差分検出処理

React は、各更新で実際に何が変更されるべきかを人間が心配する必要がないように、宣言型の API を提供しています。これによりアプリケーションの作成が大幅に容易になるわけですが、React の中でこの処理がどのように実装されているのかはよく分からないかもしれません。この章では React の “差分” アルゴリズムについて、コンポーネントの更新を予測可能なものとしながら、ハイパフォーマンスなアプリケーションの要求を満たす速度を得られるように、私たちが行った選択について説明します。

何が問題なのか

React を使う際、 `render()` 関数をある時点の React 要素のツリーを作成するものとして考えることができます。次回の state や props の更新時には、 `render()` 関数は React 要素の別のツリーを返します。React はそこから直近のツリーに合致させるように効率よく UI を更新する方法を見つけ出す必要があります。
あるツリーを別のものに変換するための最小限の操作を求めるというアルゴリズム問題については、いくつかの一般的な解決方法が存在しています。しかし、最新のアルゴリズムでもツリーの要素数を n として $O(n^3)$ ほどの計算量があります。
React でそのアルゴリズムを使った場合、1000 個の要素を表示するのに 10 億といったレベルの比較が必要となります。これではあまりに計算コストが高すぎます。代わりに、React は 2 つの仮定に基づくことで、ある程度近い結果を得ることができる O(n) ほどの計算量のアルゴリズムを実装しています。

1. 異なる型の 2 つの要素は異なるツリーを生成する。
2. 開発者は key プロパティを与えることで、異なるレンダー間でどの子要素が変化しない可能性があるのかについてヒントを出すことができる。

実際に、これらの仮定はほとんど全ての実践的なユースケースで有効です。

差分アルゴリズム

2 つのツリーが異なっている場合、React は最初に 2 つのルート要素を比較します。そのふるまいはルート要素の型に応じて異なります。

異なる型の要素

ルート要素が異なる型を持つ場合は常に、React は古いツリーを破棄して新しいツリーをゼロから構築します。`<a>` から `<img>` へ、もしくは `<Article>` から `<Comment>` へ、もしくは `<Button>` から `<div>` へ — それらの全てがツリーをゼロから再構築させるのです。
ツリーを破棄する時点で、古い DOM ノードは破棄されます。コンポーネントのインスタンスは `componentWillUnmount()` を受け取ります。新しいツリーを構築する時点で、新しい DOM ノードが DOM に挿入されます。コンポーネントのインスタンスは `componentWillMount()` とそれから `componentDidMount()` を受け取ります。古いツリーに関連付けられた全ての state は失われます。ルート配下のコンポーネントはアンマウントされ、それらの state は破棄されます。例えば、以下のように異なる場合：

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

古い `Counter` は破棄され、新しいものが再マウントされます。

同じ型の DOM 要素

同じ型の 2 つの React DOM 要素を比較した場合、React はそれぞれの属性を調べ、対応する共通の DOM ノードを保持し、変更された属性のみを更新します。例えば：

```
<div className="before" title="stuff" />

<div className="after" title="stuff" />
```

これらの 2 つの要素を比べた場合、React は対応する DOM ノードの `className` のみを更新すればよいと分かります。
`style` を更新した場合は、React は同様に変更されたプロパティのみを更新すればよいと分かります。例えば：

```
<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />
```

2 つの要素を変換する場合、React は `fontWeight` ではなく `color` のみを変更すればよいことが分かります。
この DOM ノードを処理した後、React は子に対して再帰的に処理を行っていきます。

同じ型のコンポーネント要素

コンポーネントが更新される場合、インスタンスは同じままとなり、レンダー間で state は保持されます。React は対応するコンポーネントのインスタンスの props を新しい要素に合うように更新し、`componentWillReceiveProps()` と `componentWillUpdate()` を対応するインスタンスに対して呼び出します。
次に、 `render()` メソッドが呼ばれ、差分アルゴリズムが再帰的に前の結果と新しい結果を処理します。

子要素の再帰的な処理

デフォルトでは、DOM ノードの子に対して再帰的に処理を行う場合、React は単純に、両方の子要素リストのそれぞれ最初から同時に処理を行っていって、差分を見つけたところで毎回更新を発生させます。
例えば、子ノードの最後にひとつ要素を追加するような場合、以下の 2 つのツリー間の変換はうまく動作します：

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React は 2 つの `<li>first</li>` ツリーを一致させ、2 つの `<li>second</li>` ツリーを一致させ、最後に `<li>third</li>` ツリーを挿入します。
それを単純に実行した場合、先頭への要素の追加はパフォーマンスが悪くなります。

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React は `<li>Duke</li>` と `<li>Villanova</li>` サブツリーをそのまま保てるということに気づくことなく、すべての子要素を変更してしまいます。この非効率性は問題になることがあります。

Keys

この問題を解決するため、React は `key` 属性をサポートします。子要素が key を持っている場合、React は key を利用して元のツリーの子要素と次のツリーの子要素を対応させます。例えば、 `key` を前出の非効率な例に追加することで、ツリーの変換を効率的なものにすることができます。

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

これで React は、 `'2014'` の key を持つ要素が新規の要素であり、 `'2015'` と `'2016'` の key をもつ要素は移動しただけだ、と理解するようになります。
実際に、key を探すのはたいてい難しくありません。表示しようとしている要素は既に固有の ID を持っているかもしれないので、key をそのデータから設定するだけです。

```
<li key={item.id}>{item.name}</li>
```

そうではない場合には、新しい ID プロパティをモデルに追加するか、key を生成するためにコンテンツの一部をハッシュ化します。key は兄弟要素間で一意であればよく、グローバルに一意である必要はありません。
最後の手段として、配列の要素のインデックスを key として渡すことができます。項目が並び替えられることがなければうまく動作しますが、並び替えられると遅くなります。
key として配列のインデックスが使用されている場合、並べ替えはコンポーネントの状態に関しても問題を起こすことがあります。コンポーネントのインスタンスは key に基づいて更新、再利用されます。インデックスが key の場合、要素の移動はインデックスの変更を伴います。結果として、非制御の入力などに対するコンポーネントの状態が混乱し、予期せぬ形で更新されてしまうことがあります。CodePen に配列のインデックスを key として使うことで生じる問題についての例があります。また、こちらが同じ例の更新版であり、配列のインデックスを使わないことで、ソートや並び替え、要素の先頭への追加にまつわる問題がどのように解決されるのかを示しています。

トレードオフ

この差分検出処理アルゴリズムは内部の実装の詳細であることに気をつけておく事が重要です。React はアクション毎にアプリケーション全体を再レンダーし得るものです。最終結果はいずれにせよ同じでしょう。誤解のないように言っておくと、ここでの再レンダーとは全てのコンポーネントに対して `render` メソッドを呼び出すことであり、React がそれらをアンマウントして再びマウントすることではありません。前述のルールに従って変更を適用するだけです。
私達は一般的なユースケースで高速となるように、定期的にヒューリスティクスを改善しています。現時点の実装ではサブツリーが兄弟要素の間で移動したということは表現できますが、それ以外のどこか別の場所に移動したということは伝えることはできません。結果的にアルゴリズムはサブツリーを再レンダーします。

React はヒューリスティクスに依存するため、その背後にある仮定に合致しなければ、パフォーマンスが低下します。

1. 差分アルゴリズムは型が異なるコンポーネント間でサブツリーを照合しようとはしません。2 つのとてもよく似た出力をするコンポーネントの型を入れ替えているケースがあれば、同じ型にした方がいいかもしれません。現実的には、これが問題となったことはありません。
2. key は予測可能で安定した、一意なものであるべきです。不安定な key（例えば `Math.random()` 関数で生成するような）は多くのコンポーネントのインスタンスと DOM ノードを不必要に再生成し、パフォーマンスの低下や子コンポーネントの state の喪失を引き起こします。

このページを編集する