コード分割

バンドル
多くの React アプリケーションは、Webpack や Browserify などの ツールを使ってファイルを「バンドル」しています。 バンドルはインポートされたファイルをたどって、それらを 1 つのファイルにまとめるプロセスです。 このバンドルされたファイルを Web ページ内に置くことによって、アプリ全体を一度に読み込むことができます。

例
App:

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle:

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

> **補足：**
> 実際のバンドルはこれとは大幅に異なった見た目になります。

もしあなたが Create React App や Next.js, Gatsby またはこれらに類するツールを使用している場合、 アプリケーションをバンドルするための Webpack の設定が、追加の設定なしにすぐに手に入るでしょう。
そうでない場合は、自分でバンドルを設定する必要があります。 設定方法に関しては、Webpack のドキュメントにある Installation や Getting Started などを 参照してみてください。

コード分割
バンドルは確かに素晴らしいですが、アプリが大きくなるにつれて、バンドルのサイズも大きくなります。 特にサイズの大きなサードパーティ製のライブラリを含む場合は顕著にサイズが増大します。 不用意に大きなバンドルを作成してしまいアプリの読み込みに多くの時間がかかってしまうという事態にならないためにも、 常に注意を払い続けなければなりません。
大きなバンドルを不注意に生成してしまわないように、あらかじめコードを「分割」して問題を回避しましょう。 Code-Splitting は、実行時にロードする複数のバンドルを生成できる Webpack や Browserify（factor-bundle を使用）などのバンドラによってサポートされている機能です。
コード分割は、ユーザが必要とするコードだけを「遅延読み込み」する手助けとなり、 アプリのパフォーマンスを劇的に向上させることができます。 アプリの全体的なコード量を減らすことはできませんが、ユーザが必要としないコードを読み込まなくて済むため、 初期ロードの際に読む込むコード量を削減できます。

`import()`
コード分割をアプリに導入する最も良い手段は動的な `import()` 構文を使用することです。
Before:

```
import { add } from './math';

console.log(add(16, 26));
```

After:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

補足：
`import()` 構文は ECMAScript (JavaScript) における提案中の構文であり、 現時点ではまだ言語標準となっていません。 近い将来での標準化が期待されています。

Webpack がこの構文を見つけると、自動的にアプリのコードを分割します。 Create React App を使用している場合はすでに設定がされているため、すぐに使用を開始することができます。 Next.js も同様です。
もし Webpack を自分でセットアップしていた場合には、Webpack のコード分割に関するガイドを読むと良いでしょう。 きっとあなたの Webpack の設定はだいたいこのようになると思います。
もし Babel を使用している場合は、Babel が動的インポート構文をパースできても変換してしまわないようにする必要があります。 そのためには babel-plugin-syntax-dynamic-import を利用すると良いでしょう。

`React.lazy`

補足:
`React.lazy` と Suspense はまだサーバーサイドレンダリングには使用できません。サーバーサイドでレンダリングされたアプリでコード分割をしたい場合には、Loadable Components の使用をおすすめします。こちらはサーバーサイドレンダリングでのバンドル分割のための素晴らしいガイドも提供していくれているので、参考にしてみてください。

`React.lazy` 関数を使用すると、動的インポートを通常のコンポーネントとしてレンダリングすることができます。
Before:

```
import OtherComponent from './OtherComponent';

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

After:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

このコンポーネントがレンダリングされる際には、`OtherComponent` を含むバンドルを自動的にロードしてくれます。
`React.lazy` は動的インポート構文 `import()` を呼び出す関数を引数として取ります。この関数は React コンポーネントを含む `default` export を持つモジュールに解決される `Promise` を返さなければなりません。

Suspense

`MyComponent` がレンダリングされるまでに、`OtherComponent` を含むモジュールがまだロードされていない場合、例えばロードインジケータなどのようなフォールバックコンテンツをロードが完了するまで表示する必要があります。これは `Suspense` コンポーネントを使って実装することができます。

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

`fallback` プロパティはコンポーネントがロードされるのを待っている間に表示したいあらゆる React 要素を受け取ります。`Suspense` コンポーネントは遅延コンポーネントより上位のどこにでも配置することができます。また、複数の遅延コンポーネントを単一の `Suspense` コンポーネントでラップすることもできます。

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
```

```
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

Error Boundary
もし他のモジュールがロードに失敗した場合（例えば、ネットワークの障害など）、エラーが発生します。その際には error boundary を使用することによってこれらのエラーをハンドリングし、エラーの回復やユーザ体験の向上に繋げることができます。error boundary を作成したら、遅延コンポーネントより上位のあらゆる場所で使用でき、ネットワークエラーが発生した際にエラー内容を表示することができます。

```
import MyErrorBoundary from './MyErrorBoundary';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

ルーティング単位でのコード分割
アプリ内のどこにコード分割を導入するかを決めるのは少し面倒です。 バンドルを均等に分割する場所を確実に選択したいところですが、ユーザ体験を妨げてはなりません。
コード分割を導入するにあたって適している場所はルーティングです。 Web を使用するほとんどの人は、多少のロード時間がかかるページ遷移に慣れています。 また、ユーザがページ上の他の要素を同時に操作する可能性を減らすよう、ページ全体を一度に再レンダーすることが多いでしょう。
これは React Router のようなライブラリを使ったアプリに `React.lazy` を使用することで ルーティングベースのコード分割を導入する方法の例です。

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

名前付きエクスポート
`React.lazy` は現在デフォルトエクスポートのみサポートしています。インポートしたいモジュールが名前付きエクスポートを使用している場合、それをデフォルトとして再エクスポートする中間モジュールを作成できます。これにより、tree shaking が機能し未使用のコンポーネントを取り込まず済むようにできます。

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

このページを編集する