PREV
2. JSX

NEXT
4. Data Flow

Chapter 3. Component Lifecycle

React components are simply state-machines. Their output represents the DOM for a given set of props and state. Throughout a component's lifecycle, as it's props or state change, its DOM representation might change too. As noted in the chapter on JSX, a component is just a Javascript function; for a given input it will always return the same output.

React provides lifecycle hooks for a component to respond to various moments — its creation, lifetime, and teardown. We'll cover them here in order of their appearance — first through instantiation, then through the components life and finally as the component is torn-down.

Lifecycle Methods

React has relatively few lifecycle methods, but they are very powerful. React offers you all of the methods required to control the props and state of your app throughout its lifecycle. Lets take a look at each in order as they get called on your component.

The lifecycle methods called the first time an instance is created, vs each subsequent instance, varies slightly. For your first use of a component class you'll see these methods called, in order:

- `getDefaultProps`
- `getInitialState`
- `componentWillMount`
- `render`
- `componentDidMount`

For all subsequent uses of that component class you will see the following methods called, in order. Notice `getDefaultProps` is no longer in the list.

- `componentWillMount`
- `render`
- `componentDidMount`

As the app state changes and your component is affected you will see the following methods called, in order:

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `render`
- `componentDidUpdate`

And lastly, when you are finished with the component you will see `componentWillUnmount` called, giving your instance the opportunity to clean-up after itself.

Now we'll cover each of these three stages: instantiation, lifetime, and cleanup in turn.

Instantiation

As each new component is created and first rendered there are series of methods you can use to setup and prepare your components. Each of these methods has a specific responsibility, as described here.

getDefaultProps

This method is called only once for the component class. The object returned is used to set the default values for any new instances when they are not specified by the parent component.

It is important to note that any complex values, such as objects and arrays, will be shared across all instances — they are not copied or cloned.

getInitialState

Called exactly once for each instance of your component, here you get a chance to initialize the custom state of each instance. Unlike `getDefaultProps` this method is called once each time an instance is created. At this point you have access to `this.props`.

componentWillMount

Invoked immediately before the initial render. This is the last chance to affect the component state before the `render` method is called.

render

Here you build the virtual DOM that represents your components output. Render is the only required method for a component and has specific rules. The requirements of the render method are as follows:

- The only data it can access is `this.props` and `this.state`
- You can return `null`, `false`, or any React component
- There can only be one top-level component (you cannot return an array of elements)
- It must be *pure*, meaning it does not change the state oc modify the DOM output

The result returned from render is not the actual DOM, but a virtual representation that React will later diff with the real DOM to determine if any changes must me made.

componentDidMount

After the render is successful and the actual DOM has been rendered you can access it inside of `componentDidMount` via `this.getDOMNode()`.

This is the lifecyle hook you will use to access the raw dom. For example, if you need to measure the height of the rendered output, manipulate it using timers, or run a custom jQuery plugin, this is where you'd hook into.

Note this method is not called when running on the server.

Lifetime

At this point your component has been rendered to the user and they can interact with it. Typically this involves one of the event handlers getting triggered by a click, tap or key event. As the user changes the state of a component, or the entire app, the new state flows through the component tree and you get a chance to act on it.

componentWillReceiveProps

The props of a component can change at any moment through the parent component. When this happens `componentWillReceiveProps` is called and you get the opportunity to change the new props object and update the state.

For example, within our sample survey application we have an `AnswerRadioInput` that allows users to toggle a radio input. The parent component is able to change this boolean value and we can respond to it like this, updating our own internal state based on the parent's input props.

```
componentWillReceiveProps: function (nextProps) {
  if(nextProps.checked !== undefined) {
```

```
      });
    }
  }
```

Since this method is called *before* the props are applied to the component you have the opportunity to affect the props as well. As an example, if you required a certain prop to be a boolean you can enforce that here, and the new value will be applied to the component props.

```
componentWillReceiveProps: function(nextProps) {
  // coerce the `checked` value to a boolean
  nextProps.checked = !!nextProps.checked;
}
```

shouldComponentUpdate

React is fast. But you can make it even faster using `shouldComponentUpdate` to optimize exactly when a component renders.

If you are certain that the new props or state will not require your component or any of its children to render, return `false`.

This method is not called during the initial render or after using `forceUpdate`.

By returning false you are telling react to skip calling `render`, and its before and after hooks `componentWillUpdate` and `componentDidUpdate`.

This method is not required and for most purposes you will not need to use it during development. Premature use of this method can lead to subtle bugs, so it's best to wait until you can properly benchmark your bottlenecks before choosing where to optimize.

If your careful to treat state as immutable and only read from props and state in your render method then feel free to override `shouldComponentUpdate` to compare the old props and state to their new replacements.

Another performance-tuning option is the `PureRenderMixin` provided with the React addons. If your component is *pure*, meaning it always renders the same DOM for the same props & state, this mixin will automatically use `shouldComponentUpdate` to shallowly compare props and state, returning false if they match.

componentWillUpdate

Similar to `componentWillMount` this method is triggered immediately before rendering when new props or state have been received.

Note you *cannot* update state or props in this method. You should rely on `componentWillReceiveProps` for updating state during runtime.

componentDidUpdate

Similar to `componentDidMount` this method gives you an opportunity to update the rendered DOM.

Teardown

Once React is done with a component it must be unmounted from the DOM and destroyed. You are provided with a single hook to respond to this moment, performing any cleanup and teardown necessary.

componentWillUnmount

Lastly we have the end of a components life as it's removed from the component heirarchy. This method is called just prior to your component being removed and gives you the chance to clean up. Any custom work you might have done in `componentDidMount`, such as creating timers or adding event listeners should be undone here.

Anti Pattern — Calculated Values as State

Given the possibility of using `getInitialState` to create state from `this.props` it's worth noting an anti-pattern here. React is very concerned with maintaining a single source of truth. Its design makes duplicating the source of truth more obvious, one of React's key strengths.

When considering calculated values derived from props it is considered an anti-pattern to store these as state. For example a component might convert a date to a string representation, or transform a string to uppercase before rendering it. These are not state, and should simply be calculated at render-time.

You can identify this anti-pattern when it's impossible to know inside of your render function if your state value is out-of-sync with the prop it's based from.

```
// Anti-pattern. Calculated values should not be stored as state.
getDefaultProps: function () {
  return {
    date: new Date()
  };
},
getInitialState: function () {
  return {
    day: this.props.date.getDay()
  }
},
render: function () {
  return <div>Day: {this.state.day}</div>;
}
```

The correct pattern is to calculate the values at render-time. This guarantees the calculated value

```
// It's proper to calculate values at render-time.
getDefaultProps: function () {
  return {
    date: new Date()
  };
},
render: function () {
  var day = this.props.date.getDay();
  return <div>Day: {day}</div>;
}
```

However, if your goal is not synchronization, but is to simply initialize the state, then it is proper to use props within `getInitialState`. Just be sure to make your intentions clear, for example prefix the prop with `initial`.

```
getDefaultProps: function () {
  return {
    initialValue: 'some-default-value'
  };
},
getInitialState: function () {
  return {
    value: this.props.initialValue
  };
},
render: function () {
  return <div>{this.props.value}</div>
}
```

React's lifecycle methods provide well-designed hooks into the life of your components. As state machines, each component is designed to output stable, predictable markup throughout it's life.

However no component lives in isolation. As parent components push props into their children, and as those children render their own child components, you must carefully consider how your data flows through the app. How much does each child *really* need to know about? Who owns the app state? This is the subject of our next chapter: Data Flow.