


 [acdlite / react-fiber-architecture](#) Public

A description of React's new core algorithm, React Fiber

 10.8k stars  530 forks  Branches  Tags  Activity Star Notifications

<> Code

 Issues 15 Pull requests 19 Actions Projects Security Insights master 1 Branch 0 Tags  Go to file

Go to file

Code

...



acdlite Merge pull request #7 from wuct/master

8 years ago



README.md

Update README.md

8 years ago

 README

React Fiber Architecture

Introduction

React Fiber is an ongoing reimplementation of React's core algorithm. It is the culmination of over two years of research by the React team.

The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.

Other key features include the ability to pause, abort, or reuse work as new updates come in; the ability to assign priority to different types of updates; and new concurrency primitives.

About this document

Fiber introduces several novel concepts that are difficult to grok solely by looking at code. This document began as a collection of notes I took as I followed along with Fiber's implementation in the React project. As it grew, I realized it may be a helpful resource for others, too.

I'll attempt to use the plainest language possible, and to avoid jargon by explicitly defining key terms. I'll also link heavily to external resources when possible.

Please note that I am not on the React team, and do not speak from any authority. **This is not an official document.** I have asked members of the React team to review it for accuracy.

This is also a work in progress. **Fiber is an ongoing project that will likely undergo significant refactors before it's completed.** Also ongoing are my attempts at documenting its design here. Improvements and suggestions are highly welcome.

My goal is that after reading this document, you will understand Fiber well enough to [follow along as it's implemented](#), and eventually even be able to contribute back to React.

Prerequisites

I strongly suggest that you are familiar with the following resources before continuing:

- [React Components, Elements, and Instances](#) - "Component" is often an overloaded term. A firm grasp of these terms is crucial.
- [Reconciliation](#) - A high-level description of React's reconciliation algorithm.
- [React Basic Theoretical Concepts](#) - A description of the conceptual model of React without implementation burden. Some of this may not make sense on first reading. That's okay, it will make more sense with time.
- [React Design Principles](#) - Pay special attention to the section on scheduling. It does a great job of explaining the *why* of React Fiber.

Review

Please check out the prerequisites section if you haven't already.

Before we dive into the new stuff, let's review a few concepts.

What is reconciliation?

reconciliation

The algorithm React uses to diff one tree with another to determine which parts need to be changed.

update

A change in the data used to render a React app. Usually the result of `setState`. Eventually results in a re-render.

The central idea of React's API is to think of updates as if they cause the entire app to re-render. This allows the developer to reason declaratively, rather than worry about how to efficiently transition the app from any particular state to another (A to B, B to C, C to A, and so on).

Actually re-rendering the entire app on each change only works for the most trivial apps; in a real-world app, it's prohibitively costly in terms of performance. React has optimizations which create the appearance of whole app re-rendering while maintaining great performance. The bulk of these optimizations are part of a process called **reconciliation**.

Reconciliation is the algorithm behind what is popularly understood as the "virtual DOM." A high-level description goes something like this: when you render a React application, a tree of nodes that describes the app is generated and saved in memory. This tree is then flushed to the rendering environment — for example, in the case of a browser application, it's translated to a set of DOM operations. When the app is updated (usually via `setState`), a new tree is generated. The new tree is diffed with the previous tree to compute which operations are needed to update the rendered app.

Although Fiber is a ground-up rewrite of the reconciler, the high-level algorithm [described in the React docs](#) will be largely the same. The key points are:

- Different component types are assumed to generate substantially different trees. React will not attempt to diff them, but rather replace the old tree completely.
- Diffing of lists is performed using keys. Keys should be "stable, predictable, and unique."

Reconciliation versus rendering

The DOM is just one of the rendering environments React can render to, the other major targets being native iOS and Android views via React Native. (This is why "virtual DOM" is a bit of a misnomer.)

The reason it can support so many targets is because React is designed so that reconciliation and rendering are separate phases. The reconciler does the work of computing which parts of a tree have changed; the renderer then uses that information to actually update the rendered app.

This separation means that React DOM and React Native can use their own renderers while sharing the same reconciler, provided by React core.

Fiber reimplements the reconciler. It is not principally concerned with rendering, though renderers will need to change to support (and take advantage of) the new architecture.

Scheduling

scheduling

the process of determining when work should be performed.

work

any computations that must be performed. Work is usually the result of an update (e.g. `setState`).

React's [Design Principles](#) document is so good on this subject that I'll just quote it here:

In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start delaying some updates to avoid dropping frames.

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

The key points are:

- In a UI, it's not necessary for every update to be applied immediately; in fact, doing so can be wasteful, causing frames to drop and degrading the user experience.
- Different types of updates have different priorities — an animation update needs to complete more quickly than, say, an update from a data store.
- A push-based approach requires the app (you, the programmer) to decide how to schedule work. A pull-based approach allows the framework (React) to be smart and make those decisions for you.

React doesn't currently take advantage of scheduling in a significant way; an update results in the entire subtree being re-rendered immediately. Overhauling React's core algorithm to take advantage of scheduling is the driving idea behind Fiber.

Now we're ready to dive into Fiber's implementation. The next section is more technical than what we've discussed so far. Please make sure you're comfortable with the previous material before moving on.

What is a fiber?

We're about to discuss the heart of React Fiber's architecture. Fibers are a much lower-level abstraction than application developers typically think about. If you find yourself frustrated in your attempts to understand it, don't feel discouraged. Keep trying and it will eventually make sense. (When you do finally get it, please suggest how to improve this section.)

Here we go!

We've established that a primary goal of Fiber is to enable React to take advantage of scheduling. Specifically, we need to be able to

- pause work and come back to it later.
- assign priority to different types of work.
- reuse previously completed work.
- abort work if it's no longer needed.

In order to do any of this, we first need a way to break work down into units. In one sense, that's what a fiber is. A fiber represents a **unit of work**.

To go further, let's go back to the conception of [React components as functions of data](#), commonly expressed as

$$v = f(d)$$



It follows that rendering a React app is akin to calling a function whose body contains calls to other functions, and so on. This analogy is useful when thinking about fibers.

The way computers typically track a program's execution is using the [call stack](#). When a function is executed, a new **stack frame** is added to the stack. That stack frame represents the work that is performed by that function.

When dealing with UIs, the problem is that if too much work is executed all at once, it can cause animations to drop frames and look choppy. What's more, some of that work may be unnecessary if it's superseded by a more recent update. This is where the comparison between UI components and function breaks down, because components have more specific concerns than functions in general.

Newer browsers (and React Native) implement APIs that help address this exact problem: `requestIdleCallback` schedules a low priority function to be called during an idle period, and `requestAnimationFrame` schedules a high priority function to be called on the next animation frame. The problem is that, in order to use those APIs, you need a way to break rendering work into incremental units. If you rely only on the call stack, it will keep doing work until the stack is empty.

Wouldn't it be great if we could customize the behavior of the call stack to optimize for rendering UIs? Wouldn't it be great if we could interrupt the call stack at will and manipulate stack frames manually?

That's the purpose of React Fiber. Fiber is reimplementing the stack, specialized for React components. You can think of a single fiber as a **virtual stack frame**.

The advantage of reimplementing the stack is that you can [keep stack frames in memory](#) and execute them however (and *whenever*) you want. This is crucial for accomplishing the goals we have for scheduling.

Aside from scheduling, manually dealing with stack frames unlocks the potential for features such as concurrency and error boundaries. We will cover these topics in future sections.

In the next section, we'll look more at the structure of a fiber.

Structure of a fiber

Note: as we get more specific about implementation details, the likelihood that something may change increases. Please file a PR if you notice any mistakes or outdated information.

In concrete terms, a fiber is a JavaScript object that contains information about a component, its input, and its output.

A fiber corresponds to a stack frame, but it also corresponds to an instance of a component.

Here are some of the important fields that belong to a fiber. (This list is not exhaustive.)

type and key

The type and key of a fiber serve the same purpose as they do for React elements. (In fact, when a fiber is created from an element, these two fields are copied over directly.)

The type of a fiber describes the component that it corresponds to. For composite components, the type is the function or class component itself. For host components (`div` , `span` , etc.), the type is a string.

Conceptually, the type is the function (as in `v = f(d)`) whose execution is being tracked by the stack frame.

Along with the type, the key is used during reconciliation to determine whether the fiber can be reused.

child and sibling

These fields point to other fibers, describing the recursive tree structure of a fiber.

The child fiber corresponds to the value returned by a component's `render` method. So in the following example

```
function Parent() {  
  return <Child />  
}
```



The child fiber of `Parent` corresponds to `Child` .

The sibling field accounts for the case where `render` returns multiple children (a new feature in Fiber!):

```
function Parent() {  
  return [<Child1 />, <Child2 />]  
}
```



The child fibers form a singly-linked list whose head is the first child. So in this example, the child of `Parent` is `Child1` and the sibling of `Child1` is `Child2` .

Going back to our function analogy, you can think of a child fiber as a [tail-called function](#).

return

The return fiber is the fiber to which the program should return after processing the current one. It is conceptually the same as the return address of a stack frame. It can also be thought of as the parent fiber.

If a fiber has multiple child fibers, each child fiber's return fiber is the parent. So in our example in the previous section, the return fiber of `Child1` and `Child2` is `Parent`.

pendingProps and memoizedProps

Conceptually, props are the arguments of a function. A fiber's `pendingProps` are set at the beginning of its execution, and `memoizedProps` are set at the end.

When the incoming `pendingProps` are equal to `memoizedProps`, it signals that the fiber's previous output can be reused, preventing unnecessary work.

pendingWorkPriority

A number indicating the priority of the work represented by the fiber. The [ReactPriorityLevel](#) module lists the different priority levels and what they represent.

With the exception of `NoWork`, which is 0, a larger number indicates a lower priority. For example, you could use the following function to check if a fiber's priority is at least as high as the given level:

```
function matchesPriority(fiber, priority) {  
  return fiber.pendingWorkPriority !== 0 &&  
    fiber.pendingWorkPriority <= priority  
}
```



This function is for illustration only; it's not actually part of the React Fiber codebase.

The scheduler uses the priority field to search for the next unit of work to perform. This algorithm will be discussed in a future section.

alternate

flush

To flush a fiber is to render its output onto the screen.

work-in-progress

A fiber that has not yet completed; conceptually, a stack frame which has not yet returned.

At any time, a component instance has at most two fibers that correspond to it: the current, flushed fiber, and the work-in-progress fiber.

The alternate of the current fiber is the work-in-progress, and the alternate of the work-in-progress is the current fiber.

A fiber's alternate is created lazily using a function called `cloneFiber`. Rather than always creating a new object, `cloneFiber` will attempt to reuse the fiber's alternate if it exists, minimizing allocations.

You should think of the `alternate` field as an implementation detail, but it pops up often enough in the codebase that it's valuable to discuss it here.

output

host component

The leaf nodes of a React application. They are specific to the rendering environment (e.g., in a browser app, they are ``div``, ``span``, etc.). In JSX, they are denoted using lowercase tag names.

Conceptually, the output of a fiber is the return value of a function.

Every fiber eventually has output, but output is created only at the leaf nodes by **host components**. The output is then transferred up the tree.

The output is what is eventually given to the renderer so that it can flush the changes to the rendering environment. It's the renderer's responsibility to define how the output is created and updated.

Future sections

That's all there is for now, but this document is nowhere near complete. Future sections will describe the algorithms used throughout the lifecycle of an update. Topics to cover include:

- how the scheduler finds the next unit of work to perform.
- how priority is tracked and propagated through the fiber tree.
- how the scheduler knows when to pause and resume work.
- how work is flushed and marked as complete.
- how side-effects (such as lifecycle methods) work.
- what a coroutine is and how it can be used to implement features like context and layout.

Related Videos

- [What's Next for React \(ReactNext 2016\)](#)

Releases

No releases published

Packages

No packages published

Contributors 7

