Blogged Answers: A (Mostly) Complete Guide to React Rendering Behavior

Posted on May 17, 2020

#react (https://blog.isquaredsoftware.com//tags/react) #context (https://blog.isquaredsoftware.com//tags/context) #redux (https://blog.isquaredsoftware.com//tags/redux) #greatest-hits (https://blog.isquaredsoftware.com//tags/greatest-hits)

()This is a post in the **Blogged Answers** (/series/blogged-answers) series.

Details on how React rendering behaves, and how use of Context and React-Redux affect rendering

I've seen a lot of ongoing confusion over when, why, and how React will re-render components, and how use of Context and React-Redux will affect the timing and scope of those re-renders. After having typed up variations of this explanation dozens of times, it seems it's worth trying to write up a consolidated explanation that I can refer people to. Note that all this information is available online already, and has been explained in numerous other excellent blog posts and articles, several of which I'm linking at the end in the "Further Information" section for reference. But, people seem to be struggling to put the pieces together for a full understanding, so hopefully this will help clarify things for someone.

Table of Contents

What is "Rendering"?

Rendering is the process of React asking your components to describe what they want their section of the UI to look like, now, based on the current combination of props and state.

Rendering Process Overview

During the rendering process, React will start at the root of the component tree and loop downwards to find all components that have been flagged as needing updates. For each flagged component, React will call either `classComponentInstance.render()` (for class components) or `FunctionComponent()` (for function components), and save the render output.

A component's render output is normally written in JSX syntax, which is then converted to `React.createElement()` calls as the JS is compiled and prepared for deployment. `createElement` returns React *elements*, which are plain JS objects that describe the intended structure of the UI. Example:

```
// This JSX syntax:
return <SomeComponent a={42} b="testing">Text here</SomeComponent>

// is converted to this call:
return React.createElement(SomeComponent, {a: 42, b: "testing"}, "Text Here")

// and that becomes this element object:
{type: SomeComponent, props: {a: 42, b: "testing"}, children: ["Text Here"]}
```

After it has collected the render output from the entire component tree, React will diff the new tree of objects (frequently referred to as the "virtual DOM"), and collects a list of all the changes that need to be applied to make the real DOM look like the current desired output. The diffing and calculation process is known as "reconciliation" (https://reactjs.org/docs/reconciliation.html).

React then applies all the calculated changes to the DOM in one synchronous sequence.

> ***Note:*** *The React team has downplayed the term "virtual DOM" in recent years. Dan Abramov said (https://twitter.com/dan_abramov/status/1066328666341294080?lang=en):*
>
> *I wish we could retire the term “virtual DOM”. It made sense in 2013 because otherwise people assumed React creates DOM nodes on every render. But people rarely assume this today. “Virtual DOM” sounds like a workaround for some DOM issue. But that’s not what React is.*
> *React is “value UI”. Its core principle is that UI is a value, just like a string or an array. You can keep it in a variable, pass it around, use JavaScript control flow with it, and so on. That expressiveness is the point — not some diffing to avoid applying changes to the DOM.*
> *It doesn’t even always represent the DOM, for example `<Message recipientId={10} />` is not DOM. Conceptually it represents lazy function calls: `Message.bind(null, { recipientId: 10 })`.*

Render and Commit Phases

The React team divides this work into two phases, conceptually:

- The "Render phase" contains all the work of rendering components and calculating changes
- The "Commit phase" is the process of applying those changes to the DOM

After React has updated the DOM in the commit phase, it updates all refs accordingly to point to the requested DOM nodes and component instances. It then synchronously runs the `componentDidMount` and `componentDidUpdate` class lifecycle methods, and the `useLayoutEffect` hooks.

React then sets a short timeout, and when it expires, runs all the `useEffect` hooks. This step is also known as the "Passive Effects" phase.

You can see a visualization of the class lifecycle methods in this excellent React lifecycle methods diagram (https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/). (It does not currently show the timing of effect hooks, which is something I'd like to see added.)

> *In React's upcoming "Concurrent Mode" (https://reactjs.org/docs/concurrent-mode-intro.html), it is able to pause the work in the rendering phase to allow the browser to process events. React will either resume, throw away, or recalculate that work later as appropriate. Once the render pass has been completed, React will still run the commit phase synchronously in one step.*

A key part of this to understand is that **"rendering" is not the same thing as "updating the DOM", and a component may be rendered without any visible changes happening as a result**. When React renders a component:

- The component might return the same render output as last time, so no changes are needed
- In Concurrent Mode, React might end up rendering a component multiple times, but throw away the render output each time if other updates invalidate the current work being done

How Does React Handle Renders?

Queuing Renders

After the initial render has completed, there are a few different ways to tell React to queue a re-render:

- Class components:
  - `this.setState()`
  - `this.forceUpdate()`
- Function components:
  - `useState` setters
  - `useReducer` dispatches
- Other:
  - Calling `ReactDOM.render(<App>)` again (which is equivalent to calling `forceUpdate()` on the root component)

Standard Render Behavior

It's very important to remember that:

React's default behavior is that when a parent component renders, React will recursively render *all* child components inside of it!

As an example, say we have a component tree of `A > B > C > D`, and we've already shown them on the page. The user clicks a button in `B` that increments a counter:

- We call `setState()` in `B`, which queues a re-render of B.
- React starts the render pass from the top of the tree
- React sees that `A` is not marked as needing an update, and moves past it
- React sees that `B` is marked as needing an update, and renders it. `B` returns `<C />` as it did last time.
- `C` was *not* originally marked as needing an update. However, because its parent `B` rendered, React now moves downwards and renders `C` as well. C returns `<D />` again.
- `D` was also not marked for rendering, but since its parent `C` rendered, React moves downwaard and renders `D` too.

To repeat this another way:

Rendering a component will, by default, cause *all* components inside of it to be rendered too!

Also, another key point:

In normal rendering, React does *not* care whether "props changed" - it will render child components unconditionally just because the parent rendered!

This means that calling `setState()` in your root `<App>` component, with no other changes altering the behavior, *will* cause React to re-render every single component in the component tree. After all, one of the original sales pitches for React was "act like we're redrawing the entire app on every update" (https://www.slideshare.net/floydophone/react-preso-v2).

Now, it's very likely that most of the components in the tree will return the exact same render output as last time, and therefore React won't need to make any changes to the DOM. *But*, React will still have to do the work of asking components to render themselves and diffing the render output. Both of those take time and effort.

Remember, **rendering is not a *bad* thing - it's how React knows whether it needs to actually make any changes to the DOM!**

Rules of React Rendering

One of the primary rules of React rendering is that **rendering must be "pure" and not have any side effects!** This can be tricky and confusing, because many side effects are not obvious, and don't result in anything breaking. For example, strictly speaking a `console.log()` statement is a side effect, but it won't actually break anything. Mutating a prop is definitely a side effect, and it *might* not break anything. Making an AJAX call in the middle of rendering is also definitely a side effect, and can definitely cause unexpected app behavior depending on the type of request.

Sebastian Markbage wrote an excellent document entitled **The Rules of React** (https://gist.github.com/sebmarkbage/75f0838967cd003cd7f9ab938eb1958f). In it, he defines the expected behaviors for different React lifecycle methods, including `render`, and what kinds of operations would be considered safely "pure" and which would be unsafe. It's worth reading that in its entirety, but I'll summarize the key points:

- Render logic *must not*:
    - Can't mutate existing variables and objects
    - Can't create random values like `Math.random()` or `Date.now()`
    - Can't make network requests
    - Can't queue state updates
- Render logic *may*:
    - Mutate objects that were newly created while rendering
    - Throw errors
    - "Lazy initialize" data that hasn't been created yet, such as a cached value

Component Metadata and Fibers

React stores an internal data structure that tracks all the current component instances that exist in the application. The core piece of this data structure is an object called a "fiber", which contains metadata fields that describe:

- What component type is supposed to be rendered at this point in the component tree
- The current props and state associated with this component
- Pointers to parent, sibling, and child components
- Other internal metadata that React uses to track the rendering process

You can see the definition of the `Fiber` type as of React 17 here (https://github.com/facebook/react/blob/v17.0.0/packages/react-reconciler/src/ReactFiber.new.js#L47-L174).

During a rendering pass, React will iterate over this tree of fiber objects, and construct an updated tree as it calculates the new rendering results.

Note that **these "fiber" objects store the *real* component props and state values**. When you use `props` and `state` in your components, React is actually giving you access to the values that were stored on the fiber objects. In fact, for class components specifically, React explicitly copies `componentInstance.props = newProps` over to the component right before rendering it (https://github.com/facebook/react/blob/v17.0.0/packages/react-reconciler/src/ReactFiberClassComponent.new.js#L1038-L1042). So, `this.props` does exist, but it only exists because React copied the reference over from its internal data structures. In that sense, components are sort of a facade over React's fiber objects.

Similarly, React hooks work because React stores all of the hooks for a component as a linked list attached to that component's fiber object (https://www.swyx.io/getting-closure-on-hooks/). When React renders a function component, it gets that linked list of hook description entries from the fiber, and every time you call another hook, it returns the appropriate values that were stored in the hook description object (like the `state` and `dispatch` values for `useReducer` (https://github.com/facebook/react/blob/v17.0.0/packages/react-reconciler/src/ReactFiberHooks.new.js#L795).

When a parent component renders a given child component for the first time, React creates a fiber object to track that "instance" of a component. For class components, it literally calls `const instance = new YourComponentType(props)` (https://github.com/facebook/react/blob/v17.0.0/packages/react-reconciler/src/ReactFiberClassComponent.new.js#L653) and saves the actual component instance onto the fiber object. For function components, React just calls `YourComponentType(props)` as a function (https://github.com/facebook/react/blob/v17.0.0/packages/react-reconciler/src/ReactFiberHooks.new.js#L405).

Component Types and Reconciliation

As described in the "Reconciliation" docs page (https://reactjs.org/docs/reconciliation.html#elements-of-different-types), React tries to be efficient during re-renders, by reusing as much of the existing component tree and DOM structure as possible. If you ask React to render the same type of component or HTML node in the same place in the tree, React will reuse that and just apply updates if appropriate, instead of re-creating it from scratch. That means that React will keep component instances alive as long as you keep asking React to render that component type in the same place. For class components, it actually does use the same actual instance of your component. A function component has no true "instance" the way a class does, but we can think of `<MyFunctionComponent />` as representing an "instance" in terms of "a component of this type is being shown here and kept alive".

So, how does React know when and how the output has actually changed?

React's rendering logic compares elements based on their `type` field first, using `===` reference comparisons. If an element in a given spot has changed to a different type, such as going from `<div>` to `<span>` or `<ComponentA>` to `<ComponentB>`, React will speed up the comparison process by assuming that entire tree has changed. As a result, **React will destroy that entire existing component tree section, including all DOM nodes**, and recreate it from scratch with new component instances.

This means that **you must never create new component types while rendering!** Whenever you create a new component type, it's a different reference, and that will cause React to repeatedly destroy and recreate the child component tree.

In other words, ***don't*** do this:

```
function ParentComponent() {
  // This creates a new `ChildComponent` reference every time!
  function ChildComponent() {}

  return <ChildComponent />
}
```

Instead, always define components separately:

```
  // This only creates one component type reference
function ChildComponent() {}

function ParentComponent() {

  return <ChildComponent />
}
```

Keys and Reconciliation

The other way that React identifies component "instances" is via the `key` pseudo-prop. `key` is an instruction to React, and will never be passed through to the actual component. React treats `key` as a unique identifier that it can use to differentiate specific instances of a component type.

The main place we use keys is rendering lists. Keys are especially important here if you are rendering data that may be changed in some way, such as reordering, adding, or deleting list entries. It's particularly important here that **keys *should* be some kind of unique IDs from your data if at all possible - only use array indices as keys as a last resort fallback!**

Here's an example of why this matters. Say I render a list of 10 `<TodoListItem>` components, using array indices as keys. React sees 10 items, with keys of `0..9`. Now, if we delete items 6 and 7, and add three new entries at the end, we end up rendering items with keys of `0..10`. So, it looks to React like I really just added one new entry at the end because we went from 10 list items to 11. React will happily reuse the existing DOM nodes and component instances. But, that means that we're probably now rendering `<TodoListItem key={6}>` with the todo item that *was* being passed to list item #8. So, the component instance is still alive, but now it's getting a different data object as a prop than it was previously. This may work, but it may also produce unexpected behavior. Also, React will now have to go apply updates to several of the list items to change the text and other DOM contents, because the existing list items are now having to show different data than before. Those updates really shouldn't be necessary here, since none of those list items changed.

If instead we were using `key={todo.id}` for each list item, React will correctly see that we deleted two items and added three new ones. It will destroy the two deleted component instances and their associated DOM, and create three new component instances and their DOM. This is better than having to unnecessarily update the components that didn't actually change.

Keys are useful for component instance identity beyond lists as well. **You can add a `key` to any React component at any time to indicate its identity, and changing that key will cause React to destroy the old component instance and DOM and create new ones**. A common use case for this is a list + details form combination, where the form shows the data for the currently selected list item. Rendering `<DetailForm key={selectedItem.id}>` will cause React to destroy and re-create the form when the selected item changes, thus avoiding any issues with stale state inside the form.

Render Batching and Timing

By default, each call to `setState()` causes React to start a new render pass, execute it synchronously, and return. However, React also applies a sort of optimization automatically, in the form of *render batching*. Render batching is when multiple calls to `setState()` result in a single render pass being queued and executed, usually on a slight delay.

The React docs mention that "state updates may be asynchronous" (https://reactjs.org/docs/state-and-lifecycle.html#state-updates-may-be-asynchronous). That's a reference to this render batching behavior. In particular, React automatically batches state updates that occur in React event handlers. Since React event handlers make up a very large portion of the code in a typical React app, this means that most of the state updates in a given app are actually batched.

React implements render batching for event handlers by wrapping them in an internal function known as `unstable_batchedUpdates`. React tracks all state updates that are queued while `unstable_batchedUpdates` is running, and then applies them in a single render pass afterwards. For event handlers, this works well because React already knows exactly what handlers need to be called for a given event.

Conceptually, you can picture what React's doing internally as something like this pseudocode:

```
// PSEUDOCODE Not real, but kinda-sorta gives the idea
function internalHandleEvent(e) {
  const userProvidedEventHandler = findEventHandler(e);

  let batchedUpdates = [];

  unstable_batchedUpdates( () => {
    // any updates queued inside of here will be pushed into batchedUpdates
    userProvidedEventHandler(e);
  });

  renderWithQueuedStateUpdates(batchedUpdates);
}
```

However, this means that **any state updates queued *outside* of the actual immediate call stack will *not* be batched together**.

Let's look at a specific example.

```
const [counter, setCounter] = useState(0);

const onClick = async () => {
  setCounter(0);
  setCounter(1);

  const data = await fetchSomeData();

  setCounter(2);
  setCounter(3);
}
```

This will execute **three** render passes. The first pass will batch together `setCounter(0)` and `setCounter(1)`, because both of them are occurring during the original event handler call stack, and so they're both occurring inside the `unstable_batchedUpdates()` call.

However, the call to `setCounter(2)` is happening after an `await`. This means the original synchronous call stack is done, and the second half of the function is running much later in a totally separate event loop call stack. Because of that, React will execute an entire render pass synchronously as the last step inside the `setCounter(2)` call, finish the pass, and return from `setCounter(2)`.

The same thing will then happen for `setCounter(3)`, because it's also running outside the original event handler, and thus outside the batching.

There's some additional edge cases inside of the commit-phase lifecycle methods: `componentDidMount`, `componentDidUpdate`, and `useLayoutEffect`. These largely exist to allow you to perform additional logic after a render, but before the browser has had a chance to paint. In particular, a common use case is:

- Render a component the first time with some partial but incomplete data
- In a commit-phase lifecycle, use refs to measure the real size of the actual DOM nodes in the page
- Set some state in the component based on those measurements
- Immediately re-render with the updated data

In this use case, we don't want the initial "partial" rendered UI to be visible to the user at all - we only want the "final" UI to show up. Browsers will recalculate the DOM structure as it's being modified, but they won't actually paint anything to the screen while a JS script is still executing and blocking the event loop. So, you can perform multiple DOM mutations, like `div.innerHTML = "a"; div.innerHTML = b";`, and the `"a"` will never appear.

Because of this, React will always run renders in commit-phase lifecycles synchronously. That way, if you do try to perform an update like that "partial->final" switch, only the "final" content will ever be visible on screen.

Finally, as far as I know, state updates in `useEffect` callbacks are queued up, and flushed at the end of the "Passive Effects" phase once all the `useEffect` callbacks have completed.

It's worth noting that the `unstable_batchedUpdates` API *is* exported publicly, but:

- Per the name, it is labeled as "unstable" and is not an officially supported part of the React API
- On the other hand, the React team has said that "it's the most stable of the 'unstable' APIs, and half the code at Facebook relies on that function"
- Unlike the rest of the core React APIs, which are exported by the `react` package, `unstable_batchedUpdates` is a reconciler-specific API and is *not* part of the `react` package. Instead, it's actually exported by `react-dom` and `react-native`. That means that other reconcilers, like `react-three-fiber` or `ink`, will likely not export an `unstable_batchedUpdates` function.

For React-Redux v7, we started using `unstable_batchedUpdates` internally (/2018/11/react-redux-history-implementation/#use-of-react-s-batched-updates-api), which required some tricky build setup to work with both ReactDOM and React Native (effectively conditional imports depending on which package is available.)

In React's upcoming Concurrent Mode, React will *always* batch updates, all the time, everywhere.

Render Behavior Edge Cases

React will **double-render components inside of a `<StrictMode>` tag in development**. That means the number of times your rendering logic runs is *not* the same as the number of committed render passes, and you *cannot* rely on `console.log()` statements while rendering to count the number of renders that have occurred. Instead, either use the React DevTools Profiler to capture a trace and count the number of committed renders overall, or add logging inside of a `useEffect` hook or `componentDidMount/Update` lifecycle. That way the logs will only get printed when React has actually completed a render pass and committed it.

In normal situations, you should *never* queue a state update while in the actual rendering logic. In other words, it's fine to create a click callback that will call `setSomeState()` when the click happens, but you should *not* call `setSomeState()` as part of the actual rendering behavior.

However, there is one exception to this. **Function components *may* call `setSomeState()` directly while rendering, as long as it's done conditionally** and isn't going to execute *every* time this component renders. This acts as the function component equivalent of `getDerivedStateFromProps` in class components (https://reactjs.org/docs/hooks-faq.html#how-do-i-implement-getderivedstatefromprops). If a function component queues a state update while rendering, React will immediately apply the state update and synchronously re-render that one component before moving onwards. If the component infinitely keeps queueing state updates and forcing React to re-render it, React will break the loop after a set number of retries and throw an error (currently 50 attempts). This technique can be used to immediately force an update to a state value based on a prop change, without requiring a re-render + a call to `setSomeState()` inside of a `useEffect`.

Improving Rendering Performance

Although renders are the normal expected part of how React works, it's also true that that render work can be "wasted" effort at times. If a component's render output didn't change, and that part of the DOM didn't need to be updated, then the work of rendering that component was really kind of a waste of time.

React component render output *should* always be entirely based on current props and current component state. Therefore, if we *know* ahead of time that a component's props and state haven't changed, we *should* also know that the render output would be the same, that no changes are necessary for this component, and that we can safely skip the work of rendering it.

When trying to improve software performance in general, there are two basic approaches: 1) do the same work faster, and 2) do less work. Optimizing React rendering is primarily about doing less work by skipping rendering components when appropriate.

Component Render Optimization Techniques

React offers three primary APIs that allow us to potentially skip rendering a component:

- `React.Component.shouldComponentUpdate` (https://reactjs.org/docs/react-component.html#shouldcomponentupdate): an optional class component lifecycle method that will be called early in the render process. If it returns `false`, React will skip rendering the component. It may contain any logic you want to use to calculate that boolean result, but the most common approach is to check if the component's props and state have changed since last time, and return `false` if they're unchanged.

- `React.PureComponent` (https://reactjs.org/docs/react-api.html#reactpurecomponent): since that comparison of props and state is the most common way to implement `shouldComponentUpdate`, the `PureComponent` base class implements that behavior by default, and may be used instead of `Component + shouldComponentUpdate`.
- `React.memo()` (https://reactjs.org/docs/react-api.html#reactmemo): a built-in "higher order component" (https://reactjs.org/docs/higher-order-components.html) type. It accepts your own component type as an argument, and returns a new wrapper component. The wrapper component's default behavior is to check to see if any of the props have changed, and if not, prevent a re-render. Both function components and class components can be wrapped using `React.memo()`. (A custom comparison callback may be passed in, but it really can only compare the old and new props anyway, so the main use case for a custom compare callback would be only comparing specific props fields instead of all of them.)

All of these approaches use a comparison technique called **"shallow equality"**. This means checking every individual field in two different objects, and seeing if any of the *contents* of the objects are a different value. In other words, `obj1.a === obj2.a && obj1.b === obj2.b && ........`. This is typically a fast process, because `===` comparisons are very simple for the JS engine to do. So, these three approaches do the equivalent of `const shouldRender = !shallowEqual(newProps, prevProps)`.

There's also a lesser-known technique as well: **if a React component returns the exact same element reference in its render output as it did the last time, React will skip re-rendering that particular child.** There's at least a couple ways to implement this technique:

- If you include `props.children` in your output, that element is the same if this component does a state update
- If you wrap some elements with `useMemo()`, those will stay the same until the dependencies change

Examples:

```
// The `props.children` content won't re-render if we update state
function SomeProvider({children}) {
  const [counter, setCounter] = useState(0);

  return (
    <div>
      <button onClick={() => setCounter(counter + 1)}>Count: {counter}</butto
      <OtherChildComponent />
      {children}
    </div>
  )
}

function OptimizedParent() {
  const [counter1, setCounter1] = useState(0);
  const [counter2, setCounter2] = useState(0);

  const memoizedElement = useMemo(() => {
    // This element stays the same reference if counter 2 is updated,
    // so it won't re-render unless counter 1 changes
    return <ExpensiveChildComponent />
  }, [counter1]) ;

  return (
    <div>
      <button onClick={() => setCounter1(counter1 + 1)}>Counter 1: {counter1}
      <button onClick={() => setCounter1(counter2 + 1)}>Counter 2: {counter2}
      {memoizedElement}
    </div>
  )
}
```

For all of these techniques, skipping rendering a component means React will also skip rendering that entire subtree, because it's effectively putting a stop sign up to halt the default "render children recursively" behavior.

How New Props References Affect Render Optimizations

We've already seen that **by default, React re-renders all nested components even if their props haven't changed**. That also means that passing new references as props to a child component doesn't matter, because it will render whether or not you pass the same props. So, something like this is totally fine:

```
function ParentComponent() {
    const onClick = () => {
      console.log("Button clicked")
    }

    const data = {a: 1, b: 2}

    return <NormalChildComponent onClick={onClick} data={data} />
}
```

Every time `ParentComponent` renders, it will create a new `onClick` function reference and a new `data` object reference, then pass them as props to `NormalChildComponent`. (Note that it doesn't matter whether we're defining `onClick` using the `function` keyword or as an arrow function - it's a new function reference either way.)

That also means there's no point in trying to optimize renders for "host components", like a `<div>` or a `<button>`, by wrapping them up in a `React.memo()`. There's no child component underneath those basic components, so the rendering process would stop there anyway.

However, ***if* the child component is trying to optimize renders by checking to see whether props have changed, *then* passing new references as props will cause the child to render**. If the new prop references are actually new data, this is good. However, what if the parent component is just passing down a callback function?

```
const MemoizedChildComponent = React.memo(ChildComponent)

function ParentComponent() {
    const onClick = () => {
      console.log("Button clicked")
    }

    const data = {a: 1, b: 2}

    return <MemoizedChildComponent onClick={onClick} data={data} />
}
```

Now, every time `ParentComponent` renders, these new references are going to cause `MemoizedChildComponent` to see that its props values have changed to new references, and it will go ahead and re-render... even though the `onClick` function and the `data` object *should* be basically the same thing every time!

This means that:

- `MemoizedChildComponent` will always re-render even though we wanted to skip rendering most of the time
- The work that it's doing to compare its old and new props is wasted effort

Similarly, note that rendering `<MemoizedChild><OtherComponent /></MemoizedChild>` will *also* force the child to always render, because `props.children` is always a new reference.

Optimizing Props References

Class components don't have to worry about accidentally creating new callback function references as much, because they can have instance methods that are always the same reference. However, they may need to generate unique callbacks for separate child list items, or capture a value in an anonymous function and pass that to a child. Those will result in new references, and so will creating new objects as child props while rendering. React doesn't have anything built-in to help optimize those cases.

For function components, React does provide two hooks to help you reuse the same references: `useMemo` for any kind of general data like creating objects or doing complex calculations, and `useCallback` specifically for creating callback functions.

Memoize Everything?

As mentioned above, you don't have throw `useMemo` and `useCallback` at every single function or object you pass down as a prop - only if it's going to make a difference in behavior for the child. (That said, the dependency array comparisons for `useEffect` *do* add another use case where the child might want to receive consistent props references, which does make things more complicated.)

The other question that comes up all the time is "Why doesn't React wrap *everything* in `React.memo()` by default?".

Dan Abramov has repeatedly pointed out that memoization *does* still incur the cost of comparing props (https://twitter.com/dan_abramov/status/1095661142477811717), and that there are many cases where the memoization check can never prevent re-renders because the component always receives new props. As an example, see this Twitter thread from Dan (https://twitter.com/dan_abramov/status/1083897065263034368):

> *Why doesn’t React put memo() around every component by default? Isn’t it faster? Should we make a benchmark to check?*
>
> *Ask yourself:*
>
> *Why don’t you put Lodash memoize() around every function? Wouldn’t that make all functions faster? Do we need a benchmark for this? Why not?*

Also, while I don't have a specific link on it, it's possible that trying to apply this to all components by default might result in bugs due to cases where people are mutating data rather than updating it immutably.

I've had some public discussion with Dan about this on Twitter. I personally think it's likely that using `React.memo()` on a widespread basis would likely be a net gain in overall app rendering perf. As I said in an extended Twitter thread last year (https://twitter.com/acemarke/status/1141755698948165632):

> *The React community as a whole seems to be over obsessed with "perf", yet much of the discussion revolves around outdated "tribal wisdom" passed down via Medium posts and Twitter comments rather than based on concrete usage.*
>
> *There's definitely collective misunderstanding about the idea of a "render" and the perf impact. Yes, React is totally based around rendering - gotta render to do anything at all. No, most renders aren't overly expensive.*
>
> *"Wasted" rerenders certainly aren't the end of the world. Neither is rerendering the whole app from the root. That said, it's also true that a "wasted" rerender with no DOM update is CPU cycles that didn't need to be burned. Is that a problem for most apps? Probably not. Is it something that can be improved? Probably.*
>
> *Are there apps where default "rerender it all" approaches aren't sufficient? Of course, that's why sCU, PureComponent, and memo() exist.*
>
> *Should users wrap everything in memo() by default? Probably not, if only because you should think about your app's perf needs. Will it actually hurt if you do? No, and realistically I expect it does have a net benefit (despite Dan's points about wasted comparisons)*
>
> *Are benchmarks flawed, and results highly variable based on scenarios and apps? Of course. That said, it would be REALLY REALLY HELPFUL if folks could start pointing at hard numbers for these discussions instead of playing the telephone game of "I saw a comment once..."*
>
> *I'd love to see a bunch of benchmark suites from the React team and the larger community to measure a bunch of scenarios so we could stop arguing about most of this stuff once and for all. Function creation, render cost, optimization... CONCRETE EVIDENCE, PLEASE!*

But, no one's put together any good benchmarks that would demonstrate whether or not this is true (https://twitter.com/acemarke/status/1229083161646305280):

> *Dan's standard answer is that app structure and update patterns vary drastically, so it's hard to make a representative benchmark.*
>
> *I still think some actual numbers would be useful to aid the discussion*

There's also an extended issue discussion on "When should you NOT use React.memo? (https://github.com/facebook/react/issues/14463) in the React issues.

(And yes, this blog post is basically a long-delayed and much-expanded version of that tweet thread, although I'd actually forgotten I'd tweeted all that until I ran across it just now while researching the post.)

Immutability and Rerendering

State updates in React should always be done immutably. There are two main reasons why:

- depending on what you mutate and where, it can result in components not rendering when you expected they would render
- it causes confusion about when and why data actually got updated

Let's look at a couple specific examples.

As we've seen, `React.memo / PureComponent / shouldComponentUpdate` all rely on shallow equality checks of the current props vs the previous props. So, the expectation is that we can know if a prop is a new value, by doing `props.someValue !== prevProps.someValue`.

If you mutate, then `someValue` is the same reference, and those components will assume nothing has changed.

Note that this is *specifically* when we're trying to optimize performance by avoiding unnecessary re-renders. A render is "unnecessary" or "wasted" if the props haven't changed. If you mutate, the component may wrongly think nothing has changed, and then you wonder why the component didn't re-render.

The other issue is the `useState` and `useReducer` hooks. Every time I call `setCounter()` or `dispatch()`, React will queue up a re-render. However, React requires that any hook state updates must pass in / return a new reference as the new state value, whether it be a new object/array reference, or a new primitive (string/number/etc).

React applies all state updates during the render phase. When React tries to apply a state update from a hook, it checks to see if the new value is the same reference. React will *always* finish rendering the component that queued the update. However, *if* the value is the same reference as before, *and* there are no other reasons to continue rendering (such as the parent having rendered), React will then throw away the render results for the component and bail out of the render pass completely. So, if I mutate an array like this:

```
const [todos, setTodos] = useState(someTodosArray);

const onClick = () => {
  todos[3].completed = true;
  setTodos(todos);
}
```

then the component will fail to re-render.

Technically, only the outermost reference has to be immutably updated. If we change that example to:

```
const onClick = () => {
  const newTodos = todos.slice();
  newTodos[3].completed = true;
  setTodos(newTodos);
}
```

then we have created a new array reference and passed it in, and the component *will* re-render.

Note that there is a distinct difference in behavior between the class component `this.setState()` and the function component `useState` and `useReducer` hooks with regards to mutations and re-rendering. `this.setState()` doesn't care if you mutate at all - it *always* completes the re-render. So, this will re-render:

```
const {todos} = this.state;
todos[3].completed = true;
this.setState({todos});
```

And in fact, so will passing in an empty object like `this.setState({})`.

Beyond all the actual rendering behavior, mutation introduces confusion to the standard React one-way data flow. Mutation can lead other code to see different values when the expectation was they haven't changed at all. This makes it harder to know when and why a given piece of state was actually supposed to update, or where a change came from.

Bottom line: **React, and the rest of the React ecosystem, assume immutable updates. Any time you mutate, you run the risk of bugs. Don't do it.**

Measuring React Component Rendering Performance

Use the React DevTools Profiler (https://reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html) to see what components are rendering in each commit. Find components that are rendering unexpectedly, use the DevTools to figure out *why* they rendered, and fix things (perhaps by wrapping them in `React.memo()`, or having the parent component memoize the props it's passing down.)

Also, remember that React runs way slower in dev builds. You can profile your app in development mode to see *which* components are rendering and why, and do some comparisons of *relative* time needed to render components in comparison to each other ("Component B took 3 times as long to render in this commit than component A" did). But, **never measure absolute render times using a React development build - only measure absolute times using production builds!** (or else Dan Abramov will have to come yell at you for using numbers that aren't accurate). Note that you'll need to use a special "profiling" build of React (https://kentcdodds.com/blog/profile-a-react-app-for-performance) if you want to actually use the profiler to capture timing data from a prod-like build.

Context and Rendering Behavior

React's Context API is a mechanism for making a single user-provided value available to a subtree of components, Any component inside of a given `<MyContext.Provider>` can read the value from that context instance, without having to explicitly pass that value as a prop through every intervening component.

Context is not a "state management" tool. You have to manage the values that are passed into context yourself. This is typically done by keeping data in React component state, and constructing context values based on that data.

Context Basics

A context provider receives a single `value` prop, like `<MyContext.Provider value={42}>`. Child components may consume the context by rendering the context consumer component and providing a render prop, like:

`<MyContext.Consumer>{ (value) => <div>{value}</div>}</MyContext.Consumer>`

or by calling the `useContext` hook in a function component:

`const value = useContext(MyContext)`

Updating Context Values

React checks to see if a context provider has been given a new value when the surrounding component renders the provider. If the provider's value is a new reference, then React knows the value has changed, and that the components consuming that context need to be updated.

Note that **passing a new object to a context provider *will* cause it to update**:

```
function GrandchildComponent() {
    const value = useContext(MyContext);
    return <div>{value.a}</div>
}

function ChildComponent() {
    return <GrandchildComponent />
}

function ParentComponent() {
    const [a, setA] = useState(0);
    const [b, setB] = useState("text");

    const contextValue = {a, b};

    return (
      <MyContext.Provider value={contextValue}>
        <ChildComponent />
      </MyContext.Provider>
    )
}
```

In this example, every time `ParentComponent` renders, React will take note that `MyContext.Provider` has been given a new value, and look for components that consume `MyContext` as it continues looping downwards. **When a context provider has a new value, *every* nested component that consumes that context will be forced to re-render**.

Note that from React's perspective, each context provider only has a single value - doesn't matter whether that's an object, array, or a primitive, it's just one context value. Currently, **there is no way for a component that consumes a context to skip updates caused by new context values, even if it only cares about *part* of a new value.**

State Updates, Context, and Re-Renders

It's time to put some of these pieces together. We know that:

- Calling `setState()` queues a render of that component

- React recursively renders nested components by default
- Context providers are given a value by the component that renders them
- That value normally comes from that parent component's state

This means that **by default, any state update to a parent component that renders a context provider will cause all of its descendants to re-render anyway, regardless of whether they read the context value or not!**.

If we look back at the `Parent/Child/Grandchild` example just above, we can see that **the `GrandchildComponent` *will* re-render, but not because of a context update - it will re-render because `ChildComponent` rendered!**. In this example, there's nothing trying to optimize away "unnecessary" renders, so React renders `ChildComponent` and `GrandchildComponent` by default any time `ParentComponent` renders. If the parent puts a new context value into `MyContext.Provider`, the `GrandchildComponent` will see the new value when it renders and use it, but the context update didn't *cause* `GrandchildComponent` to render - it was going to happen anyway.

Context Updates and Render Optimizations

Let's modify that example so that it does actually try to optimize things, but we'll add one other twist by putting a `GreatGrandchildComponent` at the bottom:

```
function GreatGrandchildComponent() {
  return <div>Hi</div>
}

function GrandchildComponent() {
    const value = useContext(MyContext);
    return (
      <div>
        {value.a}
        <GreatGrandchildComponent />
      </div>
}

function ChildComponent() {
    return <GrandchildComponent />
}

const MemoizedChildComponent = React.memo(ChildComponent);

function ParentComponent() {
    const [a, setA] = useState(0);
    const [b, setB] = useState("text");

    const contextValue = {a, b};

    return (
      <MyContext.Provider value={contextValue}>
        <MemoizedChildComponent />
      </MyContext.Provider>
    )
}
```

Now, if we call `setA(42)`:

- `ParentComponent` will render
- A new `contextValue` reference is created
- React sees that `MyContext.Provider` has a new context value, and thus any consumers of `MyContext` need to be updated
- React will try to render `MemoizedChildComponent`, but see that it's wrapped in `React.memo()`. There are no props being passed at all, so the props have not actually changed. React will skip rendering `ChildComponent` entirely.
- However, there was an update to `MyContext.Provider`, so there *may* be components further down that need to know about that.
- React continues downwards and reaches `GrandchildComponent`. It sees that `MyContext` is read by `GrandchildComponent`, and thus it *should* re-render because

there's a new context value. React goes ahead and re-renders `GrandchildComponent`, specifically because of the context change.
- Because `GrandchildComponent` *did* render, React then keeps on going and also renders whatever's inside of it. So, React will also re-render `GreatGrandchildComponent`.

In other words, as Sophie Alpert said (https://twitter.com/sophiebits/status/1228942768543686656):

> ***That React Component Right Under Your Context Provider Should Probably Use `React.memo`***

That way, state updates in the parent component will *not* force every component to re-render, just the sections where the context is read. (You could also get basically the same result by having `ParentComponent` render `<MyContext.Provider>{props.children}</MyContext.Provider>`, which leverages the "same element reference" technique to avoid child components re-rendering, and then rendering `<ParentComponent><ChildComponent /></ParentComponent>` from one level up.)

Note, however, that **once `GrandchildComponent` rendered based on the next context value, React went right back to its default behavior of recursively re-rendering everything**. So, `GreatGrandchildComponent` was rendered, and anything else under there would have rendered too.

React-Redux and Rendering Behavior

The various forms of "CONTEXT VS REDUX?!?!??!" seem to be the single most-asked question I see in the React community right now. (That question is a false dichotomy to begin with, as **Redux and Context are different tools that do different things** (/2018/03/redux-not-dead-yet/).)

That said, one of the recurring things that people point out when this comes up is that "React-Redux only re-renders the components that actually need to render, so that makes it better than context".

That's *somewhat* true, but the answer is a lot more nuanced than that.

React-Redux Subscriptions

I've seen a lot of folks repeat the phrase "React-Redux uses context inside." Also technically true, but React-Redux uses context to pass the *Redux store instance*, not the *current state value* (/2020/01/blogged-answers-react-redux-and-context-behavior/). That means that we always pass the same context value into our `<ReactReduxContext.Provider>` over time.

Remember that a Redux store runs all its subscriber notification callbacks whenever an action is dispatched. UI layers that need to use Redux always subscribe to the Redux store, read the latest state in their subscriber callbacks, diff the values, and force a re-render if the relevant data has changed (/2018/11/react-redux-history-implementation/). The subscription callback process happens *outside* of React entirely, and React only gets involved if React-Redux *knows* that the data needed by a specific React component has changed (based on the return values of `mapState` or `useSelector`).

This results in a very different set of performance characteristics than context. Yes, it's likely that fewer components will be rendering all the time, *but* React-Redux will always have to run the `mapState/useSelector` functions for the entire component tree every time the store state is updated. **Most of the time, the cost of running those selectors is less than the cost of React doing another render pass, so it's usually a net win**, but it *is* work that has to be done. **However, if those selectors are doing expensive transformations or accidentally returning new values when they shouldn't, that can slow things down**.

Differences between `connect` and `useSelector`

`connect` is a higher-order component. You pass in your own component, and `connect` returns a wrapper component that does all the work of subscribing to the store, running your `mapState` and `mapDispatch`, and passing the combined props down to your own component.

The `connect` wrapper components have always acted equivalent to `PureComponent/React.memo()`, but with a slightly different emphasis: `connect` will only make your own component render *if* the combined props it's passing down to your component have changed. Typically, the final combined props are a combination of `{...ownProps, ...stateProps, ...dispatchProps}`, so any new prop references from the parent will indeed cause your component to render, same as `PureComponent` or `React.memo()`. Besides parent props, any new references returned from `mapState` will also cause your component to render. (https://react-redux.js.org/using-react-redux/connect-mapstate#mapstatetoprops-and-performance). (Since you *could* customize how `ownProps`/`stateProps`/`dispatchProps` are merged, it's also possible to alter that behavior.)

`useSelector`, on the other hand, is a hook that is called inside of your own function components. Because of that, **`useSelector` has no way of stopping your component from rendering when the parent component renders!**.

This is a key performance difference between `connect` and `useSelector` (https://react-redux.js.org/api/hooks#performance). With `connect`, every connected component acts like `PureComponent`, and thus acts as a firewall to prevent React's default render behavior from cascading down the entire component tree. Since a typical React-Redux app has *many* connected components, this means that most re-render cascades are limited to a fairly small section of the component tree. React-Redux will force a connected component to render based on data changes, the next 2-3 components below it might render as well, then React runs into another connected component that didn't need to update and that stops the rendering cascade.

In addition, having more connected components means that each component is probably reading smaller pieces of data from the store, and thus is less likely to have to re-render after any given action.

If you're exclusively using function components and `useSelector`, then **it's likely that larger parts of your component tree will re-render based on Redux store updates than they would with `connect`**, since there aren't other connected components to stop those render cascades from continuing down the tree.

If that becomes a performance concern, then the answer is to wrap components in `React.memo()` yourself as needed, to prevent unnecessary re-renders caused by parent components.

Summary

- React always recursively renders components by default, so when a parent renders, its children will render
- Rendering by itself is fine - it's how React knows what DOM changes are needed
- But, rendering takes time, and "wasted renders" where the UI output didn't change can add up
- It's okay to pass down new references like callback functions and objects most of the time
- APIs like `React.memo()` can skip unnecessary renders if props haven't changed

- But if you always pass new references down as props, `React.memo()` can never skip a render, so you may need to memoize those values
- Context makes values accessible to any deeply nested component that is interested
- Context providers compare their value by reference to know if it's changed
- A new context values does force all nested consumers to re-render
- But, many times the child would have re-rendered anyway due to the normal parent->child render cascade process
- So you probably want to wrap the child of a context provider in `React.memo()`, or use `{props.children}`, so that the whole tree doesn't render all the time when you update the context value
- When a child component is rendered based on a new context value, React keeps cascading renders down from there too
- React-Redux uses subscriptions to the Redux store to check for updates, instead of passing store state values by context
- Those subscriptions run on every Redux store update, so they need to be as fast as possible
- React-Redux does a lot of work to ensure that only components whose data changed are forced to re-render
- `connect` acts like `React.memo()`, so having lots of connected components can minimize the total number of components that render at a time
- `useSelector` is a hook, so it can't stop renders caused by parent components. An app that only has `useSelector` everywhere should probably add `React.memo()` to some components to help avoid renders from cascading all the time.

Final Thoughts

Clearly, the whole situation is a lot more complex than just "context makes everything render, Redux doesn't, use Redux". Don't get me wrong, I *want* people to use Redux, but I also want people to clearly understand the behaviors and tradeoffs involved in different tools so they can make informed decisions about what's best for their own use cases.

Since everyone always seems to ask "When should I use Context, and when should I use (React-)Redux?", let me go ahead and recap some standard rules of thumb:

- Use context if:
  - You just need to pass some simple values that don't change often
  - You have some state or functions that need to be accessed through part of the app, and you don't want to pass them as props all the way down
  - You want to stick with what's built in to React and not add additional libraries

- Use (React-)Redux if:
  - You have large amounts of application state that are needed in many places in the app
  - The app state is updated frequently over time
  - The logic to update that state may be complex
  - The app has a medium or large-sized codebase, and might be worked on by many people

Please note that **these are not hard, exclusive rules - they are just some suggested guidelines for when these tools *might* make sense!** As always, *please* take some time to decide for yourself what the best tool is for whatever situation you're dealing with.

Overall, hopefully this explanation helps folks understand the bigger picture of what's actually going on with React's rendering behavior in various situations.

Further Information

- **General**
  - Dave Ceddia: A Visual Guide to References in JavaScript (https://daveceddia.com/javascript-references/)
- **React Render Behavior**
  - React docs: Reconciliation (https://reactjs.org/docs/reconciliation.html)
  - React lifecycle methods diagram (https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)
  - React hooks lifecycle diagram (https://github.com/donavon/hook-flow)
  - React issues: bailing out of context and hooks (https://github.com/facebook/react/issues/14110)
  - React issues: why `setState` is async (https://github.com/facebook/react/issues/11527#issuecomment-360199710)
  - Seb Markbage: "Context is good for low-frequency updates, not Flux-like state propagation" (https://github.com/facebook/react/issues/14110#issuecomment-448074060)
  - Ryan Florence: React, Inline Functions, and Performance (https://cdb.reacttraining.com/react-inline-functions-and-performance-bdff784f5578)
  - James K Nelson: React context and performance (https://frontarm.com/james-k-nelson/react-context-performance/)
- **Optimizing Render Performance**
  - React docs: Optimizing Performance (https://reactjs.org/docs/optimizing-

performance.html)
    - Kent C Dodds: Fix the slow render before you fix the re-render (https://kentcdodds.com/blog/fix-the-slow-render-before-you-fix-the-re-render)
    - Kent C Dodds: When to `useMemo` and `useCallback` (https://kentcdodds.com/blog/usememo-and-usecallback)
    - Kent C Dodds: One simple trick to optimize React re-renders (https://kentcdodds.com/blog/optimize-react-re-renders)
    - React issues: When should you NOT use React.memo? (https://github.com/facebook/react/issues/14463)
- **Profiling React Components**
    - React docs: Introducing the React DevTools Profiler (https://reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html)
    - React DevTools profiler interactive tutorial (https://react-devtools-tutorial.now.sh/)
    - Kent C Dodds: Profile a React App for Performance (https://kentcdodds.com/blog/profile-a-react-app-for-performance)
    - Shawn Wang: Using the React DevTools Profiler to Diagnose React App Performance Issues (https://www.netlify.com/blog/2018/08/29/using-the-react-devtools-profiler-to-diagnose-react-app-performance-issues/)
    - Use the React Profiler for Performance (https://scotch.io/tutorials/use-the-react-profiler-for-performance)
- **React-Redux Performance**
    - Practical Redux, Part 6: Connected Lists and Performance (/2017/01/practical-redux-part-6-connected-lists-forms-and-performance/)
    - Idiomatic Redux: The History and Implementation of React-Redux (/2018/11/react-redux-history-implementation/)
    - React-Redux docs: `mapState` Usage Guide - Performance (https://react-redux.js.org/using-react-redux/connect-mapstate#mapstatetoprops-and-performance)
    - High-Performance Redux (http://somebody32.github.io/high-performance-redux/)
    - React-Redux links: React/Redux Performance (https://github.com/markerikson/react-redux-links/blob/master/react-performance.md)

()This is a post in the **Blogged Answers** (/series/blogged-answers) series. Other posts in this series:

- Jun 22, 2021 - **Blogged Answers: The Evolution of Redux Testing Approaches** (https://blog.isquaredsoftware.com/2021/06/the-evolution-of-redux-testing-approaches/)
- Jan 18, 2021 - **Blogged Answers: Why React Context is Not a "State Management" Tool (and Why It Doesn't Replace Redux)** (https://blog.isquaredsoftware.com/2021/01/context-redux-differences/)
- Jun 21, 2020 - **Blogged Answers: React Components, Reusability, and Abstraction** (https://blog.isquaredsoftware.com/2020/06/blogged-answers-react-components-reusability-and-abstraction/)
- May 17, 2020 - **Blogged Answers: A (Mostly) Complete Guide to React Rendering Behavior**
- May 12, 2020 - **Blogged Answers: Why I Write** (https://blog.isquaredsoftware.com/2020/05/blogged-answers-why-i-write/)
- Feb 22, 2020 - **Blogged Answers: Why Redux Toolkit Uses Thunks for Async Logic** (https://blog.isquaredsoftware.com/2020/02/blogged-answers-why-redux-toolkit-uses-thunks-for-async-logic/)
- Feb 22, 2020 - **Blogged Answers: Coder vs Tech Lead - Balancing Roles** (https://blog.isquaredsoftware.com/2020/02/blogged-answers-coder-vs-tech-lead---balancing-roles/)
- Jan 19, 2020 - **Blogged Answers: React, Redux, and Context Behavior** (https://blog.isquaredsoftware.com/2020/01/blogged-answers-react-redux-and-context-behavior/)
- Jan 01, 2020 - **Blogged Answers: Years in Review, 2018-2019** (https://blog.isquaredsoftware.com/2020/01/blogged-answers-years-in-review-2018-2019/)
- Jan 01, 2020 - **Blogged Answers: Reasons to Use Thunks** (https://blog.isquaredsoftware.com/2020/01/blogged-answers-reasons-to-use-thunks/)
- Jan 01, 2020 - **Blogged Answers: A Comparison of Redux Batching Techniques** (https://blog.isquaredsoftware.com/2020/01/blogged-answers-redux-batching-techniques/)
- Nov 26, 2019 - **Blogged Answers: Learning and Using TypeScript as an App Dev and a Library Maintainer** (https://blog.isquaredsoftware.com/2019/11/blogged-answers-learning-and-using-typescript/)
- Jul 10, 2019 - **Blogged Answers: Thoughts on React Hooks, Redux, and Separation of Concerns** (https://blog.isquaredsoftware.com/2019/07/blogged-answers-thoughts-on-hooks/)

- Jan 19, 2019 - **Blogged Answers: Debugging Tips** (https://blog.isquaredsoftware.com/2019/01/blogged-answers-debugging-tips/)
- Mar 29, 2018 - **Blogged Answers: Redux - Not Dead Yet!** (https://blog.isquaredsoftware.com/2018/03/redux-not-dead-yet/)
- Dec 18, 2017 - **Blogged Answers: Resources for Learning Redux** (https://blog.isquaredsoftware.com/2017/12/blogged-answers-learn-redux/)
- Dec 18, 2017 - **Blogged Answers: Resources for Learning React** (https://blog.isquaredsoftware.com/2017/12/blogged-answers-learn-react/)
- Aug 02, 2017 - **Blogged Answers: Webpack HMR vs React-Hot-Loader** (https://blog.isquaredsoftware.com/2017/08/blogged-answers-webpack-hmr-vs-rhl/)
- Sep 14, 2016 - **How I Got Here: My Journey Into the World of Redux and Open Source** (https://blog.isquaredsoftware.com/2016/09/how-i-got-here-my-journey-into-the-world-of-redux-and-open-source/)

30 Comments - *powered by utteranc.es*

markerikson commented on 5 Oct 2020 Owner

Original author: **Anujit Nene @anujitnene**
Original date: 2020-05-17T10:25:09Z

Totally loved reading this consolidated article, nailed the concepts to clarity! One doubt I have - In the example of context provider (the one before the optimized example using MemoizedChildComponent), if there's a setState call in the ParentComponent, will there be two renders of the subtree scheduled - one due to the setState itself and another one due to the change in the context value provided to the provider due to this setState? Is this understanding correct?

markerikson commented on 5 Oct 2020 Owner

Original author: **BS-Harou @bsharou**
Original date: 2020-05-17T13:14:13Z

This is really nice summary, thank you for putting it together!

In regards to using memo/PureComponents everywhere:

> It's possible that trying to apply this to all components by default might result in bugs due to cases where people are mutating data rather than updating it immutably.

In my experience, the decision to use memo/PureComponents everywhere goes together with enforcing immutability of all props passing through React and so if someone is mutating such a value it is a mistake on behalf of the developer/code reviewer/bad types rather than the arch. decision itself.

As a reader I also feel a bit confused on whether you think memoizing everything is a good idea or not where on one hand you argue you in most cased don't have to and you should think about each case but but on the other you say that using it everywhere is probably overall net positive.

I would personally argue that using it every time everywhere is probably the way go. Of course by every time I don't mean when a developer is prototyping or playing with code locally or putting together some examples, but I am talking about full blown production SPAs.

markerikson commented on 5 Oct 2020 Owner

Original author: **fabb**
Original date: 2020-05-22T20:53:10Z

Thanks a lot for the insights, I learned a few more details!

I have one suggestion and one detail question.

First the suggestion. You mentioned this:
> Currently, there is no way for a component that consumes a context to skip updates caused by new context values, even if it only cares about part of a new value.

This is not entirely true, there are a few implementations of useContextSelector which only cause rerenders when the selected part of the context changed (using `observedBits`), and there is even an RFC open to integrate it into core React: https://github.com/reactjs/...

Second the question. Suppose we pass an onClick function down like in your example, but without a memoized child component, and it's assigned to the onClick of a <button> component:

```
function ParentComponent() {
const onClick = () => {
console.log("Button clicked")
}
```

markerikson commented on 5 Oct 2020 Owner

Original author: **AndyYou** @andyyou
Original date: 2020-05-29T09:26:57Z

About `Memoize Everything?` section. I don't really get the meaning "only if it's going to make a difference in behavior for the child".
Could you give me some examples for that. The next explanation about useEffect as well.

Is that means new reference will make child component get different result?

markerikson commented on 5 Oct 2020 Owner

Original author: **Gadi Tzkhori @gadi_tzkhori**
Original date: 2020-05-30T09:02:12Z

isn't contextValue as an object, recreated every rerender?, thus requires useMemo wrapping?

markerikson commented on 5 Oct 2020 Owner

Original author: **Ganesh Pendyala** @Ganeshlakshmi
Original date: 2020-05-30T11:03:28Z

Many Thanks for putting this together Mark. It really hardened my understanding of the React rendering behaviour and the pitfalls while using Context.

markerikson commented on 5 Oct 2020 Owner

Original date: 2020-05-30T15:41:19Z

That's exactly the point I'm trying to make throughout the article. Yes, in general, you probably want to memoize your context values, but there's other factors that play into whether or not the rest of the components render. If you don't have anything else blocking renders between the parent that renders the context provider, and the child that consumes the context, the child will always render anyway due to the default recursive rendering behavior.

markerikson commented on 5 Oct 2020 Owner

Original date: 2020-05-30T15:43:01Z

Sort of. Multiple components may be flagged as "dirty" and needing to be re-rendered, all in one event tick. React will then iterate over the entire tree during a single batched render pass. Any component flagged as dirty will definitely re-render, and React will recursively re-render any children of those components

markerikson commented on 5 Oct 2020 Owner

Original author: **Dennis Cual** @denniscual
Original date: 2020-06-22T07:21:28Z

It means that there's no point to memoize data like function if in the first place, it can never help and could just add some little overhead because of memo process. Like if you use the function object to the "host components" like div because it doesn't render anything than to itself. Or passing not memo function object to a Component but the perf doesn't affect.

markerikson commented on 5 Oct 2020 Owner

Original author: **Dennis Cual** @denniscual
Original date: 2020-06-22T07:36:56Z

Imo, referencing the unstated solution, in the official React documentation, to this blog is not a good choice like you said the "observedBits" because theres a possibility that it would change in the future. And about your question in button onClick handler, in React reconciliation process the button will be the same but it will only mutate the onClick prop. It means that engine will not destroy the button element rather will reuse the same button element then update onCLick handler which is not the expensive at all and will not lose some dom state like focus, etc.

markerikson commented on 5 Oct 2020 Owner

Original author: **Benjamin S. @benjamin_such**
Original date: 2020-08-31T08:27:53Z

Hey @markerikson:disqus, really great article! What I really struggled with was/is the memoization of objects. A classic example would be something like:

`// We receive `customConfig` from a propconst { config } = useContext(SomeContext)const mergedConfig = useMemo(() => ({ ...config, ...customConfig }), [config, customConfig])`

In the past I thought this would help, but it doesn't (obviously now) since shallow comparison does not realize it's the same object and will recalculate `mergedConfig`. I feel like this approach is wrong, because I don't see a solution except extracting every key from `customConfig` and put it into the dependency array which sounds absolutely horrible lol. Can you help me in this regard?

May I ask how you gathered all that knowledge? Was your initial motivation just pure interest in React and thus read all the code? I'm really curious how you approach learning all this, maybe I can take something with me :P

benjaminsuch commented on 9 Oct 2020

> *Original author:* **Benjamin S. @benjamin_such**
> *Original date: 2020-08-31T08:27:53Z*
>
> Hey @markerikson:disqus, really great article! What I really struggled with was/is the memoization of objects. A classic example would be something like:
>
> `We receive "customConfig" from a prop`
> `const { config } = useContext(SomeContext)`
> `const mergedConfig = useMemo(() => ({ ...config, ...customConfig }), [config, customConfig])`
>
> In the past I thought this would help, but it doesn't (obviously now) since shallow comparison does not realize it's the same object and will recalculate `mergedConfig`. I feel like this approach is wrong, because I don't see a solution except extracting every key from `customConfig` and put it into the dependency array which sounds absolutely horrible lol. Can you help me in this regard?
>
> May I ask how you gathered all that knowledge? Was your initial motivation just pure interest in React and thus read all the code? I'm really curious how you approach learning all this, maybe I can take something with me :P

I think I the answer on my own, which is: There is no way to memoize objects and make sure to memoize values coming from `props` as much as possible. And one could also use `react-fast-compare` and do a `isEqual` check in `useMemo`.

webMasterMrBin commented on 3 Dec 2020

Really useful
react bible
@markerikson 👍

👍 3

taurscratch commented on 19 Jan 2021

Good read. Thank you.

maksymdukov commented on 21 Feb 2021

Thanks for the guide. It's great!

sabres207 commented on 18 Oct 2021

Thank you so much for that.
I have a question tho - you keep stating that react doesn't really care about change to props in regard to rerendering.
But in the excellent lifecycle diagram you attached to, it shows that what cause render is both setState and new props.

What cause the difference between the 2?
thanks!

👍 1

YagamiNewLight commented on 9 Dec 2021

Hey Mark, I'm really really appreciate all your awesome works on software engineering.

I got a question when reading the following paragraph:
Similarly, note that rendering `<MemoizedChild><OtherComponent /></MemoizedChild>` will also force the child to always render, because props.children is always a new reference.

What I unstanding is that you wanna us to be attentive to writing `<MemoizedChild><OtherComponent /></MemoizedChild>` , if the OtherComponent's type is a new reference in the parent render, not only the OtherComponent will get remounted, but also the `MemorizedChild` will be rerendered everytime and which is a kind of wasted work.

Does my unstanding is correct?

markerikson commented on 9 Dec 2021 Owner

@YagamiNewLight : not quite.

If we have:

```
function Parent() {
  return <MemoizedChild><OtherComponent /></MemoizedChild>;
}
```

In that case, `OtherComponent` is the same component each time, so that won't cause anything to be unmounted.

However, each time `Parent` renders, it passes a new element reference into `MemoizedChild` as `props.children` (ie, `{type: OtherComponent}`). So, `MemoizedChild` won't ever get to skip a re-render, because at least one of its props is always changing.

The real point here is that *if* you use `props.children` in your component, there's no point in wrapping it in `React.memo()`.

👍 1

YagamiNewLight commented on 9 Dec 2021

Pardon me please.. I don't get why `However, each time Parent renders, it passes a new element reference into MemoizedChild as props.children (ie, {type: OtherComponent}).`

If the `OtherComponent` is defined somewhere outside, then it is reference stable between the parent renders(because the parent doesn't create it on every render) and why does *each time Parent renders, it passes a new element reference into `MemoizedChild`*?

markerikson commented on 9 Dec 2021 Owner

@YagamiNewLight: remember that JSX is transformed into `React.createElement()` calls, and every call to `createElement()` returns a brand new JS object. So:

```
const el1 = React.createElement(OtherComponent);
const el2 = React.createElement(OtherComponent);

console.log(el1, el2); // {type: OtherComponent}, {type: OtherComponent}

console.log(el1 === el2) // FALSE - they are different references
```

So, every time `Parent` runs, it calls `React.createElement(MemoizedChild)` and `React.createElement(OtherComponent)`, and those generate new element objects.

❤️ 1 🚀 1

YagamiNewLight commented on 9 Dec 2021

OK, I get it. Many many thanks for your patient reply!!!

aqarain commented on 20 Dec 2021

Firstly great article Mark and thanks for this. I have one doubt. You wrote "React-Redux uses context to pass the Redux store instance, not the current state value".
How is the redux store instance different from the current state value? store instance is one big object which is

a value, no?
Please guide

YagamiNewLight commented on 21 Dec 2021

@aqarain
Store instance is not the classic `React state` which is so-called reactive to the UI.

You can think of the redux store instance as just a global plain object which can be read from anywhere, but the change of it *won't* cause any rerenders of the UI.

So here comes the `react-redux`, making the React component subscribe to the global redux store object, and auto-rerender when it changes. Context API is just a way of `making the redux store can be read from any component`.

So yes, *the store instance is one big object which is a value*, but not `reactive value`, and react-redux is to make it `reactive` to the component.

❤️ 1 🚀 1

aqarain commented on 21 Dec 2021

@YagamiNewLight
Thank you so much for this clarification

tanthongtan commented 3 months ago

Great Article! I have one question though, how will state updates work in React 18? How will they batch all setState calls? Will they use microtasks to do that?

markerikson commented 3 months ago Owner

@tanthongtan: reactwg/react-18#21 covers the overall intended behavior.

As far as I know, yes, they're using microtasks or something to enable running all queued updates at the very end of the event loop tick.

nareshbhatia commented a month ago

Thank you for this wonderful article, @markerikson. I have a question about the following statement:

> This means that by default, any state update to a parent component that renders a context provider will cause all of its descendants to re-render anyway, regardless of whether they read the context value or not!.

I am not quite seeing this behavior in my app. Here's the top level code from my sample repo:

```
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <ViewStateContextProvider>
      <App />
    </ViewStateContextProvider>
  </React.StrictMode>
);
```

The `ViewStateContextProvider` keeps my `viewState`, which consists of a simple boolean indicating if the view is in edit mode or not: `type ViewState = { isEditing: boolean };`.

My app has a button that toggles the view state. When I run this app in the React DevTools profiler and click on this button, only the following part of the tree gets re-rendered:

markerikson commented a month ago Owner

@nareshbhatia I'm going to guess that your provider component looks like this:

```
function ViewStateContextProvider(props) {
  const [someState, setSomeState] = useState(whatever);
  const contextValue = {someState, setSomeState};

  return (
    <MyContext.Provider value={contextValue}>
      // KEY PART HERE
      {props.children}
    </MyContext.Provider>
  )
}
```

That causes the "same element" optimization that I talked about to kick in, and React will stop recursing as soon as it sess that `props.children` was the same as the last time this component rendered.

nareshbhatia commented a month ago

@markerikson, you hit the nail on the head. I had missed that completely!!! Thank you for clarifying.

Spooky-Scary-Skeleton commented 2 weeks ago

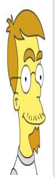
@markerikson Love this article! Feels like my itchy and annoying but not reachable point in my back is finally scratched!

Just one little question though.
In the "Component Render Optimization Techniques" paragraph's first example code

```
    <div>
      <button onClick={() => setCounter1(counter1 + 1)}>Counter 1: {counter1}</button>
      <button onClick={() => setCounter1(counter2 + 1)}>Counter 2: {counter2}</button>
      {memoizedElement}
    </div>
```

Did you use the same "setCounter1" on both buttons to show that memoized element won't change?
I thought that the second button should use "setCounter2" instead of "setCounter1".

← C r
(https://blog.isquaredsoftw .com/2020/05/blogged-
answers-w i-write/)



Mark
Erikson

Collector of
interesting links,
answerer of
questions

Newer →
(https://blog.isquaredsoftware.com/2020/06/blogged-
answers-react-components-reusability-and-
abstraction/)

Recent Posts

- Reactathon 2022: The Evolution of Redux Async Logic (/2022/05/presentations-evolution-redux-async-logic/)
- TS Congress 2022: Lesson from Maintaining TS Libraries (/2022/05/presentations-ts-lib-maintenance/)
- Idiomatic Redux: Designing the Redux Toolkit Listener Middleware (/2022/03/designing-rtk-listener-middleware/)
- I'm Joining Replay! (/2022/02/joining-replay/)
- Codebase Conversion: Migrating a MEAN AngularJS app to React, Next.js, and TypeScript (/2021/12/codebase-conversion-mean-react-next-ts/)

Top Tags (All Tags) (/tags/)

redux (/tags/redux)	**55**
javascript (/tags/javascript)	**53**
react (/tags/react)	**43**
greatest-hits (/tags/greatest-hits)	**30**
presentation (/tags/presentation)	**22**

Greatest Hits

Greatest Hits: The Most Popular and Most Useful Posts I've Written (/2020/08/greatest-hits/)

Redux - Not Dead Yet! (/2018/03/redux-not-dead-yet/)

Why React Context is Not a "State Management" Tool (and Doesn't Replace Redux) (/2021/01/context-redux-differences/)

A (Mostly) Complete Guide to React Rendering Behavior (/2020/05/blogged-answers-a-mostly-complete-guide-to-react-rendering-behavior/)

The State of Redux, May 2021 (/2021/05/state-of-redux-may-2021/)

When (and when not) to reach for Redux (https://changelog.com/posts/when-and-when-not-to-reach-for-redux)

The Tao of Redux, Part 1 - Implementation and Intent (/2017/05/idiomatic-redux-tao-of-redux-part-1/)

The History and Implementation of React-Redux (/2018/11/react-redux-history-implementation/)

Thoughts on React Hooks, Redux, and Separation of Concerns (/2019/07/blogged-answers-thoughts-on-hooks/)

React Boston 2019: Hooks HOCs, and Tradeoffs (/2019/09/presentation-hooks-hocs-tradeoffs/)

Using Git for Version Control Effectively (/2021/01/coding-career-git-usage/)

Series

Blogged Answers (/series/blogged-answers)	19
Codebase Conversion (/series/codebase-conversion)	4
Coding Career Advice (/series/coding-career-advice)	4
Declaratively Rendering Earth In 3d (/series/declaratively-rendering-earth-in-3d)	2
How Web Apps Work (/series/how-web-apps-work)	5
Idiomatic Redux (/series/idiomatic-redux)	8
Newsletter (/series/newsletter)	6
Practical Redux (/series/practical-redux)	13
Presentations (/series/presentations)	23
Site Administrivia (/series/site-administrivia)	5