Reconciliation

React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React. This article explains the choices we made in React's "diffing" algorithm so that component updates are predictable while being fast enough for high-performance apps.

Motivation

When you use React, at a single point in time you can think of the `render()` function as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the state of the art algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic O(n) algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

In practice, these assumptions are valid for almost all practical use cases.

The Diffing Algorithm

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

Elements Of Different Types

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from `<a>` to `<img>`, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive `componentWillUnmount()`. When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive `componentWillMount()` and then `componentDidMount()`. Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

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

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

This will destroy the old `Counter` and remount a new one.

DOM Elements Of The Same Type

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

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

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

By comparing these two elements, React knows to only modify the `className` on the underlying DOM node.

When updating `style`, React also knows to update only the properties that changed. For example:

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

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

When converting between these two elements, React knows to only modify the `color` style, not the `fontWeight`.

After handling the DOM node, React then recurses on the children.

Component Elements Of The Same Type

When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls `componentWillReceiveProps()` and `componentWillUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

Recursing On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

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

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

React will match the two `<li>first</li>` trees, match the two `<li>second</li>` trees, and then insert the `<li>third</li>` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

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

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

React will mutate every child instead of realizing it can keep the `<li>Duke</li>` and `<li>Villanova</li>` subtrees intact. This inefficiency can be a problem.

Keys

In order to solve this issue, React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a `key` to our inefficient example above can make the tree conversion efficient:

```
<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>
```

Now React knows that the element with key `'2014'` is the new one, and the elements with the keys `'2015'` and `'2016'` have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

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

When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Reorders can also cause issues with component state when indexes are used as keys. Component instances are updated and reused based on their key. If the key is an index, moving an item changes it. As a result, component state for things like uncontrolled inputs can get mixed up and updated in unexpected ways.

Here is an example of the issues that can be caused by using indexes as keys on CodePen, and here is an updated version of the same example showing how not using indexes as keys will fix these reordering, sorting, and prepending issues.

Tradeoffs

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling `render` for all components, it doesn't mean React will unmount and remount them. It will only apply the differences following the rules stat
the previous sections.

We are regularly refining the heuristics in order to make common use cases faster. In the current implementation, you can express the fact that a subtree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will rerender that full subtree.

Because React relies on heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match subtrees of different component types. If you see yourself alternating between two component types with very similar output, you may want to make it the same type. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

Edit this page