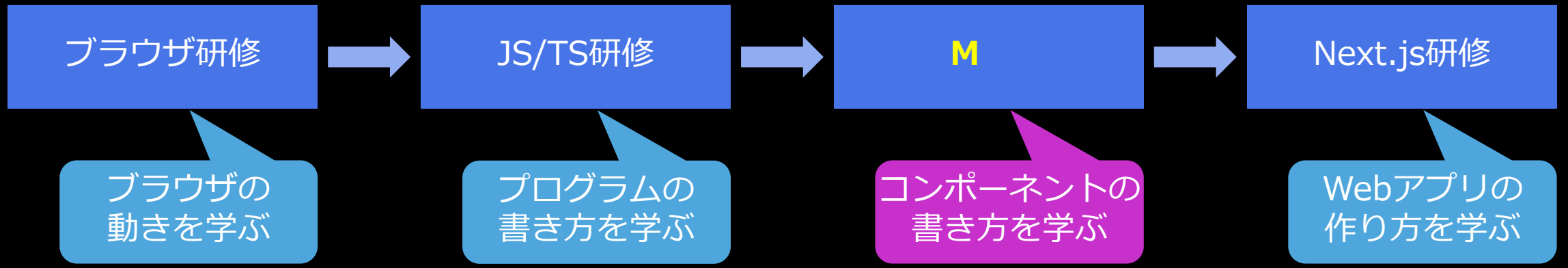
React 研修

(2024)

@koichik

React研修の位置付け



React研修の目的

- Reactの「考え方」を知ってほしい
  - 特に「宣言的UI」の考え方
  - 主に状態や副作用を扱う基本的なAPI (組込Hooks) について
    使い方だけではなく「こう書くとなぜこう動くか」を理解してほしい
- やらないこと
  - 本格的なハンズオン
  - React 18以降の新機能
    - 並行レンダリング, ストリーミングSSR, React Server Components, etc.
  - 3rd Partyのライブラリやフレームワーク、ツール等を使う機能
    - CSS, ルーティング, データフェッチ, 自動テスト, Storybook, etc.

これらの一部は
Next.js研修で扱います

Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外のリソースとの同期
- メモ化とパフォーマンス

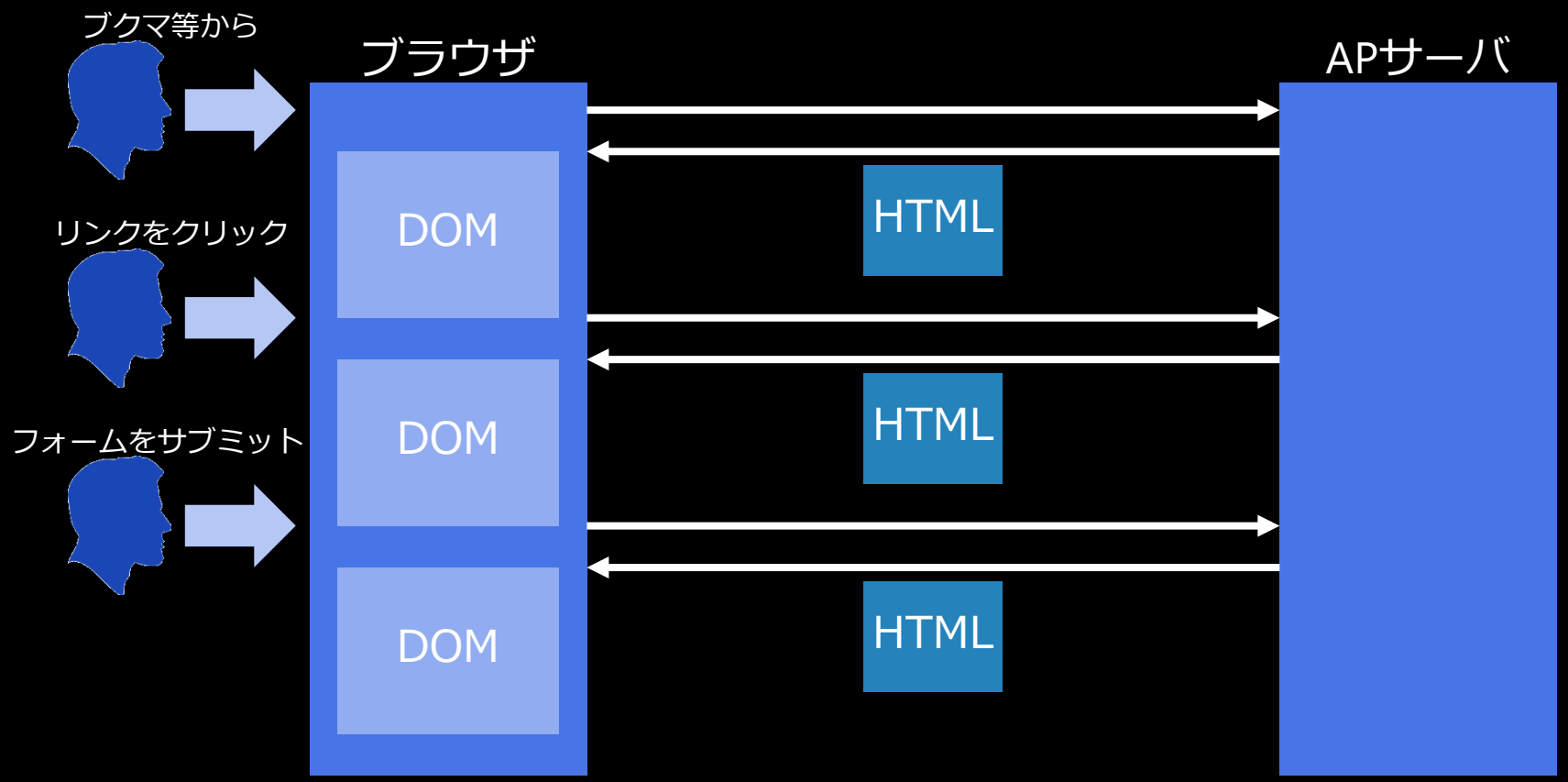
Webアプリ開発の変遷

- 90年代末～
  - MPA (クラシックSSRのみ)
- 00年代後半～
  - MPA (クラシックSSR + jQuery)
- 10年代初め～
  - SPA (CSRのみ)
- 10年代後半～
  - SPA (CSR + 事前レンダリング)

MPA (クラシックSSRのみ)

- Multiple-Page Application
  - リンクをクリックするたび、またはフォームをサブミットするたびに異なるHTMLページを表示するWebアプリケーション
  - 後述するSPAの登場以降にMPAと呼ばれるようになった

MPA (Multiple-Page Application)

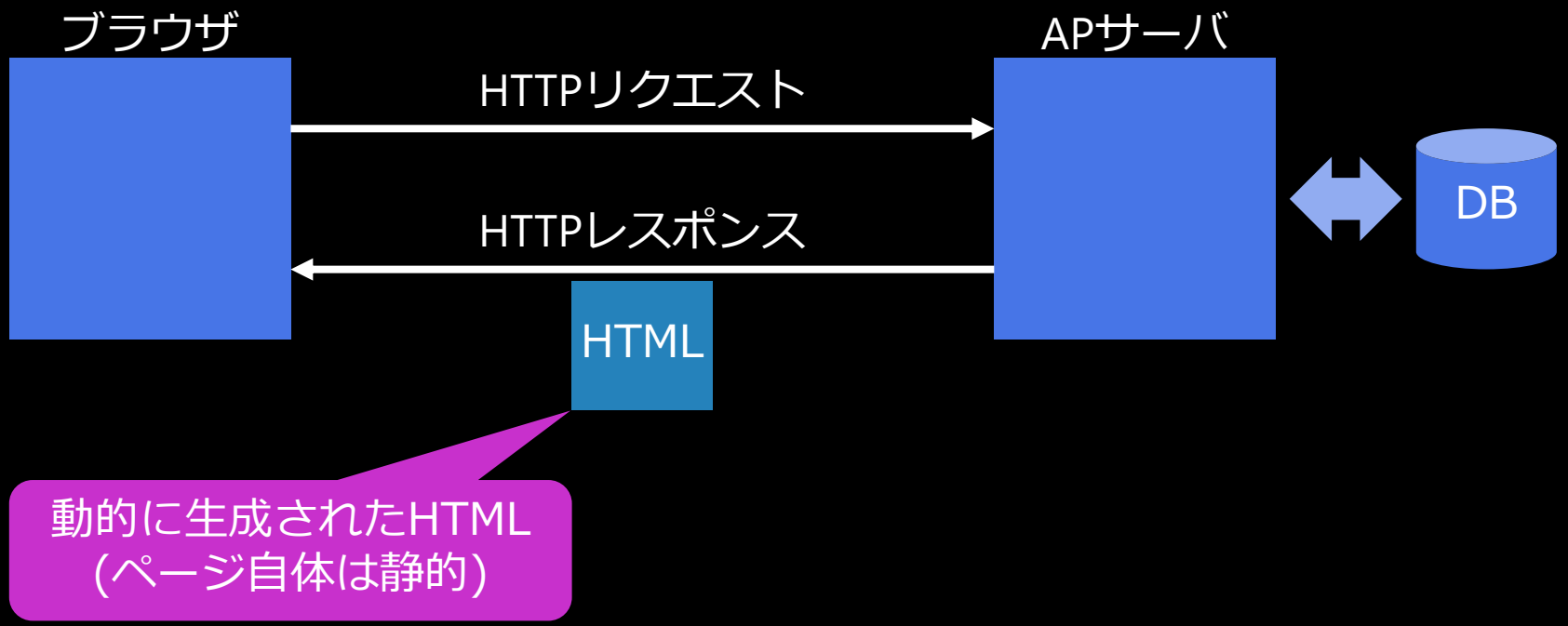


HTMLをロードするたびに
DOMツリーが構築される

MPA (クラシックSSRのみ)

- Server-Side Rendering
  - リンクをクリックするたび、またはフォームをサブミットするたびにサーバサイドでHTMLを動的にレンダリングする
    - 後述するSPAを事前レンダリングするSSR (SSR with Hydration) と区別するため本研修ではMPAのためのSSRを「クラシックSSR」と呼ぶ
- 生成されるWebページ自体は静的だった
  - 2000年代始め頃までJSはあまり活用されていなかった

SSR (Server-Side Rendering)

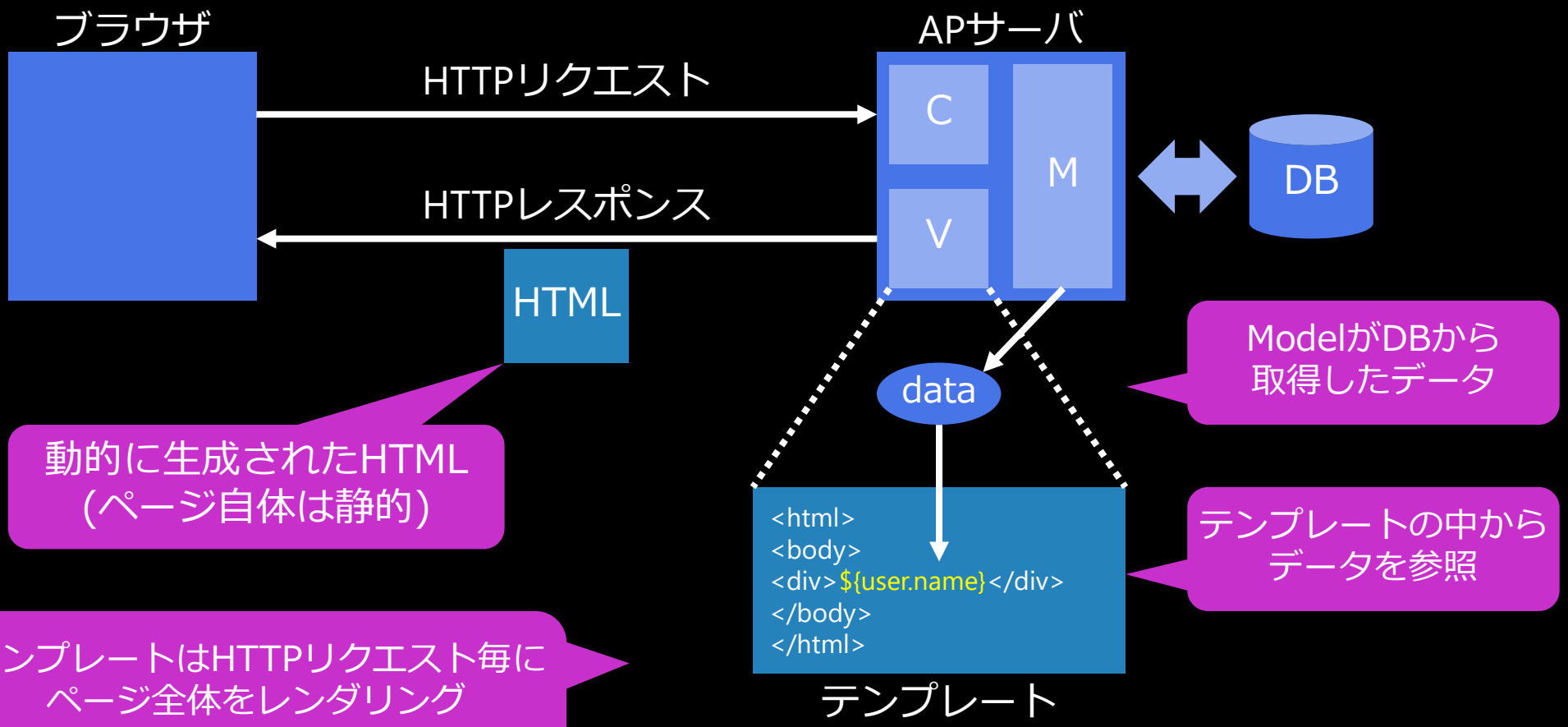


Web MVCフレームワーク

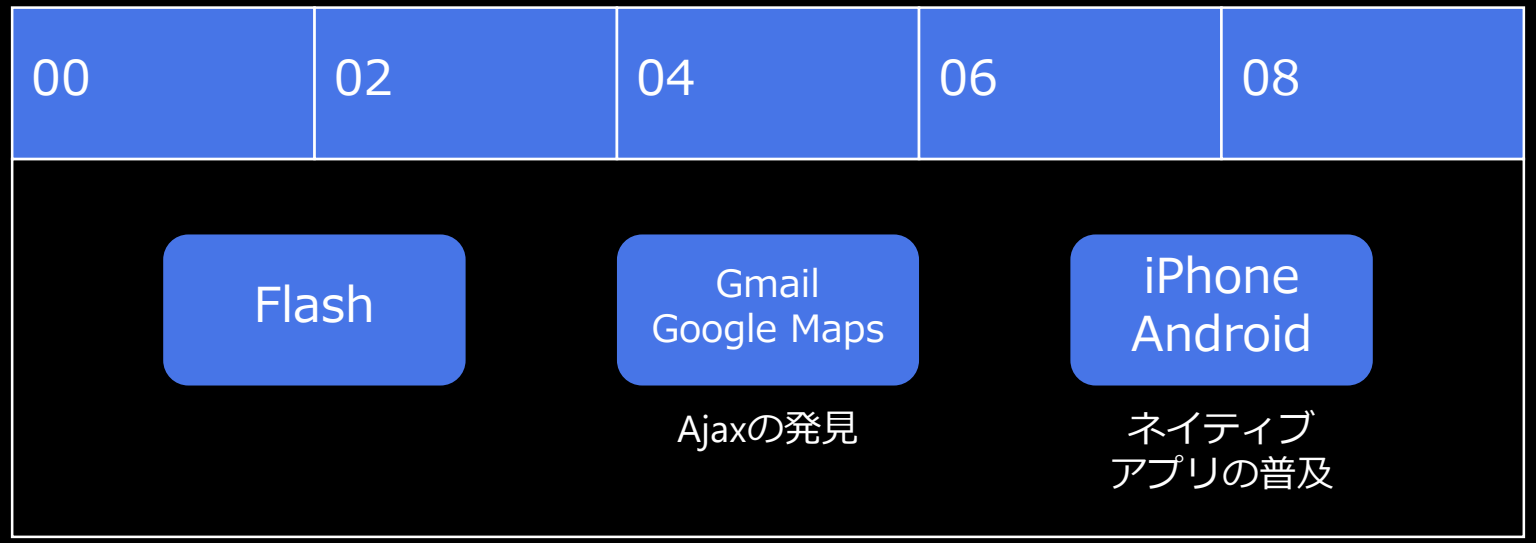
- サーバサイドでは主にWeb MVCフレームワークが使われた
  - Model-View-Controller
  - 例: Ruby on Rails, Struts, Spring-Framework, etc.
- Viewには「テンプレート」が使われた
  - ひな形となるHTMLに「式」を埋め込めるもの
  - 例: ERB, JSP, Thymeleaf, etc.
  - HTTPリクエスト毎にページ全体をレンダリングして返す

現在でも
広く使われています

Web MVCフレームワーク



2000年代のWebの変化



ブラウザのJSを無効にする人が多かった

インタラクティブなコンテンツの普及

Webでもマイクロインタラクション重要！

Webアプリ開発の変遷

- 90年代末～
  - MPA (クラシックSSRのみ)
- 00年代後半～
  - MPA (クラシックSSR + jQuery)
- 10年代初め～
  - SPA (CSRのみ)
- 10年代後半～
  - SPA (CSR + 事前レンダリング)

MPA (クラシックSSR + jQuery)

- クラシックSSRが生成したHTMLと連携するJSを「後付け」
  - 既存システム (クラシックSSR) に導入しやすかった
  - コンテンツ (HTML), スタイル (CSS), ロジック (JS) を分離することがよいプラクティスだと考えられていた (過去形)
- 2000年代のブラウザ環境
  - IE6 (2001～) やIE7 (2006～) が主流
  - JSもDOMも機能不足でブラウザ間の互換性も低かった
- ブラウザの差異を吸収する高機能なライブラリが必要だった
  - 例: Prototype.js, Mootools, Dojo, YUI, etc.
- jQueryが広く普及した

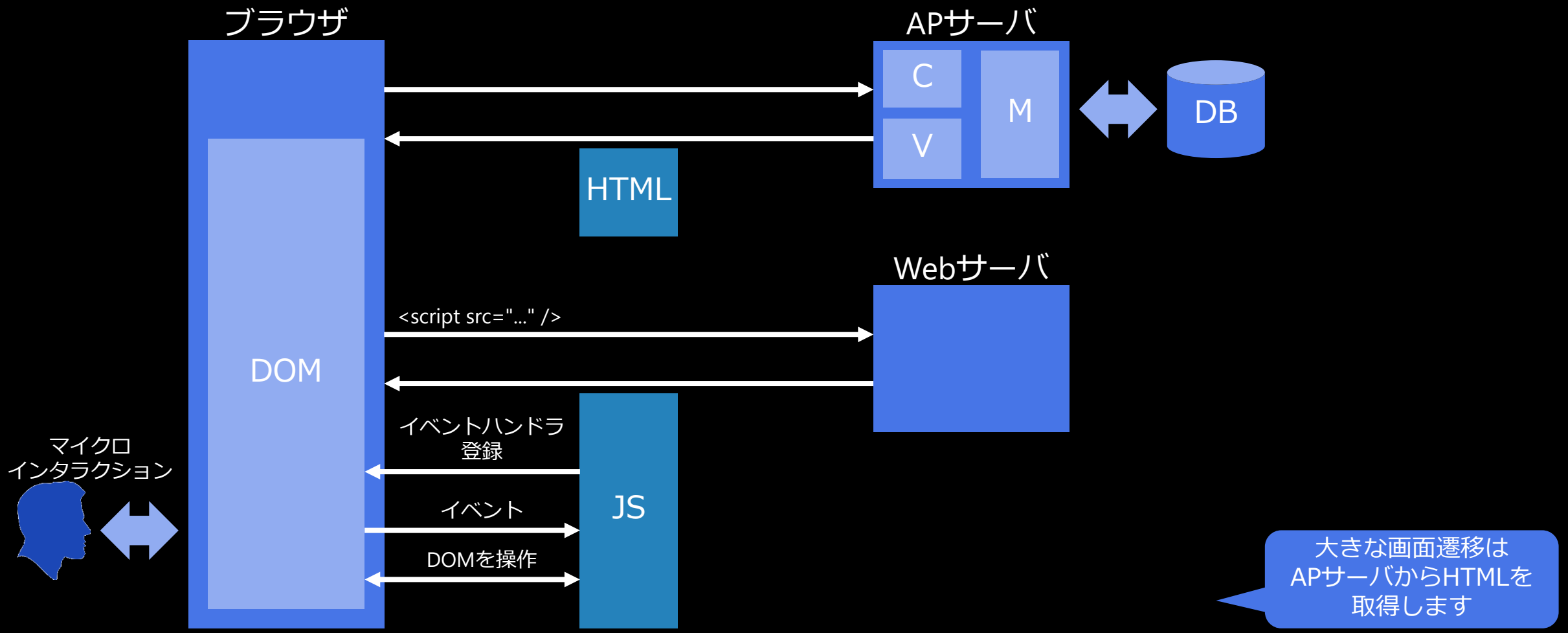
現在はjQueryを使う必要性は少なくなりましたが
クラシックSSR + JSはまだ広く使われています

jQuery

- セレクタをサポートしたメソッドチェーンによるAPI
  - HTMLにイベントハンドラを後付けするのに適していた
  - イベントハンドラからDOMを操作しやすかった

```
// DOMContentLoadedのイベントハンドラを登録
$(function() {
  // クラス属性"foo"を持つ<button />要素が押された場合のイベントハンドラ
  $("button.foo").on("click", function() {
    // クラス属性fooを持つ<p />要素を非表示にする
    $("p.foo").hide();
  });
});
```

MPA (クラシックSSR + jQuery)



MPA(クラシックSSR + jQuery)の課題: アプリケーションの構造

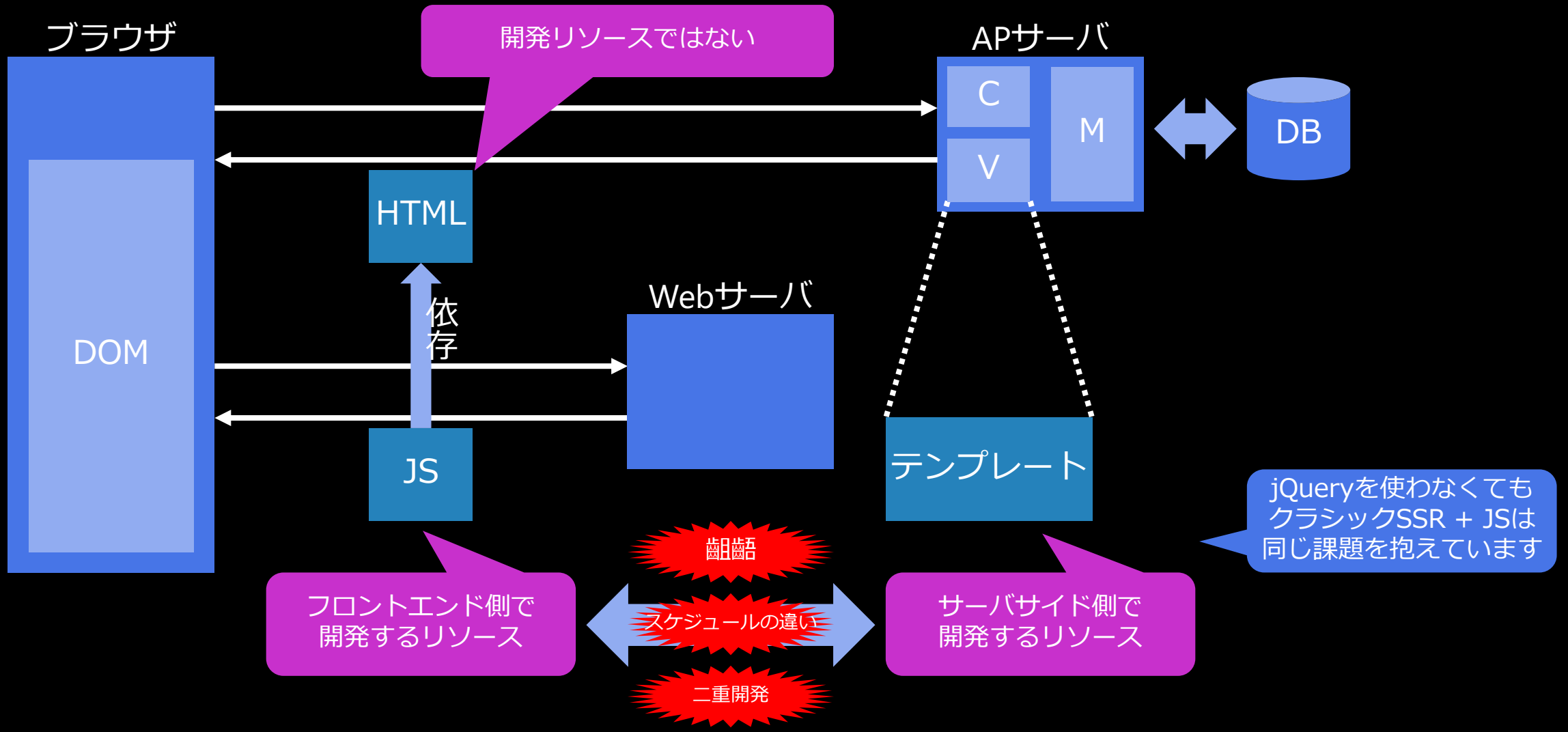
- 命令的なイベントハンドラ
  - DOMを参照して
  - 処理を行い
  - DOMを更新する
- 大量のイベントハンドラが散らばる
  - DOMを更新する処理も散らばる
- イベントハンドラ間に不明瞭な依存関係が生じる
  - あるイベントハンドラが動作するために前提となるDOM構造は どのイベントハンドラによって構築されるのか？破壊されるのか？
- DOMが巨大で暗黙的なグローバル変数になってしまう

このような構造は「Sprinkle」と呼ばれることがあります

MPA(クラシックSSR + jQuery)の課題: ワークフロー

- 「HTML」はマスタとなるリソースではない
  - Gitでバージョン管理されるリソースは「テンプレート」
    - 例: ERB, JSP, Thymeleaf, etc.
  - JSの処理対象となるHTMLはAPサーバの実行結果
- JS側が必要とするHTMLの修正を誰がどのように行うか？
  - テンプレートの修正が必要
    - 通常はサーバサイドの開発者が担当するリソース
    - フロントエンド側とは開発サイクルも開発のワークフローも異なることが多い
- 二重開発
  - サーバサイドのテンプレートとフロントエンドのJSが機能的に重複

MPA(クラシックSSR + jQuery)の課題: ワークフロー



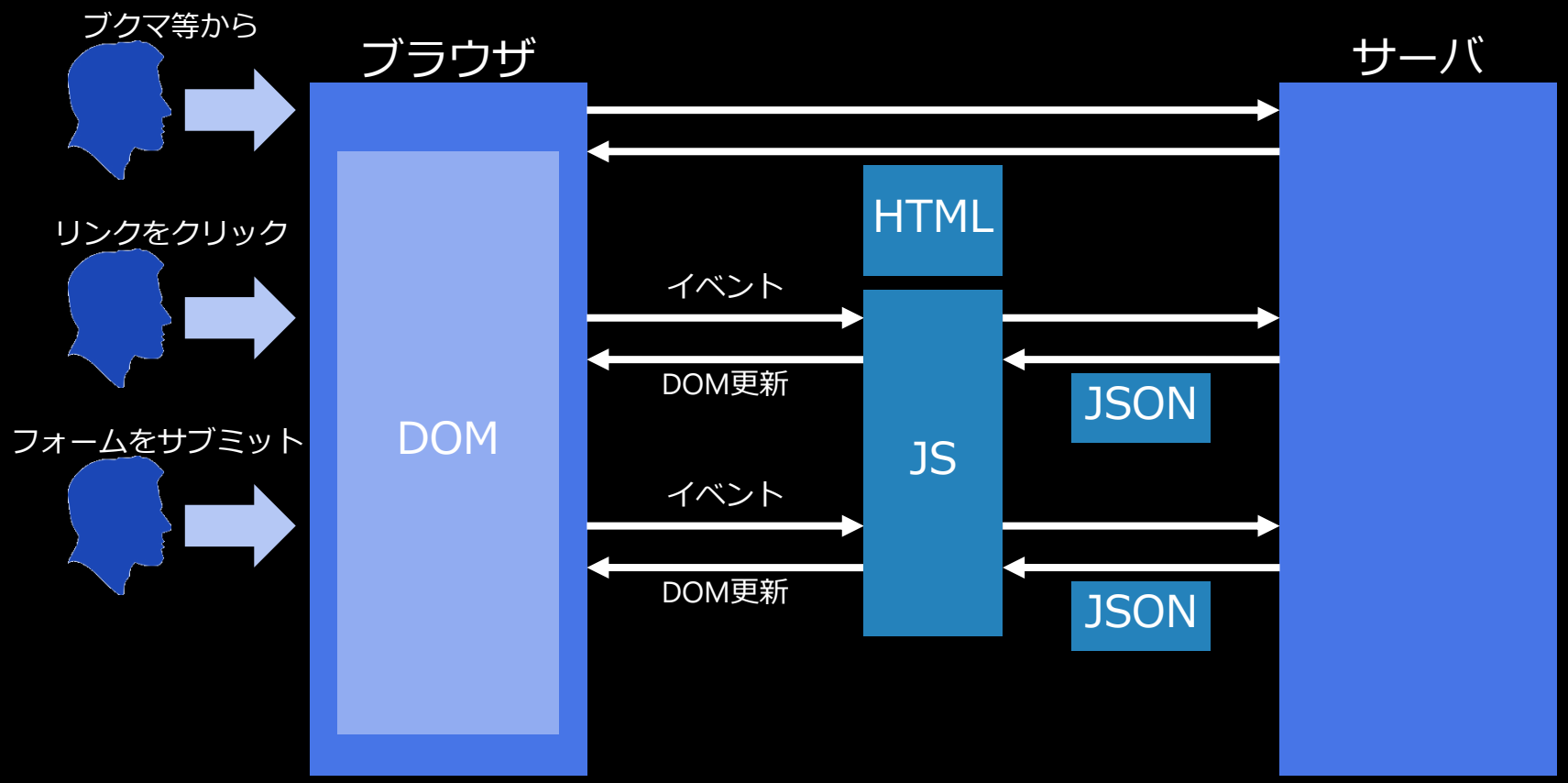
Webアプリ開発の変遷

- 90年代末～
  - MAP (クラシックSSRのみ)
- 00年代後半～
  - MPA (クラシックSSR + jQuery)
- 10年代初め～
  - SPA (CSRのみ)
- 10年代後半～
  - SPA (CSR + 事前レンダリング)

SPA (CSRのみ)

- Single Page Application
  - ナビゲーションによる画面遷移もブラウザ上のJSでレンダリング
    - 単一のHTMLページ (あるいはDocument) だけで構成されるためSPA
  - ナビゲーションの単位も「ページ」と呼ぶので紛らわしい
    - 例: トップページ、一覧ページ、商品ページ、etc.

SPA (Single-Page Application)

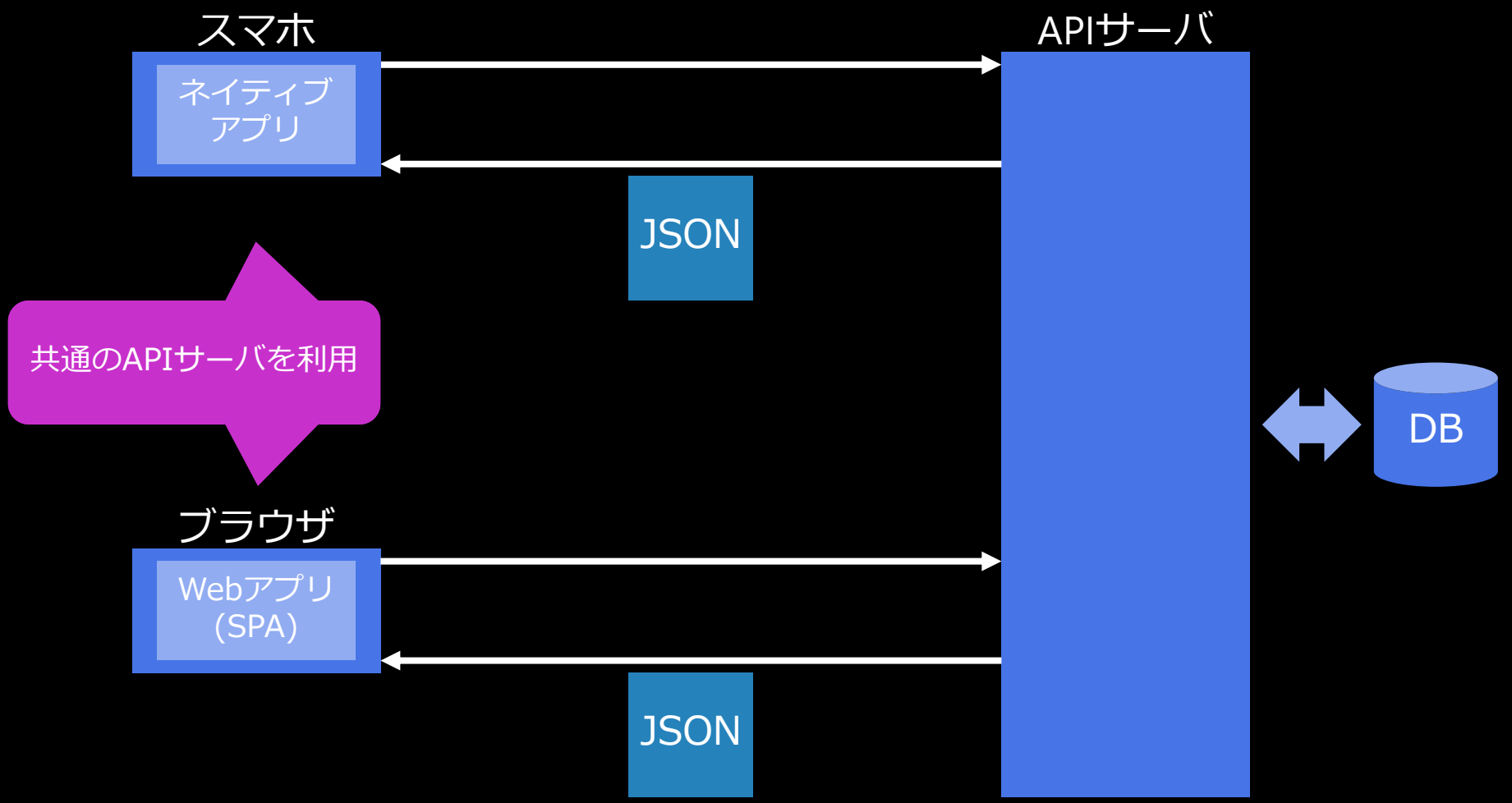


ロードされるHTMLページは一つだけ
(Documentオブジェクトは不変)

SPA (CSRのみ)

- SPA普及の背景
  - スマホ向けネイティブアプリの普及
    - サーバサイドがWeb API化
    - WebアプリもWeb APIを共通で使いたい

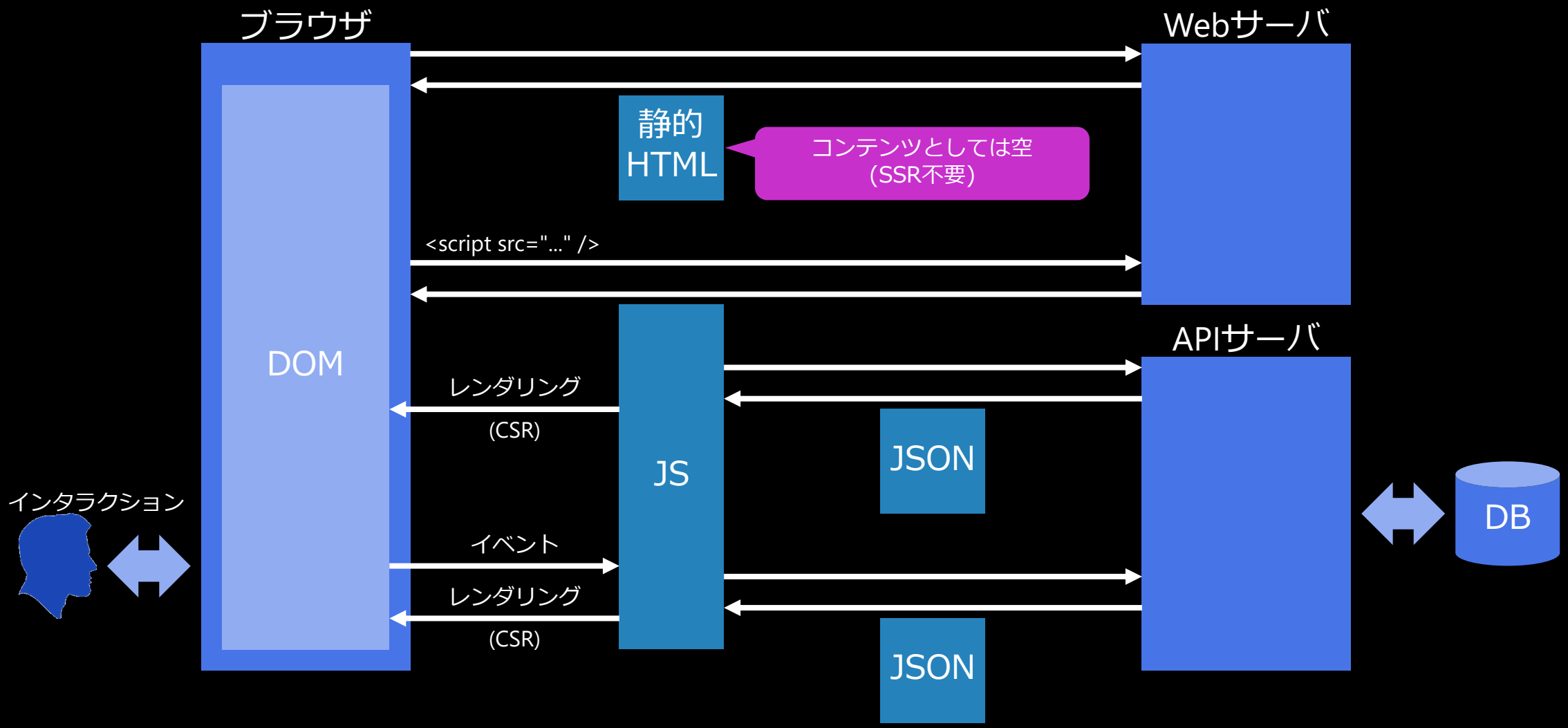
SPAとスマホネイティブアプリ



SPA (CSRのみ)

- Client-Side Renderingのみ
  - クライアントサイド (ブラウザ) だけでコンテンツをレンダリング
  - サーバサイドではHTMLページを動的にレンダリングしない

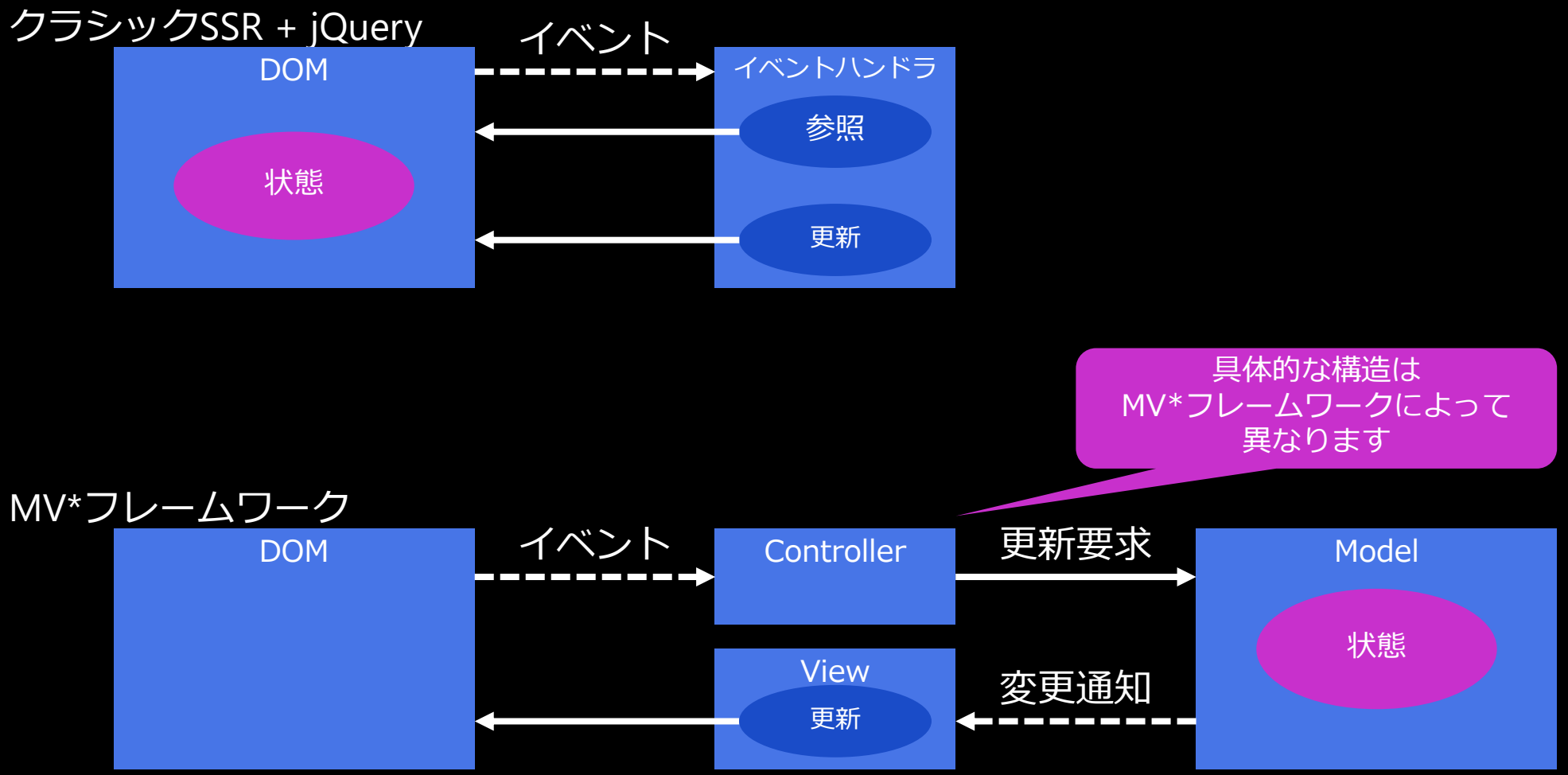
SPA (CSRのみ) の起動シーケンス



SPA向けフレームワーク

- MV*フレームワークの登場
  - MVC, MVP, MVVM, MVAC, etc. の総称 (*はワイルドカード)
  - 例: Backbone.js, AngularJS, Ember.js, Knockout.js, etc.
  - 当初はBackbone.js、後にAngularJSが主流になりそうだった
  - 10年代半ば以降はMV*フレームワークではないReact(後述)と(特に日本では) VueJSが主流になった
- MPA (クラシックSSR + jQuery) の課題を解決
  - フロントエンドのアプリケーションに構造がもたらされた
    - Model, View, Controller, Presenter, View Model, etc.
    - DOMのグローバル変数化および「Sprinkle」からの脱却
  - サーバサイドのテンプレートが不要になった
    - フロントエンドのリソース (HTML, CSS, JS) はフロントエンドで完結

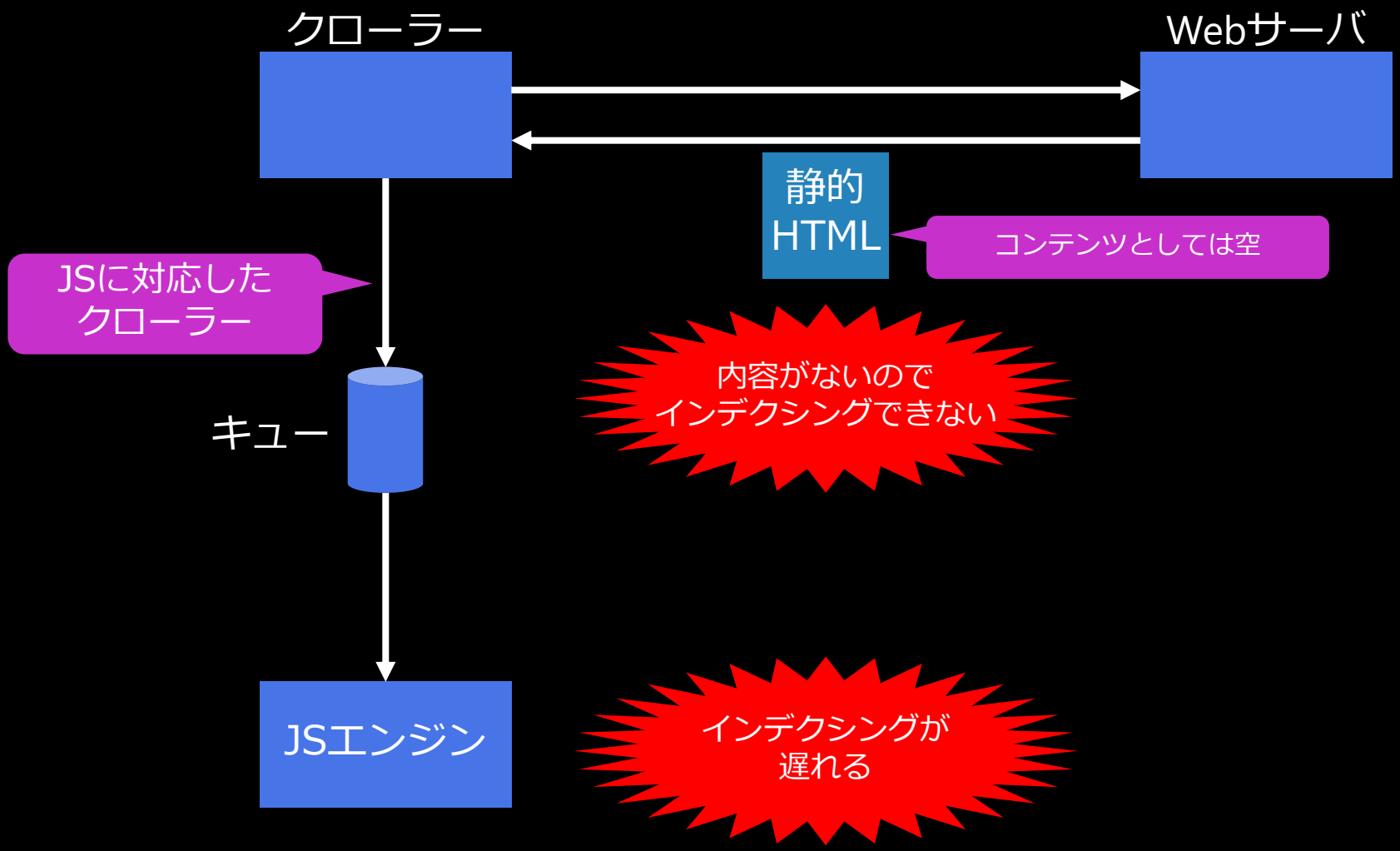
MV*フレームワークと状態



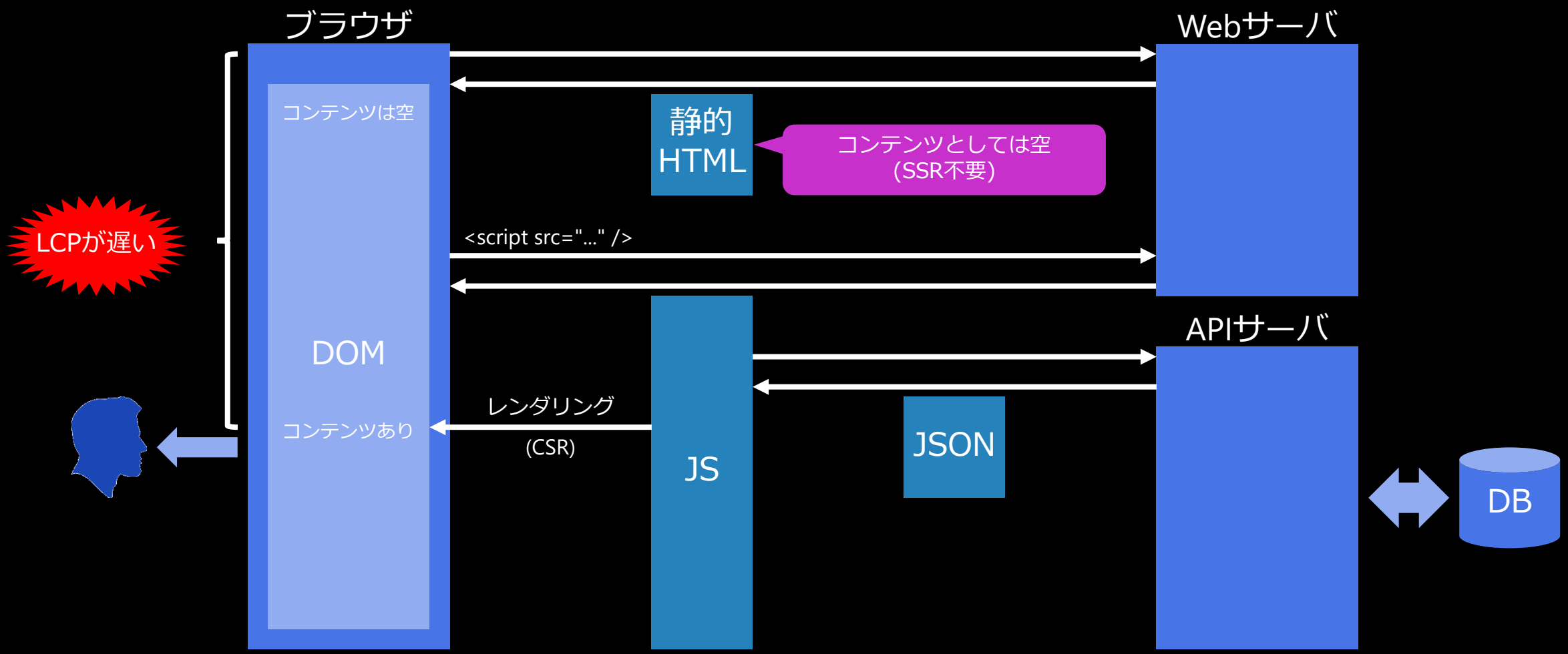
SPA (CSRのみ) の課題

- MPA (クラシックSSR) で構築されたサイトからの移行が困難
  - 現在もMPA (クラシックSSR + jQuery) が健在な理由
- SEO対策が困難 (10年代半ばのクローラーはJS非サポート)
  - 現在はGoogleなどJSをサポートしたクローラーもある
    - しかしインデクシングされるまでに時間がかかることもあり現在でも不利
  - SEOが不要なサービスでは課題にならない
- OGP対応が困難
- 初期表示 (LCP/FMP) が遅い
  - 最初に読み込まれるHTMLがコンテンツを含まず、JSが実行されてからコンテンツが表示されるため
    - Largest Contentful Paint
    - First Meaningful Paint

SPA (CSRのみ) とクローラー



SPA (CSRのみ) の初期表示



初期のMV*フレームワークの課題

- Backbone.jsの課題
  - Viewのサポートが手薄
    - 結局jQueryと組み合わせて使うことが多かった
      - 初期表示はテンプレート (Handlebars等) を使い、更新はjQuery (命令的) を使う等
- AngularJSの課題
  - Dependency Injection (DI) 等を含む多機能なフレームワークのため初期の学習コストが高いと見られやすかった
  - 複雑な画面になると表示パフォーマンスが劣化した

Webアプリ開発の変遷

- 90年代末～
  - MPA (クラシックSSR)
- 00年代後半～
  - MPA (クラシックSSR + jQuery)
- 10年代初め～
  - SPA (CSRのみ)
- 10年代後半～
  - SPA (CSR + 事前レンダリング)

SPA (CSR + 事前レンダリング)

「事前」はブラウザに読み込まれるよりも前という意味です

- 最初に配信されるHTMLにコンテンツを事前レンダリングする
- 事前レンダリングにはCSRと同じライブラリ/コードを使う
  - 例: React, VueJS, etc.
  - 事前レンダリングではDOM操作の代わりにHTML文字列を生成
  - Isomorphic JSまたはUniversal JSとも呼ばれた
- 事前レンダリングの実行にはサーバサイドのJSランタイム (主にNode.js) が使われる
- 初期表示の後はSPA (CSRのみ) と同様にCSRで画面を更新
  - 事前レンダリングはSPAの初期表示のための最適化
- 事前レンダリングの課題
  - INP (Interaction to Next Paint) が遅い

React 18のStreaming SSRや React Server Components (RSC) で解決すると期待されますが本研修では扱いません

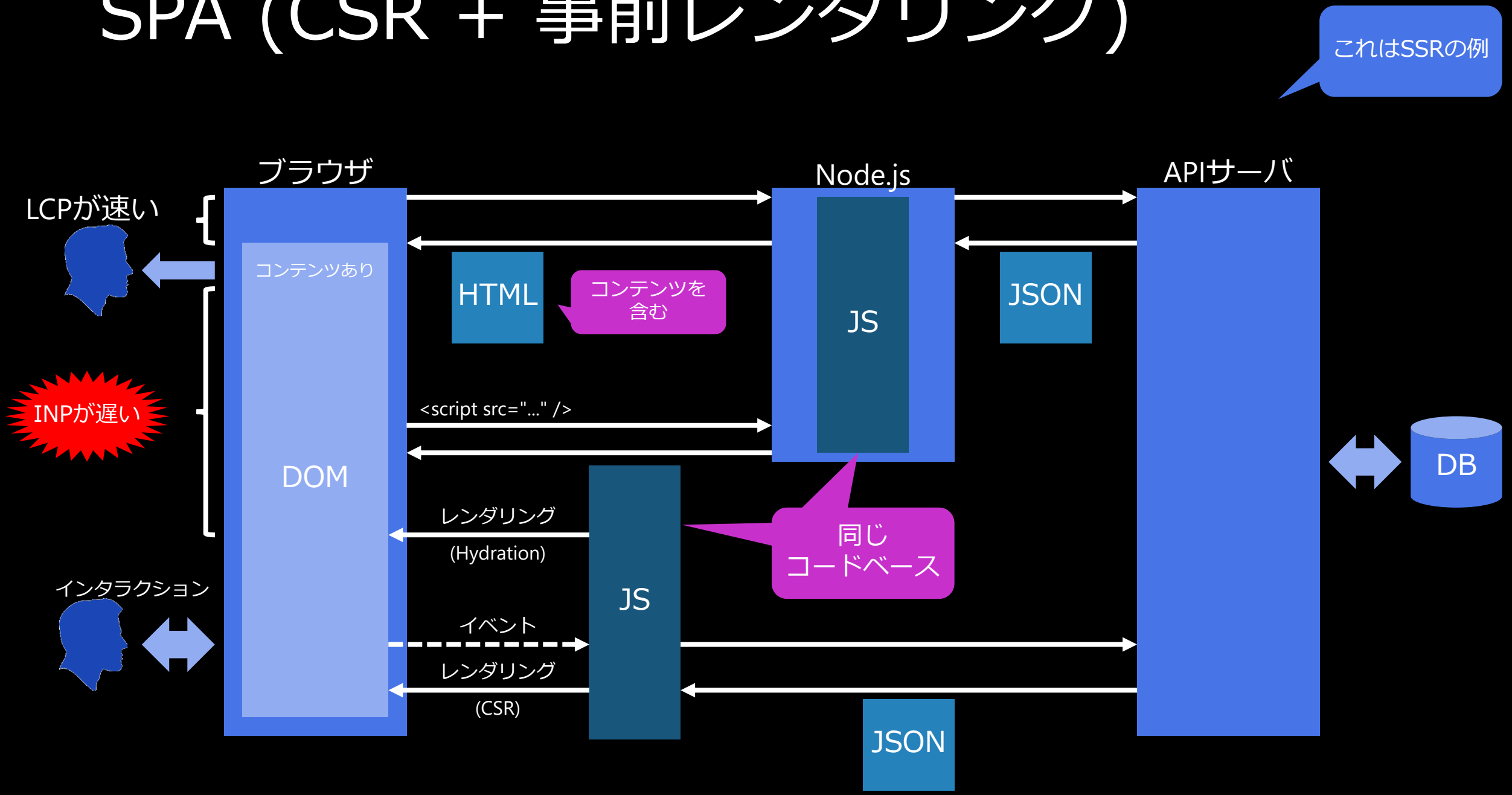
事前レンダリングの種類

- ページ単位で事前レンダリングの方式を選択できる
  - (メタ) フレームワーク (後述) によって異なる
- SSR (Server-Side Rendering)
  - HTTPリクエスト時にオンデマンドで事前レンダリング
  - ランタイムのサーバ (Node.js等) が必須
- SG (Static Generation)
  - ビルド時に事前レンダリング
  - ランタイムのサーバ (Node.js等) を不要にできる
- ISR (Incremental Static Regeneration)
  - SGとSSRの組み合わせ (事前ビルド + オンデマンド)
  - ランタイムのサーバ (Node.js等) が必須

MPAのクラシックSSRと区別する場合は SSR with Hydration とも呼ばれます

Static Site Generation (SSG)とも呼ばれます

SPA (CSR + 事前レンダリング)

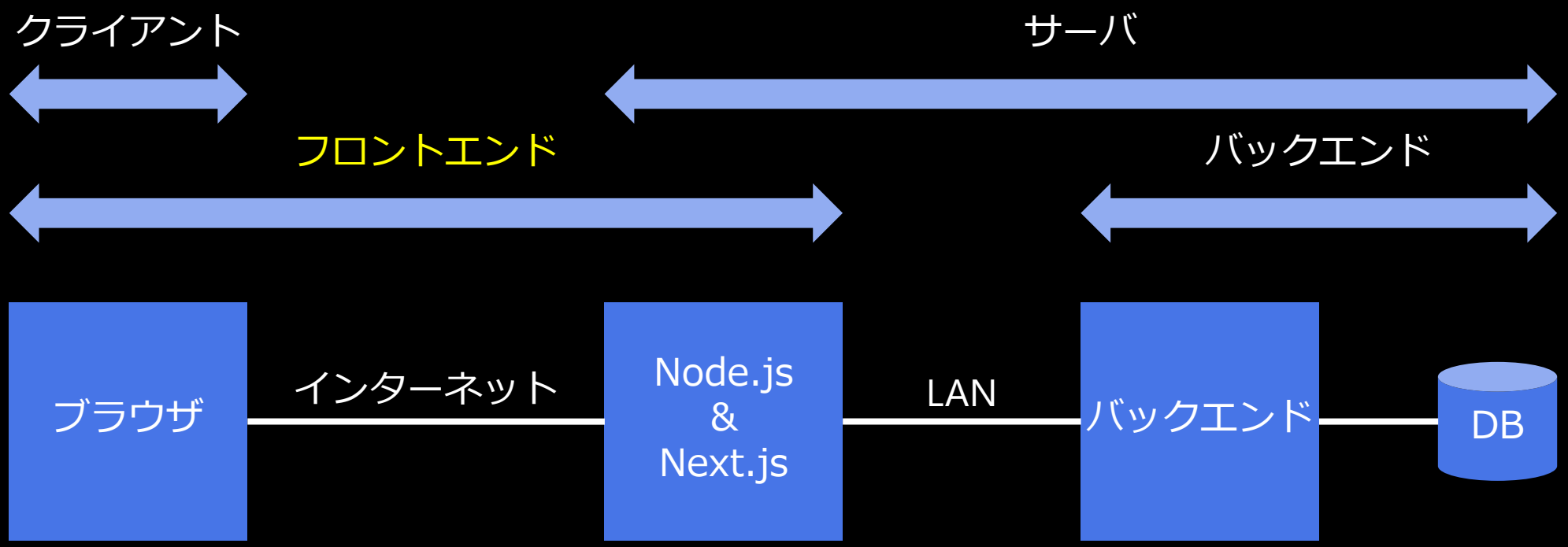


(メタ) フレームワーク

- React等をベースに事前レンダリング、ルーティング、データフェッチ等の機能を付加したもの
- Reactベースの (メタ) フレームワーク
  - 例: Gatsby, Next.js, Remix, etc.
- React以外をベースとする (メタ) フレームワーク
  - 例: Nuxt (VueJS), Angular Universal, SvelteKit, SolidStart, etc.
- 呼称について
  - React等をライブラリと呼ぶ場合
    - Next.js等をフレームワークと呼ぶことが多い
  - React等をフレームワークと呼ぶ場合
    - Next.js等をメタフレームワークと呼ぶことが多い

React公式
ドキュメントはこちら

広がるフロントエンドの領域



フロントエンドチームが開発運用するサーバを
Backend for Frontend (BFF)
と呼ぶことがあります

Webアプリ開発の変遷 まとめ

- 現代の典型的なWebアプリ
  - MPA (クラシックSSR + jQueryまたはVanilla JS)
    - 古くからある既存システムとその周辺のサービスに多い
  - SPA (CSRのみ)
    - SEOも初期表示の速度も要求されない新規サービスで選ばれやすい
  - SPA (CSR + 事前レンダリング)
    - SEOや初期表示の速度が要求される新規サービスで選ばれやすい

Webアプリ開発の変遷 まとめ

- Webアプリ (フロントエンド寄り) 開発者の立ち位置
  - クラシックSSRのテンプレート開発者
    - サーバサイドのテンプレート記述言語 (ERB, JSP, Thymeleaf, etc.) を利用
  - クラシックSSRが生成したHTMLと共に動作するJSの開発者
    - 主にjQueryまたはVanilla JSを利用
  - SPAのJS開発者
    - SPA用のライブラリやフレームワークを利用
      - 主にReact, VueJS
    - 事前レンダリングを行う場合は (メタ) フレームワークも利用
      - 主にNext.js, Nuxt
      - ブラウザに留まらずサーバサイド (BFF) もフロントエンドの領域に

Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外リソースとの同期
- メモ化とパフォーマンス

Reactとは

- UI構築のためのライブラリ
  - 主にSPAの開発に使われる
    - MPAやネイティブアプリの開発に使うこともできる
- 開発元はMeta (旧Facebook)
  - 2011年からFacebook内部で利用開始
  - 2013年にOSSとして公開
    - https://github.com/facebook/react
- TypeScriptではなくFlowtypeで記述されている
  - TSの型定義はコミュニティ (Definitely Typed) より提供されている

Reactの特徴

- Viewに特化
- コンポーネント指向
- JSX
- 宣言的UI
- 仮想DOM (Reactツリー)
- Learn Once, Write Anywhere

Viewに特化

- MV*フレームワークではない
  - そのためReactは「ライブラリ」と自称している
- メリット
  - 小さく始めやすい
  - 間違った (早すぎる) ベストプラクティスを押しつけない
  - 3rd Partyライブラリが活発に開発されてエコシステムが充実
- デメリット
  - Viewレイヤ以外をどうするかはアプリ開発側で考える必要がある

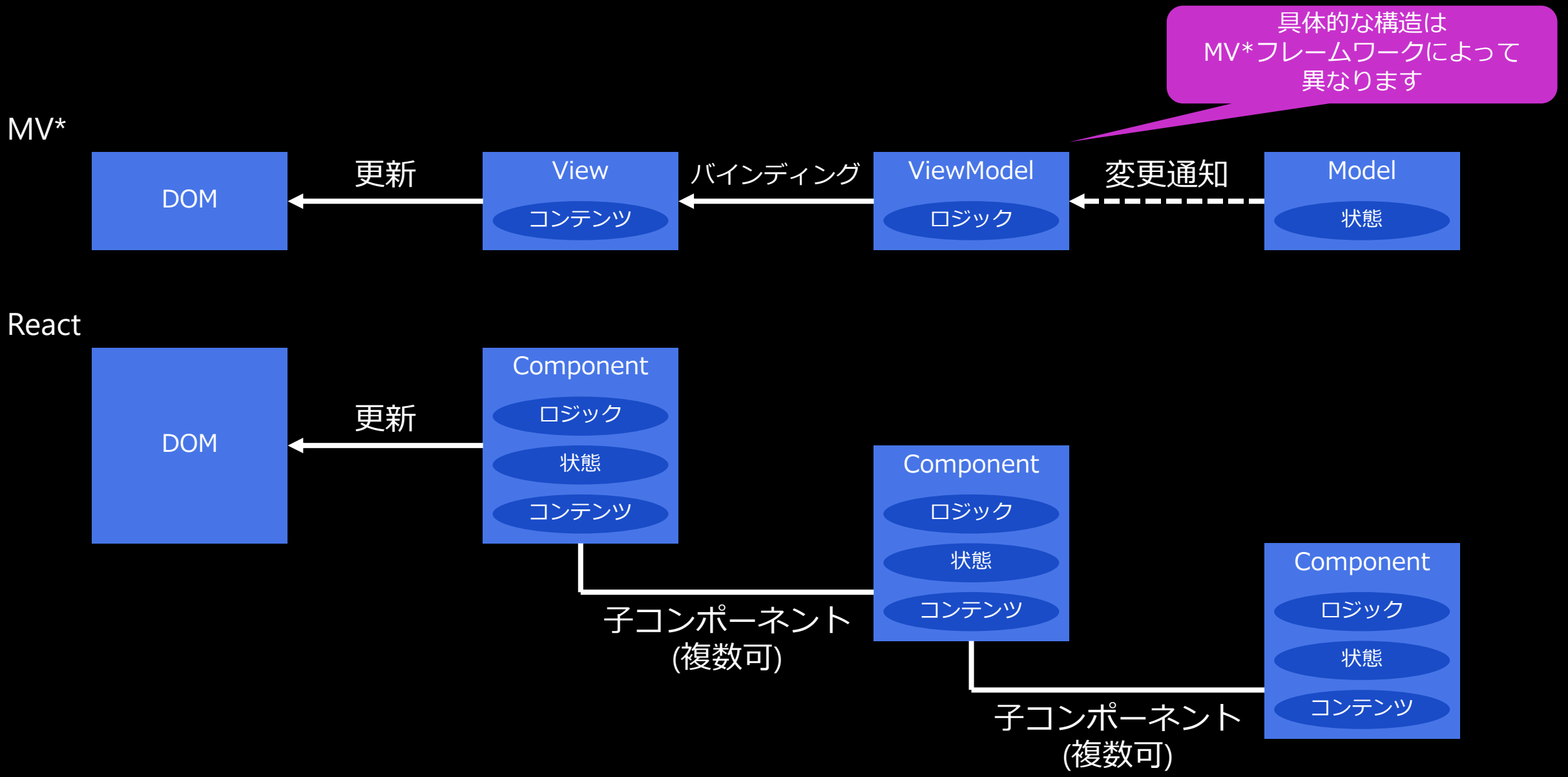
Viewレイヤ以外は？

- 当初はMV*フレームワークとの組み合わせが試された
  - Backbone.jsなど
- Facebook自身もMV*の一種「Flux」アーキテクチャを提唱
  - Fluxアーキテクチャを実装したOSSフレームワークが多数リリース
  - Flux戦争と呼ばれた
- 実際はアプリレベルでMV*アーキテクチャは不要だった
- MV*以外にも必要なものはある
  - ルーター、データフェッチ、ステート管理、フォーム管理、etc.
    - OSSエコシステムの競争から勝者 (デファクト) が決定
  - Next.js等の (メタ) フレームワークである程度解消
    - それでも足りないものはアプリ開発者自身で選択する

コンポーネント指向

- Reactアプリの構成要素はコンポーネント
  - ModelやController等は不要
- コンポーネントが持つもの
  - 表示のためのロジックやイベントハンドラ
  - 状態
  - コンテンツ (マークアップ)
- コンポーネントは子コンポーネントを持つことができる
  - コンポーネントのツリーを構築する

MV*とReactコンポーネント



JSX

- JSにマークアップ (HTML) を埋め込む構文
  - 実際はHTMLではなくXML風の構文
  - ツールによってJSの式に変換される
- ReactとJSXは独立している
  - JSXを使わずにReactを使うこともできる
  - React以外でJSXを使うこともできる

JSX

```
export default function App() {
  ...
  return (
    <div>
      <Header />
      <Main />
      <Footer />
    </div>
  )
}
```

JSX

JSX

- 当初は不評だった
  - コンテンツ (HTML), スタイル (CSS), ロジック (JS) を分離することがよいプラクティスだと考えられていたため
    - 後に間違った「Separation of Concerns」だったとみなされるように変化
- 現在では広く受け入れられている
  - Babel, TS, VSCode等のツールによる幅広いサポート
  - React以外のUIライブラリでもサポートされている
    - 例: VueJS, SolidJS, Qwik, etc.

宣言的UI

- 宣言的
  - whatを記述する
    - 選択中のタブがn番目ならXXXを表示する
  - 状態に対して一意に定まるUIを定義する
    - UI = f(state)
- 対義語は命令的
  - howを記述する
  - jQueryは命令的になりがち (特に更新)
    - n番目のタブがクリックされたら、
    - 現在選択中のタブがn番目でないことを確認し、
    - タブの下のパネルに他のタブ用の要素がもしあれば削除し、
    - パネルに新しい要素を追加し、属性を設定し、子要素を追加し、テキストを設定し、……

jQueryを使わない
Vanilla JSでも同様に
命令的になりがちです

宣言的UI

- Reactのメンタルモデル
  - 状態が変わる毎にコンポーネントを毎回実行してDOMを新規に構築
    - コンポーネント = (state) => DOM
  - 毎回新規にレンダリングするのと同等
    - 画面の更新について考えることが激減
- クラシックSSRのテンプレートに近いメンタルモデル
  - クラシックSSRではHTTPリクエストのたびに毎回テンプレートを実行してHTMLを生成する
    - テンプレート = (data) => HTML
  - 画面の更新については考える必要がない
    - 更新はブラウザ側のjQuery (またはVanilla JS) の仕事

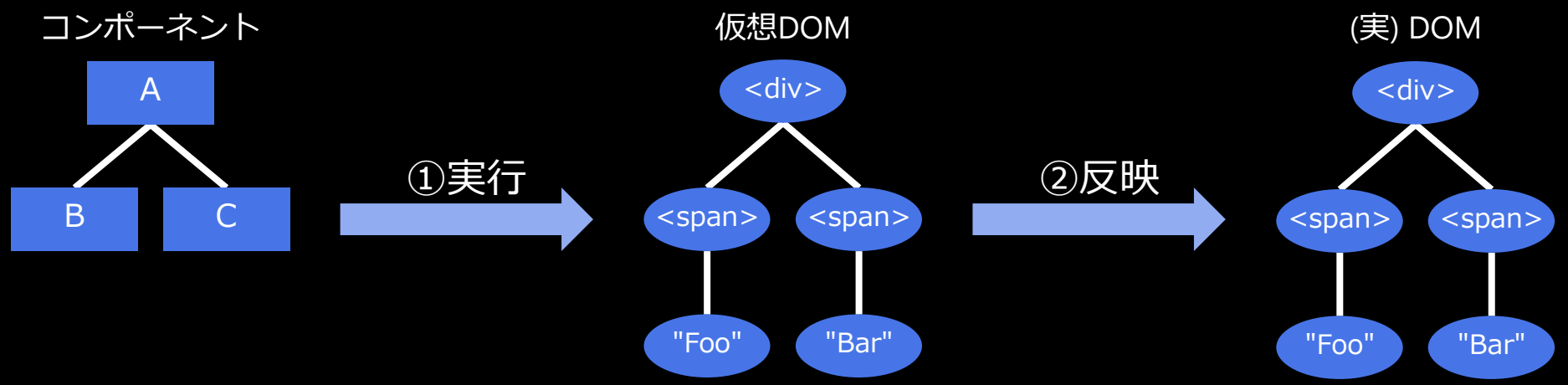
押しつけられた側は
命令的でとても大変

仮想DOM (Reactツリー)

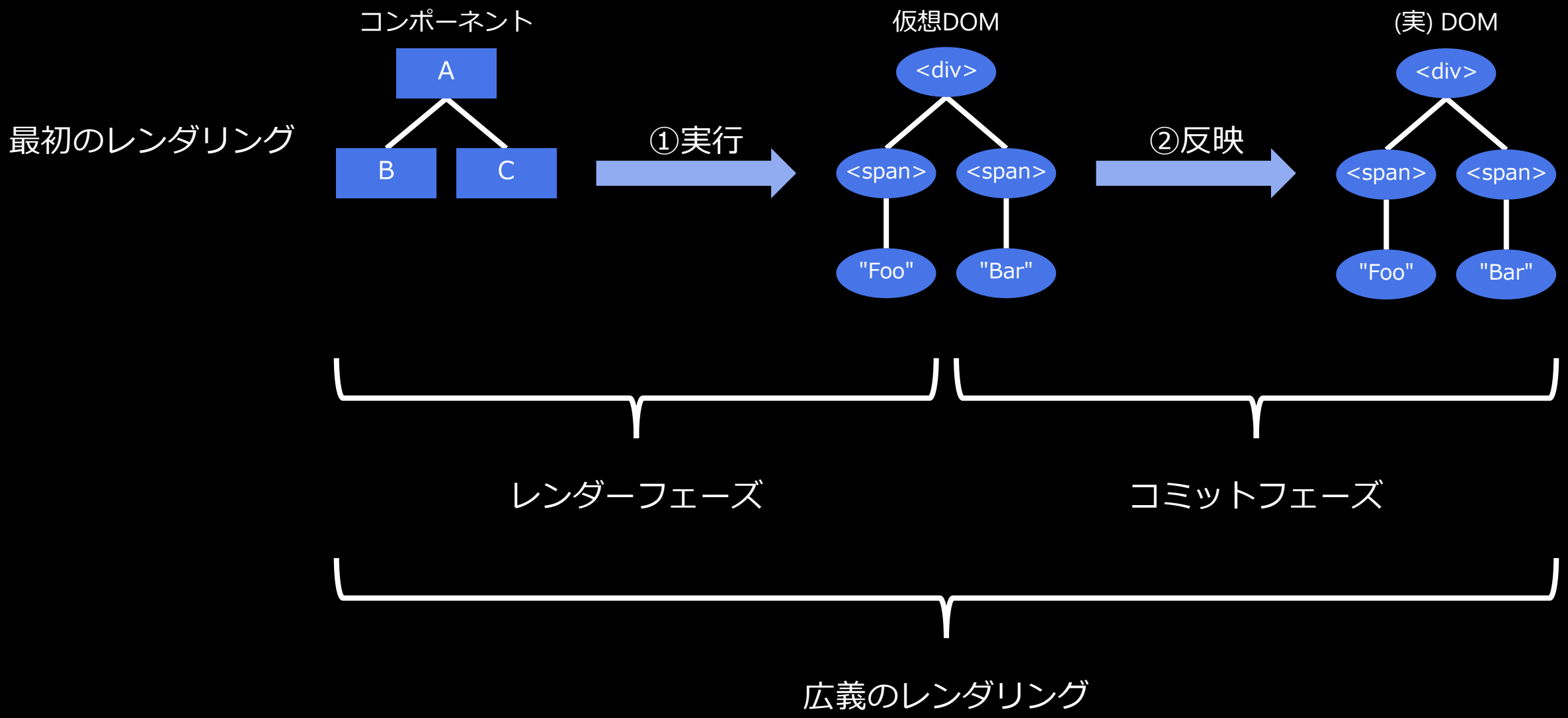
- Reactのメンタルモデル
  - 状態が変わる毎にコンポーネントを毎回実行してDOMを新規に構築
- 現実のDOMは遅い
  - 特に更新が遅い
  - メンタルモデルそのままではパフォーマンスが実用的にならない
- 仮想DOM
  - DOMの代わりにJSのオブジェクト (軽量) で仮想的なDOMを構築
    - コンポーネント = (state) => VDOM
  - 差分更新
    - 前回レンダリングした時の仮想DOMと新しい仮想DOMを比較
    - 差分だけを (実) DOMに反映

仮想DOMの動作イメージ (1)

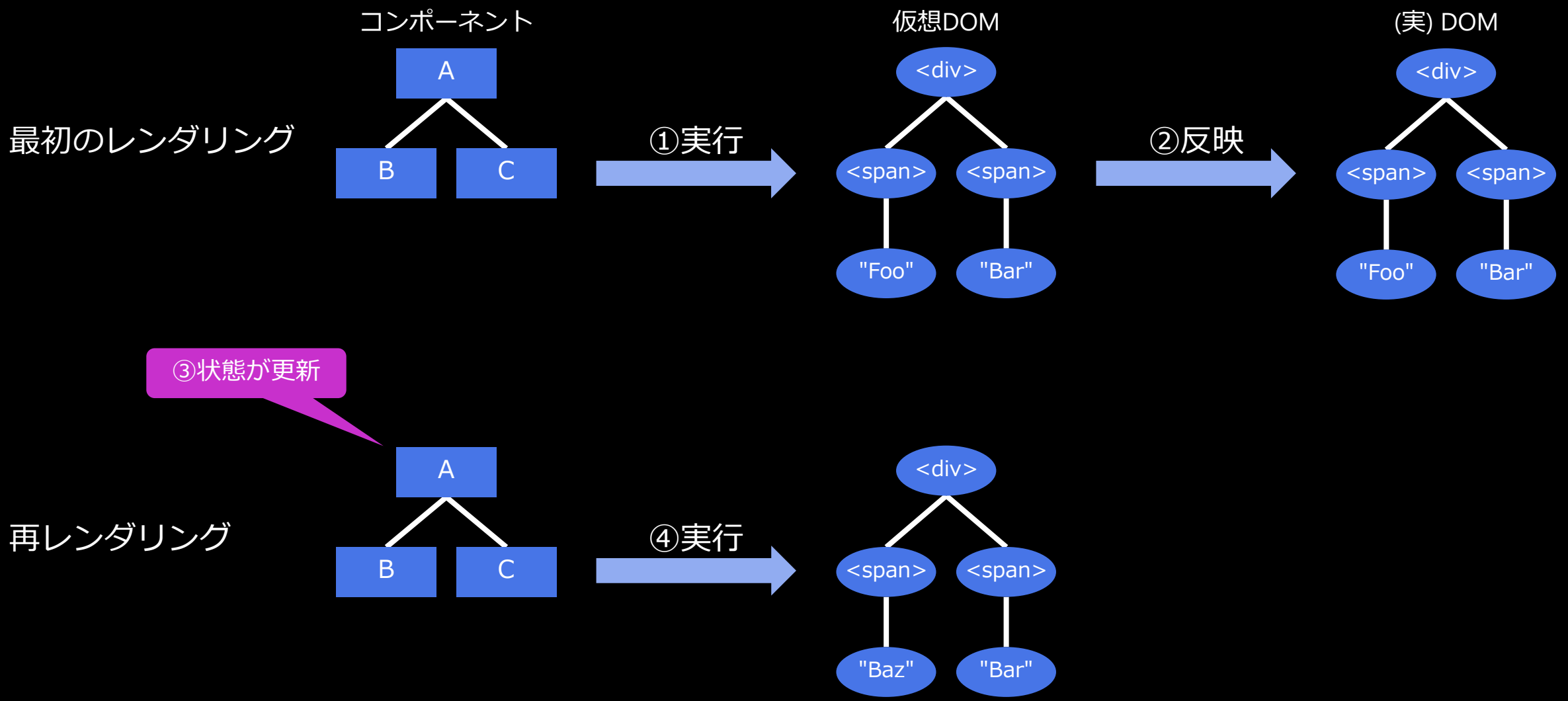
最初のレンダリング



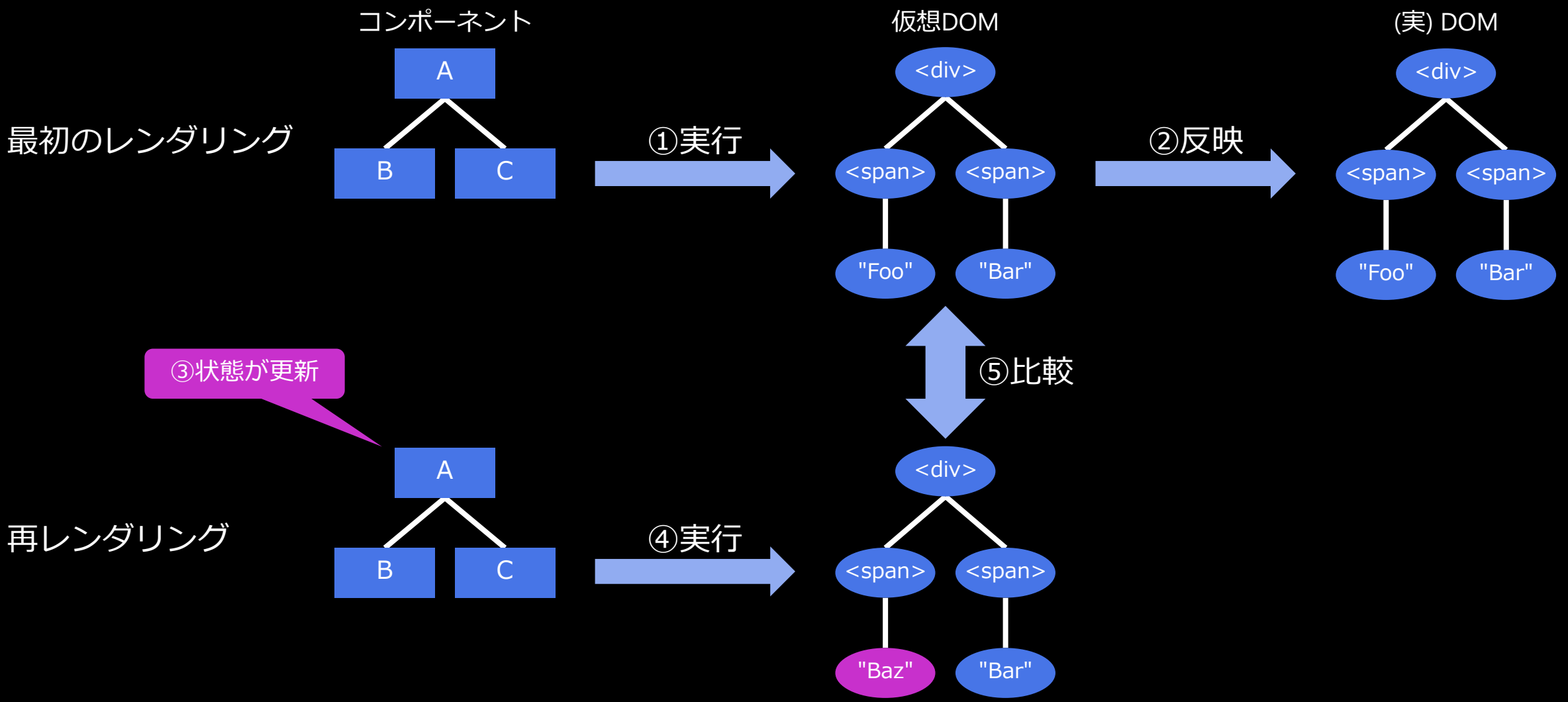
仮想DOMの動作イメージ (2)



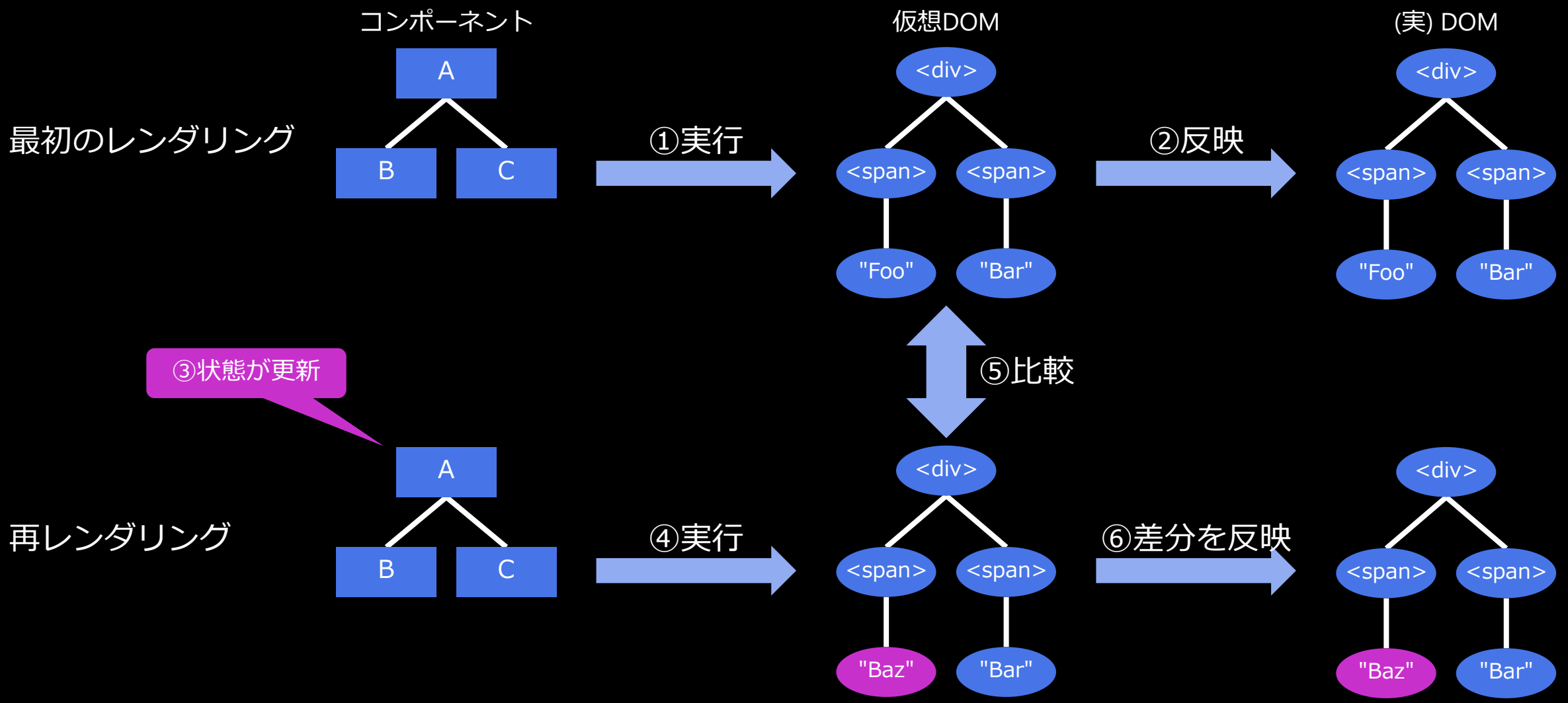
仮想DOMの動作イメージ (3)



仮想DOMの動作イメージ (4)



仮想DOMの動作イメージ (5)



仮想DOM (Reactツリー)

- 仮想DOMは最適化の1つ
  - 宣言的UIのメンタルモデルと実用的なパフォーマンスを両立
  - 両立する手段は仮想DOMだけではない
    - 近年は「Signals」をサポートするUIライブラリが増加している
- 「仮想DOMは速い」は (必ずしも) 正しくない
  - 「仮想DOMは (それほど) 遅くならない」程度が適切
  - 仮想DOMの構築を減らすためのパフォーマンスチューニングが必要になることもある
- 現在のReactは「仮想DOM」とは呼ばない
  - DOMを使わない環境 (React Native等) も存在するため
  - ドキュメント上は「Reactツリー」と呼ばれている
    - 差分検出処理のことは「Reconciliation」と呼ばれる

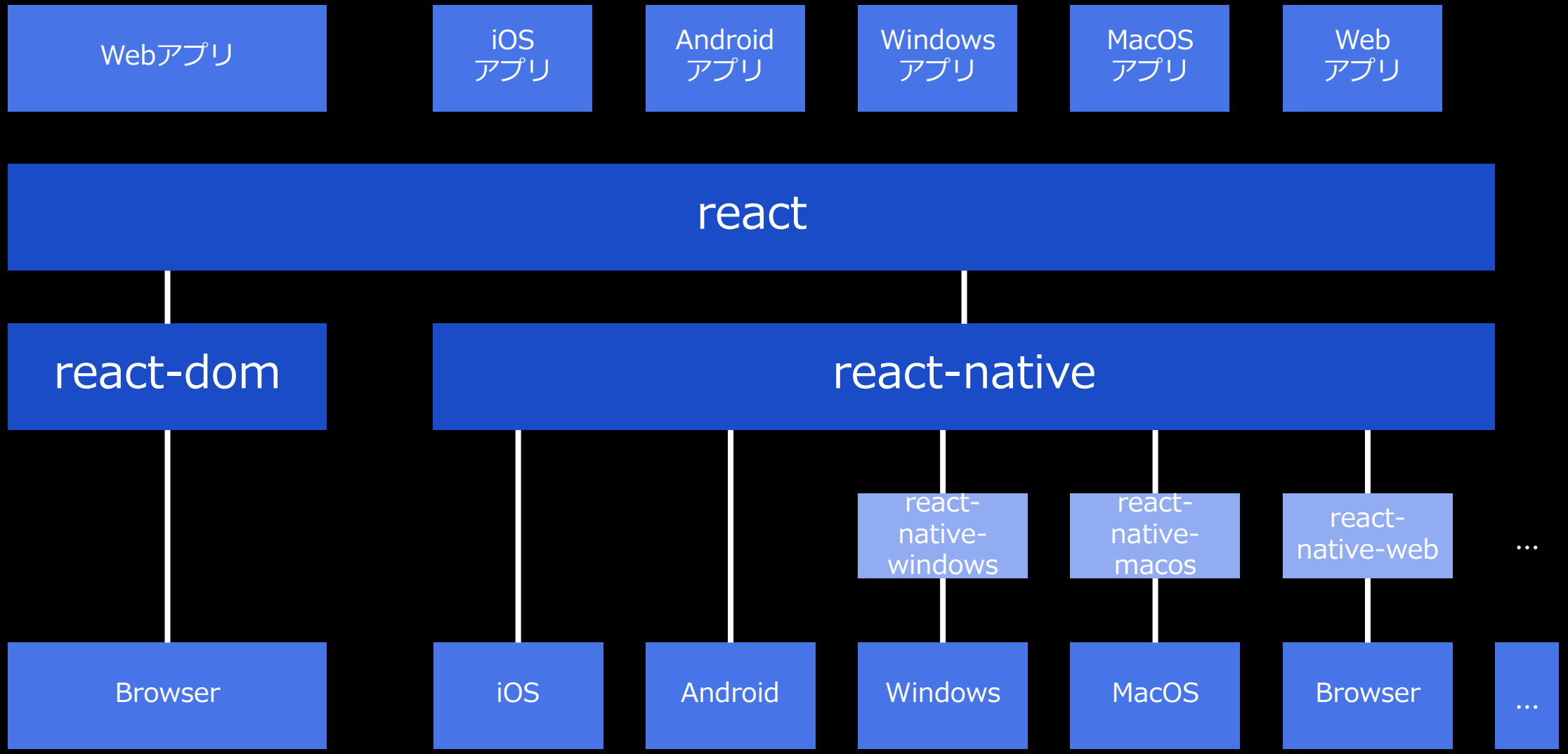
本研修では「仮想DOM」を使います

Learn Once, Write Anywhere

- ReactはWebアプリ開発だけのものではない
- React Native
  - ネイティブアプリ開発用
    - iOS, Android, Windows, Mac, etc.
- 一度Reactを学習すればWebもネイティブも書ける
- Reactのパッケージ構成
  - react
    - Web向け・ネイティブ向け共通のパッケージ
  - react-dom
    - Web向けのパッケージ
  - react-native
    - ネイティブ向けのパッケージ

本研修では
React Nativeは
扱いません

Reactのパッケージ構成



演習(2-1): StackBlitz

- StackBlitzを開いてみよう
  - https://stackblitz.com/
- GitHubアカウントでサインインしよう
  - 画面右上の「Sign in」をクリック
  - 「Continue with GitHub」をクリックしてサインイン
- 新しいProjectを作成しよう
  - 画面左上の「+New Project」ボタンを押す
  - 「Add to ～」モーダルが開く
  - 「React TypeScript」を選択
- 新しいProjectが開くのでソースを確認してみよう

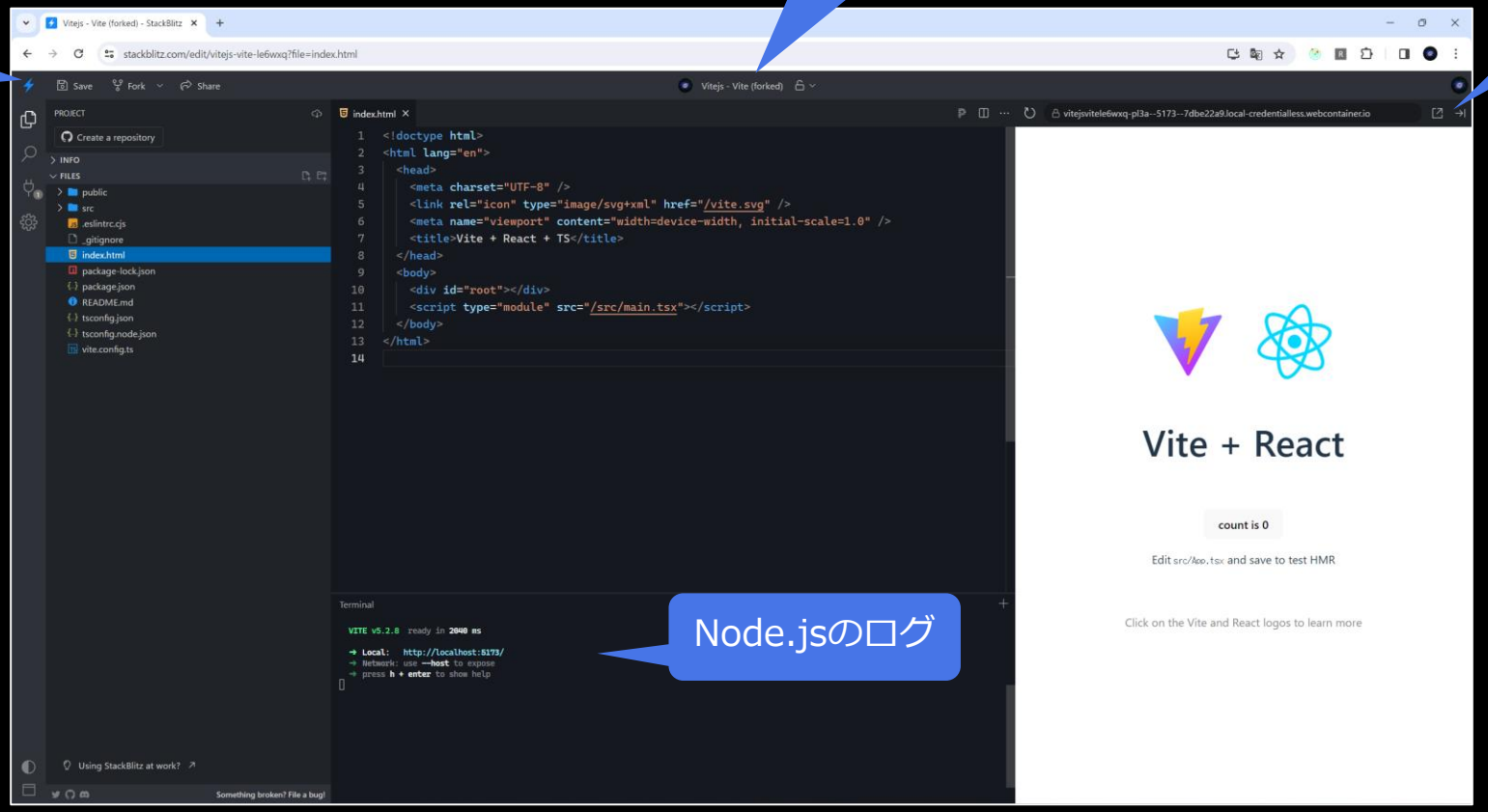
GHEではなく
github.comのアカウントです

StackBlitz

Projectの名前を
変更することができる

ダッシュボードに
移動できる

別タブでアプリを
開ける



Node.jsのログ

VSCode相当のIDE

ブラウザ内ブラウザ

index.html

最初に読み込まれる
HTML

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React + TS</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

src/main.tsx

TypeScriptでJSXを使うには
ファイルの拡張子を.tsxにします

Reactアプリの
エントリポイント

ルートとなるDOM要素に
Reactアプリをレンダー

これ以降はReactが
制御を握ってアプリを
呼び出す

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

src/App.tsx

にマウントされる
Appコンポーネント

JSX

```
...

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      ...
    </>
  )
}

export default App
```

この部分のテキスト(文言)を
自由に書き換えてみよう！

React概要 まとめ

- Reactは
  - UI構築のためのライブラリ
    - MV*フレームワークではなくViewに特化
  - コンポーネント指向、JSX
  - 宣言的UI
    - 状態が変化するたびに初回表示のようにコンポーネント全体を再実行するメンタルモデル
  - 仮想DOM (Reactツリー)
    - 宣言的なメンタルモデルとパフォーマンスを両立
  - Learn Once, Write Anywhere
    - Webアプリだけではなくネイティブアプリも開発可能

Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外リソースとの同期
- メモ化とパフォーマンス

コンポーネント

- Reactアプリの構成単位
  - コンテンツ、ロジック、状態を持つ
- コンポーネントの実装方法
  - クラスコンポーネント
    - 現在はほとんど使われない
  - 関数コンポーネント
    - 現在の主流

本研修では
クラスコンポーネントは
扱いません

関数コンポーネント

- JS/TSの普通の関数として実装するコンポーネント
- 関数名の先頭は大文字
- 引数
  - 親コンポーネントから渡されるオブジェクト (Props)
- 戻り値
  - 主にReactElement
    - ReactElementを構築するためにJSXを使う
  - null, undefined, boolean, number, string, 関数, 配列
  - 上記をまとめたReactNode型

関数コンポーネントの書き方

関数宣言 (文) で記述

```
type Props = {
  name: string;
};

export default function Message({name}: Props) {
  return <div><span>Hello, {name}!</span></div>;
}
```

最近の主流

アロー関数式で記述

```
type Props = {
  name: string;
};

export const Message: React.FC<Props> = ({name}) => {
  return <div><span>Hello, {name}!</span></div>;
};
```

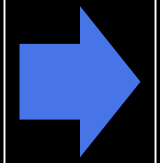
functionキーワードを使った
関数式はあまり使われない

React.FC<Props>は
@types/reactで定義された型

JSX

- JS XML
- JSの式としてXML風の構文を記述できる
  - BabelやTS (tsc) 等のツールによりJSの式に変換される (Alt JS)

```
import React from "react";
import { Child } from "./Child";

export function App() {
  return ( // ←の括弧がないと;が挿入される (ASI)
    <div className="foo">
      <span>Hello, React!</span>
      <Child name={"Foo"} />
    </div>
  );
}
```



```
import React from "react";
import { Child } from "./Child";

export function App() {
  return (
    React.createElement("div", { className: "foo" },
        React.createElement("span", null, "Hello, React!"),
        React.createElement(Child, { name: "Foo" })));
}
```

現在はよりコンパクトなJSに
トランスパイルされます

JSXとHTMLの違い

- JSXはXML風の構文であって通常のHTML構文と同じではない
  - かつてのXHTMLに近い
  - 現在のHTML Living Standardでは「The HTML Syntax」よりも「The XML Syntax」に近い
  - 特に属性名はHTML Living StandardでWeb IDLにより定義されるDOMインタフェースの属性名が採用されている

JSXとHTMLの違い (要素)

DOM要素にマッピングされる
コンポーネントは
「ホストコンポーネント」
と呼ばれることがあります

- タグ名は小文字と大文字を区別する
  - タグ名の先頭が小文字ならDOM要素にマッピングされる
    - 例: ...
  - タグ名の先頭が大文字ならReactコンポーネントにマッピングされる
    - タグ名は関数への参照として解決できなくてはならない
    - 例: <Button>...</Button>
  - ピリオド区切りでJSオブジェクトのメンバーを参照することができる
    - タグ名の先頭が小文字でもReactコンポーネントにマッピングされる
    - 例: <React.Suspense>...</React.Suspense>
- 終了タグは省略できない
  - 例: <li>..., <input />

JSXとHTMLの違い (属性名)

- 小文字と大文字を区別する
- 主にキャメルケースを使う
  - 例: <a referrerPolicy="origin">…
  - 例外: data-*属性, aria-*属性
- HTMLと異なる属性名がある
  - class → className
  - for → htmlFor

属性名はJSX仕様ではなく
react-domのAPIで
決められています

HTML Living Standardの
DOMプロパティ名に
近いです (一部例外あり)

この他にフォームや
CSSに関連する属性に
違いがあります (後述)

JSXとHTMLの違い (属性値)

- 属性値は一重または二重引用符 で囲む (省略不可)
  - 例: <input type="checked" />
- 属性値にJSの式を使うこともできる
  - 例: <input value={text} />
- 論理属性にはboolean型のJS式を使うことができる
  - 例: <input type="checkbox" checked={flag} />
- 論理属性がtrueの場合は属性値を省略できる (HTMLと同様)
  - 例: <input type="checkbox" checked />

JSXの中でJSの式を
使う方法は後述

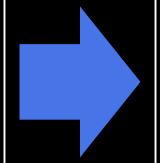
ルート要素

- JSXの式は単一のルート要素を持つ
- ルート要素として対応するHTML要素を書けない場合は「フラグメント」要素を使う
  - <>…</>
  - または<React.Fragment>…</React.Fragment>

後述するkey属性を指定する場合はこちら

```
import React from "react";

export function ListItem() {
  return (
    <dt>タイトル</dt>
    <dd>説明</dd>
  );
};
```

間違い



```
import React from "react";

export function ListItem() {
  return (
    <>
      <dt>タイトル</dt>
      <dd>説明</dd>
    </>
  );
}
```

JS in JSX

- JSXの中にJSの式を記述することができる
  - JSX構文の中でJSの式を使うにはJSの式を波括弧 {} で囲む
  - JSの式は属性値にも要素の内容にも利用できる
  - JSとJSXは任意にネストできる

```
export default function App() {
  const id = "abc";
  const flag = !Math.floor(Math.random() * 2);

  return (
    <div id={id + "def"}>
      <div>{flag ? <span>Foo! {random}</span> : <span>Bar! {random}</span>}</div>
    </div>
  );
}
```

黄色の文字はJSX
白色の文字はJS

演習(3-1): JSの値をJSXで表示

- Stackblitzで新しいProjectを作ってみよう
- JSXの中からJSの様々な値をレンダリングしてみよう
  - プリミティブ値
    - undefined, null, boolean, number, string, etc.
  - オブジェクト
    - 普通のオブジェクト, 配列, 関数, 正規表現, Date, etc.

Projectの名称を
[3-1] JSX
などに変更しよう

numberやstringは
様々な値を
表示してみよう

```
export default function App() {
  return (
    <ul>
      <li><span>undefined</span>:<span>{undefined}</span></li>
      <li><span>null</span>:<span>{null}</span></li>
       ・・・
    </ul>
  );
}
```

JS値とテキストノード

- JSXのテキストノードに書かれたJSの値は次のように扱われる
  - 表示されない
    - undefined, null, boolean, bigint, symbol, 関数 (警告が出る)
  - 表示される
    - string
      - 一部の文字 ('<', '>', etc.) はエスケープされる
  - 表示されるが実用的でない
    - number
      - 通常はアプリでのフォーマットが必要
  - 実行時エラー
    - 関数と配列以外のObject
  - 配列
    - 各要素について上記が適用される (区切り文字なしで連結)

JSXと制御構造

- JSX自体は制御構造を提供しない
- JSの制御構造と組み合わせることができる
- JSX中では主に式を使う
  - 分岐
    - 論理演算子, 条件演算子
      - 論理演算子はboolean値が表示されないことを利用する
  - 反復
    - Array.prototype.map(), etc.

JSXと制御構造

論理演算子による分岐

```
export default function App() {
  const flag = !Math.floor(Math.random() * 2);

  return (
    <div>{flag && <span>true!!</span>}</div>
  );
}
```

条件演算子による分岐

```
export default function App() {
  const flag = !Math.floor(Math.random() * 2);

  return (
    <div>{flag ? <span>true!!</span> : <span>false!!</span>}</div>
  );
}
```

JSXと制御構造

Array.prototype.map()による繰り返し

```
export default function App() {
  const numbers = [1, 2, 3, 4, 5];

  return (
    <ul>
      {numbers.map((n) => (
        <li>
          <span>{n}</span>  <span>{n % 2 ? "odd" : "even"}</span>
        </li>
      ))}
    </ul>
  );
}
```

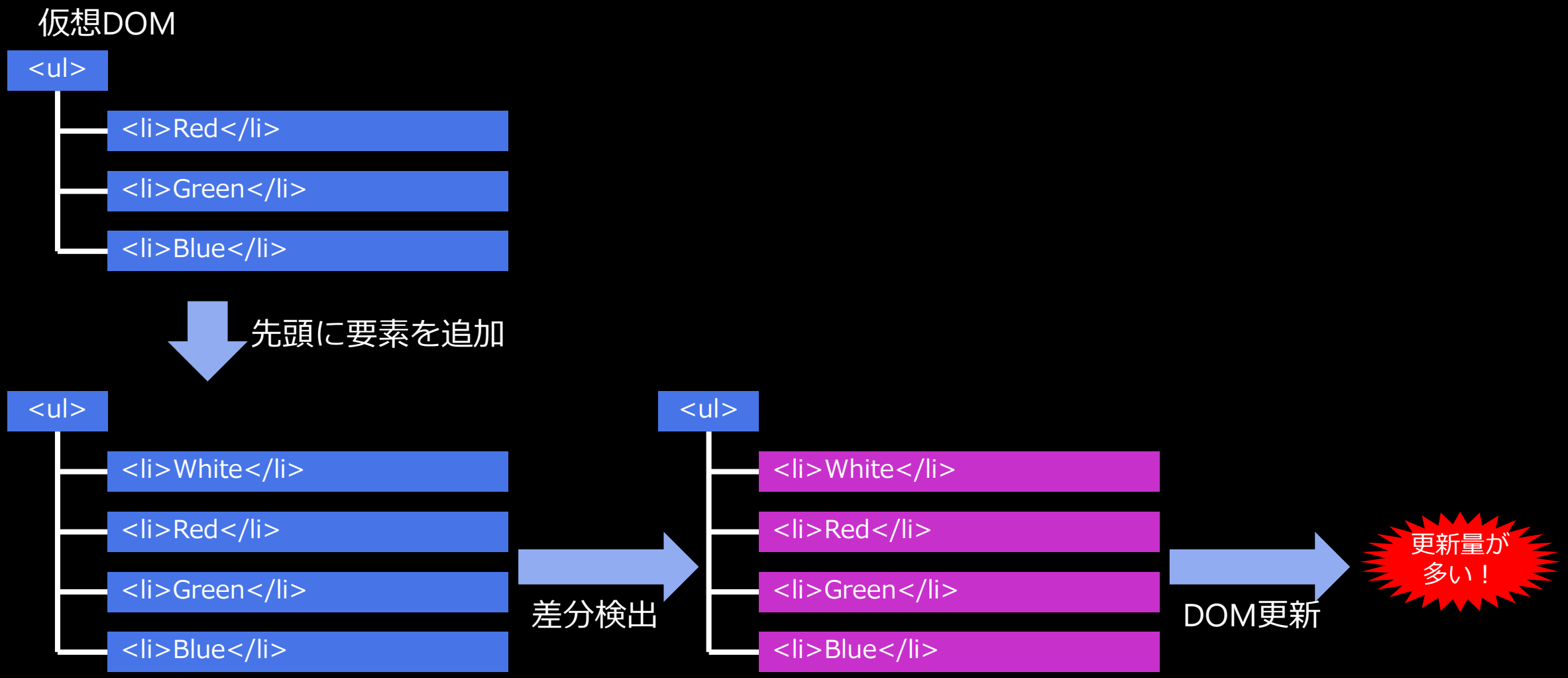
繰り返しとkey属性

- 前頁の例を実行すると警告が出力される

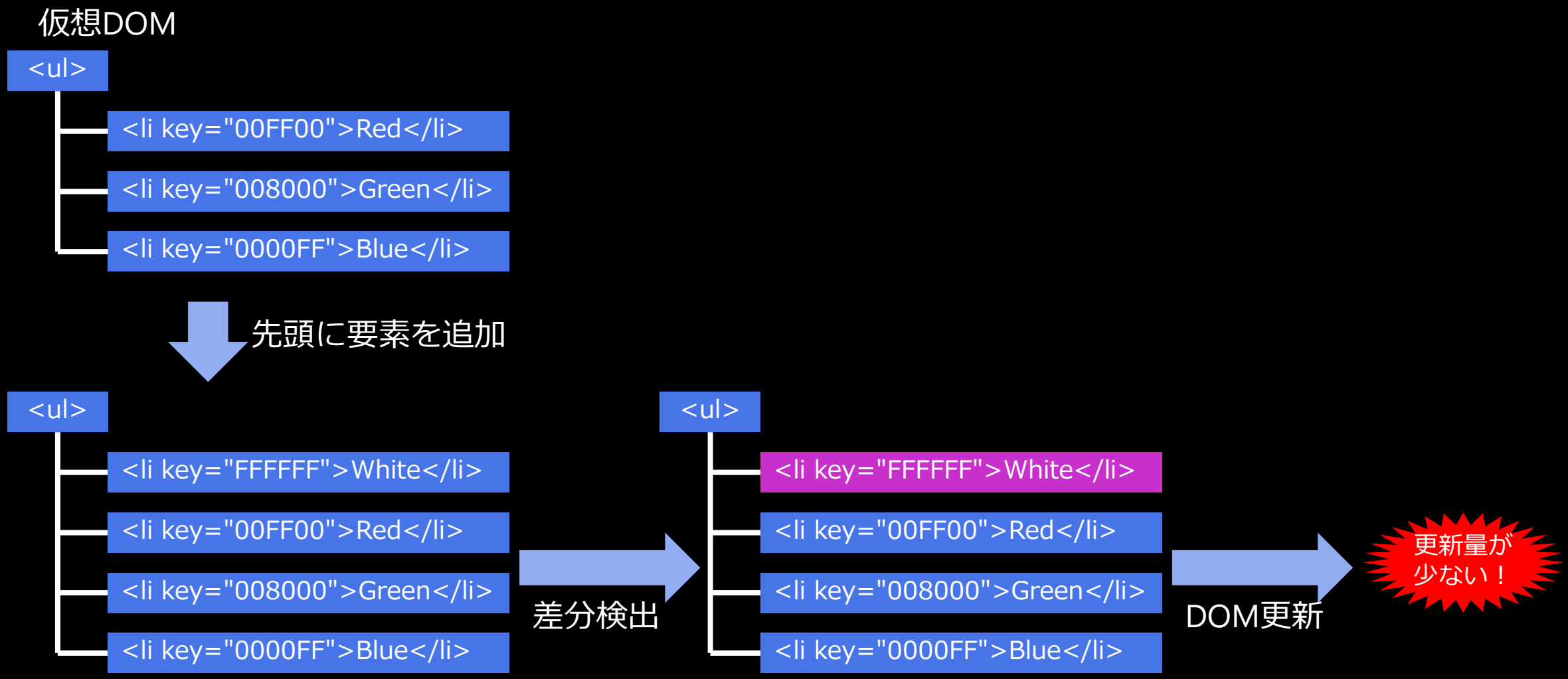
```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.
    at li
    at App
```

- 繰り返される要素にはkey属性が必要
  - 繰り返し中における個々の要素をReactが識別できるようにするため
  - 仮想DOMによる差分検出処理で使われる
- key属性の値は安定した値が望ましい
  - 商品一覧における商品であれば「商品コード」など
  - 配列のインデックスは極力避ける

繰り返される要素の差分検出 (keyなし)



繰り返される要素の差分検出 (keyあり)



演習(3-2): Todoアプリ

Projectの名称を
[3-2] Todo
などに変更しよう

- StackBlitzで新しいProjectを作成してみよう
- src/App.tsxを修正してTodoのリストを表示してみよう

```
const todoList = [
  { id: 1, task: "Learning Browser", completed: true },
  { id: 2, task: "Learning JavaScript/TypeScript", completed: true },
  { id: 3, task: "Learning React", completed: false },
  { id: 4, task: "Learning Next.js", completed: false },
];

export default function App() {
  return (
    // ここを埋めてください
    // completedの表示には<input type="checkebox" />を使ってください
    // この時点ではチェックボックスは更新できないので<input>要素にreadOnly属性を指定してください
  );
}
```

コンポーネントの分割

- コンポーネントは親と子に分割することができる
- 親コンポーネントはJSX要素の開始タグに属性を列挙して子コンポーネントにデータを渡すことができる
  - 属性にオブジェクト (配列や関数を含む) を渡すこともできる
    - オブジェクトはイミュータブルとして扱うこと

作成したオブジェクトを変更しないこと

Props

- 子コンポーネントは引数でデータを受け取ることができる
  - この引数 (オブジェクト) をPropsと呼ぶ
  - Propsの各プロパティは親コンポーネントが渡した属性
  - Propsはイミュータブルとして扱うこと
- 特殊なProps
  - children: 親コンポーネントにおいてJSX要素の属性ではなく、開始タグと終了タグの間に記述された子ノードの配列
  - key: 子コンポーネントには渡らない
    - 将来のReactでは通常のプロパティになる予定です
  - ref: 子コンポーネントが受け取るにはforwardRef()が必要
    - 将来のReactでは通常のプロパティとなり、 forwardRef()は不要となる予定です

後述します

「状態と再レンダリング」で説明します

コンポーネントとProps

親コンポーネント

```
import { Child } from "./Child";

export function Parent() {
  return (
    <div>
      <Child name="Foo" />
      <Child name="Bar" />
    </div>
  );
}
```

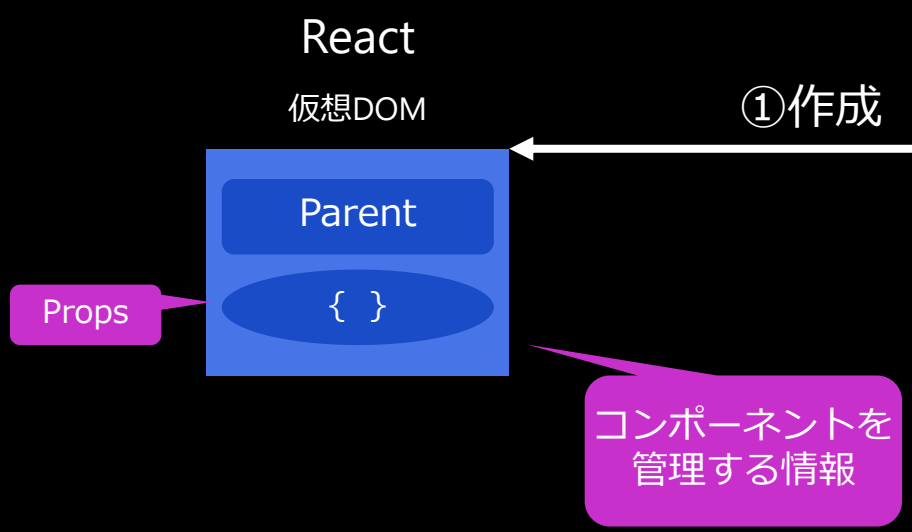
子コンポーネント

```
export type Props = {
  name: string;
};

export function Child({ name }: Props) {
  return <span>{name}</span>;
}
```

仮想DOMとコンポーネント

- 関数コンポーネントにはクラスにおける「インスタンス」は存在しない
- 関数コンポーネントそのものは状態を持たない
  - ステートレス
- しかしReactはコンポーネントの情報を仮想DOMの一部として管理している
  - React内部では「Fiber」と呼ばれるデータ構造で管理している
- 関数コンポーネントはFiberに保持される情報と紐付けられる
  - Props, State, etc.

仮想DOMとコンポーネント



①作成

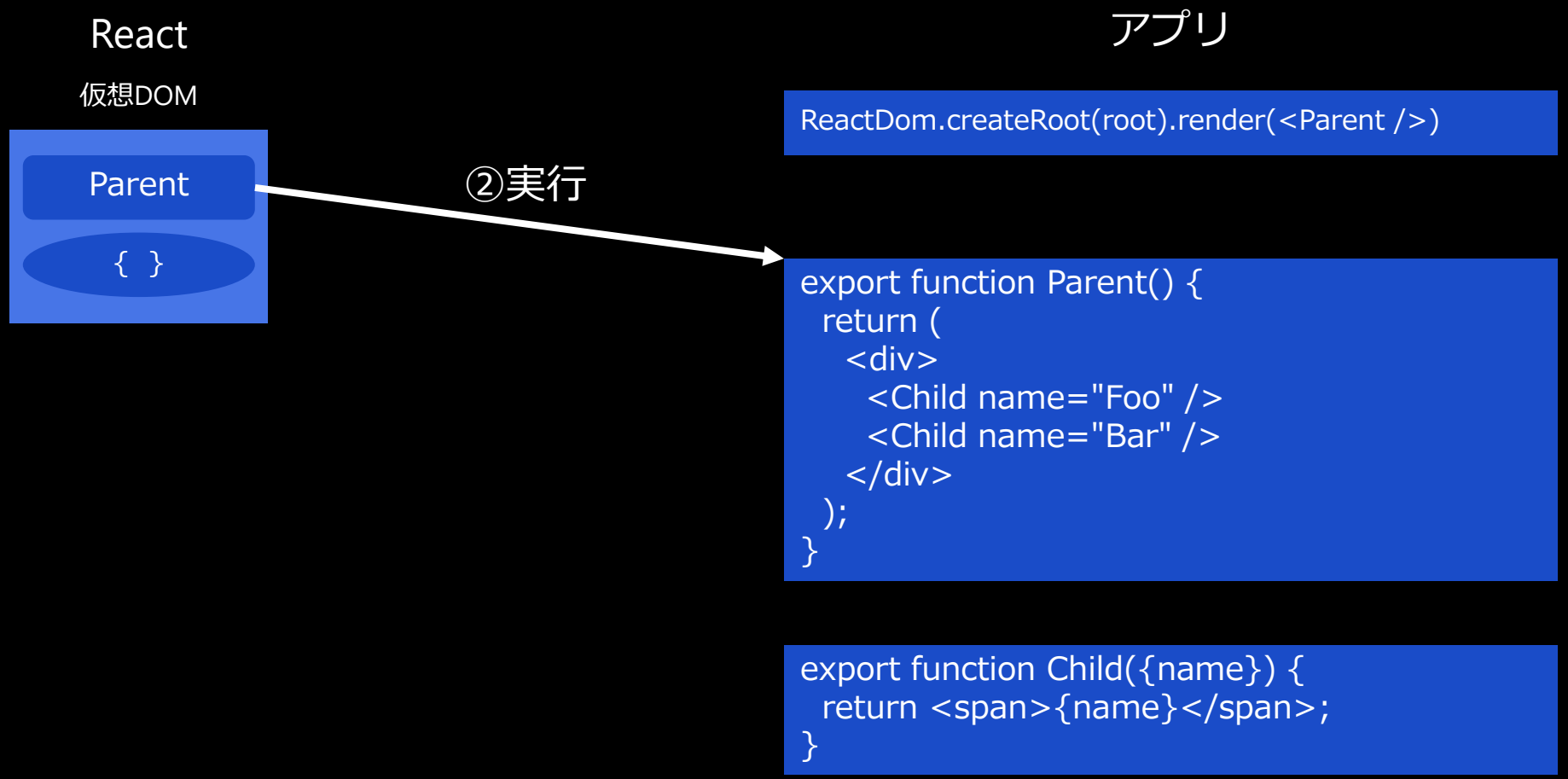
アプリ

```
ReactDom.createRoot(rootElement).render(<Parent />)
```

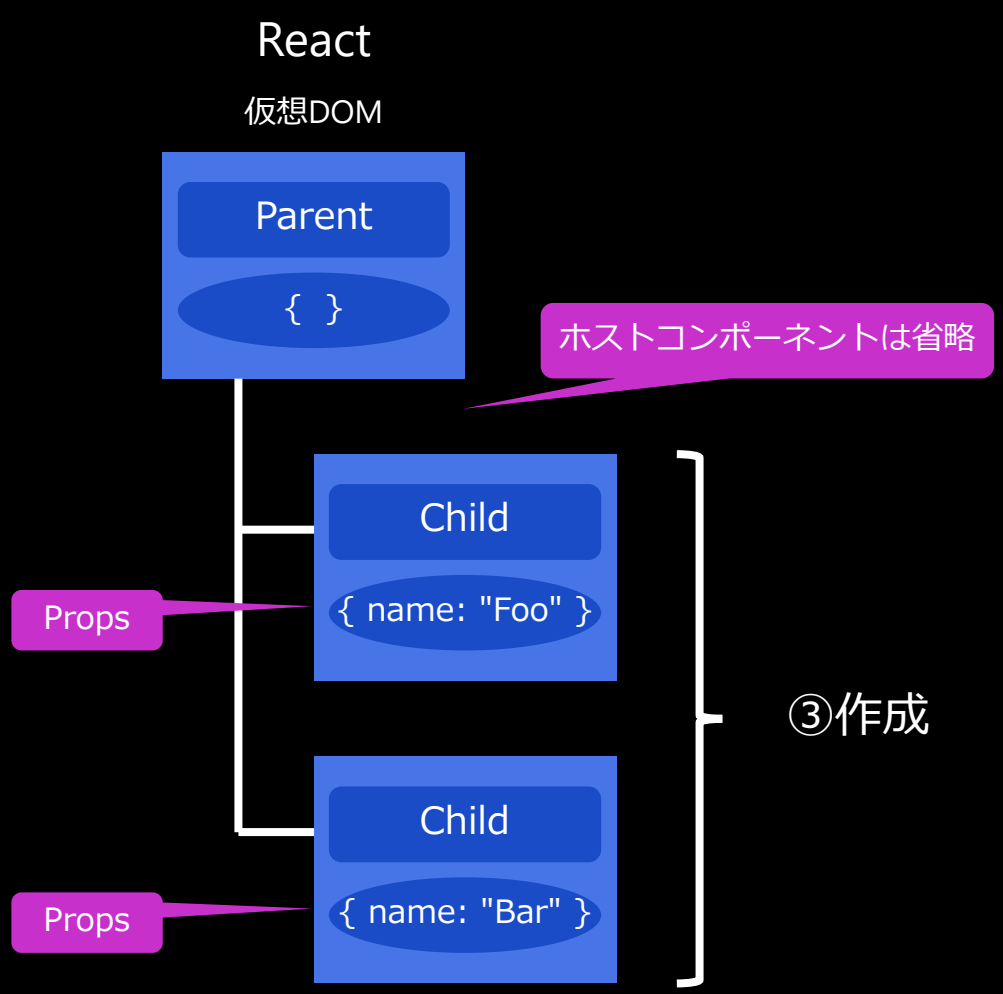
```
export function Parent() {
  return (
    <div>
      <Child name="Foo" />
      <Child name="Bar" />
    </div>
  );
}
```

```
export function Child({name}) {
  return <span>{name}</span>;
}
```

仮想DOMとコンポーネント



仮想DOMとコンポーネント



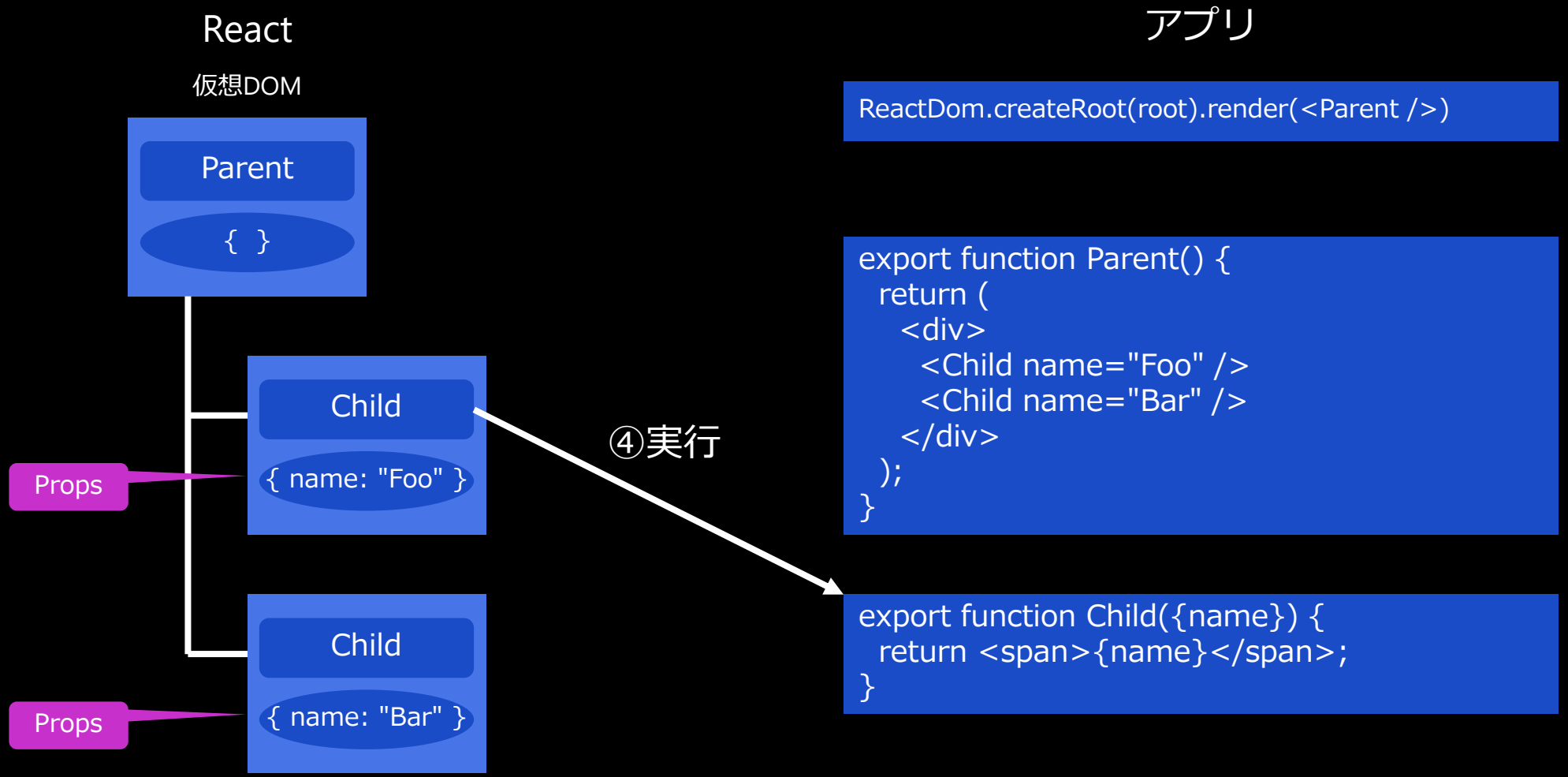
アプリ

```
ReactDom.createRoot(root).render(<Parent />)
```

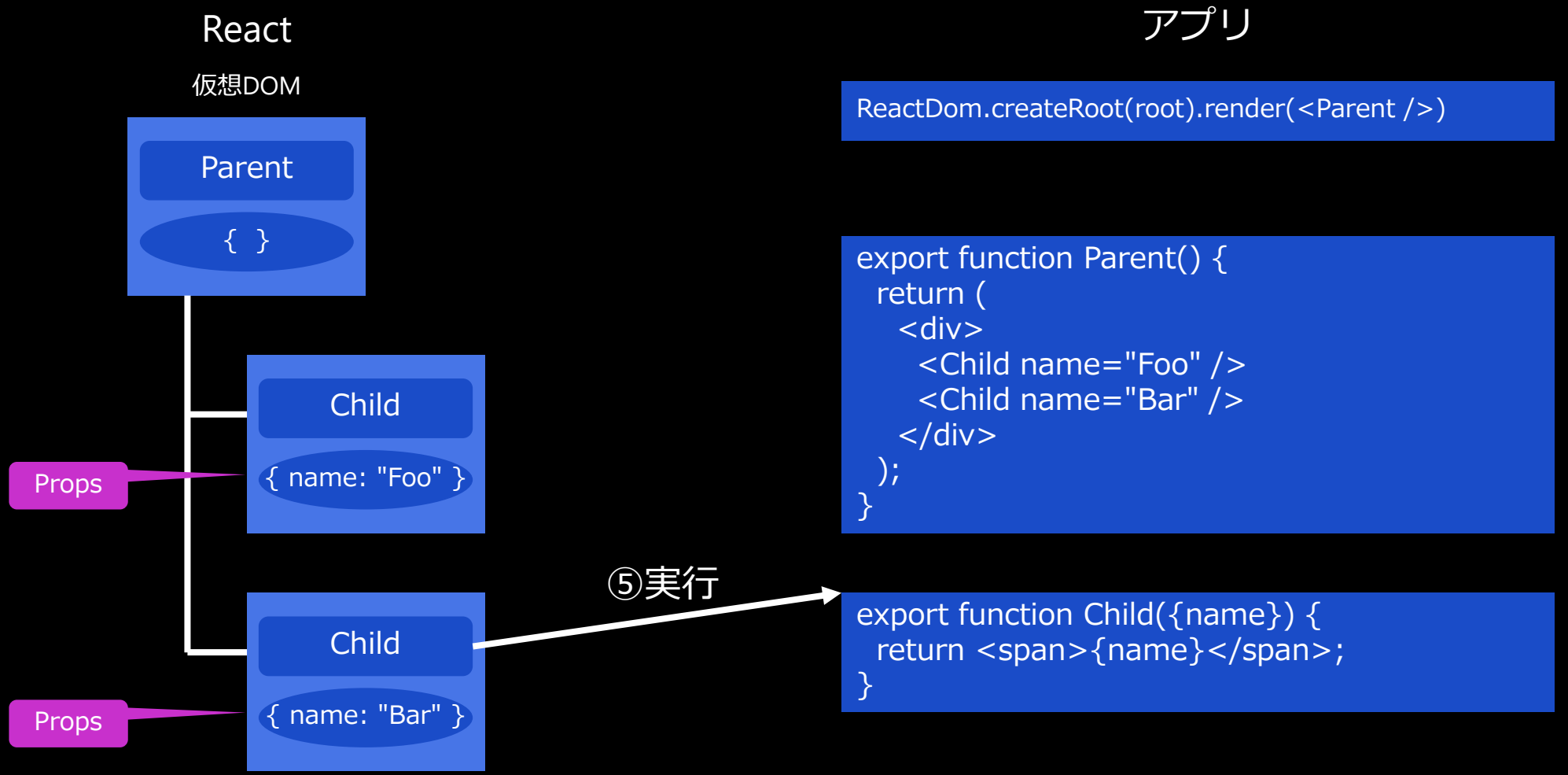
```
export function Parent() {
  return (
    <div>
      <Child name="Foo" />
      <Child name="Bar" />
    </div>
  );
}
```

```
export function Child({name}) {
  return <span>{name}</span>;
}
```

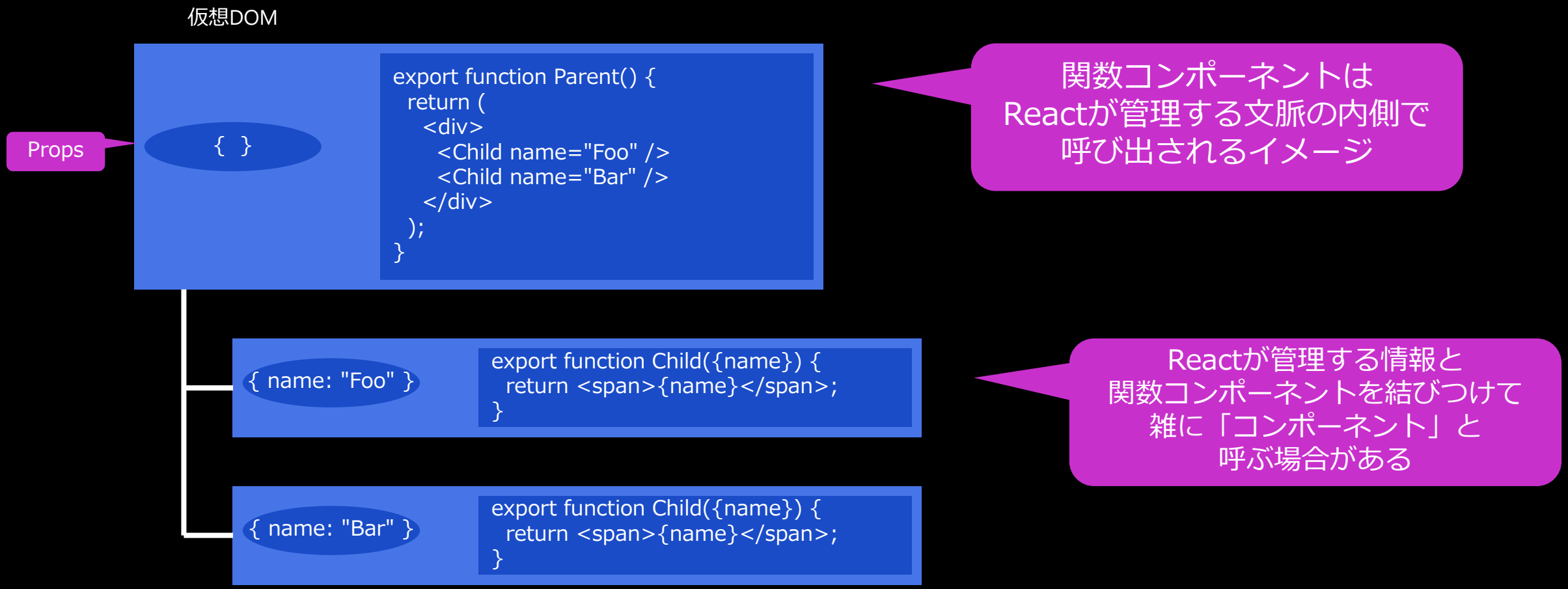
仮想DOMとコンポーネント



仮想DOMとコンポーネント



(広義の) コンポーネント


仮想DOM
Props
{ }
export function Parent() {
return (

<Child name="Foo" />
<Child name="Bar" />

);
}
{ name: "Foo" }
export function Child({name}) {
return {name};
}
{ name: "Bar" }
export function Child({name}) {
return {name};
}
関数コンポーネントは
Reactが管理する文脈の内側で
呼び出されるイメージ
Reactが管理する情報と
関数コンポーネントを結びつけて
雑に「コンポーネント」と
呼ぶ場合がある

演習(3-3): Todoアプリ

演習3-2のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[3-3] Todo
などに変更しよう

- Todoのアイテムを一つだけ表示するTodoItemコンポーネントを作成し、Appコンポーネントから使ってみよう

ディレクトリツリーの「src」に
マウスカーソルを重ねると
ファイルの追加ができます

src/App.tsx

```
import { TodoItem } from "./TodoItem";

export type TodoItemType = {
  id: number;
  task: string;
  completed: boolean;
};
const todoList: TodoItemType[] = [...];

export default function App() {
  return (
    ...
      <TodoItem ... />
    ...
  );
}
```

src/TodoItem.tsx

```
import { type TodoItemType } from "./App";

type Props = {
  todoItem: TodoItemType;
};

export function TodoItem(props: Props) {
  return (
    ...
  );
}
```

childrenプロパティ

- 親コンポーネントのJSXで子コンポーネントの開始タグと終了タグの間にノード (要素やテキスト) を記述できる
  - このノードの配列は子コンポーネントにPropsのchildrenプロパティとして渡される

childrenプロパティ

親コンポーネント

```
import { Layout } from "./Layout";

export function Parent() {
  return (
    <Layout>
      <div>...</div>
      テキスト
      <div>...</div>
    </Layout>
  );
}
```

子コンポーネント

```
export type Props = {
  children: React.ReactNode;
};

export function Layout({ children }: Props) {
  return (
    <div>
      {children}
    </div>
  );
}
```

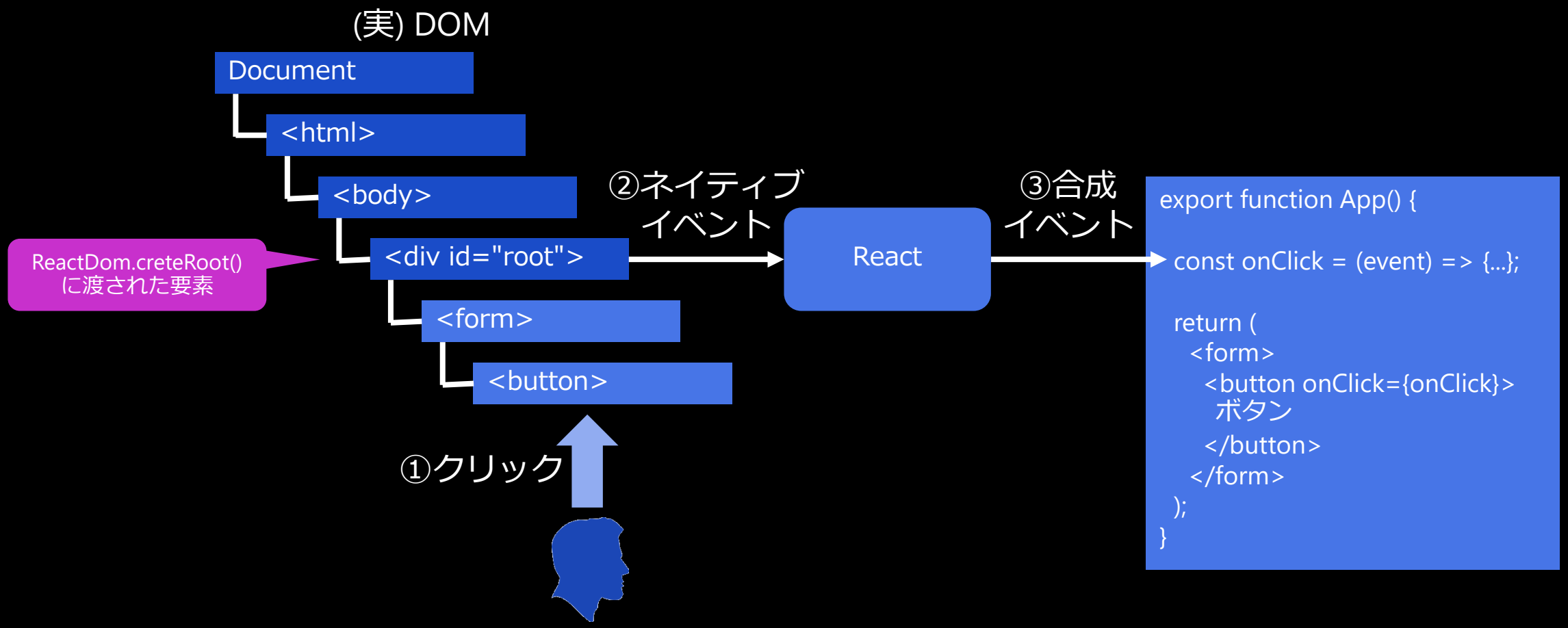
イベントハンドラ

- JSXでHTML要素に対してイベントハンドラを設定できる
  - onClick等の属性に関数を渡す
    - 例：<button onClick={(event) => {...}}>...</button>
  - キャプチャフェーズはイベント名の末尾にCaptureを付ける
    - 例：<button onClickCapture={(event) => {...}}>...</button>
- イベントハンドラはDOMに直接アタッチされない
  - Reactがイベントを受け取りコンポーネントのイベントハンドラを呼び出す
    - イベントオブジェクトはブラウザの違いを吸収したオブジェクトが渡される
    - 合成イベント (Synthetic Event) と呼ばれる
- 合成イベントで扱えるのはReactでレンダリングした要素のみ
  - Reactコンポーネントに対応しないDOMノードにイベントハンドラを設定するにはDOM APIを使う

属性名は
キャメルケースです

「React外リソースとの同期」
で説明します

合成イベント


(実) DOM
Document
<html>
<body>

<form>
<button>
ReactDom.creteRoot()
に渡された要素
②ネイティブ
イベント
React
③合成
イベント
①クリック
export function App() {
const onClick = (event) => {...};
return (
<form>
<button onClick={onClick}>
ボタン
</button>
</form>
);
}

演習(3-4): Todoアプリ

演習3-3のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[3-4] Todo
などに変更しよう

- TodoItemコンポーネントに削除ボタンを追加してみよう
  - 現段階ではタスクの削除はできないので、削除ボタンのイベントハンドラはタスクをコンソールに出力してみよう

ログはDevToolsに
出力されます

src/TodoItem.tsx

```
...

export function TodoItem(props: Props) {
  const onDeleteButton: xxx = () => {
    ...
  };

  return (
    <li>
      ...
      <button onClick={onDeleteButton}>削除</button>
    </li>
  );
}
```

xxxはonClickイベント
ハンドラの型

onClickの上にマウスカーソルを
重ねるとイベントハンドラの
型が表示されます

ReactとCSS

- React自体はCSSをサポートするための最小限の機能を提供
  - className属性
    - HTMLのclass属性に相当
    - 属性値はクラス名をスペース区切りで並べた文字列
      - 例: ...
    - 属性値を組み立てるためにclsxやclassnames等のnpmパッケージが使われる
      - 例: <div className={clsx('foo', flag && 'bar', 'baz')}>...
  - style属性
    - HTMLのstyle属性に相当
    - 属性値はJSのオブジェクトで指定する
      - オブジェクトのキーはCSSのプロパティ (キャメルケース)
      - 例: <div style={{ backgroundColor: "black" }}>...

ReactとCSSライブラリ・ツール

- 主なCSSの利用方法
  - CSS Modules
    - ローカルスコープを持つCSSファイルをJS/TSからimportして利用
    - Webpackやその他のツールで幅広くサポート
  - CSS-in-JS
    - JS/TS内でCSSをオブジェクトリテラルやテンプレートリテラルで記述
    - ランタイム系のCSS-in-JSライブラリ
      - 例: Emotion, styled-components, etc.
    - ゼロランタイム系のCSS-in-JSライブラリ
      - 例: Linaria, vanilla-extract, Panda CSS, StyleX
  - Tailwind
    - ユーティリティファーストのCSSフレームワーク
    - Tailwindが提供するCSSクラスをclassName属性で利用する

本研修ではこれらは
取り扱いません

コンポーネントとJSX まとめ

- コンポーネントはReactアプリの構成単位
  - 主に関数コンポーネントとして実装する
    - 引数としてPropsを受け取りReactElementを返す普通の関数
- コンポーネント内にJSXでHTML風のマークアップを書ける
  - JSXはReactElementを組み立てる式に変換される
  - JSXの中ではJSの式を使うことができる
    - 制御構文もJSの式を利用する
- 関数コンポーネントは仮想DOMで管理される情報 (Props, State, etc.) と関連付けられる
  - 関数コンポーネントはReactが管理する文脈の内側で呼び出されるイメージ

Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外リソースとの同期
- メモ化とパフォーマンス

コンポーネントの状態

- コンポーネントは状態 (State) を持つことができる
- 関数コンポーネントは単なる関数
  - 関数自身は状態を持っていない
- 状態はReactによって管理される
  - 状態はReactが管理する仮想DOMに保持される
  - 関数コンポーネントからReactが管理する情報 (状態を含む) とやり取りをするためにHooksを使う

Hooks

- Reactによって提供されるAPI (関数)
  - 特に「組込Hooks」と呼ばれる
    - 組込Hooksを利用したユーザ定義の関数は「カスタムHooks」と呼ばれる
  - 関数名がuse～で始まる
- 主な組込Hooks
  - 状態を扱うHooks
    - useState(), useReducer(), useRef()
  - 作用を扱うHooks
    - useEffect(), useLayoutEffect()
  - メモ化するHooks
    - useMemo(), useCallback()

他にも多数ありますが
本研修では扱いません

useState()

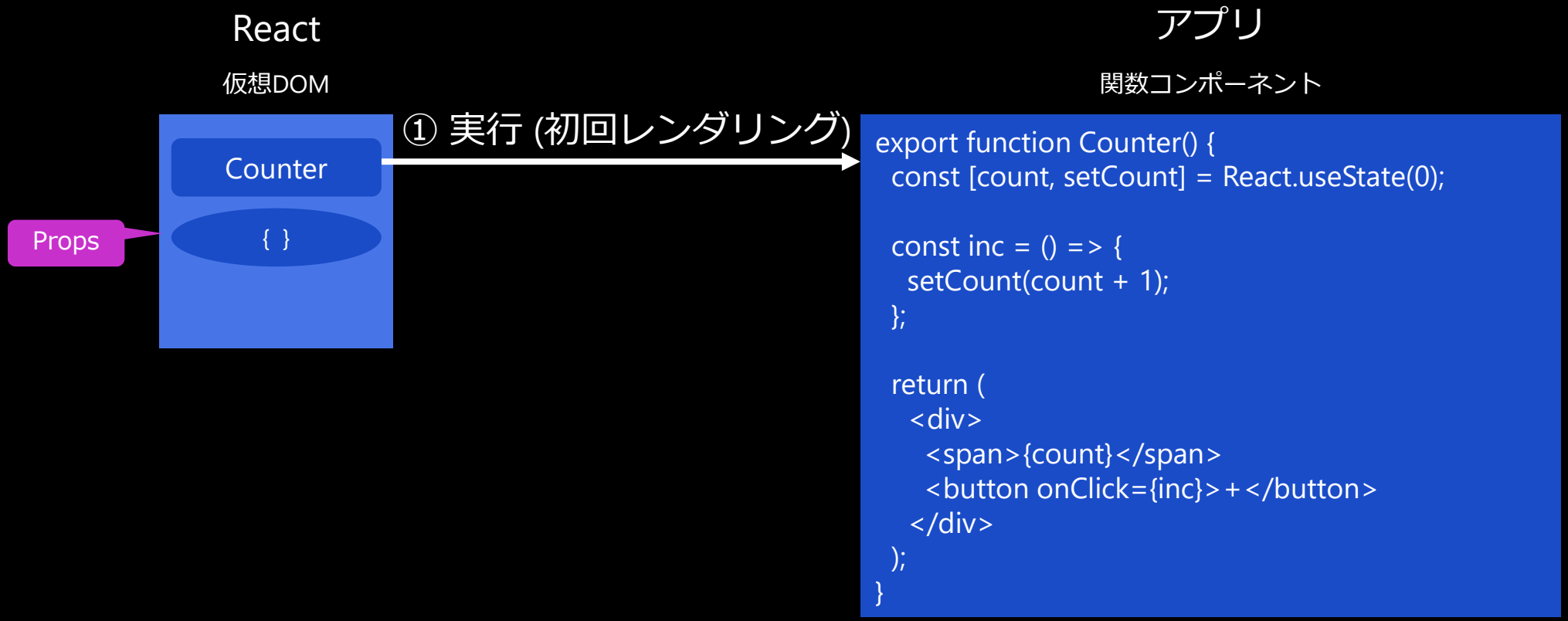
- 状態を扱う組込Hook
- 使い方: [state, setState] = useState(initialValue)
- 引数は状態の初期値または初期化関数
  - プリミティブ値に加えてオブジェクト (配列含む) も渡せる
- 戻り値は2要素の配列
  - 第1要素: 状態の現在の値
  - 第2要素: 状態を更新する関数
    - 引数は「新しい値」または「現在の値を受け取って新しい値を返す関数」
    - 状態の更新はキューイングされる (状態は直接更新されない)
    - 状態が実際に更新されると再レンダリングが発生する
    - オブジェクトや配列はイミュータブルとして扱うこと

例: Counterコンポーネント

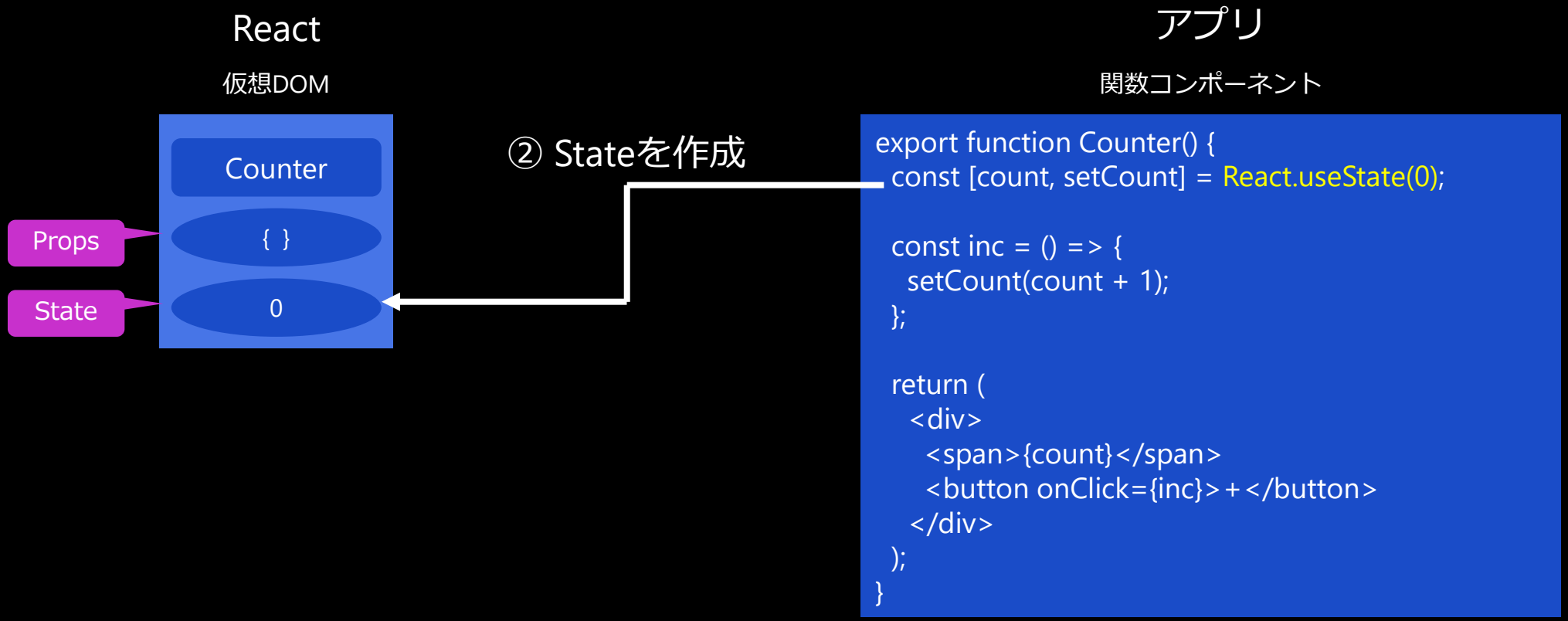
src/Counter.tsx

```
import React from "react";

export function Counter() {
  const [count, setCount] = React.useState(0);

  const inc = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <span>{count}</span>
      <button onClick={inc}>+</button>
    </div>
  );
}
```

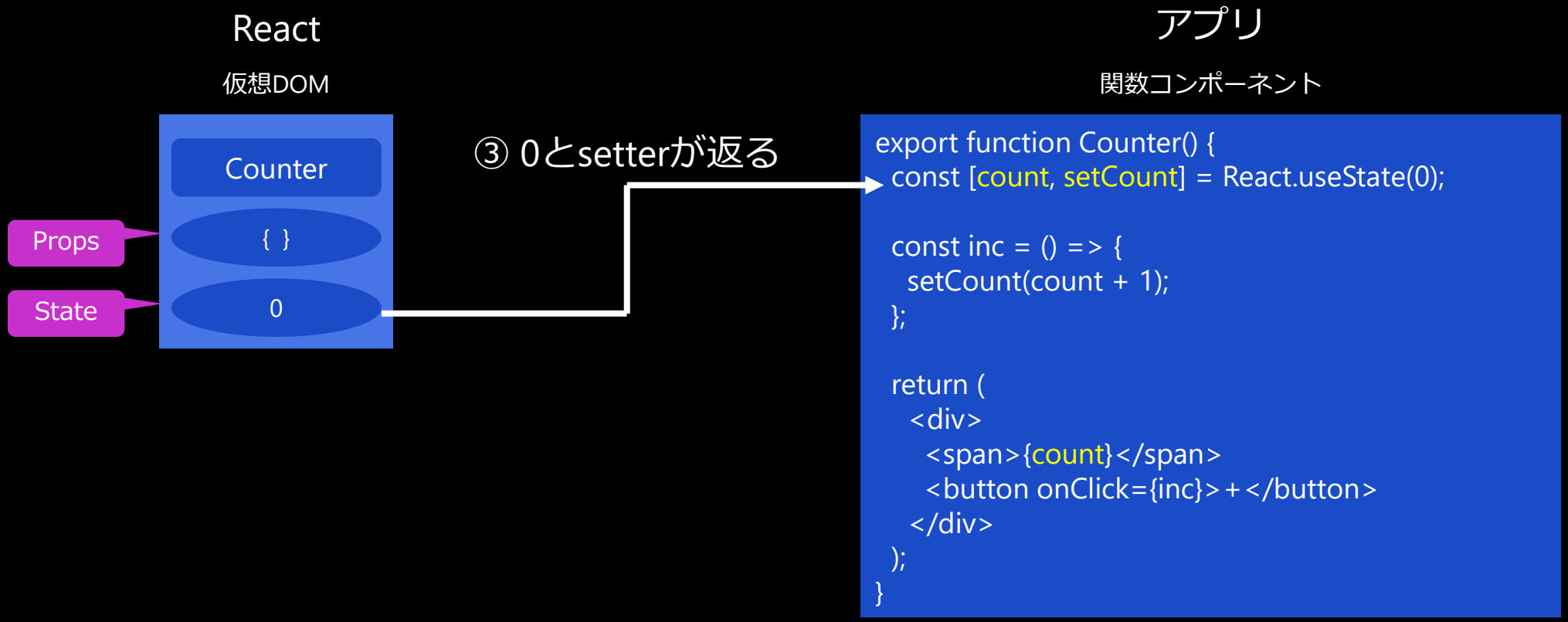
Counterコンポーネントの動作 (1)

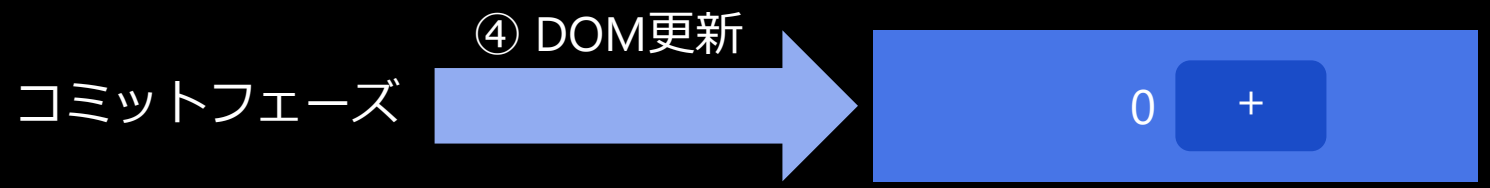


Counterコンポーネントの動作 (2)

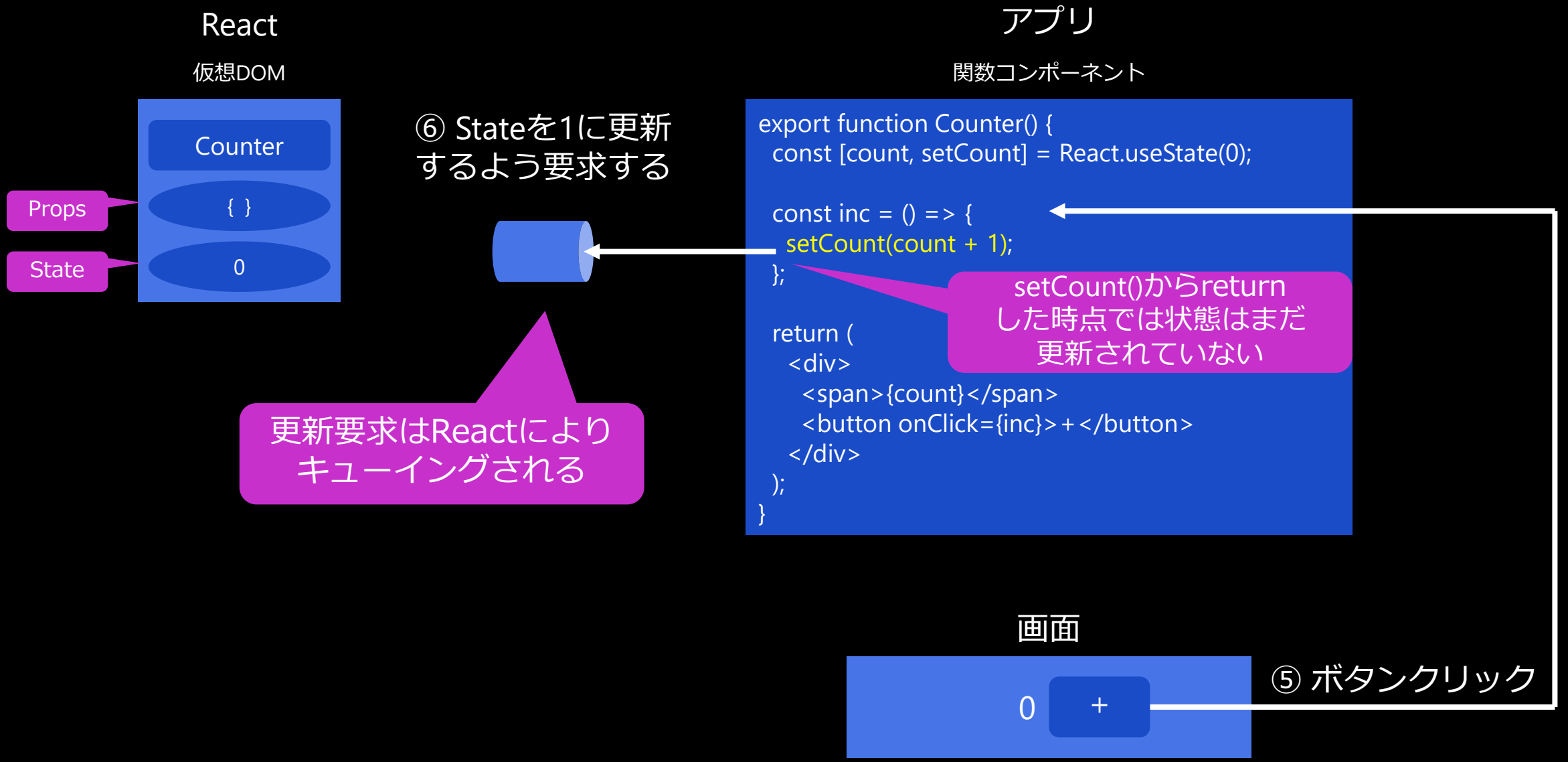


Counterコンポーネントの動作 (3)





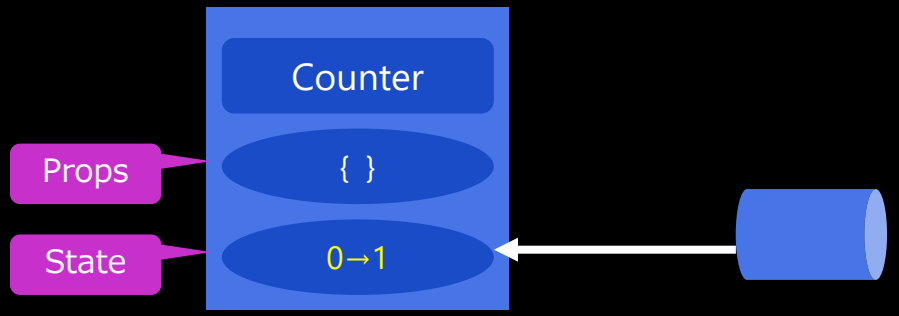
Counterコンポーネントの動作 (4)



Counterコンポーネントの動作 (5)

React

仮想DOM



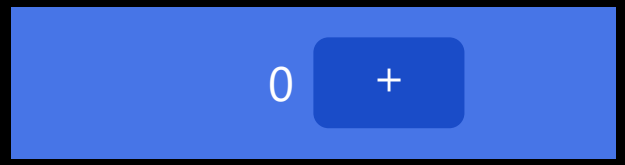
⑦ Stateを更新する

アプリ

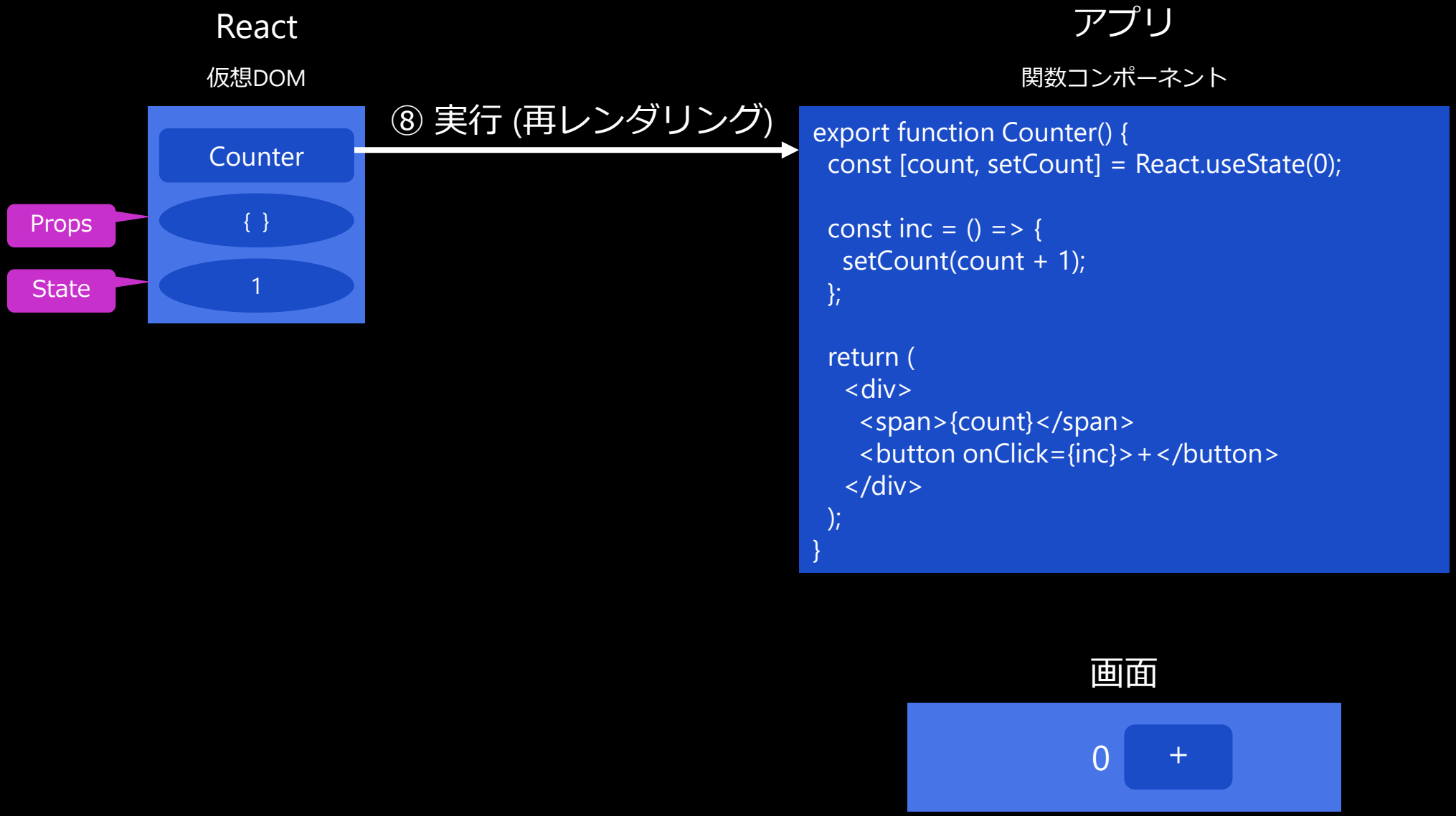
関数コンポーネント

```
export function Counter() {
  const [count, setCount] = React.useState(0);

  const inc = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <span>{count}</span>
      <button onClick={inc}>+</button>
    </div>
  );
}
```

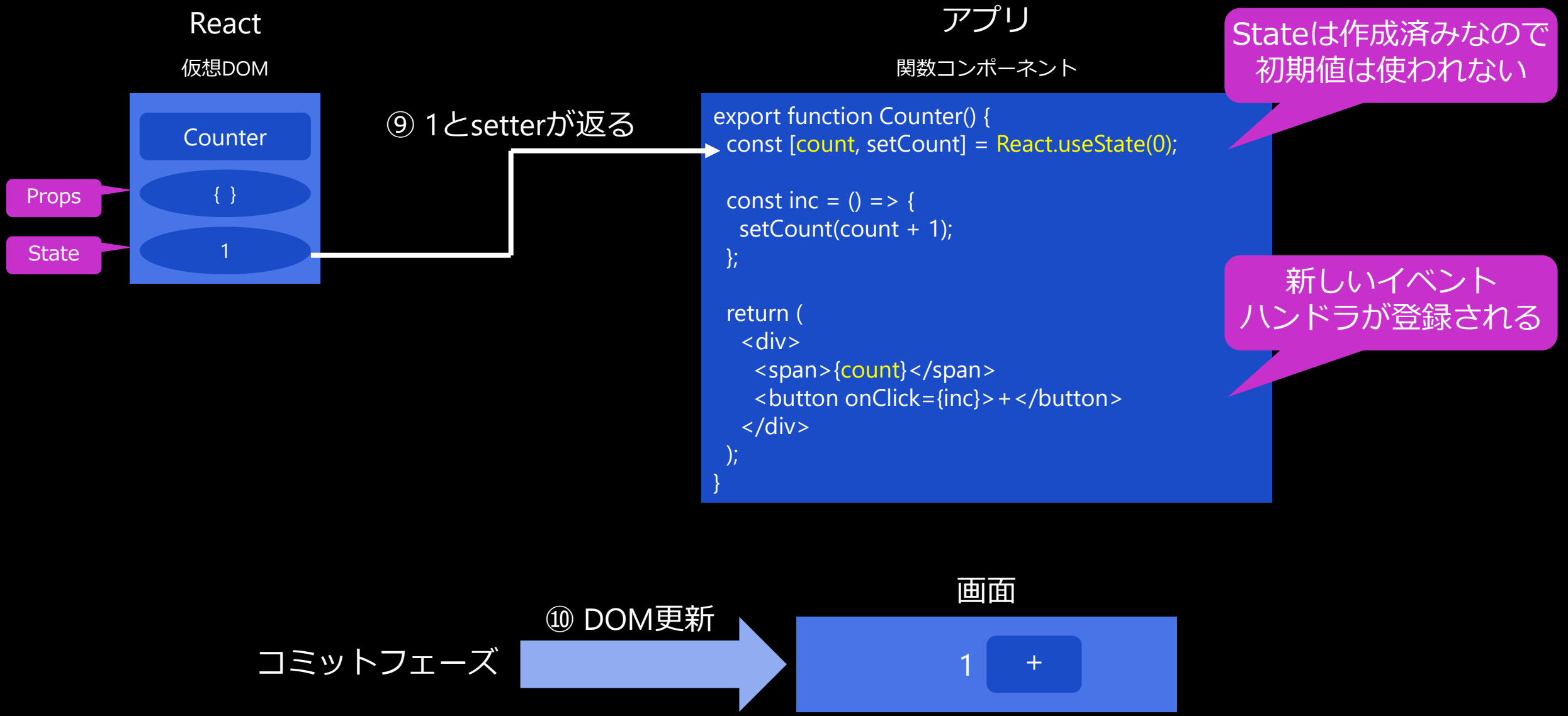
画面



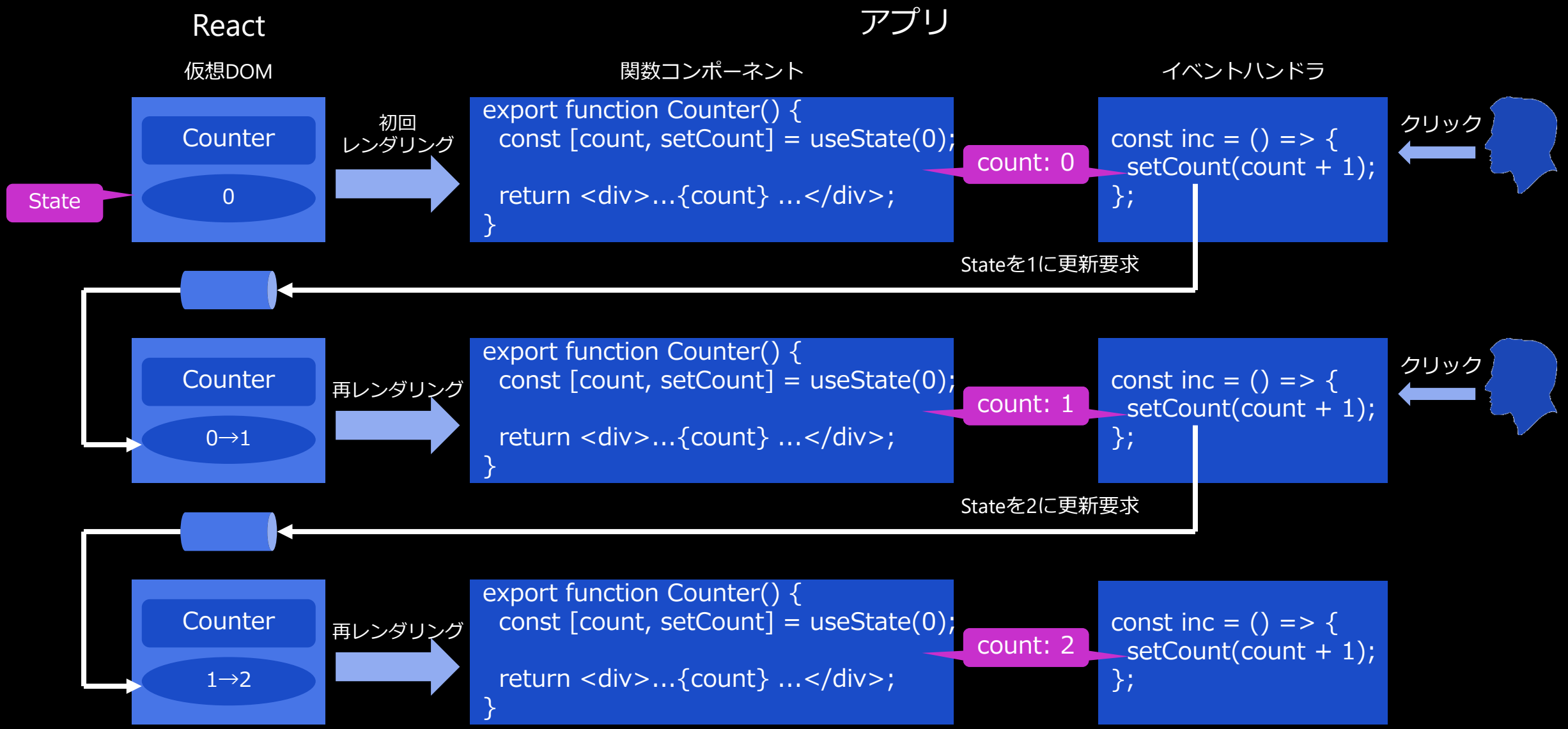
Counterコンポーネントの動作 (6)



Counterコンポーネントの動作 (7)



Counterコンポーネントの動作（まとめ）



再レンダリングと宣言的UI

- 状態が更新されるとコンポーネントは再レンダリングされる
  - 「リアクティブ (反応的)」とも呼ばれる
  - 状態が変更されたコンポーネントの子孫コンポーネントも再レンダリングされる
- 再レンダリングはコンポーネントを再実行する
  - 状態が変わるたびにコンポーネントツリーが再実行される
- 最初のレンダリング (新規表示) と再レンダリング (更新) を区別する必要が少ない
  - 現在の状態をどう画面に反映するかを考えるだけ → 宣言的UI
    - UI = f(State)

演習(4-1): Counterアプリ

Projectの名称を
[4-1] Counter
などに変更しよう

- Stackblitzの新しいProjectでCounterアプリを作ってみよう
- 「+」ボタンに加えて「-」「Reset」ボタンを追加しよう
- Appコンポーネントに Counterコンポーネントを複数置いて それぞれの状態が独立していることを確認してみよう
- 「+」ボタンのイベントハンドラにsetTimeout()を加えて ステータスの更新を遅延してみよう
  - 例: setTimeout(() => { setCount(count + 1) }, 2000)
  - タイムアウトする前にボタンを連打した場合の挙動を確認してみよう

状態とクロージャ

- JavaScriptの関数はクロージャ
  - イベントハンドラもクロージャ
- クロージャは外側の変数をキャプチャする
  - 例: イベントハンドラはuseState()の戻り値を代入した変数をキャプチャする
- 古い状態の変数をキャプチャしたイベントハンドラ(クロージャ) が動き続けることがある

Stale Closure と呼ばれます

JSのクロージャ

```
function f(m: number) {
  return (n: number) => m + n;
}
```

戻り値の関数はクロージャ
(外側の変数をキャプチャ)

```
const add1 = f(1);
add1(2); // 3
```

```
const add2 = f(2);
add2(2); // 4
```

```
add1 === add2 // false
```

ソース上では同じ関数だが
実行時は異なる関数オブジェクト

関数コンポーネント内のイベントハンドラも同様

関数コンポーネントとクロージャ

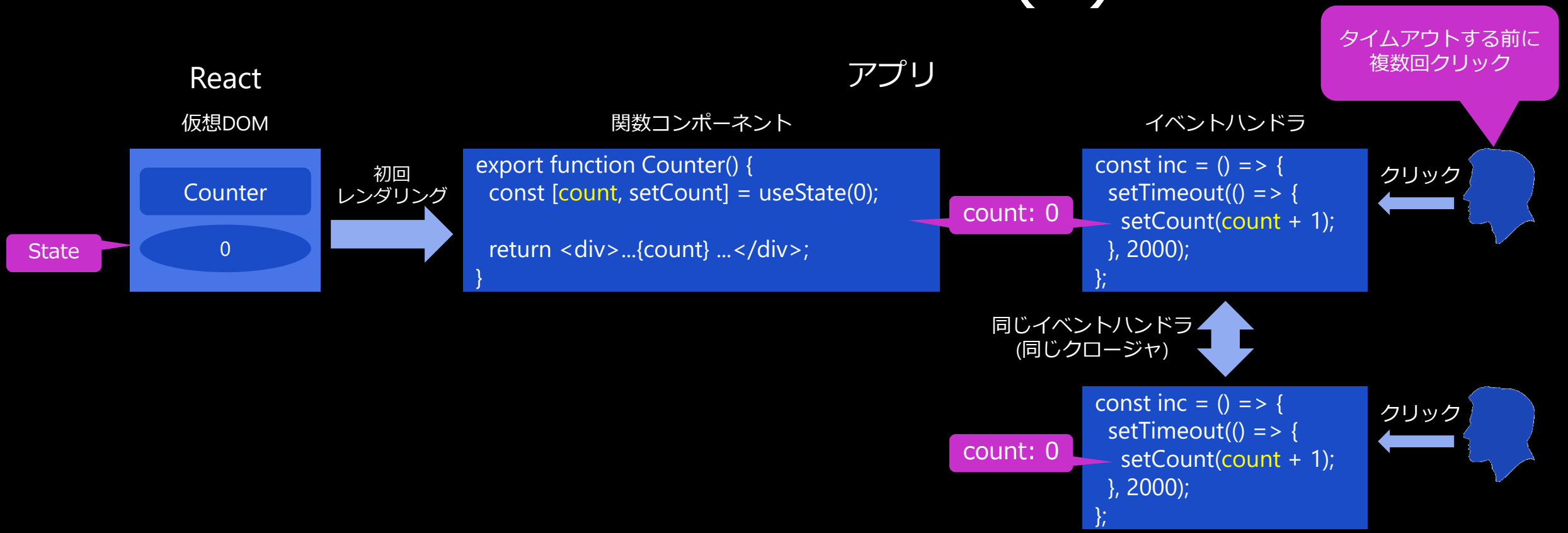
src/Counter.tsx

```
import React from "react";

export function Counter() {
  const [count, setCount] = React.useState(0);

  const inc = () => {
    setTimeout(() => setCount(count + 1), 5000);
  };

  return (
    <div>
      <span>{count}</span>
      <button onClick={inc}>+</button>
    </div>
  );
}
```

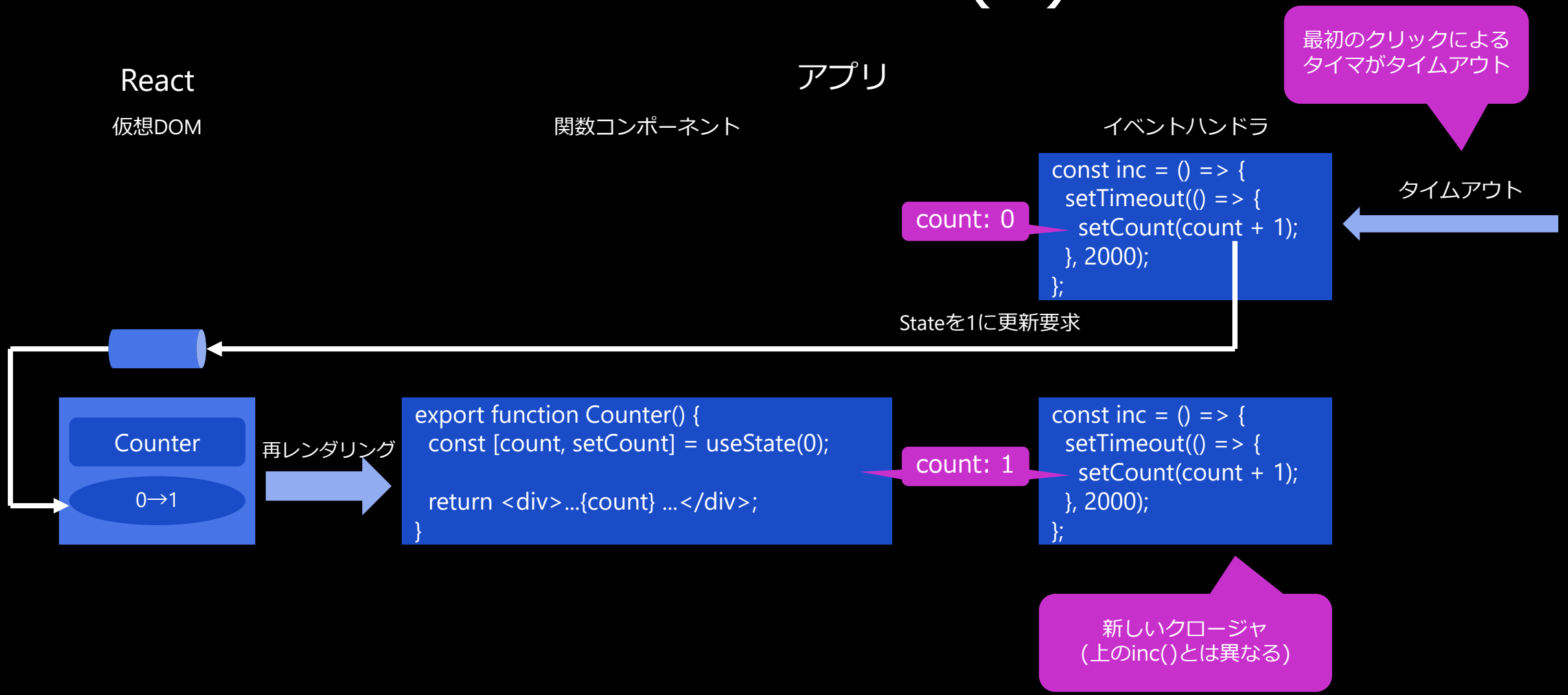
イベントハンドラもsetTimeout()に渡すコールバックもクロージャ(外側の変数をキャプチャ)

関数コンポーネントが実行されるたびに新しいイベントハンドラが設定される

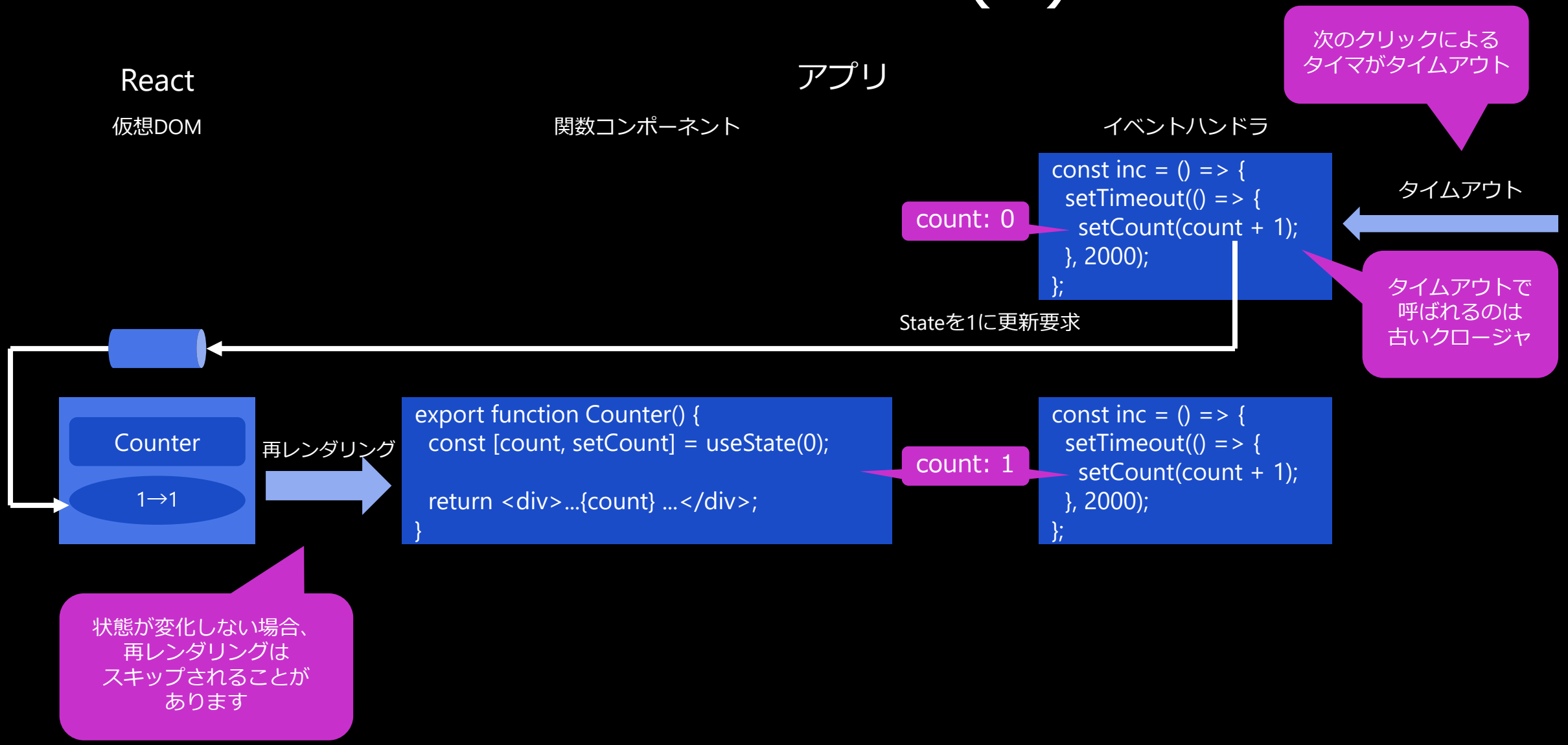
古いクロージャによる更新 (1)



古いクロージャによる更新 (2)



古いクロージャによる更新 (3)

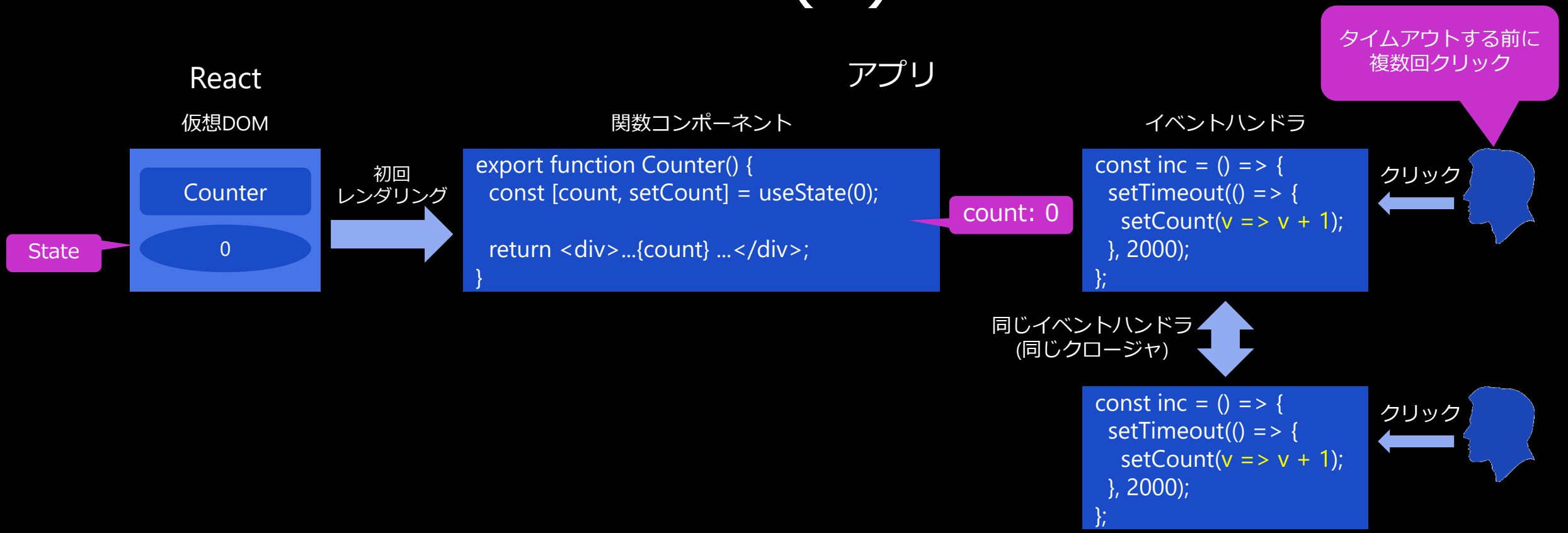


最新の状態に基づく更新

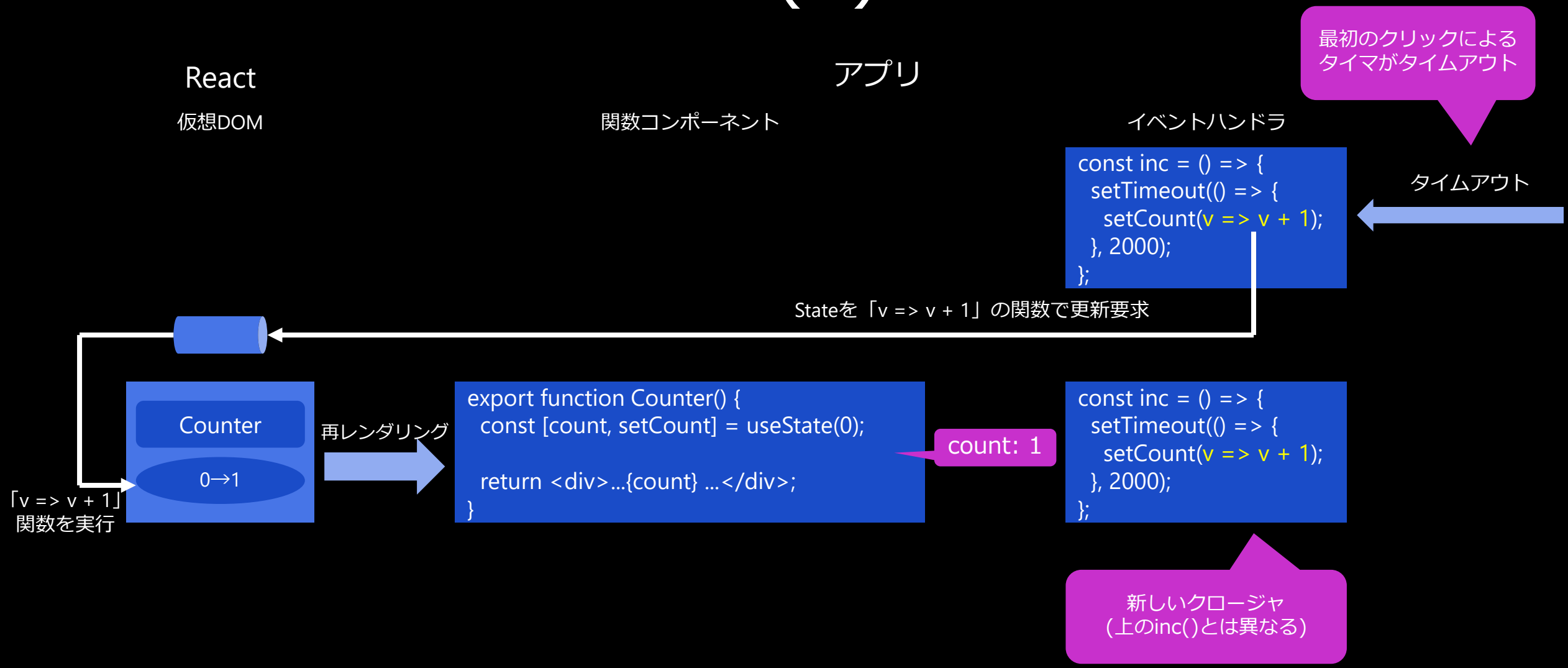
useState()が返す
配列の第2要素

- 「最新の状態に基づいて」更新を行うにはsetter関数の引数に値ではなく関数を渡す
  - 「現在の値を受け取って新しい値を返す」関数
  - 例: setState((currentValue) => currentValue + 1)

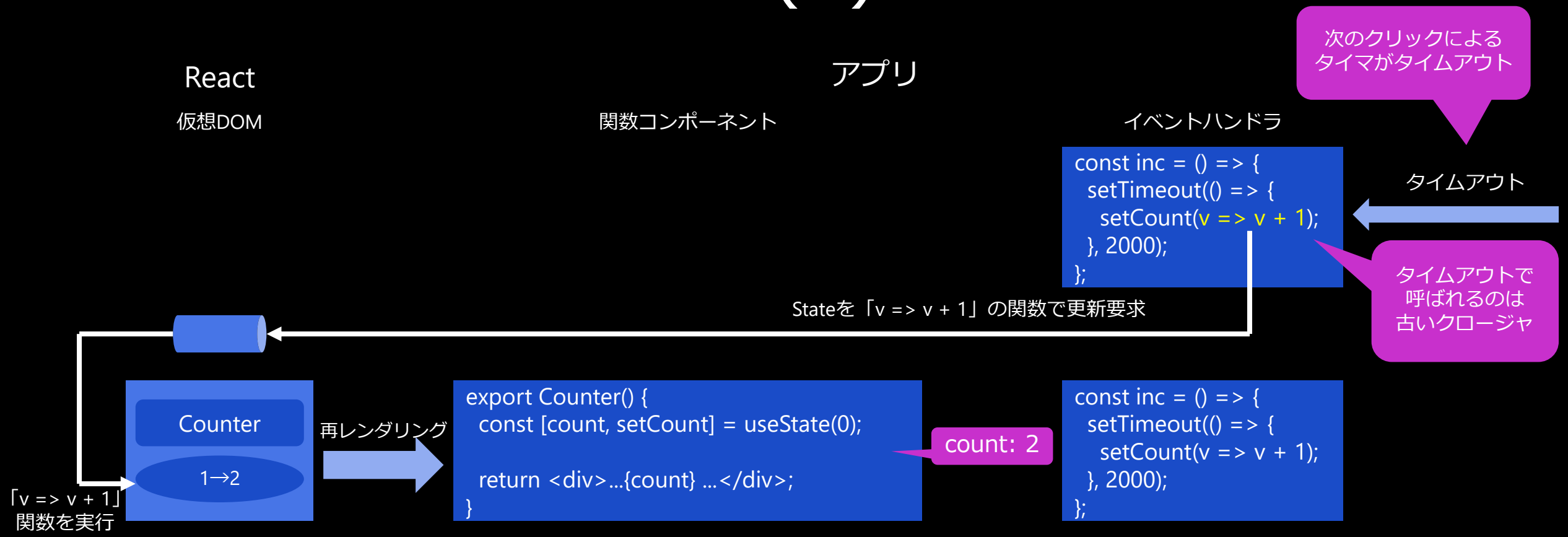
更新関数による更新 (1)


React
アプリ
仮想DOM
関数コンポーネント
イベントハンドラ
タイムアウトする前に
複数回クリック
Counter
0
State
初回
レンダリング
export function Counter() {
const [count, setCount] = useState(0);
return ...{count} ... ;
}
count: 0
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};
クリック
同じイベントハンドラ
(同じクロージャ)
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};
クリック

更新関数による更新 (2)


React
アプリ
仮想DOM
関数コンポーネント
イベントハンドラ
最初のクリックによる
タイマがタイムアウト
タイムアウト
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};
Stateを「v => v + 1」の関数で更新要求
Counter
0→1
「v => v + 1」
関数を実行
再レンダリング
export function Counter() {
const [count, setCount] = useState(0);
return ...{count} ... ;
}
count: 1
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};
新しいクロージャ
(上のinc()とは異なる)

更新関数による更新 (3)


React
仮想DOM
アプリ
関数コンポーネント
イベントハンドラ
次のクリックによる
タイマがタイムアウト
タイムアウト
タイムアウトで
呼ばれるのは
古いクロージャ
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};
Stateを「v => v + 1」の関数で更新要求
Counter
1→2
「v => v + 1」
関数を実行
再レンダリング
export Counter() {
const [count, setCount] = useState(0);
return ...{count} ... ;
}
count: 2
const inc = () => {
setTimeout(() => {
setCount(v => v + 1);
}, 2000);
};

演習(4-2): Counterアプリ

演習4-1のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[4-2] Counter
などに変更しよう

- 「+」「-」ボタンのイベントハンドラでsetterに更新関数を使ってみよう
  - 「+」ボタンのイベントハンドラにsetTimeout()を加えてタイムアウトする前に「+」ボタンを連打した場合の挙動を確認してみよう

複数のuseState()

- 関数コンポーネント内ではuseState()を何度でも呼び出せる
  - それぞれのstateは独立に更新できる
  - stateが1つでも更新されるとそのコンポーネントは再レンダリングされる
  - 複数のstateが同時に更新されても再レンダリングは1回だけ
    - 複数のstateが近いタイミングで更新された場合でも再レンダリングは1回にまとめられる場合がある (自動バッチ更新)

例: Userコンポーネント

src/User.tsx

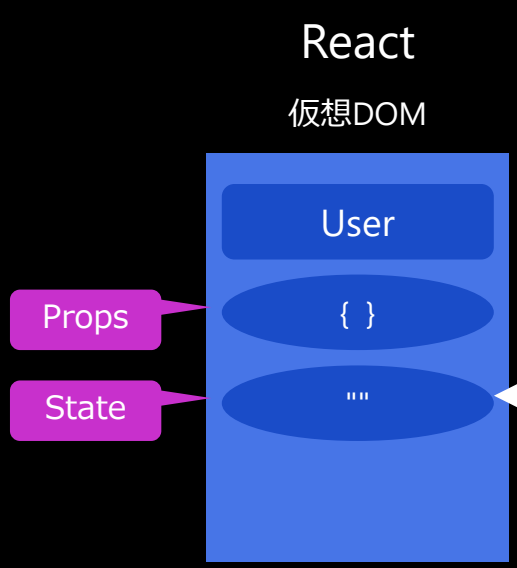
```
import React from "react";

export function User() {
  const [userName, setUserName] = React.useState("");
  const [birthday, setBirthday] = React.useState("");

  const handleChangeUserName: React.ChangeEventHandler<HTMLInputElement> = (event) => {
    setUserName(event.currentTarget.value);
  };
  const handleChangeBirthday: React.ChangeEventHandler<HTMLInputElement> = (event) => {
    setBirthday(event.currentTarget.value);
  };
   const handleClickReset: React.MouseEventHandler<HTMLButtonElement> = () => {
    setUserName("");
    setBirthday("");
  }

  return (
    <div>
      <label>名前<input type="text" value={userName} onChange={handleChangeUserName} /></label>
      <label>生年月日<input type="date" value={birthday} onChange={handleChangeBirthday} /></label>
      <button onClick={handleClickReset}>リセット</button>
    </div>
  );
}
```

複数のuseState()

複数のstateを更新

Userコンポーネントの動作 (1)

Userコンポーネントの動作 (2)



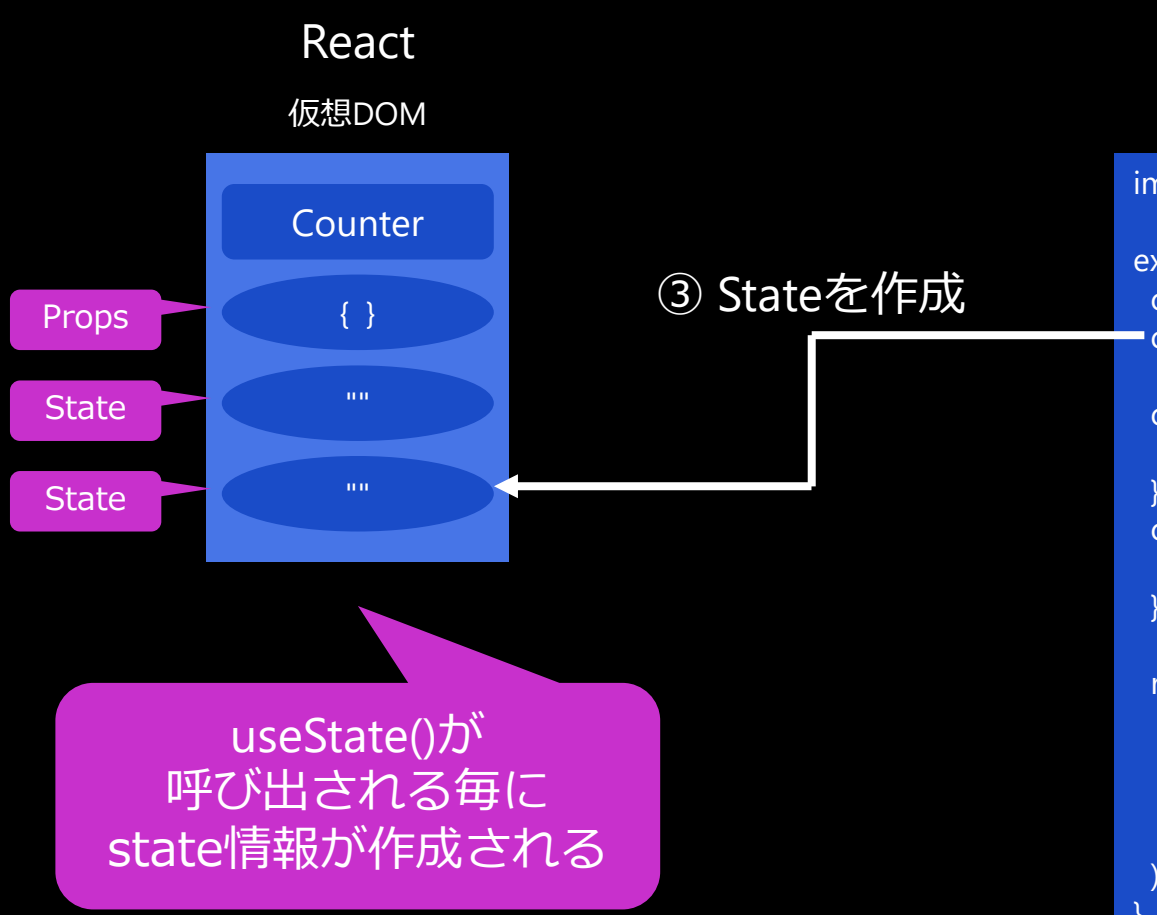
② Stateを作成

アプリ

関数コンポーネント

```
import React from "react";

export function User() {
  const [userName, setUserName] = React.useState("");
  const [birthday, setBirthday] = React.useState("");

  const handleChangeUserName = (event) => {
    setUserName(event.currentTarget.value);
  };
  const handleChangeBirthday = (event) => {
    setAgreement(event.currentTarget.value);
  };

  return (
    <div>
      <input type="text" value={userName} onChange={handleChangeUserName} />
      <input type="date" value={birthday} onChange={handleChangeBirthday} />
    </div>
  );
}
```

Userコンポーネントの動作 (3)



アプリ

関数コンポーネント

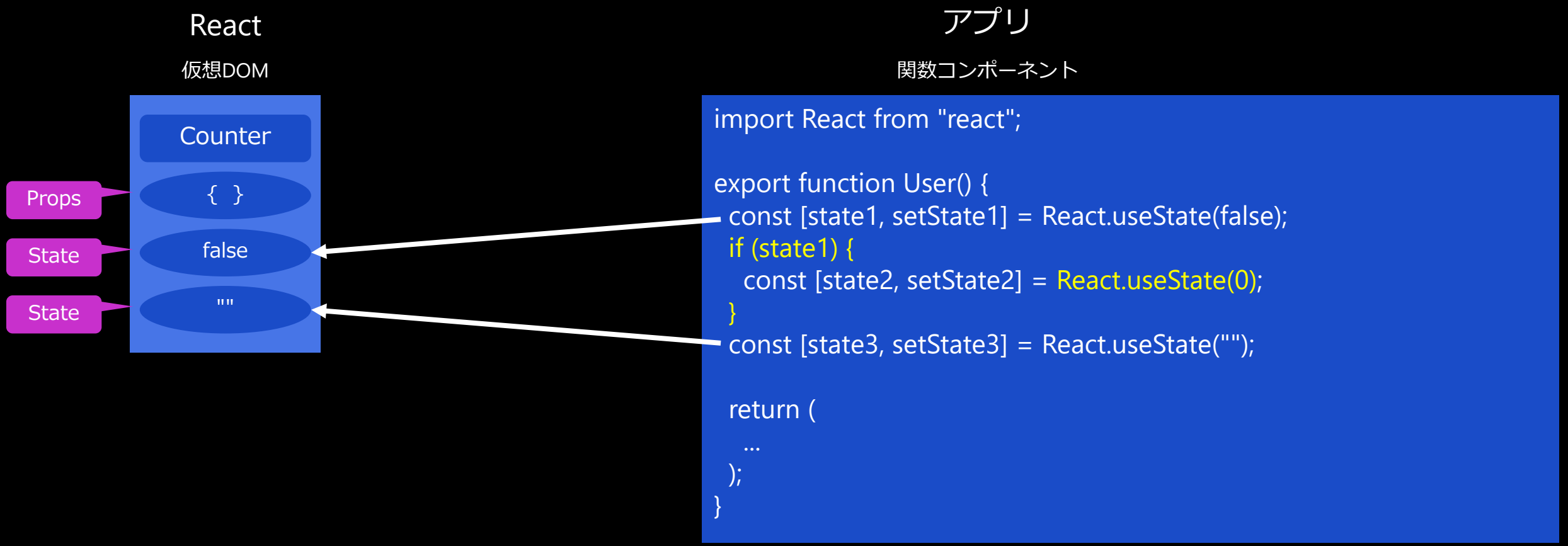
```
import React from "react";

export function User() {
  const [userName, setUserName] = React.useState("");
  const [birthday, setBirthday] = React.useState("");

  const handleChangeUserName = (event) => {
    setUserName(event.currentTarget.value);
  };
  const handleChangeBirthday = (event) => {
    setAgreement(event.currentTarget.value);
  };

  return (
    <div>
      <input type="text" value={userName} onChange={handleChangeUserName} />
      <input type="date" value={birthday} onChange={handleChangeBirthday} />
    </div>
  );
}
```

Hooksのルール

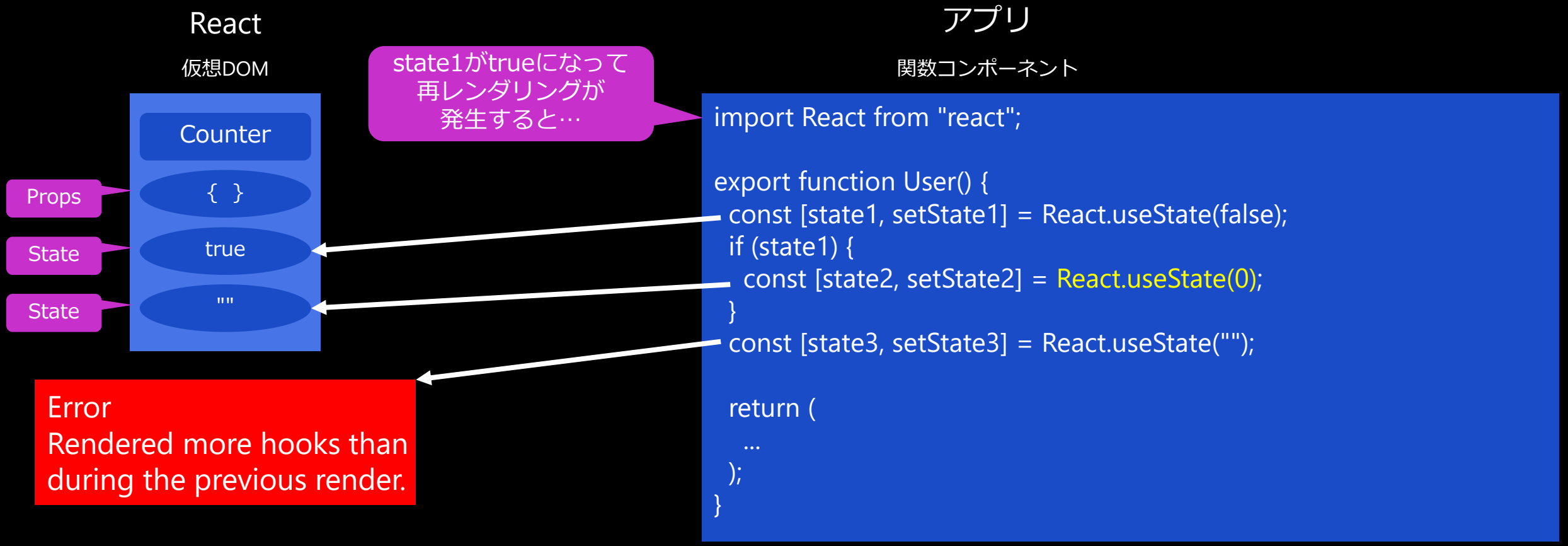
useState()に限らず
Hook全般のルールです

- Reactは組込Hookが呼び出されるたび、仮想DOMに組込Hookごとの情報を追加する
  - 単純な連結リストとして管理される
- 関数コンポーネントは常に同じ順番で同じ数のHooksを呼び出さなくてはならない
  - 組込Hooksだけではなくカスタムも同様
  - 関数コンポーネントのトップレベルでのみHooksを呼び出す
    - if文の中や条件式の中、繰り返しの中からHooksを呼び出さない
  - eslint-plugin-react-hooksでチェックする

間違ったHooksの使い方 (1)



間違ったHooksの使い方 (2)



useState()のよくない使い方

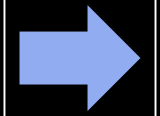
- プリミティブ値ごとにuseState()を呼び出す
  - 関連のある情報はオブジェクトや配列にまとめる
    - 関連の薄い情報まで無理にまとめる必要はない
- 導出値にuseState()を使う
  - 導出値は通常の変数に保存する
  - 導出する処理が重い場合はメモ化する

「メモ化と
パフォーマンス」で
説明します

例：useState()の使い方

よくない例

```
import React from "react";

export function User() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");
  const [fullName, setFullName] = useState("");

  const handleChangeFirstName = (event) => {
    setFirstName(event.target.value);
    setFullName(`${event.target.value} ${lastName}`);
  };
  const handleChangeLastName = (event) => {
    setLastName(event.target.value);
    setFullName(`${firstName} ${event.target.value}`);
  };

  return (
    ...
  );
}
```



改善例

```
import React from "react";

export function User() {
  const [user, setUser] = useState({
    firstName: "",
    lastName: "",
  });
  const fullName = `${user.firstName} ${user.lastName}`;

  const handleChangeFirstName = (event) => {
    setUser((user) => ({ ...user, firstName: event.target.value }));
  };
  const handleChangeLastName = (event) => {
    setUser((user) => ({ ...user, lastName: event.target.value }));
  };

  return (
    ...
  );
}
```

関連のあるstateはまとめる

導出値にstateは使わない

イミュータブルな更新

イミュータブルな更新

- オブジェクト (配列を含む) は直接更新しない
  - プロパティや要素の書き換え、追加・削除はしない
- 変更後のプロパティや要素を持つ新しいオブジェクトを作る
  - イミュータブルなマナーに従う
- オブジェクトのイミュータブルな更新
  - 例: { ...oldObject, foo: newFoo, bar: newBar }
- 配列のイミュータブルな更新
  - 要素の追加: [...oldArray, newItem]
  - 要素の置換: Array.prototype.map()
    - または: Array.prototype.with() // ES2023

フォームと制御コンポーネント

- Reactで入力要素等を扱う方法
  - 制御コンポーネント (Controlled Component)
    - React-wayな方法 (宣言的) で入力要素等を扱う
  - 非制御コンポーネント (Uncontrolled Component)
    - React-wayではない方法 (命令的) で入力要素等を扱う
- 制御コンポーネント
  - 入力要素等の状態はReactコンポーネントが管理する
    - useState()/useReducer()を使う
  - Reactコンポーネントの状態更新による再レンダリングで入力要素等が更新される
    - Reactコンポーネントの状態が更新されなければ入力要素等は更新されない

「パフォーマンスとメモ化」で扱います

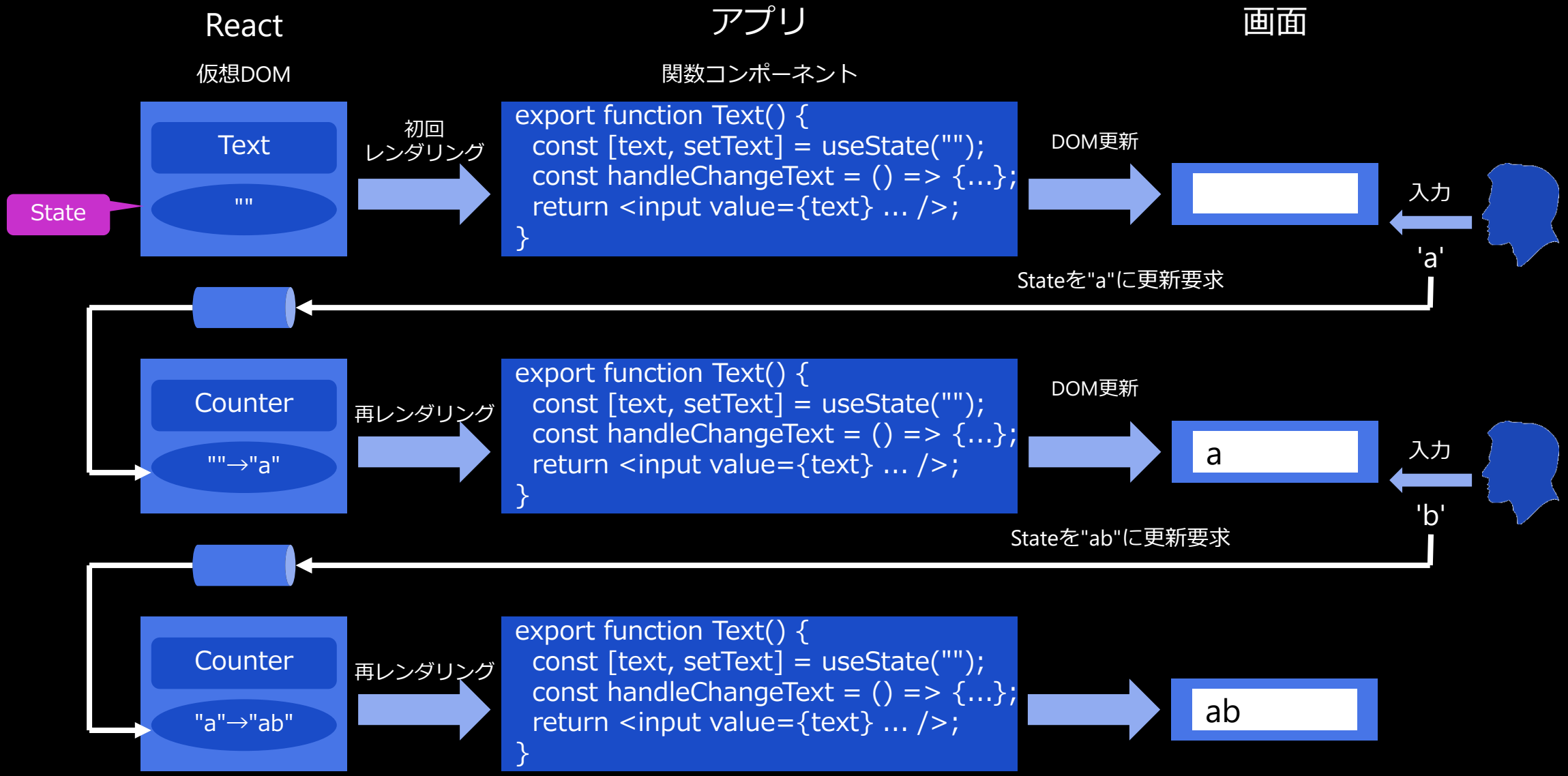
入力要素等と制御コンポーネント

```
import React from "react";

export function ReadonlyText() {
  const text = "Foo";
  const handleTextChange: React.ChangeEventHandler<HTMLInputElement> = () => {
  };

  return <input type="text" value={text} onChange={handleTextChange} />;
}

export function WritableText() {
  const [text, setText] = React.useState("Foo");
  const handleTextChange: React.ChangeEventHandler<HTMLInputElement> = (event) => {
    setText(event.currentTarget.value);
  };

  return <input type="text" value={text} onChange={handleTextChange} />;
}
```

テキストフィールドに
入力しても反映されない

テキストフィールドに
入力すると反映される

入力要素等と制御コンポーネント



フォームとイベントハンドラ (1)

- フォーム
  - onSubmitイベントハンドラでフォームサブミット時のイベントを扱う
  - イベントハンドラではevent.preventDefault()を呼び出す
  - 例: <form onSumbit={handleSubmit} >...</form>
- ボタン
  - onClickイベントハンドラでボタン押下時のイベントを扱う
  - disabled属性 (boolean) で有効/無効を指定する
  - 例: <button onClick={handleClick} disabled={flag}>ボタン</button>

onSubmitを使わない方法も増えています

フォームとイベントハンドラ (2)

- テキストフィールド
  - value属性 (string) でテキストを指定
  - onChangeイベントハンドラでテキスト入力時のイベントを扱う
    - event.currentTarget.value (string)で入力値を取得
  - 例: <input type="text" value={text} onChange={handleChange} />
- テキストエリア
  - value属性 (string) でテキストを指定
  - onChangeイベントハンドラでテキスト入力時のイベントを扱う
    - event.currentTarget.value (string)で入力値を取得
  - 例: <textarea value={text} onChange={handleChange}></textarea>

通常のHTMLとは異なります

フォームとイベントハンドラ (3)

- チェックボックス
  - checked属性 (boolean) で選択・未選択を指定
  - onChangeイベントハンドラで選択状態変更時のイベントを扱う
    - event.currentTarget.checked (boolean)で選択状態を取得
  - 例: <input type="checkbox" name="option" value={1}
    checked={checked} onChange={handleChange} />

フォームとイベントハンドラ (4)

- ラジオボタン
  - checked属性 (boolean) で選択・未選択を設定
  - onChangeイベントハンドラで選択状態変更時のイベントを扱う
    - event.currentTarget.valueに選択されたラジオボタンのvalue属性値が入る
  - 例:

```
const colors = [{ label: "赤", value: "red" }, { label: "緑", value: "green" }, { label: "青", value: "blue" }];

function ColorForm() {
  const [selected, setSelected] = React.useState("");
  const handleChange: React.ChangeEventHandler<HTMLInputElement> = (event) => setSelected(event.currentTarget.value);

  return (
    <form>
      {colors.map((color) => (
        <label>{color.label}
          <input type="radio" name="color" value={color.value} checked={color.value === selected} onChange={handleChange} />
        </label>
      ))}
    </form>
  );
}
```

フォームとイベントハンドラ (5)

- 選択リスト
  - <select>要素のvalue属性 (stringまたはstring[]) で選択中の<option>要素を指定
  - onChangeイベントハンドラで選択状態変更時のイベントを扱う
    - event.currentTarget.valueに選択された<option>要素のvalue属性値が入る
  - 例:

通常のHTMLとは異なります

```
const colors = [{ label: "赤", value: "red" }, { label: "緑", value: "green" }, { label: "青", value: "blue" }];

function ColorForm() {
  const [selected, setSelected] = React.useState("");
  const handleChange: React.ChangeEventHandler<HTMLSelectElement> = (event) => setSelected(event.currentTarget.value);

  return (
     <select value={selected} onChange={handleChange}>
     {colors.map((color) => (
        <option value={color.value}>{color.label}</option>
     ))}
    </select>
  );
}
```

演習(4-3): Todoアプリ

演習3-4のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[4-3] Todo
などに変更しよう

- テキストフィールドと「追加」ボタンで新しいTodoを追加できるようにしよう
  - テキストフィールドが未入力なら「追加」ボタンを押せないようにしよう
- テキストフィールド内で「Enter」キーを押すだけでTodoを追加できるようにしよう
- Todoの完了/未完了を変更できるようにしよう
- Todoを削除できるようにしよう
- 完了/未完了のTodoをフィルタリングできるようにしよう
  - 全て/完了/未完了の選択リストを追加してみよう

useReducer()

- useState()の課題
  - Stateをどのように更新するかがイベントハンドラに散らばりやすい
- useReducer()
  - Stateの更新処理を「Reducer」関数に集約できる組込Hook
  - ActionとReducerを使ってStateを更新する
    - Action
      - Stateをどのように更新するか指示するもの (通常はオブジェクト)
    - Reducer
      - 現在のStateとActionを受け取って新しいStateを返す関数
        - Array.prototype.reduce()に渡す関数と同様

useReducer()

- 使い方: [state, dispatch] = useReducer(reducer, initialState)
- 引数でReducerとStateの初期値を渡す
  - Reducer
    - 「現在のState」と「Action」を引数で受け取り「新しいState」を返す関数
      - (state, action) => state
- 戻り値は配列
  - 1番目の要素は現在のState
  - 2番目の要素はStateの更新を要求するための関数
    - 引数にActionを渡して呼び出すと現在のステートと共にReducerに渡されStateが更新される
    - Stateの更新はuseState()と同様にキューイングされる

例：useRecuder()版のCounter

Action

```
// Action types
type Inc = {
  type: "Inc";
  step: number;
};

type Dec = {
  type: "Dec";
  step: number;
};

type Reset = {
  type: "Reset";
  value: number;
};

type Action = Inc | Dec | Reset;

// Action creators
const inc: (step?: number) => Inc =
  (step = 1) => ({ type: "Inc", step });

const dec: (step?: number) => Dec =
  (step = 1) => ({ type: "Dec", step });

const reset: (value?: number) => Reset =
  (value = 0) => ({ type: "Reset",  value });
```

Reducer

```
const reducer = (state: number, action: Action): number => {
  switch (action.type) {
    case "Inc":
      return state + action.step;
    case "Dec":
      return state - action.step;
    case "Reset":
      return action.value;
  }
};
```

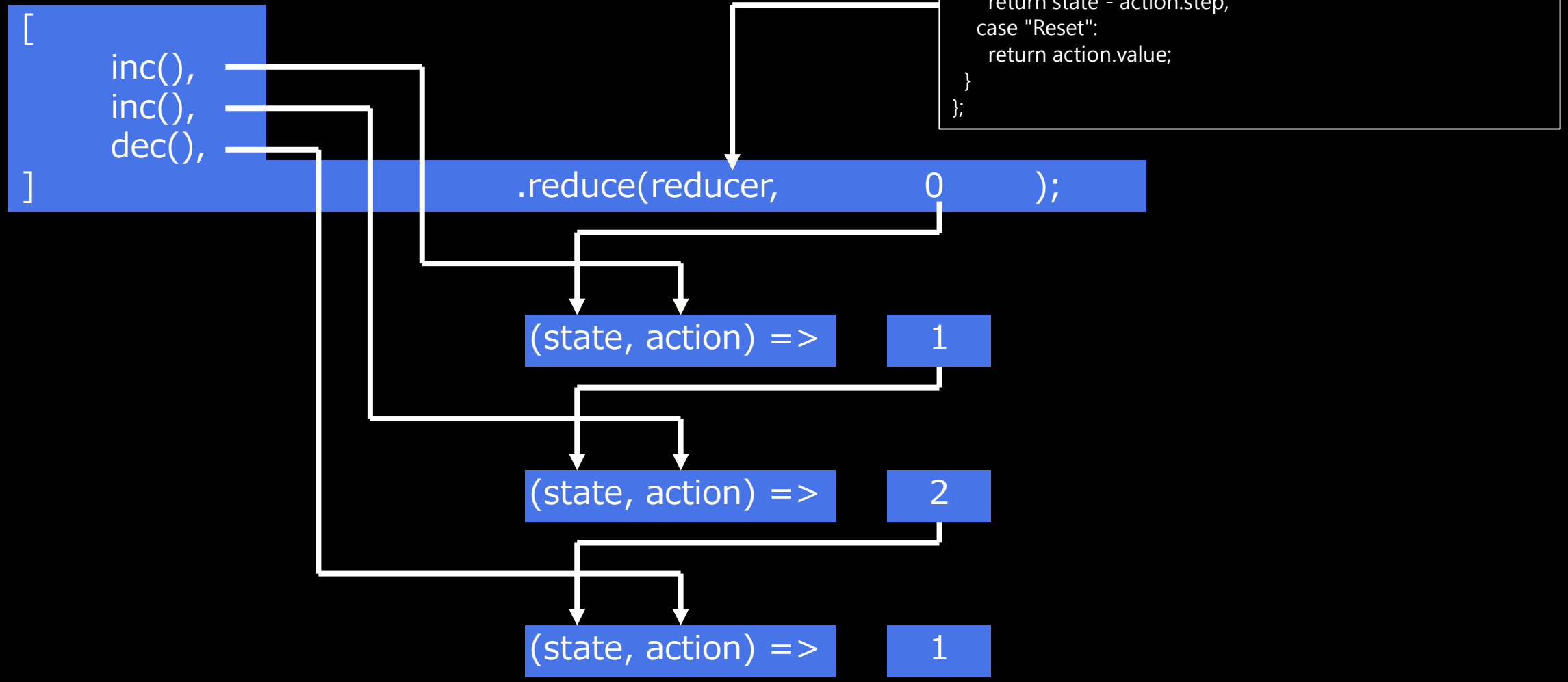
Counterコンポーネント

```
export Counter() {
  const [count, dispatch] = React.useReducer(reducer, 0);

  return (
    <div>
      <span>{count}</span>
      <button onClick={() => dispatch(inc())}>+</button>
      <button onClick={() => dispatch(dec())}>-</button>
      <button onClick={() => dispatch(reset())}>Reset</button>
    </div>
  );
}
```

参考: Array.reduce()

Reducer

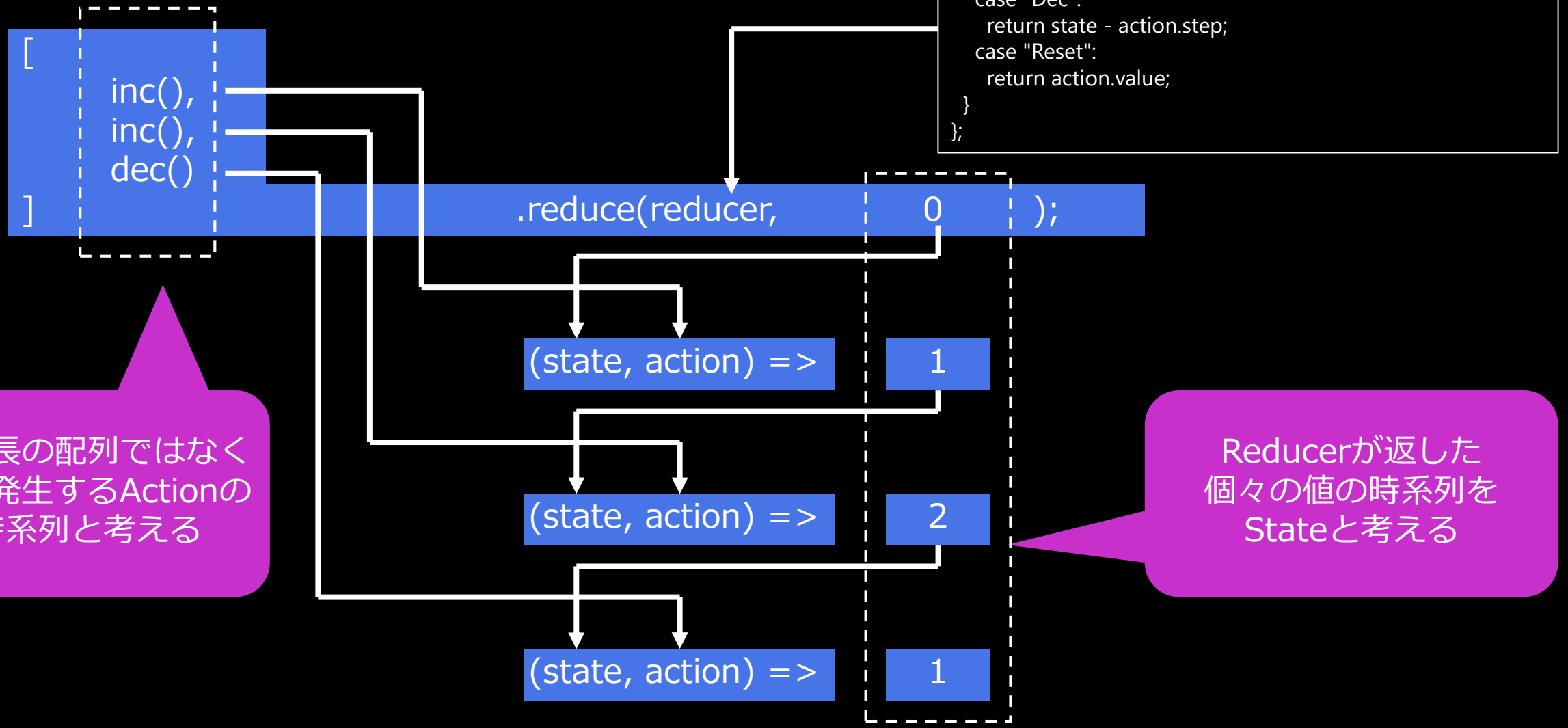
```
const reducer = (state: number, action: Action): number => {
  switch (action.type) {
    case "Inc":
      return state + action.step;
    case "Dec":
      return state - action.step;
    case "Reset":
      return action.value;
  }
};
```



useReducer()の考え方

Reducer

```
const reducer = (state: number, action: Action): number => {
  switch (action.type) {
    case "Inc":
      return state + action.step;
    case "Dec":
      return state - action.step;
    case "Reset":
      return action.value;
  }
};
```



演習(4-4): Todoアプリ(Reducer版)

演習4-3のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[4-4] Todo
などに変更しよう

- TodoアプリのuseState()をuseReducer()に置き換えてみよう
- useState()とuseReducer()の使い分けについて考えてみよう
  - useState()が向いているStete
  - useReducer()が向いているStete

useRef()

- 再レンダリングを引き起こさない状態を扱う組込Hook
- 使い方: ref = useRef(initialValue)
  - 引数は戻り値となるRefオブジェクトのcurrentプロパティの初期値
  - 戻り値はcurrentプロパティを持つRefオブジェクト
    - レンダリング毎に常に同じオブジェクトが返される
- Refオブジェクト
  - currentプロパティに任意の値を設定できる
  - useState()/useReducer()が返す状態と異なり直接更新することができる
    - イミュータブルなマナーに従う必要はない
    - 更新はキューイングされない

useRef()の用途

- OOPにおけるインスタンス変数の代わりに使用する
  - Reactが管理する仮想DOM上の情報をインスタンスとみなし、インスタンス固有の情報を持たせる
  - 必要なことも多いが命令的になりがちなのでできるだけ避ける
- DOM要素の参照を取得する
  - ホストコンポーネントのref属性にRefオブジェクトを渡すとコミットフェーズで対応するDOM要素の参照がRefオブジェクトのcurrentプロパティに設定される

useRef()の例 (インスタンス変数的)

src/RefCounter.tsx

```
import React from "react";

export function RefCounter() => {
  const [stateCount, setStateCount] = React.useState(0);
  const refCount = React.useRef(0);

  const handleClickStateCount = () => {
    setStateCount((v) => v + 1);
  };
  const handleClickRefCount = () => {
    refCount.current++;
  };

  // →へ続く
```

```
return (
    <div>
      <div>
        <div>State Counter</div>
        <div>
          <span>{stateCount}</span>
          <button onClick={handleClickStateCount}>+</button>
        </div>
      </div>
      <div>
        <div>Ref Counter</div>
        <div>
          <span>{refCount.current}</span>
          <button onClick={handleClickRefCount}>+</button>
        </div>
      </div>
    </div>
  );
}
```

useRef()によるDOM要素との連携

- ホストコンポーネントに対応するDOM要素の参照を取得できる
  - ホストコンポーネントのref属性にuseRef()の戻り値を渡す
- 例: const inputRef = useRef<HTMLInputElement>(null!);
  <input ref={inputRef} ... />
  - コミットフェーズでinputRef.currentにDOM要素が設定される
    - 最初にコンポーネントが実行される時点では未設定
- イベントハンドラからDOM要素にアクセスすることができる
  - 関数コンポーネント本体からはDOM要素にアクセスすべきではない

useRef()の例（DOM要素の取得）

src/RefCounter.tsx

```
import React from "react";

export function RefCounter() {
  const buttonRef = React.useRef(null!);

  const handleClick = () => {
    const buttonElement = buttonRef.current;
    ...
  };

  return (
    <div>
      <button ref={buttonRef} onClick={handleClick}>...</button>
    </div>
  );
}
```

イベントハンドラが呼び出された時点ではcurrentプロパティにDOM要素が設定されている

演習(4-5): Todoアプリ

演習4-3のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[4-5] Todo
などに変更しよう

- 「追加」ボタンで新しいTodoを追加した場合でも
  テキストフィールドにフォーカスが設定されるようにしてみよう
  - フォーカスは<input>要素のfocus()メソッドで設定できます

forwardRef()

- 子コンポーネントはPropsで「ref」を受け取ることができない
  - refという名前以外でなら受け取ることができる
  - 例: inputRef, innerRef, etc.
- 子コンポーネントがrefという名前でRefオブジェクトを受け取れるようにするにはforwardRef()を使う
  - <Input>や<Button>等、プロジェクト固有のスタイルを与えた基本的なコンポーネントでよく使用する
- 使い方: Component = forwardRef((props, ref) => {...})
- 引数はrefを転送する関数
  - 引数にPropsとRefを受け取りReactElement等を返す
- 戻り値はrefを転送できる関数コンポーネント

forwardRef()

src/Button.tsx

```
import React from "react";

type Props = React.ButtonHTMLAttributes<HTMLButtonElement>;

const Button = React.forwardRef<HTMLButtonElement, Props>((props, ref) => {
  const { children, ...rest } = props;
  return (
    <button ref={ref} {...rest}>
      {children}
    </button>
  );
});
```

src/App.tsx

```
import React from "react";

export default function App() {
  const ref = React.useRef<HTMLButtonElement>(null!);

  return (
    <Button ref={buttonRef} onClick={() => console.log("clicked")}>Refを渡せる喜び</button>
  );
}
```

状態とスコープ

- useState()/useReducer()/useRef()による状態はコンポーネント固有
  - コンポーネントの「ローカルステート」と呼ばれる
- 空間的なスコープ (可視範囲)
  - useState()等を呼び出したコンポーネントのみが直接参照できる
  - Propsを通じて子コンポーネントに状態を渡すことができる
  - 該当コンポーネントとその子孫が空間的スコープ
- 時間的なスコープ (存続期間、ライフタイム)
  - useState()等を呼び出したコンポーネントが表示されている (マウントされている) 間のみ状態が存続する
  - 該当コンポーネントが親コンポーネントによってレンダリングされている期間が時間的スコープ

状態のスコープを広げる

- より広範囲のコンポーネントでも状態を共有したい場合
- 状態を親 (祖先) に移動する (状態のリフトアップ)
  - useState()等の呼び出しを祖先コンポーネントで行う
  - より空間的に広い範囲のコンポーネントに状態を渡せる
  - より時間的に長い存続期間を持つことができる
- デメリット
  - 子孫に状態をPropsで受け渡す必要がある
    - Propsのバケツリレー
  - 対策: Contextを導入する
    - Propsのバケツリレーを回避できる
    - 注意深く使わないと再レンダリングが増える

本研修ではContextは扱いません

ステート管理ライブラリ

- Reactコンポーネントに依存せずに状態を扱うライブラリ
  - 「グローバル」ステートとも呼ばれる
    - 提供される空間的・時間的スコープはライブラリによって異なる
  - 例: Redux, Jotai, Valitio, Zustand, etc.

状態と再レンダリング まとめ

- コンポーネントは状態 (State) を持つ
  - 関数コンポーネントそのものが状態を持つわけではない
  - Reactが管理する仮想DOMに状態が保持される
- 組込HooksのuseState(), useReducer()で状態にアクセス
  - 状態が更新されると再レンダリングが発生する
    - 関数コンポーネントは再実行される
    - 最初の表示と同様に実行されるので「更新」を意識しない (宣言的UI)
  - 状態としてオブジェクト (配列を含む) を使っている場合
    - 状態の更新はミュータブルなマナーに従う
  - useState()では「古くなったクロージャ」に注意
    - 「現在の値に基づく更新」は更新関数を利用する
- useRef()で再レンダリングを伴わない状態を扱うことができる

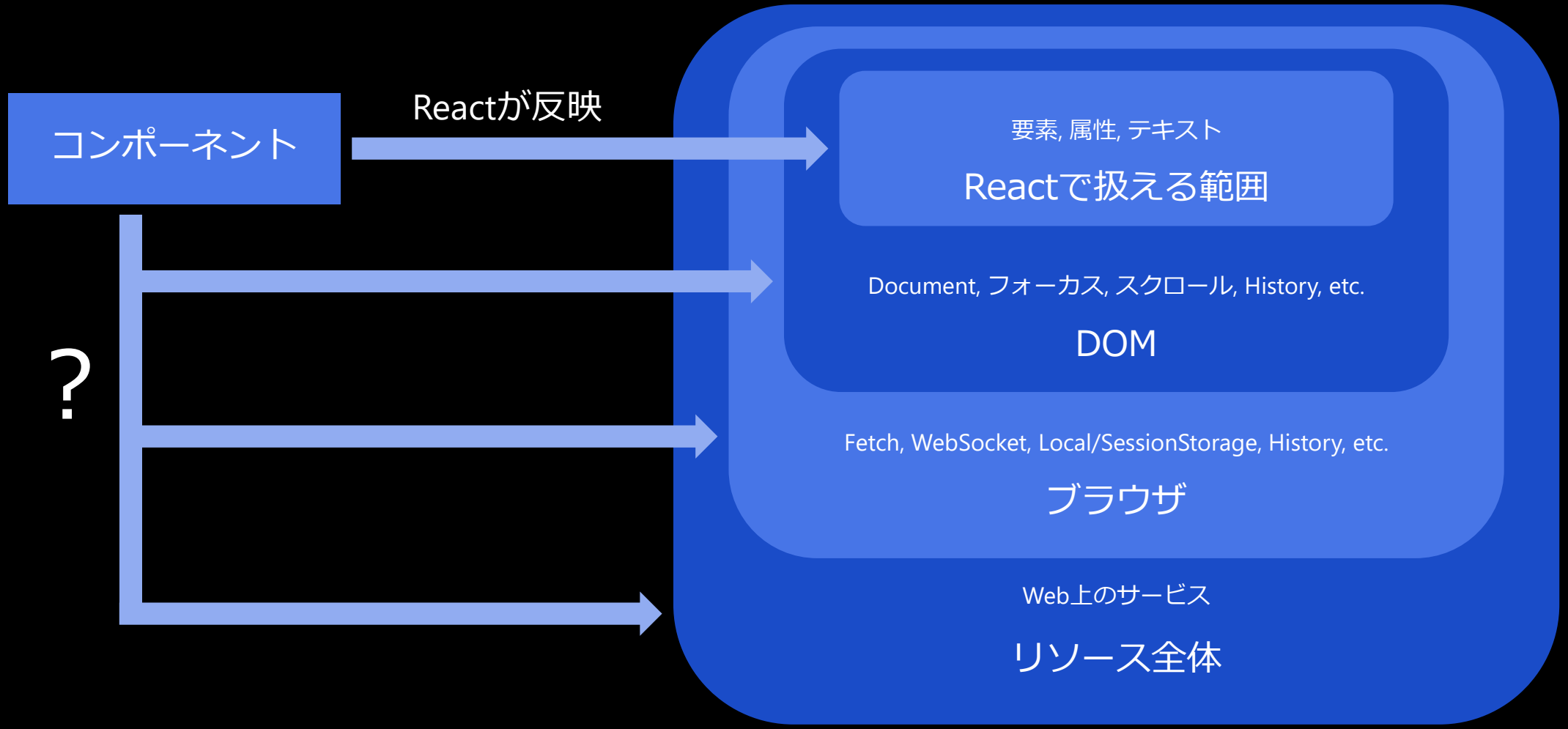
Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外リソースとの同期
- メモ化とパフォーマンス

Reactの宣言的UIとリソース

- Reactの宣言的UIで扱えるのはDOMの一部だけ
  - ルートとなるDOM要素とその子孫の要素、属性、テキスト
- Webアプリで扱う範囲はもっと広い
  - DOM
    - Reactで扱える範囲の外側
      - Document, html要素, head要素, body要素, etc.
    - DOM APIのメソッドを利用する機能
      - フォーカス, スクロール, etc.
  - DOM以外のブラウザが提供する機能
    - Fetch, WebSocket, Local/SessionStorage, History, Workers, etc
  - ブラウザの外
    - Web上のサービス (Web API)

Reactとリソース


コンポーネント
Reactが反映
?
要素, 属性, テキスト
Reactで扱える範囲
Document, フォーカス, スクロール, History, etc.
DOM
Fetch, WebSocket, Local/SessionStorage, History, etc.
ブラウザ
Web上のサービス
リソース全体

コンポーネントと副作用

- コンポーネントの主目的はDOM要素のレンダリング
  - それ以外のリソースを扱うことは「副作用」
- コンポーネント自体は副作用を持つべきではない
  - コンポーネントはレンダーフェーズで実行される
  - レンダーフェーズは途中で破棄されて再実行されることもある
  - コンポーネントが何度実行されるかはReactのスケジューリング次第
- コンポーネントは「べき等」であるべき
  - 繰り返し実行されても不都合がないこと
  - イベントハンドラのようにレンダーフェーズで実行されないコードは副作用を持っても構わない

宣言的UIとリソース

- React管理外のリソースも宣言的UIのマナーに従って扱う
- メンタルモデル
  - Reactのレンダリングとリソースを「同期」する
    - 例:
      - 時計コンポーネントはタイマと同期する
        - 時計コンポーネントが表示されている間はタイマで監視された状態にある
      - チャットコンポーネントはチャットサーバと同期する
        - チャットコンポーネントが表示されている間はチャットサーバからの通知を受け取れる (サブスクリプションしている) 状態にある

useEffect()

- 作用を扱うための組込Hook
  - コンポーネント視点では「副作用」だがuseEffect()視点では「作用」
    - useEffect()の主目的のため「副」作用ではないという扱い
- 使い方: useEffect(effectFunction, deps)
  - 第1引数は「作用」をセットアップする関数
    - 「作用」をクリーンナップする関数を返す (省略可)
  - 第2引数は作用が依存する値の配列 (省略可)
    - 配列の各要素は前回のレンダリング時の対応する要素とObject.is()で比較される
    - オブジェクト (配列や関数を含む) は同一性に注意

「パフォーマンスとメモ化」で説明します

セットアップ/クリーンナップ関数

- セットアップ関数
  - リソースと同期した状態を開始する関数
  - 例
    - タイマを設定する、イベントリスナーを登録する、ネットワークに接続する
  - 引数: なし
  - 戻り値: クリーンナップ関数
- クリーンナップ関数
  - リソースと同期した状態を終了する関数
  - 例
    - タイマを解除する、イベントリスナーを削除する、ネットワークを切断する
  - 引数: なし
  - 戻り値: なし

セットアップ/クリーンナップ関数の例

```
useEffect(() => {  // setup
  const timerId = setTimeout(() => { ... }, 5000);

  return () => clearTimeout(timerId);  // cleanup
});
```

コンポーネントを
「タイムアウト時間が
設定されている状態」
と同期する

```
useEffect(() => {  // setup
  const mouseMoveListener = () => { ... };
  document.addEventListener("mousemove", mouseMoveListener);

  return () =>document.removeEventListener("mousemove", mouseMoveListener); // cleanup
});
```

コンポーネントを
「mousemoveイベントを
監視している状態」
と同期する

```
useEffect(() => {  // setup
  const controller = new AbortController();
  const signal = controller.signal;
  document.addEventListener("mousemove", () => { ... }, { signal });
  document.addEventListener("wheel", () => { ... }, { signal });

  return () =>controller.abort();  // cleanup
});
```

AbortControllerを使うと
クリーンナップ関数が
簡潔に書ける

Clockコンポーネント

src/Clock.tsx

```
export function Clock() {
  const [date, setDate] = React.useState(new Date());

  const formatter = new Intl.DateTimeFormat("ja-JP", {
    hour: "2-digit",
    minute: "2-digit",
    second: "2-digit",
    hour12: true
  });
  const time = formatter.format(date);

  React.useEffect(() => {  // setup
    const timerId = setTimeout(() => {
      setDate(new Date());
    }, 5000);

    return () => {  // cleanup
      clearTimeout(timerId);
    };
  });

  return (
    <div>
      <div><span>{time}</span></div>
    </div>
  );
}
```

演習(5-1): Clockアプリ

Projectの名称を
[5-1] Clock
などに変更しよう

- Stackblitzの新しいProjectでClockアプリを作ってみよう
- 12時間制/24時間制を切り替えられるようにしてみよう
  - Intl.DateTimeFormat()の第2引数に渡しているオブジェクトの hour12で切り替えることができます
- 12時間制/24時間制を連続的に素早く切り替えて時刻が更新される様子を確認してみよう

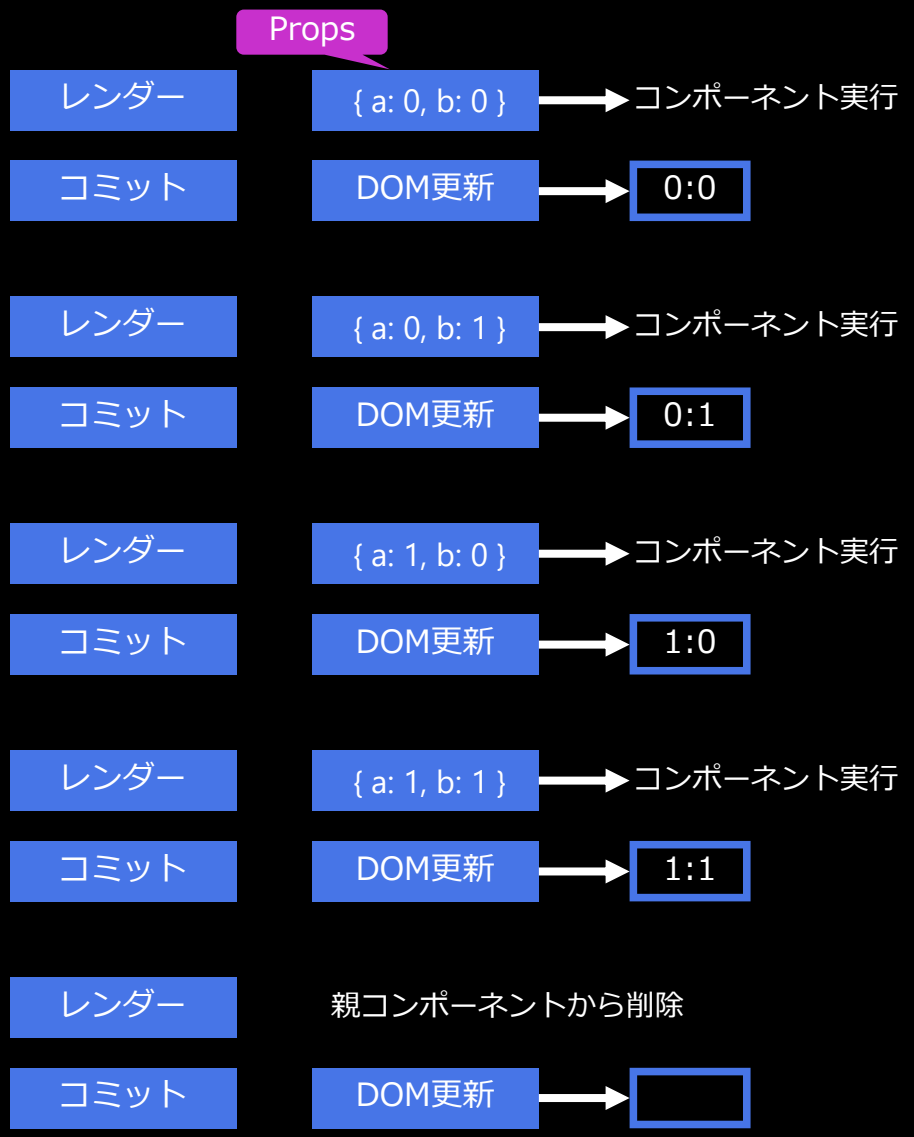
依存配列 (deps)

- useEffect()の第2引数
  - リソースが何と同期するかを指定する
- 依存配列を省略
  - リソースは毎回のレンダリングと同期する
  - 同期の頻度: 多
- 1要素以上の配列
  - リソースは配列の要素である変数 (の値) と同期する
  - 同期の頻度: 中
- 空配列
  - リソースはコンポーネント自体と同期する
  - 同期の頻度: 少

論理的にはこれで
正しく動作すべき

最適化

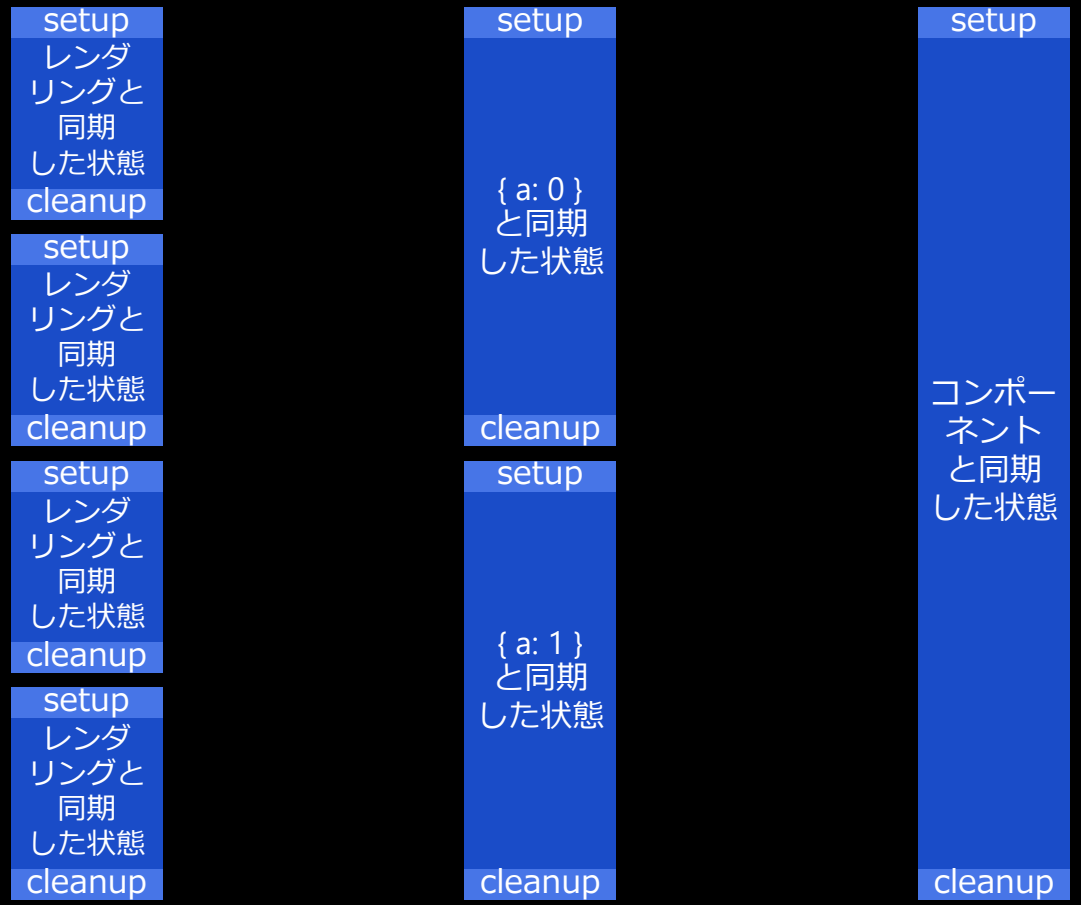
依存配列（deps）とリソースの同期




```
function Foo({a, b}) {
  useEffect(() => {...});
  return <p>{a}:{b}</p>;
}
```

```
function Bar({a, b}) {
  useEffect(() => {...}, M );
  return <p>{a}:{b}</p>;
}
```

```
function Baz({a, b}) {
  useEffect(() => {...},    );
  return <p>{a}:{b}</p>;
}
```



依存配列とクロージャ

- セットアップ/クリーンナップ関数の内部から外側のコンポーネントで定義された変数を参照できる
  - セットアップ/クリーンナップ関数はクロージャ
- セットアップ/クリーンナップ関数から参照している変数を依存配列に指定する
  - 変数の値が変わった場合はリソースと同期し直す
  - 例外: 変化しない (イミュータブルな) 変数
    - 例: useState()が返すsetter, useReducer()が返すdispatch, useRef()が返すRef
  - eslint-plugin-react-hooksでチェックする
    - 将来的にはReact Compilerによって自動的に補われる予定

演習(5-2): Clockアプリ

演習5-1のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[5-2] Clock
などに変更しよう

- useEffect()に依存配列を指定してみよう
  - 12時間制/24時間制を切り替えても時刻の更新に影響が出ないようにしてみよう

演習(5-3): Clockアプリ

演習5-2のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

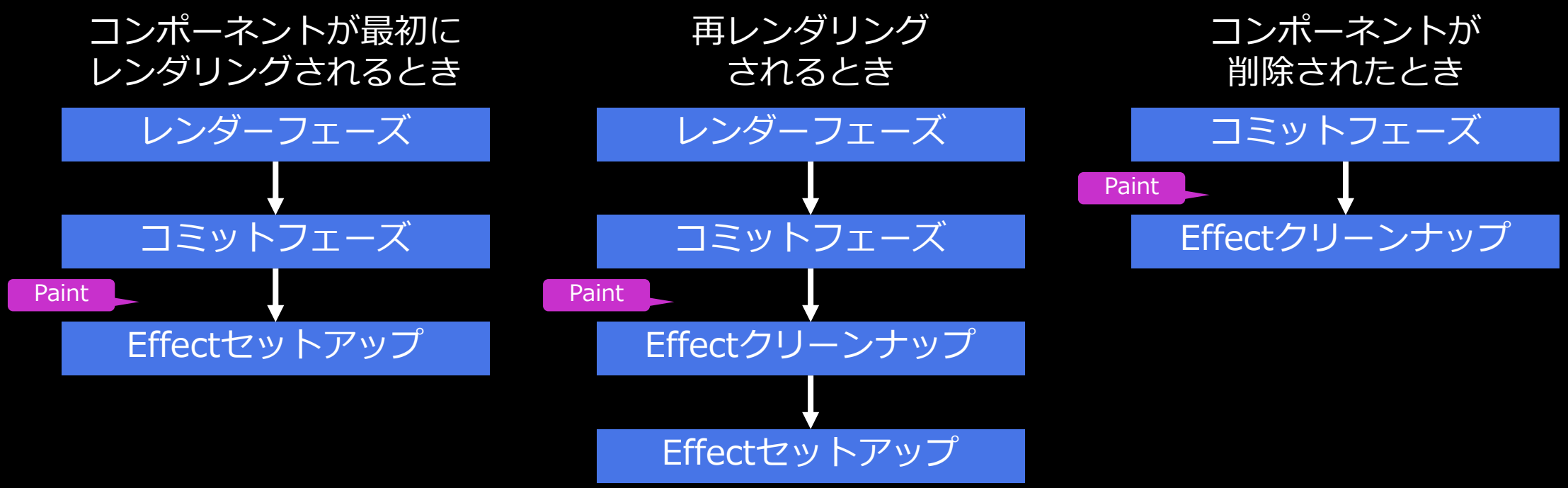
Projectの名称を
[5-3] Clock
などに変更しよう

- setTimeout()の代わりにsetInterval()を使ってみよう
  - 依存配列を適切に変更しよう
- 時刻を更新するインターバルをテキストフィールドで設定できるようにしてみよう
  - 依存配列を適切に変更しよう
- Clockコンポーネントの先頭やセットアップ/クリーンナップ関数にログ出力を入れて動作を確認してみよう
  - リロードした直後の動作を確認してみよう
    - ブラウザのリロードボタンではなくブラウザ内ブラウザのリロードボタンを使用

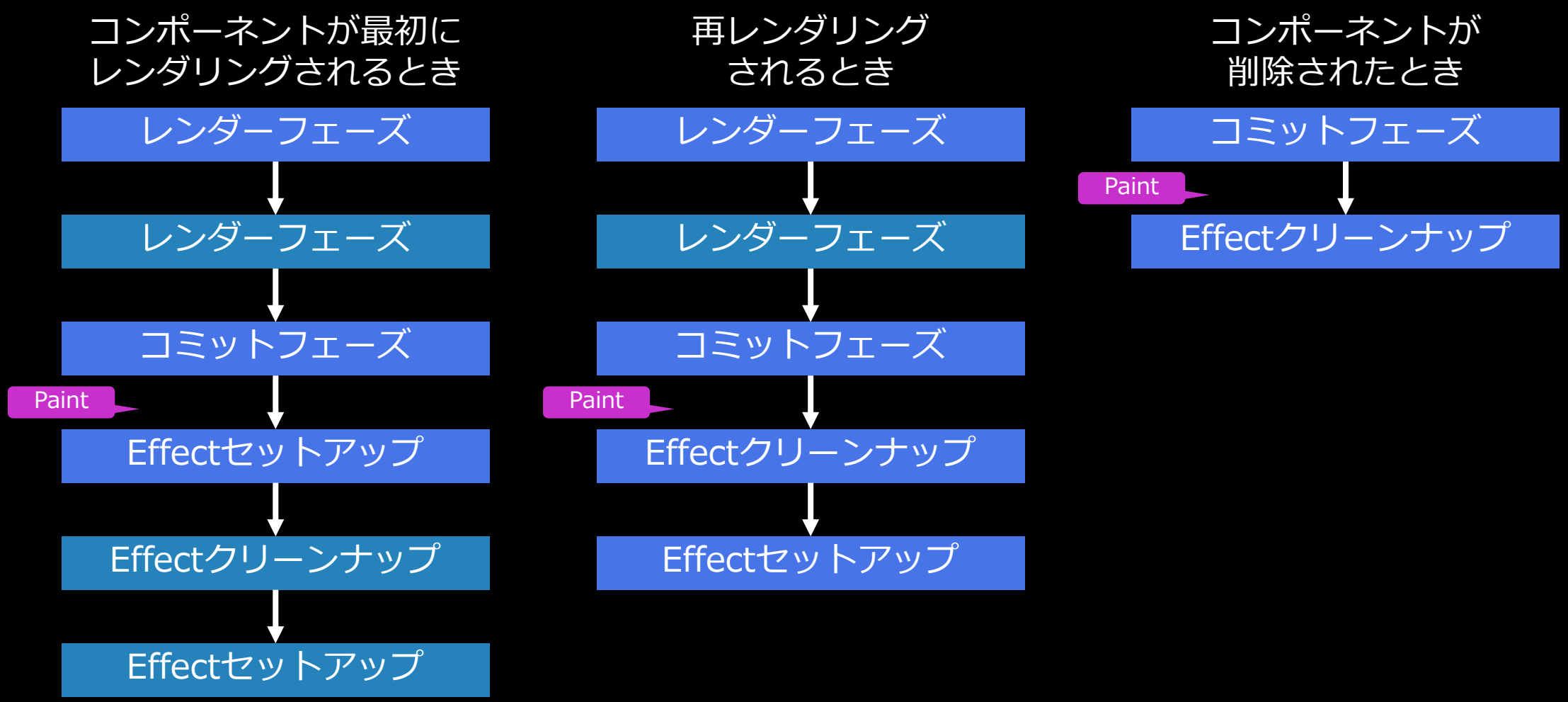
Strict Mode

- 不正な副作用のあるコンポーネントを早期検出するための機能
  - <React.StrictMode>...</React.StrictMode>
- StackBlitzの「React TypeScript」でも適用されている
  - src/main.tsx
- 開発モードでは:
  - レンダーフェーズが2回ずつ実行される
  - コンポーネントが最初にレンダリングされる際はセットアップ関数も2回実行される
    - セットアップ関数 → クリーンナップ関数 → セットアップ関数

Strict Mode無効時の動作



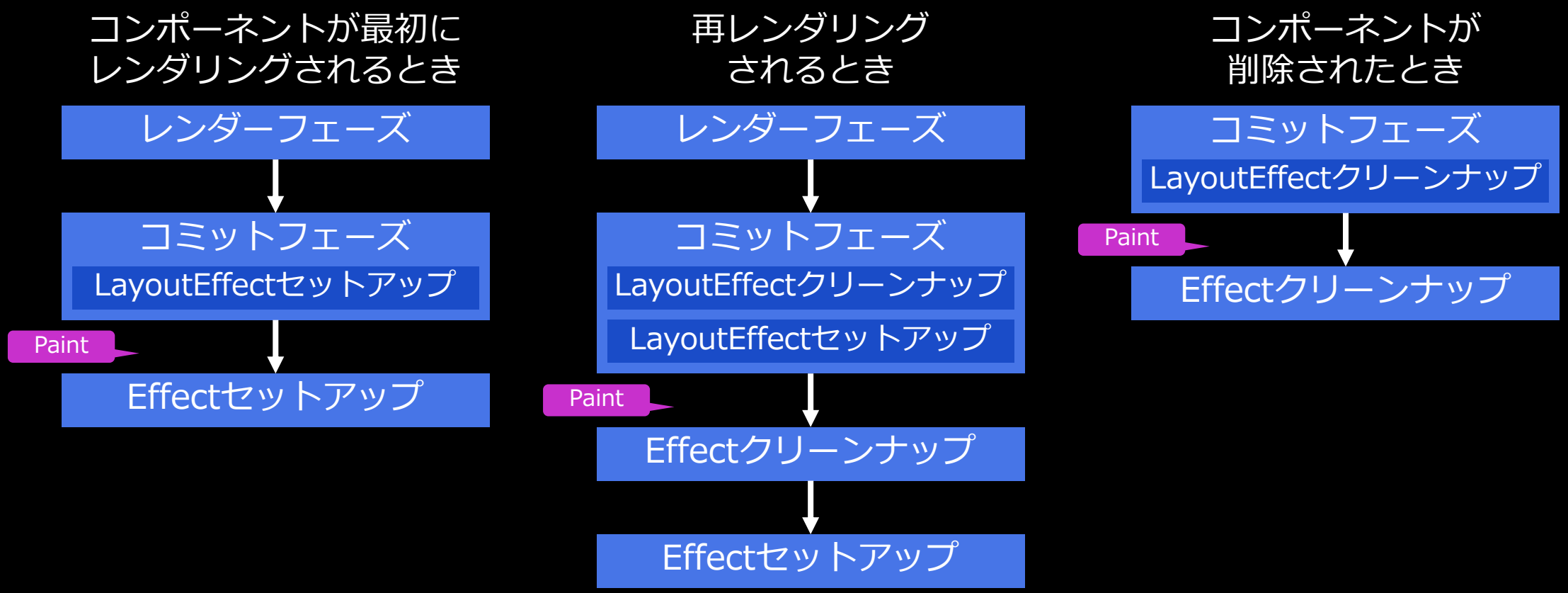
Strict Mode有効時の動作



useLayoutEffect()

- セットアップ/クリーンナップ関数をコミットフェーズで「同期的」に実行する組込Hook
- 使い方: useLayoutEffect(setupFunction, deps)
- DOMを更新されたブラウザが画面を「ペイントする前」にセットアップ/クリーンナップ関数を実行する
  - DOM要素がペイントされる前にそのサイズや位置等を制御したい場合に使える
  - useEffect()は両方の関数を「非同期」に実行する
    - 両関数ともブラウザが画面をペイントした後に実行される
- パフォーマンスに悪影響を与える可能性があるので可能ならuseEffect()を使用すべき

useLayoutEffectの動作 (非Strict Mode)



useLayoutEffect()の例

src/Scroll.tsx

```
import React from "react";

export function Scroll() {
  const ref = React.useRef<HTMLDivElement>(null!);

  React.useLayoutEffect(() => {
    ref.current.scrollIntoView();
  }, []);

  return (
    <div>
      <ul>
        {new Array(1000).fill(0).map((_, index) => (
          <li key={index}>{index}</li>
        ))}
      </ul>
      <div ref={ref}></div>
    </div>
  );
}
```

useEffect()を使うと
スクロール位置が
移動する前の状態が
一瞬見える場合がある

実行環境によっては
useEffect()との違いを
目視できません

React管理外のリソースと作用 まとめ

- コンポーネントからReact管理外のリソースを扱うことは副作用となる
  - コンポーネントから直接リソースを操作してはいけない
  - コンポーネントを何度実行するかはReactのスケジューラ次第
  - コンポーネントは「べき等」であるべき
- React管理外のリソースはuseEffect()/useLayoutEffect()に渡すセットアップ/クリーンナップ関数でReactと「同期」する
  - セットアップ関数は同期した状態を開始する
  - クリーンナップ関数は同期した状態を終了する
- useEffect()/useLayoutEffect()に渡す依存配列でリソースと「同期」する頻度をコントロールする

Agenda

- Webアプリ開発の変遷
- React概要
- コンポーネントとJSX
- 状態と再レンダリング
- React外リソースとの同期
- メモ化とパフォーマンス

Reactのレンダリングとパフォーマンス

- レンダーフェーズ
  - コンポーネントを実行して仮想DOMを構築する
- コミットフェーズ
  - 仮想DOMを (実) DOMに反映する
  - 差分更新が行われる
- 仮想DOMはコミットフェーズを効率化する

レンダーフェーズとパフォーマンス

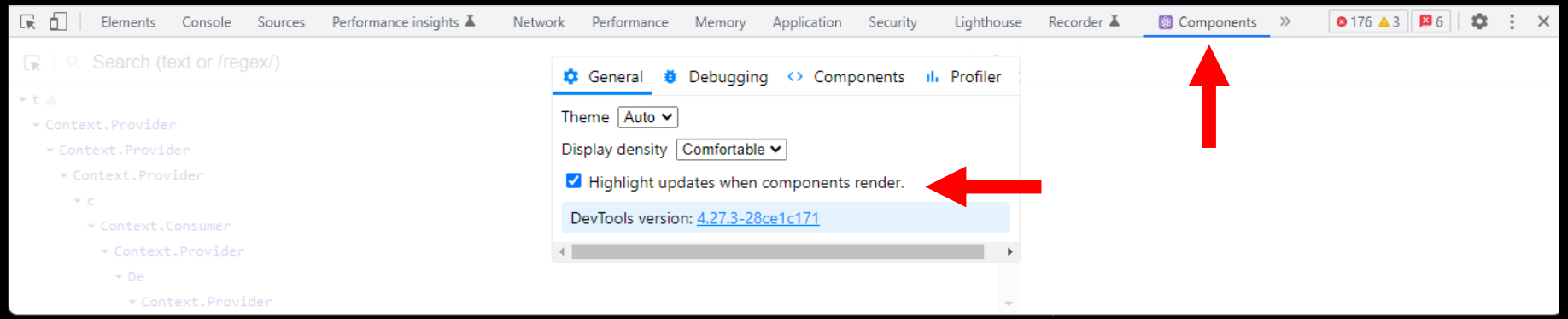
- 前提: 仮想DOMの構築は (実) DOMの構築よりも軽量
  - そのため毎回コンポーネントを再実行しても影響は少ない
- 現実: 大規模な画面では仮想DOMも巨大になる
  - 30行×30列のテーブルがある場合
    - セルコンポーネントは約1000個
    - セルコンポーネントごとに10個の子コンポーネントを持つ場合
    - テーブル全体は約1万個のコンポーネント
- レンダーフェーズの重さが課題になり得る
  - Reactはレンダーフェーズを分割実行するので画面が固まることは生じにくいが画面が更新されるまでの遅延は低減できない
- コンポーネントの　　したい！

React Developer Tools

- React開発者のためのChrome拡張
- Reactアプリを表示している場合、コンポーネントツリーや各コンポーネントのProps/Stateを確認できる
- 再レンダリングされたコンポーネントを可視化できる

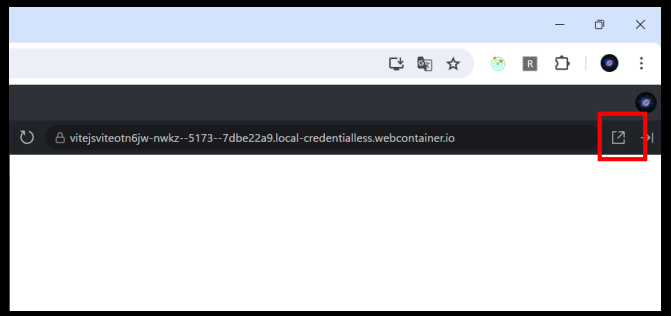
演習(6-1): React DevTools

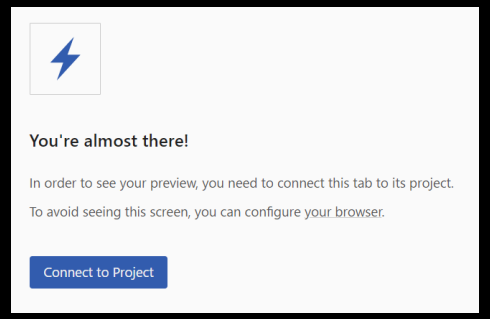
- React Developer Toolsをインストールしよう
  - https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
- Stackblitzを開いたタブでChrome DevToolsを開き、「Components」タブで「View Settings ⚙ 」→「Highlight updates when components render.」をチェックしよう



演習(6-2): Todoアプリ

- 演習(4-5)のTodoアプリを独立したタブで開いてみよう
  - ブラウザ内ブラウザ領域の上にあるアドレスバーの右端にある「Open Preview in new tab」をクリック





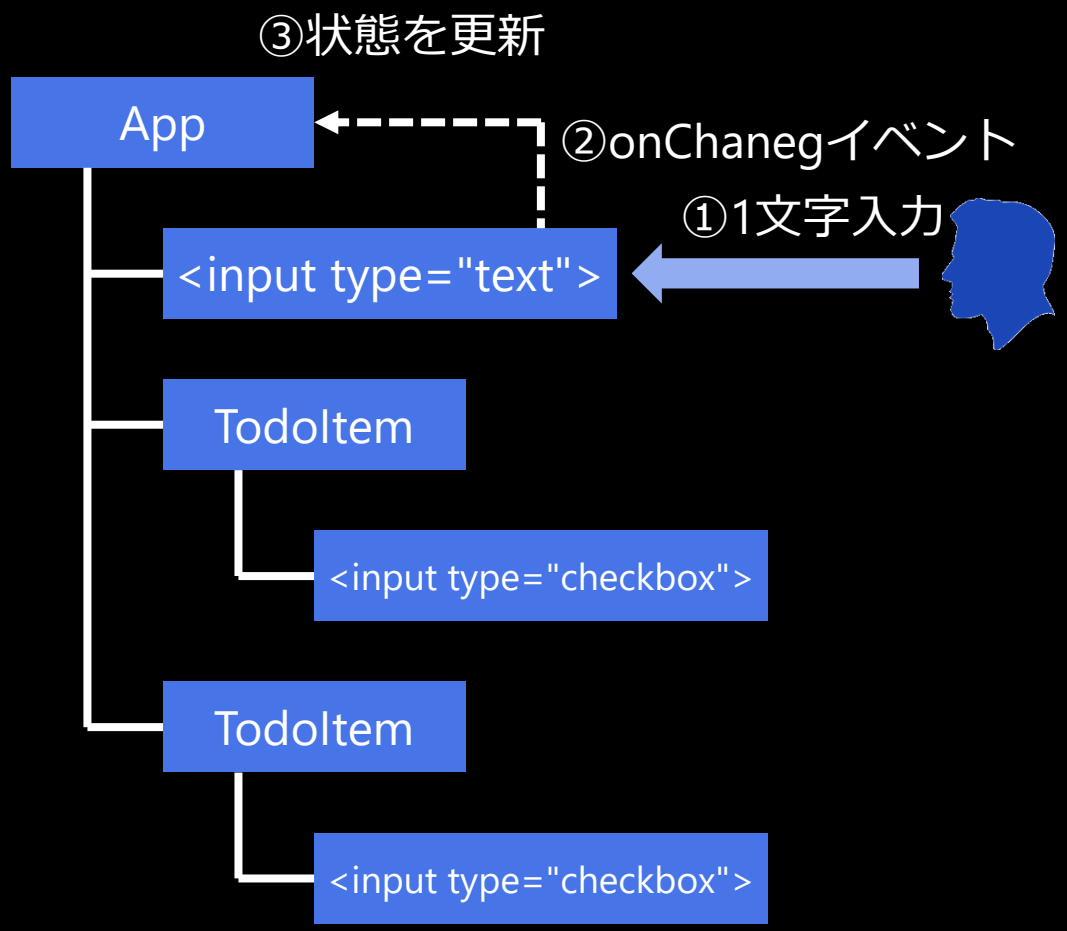
この表示が出たときは
「Connect to Project」を
クリックした上で
タブを開き直してください

- Chrome DevToolsを開いてTodoアプリを操作し、再レンダリングされる様子を見てみよう
  - 新しいTodoを入力するテキストフィールドに文字を入力してみよう
  - TodoItemの状態を変更してみよう

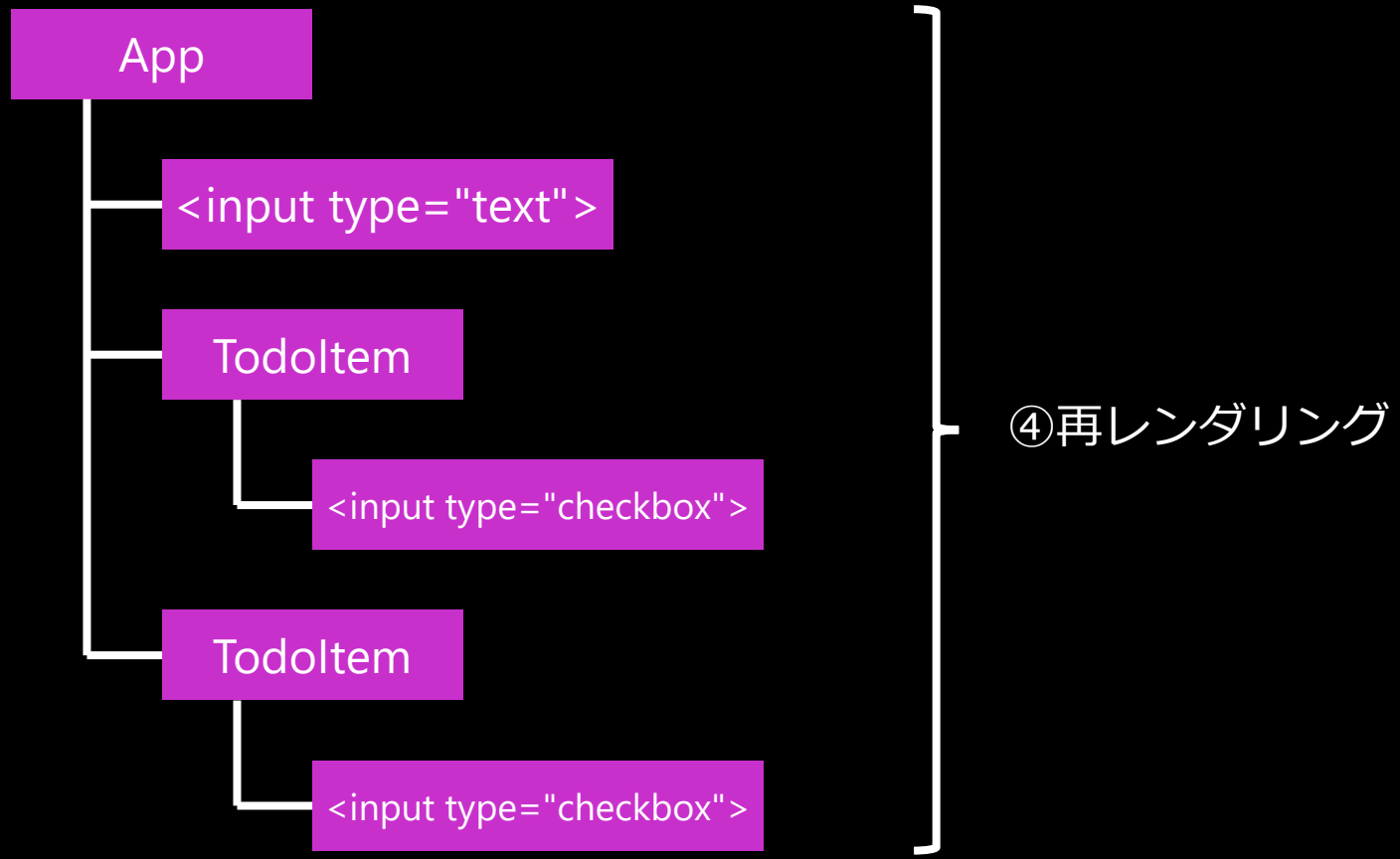
Reactの再レンダリング

- 状態 (State) が変更されるとそのコンポーネントは再レンダリングされる
- そのコンポーネントがレンダリングする子コンポーネントも再レンダリングされる
  - 再帰的
- 状態が変更されるとそのコンポーネント以下のツリー全体が再レンダリングされる

Todoアプリの再レンダリング (1)



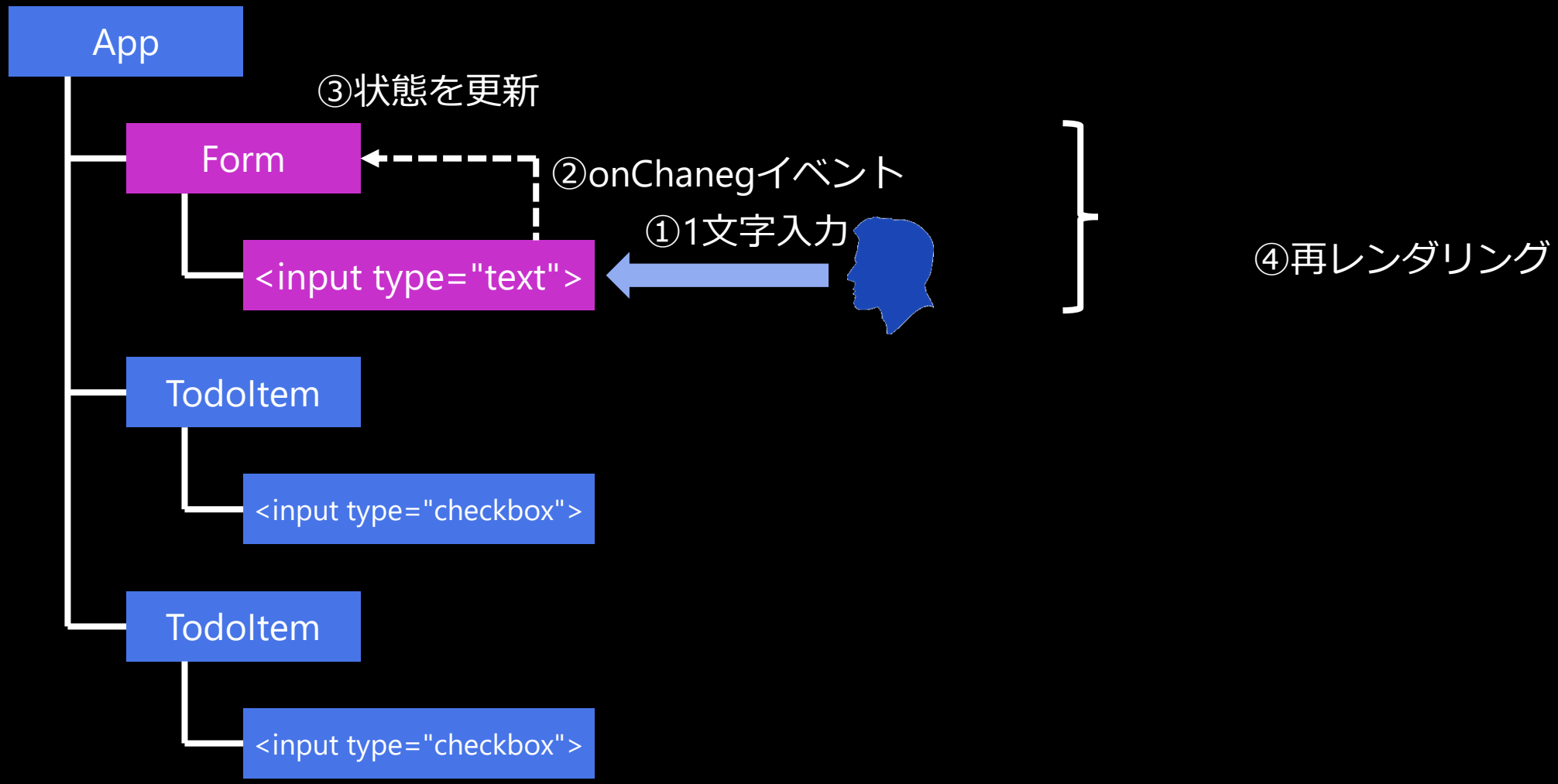
Todoアプリの再レンダリング (2)



再レンダリングを抑止する

- 状態のスコープを狭くする
  - 状態が更新される要因 (ユーザのインタラクション) ごとにコンポーネントを分割する
  - 例: テキストフィールドに1文字入力されるたびに更新される必要があるコンポーネントはどの範囲か？

Todoアプリの再レンダリング



演習(6-3): Todoアプリ

演習4-5のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[6-3] Todo
などに変更しよう

- 新しいTodoを入力するためのForm
  コンポーネントを導入しよう
- テキストフィールドに文字を入力しても実際に登録するまでは
  AppコンポーネントやTodoItemコンポーネントのリストが
  再レンダリングされないようにしてみよう

非制御コンポーネント

- Reactで入力要素等を扱う方法
  - 制御コンポーネント (Controlled Component)
    - React-wayな方法 (宣言的) で入力要素等を扱う
  - 非制御コンポーネント (Uncontrolled Component)
    - React-wayではない方法 (命令的) で入力要素等を扱う
- 非制御コンポーネント
  - 入力要素等の状態をReactで管理しない
    - DOMの状態が「Single Source of Truth」→ 再レンダリングを抑制
  - 使い方: 入力要素等にvalue/checked属性を指定しない
    - defaultValue/defaultChecked属性で初期値を指定できる
    - 初期値を表示した後はDOMの状態が「Single Source of Truth」
  - イベントハンドラでRefからフォーム入力要素の状態を取得

近年はこちらが主に
使われるように
なってきました

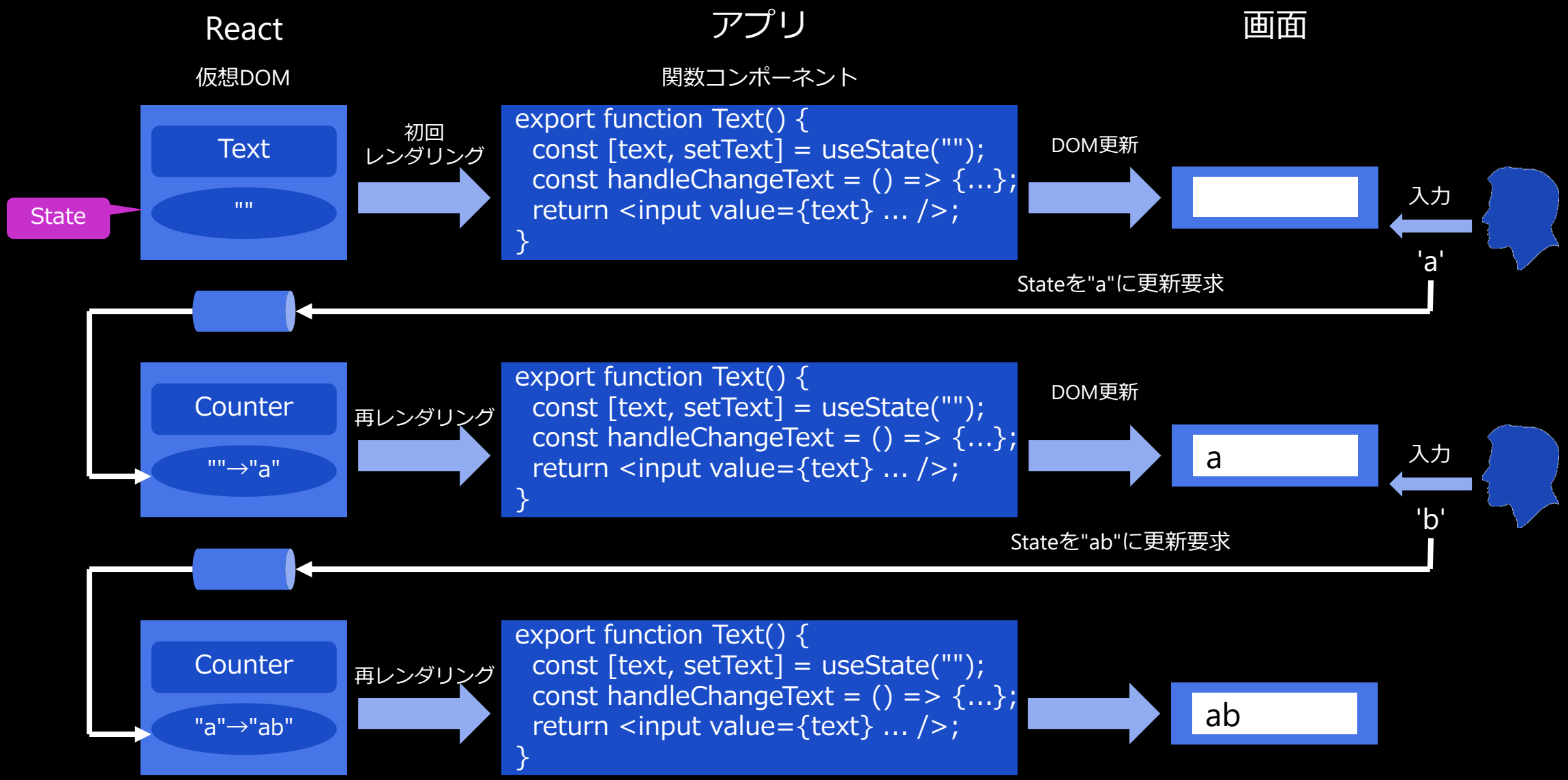
非制御コンポーネント

```
import React from "react";

export function Form() {
  const inputRef = React.useRef<HTMLInputElement>(null!);

  const handleSubmit: React.FormEventHandler<HTMLFormElement> = (event) => {
    event.preventDefault();
    console.log(inputRef.current.value);
    inputRef.current.value = "";
    inputRef.current.focus();
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} defaultValue="" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

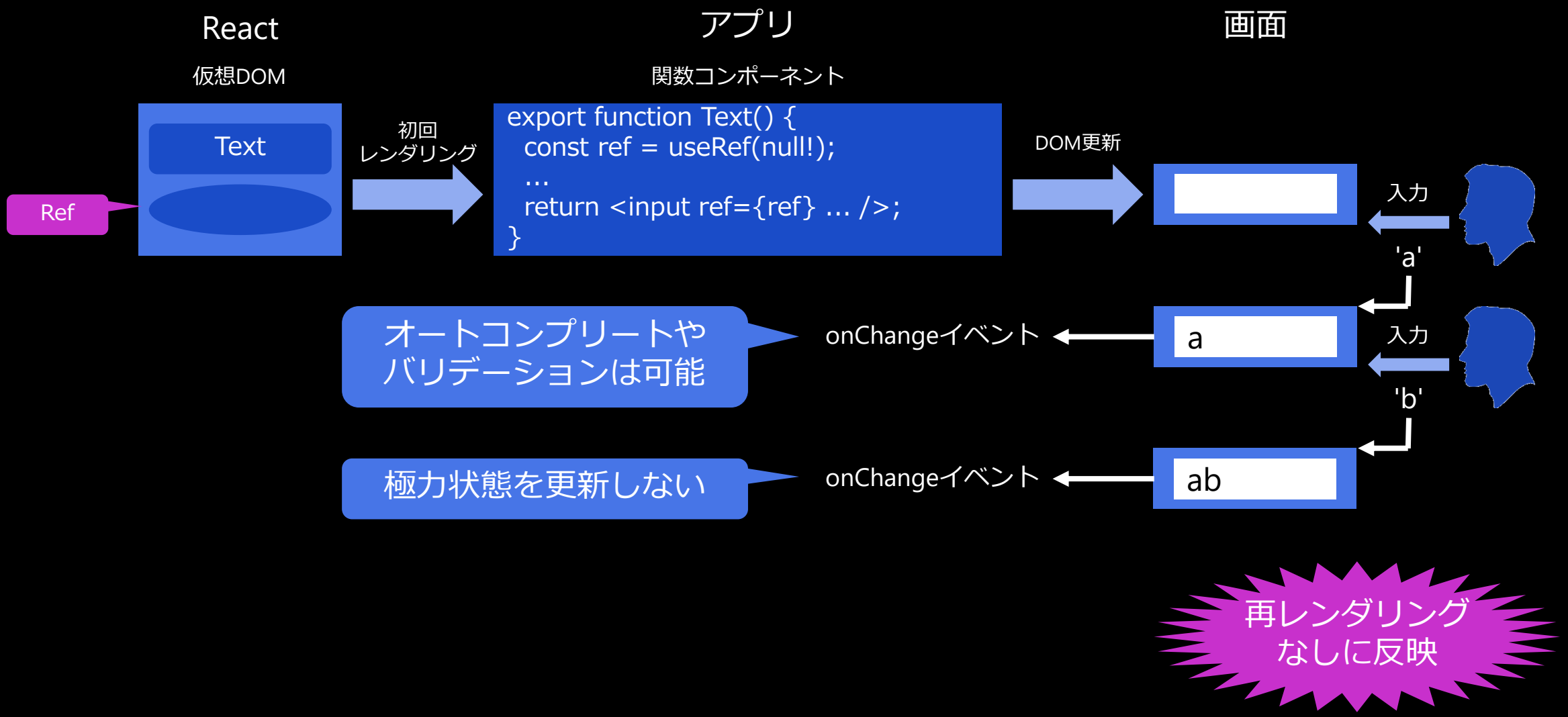
テキストフィールドの
内容をRefから取得

Refを通じて要素の
属性を更新できる

制御コンポーネント（再掲）



非制御コンポーネント


React
仮想DOM
Text
Ref
初回
レンダリング
アプリ
関数コンポーネント
export function Text() {
const ref = useRef(null!);
...
return <input ref={ref} ... />;
}
DOM更新
画面
入力
'a'
a
入力
'b'
ab
オートコンプリートや
バリデーションは可能
onChangeイベント
極力状態を更新しない
onChangeイベント
再レンダリング
なしに反映

演習(6-4): Todoアプリ

演習6-3のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[6-4] Todo
などに変更しよう

- Formコンポーネントのテキストフィールドを
  非制御コンポーネント化してみよう
  - テキストフィールドに文字を入力してもFormコンポーネントが
    再レンダリングされないことを確認しよう
- テキストフィールドの状態に応じて「追加」ボタンの
  有効/無効を再レンダリングなしに切り替えられるように
  してみよう

再レンダリングを抑止する

- コンポーネントをメモ化する
  - メモ化
    - 関数が返した結果をキャッシュして、同じ引数で呼び出された場合はキャッシュを返すことで関数の再実行を回避する

React.memo()

- Reactコンポーネントをメモ化する
- 使い方: Memoized = React.memo(Component, compare)
- 第1引数はメモ化する対象のコンポーネント
- 第2引数はPropsを比較する関数 (任意)
  - デフォルトはPropsオブジェクトを「浅い比較」する
    - 各プロパティについてObject.is()で比較する
- 戻り値はメモ化されたコンポーネント
  - 直前に実行された場合と同じPropsで呼び出された場合は引数のコンポーネントを実行しない
    - Reactが管理している前回の実行結果 (仮想DOM) を再利用する

React.memo()の使いどころ

- 親コンポーネントが再レンダリングされた場合でも
  子コンポーネントは再レンダリングする必要が少ない場合
  - 親と子で再レンダリングの頻度が異なる場合
- 使う必要がない場合
  - 親が再レンダリングされる場合には子も再レンダリングされる場合
    - 親と子で状態が変化するタイミングが一致している場合

演習(6-5): Todoアプリ

演習6-4のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[6-5] Todo
などに変更しよう

- FromおよびTodoItemコンポーネントに
  React.memo()を適用してみよう
- 実際に再レンダリングが抑制されたか確認してみよう

オブジェクト・関数の同一性

- React.memo()はPropsの各プロパティをObject.is()で比較する
  - useEffect()/useLayoutEffect()の依存配列も同様
- Object.is()はオブジェクトをStrict Equality (===) で比較する
  - 関数もオブジェクト
- オブジェクトリテラルや関数式は評価されるたびに新しいオブジェクトを作成する
  - 再レンダリングで関数コンポーネントが実行されるたびに関数内に記述されたオブジェクトリテラルや関数式は新しいオブジェクトや関数を作成する
- Propsがオブジェクトや関数を含む場合はレンダリングごとにPropsが異なってしまう

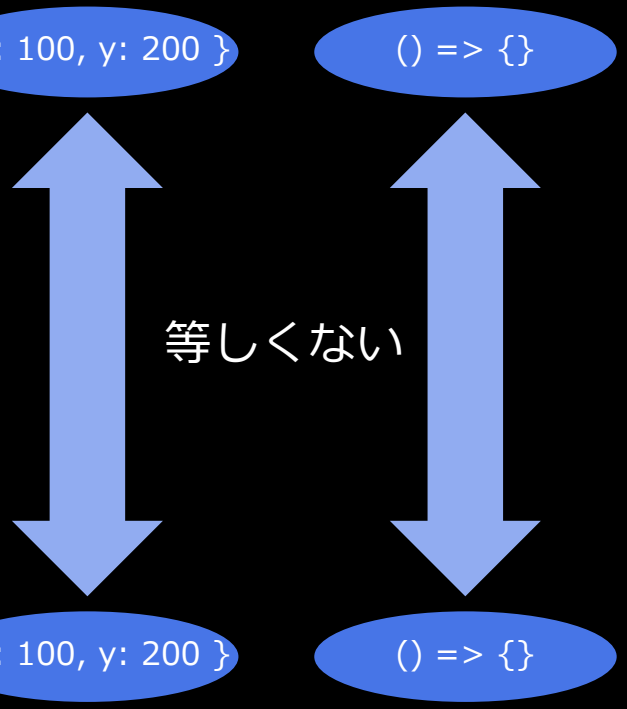
オブジェクト・関数の同一性

レンダリング1

```
import React from "react";
import { Child } from "./Child";
const MemoizedChild = React.memo(Child);

export function Foo () {
  const point = { x: 100, y: 200 };
  const handleEvent = () => {};

  return (
    <MemoizedChild point={point} handleEvent={handleEvent} />
  );
}
```

レンダリング2

```
import React from "react";
import { Child } from "./Child";
const MemoizedChild = React.memo(Child);

export function Foo() {
  const point = { x: 100, y: 200 };
  const handleEvent = () => {};

  return (
    <MemoizedChild point={point} handleEvent={handleEvent} />
  );
}
```



MemoizedChildは再レンダリングされる

React.useMemo()

- 任意の値をキャッシュするための組込Hook
  - 同一性のためだけではなく重い計算をキャッシュする用途でも使える
- 使い方: cached = useMemo(calculate, deps)
- 第1引数はキャッシュされる値を計算する関数
- 第2引数はキャッシュする値が依存する値の配列
  - 配列の各要素をObject.is()で比較する
  - useEffect()のdepsと似ている (省略は不可)
- 戻り値はキャッシュされた値
  - 前回のレンダリング時とdepsの全要素が等しければ
    キャッシュされた値が返される
  - それ以外はcalculateが再実行されてその戻り値が返される

React.useCallback()

- 関数をキャッシュするための組込Hook
- 使い方: cached = useCallback(fn, deps)
  - useMemo(() => fn, deps)と同等
- 第1引数はキャッシュ対象の関数
- 第2引数はキャッシュする関数が依存する値の配列
  - 配列の各要素をObject.is()で比較する
  - useEffect()のdepsと似ている (省略は不可)
- 戻り値はキャッシュされた関数
  - 前回のレンダリング時とdepsの全要素が等しければ
    キャッシュされた関数が返される
  - それ以外はuseCallback()に渡された関数が返される

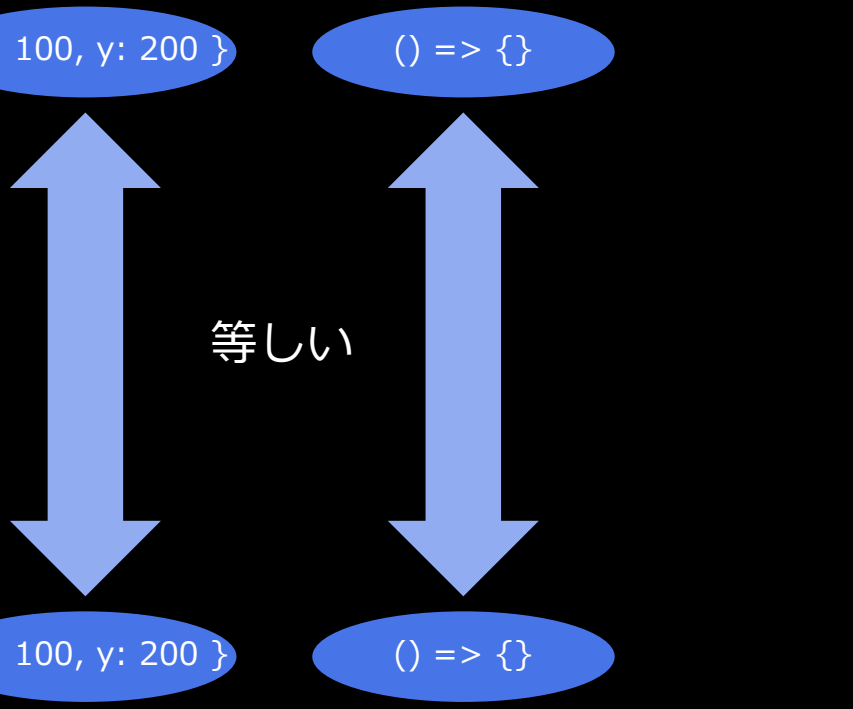
オブジェクト・関数の同一性

レンダリング1

```
import React from "react";
import { Child } from "./Child";
const MemoizedChild = React.memo(Child);

export Foo() {
  const point = React.useMemo(() => { x: 100, y: 200 }, []);
  const handleEvent = React.useCallback(() => {}, []);

  return (
    <MemoizedChild point={point} handleEvent={handleEvent} />
  );
}
```

レンダリング2

```
import React from "react";
import { Child } from "./Child";
const memoizedChild = React.memo(Child);

export function Foo() {
  const point = React.useMemo(() => { x: 100, y: 200 }, []);
  const handleEvent = React.useCallback(() => {}, []);

  return (
    <MemoizedChild point={point} handleEvent={handleEvent} />
  );
}
```



演習(6-6): Todoアプリ

演習6-5のProjectを開いて
左上の「Fork」で
新しいProjectを作成しよう

Projectの名称を
[6-6] Todo
などに変更しよう

- useMemo()/useCallback()を導入して、
  メモ化されたFormおよびTodoItemコンポーネントの
  不要な再レンダリングが抑止されるようにしてみよう
- 再レンダリングが抑制されたか確認してみよう

useMemo()/useCallback()を使うとき

- ホストコンポーネントに渡されるオブジェクト・関数
  - ホストコンポーネントは実行されないので適用する必要は
- Reactコンポーネントに渡されるオブジェクト・関数
  - Reactコンポーネントがメモ化されているなら適用
- useEffect(), useMemo(), useCallback()の依存配列に渡されるオブジェクト・関数
  - 常に適用
- 上記に該当するか自明でない場合は適用
  - 3rd-Partyのコンポーネントに渡すオブジェクト/関数や広く共有されるカスタムHooksに渡す/返すオブジェクト/関数など

パフォーマンスとメモ化 まとめ

- Reactコンポーネントの状態 (State) が更新されると
  そのコンポーネントと子孫コンポーネントのツリー全体が
  再レンダリングされる
- レンダーフェーズで実行されるコンポーネントの量が問題と
  なる場合がある
- 再レンダリングされるコンポーネントツリーを小さくする
  - 異なるタイミングで更新される状態毎にコンポーネントを分割する
  - 更新される状態を持つコンポーネントをツリーの下の方に置く
- 親コンポーネントが頻繁に再レンダリングされても
  再レンダリングの必要が少ない子コンポーネントはメモ化する

React研修 まとめ

- 学んだこと
  - 現代のWebアプリの形態とReactが利用されてる状況
  - Reactの特徴および関数コンポーネントの書き方、JSXの書き方
  - コンポーネントの状態を更新する方法とその背後でのReactの動作
  - DOM以外のリソースを宣言的UIに沿った方法で扱う方法
  - 仮想DOMではカバーしきれないレンダーフェーズの最適化
- 次のステップ
  - 実際にコードを書いて動作と頭の中のイメージを一致させよう
  - React公式ドキュメントを読んで正しい知識を得よう
  - ライブラリ、フレームワーク、ツールなどのエコシステムを知ろう
  - React Server Componentsなど次世代の機能を知ろう

お疲れ様でした！