React

セットアップ

ブラウザ向けのクイックスタートで、すでにReactアプリケーションの開発のセットアップの仕方を説明しています。要点をまとめると、次の通りです。

- ファイル拡張子 `.tsx` (`.ts` の代わりに)を使用してください。
- `tsconfig.json` の `compilerOptions` で `"jsx" : "react"` を使ってください。
- JSXとReactの定義をあなたのプロジェクトにインストールします：(`npm i -D @types/react @types/react-dom`)。
- reactを `.tsx` ファイルにインポートします(`import React from "react"` となります)。

ReactとTypeScriptのプロジェクトをセットアップする一番簡単な方法は、Create React Appを使うことです。このツールはReactチームから提供されているツールです。公式にTypeScriptのテンプレートが提供されています。Create React Appを使うには、 `$ npm i -g create-react-app` というコマンドでローカルPCにグローバルにインストールできます。そうすれば、Reactの公式Webサイトに書かれているように `$ npx create-react-app my-app --template typescript` というコマンドで、最初からTypeScriptを利用可能なプロジェクトを簡単に作成できます。

HTMLタグ vs Component

Reactは、HTMLタグ(文字列)または、React Component(クラス)を表示できます。これらのJavaScriptへのコンパイル結果は同じではありません(例えば、、 `React.createElement('div')` と `React.createElement(MyComponent)` のJavaScriptのコンパイル結果に同じではありません)。どちらとして扱われるかは、最初の文字のケース(大文字小文字)です。 `foo` はHTMLタグとして扱われます。そして、 `Foo` はコンポーネントとして扱われます。

型チェック(Type Checking)

HTMLタグ

HTMLタグ `foo` の型は `JSX.IntrinsicElements.foo` です。これらの主要な型は、セットアップの一部としてインストールした型定義ファイル `react-jsx.d.ts` の中で、予め定義されています。下記にその中に含まれる型定義のサンプルを示します。

```
declare module JSX {
    interface IntrinsicElements {
        a: React.HTMLAttributes;
        abbr: React.HTMLAttributes;
        div: React.HTMLAttributes;
        span: React.HTMLAttributes;

        /// などなど
    }
}
```

Functionコンポーネント(Functional Components)

単に `React.FC` インターフェースを使ってステートレスなFunctionコンポーネントを定義することができます。 `React.FC` は単に、 `React.FunctionComponent` の短いバージョンです。

```
type Props = {
  foo: string;
}
const MyComponent: React.FC<Props> = (props) => {
    return <span>{props.foo}</span>
}

<MyComponent foo="bar" />
```

クラスコンポーネント(Class Components)

コンポーネントは、 `props` プロパティに基づいて型チェックされます。コンポーネントは、JSXが、どのように変換されるか、ということを考慮して型チェックされます。例えばJSXのタグの属性は、コンポーネントの `props` の一部である必要があります。

Reactのステートフルなコンポーネントを作成するには、クラスを使用します。型定義ファイル `react.d.ts` は、 `React.Component<Props,State>` クラスを定義しています。このクラスを、コ

ンポーネントを作成するときに継承します。このクラスは、そのコンポーネント独自の `Props` と `State` の型をジェネリクスとして設定できるようになっています。これを以下の例で示します(下記の例では、Stateには空のオブジェクト型を設定しています):

```
type Props = {
  foo: string;
}
class MyComponent extends React.Component<Props, {}> {
    render() {
        return <span>{this.props.foo}</span>
    }
}

<MyComponent foo="bar" />
```

React JSX のヒント： レンダリング可能なインターフェース

Reactは `JSX` や `string` をレンダリングすることができます。これらはすべて `React.ReactNode` 型に統合されていますので、レンダリング可能なものを受け入れる場合などに使用してください。

```
type Props = {
  header: React.ReactNode;
  body: React.ReactNode;
}
class MyComponent extends React.Component<Props, {}> {
    render() {
        return <div>
            {this.props.header}
            {this.props.body}
        </div>;
    }
}

<MyComponent header="これはヘッダーです" body="これはボディです" />
```

React JSX のヒント：コンポーネントのインスタンスを受け入れる

Reactの型定義ファイルは、 `React.ReactElement<T>` という型を提供しています。これを、 `<T/>` クラスコンポーネントのインスタンスを保持できる型アノテーションに利用できます。例えば以下の通りです:

```
class MyAwesomeComponent extends React.Component {
  render() {
    return <div>こんにちは</div>;
  }
}

const foo: React.ReactElement<MyAwesomeComponent> = <MyAwesomeComponent />; //
const bar: React.ReactElement<MyAwesomeComponent> = <NotMyAwesomeComponent />;
```

> もちろん、これを関数の引数や、コンポーネントのPropsの型として使用することもできます。

React JSXのヒント： propsを受け取り、JSXでレンダリングできるコンポーネントを受け入れる

型 `React.Component<Props>` は `React.ComponentClass<P> | React.StatelessComponent<P>` のように２つの種類のコンポーネントの型を統合しています。なので、 `Props` 型を受け取り、JSXを使ってレンダリングする何かを受け入れることができます。

```
const X: React.Component<Props> = foo; // どこかから渡されたコンポーネント

// XコンポーネントにPropsを渡して表示:
<X {...props}/>;
```

React JSXのヒント： ジェネリックコンポーネント

これは、期待どおりに動作します。以下に例を示します:

```
/** ジェネリックコンポーネント */
type SelectProps<T> = { items: T[] }
class Select<T> extends React.Component<SelectProps<T>, any> { }

/** 使い方 */
const Form = () => <Select<string> items={['a','b']} />;
```

ジェネリック関数

次のようなものがうまくいきます：

```
function foo<T>(x: T): T { return x; }
```

しかし、アロー関数のジェネリック関数を利用しようとすると、構文エラーになります。

```
const foo = <T>(x: T) => x; // ERROR : `T` タグが閉じられていません
```

回避策：ジェネリックのパラメータに `extends` を使用すると、コンパイラがジェネリックであることを教えられます。 `extends` は、ジェネリックの型に制約を付けるためのキーワードです。 `{}` を `extends` する、ということは、オブジェクトであれば、何でも良い、ということです。なので、単に構文エラーを回避するために、このようにすることができます:

```
const foo = <T extends {}>(x: T) => x;
```

Reactのヒント: 厳密に型付けされたRef

基本的にRefの変数をユニオン型で定義することによって、 `null` で初期化できます。そして、それをコンポーネントのrefプロパティに渡したコールバック関数で初期化できます。

```
class Example extends React.Component {
  example() {
    // ... 何らかの処理
  }

  render() { return <div>Foo</div> }
}


class Use {
  exampleRef: Example | null = null;

  render() {
    return <Example ref={exampleRef => this.exampleRef = exampleRef } />
  }
}
```

これは、DOM要素の参照を保持する変数についても同じです。

```
class FocusingInput extends React.Component<{ value: string, onChange: (value:
  input: HTMLInputElement | null = null;

  render() {
    return (
      <input
        ref={(input) => this.input = input}
        value={this.props.value}
        onChange={(e) => { this.props.onChange(e.target.value) } }
        />
      );
    }
    focus() {
      if (this.input != null) { this.input.focus() }
    }
}
```

型アサーション

既に説明した のように、型アサーションには `as Foo` 構文を使います。

デフォルトProps

- デフォルト値のあるPropsを持ったステートフルなコンポーネント： *Null*アサーション演算子 `?` を使って、デフォルト値のあるPropsを定義できます。このPropsには、(外部からReactの仕組みによって)値が渡されます(これが最も理想的なものとは限りませんが、今思いつく限りでは、シンプルでミニマムなコードです)。

```
class Hello extends React.Component<{
  /**
   * @default 'TypeScript'
   */
  compiler?: string,
  framework: string
}> {
  static defaultProps = {
    compiler: 'TypeScript'
```

```
  }
  render() {
    const compiler = this.props.compiler!;
    return (
      <div>
        <div>{compiler}</div>
        <div>{this.props.framework}</div>
      </div>
    );
  }
}

ReactDOM.render(
  <Hello framework="React" />, // TypeScript React
  document.getElementById("root")
);
```

- デフォルトPropsのあるFunctionコンポーネント ：シンプルなJavaScriptの構文を活用して、TypeScriptの型システムとうまく組み合わせることをお勧めします。下記はその例です:

```
const Hello: React.FC<{
  /**
   * @default 'TypeScript'
   */
  compiler?: string,
  framework: string
}> = ({
  compiler = 'TypeScript', // デフォルト値のあるProps
  framework
}) => {
    return (
      <div>
        <div>{compiler}</div>
        <div>{framework}</div>
      </div>
    );
  };


ReactDOM.render(
  <Hello framework="React" />, // TypeScript React
  document.getElementById("root")
);
```

WebComponentの宣言

もしWebComponentを利用している場合、それはReactの型定義ファイル(`@types/react`)には定義されていません。しかし、アンビエント宣言(`declare`)を使って簡単に定義できます。例えば、 `my-awesome-slider` というWebComponentがあるとします。これは、 `MyAwesomeSliderProps` を受け取ります。この場合、以下のようになります:

```
declare global {
  namespace JSX {
    interface IntrinsicElements {
      'my-awesome-slider': MyAwesomeSliderProps;
    }

    interface MyAwesomeSliderProps extends React.Attributes {
      name: string;
    }
  }
}
```