

React実習

03. フック

株式会社ジーードライブ

今回学ぶこと

- フックとは
- Reactの組み込みフック
 - useState
 - useEffect
- カスタムフック

フック(Hooks)とは

- useStateやuseEffectといったuseで始まる関数
- 組み込みのフック関数としては、以下のようなものが存在する
 - useState: コンポーネント内部の状態を管理するための関数
 - useEffect: コンポーネントのマウントや更新といったといったライフサイクルイベントに際して、副作用のある処理を実行するための関数
 - その他、多くのフック関数が存在し、独自のフック(カスタムフック)を作成することもできる

フックのルール

- フックを呼び出せるのは関数のトップレベル
 - 反復処理内や条件分岐内などで呼び出してはいけない
- フックを呼び出せるのは、関数コンポーネントの定義内、または後述のカスタムフック内のみ
 - 通常のJavaScript関数から呼び出してはいけない

Reactの組み込みフック1

useState

コンポーネントとステート(状態)

- ステート(状態)は、コンポーネントが保持することができるデータで、変化する可能性がある
 - ステートが変化する(更新される)と画面が再レンダリングされる

例：商品テーブルコンポーネント

状態(ステート)

```
[{ name: "りんご", price: 100 }]
```



状態(ステート)

```
[  
  { name: "りんご", price: 100 },  
  { name: "ぶどう", price: 600 },  
  { name: "みかん", price: 400 }  
]
```

| 商品名 | 値段 |
|-----|------|
| りんご | 100円 |

↓ 状態が更新される
⇒ コンポーネントの再描画

| 商品名 | 値段 |
|-----|------|
| りんご | 100円 |
| ぶどう | 600円 |
| みかん | 400円 |

ステートの例

フォームを扱うコンポーネントでの「状態」の例

- 送信ボタンの状態
 - クリック可能か不可能か
- テキストボックスに入力されている値の状態
 - どのような文字列が入力されているか
- 選択肢の状態
 - どの項目が選択されているか
- エラーメッセージの状態
 - エラーメッセージを表示中か否か

The screenshot shows a user registration form with the following elements and annotations:

- ユーザー登録** (User Registration) title.
- An error message: **入力文字数が不足しています** (Input character count is insufficient).
- A text input field: **ユーザー名: yama**.
- A dropdown menu: **お住いの都道府県: 東京**.
- A blue **送信** (Send) button.

Annotations with red boxes and arrows explain the states:

- エラー表示中の状態** (Error displayed state) points to the error message.
- 東京を選択している状態** (Selected Tokyo state) points to the dropdown menu showing "東京".
- クリック不可の状態** (Click不可 state) points to the **送信** button, which is currently disabled.
- yamaと入力されている状態** (yama is inputted state) points to the text input field containing "yama".

useState

- 関数コンポーネント内のステートを扱うための関数
 - 戻り値は、ステートを格納する変数とそれを更新するための関数
⇒ これらが配列で返ってくる

書式：一般的に配列の分割代入で受け取る

```
const [ステート変数, ステート更新用関数] = useState(初期値);
```

例

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

更新用関数は、慣例として「set○○」と命名する

初期値として
「0」が入る

setCount(3) を実行すると
countに「3」が入る

更新用関数の利用方法

- ステート変数には、直接、値を代入してはいけない
⇒ 更新用関数を利用する必要がある

方法1. 直接、値を設定する

例：countに「2」をセットする

```
setCount(2);
```

方法2. ステート値を利用する

- コールバック関数で、ステートの値を利用できる

例：countを「1」加算する

```
setCount(prev => prev + 1);
```

countを参照

countにセットする値を返す

更新用関数の注意事項

- ステート変数は、更新用関数の実行直後ではなく、次にレンダリングされるときにアップデートされる

例：ステートの値を加算する関数

```
const countUp = () => {
  console.log(count);
  setCount(prev => prev + 1);
  console.log(count);
};
```

同じ値が出力される

この時点では、更新後の値を確認できない

更新用関数の注意事項

- 前ページに示した性質から、更新用関数の利用方法によっては、以下のような違いが現れる

例：ステート変数を直接利用

```
const count3up = () => {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);
};
```

この一連の処理で、
countには「1」しか加算されない

例：コールバック関数でステートの値を利用

```
const count3Up = () => {
  setCount(prev => prev + 1);
  setCount(prev => prev + 1);
  setCount(prev => prev + 1);
};
```

この一連の処理で、
countには「3」加算される

useStateと配列

- 配列のステートに更新を加える場合、スプレッド構文で配列を複製しつつ、更新処理を行う

例：配列のステート管理

```
const [items, setItems] =  
  useState(["りんご", "みかん", "バナナ"]);  
  
// itemsステートの末尾にアイテムを追加する関数  
const addItem = item => {  
  // items.push(item); ← NG  
  setItems(prevItems => [...prevItems, item]);  
};
```

useStateとオブジェクト

- オブジェクトのステートに更新を加える場合、スプレッド構文でオブジェクトを複製しつつ、更新処理を行う

例：オブジェクトのステート管理

```
const [info, setInfo] =  
  useState({ name: "山田太郎", age: 25, address: "東京" });  
  
// infoステートの名前を更新する関数  
const updateName = (newName) => {  
  // info.name = newName; ← NG  
  setInfo((prevInfo) => ({ ...prevInfo, name: newName }));  
};
```

練習

- 練習03-1

Reactの組み込みフック2

useEffect

純粋関数と副作用

- 以下の 2 つの特徴をもつ関数は**純粋関数**と呼ばれる
 - 1. 外部の状態に依存せず、同じ引数に対しては、常に同じ結果を返す
 - この性質は**参照透過性**と呼ばれる
 - 2. 外部の状態を変更しない
 - すなわち、**副作用**をもたないということ
 - 副作用とは、関数外の変数を変更したり、DOMやファイルの書き換えを行うなど、外部の状態に対して影響を与えてしまうこと
 - 関数内で実行される `console.log()` や `window.alert()`、Ajax通信も副作用に含まれる

参照透過性をもつ関数

- 以下の関数は、周囲の状態に関係なく、引数が同じであれば、必ず同じ結果を返す(参照透過性をもつ)

```
function getTotal(num1, num2) {  
    return num1 + num2;  
}
```

- 以下の関数は、外部の影響を受けており、条件によって異なる結果を返す(参照透過性をもたない)

```
function getTotal(num1, num2) {  
    return (num1 + num2) * rate;  
}
```

関数外の変数の影響を受ける

```
let rate = 1.25;  
console.log(getTotal(10, 20)); // 37.5  
rate = 2.0;  
console.log(getTotal(10, 20)); // 60
```

同じ引数だが、
結果が異なる

副作用のある関数

- 以下の関数は、関数外のオブジェクトを変更している
⇒ 副作用がある

```
const user = { name: "山田太郎", age: 25 };
```

```
function changeName(newName) {  
  const u = user;  
  u.name = newName;  
  return u;  
}
```

変数「u」と「user」が
同じものを参照する

「u」に変更を加えると
「user」にも影響する

以下のように修正すると、副作用はなくなる

```
function changeName(newName) {  
  const u = { ...user };  
  u.name = newName;  
  return u;  
}
```

スプレッド構文で「user」を複製して
いるので、変数「u」と「user」は、
別のものを参照する

コンポーネントのライフサイクル

① Mounting

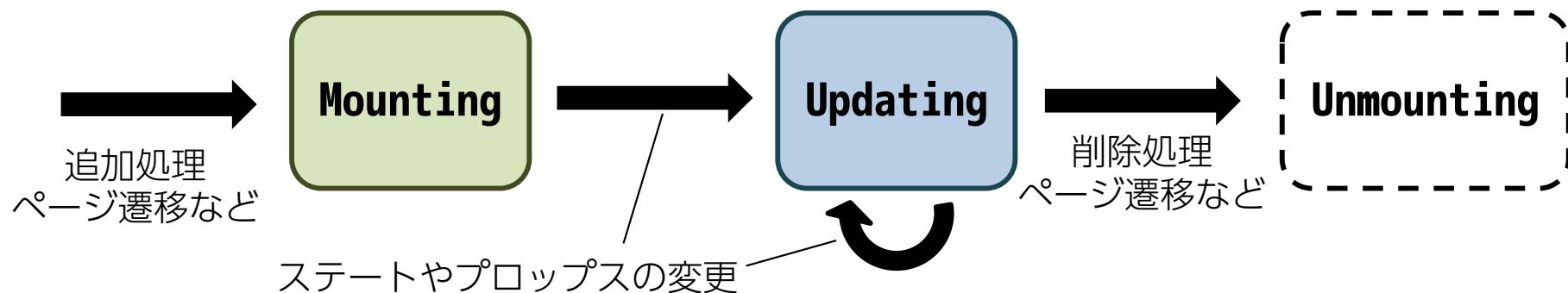
– コンポーネントが初回レンダリングされるタイミング

② Updating

– コンポーネントの再レンダリングされるタイミング

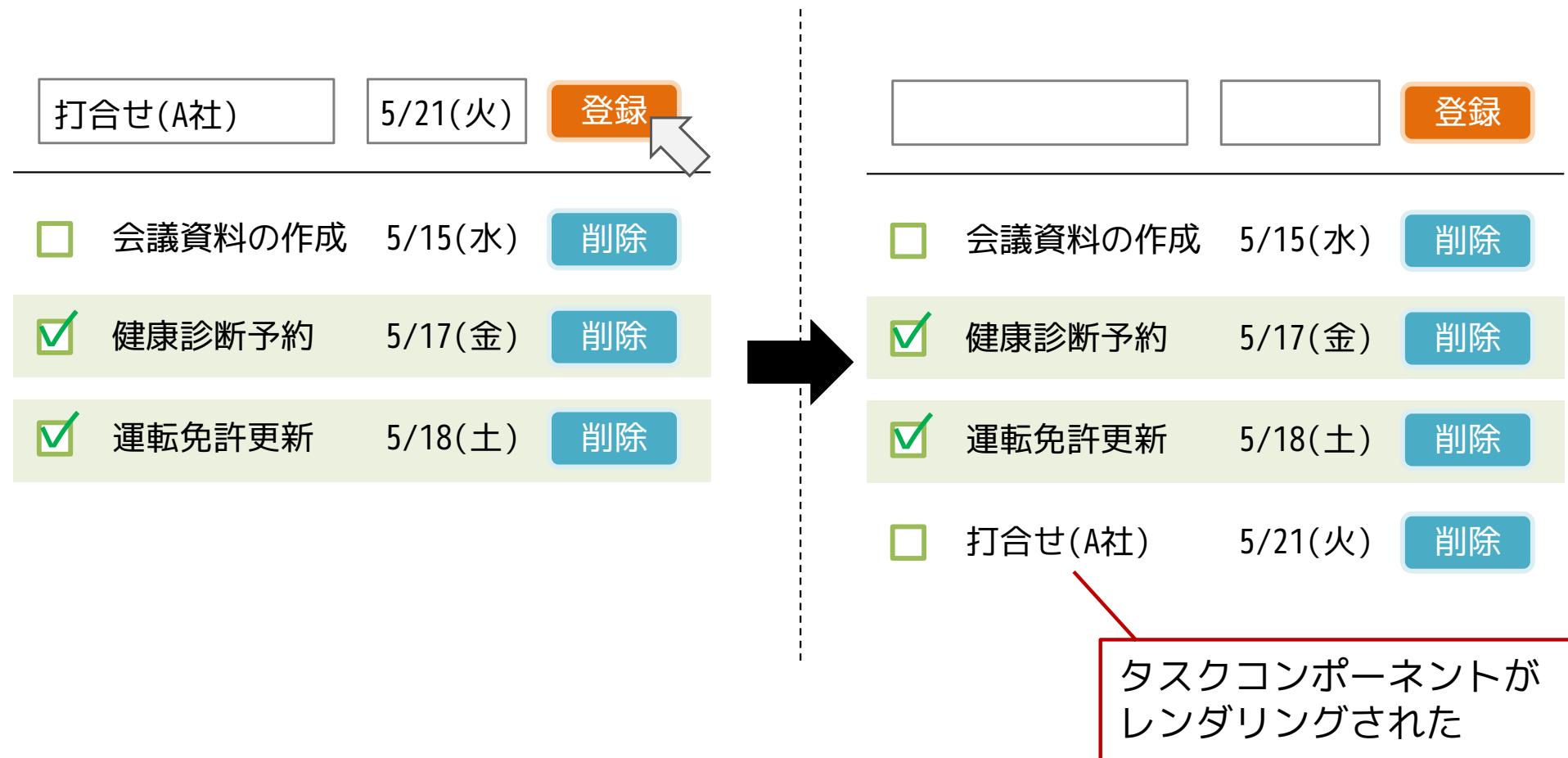
③ Unmounting

– コンポーネントが削除されるタイミング



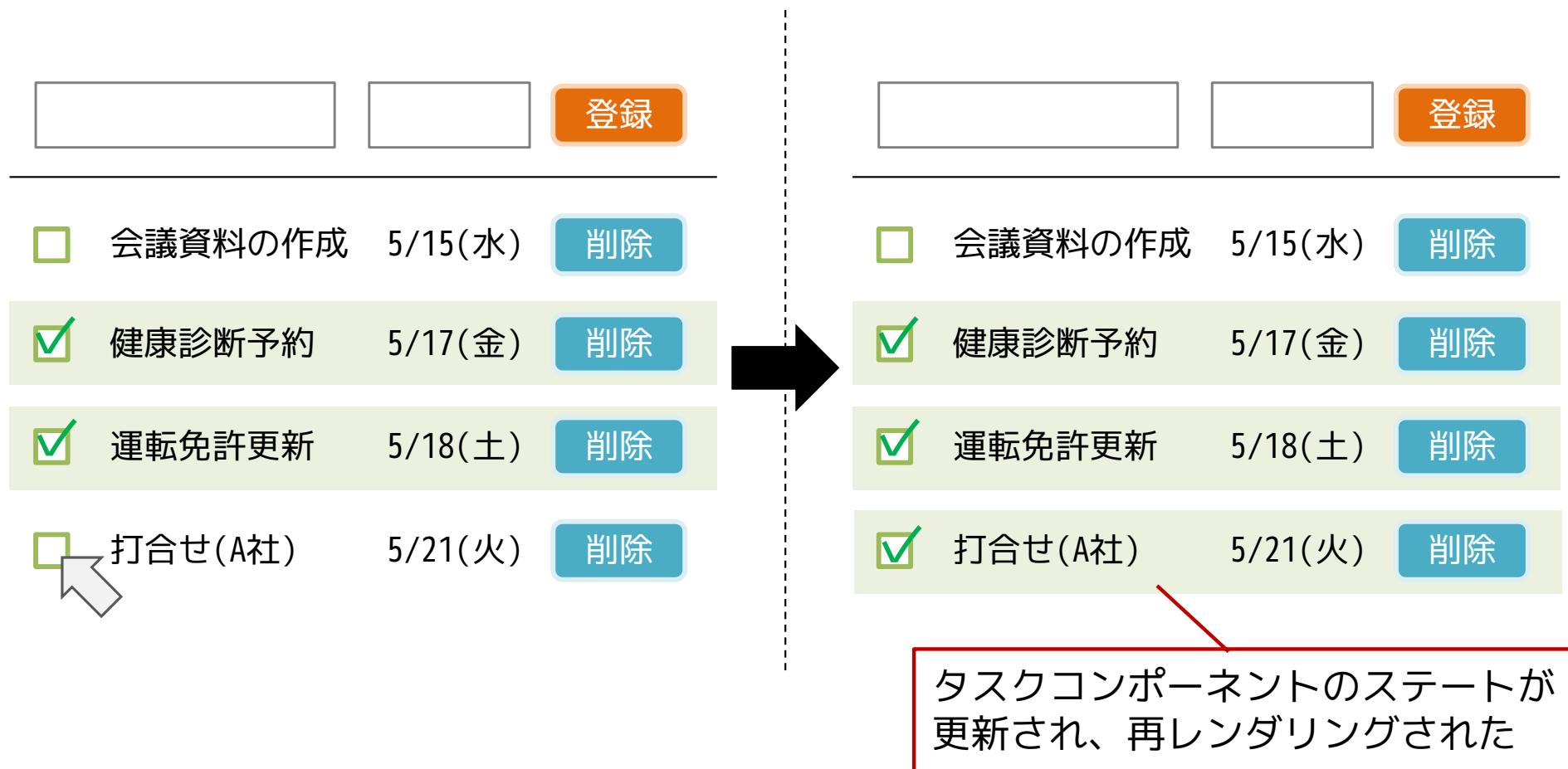
Mountingの例

- タスクが登録される



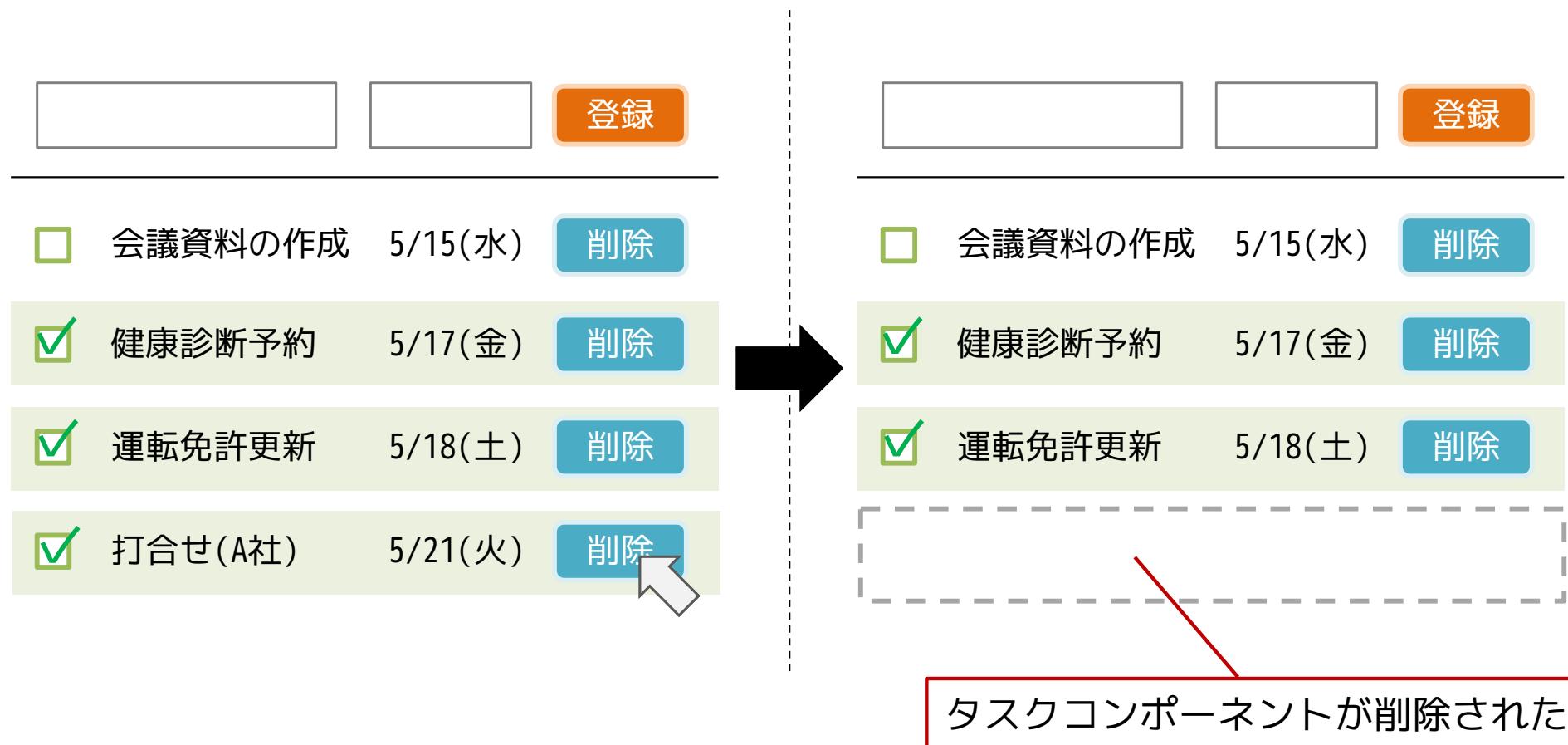
Updatingの例

- タスク完了のチェックを入れる



Unmountingの例

- タスクを削除する



useEffect

- 副作用のある処理やライフサイクルに合わせた処理を記述したい場合に利用する関数
 - 第1引数：コールバック関数。副作用のある処理を記述
 - 第2引数：依存する変数の配列。初回レンダリング後と、ここに列挙された変数に変更がある度に、useEffectは実行される

```
useEffect(() => {
```

初回レンダリング時、及び依存する変数に
変更がある度に実行される

※開発時は初回レンダリング前に、Reactによってテスト実行される

①副作用のある処理

(Ajax通信でデータを取得し、ステートを更新する等)

```
return () => {
```

②クリーンナップ・コード
(タイマー関数の終了など)

```
}
```

```
}, [③依存する変数を列挙]);
```

省略可

①の実行直前、またはコンポーネント
がアンマウントされる際に実行される

空の配列の場合、useEffectは1回だけ実行される
この引数自体を省略すると、レンダリング毎に実行される

空の配列の警告

- useEffectの第2引数を空の配列にすると生じる警告を抑制するには、以下の設定を記述する

eslint.config.jsへの追記

```
rules: {
  ...js configs.recommended.rules,
  ...react configs.recommended.rules,
  ...react configs['jsx-runtime'].rules,
  ...react Hooks configs.recommended.rules,
  ''react-hooks/exhaustive-deps': 0,
  'react/jsx-no-target-blank': 'off',
  'react-refresh/only-export-components': [
    'warn',
    { allowConstantExport: true },
  ],
},
```



警告が出てくる

useEffectの挙動

- 開発時の環境では、useEffectが2回実行される
 - 開発者が適切なクリーンナップ関数を実装しているかなどを確認するストレステストの目的として機能する
 - クリーンナップ関数を適切に記述していないと、イベントの重複やメモリリークなどの問題が起こるが、それを開発段階で検出するという目的がある
- 本番用にビルドした場合やStrict Modeを無効化した場合には、1回のみの実行になる

main.jsx

```
<StrictMode>
  <App />
</StrictMode>
```

StrictModeを無効化する場合は削除

useEffectの例 1

例：金額の配列ステートが更新されたら、
合計金額のステートを更新する

```
// 金額の配列と合計金額のステート
const [priceList, setPriceList] = useState([]);
const [totalPrice, setTotalPrice] = useState(0);

useEffect(() => {
  // 合計金額の算出
  const total = priceList.reduce((accum, price) => accum + price, 0);
  // 合計金額のステートを更新
  setTotalPrice(total);
}, [priceList]);
```

priceListステートが更新されると、
useEffectが実行される

useEffectの例 2

例：Web APIからAjax通信でデータを取得して、ステートを更新する

```
// ローディングとTodoリストのステート
const [isLoading, setIsLoading] = useState(true);
const [todoList, setTodoList] = useState([]);

useEffect(() => {
    // Ajax通信で、Todoリストを取得する関数
    const getTodoList = async () => {
        const url = "https://jsonplaceholder.typicode.com/todos";
        const res = await fetch(url);
        const data = await res.json();
        setTodoList(data);
        setIsLoading(false);
    };

    getTodoList();
}, []);
```

コンポーネント描画時の1回だけ実行する

練習

- 練習03-2

カスタムフック

カスタムフックの作成

- カスタムフックとは、名前が「use」で始まり、他のフックを呼び出せる関数
 - 基本的には、1つのフック関数につき、1つのファイルを作成
⇒ ファイル名は「関数名.js」とし、hooksという名前のフォルダに配置する
 - どのような引数を取って何を返すかについて制限はない
- カスタムフックを定義することで、フック関数を含む処理を関数化でき、再利用しやすくなる
- カスタムフックの例:
 - 引数として渡されたAPIのURLからデータを取得する関数…など

カスタムフックの例

例：引数として渡されたAPIのURLからデータを取得するフック関数

```
export default function useFetchData(apiUrl) {
  const [isLoading, setIsLoading] = useState(true);
  const [data, setData] = useState(null);

  useEffect(() => {
    // Ajax通信で、APIからデータを取得する関数
    const getData = async () => {
      const res = await fetch(apiUrl);
      const data = await res.json();
      setData(data);
      setIsLoading(false);
    };

    getData(); // 上記関数の実行
  }, []);

  return { data, isLoading }; // 取得したデータとローディングの情報を返す
}
```

練習

- 練習03-3