コンポーネントに関数を渡す

コンポーネントに（onClick のような）イベントハンドラを渡すには？
イベントハンドラやその他の関数を props として、子コンポーネントに渡してください。

```
<button onClick={this.handleClick}>
```

ハンドラ内で親コンポーネントにアクセスする必要がある場合は、関数をコンポーネントインスタンスにバインドする必要があります（以下を参照）。

関数をコンポーネントインスタンスにバインドするには？
使用する構文やビルドステップにより、 `this.props` や `this.state` のようなコンポーネントの属性に、関数がアクセスできるようにする方法がいくつかあります。

コンストラクタでバインドする (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

クラスプロパティ (Stage 3 Proposal)

```
class Foo extends Component {
  // Note: this syntax is experimental and not standardized yet.
  handleClick = () => {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

レンダー内でバインドする

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

> **補足：**
> レンダー内で `Function.prototype.bind` を利用すると、コンポーネントがレンダーされるたびに新しい関数が作成され、パフォーマンスに影響を与える可能性があります（下記参照）。

レンダー内でアロー関数を使用する

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
```

```
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

> **注意：**
> レンダー内でアロー関数を利用するとコンポーネントがレンダーされるたびに新しい関数が作成され、パフォーマンスに影響を与える可能性があります（下記参照）。

レンダー内でアロー関数を使用しても良いのか？
一般的には問題ありません。コールバック関数にパラメータを渡すのに最も簡単な方法です。
しかしパフォーマンス上の問題が出た際には、ぜひ最適化しましょう！

そもそもバインドはなぜ必要なのか？
JavaScript において、以下の 2 つのコードは同等 **ではありません**。

```
obj.method();
```

```
var method = obj.method;
method();
```

メソッドはバインドすることで、2 つ目のコードが 1 つ目と同様に動作するようになります。
React では、一般的に他のコンポーネントに **渡す** メソッドしかバインドする必要はありません。たとえば、`<button onClick={this.handleClick}>` というコードは、`this.handleClick` を渡しているので、バインドする必要があります。しかし、`render` メソッドやライフサイクルメソッドをバインドする必要はありません。それらは他のコンポーネントに渡すことがないからです。
Yehuda Katz の記事 はバインドとは何か、また JavaScript における関数の動作について詳細に説明してくれています。

どうしてコンポーネントをレンダーするたびに、関数が呼ばれるのか？
関数をコンポーネントに渡すときにその関数を**呼び出していない**ことを確認してください：

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

代わりに、（カッコなしの）**関数自体を返すようにしてください**：

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

イベントハンドラやコールバックにパラメータを渡すには？
イベントハンドラをラップするアロー関数を作成してパラメータを渡すことができます：

```
<button onClick={() => this.handleClick(id)} />
```

これは `.bind` を呼び出すのと同じことです：

```
<button onClick={this.handleClick.bind(this, id)} />
```

例：アロー関数を利用してパラメータを渡す

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
```

```
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}
```

例：データ属性を利用してパラメータを渡す

ほかにも、DOM API をイベントハンドラに必要なデータの保存に使用することができます。大量の要素を最適化したり、React.PureComponent による等価性チェックに依存したレンダーツリーがある場合に、この方法を検討してください。

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}
```

関数があまりに頻繁にあるいは連続して呼ばれる場合の対処法は？

`onClick` や `onScroll` のようなイベントハンドラがあって、それが高頻度で呼び出されるのを防ぎたい場合、コールバックが実行される頻度を制御するこができます。これは以下の方法で可能です：

- **スロットル**：時間ベースの頻度に基づくサンプルの制御（例：`_.throttle`）
- **デバウンス**：一定時間アクティブでなかった場合の出力の制御（例：`_.debounce`）
- **requestAnimationFrame スロットル**：`requestAnimationFrame` に基づくサンプルの制御 (例：`raf-schd`)

スロットルとデバウンスの比較はこのビジュアルデモ をご覧ください。

> **補足：**
>
> `_.debounce`、`_.throttle`、`raf-schd` は遅延されるコールバックの呼び出しをキャンセルするための `cancel` メソッドを提供します。`componentWillUnmount` でこのキャンセルメソッドを呼び出すか、**あるいは**遅延されるコールバック関数内でコンポーネントがまだマウントされていることを確認するようにしてください。

スロットル

スロットルは、特定の時間枠内に関数が複数回呼ばれるのを防ぎます。以下の例では、「クリック」ハンドラをスロットルして、1 秒間に 2 回以上呼ばれるのを防ぎます。

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }
```

```
  componentWillUnmount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

デバウンス

デバウンスは、関数が最後に呼ばれてから一定時間経過するまで実行されないようにします。これは、高頻度で発生するイベント（例：スクロールやキーボードイベントなど）に応じた高コストな計算処理が必要なときに役立ちます。下記の例は、250 ミリ秒の遅延でテキスト入力をデバウンスします。

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    // React pools events, so we read the value before debounce.
    // Alternately we could call `event.persist()` and pass the entire event.
    // For more info see reactjs.org/docs/events.html#event-pooling
    this.emitChangeDebounced(e.target.value);
  }

  emitChange(value) {
    this.props.onChange(value);
  }
}
```

`requestAnimationFrame` スロットル

`requestAnimationFrame` は描画パフォーマンスが最適化されるタイミングでブラウザで実行される関数をキューに入れる方法です。`requerstAnimationFrame` でキューに入れられている関数は、次のフレームで実行されます。ブラウザは、毎秒 60 フレーム (60fps) を確保するように努めます。しかし、それができなかった場合には、自然に 1 秒間のフレーム数が **制限** されます。たとえば、デバイスが 30 fps しか処理できなければ、1 秒間に 30 フレームしか取得できません。`requestAnimationFrame` スロットルを使うことで、1 秒間に 60 回以上になる更新を防ぐことができます。1 秒間に 100 回の更新をする場合、ユーザーには確認できない追加作業をブラウザが作成することになります。

> **注意：**
> このテクニックにより、フレーム内の最後に公開された値のみがキャプチャされます。この最適化がどのように動作するのかについての例は `MDN` で確認できます。

```
import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }
```

```
  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        <img src="/my-huge-image.jpg" />
      </div>
    );
  }
}
```

レート制限をテストする

レート制限コードを正しくテストするとき、早送り機能があると便利です。もし `jest` を使っているのであれば、`mock timers` を使って時間を早送りすることができます。`requestAnimationFrame` スロットルを使っているのであれば、`raf-stub` がアニメーションフレームの間隔を制御するのに役立つでしょう。

このページを編集する