに変換しその結果をこれに追加する。次に改行を追加する。
abstract Future<T> **writeStream**(Stream<List<int>> stream)	（IOSinkから継承） consumeと同じだが、終了したときにこのターゲットをクローズしない。

dart.io.HttpSession

abstract class **HttpSession**	
実装	
Map	
属性	
final String **id**	現行セッションのIDを取得する。
final bool **isEmpty**	（Mapから継承） このMapに{key, value}ペアが存在しないときにtrueを返す。
final bool **isNew**	このセッションがまだ該クライアントに送信されていないときにtrueとなる。
final Iterable<K> **keys**	（Mapから継承） thisのキーたち。
final int **length**	（Mapから継承） このMap内の{key, value}ペアの数。
abstract void set **onTimeout**(void callback())	このセッションがタイムアウトしたときに呼ばれるコールバックを設定する。
final Iterable<V> **values**	（Mapから継承） thisの値たち。
演算子	
abstract V operator **[]**(K key)	（Mapから継承） 与えられたキーに対する値を返すか、キーがこのマップに存在しないときにnullを返す。nullという値に対応しているので、キーがないこととnullの値を区別する為にcontainsKeyを使うか、あるいはputIfAbsentメソッドを使う。
abstract void operator **[]=**(K key, V value)	（Mapから継承） 該キーに指定された値を結び付ける。
メソッド	
void **destroy**()	このセッションを廃棄する。これによりこのセッションは終了し、このIDを持ったその後の接続に対しては新しいセッションとIDが付与される。
abstract void **clear**()	（Mapから継承） このMapから総てのペアを削除する。
abstract bool **containsKey**(K key)	（Mapから継承） このマップが指定したキーを含んでいるかどうかを返す。
abstract bool **containsValue**(V value)	（Mapから継承） このマップが指定した値を含んでいるかどうかを返す。
abstract void **destroy**()	このセッションを破棄する。これは該セッションを終了させ、このidをもった以降の接続には新しいセッションとidが付与される。
abstract void **forEach**(void f(K key, V value))	（Mapから継承） このマップの各{key, value}ペアにfを適用する。

	この繰り返し操作中にキーを付加または削除するのはエラーである。
abstract V **putIfAbsent**(K key, V ifAbsent())	(Mapから継承) keyがある値に結び付けられていないときはifAbsentを呼びkeyをifAbsentで返された値にマッピングすることでこのマップを更新する。 ifAbsentの呼び出し中にキーを付加または削除するのはエラーである。
abstract V **remove**(K key)	(Mapから継承) 与えられたkeyへの関連付けを外す。このマップの中のkeyにたいする値を返すか、keyが存在しないときにはnullを返す。値はnullであることが許されるので、nullが返されたということはこのkeyが存在していないということを意味しないことに注意。

dart.io.HttpHeaders

abstract class **HttpHeaders**

HTTP要求および応答のヘッダたちを表現。

ある場合にはヘッダたちは不変(immutable:設定付加)となる

- HttpRequestとHttpClientResponseは常に不変なヘッダたちを持つ
- そのボディ部のデータが書き込まれた以降はHttpResponseとHttpClientRequestのヘッダたちは不変となる

これらの場合には可変のメソッドっ呼び出しは例外をスローする。

HTTPヘッダたちの総ての操作においてヘッダ名は大文字と小文字は区別されない。

ヘッダたちの値をセットするにはset()メソッドを使用する:

```
request.headers.set(HttpHeaders.CACHE_CONTROL,
                    'max-age=3600, must-revalidate');
```

ヘッダたちの値を取得するにはvalue()メソッドを使用する:

```
print(request.headers.value(HttpHeaders.USER_AGENT));
```

標準が認めているようにHttpHeadersのオブジェクトは各名前に対し値たちのリストを保持する。ほとんどの場合はある名前は単一の値を保持する。もっとも一般的な操作は値の設定にはset()を、値の取得にはvalue()を使うことである。

継承	
Object	
属性	
bool **chunkedTransferEncoding**	チャンクド転送エンコーディングのヘッダ値の取得と設定
int **contentLength**	コンテント長ヘッダ値の取得と設定

ContentType **contentType**	コンテント・タイプの取得と設定。このフィールドが直接セットされるときに限りこのヘッダ内のコンテント・タイプが更新されることに注意。返された現行値を可変化しても効果を持たない。
DateTime **date**	日付の取得と設定。この属性の値は'date'ヘッダを反映する。
DateTime **expires**	有効期限の取得と設定。この属性の値は'expires'ヘッダを反映する。
String **host**	該接続の'host' ヘッダのホスト部の取得と設定。
DateTime **ifModifiedSince**	"if-modified-since"時刻の取得と設定。この属性の値は"if-modified-since"ヘッダを反映する。
bool **persistentConnection**	永続接続ヘッダの値の取得と設定。
int port	該接続の'host'ヘッダのポート部の取得と設定。
satic属性	
static const	ACCEPT ACCEPT_CHARSET ACCEPT_ENCODING ACCEPT_LANGUAGE ACCEPT_RANGES AGE ALLOW AUTHORIZATION CACHE_CONTROL CONNECTION CONTENT_ENCODING CONTENT_LANGUAGE CONTENT_LENGTH CONTENT_LOCATION CONTENT_MD5 CONTENT_RANGE CONTENT_TYPE COOKIE DATE ENTITY_HEADERS ETAG EXPECT EXPIRES FROM GENERAL_HEADERS HOST IF_MATCH IF_MODIFIED_SINCE IF_NONE_MATCH IF_RANGE IF_UNMODIFIED_SINCE LAST_MODIFIED LOCATION MAX_FORWARDS PRAGMA PROXY_AUTHENTICATE PROXY_AUTHORIZATION RANGE REFERER REQUEST_HEADERS RESPONSE_HEADERS

	RETRY_AFTER SERVER SET_COOKIE TE TRAILER TRANSFER_ENCODING UPGRADE USER_AGENT VARY VIA WARNING WWW_AUTHENTICATE
演算子	
List<String> **[]**(String name)	nameという名前のヘッダ値のリストを返す。指定した名前のヘッダが存在しないときはnullが返される。
メソッド	
void **add**(String name, Object value)	ヘッダ値を付加する。nameという名前のヘッダがその値たちのリストにvalueという値が付加される。あるヘッダは単一の値を持つものであり、それらの場合はvalueを付加するとこれまでの値に置き換わる。valueが DateTime型の場合は、HTTP日付のフォーマットが適用される。valueがList型の時は、そのリストの各要素が別々に付加される。その他の総ての型に対しては、デフォルトのtoStringメソッドが使用される。
void **clear**()	総てのヘッダを削除する。一部のヘッダはシステムが与えた値を持っており、これらの値は該ヘッダの値たちのコレクションに付加されたままとなる。
void **forEach**(Function void f(String name, List<String> values))	ヘッダたちを列挙化し、各ヘッダに関数fを適用する。nameとして渡されたヘッダ名はすべて小文字となる。
void **noFolding**(String name)	HTTPヘッダたちを送信する際に、nameという名前のヘッダをカンマで区切ってフォルドしないようにする。デフォルトでは複数の値があった場合はカンマで各値を区切って単一のヘッダ行としてフォルドされる。但し'set-cookie'だけはデフォルトでフォルドしない。
void **remove**(String name, Object value)	nameという名前の特定の値を削除する。一部のヘッダはシステムが与えた値を持っており、これらの値は該ヘッダの値たちのコレクションに付加されたままとなる。
void **removeAll**(String name)	nameという名前の総ての値を削除する。一部のヘッダはシステムが与えた値を持っており、これらの値は該ヘッダの値たちのコレクションに付加されたままとなる。
void **set**(String name, Object value)	ヘッダをセットする。nameという名前のヘッダはこれまでの総ての値がクリアされ、valueが付加される。
String **value**(String name)	値を一つしか持たないヘッダの値に対する便利なメソッド。指定したnameのヘッダがないときはnullが返される。該ヘッダが1以上の値を持つときは例外がスローされる。

dart.core.Uri

（注意：2013年5月末にdart:uriはdart:coreに移された）

class **Uri** RFC-3986, http://tools.ietf.org/html/rfc3986の規定に基づいて構文解析されたURI。	
staticメソッド	
Uri **parse**(String uri)	指定したURI文字列を構文解析して新しいURIオブジェクトを生成する。
String **encodeComponent**(String component)	文字列のcomponentを%エンコーディングを使用してエンコードし、URI要素として使うリテラルとして安全なものにする。 大文字と小文字の英文字、数文字、及び!$&'()*+,;=:@ を除いたすべての文字が%エンコードされる。これは RFC 2396で規定され、またECMA-262 version 5.1で encodeUriComponentのために規定された文字たちのセットである。 パス要素またはクエリ要素をマニュアルでエンコードするときは、パスまたはクエリ文字列を組み立てる前に各要素を別々にエンコードすることを忘れないこと。 クエリ部分をエンコードするにはencodeQueryComponentの使用を検討する。 明示的なエンコーディングの必要性を回避するには、Uriのコンストラクトのなかでオプショナルな指名引数である pathSegmentsと queryParametersを使用する。
String **encodeQueryComponent**(String component)	クエリ文字列要素としてHTMLのformのポストをエンコードする為に、HTML 4.01規則に従って文字列のcomponentをエンコードする。 スペースは'+'文字によって置き換えられ、大文字と小文字の英文字、数文字、及び-._~を除く総ての文字は%エンコードされる。このエンコードされる文字セットはHTML 4.01がRFC 1738で予約文字としているからとしているのでスーパーセットである。 マニュアルでクエリ要素をエンコードするときは、パスまたはクエリ文字列を組み立てる前に各要素を別々にエンコードすることを忘れないこと。 明示的なエンコーディングの必要性を回避するには、Uriのコンストラクトのなかでオプショナルな指名引数である pathSegmentsと queryParametersを使用する。 更なる詳細は http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2を参照のこと。
String **decodeComponent**(String encodedComponent)	encodedComponentのなかの%エンコーディングをデコードする。 あるURI要素をデコードすると、一部のデコードされた文字が与えられたURI要素の型ではデリミタになってしまうことがあるのでその意味を変えてしまうことに注意。デリミタたちを要素として使うときは、個々の部分をデコードする前にURI要素を必ずスプリットしておくこと。 パスとクエリ要素を取り扱うときは、分離されデコードされた要素を取得する為にpathSegments及び queryParametersを使用すること。
String **decodeQueryComponent**(String encodedComponent)	
String **encodeFull**(String uri)	文字列のuriを%エンコーディングを使用してエンコードし、URI全体として使うリテラルとして安全なものにする。

	大文字と小文字の英文字、数文字、及び!$&'()*+,;=:@_~ を除いたすべての文字が%エンコードされる。これはECMA-262 version 5.1で encodeURI関数のために規定された文字たちのセットである。
String **decodeFull**(String uri)	uriのなかの%エンコーディングをデコードする。 デコードされた文字の幾つかが予約語である可能性があるのでURI全部のデコードではその意味を変えてしまう可能性があることに注意。殆どの場合エンコードされたURIは分離した要素をデコードする前に Uri.parseを使ってエンコードされたURIを構文解析すべきである。
Map<String, String> **splitQueryString**(String query)	HTML 4.01規則17.13.4.節のFORMポストで規定された規則に従ってqueryを分割しMapとして返す。もしqueryが空の文字列のときは、空のMapが返される。 query 文字列のなかの値を持たないキーたちは空の文字列としてマップされる。
コンストラクタ	
new **Uri**({scheme, String userInfo: "", String host: "", port: 0, String path, List<String> pathSegments, String query, Map<String, String> queryParameters, fragment: ""})	その要素たちから新しいURIを生成する。 各要素は指名引数を介してセットされる。任意の数の要素を指定できる。指定されない要素のデフォルト値は空の文字列だが、portだけは別でそのデフォルト値は0である。pathとqueri要素はふたつの別々の指名引数でセットできる。 スキーム要素はschemeを介してセットする。この schemeは総て小文字で正規化される。 権限要素のユーザ情報部分はuserInfoを介してセットされる。 権限要素のホスト部はhostを介してセットされる。このhostは" と "に含められたhostname、IPv4アドレス、またはIPv6アドレスのどれかになる。もしこのhostが':' 文字を含んでいるときは、既に与えられていなければ" と " が付加される。 権限要素のポート部はportを介してセットされる。ポートの80あるいは443がセットされるとそれはhttpあるいはhttpsに正規化される。 パス要素は pathまたはpathSegmentsのどちらかでセットされる。pathが使われているときは、指定された文字列は総て%エンコードであると見做され、そのリテラル様式が使われる。pathSegmentsが使われているときは、指定されている各セグメントが%エンコードされてると見做され、スラッシュのセパレータを使って結合される。 pathセグメントの%エンコーディングでは、非予約文字と!$&'()*+,;=:@を除いた総ての文字が%エンコードされる。もし他の要素が絶対パスを指定しているときは既に無いときは頭に/が付される。 クエリ要素は queryまたはqueryParametersを介してセットされる。queryが使われているときは、与えられた文字列は総て%エンコードされているとみなされ、そのリテラル様式のなかで使われる。 queryParametersが使われているときは、そのクエリは指定したmapから組み立てられる。そのマップの各キーと値は%エンコードされ=と&文字で結合される。キーと値の%エンコーディングは非予約文字を除いた総ての文字をエンコードする。

	フラグメント要素はfragmentを介してセットされる。
属性	
final String **authority**	権限要素を返す。 権限はuserInfo、host、及びport部からフォーマットされる。 権限要素が無い場合は空の文字列を返す。
final String **fragment**	フラグメント識別子部分を返す。 フラグメント識別子部分が無いときは空の文字列を返す。
final bool **hasAuthority**	このURIが権限部分を持っているかどうかを返す。
final int **hashCode**	このオブジェクトのハッシュ・コードを返す。 総てのオブジェクトがハッシュ・コードを持っている。対等演算子==を使って比較したときに等しいオブジェクトたちはハッシュ・コードが同じであることを保証される。その例外は、そのハッシュ・コードに関する保証は無い。runたち間には一貫性がなく、配布に際しても保証されない。 あるサブクラスが hashCodeするときは、そのサブクラス一貫性を維持するとともに対等演算子をオーバライドする。
final String **host**	権限要素のホスト部を返す。 権限要素がなく従ってホスト部が無いときは空の文字列を返す。
final bool **isAbsolute**	そのURIが絶対URIかどうかを返す。
final String **origin**	http及びhttpsのスキームの為の scheme://host:portの形式ののなかのURIのオリジンを返す。 スキームが"http"または"https"でないときはエラーである。 参照：http://www.w3.org/TR/2011/WD-html5-20110405/origin-0.html#origin
final String **path**	パス部を返す。 戻されるパスはエンコードされている。デコードされたパスに直接アクセスするときはpathSegmentsを使用のこと。 パス要素が無いときは空の文字列を返す。
final List<String> **pathSegments**	そのセグメントに分割されたURIパスを返す。戻されるリストのなかの各セグメントはデコードされている。もしそのパスが空のときは空のリストが返される。頭のスラッシュ/は戻されるセグメントに影響を与えない。 戻されるリストは修正不可でそれを変えるような呼び出しには UnsupportedErrorがスローされる。
final int **port**	権限部のポート部を返す。 権限部のなかにポートが無い場合は0を返す。

final String **query**	クエリ要素を返す。返されるクエリはエンコードされている。デコードされたクエリに直接アクセスするときは queryParametersを使用のこと。
final Map<String, String> **queryParameters**	HTML 4.01仕様書の17.13.4. 節のFORMポストで規定されている規則に従ってマップに分割されたURIクエリを返す。返されるマップのなかのキーと値はデコードされている。クエリがないときは空のマップが返される。 値を持たないクエリ文字列のキーは空の文字列にマップされる。 戻されるマップは修正不可でそれを変えるような呼び出しには UnsupportedErrorがスローされる。
final String **scheme**	スキーム部を返す。 スキームが無い場合は空の文字列を返す。
final String **userInfo**	権限要素のinfo部を返す。 権限要素のなかにユーザ情報が無いときは空の文字列を返す。
演算子	
bool operator ==(other)	対等性演算子。 総てのObjectにたいする振る舞いはthisとotherが同じオブジェクトであるときに限りtrueを返す。 あるサブクラスがこの演算子をオーバライドするときは、一貫性を保つためにはhashCodeもオーバライドしなければならない。
メソッド	
Uri **resolve**(String uri)	
Uri **resolveUri**(Uri reference)	
String **toString**()	このオブジェクトのStringによる表現を返す。

第20章 HTTPSサーバ (HTTPS Servers)

2013年2月にHttpServerインターフェイスにbindSecureというクラス・メソッドが追加され、HTTPSサーバの開発が可能となった。

2015年9月にDartチームは 1.13.0-dev.1.0版からDart SDK及びDart VMがTLS/SSLとしてBoringSSLを実装したと発表した（これまでのdart:ioの暗号化ライブラリではApacheなどで使われているOpenSSLではなくてNetwork Security Services (NSS)を採用していた）。BoringSSLはOpenSSLの派生物でGoogleが作成・管理している。従って今後はDartの開発者たちはChromeと同じSSL実装を使用することになる。

BoringSSLはOpenSSLに比べてコンパクトであり、よりきちんとまた積極的に更新・維持されている。PEMファイル（通常認証機関から署名つき証明書としてメールで送られてくる標準的なテキスト形式）を使って認証とキーの管理がなされ、NSSで使われているセキュリティ・データベースに比べてより理解し易いものとなっている。

上記の変更に伴い、第33版から本章は大幅に改版されている。

20.1節　PKIの基礎

TLS/SSLの基となっている技術はPKI（PPublic Key Infrastructure、「公開鍵基盤」と訳される）である。従ってまずPKIの基礎を略説する。詳細は日本語の資料が豊富に存在する（例えば情報処理推進機構のPKIに関する資料）ので、それを見ていただきたい。

PKIは次の2つの要素で構成される：

1. 2つの鍵のペアを使う暗号化技術。これは一方の鍵で暗号化したメッセージはもう一方の鍵でのみ複合できる特殊な暗号化技術で、一方の鍵を公開鍵として相手に渡し、他方の鍵を秘密鍵として自分が持つことで、自分のみが相手が送信したメッセージを受け取ることが可能となり、盗聴を防止する。この暗号化方式は送り手と受け手が同じ鍵を持つ「共通鍵方式」と対比して「公開鍵方式」（あるいはより正確には「非対象アルゴリズム暗号方式」）と呼ばれる。

2. 電子署名のメカニズム。相手（例えばサーバ）が本当に自分がメッセージを送ろうとしている正しい相手なのかどうか（即ち渡された公開鍵が正しい持ち主のものなのか）を確認するもので、上記の暗号化技術を応用している。これによりなりすましや改ざん等の不正に対処する。確認のためには認証局(CA)と呼ばれる機関が発行し署名した証明書（電子署名）が使われる。サーバは認証局が発行した証明書（秘密鍵で暗号化した）と公開鍵をクライアントに渡し、クライアントは次のように相手を確認する：
    1. 相手の公開鍵を入手する
    2. その公開鍵で送付された電子署名を復号化する
    3. 送付された平文から、相手と同じアルゴリズムを用いてハッシュを作成する
    4. 2の結果と3で作成したハッシュを比較し、一致することを確認する

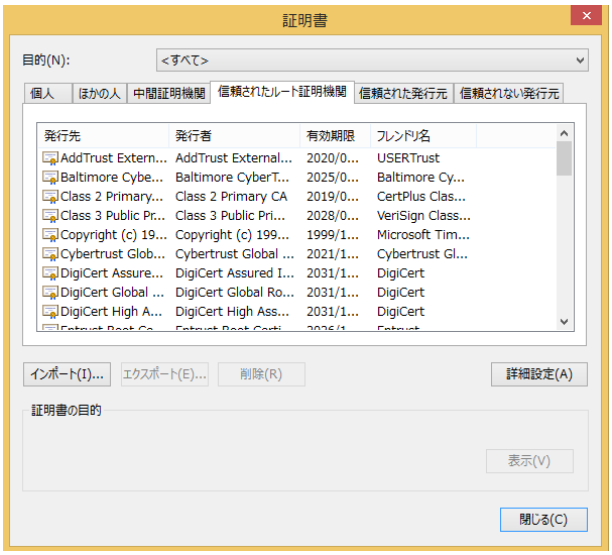
電子証明書の中身はざっと以下のようなものからなる：

1. 登録された公開鍵
2. その公開鍵の持ち主の情報

3. 証明書を発行した認証局の情報
4. 発行元認証局の署名

電子証明書の詳細なフォーマットについては、ITU-Tが定めたX.509という書式が使われる。

認証局は数多く存在しており、システム設計者はそのなかから選択できる。トップ階層にある認証局を「ルート（Root）認証局」という。例えばIEでは「ツール」メニュー→「インターネットオプション」→「コンテンツ」の証明書ボタン」を押すと、現在ブラウザで使用できる証明書が表示される。この中の「信頼されたルート証明機関」というタブで表示される証明書の認証局はすべてルート認証局であり、あらかじめブラウザに「信頼する認証局」として登録されていることを示している。表示された証明機関のどれかをダブルクリックすれば、その詳細を知ることができる。



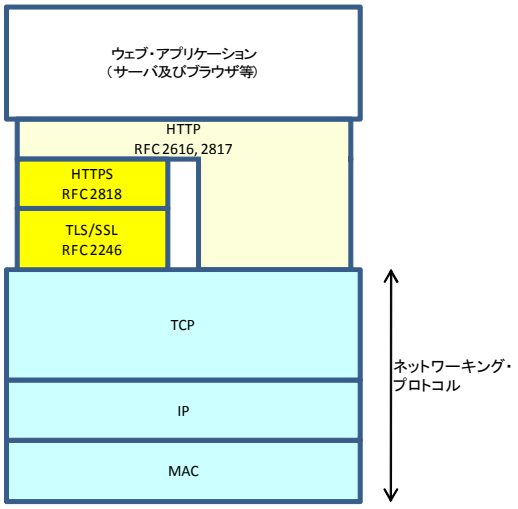
IEで登録されている証明書の例

Chromeの場合は、「ツールボタン」→「設定」→「詳細設定を表示」→「HTTPS/SSLの証明書の管理ボタン」→「信頼されたルート証明機関のタブ」で同じような表示が得られる。

20.2節　　TLS/SSLの基礎

TLS (Transport Layer Security)は上記のPKIをHTTPやSocket通信に適用させるものである。TLSはその名のとおりトランスポート層を安全なものとする為の仕掛けであり、ネットワーキングのプロトコル下位層のTCP層の上に介在させるプロトコルである。基になった技術がSSL (Secure Sockets Layer)であるため、むしろSSLという言葉が良く使われる。詳細は別途説明するが、とりあえずはWikipediaなどを参照されたい。

下図はそのプロトコルの位置づけである：



プロトコル構成

TLSはRFC2246としてTCPの上に置かれ、またHTTPアプリケーションとの仲介にHTTPプロトコルが存在するが、そのHTTPそのものもTLS対応の為にRFC 2817でアップグレードされている。TLSは接続型のプロトコルで、クライアントとサーバ間で安全（セキュア）な接続を確立し、その接続上でデータ（HTTPメッセージ）を交換する。TLS対応のブラウザはTLSクライアントになり、サーバのしかるべきTCPポート（デフォルトは443、Tomcatは8443）上への接続を開始し、TLSハンドシェイクを始めるために、TLS ClientHelloメッセージを送る。TLSハンドシェイクが完了すれば、接続が確立されたことになり、その接続上でクライアントは最初のHTTP要求をサーバに送信できるようになる。

クライアントとサーバ間で安全（セキュア）な接続を確立ためのTLSの主要な要素は、前節で示したように暗号化と認証である。認証に際してはサーバはサーバ証明書(Certificate)、あるいは加えてサーバ側が信頼する認証局のリストをブラウザに送信してそのサーバが合法であることを提示する。一般のTLS対応のアプリケーションでは、お互いが正当かどうかに関しては、サーバは認証局(CA)が発行する認証局の認証済みサーバ証明書で確認されるが、クライアントは認証局が出す公開鍵を使うだけである。このままではサーバから見たクライアントは、信頼できる相手と確信できない。そのためアプリケーションのレベルでこれまでの認証と同じようにユーザ名とパスワードによる認証を使うことが出来る。しかし、TLSではその為のクライアント認証が用意されている。

「クライアント認証」(CLIENT-CERT認証)では、更に安全性を高める為に、サーバがクライアントに対し自分が誰であるかを証明させる。サーバにSSLで接続する際、クライアントに証明書を提示させ、接続元を認証する。通常、サーバーが信用する認証局が署名した電子証明書をクライアントが提示してはじめてSSLの接続が成立する。TLS対応のサーバの殆どはこのクライアント認証をクライアントに要求していない。クライアント認証に関しては日本ベリサイン社の担当者が投稿した記事が判り易く説明されており、参考になろう。

クライアント認証は通常電子商取引や企業内網で使用されている。

TLSの基本シーケンス

TLSのシーケンスに関しては、日経NETWORKの半沢智氏の記事が判り易く書かれているので、それを読んでいただいたほうが良かろう。

TLSの基本シーケンスは次のようである:

1. ネットワーク上のノードのサーバとクライアントが証明書を交換することで互いを特定し確認する。証明書はVeriSignのような認証局によって発行される。サーバのID、サーバの公開鍵、及び認証局の署名などを含んだこの証明書は、認証局が持っている秘密鍵によって暗号化されている。そうするとクライアントは認証局の公開鍵を使ってこの証明書を復号し、サーバの公開鍵を取得できる。公開鍵暗号方式では，一方の鍵で暗号化したデータは，ペアとなっているもう一方の鍵でしか復号できない（これを非対象アルゴリズム暗号方式という）。公開鍵で暗号データを正しく復号できた場合，その暗号データはペアとなっている秘密鍵を使って暗号化されたことが保証される。秘密鍵は，所有者だけが持っているはずなので，これによってその証明書は認証局が発行した証明書であることがわかる。これで、クライアントは証明書から取得したサーバの公開鍵が信用できるものであることが判るので、これを使ってサーバのメッセージを復号化出来る。同じようにサーバはクライアントの公開鍵を取得できる。ただウェブ・アプリケーションの場合は、クライアント（ブラウザ）がサーバを信頼できるものであるかどうかを判断するだけの方法が一般的である。従って、以下そのような手順を説明することとする。

2. クライアントはサーバに対して取得したサーバの公開鍵で暗号化されたランダムなデータ（プレマスタ・シークレット）を送る。クライアントは自分が生成した該プレマスタ・シークレットをもとに共通鍵を生成する。サーバは受けた暗号化されたデータを自分の秘密鍵を使って復号する。サーバは復号化されたランダム・データをもとにクライアントと同じ共通鍵を生成する。これでクライアントとサーバがともに同じ鍵（共通鍵）を持ったことになる。

3. クライアントとサーバは電子的にメッセージ交換時の暗号化と復号に使うアルゴリズムの折衝を行う。折衝されたアルゴリズムはクライアントとサーバの双方が受け付けられるものでなければならない。

4. 共通鍵を使って双方がデータの交換を行う。即ち、サーバはサーバ鍵を使って暗号化したデータをクライアントに送信し、クライアントはそれをサーバ鍵を使って復号する。クライアントはクライアント鍵を使って自分が送りたいデータを暗号化しそれをサーバに送り、サーバをそれを受けたらクライアント鍵を使って復号化する。

実際のネットワーク上のTCPレベルのシーケンスはRFC 2246によれば以下のようになっている:

表15-4：TLSのシーケンス

<table>
<tr><td>SSLクライアント
（ブラウザ）</td><td>SSLサーバ</td><td></td></tr>
<tr><td colspan="3">ユーザがhttps://で始まるアドレスをクリック</td></tr>
<tr><td>SYN
⟶
TCP_Port = 443</td><td colspan="2" rowspan="3">このセッションには安全化された（secure)接続が必要である。クライアントはHTTPS TCPポート番号443でTCP接続を確立する</td></tr>
<tr><td>SYN+ACK
⟵</td></tr>
<tr><td>ACK
⟶</td></tr>
</table>

新しいTCP接続上でのSSLハンドシェイク	
HELLO_REQUEST ⟵	サーバは何時でもこのメッセージを送信し、ハンドシェイクの手順を最初からやり直すことをクライアントに要求できる
CLIENT_HELLO ⟶ Highest SSL Version, Ciphers Supported, Data Compression Methods, SessionId = 0, Random Data	クライアントはハンドシェイクの最初にまずCLIENT_HELLOメッセージを送信するが、これには以下のものが含まれる: • このクライアントが対応している最も新しいSSLとTLSのバージョン • このクライアントが対応している暗号化アルゴリズムたち。これは好ましいもの順でリスト化されている • このクライアントが対応しているデータ圧縮法たち • このセッションのID。もしそのクライアントが新しいセッションを開始するときはこのセッションIDは0 • ランダム・データはクライアントが生成し、これはキー生成プロセスで使用される
SERVER_HELLO ⟵ Selected SSL Version, Selected Cipher, Selected Data Compression Method, Assigned Session Id, Random Data	サーバはCLIENT_HELLOメッセージを受けてサーバはこのSSLセッションで使用する暗号方式などを決定し、SERVER_HELLOメッセージをクライアントに送信するが、これには以下のものが含まれる: • このSSLセッションの為に使われるSSLまたはTLSのバージョン • このSSLセッションで使う暗号化アルゴリズム • このSSLセッションで使用するデータ圧縮 • このSSLセッションの為のセッションID • サーバが生成したランダム・データで、キー生成プロセスで使われる
SERVER_CERTIFICATE ⟵ Public Key, Authentication Signature	サーバはサーバ認証の為のSERVER_CERTIFICATEメッセージを送信する。このコマンドは次のものを含む: • このサーバの証明書 • このサーバの証明書を割り当てた認証局の証明書で始まる証明書たちのチェイン
CLIENT_CERTIF_REQUEST ⟵	サーバはCLIENT_CERTIF_REQUESTメッセージを送信し、クライアント認証を求めることができる。
SERVER_HELLO_DONE ⟵	サーバはSERVER_HELLO_DONEメッセージを送信する。このコマンドはこのサーバがこのSSLハンドシェイクのフェイズを完了したことを示す。
CLIENT_CERTIFICATE ⟶	SERVER_HELLO_DONEメッセージを受けたらクライアントは最初にこのメッセージを送信できる。これはサーバがクライアント認証を求めたときにのみ送信される。このメッセージで送信されるものはSERVER_CERTIFICATEと似て以下のものとなる: • このクライアントの証明書 • オプションとしてこのクライアントの証明書を割り当てた認証局の証明書で始まる証明書たちのチェイン
サーバの証明書を検証する	
	クライアントの証明書を検証する
CERTIFICATE_VERIFY ⟵	サーバはクライアントに対しそのクライアントの証明書を検証したことを知らせる。このメッセージはクライアント認証が求められているときに送信される
CLIENT_KEY_EXCHANGE	PKCS #1エンコードされたプレマスタ・シークレット(Premaster Secret)で、

⟶	サーバからSERVER_CERTIFICATEメッセージで受理した公開鍵で暗号化されている。このプレマスタ・シークレットからサーバはクライアントにデータを送信するための鍵を生成する。先頭に置かれるヘッダ・ハンドシェークは暗号化されない
CHANGE_CIPHER_SPEC ⟶	クライアントはCHANGE_CIPHER_SPECメッセージを送信する。このメッセージはこのセッション中にこのクライアントが送信するこれからのSSLデータ・レコードの中身が暗号化されることを通知するものである。5バイトからなるSSLレコード・ヘッダは暗号化されない
FINISHED ⟶	クライアントはFINISHEDメッセージを送信する。このメッセージはこの時点までのクライアントとサーバ間で交わされた総てのSSLハンドシェイク・コマンドたちのダイジェストを含む。このメッセージにより、クライアントとサーバ間で暗号化されないでこれまで交わされたメッセージたちのどれもが途中で誰かによって変えられていないことを確認する為に送信される。
CHANGE_CIPHER_SPEC ⟵	サーバはCHANGE_CIPHER_SPECメッセージを送信する。このメッセージはサーバが送信するこれからのSSLデータ・レコードの中身が暗号化されることを通知するものである。
FINISHED ⟵	サーバはFINISHEDコメッセージを送信する。このメッセージはこの時点までのクライアントとサーバ間で交わされた総てのSSLハンドシェイクメッセージたちのダイジェストを含む
これ以降クライアントとサーバーは，共有したプレマスタ・シークレットから共通鍵を生成し，その共通鍵を使って暗号通信を実施する	

この表で赤で示したメッセージがCLIENT-CERT認証に関わるものである。

20.3節　　OpenSSLとそのインストール

これまでのdart:ioの暗号化ライブラリではApacheなどで使われているOpenSSLではなくてNetwork Security Services (NSS)を採用していた。NSSでは証明書や鍵はcert9.dbやkey4.dbなどといったデータベースに置かれていた。一方OpenSSLではPEMファイルが使われ、Apacheではキーストア(keystore.jks)も使用されている。またコマンド行ツールはNSSではcertutilであり、一方OpenSSLではopensslあるいはkeytoolである。

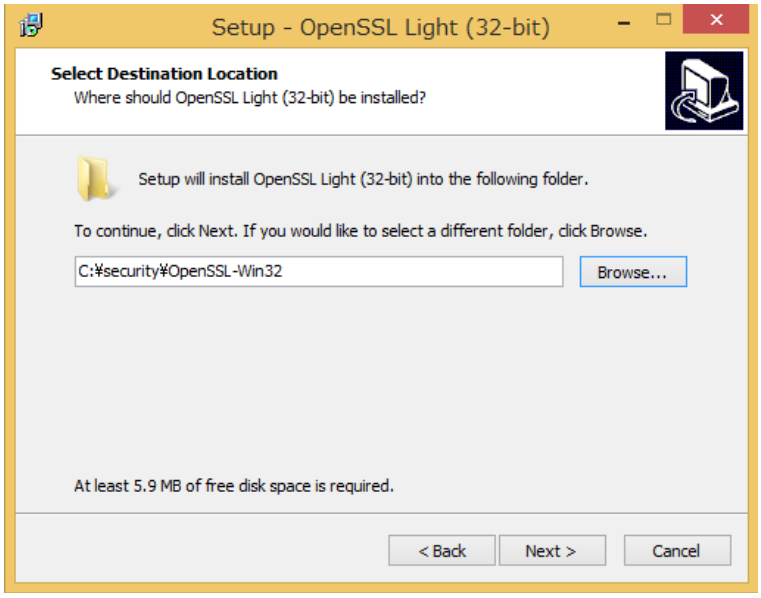
新しいBoringSSL対応のライブラリでもPEM(.pem)ファイルが使われるが、これは.dbファイルをPEM形式にエクスポートできる。従ってNSSをで使っていた鍵と証明書が既に存在する場合は、その手順に従えばよい。

PEM(Privacy-enhanced Electronic Mail)書式は認証局が証明書を発行する際に使われる最も一般的な書式である。拡張子は .pem, .crt, .cer, 及び .keyである。これらはBase64エンコードされたASCIIファイルで、例えば公開鍵の場合は"-----BEGIN CERTIFICATE-----"と"-----END CERTIFICATE-----"文で挟まれたテキストである。サーバ証明書、中間証明書、プライベート鍵などもこの書式に変換できる。

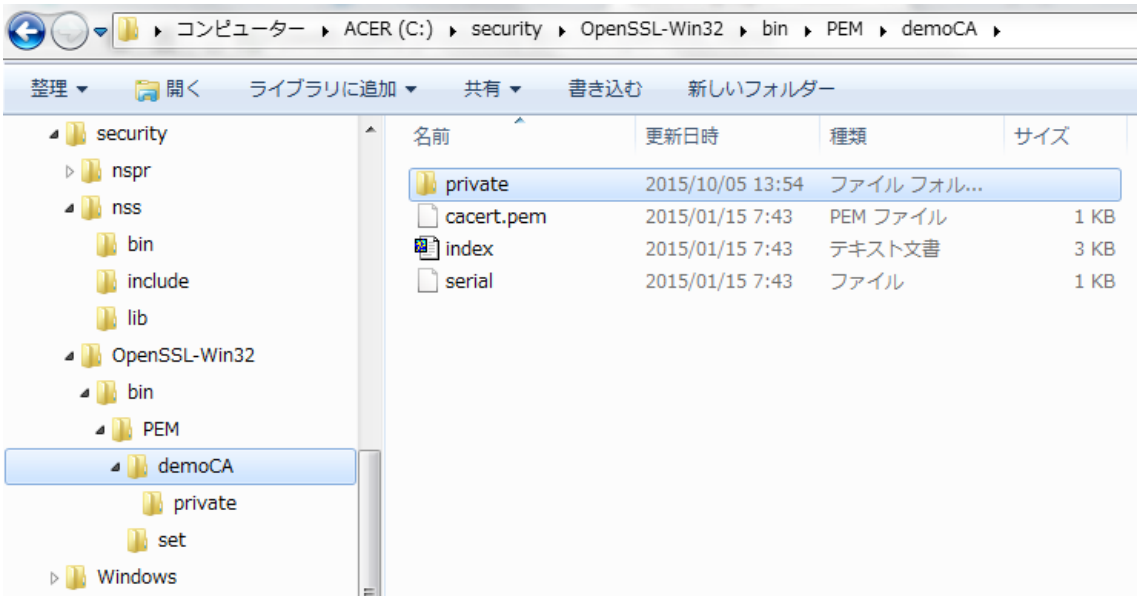
OpenSSLに関する日本語の解説は多くの認証機関またはそれらの代理事業者たちが提供している（例えばDigiCert社のサーバ証明書に関する解説）ので、それを参照して頂きたい。

ダウンロードとインストール

ここではWindows用のWin32 OpenSSL v1.0.2d Light（64ビット機でも32ビット版をインストールすることが推奨されている）をダウンロードしてインストールした例である。ここではNSSと同じc:\securityというディレクトリに配置してある。



インストーラ画面（インストールの場所指定）



openSSLのインストール

基本的なコマンド

openssslコマンドの詳細は公式サイトのマニュアル（あるいは日本語のサイト）を見ることになるが、かなり専門的である。基本的なopenssslコマンドは次のようである（下線が引かれた個所はユーザが設定するもの）：

1. 一般的なコマンド
   CSR（証明書署名要求）、証明書、プライベート・キーの生成などのタスク

- 新規プライベート・キーと証明書署名要求の生成

```
openssl req -out CSR.csr -new -newkey rsa:2048 -nodes -keyout privateKey.key
```

- 自己署名の証明書の生成

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout privateKey.key -out certificate.crt
```

- 既存のプライベート・キーに対する証明書署名要求（CSR）を生成する

```
openssl req -out CSR.csr -key privateKey.key -new
```

- 既存の証明書に基づいた証明書署名要求（CSR）を生成する

```
openssl x509 -x509toreq -in certificate.crt -out CSR.csr -signkey privateKey.key
```

- プライベート・キーからパス・フレーズを削除する

```
openssl rsa -in privateKey.pem -out newPrivateKey.pem
```

2. OpenSSLを使ってチェックする
   証明書、証明書署名要求、あるいはプライベート・キーのなかの情報をチェックしたいときにこれらのコマンドを使う。他のオンライン・ツールを使ってCSRや証明書をチェックすることも可能である（例として日本語のサイト）。

- CSR（証明書署名要求）をチェックする

```
openssl req -text -noout -verify -in CSR.csr
```

- プライベート・キーをチェックする

```
openssl rsa -in privateKey.key -check
```

- 証明書をチェックする

```
openssl x509 -in certificate.crt -text -noout
```

- PKCS#12ファイル(.pfx or .p12)をチェックする

```
openssl pkcs12 -info -in keyStore.p12
```

3 OpenSSLを使ってデバッグする
プライベート・キーが証明書と合致しないとかあるサイトにインストールした証明書が信頼されないとかのエラーを受けた場合は、これらのコマンドのなかのひとつを試されたい。あるSSL証明書が正しくインストールされたかを確認したいときは、SSL Checkerを試されたい。

- パブリック・キーのMD5ハッシュがCSRまたはプライベート・キーのなかのそれと合致するかをチェックする

```
openssl x509 -noout -modulus -in certificate.crt | openssl md5
```

```
openssl rsa -noout -modulus -in privateKey.key | openssl md5
```

```
openssl req -noout -modulus -in CSR.csr | openssl md5
```

```
SSL 接続をチェックする。総ての証明書（仲介も含む）が表示される
openssl s_client -connect www.paypal.com:443
```

4. OpenSSLを使って変換する
   以下のコマンドはサーバあるいはソフトウエアのタイプに適合させるために証明書と鍵を別の書式に変

換させるものである。例えば、ApacheやDartで動作する通常のPEMファイルをPFX (PKCS#12)に変換してTomcatやIISで動作するようにできる。SSLコンバータを使うとより簡単に変換できる場合がある。

- DERファイル(.crt .cer .der) をPEMに変換する

```
openssl x509 -inform der -in certificate.cer -out certificate.pem
```

- PEMファイルをDERに変換する

```
openssl x509 -outform der -in certificate.pem -out certificate.der
```

- プライベート鍵と証明書を含んだPKCS#12 ファイル(.pfx .p12)をPEMに変換する

```
openssl pkcs12 -in keyStore.pfx -out keyStore.pem -nodes
```

-nocertsを追加してプライベート鍵のみ変換したり、-nokeysを付加して証明書のみ変換したりできる

- PEM証明書ファイルとプライベート鍵をPKCS#12 (.pfx .p12)に変換する

```
openssl pkcs12 -export -out certificate.pfx -inkey privateKey.key -in certificate.crt
-certfile CACert.crt
```

20.4節　　秘密鍵と証明書の用意

筆者の「改定サーブレット・チュートリアル」の第15章参照の「認証局からの証明書の取得」の項で示したように、認証局(CA)から証明書を取得する為には、先ず自分の秘密鍵を用意し、それをもとに証明書署名要求(CSR: Certificate Signing Request)を作成しなければならない。

最初にHTTPSサーバ構築に必要なファイル名やパスワード等をあらかじめ決めておくことをお勧めする。以下はその例である:

秘密鍵のファイル名	my_key.pem
認証局署名つきサーバ証明書のファイル名	my_cert.pem
証明書署名要求のファイル名	my_csr.pem
秘密鍵にたいするパスフレーズ(必要なら)	changeit

また最初にコマンドプロンプト上で次のようにディレクトリの移動とパス設定を行う:

```
cd c:\security\openssl-win32\bin\pem
path c:\security\openssl-win32\bin
```

秘密鍵と証明書署名要求(CSR: Certificate Signing Request)を作成する

前節で示したように、openssslツールでは秘密鍵の生成と証明書署名要求の作成を同時に行うことが可能であるがその場合はパスフレーズの入力が要求されないことに注意。

```
openssl req -out my_csr.pem -new -newkey rsa:2048 -nodes -keyout my_key.pem
```

以下はその実行例である:

```
c:\security\OpenSSL-Win32\bin\PEM>openssl req -out my_csr.pem -new -newkey rsa:2048 -nodes
-keyout my_key.pem
Loading 'screen' into random state - done
Generating a 2048 bit RSA private key
...+++
.....................................................+++
writing new private key to 'my_key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:JP
State or Province Name (full name) [Some-State]:Tokyo
Locality Name (eg, city) []:Minato-ku
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Cresc
Organizational Unit Name (eg, section) []:System
Common Name (e.g. server FQDN or YOUR name) []:Terry
Email Address []:

Please enter the following 'extra' attributes to be sent with your certificate request
A challenge password []:
An optional company name []:

c:\security\OpenSSL-Win32\bin\PEM>
```

幾つかの入力項目があるがこれは、新しい証明要求と証明書の所有者を特定する情報で、RFC-1485に準拠して指定（CN=..., OU=...）する。Cは国名、CNは一般名（Common Name (eg, YOUR name)）、Oは会社名（Organization Name）、OUは部門名（Organizational Unit Name）などである。メール・アドレス、チャレンジ・パスワード、オプショナルな会社名はスキップすればよい。

これでmy_key.pem及びmy_csr.pemの二つのファイルが生成される。

my_key.pemの内容は次のコマンドでチェックできる：

```
c:\security\OpenSSL-Win32\bin\PEM>openssl rsa -in my_key.pem -check
RSA key ok
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEA6YV6R0zApfofoY8DX7ws7FebeFBsflx/w5S4hFxQ64BjgK7s
kPLSLCtIybZmms/H8p0AQRBfkP8oVY/m+YsBSWm50C2Sz1wYQ9pC0uZWH99SPfRR
IC+xnsTjym3q/Ht+7mO39keqRhqvr+LHkC+q/MyX9swc2ealXTVVXwir2TE0uCMi
MviILop6WCU5uU5u3Yc3NdMigcX08OMzTfxGsBYsobna/4tMSELSquiYYt6VJMZa
5N2UYK3XCiGIAh7gaz5JV1Tr5ayqO2wLLP+15Y6z7dXuvl01lBD+qzjImfuKwVAk
irkRODh4T4gia3ToJqoEITg59yMQozYbUwIK5wIDAQABAoIBAF+xevJM2YUqglvK
Jy/MBPy0ydj72/nMHe8fup1C10YqTpwlEnwzhTzkX+eI/3LhQqaA/+Gpu//HD9hA
J/6Kn/Rdhu9wPYEL1EW54zhZe2GEOkd5HuV5pefR9ya09F6SnOlDo9cgdv5TYTtK
S00iu4bssDlKW2hPw+P2Pw26l/FgGD2Npcjd+/zKRBof6ZXIXM3R918ThTAc9v2O
tn+zKGuT/ADU1nsfq2yOgz2dry1mZ/Mut5ok6zlDNCY6cnqlGIrcZfhWz183uBiX
4/+Pr/7hzELxDxplQvm5lMlK1dzmr4PXOz0THVjgHECvat5MHKoz9Gw81NYZCCYt
bmGkDAECgYEA9jdo1nHfYa0mR6B6OtxnXVtkZpC57X2ilnPD1K9sTHUN/DP8ir/3
nftWoYYkYzIHZ5MeqG9uunKiQaoIDz3S651je39uT0U9SOitaRFTndEpmjPGFsja
2DKa+3RyOpVCqMP0LwyP/Rz/scSN0aGg5Npnf/uosmnTLoIW4SzBuicCgYEA8szu
HrMbZFMefUGoXg45LJjy84+379CkosraFpH2xdpe0qk2kjZeAjiLCl1q8IdXPcwB
bUiz0Go1jOWkQW5H9d9FVFF82A0JM0KT9f9r43mSrFOsE/ELbS0HJZRFr5MhPRg2
1G6LDDay+Bp2ICg++bSTXQY02QtjRQtVyOoPYUECgYEAl8Z6Us7pQ53n3fl3v3cE
JNFkp7EJW/O5WQNNsXrdyJfToictwx4o9vLraTB2l8tMMzXGjU/7suVdThsRUsvq
jCF+JK/eAPGOoLYFx6HNqJg6C/tkXcE83k77qIwUqjY+XChHCwW0cPQCTsP6JEJ9
GvXjHUq1qB2B98Zrci0T0rsCgYAlye9lp8qjmqwsIoPp35zWbBekAMJH+Nkm3RuE
V8Nol8waTWvI5d1LyzEYs+Fo/id93rp6H86cqWscOsGNzXQ1uEI6FVSw65Z6++m7
```

```
Z72K8ej6GSu3DtUQcOmj16fg461QrOwbs/jANeM06iloZ9slUg19dPHUtgkFPfZk
BsnOAQKBgFOipZlfUmBmjIKeWaA87H/GZFfSnNLRYFmRwJY6UhGTm6X/v2A/qZRW
9t+joWCWPasuJ1TUzVz9xRunFnUR9d0HE7pZSk+tXNl2giNvm4QAubQg2x8y+voC
G0aRqy300De7TsyG9tZJweGjXUTiKcjnwSQ3PIizRG3W1LAEs5Aa
-----END RSA PRIVATE KEY-----
```

my_csr.pemは次のコマンドでその内容を確認できる:

```
c:\security\OpenSSL-Win32\bin\PEM>openssl req -in my_csr.pem -text
Certificate Request:
  Data:
    Version: 0 (0x0)
    Subject: C=JP, ST=Tokyo, L=Minato-ku, O=Cresc, OU=System, CN=Terry
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
        Public-Key: (2048 bit)
        Modulus:
          00:e9:85:7a:47:4c:c0:a5:fa:1f:a1:8f:03:5f:bc:
          2c:ec:57:9b:78:50:6c:7e:5c:7f:c3:94:b8:84:5c:
          50:eb:80:63:80:ae:ec:90:f2:d2:2c:2b:48:c9:b6:
          66:9a:cf:c7:f2:9d:00:41:10:5f:90:ff:28:55:8f:
          e6:f9:8b:01:49:69:b9:d0:2d:92:cf:5c:18:43:da:
          42:d2:e6:56:1f:df:52:3d:f4:51:20:2f:b1:9e:c4:
          e3:ca:6d:ea:fc:7b:7e:ee:63:b7:f6:47:aa:46:1a:
          af:af:e2:c7:90:2f:aa:fc:cc:97:f6:cc:1c:d9:e6:
          a5:5d:35:55:5f:08:ab:d9:31:34:b8:23:22:32:f8:
          88:2e:8a:7a:58:25:39:b9:4e:6e:dd:87:37:35:d3:
          22:81:c5:f4:f0:e3:33:4d:fc:46:b0:16:2c:a1:b9:
          da:ff:8b:4c:48:42:d2:aa:e8:98:62:de:95:24:c6:
          5a:e4:dd:94:60:ad:d7:0a:21:88:02:1e:e0:6b:3e:
          49:57:54:eb:e5:ac:aa:3b:6c:0b:2c:ff:b5:e5:8e:
          b3:ed:d5:ee:be:5d:35:94:10:fe:ab:38:c8:99:fb:
          8a:c1:50:24:8a:b9:11:38:38:78:4f:88:22:6b:74:
          e8:26:aa:04:21:38:39:f7:23:10:a3:36:1b:53:02:
          0a:e7
        Exponent: 65537 (0x10001)
    Attributes:
      a0:00
  Signature Algorithm: sha256WithRSAEncryption
     d2:92:b1:70:07:1a:d1:67:c8:83:c4:f0:7e:15:f7:ad:26:77:
     87:b8:5a:b4:e3:6e:1e:31:64:99:f0:28:7f:90:6c:4f:c5:c6:
(以下省略)
```

ここにはPublic-Key: (2048 bit)としてRSA2048ビットの公開鍵、そしてSignature AlgorithmとしてSHA256変換した256バイトの秘密鍵のハッシュ値が含まれていることが理解されよう。

Verisignなどの公式の認証機関から証明書を取得するにはBEGINの行のつぎからENDの行の前までをコピー/ペーストして渡すことになる。詳細は各認証機関の取得手順に関する解説に従うこと。

信頼される認証機関からの証明書の取得

通常そのような機関からはメールで署名つき証明書が.cerや.pemといった拡張子で送信されてくる。「改定サーブレット・チュートリアル」の第15章参照の「認証局からの証明書の取得」の項を参照のこと。次の例はVeriSignから取得したテスト用の有効期間が短期間の証明書である。

その詳細は長いので最初の部分のみを示すと次のようになっている:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            06:af:57:23:51:13:28:28:2e:2d:d3:2d:ca:6b:d2:3e
        Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption
        Issuer: "CN=VeriSign Class 3 Secure Server 1024-bit Test CA,OU=Terms
            of use at https://www.verisign.com/cps/testca/ (c)07,OU=VeriSign
            Trust Network,OU=FOR TEST PURPOSES ONLY,O="VeriSign, Inc.",C=US"
        Validity:
            Not Before: Sat Apr 09 00:00:00 2011
            Not After : Sat Apr 23 23:59:59 2011
        Subject: "CN=localhost,OU=Terms of use at www.verisign.com/cps/testca
             (c)05,OU=Tech,O=Cresc,L=Mita,ST=Tokyo,C=JP"
        Subject Public Key Info:
            Public Key Algorithm: PKCS #1 RSA Encryption
            RSA Public Key:
                Modulus:
                    bd:11:dd:3b:22:32:82:d3:9e:de:3f:2a:56:a9:83:ad:
                    05:29:75:20:d7:6a:a7:c1:83:b5:4d:d1:61:c5:8a:00:
                    ba:ee:d9:a4:94:f7:f1:3d:eb:a9:10:1e:11:ee:f8:b1:
                    4e:1b:e8:28:e9:d2:c8:d1:b7:30:19:e3:f8:58:bd:c8:
                    89:c3:9e:00:24:29:db:78:3d:61:60:ed:42:f7:2a:a9:
                    e2:82:69:71:69:16:a1:fe:fd:b1:cf:09:11:35:3d:2f:
                    08:b3:cb:85:bf:79:1d:3b:4d:dd:0d:a4:da:50:9c:d9:
                    95:40:fa:93:f9:49:53:00:1a:71:2e:4e:bd:60:36:09
                Exponent: 65537 (0x10001)
        Signed Extensions:
            Name: Certificate Basic Constraints
            Data: Is not a CA.

            Name: Certificate Key Usage
            Usages: Digital Signature
                    Key Encipherment
（以下省略）
```

これはこの公開鍵の所有者が信用できることをVeriSignが証明した詳細な内容である。

自己証明書の作成(Self-signed Certificate)

本来は「改定サーブレット・チュートリアル」の第15章参照の「認証局からの証明書の取得」の項で示したように、作成した証明書発行要求書をもとにした証明書を認証局から取得しなければならない。しかしながら先ずブラウザが警告を出すものの自分で証明した自己証明書（いわゆるオレオレ証明書）を使って、手っ取り早くSSL接続の実験を進めることとする。

コマンドは次のようになる:

```
openssl x509 -req -signkey my_key.pem -in my_csr.pem -days 3650 -out my_crt.pem
```

以下はその実行例である:

```
c:\security\OpenSSL-Win32\bin\PEM>openssl x509 -req -signkey my_key.pem -in my_csr.pem
-days 3650 -out my_crt.pem
Loading 'screen' into random state - done
Signature ok
subject=/C=JP/ST=Tokyo/L=Minato-ku/O=Cresc/OU=System/CN=Terry
Getting Private key
```

ここでは10年間の有効期間を設定している。

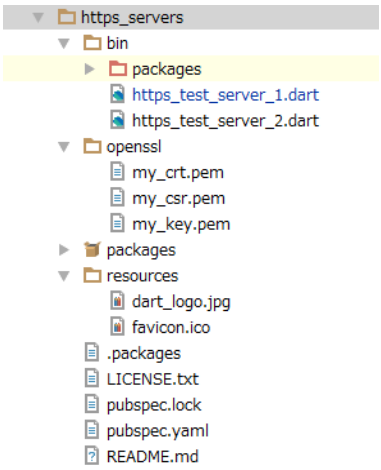
その内容は次のように確認できる:

```
c:\security\OpenSSL-Win32\bin\PEM>openssl x509 -text -in my_crt.pem
Certificate:
   Data:
      Version: 1 (0x0)
      Serial Number:
         b9:cd:48:c9:ff:54:1e:e0
   Signature Algorithm: sha256WithRSAEncryption
      Issuer: C=JP, ST=Tokyo, L=Minato-ku, O=Cresc, OU=System, CN=Terry
      Validity
         Not Before: Oct 15 04:05:58 2015 GMT
         Not After : Oct 12 04:05:58 2025 GMT
      Subject: C=JP, ST=Tokyo, L=Minato-ku, O=Cresc, OU=System, CN=Terry
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
            Modulus:
               00:e9:85:7a:47:4c:c0:a5:fa:1f:a1:8f:03:5f:bc:
               2c:ec:57:9b:78:50:6c:7e:5c:7f:c3:94:b8:84:5c:
               (途中省略)
               e8:26:aa:04:21:38:39:f7:23:10:a3:36:1b:53:02:
               0a:e7
            Exponent: 65537 (0x10001)
   Signature Algorithm: sha256WithRSAEncryption
       c1:64:e9:11:62:a9:3d:a7:35:40:d5:08:2e:f6:94:a0:10:9c:
       66:87:7c:1c:61:00:5e:ea:d6:c5:b9:57:fc:83:f7:17:69:c2:
       (途中省略)
       70:1a:c0:d2:9f:b6:6c:fc:db:ef:d4:ab:19:58:84:80:37:ba:
       7f:2d:2e:4d
-----BEGIN CERTIFICATE-----
MIIDQDCCAigCCQC5zUjJ/1Qe4DANBgkqhkiG9w0BAQsFADBiMQswCQYDVQQGEwJK
UDEOMAwGA1UECAwFVG9reW8xEjAQBgNVBAcMCU1pbmF0by1rdTEOMAwGA1UECgwF
      (途中省略)
t85B6NxTzBzSbEJQXj+MSo/NHUtWJXhbcApcQEe4PIeNMdezcs2zSyrOpbPBHXAa
wNKftmz82+/UqxlYhIA3un8tLk0=
-----END CERTIFICATE-----
```

20.5節　　簡単なHTTPSサーバの実験

簡単なHTTPSサーバを実験してみよう。このアプリケーションはGitHubからDownload ZIPのボタンをクリックしてダウンロードできる。なおHTTPSサーバのプログラム開発にはDartチームが作成したサンプルも参考になる。

このアプリケーションをダウンロードしてインストールし、pubspec.yamlを選択してPub : Get Dependenciesを実行して、必要なライブラリを取り込む。そうすると自分のIDE上には次のようなファイル構成が表示される:



https_serversの構成

- サーバの実行ファイルは/binのなかに置かれる。
- BoringSSLのPEMファイルたちは/openssslのなかに置かれる。
- アプリケーションに必要な静的リソースは/resourcesのなかに置かれる。
- いつものようにパッケージ・マネージャはpubspec.yamlを調べて、必要なパッケージを/packagesというディレクトリを(ここでは2つ)生成して収容している。

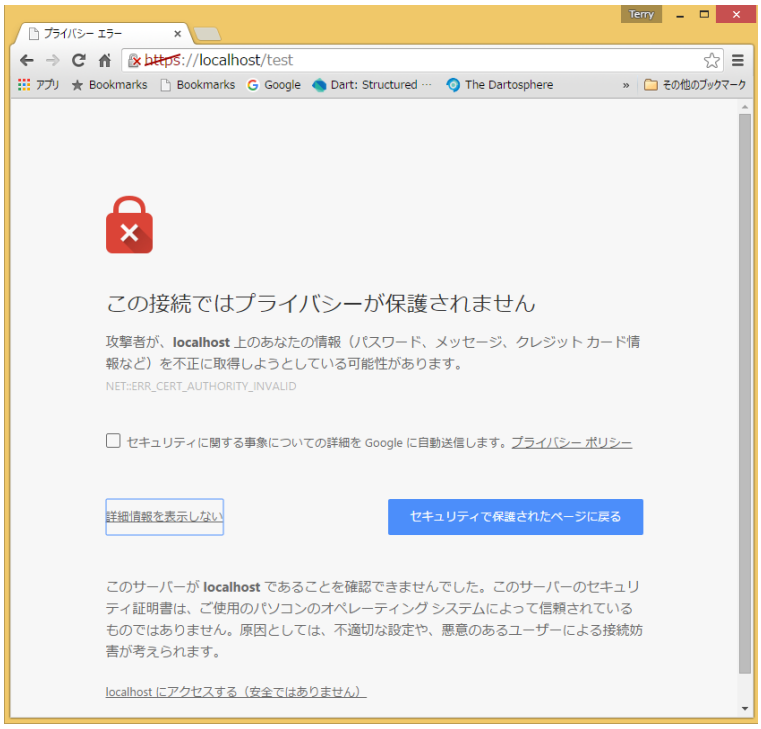
https_test_server_1

https_test_server_1.dartはブラウザがこのサーバに正常にHTTPSアクセスできたときに簡単なメッセージを返す。https_test_server_1.dartを実行させると、エディタのコンソールには次のようなメッセージが表示され、NSSライブラリの初期化が完了して、このサーバがクライアントからのHTTPS要求を受け付けることが可能になっていることが判る:

```
2013-06-28 10:29:58.629 - BoringSSL security context initialized.
2013-06-28 10:29:58.676 - https_test_1 server started.
```

ここでChromeブラウザからHTTPS://localhost/testをアドレス・バーに入れてアクセスさせると、ブラウザのインスタンスは最初にlocalhostをHTTPSでアクセスした場合は次のような警告を出す。これはここで使われている証明書が自己証明書であり、信頼された認証機関から発行されたものでないからである。

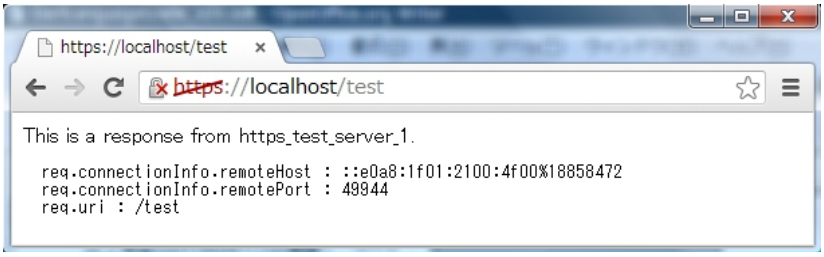
左下の「詳細設定」をクリックして、一番下の「localhostにアクセスする（安全ではありません）」をクリックすれば、次のような画面になる。



ブラウザからの警告

VeriSignなどの信頼された認証機関からの証明書を既に取得しているユーザは、自分のcert9.db及びkey4.dbを上書きコピーし、ニックネームをパスワードをそれにあわせて指定してやれば、ブラウザはこのような警告を出さない。

「localhostにアクセスする（安全ではありません）」リンクをクリックすると次のような画面となる：



アドレス・バーには安全でないサーバにアクセスしていることが表示されているものの、正しくSSL接続がとれ、サーバから応答が返ってきていることが理解されよう。

なおアドレス・バーの×印のついた鍵のマークをクリックすれば、下図のようにlocalhostに関する情報（証明書情報等）が得られる。

localhostの情報

https_test_server_1.dartのコードは次のようになっている:

https_test_server_1.dart

```
import 'dart:io';

final LOG_REQUESTS = true;        // set true for debugging
final HOST_NAME = 'localhost';    // use loop back address for the test
final int SERVER_PORT = 443;      // use well known HTTPS port number
final REQ_PATH = '/test';         // request path for this application
final CERT_PATH = 'openssl/my_crt.pem'; // path to pem cert file
final KEY_PATH = 'openssl/my_key.pem';  // path to pem private key file
final KEY_PASSWORD = 'changeit';        // password for the key file

SecurityContext serverContext;   // security context of this server

void main() {
  setSecurityContext();
  listenHttpsRequest();
}

void setSecurityContext() {
  serverContext = new SecurityContext()
    ..useCertificateChain(CERT_PATH)
    ..usePrivateKey(KEY_PATH, password: KEY_PASSWORD);
  log('BoringSSL security context initialized.');
}

void listenHttpsRequest() {
  HttpServer.bindSecure(HOST_NAME,
                        SERVER_PORT,
                        serverContext)
  .then((HttpServer server) {
    server.listen(
      (HttpRequest req) {
        if (req.uri.path.contains(REQ_PATH)) processRequest(req);
        else {
          req.response.statusCode = HttpStatus.BAD_REQUEST;
          req.response.close();
        }
      },
      onError: (err) {
```

```
      print('listen: error: $err');
    },
    onDone: () {
      print('listen: done');
    },
    cancelOnError: false
    );
   log('https_test_1 server started.');
 });
}

void processRequest(HttpRequest req) {
  req.response.headers.add("Content-Type", "text/html; charset=UTF-8");
  String mes = requestInf(req);
  if (LOG_REQUESTS) log('\n$mes');
  req.response.write(
    '''<!DOCTYPE html>This is a response from https_test_server_1.
    <pre>$mes</pre></html>''');
  req.response.close();
}

// adapt this function to your logger
void log(String s) {
  print('${new DateTime.now().toString()} - $s');
}

String requestInf(HttpRequest req) =>
  '''
  req.connectionInfo.remoteAddress : ${req.connectionInfo.remoteAddress}
  req.connectionInfo.remotePort : ${req.connectionInfo.remotePort}
  req.uri : ${req.uri}''';
```

幾つかのポイント列記すると:

- final変数は次のようになっている。
  - final LOG_REQUESTS = true; // ログを取るかどうかを指定する。実験の場合はtrueのままとする
  - final HOST_NAME = 'localhost'; // テスト用にはループバック・アドレスを使用する。本番には指定されたグローバル・アドレスをセットする。
  - final int SERVER_PORT = 443; // ここではポート番号はHTTPS用に指定されている443を使用する。Apacheなどでは実験用に8443を使う場合が多いが、その場合はブラウザのアドレス・バーにはhttps://localhost:8443/testとポート番号を指定してやらねばならない。
  - final REQ_PATH = '/test'; // ここでは/testという要求パスを持った要求のみを取扱う。
  - final CERT_PATH = 'openssl/my_crt.pem'; // PEM証明書ファイルへのパス
  - final KEY_PATH = 'openssl/my_key.pem'; // PEM秘密鍵ファイルへのパス。これらの2つのパスは相対パスで指定しているが、無論絶対パスで指定しても良い。なおWindows以外のOSの場合は、それに準拠したパスの指定の仕方をしなければならない。
  - final KEY_PASSWORD = 'changeit'; // PEM秘密鍵ファイルにアクセスするためのパスワード

- サーバの起動に先だって、setSecurityContextメソッドを使ってDart VMが用意しているSecurityContextの初期化をしておかねばならない。これによりBoringSSLは証明書と秘密鍵場所と秘密鍵のパスワードを知り、クライアントからのSSL接続要求に対応できるようになる。

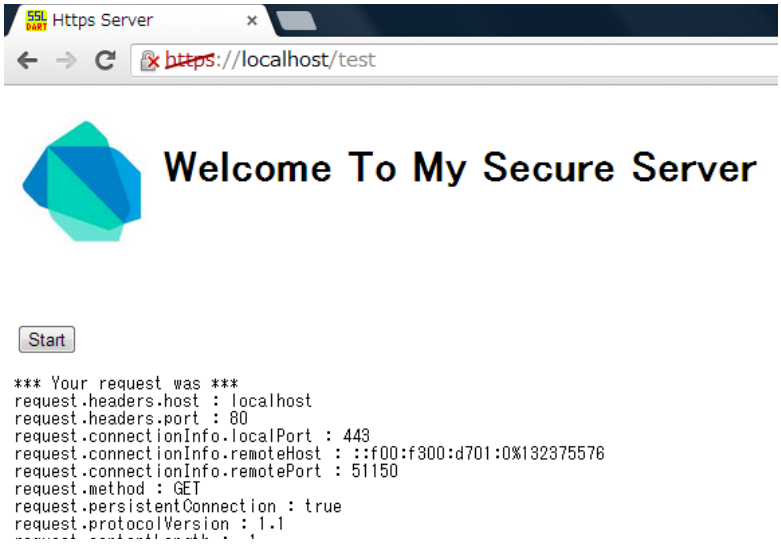
- 同じくサーバの起動に先だって、HttpServer.bindSecureという静的メソッドを使ってSSL接続からのHTTP要求を受理するように設定する。

- その後の処理は基本的に通常のHTTP要求の処理と同じである。

https_test_server_2

このプログラムはHTTPSサーバとして一般的に活用できるように必要な機能（以下に示す）を含めたものである。読者はこのコードを理解することで、実際の商用に耐えるようなサーバが開発できるようになろう。

- favicon.icoに対応　（これは下図のブラウザのタブに表示されるアイコンである）
- ブラウザ画面に自分が用意したグラフィックスを表示できるようにファイル・サーバ機能を含めている
- セッション管理とそれを使った画面遷移の基本的な手順



このプログラムはセッション管理の学習に使ったHttpSessionTestServer.dartと類似しているので、簡単に実験出来よう。

favicon.icoはhttp://www.favicon.cc/のようなツールを使って簡単に作成できる。このアプリケーションでは/resoucesというディレクトリに他の静的リソースとともに収容してある。従ってクライアントからのfavicon要求に対しては、そのディレクトリに変換してファイル・サーバ機能を呼び出している。

20.6節　　関連APIの和訳

注意：本APIはDart SDKの1.13.0-dev.1.0以降でのみ適用される。

HttpServer.bindSecure

注意：ここではstaticメソッドのbindSecureのみを抜き出してある。

HttpServer class

実装	
Socket	Stream<HttpRequest>
staticメソッド	
Future<HttpServer> **bindSecure**(address, int port, SecurityContext context, {int backlog: 0, bool v6Only: false, String certificateName, bool requestClientCertificate: false, bool shared: false})	addressはString またはInternetAddressのいずれかであり得る。もしaddressがStringのときは、bindはInternetAddress.lookupを実行し、そのリストの最初の値を採用する。ローカル・ホストからの到来接続のみを受け付けるループバック・アダプタ上でリスンするときは、この値としてInternetAddress.LOOPBACK_IP_V4またはInternetAddress.LOOPBACK_IP_V6 を使用すること。 ネットワークからの到来接続を受け付けるときは、総てのインターフェイスにバインドするためにInternetAddress.ANY_IP_V4またはInternetAddress.ANY_IP_V6という値のどれかを使用するか、特定のインターフェイスのIPアドレスを使用する。 もしIPバージョン6(IPv6)アドレスが使われているときは、IPバージョン6(IPv6)とIPバージョン4(IPv4)の接続の双方とも受け付けられる。IPバージョン6(IPv6)アドレスのみに制限したいときは、IPバージョン6のみとセットするために v6Onlyを使用する。 もしportが0という値であるときは、このシステムではエフェメラル・ポートがこのシステムで選択される。実際に使われているポート番号はportゲッタで取得できる。 オプショナルな引数である backlogは下位に存在しているOSのリスン設定に対しリスン・バックログを指定するのに使われる。backlogが0（デフォルト値）という値を持っているときは、このシステムで妥当な値が選択される。 ニックネームまたは識別名（DN: Distinguished Name）をもった証明書のcertificateNameは証明書データベース内で検索され、サーバ認証用に使用される。もし requestClientCertificateがtrueのときは、そのサーバはクライアントたちにクライアント証明書で認証するよう要求する。 オプショナルな引数であるsharedは同じaddress, port 及びv6Onlyの組み合わせにたいし、同じDartのプロセスからの追加的なバインドを可能とするかどうかを指定する。もしsharedがtrueのときは、付加的なバインドが実行され、到来接続はHttpServerたちのセット間で分配される。これを使う一つの手段としては、複数のアイソレートたちに到来接続たちを配分させることである。

SecurityContext

SecurityContext class
安全なクライアント接続を確立する際に信頼するための証明書、安全なサーバからサービスするための証明書チェインとプライベート鍵で構成されるオブジェクト。 SecureSocket及びSecureServerクラスは各々のconnectおよびbindメソッドたちへの引数としてあるSecurityContextオブジェクトをとる。 証明書たちと鍵たちはディスク上のPEMファイル達からSecurityContextオブジェクトに付加できる。PEMファイ

ルはひとつまたはそれ以上の"-----BEGIN CERTIFICATE -----"と"-----END CERTIFICATE-----"といったデリミタ文字列で囲まれたbase-64エンコードの DER直列化 ASN1オブジェクトたちで構成される。 DER(Distinguished Encoding Rules)はASN1オブジェクトたちをバイト文字列へのカノニカルなバイナリ直列化である。	
static属性	
SecurityContext **defaultContext**	リード・オンリー
コンストラクタ	
SecurityContext()	
メソッド	
void **setAlpnProtocols**(List<String> protocols, bool isServer)	あるクライアント接続またはサーバ接続でサポートされるアプリケーション・レベルのプロトコルのリストをセットする。TLSに対するこのALPN (application level protocol negotiation)拡張により、クライアントはTLSクライアントhelloメッセージのなかでプロトコルのリストを送信でき、サーバはそのなかから一つを選択しそれをサーバのhelloメッセージのなかで送り返すことができるようになる。 同じSecurityContextを使ってクライアント接続またはサーバ接続に対し別のプロトコルたちのリストを送信できる。ブール値のisServer引数は、このリストをサーバ接続のためにセットするかあるいはクライアント接続のためにセットするかを指定する。
void **setClientAuthorities**(String file)	接続しようとしているクライアントからクライアント認証を要求する際にSecureServerが受け付けたと広告する認証局の名前のリストをセットする。fileは受け付けられた認証局の署名つきの証明書（認証局名は証明書から抽出される）を含んだPEMファイルである。
void **setTrustedCertificates**({String file, String directory})	SecureSocketクライアント接続があるセキュアなサーバに接続する際に使われる信頼される X509証明書たちのセットをセットする。 信頼される証明書たちのセットをセットするには、証明書たちのために単一のPEMファイルで、あるいは個々のPEMファイルたちを含むディレクトリでセットする2つの手段がある。 fileはX509証明書たちを含むオプショナルなPEMファイルであり、通常認証局たちからのルート証明書たちである。 directoryはPEMファイルたちを含むオプショナルなディレクトリである。このdirectoryは付加されたファイル・システムのリンクが含まれていなければならない。このリンクたちはその証明書を含むファイルへの証明書の識別名（distinguished name：DN）のハッシュに基づいたファイル名たちにリンクする。 OpenSSLにはあるディレクトリ内でこれらのリンクたちを生成するためのc_rehashというツールがある。
void **useCertificateChain**(String file)	セキュアな接続を作るためにSecureServerによってサービスされるX509証明書たち（サーバ証明書が含まれる）のチェインをセットする。fileはX509証明書たちが含まれるPEMファイルであり、そのサーバ証明に対する署名されたチェインを構成するルート局及び中間局たちで始まり、そのサーバ証明書で終了する。この

	サーバ認証のためのプライベート鍵は usePrivateKeyでセットされる。
void **usePrivateKey**(String keyFile, {String password})	あるサーバ証明書またはクライアント証明書のためのプライベート鍵をセットする。この SecurityContextを使うセキュアな接続は、署名及びメッセージの復号のためにサーバまたはクライアント証明書を持ったこの鍵を使用する。 keyFileはパスワードで暗号化された暗号化されたプライベート鍵を含むPEMファイルである。暗号化されていないファイルも使用可能であるが、それは一般的でない。

第21章 WebSocketサーバ (WebSocket Servers)

DartはWebSocket対応のAPIを備えているので、この章ではWebSocketベースのサーバを簡単に紹介する。但しこれも現時点では開発段階なので、変更の可能性があるので注意されたい。

WebSocketは単一のTCP接続上での完全双方向全二重通信を行う為の技術である。TCP接続が常にクライアントとの間で保持されるので、リアルタイムの送受信が必要なアプリケーションに適している。クライアント(即ちブラウザ)でのWebSocket APIはW3Cが標準化を行っており、通信プロトコルはIETFがRFC 6455として標準化されている。RFC 6455は既に日本語に翻訳されたものもあるので、それを参照していただきたい。

WebSocketはブラウザとウェブ・サーバが実装するよう設計されているが、その他のクライアントとサーバのアプリケーションでも使用可能である。ブラウザとサーバ間の通信がこれまでのHTTPの要求/応答のパラダイムに制約されず、より密に双方が関わりあえる為、チャットなどのライブのコンテントやリアル・タイムのゲームなどへの適用が容易になる。

WebSocketは接続開始の手段としてHTTPに類似した手順を使うので、通常のHTTPのTCPポート番号80(あるいはTLS接続の場合443)が使われており、ファイアウォール通過の制約が少ない。

WebSocketのもう一つの利点として、クライアントとのTCP接続が維持されることがある。TCPでは、サーバから見ると自分のポート番号、相手のポート番号、そして相手のIPアドレスのセットで接続が管理されている。従ってその接続をクライアント識別に使えるので、HTTPサーバのようなクッキーなどによるセッション管理は不要である。即ちWebScket接続がセッションそのものだと考えることができる。

WebSocketは現在殆どのブラウザが多少の差はあるものの実装している。特にセキュア・バージョンはFirefox 6(MozWebSocketという名前がついている)、Google Chrome 14及びInternet Explorer 10(developer preview)で採用されている。

21.1節 WebSocketプロトコルの概要

接続の確立

WebSocket接続開始に当たっては、クライアントが先ず接続要求を行い、ブラウザがこれに答える形式になっている。

クライアント要求例:

```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
```

```
Origin: http://example.com
```

サーバ応答例:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

これら、特に要求行とステータス行はHTTPのメッセージと似ている。

クライアントはbase64エンコードされたSec-WebSocket-Key（キー）を渡す。応答に際しては、258EAFA5-E914-47DA-95CA-C5AB0DC85B11というマジック文字列（GUI: Globally Unique Identifier）がこのキーに追加される。追加された文字列は次にSHA-1でハッシュ化（160ビット）され、base64エンコードされる。その結果がSec-WebSocket-Acceptヘッダの値になっている。

一旦WebSocket接続が確立されたら、WebSocketデータ・フレームはクライアントとサーバ間で全二重で送信される。すなわち双方とも相手を待つことなく勝手に同時に相手に送信できる。

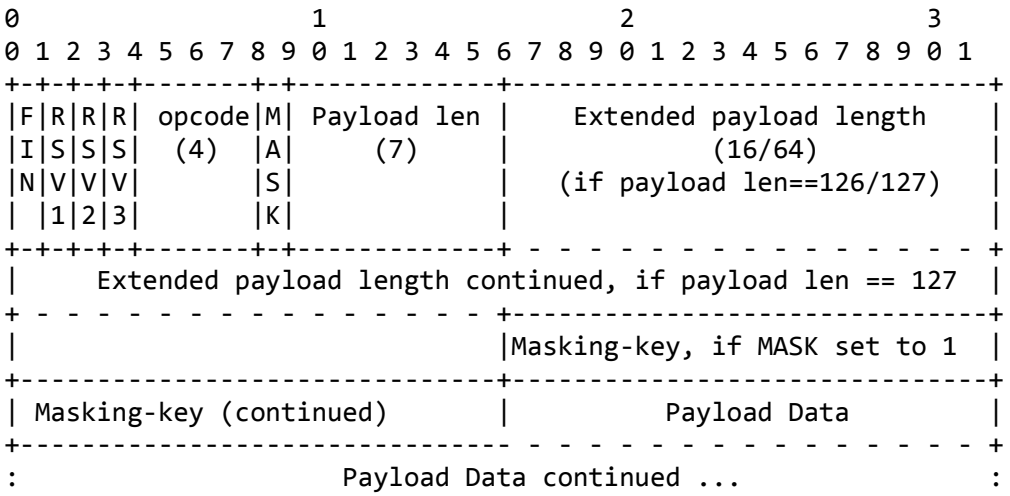
接続のクローズ

クローズはクライアントとサーバのどちらからでもその手続きを開始できる。クローズする側は先ず特定の制御シーケンスを含んだデータ・フレームを送信する。そのフレームを受けとった側はその応答としてCloseフレームを送信する。その制御フレームを受け取った最初の側がその接続（TCP接続を含めて）をクローズする。

これらの手順はAPIの中で行われるので、プログラマはその為のAPIのみを理解しておれば良い。

WebSocketフレーム

WebSocketフレームは次のような簡単な構成をとる:



```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

- FIN: 1ビットの情報で、これはあるメッセージの最後の断片であることを示す
- opcode: 4ビットの情報で、ペイロードのデータの解釈を指定する:
  - %x0: 継続フレームである
  - %x1: テキスト(UTF-8エンコード)のフレームである
  - %x2: バイナリ・データのフレームである
  - %x3-7: 予約
  - %x8: 接続のクローズ
  - %x9: ping
  - %xA: pong
  - %xB-F: 予約
- Mask: 1ビットの情報で、クライアント側からのペイロードのデータがMasking-keyで指定した32ビット長のキーでマスク解除できるようマスクされていることを示す。クライアントからサーバへ送信される総てのフレームでこのビットは1にセットされる。Masking-keyはクライアントがランダムに選択した値である
- Payload length: 7、7+16、または7+64ビット長の情報で、ペイロードのバイト長を示す。
- Payload Data: これは拡張データ(Extension data)とアプリケーション・データ(Application data)をあわせたものである。

Masking-keyはPayload data長に影響を与えない。送信されるデータのi番目のバイトは、オリジナルのデータのi番目のバイトとMasking-keyのi modulo 4番目のバイトとの排他的論理和をとったものである。Masking-keyを毎回変化させることで、悪意を持ったハッカたちがネットワーク上のデータを予測できないようにしている。

なおW3CのWebSocket API仕様は日本語に翻訳したものがあるので参考にされたい。

21.2節　　基本的なWebSocketサーバ

次のコードは、Dartチームのメンバであるの Seth Laddがそのブログに掲載した記事に示されている簡単なエコー・サーバを新しいAPIに対応するよう変更したものである。このアプリケーションはGithubからダウンロードできる。この資料の最後の「本資料に含まれているプログラムのダウンロード」の章を参考にして、自分のIDEから\dart_code_samples-master\apps\WebSocketEchoのフォルダを開くと良い。

WebSocketEcho.dart

```
// Sample Dart WebSocket echo server
// Source : http://blog.sethladd.com/2012/04/dart-server-supports-web-
sockets.html#disqus_thread
// you can connect to ws://localhost:8000/ws
// Feb. 2013, revised to incorporate redesigned dart:io v2 library

import 'dart:io';

void main() {
  HttpServer.bind('127.0.0.1', 8000)
  .then((HttpServer server) {
    server
      .where((request) => request.uri.path == '/ws')
      .transform(new WebSocketTransformer())
```

```
      .listen((WebSocket ws){
        wsHandler(ws);
      });
    print("Echo server started");
  });
}

wsHandler(WebSocket ws) {
  print('new connection : ${ws.hashCode}');
  ws.listen((message) {
      print('message is ${message}');
      ws.add('Echo: ${message}');
    }, onDone: (() {
      print('connection ${ws.hashCode} closed with ${ws.closeCode} for ${ws.closeReason}');
    })
  );
}
```

このコードが前記APIの基本的な使い方を良く説明している。

1. HttpServer.bindでIPアドレスとTCPポート番号を指定してHTTPサーバを用意する。
2. whereメソッドで要求パスが'/ws'である要求のみを取り込むストリームを用意する。本来なら request.uri.scheme == 'ws'で判断すべきであるが、localhostに対しては機能しない。
3. このストリームに対しtransformメソッドでウェブ・ソケットに対応する変換機を付加したストリームにする。
4. このストリームに対しlistenメソッドでその要素をイベントとして取り込むハンドラwsHandlerを用意する。
5. 従ってクライアントからのウェブ・ソケット接続に対応したWebSocketのオブジェクト毎にこのハンドラが呼ばれる。つまりこのハンドラは複数のWebSocketオブジェクトが共有する。
6. WebSocketオブジェクト、即ちクライアントを識別するひとつの手段はこのオブジェクトのハッシュ・コードを使うことである。
7. ws.listenメソッドでWebSocketオブジェクトからのメッセージ・イベントを受け付けるハンドラを用意する。
8. ws.addメソッドはクライアントにデータを送り返す。ここではエコーバックなので、受信したメッセージに'Echo : 'という文字列を付加して送り返している。
9. onDoneで完了のイベントが到来したときは、ウェブ・ソケット接続が切れたことを意味する。

クライアント側のHTMLテキスト

クライアント側のHTMLテキストに関してはWebSocket.orgの解説が参考になろう。WebSocket.orgではEcho Testというエコー・サーバのテスト用のページが用意されている。これを参考にすればチャットなどのページも比較的容易に作成できるようになる。

WebSocketEchoサーバの実行

下図はこのサーバを実行させ、上記のエコー・サーバのテスト・ページからこのサーバにアクセスした例である:

1. 自分のIDEからこのWebSocketEcho.dartを実行させる

2. エコー・サーバのテスト用のページをChromeブラウザで開き、This browser supports WebSocketと表示されていることを確認する
3. Locationにws:localhost:8000/wsとこのサーバを指定してやる
4. Connectボタンをクリックして、WebSocket接続を実行させる
5. LogにはCONNECTEDと表示される
6. 次に適当なメッセージをMessageテキスト・エリアに書き込む
7. Sendボタンをクリックすると、送信したメッセージ、及びエコー・サーバが返したメッセージが表示される
8. Disconnectボタンをクリックすると、接続がクローズされ、DISCONNECTEDとその状態が表示される

自分のIDE上のコンソールには、次のようなメッセージが表示される。

```
new connection : 614484817
message is 本日は曇天なり
connection 614484817 closed with 1005 for
```

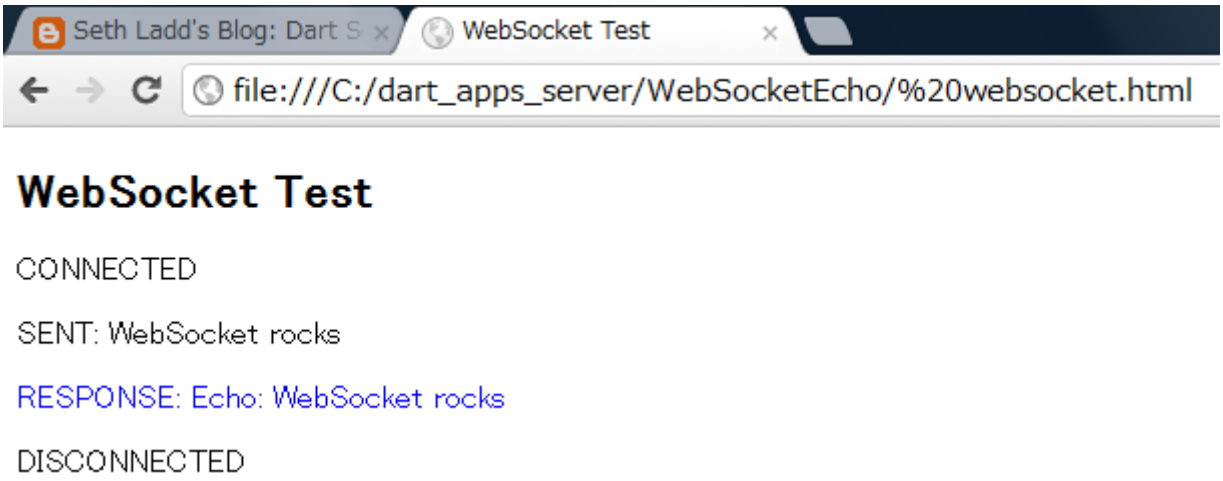
エコー・サーバのテスト用のページには次に示すようなプログラマが自分で作成できる簡単なJavaScriptベースのHTMLテキストが含まれている。

websocket.html

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>

<script language="javascript" type="text/javascript">
var wsUri = "ws://localhost:8000/ws";
var output;

function init()
{
output = document.getElementById("output");
testWebSocket();
}

function testWebSocket()
{
websocket = new WebSocket(wsUri);
websocket.onopen = function(evt) { onOpen(evt) };
websocket.onclose = function(evt) { onClose(evt) };
websocket.onmessage = function(evt) { onMessage(evt) };
websocket.onerror = function(evt) { onError(evt) };
}

function onOpen(evt)
{
```

```
writeToScreen("CONNECTED");
doSend("WebSocket rocks");
}

function onClose(evt)
{
writeToScreen("DISCONNECTED");
}

function onMessage(evt)
{
writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
websocket.close();
}

function onError(evt)
{
writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}

function doSend(message)
{
writeToScreen("SENT: " + message);
websocket.send(message);
}

function writeToScreen(message)
{
var pre = document.createElement("p");
pre.style.wordWrap = "break-word";
pre.innerHTML = message; output.appendChild(pre);
}

window.addEventListener("load", init, false);
</script>

<h2>WebSocket Test</h2>
<div id="output"></div>
</html>
```

このプログラムは自動で接続、送信、受信、解放を行う。下図のようにこのテキストを開くと、エコー・サーバにアクセスして、その経過が表示される。



Dart Editorのコンソールには次のように表示される:

```
new connection : 693238151
message is WebSocket rocks
connection 693238151 closed with 1005 for
```

プロキシによるネットワーク・レベルでの確認

いつものように、前述のwebsocket.htmlからプロキシ経由でWebSocketEcho.dartサーバをアクセスした際のTCPレベルでのデータのやり取りを調べてみよう。相変わらずサーバからのデータは細かく分断されるというバグは修正されていないので、それをつないで見やすくしたものを以下に示す:

```
ポート 12345で待機中
18042 [NioProcessor-1] INFO MINA_proxy.AbstractProxyIoHandler - GET /ws HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:12345
Origin: null
Sec-WebSocket-Key: 44P/7ZV2EJ6F0i6ABoyqUQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame


18043 [NioProcessor-6] INFO MINA_proxy.AbstractProxyIoHandler - HTTP/1.1 101 Switching
Protocols
upgrade: websocket
connection: Upgrade
content-length: 0
sec-websocket-accept: iGC1iPNmUjXsXCyGZZblwUZqHfM=


??+###|FfHD@o~_#vtHHw
?#Echo: WebSocket rocks
?????≫
?
```

最初の10行からなるかたまりはブラウザが送信したWebSocket接続要求である。これは仕様書通りである。但しoriginヘッダはnullになっている。

次の7行からなるかたまりは、サーバが送信した応答である。

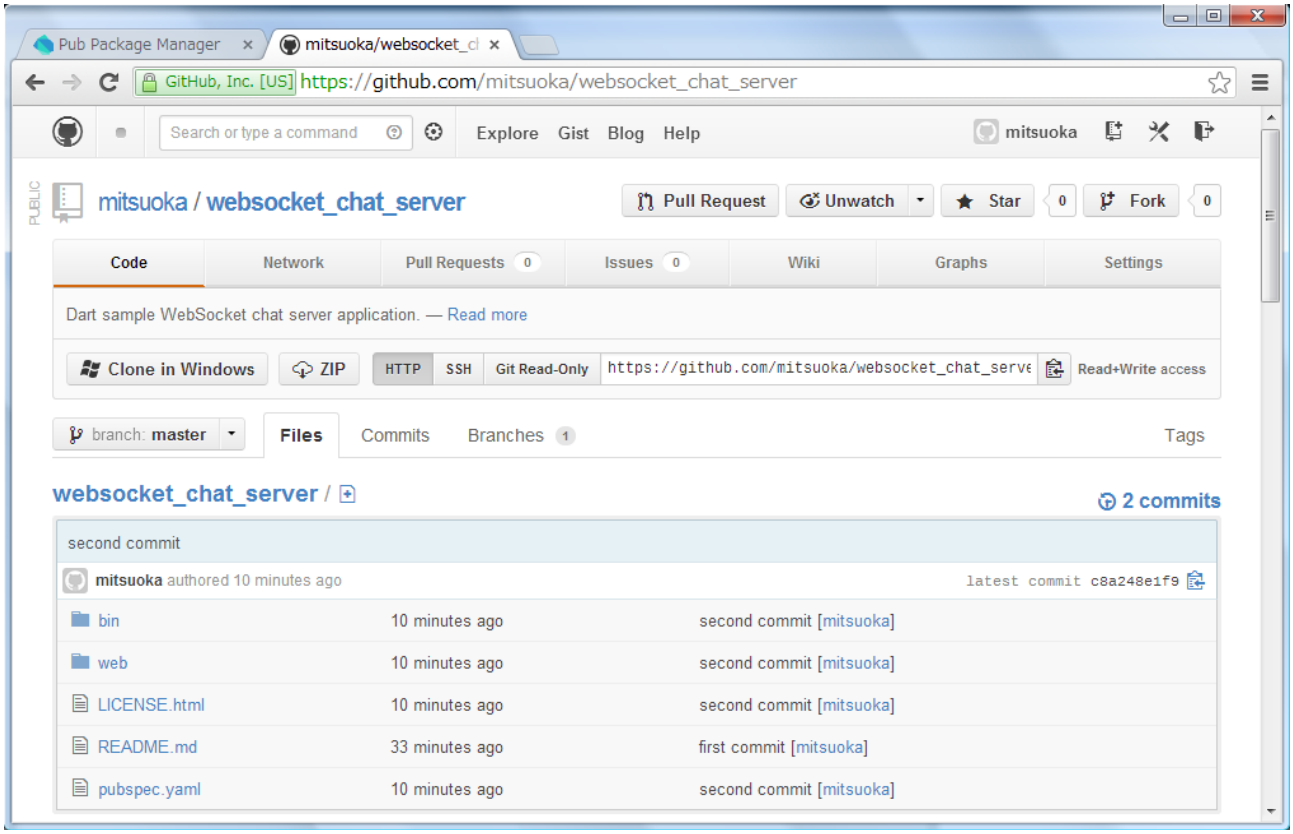
次の4行はWebSocketフレームであり、またブラウザからのデータはマスクされているので、テキストとしては読めない。最初の行はブラウザからのWebSocket rocksというテキスト、次の行はサーバからのEcho: WebSocket rocksというテキストを含んだフレームである。次の行はブラウザからの接続開放要求である。

21.3節　　チャット・サーバ

Echoサーバのアプリケーションを参考にすれば、チャット・サーバは容易に開発できよう。ここではその簡単な見本を示すことにする。

このアプリケーションはサーバのWebSocketChatServer.dart、Dartで書かれたDartium用のWebSocketChatClient.dart、WebSocketChatClient.html及びdart.jsブートストラップ、及びJavaScriptを使ったクライアントの為のWebSocketChat.html（これはテスト用）、及びファイル転送のために使うMIMEタイプのライブラリであるmimeType.dartというファイルたちで構成されている。

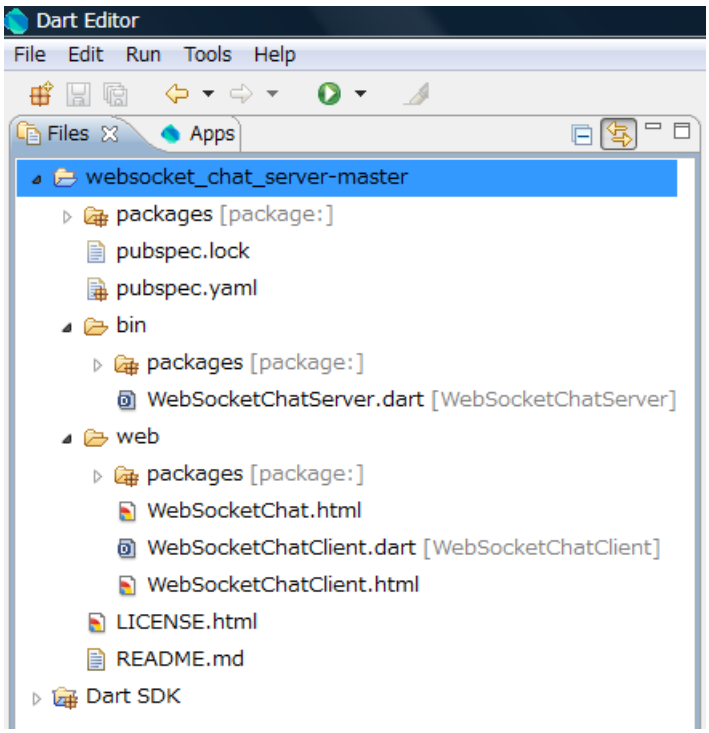
1. Githubで公開したこのアプリケーションは次のアドレスにある:
   https://github.com/mitsuoka/websocket_chat_server
   これをChrome等のブラウザでアクセスすると次のような画面が表示される:



2. この画面のZIPと表示されたダウンロードの個所をクリックするとこのアプリケーション全体がZIP圧縮された形式のwebsocket_chat_server-master.zipというファイルが取得できる。これを適当な解凍ツールで展開する。

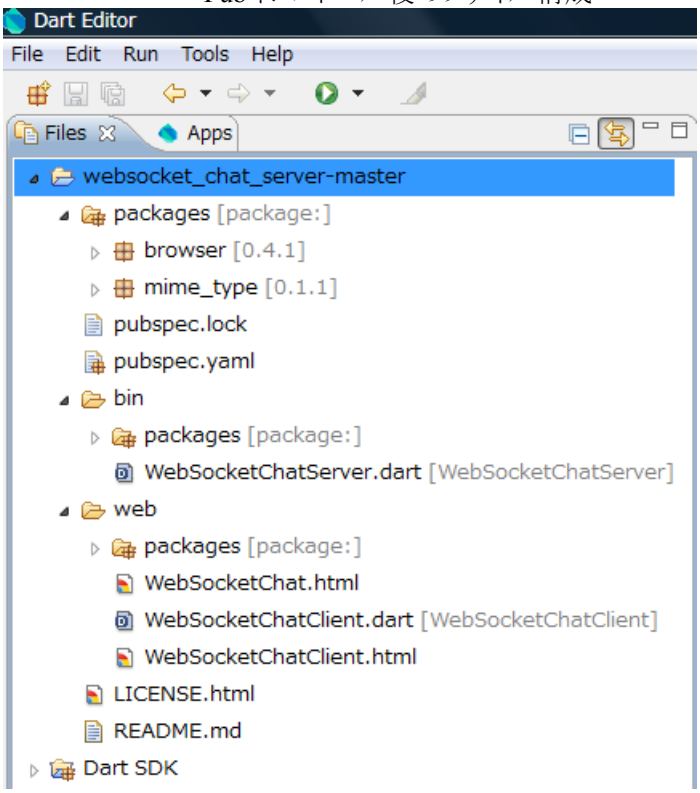
3. 自分のIDEでFile -> Open、そしてこのwebsocket_chat_server-masterを選択する。

Dart Editor上でのファイルの配置



4. 自分のIDE上でpubspec.yamlを選択、右クリックしてPub : Get Dependenciesを選択するとpub getが実行される:

Pubインストール後のファイル構成



5. WebSocketChatServer.dartを実行させる。コンソールには次のようなメッセージが表示される:

```
2013-03-09 22:02:01.791 - Serving Chat on 127.0.0.1:8080.
```

6. このサービスのクライアント側の画面であるWebSocketChatClient.htmlまたはWebSocketChat.htmlをブラ

ウザ（**Dartium、ChromeまたはIEの11版などを**）から次のようにアドレスを指定してHTTPサーバから取得して開く:
http://localhost:8080/chat

7. サーバはDartiumからの要求なのかそれともChromeからの要求なのかを識別して、各々に適したクライアント用のHTMLファイルを返す。
Dartiumが使うファイル　：　WebSocketChatClient.html、dart.js及びWebSocketChatClient.dart
Chromeが使うファイル　：　WebSocketChat.html

8. WebSocket接続を開始するには、ユーザ名を入力してJoinボタンをクリックする。

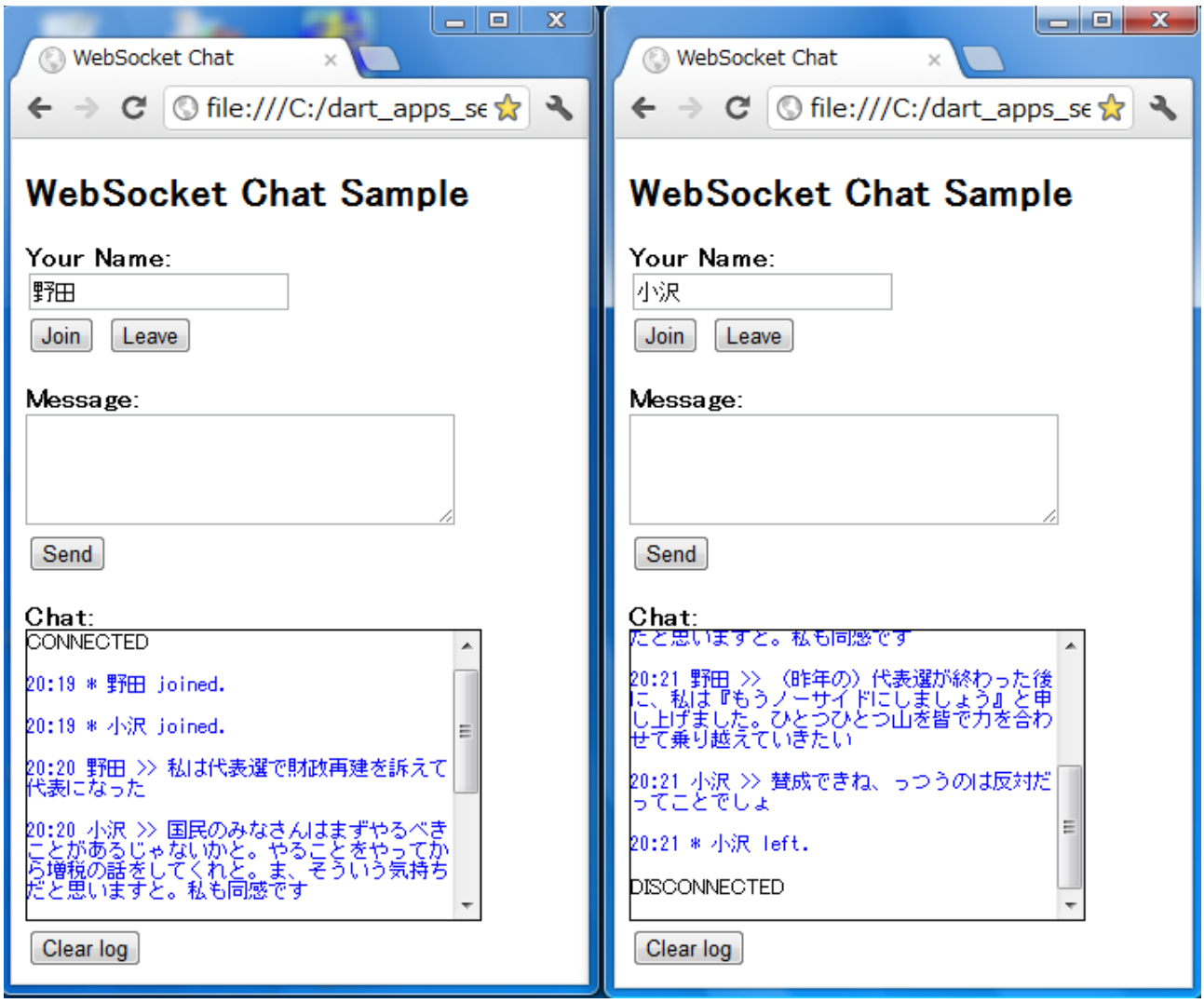
9. メッセージを送信するには、メッセージを入力してSendボタンをクリックする。

10. 接続を解放するには、Leaveボタンをクリックする。

複数のブラウザのインスタンス（あるいはタグ）から同時にこのサーバをアクセスして、互いに干渉することなくこのアプリケーションが動作することを確認する。

2つのブラウザのインスタンスからのアクセス例

下図は、スクリーン上に2つのブラウザを立ち上げ、左が野田、右が小沢氏に見立てたものである。この画面で判るように、同じサーバに対し2つのWebSocket接続がともに確立され、またそれぞれが独立してサーバと交信している。



そのシナリオは：

1. 最初に野田氏、次に小沢氏がこのチャットに参加している
2. 野田氏が最初に「私は代表選で財政再建を訴えて代表になった」ときり出している
3. 小沢氏は最後に「賛成できね、っつうのは反対だってことでしょ」と述べる
4. 結局物別れになり、20時21分に小沢氏が席を立っている

Dart Editorのコンソールには次のようなログが出力される：

```
2012-06-20 20:17:45.287 - Chat server started on ws://127.0.0.1:8000/chat.
2012-06-20 20:19:43.611 - new connection 1 (active connections : 1)
2012-06-20 20:19:43.611 - Received message on connection 1: userName=野田
2012-06-20 20:19:49.197 - new connection 2 (active connections : 2)
2012-06-20 20:19:49.197 - Received message on connection 2: userName=小沢
2012-06-20 20:20:10.508 - Received message on connection 1: 私は代表選で財政再建を訴えて代表になった
2012-06-20 20:20:37.933 - Received message on connection 2: 国民のみなさんはまずやるべきことがあるじゃないかと。やることをやってから増税の話をしてくれと。ま、そういう気持ちだと思いますと。私も同感です
2012-06-20 20:21:16.388 - Received message on connection 1: （昨年の）代表選が終わった後に、私は『もうノーサイドにしましょう』と申し上げました。ひとつひとつ山を皆で力を合わせて乗り越えていきたい
2012-06-20 20:21:38.183 - Received message on connection 2: 賛成できね、っつうのは反対だってことでしょ
2012-06-20 20:21:51.720 - closed connection 2 with null for null
```

WebSocketConnectionオブジェクトは野田氏が参加した時点で1個、次に小沢氏が参加した時点で2個になって

いる。

チャット・サーバのプログラムのポイント

まずこのサーバのmain関数のコードを以下に示す。main()ではHTTPとWebSocketの2つのプロトコルに対応する必要がある。何故ならチャットの為の画面（HTMLテキストとDartiumにはdartコード）をクライアントにHTTP応答として送り返さねばならないからである。HTTPサービスとWebSocketサービスをIPアドレスは同じにしてポート番号を違わせて区別することも可能だが、ここでは同じアドレス（実験の為にここではループバック・アドレス）とポート（8080）を使う方式を紹介する：

WebSocketChatServer.dartのmain()

```
001 void main() {
002   WebSocketHandler webSocketHandler = new WebSocketHandler();
003   HttpRequestHandler httpRequestHandler = new HttpRequestHandler();
004
005   HttpServer.bind(HOST, PORT)
006   .then((HttpServer server) {
007     var sc = new StreamController();
008     sc.stream.transform(new WebSocketTransformer())
009       .listen((WebSocket ws){
010         webSocketHandler.wsHandler(ws);
011       });
012
013     server.listen((HttpRequest request) {
014       if (request.uri.path == '/Chat') {
015         sc.add(request);
016       } else if (request.uri.path.startsWith('/chat')) {
017         httpRequestHandler.requestHandler(request);
018       } else {
019         new NotFoundHandler().onRequest(request, request.response);
020       }
021     });
022   });
023
024   print('${new DateTime.now().toString()} - Serving Chat on ${HOST}:${PORT}.');
025 }
```

- 005行目でHOST及びPORTを持ったHttpServerを用意している。
- 007-011行では、当該要求がWebSocket要求であったときにそれを処理しWebSocket接続をとるWebSocketTransformerを付加し、またそのイベントであるWebSocketオブジェクトをハンドラに渡すStreamControllerを用意している。
- 013行目ではHttpServerからのHttpRequsetオブジェクトの到来を待っている。
- もし/Chatという要求パスを持ったHttp要求だったときは、その要求はWebSocketの要求であるので、これはStreamControllerのscに渡している。
- もし/chatで始まる要求パスのときは、それは通常のHTTP要求なので、必要なファイルをクライアントに送信するハンドラ（ファイル・ハンドラ）にその要求を渡す。
- チャット・サービスはWebSocketHandlerクラス、HTTP要求ハンドラはHttpRequestHandlerクラスで記述されている。HttpRequestHandlerはファイル・サーバを参考にされたい。

なお参考のためにポートで2つのプロトコルを区分する場合のコードを以下に示す。

ポート番号で区別する場合のmain()

```
001 void main() {
002   users = new Map<String, WebSocket>();
003
004   WebSocketHandler webSocketHandler = new WebSocketHandler();
005   HttpRequestHandler httpRequestHandler = new HttpRequestHandler();
006
007   HttpServer.bind(HOST, WS_PORT)
008   .then((HttpServer server) {
009     server
010       .where((request) => request.uri.path == '/chat')
011       .transform(new WebSocketTransformer())
012       .listen((WebSocket ws){
013         webSocketHandler.wsHandler(ws);
014       });
015   });
016   print('Serving chat on : ${HOST}:${WS_PORT}.');
017   HttpServer.bind(HOST, HTTP_PORT)
018   .then((HttpServer server) {
019     server
020       .where((request) => request.uri.path.startsWith('/chat'))
021       .listen((HttpRequest request) {
022         httpRequestHandler.requestHandler(request);
023        });
024   });
025   print('Serving ChatPage request on : ${HOST}:${HTTP_PORT}.');
026 }
```

- ここではWebSocketプロトコルによるチャット・サービスを行うサーバ（ws://localhost:8000/chat）と、このサービスに必要なクライアント側のプログラムをクライアントに渡すHTTPプロトコルによるファイル・サーバ(http://localhost:8080/chat)のふたつを起動させている。

ウェブソケットのハンドラは次のようになっている:

WebSocketChatServer.dartのウェブソケットのハンドラ

```
001 // handle WebSocket events
002 class WebSocketHandler {
003
004   Map<String, WebSocket> users = {}; // Map of current users
005
006   wsHandler(WebSocket ws) {
007     if (LOG_REQUESTS) {
008       log('${new DateTime.now().toString()} - New connection ${ws.hashCode} '
009         '(active connections : ${users.length + 1})');
010     }
011     ws.listen((message) {
012       processMessage(ws, message);
013       } ,
014       onDone:(){
015         processClosed(ws);
016       }
017     );
018   }
019
020   processMessage(WebSocket ws, String receivedMessage) {
021     try {
022       String sendMessage = '';
023       String userName;
024       userName = getUserName(ws);
025       if (LOG_REQUESTS) {
026         log('${new DateTime.now().toString()} - Received message on connection'
027           ' ${ws.hashCode}: $receivedMessage');
028       }
029       if (userName != null) {
030         sendMessage = '${timeStamp()} $userName >> $receivedMessage';
031       } else if (receivedMessage.startsWith("userName=")) {
032         userName = receivedMessage.substring(9);
```

```
033         if (users[userName] != null) {
034           sendMessage = 'Note : $userName already exists in this chat room. '
035             'Previous connection was deleted.\n';
036           if (LOG_REQUESTS) {
037             log('${new DateTime.now().toString()} - Duplicated name, closed previous '
038             'connection ${users[userName].hashCode} (active connections : ${users.length})');
039           }
040           users[userName].add(preFormat('$userName has joind using another connection!'));
041           users[userName].close();  //  close the previous connection
042         }
043         users[userName] = ws;
044         sendMessage = '${sendMessage}${timeStamp()} * $userName joined.';
045       }
046       sendAll(sendMessage);
047     } on Exception catch (err) {
048       print('${new DateTime.now().toString()} - Exception - ${err.toString()}');
049     }
050   }
051
052   processClosed(WebSocket ws){
053     try {
054       String userName = getUserName(ws);
055       if (userName != null) {
056         String sendMessage = '${timeStamp()} * $userName left.';
057         users.remove(userName);
058         sendAll(sendMessage);
059         if (LOG_REQUESTS) {
060           log('${new DateTime.now().toString()} - Closed connection '
061             '${ws.hashCode} with ${ws.closeCode} for ${ws.closeReason}'
062             '(active connections : ${users.length})');
063         }
064       }
065     } on Exception catch (err) {
066       print('${new DateTime.now().toString()} Exception - ${err.toString()}');
067     }
068   }
```

- wsHandler(WebSocket ws)メソッドではWebSocketのオブジェクトwsはStreamを実装しているので、このストリームからのmessageとclosedのイベントを受け取りそれらのイベントの処理メソッドprocessMessage及びprocessClosedに渡している。

- サーバは例外が発生してもサービスを停止させてはいけない。従ってこれらのメソッド内ではtry文を使って例外をきちんと捕捉することが必須である。

- チャット・サーバの場合は、現在参加しているユーザを何時も管理しなければならない。ここではユーザ名をキー、WebSocketオブジェクトを値としたMapであるusersというオブジェクトでこれを管理している。

- クライアントは接続を開始すると同時に、ユーザ名をサーバに送る。サーバはこのユーザ名と接続のペアをusersに登録する。

- sendAllという関数は、現在登録されている総てのユーザに指定したテキストを送信する。その際、送信するテキストをpreFormatという関数に通している。これは例えば
というテキストをブラウザが改行と解釈したり、逆に改行を無視したり、連続したスペースを1個のみしか表示したりしないようにするものである。この作業はクライアント側で行っても良いが、ここではサーバ側で実施している。これにより、改行入りのテキストを送信しても、クライアント側では正しく表示される。

- 最初にあるユーザが接続を開始したときには、025行目に示すようにuserName=という形式でユーザ名が送られてくるので、これを識別する必要がある。間違って既に参加済みのユーザがuserName=というテキストを送信しても、それは新たな参加とは見做されない。

- 033行目にあるように、既に参加済みのユーザ名と同じユーザ名で接続してきた場合（例えばクライアントがソケット切断しないまま別の画面に移り、再びこのチャットに参加してきた場合）は、警告を出してそ

の接続削除し、新しいWebSocketオブジェクトとuserNameを登録する。

- logという関数は、プログラマが使っているロガーに適合させる。ここではコンソールに出力している。

JavaScriptベースのクライアント側のコード

ここではサーバのテストの為に、次のようなJavaScriptベースのHTMLテキストをクライアントとして使用している。このクライアントはChromeあるいはFirefoxで利用できるが、IEは対応していない。

WebSocketChat.html

```
001 <!DOCTYPE html>
002 <meta charset="utf-8" />
003 <title>WebSocket Chat</title>
004
005 <script language="javascript" type="text/javascript">
006
007   var wsUri = "ws://localhost:8000/chat";
008   var mode = "DISCONNECTED";
009
010   window.addEventListener("load", init, false);
011
012   function init() {
013     var consoleLog = document.getElementById("consoleLog");
014     var clearLogBut = document.getElementById("clearLogButton");
015     clearLogBut.onclick = clearLog;
016     var connectBut = document.getElementById("joinButton");
017     connectBut.onclick = doConnect;
018     var disconnectBut = document.getElementById("leaveButton");
019     disconnectBut.onclick = doDisconnect;
020     var sendBut = document.getElementById("sendButton");
021     sendBut.onclick = doSend;
022     var userName = document.getElementById("userName");
023     var sendMessage = document.getElementById("sendMessage");
024   }
025
026   function onOpen(evt) {
027     logToConsole("CONNECTED");
028     mode = "CONNECTED";
029     websocket.send("userName=" + userName.value);
030   }
031
032   function onClose(evt) {
033     logToConsole("DISCONNECTED");
034     mode = "DISCONNECTED";
035   }
036
037   function onMessage(evt) {
038     logToConsole('<span style="color: blue;">' + evt.data+'</span>');
039   }
040
041   function onError(evt) {
042     logToConsole('<span style="color: red;">ERROR:</span> ' + evt.data);
043     websocket.close();
044   }
045
046   function doConnect() {
047     if (mode == "CONNECTED") {
048       return;
049     }
050     if (window.MozWebSocket) {
051       logToConsole('<span style="color: red;"><strong>Info:</strong> This browser supports
WebSocket using the MozWebSocket constructor</span>');
052       window.WebSocket = window.MozWebSocket;
053     }
054     else if (!window.WebSocket) {
```

```
055       logToConsole('<span style="color: red;"><strong>Error:</strong> This browser does not have
support for WebSocket</span>');
056       return;
057     }
058     if (!userName.value) {
059       logToConsole('<span style="color: red;"><strong>Enter your name!</strong></span>');
060       return;
061     }
062     websocket = new WebSocket(wsUri);
063     websocket.onopen = function(evt) { onOpen(evt) };
064     websocket.onclose = function(evt) { onClose(evt) };
065     websocket.onmessage = function(evt) { onMessage(evt) };
066     websocket.onerror = function(evt) { onError(evt) };
067   }
068
069   function doDisconnect() {
070     if (mode == "CONNECTED") {
071     }
072     websocket.close();
073   }
074
075   function doSend() {
076     if (sendMessage.value != "" && mode == "CONNECTED") {
077       websocket.send(sendMessage.value);
078       sendMessage.value = "";
079     }
080   }
081
082   function clearLog() {
083     while (consoleLog.childNodes.length > 0) {
084       consoleLog.removeChild(consoleLog.lastChild);
085     }
086   }
087
088   function logToConsole(message) {
089     var pre = document.createElement("p");
090     pre.style.wordWrap = "break-word";
091     pre.innerHTML = message;
092     consoleLog.appendChild(pre);
093     while (consoleLog.childNodes.length > 50) {
094       consoleLog.removeChild(consoleLog.firstChild);
095     }
096     consoleLog.scrollTop = consoleLog.scrollHeight;
097   }
098
099 </script>
100
101 <h2>WebSocket Chat Sample</h2>
102 <div id="chat">
103   <div id="chat-access">
104     <strong>Your Name:</strong><br>
105     <input id="userName" cols="40">
106     <br>
107     <button id="joinButton">Join</button>
108     <button id="leaveButton">Leave</button>
109     <br>
110     <br>
111     <strong>Message:</strong><br>
112     <textarea rows="5" id="sendMessage" style="font-size:small; width:250px"></textarea>
113     <br>
114     <button id="sendButton">Send</button>
115     <br>
116     <br>
117   </div>
118   <div id="chat-log"> <strong>Chat:</strong>
119     <div id="consoleLog" style="font-size:small; width:270px; border:solid; border-width:1px;
height:172px; overflow-y:scroll"></div>
120     <button id="clearLogButton" style="position: relative; top: 3px;">Clear log</button>
121   </div>
122 </div>
123
124 </html>
```

- JavaScriptでWebSocket接続を行うには、62行目から示すようにURIを引数にして新しいWebSocketのインスタンスを生成する。
- その後接続完了、接続開放、メッセージ受信、及びエラー発生のイベント処理の為のハンドラをこのオブジェクトにセットする。
- チャットのログの為の"consoleLog”というIDの要素に対しては、88行目から示すlogToConsoleという関数を使用する。このテキスト領域は最大50メッセージを収容させ、それを超えたら最初のほうから削除してゆく。
- 100行目以降のHTMLテキストは、画面を見れば特に説明する必要はなかろう。

Dartベースのクライアント側のコード

Dartiumで実行させる為のDartベースのクライアントのDart部を以下に示す。HTML部はJavaScriptのそれと同じである。またその動作もJavaScriptベースのクライアントとまったく同じである。双方のコードを比較すると、DOMアクセスに対する両者の相違が判って参考になろう。

```
001 /*
002   Dart code sample : WebSocket chat client for Dartium
003     1. Save these files into a folder named WebSocketChat.
004     2. From Dart editor, File > Open Folder and select this WebSocketChat folder.
005     3. Run WebSocketChatServer.dart as server.
006     4. Run WebSocketChatClient.html from Dartium.
007     5. To establish WebSocket connection, enter your name and click 'join' button.
008     6. To chat, enter chat message and click 'send' button.
009     7. To close the connection, click 'leave' button
010   June 2012, by Cresc Corp.
011   Ref: www.cresc.co.jp/tech/java/Google_Dart/DartLanguageGuide.pdf (in Japanese)
012 */
013
014 #import('dart:html');
015
016 var wsUri = 'ws://localhost:8000/Chat';
017 var mode = 'DISCONNECTED';
018 WebSocket webSocket;
019 var userName;
020 var sendMessage;
021 var consoleLog;
022
023 void main() {
024   show('Dart WebSocket Chat Sample');
025   userName = document.query('#userName');
026   sendMessage = document.query('#sendMessage');
027   consoleLog = document.query('#consoleLog');
028   document.query('#clearLogButton').on.click.add((e) {clearLog();});
029   document.query('#joinButton').on.click.add((e) {doConnect();});
030   document.query('#leaveButton').on.click.add((e) {doDisconnect();});
031   document.query('#sendButton').on.click.add((e) {doSend();});
032 }
033
034 doConnect() {
035   if (mode == 'CONNECTED') {
036     return;
037   }
038   if (userName.value == '') {
039     logToConsole('<span style="color: red;"><strong>Enter your name!</strong></span>');
040     return;
041   }
042   webSocket = new WebSocket(wsUri);
043   webSocket.onOpen.listen(onOpen);
044   webSocket.onCclose.listen(onClose);
045   webSocket.onMessage.listen(onMessage);
046   webSocket.onError.listen(onError);
047 }
```

```
048
049 doDisconnect() {
050   if (mode == 'CONNECTED') {
051   }
052   webSocket.close();
053 }
054
055 doSend() {
056   if (sendMessage.value != '' && mode == 'CONNECTED') {
057     webSocket.send(sendMessage.value);
058     sendMessage.value = '';
059   }
060 }
061
062 clearLog() {
063   while (consoleLog.nodes.length > 0) {
064     consoleLog.nodes.removeLast();
065   }
066 }
067
068 onOpen(open) {
069   logToConsole('CONNECTED');
070   mode = 'CONNECTED';
071   webSocket.send('userName=${userName.value}');
072 }
073
074 onClose(close) {
075   logToConsole('DISCONNECTED');
076   mode = 'DISCONNECTED';
077 }
078
079 onMessage(message) {
080   logToConsole('<span style="color: blue;">${message.data}</span>');
081 }
082
083 onError(error) {
084   logToConsole('<span style="color: red;">ERROR:</span> ${error}');
085   webSocket.close();
086 }
087
088 logToConsole(message) {
089   var pre = document. $dom_createElement('p');
090   pre.style.wordWrap = 'break-word';
091   pre.innerHTML = message;
092   consoleLog.nodes.add(pre);
093   while (consoleLog.nodes.length > 50) {
094     consoleLog.$dom_removeChild(consoleLog.nodes[0]);
095   }
096   pre.scrollIntoView();
097 }
098
099 show(String message) {
100   document.query('#status').text = message;
101 }
```

WebSocket接続の記述は:

```
webSocket = new WebSocket(wsUri);
webSocket.onOpen.listen(onOpen);
webSocket.onClose.listen(onClose);
webSocket.onMessage.listen(onMessage);
webSocket.onError.listen(onError);
```

と、JavaScriptに比べるとより短い記述で済む。

またログ表示の関数は:

```
logToConsole(message) {
  var pre = document. $dom_createElement('p');
  pre.style.wordWrap = 'break-word';
  pre.innerHtml = message;
  consoleLog.nodes.add(pre);
  while (consoleLog.nodes.length > 50) {
    consoleLog.$dom_removeChild(consoleLog.nodes[0]);
  }
  pre.scrollIntoView();
}
```

と、DOMアクセスの書き方がやや異なってくる。

21.4節　　WebSocketの為のAPIの和訳

更に2013年2月のM3版でdart:ioに更なる変更がなされた。これは非同期処理の為のFutureとStreamの積極的な導入である。

WebSocketに関してはインターフェイスの簡素化がおこなわれるとともに、サーバ側とクライアント側がソケット接続に対し同じクラスが使われるようになった。サーバ側ではWebSocketの処理はストリーム・トランスフォーマとして組み入れられた。このトランスフォーマはHttpRequestのストリームをWebSocketのストリームに変換するものである。

更に2013年3月のAPI変更ではWebSocketからのイベントがメッセージのストリームとなった（List<int> 型またはString型）。クローズにイベントは onDoneとなり、クローズ・コードと理由句はWebSocketオブジェクトの属性となった。

dart:io.WebSocketTransformer

Abstract class **WebSocketTransformer**

HTTPまたはHTTPSからの各HttpRequestをWebSocketプロトコルにアップグレードしてHttpRequestのストリームをWebSocketのストリームに変換するストリーム変換器としてこのWebSocketTransformerが実装されている。

使用例:

```
server.transform(new WebSocketTransformer()).listen((webSocket) => ...);
```

あるいは、

```
server
    .where((request) => request.uri.scheme == "ws")
    .transform(new WebSocketTransformer()).listen((webSocket) => ...);
```

個の変換器はRFC6455で規定されたウェブ・ソケットを実装している。

実装	
StreamTransformer<HttpRequest, WebSocket>	
コンストラクタ	
factory **WebSocketTransformer**()	
メソッド	
abstract Stream<T> **bind**(Stream<S> stream)	（StreamTransformerから継承）

dart:io.WebSocket

Abstract Class **WebSocketConnection** 代替的なウェブ・ソケットのクライアント・インターフェイス。このインターフェイスは http://dev.w3.org/html5/websockets/で規定されているウェブ・ソケットの為のW3CブラウザAPIに対応している。	
実装	
Stream<Event>	
static属性	
const int **CLOSED**	
const int **CLOSING**	
const int **CONNECTING**	
const int **OPEN**	
staticメソッド	
Future<WebSocket> **connect**(String url, [protocols])	新しいウェブ・ソケット接続を作る。uriで指定するURIはwsまたはwssのスキームを使用しなければならない。protocols引数は該クライアントが使いたいサブプロトコルを指定するもので、StringまたはList<String>でなければならない。
属性	
final int **bufferedAmount**	送信の為に現在バッファリングされているバイト数を返す。
final int **closeCode**	該WebSocket接続がクローズしたときにセットされるクローズ・コード。クローズ・コードが得られないときはこの属性はnullとなる。
final String **closeReason**	該WebSocket接続がクローズしたときにセットされるクローズ理由句。クローズ理由が得られないときはこの属性はnullとなる。
final String **extensions**	このextensions属性は初期には空のStringである。ウェブ・ソケット接続が確立された後はこの文字列はこのサーバが使っている拡張(extension)たちを反映する。
final Future<T> **first**	（Streamから継承） 最初の要素を返す。 もし空のときはStateErrorをスローする。そうでないときはこのメソッドは this.elementAt(0)と等価である。

final bool **isBroadcast**	（Streamから継承） このストリームが放送ストリームかどうかを返す。
final Future<bool> **isEmpty**	（Streamから継承） このストリームが何ら要素を含んでいないかどうかを報告する。
final Future<T> **last**	（Streamから継承） 最後の要素を返す。 もし空のときはStateErrorをスローする。
final Future<int> **length**	（Streamから継承） このストリームの中の要素の数を数える。
final String **protocol**	このprotocol属性は初期は空の文字列である。ウェブ・ソケット接続が確立された後はこの値はサーバが選択したサブプロトコルがこの値となる。サブプロトコルのネゴシエーションがされていなければこの値はnullのままである。
final int **readyState**	現在の接続の状態を返す。
final Future<T> **single**	（Streamから継承） 単一の要素を返す。 空のときまたはひとつ以上の要素を持っているときはStateErrorをスローする。
メソッド	
abstract void **add**(data)	このウェブ・ソケット接続上でデータを送信する。このデータはStringまたはバイト列であるList<int>でなければならない。
abstract void **addError**(errorEvent)	（EventSinkから継承） asyncエラーを生成する
abstract Future **addStream**(Stream stream)	あるストリームからデータをウェブ・ソケット接続上で送信する。ストリームからの各データ・イベントは単一のWebSocketフレームとして送信される。streamからのデータはStringsまたはバイトを保持するList<int>sでなければならない。
Future<bool> **any**(bool test(T element))	（Streamから継承） このストリームが用意している要素のどれかをtestが受け付けるかどうかをチェックする。 その答えが判ったときにFutureを完了させる。もしこのストリームがエラーを報告したときは、このFutureはエラーを報告する。
Stream<T> **asBroadcastStream**()	（Streamから継承） これと同じイベントたちを作り出す複数受信のストリームを返す。もしこのストリームが単一受信のときは、複数の受信者が許される新しいストリームを返す。その最初の受信者が付加されたときにこれはこのストリームに受信し、最後の受信者がキャンセルされたときに再度受信解除される。 もしこのストリームが既に放送ストリームであるときは、これは手を加えられないで返される。
abstract void **close**([int code, String reason])	このウェブ・ソケット接続を閉じる。
Future<bool> **contains**(T match)	（Streamから継承） このストリームが用意した要素たちのなかでmatchが生じるかどうかをチェックする。

	この答えが判ったときにこのFutureを完了させる。このストリームがエラーを報告したときは、該Futureはそのエラーを報告する。
Stream<T> **distinct**([bool equals(T previous, T next)])	(Streamから継承) もし以前のデータ・イベントと同じのときはこれらのデータ・イベントをスキップする。 返されるストリームは、ふたつの連続したデータが決して同じで無いということを除いて、このストリームと同じイベントを作る。 用意されるequalsメソッドによって等しいかどうかが判断される。もしこれがオミットされているときは、最後に用意されたデータ要素には'=='演算子が使われる。
Future<T> **elementAt**(int index)	(Streamから継承) このストリームのindex番目のデータ・イベントの値を返す。 もしエラーが起きれば、このfutureはエラーで終了する。 もしこのストリームが閉じる前にindex要素たちより少ない数しか用意していないときは、エラーが報告される。
Future<bool> **every**(bool test(T element))	(Streamから継承) このストリームが用意している総ての要素をtestが受け付けるかどうかをチェックする。 この答えが判った時点でこのFutureを完了させる。もしこのストリームがエラーを報告するときは、このFutureはそのエラーを報告する。
Stream **expand**(Iterable convert(T value))	(Streamから継承) 各要素をゼロまたはそれ以上のイベントたちに変換する新しいストリームをこのストリームから生成する。 各到来イベントは新しいイベントたちのIterableに変換され、これらの新しいイベントたちの各々が次に順番に返されたストリームによって送信される。
Future<T> **firstMatching**(bool test(T value), {T defaultValue()})	(Streamから継承) testにマッチするこのストリームの最初の要素を見つける。 testがtrueを返すこのストリームの最初の要素で満たされたfutureを返す。 このストリームが完了する前にそのような要素が見つからず、またdefaultValue関数が用意されているときは、defaultValue呼び出しの結果がそのfutureの値となる。 エラーが発生したとき、あるいはこのストリームが一致する要素が見つからないで終了し、またdefaultValue関数が用意されていないときは、このfutureはエラーを受信する。
Stream<T> **handleError**(void handle(AsyncError error), {bool test(error)})	(Streamから継承) このストリームからの何らかのエラーを横取りするラッパーのストリームを生成する。 もしこのラッパ・ストリームがtestにマッチするエラーを送信するときは、それはhandle関数により横取りされる。 test(e)がtrueを返すと[AsyncError] [:e:]はtest関数によりマッチがとられる。testがオ

	ミットされているときは各エラーはマッチしていると見做される。 もしそのエラーが横取りされたときは、handle関数はそれに対してどうするかを判断できる。この関数は新しい（あるいは同じ）エラーを生起させたいときはスローできるし、あるいはこのストリームにこのエラーを忘れさせる為に単に戻ることができる。 あるエラーをデータ・イベントに変換したいときは、より一般的なStream.transformEvenを使って出力sinkへのデータ・イベントを書いて、このイベントを処理させる。
Future<T> **lastMatching**(bool test(T value), {T defaultValue()})	（Streamから継承） このストリームの中でtestにマッチする最後の要素を探す。 最後のマッチする要素を見つけることを除いてfirstMatchingとおなじ。このことはこのストリームが完了するまでこの結果は得られないことを意味する。
abstract StreamSubscription<T> **listen**(void onData(T event), {void onError(AsyncError error), void onDone(), bool unsubscribeOnError})	（Streamから継承） このストリームにひとつの受信を付加する。 このストリームからの各テータ・イベントにたいし、受信者たちのonDataハンドラが呼び出される。もしonDataがnullのときは何も起きない。 このストリームからのエラーにたいし、onErrorハンドラにはそのエラーを記述したAsyncErrorが与えられる。 このストリームがクローズしたときは、onDone ハンドラが呼び出される。 unsubscribeOnErrorがtrueのときは、最初のエラーが報告されたときにこの受信は終了する。デフォルト値はfalseである。
Stream **map**(convert(T event))	（Streamから継承） このストリームの各要素をconvert関数を使って新しい値に変換する新しいストリームを生成する。
Future<T> **max**([int compare(T a, T b)])	（Streamから継承） このストリームのなかの最大の要素を探す。 もしこのストリームが空のときはその結果はnullである。そうでないときは、その結果はこのストリームからの他のどの値よりも小さくないこのストリームからの値となる（Comparatorでなければならないcompareに従って）。 もしcompareがオミットされているときは、そのデフォルトはComparable.compareである。
Future<T> **min**([int compare(T a, T b)])	（Streamから継承） このストリームのなかの最小の要素を探す。 もしこのストリームが空のときはその結果はnullである。そうでないときは、その結果はこのストリームからの他のどの値よりも大さくないこのストリームからの値となる（Comparatorでなければならないcompareに従って）。 もしcompareがオミットされているときは、そのデフォルトはComparable.compareである。
Future	（Streamから継承）

pipe(StreamConsumer<T, dynamic> streamConsumer)	このストリームを用意されているStreamConsumerの入力に結び付ける。
Future **pipeInto**(StreamSink<T> sink, {void onError(AsyncError error), bool unsubscribeOnError})	（Streamから継承）
Future **reduce**(initialValue, combine(previous, T element))	（Streamから継承） combineを繰り返し適用することで値たちの並びを減らす。
Future<T> **singleMatching**(bool test(T value))	（Streamから継承） testにマッチするこのストリームのなかの最初の要素を探す。 このストリームの中でひとつ以上のマッチする要素が生じないときにエラーとなることを除いてlastMatchと似ている。
Stream<T> **skip**(int count)	（Streamから継承） このストリームからの最初のcount個数のデータ・イベントをスキップする。
Stream<T> **skipWhile**(bool test(T value))	（Streamから継承） testにマッチする間はこのストリームからのデータ・イベントをスキップする。 返されたストリームはエラーと完了のイベントを加工しないで渡す。 そのイベント・データに対しtestがtrueを返す最初のデータ・イベントから始まり、返されるこのストリームはこのストリームと同じイベントを持つことになる。
Stream<T> **take**(int count)	（Streamから継承） このストリームの最大n個の値を渡す。 このストリームの最初のn個のデータ・イベント、及び総てのエラー・イベントを返されるストリームに転送し、完了イベントで終了する。 このストリームがその完了前にcount値より少ない場合は、返されるストリームもそうする。
Stream<T> **takeWhile**(bool test(T value))	（Streamから継承） testが成功している間はデータ・イベントを転送する。 返されたストリームはそのイベントデータに対しtestがtrueを返す限りこのストリームと同じイベントを渡す。このストリームはthisストリームが完了した、あるいはthisストリームが最初にtestを受け付けない値を最初に渡したときに完了する。
Future<List<T>> **toList**()	（Streamから継承） このストリームのデータをListとして収集する。
Future<Set<T>> **toSet**()	（Streamから継承） このストリームのデータをSetとして収集する。
CodeStream **transform**(StreamTransformer<T, dynamic> streamTransformer)	（Streamから継承） このストリームを指定されたStreamTransformerの入力として連結する。 streamTransformer.bind自身の結果を返す。
Stream<T> **where**(bool test(T event))	（Streamから継承）

	何らかのデータ・イベントたちを破棄する新しいストリームをこのストリームから生成する。 新しいストリームはこのストリームと同じエラーと完了のイベントを送信するが、test を満足させるデータ・イベントのみを送信する。

第22章 ファイル・アップロード (HTTP File Upload Servers)

注意：現在Dartチームは筆者の指摘を受けhttp_serverライブラリの改定作業中であり、この章は近い将来改正される。

写真、ビデオなどのファイルをサーバにアップロードして、友人や親しい人たちとそれを共有するといったインターネットの使い方が普及しているなかで、ウェブ・アプリケーションにそのような機能を持たすことの要求が一般化してきている。ウェブ・サービスの世界では、クライアントとサーバ間でJSONファイルを交わすことが一般化してきている。Dartでは、最近用意されているhttp_serverというPubパッケージでmultipart/form-dataつきのHTTP要求メッセージを構文解析し利用できるようになっている。これらの新しいAPIにより、フォーム・データ送信（submission）が容易に処理できるようになった。

本章ではこのhttp_serverパッケージの概要を紹介し、続いてこれを使ったテスト・サーバを説明することとする。このテスト・サーバはfile_upload_testというGitリポジトリに登録してあるので、読者はこれをダウンロードし、自分で試すとともに、サーバ開発に利用することが可能である。

その前にファイル・アップロードの仕組みであるHTTPマルチパート要求について理解しておく必要がある。Multipart/form-dataというのはファイル、非ASCIIテキスト、及びバイナリ・データをサーバに送信する為のHTTPプロトコルのコンテント・タイプである。HTMLのFORMでこれを送信するにはRFC 1867「HTMLにおけるフォーム・ベースのアップロード」仕様書に規定されている記述を使用する。

22.1節　ブラウザが送信するマルチパート・データ

具体的にブラウザがファイルを含む幾つかのパートからなるデータをどのようにサーバに送信するかを簡単なHTMLファイル(file_upload_test.html)及びテスト・サーバ(test_server_1.dart)で確認してみよう。これらのコードはgithubからダウンロードしたfile_upload_testに含まれているので、自分のIDE上で確認されたい。

テスト用のHTMLファイルは次のような簡単なものである：

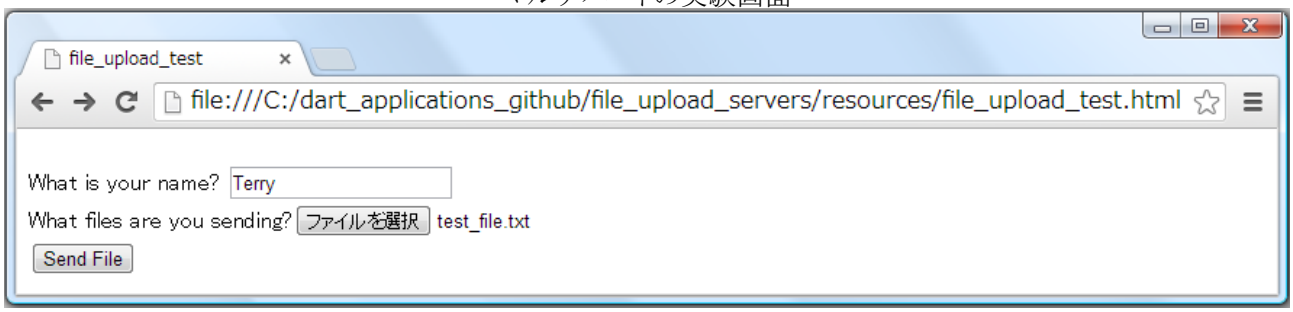
file_upload_test.html

```
<!DOCTYPE html>

<html>
  <head>
    <title>file_upload_test</title>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  </head>

  <body>
    <form action="http://localhost:8080/DumpHttpMultipart"
          enctype="multipart/form-data"
          method="POST"> <br>
      What is your name? <input type="text" name="submitter"> <br>
      What files are you sending?<input type="file" name="content"> <br>
      <input type="submit" value="Send File"">
    </form>
  </body>
</html>
```

このコードはRFC 1867の第6章の"Examples"に示されているものと殆ど同じである。formタグのなかのenctype="multipart/form-data"でマルチパートのフォーム・データとしてエンコードするよう指定している。

このファイルをブラウザで開くと、次のような画面が表示される:

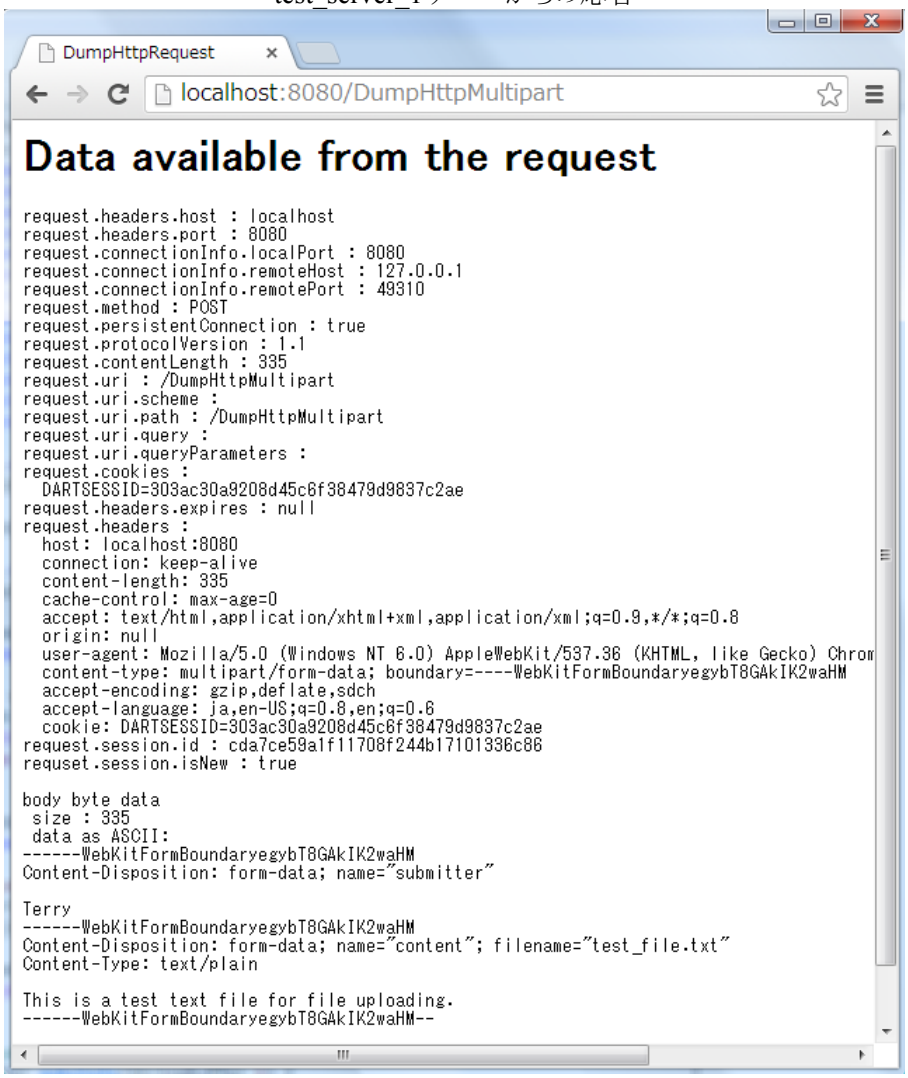
マルチパートの実験画面



この画面上で自分の名前(ここではTerry)を入力し、「ファイルを選択」ボタンを押して送信したいファイル(ここではtest_file.txt)を指定する。

この状態でtest_server_1.dartを起動し、Send Fileボタンをクリックすると、サーバは次のような応答を返す:

test_server_1サーバからの応答



この応答は「要求オブジェクト」の節で示したDumpHttpRequestの応答と同じであるが、ボディ部分のバイナリ・データを8ビットASCIIの列として表示しているところが異なっている。

要求ヘッダのなかにある

```
content-type: multipart/form-data; boundary=----WebKitFormBoundary1AfDiA2EF9BpaXuW
```

というヘッダがこの要求のボディ部はマルチパートのデータであって、各パートの区切り(バウンダリ)が----WebKitFormBoundary1AfDiA2EF9BpaXuWであることを示している。サーバはこのヘッダによりこの要求のボディ部を処理することになる。

ボディ部は次のように335バイトからなるデータである:

```
------WebKitFormBoundary1AfDiA2EF9BpaXuW
Content-Disposition: form-data; name="submitter"

Terry
------WebKitFormBoundary1AfDiA2EF9BpaXuW
Content-Disposition: form-data; name="content"; filename="test_file.txt"
Content-Type: text/plain

This is a test text file for file uploading.
------WebKitFormBoundary1AfDiA2EF9BpaXuW--
```

即ち3つのバウンダリで区切られた2つのパートで構成されており、最初のパートは"submitter"という名前のテキスト・データである。残りのパートは送信するファイルの情報である。このパートの名前は"content"であり、ファイル名は"test_file.txt"である。このファイルはテキスト・ファイルなので、Content-Type: text/plainという行が付加されている。ファイルの中身は"This is a test text file for file uploading."という文字列である。

より詳しく説明すると:

- 3つのバウンダリで2つのパートが挟まれている。ひとつは送信者名であり、もうひとつは送信ファイルのデータである。これらのパートはinputのnameによって識別される
- バウンダリの文字列にはその前に"--"が、また最後のバウンダリには前後に"--"が付けられている。バウンダリの文字列としては、パートたちのなかに同じ文字列が存在しないものが選択されている
- 送信ファイルのデータは、Content-Disposition:、Content-Type:のヘッダ行と、ファイルの中身で構成されている
- Content-Disposition:ヘッダはRFC 2183で規定されている。Content-Dispositionは本来、メールのMIME仕様に従うデータの提示的情報(presentational information)を伝えるに作られたヘッダであるが、HTTPメッセージの形式がMIMEに似ている為、HTML フォームからマルチパート・データをアップロードする際に使用されている。filename 属性を使うと、マルチパート・データとして転送されたデータのファイル名を提示することになる。サーバ側はこの提示されたファイル名を受け入れる必要は無く、自由にファイル名を決定する事ができるが、誤ってデータを上書きしてしまうことの無いよう注意が必要である。
- Content-Type:ヘッダは、このファイルはテキスト・ファイルであるので、text/plainとなっている。例えばDLLファイルだとContent-Type: application/octet-streamとなる。MIMEタイプの詳細は筆者のmime_typeライブラリなどを参照されたい。

22.2節　　http_serverパッケージ

Dartチームはhttp_serverというパッケージを用意し、マルチパートのボディ部のデータをより簡単に取り扱えるようにている。このパッケージはfile_upload_serversパッケージに既にインポートされているので、自分のIDE上でその内容を知ることができる。

このライブラリは現在次のような構成となっている:

```
library http_server;

import 'dart:async';
import 'dart:convert';
import 'dart:io';

import 'package:mime/mime.dart';
import "package:path/path.dart";

part 'src/http_body.dart';
part 'src/http_body_impl.dart';
part 'src/http_multipart_form_data.dart';
part 'src/http_multipart_form_data_impl.dart';
part 'src/virtual_directory.dart';
part 'src/virtual_host.dart';
```

更にここでインポートされているmimeというライブラリは:

```
library mime;

import 'dart:async';
import 'dart:typed_data';

part 'src/mime_type.dart';
part 'src/extension_map.dart';
part 'src/magic_number.dart';
part 'src/mime_multipart_transformer.dart';
```

となっている。

赤で示したファイルたちがファイル・アップロードのサーバ開発に重要なので、一応その概要を理解する必要がある。

HttpBodyHandler

HttpBodyHandlerというヘルパ・クラスは、単にマルチパートのフォーム・データを含むボディ部を含むPOST要求を取り扱うだけでなく、他のフォーム・データも取り扱うに、より一般化されたものとなっている。

以下はhttp_body.dartの最初のコメント部分の翻訳である:

> HttpBodyHandlerは使いやすいHttpBodyオブジェクトの形でHTTPメッセージ・データを処理・収集するためのヘルパ・クラスである。ボディ部分は`Content-Type`ヘッダ・フィールドの基づいて構文解析される。ボディ部すべてが読みだされ構文解析された時点でこのボディのコンテントが取得できるようになる。このクラスはサーバ要求とクライアント応答の双方の処理に使える。
>
> このクラスは以下のコンテント・タイプを認識する:
>
> - text/*

- application/json
- application/x-www-form-urlencoded
- multipart/form-data

`text/*`のコンテント・タイプに対しては、該ボディ部は文字列にでコードされる。content typeヘッダの'charset'パラメタがこのエンコーディングを指定している。'charset'パラメタが存在しないときは、デフォルトのISO-8859-1がエンコーディングに使われる（これが問題で、Dartチームに指摘してある）。

`application/json`のコンテント・タイプに対しては、該ボディは文字列にデコードされ、次にJSONとして構文解析される。結果としてのbodyはMapとなる。content typeヘッダの'charset'パラメタがこのデコードのためのエンコーディングを指定している。'charset'パラメタが存在しないときは、デフォルトのUTF-8がエンコーディングに使われる。

`application/x-www-form-urlencoded`のコンテント・タイプの場合は、該ボディ部はURLエンコードされたクエリ文字列であり、この場合はクエリ文字列のルールに従ってクエリたちに分割される。結果としてのbodyはMap<String, String>型である。もしクエリ文字列のなかで同じ名前が幾つか存在している場合は、最後のその名前の値がこのマップに含められる。デコーディングには常にUS-ASCIIが使われる。

`multipart/form-data`のコンテント・タイプの場合は、該ボディ部は幾つかのフィールドたちに構文解析される。結果としてのbodyはMap<String, dynamic>となり、その値は通常のフィールドではStringとなり、ファイル・アップロードのフィールドではHttpBodyFileUploadのインスタンスとなる。同じ名前のフィールドが何回か存在する場合は、この名前の最後のものが結果としてのマップに含められる。

`multipart/form-data`のコンテント・タイプを使う場合は、Stringの値を持ったフィールドたちのエンコーディングはこのフォーム・データをもったHTTP要求を送信しているブラウザによって決まる。該エンコーディングはHTMLフォーム上の属性`accept-charset`によって、あるいはこのフォームを含むウェブ・ページの content typeによってかのどちらかで指定される。もしこのHTMLフォームが`accept-charset`属性を持っていないときは、そのブラウザは該フォームを含むウェブ・ページのコンテント・タイプからそのエンコーディングを判断する。HttpBodyHandlerのデフォルトがUTF-8であるので、そのページに対し`text/html; charset=utf-8`のコンテント・タイプを使用し、そのHTMLフォーム上の`accept-charset`を`utf-8`にセットすることが推奨される。ブラウザが送信した実際の`multipart/form-data`のHTTP要求には該アンコーディングに関する情報が含まれていないので、これらのエンコーディングの値を正しいものにすることが重要である。もしUTF-8以外のものを使うときは、HttpBodyHandlerのコンストラクタで`defaultEncoding`をセットし、processRequestとprocessResponseを呼ぶ必要がある。

その他のすべてのコンテント・タイプに対しては、該ボディ部は解釈されないバイナリ・データだとして取り扱われる。結果としてのbodyは`List<int>`となる。

要求メッセージを処理する為にHttpServerを使用するには、HttpBodyHandlerはStreamTransformerとしてあるいは要求あたりのハンドラ（processRequest参照）として使える。

```
HttpServer server = ...
server.transform(new HttpBodyHandler())
    .listen((HttpRequestBody body) {
      ...
    });
```

応答メッセージを処理するためにHttpClientを使用するには、HttpBodyHandlerは要求あたりのハンドラ（processResponse参照）として使える。

```
HttpClient client = ...
```

```
client.get(...)
    .then((HttpClientRequest response) => response.close())
    .then(HttpBodyHandler.processResponse)
    .then((HttpClientResponseBody body) {
      ...
    });
```

class **HttpBodyHandler**	
実装	
StreamTransformer<HttpRequest, HttpRequestBody>	
コンストラクタ	
HttpBodyHandler({Encoding defaultEncoding: UTF8})	Stream<HttpRequest>たとえばHttpServerとともに使われる新規のHttpBodyHandlerを生成する。 そのページがUTF-8以外のエンコーディングを使っているときは、defaultEncodingをそれに基づきセットすること。これはmultipart/form-dataのコンテントを正しく構文解析するのに必要である。multipart/form-dataの詳細情報はクラス・コメントを参照のこと。
メソッド	
static Future<HttpRequestBody> **processRequest**(HttpRequest request, {Encoding defaultEncoding: UTF8})	到来HttpRequestを処理し構文解析する。返されるHttpRequestBodyオブジェクトはHttpResponseをアクセスするためのresponseフィールドを含む。 defaultEncodingに関する更なる詳細はHttpBodyHandlerのコンストラクタを参照のこと。
static Future<HttpClientResponseBody> **processResponse**(HttpClientResponse response, {Encoding defaultEncoding: UTF8})	到来HttpResponseを処理し構文解析する。 defaultEncodingに関する更なる詳細はHttpBodyHandlerのコンストラクタを参照のこと。
Stream<HttpRequestBody> **bind**(Stream<HttpRequest> stream)	

abstract class **HttpRequestBody** HttpRequestのHttpBodyはHttpRequestBody型となる。これにはすべての要求ヘッダの情報を読み出すためのフィールドとクライアントに応答を返すためのフィールドを含んでいる。	
継承	
HttpBody	
属性	
ContentType get **contentType**	たとえばapplication/json、application/octet-stream,、application/x-www-form-urlencoded、text/plainといったコンテント・タイプ。
String get **type**	たとえば"text"、"binary"、および"json"といった以下のこのボディを構文解析したかを反映したハイレベルのタイプ値。

dynamic get **body**	実際のボディ。この型はtypeによって異なる。
String get **method**	該要求のメソッド。
Uri get **uri**	該要求のURI
HttpHeaders get **headers**	要求ヘッダたち
HttpResponse get **response**	クライアントに応答を返すためのHttpResponseオブジェクト。

abstract class **HttpFileUpload** HttpBodyFileUploadはアップロードされたファイルのラッパで、アップロードされたファイルの filename、contentType、およびdataの取得ができる。	
属性	
String get **filename**	アップロードされたファイルのfilename
ContentType get **contentType**	アップロードされたファイルのContentType。text/*とapplication/jsonの場合はdata のフィールドはStringとなる。
dynamic get **content**	アップロードされたファイルの中身。StringまたはList<int>のいずれかの型となる。

Mime_multipart_transformer.dart

mime_multipart_transformer.dartとhttp_multipart_form_data.dartはファイル・アップロードに特化したサーバには重要なライブラリである。

MimeMultipartTransformerというクラスはStreamを実装しており、MIMEマルチパート形式のバイト・データを構文解析し、各パートを表現したMimeMultipartクラスのオブジェクトのストリームに変換している。実際に使用するにはコンストラクタのみが必要である。コンストラクタの引数はバウンダリ(--プレフィックスは除く)の文字列である。

class **MimeMultipartTransformer** MimeMultipartTransformerクラスはRFC 2046 section 5.1.1で規定されているMIMEマルチパートの形式のデータを構文解析する。このデータはMimeMultipartオブジェクトたちに変換され、各オブジェクトがマルチパートのデータとしてストリームされる。	
実装	
Stream	
コンストラクタ	
new **MimeMultipartTransformer**(String boundary)	バウンダリboundaryをもとに新しいMIMEマルチパート構文解析器を構成する。boundaryはコンテント・タイプのパラメタで指定されていなければならず、--プレフィックスは含まれない。
メソッド	
Stream<MimeMultipart> **bind**(Stream<List<int>> stream)	

http_multipart_form_data.dart

http_multipart_form_data.dartはHTTP（SMTPのメールではなくて）のmultipart/form-dataで指定されたマルチパートのデータの為のライブラリである。その他のHTTP要求に対しては、別途処理しなければならない（即ちこれらのライブラリを通してはいけない）。ここで定義されているHttpMultipartFormDataというクラスは次のようになっている：

abstract class **HttpMultipartFormData**

HttpMultipartFormDataクラスはMimeMultipartを'multipart/form-data'のパートとして構文解析することでアップグレードしている。以下のコードはその使用法を示したものである：

```
HttpServer server = ...;
server.listen((request) {
  String boundary = request.headers.contentType.parameters['boundary'];
  request
     .transform(new MimeMultipartTransformer(boundary))
     .map(HttpMultipartFormData.parse)
     .map((HttpMultipartFormData formData) {
       // ここでform dataオブジェクトが使える
     });
```

HttpMultipartFormDataはストリームであり、バイトまたはデコードされた文字列として動作する。どの型のデータが使われているかはisTextまたはisBinaryで知ることができる。

実装	
Stream	
属性	
ContentType get **contentType**	構文解析されたHttpMultipartFormDataのContent-Typeヘッダ。存在しないときはnullを返す。
HeaderValue get **contentDisposition**	構文解析されたHttpMultipartFormDataのContent-Dispositionヘッダ。このフィールドは常に存在する。例えばname(form field name)及びfilename（クライアントが提供するアップロードされたファイルの名前）などのパラメタを取り出すときにこれを使用する。
HeaderValue get **contentTransferEncoding**	構文解析されたHttpMultipartFormDataのContent-Transfer-Encodingヘッダ。このフィールドはこのデータをどのようにデコードするかを判断するのに使われる。もし存在しなければnullを返す。
bool get **isText**	このデータがStringとしてデコードされているときはtrueを返す。
bool get **isBinary**	このデータが生のバイト列であるときはtrueを返す。
メソッド	
String **value**(String name)	nameという名前のヘッダの値を返す。指定した名前のヘッダが無いときはnullが返される。 このメソッドはオリジナルのMimeMultipartのなかで得られる他のヘッダたちを指す(index)のに使われる。
Static	MimeMultipartを構文解析しHttpMultipartFormDataを返す。Content-Disposition

HttpMultipartFormData **parse**(MimeMultipart multipart)	ヘッダが存在しないか無効なものであるときはHttpExceptionがスローされる。

ここでのポイントは、このクラスにはコンストラクタが存在せず、parseというstaticなメソッドを呼ぶことでこのクラスのオブジェクトが取得されることである。このオブジェクトからは：

- value： このパートのバイト列またはStringのデータ
- contentType： このパートのcontentTypeオブジェクト
- contentDisposition： このパートのcontentDispositionのHeaderValueオブジェクト
- contentTransferEncoding： このパートのcontentTransferEncodingのHeaderValueオブジェクト
- isText： このパートのデータがテキストかどうか
- isBinary： このパートがバイト列かどうか

の情報が取得できる。

22.3節　　HttpBodyHandlerを使ったサーバ例 (test_server_2.dart)

前節で説明したとおり、http_serverパッケージ・ライブラリは次のようなアプリケーションに有用である：

- POST要求を使ったアプリケーション。GETを使うとブラウザのアドレスにそれが表示されてしまうのが困る場合。特にクエリが長くなってしまう場合はブラウザのアドレス表示が見づらくなる（仕様書上はクエリ文字列の長さには制限は無い）。

- この章の本題であるファイル・アップロードを使ったアプリケーション。

ユーザは前章に示したHttpBodyHandlerというクラスを使用すると便利である。このクラスはStreamTransformerとして総ての到来HttpRequestオブジェクトに共通なコンバータとして使えるし、またあるタイプのHttpRequestオブジェクトに対応するメソッドとしても使用可能である。

総てのHttpRequestオブジェクトに共通なコンバータとして使うときは次のような記述になる：

```
HttpServer.bind(ipAddress, portNumber).then((server) {
  server.transform(new HttpBodyHandler(defaultEncoding: defaultEncoding))
      .listen((body) { // ボディ部の処理
```

またあるタイプ（例えばPOST要求のみ）のHttpRequestオブジェクトに対応するメソッドとして使用する場合は次のような記述となる：

```
  HttpBodyHandler.processRequest(request, defaultEncoding: UTF8)
    .then((body) { // ボディ部の処理
```

通常のアプリケーションではGETとPOSTの双方の要求に対処しなければならないので、こちらの記述が使われることになろう。

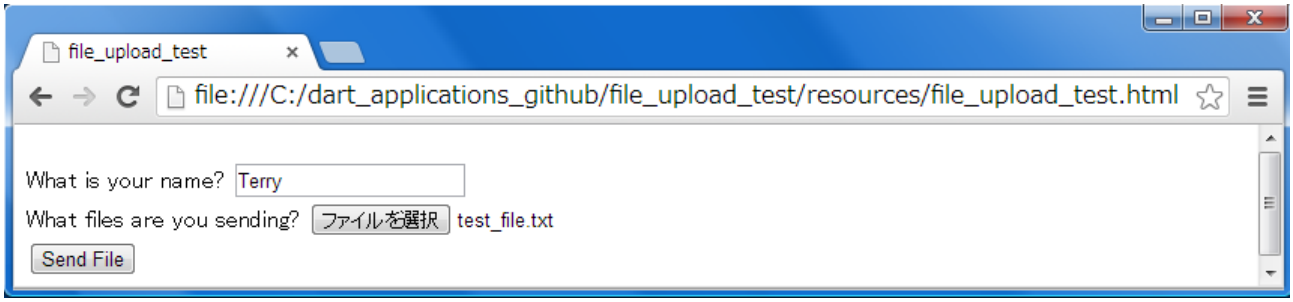
いずれにしても{Encoding defaultEncoding: UTF8}というオプショナルなdefaultEncodingという引数が使われている。これはボディ部がどの文字セットでエンコードされているかを指定する為のものである。デフォルトではUTF-8が使われる。現在dart:convertライブラリにはそれ以外のエンコーディングとしてASCII、LATIN1、及び

JSONしか用意されておらず、日本で良く使われているShift-JIS（あるいはWindows-31J）は使えないので注意が必要である。

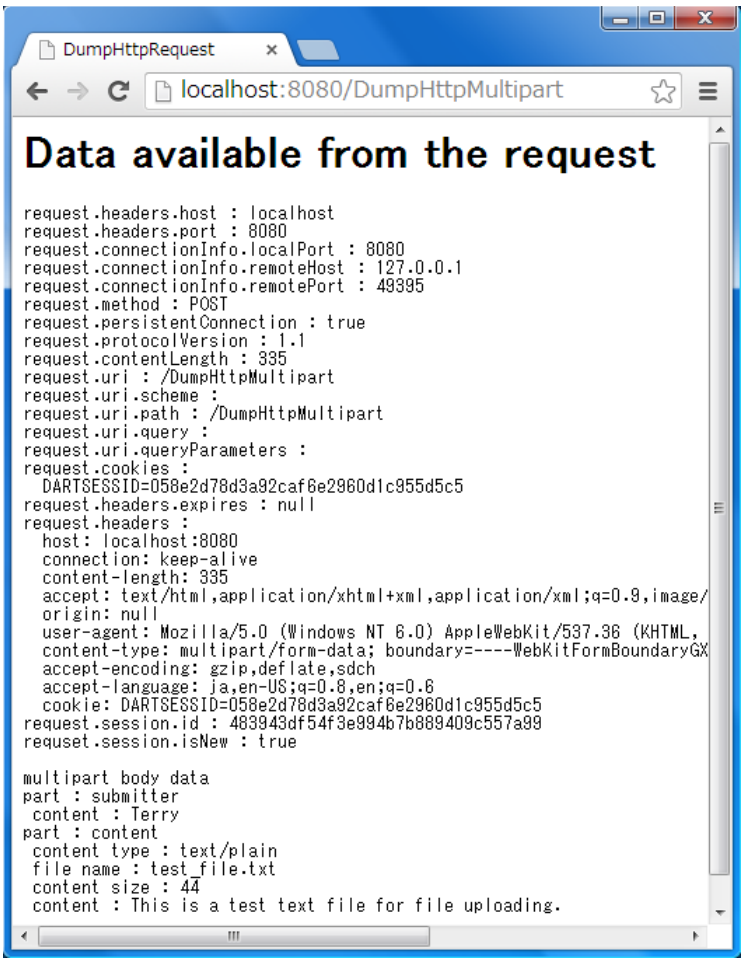
具体的な使い方はfile_upload_testというGitリポジトリに登録してあるbin/test_server_2.dartを見れば良い。読者はこれをダウンロードし、先ず実行させてその動作を確認できる。

1. このサーバを実行させる。
2. file_upload_test.htmlまたはget_post_query_test.htmlというファイルを自分のブラウザで開く。Dart Editor上でそのHTMLファイルを選択、右クリックしてCopy File Pathをクリックしてそのファイルのファイル・パスをクリックし、次にブラウザのアドレス・バーにfile:///を入力したあとにコピーしてあるそのファイル・パスを貼り付ける。
3. そのHTMLページのしかるべき箇所にデータを入力または選択してサブミット・ボタンをクリックする。
4. サーバからはその要求に関するデータが報告されてくる。

例えばブラウザから次のようなファイルをサーバに送信してみよう:



そうするとbin/test_server_2.dartサーバは次のような応答を返してくる:



これと「ブラウザが送信するマルチパート・データ」の節で示した応答と比較すると、このサーバが使っているHttpBoduHandlerから得られる情報が理解できよう。即ちマルチパートのボディ部は構文解析され、パート毎にその名前、コンテント・タイプ、ファイル名、及びコンテントなどが取得できている。

```
HttpBodyHandler.processRequest(request, defaultEncoding: UTF8)
       .then((body) {
```

StaticなメソッドprocessRequestのFutureから得られるbodyはHttpBodyを継承したHttpRequestBody型であるので、contentType、type、body、method、uri、headers、responseなどが取得できる。

マルチパート部はbody.body属性で得られ、これは各パートのMapである。一般的な処理は次のようになる:

```
001   if (body.type == "form) {
002     var mimeType = body.contentType.mimeType;
003     if (mimeType == 'application/x-www-form-urlencoded') {
004       print("\nform body data :\n${body.body}");
005     }
006     else if (mimeType == 'multipart/form-data') {
007       print('\nmultipart body data');
008       for (var key in body.body.keys)
009       {
010         var part = body.body[key];
011         print('\npart : $key');
012         if( part is HttpBodyFileUpload){
013           print('\n content type : ${part.contentType}')
014           print('\n file name : ${part.filename}')
015           print('\n content size : ${part.content.length}');
```

```
016           if (part.contentType.value.toLowerCase().startsWith('text')) {
017             print('\n content : ${part.content}');
018           }
019         }
020         else print('\n content : ${part}');
021       }
022     }
023   }
```

第23章 ミドルウエア・フレームワーク (shelf)

ShelfはHTTPサーバの為のミドルウエア・フレームワークのライブラリで、Dartチームが2014年1月から開発中である。しかしこれは単なるミドルウエア・フレームワークではなく、HttpServerを使う場合よりもずっと簡単にHTTPサーバを書くことができることに加え、複数のサーバを実装出来るように配慮されているので、このライブラリが将来Dartに於けるHTTPサーバの主流になることが期待されている。Shelfは従ってdart:ioのHTTP APIとは独立している。クッキー・ベースのセッション管理もHttpServerの実装とは独立したミドルウエアとなる。

Node.jsはChromeブラウザ用に開発されたV8エンジンをサーバ・サイドでも使えるように拡張したものであるが、これにはConnectと呼ばれるウェブ・サーバの為のミドルウエア・フレームワークが存在する。Shelfはこれに影響を受けたパッケージ・ライブラリである。

Shelfを使うとウェブ・サーバあるいはウェブ・サーバの要素部品をより簡便に構成・作成できるようになる。

- 用意されているシンプルなクラスたちの小規模なセットを使うことができる
- サーバ・ロジックをシンプルな関数（即ちハンドラ）にマッピングしている：要求が単一の引数になり、応答は値として返される。
- 同期と非同期の処理を容易に組み合わせることができる。返す応答はFutureとすることもできる。
- 同じモデルでシンプルな文字列あるいはバイト・ストリームで返すことができる柔軟性を持っている。

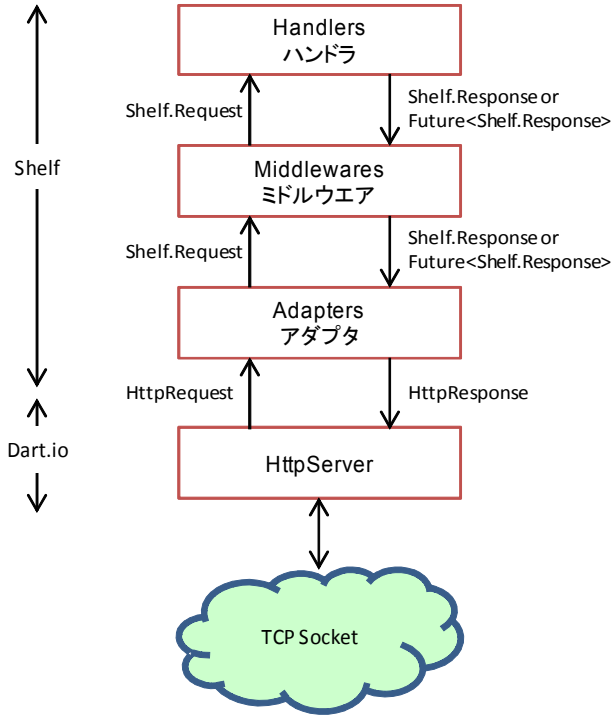
このパッケージを使ったシンプルなエコー・サーバの記述例として、このパッケージに含まれているexample/example_server.dartを示そう：

```
import 'package:shelf/shelf.dart' as shelf;
import 'package:shelf/shelf_io.dart' as io;

void main() {
  var handler = const shelf.Pipeline()
      .addMiddleware(shelf.logRequests())
      .addHandler(_echoRequest);

  io.serve(handler, 'localhost', 8080).then((server) {
    print('Serving at http://${server.address.host}:${server.port}');
  });
}

shelf.Response _echoRequest(shelf.Request request) {
  return new shelf.Response.ok('Request for "${request.url}"');
}
```

_echoRequestはサーバ・ロジックの関数である。引数はShelf要求オブジェクトで、戻り値はShelf応答オブジェクトである。handlerはログの為の組み込みミドルウエアshelf.logRequests()と、サーバ・ロジック（ハンドラ）の_echoRequestが付加されている。このように、Pipelineはミドルウエアとハンドラを付加するためのヘルパ・クラスである。io.serveはこのHTTPサーバを開始させるメソッドである。

23.1節　基本的なコンセプト

下図はShelfの基本的な構成を示す:



ミドルウエアのパイプラインはJava Servetのフィルタと似たコンセプトでもある。

いわゆる要求処理はハンドラが行い、その前後処理(例えばキャッシュ、ログ、認証など)はミドルウエアが受け持つ。ハンドラはユーザが作成するが、ミドルウエアは通常Dartチームまたはサード・パーティが開発したものを使用する。アダプタはShelfが持っている基本機能ではあるが、ユーザが専用に用意することも可能である。

ミドルウエアからハンドラに何らかの情報を渡したいときは、ShelfのRequestオブジェクトにコンテキスト(contxt)を付加する。このコンテキストは(名前、値)のMapである。逆にハンドラからミドルウエアに何らかの情報を渡したいときは、ShelfのResponseオブジェクトにコンテキストを付加する。ミドルウエアまたはアダプタに返されるShelfの応答はオブジェクトまたはそのFutureの形で渡される。一般にハンドラやミドルウエアは何らかのイベント待ちを有することが多く、Futureの形式で返すほうが便利である。

アダプタはshelf_ioはshelfをdart:ioの環境の中で使うためのアダプタである。殆どのアプリケーションはこの図のようにHttpServerのHttpRequestをshelf.Requestに変換してハンドラに渡す。

ハンドラ(Handlers)とミドルウエア(Middleware)

shelfの作成者(Kevin Moor)はハンドラとミドルウエアは同じ構造をしており、ハンドラはむしろその特別なものとして抽象化している。ハンドラをミドルウエアでラップしたものもハンドラである。これを理解すれば、以下のAPIドキュメントのトップにある記述の翻訳は面食らわなくて済む:

> ハンドラはshelf.Requestを処理してshelf.Responseを返す関数である。これには要求そのものを処理するもの(例えば要求URIをファイル・システム上で検索する静的なファイル・サーバ)と、要求に対し何らかの処理をして他のハンドラに渡すもの(例えば該要求と応答に関する情報をコンソールに出力するロ

ガー）がある。

後者の類のハンドラは「ミドルウエア」と呼ばれている。これはサーバ・スタックの途中に存在することに依る。ミドルウエアは、あるハンドラに対し更に付加的な機能を持たせるためにそれをラップして別のハンドラにするための関数だと見做すことができる。Shelfのアプリケーションは通常複数のハンドラたちを中心にしたミドルウエアの多くのレイヤで構成されることになり、その為にこの種のアプリケーションを構成しやすくするshelf.Pipelineというクラスが用意されている。

ミドルウエアのなかには複数のハンドラたちを対象にしていて、各要求に対しそれらのハンドラを選択的に呼び出すものもある。例えば、ルーティング・ミドルウエア(routing middleware)では、到来要求のURIまたはHTTPメソッド（POSTとかGETとか）に基づいてどのハンドラを呼び出すかを判断する。一方カスケード化のミドルウエア(cascading middleware)では、どれかが応答を返すまで順番に各ハンドラを呼び出す。

ミドルウエアとしては、キャッシュ、ロガー、あるいは認証などが典型的である。各種ミドルウエアは将来Shelfに含められよう。またサードパーティのミドルウエアも拡充されよう。現在pub.dartlang.orgに登録されているものとしては以下のものがある（2014年6月時点）:

- ハンドラ
  - shelf_static： 静的ファイル・サーバ
  - shelf_web_socket： WebSocketハンドラ
  - shelf_auth： 認証ハンドラ

- ミドルウエア
  - shelf_route： ルーティング
  - shelf_bind： 要求と応答のデータをクラスの属性にバインドする
  - shelf_exception_response： HTTP例外応答を発生する。
  - shelf_injection_router： ルーティング・パラメタ設定、ハンドラ注入、パラメタの検証機能を持ったルーティング

パイプライン

通常のサーバ・アプリケーションでは複数のミドルウエアのチェイン（パイプ）の最後にひとつ（ルーティングのミドルウエアがある場合は複数）のハンドラが置かれた構成になろう。そのためにPipelineというクラスが用意されている。パイプライン（Pipeline）というクラスは、ハンドラと幾つかのミドルウエアのセットを構成しやすくするためのヘルパ・クラスである。たとえば:

```
var handler = const Pipeline()
    .addMiddleware(loggingMiddleware)
    .addMiddleware(cachingMiddleware)
    .addHandler(application);
```

と記述することで、applicationというハンドラに、cachingMiddlewareおよびloggingMiddlewareというミドルウエアからなるハンドラを構成することができる。

アダプタ(Adapters)

アダプタはshelf.Requestオブジェクトを生成し、それをハンドラに渡し、またその結果のshelf.Responseオブジェクトを取り扱うコードのことを言う。その殆どの部分は、下位のHTTPサーバからの要求オブジェクトをハンドラに渡す。shelf_io.serveがこの種のアダプタになる。アダプタにはクライアント・サイド（ブラウザ）でwindow.locationとwindow.historyを使ってHTTP要求を合成するもの、あるいはHTTPクライアントからの要求を直接Shelfのハンドラにパイプするものもあり得る。

ユーザがアダプタを実装する場合には、幾つかの規則に従わねばならないい。アダプタはurlまたはscriptNameパラメタを新しいshelf.Requestに渡してはならず、requestedUriのみが渡すことが可能である。またコンテキスト・パラメタを渡すときは、総てのキーはそのアダプタのパッケージ名とそれに続くピリオドで始まらねばならない。同じ名前を持った複数のヘッダを受信したときは、RFC 2616 section 4.2に従ってカンマで区切った単一のヘッダにまとめて渡さねばならない。

アダプタはnull応答を返すものを含めそのハンドラからの総てのエラーを処理しなければならない。可能であれば各エラーをコンソールに出力し、次にあたかも500（サーバー・エラーの応答コード）応答をそのハンドラが返したかのごとく動作しなければならない。アダプタは500応答のボディ部を含めることは可能であるが、そのボディ部データは発生したエラーに関す売る情報を含めてはいけない。これは予期されていないエラーの結果デフォルトで内部情報が外部から見えてしまわないようにするためである。もしユーザが詳細なエラー記述を返したいときは、そうするためのミドルウエアを明示的に含めるべきである。

アダプタはデフォルトでHTTP応答のServerヘッダ行のなかに自分自身の情報を含めるべきである。そのハンドラがServerヘッダ行を返してくるときは、それはアダプタのデフォルトのヘッダ行より優先されなければならない。

アダプタはハンドラが応答を返した時刻をDateヘッダ行に含めるべきである。そのハンドラがDateヘッダ行を返してくるときは、そちらが優先されねばならない。

アダプタはたとえそれがFutureチェインによって報告されていないものであっても、ハンドラがスローした同期エラーたちによりアプリケーションが絶対クラッシュしないようにしなければならない。特に、これらのエラーはルート・ゾーンのエラー・ハンドラに渡されてはいけない：しかしそのアダプタが別のエラー・ゾーンのなかで走っているときは、それらのエラーはそのゾーンに渡すようにしなければならない。次の例では、そうしなければトップ・レベルに渡されてしまうエラーのみを捕捉するのに使える関数である：

```
/// Run [callback] and capture any errors that would otherwise be top-leveled.
///
/// If [this] is called in a non-root error zone, it will just run [callback]
/// and return the result. Otherwise, it will capture any errors using
/// [runZoned] and pass them to [onError].
catchTopLevelErrors(callback(), void onError(error, StackTrace stackTrace)) {
  if (Zone.current.inSameErrorZone(Zone.ROOT)) {
    return runZoned(callback, onError: onError);
  } else {
    return callback();
  }
}
```

この関数はcallbackを実行し、そうでなければトップ・レベルに渡されてしまうエラーを捕捉する。この関数が非ルート・ゾーンの中で呼ばれた時は、単にcallbackを呼びその結果を返すのみである。それ以外のときは、runZonedを使ってエラーを捕捉し、それをonErrorに渡す。

23.2節　shelfのAPI

Shelfは”shelf”と”shelf.io”の2つのライブラリで構成されている。

- “shelf”はミドルウエア・フレームワークの本体である。
- “shelf.io”はdart:ioからのHttpRequestのオブジェクトたちを処理するためのアダプタである。serveRequests関数のなかのrequestsパラメタとしてHttpServerのインスタンスを指定できる（HttpServerがHttpRequestのストリームを実装しているため）。dart:ioアダプタは要求ハイジャック(request hijacking)に対応している。

その基本構成は次のようになっている:

ライブラリ	区分	名称	概要
shelf	関数	createMiddleware	要求ハンドラ、応答ハンドラ、およびエラー・ハンドラの関数を指定してMiddlewareを生成する。
		logRequests	要求の到来時刻、内側ハンドラの経過時間、応答のステータス・コード、要求URIをプリントする組み込み済みのミドルウエア。将来各種のミドルウエアが関数として拡充されよう。
	クラス	Cascade	幾つかのハンドラたちを順番に呼び出し、最初に受理可能な応答を返すヘルパ・クラス。
		HijackException	ある要求がハイジャックされたことを示すのに使われる例外。
		Pipeline	MiddlewareのセットとHandlerからなる構成を記述し易くする為のヘルパ・クラス。
		Request	Shelfアプリケーションが処理するHTTP要求を表現したもの。
		Response	ハンドラが返す応答を表現したもの。
	Typedefs	Handler	Requestを処理する関数のシグネチュア。ハンドラはHTTPサーバから直接要求を受信しても良いし、大きなアプリケーションの要素として構成しても良い。ResponseまたはFuture<Response>を返す。
		HijackCallback	Shelfのハンドラが用意し、hijackメソッドに渡されるcallback。
		Middleware	あるHandlerをラップして新規のHandlerを生成する関数。あるHandlerをラップしてMiddlewareとすることで、そのHandlerの関数を拡張し、あるハンドラに渡す前に割り込んで要求を処理する、またはあるハンドラが応答を返す前に割り込んでその応答を処理するようにできる。
		OnHijackCallback	あるsocketでHijackCallbackを準備するためにhijackメソッドが使うcallbackで、Shelfのアダプタが用意する。

shelf.io	Functions	handleRequest	HTTP要求(HttpRequest)を処理するためにhandlerを使用する。
		serve	指定したIPアドレスとTCPポート上で要求到来を待ち、その要求をHandlerに送信するHttpServerを起動する。
		serveRequests	HttpRequestのStreamに対処する。

ここで重要な関数はミドルウエアを生成するためのcreateMiddlewareである。更により高度なミドルウエア（例えば要求を加工してハンドラに渡す）を作るにはTypeDefのMiddlewareを使用する。

```
Middleware createMiddleware({Function requestHandler(Request request),
Function responseHandler(Response response), Function errorHandler(error,
StackTrace stackTrace)})
```

requestHandler を指定したときは、これはRequestオブジェクトを受理する。このrequestHandler 関数はこの要求に対処しResponseまたはFuture<Response>を返すことができる。requestHandlerはnullを返すことができるが、その場合はそのrequestは内側のハンドラに送られる。但しそのrequestをrequestHandlerで加工しても、それは内側のハンドラに渡されるRequestオブジェクトには反映されない。requestHandlerで加工したrequestを内部ハンドラに渡したい場合は、次に示すMiddlewareの型エイリアスを使うことになる。

ここではコンセプト図の上側のハンドラをinnerHandler（内側のハンドラ）と表現しているが、これは次のような関数型エイリアスで上側のハンドラをラップしているからである：

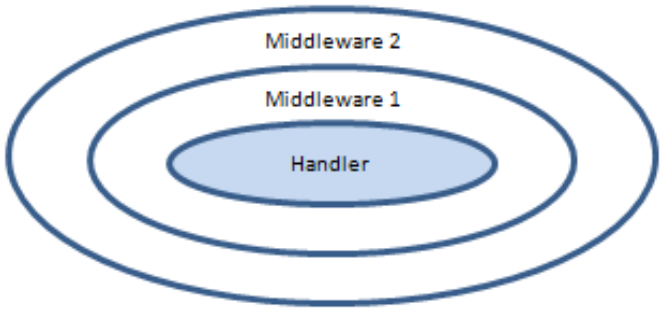
```
typedef Handler Middleware(Handler innerHandler);
```

responseHandlerが指定されているときは、このresponseHandler関数は内側のハンドラで生成されたResponseオブジェクトで呼ばれる。requestHandlerで作られたResponseオブジェクトはresponseHandlerには送られない。

responseHandlerはResponseまたはFuture<Response>を返さなければならない。このresponseHandler関数はそれが受信した応答パラメタを返すか新しい応答オブジェクトを生成するかする。

errorHandlerが指定されているときは、このerrorHandler関数は内側のハンドラでスローされたエラーを受理する。このerrorHandler関数はrequestHandlerまたはresponseHandlerがスローしたエラーは受信しないしHijackExceptionsも受信しない。このerrorHandler関数は新しい応答を返すかまたはエラーをスローするかする。

以上の説明から、API上はミドルウエアとハンドラは次のような構成になっていることが理解されよう：



- ミドルウエアもハンドラも到来要求を処理し、応答を返すことができる
- ミドルウエアはハンドラをラップする
- ハンドラをラップしたミドルウエアもハンドラである

- Middleware 1の内部ハンドラはHandlerである
- Middleware 2の内部ハンドラはHandlerをラップしたMiddleware 1である
- このようにして一連のミドルウエアとハンドラのセットが構成される

shelfのRequest

dart:ioのHttpRequestはチャンク形式のHTTP要求に対応するためにStreamを実装していたが、shelf.Requestは単にMessageを継承している。Requestのオブジェクトから HTTPボディ部をバイト列あるいは文字列として取り出すには、Messageのread（Streamベース）またはreadAsString（Futureベース）メソッドを使用する。

もうひとつの特徴はハイジャック機能である。これは要求socketの制御をハイジャックするものである。Socketレベルでの HTTP要求の操作が可能となる。

なおdart:ioのHttpSessionなどHttpRequestのオブジェクトを使うときには新たなミドルウエアを使用しなければならない。現在（2014年6月）shelf_simple_sessionというセッション・マネージャのミドルウエアが開発中である。

Requestクラスは次のような構成になっている:

- コンストラクタ
  - Request　　ユーザは通常使う必要はない。アダプタで使われる。

- メソッド
  - change　　このオブジェクトの内容を変更した新しいRequestオブジェクトをつくる。ミドルウエアで使われる。
  - hijack　　ハイジャック。
  - read　　バイト列としてボディ部をとりだす（Streamベース）。
  - readAsString　　文字列としてボディ部をとりだす（Futureベース）。
- 属性
  - canHijack　　ハイジャック可能かどうか。
  - contentLength　　ボディ部のコンテント長。
  - context　　ミドルウエアとハンドラ間のデータで、名前と値のペアのMap。
  - encoding　　ボディ部のエンコーディング。
  - headers　　ヘッダ部の名前と値のMapで不変変数。
  - ifModifiedSince　　ifModifiedSincヘッダ行。
  - method　　GETあるいはPOST等の要求メソッド。
  - mimeType　　Content-Typeヘッダから取得。
  - protocolVersion　　HTTPバージョンで1.0または1.1。
  - requestedUri　　要求URIで不変変数。
  - scriptName　　requestedUri のパスで不変変数。
  - url　　要求URIとクエリ文字列を除いた部分で不変変数。

shelfのResponse

ResponseはHTTP応答を返すためのクラスである。このクラスもMessageを継承している。このクラスのコンストラクタは次のようになっている:

```
Response(int statusCode, {body, Map<String, String> headers, Encoding
encoding, Map<String, Object> context})
```

これは指定したステータス・コード(statusCode)を持ったHTTP応答を生成する。

- statusCode　ステータス・コード（100またはそれ以上）
- body　HTTP応答のボディ部でString、Stream<dart-core<dart-core>>またはnull（ボディ部がないことを示す）である。nullまたはbodyが渡されないときはデフォルトのエラー・メッセージが使われる。
- headers　追加のHTTPヘッダ行を付加したいとき指定する。
- encoding　送信されるバイト列のストリームに対するエンコーディングで、指定しないときはUTF-8が使われる。encodingが指定されているとそれを示すContent-Typeヘッダ行がHTTP応答に付加される。指定されていないときは"Content-Type : application/octet-stream"というヘッダ行が付加される。
- context　これはミドルウエアに情報を渡すために使われる。

サーバは通常200(OK)応答を返すが、必要に応じ別の応答を返す必要が出る。Shelfではそれらの応答のための指名コンストラクタたちが用意されている:

- 403 Forbidden応答。サーバは該要求に応えることを拒否する:

```
Response.forbidden(body, {Map<String, String> headers, Encoding encoding,
Map<String, Object> context})
```

- 302 Found応答。要求されているリソースが一時的に新しいURIに移っていることを示す:

```
Response.found(location, {body, Map<String, String> headers, Encoding
encoding, Map<String, Object> context})
```

- 500 Internal Server Error応答。該要求処理中に内部エラーが起きたことを示す:

```
Response.internalServerError({body, Map<String, String> headers, Encoding
encoding, Map<String, Object> context})
```

- 301 Moved Permanently応答。要求されているリソースが新しいURIに恒久的に移ってしまっていることを示す:

```
Response.movedPermanently(location, {body, Map<String, String> headers,
Encoding encoding, Map<String, Object> context})
```

- 404 Not Found応答。要求されているURIに合致したリソースが見つからないことを示す:

```
Response.notFound(body, {Map<String, String> headers, Encoding encoding,
Map<String, Object> context})
```

- 304 Not Modified応答。指定した時刻以降要求したリソースが存在するかのGET要求に対し、変更されていないことを示す:

```
Response.notModified({Map<String, String> headers, Map<String, Object>
context})
```

- 200 OK応答。これは最も良く使われる応答で、該要求を成功裏に処理したことを示す:

```
Response.ok(body, {Map<String, String> headers, Encoding encoding,
Map<String, Object> context})
```

- 303 See Other応答。該要求の応答は別の新しいURIで得られることを示す:

```
Response.seeOther(location, {body, Map<String, String> headers, Encoding
encoding, Map<String, Object> context})
```

メソッドと属性は現在以下のようになっている:

- メソッド
  - change　このオブジェクトの内容を変更した新しいResponseオブジェクトをつくる。ミドルウエアで使われる。
  - read　バイト列としてボディ部をとりだす(Streamベース)。
  - readAsString　文字列としてボディ部をとりだす(Futureベース)。
- 属性
  - contentLength　ボディ部のコンテント長。
  - context　ミドルウエアとハンドラ間のデータで、名前と値のペアのMap。
  - encoding　ボディ部のエンコーディング。
  - expires　該応答の有効期限
  - headers　ヘッダ部の名前と値のMapで不変変数。
  - lastModified　lastModifiedSincヘッダ行で該応答が最後に変更された時刻。
  - mimeType　Content-Typeヘッダから取得。
  - statusCode　ステータス・コード。

23.3節　shelf_routeミドルウエア

HTTPサーバは要求パスに応じその処理を振り分ける必要がある。shelf_routeはそのためのミドルウエアでAtlassian社の上級技術者のAnders Holmgren氏が開発中である。これは各ルートをモジュラ形式で簡単に記述できるルータである。作者は自分のドキュメントのなかでその特徴を次のように記している:

- ルーティングに特化しており、バインディング、認可、エラー処理等は他のミドルウエアに任せている
- Shelf Bindなどの互換性のあるミドルウエアと連携するようShelf Requestのコンテキストの中でパス・パラメタをソーティングしている
- 大規模モジュール化のための階層化したルート設定ができる
- カスタムのパス書式に対応
- カスタムのハンドラ・アダプタに対応

このミドルウエアの使用法を以下に記すが、彼のサンプルも参照すると理解が早かろう。

基本的な使用法

新規のRouterオブジェクトを生成するにはこのライブラリのrouter関数を使用する:

```
var myRouter = router();
```

このルータに対しHTTPのGETメソッドに対応したルートを付加するにはこのルータのgetメソッドを使う(ここでは応答も生成してしまっている):

```
myRouter.get('/', (_) => new Response.ok("Hello World");
```

ShelfのHandlerを取得するにはこのルータのhandler属性を使用する:

```
var handler = myRouter.handler;
```

こうするとShelf IOのserveメソッドでこのルータを起動できる:

```
io.serve(handler, 'localhost', 8080);
```

HTTPメソッドによる振り分け

GET、POST、PUT、およびDELETEの各メソッドに対応できる:

```
myRouter..get('/', (_) => new Response.ok("Hello World"))
       ..post('/', (_) => new Response.ok("Hello World"))
       ..put('/', (_) => new Response.ok("Hello World"))
       ..delete('/', (_) => new Response.ok("Hello World"));
```

ルータのaddメソッドを使うことも可能である:

```
myRouter.add('/', ['GET', 'PUT'], (_) => new Response.ok("Hello World"));
```

パス・パラメタによる振り分け

一般にはパス・パラメタによる振り分けが使われる。Shelf Routeでは各ルートをパスで振り分けるのにuriライブラリのUriPatternインターフェイスを使っている。このインターフェイスを実装している限りそのパスに対する書式を任意に設定できる。デフォルトではUriTemplateが使われる。UriTemplateを使うと

- /greeting/fredといったパス・セグメント
- /greeting?name=fredといったクエリ・パラメタ

双方へのバインド可能となる。

パス・パラメタを指定するには{parameter name}記述を使う:

```
myRouter.get('/{name}', (request) =>
         new Response.ok("Hello ${getPathParameter(request, 'name')}"));
```

パス・パラメタはShelf PathのgetPathParameterで取得される。

同様に、クエリ・パラメタたちにバインドできる:

```
myRouter.get('/{name}{?age}', myHandler);

myHandler(request) {
  var name = getPathParameter(request, 'name');
  var age = getPathParameter(request, 'age');
  return new Response.ok("Hello $name of age $age");
}
```

階層化ルータ

一連のネストしたルートにルートたちを分割してモジュール化を改善できる。たとえば、/bankingで始まるすべてのルートにチャイルド・ルータを付加できる:

```
var rootRouter = router();
var bankingRouter = rootRouter.child('/banking');
```

そうするとこの銀行ルータ(banking router)にはいつものように幾つかのルートを付加できる:

```
bankingRouter
    ..get('/account/{accountNumber}', fetchAccountHandler)
    ..post('/account/{accountNumber}/deposit', makeDepositHandler);
```

こうすると全ルートに対しrootRouterを介してサービスできるようになる:

```
io.serve(rootRouter.handler, 'localhost', 8080)
```

この場合deposit(預金)のリソースに対する完全パスは実際は次のようになることに注意:

```
/banking/account/{accountNumber}/deposit
```

これを試すには次のようにこのサーバをアクセスしてみるとよい(注:コマンド行ツールのcurlが使われている):

```
curl -d 'lots of money' http://localhost:8080/banking/account/1235/deposit
```

ルート毎のミドルウエアの付加

次の場合はこのルートのすべての要求に対しlogRequests()というミドルウエアが付加される:

```
myRouter.get('/', (_) => new Response.ok("Hello World", middleware:
logRequests());
```

次のようにチャイルド・ルータにこのミドルウエアを付加するとチャイルド・ルート(banking routes)に対する総ての要求に対してこのミドルウエアが機能する:

```
var bankingRouter = rootRouter.child('/banking', middleware: logRequests());
```

23.4節　　shelf_staticハンドラ

ウェブ・アプリケーションはよりリッチな体験をユーザに与えるために、通常動的コンテンツと静的コンテンツが組み合わされる。静的コンテンツをアプリケーション・サーバから提供するには、ファイル・サーバ機能が必要になるが、sfelfではそのためのshelf_staticというファイル・ハンドラが用意されている。これはGoogleの技術者のKevin Mooreが作成したシンプルなものである。

以下のコードを見て頂きたい:

```
import 'package:shelf/shelf_io.dart' as io;
import 'package:shelf_static/shelf_static.dart';

void main() {
  var handler = createStaticHandler('example/files',
      defaultDocument: 'index.html')

  io.serve(handler, 'localhost', 8080);
}
```

このハンドラは

```
Handler createStaticHandler(String fileSystemPath,{bool
serveFilesOutsidePath: false, String defaultDocument})
```

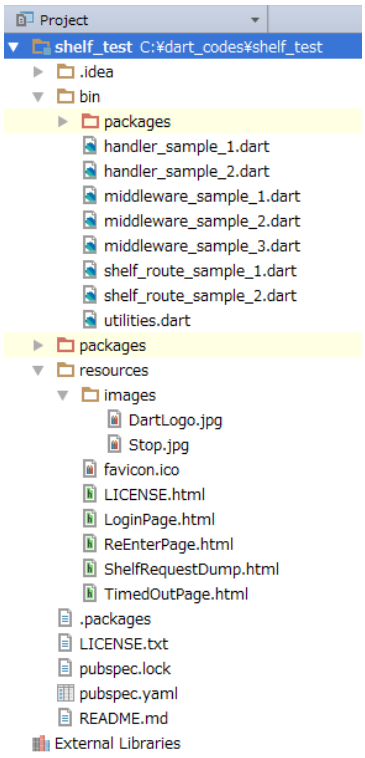
というメソッドで生成される。サービスしたい静的リソースへのファイル・パスはfileSystemPathで、ファイル名が指定されていないときのデフォルトのファイルをdefaultDocumentで指定する。serveFilesOutsidePathは指定したパス以外のファイルにアクセスするかを指定する。

23.5節　　テスト・アプリケーション

この章の読者の理解を早めるために、本章の添付サンプル・コードとして、shelf_testというリポジトリがgithubに

アップロードされている。これを「本資料に含まれているプログラムのダウンロード」という章に記載した手順に沿ってダウンロードし、自分のIDE上で活用されたい。

shelf_testというプロジェクトは2014年8月現在次のような構成になっている（Pub: Getを実行して必要な外部パッケージを取り込んだ後）:



shelf_testの構成

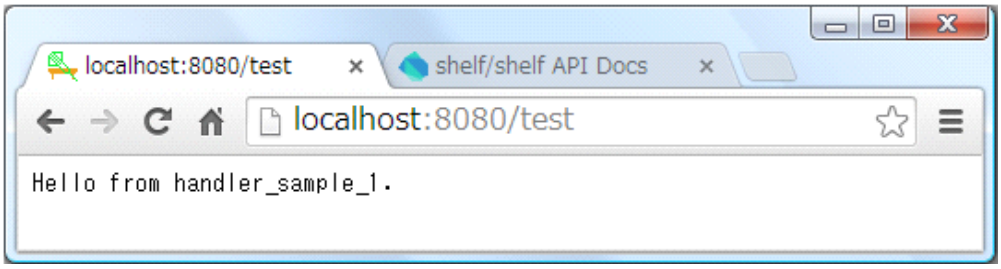
shelf_test/binのフォルダなかのコードたちがサンプル・サーバである。これらのサーバのテストにはChromeを使用することをお勧めする。IEを使った場合は:

- テキストだけのボディに対しては改行がなされない。
- favicon.icoの要求頻度が少ない
- TextAreaのデータをGETで要求するよう指定してもPOSTで要求してしまう版がある

ことに注意しなければならない。

handler_sample_1

handler_sample_1は簡単なエコー・サーバあるいはpingサーバと呼ばれるもので、このサーバにアクセスすると簡単なテキスト・メッセージをクライアントに返す。ブラウザはtext/plainをデフォルトのコンテント形式としているので、この応答を次のように表示する。



このコードは非常に簡単なものではあるが、このミドルウエア・フレームワークの基本的な使い方を示している。

ハンドラは次のように記述されている:

```
dynamic myHandler(shelf.Request request) =>
  new shelf.Response.ok('Hello from handler_sample_1.');
```

dynamicと型指定しているのは、ハンドラはshelf.ResponseまたはFuture<shelf_Response>を返すためである。

このハンドラを動作させるためには'package:shelf/shelf_io.dart'にあるserveというメソッドが使われる:

```
io.serve(myHandler, '127.0.0.1', 8080).then((server) {
  print('Serving at http://${server.address.host}:${server.port}');
```

このメソッドはFuture<HttpServer>を返すので、thenでその完了を受けてコンソールにこのサーバが起動したことを表示している。

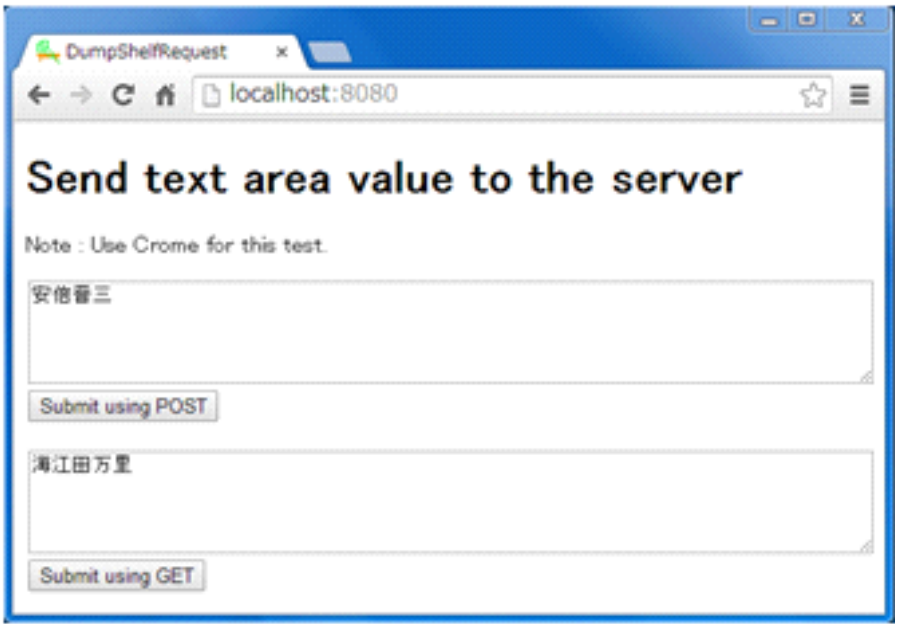
handler_sample_2

handler_sample_2はより一般的なアプリケーションの構成を示す為のサンプルである。即ち:

- 最初にアプリケーションの入り口のHTMLページを渡す。
- そのHTMLページからアプリケーション(到来shelf.Requestオブジェクトの内容をテキストまたはHTMLとしてクライアントに返す)を呼ぶ。
- クライアントからのfavicon.ico要求に対応する。

具体的にこのアプリケーションを試してみよう:

1. handler_sample_2を実行すると、コンソールには'Serving at http://127.0.0.1:8080'と表示される。
2. ブラウザから'http://localhost:8080/'でこのサーバを呼ぶと次のような画面となる:



これはこのガイドの読者には馴染みのページであるが、\shelf_test\resources\ShelfRequestDump.htmlをshelf_staticハンドラを介してクライアントに返したものである。

もう一つこの画面で注意することは、ブラウザのタブの左に小さなアイコンが表示されていることである。これはブラウザがhttp://localhost:8080/favicon.icoで要求したもので、\shelf_test\resources\favicon.icoを同じくshelf_staticハンドラを介してクライアントに返したものである。この下手くそなアイコンはダートとそれを置く為の棚(dart shelf)を示したものだが、どなたかより適したアイコンを創作して頂きたい。

3. このフロント画面で適当なテキストを入力しサブミット・ボタンを押すと、このサーバは次のような到来shelf.requestオブジェクトの内容を報告する応答を返す:

```
Available shelf.request data for this HTTP request:
shelf.request.canHijack : true
shelf.request.contentLength : 23
shelf.request.encoding : null
shelf.request.ifModifiedSince : null
shelf.request.method : POST
shelf.request.mimeType : text/plain
shelf.request.protocolVersion : 1.1
shelf.request.scriptName :
shelf.request.url : /requestDump
shelf.request.requestedUri : http://localhost:8080/requestDump
shelf.request.requestedUri.path : /requestDump
shelf.request.requestedUri.queryParameters :
shelf.request.context :
shelf.request.headers :
  user-agent : Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/36.0.1985.125 Safari/537.36
  connection : keep-alive
  cache-control : max-age=0
  accept : text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
  accept-language : ja,en-US;q=0.8,en;q=0.6
  accept-encoding : gzip,deflate,sdch
  origin : http://localhost:8080
  content-length : 23
  host : localhost:8080
  content-type : text/plain
  referer : http://localhost:8080/
request body String data :
  submitPost=安倍晋三
```

このアプリケーションの核となっているmyHandllerというハンドラは次のようになっている:

```
001 dynamic myHandler(shelf.Request request) {
002   if(request.requestedUri.path == '/' || request.requestedUri.path == '/favicon.ico') {
003     // show front page or return favicon.ico
004     return staticHandler(request);
005   }
006   else {
007     var completer = new Completer();
008     String data;
009     util.reqInfo(request).then((sb){
010       data = sb.toString();               // return plain text
011       print(data);                        // console out for debugging
012       data = util.createHtmlResponse(sb); // or, return html text
013       completer.complete(new shelf.Response.ok(data));
014     });
015   return completer.future;
016   }
017 }
```

- 001　戻りの型がdynamicとなっているのはshelf.ResponseまたはFuture<shelf.Response>が返されるからである。
- 002　要求パスが単に'/'のとき、および'/favicon.ico'のときはstaticHandlerというハンドラを呼んだ結果を返す。staticHandlerは次のようにshelf_staticライブラリのcreateStaticHandlerメソッドで生成される。

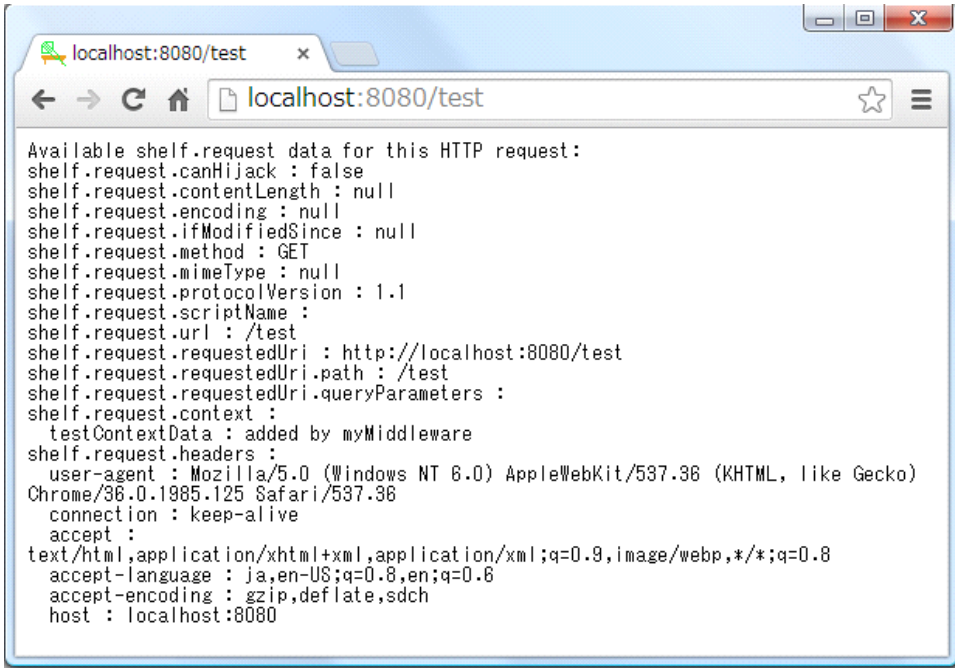
  ```
  var staticHandler = static.createStaticHandler('../resources',
      defaultDocument: 'ShelfRequestDump.html');
  ```

  引数の'../resources'は自分のコードからのリソースへの相対パスを示している。またdefaultDocumentはファイルが指定されていないときにデフォルトとして取り出すファイルを指定している。したがってこの場合は'http://localhost:8080/'というHTTP要求に対してフロント・ページが返されることになる。なおブラウザはこのようなホストを要求パスなしで呼び出した応答をキャッシュしてしまうことに注意のこと。もしこのファイルを加工した場合、あるいは'http://localhost:8080/'を別のアプリケーションで使う場合は、ブラウザの経歴を削除する必要がある。

- 007 – 015　ここではそれ以外の要求パスを持ったHTTP要求に対し、そのshelf.Requestの内容をHTMLテキストで返している。
- 009　util.reqInfo(request)という関数は、指定したshelf.Requestの内容をStringBufferでテキストとして報告する。しかしながらshelf.Requestのボディを読みだすread()およびreadAsString([Encoding encoding])はともにFutureを返す。したがってこの関数もFuture<StringBuffer>を返しているので、thenで受けている。
- 007, 013, 015　このハンドラもしたがってFuture<shelf.Response>を返したほうが適当であろう。最初に015でFutureオブジェクトを渡し、完了したらcomplete()でshelf.Responseオブジェクトを渡している。
- 010　これをStringに変換したものは、これをコンソールに打ち出してデバッグに活用すると便利である。
- 011　util.createHtmlResponse(sb)はブラウザ用にHTMLテキストに変換するツールである。

middleware_sample_1

middleware_sample_1はshelf.Requestオブジェクトを加工して内部ハンドラに渡すミドルウエアのサンプルである。このサーバを起動し、ブラウザから'http://localhost:8080/test'でアクセスすると、次のような応答が返される:



ここで

```
shelf.request.context :
  testContextData : added by myMiddleware
```

と、このミドルウエアが追加したshelf.request.contextにはtestContextData : added by myMiddlewareというデータが含まれていることが判る。

このデータを作っているのはmodifyRequestという関数である。

```
shelf.Request modifyRequest(shelf.Request request){
  var newContext = {'testContextData': 'added by myMiddleware'};
  if (request.context != null) newContext.addAll(request.context);
  return request.change(context: newContext);
```

それではミドルウエアの生成はどのように記述されているだろうか:

```
shelf.Middleware myMiddleware() {
  return (shelf.Handler innerHandler) {
    return (shelf.Request request) {
      return innerHandler(modifyRequest(request));
    };
  };
}
```

一見このコードは複雑であるが、

```
typedef Handler Middleware(Handler innerHandler);
```

という関数型エイリアスを思いだせば理解されよう。すなわちあるinnerHandlerが与えられたとき、あるrequestが与えられたときにそのrequestをmodifyRequestで加工したものでinnerHandlerを呼んだ結果としてのshelf.Responseを返す関数(即ちミドルウエア)を返す関数である。こうすればmodifyRequest(request)で加工したshelf.requestオ

ブジェクトが確実にinnerHandlerに渡されることになる。

このミドルウエアと内部ハンドラであるrequestDumpは次のようにPipelineを使って組み合わされ、myHandlerとなっている:

```
var myHandler = const shelf.Pipeline()
    .addMiddleware(myMiddleware())
    .addHandler(requestDump);
```

middleware_sample_2

middleware_sample_2は要求パスが'/middleware'の時に限り直接

```
Response for "http://localhost:8080/middleware" from myMiddleware.
```

という応答をクライアントに返すミドルウエアである。それ以外の要求パスに対しては該要求は内部ハンドラであるsimpleHandlerに渡され、このハンドラは

```
Response for "http://localhost:8080/test" from simpleHandler.
```

とういう応答を返している。

そのようなミドルウエアを作成するには

Middleware **createMiddleware**({Function requestHandler(Request request), Function responseHandler(Response response), Function errorHandler(error, StackTrace stackTrace)})

という関数を使うのが便利である。ここでは次のように生成している:

```
shelf.Middleware myMiddleware = shelf.createMiddleware(requestHandler:
  (shelf.Request request){
    if (request.requestedUri.path == '/middleware') { // direct response
      return new shelf.Response.ok('Response for "${request.requestedUri}" from
myMiddleware.');
    }
    else return null; // call inner handler
  }
);
```

即ちrequestHandlerの部分は、request.requestedUri.path == '/middleware'の時に限り直接shelf.Response.ok応答を返している。それ以外の場合はnullを返すことで該要求は内部ハンドラに渡されることを指示している。

middleware_sample_3

middleware_sample_3は内部ハンドラからのshelf.Responseを加工するミドルウエアの例である。このサーバを起動させ、ブラウザから'http://localhost:8080/123'でアクセスすると、ブラウザには次のようなテキストが表示される:

```
Response for "http://localhost:8080/123" from simpleHandler.
Time : 20:50:17.261 ... added by myMiddleware.
```

最初の行はsimpleHandlerが返したテキストであり、次の行がmyMiddlewareというミドルウエアが追加したものである。

このミドルウエアは次の関数で生成される:

```
shelf.Middleware myMiddleware() {
  return (shelf.Handler innerHandler) {
    return (shelf.Request request) {
      return new Future.sync(() => innerHandler(request))
        .then((shelf.Response response) {
              return modifyResponse(response);
      });
    };
  };
}
```

即ちこれは、要求が到来したらinnerHandlerをその要求で呼び、返されたFutureで待ち、応答がinnerHandlerから戻されたら、その応答でmodifyResponseを呼んで、結果としての応答(またはそれのFuture)を返す関数(即ちミドルウエア)を返す関数である。

modifyRequestは次のようになっている:

```
001 Future<shelf.Response> modifyResponse(shelf.Response response){
002   var completer = new Completer();
003   var newBody =
004     'Time : ${new DateTime.now().toString().substring(11)} ... added by myMiddleware.';
005   response.readAsString().then((data){
006     newBody = '${data}\n${newBody}';
007     completer.complete(new shelf.Response.ok(newBody, headers: response.headers));
008     });
009   return completer.future;
010 }
```

この関数はFuture<shelf.Response>を返しているが、これは応答オブジェクトのボディ部を読みだすresponse.readAsString()というメソッドがFutureを返している為である。

shelf_route_sample_1

shelf_route_sample_1はshelf_routeミドルウエアを理解するための最初のサンプルであり、このルータがどのように要求パス・パラメタとクエリ・パラメタを処理しているかを示している。

このサーバを起動させ、ブラウザから次のようにアクセスする:

```
http://localhost:8080/bookstore/map/tokyo?detail=false
```

そうするとこのサーバは次のような要求が内部ハンドラに渡されたと報告してくる:

```
Available shelf.request data for this HTTP request:
shelf.request.canHijack : false
```

```
shelf.request.contentLength : null
shelf.request.encoding : null
shelf.request.ifModifiedSince : null
shelf.request.method : GET
shelf.request.mimeType : null
shelf.request.protocolVersion : 1.1
shelf.request.scriptName : /bookstore/map/tokyo
shelf.request.url : ?detail=false
shelf.request.requestedUri : http://localhost:8080/bookstore/map/tokyo?detail=false
shelf.request.requestedUri.path : /bookstore/map/tokyo
shelf.request.requestedUri.queryParameters :
  detail : false
shelf.request.context :
  shelf_path.parameters : {category: map, area: tokyo, detail: false}
shelf.request.headers :
  user-agent : Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/36.0.1985.125 Safari/537.36
  connection : keep-alive
  accept : text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
  accept-language : ja,en-US;q=0.8,en;q=0.6
  accept-encoding : gzip,deflate,sdch
  host : localhost:8080
```

即ちこのルータは

```
shelf.request.context :
  shelf_path.parameters : {category: map, area: tokyo, detail: false}
```

と、要求パスに含まれているパラメタであるcategoryおよびareaと、クエリとして渡されたdetailの3つをshelf_path.parametersとしてshelf.request.contextに書き込んで内部ハンドラに渡している。これは次のようにrouterの生成と設定をしている為である:

```
var router = route.router()
    ..add('/bookstore/{category}/{area}{?detail}', ['GET'], requestDumpHandler,
        middleware: shelf.logRequests());
```

「shelf_routeミドルウエア」の節で説明したように、パスとして'/bookstore/{category}/{area}{?detail}'が指定されている。したがってルータはこのパスに合致した要求のみを内部ハンドラに渡す。'{?detail}'はクエリであるが、ルータはこれを調べ、その結果もcontextに含める。

このようにshelf_routeはUriTemplateを使ってパス・パラメタを取り出すので、単にクエリだけでなくパスもユーザ、商品、セッションなどの要求パラメタとして使うアプリケーションには有用である。

なおここでは組み込みミドルウエアのlogRequests()を指定しているので、コンソールにはこのミドルウエアからのログが表示される。

shelf_route_sample_2

shelf_route_sample_2.dartはshelf_routeミドルウエアを使ったより実用的なサーバのサンプルである。このサーバは:

- shelf_staticミドルウエアを使った静的ドキュメントのファイル・サーバ機能を有しているので、静的リソース（favicon.icoも含む）および動的リソースを使ったアプリケーションを単一のパッケージとして構成できる。
- URIテンプレートを使っているので、ユーザIDなどをパス・パラメタとして受け取ることができる。これによ

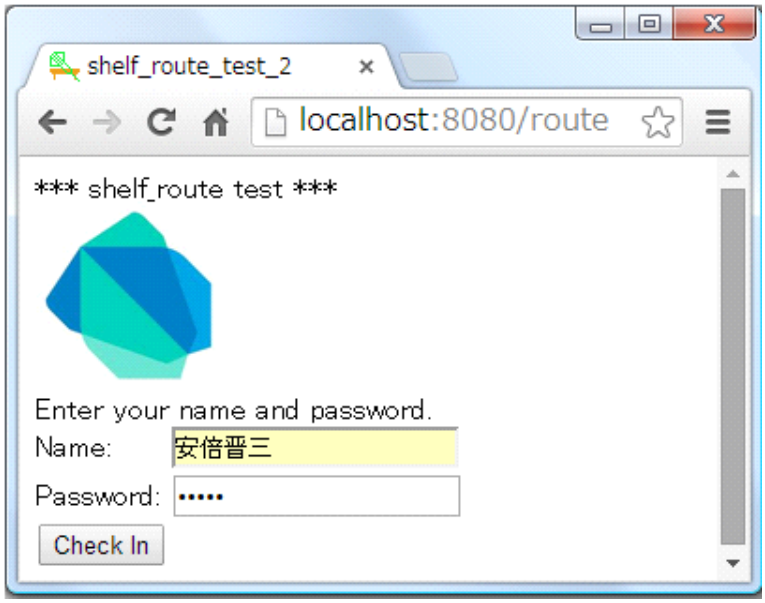
りクッキーを使わないパス・パラメタベースでのセッション管理を実現している。このセッション管理にはタイムアウト機能も有している。
- クライアントからの要求データはPOSTメソッドで要求ボディで渡されるので、セキュリティ上有利である。

shelf_route_sample_2.dartを試す

それではこのアプリケーションを先ず試してみよう。

1. アプリケーションの起動
   自分のIDEからshelf_route_test_2を選択し、実行させる。

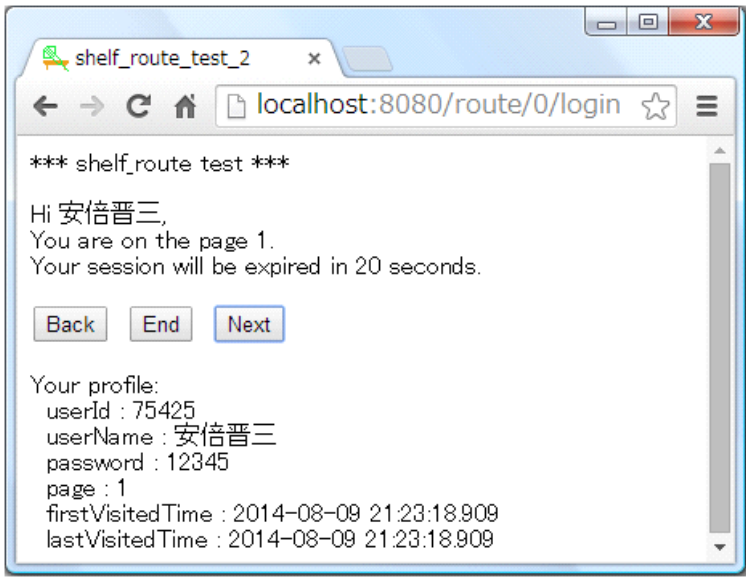
2. このアプリケーションの呼び出し
   ブラウザ（Chromeが好ましい）のアドレス・バーに”localhost:8080/route”を入力してEnterキーを押すと次のようなログイン画面（LoginPage.html）が表示される。



ログイン画面

   ここに表示されているタブ上のアイコン(favicon.ico)およびDartのロゴ(DartLogo.jpg)もこのサーバがブラウザに渡したものである。

3. ログイン後の画面
   ログイン画面で名前とパスワードを入力し、Check Inボタンをクリックすると次のような画面が表示される。



ページ遷移画面

この画面はページ遷移を試すためのもので、サーバが生成したものである。

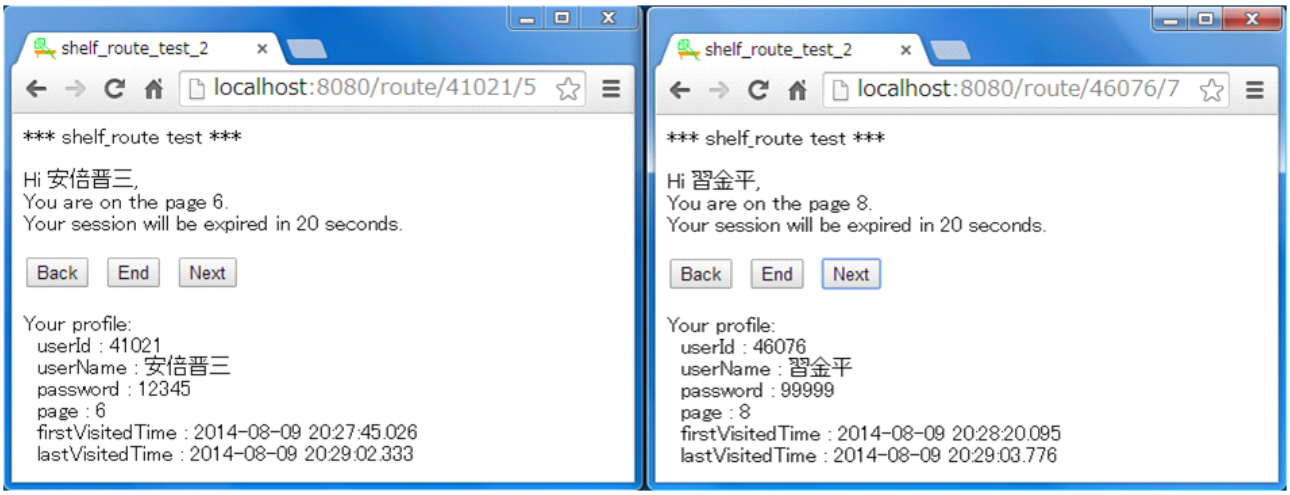
- “localhost:8080/route/0/login”というアドレスは“/route”というアプリケーションで、”/0”というユーザIDのユーザが”/login”という画面から出したHTTP要求の応答結果であることを意味している。
- “You are on the page 1.”という行は、現在1ページ目の画面であることを意味する。
- “Your profile:”はこのユーザID（ここでは新規に割り当てられた）に関してこのアプリケーションが保持している情報を示したものである。

4. ページ遷移の確認
“Back” “Next”のボタンをクリックしてページが1から10の範囲で前後に遷移することを確認する。“End”ボタンはこの画面からログイン・画面に戻るためのものである。

5. タイムアウトの確認
ここではセッションにタイムアウトを設けており、“Back” “Next”のボタンを20秒間以上経過してクリックした場合にはタイムアウト画面に遷移するようになっている。通常のアプリケーションでは20分程度の時間が使用されている。タイムアウト画面から正しい名前とパスワードを入力してCheck Inボタンをクリックすると、タイムアウト前の状態に復帰できることを確認する。

6. パスワード入力を間違えた場合の検出
一旦ログインしたユーザ名とパスワードはサーバが保持している。もし再度ログインしようとしてパスワードを間違えた場合は、ユーザ名とパスワードの再入力を要求される(ReEnterPage.html)ことを確認する。

7. ユーザ間での干渉がないことの確認
今回採用しているセッション管理はクッキーを使用していないので、クッキーが共有されるブラウザ・インスタンス間であっても干渉は存在しない。下図は2つのChromeを立ち上げているがタブ間でも相互で干渉することはないことを確認されたい。



セッション維持の確認

8. 同一ユーザが2つのブラウザ画面からアクセスしたらどうなるか。
   通常ユーザ名とパスワードによるログインでは厳重なユーザ登録があらかじめ行われる。しかしながらここではデモの為簡略化してあり、2つの画面から同じユーザがログインすることが可能である。これはあるユーザが現在の端末をそのままにして別の場所に移り、そこから別の端末を使ってこのアプリケーションを継続できるという利点もある。このような状態が発生したときには、その後ボタン操作した端末でないほうは、その後ボタン操作しようとしても強制的にログイン画面に戻されるようになっていることを確認する。

shelf_route_sample_2.dartのポイント

このサーバの動作を確認すれば、shelf_route_sample_2.dartのコードはより理解が早くなる。このコードは以下のもので構成されている:

- ルータの生成と設定
- サーバの設定と起動(main)
- 動的サービス処理のクラス(Service)
- 静的サービス処理のクラス(StaticHandler)
- デバッグに使う到来要求の内容を返すハンドラ(reqDumpHandler)

ルータの生成と設定

shelf_routeのrouter()関数で生成されるルータの型はHandlerでもMiddlewareでもない。従ってPipeline()は使用できないことに注意しなければならない。サーバの構成に当たってはrouter()関数で生成されるオブジェクトのaddメソッドしか使用できない。

```
001 var router = route.router()
002     ..add('${SERVICE}/{userId}/{page}', ['GET', 'POST'], service.doService,
003         middleware: shelf.logRequests())
004     ..add('${SERVICE}', ['GET'], staticHandler.doHandling)     // log in page
005     ..add('/{file}', ['GET'], staticHandler.doHandling)        // static file includes favicon
006     ..add('/{path}/{file}', ['GET'], staticHandler.doHandling);// static file
```

002行目で追加されているハンドラが動的サービスのメソッドである。このサービスは'${SERVICE}/{userId}/

{page}'すなわちここでは'/route/{userId}/{page}'というURIテンプレートに合致したGETおよびPUT要求に対して呼ばれる。この要求に対してはshelf.logRequests()というミドルウエアが付加されており、サーバのコンソールには次のようなメッセージが出力される:

```
2014-08-09-12:28:40.910 0:00:00.002000   POST    [200]
```

これは到来時刻、応答時刻、要求メソッド、および応答コード番号を意味している。

{userId}はセッション維持のために使われている。ここではランダムに生成した重複のない5桁の10進数の文字列が使われている。{page}はどのページから出された要求であるかを示すものである。そのクライアントは今どのページにいるかはサーバが判っている筈なので、一見無駄のように見えるが、たまたま同じユーザIDを持ったクライアントがログインしてきたときに{page}とサーバが持っているページ番号とは一致しなくなる。これはあるユーザが今使っている端末を途中で離れて、別の端末からそのサービスを継続しようとした場合に発生する。このような事態が発生したときに{page}とサーバが持っているページ番号しなくなった要求が到来したら、強制的にその端末をログイン画面に戻すようにしている。

004行目の${SERVICE}即ち'/route'というURIテンプレートに合致した(即ちhttp://localhost:8080/routeというURI)のGET要求に対してはログイン・ページを返すよう静的ハンドラに指示している。

005および006行では'/{file}'および'/{path}/{file}'というURIテンプレートに合致したGET要求は静的ファイルの要求だとしてそのファイルを返すよう静的ハンドラに指示している。

サーバの設定と起動(main)

```
001 void main() {
002   io.serve(router.handler, '127.0.0.1', 8080).then((server) {
003       print('Serving at http://${server.address.host}:${server.port}');
004   });
005 }
```

router.handlerはreuterからハンドラを取得するゲッタである。'127.0.0.1'(InternetAddress.LOOPBACK_IP_V4でも可)はIPv4のループバック・アドレスとしている。もし'localhost'と指定すればシステムがループバック・アドレスをIPv4とするかIPv6とするかを選択する。明示的にIPv6を使いたいときは'[::1]'またはInternetAddress.LOOPBACK_IP_V6と指定する。待ち受けTCPポート番号はHTTPテスト用として良く使用される8080を設定している。商用本番では80とする。

動的サービス処理のクラス(Service)

これが本サービスの本体である。最初にクラス変数として以下のものが用意されている:

```
001   Map userTable = {};  // {userId : userState}
002   Map userState = {};  // {userState : {userName:, password:, page:, ...}
003   String userId;
004   Map queries;
005   var loginState;
006   static const NEW_USER = 0;
007   static const PW_OK = 1;
008   static const PW_FAIL = 2;
009   static const SESSION_TIMEOUT = 20;
```

userTableはこのサーバが現在管理しているユーザの情報であり、userIdをキー、userStateを値とするMapである。userStateもMapで以下のものを保持する:

- userName：該ユーザの名前
- password：該ユーザのパスワード
- page：そのユーザが現在いるはずのページで画面遷移のベースとなる
- firstVisitedTime：該ユーザが最初にログインした時刻
- lastVisitedTime：該ユーザが最後にアクセスした時刻で、タイムアウトのチェックに使用する

userIdおよびqueriesは到来要求に含まれているuserIdとクエリたちである。本アプリケーションではクエリはPOST即ち要求ボディに含まれている。このクエリはHTMLでエンコードが指定されていなければ自動的にデフォルトのapplication/x-www-form-urlencodedが使用される。したがってqueriesはそれを解析してMapとしたものである。

loginStateはログイン要求を受けつけた結果を示す：

- NEW_USER：userTableに登録されていない名前のユーザを受け付けた
- PW_OK：該ユーザのパスワードはuserTableに記録されているパスワードと一致した
- PW_FAIL：該ユーザのパスワードはuserTableに記録されているパスワードと一致しなかった

SESSION_TIMEOUTはセッションのタイムアウト時間を秒で指定する。ここでは実験の為20秒と短くしてある。

dynamic **doService**(shelf.Request request)というメソッドが到来要求を処理するハンドラである。このハンドラは次のような記述になっている：

```
dynamic doService(shelf.Request request) {
  var completer = new Completer.sync();
  request.readAsString().then((data){
          要求処理
    completer.complete(new shelf.Response.ok(......));
  });
  return completer.future;
}
```

各到来要求はPOSTメソッドで送られてくるので、クエリはボディ部から取り出すことになる。readAsString()はFutureを返してくるので、その処理はthen以降の関数リテラルのなかとなる。従ってこのハンドラはFuture<Response>を返している。

要求処理の全般はログイン・ページからの要求に対する処理で、後半が画面遷移のページからの要求に対する処理となる。

ログイン・ページからの要求に対する処理は次のようになっている：

```
001       if (route.getPathParameter(request,'page') == 'login') {
002         userId = processLogin(request);
003         if (loginState == PW_FAIL) {
004           completer.complete(new shelf.Response.movedPermanently('/ReEnterPage.html'));
005         }
006         else {
007           userTable[userId]['lastVisitedTime'] = new DateTime.now();
008           completer.complete(new shelf.Response.ok(getHtml()));
009         }
010       }
```

processLogin(request)はログイン要求の名前とパスワードを調べるメソッドである。もしPW_FAILだったらそのユーザに再入力を促す/ReEnterPage.htmlを返すために301 Moved Permanently応答を返している。新規ユーザおよび再ログインのユーザに対しては、lastVisitedTimeを更新して画面遷移の画面のHTMLテキストを返している。

画面遷移の画面からの要求処理は次のようになっている:

```
001       // process requests from transition page n
002       else {
003         userId = route.getPathParameter(request,'userId');
004         var fromPage = int.parse(route.getPathParameter(request,"page"));
005         if (queries['submit'] == 'End'){
006           userTable[userId]['page'] = 1; // reset page
007           completer.complete(new shelf.Response.movedPermanently('/route'));
008         }
009         else if (fromPage != userTable[userId]['page']){ // collided by two browsers
010           completer.complete(new shelf.Response.movedPermanently('/route'));
011         }
012         else if (new DateTime.now().difference(userTable[userId]['lastVisitedTime'])
013                                    .inSeconds > SESSION_TIMEOUT) {
014           completer.complete(new shelf.Response.movedPermanently('/TimedOutPage.html'));
015         }
016         else if (queries['submit'] == 'Back') {
017           if (fromPage != 1) userTable[userId]['page'] = --fromPage;
018           userTable[userId]['lastVisitedTime'] = new DateTime.now();
019           completer.complete(new shelf.Response.ok(getHtml()));
020         }
021         else {
022           if (fromPage != 10)  userTable[userId]['page'] = ++fromPage;
023           userTable[userId]['lastVisitedTime'] = new DateTime.now();
024           completer.complete(new shelf.Response.ok(getHtml()));
025         }
026       }
```

- 005-008:'End'ボタンが押された場合は登録されている該ユーザのページ位置をリセットし、ログイン画面を301 Moved Permanently応答経由で返す。
- 009-011:蓄積しているそのユーザのページ番号と帰ってきたページ番号が一致しないときも、ログイン画面を301 Moved Permanently応答経由で返す。
- 012-015:到来要求の到来時刻がセッションのタイムアウト時間を過ぎている場合は、タイムアウトの為再度名前とパスワードを要求するページ'/TimedOutPage.html'を表示するよう301 Moved Permanently応答を返す。
- 016-020:'Back'ボタンが押された場合は、登録されている該ユーザのページ番号を一つ戻し(1が最小)'lastVisitedTime'を更新して画面遷移の画面のHTMLテキストを返している。
- 021-025:'Next'ボタンが押された場合は、登録されている該ユーザのページ番号を一つ増し(10が最大)'lastVisitedTime'を更新して画面遷移の画面のHTMLテキストを返している。

静的サービス処理のクラス(StaticHandler)

このクラスのdoHandlingが静的ファイルを返すハンドラで、shelf_staticのハンドラをラップしたものである。なぜラップしなければならないかというと:

- '${SERVICE}'という要求パス(即ち'http://localhost:8080/route'というURI)で到来したGET要求にはログイン画面を返さなければならない。
- shelf_routeが渡すRequestオブジェクトとshelf_staticが必要とするRequestオブジェクトの中身が一致しない。従ってcreateRequestというメソッドを使ってshelf.Requestを新規に作っている。

からである。

```
001 class StaticHandler{
002   dynamic doHandling(shelf.Request request)  {
003     var path = request.requestedUri.path;
004     print('staticHandler : requestedUri.path = $path');      // for debugging
005     if (request.method == 'GET' && path == '${SERVICE}')     // front page
```

```
006           return staticHandler(createRequest(request, LOG_IN_PAGE));
007     else return staticHandler(createRequest(request, path)); // static files
008   }
009
010   // create new request with newPath
011   shelf.Request createRequest(shelf.Request request, [newPath = '']) {
012     var uri = request.requestedUri;
013     return new shelf.Request('GET', new Uri(scheme: uri.scheme, userInfo: uri.userInfo,
014         host: uri.host, port: uri.port, path: newPath, query: uri.query));
015   }
016
017   // staticHandler
018   var staticHandler = static.createStaticHandler('../resources');
019 }
```

- 005-006：'http://localhost:8080/route'のときはLOG_IN_PAGEを返す。
- 007：通常のファイル要求の時は新規に作ったshelf.Requestをもとにそのファイルを返す。
- 018：staticHandlerは'../resources'という相対ディレクトリのなかで指定されたファイルを探し、それを応答として返す。もし存在しない時は404 Not Found応答を返す。

第24章 RESTfulウェブ・サービスとDart (Dart with RESTful web services)

ウェブ・サービスの世界ではREST (Representational State Transfer)というコンセプトが良く使われる。
Yahoo、Google、Facebookなどのウェブ・サービスはSOAP ベースやWSDL ベースのインターフェースを非推奨あるいは不使用としており、もっぱら使いやすいRESTモデルを使ってJSONファイルによるサービスを公開している。

RESTベースでJSONファイルを交わすREST-JSONは、これまでのSOAPベースでXMLファイルを交わすSOAP-XMLに比べて、次のような利点を有する:

- サイズ: SOAP-XMLに比べずっとコンパクトであり、ネットワーク上で渡すデータ量が少なく、特にスマートフォンなどのモバイル・アプリケーションにはこれが重要である。
- 効率: REST-JSONは構文解析が容易なので、データの抽出と変換が容易である為、クライアントのCPU負荷がずっと軽くなる。
- キャッシュ: キャッシュに対応しているので、応答時間とサーバ・ロード時間が改善される。
- 実装: REST-JSONのインターフェイスは設計と実装がより容易である。

SOAP-XMLは一般に大量のテキストを交換するときや、銀行などのセキュアなサービスで使われている。

本章では、Dartが現時点でクライアント側およびサーバ側でRESTベースのウェブ・サービスにどのように対応しているかを紹介する。

下表は現時点で用意されているDartのライブラリの主要なものである:

名称	形態	用途	概要	作者
shelf_rest	pub	サーバ	shelfのRESTハンドラ	Anders Holmgren
rest_dart	pub	サーバ	dart:ioのHttpServer上でのRESTサーバ構築ライブラリ	Matthew Barbour
rest	pub	サーバ	HTTP RESTサーバ実装	Matthew Coleman
rest_let	pub	サーバ	dart:ioのHttpServer上でのシンプルなRESTサーバ	Meal Adam
uri	pub		UriTemplate対応ライブラリ	Google
JsonCodec JsonEncoder JsonDecoder	dart:convert		JSONコーデック	Google
jsonp	pub	クライアント	JSONP要求作成作業の簡素化	Matthew Franglen
googleapis	pub	クライアント	Googleのウェブ・サービス・アクセスの為のAPI (発表) (サンプル)	Google

サーバ側では多くの人たちが自分のライブラリを公開している。またクライアント側では、Googleのいろんなウェブ・サービスの APIをDart言語に変換したものが用意されている。

24.1節　RESTスタイルの概説

REST (Representational State Transfer) とは、コンピュータ科学者のRoy Thomas Fieldingが2000年に自分の博士論文の中で提唱したネットワーク・ベースのアプリケーション・ソフトウエアの指針となるアーキテクチャ・スタイルである。RESTだけだとアメリカでは公衆便所（正式にはRest Roomだろうが）になって仕舞うので、通常はこのスタイルをきちんと実装したという意味で「RESTfulな」というように、形容詞として良く使われる。

本節では、Fieldingが提案したRESTの概説と、実際のウェブ・サービスに於けるRESTfulサービスの基礎を説明する。

Fieldingの提案

Roy Thomas Fieldingは1965年California生まれのアメリカ人で、California州立大学Irvine校の情報とコンピュータ科学科修士課程にいたときからウェブの標準策定に関わった。丁度このころが世界の大学や研究機関が開発の中心だったインターネットの世界でのウェブの誕生・発展期であり、非常に良い時代と環境に恵まれていたことになる。彼はこの大学で2000年に博士号を取得している。日本でも慶応大学、大阪大学、東京大学等がインターネットの日本での普及に大きく寄与したが、標準化には殆ど寄与できなかったことは反省すべきである。彼の標準化での貢献は、特に1996年のHTTP 1.0 プロトコル(RFC 1945)及び1999年のHTTP 1.1 プロトコル(RFC 2616)で大きい。HTTPはウェブの発明者ともいわれるTim Berners-Leeが考え出したプロトコルである。彼がBerners-Leeなどとともに策定したHTTPプロトコルはHTTP 1.1で強化され、現在もそのまま使われている。HTTPに関しては、筆者の「改定サーブレット・チュートリアル」の2.4節を参照のこと。彼はまたW3C (World Wide Web Consortium)でのHTMLとURI (RFC 1808, 2396)の標準化にも貢献した。彼はまたApache HTTPサーバ・プロジェクトの設立メンバのひとりであり、1999年から初代会長を3年間務めた。現在もASF (The Apache Software Foundation)の役員のひとりである。1999年にMITのTechnology Review誌の「トップ100人の35歳以下の若手革新者たち」初年度に彼が指名されている。

その彼の成果に対するご褒美としての博士論文「ネットワーク・ベースのソフトウエア・アーキテクチャのアーキテクチャ・スタイルと設計(Architectural Styles and the Design of Network-based Software Architectures)は、自分がこれまのウェブでの貢献を整理するにあたり、その基本となってきたコンセプトとしてRESTという基本原則を示して、博士論文らしくしようとした（あるいは論理武装しようとした？）ものと推察される。これはハイパーメディア上の構成要素たちが、リソースの表現(Representation)を使ってそのリソースのある状態(State)を転送(Transfer)しあうという基本コンセプトである。

このRESTという原則がウェブ・サービスの世界で注目され、その使い易さ（例えばサーバはクライアント間の状態を保持しない）からこれまでのCORBAあるいはSOAP / WSDLベースのインターフェイス設計を凌駕するほどになっている。これは上位のアプリケーション・プロトコルから下位の通信・プロトコルであるHTTPへの回帰現象ともいえよう。現在多くのウェブ・サービスのプロバイダたちが、自分のサービスの為のRESTベースのAPI（ここでいるAPIというのは、URLの一部としてサーバにパラメタを渡すWeb APIの仕様のこと）を提供している。またJCP (Java Community Process)ではJSR 311 (JAX-RS: The Java™ API for RESTful Web Services)という仕様書が出されており、Java EE6に組み入れられている。

彼が主張しようとしていることは、彼の1998年のプレゼンテーション資料のアブストラクトにより良く示されている：

アーキテクチャはあるスタイルの実体化物（インスタンス）とも言え、たとえばWWW技術の世界ではURL, HTTP, HTML, Java applets等がこれにあたる。スタイルというのはアーキテクチャたちのなかで共通したパタンのことをいう。スタイルとしては、例えばパイプとフィルタ、リモート・セッション、イベント・ベースの統合（暗示的呼び出し）、クライアント/サーバ（明示的呼び出し）、分散オブジェクト、及び多種の分散プロセスのパラダイムなどが使われてきている。これらのスタイルの各々は要素間の通信の特定のパラダイムに最適化するように意図されている。

彼の考えでは、ウェブのアーキテクチャ・スタイルは以下の5つの基本的概念が中心になっている:

- リソース
- リソースの表現(representation)
- 表現の取得/修正の為の通信
- アプリケーションの状態(state)のインスタンスとしてのウェブ「ページ」
- ある状態から次の状態に移動する為のエンジン
  - ブラウザ（もっとも一般的に使われている）
  - スパイダ
  - なんらかのメディア・タイプのハンドラ

「WWWは自分がRESTと称する新しいアーキテクチャのスタイルへと発展した」と彼はいう。このスタイルはクライアント/サーバ、パイプとフィルタ、及び分散オブジェクトのパラダイムたちの要素たちを使うことで、あるリソースの表現のネットワーク転送を最適化する。ウェブ・ベースのアプリケーションというのは、状態表現（ページ）と状態間の潜在的転移（リンク）のダイナミックなグラフだと見ることが出来る。それがもたらされる結果は、サーバの実装とクライアントのリソースの認知との分離、大きなクライアント数に十分拡張できる、無制限の規模とタイプのストリーム・データの転送ができる、データ転送とキャッシングの要素のような仲介者（プロキシとゲートウェイ）に対応できる、そしてユーザ・エージェント要素内でアプリケーションの状態に集中した、アーキテクチャであると彼はいう。

彼がRESTをどのように導き出したかは、彼の博士論文の第5章の最初の部分（5.1節）で説明されている。彼は全く空のスタイル(null style)に以下のような幾つかの制約を付加することで、RESTアーキテクチャ・スタイルを導入している:

- クライアント-サーバのアーキテクチャ・スタイル：ユーザ・インターフェイスとデータ蓄積を分離することでユーザ・インターフェイスのプラットフォーム間での可搬性（portability）が高まり、またサーバが簡素化され拡張性が高まる
- 状態なし（ステートレス）のアーキテクチャ・スタイル：通信に状態を持たせない。各クライアントからサーバへの要求は、その要求を理解する為の総ての情報を持たせる。セッション状態はクライアント上でのみ保持される。これにより可視性（例えば監視システムはその要求を調べるだけで良い）、信頼性（部分的な障害からの復旧が簡単）、及び拡張性（サーバ側で状態情報を保持しなくても良い）が得られる
- キャッシュ付加可能なスタイル：ある要求に対する応答にキャッシュできるかどうかを明示的あるいは暗示的ラベルを付加することで、効率、拡張性、ユーザが受け取る感じを改善できる
- インターフェイスが統一されたスタイル：ハイパー・メディアを構成する各要素間のインターフェイスを統一する。これにより情報が標準化された形式で転送されるので、システム全体のアーキテクチャが簡素化され、また要素のインターフェイスの一般化がなされる
- 階層化されたシステムのアーキテクチャ・スタイル：大規模なインターネットに対応できるよう、階層化されたシステムのアーキテクチャ・パタンとする：各要素は自分が属する階層以外の階層が不可視になることで、大規模化に対応できるようになる
- コード・オン・デマンド：アプレットやスクリプトなどの形式で実行コードがダウンロードできるようにする。これによりクライアント側が簡素化される。また、システムの拡張性が得られる。但しこれは可視性を阻害することになるので、RESTの中ではオプショナルな制約となっている

RESTにおけるデータ要素たちを纏めると次の表のようになる。これらはHTTPプロトコルを理解していれば理解が早い（筆者の改定サーブレット・チュートリアルの第2章参照のこと）:

データ要素	ウェブでの事例
リソース	ハイパーテキスト参照の意図されたコンセプト上のターゲット
リソース識別子	URL、URN
表現	JSON、XML、HTMLなどのドキュメント、JPEGイメージ
表現のメタデータ	メディア・タイプ(media type)、前回変更時刻(last-modified time)
リソースのメタデータ	ソースのリンク(source link)、代替(alternates)、変動(very)
制御データ	if-modified-since、cache-control

RESTのデータ要素

リソース

リソースとはRESTにおける「情報」の重要な抽象化である。リソースは名前（識別子、実際にはURI: Uniform Resource Identifier）をもったもので、ドキュメントまたはイメージ、例えば「今日の東京の天気」といったサービス、他のリソースたちの集合、あるいは「人々」といったネットワークされていないオブジェクト、などである。URLなどで表されるリソースが指し示すのは特定のドキュメントやイメージではなく概念である、という考え方である。リソースはある実体のセットへのコンセプト的なマッピングであり、ある特定の時刻におけるそのマッピングに対応した実体のことではない。即ちリソースは時間の関数として実体のセットあるいはそれと等価な値たちを持つ。あるセットのなかの値たちは、リソース表現(resource representations)あるいは/及びリソース識別子たち(resource identifiers)ということになる。

RESTでは、要素間の係わり合いのなかで関与される特定のリソースを特定する為に、リソース識別子(resource identifier)が使われる。HTTPでいえばURIがこれに相当する。

リソースの表現

RESTの構成部品たちは、そのリソースの現在のあるいは意図した状態(state)を捕捉するのに表現(representation)を使い、またその部品(components)間でその表現を転送(transfer)することで、あるリソースにたいするアクションを行う。RESTの構成部品たちは、そのリソースに直接関わりあおうことは無く、あくまでもその表現を介すことになる。表現はバイト列、及びそれに加えてそれらのバイトを記述する為の表現メタデータ（通常は名前と値のペアたちで構成）である。彼はウェブはリソースの表現を操作し転送するよう設計されているという。例えばHTTPではヘッダ部分がメタデータ部分であり、ボディ部分はバイト列用（通常はテキスト/HTML、MIMEなどのタイプで）として使用されている:

- 単一のリソースは複数の表現に結び付けられていても良い（コンテント交渉）
- 表現のデータ・タイプはメディア・タイプとして知られているものである（例えばMIMEなど）
  - このリソースの情報を提供
- ハイパーメディア認知のメディア・タイプをとる
  - 潜在的な状態遷移を提供する
- 殆どの表現はキャッシュできる

ウェブ・サービスの世界に於けるREST

Fieldingは自分のインターネットの世界での貢献のベースとなったコンセプトをRESTという言葉で示したが、ウェブ・サービスの世界の人たちはこのコンセプトをより自分なりにより狭く解釈している。従ってその定義はとくに標準化されている訳ではない。一般的にはFieldingの定義(「RESTの導入」の項で述べたクライアント / サーバ、ステートレス、キャッシュ対応、階層化などの制約)に対し、更に以下のものが付加されている:

- HTTPベースである
  RESTの導入にあたってFieldingはインターフェイスの統一という制約を加えたが、ウェブ・サービスの世界ではHTTPがインターフェイスのベースとなっている。従ってGET、POST、PUT、DELETEなどのHTTPメソッドを使ってクライアントはサーバにアクセスする。これらのメソッドは厳格に区別して使用される:
  - GET:取得(GET要求でリソースに何らかの影響を与える使い方はしてはならない)
  - POST:新規作成
  - PUT:更新
  - DELETE:削除

  これらはしばしばCRUD (Create, Read, Update, Delete)操作と呼ばれる
- 従ってリソースの表現に対する要求はHTTPの要求行のURIからのみとなる
  HTTPに関しては別途説明するが、HTTPの他のヘッダ行を使って要求先を指定してはならない。逆に言えばリソースはURIを介してのみ到達可能である(但しPOSTなどで送信されるデータはHTTPメッセージのヘッダ行とボディ部分が関与する)
- 表現には一般的な標準であるデータ・フォーマットを使用する
  XML、HTML、Atom、RSS、JSON、CSV、GIFなど
  特にJSONが多用されていて(REST-JSON方式)、RESTといえばJSONだと一般的に理解されるまでになっている。
- MIMEタイプとしては以下のものが使われる:
  - text/xml
  - text/html
  - application/json
  - image/gif
  - image/jpeg

    等々
- ハイパーリンクを使ってリソース間の関連(連鎖)を伝えたり、アプリケーションの状態遷移(一連のアクションに於ける関連ステップのことで、クライアントとの間の状態遷移とは意味が異なる)をクライアントが選択する手段を伝える。状態を制御・維持するのはクライアントである
  これは「連結性(Connectedness))」とも呼ばれる。あらゆるRESTベースのシステムは、クライアントが関連リソースにアクセスする必要があることを前提としており、リソース表現に関連リソースを含めてクライアントに返す。

RESTfulなウェブ・サービスでは既存の良く知られたW3CとIETFの標準たち(HTTP, XML, URI, MIME)が使われており、また最小限のツーリングでウェブ・サービスを構築できるので、RESTfulなウェブ・サービスの開発コストが低くて済み、従って導入の障壁が非常に低い。RESTfulなウェブ・サービスの開発にはEclipseのような統合開発環境が使え、開発がより簡単化される。

OracleのJava EE6チュートリアルでは、以下のような条件が満たされるときが、RESTfulなウェブ・サービスを設計するのが適正な場合であろうと書いている:

- そのウェブ・サービスが完全にステートレスであるとき。そのリソースへの係わり合いが、サーバの再起動でも影響を受けないようにできるかどうかがひとつの判断基準となる。

- キャッシュによってサービス性能が改善できるとき。そのウェブ・サービスが返すデータが動的に生成されたものでなくキャッシュ可能な場合は、ウェブ・サーバたちや仲介サーバたちが提供するキャッシングにより性能を改善できる。但し殆どのサーバにとってはキャッシュはHTTP GET要求のみに制限されているので、開発には注意が必要である。
- サービスの提供側と消費側が互いに渡すデータのコンテキストと中身を理解しているとき。ウェブ・サービスのインターフェイスを記述する公式な方法が無い為、双方は別途交換されているデータを記述するスキーム、及びそれを有意義な形で処理する手段で同意がとられていなければならない。実際には、RESTful実装としている殆どの商用のアプリケーションでは、一般的なプログラミング言語でそのインターフェイスを記述したいわゆる付加価値ツールキットを提供している。
- 帯域が特に重要であって、帯域制限の必要がある場合。PDAや携帯電話機のようにプロファイル制限があって、XMLペイロード上でのSOAP要素たちのヘッダや付加的レイヤのオーバヘッドが問題となる機器にとって、RESTは特に有用である。
- RESTfulスタイルにより、既存のウェブのサイトたちにウェブ・サービスのデリバリ（提供）あるいはアグレゲーション（統合）が容易になる。開発者たちはJAX-RS及びAJAX (Asynchronous JavaScript with XML)のような技術が使えるし、自分たちのウェブ・アプリケーションの中でそのサービスを使うのにDWR (Direct Web Remoting)のようなツールキットを使うことが出来る。白紙から始めるのではなく、既存のウェブ・サイトのアーキテクチャを大きく変えることなく、サービスはXMLで表現され、HTMLページで消費されるようにできる。既存の開発者たちは、新しい技術で最初から始めるのではなく、既に馴染みがあるものを付加することになるので、彼らはより生産的になる（仕事がはかどる）

24.2節　　簡単なクライアントとサーバ

Dartに関する多くの記事を書いていて、日本語にも翻訳されているDart in Actionの著者でもあるイギリスのEntity Group Ltdの技術者Chris Buckett氏がDartのサイトで書いているUsing Dart with JSON Web Servicesという記事は、JSONウェブ・サービスの簡単な教材として非常に適している。従って本節では、Dartを使ってRESTfulサービスを利用あるいは開発したい読者の学習の第一歩として、この記事で示されているシンプルなクライアントとサーバのサンプルを使って解説することにする。読者はこのサンプルを実際に動作させ、そのコードを理解することで、Dartでクライアントとサーバの双方でREST-JSONをどのように実現するかを把握できよう。

但しgithubにアップロードされているこの記事にあるサンプルは、2014年9月の時点（1.0.18+2版）ではその後のAPI変更（dart:JSON.stringfyはdart:convert.JSON.encodeに変更、dart:html.queryメソッドはquerySelectorに改名、HttpResponse.addStringメソッドはwriteに改名、String.concatは+に変更）に対応していないので改定を申し入れてある。改定されていない場合は、読者のほうでダウンロードしたコードを修正して頂きたい。

参考：1.0.18+2版に対する修正事項は次のようである
- import 'dart:json' as JSON;インポート文はimport 'dart:convert' show JSON;に変更
- stringfyメソッドはJSON.encodeメソッドに変更
- queryはquerySelector に改名
- addStringメソッドはwriteに改名
- Stringのconcatは+演算子に変更
- Language抽象クラスのString、List、Map型指定はcore.String、core.List、core.Mapに変更

このウェブ・サービスは次のような文字列、リスト、およびマップからなるJSONのリソースをクライアントとサーバで交わすものである：

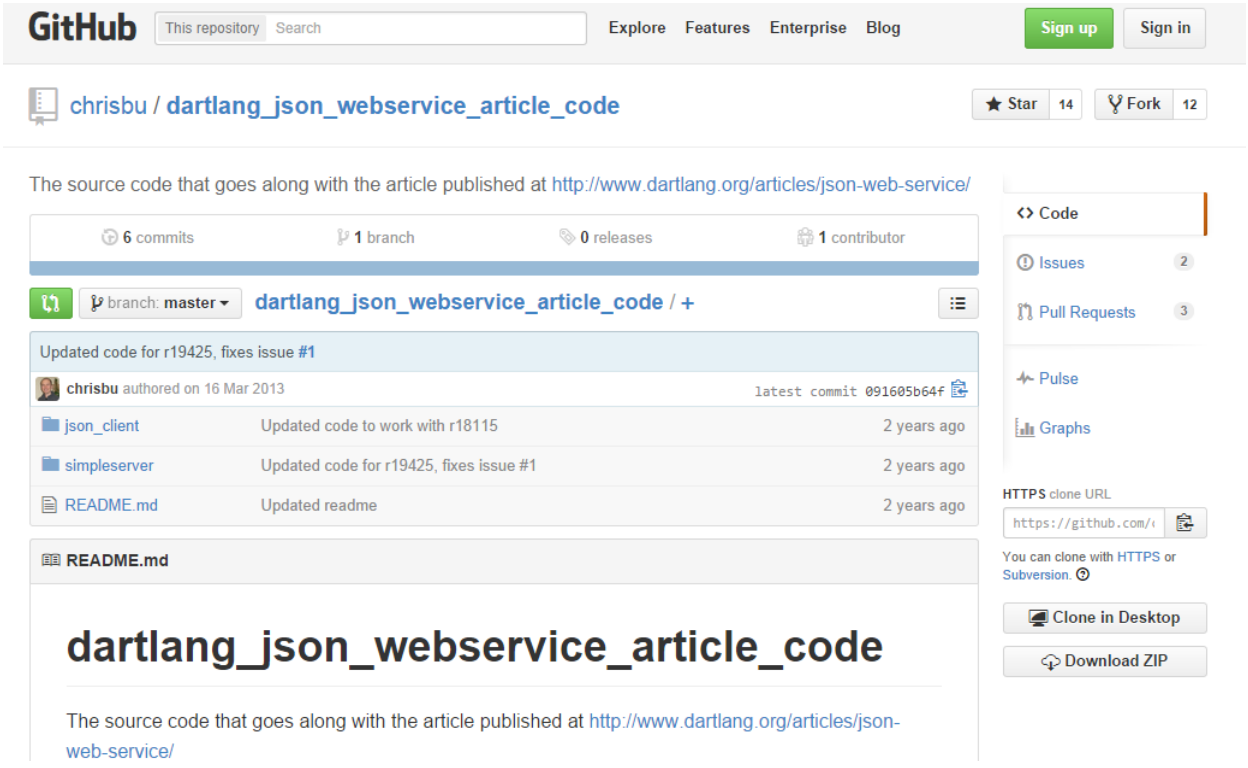
```
{
```

```
  "language": "dart",                                  // String
  "targets": ["dartium","javascript"],                 // List
  "website": {                                         // Map
    "homepage": "www.dartlang.org",
    "api": "api.dartlang.org"
  }
}
```

リソースのURIは/programming-languages/dartで、クライアントはGETメソッドでこのリソースを取得し、またPOSTメソッドで新規のリソースをサーバに渡す。サーバはPOSTで渡されたリソースを新規のdata.jsonというファイルを生成する。サーバはHTTP応答のボディ部でこのdata.jsonファイルをクライアントに渡す。

サンプルのダウンロードと実行

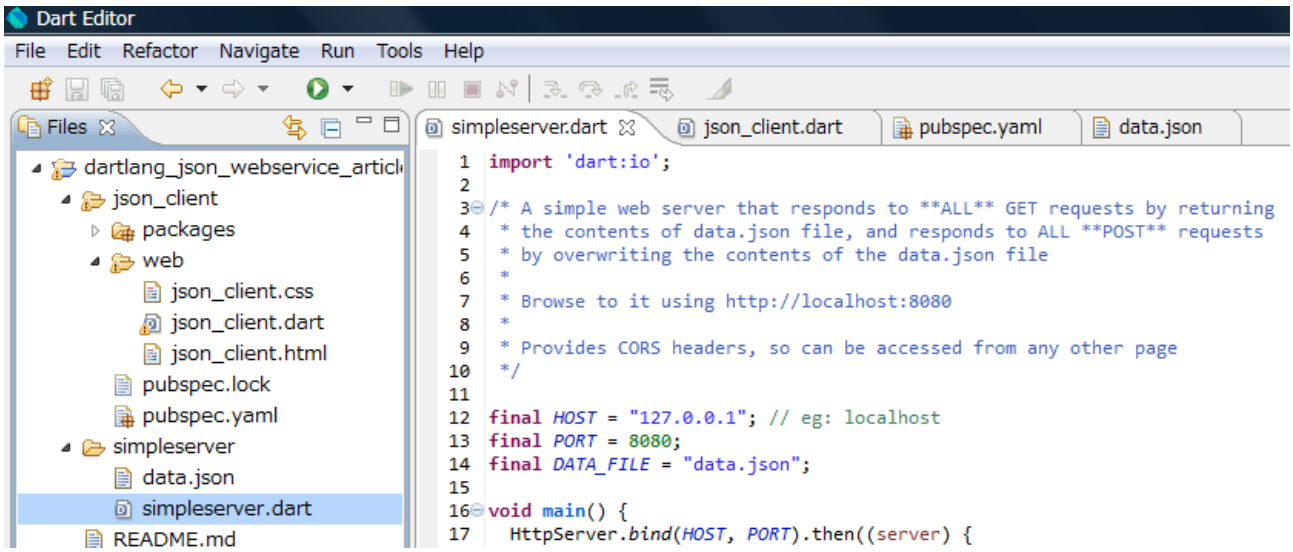
githubのdartlang_json_webservice_article_codeというリポジトリを開くと、次のような画面が表示される：



Githubにあるdartlang_json_webservice_article_code

1. 右下のDownload ZIPボタンをクリックしてこれをダウンロードし、適当な解凍ツールでこれを解凍するとdartlang_json_webservice_article_code-masterというフォルダが得られる。

2. IDEからFile→Openでこのフォルダを選択すると、下図のように展開される：



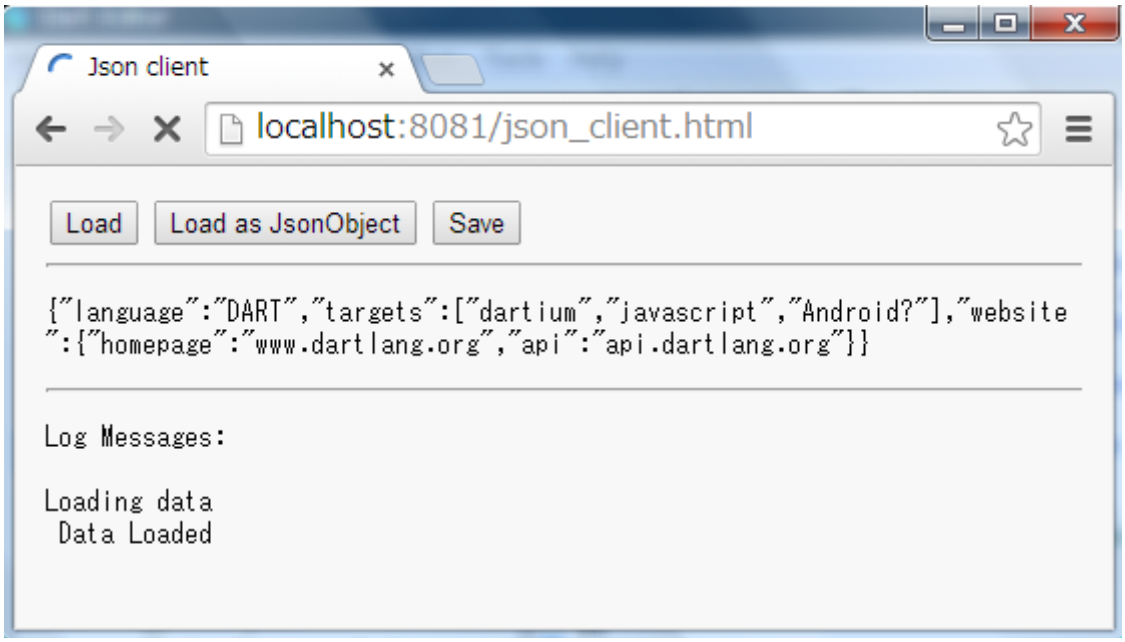
dartlang_json_webservice_article_codeの展開

クライアントはjson_client.html、json_client.cssおよびjson_client.dartで、サーバはsimpleserver.dartおよびdata.jsonで構成されている。この

3. webのフォルダにはpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込む。

4. simpleserver.dartを選択し右クリックでRunをクリックしてこのサーバを起動する。

5. json_client.htmlを選択し右クリックでRun in DartiumでDartiumからこのファイルを実行させると次のような画面が得られる:



json_client実行画面

6. 3つのボタンをクリックして、その動作を確認する:

- Loadボタン

GETメソッドでサーバを呼び、戻ってきたボディ部を表示する
- Laod as JsonObject
  GETメソッドでサーバを呼び、戻ってきたボディ部を表示するとともにjsonObjectに変換し、それをstructured dataとしてログとして表示する
- Saveボタン
  表示されているjson文字列からjsonObjectを作り、その中の"targets"要素に対し"Android?"を追加し、これをPOSTメソッドでサーバに送る。従ってこのボタン操作を繰り返すと同じ"Android?"が増加して行く

一方サーバ側では、
- GET要求に対しては、DATA_FILE即ちdata.jsonの内容をクライアントに返す
- POST要求に対しては、送信されてきたデータをバイト列として受け取り、これをそのままDATA_FILE即ちdata.jsonに書き込み、またクライアントにそのまま返している。
- OPTIONS要求はしかるべきCORSヘッダを付加するためのものである(別途説明)。

なお最初のdata.jsonファイルの中身は標準的な記述を使うと次のようになっていることも確認する:

```
{
  "language": "dart",                                  // String
  "targets": ["dartium","javascript"],                 // List
  "website": {                                         // Map
    "homepage": "www.dartlang.org",
    "api": "api.dartlang.org"
  }
}
```

つまりこのオブジェクトは、String、List、およびMapの3つの要素から構成されている。

クライアント(ブラウザ)のコード

クライアントのコードであるjson_client.dartは、クライアント用のdart:htmlライブラリのHttpRequestが使われる。

dart:htmlライブラリにはブラウザからサーバにHTTPを介してアクセスする為のHttpRequestが用意されている。API参照を調べる際にサーバ用のdart:ioライブラリのHttpRequestと間違えないように注意しなければならない。このクラスは非常に強力だが、あらかじめHTTPに関する理解が必要である。dart:html.HttpRequestはこの章の終わりに日本語化してあるので見て頂きたい。HTTPに関しては、筆者の改定サーブレット・チュートリアルの第2章参照のこと。

dart:html.HttpRequestの基本的なメソッドは以下のものがある:

- HttpRequest　コンストラクタで汎用性を持つ
- request　Future<HttpRequest>を返すstaticなメソッドで、これも汎用性がある
- getString　Future<String>を返すstaticなメソッドで、JSONを含むStringベースのデータ取得に特化している
- postFormData　Future<HttpRequest>を返すstaticなメソッドで、text/のMIMEタイプのデータの交換に適している

LoadボタンをクリックしたときにサーバからGETでデータを取り込むときは次のようにgetStringメソッドが使われている:

```
void loadData() {
  var url = "http://127.0.0.1:8080/programming-languages";

  // 非同期でウェブ・サーバを呼び出す
  var request = HttpRequest.getString(url).then(onDataLoaded);
}
```

一方Saveボタンをクリックしたときは、HttpRequestコンストラクタで生成したrequestを使って、次のようにオブジェクトをJSON変換してサーバにPOST要求で渡している:

```
  // サーバにデータをPOSTする
  var url = "http://$host/programming-languages";
  request.open("POST", url, async:false);
  request.send(JSON.encode(jsonObject));
```

なお、ここではdart:enkode.JSON.encodeメソッドを使ってオブジェクトをJSONテキストに変換している。

POSTでデータを渡した後でのサーバからの応答を受けるには先ず上記のように同期で該要求をオープンし、その後状態変化で受信の処理を行う:

```
  var request = new HttpRequest();
  request.onReadyStateChange.listen((_) {
    if (request.readyState == HttpRequest.DONE &&
        (request.status == 200 || request.status == 0)) {
      // サーバがデータをセーブしてステータス200 OKを返してきた
      print(" Data saved successfully");

      // update the UI
      var jsonString = request.responseText;
      querySelector("#json_content").text = jsonString;
    }
  });
```

サーバのコード

サーバのコードはsimpleserver.dartで、その名のとおりシンプルで理解しやすいものである。RESTベースのサーバではHTTPメソッドごとに処理を振り分ける必要がある:

```
void main() {
  HttpServer.bind(HOST, PORT).then((server) {
    server.listen((HttpRequest request) {
      switch (request.method) {
        case "GET":
          handleGet(request);
          break;
        case "POST":
          handlePost(request);
          break;
        case "OPTIONS":
```

```
        handleOptions(request);
        break;
      default: defaultHandler(request);
    }
  },
  onError: printError);

  print("Listening for GET and POST on http://$HOST:$PORT");
 },
 onError: printError);
}
```

JsonObject

現在Dartではobject-to-jsonおよびjson-to-objectのマッパが存在しない。例えばつぎのようにJSON.decodeを使って型キャストをしようとしてもエラーとなる。

```
import 'dart:convert';
main() {
  var jsonText = '{"language":"DART","targets":
["dartium","javascript","Android?"],"website":
{"homepage":"www.dartlang.org","api":"api.dartlang.org"}}';
  var jsonObject = JSON.decode(jsonText);
  print(jsonObject);
  (jsonObject as Language).language = "Cobol";
  print(jsonObject);
}
class Language {
  String language;
  List targets;
  Map website;
}
```

JSON.decodeメソッドはJson obujectと称するDynamicなオブジェクト（IterableやMapで）を返す。Dartはこれをある型に変換するためのグローバルに独自のクラス名たち（名前空間）を持っていない為、その型に変換（deserialization）できない。これに関しては現在議論（https://groups.google.com/a/dartlang.org/forum/#!topic/misc/0pv-Uaq8FGI など）がなされている。このサンプルではChris Buckett氏がこの問題の解決策として提唱しているJsonObject（json_objectというパッケージでpubに登録されている）を使用している。JsonObcectはMap変換にJSON.decodeメソッドを使い、ドット表記でアクセスしたときに呼ばれるnoSuchMethodメソッドを使ってこの問題を解決している。

このライブラリを使うと次のようにドット表記でその要素にアクセス可能となる：

```
void onDataLoaded(HttpRequest req) {
  // decode the JSON response text using JsonObject
  JsonObject data = new JsonObject.fromJsonString(req.responseText);

  // ドット表記による属性へのアクセス
  print(data.language);          // 値の取得
  data.language = "Dart";        // 値のセット
  print(data.targets[0]);        // listのなかの値を取得
```

```
// website mapでの繰り返し操作
data.website.forEach((key, value) => print("$key=$value"));
```

但しそうするためには次のようなコードを付加する必要がある:

```
// JSONデータ構造のインターフェイスを定義する抽象クラス
abstract class Language {
  String language;
  List targets;
  Map website;
}

/** JsonObjectを拡張した実装クラスで、Language抽象クラスを実装して定義された構成を使用する
 *  JsonObjectのnoSuchMethod()関数が実際の内部の実装で使われている
 */
class LanguageImpl extends JsonObject implements Language {
  LanguageImpl();

  factory LanguageImpl.fromJsonString(string) {
    return new JsonObject.fromJsonString(string, new LanguageImpl());
  }
}
```

JsonObjectはMapを実装しているのでJSON.encodeにJsonObjectを渡すことができる:

```
var data = new JsonObject.fromJsonString(req.responseText);

// later...
// JsonObjectをStringに戻す
String json = JSON.encode(data);

// これをサーバにPOSTして戻す
HttpRequest req = new HttpRequest();
req.open("POST", url);
req.send(json);
```

CORS (Cross-Origin Resource Sharing)

セキュリティに問題があるので、ブラウザは要求に対しては組み込みアプリケーション(スクリプト)を作ったと同じサイトにあるものに限定している。逆に言えば要求をするコードは要求されるリソースと同じオリジン(ドメイン名、ポート番号、およびアプリケーション層プロトコル)からサービスされるものに限定される。つまり要求を行うコードを提供したと同じオリジン以外に属するリソースにアクセスできない。そうしておかないと、悪意のあるスクリプトがユーザの情報を別のサーバに送信させること(クロス・サイト・スクリプティング)を防げない。上記の例では、myData.jsonファイルはそれを使うアプリケーションと一緒に存在しなければならない。しかしCORSヘッダまたはJSONPを使えば、この制約を回避できる得る。

CORSはW3Cが標準化(勧告として)しているものである。

CORSでは、クロス・サイト・アクセスを行うクライアント(ブラウザ)側とクロス・サイト・アクセスされるサーバー側のふるまいが規定されている。クロス・サイト・アクセスされるサーバー側ではアクセスを制御するルールを設定し、ブラウザとサーバー側でHTTPヘッダを使ってアクセス制御に関する情報をやりとりしながらサイトをまたいだアク

セスをおこなう。

サーバー側で設定するアクセスを制御するルール:

- クロスドメインアクセスを許可するWebページのオリジン・サーバーのドメイン
- 使用を許可するHTTPメソッド
- 使用を許可するHTTPヘッダ

simpleserver.dartでは、この3つのルールを実装するとともに、これらをHTTP応答ヘッダに付加してクライアントに返している:

```
/**
 * このサーバ以外のサーバから渡されたスクリプトでこのサーバへのアクセスを許す為のクロス・サイト・
ヘッダ
 * See: http://www.html5rocks.com/en/tutorials/cors/
 * and http://enable-cors.org/server.html
 */
void addCorsHeaders(HttpResponse res) {
  res.headers.add("Access-Control-Allow-Origin", "*");
  res.headers.add("Access-Control-Allow-Methods", "POST, GET, OPTIONS");
  res.headers.add("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept");
}
```

CRRSヘッダ対応のブラウザは通常次のどれかを実行する:

- とにかく直接クロス・サイトのサーバのリソースにアクセスするHTTP要求を送信する
- あらかじめそのサイトがクロス・サイトのアクセスが可能かどうかを知るためにHTTP要求を送信する

24.3節　　簡単な天気予報アプリケーション

ここではサービスへの登録が不要な天気予報のウェブ・サービスを使った簡単なアプリケーションのクライアントとサーバのコード例を示す。

天気予報ウェブ・サービスはグローバルにはOpenWeatherMapが良く使われている。このサービスのAPIは非常に豊富である。都市名、緯度経度、ZIPコードなどで指定した場所の現在の気象データ、5日間または16日間の予報、歴史データ、気象地図たちが得られる。このサイトはCORS対応しているのでブラウザだけでデータを取得できる。JavaScriptによるサンプル・コードは豊富に存在するので、これをDartに変換すれば良く、読者の練習にもなろう。

しかしここでは日本のLivedoorのお天気Webサービス(Livedoor Weather Web Service / LWWS)を使ってみる。LWWSは現在全国142カ所の今日、明日、および明後日の天気予報、予想気温、および都道府県の天気概況情報を提供している。

JavaScriptやDartなどのスクリプトを使ってこのウェブ・サービスをブラウザから直接アクセスするとCORSに対応していない為、エラーが発生し、ブラウザのJavascriptコンソールには次のように表示される:

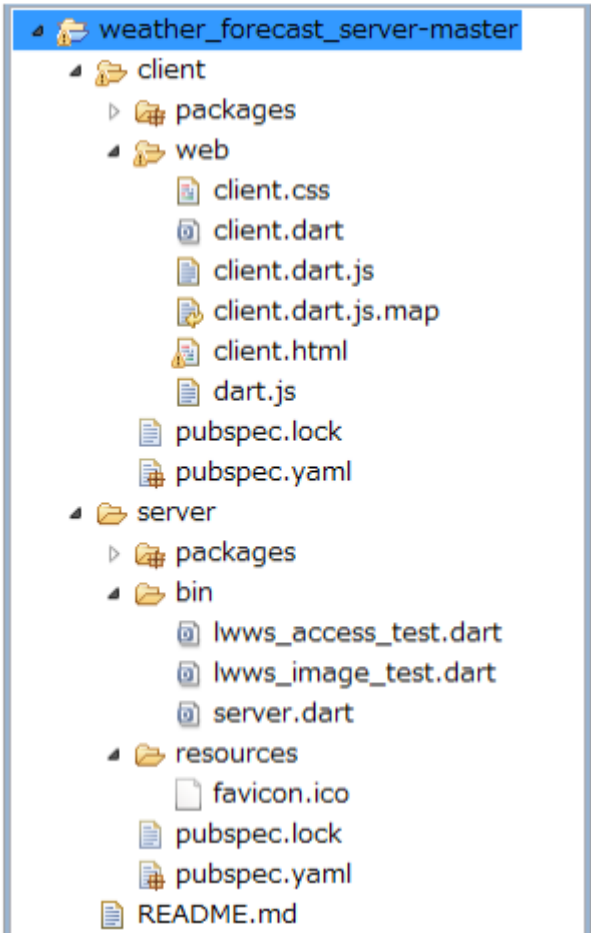
```
XMLHttpRequest cannot load http://weather.livedoor.com/forecast/webservice/json/v1?
```

```
city=130010. No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

従ってここではサーバが仲介役となってLivedoorのお天気Webサービスにアクセスする。この方式はサーバ側からウェブ・サービスにアクセスしており、これはサーバ・サイドのマッシュアップ・アプリケーションのプログラム開発のベースにもなる。

このアプリケーションはgithubに'weather_forecast_server'として公開してあるので、読者はダウンロードして実際に試してみると理解がはやくなる。

このアプリケーションは下図のような構成となっている:



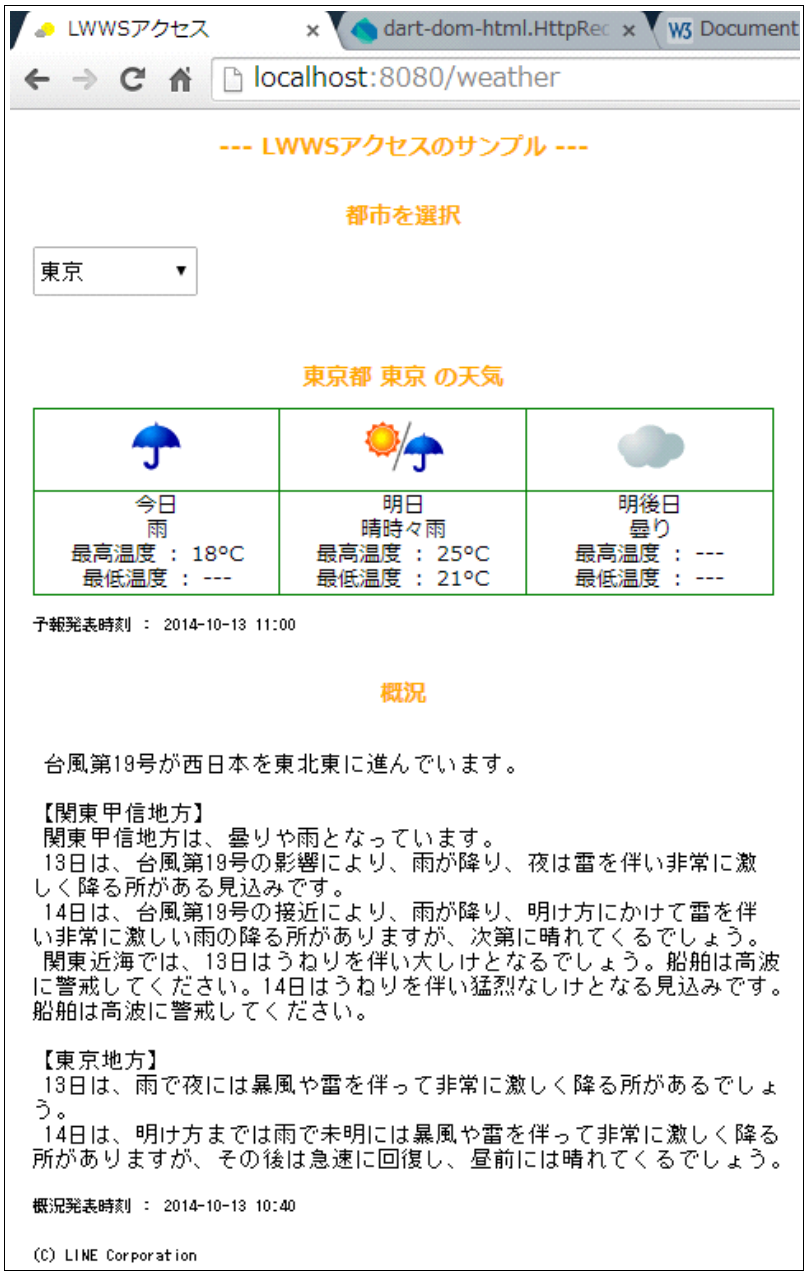
weather_forecast_serverの構成

- /client/webに置かれたファイルたちがクライアント側で使用するファイルで、サーバがクライアントに渡す
- /server/binに置かれたファイルがサーバの実行ファイルたちである。server.dartがこのサービスの為のコードであり、残りはサーバ開発のテストに使われる。
- /server/resoucesはクライアントに渡すリソースを収用する。初期はfaviconイメージのみであるが、クライアントと交信する中で天気のアイコンが蓄積されてゆく。

このアプリケーションには2つのpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込む。IDE上でserver.dartを選択し、右クリックしてRunを選択すればサーバが起動する。

Chrome、Dartium、Firefox、Safari及びIE-11（IE-9では動作しないので注意）から"http://127.0.0.1:8080/weather"でこのサーバをアクセスすると、初期画面（即ち現時点で得られる東京の予報）が表示される。「都市を選択」の個所の選択メニューから知りたい都市を選択すれば、その都市の予報が表示される。

下図はこのアプリケーションのクライアントの初期画面を示す:



LWWSアクセス画面

このアプリケーションに必要なデータは総てlocalhostすなわち本サーバが提供する。即ち:

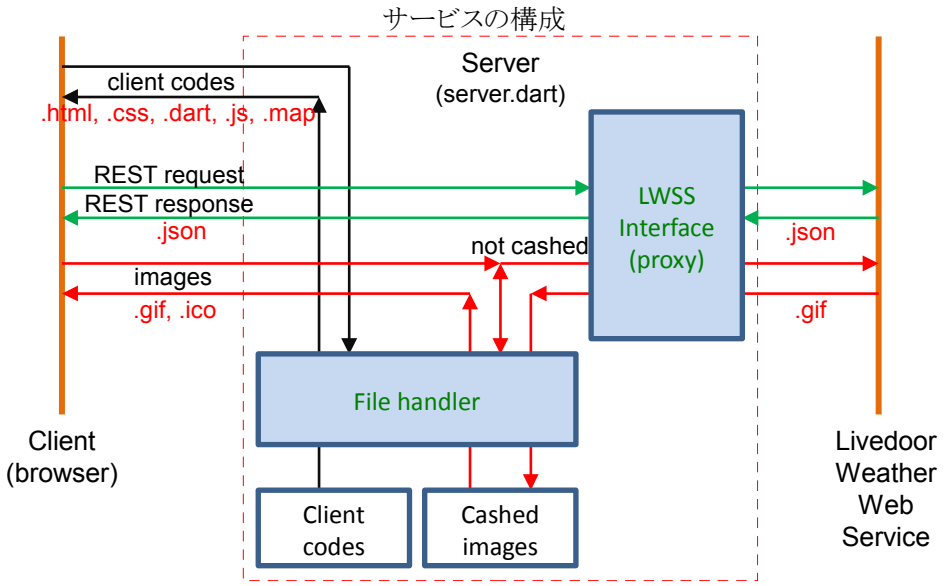
ブラウザが実行するために必要なファイルでサーバが保管している	/favicon.ico	総てのブラウザに必要
	/weather/dart.js	
	/weather/client.html	
	/weather/client.css	
	/weather/client.dart	
	/weather/client.dart.js	Chrome等Dartium以外のブラウザで必要
	/weather/client.dart.js.map	

画面作成に必要なデータ	/weather/lwws?city=10040など	LWWSからのその都市のJSONデータ
	/weather/lwws?image=1.gifなど	LWWSからのその日の天気（雨のち晴れなど）のアイコンで、このサーバが保管していない場合はLWWSから取り寄せる保管する

従ってこのサーバはファイル・サーバであるとともにLWWSのプロキシでもある。

サービスの構成

ここではサーバは基本的に中継役（プロキシ）になり、JSONファイルの中身は関与させないサービスを考えることとする。こうしたのは、ブラウザ側でどのようにJSONファイルを処理するかのサンプルとする為である。



クライアント（ブラウザ）がこのサーバをアクセスすると次のようなステップでサーバとの交信が進む：

1. クライアント画面を表示するに必要なコードが実行できるようにするために必要なコード(client codes)をクライアントが要求し、サーバは自分が蓄積しているそれらのコードを応答として返す
2. クライアントは画面のタブの中に表示するアイコン(favicon)を要求する場合がある。その場合はサーバが保管しているアイコン(favicon.ico)をファイル・ハンドラ(file handler)経由で返す
3. クライアントがそのコードを実行し、画面表示に必要な天気予報情報をサーバに要求する
4. サーバはLWWSインターフェイス経由でその要求をライブドアのLWWSサーバに要求し、戻ってきたJSONデータをクライアントに応答として返す
5. クライアントは渡された天気予報情報に記載されている画像（曇りのち雨などのアイコン）をサーバに要求する
6. サーバはその画像がキャッシュ内に存在するかどうかをファイル・ハンドラに問い合わせ、なければLWWSインターフェイス経由でLWWSにこれを要求する。応答として渡された画像はファイル・ハンドラ経由でこれをキャッシュするとともに、クライアントに返す。

7. 要求された画像が既にキャッシュされている場合は、ファイル・ハンドラ経由でこれをクライアントに渡す。

このような一連の流れは、サーバのログを見れば良くわかる。クライアントが要求したURIは赤で、それに対する応答は緑で示してある。

Chromeからのアクセス例

```
001 09:07:41.617: Serving /weather on http://InternetAddress('127.0.0.1', IP_V4):8080.
002 09:07:46.906: received a request from a client : method = GET, uri = /weather
003 09:07:46.908: requested client side file :
004 09:07:46.910: file handler : requested file : ../../client/web/client.html
005 09:07:46.964: sent response to the client for request : /weather with status = 200
006 09:07:46.992: received a request from a client : method = GET, uri = /weather/dart.js
007 09:07:46.992: requested client side file : /dart.js
008 09:07:46.992: file handler : requested file : ../../client/web/dart.js
009 09:07:46.993: received a request from a client : method = GET, uri = /weather/client.css
010 09:07:46.994: requested client side file : /client.css
011 09:07:46.994: file handler : requested file : ../../client/web/client.css
012 09:07:47.024: sent response to the client for request : /weather/client.css with status = 200
013 09:07:47.025: sent response to the client for request : /weather/dart.js with status = 200
014 09:07:47.050: received a request from a client : method = GET, uri = /weather/client.dart.js
015 09:07:47.050: requested client side file : /client.dart.js
016 09:07:47.050: file handler : requested file : ../../client/web/client.dart.js
017 09:07:47.153: sent response to the client for request : /weather/client.dart.js with status = 200
018 09:07:47.281: received a request from a client : method = GET, uri = /weather/lwws?city=130010
019 09:07:47.307: received a request from a client : method = GET, uri = /favicon.ico
020 09:07:47.308: file handler : requested file : ../resources/favicon.ico
021 09:07:47.327: sent response to the client for request : /favicon.ico with status = 200
022 09:07:47.438: LWWS interface : received response : http://weather.livedoor.com/forecast/webservice/json/v1?city=130010 with status = 200
023 09:07:47.445: sent response to the client for request : /weather/lwws?city=130010 with status = 200
024 09:07:47.506: received a request from a client : method = GET, uri = /weather/lwws?image=10.gif
025 09:07:47.507: received a request from a client : method = GET, uri = /weather/lwws?image=2.gif
026 09:07:47.553: LWWS interface : received response : http://weather.livedoor.com/img/icon/10.gif with status = 200
027 09:07:47.554: file handler : saved file : 10.gif
028 09:07:47.558: file handler : requested file : ../resources/10.gif
029 09:07:47.560: sent response to the client for request : /weather/lwws?image=10.gif with status = 200
030 09:07:47.570: LWWS interface : received response : http://weather.livedoor.com/img/icon/2.gif with status = 200
031 09:07:47.570: file handler : saved file : 2.gif
032 09:07:47.573: file handler : requested file : ../resources/2.gif
033 09:07:47.575: sent response to the client for request : /weather/lwws?image=2.gif with status = 200
```

1. 002： 最初にブラウザは/weatherでこのサーバを呼ぶ
2. 004： ファイル・ハンドラはclient.htmlだとして指示を受け、これをサーバに返す
3. 005： それがクライアントに渡されたことがここで判る。クライアントはそのHTMLファイルを読みはじめる
4. 006： クライアントはそのHTMLファイルに記載されているブートストラップ・コードのdart.jsを要求する
5. 009： クライアント同じくHTMLファイルに記載されているCSSファイル(client.css)を要求する
6. 014： ブートストラップ・コードではクライアントがDartのVMを実装していないことを知り、client.dartではなくてJavascriptに変換されたclient.dart.jsをサーバに要求する
7. 017： サーバはclient.dart.jsをクライアントに渡すと、プログラムの実行が開始される
8. 018： その結果、最初のデフォルトの東京のデータをサーバに要求する
9. 019： クライアントはまたfaviconをサーバに要求する
10. 022： サーバはLWWSに要求した東京のJsonデータを受理する
11. 023： サーバはそのJsonデータをそのままクライアントに渡す
12. 024、025： クライアントはそのデータを解析し、さらに10.gifおよび2.gifが必要であることを知り、これをサーバに要求する
13. 046,029： サーバこれらのファイルがキャッシュされていないことを知り、これをLWWSに要求する
14. 027、031： サーバはLWWSから取得したファイルをキャッシュする
15. 029、033： サーバはこれらの画像ファイルをファイル・ハンドラ経由でクライアントに送信する。クライアントはこれで表示画面を完成させることができる

この状態で次にDartimからこのサーバをアクセスすると、次のようなログが得られる。

Dartiumからのアクセス例

```
001 09:19:19.447: received a request from a client : method = GET, uri = /weather
002 09:19:19.447: requested client side file :
003 09:19:19.447: file handler : requested file : ../../client/web/client.html
004 09:19:19.449: sent response to the client for request : /weather with status = 200
005 09:19:21.145: received a request from a client : method = GET, uri = /weather/client.dart
006 09:19:21.145: requested client side file : /client.dart
007 09:19:21.145: file handler : requested file : ../../client/web/client.dart
008 09:19:21.148: received a request from a client : method = GET, uri = /weather/dart.js
009 09:19:21.148: requested client side file : /dart.js
010 09:19:21.148: file handler : requested file : ../../client/web/dart.js
011 09:19:21.149: received a request from a client : method = GET, uri = /weather/client.css
012 09:19:21.149: requested client side file : /client.css
013 09:19:21.149: file handler : requested file : ../../client/web/client.css
014 09:19:21.151: sent response to the client for request : /weather/client.dart with status = 200
015 09:19:21.152: sent response to the client for request : /weather/dart.js with status = 200
016 09:19:21.152: sent response to the client for request : /weather/client.css with status = 200
017 09:19:28.709: received a request from a client : method = GET, uri = /weather/lwws?city=130010
018 09:19:28.811: LWWS interface : received response : http://weather.livedoor.com/forecast/webservice/json/v1?city=130010 with status = 200
019 09:19:28.813: sent response to the client for request : /weather/lwws?city=130010 with status = 200
020 09:19:30.168: received a request from a client : method = GET, uri = /weather/lwws?image=10.gif
021 09:19:30.169: file handler : requested file : ../resources/10.gif
022 09:19:30.169: received a request from a client : method = GET, uri = /weather/lwws?image=2.gif
023 09:19:30.169: file handler : requested file : ../resources/2.gif
024 09:19:30.171: sent response to the client for request : /weather/lwws?image=10.gif with status = 200
025 09:19:30.171: sent response to the client for request : /weather/lwws?image=2.gif with status = 200
```

1. 001： 最初にブラウザは/weatherでこのサーバを呼ぶ
2. 003： ファイル・ハンドラはclient.htmlだとして指示を受け、これをサーバに返す
3. 004： それがクライアントに渡されたことがここで判る。クライアントはそのHTMLファイルを読みはじめる
4. 005： クライアントはHTMLファイルに記載されているDartコード(client.dart)を要求する
5. 008： クライアントはそのHTMLファイルに記載されているブートストラップ・コードのdart.jsを要求する
6. 011： クライアント同じくHTMLファイルに記載されているCSSファイル(client.css)を要求する
7. 014、015、016： これらのファイルがクライアントに渡され、プログラムの実行が開始される。ブートストラップのなかでこのクライアントがDartのVMを実装していることを知り、Dartコードが実行される
8. 017： その結果、最初のデフォルトの東京のデータをサーバに要求する
9. 018： サーバはLWWSに要求した東京のJsonデータを受理する
10. 019： サーバはそのJsonデータをそのままクライアントに渡す
11. 020、022： クライアントはそのデータを解析し、さらに10.gifおよび2.gifが必要であることを知り、これをサーバに要求する
12. 024、029： サーバこれらのファイルがキャッシュされていることを知り、サーバはこれらの画像ファイルをファイル・ハンドラ経由でクライアントに送信する。クライアントはこれで表示画面を完成させることができる

クライアントのコード

クライアント即ちブラウザのコードはサーバと交信する必要がある。ブラウザで使われるAPIのライブラリであるdart:htmlはJavaScriptに対応する巨大なものであるが、その中にはHTTPでサーバにアクセスする為のHttpRequestというクラスがある。dart:ioのHttpRequestと間違えないよう注意が必要である。このクラスはこの章の終わりに翻訳してある。

ウェブ・サービスにアクセスする場合は次のようにHttpRequest.getStringというstaticメソッドが良く使われる：

```
var path = 'myData.json';
HttpRequest.getString(path)
    .then((String fileContents) {
      print(fileContents.length);
    })
    .catchError((Error error) {
      print(error.toString());
    });
```

これはURLを指定して文字列の応答を非同期で取得するものである。受信が終了したらthenで受ける。
catchError はErrorの型が帰ると記載されているが、現在はそうなっていないのでバグ報告をあげてある。

具体的にはclient.dartでは次のような記述となっている:

```
Future loadData() {
  log("Loading data");
  var url = "http://${host}?city=$cityCode";
  // call the web server asynchronously
  var completer = new Completer();
  var request = HttpRequest.getString(url, withCredentials: false)
    .then((responseText) {
      logFlesh('JSON data loaded : $responseText');
      completer.complete(responseText);
    })
    .catchError((error) {
      log('requested data is not available : $error');
    });
  return completer.future;
}
```

- loadDataというFuture<String>を返す関数として記述されている。
- HttpRequest.getStringはFuture<String>を返すので、これをthenで受け、得られたテキストを処理する。
- catchErrorは200 OK応答以外の応答が帰ってきたときなどで発生するイベントを受ける。

更にクライアント側では受理したJsonオブジェクトをもとにリッチな画面を作るので、DOMが中心になるので、これのポイントを以下に解説する。

クライアントのコードのポイント

client.dartに比べclient.htmlとclient.cssはシンプルなものとなっている。その分DOMベースのdartコードがダイナミックな画面を生成する。JavaScriptのDOMを理解している読者であれば、client.dartは比較的容易に追うことが可能であろう。

LWWS(Livedoor Weather Web Service)から（及びプロキシであるserver.dartから）返されてくるJSONテキストは、適当なJSONエディタでその構成を調べると次のようになっている:

```
object   {8}
```

```
        pinpointLocations       [53]
        link    :       http://weather.livedoor.com/area/forecast/130010
        forecasts       [2]
        location        {3}
        publicTime      :       2014-10-07T05:00:00+0900
        copyright       {4}
        title   :       東京都 東京 の天気
        description     {2}
```

- このjsonObjectオブジェクトは8個の要素からなるMapである
- pinpointLocationsはピンポイント情報を得るためのリンク先である(今回は使用しない)
- linkは東京地方の詳細天気予報を知るためのURLである(今回は使用しない)
- forecastsは今日、明日、明後日(もしあれば)の予報のListである
- locationは都市、地域、都道府県の名前のMapである
- publicTimeはこの予報の発表時刻のMapである
- copyrightは著作権情報のMapである
- titleは概況のタイトルのMapである
- descriptionは概況と発表時刻のMapである

これをもとにJSON.decodeで得られるJsonObjectから必要なデータを取り出せばよい。

以下は幾つかのポイントを列挙することとする。

プルダウン選択メニューの選択イベントの取得

```
SelectElement smenu = document.getElementById("selectMenu");
   smenu.onChange.listen((ev){
     //selectMenuというIDのメニュー・リストであるメニューが選択されたら実行するコードを
記述
   });
```

ウェブ・サーバからJsonなどのテキスト・データを取り込む

```
 var url = "http://${host}?city=$cityCode";
 // 該ウェブ・サーバから非同期でJsonデータを取り込む
 var completer = new Completer();
 var request = HttpRequest.getString(url, withCredentials: false)
   .then((responseText) {
     completer.complete(responseText); // 受信したテキストを返す
   })
   .catchError((error) {
     // エラー処理（200 OK以外の応答だった時など）
   });
 return completer.future; // Futureを返す
```

表のある行の各<TR>要素を全部削除する

都市選択の選択メニューからある都市を選択するとその予報がこの行に追加される。従って事前に残っている

<tr>要素を削除しておかないと、この行がどんどん横に増えていってしまう。本日、明日、明後日（時間によっては本日と明日のみの場合がある）の予報はその<td>要素の数が不変である。このような場合は、現在表示されている子の要素の数を調べる必要がある。

```
var cells = document.getElementById('imageCells');
while (cells.childNodes.length != 0) cells.deleteCell(0);
```

Node.childNodesはListの値を返すので、whileループを使うとそれがゼロになるまでそのリストの最初の要素を削除することを繰り返す。

表のある行にイメージを含んだ<TR>要素を追加する

```
forecasts.forEach((forecast) { // forecastsの型はListであるのでforEachを使う
    Element tdElement = new Element.tag('td'); // <td>要素を生成
    var img = document.createElement("IMG"); // イメージ要素を生成
    Uri uri = Uri.parse(forecast['image']['url']); // JsonObjectからそのイメージのURLを取得
    img.src="/weather/lwws?image=${uri.pathSegments.last}"; // そこからリンク・アドレスを作ってイメージ要素に付加
    tdElement.children = [img]; // <td>要素の子供としてイメージ要素<img>を指定
    document.getElementById('imageCells').append(tdElement); // これを<tr>要素に追加する
  });
```

これで天気のアイコンの列が完成する。

サーバのコード

サーバはサービスの構成の項で示したように、大きくはLWWSのプロキシの機能とファイル・サーバの機能が中心となっている。とりわけプロキシの機能（ここではLWWSインターフェイスとも記してある）は今回初めて登場するので、これを中心に説明する。

LWWSへのサーバからのアクセス

Dart:ioにはHTTPでサーバと交信する用途の為にHttpClientという抽象クラス（日本語に訳したものはこの章の終わりにある）が用意されている。つまりサーバが対象とするウェブ・サービスのクライアントとなる。サーバウェブ・サービスにたいしHttpClientRequestを送信し、その応答をHttpClientResponseとして受理する。

HttpClientは別のHTTPサーバに対しHttpClientRequestを送信し、戻ってくる HttpClientResponseを受信するための一連のメソッドたちを含む。例えば、GETおよびPOST要求に対しget, getUrl, post,およびpostUrlメソッドが使える。

一番シンプルなコードは次のような記述になる：

```
HttpClient client = new HttpClient();
client.getUrl(Uri.parse("http://www.example.com/"))
    .then((HttpClientRequest request) {
      // 必要ならヘッダたちをセット...
      // 必要なら要求オブジェクトに対しデータを書き込む...
      // その後HttpClientRequestのcloseを呼ぶ（Future<HttpClientResponse>が返される）
      ...
      return request.close();
    })
    .then((HttpClientResponse response) {
      // Process the response.
      ...
    });
```

- getUrlメソッドはサーバにGET要求をするためのオブジェクトを生成し、サーバとの接続を行う
- これが完了したら、HTTP要求をそのサーバに送るためのHttpClienRequestのオブジェクトが渡されるので、そのオブジェクトに対しヘッダやボディ部をセットする
- HttpClientRequestのcloseメソッドはHTTP要求をサーバに送信する（ヘッダ部は先に送信されている場合がある）。デフォルトではボディ部はGZIP圧縮して送信される
- その後サーバからの応答を受信したらFuture<HttpClientResponse>が返されるので、thenで受けて受信した応答の処理を行う

サーバからLWWSのような認可や登録を必要としないウェブ・サービスをアクセスする一般的なコードは次のようになる。このコードはserver/bin/lwws_access_test.dartとしてこのアプリケーションのなかに同梱してあるので、これを直接自分のIDEから実行させて試して頂きたい。

lwws_access_test.dart

```
/**
 * Sample to access a web service server (no credentials)
 * Gets Tokyo area weather forecast from LWWS as jsonObject
 */

import 'dart:io';
import 'dart:convert';

const host = "weather.livedoor.com/forecast/webservice/json/v1";
const cityCode = 130010; // Tokyo

main() {
  HttpClient client = new HttpClient();
  var bodyStr = '';
  client.getUrl(Uri.parse("http://${host}?city=${cityCode}"))
      .then((HttpClientRequest request) {
        return request.close();
      })
      .then((HttpClientResponse response) {
        // Process the response.
        response.transform(UTF8.decoder).listen((bodyChunk) {
          bodyStr = bodyStr + bodyChunk;
         }, onDone: (){
          // handle data
          print('***headers***\n${response.headers}');
          var jsonObj = JSON.decode(bodyStr);
          print('***bodyString***\n$bodyStr');
          print('***jsonObject***\n$jsonObj');
```

```
        print('**forecasts***\n${jsonObj["forecasts"]}');
        print('***description***\n${jsonObj["description"]}');
      });
    });
}
```

幾つかのポイントを示すと:

- サーバからの応答ヘッダにはcontent-type: application/json; charset=utf-8となっていて、UTF-8でエンコードされているので、transform(UTF8.decoder)でデコードしてとりだす。
- またボディ部はtransfer-encoding: chunkedとヘッダに記されているように、チャンクで送信されてくるので、これを文字列として連結する必要がある。
- 最終的にデータはonDoneのコールバックで得られる。
- JSON.decodeメソッドはJsonオブジェクトと呼ばれるMapで得られる。

具体的にserver.dartのなかでは次のような関数として記述されている:

```
Future<List<int>> _getBytes(Uri uri, HttpRequest request) {
  try {
    var completer = new Completer();
    HttpClient client = new HttpClient();
      List<int> bodyBytes = [];
      client.getUrl(uri)
        .then((HttpClientRequest req) {
          return req.close();
          })
        .then((HttpClientResponse res) {
          res.listen((bodyChunk) {
            bodyBytes.addAll(bodyChunk);
          }, onDone: (){
            if (res.statusCode == HttpStatus.OK){
              if (LOG_REQUESTS)
                log('LWWS interface : received response : $uri with status = ${res.statusCode}');
              completer.complete(bodyBytes);
            } else notFoundHandler.onRequest(request); // not a status OK (200)
          });
        });
      return completer.future;
  } catch (err, st) {
    log('LWWS Handler getLwws error : ${err.toString()}\n${st}');
  }
}
```

この関数もFutureを返し、非同期で進行する。この関数ではイメージ・ファイルも受け取るの為、バイト列としてそのままクライアントに転送しなければならないので、UTF-8変換することなくFuture<List<int>>の形で結果をわたす。また200（OK）以外の応答が帰ってきたときは、404（NOT FOUND）応答をクライアントに返している。

24.4節　　クライアント側だけで済むアプリケーション

いわゆるマッシュアップにはサーバ側で行うもの(server side mashup)とクライアント（ブラウザ）側で行うもの(client side mashup)がある。dart:htmlライブラリは極めて強力であり、クライアント側マッシュアップ・アプリケーションの開

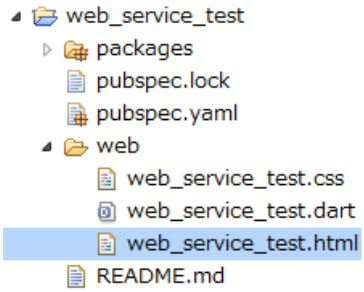
発が容易である。またウェブ・サービスの中には認証やキー取得などの手続きが不要で手軽に使えるものも多い。ここではそのようなウェブ・サービスを使ってクライアント側だけで済むアプリケーションのサンプルを紹介する。

このサンプルはデザインは配慮されていなく、なるべくユーザが理解し活用しやすいようにシンプルなものとなっている。

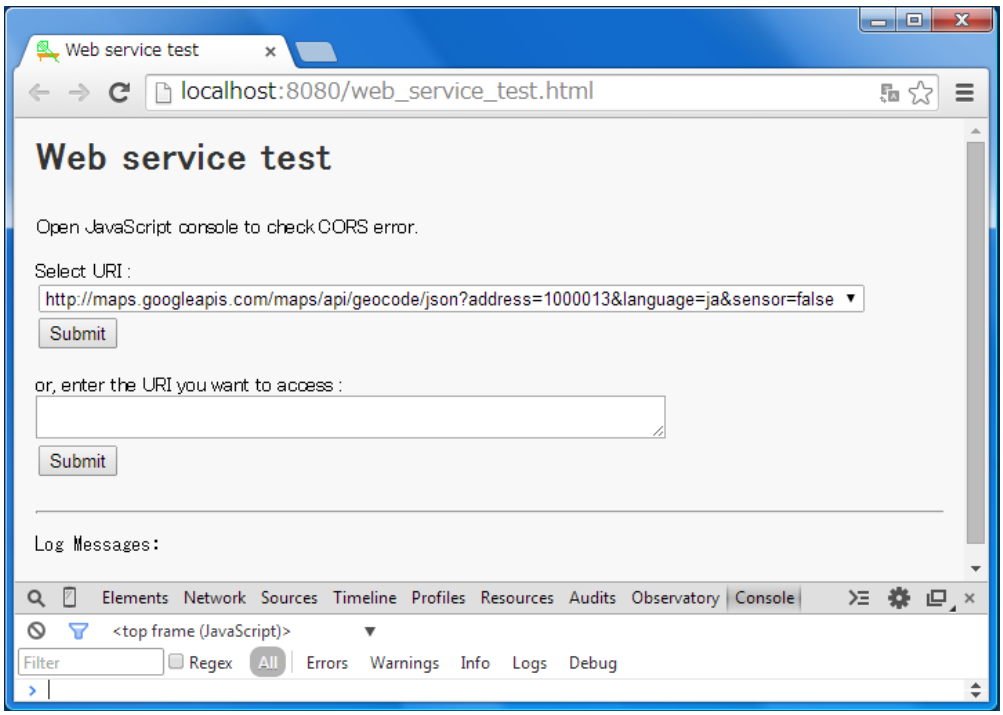
クライアント側で使えるウェブ・サービスの確認

ブラウザからあるウェブ・サービスにアクセスして、必要なデータが取得できるか、あるいはそのデータの構成はどうなっているかを調べる必要が出よう。そのためのweb_service_testというアプリケーションを紹介する。このアプリケーションはdart_code_samplesのなかの\apps\web_service_testである。

1. dart_code_samplesのDownload ZIPをクリックしてダウンロードする
2. これを解凍する
3. 自分のIDEでFile→Open Existing Folderで展開されたフォルダの中の\apps\web_service_testというフォルダを選択する
4. Tools→ Pub Getを実行して必要なライブラリを取り込むと以下のように展開される



5. web_service_test.htmlを選択、右クリックでOpen in BrowserでDartiumまたは通常のブラウザでこれを実行すると、ブラウザには初期画面が表示される
6. 更にブラウザ上でCtrlとShiftを押しながらJをクリックしてJavaScriptコンソールを開くので、下図のように配置する。これはCORSエラーの情報がここでしか得られない為である。



Select URI:と表示された選択メニューには幾つかのウェブ・サービスのAPIにアクセスするURIが登録されている。ここではすでにデフォルトとして"http://maps.googleapis.com/maps/api/geocode/json?address=1000013&language=ja&sensor=false"というURIが選択されてるので、そのままそのメニューの下のSubmitボタンをクリックすると以下のようなログが得られる筈である:

```
URI : http://maps.googleapis.com/maps/api/geocode/json?address=1000013&language=ja&sensor=false
received response from the server
response text : {
   "results" : [
      {
         "address_components" : [
            {
               "long_name" : "100-0013",
               "short_name" : "100-0013",
               "types" : [ "postal_code" ]
            },
            {
               "long_name" : "霞が関",
               "short_name" : "霞が関",
               "types" : [ "sublocality_level_1", "sublocality", "political" ]
            },
            {
               "long_name" : "千代田区",
               "short_name" : "千代田区",
               "types" : [ "locality", "political" ]
            },
                                       ＊＊＊途中省略＊＊＊
               "southwest" : {
                  "lat" : 35.6692782,
                  "lng" : 139.7434881
               }
            }
         },
         "types" : [ "postal_code" ]
      }
   ],
   "status" : "OK"
}

jsonObject : {results: [{address_components: [{long_name: 100-0013, short_name: 100-0013, types: [postal_code]},
{long_name: 霞が関, short_name: 霞が関, types: [sublocality_level_1, sublocality, political]}, {long_name: 千代田区,
short_name: 千代田区, types: [locality, political]}, {long_name: 東京都, short_name: 東京都, types:
[administrative_area_level_1, political]}, {long_name: 日本, short_name: JP, types: [country, political]}],
formatted_address: 〒100-0013, 日本, geometry: {bounds: {northeast: {lat: 35.6777019, lng: 139.7562635}, southwest:
{lat: 35.6692782, lng: 139.7434881}}, location: {lat: 35.6752772, lng: 139.7525236}, location_type: APPROXIMATE,
viewport: {northeast: {lat: 35.6777019, lng: 139.7562635}, southwest: {lat: 35.6692782, lng: 139.7434881}}}, types:
```

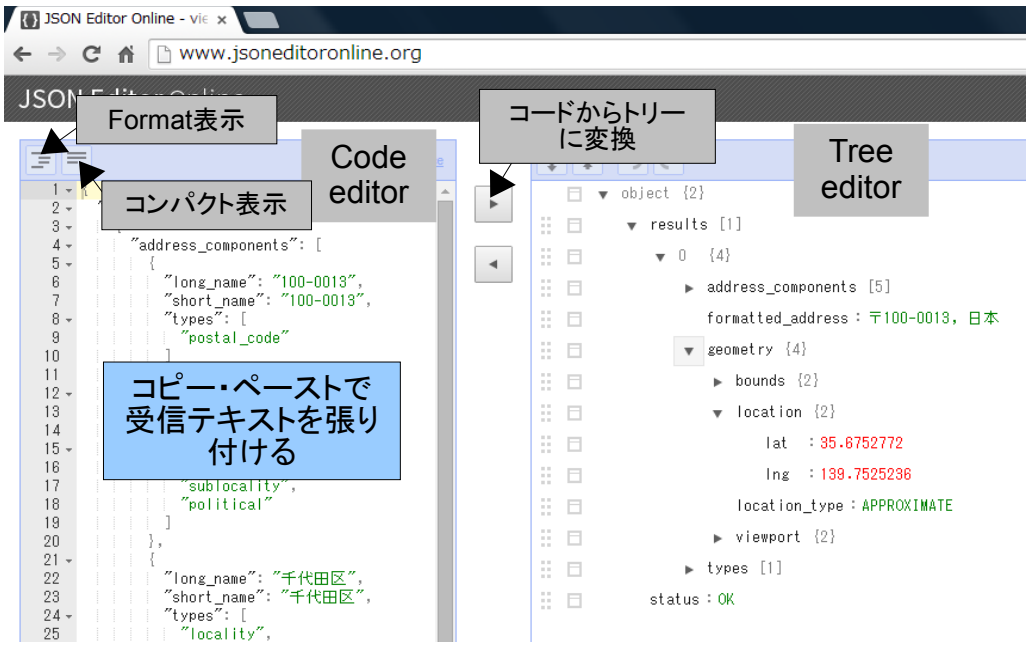
```
[postal_code]}], status: OK}
```

これはGoogleのGeocoding API（住所から緯度経度を取得するサービス）にたいし郵便番号（100-0013）を与えまたJSONと日本語を指定してデータを要求してえられた応答である。

選択メニューに登録されていないURLを試す時はor, enter the URI you want to access :と記されたテキスト・ボックスにそのURLを入力してその下のSubmitボタンをクリックする。この際?以降のクエリ文字列はURLエンコードされていなければならないことに注意のこと。

得られたJSONテキストをDartのJsonObject（即ちMapオブジェクト）に変換した値がjsonObject : 以降に表示されている。複雑なJsonObjectオブジェクトの構成を調べたいときはこのままでは大変である。

このような場合は、オンラインのJSONエディタを使うと便利である。ここではJSON Editor Onlineを使用する。



これはログのなかのresponse text :以降をコピーし左側のCode editorパネルに張り付け、そのパネルの左側のFormat表示ボタンをクリックしたものである。右側はTree editorのパネルで、これらのパネルの中央にある右向き矢印ボタンを押すと左のパネルのデータが反映される。これを見れば次の構成であることが判る：

- objectはresultsとstatusの2つの要素からなるMapである
- resultsは一つの要素からなるListである
- results[0]はaddress_components, formatted_address, geometry,typesの4つの要素からなるMapである
- address_componentsは5個の要素からなるListであって、各要素はlong_name, short_name, typesの3要素からなるMapである
- 更にtypesはpostal_codeというMapを含んだListである
- 等々

正しいURLを入力したにも関わらず結果がなにも表示されない場合がある。選択メニューの中にhttp://weather.livedoor.com/forecast/webservice/json/v1?city=400040というURLがあるので試して見られたい。この場合JavaScriptコンソールには次のように表示される。

```
XMLHttpRequest cannot load http://weather.livedoor.com/forecast/webservice/json/v1?
city=400040. No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'http://localhost:8080' is therefore not allowed access.
```

これはCORSアクセス制限に引っかかった為である。なお、type '_XMLHttpRequestProgressEvent' is not a subtype of type 'Error' of 'error'.というDartのエラー表示がされるが、これはAPIのバグによるもので、現在バグ報告をしてある。

OpenWeatherMapを使ったサンプル

OpenWeatherMapはグローバルな天気情報のウェブ・サービスである。日本語に対応していないので日本ではあまり普及していないが、グローバルには良く利用されるサイトである。

- 20万都市と任意の緯度経度での現在の天気情報と15日間までの予報
- 40,000観測局のデータ(歴史的データを含む)を蓄積
- 気象地図と衛星地図を提供

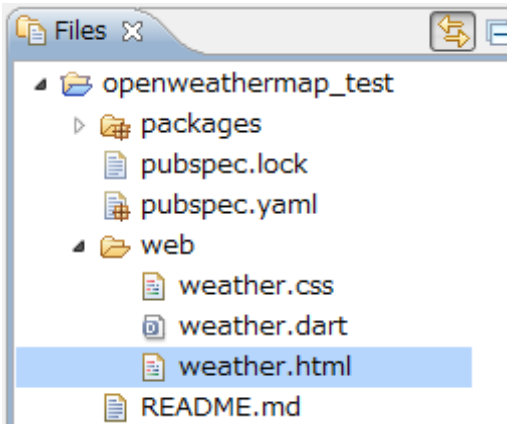
APIに関する情報はここから得られる。

2015年10月9日からAPIを利用するには無料の課金プランを含めてこのAPPIDを取得せねばならなくなった。従ってこのAPIを使うサービスを開設するには各自OpenWeatherMapのAPIのページからAPPIDを取得し、各自のプログラムに書きまねばならなった。APPIDの取得法は:

1. OpenWeatherMapのAPIのページの16 day / daily forecastの下のSubscribeボタンをクリックする
2. Get API key and Strartのボタンを押してSign upのボタンをクリックし、登録ページに進む
3. ユーザ名、メールアドレス、及びパスワード(2回)を入力し、I agree to the Terms of Service and Privacy Policyにチェックを入れ、Create Accountボタンをクリックする
4. My Homeのページが開くので、User settingsでユーザ名等を書き込む
5. API keysのタブを見ると16進32桁からなるAPIキーが表示されているはずである(課金のデフォルトは無料プランになっている)

サンプルはopenweathermap_testという名前でdart_code_samplesリポジトリに含まれている。まずこれを試して頂きたい。このサンプルは教材のため筆者が取得したAPPIDが使われているので、このままでも機能するが、読者は自分のAPPIDを取得しておいて、それで動作してみることをお勧めする。

1. dart_code_samplesのDownload ZIPをクリックしてダウンロードする
2. これを解凍する
3. 自分のIDE上でFile→Open Existing Folderで展開されたフォルダの中の\apps\openweathermap_testというフォルダを選択する
4. このアプリケーションを自分のIDE上で開く。またwebフォルダにはpubspec.yamlファイルが存在するので、「外部パッケージの取り込み」の節を参照して、依存ライブラリを取り込む。そうするとIDE上には以下のように展開される



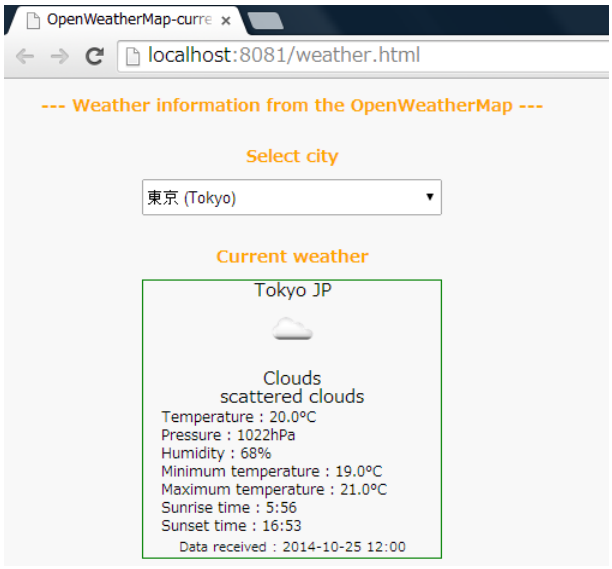
5. このままでも動作するがweather.dartコードを開き、できれば以下の箇所のAPPIDを自分が取得したAPPIDに置き換える:

```
final APPID = '1a1a27e8eccf4158d52c5763415c8121';
```

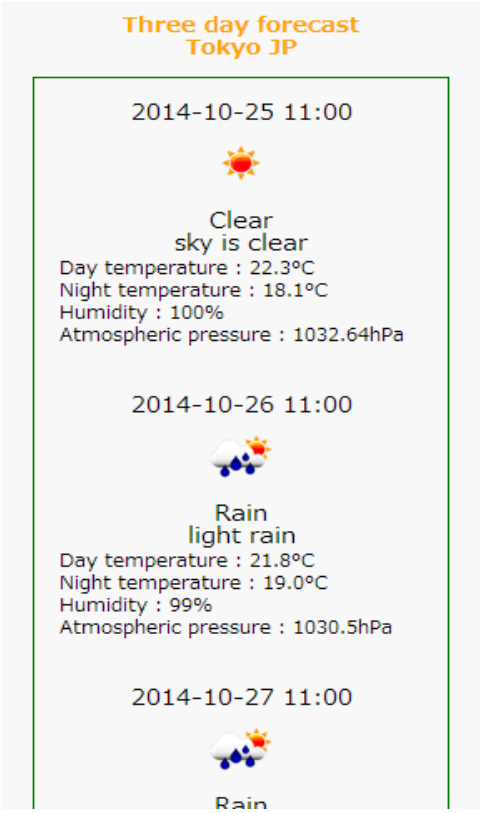
6. weather.htmlを選択、右クリックでRun in DartiumまたはRun as JavaScriptでこれを実行すると、ブラウザには初期画面が表示される。これはあらかじめ選択されている東京の現在の天気と3日間の予報である。

画面の最初のSelect Cityの選択メニューは知りたい都市を選択するもので、東京があらかじめ選択されている。

次のCurrent Weatherの枠は現在の天気情報である。このデータの詳細はここにある。



次のThree day forecastの枠は3日間の予測であり、このデータの詳細はここにある。表示されている日時はローカル時間である。Bostonを選択しても日本時間で表示されることに注意。



ここでは3日間のデータであるが、これはweather.dartのなかの3という値を最大16まで変更可能である。

```
final int numberOfDays = 3;
```

weather.dartのポイント

このプログラムのなかで2つのDOM処理による画面操作関数が核となっている:

- currentDom(Map json):現在の気象データの表示
- forecastDom(Map json):15日間までの予測データの表示

これらはともにJsonObjectを引数としている。

currentDom(Map json)に関しては、前節のclient.dartと似ており、特に説明する必要も無かろう。

forecastDom(Map json)は予測日数が変化する(final int numberOfDays = 3;の数字を15まで増やすことができる)ので、table.insertRow(n)で表の行を挿入する方式をとっている。nの値は0から4まで順番に増やしてゆく。最終日分のデータが終わったら前日分が再度0番目の行として追加される。従ってこの表は次のように終わりから順に下に伸びてゆくことになる。

```
for (int i = numberOfDays-1; i >= 0; i--) {
```

HTMLページを再ロードすることなく別の都市のデータを表示する際は、以前の行を削除しておかないとこの表はどんどん下に伸びていってしまう。

```
while (table.rows.length != 0) table.deleteRow(0);
```

という行がその作業を行っている。

24.5節　　関連APIの和訳

Dart:html.HttpRequest

オリジナル・ドキュメント: https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-dom-html.HttpRequest

HttpRequest class

あるURLからデータを取得する為のクライアント側のXHR(XMLHttpRequest)要求で、以前はXMLHttpRequestとして知られていたもの。

HttpRequestはHTTPおよびFTPプロトコルでデータを取得するのに使え、AJAXスタイルのページj更新に有用である。

JSONフォーマットされたファイルのようなテキスト・ファイルの中身を取得する最もシンプルな方法は、getStringメソッドを使うことである。例えば、以下のコードはJSONファイルの中身を取得しその長さをプリントする:

```
var path = 'myData.json';
HttpRequest.getString(path)
    .then((String fileContents) {
      print(fileContents.length);
    })
    .catchError((Error error) {
      print(error.toString());
    });
```

他のサーバからのデータの取得について。

セキュリティの理由から、ブラウザは組み込みアプリケーションからの要求には制限を課している。このクラスのデフォルトの振る舞いでは、この要求をするコードは要求されているリソースと同じオリジン(ドメイン名、ポート番号、およびアプリケーション層プロトコル)から渡されねばならない。上記の例では、myData.jsonファイルは取れを使うアプリケーションと共に置かれて居なければならない。この制約はCORSヘッダまたはJSONPを使って回避し得る。

実装	
HttpRequestEventTarget	
コンストラクタ	
HttpRequest()	どのタイプの要求(GET、POST等)にも使える汎用コンストラクタ。 この呼び出しはopenと一緒にして使われる: `var request = new HttpRequest();`

	`` request.open('GET', 'http://dartlang.org'); request.onLoad.listen((event) => print('Request complete ${event.target.reponseText}')); request.send(); ``` これは以下と等価であるが、ややくどい記述である: ``` HttpRequest.getString('http://dartlang.org').then((result) => print('Request complete: $result')); ```
メソッド	
void **abort**()	現行の要求を止める。 readyState がHEADERS_RECIEVED または LOADINGの状態である時のみこの要求は停止できる。このメソッドが送信中でないときは、このメソッドは効果を持たない。
void **addEventListener**(String type, EventListener listener, [bool useCapture])	EventTargetから継承
bool **dispatchEvent**(Event event)	EventTargetから継承
String **getAllResponseHeaders**()	(安定していない) ある要求からの総ての応答ヘッダをとりだす。 ヘッダを受信していないときはnullを返す。マルチパート要求の場合は、getAllResponseHeadersはその要求の現行のパートに対する応答ヘッダを返す。 一般的な応答ヘッダのリストに対すしてはHTTP応答ヘッダを見ること。
String **getResponseHeader**(String header)	(安定していない) 指定したヘッダをもった応答を返し、見つからないときはnullを返す。 一般的な応答ヘッダのリストに対すしてはHTTP応答ヘッダを見ること。
noSuchMethod(Invocation invocation)	Interceptorから継承 ユーザがあるオブジェクトに対し存在しないメソッドを呼んだときにnoSuchMethodが呼び出される。この呼び出しの名前と引数たちは InvocationのなかのnoSuchMethodに渡される。noSuchMethod が値を返す時は、その値がオリジナルの呼び出しの結果となる。 noSuchMethodのデフォルトの振る舞いは NoSuchMethodErrorをスローすることである。
void **open**(String method, String url, {bool async, String user, String password})	この要求をするときはurlとmethodを指定する。 デフォルトでは、この要求は非同期で行われ、パスワード認証情報を持たない。もしasyncがfalseなら、この要求は同期的に送信される。 現行のアクティブな状態の要求にopenを再度呼ぶことはabortを呼ぶことと等価で

	ある。 注意:ほとんどのシンプルなHTTP要求はgetString、request、requestCrossOrigin、またはpostFormData のメソッドを使って達成される。このopenメソッドを使うのは、細かなコントロールが必要なより複雑なHTTP要求にの場合のみに意図されている。
void **overrideMimeType**(String override)	その要求にとって必要な特定のMIMEタイプ（例えばtext/xml のような）を指定する。 この値はその要求が送信される前にセットされねばならない。一般的な応答ヘッダのリストに対すしてはHTTP応答ヘッダを見ること。
void **removeEventListener**(String type, EventListener listener, [bool useCapture])	EventTargetから継承
void **send**([data])	与えられたdataでその要求を送信する。 注意:殆どのシンプルなHTTP要求はgetString、request、 requestCrossOrigin、またはpostFormData のメソッドを使って達成される。このopenメソッドを使うのは、細かなコントロールが必要なより複雑なHTTP要求にの場合のみに意図されている。 他のリソース： XMLHttpRequest.send from MDN.
void **setRequestHeader**(String header, String value)	あるHTTP要求ヘッダの値をセットする。 このメソッドはこの要求がopenした後で且つ該要求が送信される前に呼ばれねばならない。 同じヘッダに対し複数回呼ぶとそれらは単一ヘッダとしてまとめられる。 他のリソース： XMLHttpRequest.setRequestHeader method from MDN. setRequestHeader() method-from W3C.
String **toString**()	EventTargetから継承 このオブジェクトの文字列表現を返す。
演算子	
bool ==(other)	EventTargetから継承 対等性演算子。 総てのObjectにたいする振る舞いはthisとotherが同じオブジェクトであるときに限りtrueを返す。 あるクラス上で別の対等性の関係を規定するときはこのメソッドをオーバライドする。オーバライドするほうのメソッドはそれでも対等性の関係を保持しなければならない。即ち、以下のごとくあらねばならない: Total: それは総ての引数に対しブール値を返さねばならない。決してスローしたりnullを返してはいけない。

	• Reflexive: 総てのオブジェクトoに対し、o == o はtrueでなければならない。 • Symmetric: 総てのオブジェクトo1とo2に対し、 o1 == o2と o2 == o1はともにtrueでもともにfalseでもあってはいけない。 • Transitive: 総てのオブジェクトo1, o2, 及び o3にたいし、もし o1 == o2 及び o2 == o3 がtrueなら、o1 == o3 もtrueでなければならない。a このメソッドは近いとともに一貫していなければならないので、2つのオブジェクトは時間とともに変化してはいけないか、あるいは少なくとも一つのオブジェクトに変化を与えられたときにのみ変化する。 あるサブクラスがこの演算子をオーバライドするときは、一貫性を保つためにはhashCodeもオーバライドしなければならない。
Staticメソッド	
static Future<String> **getString**(String url, {bool withCredentials, Function void onProgress(ProgressEvent e)})	指定したurlむけのGET要求を生成する。 この要求が成功するにはサーバの応答は text/ mimeタイプでなければならない。 これはrequestと似ているがテキストの中身を返すHTTP GET 要求に特化している。 クエリ・パラメタを付加するには、urlのあとに?を付けそのあとにそれらを追加する。その際各キーと値は=で結び付け、各キーと値のペアを&で分離する。 `var name = Uri.encodeQueryComponent('John');` `var id = Uri.encodeQueryComponent('42');` `HttpRequest.getString('users.json?name=$name&id=$id')` `  .then((HttpRequest resp) {` `  // Do something with the response.` `});` See also: request
static Future<HttpRequest> **postFormData**(String url, Map<String, String> data, {bool withCredentials, String responseType, Map<String, String> requestHeaders, Function void onProgress(ProgressEvent e)})	指定したdataをformデータとして付したPOST要求をサーバにする。 これはほぼgetStringのPOST等価なメソッドである。このメソッドはより広いブラウザ・サポートを持つ FormDataオブジェクトを送信することと等価であるがStringの値に限定されている。 もしdataが与えられたら、そのキー/値のペアは encodeQueryComponentでURIエンコードされ、HTTPクエリ文字列に変換される。 指定されていない限り、このメソッドは以下のヘッダを追加する: `Content-Type: application/x-www-form-urlencoded; charset=UTF-8` 以下はこのメソッドの使用例である: `var data = { 'firstName' : 'John', 'lastName' : 'Doe' };` `HttpRequest.postFormData('/send', data).then((HttpRequest resp)` `{` `  // Do something with the response.` `});`

	See also: request
static Future<HttpRequest> **request**(String url, {String method, bool withCredentials, String responseType, String mimeType, Map<String, String> requestHeaders, sendData, Function void onProgress(ProgressEvent e)})	指定したurlに対する要求を生成し送信する。 デフォルトではrequestはHTTP GET要求を行うが、methodパラメタを指定することで他のメソッド(POST, PUT, DELETE, etc)も使用可能である。(POST要求だけの場合のpostFormDataも参照のこと) 返されるFutureはその応答が得られたら完了する。 指定されていれば、sendData は ByteBuffer, Blob, Document, String,またはFormDataの形式のデータをHttpRequestに載せて送信する。 指定されていれば、responseType はその要求に対する欲しい応答書式をセットする。デフォルトではそれはStringであるが、'arraybuffer', 'blob', 'document', 'json', または'text'が指定できる。更なる情報は responseType を参照のこと。 withCredentialsパラメタは(既に)クッキーのようなクレデンシャルがヘッダにセットされているか、その要求にauthorization ヘッダを指定するかしなければならない。クレデンシャルを使うときの注意点: • クレデンシャルの使用はクロス・オリジンの要求に対しのみ有用である。 • urlのAccess-Control-Allow-Originヘッダはワイルド・カード(*)を含むことはできない。 • urlのAccess-Control-Allow-Credentialsヘッダはtrueにセットされていなければならない。 • Access-Control-Allow-Credentialsヘッダがまだtrueにセットされていないときは、getAllRequestHeadersを呼んだときに総ての応答ヘッダたちのみが返される。 これは上記の getStringサンプルと等価なものである: `var name = Uri.encodeQueryComponent('John');` `var id = Uri.encodeQueryComponent('42');` `HttpRequest.request('users.json?name=$name&id=$id')` `  .then((HttpRequest resp) {` `  // Do something with the response.` `});` 以下はFormDataでフォーム全部をサブミットする例である: `Here's an example of submitting an entire form with FormData.` `var myForm = querySelector('form#myForm');` `var data = new FormData(myForm);` `HttpRequest.request('/submit', method: 'POST', sendData: data)` `  .then((HttpRequest resp) {` `  // Do something with the response.` `});` file:// URIsへの要求はそのマニフェストでしかるべき許可をしたChrome拡張でのみサポートされる。file:// URIsへの要求はまた決して失敗は起きず、例えそのファイルが見つからないときでも常に成功で完了する。

	See also: authorization headers.
static Future<String> **requestCrossOrigin**(String url, {String method, String sendData})	(試験的) 指定したURLにクロス・オリジン要求を行う。 このAPIは IE9で動作するサブセットである。もしIE)のクロス・オリジン・サポートが必要ないときは、その代りrequestを使うべきである。
属性	
int get **hashCode**	EventTargetから継承 このオブジェクトのハッシュ・コードを取得する。
Events get **on**	EventTargetから継承 これはイベント・ストリームに対する使いやすいアクセサであって、明示的なアクセサがないときにのみ使われるべきである。
Stream<ProgressEvent> get **onAbort**	(試験的) HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたabortイベントのストリーム。
Stream<ProgressEvent> get **onError**	(試験的) HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたerrorイベントのストリーム。
Stream<ProgressEvent> get **onLoad**	(試験的) HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたloadイベントのストリーム。
Stream<ProgressEvent> get **onLoadEnd**	(試験的) Supported on: Chrome, Firefox, Ie 10, Safari HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたloadendイベントのストリーム。
Stream<ProgressEvent> get **onLoadStart**	(試験的) HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたloadstartイベントのストリーム。
Stream<ProgressEvent> get **onProgress**	(試験的) Supported on: Chrome, Firefox, Ie 10, Safari HttpRequestEventTargetから継承 このHttpRequestEventTargetで扱われたprogressイベントのストリーム。
Stream<ProgressEvent> get **onReadyStateChange**	このHttpRequestEventTargetで扱われたreadystatechange イベントのストリーム。 イベント・リスナたちはこのHttpRequestオブジェクトの readyStateの値が変わった時毎に通知される。
Stream<ProgressEvent> get	(試験的)

<table>
<tr><td>onTimeout</td><td>HttpRequestEventTargetから継承

このHttpRequestEventTargetで扱われたtimeoutイベントのストリーム。</td></tr>
<tr><td>final int readyState</td><td>この要求の現在の状態を示すインディケータ:
<table>
<tr><th>Value</th><th>State</th><th>意味</th></tr>
<tr><td>0</td><td>unsent</td><td>open()がまだ呼ばれていない</td></tr>
<tr><td>1</td><td>opened</td><td>send()が未だ呼ばれていない</td></tr>
<tr><td>2</td><td>headers received</td><td>sent()が呼ばれている:応答ヘッダとstatusが取得できる</td></tr>
<tr><td>3</td><td>loading</td><td>responseTextが何らかの値を持っている</td></tr>
<tr><td>4</td><td>done</td><td>要求が完了している</td></tr>
</table></td></tr>
<tr><td>get response</td><td>その要求からの応答として受信したデータ。
データはString, ByteBuffer, Document, Blob, or json (also a String)であり得る。
nullは要求が失敗したことを示す。</td></tr>
<tr><td>Map<String, String> get responseHeaders</td><td>総ての応答ヘッダたちをキー/値のMapとして返す。
同じヘッダ・キーに複数の値があるときは一つにまとめられ、カンマとスペースで区切られる。

See: http://www.w3.org/TR/XMLHttpRequest/#the-getresponseheader()-method</td></tr>
<tr><td>final String responseText</td><td>Stringの形式の応答で、空のStringの時は失敗したことを示す。</td></tr>
<tr><td>String responseType</td><td>サーバに対し欲しい応答のフォーマットを告げるためのString。

デフォルトはStringだが、それ以外に'arraybuffer', 'blob', 'document', 'json', 'text'のどれかを選択できる。

同期要求を行っている際に responseTypeがセットされているときに一部のブラウザでは NSERRORDOMINVALIDACCESS_ERRをスローする。

See also: MDN responseType</td></tr>
<tr><td>final String responseUrl</td><td>(試験的)</td></tr>
<tr><td>final Document responseXml</td><td>要求に対する応答で、失敗したときはnullである。

その応答はresponseType = 'document'でその要求が同期でないときに text/xmlのストリームとして処理される。</td></tr>
<tr><td>Type get runtimeType</td><td>Interceptorから継承

このオブジェクトの実行時の型を表現する。
Comment inherited from Object</td></tr>
<tr><td>final int status</td><td>その要求に対する応答の結果コード(200、404等)

See also: Http Status Codes</td></tr>
<tr><td>final String statusText</td><td>応答ステータスの文字列(such as \"200 OK\")。
See also: Http Status Codes</td></tr>
</table>

int **timeout**	(試験的) ある要求が自動的に終了するまでの時間長。timeを超過したらTimeoutEvent で通知される。timeを0にセットすると、この要求はタイムアウトを起こさない。 Other resources XMLHttpRequest.timeout from MDN. The timeout attribute from W3C
final HttpRequestUpload **upload**	(安定していない) 該要求の進捗状況を追跡するためのリスナを保持できるEventTarget。ファイアするイベントは HttpRequestUploadEventsのメンバとなる。
bool **withCredentials**	クロス・サイトの要求がクッキーのようなクレデンシャルを使うかまたはauthorizationヘッダを使うかするときはtrueとする。それ以外はfalseである。 同じサイトへの要求ではこの値は無視される。
Static属性	
static const int **DONE**	
static const int **HEADERS_RECEIVED**	
static const int **LOADING**	
static const int **OPENED**	
static const int **UNSENT**	
static const EventStreamProvider<ProgressEvent> **readyStateChangeEvent**	必ずしもHttpRequestのインスタンスでないイベント・ハンドラに readystatechangeイベントが使えるようにするよう設計されたstaticなファクトリ。 使用法はEventStreamProviderを参照のこと
static bool get **supportsCrossOrigin**	現行のプラットホームがクロス・オリジン要求をするのに対応しているかどうかをチェックする。 注意:たとえクロス・オリジン要求に対応していても、相手のサーバがCORS要求に対応していなければその要求は失敗する。
static bool get **supportsLoadEndEvent**	現行のプラットホームが LoadEndイベントに対応しているかどうかをチェックする。
static bool get **supportsOverrideMimeType**	現行のプラットホームが overrideMimeType メソッドに対応しているかどうかをチェックする。
static bool get **supportsProgressEvent**	現行のプラットホームがProgressイベントに対応しているかどうかをチェックする。

Dart:io.HttpClient

オリジナル・ドキュメント:https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-io.HttpClient

HttpClient abstract class

HTTPプロトコルを使ってウェブ・ページのようなコンテントをあるサーバから受信するクライアント。

HttpClientはあるHTTPサーバに対し HttpClientRequestを送信し、戻ってくる HttpClientResponseを受信するための一連のメソッドたちを含む。例えば、GETおよびPOST要求に対しget, getUrl, post,およびpostUrlメソッドが使える。

シンプルなGET要求をする例:

getUrl 要求は2つのFutureがトリガとなる2段階のプロセスである。HttpClientRequestで最初のFutureが完了したときは、その背後のネットワーク接続が確立したがデータはまだ送信されていない状態である。最初のFutureのコールバック関数の中で、HTTPヘッダたちとボディ部がその要求上でセットできる。その要求オブジェクトに最初に何かを書き込んだとき、あるいはcloseを呼んだときにその要求はサーバに送信される。

そのサーバからHTTP応答が受信したときは、closeで返された第2のFutureがHttpClientResponseオブジェクトで完了する。このオブジェクトが該応答のヘッダたちおよびボディへのアクセスを提供する。このボディはHttpClientResponseで実装されているストリームによって取得できる。もしボディ部が存在していれば、それは読み出しされねばならず、そうでないとリソースのリークをもたらす。ボディを使わないときはHttpClientResponse.drainを使うことを検討すること。

```
HttpClient client = new HttpClient();
client.getUrl(Uri.parse("http://www.example.com/"))
    .then((HttpClientRequest request) {
      // Optionally set up headers...
      // Optionally write to the request object...
      // Then call close.
      ...
      return request.close();
    })
    .then((HttpClientResponse response) {
      // Process the response.
      ...
    });
```

HttpClientRequestのFutureは getUrlおよびopenのようなメソッドによって生成される。

ヘッダ

総てのHttpClientの要求はデフォルトでは次のヘッダ行が付加される:

```
Accept-Encoding: gzip
```

これにより該HTTPサーバはそのボディにたいし可能ならgzip圧縮をかける。こうしてほしくないときは、Accept-Encodingヘッダをなにか別のものに変更する。応答のgzip圧縮をオフにするときは、このヘッダをオフにする:

```
request.headers.removeAll(HttpHeaders.ACCEPT_ENCODING)
```

HttpClientを閉じる

HttpClientは出来うる限り複数の要求の為に再利用する為の永続した接続およびキャッシュ網接続に対応している。このことはある要求が完了した後もある時間ネットワーク接続を保持され得ることを意味する。強制的にこの接続をシャットダウンし、アイドル状態のネットワーク接続を閉じるには、HttpClient.close を使う。

プロキシのオンとオフ

デフォルトではHttpClient はその環境で使えるプロキシ構成を使用する、findProxyFromEnvironmentメソッドを参照のこと。プロキシ使用をオフにするにはfindProxy属性をnullにする:

```
HttpClient client = new HttpClient();
client.findProxy = null;
```

継承	
Object	
コンストラクタ	
HttpClient()	
メソッド	
void **addCredentials**(Uri url, String realm, HttpClientCredentials credentials)	HTTP要求を認可するために使われるクレデンシャル（証明書）たちを付加する。
void **addProxyCredentials**(String host, int port, String realm, HttpClientCredentials credentials)	HTTPプロキシたちを認可するために使われるクレデンシャルたちを付加する。
void **close**({bool force: false})	該HTTPクライアントを閉じる。forceがfalseのとき（デフォルト）は総てのアクティブな接続が完了するまでこのHttpClientは活きつづける。forceがtrueのときは、アクティブな接続は閉じられ、総てのリソースが解放される。これらの閉じた接続たちは該接続がシャットダウンしたことを示す為のonErrorコールバックを受ける。双方の場合においてシャットダウンを呼んだ後で新規の接続を確立しようとすると例外が生起する。
Future<HttpClientRequest> **delete**(String host, int port, String path)	HTTPのDELETEメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。 詳細はopenを参照のこと。
Future<HttpClientRequest> **deleteUrl**(Uri url)	HTTPのDELETEメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
Future<HttpClientRequest> **get**(String host, int port, String path)	HTTPのGETメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。

	詳細はopenを参照のこと。
Future<HttpClientRequest> **getUrl**(Uri url)	HTTPのGETメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
Future<HttpClientRequest> **head**(String host, int port, String path)	HTTPのHEADメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。 詳細はopenを参照のこと。
Future<HttpClientRequest> **headUrl**(Uri url)	HTTPのHEADメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
Future<HttpClientRequest> **open**(String method, String host, int port, String path)	HTTP接続を行う。 使用するHTTPメソッドはmethodで指定され、このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。 要求の為のHostヘッダは host:portという値でセットされる。これはこの要求が送信される前であればHttpClientRequestインターフェイスを介してオーバライドされ得る。 注意：もしhostがIPアドレスの時は、これはHostヘッダにセットされたままとなる。 HTTPトランザクション中のイベントたちのシーケンス、およびfutureたちによって返されるオブジェクトたちに関する更なる情報は、HttpClientクラスの全体ドキュメンテーションを参照のこと。
Future<HttpClientRequest> **openUrl**(String method, Uri url)	HTTP接続を行う。 HTTPメソッドはmethodで指定し、URLはurlで指定する。 要求の為のHostヘッダは host:portという値でセットされる。これはこの要求が送信される前であればHttpClientRequestインターフェイスを介してオーバライドされ得る。 注意：もしhostがIPアドレスの時は、これはHostヘッダにセットされたままとなる。 HTTPトランザクション中のイベントたちのシーケンス、およびfutureたちによって返されるオブジェクトたちに関する更なる情報は、HttpClientクラスの全体ドキュメンテーションを参照のこと。
Future<HttpClientRequest> **patch**(String host, int port, String path)	HTTPのPATCHメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。

	詳細はopenを参照のこと。
Future<HttpClientRequest> **patchUrl**(Uri url)	HTTPのPATCHメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
Future<HttpClientRequest> **post**(String host, int port, String path)	HTTPのPOSTメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。 詳細はopenを参照のこと。
Future<HttpClientRequest> **postUrl**(Uri url)	HTTPのPOSTメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
Future<HttpClientRequest> **put**(String host, int port, String path)	HTTPのPUTメソッドを使ってHTTP接続を行う。 このサーバはhostとportを使って指定され、パス（フラグメントとクエリが付加されている場合もあり）はpathを使って指定される。 詳細はopenを参照のこと。
Future<HttpClientRequest> **putUrl**(Uri url)	HTTPのPUTメソッドを使ってHTTP接続を行う。 使用するサーバのURLはurlで指定される。 詳細はopenUrl を参照のこと。
staticメソッド	
static String **findProxyFromEnvironment**(Uri url, {Map<String, String> environment})	environment変数で指定されたプロキシ構成から、HTTP接続に使われるプロキシ・サーバを得るための関数。 以下のenvironment変数たちが調べられる: `http_proxy` `https_proxy` `no_proxy` `HTTP_PROXY` `HTTPS_PROXY` `NO_PROXY` http_proxyとHTTP_PROXYは http:// urlsに使われるプロキシ・サーバを指定する。hostname:portを使う。portがないときはデフォルトとして1080が使われる。双方ともにセットされていると小文字のほうが優先する。 https_proxyとHTTPS_PROXYは http:// urlsに使われるプロキシ・サーバを指定する。hostname:portを使う。portがないときはデフォルトとして1080が使われる。双方ともにセットされていると小文字のほうが優先する。

	no_proxyとNO_PROXYはプロキシ・サーバに使わないhostnameたちをカンマで区切ったリストを指定する。例えば"localhost,127.0.0.1"という値は"localhost"と"127.0.0.1"の双方への要求にはプロキシを使わなくする。双方ともにセットされていると小文字のほうが優先する。 このプロキシ解決手段を活かすには、この関数を HttpClient上でfindProxyに代入する。 `HttpClient client = new HttpClient();` `client.findProxy = HttpClient.findProxyFromEnvironment;` システム環境を使いたくないときは、この関数をラップして異なったものを使うことができる: `HttpClient client = new HttpClient();` `client.findProxy = (url) {` `  return HttpClient.findProxyFromEnvironment(` `  url, {"http_proxy": ..., "no_proxy": ...});` `}` もしプロキシが認可を必要とする場合は、usernameと passwordも設定することができる。usernameと passwordも含めるときは username:password@hostname:portという書式を使う。別のやりかたとして、addProxyCredentialsというAPIが認可を要求とするプロキシにクレデンシャルをセットするのに使える。
属性	
set **authenticate**(Function Future<bool> f(Uri url, String scheme, String realm))	そのサイトが認可を要求している場合に呼ばれる関数をセットする。要求されるURLとサーバからのセキュリティ・レルムはurlとrealm変数で渡される。 この関数はFutureを返し、認証が終わった時に完了する。クレデンシャルたちが与えられない場合はFutureがfalseで終了する。クレデンシャルたちが指定されたときは、この関数はtrueのvalueでFutureが完了される前に addCredentialsを使ってこれらを付加しなければならない。 Futureがtrueで完了したときは、この要求は更新されたクレデンシャルたちを使って再度試行される。そうでないときは、応答処理は通常通り継続する。
set **authenticateProxy**(Function Future<bool> f(String host, int port, String scheme, String realm))	プロキシが認可を要求している場合に呼ばれる関数をセットする。使用しているプロキシに関する情報とその認証の為のセキュリティ・レルムはhost、portとrealm変数で渡される。 この関数はFutureを返し、認証が終わった時に完了する。クレデンシャルたちが与えられない場合はFutureがfalseで終了する。クレデンシャルたちが得られるときは、この関数はtrueのvalueでFutureが完了される前に addProxyCredentialsを使ってこれらを付加しなければならない。 Futureがtrueで完了したときは、この要求は更新されたクレデンシャルたちを使って再度試行される。そうでないときは、応答処理は通常通り継続する。
bool **autoUncompress**	この応答のボディ部が自動的に圧縮されるかどうかを取得および設定する。 HTTP応答のボディ部は圧縮し得る。殆どの場合解凍したボディを渡すことが最も都合がよい。従ってデフォルトの振る舞いはボディ部は解凍されることになって

	いる。しかしながらある状況（例えば透過プロキシの実装）では解凍しないストリームを維持することが求められる。 注意：応答からのヘッダは加工されることはない。このことは、自動回答がオンになっていても Content-Lengthヘッダはオリジナルの圧縮されたボディの長さを反映していることを意味する。同様に、Content-Encodingヘッダは圧縮を意味するオリジナルの値を保持している。 注意：自動解凍はContent-Encodingヘッダの値がgzipであるときのみ行われる。 この値が変更された後は、このクライアントが生成する総ての応答にその値が適用される。 自動解凍をオフにするときは、falseをセットする。 デフォルトはtrueである。
set **badCertificateCallback**(Function bool callback(X509Certificate cert, String host, int port))	我々の信頼されるルート認証たちのどれを使っても認可が得られないサーバ認証でセキュアな接続を受け付けるかどうかを判断するコールバック関数をセットする。 セキュアなHTTP要求がこのHttpClientを使ってでき、そのサーバが認可できなかったs-場認証を返したときは X509認証オブジェクトおよびそのサーバのhostnameとportでこのコールバックが非同期で呼ばれる。badCertificateCallback=の値がfalseのときは、その合わない認証は、あたかもそのコールバックがfalseを返したごとく排除される。 もしこのコールバックがtrueを返す時は、そのセキュアな接続は受け付けられ、その要求をしている呼び出しから返されたFuture<HttpClientRequest> は有効なHttpRequestオブジェクトで完了する。このコールバックがfalseを返す時は、Future<HttpClientRequest>は例外で完了する。 試みている接続上で認証できず（bad certificate）が受信されているとき、このライブラリはたとえそれ以来 badCertificateCallbackの値が変わっていたとしても、その要求がなされた時点での badCertificateCallbackの値である関数を呼び出す。
set **findProxy**(Function String f(Uri url))	指定したurlに対しHTTP接続を開くのに使われるプロキシ・サーバを解決するのに使われる関数をセットする。この関数がセットされていないときは、常に直接接続が使われる。 fが返すStringはブラウザPAC (proxy auto-config)スクリプトで使われる書式でなければならない。即ち以下のどれかでなければならない： 直接接続使用の為の `"DIRECT"` またはポートport上のプロキシ・サーバhostを使うための `"PROXY host:port"` 設定はセミコロンで区切られた幾つかの要素を含むことがあり得る。例えば：

	`"PROXY host:port; PROXY host2:port2; DIRECT"` このクラスのstatic関数のfindProxyFromEnvironmentが環境変数に基づくプロキシ・サーバ解決を実装するのに使える。
Duration **idleTimeout**	非アクティブな永続した(keep-alive) 接続のアイドルのタイムアウトの取得および設定。デフォルト値は15秒である。
int **maxConnectionsPerHost**	単一のホストに対するライブな接続の最大数の取得と設定。 この数を増やすと性能を下げ不必要なシステム・リソースの消費をもたらし得る。 これを殺す時はnullをセットする。 デフォルトはnullである。
String **userAgent**	この HttpClientが生成した総ての要求におけるデフォルトのUser-Agentヘッダの値の取得と設定。デフォルトの値はDart/<version> (dart:io)である。 userAgent がnullに設定されているときは、各要求にはデフォルトの User-Agent ヘッダは付加されない。
static属性	
static const int **DEFAULT_HTTPS_PORT**	
static const int **DEFAULT_HTTP_PORT**	

第25章 Googleのウェブ・サービスの為のAPI (googleapis)

Googleは2014年9月22日にGoogleのウェブ・サービスの為のDart言語化したAPIの新しいクライアント・ライブラリが出来たと発表した。これによりGoogleのサービスを使った高度なアプリケーションをDartで書くことが可能となった。

殆どのGoogleのサービスはGoogle Discovery Serviceを使って書かれているRESTスタイル準拠のAPIを有している。これにはApps API（GmailとかDriveのようなサービス）とCloud API（Cloud Datastore とかCloud Storageのようなサービス）がある。

Googleは次の2つのpubライブラリを公開した：

- googleapis　（APIドキュメントはここ）
- googleapis_beta

googleapisパッケージは安定版として使える総てのAPIたちが含まれている。googleapis_betaパッケージのほうは現在ベータ版の状態でLimited Preview（限定プレビュー）プログラムでのみ使えるAPIたちが含まれている。Googleは毎月のペースでこれらのパッケージを更新し、使えるようになったら新しいサービスたちへのアクセスが可能になるように努めるという。

これらのAPIはJavascriptからDartへの変換ツールで得られたものであるが、それ以外に異なった環境からのAPIにアクセスするために対応している総ての認定モデルを扱うためのgoogleapis_authパッケージが用意されている。このパッケージを使うと、スタンドアロンのアプリケーション、Google Compute Engine上、あるいはブラウザの中でクライアント・ライブラリたちが使いやすいものになる。

25.1節　　googleapisライブラリを使うときに必要なもの

Google Developers Console

これらのAPIを使うにはCloud Projectが必要である。またAPIによってAPIキーまたはクライアントIDのどちらかが必要になる。そしてどの種のアプリケーションが使われているかによってはサービス・アカウント(Service Account)が必要にある。これらは自分があるプロジェクト（アプリケーション）を作る際にGoogle Developers Consoleを介して取得できる。

クライアントID、サービス・アカウント、およびAPIキーがどのような場合に必要かを示すと：

- そのユーザに代わって（同意のもとで）該ユーザが所有するデータにアクセスするようなアプリケーション（例えばブラウザ・アプリケーションあるいはコンソール・アプリケーション）の類ではそのアプリケーションには**クライアントID**が必要になる。

- サーバ・アプリケーションではそのアプリケーション自身が所有しているデータに対し**サービス・アカウント**を使ってアクセスできる。例えばDatastoreまたはGoogle Cloud Storageにあるデータにアクセスする

サーバ・アプリケーション。

- パブリックなデータを取得するようなAPIでは**APIキー**（アクセス量の割り当てや課金目的の為に）のみが必要である。

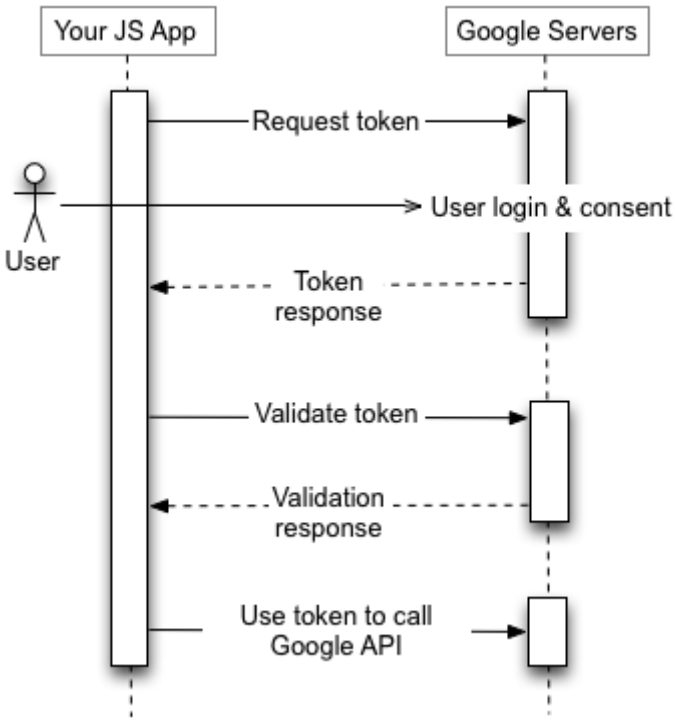
自分が開発するアプリケーションの為の認証と認可(authentication and authorization)システムを設定するには、まずプロジェクト(project)を用意し、認証クレデンシャル(authentication credentials)を作り、そののち使いたいAPIが使えるようにする。これらすべてをGoogle Developers Console上で行うことになる。

OAuth 2.0

Googleのウェブ・サービスにアクセスするアプリケーションでは場合によっては認証・認可にRFC-6749及びRFC-6750のOAuth 2.0プロトコルが必要になる。そのアプリケーションでOAuth 2.0が必要かどうかは使うAPIとどの種のデータにアクセスするかに依存する。例えば、ユーザの同意が必要な場合はOAuth 2.0が必要になる。以下のようなアプリケーションによってはOAuth 2.0が必要になる。

- パブリックなAPIを使うとき（クレデンシャルは不要）
- APIキーを使ってアクセスされるAPIを使うとき（具体的なユーザがいない）
- ユーザ・データにアクセスするAPIを使うとき（ユーザの同意が必要）
- アプリケーションのデータにアクセスするAPIを使うとき（サービスのアカウントが必要）

OAuth 2.0のプロトコルではウェブ・サーバのアプリケーションなのか、組み込みアプリケーションか、あるいはブラウザ上でのJavaScript（あるいはDart）で書かれたクライアント側アプリケーションなのかでそのフローが多少異なる。OAuth 2.0の認証認可のフローに関してはUsing OAuth 2.0 to Access Google APIsという資料を参考にすると良い。以下のフローはクライアント側（ブラウザ）アプリケーションの場合を示す:



クライアント・アプリケーションからアクセスするときのフロー

そのアプリケーションが（ポップアップ・ウィンドウ等を介して）リソース所有者であるGoogleのサーバのURL（https://accounts.google.com/o/oauth2/auth）にログインすると認可シーケンスが始まる。このURLにはそのアプリケーションが要求するGoogle APIアクセスのタイプがスコープとしてクエリ・パラメタのセットに含まれている。

結果としてアクセス・トークンが渡される。このトークンはGoogle の認可サーバ（https://www.googleapis.com/oauth2/v1/tokeninfo）の検証(validation)をとらないとGoogleのリソース・サーバへの要求は送信できない。このトークンにある有効期限を超えた場合は、そのアプリケーションはこのプロセスを繰り返す（更新する）ことになる。

確認が取れたらそのアプリケーションはそのアクセス・トークンを（HTTP要求のヘッダ、クエリあるいはボディとして）付けてGoogleのAPIに欲しいデータの要求を送信する(例えばhttps://www.googleapis.com/plus/v1/people/userId?access_token=1/fFBGRNJru1FQd44AzqT3Zg)。

このフローのHTTP上でのより具体的且つ詳細な解説はUsing OAuth 2.0 for Client-side Applicationsという資料で得られる。

googleapis_authパッケージ

Dartのコードからこの認証・認可の一連のフローを実行するにはgoogleapis_authというライブラリを使う。このライブラリを使えば、上記のフローの詳細を知ることなくコードを書くことができる。このパッケージ・ライブラリは以下の3つのライブラリで構成されている:

- googleapis_auth.auth（共通の関数、クラス、および例外）
- googleapis_auth.auth_browser（ブラウザ用）、および
- googleapis_auth.auth_io（サーバ用）

通常はgoogleapis_auth.authライブラリは使う必要はなく、googleapis_auth.auth_browser（ブラウザ用）またはgoogleapis_auth.auth_io（サーバ用）のどれかをインポートすればよい。主要な関数やメソッドの邦訳はこの章の終わりにあるので見て頂きたい。

Googleのウェブ・サービスを使うアプリケーションの開発者はこのライブラリを使ってGoogleのあるウェブ・サービスにアクセスするための専用のHTTPクライアントを取得する。一旦そのHTTPクライアントが取得出来たら、そのメソッド（Googleのウェブ・サービス毎に用意されている）を使ってそのアプリケーションのサーバと交信できる。このHTTPクライアントの必要が無くなった時点でこれを閉じる。

各ライブラリで得られるHTTPクライアント

ライブラリ	HTTPクライアント	メソッド	引数
共通	Client	clientViaApiKey	apiKey
	AuthClient	authenticatedClient	AccessCredentials
	AutoRefreshingAuthClient	autoRefreshingClient	ClientId, AccessCredentials
googleapis_auth.auth_browser（ブラウザ・アプリケーション）	AutoRefreshingAuthClient（Implicitベース）	createImplicitBrowserFlow.clientViaUserConsent	ClientId, scopes
googleapis_auth	AutoRefreshingAuthClient	clientViaMetadataServer	

.auth_io（サーバ・アプリケーション）	（METAdataサーバ経由）		
	AutoRefreshingAuthClient（service account経由）	clientViaServiceAccount	ServiceAccountCredentials, scopes
	AutoRefreshingAuthClient（ユーザ同意経由）	clientViaUserConsent	Scopes, ClientId, PromptUserForConsent
	AutoRefreshingAuthClient（ユーザ同意マニュアル経由）	clientViaUserConsentManual	Scopes, ClientId, PromptUserForConsent

どのクライアントを使うかはそのウェブ・サービスの内容による。一番簡単なのは認証を伴わないClientで、これはAPIキーだけでそのウェブ・サービスにアクセスする。それ以外は認証を受けたAuthClientである。良く使われるのはブラウザではインプリシット・フロー（ブラウザ用にクライアントの認証を伴わない簡素化されたフロー）をベースとしたユーザ同意（そのアプリケーションがユーザに代わってそのウェブ・サービスにアクセスすることをユーザが同意する手順）を伴うAutoRefreshingAuthClientであり、サーバ側ではユーザ同意を伴うAutoRefreshingAuthClientであろう。

ブラウザ側での認証の為のDartコードは次のようになろう：

```
import "package:googleapis_auth/auth_browser.dart";

...

var id = new ClientId("....apps.googleusercontent.com", null);
var scopes = [...];

// ブラウザ oauth2 フロー機能を初期化し、ユーザ同意を介して自動更新のクライアントを取得する。
createImplicitBrowserFlow(id, scopes).then((BrowserOAuth2Flow flow) {
 flow.clientViaUserConsent().then((AuthClient client) {
  // 認証が終了。自動更新のクライアントが client で使用可能になった。
  ...
  client.close();
  flow.close();
 });
});
```

createImplicitBrowserFlowという関数はGoogleのgapi ライブラリをロードし、それを初期化する。初期化が終わったらBrowserOAuth2Flowオブジェクトで完了する。このフロー・オブジェクトは AccessCredentialsあるいはauthenticated HTTPクライアントを取得するのに使える。authenticated HTTPクライアントを取得するにはBrowserOAuth2FlowのメソッドであるclientViaUserConsent()を使用する。

clientViaUserConsent()はAccessCredentialsを取得し、認可を受けたHTTPクライアントを返す。アクセス・クレデンシャルを取得したあとは、この関数はHTTPクライアントを返す。返されたクライアント上でなされたHTTP要求には取得したAccessCredentialsつきのAuthorizationヘッダが付加される。AccessCredentialsの有効期限が切れているときは、ユーザの同意なしに新らたなクレデンシャルを取得しようとする。このメソッドが返したFutureが完了したときにコールバックのなかでこのクライアントを使って必要なデータの取得ができる。

obtainAccessCredentialsViaUserConsentと clientViaUserConsentの2つのメソッドはユーザ認証のダイヤログの為のポップアップ・ウィンドウを開こうとする。ブラウザがこのポップアップ・ウィンドウが開かないようにしようとするには、これらのメソッドたちはイベント・ハンドラ内でのみ呼ばれねばならない。なぜなら、ほとんどのブラウザはユーザとの関わり合いに応じて作られたポップアップ・ウィンドウを阻止できないからである。

終了時にはクライアントだけでなくフローもクローズしなければならない。

一方サーバ側では次のようなコードとなろう:

```
import "package:googleapis_auth/auth_io.dart";

 ...

 var id = new ClientId("....apps.googleusercontent.com", "...");
 var scopes = [...];

 clientViaUserConsent(id, scopes, prompt).then((AuthClient client) {
  // 認証が終了。自動更新のクライアントが client で使用可能になった。
  // ...
  client.close();
 });

 void prompt(String url) {
  print("つぎの URL に行ってアクセスに同意してください:");
  print("  => $url");
  print("");
 }
```

clientViaUserConsent関数はAccessCredentialsを取得し、認可を受けたHTTPクライアントを返す。アクセス・クレデンシャルを取得したあとは、この関数はHTTPクライアントを返す。返されたクライアント上でなされたHTTP要求には取得したAccessCredentialsつきのAuthorizationヘッダが付加される。AccessCredentialsの有効期限が切れているときは、ユーザの同意なしに新らたなクレデンシャルを取得しようとする。

promptはこのクライアントがユーザに代わってそのウェブ・サービスにアクセスすることに同意するかどうかの問い合わせに使用する。promptの引数はurlである。このURLはこのサーバに代わってブラウザで同意画面経由でユーザの同意を得るためのものである。この関数はtypedefとして次のようにgoogleapis_auth.auth_ioのなかで定義されている。

```
typedef void PromptUserForConsent (String uri)
```

25.2節　　サンプルを試してみる

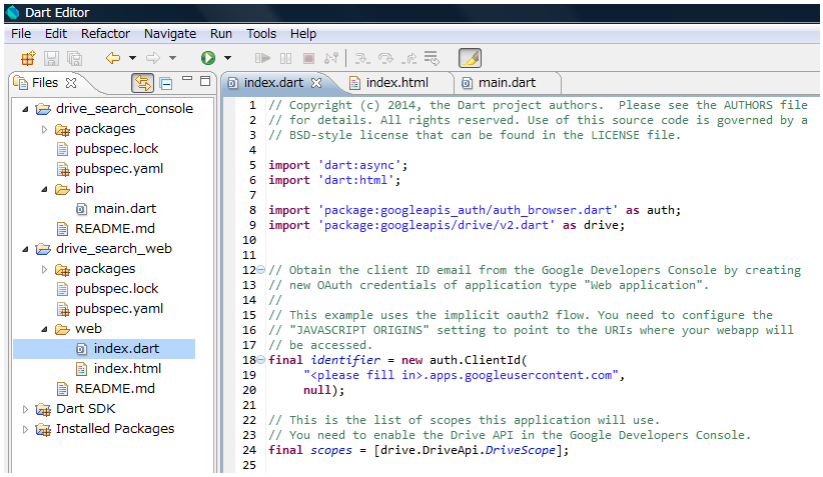
GitHubのgoogleapis_examplesというリポジトリを先ず見て頂きたい。これらのサンプルはブラウザ内で走るウェブ・アプリケーションのDartコードと、スタンドアロンのプログラムとして走るサーバ・アプリケーションのDartコードが用意されており、どのようにGoogle Driveと Google Cloud Storageにアクセスするかを示している。本節ではこのサンプルをベースにこれらのライブラリの使い方を説明してゆく。この説明はGoogle APIs Client Libraries with Dartという資料に沿ったものである。

まずこのリポジトリのZIPファイルをダウンロードし、解凍したら以下の2つを自分のIDEで展開する:

- Example: drive_search_web　: Google Drive APIの使用法のデモで、如何にクライアント・サイドだけのウェブ・アプリケーションからDriveの中にあるファイルをサーチするかを示している。

- drive_search_console ：Google Drive APIの使用法のデモで、如何にコンソール・アプリケーションからDriveの中にあるファイルをサーチするかを示している。

Google Drive APIドキュメントはここにある。Driveアプリケーションの使用法の詳細はGoogle デベロッパー アカデミーのDriveに関する資料（日本語）を読むと良い。



Driveアクセスのサンプルの展開

アプリケーションの流れ

アプリケーション（ここではindex.htmlやindex.dart）は基本的に次のような手順を踏む。

1. このアプリケーションは必要なクライアント・ライブラリ（googleapis_auth/auth_browser.dartまたはgoogleapis_auth/auth_io.dart、及びgoogleapis/drive/v2.dart等）を取り込む。

2. このアプリケーションはクライアントIDを使ってGoogleのサービスの認証を受ける。

3. このアプリケーションがユーザのパーソナルな情報にアクセスする必要があるとき（このサンプルのように）は、Googleの認証サーバとのセッションを開き、この認証サーバはダイヤログ・ボックス（ポップアップ・ウィンドウ）でこのユーザに対しパーソナルな情報をアクセスすることへの同意を得る。

4. このアプリケーションはそのAPIによって返されるデータを指定するHTTPS要求オブジェクト（クレデンシャルがヘッダに付される）を生成する。

5. このアプリケーションはその要求オブジェクトを該APIに送信し、そのAPIが返してきたデータを処理する。

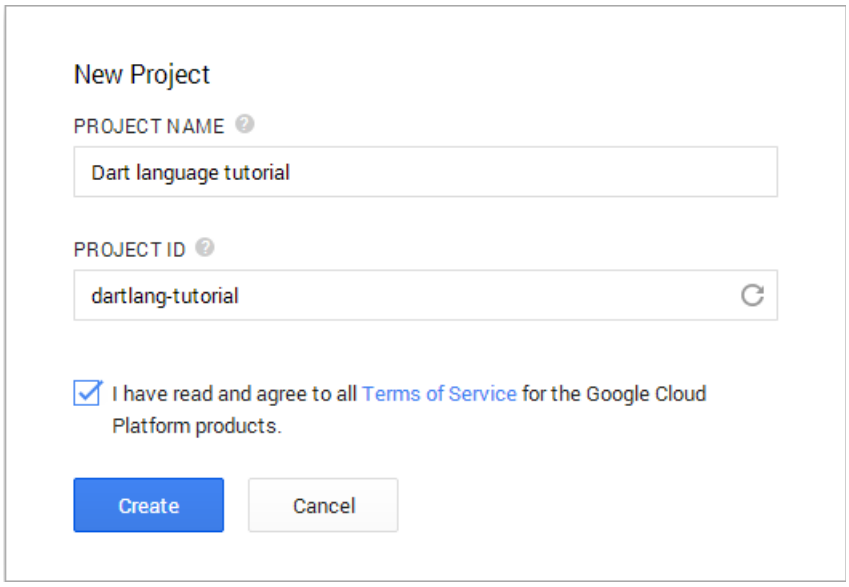
アプリケーションの登録とクライアントIDの取得

これらのサンプルを使うには、このGoogleのクライアント・アプリケーションを使うプロジェクト（アプリケーション）を登録し、このプロジェクトの為のAPIが使えるようにし、またこのサンプルに必要なクライアントIDを取得しなければならない。これらは Google Developers Consoleのなかで行う。Google Developers Consoleを使うには読者は

Googleのアカウントが必要である。未だこのアカウントを持っていない人はここから登録する。

プロジェクトの生成

- Google Developers Consoleを開き、create Projectのボタンをクリックする
- プロジェクト名とそのIDを入力し、同意にチェックしてCreateボタンをクリックする



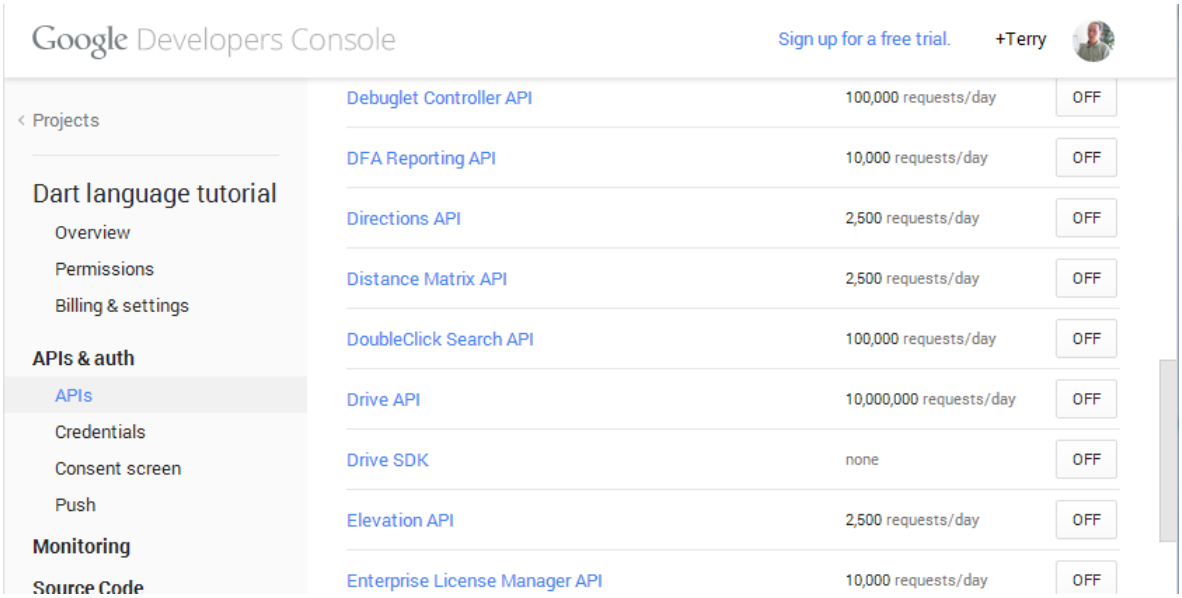
プロジェクトの登録

そうするとプロジェクトIDとプロジェクト番号が次のように表示される:

```
Project ID: dartlang-tutorial Project Number: 11桁の数字
```

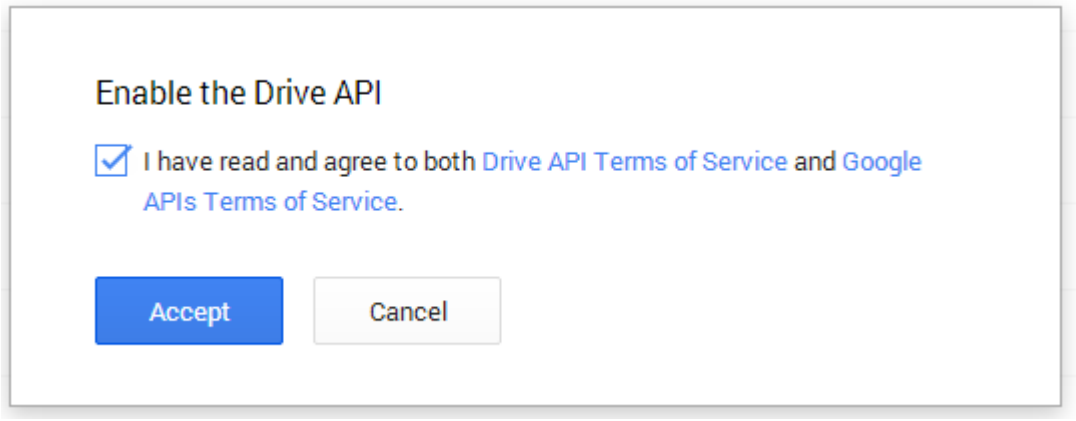
Drive APIが使えるようにする

- このプロジェクト・ページの左側の APIs & authをクリックし、次に APIsをクリックする。
- 下図のように一覧で表示されるので、Drive APIの認証のオフボタンをオンにする。



Drive APIの選択

- 次のような表示が出たら同意にチェックしてAcceptをクリックする。

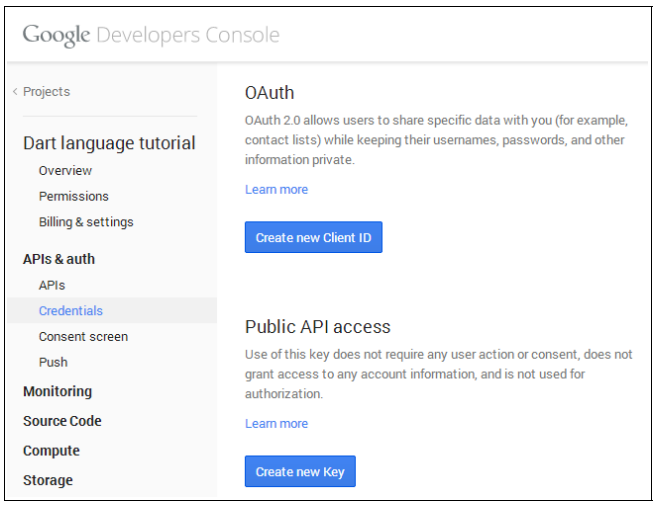


API使用の同意

- これにはときには数分間の時間がかかることがある。

クライアントIDの取得

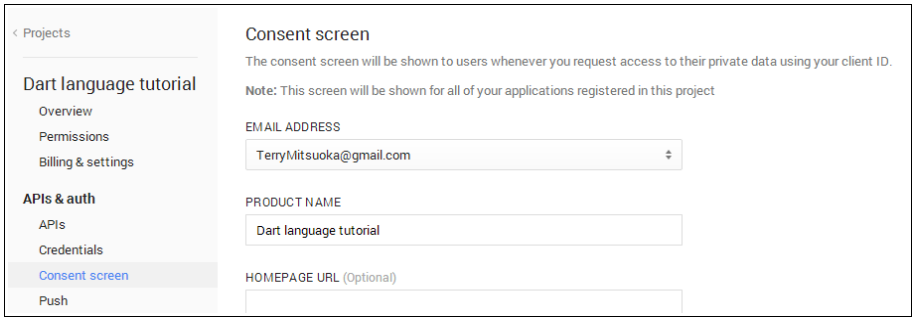
- このプロジェクト・ページの左側の APIs & authをクリックし、次に Credentials をクリックする。



クライアントID取得の選択

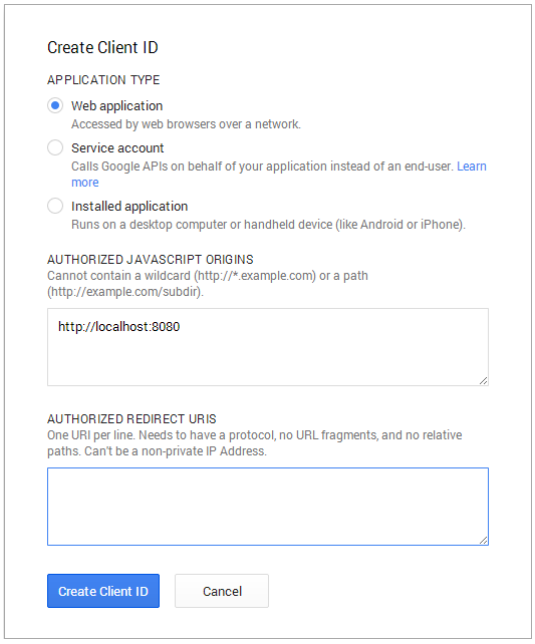
- Create new Client IDをクリックする。

- もしconsent記入が必要と表示されたら、Consent screenで必要な個所（EMAIL ADDRESSとPRODCT NAME）を埋めてSaveをクリックする。



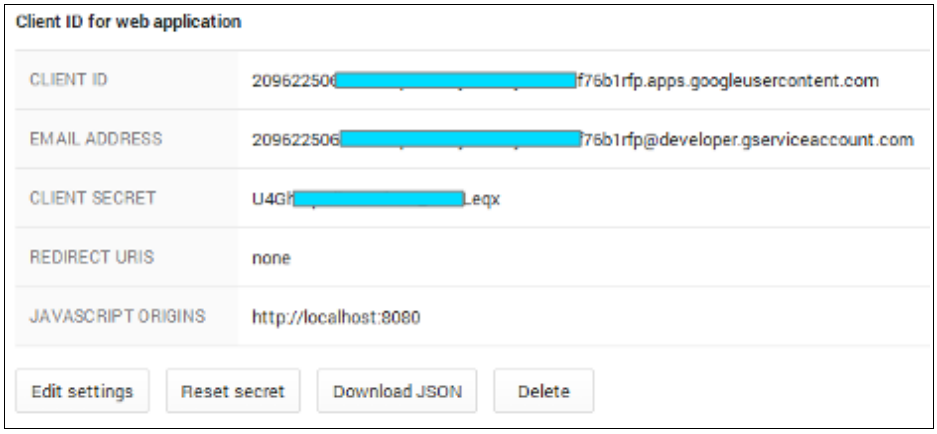
Consent screenを埋める

- ポップアップ・ウィンドウの中で Web applicationを選択する。次にこのフォームの中を埋める。AUTHORIZED JAVASCRIPT ORIGINSは http://localhost:8080を使う。なぜならこのサンプル自分のIDEがDartiumにこのホストをアクセスさせるからである。ほかのアプリケーションで現在8080というポート番号が使われていないことを確認する。将来別のオリジンからこのサンプルを走らせるときはここを変更しなければならない。またAUTHORIZED REDIRECT URISは何も記入してはいけない。



クライアントIDの生成

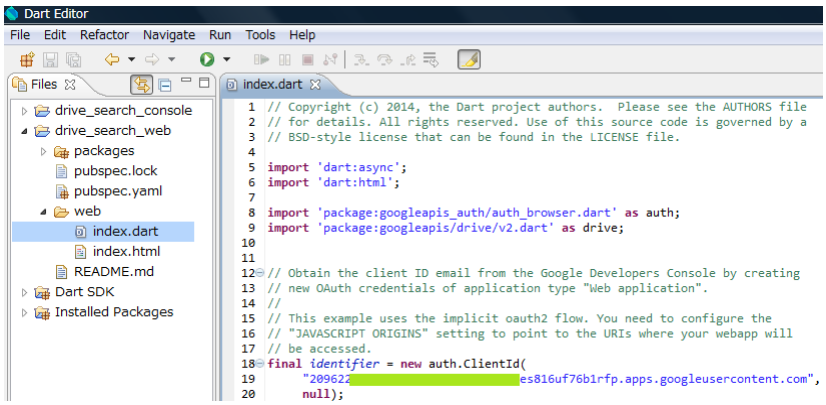
- Create Client IDボタンをクリックすると、次のように結果が表示される:



クライアントIDの取得

ウェブ・アプリケーションの実行

- 自分のIDE上でdrive_search_web/web/index.dartのなかのクライアントIDをセットする。



クライアント・アプリケーションの編集

つまり以下のように、赤の個所を取得したCLIENT IDの値で置き換える:

```
final identifier = new auth.ClientId(
   "ここに Google Developers Console で取得したクライアント ID をセットする", null);
```

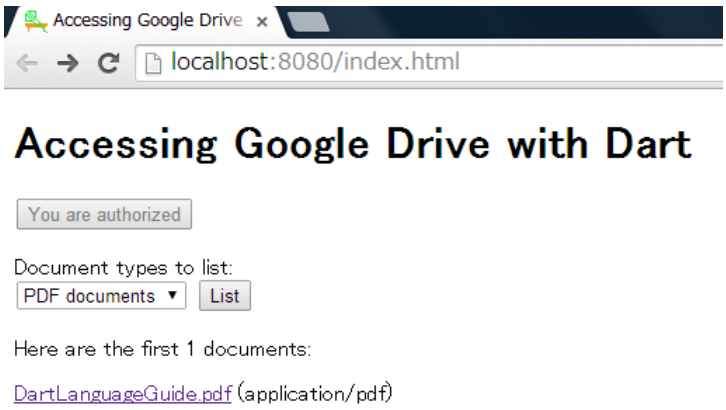
- 自分のIDEからDartiumを起動するとともにこのアプリケーションを読み込ませる。即ち自分のIDE上でdrive_search_web/web/index.htmlを選択し、右クリックでRun in Dartiumを選択する。

- 最初に Authorizeボタンをクリックするとログイン画面が出てくるので、自分のGoogleのアカウントの電子メール・アドレスとパスワードを入力してログインする。

- 次に下図に示すように、このアプリケーションがユーザに対し許可を求めるので、承認ボタンをクリックする。

Consent画面

- ボタンがYou are autholizedと表示が変わりハイライトが消えるので、探したいドキュメントのタイプを指定し、Listボタンでサーチを実行させる。下図はその結果である。PDFドキュメントとしては一つだけ存在する。

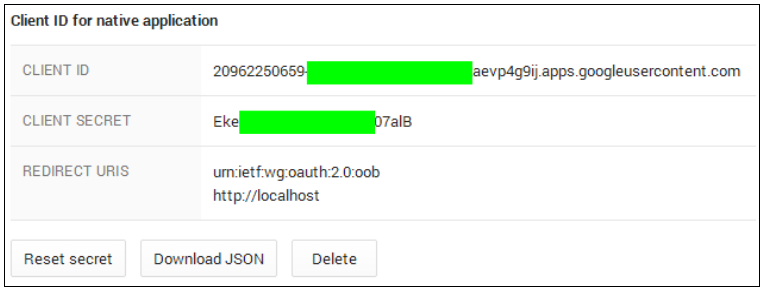


このプログラムの実行例

このプログラムは現在ログオンしているユーザのGoogle Drive からの検索条件を満たすファイルのリストを表示している。そうしてほしくないユーザはアクセスを拒否できる。Security - Account Settings 参照のこと。

サーバ・アプリケーションの実行

- Google Developers Console 上のこのプロジェクト・ページの左側の APIs & authをクリックし、次に Credentials をクリックする。次にAPPLICATION TYPEにはInstalled applicationを選択、INSTALLED APPLICATION TYPEはothersを選択し、Create client IDボタンをクリックする。そうすると以下のように表示される:



サーバ用クレデンシャルの取得

- 自分のIDE上でdrive_search_console/bin/main.dartのなかのクライアントIDをセットする。
つまり以下の行の最初の個所にCLIENT IDで、次の個所にCLIENT SECRETで置き換える。

```
final identifier = new auth.ClientId(
  "<please fill in>.apps.googleusercontent.com",
  "<please fill in>");
```

- このプログラムを実行させる。
- コマンド・プロンプトからVMのパス設定を行う。

```
C:\>path c:\dart_editor\dart-sdk\bin
```

- Dartの起動

```
c:\dart_applications\googleapis_examples-master\drive_search_console>dart bin\main.dart
pdf
```

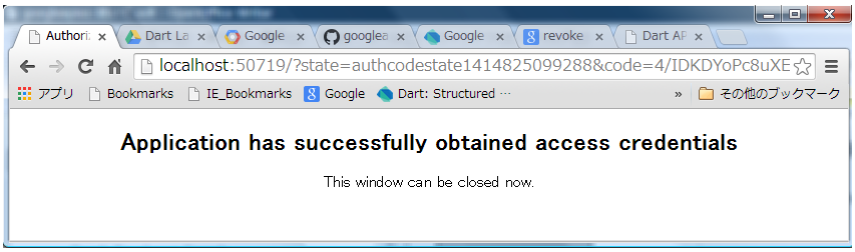
- このプログラムからのプロンプトが表示される:

```
Please go to the following URL and grant access:
 => https://accounts.google.com/o/oauth2/auth?response_type=code&client_id=20962 途中省略
116lkaevp4g9ij.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost
%3A50654&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth
%2Fdrive&state=authcodestate1414823793709
```

- ブラウザから同意する。 上記のプロンプトにあるURL（https://accounts.....最後まで）をChrome等のブラウザのアドレスにコピー/ペーストしそのURLにアクセスさせる。そうすると下図のような同意のプロンプトが出るので承認するのボタンをクリックする。

ブラウザからの同意

承認したらブラウザにはこのアプリケーションがアクセス・クレデンシャルを取得したと以下のように表示される:

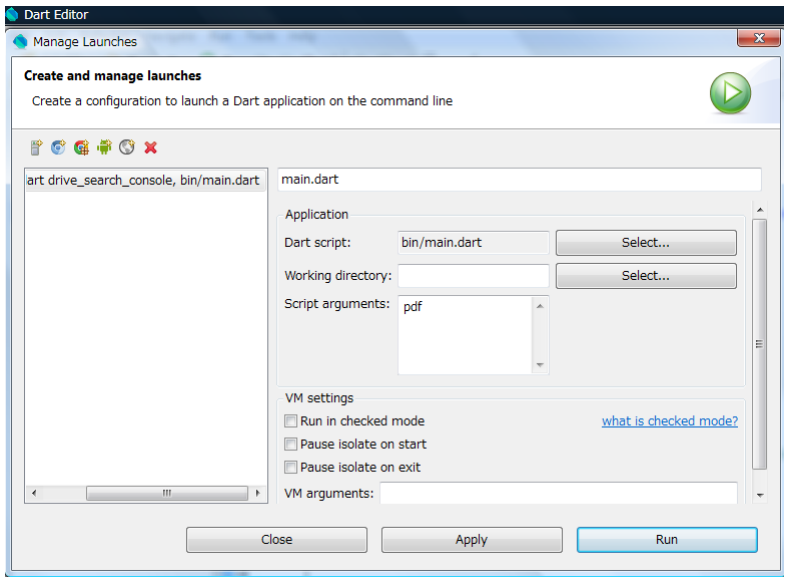


ブラウザ上での表示

そうするとコマンド・プロンプトにはサーチの結果が表示される。

```
Here are the first 1 documents:

 - DartLanguageGuide.pdf (https://docs.google.com/file/d/0B-
txJOS6P31eZjQyeDVydTFlY1U/edit?usp=drivesdk)
```

無論自分のIDEから立ち上げることも可能である。その場合はローンチ・マネージャを使用する。Script argumentsに必要なファイルの拡張子（pdf、docs、slides、spreadsheets、allなど）をひとつ書き込む。



ローンチ・マネージャからのサーバ・アプリケーションの立ち上げ

サンプルのコードに就いて

以上の2つのサンプルを実際に走らせてみることで、DartでOAuth2.0の認証・認可の手順をどう書けば良いかが理解されよう。その後のコードは使うGoogleのアプリケーションによって異なるので、APIドキュメントを見ながら作業することになる。

ブラウザの認証・認可・同意に関する注意点としては、同意の為のポップアップ・ウィンドウがある。以下のコードは多分定型的に使用可能である。authorizedClientという関数はAutoRefreshingAuthClientのインスタンスをFutureベースで返す。その際ブラウザに対しユーザ同意のポップアップを開こうとするが、ほとんどのブラウザはこれを受け付けない。従ってユーザのonClickを引き金にしたハンドラ内でこれを行っている。

```
// Obtain an authenticated HTTP client which can be used for accessing Google
// APIs.
Future authorizedClient(ButtonElement loginButton, auth.ClientId id, scopes) {
 //  oauth2ブラウザ・フローを初期化し、認証呼び出しができるようになったら直ちに完了する。
 return auth.createImplicitBrowserFlow(id, scopes)
   .then((auth.BrowserOAuth2Flow flow) {
  // ユーザの同意なしでクレデンシャルを取得しようとまず試みる。
  // ユーザが既にこのアプリケーションに対し同意を与えてしまっている場合はこれは成功する。
  return flow.clientViaUserConsent(forceUserConsent: false).catchError((_) {
   // エラー(UserConsentException)の場合はユーザに同意を求める。
   // 同意を求める際は、ポップアップ・ウィンドウが作られ、そこでユーザはGoogleで認証を受け
   // このアプリケーションが彼に代わってデータにアクセスすることに同意しなければならない。
   //
   // 殆どのブラウザはデフォルトではポップアップ・ウィンドウをブロックしているので、我々はイベント・ハンドラ内
   // でのみこれができる(もしあるユーザ・アクションがポップアップの引き金になっているときは、
   // 通常それはブロックされない)。我々はそのためにloginButtonを使っている。
   loginButton.text = 'Authorize';
   return loginButton.onClick.first.then((_) {
    return flow.clientViaUserConsent(forceUserConsent: true);
   });
  }, test: (error) => error is auth.UserConsentException);
```

```
 });
}
```

Google DriveのDartのAPI

例えばGoogle DriveのDartのAPIを見ると非常に多くのクラスが存在する。しかしまだこれは不十分な状態なので、これとGoogle Drive Web APIsのAPIドキュメントを並べて作業すると良い。

クライアント用にしろサーバ用にしろこれらのサンプルでは次のようなメソッドがファイル・サーチの為に使われている:

```
api.files.list(q: query, pageToken: token, maxResults: max)
```

queryではサーチの為の非常に多様なクエリが作れる。Search for Filesという資料を参考にすると良い。このサンプルでは単にMIMEタイプでの検索なので、mimeType = 'application/pdf'といったクエリになっている。pageTokenは複数のページで結果を返す場合に使われる。

このメソッドはFutureを返し、[FileList]で完了する。FileListの属性としては以下のものがある:

- String etag
- List<File> items （これが実際のファイル・リスト）
- String kind （これは常にdrive#fileList）
- String nextLink （ファイルたちの次のページへのリンク）
- String nextPageToken （ファイルたちの次のページの為のページ・トークン）
- String selfLink （このリストに戻るためのリンク）

サンプルではitemsとnextPageTokenが使われている。

サンプルではこの部分は次のようなFutureベースの関数となっている:

```
Future<List<drive.File>> searchTextDocuments(drive.DriveApi api, int max, String query) {
 var docs = [];
 Future next(String token) {
  // この api 呼び出しでは結果のサブセットのみが返される。
  // ページングを使えば結果の全部を知ることができる。
  return api.files.list(q: query, pageToken: token, maxResults: max)
    .then((results) {
   docs.addAll(results.items);
   // もっとドキュメントを得たければこれを繰り返す。
   if (docs.length < max && results.nextPageToken != null) {
    return next(results.nextPageToken);
   }
   return docs;
  });
 }
 return next(null); // next を起動
}
```

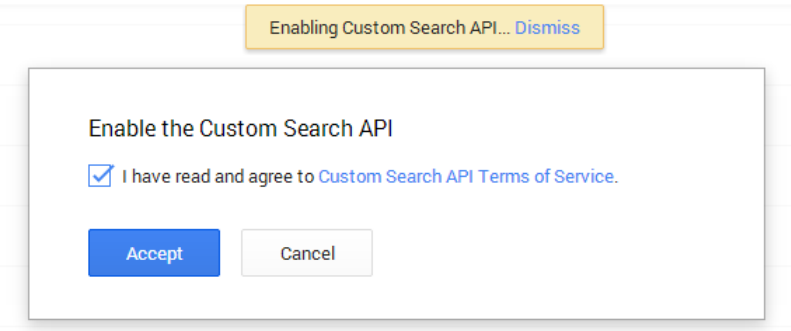
APIキーだけで済むサンプル・アプリケーション

おなじGitHubのgoogleapis_examplesポジトリにはgoogleapis_examples-master\customsearch_browser_apikeyというGoogle Custom Searchにブラウザからアクセスするサンプルが存在する。このサービスはユーザの同意が不要なオープンなものなので、APIキーさえあればアクセスできる。

このアプリケーションはブラウザからGoogle Custom Searchを使って、pub.dartlang.orgにあるライブラリを検索する。

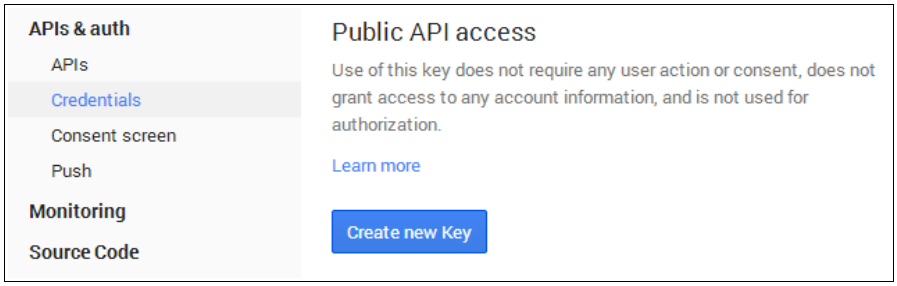
APIキーの取得

最初にGoogle Developer Consoleに行き、APIs & AuthのAPIsを選択する。そのなかのCustom Search APIを「オン」にする。同意するをチェックしてAcceptボタンをクリックする。



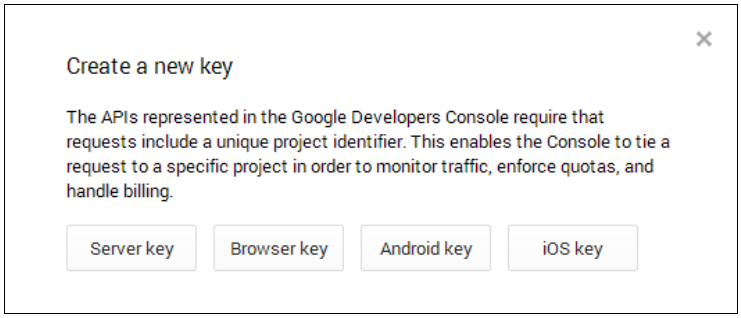
Custom Search APIをオンにする

次にAPIs & AuthのCledentialsを選択する。その右にCreate new Keyというボタンがあるので、それをクリックする。



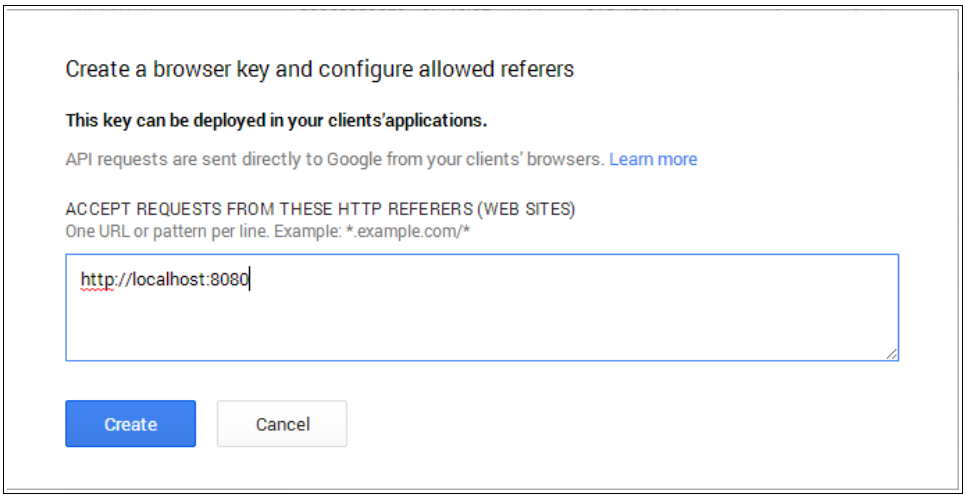
新しいAPIキーの生成

そうすると下図のようにどの種のクライアントなのかを聞いてくるので、ここではBrowser keyのボタンをクリックする。



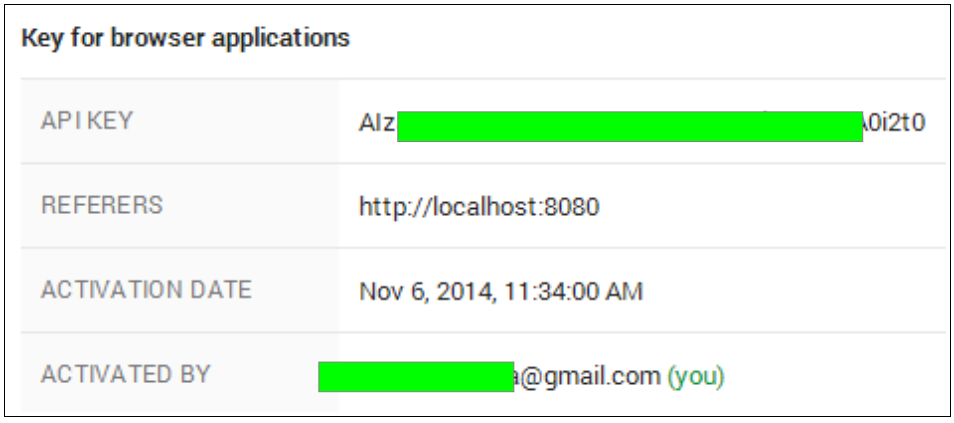
クライアントのタイプを指定

次にHTTPリフェラ(ウェブ・サイト)を聞いてくるので、以前と同様にhttp://localhost:8080を入力し、Createボタンをクリックする。



キーの生成

そうすると以下のようなAPIキーが得られる:



得られたAPIキー

このAPIキーをgoogleapis_examples-master\customsearch_browser_apikey\web\index.dartの次の個所にはめ込む。

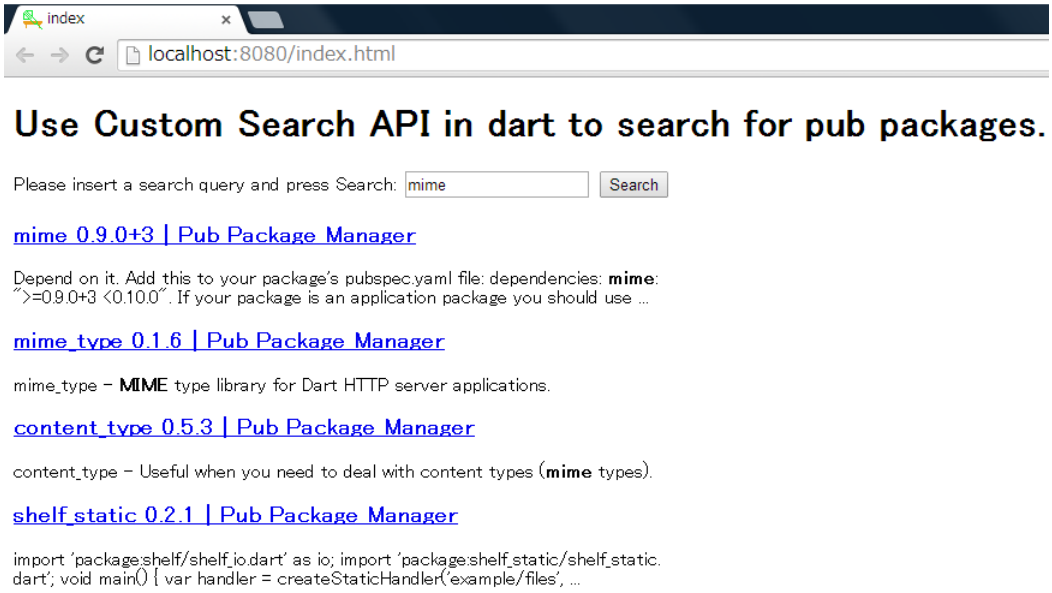
index.dart

```
// Copyright (c) 2014, the Dart project authors.  Please see the AUTHORS file
// for details. All rights reserved. Use of this source code is governed by a
// BSD-style license that can be found in the LICENSE file.

import 'dart:async';
import 'dart:html';

import 'package:googleapis_auth/auth_browser.dart' as auth;
import 'package:googleapis/customsearch/v1.dart' as search;

// これは pub のカスタム・サーチの ID で、以下のアドレスにあり。
// https://github.com/dart-lang/pub-dartlang/blob/master/app/handlers/search.py
final customSearchId = '009011925481577436976:h931xn2j7o0';

// これは Google Developers Console から得られた API キーである。
final apiKey = 'AizaS 途中省略 A0i2t0';
```

これでこのアプリケーションは動作可能となる。

このアプリケーションを走らせる

自分のIDE上でgoogleapis_examples-master\customsearch_browser_apikey\web\index.htmlを選択し左クリックでRun in Dartiumを選択すれば、このアプリケーションは起動する。下図はその実行例である。mimeを入力するとmimeというキーワードを含むpubパッケージの一覧が表示される。筆者がアップロードしたmime_typeというパッケージは2番目に示されている。



カスタム・サーチ例

このサンプルのコードについて

このサンプルのポイントを以下に示す。Google Custom Searchはサーチの為のエンジンを使用するので、これを先ず生成する必要がある。その手順についてはマニュアルに記載されている。このサンプルにはその結果得られたpubパッケージのサーチの為のCustom Search IDが必要である。

```
// Use the Custom Search API to search for pub packages.
Future<List<Package>> searchPackages(search.CustomsearchApi api, String query) {
 return api.cse.list(query, cx: customSearchId).then((search.Search search) {
  var packages = [];
  if (search.items != null) {
   for (var result in search.items) {
    packages.add(
      new Package(result.htmlTitle, result.link, result.htmlSnippet));
   }
  }
  return packages;
 });
}
```

googleapis.customsearch.v1のAPIドキュメントを見ると、search.CustomsearchApi.CseResourceApiはサーチ・エンジンを表現しており、これにはFuture<Search>を返すlistというメソッドのみがある。この例ではCustomsearchApi即ちAPIキーとクエリのみが引数となっている。

main()メソッドでは最初にclientViaApiKey(client)関数で得られるClientオブジェクトと、そのオブジェクトを引数にしたCustomsearchApi(client)で得られるAPIオブジェクトを用意する必要がある:

```
main() {
 InputElement searchText = querySelector('#search_text');
 ButtonElement searchButton = querySelector('#search_button');
 DivElement results = querySelector('#results');

 var client = auth.clientViaApiKey(apiKey);
 var api = new search.CustomsearchApi(client);

 searchText.onInput.listen((_) {
  searchButton.disabled = searchText.value == '';
 });

 searchButton.onClick.listen((_) {
  searchPackages(api, searchText.value).then((List<Package> packages) {
   results.children.clear();

   if (packages.isEmpty){
    display('<h5>No results found.</h5>', results);
   } else {
    for (var package in packages) {
     display('<h3><a href="${package.url}">${package.title}</a></h3>',
          results);
     display(package.snippet, results);
    }
   }
  });
 });
}
```

25.3節　googleapis_authパッケージの主要部分の和訳

オリジナル・ドキュメント:
http://www.dartdocs.org/documentation/googleapis_auth/0.1.1/index.html#googleapis_auth

googleapis_auth.authライブラリ

googleapis_auth.authライブラリ

関数
AuthClient **authenticatedClient**(Client baseClient, AccessCredentials credentials)

クレデンシャルを使って要求を自動的に認証するHTTPクライアントを取得する。 返された RequestHandlerは与えられたクレデンシャルを自動更新しないことに注意。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
AutoRefreshingAuthClient **autoRefreshingClient**(ClientId clientId, AccessCredentials credentials, Client baseClient)
クレデンシャルの有効期限が切れる前にクレデンシャルを自動的に更新するHTTPクライアントを取得する。認証されたHTTP要求を作り、またクレデンシャルの更新ためにはbaseClientを使う。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
Future<AccessCredentials> **refreshCredentials**(ClientId clientId, AccessCredentials credentials, Client client)
client を使ってcredentialsに基づいてアクセス・クレデンシャルの更新を試みる。
クラス
AccessCredentials
OAuth2クレデンシャルを表現
AccessToken
OAuth2アクセス・トークンを表現
AuthClient
認証を得たHTTPクライアント
AutoRefreshingAuthClient
自動更新の認証を受けたHTTPクライアント
ClientId
該クライアント・アプリケーションのクレデンシャルを表現
ServiceAccountCredentials
サービス・アカウントのクレデンシャルを表現

googleapis_auth.auth_browserライブラリ

googleapis_auth.auth_browserライブラリ

関数
AuthClient **authenticatedClient**(Client baseClient, AccessCredentials credentials)
クレデンシャルを使って要求を自動的に認証するHTTPクライアントを取得する。 返された RequestHandlerは与えられたクレデンシャルを自動更新しないことに注意。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。

AutoRefreshingAuthClient **autoRefreshingClient**(ClientId clientId, AccessCredentials credentials, Client baseClient)
クレデンシャルの有効期限が切れる前にクレデンシャルを自動的に更新するHTTPクライアントを取得する。認証されたHTTP要求を作り、またクレデンシャルの更新ためにはbaseClientを使う。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
Client **clientViaApiKey**(String apiKey, {Client baseClient})
HTTP要求をするのに与えられた apiKeyを使用するHTTPクライアントを取得する。 返されたクライアントは Google ServicesへのHTTP要求をする為だけに使わねばならないことに注意。この apiKeyは第3者に公開してはいけない。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
Future<BrowserOAuth2Flow> **createImplicitBrowserFlow**(ClientId clientId, List<String> scopes, {Client baseClient})
BrowserOAuth2Flowのオブジェクトを生成し、それでfutureを完了する。 この関数はOAuth 2.0フローにに基づいた暗示的ブラウザを実行する。 これはGoogle'のgapi ライブラリをロードし、それを初期化する。初期化が終わったら BrowserOAuth2Flowオブジェクトで完了する。このフロー・オブジェクトは AccessCredentialsあるいは authenticated HTTPクライアントを取得するのに使える。 gapiのロードまたは初期化でエラーが生じたら、このfutureはエラーで完了する。 baseClientが与えられていないときは、自動的にひとつ生成される。これは認証されたHTTP要求を行うのに使われる。BrowserOAuth2Flowを読むこと。 ClientIdはGoogle Cloud Consoleから取得できる。 返された BrowserOAuth2Flowオブジェクトを閉じるのはユーザの責任である。返された BrowserOAuth2Flowを閉じてもbaseClient（与えられているとき）は閉じない。
Future<AccessCredentials> **refreshCredentials**(ClientId clientId, AccessCredentials credentials, Client client)
clientを使って credentialsに基づいて更新された AccessCredentialsの取得を試みる。

クラス	
AccessCredentials	OAuth 2.0クレデンシャルを表現。
AccessToken	OAuth 2.0アクセス・トークンを表現。
AuthClient	認証されたHTTPクライアント。
AutoRefreshingAuthClient	自動更新の認証されたHTTPクライアント。
BrowserOAuth2Flow	OAuth 2.0のアクセス・クレデンシャルを取得するのに使われる。
ClientId	クライアント・アプリケーションのクレデンシャルを表現。
ServiceAccountCredentials	サービス・アカウントのクレデンシャルを表現。

例外	
AccessDeniedException	認可された要求の試みが失敗したときスローされる。
RefreshFailedException	コーケンの更新の試みが失敗したときスローされる。
UserConsentException	ユーザが同意をしなかったときにスローされる。

googleapis_auth.auth_browser/**BrowserOAuth2Flow** class

Extends: Object
OAuth 2.0のアクセス・クレデンシャルを取得するのに使う。
警告:
obtainAccessCredentialsViaUserConsentと clientViaUserConsentの2つのメソッドはユーザ認証のダイヤログの為のポップアップ・ウィンドウを開こうとする。ブラウザがこのポップアップ・ウィンドウが開かないようにしようとするには、これらのメソッドたちはイベント・ハンドラ内でのみ呼ばれねばならない。なぜなら、ほとんどのブラウザはユーザとの関わり合いに応じて作られたポップアップ・ウィンドウを阻止できないからである。

メソッド

Future<AutoRefreshingAuthClient> **clientViaUserConsent**({bool forceUserConsent: true})

AccessCredentialsを取得し、認可を受けたHTTPクライアントを返す。

アクセス・クレデンシャルを取得したあとは、この関数はHTTPクライアントを返す。返されたクライアント上でなされたHTTP要求には取得したAccessCredentialsつきのAuthorizationヘッダが付加される。

AccessCredentialsの有効期限が切れているときは、ユーザの同意なしに新らたなクレデンシャルを取得しようとする。

如何にクレデンシャルが取得されるかに関しては obtainAccessCredentialsViaUserConsentを見ること。
obtainAccessCredentialsViaUserConsent からのエラーは、この関数が返されるFutureと、返されるHTTPクライアント(クレデンシャル更新の場合)を通過する。

返されたHTTPクライアントはそのFuture<Response> またはそのResponse.read()ストリームを介してエラーを下位レベルに渡す。

返されたHTTPクライアントを閉じるのはユーザの責任になる。

void **close**()

この BrowserOAuth2Flowオブジェクトと、これが使っているHTTPクライアントを閉じる。

clientViaUserConsent を介して取得したクライアントは動作を継続する。このフロー・オブジェクトと取得した総てのクライアントたちが閉じた後は、その下に存在するHTTPクライアントも同じく閉じられる。

このcloseメソッドを呼んだあとでは、clientViaUserConsent及びobtainAccessCredentialsViaUserConsentを呼ぶとエラーになる。

Future<AccessCredentials> **obtainAccessCredentialsViaUserConsent**({bool forceUserConsent: true})

OAuth 2.0のアクセス・クレデンシャルを取得する。

forceUserConsent がtrueのときは、新たなポップアップ・ウィンドウが作られる。そのユーザには彼に代わってアクセスしようとするアプリケーションのスコープ(有効範囲)のリストが提示される。該ユーザはそのアプリケーションへのアクセス要求を承認するか拒否するかをする。

forceUserConsent がfalseのときは、ユーザの同意なしにアクセス・クレデンシャルを取得しようとする。 返されたfutureは、もしそのユーザがそのデータへのアクセスをそのアプリケーションに与えた場合は、AccessCredentialsで完了する。そうでない場合は、 UserConsentExceptionで完了する。 別の要因でエラーが発生したときは、返されるfutureはException.で完了する。

googleapis_auth.auth_ioライブラリ

googleapis_auth.auth_ioライブラリ

関数
AuthClient **authenticatedClient**(Client baseClient, AccessCredentials credentials)
クレデンシャルを使って要求を自動的に認証するHTTPクライアントを取得する。 返された RequestHandlerは与えられたクレデンシャルを自動更新しないことに注意。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
AutoRefreshingAuthClient **autoRefreshingClient**(ClientId clientId, AccessCredentials credentials, Client baseClient)
クレデンシャルの有効期限が切れる前にクレデンシャルを自動的に更新するHTTPクライアントを取得する。認証されたHTTP要求を作り、またクレデンシャルの更新ためにはbaseClientを使う。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
Client **clientViaApiKey**(String apiKey, {Client baseClient})
HTTP要求をするのに与えられた apiKeyを使用するHTTPクライアントを取得する。 返されたクライアントは Google ServicesへのHTTP要求をする為だけに使わねばならないことに注意。このapiKeyは第3者に公開してはいけない。 返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。
Future<AutoRefreshingAuthClient> **clientViaMetadataServer**({Client baseClient})
OAuth2のクレデンシャルを取得し、認証を受けたHTTPクライアントを返す。 アクセス・クレデンシャル取得に使われる引数たちの詳細は obtainAccessCredentialsViaMetadataServerを見ること。 一旦アクセス・クレデンシャルが取得されたら、この関数は自動更新のHTTPクライアントで完了する。一旦アクセス・クレデンシャルが期限切れになったら、新たなアクセス・クレデンシャルを取得する。 baseClientは与えられていないときは自動的にこれが生成される。これは認証を受けたHTTP要求をする際、あるいはアクセス・クレデンシャルを取得する際に使われる。

返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。

Future<AutoRefreshingAuthClient> **clientViaServiceAccount**(ServiceAccountCredentials clientCredentials, List<String> scopes, {Client baseClient})

OAuth2のクレデンシャルを取得し、認証を受けたHTTPクライアントを返す。

アクセス・クレデンシャル取得に使われる引数たちの詳細は obtainAccessCredentialsViaServiceAccount を見ること。

一旦アクセス・クレデンシャルが取得されたら、この関数は自動更新のHTTPクライアントで完了する。一旦アクセス・クレデンシャルが期限切れになったら、新たなアクセス・クレデンシャルを取得する。

baseClientは与えられていないときは自動的にこれが生成される。これは認証を受けたHTTP要求をする際、あるいはアクセス・クレデンシャルを取得する際に使われる。

返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。

Future<AutoRefreshingAuthClient> **clientViaUserConsent**({bool forceUserConsent: true})

AccessCredentialsを取得し、認証を受けたHTTPクライアントを返す。

アクセス・クレデンシャルを取得したあとは、この関数はHTTPクライアントを返す。返されたクライアント上でなされたHTTP要求には取得したAccessCredentialsつきのAuthorizationヘッダが付加される。

AccessCredentialsの有効期限が切れているときは、ユーザの同意なしに新らたなクレデンシャルを取得しようとする。

如何にクレデンシャルが取得されるかに関しては obtainAccessCredentialsViaUserConsentを見ること。obtainAccessCredentialsViaUserConsent からのエラーは、この関数が返されるFutureと、返されるHTTPクライアント(クレデンシャル更新の場合)を通過する。

返されたHTTPクライアントはそのFuture<Response> またはそのResponse.read()ストリームを介してエラーを下位レベルに渡す。

返されたHTTPクライアントを閉じるのはユーザの責任になる。

Future<AutoRefreshingAuthClient> **clientViaUserConsentManual**(ClientId clientId, List<String> scopes, PromptUserForConsentManual userPrompt, {Client baseClient})

OAuth2のクレデンシャルを取得し、認証を受けたHTTPクライアントを返す。

アクセス・クレデンシャル取得に使われる引数たちの詳細は obtainAccessCredentialsViaUserConsentManual を見ること。

一旦アクセス・クレデンシャルが取得されたら、この関数は自動更新のHTTPクライアントで完了する。もしアクセス・クレデンシャルの有効期限が切れたら、新たなクレデンシャルを取得する為に自分の更新トークンを使用する。更なる情報は autoRefreshingClientを見ること。

baseClientは与えられていないときは自動的にこれが生成される。これは認証を受けたHTTP要求をする際に使われる。

返されたHTTPクライアントを閉じるのはユーザの責任である。返されたクライアントを閉じてもbaseClientは閉じない。

Future<AccessCredentials> **obtainAccessCredentialsViaMetadataServer**(Client baseClient)

ComputeEngine上でメタデータAPIを使ってOAuth2のアクセス・クレデンシャルを取得する。

このVMが要求されたスコープ内で設定されていない、あるいはエラーが発生したときは、futureはExceptionで完了する。

clientはアクセス・クレデンシャルを取得する際に使われる。

クレデンシャルの必要はない。しかしながらこの関数はGoogle APIsにアクセスするよう設定された Google Compute Engine VM上で動作するようにのみ意図されたものである。

Future<AccessCredentials> **obtainAccessCredentialsViaServiceAccount**(ServiceAccountCredentials clientCredentials, List<String> scopes, Client baseClient)

サービス・アカウント・クレデンシャルを使ってOAuth2 AccessCredentialsを取得する。

サービス・アカウントが要求されているスコープへのアクセスを持っていない、あるいは別のエラーが発生したときは、返されたfutureは Exceptionで完了する。

ServiceAccountCredentialsを取得するのに使われる。
ServiceAccountCredentialsはGoogle Cloud Consoleのなかで取得できる。

Future<AccessCredentials> **obtainAccessCredentialsViaUserConsent**(ClientId clientId, List<String> scopes, Client client, PromptUserForConsent userPrompt)

OAuth2認証コード・フローを使ってOAuth2認証アクセス・クレデンシャルを取得する。

そのユーザがこのアプリケーションに対しそのデータへのアクセスを与えたときは、返されるfutureは AccessCredentialsで完了する。それ以外の時は UserConsentExceptionで完了する。

userPromptはuser/user-agentをURIに振り向けるときに使われる。更なる情報は PromptUserForConsentを見ること。

clientは認可コードからアクセス・クレデンシャルを取得するのに

ClientIdはGoogle Cloud Consoleで取得できる。

Future<AccessCredentials> **obtainAccessCredentialsViaUserConsentManual**(ClientId clientId, List<String> scopes, Client client, PromptUserForConsentManual userPrompt)

OAuth2認証コード・フローを使ってOAuth2認証アクセス・クレデンシャルを取得する。

そのユーザがこのアプリケーションに対しそのデータへのアクセスを与えたときは、返されるfutureは AccessCredentialsで完了する。そうでない時は UserConsentExceptionで完了する。

別のエラーが発生したときは返されるfutureはExceptionで完了する。

userPromptはユーザ/ユーザ・エージェントをあるURIに向けさせるのに使われる。更なる情報は PromptUserForConsentManualを参照のこと。

clientは認可コードからアクセス・クレデンシャルを取得するのに

ClientIdはGoogle Cloud Consoleで取得できる。
Future<AccessCredentials> **refreshCredentials**(ClientId clientId, AccessCredentials credentials, Client client)
client を使ってcredentialsに基づいてアクセス・クレデンシャルの更新を試みる。
Future<AccessCredentials> **obtainAccessCredentialsViaUserConsent**({bool forceUserConsent: true})
OAuth 2.0のアクセス・クレデンシャルを取得する。 forceUserConsent がtrueのときは、新たなポップアップ・ウィンドウが作られる。そのユーザには彼に代わってアクセスしようとするアプリケーションのスコープ（有効範囲）のリストが提示される。該ユーザはそのアプリケーションへのアクセス要求を承認するか拒否するかをする。 forceUserConsent がfalseのときは、ユーザの同意なしにアクセス・クレデンシャルを取得しようとする。 返されたfutureは、もしそのユーザがそのデータへのアクセスをそのアプリケーションに与えた場合は、AccessCredentialsで完了する。そうでない場合は、 UserConsentExceptionで完了する。 別の要因でエラーが発生したときは、返されるfutureはException.で完了する。

第26章 Google App EngineでDartを走らせる

筆者はDartのプロジェクトが発足したときからそのサーバ・サイドの潜在性に注目し、開発の進捗にあわせて主としてサーバ・サイドのアプリケーションをその都度解説してきました。前章で示したように、DartチームはこのところGoogleのサービスとの統合にかなりの労力を投じています。そして2014年11月7日にRunning Dart server applications on Google Cloud Platform（Googleクラウド・プラットホーム上でDartのサーバ・アプリケーションを走らせる）というタイトルで、Googleのクラウド・サービス上でのDartのサーバの実行が可能になったと発表しました。

この章は同時に公開されたDart and Google Cloud Platform（DartとGoogleクラウド・プラットホーム）という資料の翻訳です。下手な翻訳なので、なるべく本文のほうをご利用ください。

なお概要はGoogleデンマークのSøren Gjesse氏の解説ビデオを見てください。

訳者は以下の環境で確認しました：

- Windows 7 64bit
- Chrome

Linux等その他のOSの読者はこの背景色の訳者のコメントの一部は適用されません。

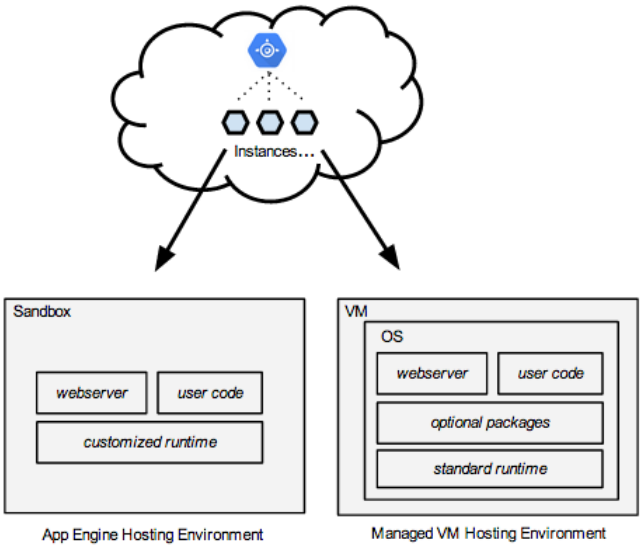
またDart Server-side and Cloud Developmentというフォーラムが立ち上がっていますが、今後このグループは拡大すると思いますので、時々覗いてみることをお勧めします。

26.1節 アプリケーション・エンジンと管理されたVMについて (About App Engine and Managed VMs)

App Engineを使うと、Googleのインフラストラクチャ上でアプリケーションの構築と実行ができ、高性能、負荷分散、セキュリティその他の利点が得られます。これまではApp Engineはセキュアでサンドボックス化された環境で総てのアプリケーションを走らせ、Python、Java、Go、およびPHPの4つのプログラミング言語に対応していました。

現在App Engineはサンドボックス化されたホスティング環境に加えて、皆さんのアプリケーションをホストするための管理されたVMたち(Managed VMs)にも対応しています。このVMベースのホスティング環境により、より柔軟性が得られ、またより広いCPUとメモリの選択肢が得られます。加えて、Dartを含むいろんなプログラミング言語で書かれたアプリケーションが使えます。

訳者注：Managed VMsは2014年11月時点では未だベータ段階です。



App Engineが用意している2つのホスティング環境

App EngineのManaged VMsに関する一般的な情報はManaged VMsというドキュメントを読んでください。

前提

Managed VMs上でDartアプリケーションの作業をする前に、DartでHTTPクライアントとサーバを書き慣れていなければなりません。未だの場合は、Writing HTTP Clients & Servers in Dartを読んでください。

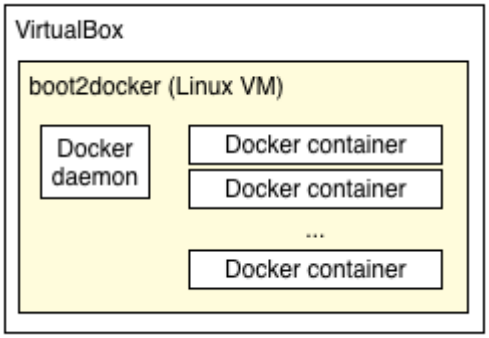
訳者注:訳者の「プログラミング言語 Dartの基礎」の一読も検討ください。

26.2節　　App Engine開発の為のセットアップ

必要なもの

App Engine Managed VMsではこのクラウドの為のアプリケーションを開発・配備するためにDockerが使われています。Dockerはポータブルで、軽量ランタイムで、アプリケーションたちをシェアするためのクラウド・サービスであり、Linuxのみの技術です。読者がLinux上で開発を行っていない場合は、Dockerを使う為にLinux VMが必要です。本手順書ではLinux VMをホストするのに、クロス・プラットホームの仮想化アプリケーションであるOracle社のVirtualBoxを使用します。

Google Cloud SDKにはローカルのApp Engine開発サーバがあり、クラウドにいると同じ環境(到来要求を配分する、APIたちを提供する、等)をローカルに持てます。



Docker、boot2docker及びVirtualBoxがともに走る

本ページにある手順は、如何に必要なソフトウエアをインストールと設定するかを示しております:

- **Dart**
  Dart Editorのバンドルには App Engine Managed VMsに配備できるウェブ・ベースとサーバ・サイドのアプリケーションを作成・編集・テスト・構築するために必要なすべてが含まれています。

- **Virtual Box version 4.3.10 またはそれ以降のバージョン**
  VirtualBoxはDockerVMデモンをホストし、また総てのdockerコンテナたちを実行させます。
  訳者注:これはDockerに同梱されています。

- **boot2docker**
  boot2dockerは VirtualBoxの為のあらかじめパッケージされているLinux VMイメージです。このイメージにはdockerデモンのインストール物、および boot2dockerコマンド行ツールが含まれています。
  訳者注:これはDockerに同梱されています。

- **docker**
  dockerコマンド行・ツールはVirtualBox内で走っているdockerデモンと交信します。

- **Google Cloud SDK**
  Google Cloud SDKにはgcloudコマンド行ツールを含むApp Engineの総てのツールとコマンドが含まれています。総ての機能はgcloud preview appというサブ・コマンドを介して使えます。

Dartのダウンロードとインストール

Dartをダウンロードし ZIPファイルを解凍するとdartディレクトリができます。

dart/dart-sdk/binにPATHを設定します。

Mac OSの場合:

```
$ export PATH=$PATH:<インストール・ディレクトリ>/dart/dart-sdk/bin
```

Windowsの場合:

```
> set PATH=%PATH%;C:<インストール・ディレクトリ>/dart/dart-sdk/bin
```

Linuxの場合:

```
$ export PATH=${PATH}:<インストール・ディレクトリ>/dart/dart-sdk/bin
```

訳者の場合は次のようにpath設定しています:

```
>set PATH=%PATH%;c:\dart\dart-sdk\bin
```

確認はdart --versionを実行します。

セットアップ法(Windows (Vista, 7, or 8)の場合)

Linux及びMacの場合は翻訳を省略してあるので、本文のほうをご覧ください。

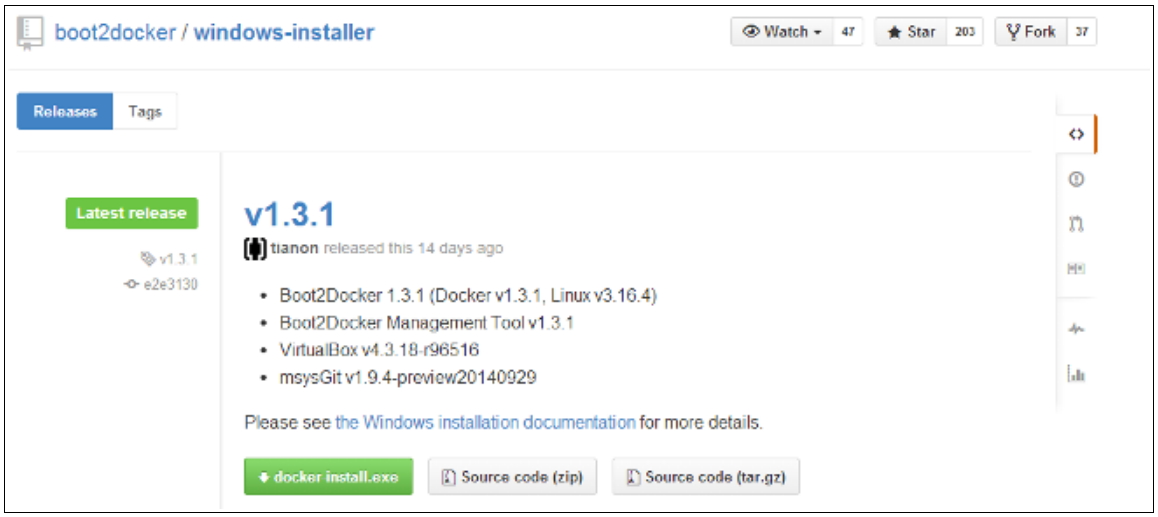
訳者注:WindowsのOSは64ビットでなければなりません。

Dockerと関連ツールのダウンロードとインストール

- Dockerのウェブ・サイトに行きインストール手順に従う。これはVirtualBox、docker、boot2docker管理ツール、および幾つかの必要なプログラムをインストールします。

注意:これらの手順はDockerと boot2dockerの v1.3.1が読者のマシンにインストールされることを想定しております。しかしながら、boot2docker VM内では Docker 1.3.0を必要とします。以下のboot2docker設定がこれを処理しています。

訳者注:インストール手順のInstallationの1項目目にあるリンクをクリックします。



Dockerのダウンロード

上図のdocker-install.exeをクリックしてインストーラを実行します。これは120MBを超える大きなファイルです。セットアップ・ウィザードに従ってインストールします。

- インストール・ディレクトリはC:\Program Files\Boot2Docker for Windowsです

- 以下のコンポネンツをインストールします
  - Boot2Docker management tool and ISO
  - VirtualBox
  - MSYS-git UNIX tools
- boot2docker.exeのパスも付加します。

インストールには暫く時間がかかります

- VirtualBoxツール、VBoxManageを読者のパスに置きます。デフォルトのインストール・ディレクトリは C:\Program Files\Oracle\VirtualBox\です:

```
>set PATH=%PATH%;c:\Program Files\Oracle\VirtualBox
```

訳者注:私の場合は環境変数はコントロールパネル→システムとセキュリティ→システム→システムの詳細設定→環境変数でシステム環境変数のなかでpathの値を次のように追加しています:

```
;C:\Program Files\Boot2Docker for Windows;c:\dart\dart-sdk\bin;c:\Program
Files\Oracle\VirtualBox;c:\Program Files (x86)\Git\bin;c:\Program Files\Google\Cloud
SDK\google-cloud-sdk\bin
```

Dockerを設定する

注意:Windowsでは、このインストール・プロセスはWindowsのdockerコマンド行ツールをインストールしないので、以下のdockerコマンドたちはVM内で走らねばなりません。

1. boot2dockerを設定するために以下のコマンドを実行させます:

```
> mkdir "%USERPROFILE%\.boot2docker"
> echo ISOURL =
"https://github.com/boot2docker/boot2docker/releases/download/v1.3.0/boot2docker.iso" >
"%USERPROFILE%\.boot2docker\profile"
> "%ProgramFiles%\Boot2Docker for Windows\boot2docker" init
```

訳者注:このコマンドを実行する前に"c:\Program Files (x86)\Git\bin"をpathに含めないとssh実行ファイルが見つからないというエラーが発生する問題があります。

以下はその実行例です(訳者の場合):

```
C:\Users\Terry>mkdir "%USERPROFILE%\.boot2docker"

C:\Users\Terry>echo ISOURL = "https://github.com/boot2docker/boot2docker/release
s/download/v1.3.0/boot2docker.iso" > "%USERPROFILE%\.boot2docker\profile"

C:\Users\Terry>"%ProgramFiles%\Boot2Docker for Windows\boot2docker" init
Downloading boot2docker ISO image...(ダウンロードに時間がかかります)
Success: downloaded https://github.com/boot2docker/boot2docker/releases/download
/v1.3.0/boot2docker.iso
    to C:\Users\Terry\.boot2docker\boot2docker.iso(ダウンロードが成功しました)
Generating public/private rsa key pair.
Your identification has been saved in C:\Users\Terry\.ssh\id_boot2docker.
Your public key has been saved in C:\Users\Terry\.ssh\id_boot2docker.pub.
The key fingerprint is:
5f:48:cc:27:9a:65:18:a9:c1:69:20:aa:fb:21:5b:b9 Terry@TERRY_NOTE
The key's randomart image is:
+--[ RSA 2048]----+
|  . .o ...       |
| . .  = .=       |
|.    . o. B .    |
|.     .  * +     |
|.       S . .    |
| . .     . .     |
```

```
|o +       .      |
| = o             |
|. E              |
+-----------------+

C:\Users\Terry>
```

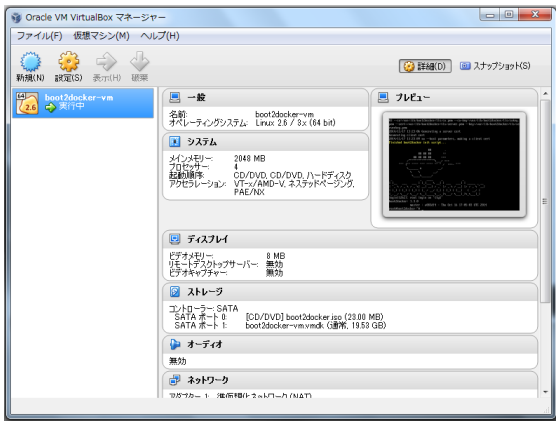
2. booto2dockerを立ち上げるのに以下のコマンドを実行させます:

```
> "%ProgramFiles%\Boot2Docker for Windows\boot2docker" up
```

これが成功すれば、その出力は以下のものに似た行となる筈です(訳者の例):

```
C:\Users\Terry>boot2docker up
Waiting for VM and Docker daemon to start...
.o
Started.
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\key.pem
Docker client does not run on Windows for now. Please use
  "boot2docker" ssh
to SSH into the VM instead.
```

ポイント:このコマンドを実行したら、デスクトップ上のOracle VirtualBoxのアイコンをクリックしてVirtualBoxを立ち上げてみてください。boot2dockerが走っていることが確認できる筈です。



VisualBox起動例: boot2docker-vmが走っている

Dockerイメージの取得

1. 環境変数DOCKER_TLS_VERIFY、DOCKER_HOST、及びDOCKER_CERT_PATHを設定します。残念ながら boot2docker はWindowsに対応していません。

最初に以下のコマンドを実行する:

```
> "%ProgramFiles%\Boot2Docker for Windows\boot2docker.exe" ssh
```

その結果以下のような3行がプリントされます:

```
    export DOCKER_HOST= ...
    export DOCKER_CERT_PATH= ...
    export DOCKER_TLS_VERIFY= ...
```

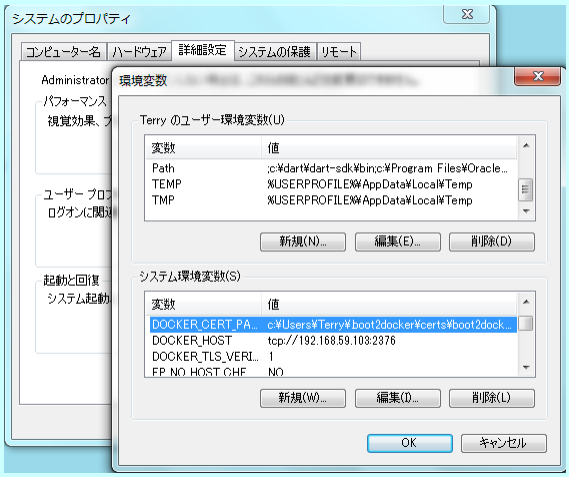
訳者注:訳者の場合はこれはプリントされません。shellinitを実行する必要があります:

```
C:\Users\Terry>boot2docker shellinit
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\Terry\.boot2docker\certs\boot2docker-vm\key.pem
    export DOCKER_CERT_PATH=C:\Users\Terry\.boot2docker\certs\boot2docker-vm
    export DOCKER_TLS_VERIFY=1
    export DOCKER_HOST=tcp://192.168.59.103:2376
```

もしあなたが通常のWindowsシェルを使っているとしたら、これらのコマンドをひとつずつ行いexportをsetで置き換えます。例えば、DOCKER_TLS_VERIFYに対しては:I

```
set DOCKER_TLS_VERIFY=1
```

訳者注:訳者の場合は下図のようにシステム環境変数をセットしています:



Docker環境変数の設定

もしあなたがcygwinを使っているとしたら、あなたは次のスクリプトを使えます:

```
$ $(boot2docker shellinit)
```

2. 以下のコマンドを使って一連のDockerイメージ(コンテナのテンプレート:イメージを元にして新しいコンテナを作成します)をダウンロードします。docker pullコマンドは少し時間がかかります。

```
> "%ProgramFiles%\Boot2Docker for Windows\boot2docker.exe" ssh
$ docker pull google/docker-registry
```

以下は訳者が試した例です:

```
docker@boot2docker:~$ docker pull google/docker-registry
Pulling repository google/docker-registry
5d4bb763edd7: Pulling image (latest) from google/docker-registry, endpoint: http
5d4bb763edd7: Download complete
511136ea3c5a: Download complete
95b32d411fbe: Download complete
(途中省略)
543bc2d2fbfc: Download complete
65c52c3c1f3a: Download complete
6470a34a110d: Download complete
36970167ca69: Download complete
2a83ff3b687a: Download complete
d0ea64f85124: Download complete
c3b4fd502f39: Download complete
Status: Downloaded newer image for google/docker-registry:latest
docker@boot2docker:~$
```

3. まだイメージが残っているか確認します:

```
$ docker images
```

このコマンドはイメージの数をリストします。

4. 以下のコマンドを使ってDart VMのバージョンをプリントします。これには boot2docker VMが走っている必要があります:

```
$ docker run google/dart /usr/bin/dart --version
```

次のような行が出力されます:

```
Dart VM version: 1.7.2 (Tue Oct 14 12:12:42 2014) on "linux_x64"
```

訳者の場合はgoogle/dartイメージを未だダウンロードしていなかったので次のような結果になりました:

```
docker@boot2docker:~$ docker run google/dart /usr/bin/dart --version
Unable to find image 'google/dart' locally
Pulling repository google/dart
cd7baf4008f8: Download complete
511136ea3c5a: Download complete
95b32d411fbe: Download complete
7a243a3158d4: Download complete
2eac30c6b22e: Download complete
3fad76ad435f: Download complete
7f720ff6e45f: Download complete
b37bdcd86f11: Download complete
Status: Downloaded newer image for google/dart:latest
Dart VM version: 1.7.2 (Tue Oct 14 12:12:42 2014) on "linux_x64"
docker@boot2docker:~$
```

5. boot2docker VMを抜けます:

```
$ exit
```

クラウド・プロジェクトの設定

App Engineにアプリケーションを開発して配備するには App Engineのプロジェクトが必要です。Google Developer Consoleへゆき、指示に従ってプロジェクトを生成します。固有の名前を付けてください。

訳者注:訳者の「プログラミング言語 Dartの基礎」の25.1節のアプリケーションの登録とクライアントIDの取得の項が参考になります。

Google Cloud SDKのインストール

1. Google Cloud SDKをインストールするには、Installing the Cloud SDKに書かれている手順に従い、次に以下の手順に戻ってください。

訳者注:Installing the Cloud SDKのページのDownload the Google Cloud SDK installer for Windowsという箇所をクリックすると GoogleCloudSDKInstaller.exeがダウンロードされます。これを実行し設定ウィザードにしたがってインストールします。

- インストール・ディレクトリはC:\Program Files\Google\Cloud SDKです

- インストールする部品にはPreview commandsも含めても構いません（含めるとその分gcloud components　updateの部品数がひとつ減ります）。
- Python 2.7.6はインストールします

インストールが終わるとコマンド・プロンプトとGoogle App Engine Launcherの画面が表示されます。アプリのローンチャは消して、コマンド・プロンプトでその後の作業を進めます。

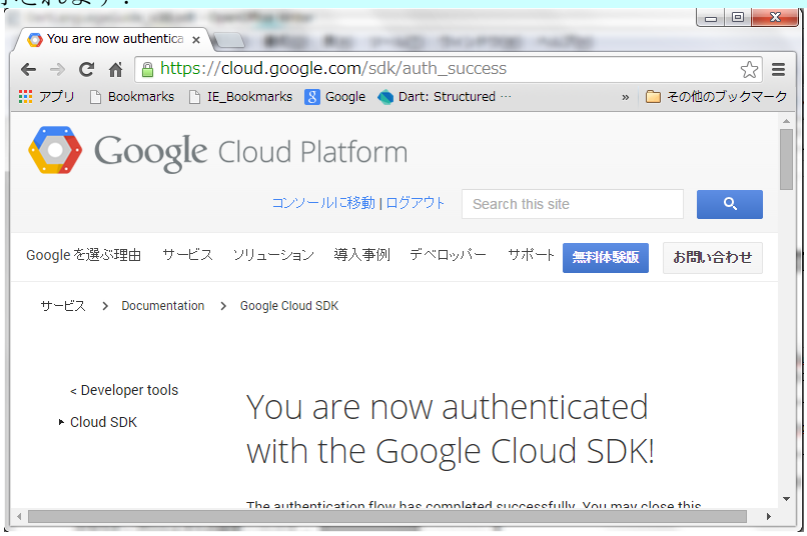
2. Google Cloud SDKはベータの機能が付いています。従ってもしあなたが Cloud SDKを使っているのなら、このベータ機能を使うのに特に何かする必要はありません。未だGoogle Cloud SDKを使ったことがないときはgoogle-cloud-sdk/binへのパスを追加設定してください。

訳者注：パスはC:\Program Files\Google\Cloud SDK\google-cloud-sdk\binです。筆者は先に示したシステム環境変数としてこれを設定しています。

3. これでmy-project-nameを「クラウド・プロジェクトの設定」の項で選択したあなたのプロジェクト名で置き換えたgcloudコマンドを使って、クラウド・プロジェクトにログインして設定できます。次に、Managed VMsサポートをインストールします。これには App Engineの為のベースDockerイメージたちが含まれています。
注意：もしあなたがVPN上にいるときはこれらのコマンドの最後は正しく実行されないかもしれません。

```
> gcloud auth login
> gcloud config set project my-project-name
> gcloud components update app
```

訳者注：最初のコマンドを実行するとブラウザが開きログインを行います。ログインが終了すると次のような画面が表示されます：



ログイン完了画面

訳者注：コマンド・プロンプトは次のような表示になります：

```
C:\Users\Terry>gcloud auth login
Your browser has been opened to visit:
  https://accounts.google.com/o/oauth2/auth?redirect_uri=http%3A%2F%2Flocalhos
.........
pis.com%2Fauth%2Fprojecthosting&access_type=offline

Saved Application Default Credentials.
You are now logged in as [terry xxxxxxxx@gmail.com].
Your current project is [None].  You can change this setting by running:
 $ gcloud config set project PROJECT
C:\Users\Terry>
```

訳者注:2行目のコマンドは初回のみ必要です。

訳者注:3行目のコマンド実行には少し時間がかかります。次のような結果になります:

```
C:\Program Files\Google\Cloud SDK> gcloud components update app
The following components will be installed:
  ---------------------------------------------------------------------
  | App Engine Command Line Interface (Preview) | 2014.11.06 |  < 1 MB |
  | App Engine Managed VMs Component (Preview)  | 2014.11.03 | 84.6 MB |
  | gcloud app Go Extensions (Windows, x86_64)  |     1.9.15 | 32.4 MB |
  | gcloud app Java Extensions                  |    1.9.15a | 88.9 MB |
  ---------------------------------------------------------------------
Do you want to continue (Y/n)?  y
Creating update staging area...
Installing: App Engine Command Line Interface (Preview) ... Done
Installing: App Engine Managed VMs Component (Preview) ... Done
Installing: gcloud app Go Extensions (Windows, x86_64) ... Done
Installing: gcloud app Java Extensions ... Done
Creating backup and activating new installation...

Done!
C:\Program Files\Google\Cloud SDK>
```

場合によって以下のようなエラーが出ることがあります。この場合はCloud SDKのフォルダを削除して再インストールしてみてください:

```
Do you want to continue (Y/n)?  y
Creating update staging area...
ERROR: (gcloud.components.update) Access is denied: [C:\Program Files\Google\Clo
ud SDK\google-cloud-sdk\.install\.backup\.install\.download]
```

これで後述のgcloud previewコマンドなどが使えるようになります。

4. 現在の設定を知るために次のコマンドを実行します:

```
> gcloud config list
```

その出力は次のようなものになる筈です:

```
[core]
account = mem@example.com
disable_usage_reporting = True
project = my-project-name
user_output_enabled = True
```

これで開発環境ができたので、コードを書き込めるようになりました。

26.3節　　HelloWorldの生成と実行

App Engine開発環境のセットアップができたら、シンプルなDartアプリを作ってそれをApp Engine開発サーバを使ってローカルに走らせることが可能になります。

訳者注:このサンプルはgithubに登録されているので、これをダウンロードし解凍しても構いません。

Dartプロジェクトの作成

総てのApp Engineアプリケーションにはそのアプリケーションの幾つかのアスペクトを設定するためのapp.yamlファイルが要ります。このファイルはDart App Engineプロジェクトのトップ・ディレクトリに置かれます。

helloworldディレクトリを作ります。これはあなたが作っているDartプロジェクト（シンプルな“HelloWorld”のサンプル）のディレクトリです。次にディレクトリをhelloworldに変えます：

```
$ mkdir helloworld
$ cd helloworld
```

以下の内容のapp.yaml ファイルを作ります：

```
version: helloworld
runtime: custom
vm: true
api_version: 1
```

app.yamlファイルの中のversionフィールドはApp Engine が配備されたアプリケーションのインスタンスたちを区別するためのひとつの手段です。

以下の中身のDockerfile（ファイル拡張子はありません）を作ります：

```
FROM google/dart-runtime
```

pubspec.yamlファイルを作る

あなたのアプリケーションをApp Engineと共に使うときは幾つかのDartのパッケージが必要です。 pubspec.yamlファイルはDartプログラムの依存パッケージを指定するものです。以下の内容のpubspec.yamlファイルを作ります：

```
name: helloworld
version: 0.1.0
author: <your name>
dependencies:
  appengine: '>=0.2.1 <0.3.0'
```

Dartのソース・ファイルを作る

このステップではDartプログラムの為のmain()を含んだ bin/server.dartファイルを作ります。 App Engineの為のDartランタイムは常にこのbin/server.dartファイルを実行することで開始します。即ちこのファイルが App Engine上のDartランタイムのインスタンスを立ち上げ、HTTP要求を処理するためのメソッドを App Engineに渡します。

binディレクトリをhelloworldディレクトリ内に作ります：

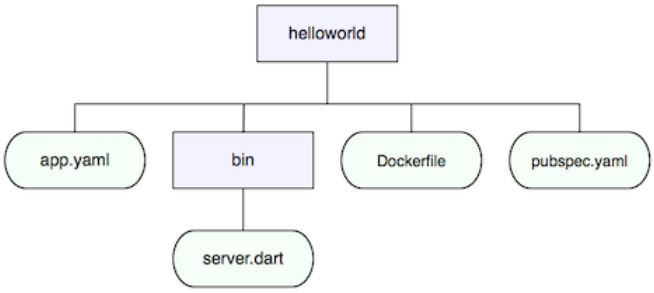
```
$ mkdir bin
```

次にHelloWorldのサンプルのコードが入った bin/server.dartを作ります：

```
import 'dart:io';

import 'package:appengine/appengine.dart';
```

```
main() {
  runAppEngine((HttpRequest request) {
    request.response..write('Hello, world!')
                    ..close();
  });
}
```

あなたのディレクトリ構造は次のようなものになっている筈です:



helloworldの構成

HelloWorldコードの説明

- このサーバがインポートしているappengineライブラリはトップ・レベルのメソッドrunAppEngine()が入っており、これがDartのランタイムを開始させ、また App Engineにこれを接続します。
- runAppEngine() メソッドは引数としてコールバック関数をとります。この関数がHTTP要求を処理します。HelloWorldのサンプルの場合はHTTP要求ハンドラ関数は requestHandlerです。
- この要求ハンドラは引数としてHttpRequestオブジェクトを受けまむ。このオブジェクトが該要求の情報を保持しています。このサーバはまたその応答をHttpRequestオブジェクトのresponseメンバを介して送信しています。Dartのサーバに関する更なる詳細はWriting HTTP Clients & Servers in Dart、または訳者の「プログラミング言語 Dartの基礎」を参照ください。

pub getを走らせる

HelloWorldプロジェクトに必要なライブラリをインストールする必要があります。これはDartをダウンロードした際他のコマンド行ツールと共に得られるpub getコマンドで行われます。

helloworldディレクトリからpub getを実行します:

```
$ pub get
$ cd ..
```

自分のアプリケーションの中の依存物解決の為にpub getを実行できます。配備中ではサーバ・サイドでは自動的にpub getが走り、必要なパッケージが使えるようにしています。

訳者注:このプロジェクトを自分のIDE上で開くと自動的にpub getが実行されます(自分のIDEのpub設定がデフォルトのままのとき)。

Dockerイメージをプルする

次のコマンドを使ってDockerイメージをプルします:

```
$ docker pull google/dart-runtime
```

このステップは暫く時間がかかり得ますが一回だけ行なう必要があります。

以下は訳者の実行例です:

```
C:\Users\Terry>boot2docker ssh
＜途中省略＞
boot2docker: 1.3.0
       master : a083df4 - Thu Oct 16 17:05:03 UTC 2014
docker@boot2docker:~$ docker pull google/dart-runtime
Pulling repository google/dart-runtime
e2ab3ccbce58: Download complete
＜途中省略＞
2df2f95bbc6a: Download complete
Status: Downloaded newer image for google/dart-runtime:latest
docker@boot2docker:~$
```

このアプリケーションをApp Engineを使って走らせる

以下の gcloud preview appコマンドを使ってこのアプリケーションを走らせます。このコマンドであなたは App Engine開発サーバを使ってローカルでこのアプリケーションを走らせることが出来ます。

```
$ gcloud preview app run app.yaml
```

訳者注:previewコマンドはgcloud components update appを実行しないとインストールされません。

gcloudの出力は以下のような行を含んだものとなります:

```
Module [default] found in file
[/usr/local/prj/dart/appengine/apps/hello/app.yaml]
INFO: Looking for the Dockerfile in /usr/local/prj/dart/appengine/apps/hello
INFO: Using Dockerfile found in /usr/local/prj/dart/appengine/apps/hello
INFO: Skipping SDK update check.
INFO: Starting API server at: http://localhost:40800
INFO: Health checks starting for instance 0.
INFO: Starting module "default" running at: http://localhost:8080
INFO: Building image <cloud_project_name>.default.my-version...
INFO: Starting admin server at: http://localhost:8000
INFO: Image <cloud_project_name>.default.my-version built, id = 77cc15d8a6e5
INFO: Creating container...
INFO: Container f5f012c233e0c6d0bded64f3f3e3b228c0790f142def7746e99362466fb76e8c
created.
INFO: default: "GET /_ah/start HTTP/1.1" 200 2
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 200 2
```

訳者注:訳者の場合は次のようになりました:

```
C:\Users\Terry>cd helloworld

C:\Users\Terry\helloworld>gcloud preview app run app.yaml
Module [default] found in file [C:\Users\Terry\helloworld\app.yaml]
INFO: Looking for the Dockerfile in C:\Users\Terry\helloworld
INFO: Using Dockerfile found in C:\Users\Terry\helloworld
INFO: Skipping SDK update check.
WARNING: Could not read search indexes from
c:\users\terry\appdata\local\temp\appengine.dartlang-tutorial\search_indexes
INFO: Starting API server at: http://localhost:49613
INFO: Health checks starting for instance 0.
INFO: Starting module "default" running at: http://localhost:8080
INFO: Building image dartlang-tutorial.default.helloworld...
INFO: Starting admin server at: http://localhost:8000
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 503 -
INFO: Image dartlang-tutorial.default.helloworld built, id = 8216c2258b4e
INFO: Creating container...
INFO: Container 240783a3a1648a18c6fc7970342310eb20afccd73f7aa1f69761fd54ecf12196 created.
INFO: default: "GET /_ah/start HTTP/1.1" 200 2
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 200 2
(以下周期的にヘルス・チェックが LOG として報告される)
```

“200 2”ステータスは立ち上げが成功していることを示します。あなたが構築している最初の数回は 50xエラーが起きるかもしれません。またこのステップは最初は少し時間がかかりますが、これはコンテナ内で pub getが走っていて数メガバイトのデータを取っているからです

訳者注:途中でブラウザからのアクセスがあると、次のようなログとなります:

```
INFO: default: "GET / HTTP/1.1" 200 13
```

訳者注:サーバを止めるにはCtrlキーを押しながらCキーを押します。プロセスを止めるにはCtrlキーを押しながらBreakキーを押します。

ポイント:自分のアプリケーションをプレビューしようとしているとき“Unable to bind localhost:8080”というエラーがでたときは、別のプロセスがポート8080を使っている可能性があります。どのプロセスがこのポートで聞いているかを調べるには以下のコマンドのどれかを使います:

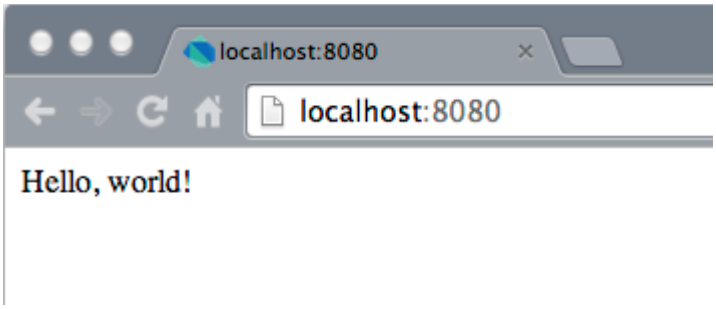
```
$ sudo lsof -i :8080  [Mac OS X or Linux]
$ netstat -ano        [Windows]
```

この場合はそのプロセスを止めるか--hostフラグを使って gcloud preview appコマンドの為のポート番号を変えるかします。例えば:

```
--host localhost:7777
--host 127.0.0.1:7777
```

このアプリケーションをブラウザから見る

自分のブラウザから http://localhost:8080にナビゲートします。次のように“Hello, World!”が表示されるはずです:

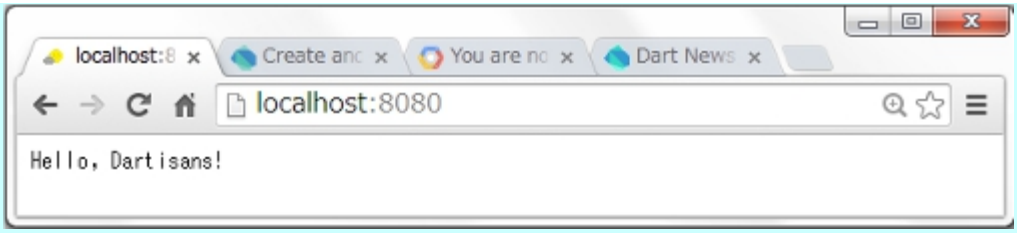


HelloWorldアプリの出力

このコードに何か変更を加え、例えば“Hello, World!”を“Hola, Mundo!” に変更し、ファイルに保管すると、配備サーバがこのDartサーバ・アプリケーションを再スタートさせます。サーバ・コードが変わっていることを調べるには、ブラウザを再ロードさせます。

自分のコードを変更する

そのアプリケーションがローカルで走っている最中では、そのアプリケーションのディレクトリ内のファイルの変更は監視されています。アプリケーションのファイルを変更するとそのアプリケーションは再スタートし新しいコードが実行されます。



訳者注:IDE上でserver.dartファイルに変更を加え、それを保管するとその変更は直ちに反映されます:ソース・コードに変更を加える

以下はそのログです:

```
INFO: [default] Detected file changes:
 bin\server.dart
INFO: Building image dartlang-tutorial.default.helloworld...
INFO: Waiting for instances to restart
INFO: Health checks starting for instance 0.
INFO: Image dartlang-tutorial.default.helloworld built, id = 3d8af337877f
INFO: Creating container...
INFO: Container 450aa03bd4b317c2441275168b027d3e079f8bf83cfeaf6995f50fcc7f21cc2a
 created.
INFO: default: "GET /_ah/start HTTP/1.1" 200 2
INFO: Instances restarted
INFO: [default] Detected file changes:
 bin\server.dart
INFO: Waiting for instances to restart
INFO: Health checks starting for instance 0.
INFO: default: "GET / HTTP/1.1" 500 48
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 503 -
INFO: Building image dartlang-tutorial.default.helloworld...
INFO: Image dartlang-tutorial.default.helloworld built, id = 3d8af337877f
INFO: Creating container...
INFO: Container c8d68a7a8174b07e24a47418987b3a520a9ae8e02dff313de1e2a91d819649f6
 created.
INFO: default: "GET /_ah/start HTTP/1.1" 200 2
INFO: Instances restarted
INFO: default: "GET / HTTP/1.1" 200 17
```

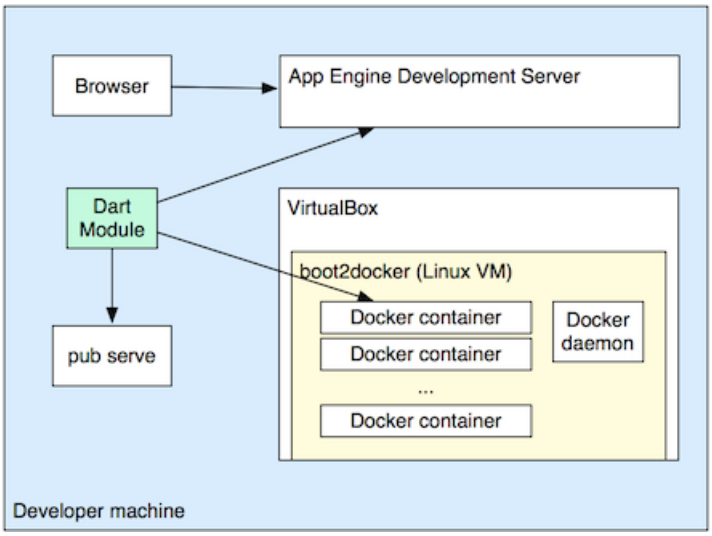
```
INFO: default: "GET /favicon.ico HTTP/1.1" 200 17
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 200 2
INFO: default: "GET /favicon.ico HTTP/1.1" 200 17
INFO: default: "GET /_ah/health?IsLastSuccessful=yes HTTP/1.1" 200 2
```

訳者注:
皆さんのマシンでこの状態まで行けたら、その後一旦皆さんのマシンを再起動させたあとでの手順は、一般的には次のようになります:

1. boot2dockerを立ち上げる >boot2docker up
2. dockerクライアントをWindows上で走らせる >boot2docker ssh
3. ディレクトリを自分のアプリケーションに移す >cd helloworld
4. gcloudにログインする >gcloud auth login
5. アプリケーションを走らせる >gcloud preview app run app.yaml
6. IDEでDartソース・コードを編集すればその結果は直ちに実行中のサーバに反映されます

どのように動作しているのか

次の図がどのように App Engine配備サーバの中でこのアプリケーションが走るのかを示しています:



如何に各部品が関連しているか

このシンプルなサンプルの場合はpub serveに悩む必要はありません。pub serveの役割に関しては client/server コードをダウンロードしてそれをローカルで走らせる中で理解することになります。

App Engineに自分のアプリケーションを配備する

helloworldを App Engineに配備するのは簡単で、以下のコマンドを使います:

```
$ gcloud preview app deploy app.yaml
```

注意:このコマンドは少し時間がかかる場合があります。

次に自分のブラウザからhttp://helloworld.my_project_name.appspot.com/をナビゲートすると、“Hello, world!”が表示されるはずです。

訳者注:ローカルにある自分のアプリケーション(Dockerイメージ)をApp Engineに配備するには、課金設定がされている必要があります。いちばん簡単なのは無料体験版を利用することです。但し上限枠を超えると課金されるので注意してください。ローカルで試している限り課金されることはありません。

26.4節　　APIの概要

以下のDartのライブラリは、コードを書いてApp EngineのManaged VMsの機能とサービスを活用できるようにしています。

appengine

このライブラリはApp Engine Managed VMs上にDartのアプリケーションを実行しまたホストするために必要な関数とクラスを用意しています。あなたのDartアプリケーションは隔離されたコンテナ内で走り、またgcloudとmemcacheといったApp Engineのサービスにアクセスできます。

あなたのアプリケーションはこのライブラリをインポートする必要があります:

```
import 'package:appengine/appengine.dart';
```

あるApp Engineのインスタンスを開始させるには、コールバック関数を引数にしてトップ・レベルのrunAppEngine()メソッドを呼びます。HTTP要求が到来したら、このコールバック関数がその要求処理のために呼び出されます。

要求ハンドラ内から、あなたは context.servicesを使ってApp Engineのいろんなサービスにアクセスできます。例えば、Cloud DatastoreサービスAPIにアクセスするには次のように記述します:

```
context.services.db
```

このライブラリにはまたログ、ユーザ、及びエラーのAPIが含まれています。

gcloud

このgcloudパッケージにはCloud Datastore及びCloud Storageを含む一連のAPIが含まれており、Dartのオブジェクトから cloud datastoreへのマッピングを提供するために使えます。

あなたのコードはこのライブラリをインポートする必要があります:

```
import 'package:gcloud/db.dart'
```

cloud datastorというのはあなたのアプリケーションのデータのための拡張性があり、スキーマなしの、ストレージで

す。

datastore内のエンティティは親と子が持て、従って階層的構造を持てます。共通の先祖からの子孫のエンティティは同じエンティティ・グループに属します。単一のエンティティ・グループへのクエリは強く一貫した結果を返します。他のクエリでは最終的に一貫した結果を返します。

このライブラリに含まれている重要なクラスを挙げます:

- DatastoreDB
  App EngineのDatastoreへの主たるインターフェイスで、このクラスはトップ・レベルのwithTransaction()とquery()という関数があります。

- Query
  Queryは サーチ・クライテリアを使ってのエンティティまたはエンティティのグループに対するDatastoreへのクエリのオブジェクトです。このQueryクラスには注文しその結果をフィルタリングするメソッドがあります。更に結果をリファインするのにfilter()を、ソートするのにorder()を使います。

  ```
  var db = context.services.db;
  db.query(Greeting)..order('date');  // sort by date
  db.query(Greeting)..filter('date'); // filter out anything that is not a date
  ```

- Transaction
  トランザクションはデータベースへのアトミックな操作または操作のセットです。データベース内にエンティティを挿入、更新、または削除するのにTransactionを使います。Datastoreエンティティへの修正はTransaction内で行われねばなりません。挿入と削除の設定にはqueueMutations()を使い、次にその変更を恒久化するのにcommit()を呼びます。このTransaction内での総ての修正は成功裏に完了せねばならず、そうでないときはこのTransactionは何の効果も持たず、rollback()メソッドを使ってロールバックされねばなりません。

memcache

このライブラリはApp Engineの高性能な分散イン・メモリ・キャッシュのシステムへのインターフェイスになります。このライブラリのメインのクラスである Memcacheは、このキャッシュからオブジェクトを取得、セット、削除するためのトップ・レベルの関数を持っています。

Memcacheサービスはキーから値へのマップで構成された共有キャッシュを提供します。キーと値の双方はバイナリ(バイトのリスト)です。 Memcache内で使われるキーの値は最大250バイト長です。値のほうの最大長はこのmemcacheサービスの設定に依存します。もっとも一般的なデフォルトのサイズは1M(1メガバイト)です。

このキャッシュ内の各エントリには有効期限が設定でき指定した時間間隔または指定した時刻に達したらこのキャッシュから取り除かれます。このキャッシュはサイズが限られており、このサービスによりアイテムが外される場合(一般にLeast Recently Used (LRU)ポリシーに基づいて)があります。

26.5節　クライアントとサーバのアプリケーション

HelloWorldのサンプルでは最も簡単なDartのアプリケーションをあなrたのローカル・マシン上でApp Engine開発サーバ上で作成、実行、及びテストする手段を示しました。この節ではどのようにクライアント/サーバのアプリケーションを作成、実行、及びテストするかを示します。最後にこれをどのようにしてクラウドに配備するかを学びます。

クライアントはHTMLとDartで実装されており、ブラウザ内で走ります。サーバはDartでのみ書かれており、App EngineのDartランタイムのなかで走ります。 HelloWorldのサンプル場合と同様に、サーバのコードはbinディレクトリ内に置かれます。クライアントのコードはwebディレクトリ内に置かれます。

このアプリケーションではユーザがあるリストにアイテムを付加できます。ユーザがタイプインするとクライアントはサーバにHTTP要求を送信し、サーバがこれに応答します。サーバは各アイテムをCloud Datastoreにストアするので、ユーザがこのアプリケーションにまた戻ってきたときもそのアイテムは存続します。クライアントとサーバはアイテムのリストを交信するのにJSON書式のデータを使います。

クライアント/サーバのコードを取得してローカルに走らせる

1. GitHubのappengine_samplesをダウンロードし解凍します。このレポジトリ内にclientserverディレクトリがります。

2. このclientserverディレクトリ内でこのclient/serverプロジェクトに必要なライブラリを取り込むのにpub getを実行します。このプロジェクトはappengine、gcloud、memcacheその他を依存物としています。

```
$ cd clientserver
$ pub get
```

3. ひとつの端末ウィンドウ内で次のように pub serveを実行します:

```
$ pub serve web --hostname 192.168.59.3 --port 7777
```

訳者の場合は次のような結果が得られます(IPアドレスに注意):

```
C:\Users\Terry\clientserver>pub get
Resolving dependencies...
Got dependencies!

C:\Users\Terry\clientserver>pub serve web --hostname 169.254.60.121 --port 7777
Loading source assets...
Serving clientserver web on http://192.168.59.3:7777
Build completed successfully
```

未だ一般化していないDartで書かれたHTMLクライアントを一般のブラウザが実行できるようにするためには変換/コンパイルのスッテプが必要です。このステップはこのプロジェクトが使っている総ての変換器を走らせます。これらの変換器のひとつがDartからJavaScriptへのコンパイルになります。

pub serveコマンドはあなたのDartウェブ・アプリケーションのサーバを立ち上げます。pub serveはHTTPサーバであり、あなたのクライアントのコードを渡す役割を持ちます。このpub serveコマンドはあなたのDartコードをJavaScriptにコンパイルする Dart-to-JavaScript変換器を自動的に含めます。これを使えばあなたはDartコードに変更を加え、あなたのクライアントを再ロードさせ、直ちにその変更の結果が見られます。

App Engine managed VMのアプリケーションでは、pub serveは総てのdockerコンテナたちが共有するアダプタを聞かねばなりません。殆どの場合、これは192.168.59.3というIPアドレスになります。もしこのアドレスでうまくいかなかったときは、ifconfig (MacとLinux)またはipconfig (Windows)を使ってどのIPアドレスを使うべきかを判断してください。一般的にipconfigの出力はこのアダプタをvboxnet1としてリストアップします。ipconfigではVirtualBox Host-Only Ethernet Adapterという名前で表示されます。更なる詳細はboot2docker configurationを見てください。

訳者注:訳者のマシンでは以下の2つが表示されますが、最初のアドレスを使用します:

```
イーサネット アダプター VirtualBox Host-Only Network:
   接続固有の DNS サフィックス . . . :
   リンクローカル IPv6 アドレス. . . . : fe80::8419:27be:6cdf:3c79%22
   自動構成 IPv4 アドレス. . . . . : 169.254.60.121
   サブネット マスク . . . . . : 255.255.0.0
   デフォルト ゲートウェイ . . . . . :

イーサネット アダプター VirtualBox Host-Only Network #2:
   接続固有の DNS サフィックス . . . :
   リンクローカル IPv6 アドレス. . . . : fe80::8102:3e1f:f597:f96%26
   IPv4 アドレス . . . . . : 192.168.59.3
   サブネット マスク . . . . . : 255.255.255.0
   デフォルト ゲートウェイ . . . . . :
```

4. DART_PUB_SERVE環境変数をapp.yamlファイルに追加します:

```
env_variables:
  DART_PUB_SERVE:  'http://192.168.59.3:7777'
```

訳者注:Dart Eritor上で次のように追加します(IPアドレスは自分のマシンに合わせる):

```
# Copyright (c) 2014, the Dart project authors.  Please see the AUTHORS file
# for details. All rights reserved. Use of this source code is governed by a
# BSD-style license that can be found in the LICENSE file.
version: clientserver
runtime: custom
vm: true
api_version: 1
env_variables:
 DART_PUB_SERVE:  'http://169.254.60.121:7777'
```

訳者注:7777はpub serveの専用のポート番号です。

5. ここで client/serverアプリケーションを走らせます。別の端末ウィンドウ(もうひとつコマンド・プロンプトを立ち上げる)から、gcloudを次のように実行します:

訳者注:boot2dockerが走っていることを確認してください。

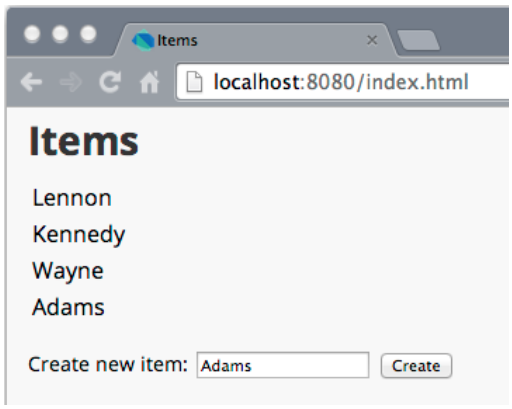
```
$ gcloud preview app run app.yaml
```

訳者の場合は次のような結果になります:

```
C:\Users\Terry\clientserver>gcloud preview app run app.yaml
Module [default] found in file [C:\Users\Terry\clientserver\app.yaml]
INFO: Looking for the Dockerfile in C:\Users\Terry\clientserver
INFO: Using Dockerfile found in C:\Users\Terry\clientserver
INFO: Skipping SDK update check.
INFO: Starting API server at: http://localhost:50993
```

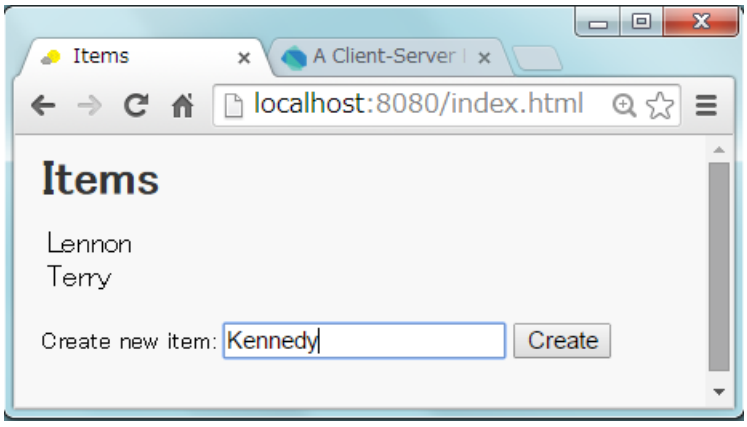
```
INFO: Health checks starting for instance 0.
INFO: Starting module "default" running at: http://localhost:8080
INFO: Building image dartlang-tutorial.default.clientserver...
INFO: Starting admin server at: http://localhost:8000
INFO: Image dartlang-tutorial.default.clientserver built, id = 8a15635f7b52
INFO: Creating container...
INFO: Container 723f87f1e3cc9f58c1cf2063144bc786a385af6dee1fa19cc205427984053af8 created.
INFO: default: "GET /_ah/start HTTP/1.1" 200 2
INFO: default: "GET /_ah/health?IsLastSuccessful=no HTTP/1.1" 200 2
```

6. ブラウザからhttp://localhost:8080/にナビゲートします。次のような画面が得られる筈です:



clientserverアプリのアクセス例

訳者の場合は次のようになりました(Chrome使用):



クライアント画面

以下はpub serveのコンソール例です:

```
C:\Users\Terry\clientserver>pub serve web --hostname 169.254.60.121 --port 7777
Loading source assets...
Serving clientserver web on http://169.254.60.121:7777
Build completed successfully
[web] GET /index.html => clientserver|web/index.html
[web] GET /packages/browser/dart.js => browser|lib/dart.js
[web] GET /index.css => clientserver|web/index.css
[Info from Dart2JS]:
Compiling clientserver|web/index.dart...
[Info from Dart2JS]:
Took 0:00:24.305086 to compile clientserver|web/index.dart.
Build completed successfully
[web] GET /index.dart.js => clientserver|web/index.dart.js
```

```
[web] GET /favicon.ico => clientserver|web/favicon.ico
^C バッチ ジョブを終了しますか (Y/N)? y
```

- pub serveを止めるにはCtrlキーを押しながらCキーを押します。
- [web]のログはクライアント要求に基づき静的リソースを渡したことを示しています。
- 途中でDart2JSを起動してindex.dartをJavaScriptにコンパイルしてindex.dart.jsを生成しています。

一方gcloudのコンソールは次のようなログが出力されます:

```
INFO: default: "GET /index.html HTTP/1.1" 200 639
INFO: default: "GET /packages/browser/dart.js HTTP/1.1" 200 1270
INFO: default: "GET /index.css HTTP/1.1" 200 570
INFO: default: "GET /_ah/health?IsLastSuccessful=yes HTTP/1.1" 503 -
INFO: default: "GET /index.dart.js HTTP/1.1" 200 388086
INFO: default: "GET /items HTTP/1.1" 200 28
INFO: default: "GET /favicon.ico HTTP/1.1" 200 -
```

- これはブラウザからの到来要求をどう処理したかを示しています。
- gcloudを止めるにはCtrlキーを押しながらBreakキーを押します。

ソース・ファイルたち

appengine_samplesをダウンロードして得られるクライアントとサーバの双方のアプリケーションのソース・ファイルは次のようです。

- app.yaml
  appID、最新のバージョンの識別子、及び要求ハンドラへのURLマッピングのようなこのアプリケーションのアスペクトを設定します。このファイルはDartプロジェクトのトップ・レベルに位置します。

- pubspec.yaml
  Dartプログラムのパッケージ依存物を指定します。このファイルは総てのDartプロジェクトで必要で、Dartプロジェクトのトップ・レベルに位置します。

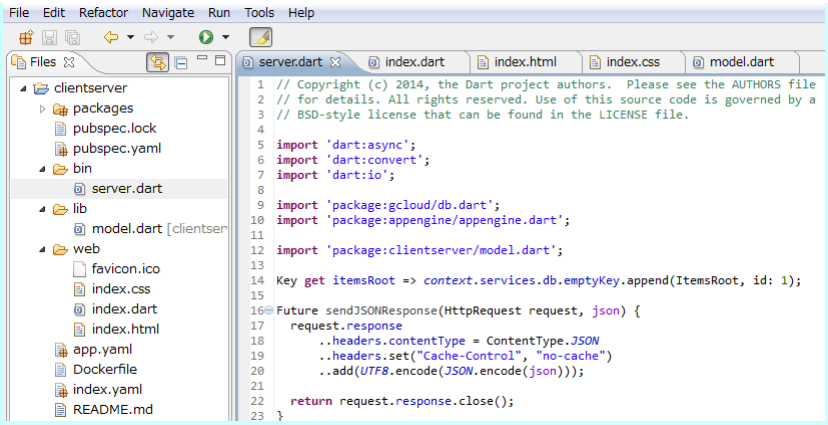
- bin/server.dart
  DartのHTTPサーバを実装します。 App Engine managed VM経由のクライアント・プログラムからのHTTP要求を受信し取り扱います。このサーバは App Engineのdatastoreからのエンティティをストアし検索します。

- web/index.html
  クライアントの為のユーザ・インターフェイスとなります。このファイルにはスクリプトのpackages/browser/dart.jsがあり、非Dartのブラウザとの互換性を扱います。

- web/index.dart
  Dart HTTPクライアントを実装しています。cloud datastore内にストアされているデータはItemクラスとして実装されており、これはweb/model.dartファイル内で定義されています。

- web/model.dart
  Dartの型のItemの gcloudモデルで、この型のオブジェクトはCloud Datastoreにストアできます。

訳者注:このアプリケーションは次のような構成になっています。自分のIDEで確認すると良いでしょう。



clientserverの構成

クライアントとサーバ間のコードの流れ

このクライアント/サーバのアプリケーションの流れは次のようです:

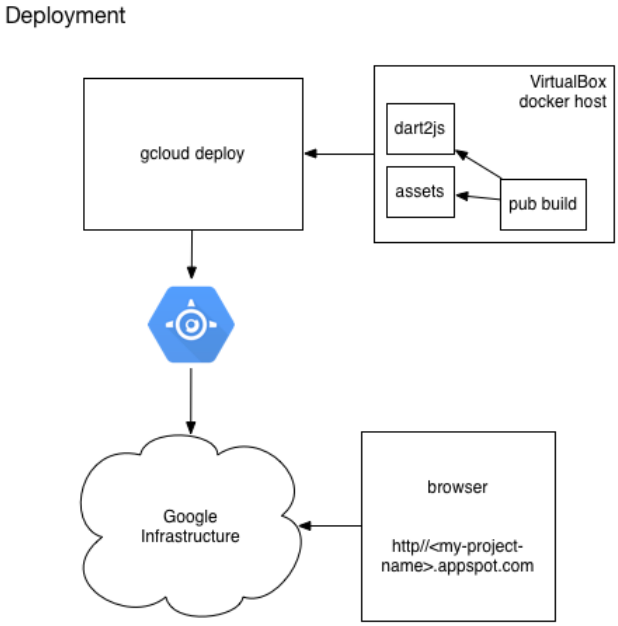
1. mainメソッドがApp Engineのインスタンスを起動させ、requestHandlerコールバック関数を通過します。
2. サーバの中の鍵となるメソッドはrequestHandlerで、これがその要求及びhandleItemsを処理しそこからitemsを取り出しそのitemsをdatastoreに送信します。
3. HTTP要求が到来したら、App Engine上のDartのランタイムはrequestHandlerを呼び出し、HttpRequestのインスタンスを渡します。
4. クライアントが送信したHTTP要求に基づき、このハンドラは既にlistの中にあるitemsを表示するメインのページを返すか、そのlistをクリアするか、またはitemsをJSON書式で表示するかをします。
5. クライアントの中ではテキスト・フィールドにテキストをタイプインし、ボタンをクリックすると/itemsというURIパスでのPOST要求が生成されます。ハンドラはそのitemをlistに追加し、context.services.db.commitメソッドを使って Cloud Datastoreにそのlistを保管します。サーバは次にJSON書式でクライアントに応答を返すので、クライアントはUIを更新できます。
6. クライアントが立ち上がったら、そのクライアントは/itemsというURIパスでGET要求を送信します。サーバはdatastore内のitemsをロックし、それをJSON書式でクライアントに渡します。
7. localhost:8080/cleanを使うとlistがクリアされ、datastoreから総てのitemsが除去されます。
8. localhost:8080/itemsを使うとlistがJSON書式で表示されます。

如何にこのアプリケーションが機能させられているか

上記で示したように、pubサーバ(pub serveで走る)はあなたのDartアプリに必要です。このpubサーバはlocalhost上のHTTPサーバであり、イメージとかスタイルだとかのクライアントのリソースをクライアントに渡します。静的リ

ソースのサービスに加えてこのpubサーバはまた変換器を走らせることでそれらを生成します。ある変換器は入力のリソース(Dartファイルなど)を出力リソース(JavaScriptとHTMLのような)に変換します。これらの出力リソースはファイル・システム内にはおらず、それらはpubサーバ内にのみ存在します。このpubサーバは自動的にDartコードをJavaScriptに変換するdart2js変換器を自動的に含めます。これを使うと、Dartコードの一部を変更し、ブラウザを再ロードさせ、またすぐにその変更を見ることができます。

App Engine開発サーバはpub serveを使ってあなたの Dart App Engineアプリのリソースを取得します。



配備

配備に先立ち、次のようにマニュアルで pub build webを実行します:

```
$ pub build web
$ gcloud preview app deploy app.yaml
```

これにより総ての必要なリソースをディレクトリに静的ファイルとして生成します。これらの静的ファイルは次にサーバに配備されます。 deployコマンドは少し時間がかかることがあるので注意してください。

注意:もしあなたがカスタムのdatastoreインデックスを使っているときは、それを指定する index.yamlが必要です:

```
$ gcloud preview app deploy app.yaml index.yaml
```

26.6節　　クライアント・コードの説明

クライアント・アプリケーションはmodel.dart及びindex.dartの2つのファイルで構成されています。加えてこのアプリケーションはindex.htmlというHTMLファイルを含みます。

model.dart

gcloudパッケージはDartのオブジェクトからCloud Datastoreへのマッピングを提供しています。Modelを継承しEntityでアノテートされたDartのクラスはCloud Datastoreにストアできます。

model.dartソース・ファイルはItemクラスの為のこれらのマッピングを提供します。Itemはユーザによってlistの中に置かれた、そしてサーバによってgcloud datastoreに保管されるひとつのitemを表現しています。 model.dartファイルはサーバによって含められます。

以下はmodel.dartファイルの内容です:

```
library model;

import 'package:gcloud/db.dart';

@Kind()
class ItemsRoot extends Model { }

@Kind()
class Item extends Model {
  @StringProperty()
  String name;

  validate() {
    if (name.length == 0) return "Name cannot be empty";
    if (name.length < 3) return "Name cannot be short";
  }

  Map serialize() => {'name': name};
  static Item deserialize(json) => new Item()..name = json['name'];
}
```

index.dart

これはクライアント・プログラムの為の完全な実装です。App Engine Dartランタイムに固有な唯一のコードは list: Item内の各itemの為に使われるdata typeです。このクラスはModelで裏付けされており、この型のオブジェクトはgcloud datastoreに保管できます。ハイライトされたコードがどのようにしてある名前を持ったItemを生成し、直列化し、また直列化されたものを戻し、そしてそれを検証するかを示しています。

以下は index.dartファイルの中身の総てです:

```
import 'dart:async';
import 'dart:convert';
import 'dart:html';

import 'model.dart';

var nameInput;
var itemsTable;
var errorMessage;

void main() {
  querySelector("#create")
      ..onClick.listen(onCreate);
  nameInput = querySelector("#name");
  itemsTable = querySelector("#items");
  errorMessage = querySelector("#error_text");
```

```
  restGet('/items').then((result) {
    result.forEach((json) => addItem(Item.deserialize(json)));
  });
}

void addItem(Item item) {
  var row = new TableRowElement();
  var cell = new TableCellElement();
  cell.text = item.name;
  row.children.add(cell);
  itemsTable.children.add(row);
}

void onCreate(MouseEvent event) {
  var item = new Item()..name = nameInput.value;
  var error = item.validate();
  if (error != null) {
    window.alert(error);
  } else {
    restPost('/items', item.serialize()).then((result) {
      if (!result['success']) {
        errorMessage.text = 'Server error: ${result['error']}';
      } else {
        errorMessage.text = '';
        addItem(item);
      }
    });
  }
}

Future restGet(String path) {
  return HttpRequest.getString(path).then((response) {
    var json = JSON.decode(response);
    if (json['success']) {
      errorMessage.text = '';
      return json['result'];
    } else {
      errorMessage.text = 'Server error: ${json['error']}';
    }
  });
}

Future restPost(String path, json) {
  return HttpRequest.request(path, method: 'POST', sendData: JSON.encode(json))
      .then((HttpRequest request) {
        return JSON.decode(request.response);
      });
}
```

26.7節　　サーバ・コードの説明

この節ではサーバ・コードの幾つかのメソッドを示し、また App Engineとそのサービスに関連した行を説明します。

Cloud Databaseサービスへのアクセス

Cloud Database、memcache、等々といったApp Engineのサービスにアクセスするにはcontext.servicesが使えます。例えば、次の行は Cloud Datastoreサービスのハンドルをどのように取得するかを示しています:

```
context.services.db
```

このハンドルから、commit()及びquery()といったこのサービスのメソッドを呼び出せます:

```
context.services.db.query(...);
```

runAppEngine()を実行する

main()関数は App Engineを開始させるためにrunAppEngine()を呼びますが、その際App Engineがクライアントからの各HTTP要求到来毎に呼び出すrequestHandler()を用意します:

```
main() {
  runAppEngine(requestHandler);
}
```

リソースに対するサービス

requestHandler()メソッドはURIパスに基づいて到来したクライアント要求を受付け、特別なものはそこで処理します。もしURIパスがこれらのテストに合わないものなら、このプログラムはハイライトされた行を呼びますが、これはindex.htmlのようなクライアントのファイルたちを渡すものです。実際特別なURIパスの幾つかは処理され次にindex.htmlにリダイレクトされます。

```
void requestHandler(HttpRequest request) {
  if (request.uri.path == '/items') {
    handleItems(request);
  } else if (request.uri.path == '/clean') {
    handleClean(request).then((_) {
      request.response.redirect(Uri.parse('/index.html'));
    });
  } else if (request.uri.path == '/') {
    request.response.redirect(Uri.parse('/index.html'));
  } else {
    context.assets.serve();
  }
}
```

context.assets.serve();メソッドはpub serveからまたはウェブ・ディレクトリから(--dart-pub-serveオプションがgcloud app runで渡されているかどうかによって)リソースをとってきます。これには非Dart対応のブラウザで走らせるコンパイルされたJavaScriptファイルが含まれます。

pub serveを使わないと変換器が使われず、シンプルなクライアントリソースのみが動作します。

Cloud Datastoreへのitemsのコミット

datastoreにitemをコミットするにはcommit()メソッドを使います。このcommit()メソッドは非同期で操作を行うためにFutureを返します。

```
handleItems(HttpRequest request) {
  if (request.method == 'GET') {
    // Get items from datastore using the readItems method;
    // send them to client in JSON format.
    ...
  } else if (request.method == 'POST') {
    // Validate request, then commit new item to the datastore.
    ...
        return context.services.db.commit(inserts: [item]).then((_)
  ...
  }
}
```

このサーバのhandleClean()メソッドはlistから総てのitemを削除するのにcommit()メソッドを使っています:

```
Future handleClean(HttpRequest request) {
  return readItems().then((items) {
    var deletes = items.map((item) => item.key).toList();
    return context.services.db.commit(deletes: deletes);
  });
}
```

クエリを実行する

サーバのreadItems()メソッドの中のコードは当初から走っているこのアプリケーションに結び付けられたdatastoreのなかの総てのitemsを検索するクエリを生成します。次にquery.run()を呼びますが、これは非同期操作のためStreamを返します。

```
Future<List<Item>> readItems() {
  // Get items from datastore.
  var query = context.services.db.query(
      Item, ancestorKey: rootKey())..order('name');
  return query.run.toList();
}
```

order()メソッドは返されたlistをアルファベット順にソートします。

rootkeyを使う

rootKeyメソッドはhandleItems()とreadItems()が呼び出していますが、これは cloud databaseにitemsを追加するためのものです。 ItemsRootはクライアントのModel.dartで定義されており、itemを置くエンティティ・グループを識別します。

```
rootKey() {
  var rootKey = context.services.db.emptyKey.append(ItemsRoot, id: 1);
```

```
}
```

26.8節　App Engine上にDartアプリケーションを配備する

あなたのDartアプリケーションを App Engine Managed VMs上に配備するには、固有の名前のGoogle Cloudプロジェクトが必要です。このプロジェクト名があなたのアプリケーションのURLの一部になります。

```
<my-project-name>.appspot.com
```

以下の手順を踏むなかで、<my-project-name> はあなたのプロジェクト名で置き換えて下さい。

アプリケーションを配備する

1. 未だそうしていないのなら、dart/dart-sdk/binをあなたのPATH環境変数に含めてください。次のステップではこれが必要です。

2. あなたがこれまでに使っていたClient/Server Example を含むディレクトリにディレクトリを変更し、pub buildを実行します。

   ```
   $ cd appengine_samples/clientserver
   $ pub build
   $ gcloud preview app deploy app.yaml
   ```

   このコマンドは暫く時間がかかることがあります。

   注意:もしあなたがカスタムのdatastoreインデックスを使っているときは、それを指定する index.yamlが必要です:

   ```
   $ gcloud preview app deploy app.yaml index.yaml
   ```

3. プロジェクトURLをナビゲートし、Dartのアプリケーションが走っていることを確認します。

   ```
   http://<my-project-name>.appspot.com/
   ```

おめでとうございます！　あなたは App Engine Managed VMsを使ったウェブ上に自分のDartアプリを配備しました。

如何に配備が機能しているか

配備に先立ち、コンテナを構築する前にクライアントのコードが含まれていたwebディレクトリ上で pub buildを走らせます。これにより buildディレクトリが生成され、そこには総ての変換されたリソースが含められます。アプリケーションは配備時に pub serveからリソースを得るのに使ったと同じコードを使ってここからリソースを取得します。

第27章 本資料作成にあたってこれまでにDart開発チームに行った指摘と提案

クライアント・サイドのアイソレートに使えるタイマが存在していないこと。	これに関しては既にhttp://dartbug.com/1880で2つのライブラリに重複したインターフェイスがあるからという理由で2月からバグ登録されている。しかしながらチーム内での意見が合わないで進捗が無いので、督促のコメントをこのバグにもいれてある。 6月22日にやっとこのチームはTimerをdart:isolateに移した。 但しアイソレートからは未だTimerをアクセス出来ない状態が続いている。
HttpServerの応答がTCPレベルで細分化されていること。	これに関してはそこまでチューニングしていないからという回答だったが、これはバッファリングされていない為で、恥ずかしいことだということを認識してもらう為に更にコメントを入れてある。 2013年6月25日にやっと改良作業を完成させている。
HttpRequest.queryParametersが多バイト文字をデコードしていないこと。	どうやらASCIIしか念頭になかったようである。これに対しては5月11日にパッチが実施された。またURLエンコードとデコードのメソッド要求に対しては、dart:uriというライブラリが追加された。
Windows-31Jを含む文字セット対応の要求。	これはDartの広報担当みたいな役のSeth Ladd預かりとなっている。
Set-Cookieヘッダがフォールドされていること、及びHttpSession組み入れの提案。	Set-Cookieフォールド問題は直前に修正作業がされていた。またCookieクラスが新たに導入されている。HttpSessionに関しては、セッションID保護の為にもネーティブ実装が好ましいことを説明した。これを受け10月4日にDartチームはセッション・オブジェクト追加作業を開始した。 10月22日にDartチームはr13870としてIOライブラリに追加した。
WebSocketに関するバグ http://code.google.com/p/dart/issues/detail?id=5209 及び http://code.google.com/p/dart/issues/detail?id=5210	これは直ちに修正された。
HttpSession強化提案	HttpSessionのクッキー・ヘッダにhttpOnly属性を追加してセキュリティを上げることの提案。これは直ちに受け入れられた。
Dart:io v2に絡むWebSocketにおけるバグ	onDoneイベントがクライアントからの解放に対応しなくなった。これは先方が気づいて修正していた。
HttpSessionにURLエンコーディングやSSLセッションIDの組み入れ提案	これは色々考えた末、取り下げた。
HTTPセッションのタイムアウトでその後のHTTP応答がクライアントに渡されなくなるバグ	これはまだリリースされていないcontinuous buildで修正されていた。
http_serverパッケージのフルパス・ファイル名処理とデフォルト・エンコーディングに関する3つの問題点の不具合報告	翌日の10月10日にAnders Johnsenが早速手直しを開始し、迅速にパッチを完了させている。。

アップロードされたファイルの処置に関する要求	先方は直ちにAPI変更を開始した。
Dart:htmlのHttpResponse.getString.then.catchErrorがError 型のオブジェクトを返さず、エラーの内容が得られない不具合報告	このドキュメントは不親切で、同じような指摘がGoogle社内の別のユーザからも出されている。JS consoleのエラー・メッセージのほうがはるかに親切である。

第28章 推奨IDE (IntelliJ / WebStorm)のインストール

2015年4月28日に開催されたDart Summitのなかで1.11版から推奨開発環境（IDE）としてDart EditorからDart Plugin for IntelliJ/WebStormに切り替え、Dart Editorは廃止すると発表された。これはかなり衝撃的な動きで、Dart VM実装Chromeの開発停止の発表とともに大きな議論を起こした。Dart EditorはDart言語開発の初期からこのコミュニティのなかで馴染み親しまれてきたものである。

従って2015年6月以降、本資料ではIDEとしてIntelliJ IDEA CEを採用することとした。多少使える機能は限定されるが、Dartを試してみたい読者には、ツールが無料であることが絶対条件である。IntelliJ CE（無料）とWebStorm（有料）のDartコード開発の為の機能の相違はStack Overflow上で議論されているのでそれを見ていただきたい。

IntelliJ CEで出来る機能：

- サーバ・サイドのアプリのデバッグ
- Pubspec.yaml と Pub の統合 (Pub Get)

IntelliJ CEで出来ない機能：

- YAML サポート（YAMLファイルの編集）
- ブラウザ・プラグインを介したJavascriptのデバッグ
- ブラウザ・プラグインを介したDartコードのデバッグ

Dart言語にある程度馴染んだあとで、WebStormまたはIntelliJ IDEAを購入されることをお勧めする。

28.1節　Dart SDKとDartiumのダウンロード

IntelliJ IDEA CEのダウンロードとインストールに先立ち、Dartのダウンロードのページからから Dart SDKとDartiumをダウンロードしておく。

- Dart SDK：　Dart VM、ライブラリ(libフォルダ)、および dart, dart2js, dartanalyzer, pub, およびdartdocgenを含むコマンド行ツールたち（binフォルダ）で構成されている
- Dartium（必須ではない）：　Dart VMが含まれている Chromium（新しい機能が含まれているChromeブラウザのリリース前のもの）の特別な版

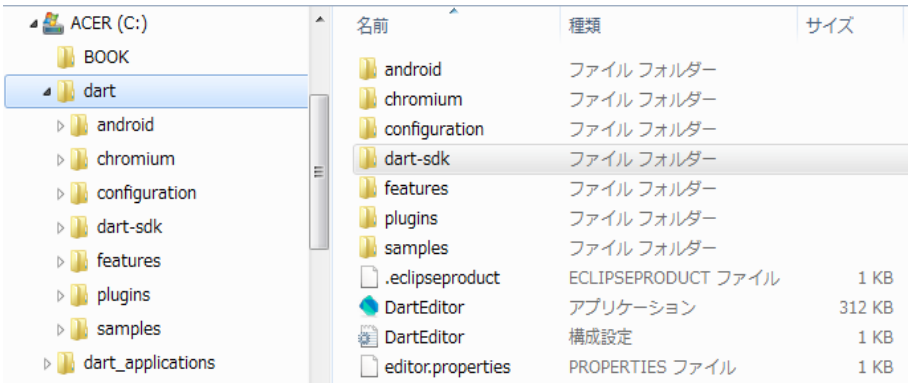
マニュアルによるダウンロード

まずIndex of Downloadsのページからから Dart SDKとDartiumのStable channelから安定版(stable version)のZIP圧縮ファイルをダウンロードする。

- 64ビットのWindowsの場合は”Dart SDK(SHA-256)”の64ビット版と”Dartium(SHA-256)”の32ビット版をダウンロードする
- 32ビットのWindowsの場合は”Dart SDK(SHA-256)”の32ビット版と”Dartium(SHA-256)”の32ビット版を

ダウンロードする

これらのファイルを解凍して、適当なパスに置く。筆者の場合はc:\dartの下に次のように配置している。



dart_sdkとDautiumのダウンロード

即ち:

- dart-sdk(binとlibのフォルダが含まれる)のパス:c:\dart\dart-sdk
- Dartium(実行ファイル)のパス:c:\dart\chromium\chrome.exe

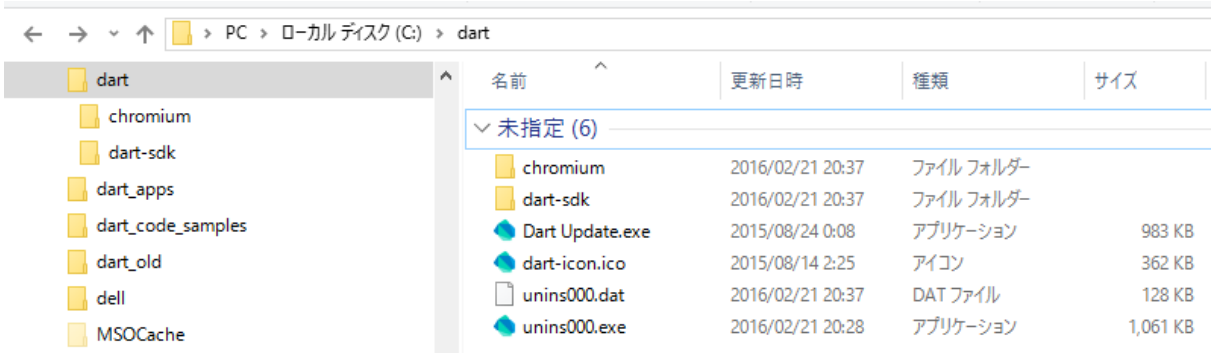
である。

Windows用インストーラによるダウンロード

また2016年2月時点でサードパーティのWindows(64ビット)ユーザ向けのインストーラがテスト版として存在している。初めてダウンロードするときは、このページの画面の上部にある安定版またはアーリー版のボタン(”GET DART(64-BIT Windows, STABLE)”または”GET DART(64-BIT Windows, DEV)”)をクリックしてインストーラをダウンロードし、それを実行する。そうするとウィザードが表示されるので、それに従ってインストールする。

- インストール・ディレクトリはC:\dartとする
- satrt menuフォルダは生成しないことにチェックを入れる

この場合は次のようなフォルダとファイルがC:\dartにダウンロードされる:



インストーラによるdart_sdkとDautiumのダウンロード

これにはDartUpdate.exeというアップデートのための実行ファイルが存在している。単にアップデートするだけな

らこのコマンドを使用する。

Chocolateteyを使ったインストール

2015年5月時点では、dart-sdkの最新のバージョンがインストールされているかを調べダウンロードする機能は無い。Windowsのコマンド プロンプトから操作する簡単なパッケージ・マネージャ（Linuxのapt-get相当）であるChocolateyを活用することが一つの対策として検討されている。2015年9月にはChocolateyでこの時点での安定版のdart-sdkおよびDartiumの1.12.1版がコマンド・プロンプトから簡単にインストールできるようになったとの記事がある。Chocolateyはコマンド・プロンプトを管理者として開き、簡単なコマンドを張り付けてそれを実行させればよい。具体的にはChocolateyのサイトまたは日本語の記事などを参考にすると良い。

dart-sdkのバージョン1.12.1をインストール：

```
choco uninstall dart-sdk
choco install -y dart-sdk  -version 1.12.1
```

Dartiumのバージョン1.12.1をインストール：

```
choco uninstall dartium
choco install -y dartium  -version 1.12.1
```

28.2節　　IntelliJ IDEA CEのダウンロードとインストール

ここではWindows（Vista以降）のユーザを対象としている。

このIDEはJavaの実行環境(JRE)が必要である。JREはブラウザ等によって既にインストールされている場合が多いが、ない場合は以下のようにインストールすればよい。

- java.comのダウンロードのページから使用するブラウザのビットに対応した最新のJavaのJREをダウンロードする。これはideaIC-2016.2.4.exeといった実行ファイルであり、これを実行する。

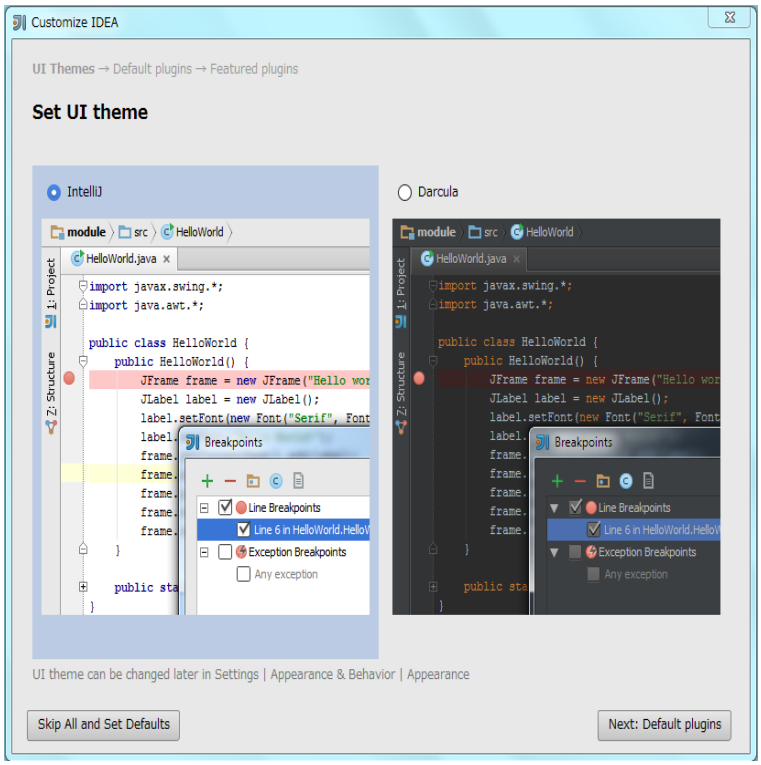
IntelliJ IDEA CEは例えばこのURLからダウンロードできる。

1. このページにある”Community”の下にある”Download”ボタンをクリックするとインストールの為の実行ファイル（200MBを超えるideaIC-xxx.exeファイル）がダウンロードされ、実行が開始される。その後は設定ウィザードに従えば良い

設定ウィザード

1. インストールに先立ち他の実行中のアプリケーションを終了させておくことが好ましい
2. Install Locationはデフォルトのままでよい
3. Installation OptionsのCreate Associationsはなにもチェックを入れない
4. Start Menu FolderはJetBrainsのままにしてInstallボタンをクリックするとインストールを開始する

2. インストールが終了したらIntelliJ IDEA CEの設定を行う
    1. IntelliJ IDEA CEを最初に起動すると以前使っていたIntelliJ IDEAの設定をインポートするかどうかを聞いてくるので、デフォルトのインポートしないという指定でOKボタンをクリックする
    2. UI themeはどちらか好きなものを選択してSkip All annd Set Defaultsをクリック

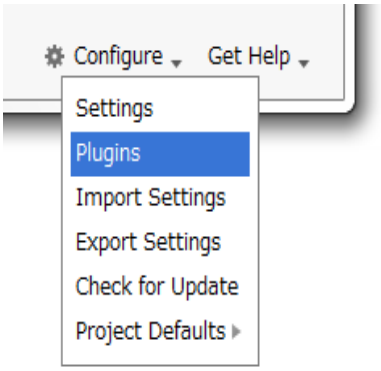


UI themeの選択

3. そうすると下図のようなWelcome画面が表示される

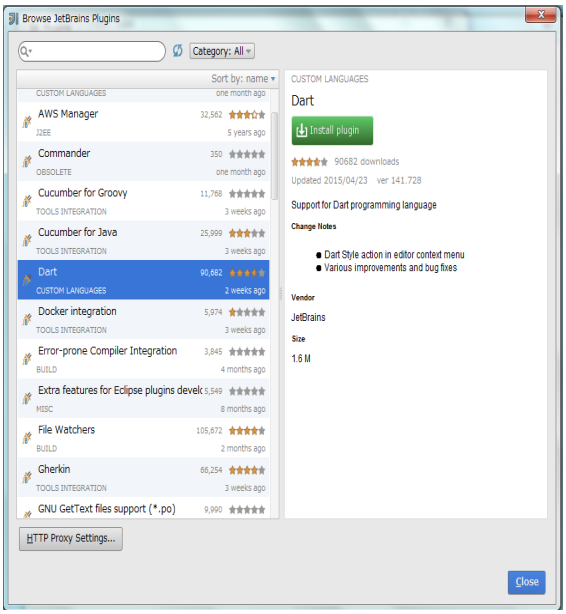
Welcome画面

4. このWelcome画面の右下にあるConfigure（設定）をクリックするとプルダウン・メニューが表示されるので、Pluginsを選択する。なおこの設定はWelcome画面からでなく、あとでFile → Settings → Pluginsで実行できる



プラグインの設定を選択

5. つぎに左下のInstall JetBrains pluginをクリックし(あるいはサーチ機能を使って)、Dartを探す



Dartプラグインを選択

6. Install pluginをクリックしてプラグインをダウンロードしてインストールする

7. もし読者のコンピュータが企業内のプロキシの配下にあるときは、HTTP Proxy Settingsボタンをクリックして、所定のプロキシ設定を行う

8. Dartプラグインのインストールが終了したら、Restart IntelliJ IDEAをクリックして、IntelliJ を再起動する必要がある。

28.3節　　幾つかのオプショナルな設定事項

以下にあらかじめ設定することが好ましい幾つかの事項を紹介する。

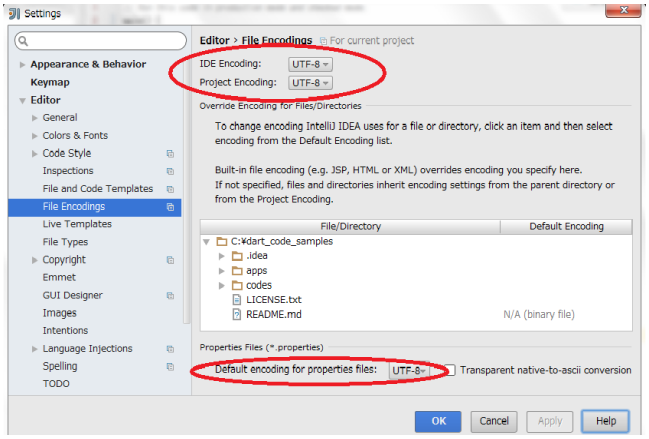
Dartプラグインの管理

File → Settings → Pluginsで開いたウインドウでDartプラグインのアップデートがあるかを調べ、必要ならそれをダウンロードする。その場合はRestart IntelliJ IDEAをクリックして、IntelliJ を再起動する必要がある。

ファイルのエンコーディング

DartではUTF-8が標準的に使用されており、その他のエンコーディングを使用することは推奨されない。このことはShift-JIS (Windows-31J)が良く使用されている日本では不便ではあるが、ウェブ・アプリケーションの世界ではむしろ好ましいものである。

- File → Settings → File EncodingsでEditor > File EncodingsでIDE及びプロジェクトのエンコーディングを該プロジェクトに対し設定する。
- その下のProperties Files（各国言語対応に関する情報を保持している）もUTF-8にする。



UTF-8を選択

- Applyボタンをクリックして確定する

コンソール出力のUTF-8化

Windowsの場合はこれをしないと日本語文字のコンソール出力が文字化けを起こす。

- C:\Program Files (x86)\JetBrains\IntelliJ IDEA Community Edition 14.1.3\binにある
  - idea64.exe.vmoptions　および
  - idea.exe.vmoptions

  に対し、以下のオプションの行を追加する：

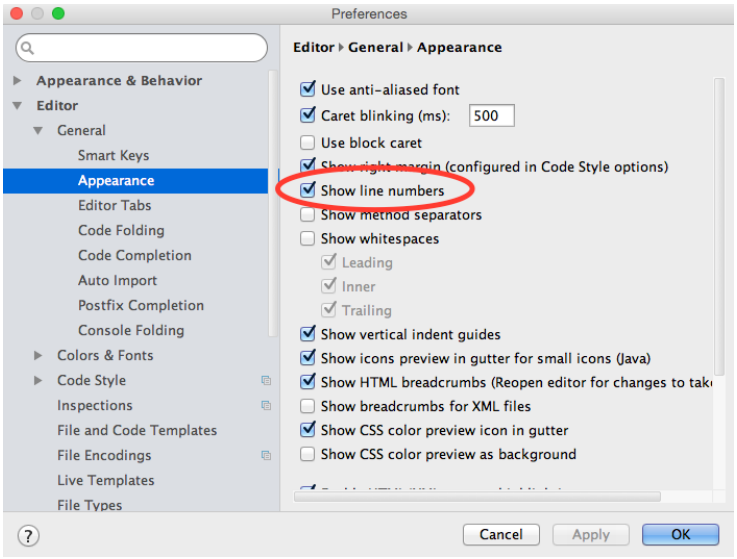
  ```
  -Dfile.encoding=UTF-8
  ```

  これらのファイルの変更は管理者(administrator)権限で行うこと。

- ファイルの変更が終了したら、IntelliJ IDEAを再起動する。

行番号の表示

- File → Settings → Editor → General → Appearanceで下図のようにShow line numbersをチェックする:

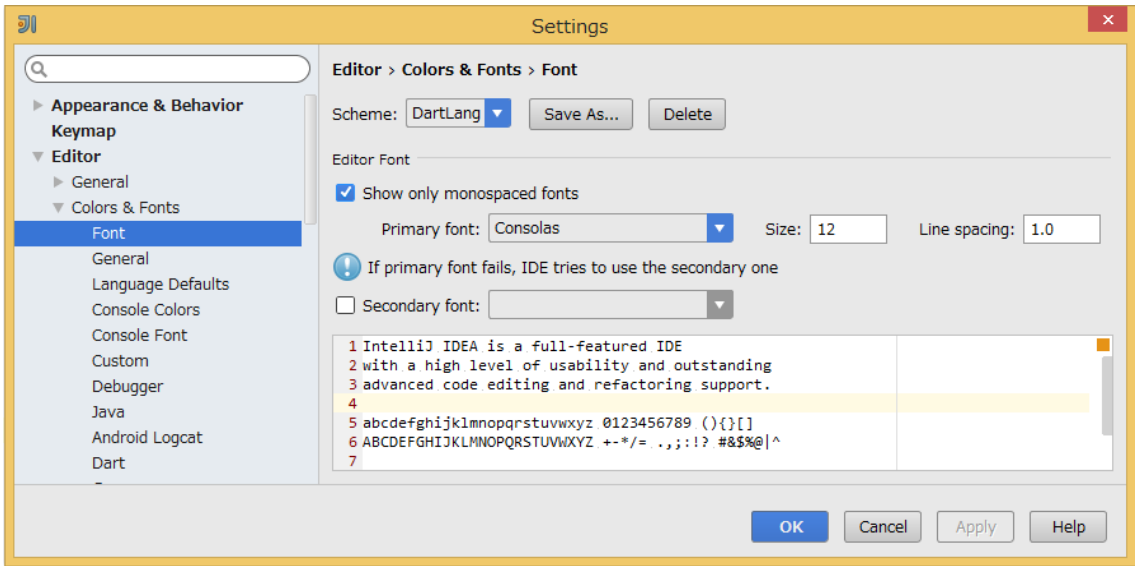


行番号の表示

フォントの変更

- File → Settings → Editor → Colors and Fonts → Fontでまず自分のスキームをSave as..で(例えばdartlangという名前で)セットする。

- 次に好みのモノスペースのフォントを選択する。

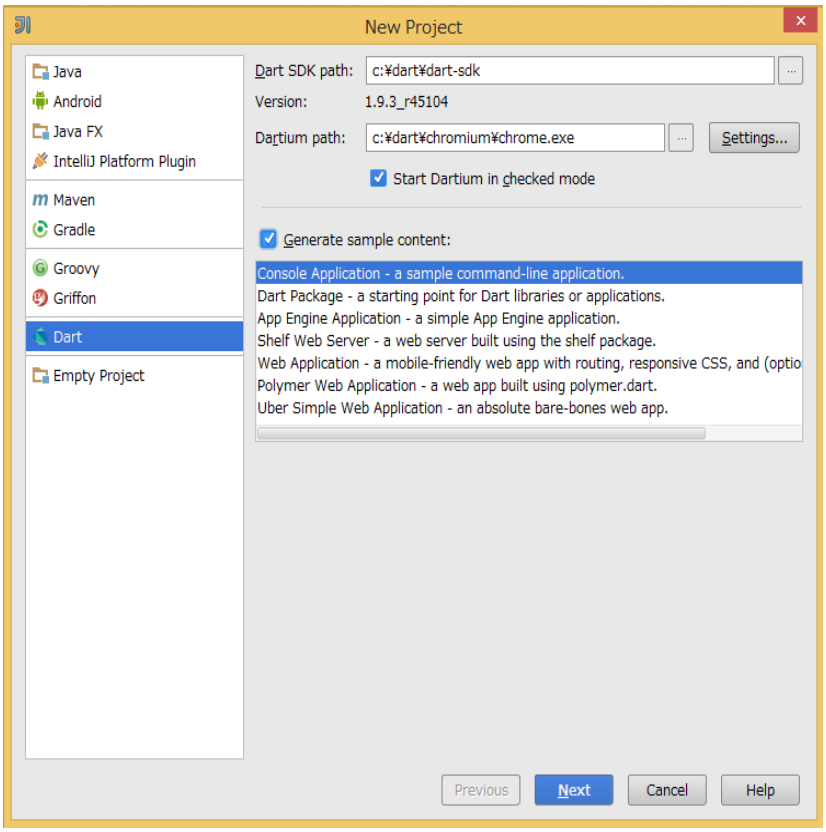


28.4節　Dartのサンプルを試してみる

IntelliJ IDEAの基本的な使い方は日本語の「クイックスタート」などを見ていただきたい。

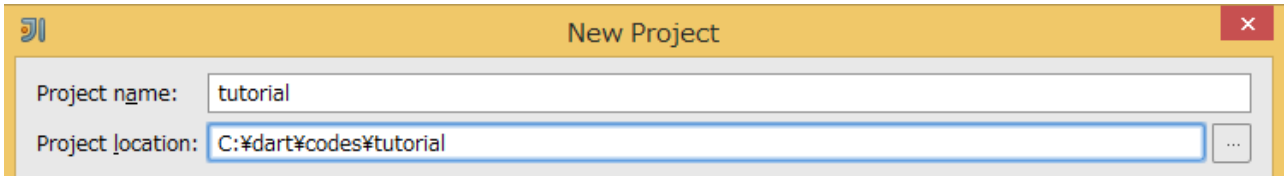
1. Welcome画面またはFile → New → Project から、Create New Projectを選択する。

2. ここでDart SDK pathとDartium pathを指定する。またStart Dartium in checked modeとGenerate sample contentにチェックを入れ、簡単なコマンド行アプリのサンプルを選択する。



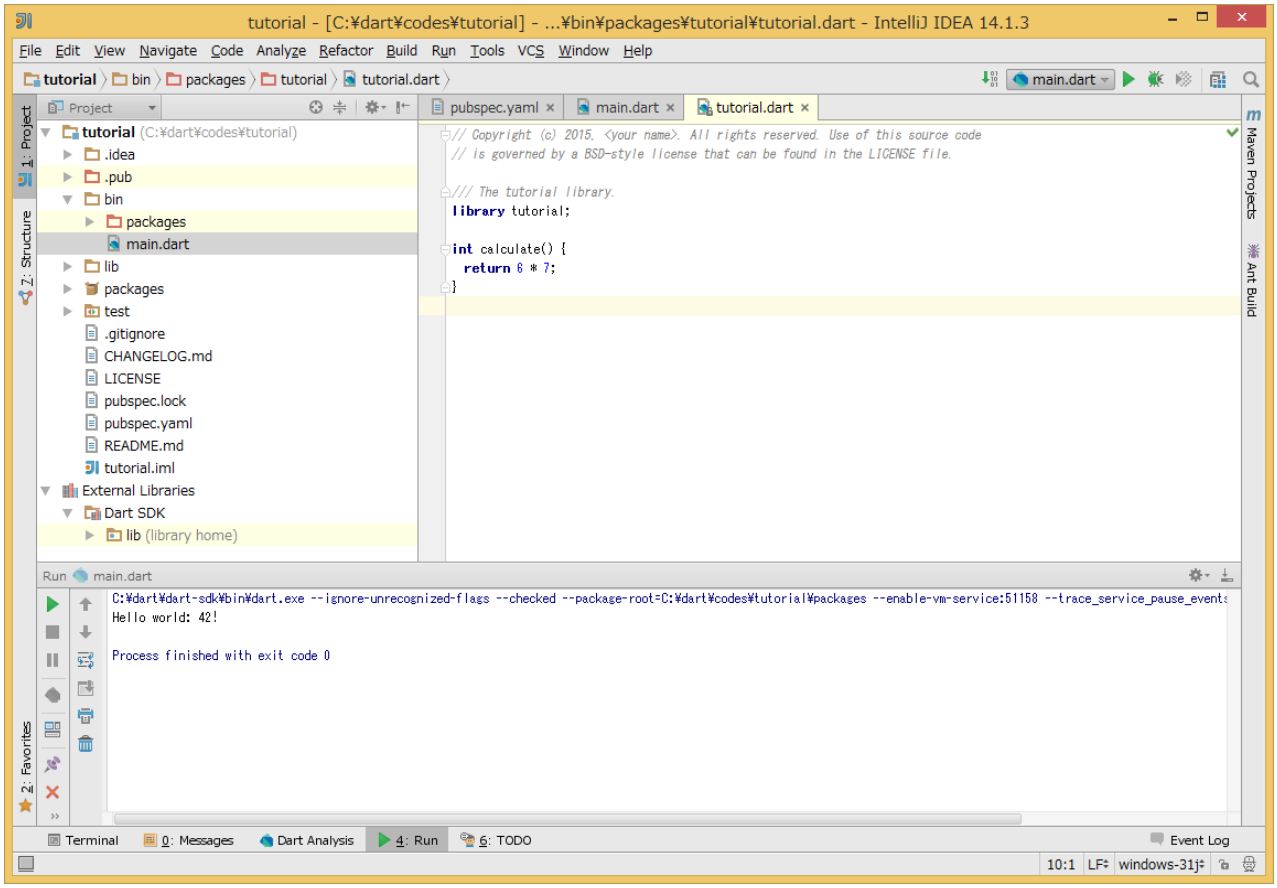
パスの指定とサンプルの選択

3. Nextボタンをクリックし、新規プロジェクト名と場所を指定する。この例ではc:\dart\codesにtutorialというプロジェクトを作っている。



新規プロジェクトの作成

そうすると下図のようなコンソール・アプリケーション（コマンド行アプリケーションとも言う）のサンプルが展開される：



コマンド行アプリのサンプル

このプロジェクトの核となっているのが次の3つのファイルで、ユーザが作成しなければならない:

- C:\dart\codes\tutorial**pubspec.yaml** このプロジェクトがどのようなライブラリのパッケージを使っているかを指定するヤムル・ファイルである。pubパッケージ・マネージャがこの情報をもとに必要なライブラリを展開する。

pubspec.yaml

```
name: tutorial
version: 0.0.1
description: A sample command-line application.
#author: <your name> <email@example.com>
#homepage: https://www.example.com
environment:
 sdk: '>=1.0.0 <2.0.0'
#dependencies:
#  foo_bar: '>=1.0.0 <2.0.0'
dev_dependencies:
 test: '>=0.12.0 <0.13.0'
```

- C:\dart\codes\tutorial\bin**main.dart** メインのDartコードである。コマンド行のコードはbinフォルダに置かれる。mainという名前である必要はないが、main()というトップ・レベルのメソッドが含まれていなければならない。

main.dart

```
import 'package:tutorial/tutorial.dart' as tutorial;

main(List<String> arguments) {
```

```
 print('Hello world: ${tutorial.calculate()}!');
}
```

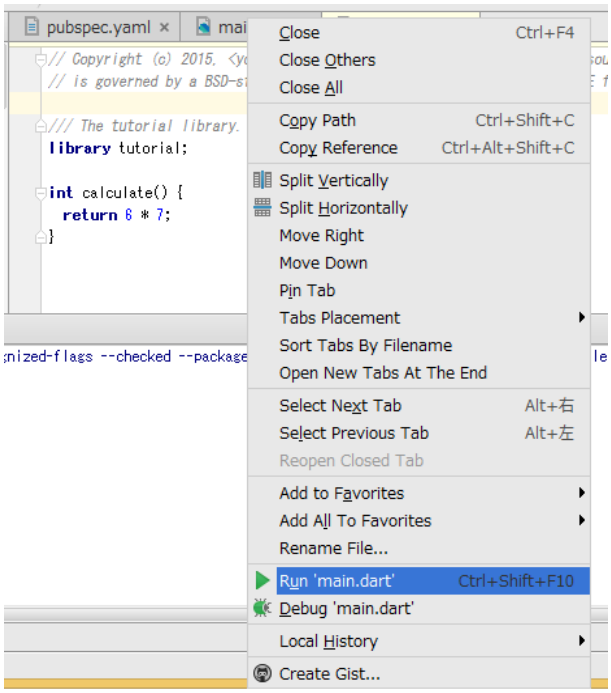
- C:\dart\codes\tutorial\packages\tutorial**tutorial.dart** このサンプルではユーザが作成したライブラリのパッケージも使用している。このコードはcalculate()というメソッドを含んだライブラリである。

tutorial.dart

```
/// The tutorial library.
library tutorial;

int calculate() {
 return 6 * 7;
}
```

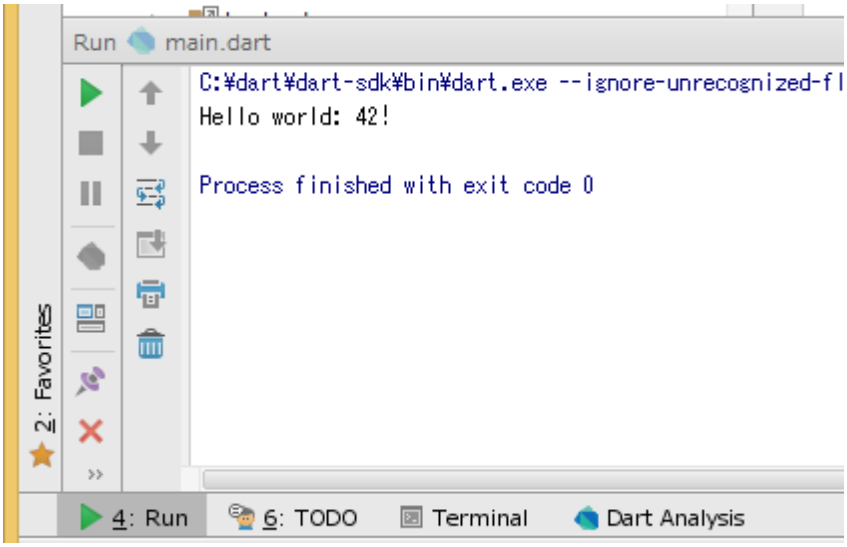
どうしてこのような構成になっているかの詳細は「パッケージ・マネージャ(Pub)」の章、特に「パッケージの詳細」の節を読む必要があるが、ここではとりあえずこのままIntelliJ CEの基本的な使用法をつかむこととする。

4. main.dartの実行は下図のようにmain.dartを選択し（ハイライトさせる）、右クリックしてRun 'main.dart'を選択する。



main.dartの実行

実行結果は下図のようにコンソールに表示される：



コンソール出力

ここではまずsdkのなかのVMであるdart.exeがC:\dart\codes\tutorial\bin\main.dartを引数にして呼び出され、その実行結果がHello world: 42!と出力されている。

```
C:\dart\dart-sdk\bin\dart.exe --ignore-unrecognized-flags --checked --package-
root=C:\dart\codes\tutorial\packages --enable-vm-service:49862 --trace_service_pause_events
C:\dart\codes\tutorial\bin\main.dart
Hello world: 42!

Process finished with exit code 0
```

第29章 本資料に含まれているプログラムのダウンロード

本資料に含まれているプログラムはGithubのリポジトリからダウンロードできる。

1. dart_code_samples
   このレポジトリは2つのフォルダで構成されている:
   - codes : 「プログラミング言語Dartの基礎」の各章にあるcode_xx.yy.dartの名前（xxは章番号、yyは節番号）で表示されたコード・サンプル
   - apps : 「プログラミング言語Dartの基礎」のなかにあるアプリケーションで、以下に記したものを除く
2. MIME type
   「パッケージ・マネージャ(Pub)」の章の「pubの概要」の節の「筆者が公開したライブラリ mime_type」の項にあるPubライブラリ
3. file_server
   「HTTPサーバ (HttpServer)」の章の「ファイル・サーバ」の節
4. GooSushi
   「HTTPサーバ (HttpServer)」の章の「セッション管理」及び「ショッピング・カートのアプリケーション・サーバ」の節
5. https_servers
   「HTTPSサーバ (HTTPS Servers)」の章の「簡単なHTTPSサーバの実験」の節
6. websocket_chat_server
   「WebSocketサーバ (WebSocket Servers)」の章の「チャット・サーバ」の節
7. file_upload_test
   「ファイル・アップロード (HTTP File Upload Servers)」の章
8. shelf_test
   「ミドルウエア・フレームワーク (shelf)」の章
9. weather_forecast_server
   「RESTfulウェブ・サービスとDart (Dart in RESTful web services)」の章の「簡単なアプリケーションの節」

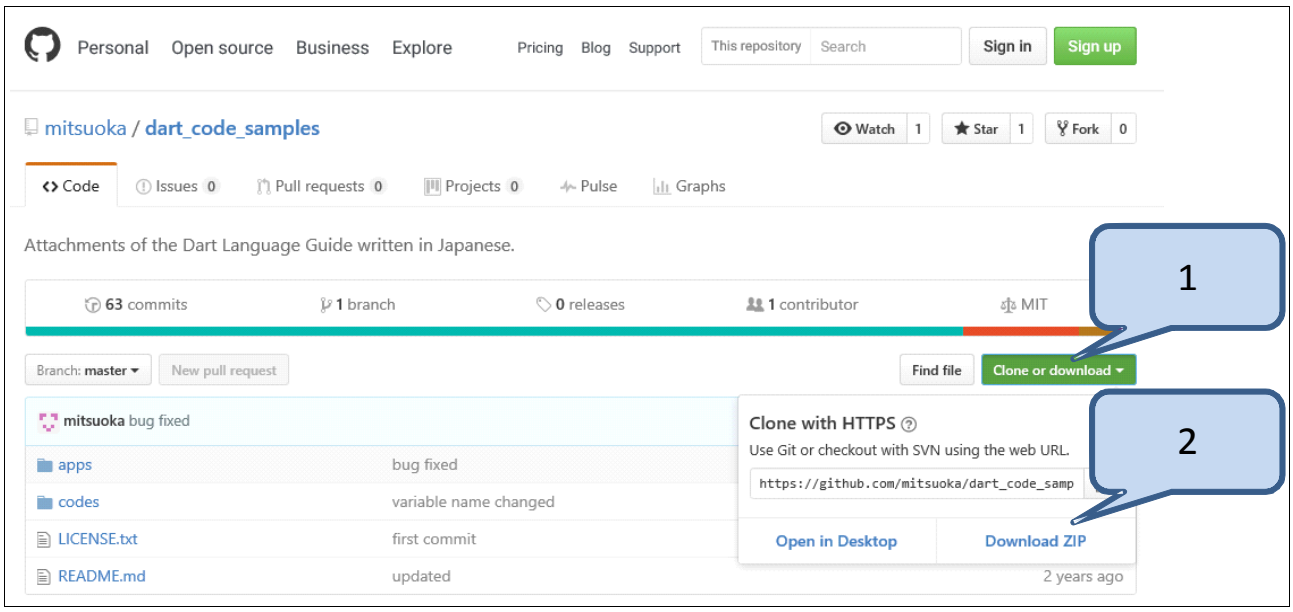
なおGitHub上のShams Zakhour氏による以下のサンプルも有用なものであり、試してみることをお勧めする:

- https://github.com/dart-lang/dart-samples
- https://github.com/dart-lang/dart-tutorials-samples

29.1節　ダウンロードの手順

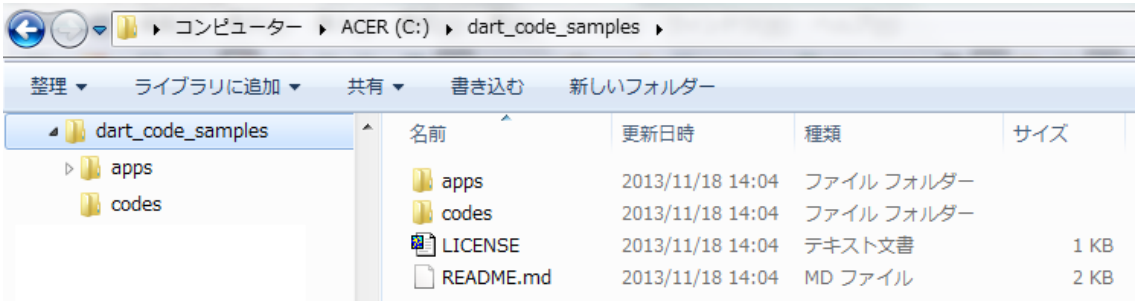
dart_code_samplesを例として、そのダウンロード手順を示す:

1. 下図のようにGithubの画面を開き、
   - まずClone or downloadのボタンをクリックし、選択ウィンドウを開く
   - 次にその窓のDownload ZIPのボタンをクリックするとZIP圧縮されたファイル（dart_code_samples-master.zip）がダウンロードされる。



Github上のdart_code_samples

2. このファイルを適当な解凍ツールを使って解凍すると下図のように展開される:
この例ではフォルダ名を dart_code_samples-masterから dart_code_samplesに変更し、c:\に配置してある。



解凍したdart_code_samples

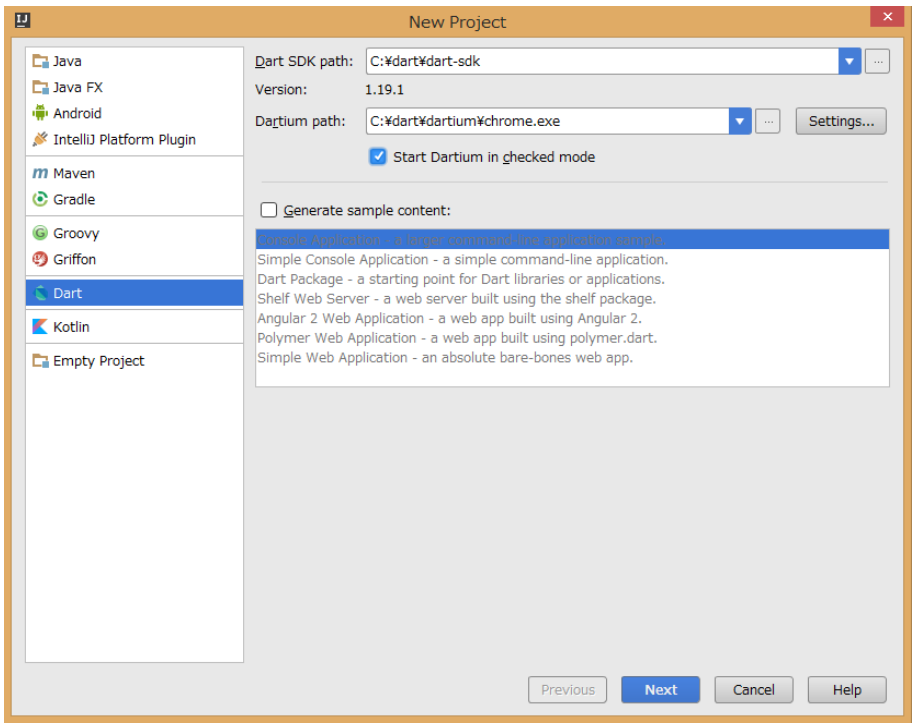
29.2節　IntelliJ CEでダウンロードしたプログラムを開く

ここでは前章で解説したDartチームが推奨しているIDEのひとつであるIntelliJ CEで、ダウンロードしたサンプルを開いて試してみる。

1. 前章に基づきIntelliJ CEのダウンロードとDart開発のための設定がすでに実施されていることを確認する

2. 上部選択バーのFile → New → Project から、Create New Projectを選択する

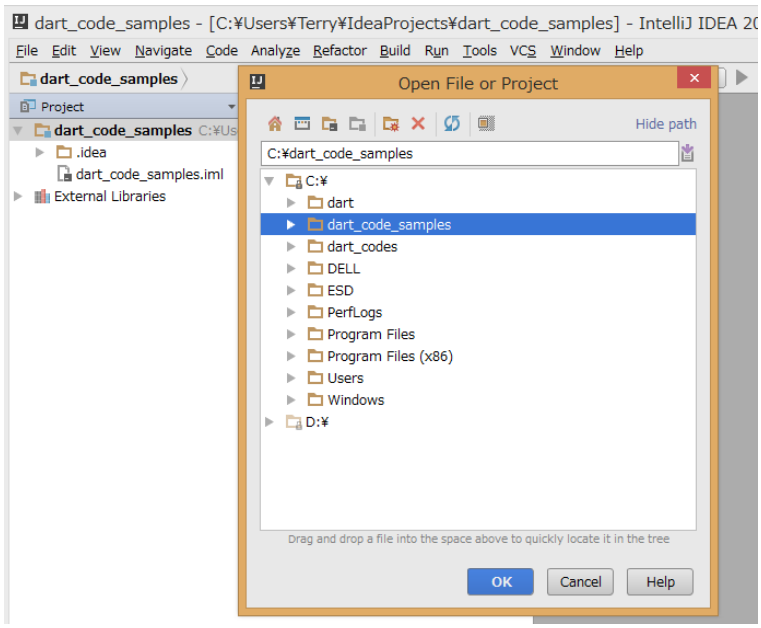
3. ここでDart SDK pathとDartium pathを指定する。またStart Dartium in checked modeにチェックを入れ、Dartiumはチェック・モードで実行させる



新規プロジェクトの作成

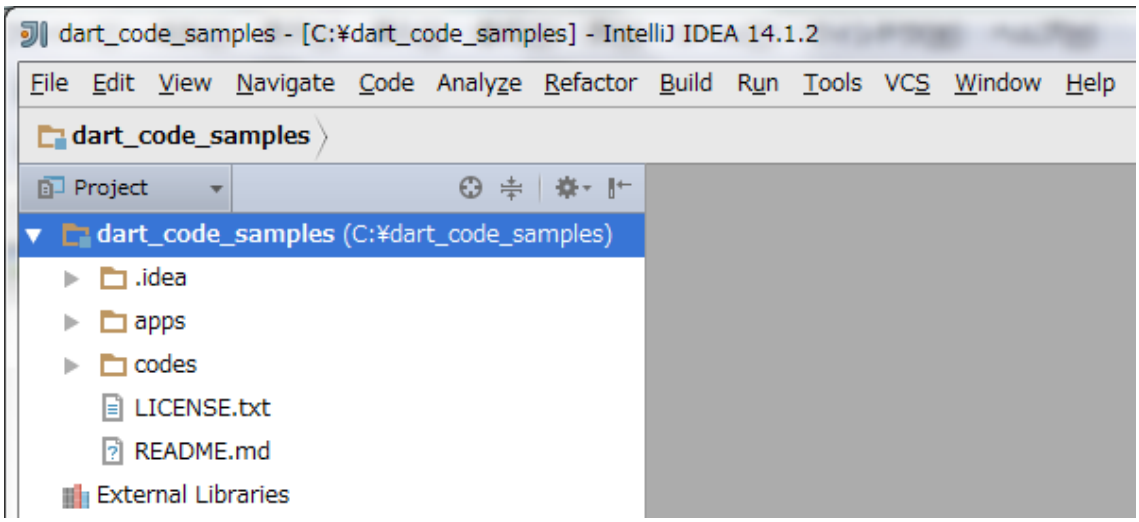
4. Nextボタンをクリックして新しく生成するプロジェクト名を記入する。ここではdart_code_samplesという名前を付ける。そうすると新しいプロジェクトが開く

5. このプロジェクトで開く必要なフォルダを開く
    - IntelliJ CEの画面の上部選択バーからOpenを選択する:



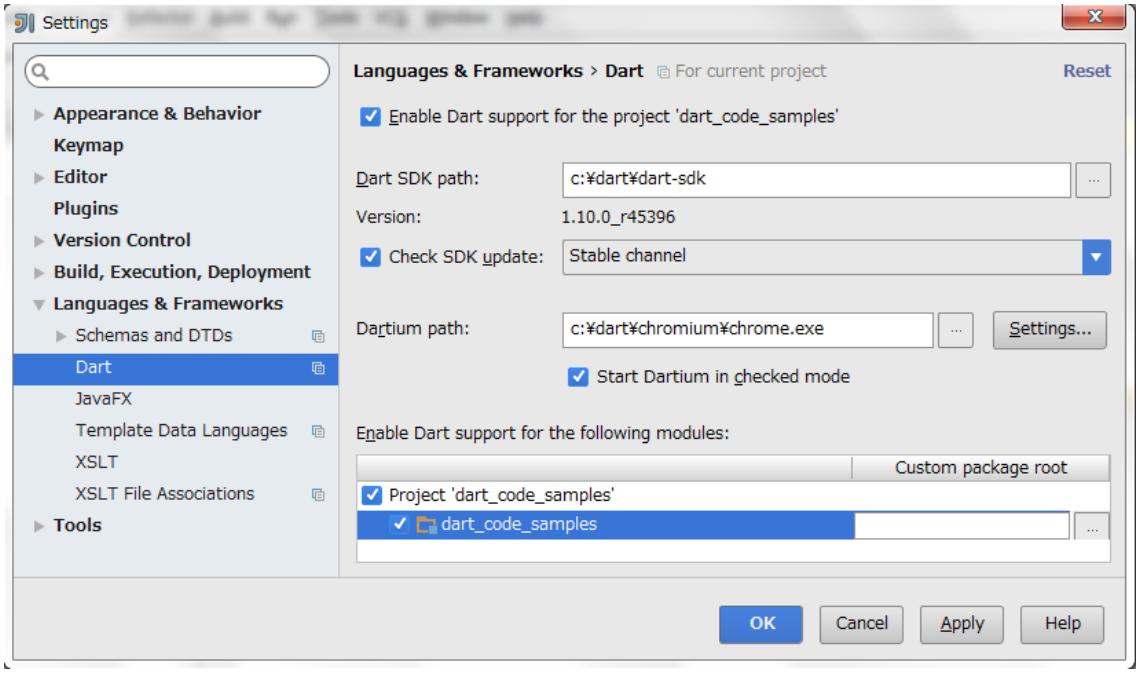
必要なフォルダを選択

- 上画面で必要なフォルダを選択してOKボタンをクリックすると下図のように展開される:



このプロジェクトにファルダをロード

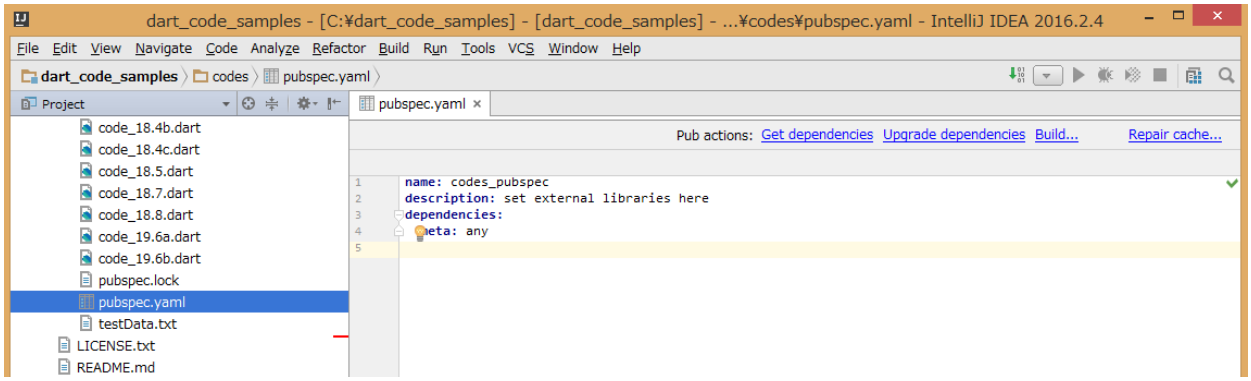
6. このプロジェクトに対しDart言語対応の設定を行う
   上図の画面でFile→Settings...で設定画面を開き、下図のようにDart設定を行う:



このプロジェクトに対するDart対応の設定

29.3節　　外部パッケージの取り込み

Dartで書かれたアプリケーションの殆どにpubspec.yamlというYAMLファイルが存在する。これはパッケージ・マネージャ(Pub)の章で説明しあるように、そのアプリケーションで使用する外部リソースを記述したファイルである。IntelliJなどのIDEに対しては、それらの外部リソースを取り込むことを指示する必要がある。IntelliJではそのようなファイルを選択すると必要なPubアクションを提示してくれる。



Pubspec.yamlファイルに対するアクション

例えばdart_code_samples/apps、およびdart_code_samples/codesにもpubspec.yamlが存在する。上図はdart_code_samplesのフォルダの中にあるpubspec.yamlを選択した例である。Pub actions:で表示されているアクションの"Get dependencies"をクリックすると必要なパッケージが取り込まれる(プロジェクト・ビュー上でこのファイルを右クリックしてもよい)。取り込みが成功すれば次のようなメッセージがコンソールに表示される:

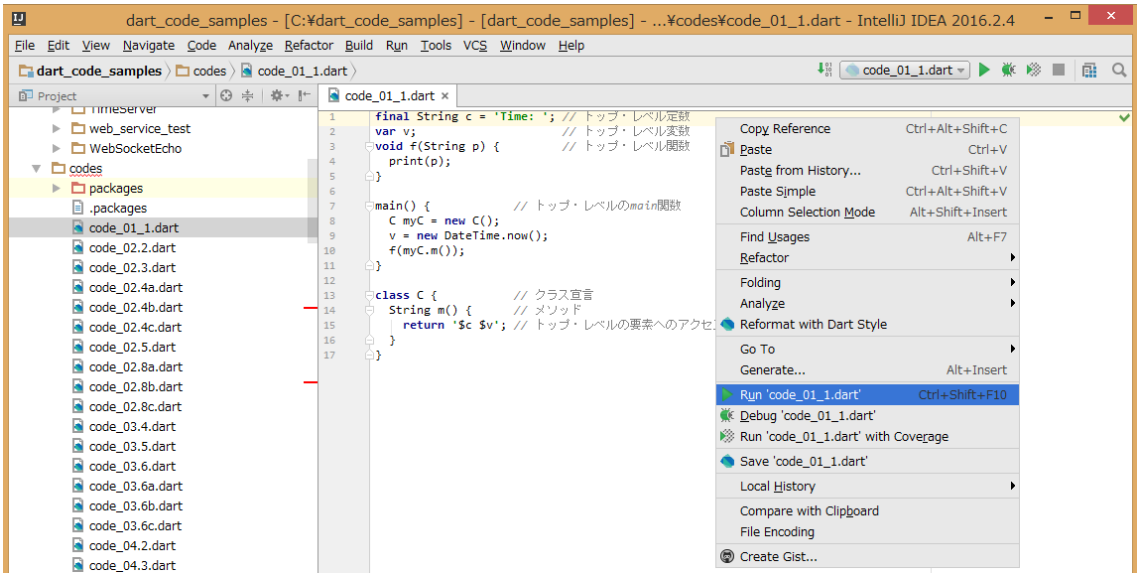
```
Working dir: C:\dart_code_samples\codes
C:\dart\dart-sdk\bin\pub.bat get
Resolving dependencies...
Got dependencies!
Process finished with exit code 0
```

そうすることで.packagesというフォルダ、pubspec.lockおよびその他のファイルも生成される。

29.4節　　IntelliJ CEでコマンド行プログラムのサンプルを試す

コマンド行のサンプルはcodesというフォルダに収容されている。このフォルダにもpubspec.yamlファイルが存在しているので、最初に前節に従い外部ライブラリを取り組む必要がある。

最初のcode_01.1.dart(第1章の最初のサンプル・コード)はこれをダブル・クリックしてコード表示域に表示されるので、これを右クリックするとこのコードに対する処理の一覧が下図のように表示される:



コマンド行コードの実行

ここでRun 'code_01_1.dart'を選択すれば、このコードが実行され、その結果が左下のコンソール域に表示される。

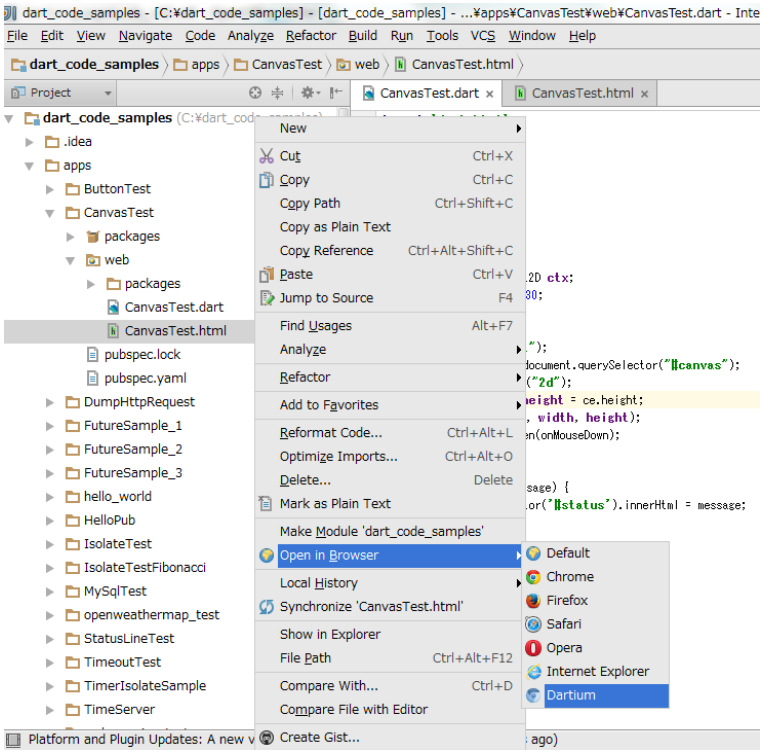
```
C:\dart\dart-sdk\bin\dart.exe --ignore-unrecognized-flags --checked --enable-vm-
service:50358 --trace_service_pause_events C:\dart_code_samples\codes\code_01_1.dart
Observatory listening on http://127.0.0.1:50358
Time:  2015-05-25 13:15:50.321

Process finished with exit code 0
```

29.5節　　IntelliJ CEでウェブ・アプリケーションのサンプルを試す

ウェブ・アプリケーションのサンプルはappsというフォルダに収容されている。例えばCanvasTestはHTML5のCanvasを使った簡単なサンプルである。

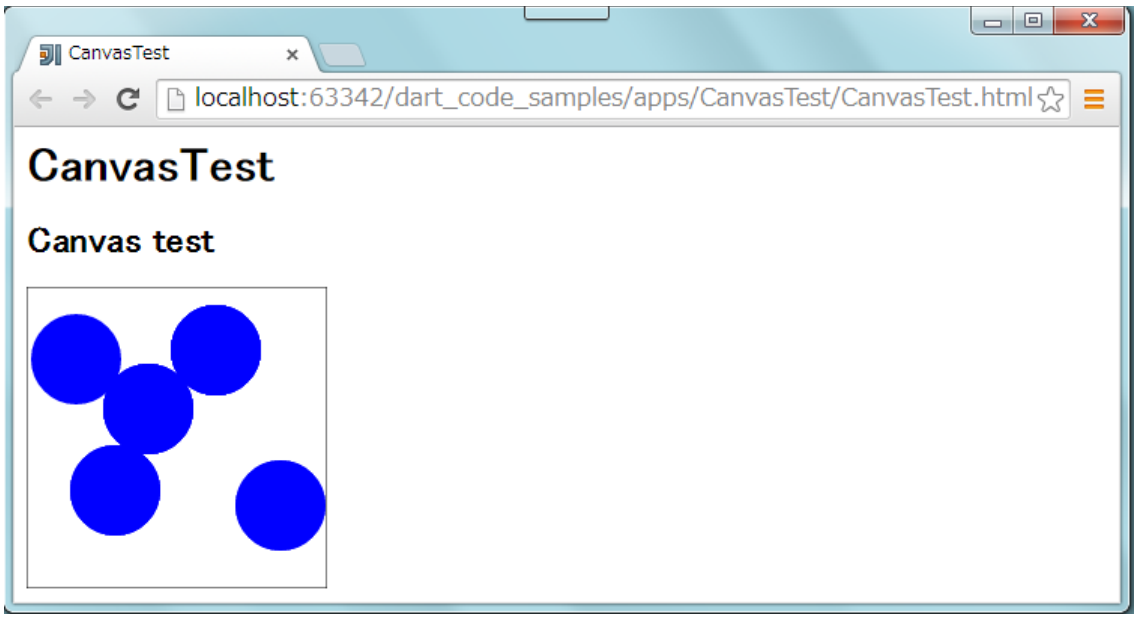
このフォルダにもpubspec.yamlファイルが存在しているので、最初に前節に従い外部ライブラリを取り組む必要がある。

webディレクトリ内のCanvasTest.dartとCanvasTest.htmlの2つのファイルからなる。これらをダブル・クリックするとコード表示域に表示される。これら右クリックするとこのコードに対する処理の一覧が下図のように表示される。



Dartiumで実行させる

Open in Browser→Dartiumを選択すれば、下図のようにDartiumでこれらのコードが実行される。キャンバス内の任意の場所をクリックするとその点を中心にした円が描かれる。



CanvasTestの実行

Dart2JSを使って普通のブラウザで実行させる場合は、Chromeなどのブラウザを選択すれば良い。その場合は自動的に最初にDart2JSでJSコードにクロスコンパイルされ、それが当該ブラウザから実行される。その為マシンによってはブラウザに画面が表示されるまで最初は多少時間がかかる。Dartiumとその他のブラウザとのきりわけの詳細は「dart.jsブートストラップ・コード」の項を参照のこと。この場合は実行されるHTMLソース・コード（Chromeではその他のツール→デベロッパ　ツール、IEではF12 開発者ツール→DOM Explorerで調べることができる）は次のようになっている：

```
<html><head>
    <title>CanvasTest</title>
  </head>
  <body>
    <h1>CanvasTest</h1>
    <h2 id="status">Canvas test</h2>
    <canvas width="200" height="200" id="canvas"></canvas>
    <script
src="http://localhost:63342/dart_code_samples/apps/CanvasTest/web/CanvasTest.dart.js"></scr
ipt>
    <script src="packages/browser/dart.js"></script>

</body></html>
```

dart_code_samples/apps/CanvasTest/web/CanvasTest.dart.jsはDart2JSがクロスコンパイルしたファイルである。

Githubに登録されているそれ以外のサンプルの使い方に関しては、その都度本文の然るべき箇所に記されているので、そちらを見て頂きたい。