React実習

06. Reactの状態管理

株式会社ジードライブ

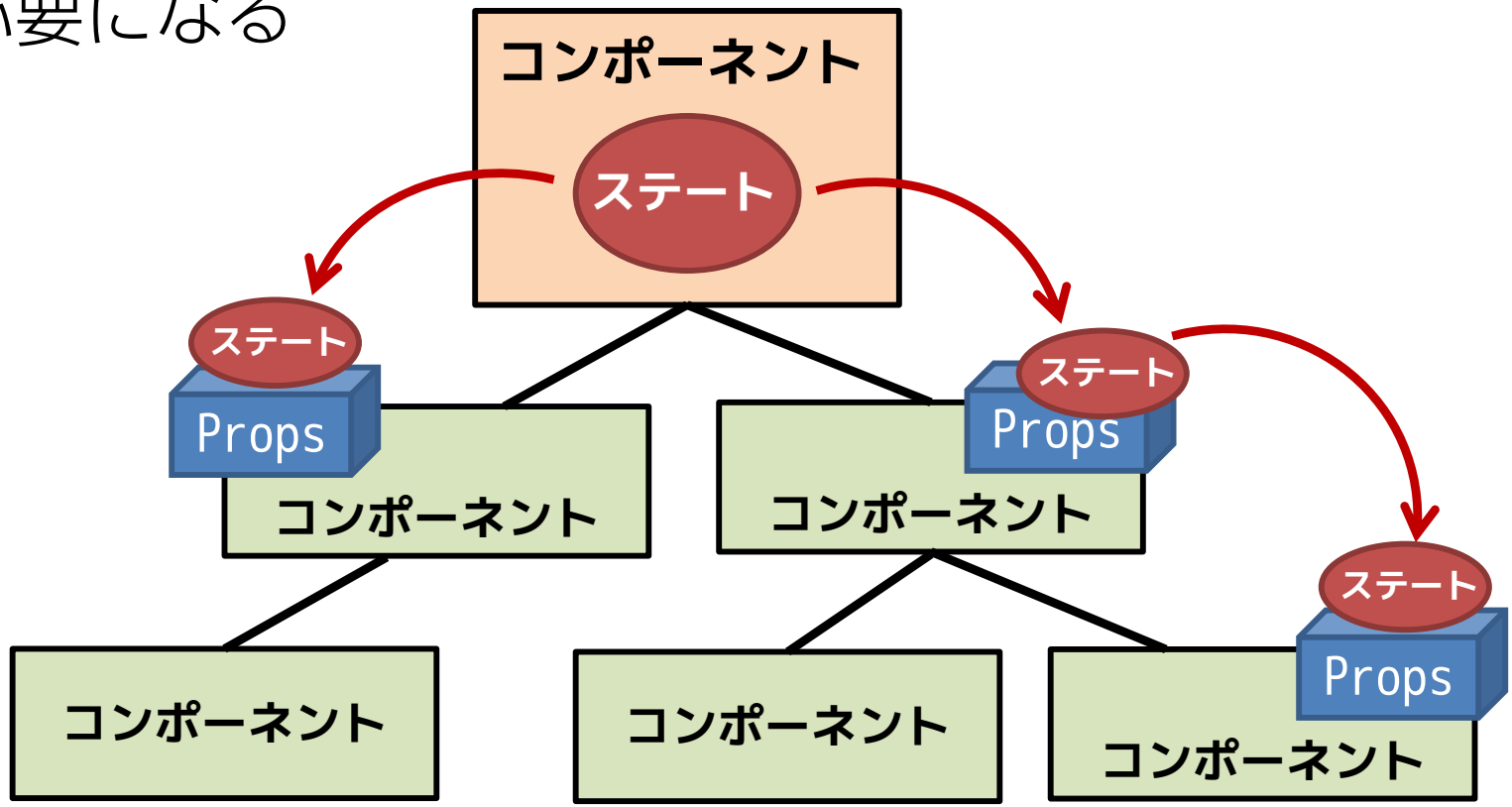
今回学ぶこと

- Reactでの状態(グローバルステート)管理の方法
  - **Jotai**
  - useReducer + Context API
  - Redux

ReactにはuseState以外にも、状態管理の方法が多数あるが、ここではJotaiを学習する

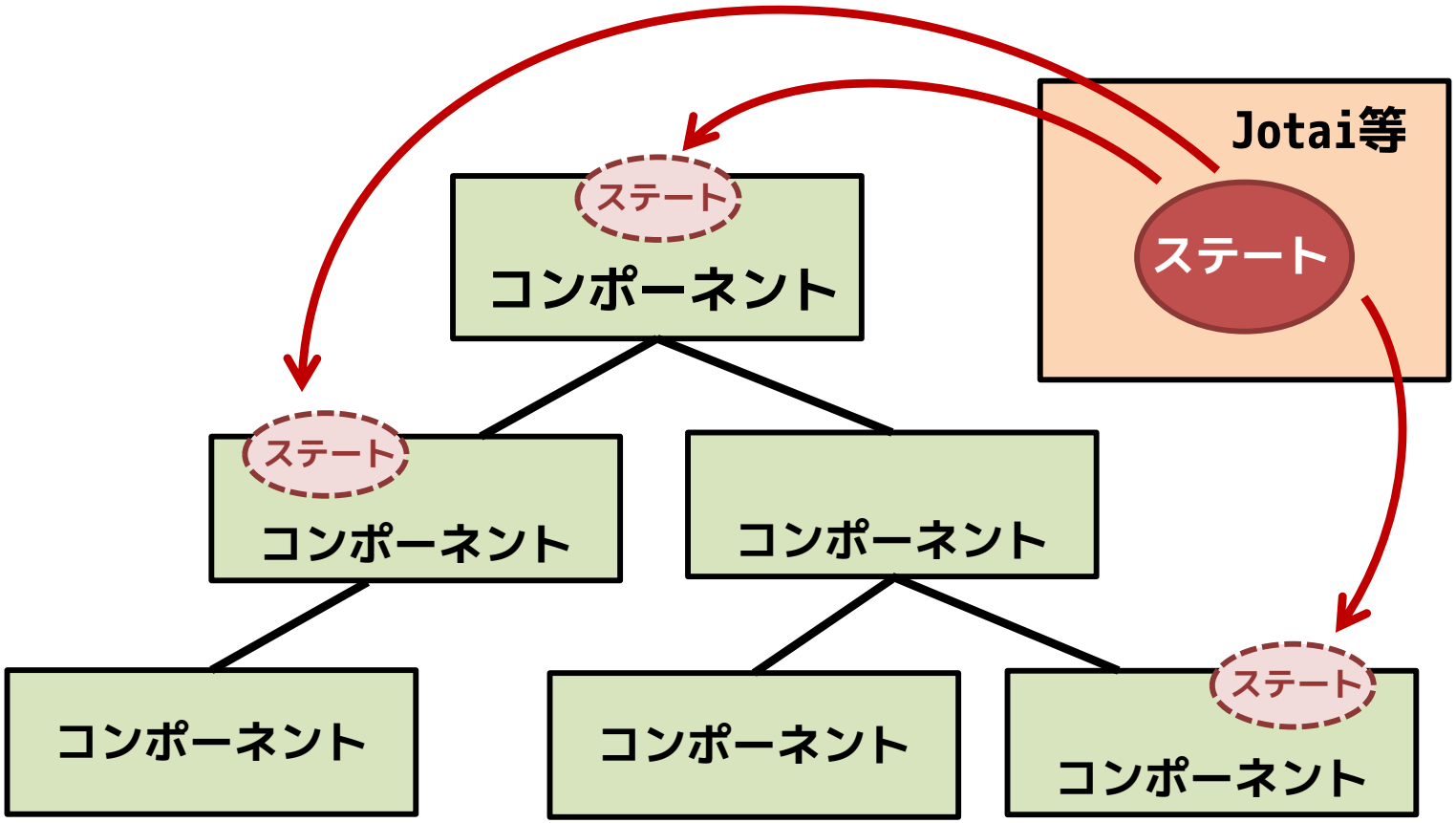
useStateの課題

- useStateでは、コンポーネント内で使うステート(ローカルステート)の管理はできるが、それを他のコンポーネントで使おうとすると、Propsを通じたバケツリレーが必要になる



コンポーネントをまたぐステート

- 複数のコンポーネントでステートを共有したい(グローバルステートを利用したい)場合は、JotaiやReduxといったライブラリやContext APIを利用する



Jotaiによる状態管理

Jotaiとは

- React用の状態管理ライブラリ
  - https://jotai.org/
- Jotaiで扱う１つ１つのステートはアトムと呼ばれており、useStateに似たuseAtom等のフック関数を使用してコンポーネント内に呼び出すことができる

例: useStateによる状態管理

```
const [count, setCount] = useState(0);
```

コンポーネント内

例: Jotaiによる状態管理

```
export const countAtom = atom(0);
```

コンポーネント外

```
const [count, setCount] = useAtom(countAtom);
```

コンポーネント内

Jotai利用の流れ

1. Jotaiをインストールする

```
npm install jotai
```

2. 必要なアトムを作成する

コンポーネント外

```
export const countAtom = atom(0);
```

3. コンポーネントでアトムを呼び出して利用する

コンポーネント内

```
const [count, setCount] = useAtom(countAtom);
return (<button onClick={() => setCount(count + 1)}>
    {count}
</button>);
```

アトムの作成

- Jotaiで扱うアトム(ステート)は、atom関数で作成する
  - atom関数の引数として、ステートの初期値を設定することができる
  - アトムから値を読み出すコンポーネントは、暗黙的にこのアトムを購読してる状態になり、アトムの値が変更されると再レンダリングが行われる

例: アトムの作成

```
import { atom } from "jotai";

export const countAtom = atom(0);
export const itemsAtom = atom(["りんご", "みかん", "バナナ"]);
export const membersAtom = atom([
    {name: "山田太郎", age: 25}, {name: "鈴木花子", age: 24}
]);
```

アトムの呼び出し

- Jotaiでは、以下のようなフックが用意されており、アトムを利用することができる

フック	説明
useAtom()	ステート値とステート更新関数を呼び出す
useSetAtom()	ステート更新関数を呼び出す
useAtomValue()	ステート値を呼び出す

例：ステート値と更新関数の両方を呼び出す

引数はアトム

```
const [count, setCount] = useAtom(countAtom);
```

例：更新関数とステート値を個別に呼び出す

```
const setItems = useSetAtom(itemsAtom);     //更新関数
const members = useAtomValue(membersAtom); //ステート値
```

練習

- 練習06-1
- 練習06-2

派生アトム(派生ステート)

- 既存のアトムを元に作られるアトム
  - 元となるアトムの状態が変化すれば、派生アトムの状態も変化する

```
// 派生元のアトム
export const scoreAtom = atom(0);

// 派生アトムの定義
export const judgeAtom = atom((get) => {
    const score = get(scoreAtom);
    return score >= 60 ? "合格" : "不合格";
});
```

派生元のステート値を呼び出す

派生させたステートをリターンする

派生アトム(派生ステート)

- 派生アトムは、通常のアトムと同じように呼び出せる

```
const [score, setScore] = useAtom(scoreAtom);
const result = useAtomValue(judgeAtom);

return (
   <div>
      <input
        type="number"
        value={score}
        onChange={(e) => setScore(Number(e.target.value))}
      />
      点 ⇒ {result}
   </div>
);
```

ステートの初期化

- useResetAtomフックで、ステートを初期化する関数を取得することができる
  - アトムは、atomWithReset関数で作成しておく必要がある

```
import { atomWithReset } from "jotai/utils";

export const scoreAtom = atomWithReset(0);
```

jotai/utils からインポート

```
import { useAtom } from "jotai";
import { useResetAtom } from "jotai/utils";

const [score, setScore] = useAtom(scoreAtom);
const reset = useResetAtom(scoreAtom);
return <button onClick={reset}>リセット</button>;
```

atomWithResetで作成したアトムもuseAtomで利用可能

練習

- 練習06-3
- 練習06-4

localStorageとsessionStorage

- ブラウザ上にデータを保存する仕組みとして、localStorageとsessionStorageが存在する
  - 同じメソッドを使い、データを操作することができる

例: localStorageの利用（sessionStorageにも同名のメソッドが存在する）

```
localStorage.setItem("info", "東京都新宿区"); //データの保存
const info = localStorage.getItem("info");    //データの取得
localStorage.removeItem("info");              //データの削除
localStorage.clear();                        //全データをクリア
```

両者の違い

localStorage	sessionStorage
10MBまで保存可能	5MBまで保存可能
タブやウィンドウを閉じてもデータは残る	タブを閉じるとデータが消える
タブをまたいでデータを共有できる	タブをまたいでデータは共有されない

localStorageとsessionStorage

- ストレージにデータを保存する際は、データを文字列に変換する
  - オブジェクトを文字列に変換するには、**JSON.stringify()**を使う
- ストレージから取り出すデータは文字列なので、必要に応じて適切な型へ変換する
  - 文字列をオブジェクトに変換するには、**JSON.parse()**を使う

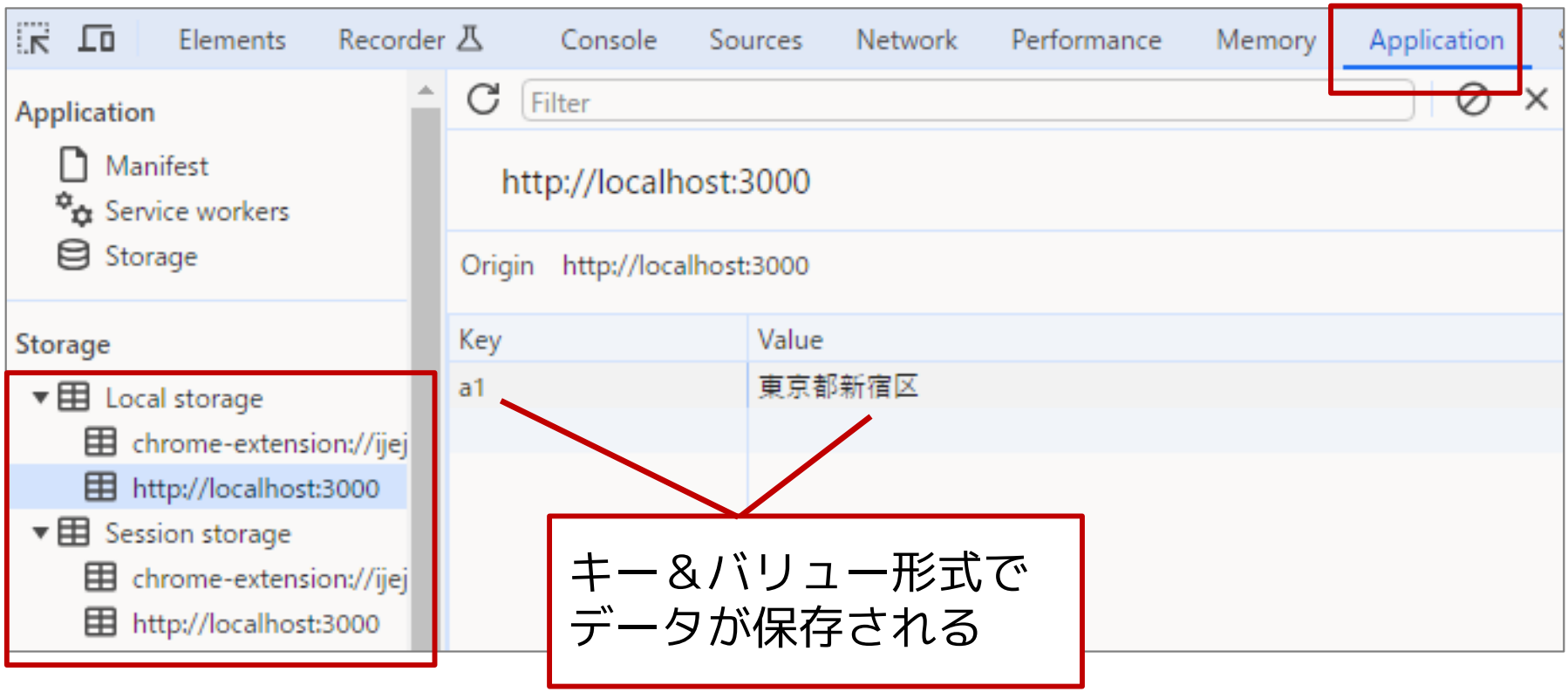
```
const user = { name: "山田太郎", age: 25 };
localStorage.setItem("info", JSON.stringify(user));

const userInfo = JSON.parse(localStorage.getItem("info"));
```

localStorageとsessionStorage

- localStorageやsessionStorageに格納したデータは、ブラウザの開発者ツールで確認できる

例：Chromeの開発者ツール



atomWithStorage

- atomWithStorageを使うと、ステートをlocalStorageやsessionStorageに保存できる
  - https://jotai.org/docs/utilities/storage

```
import {atomWithStorage } from "jotai/utils";

export const darkModeAtom =
                atomWithStorage("darkMode", false);
```

jotai/utils からインポート

ストレージ保存時のキー名

ステートの初期値

デフォルトの設定では、**localStorage**に保存される

sessionStorageに保存する場合

- sessionStorageにデータを保存する場合は、atomWithStorageの第三引数でストレージの操作方法を指定する

```
// ストレージの操作方法をあらかじめ定義しておく
const sessionStorageInterface = {
  getItem: (key) => sessionStorage.getItem(key),
  setItem: (key, value) => sessionStorage.setItem(key, value),
  removeItem: (key) => sessionStorage.removeItem(key),
};

export const darkModeAtom =
   atomWithStorage("darkMode", false, sessionStorageInterface);
```

sessionStorageを使用する

atomWithStorageの第三引数として指定

ストレージへの保存

- ステートに変更がある度に、ストレージにデータが保存される

```
const [darkMode, setDarkMode] = useAtom(darkModeAtom);

return (
    <div>
      <h1>現在のモード: {darkMode ? "ダーク" : "ライト"}</h1>
      <button onClick={() => setDarkMode(!darkMode)}>
        モード変更
      </button>
    </div>
);
```

練習

- 練習06-5

補足：その他のステート管理方法

useReducerとは

- useReducerは、useStateのようにステートを管理するためのフックだが、複数の関連するステートを１つのオブジェクトとして管理する場面やステートの更新ロジックが複雑な場合などに利用される

例：useReducerフックの基本書式

```
const [state, dispatch] = useReducer(reducer, initialValue);
```

管理するステート

Dispatcherと呼ばれる更新用の関数

ステートの更新ロジックを定義したReducer関数

管理するステートの初期値
通常はオブジェクトとして用意する

useReducer: Dispatcher関数

- Dispatcher関数は、引数としてtypeとpayloadから構成されるオブジェクト(actionオブジェクト) を受け取る
  - このactionオブジェクトがReducer関数に渡される
  - typeプロパティはデータの追加や削除といったステートに対する操作を示す文字列
  - payloadプロパティには、ステートに追加するデータや削除対象のIDといった情報を格納する

Dispatcher関数の利用例：商品を追加

```
const action = {
    type: "add",
    payload: { name: "りんご", price: 100 }
};

dispatch(action);
```

useReducer: Reducer関数

- Reducer関数は、引数としてstateとactionオブジェクトを受け取る
  - actionオブジェクトのtypeに応じて更新したstateを返す

Reducer関数の定義例

```
const reducer = (state, action) => {
   const {type, payload} = action;
   switch(type) {
      case "add":
         const newItem = { ...payload, id: crypto.randomUUID() };
         return {...state, items: [...state.items, newItem]};
      default:
         return state;
   }
};
```

Context APIとは

- Propsのバケツリレーを回避し、ステートやその更新関数といったデータをコンポーネントをまたいで使えるようにするためのしくみ
  - 複数のコンポーネントで同じステート(認証済みのユーザー情報など)を共有したい場合に便利
  - RecoilやReduxといった外部ライブラリとは異なり、あらかじめReactに組み込まれている
- 共有するステートやその更新関数をContextという形でまとめ、Context Providerを通じて各コンポーネントから利用できるようにする

Context API: Contextの作成

- まず、ContextとContextProviderを作成する

```
import { createContext, useState } from "react";

export const ItemContext = createContext({});

export default function ItemContextProvider({ children }) {
  const [items, setItems] = useState([]);

  const addItem = (item) => {
    const newItem = { ...item, id: crypto.randomUUID() };
    setItems((prev) => [...prev, newItem]);
  };

  return (
    <ItemContext.Provider value={{ items, addItem }}>
      {children}
    </ItemContext.Provider>
  );
}
```

Contextの作成
引数は共有データの初期値

useReducerを使ってもよい

ContextProviderの作成
このコンポーネントの子孫内で、
データが共有される

value属性で共有データを指定する
このデータで、createContextの引数で
設定した初期値が上書きされる

Context API: Contextの利用

- ContextProviderコンポーネントを配置する

```
root.render(
  <React.StrictMode>
    <ItemContextProvider>
      <App />
    </ItemContextProvider>
  </React.StrictMode>
);
```

この例では、アプリケーションのルートとなるAppコンポーネントを囲っているが、もっと下層のコンポーネントを囲ってもよい

- コンポーネント内で、useContextフックを使い、コンテキストを呼び出す

```
import { useContext } from "react";
import { ItemContext } from "../contexts/ItemContext";

export default function Items() {
  const { items, addItem } = useContext(ItemContext);
  ...
}
```

必要なデータを分割代入で取り出す

引数は呼び出したいコンテキスト

Redux

- ReduxはJavaScriptアプリケーション用の状態管理ライブラリ
  - Fluxという設計パターンに基づいて作られている
  - 後発のuseReducerやRecoilにも影響を与えている
  - Reactと併用されることが多いが、その他のライブラリやフレームワークとも併用できる
- アプリケーションの規模が大きく、デバッグやロギングといった多様な機能が必要な場合に利用される
  - 比較的規模が小さく、非同期処理があまり多くないアプリケーションでは、組み込みのContext APIの機能だけで十分と判断されることがある