React 的未来（2）：Suspense 带来的异步操作革命

上一节我们介绍了 Fiber 架构下的异步渲染机制，我们知道生命周期函数的修改是势在必行，那么，接下来呢？接下来 React 会有什么“大事”呢？

这个答案估计连 React 的核心开发者也在讨论中，不过从各种渠道信息看来，至少有两件“大事”在会在看得见的未来发生，那就是：

- Suspense
- Hooks

当然 React 增加的功能肯定远不止这点，将这两件“大事”在这里提出来，是因为它们对我们使用开发者的影响最大，会彻底改变我们的代码模式。

在写这本小册时，React 正式版是 v16.6.0，还只是 alpha 阶段，也许当你读到这本小册时，React 已经走得更远，但是你依然应该阅读这一小节，因为作为开发者你应该要明白技术演化的来龙去脉。

我们首先来了解 Suspense。Suspense 应用的场合就是异步数据处理，最常见的例子，就是通过 AJAX 从服务器获取数据，每一个 React 开发者都曾为这个问题纠结。

如果用一句话概括 Suspense 的功用，那就是：**用同步的代码来实现异步操作。**

而要理解 Suspense，我们先来体会一下 React 中做 AJAX 之类异步操作的痛苦。

React 同步操作的不足

上一节介绍过，React 最初的设计，整个渲染过程都是同步的。同步的意思是，当一个组件开始渲染之后，就必须一口气渲染完，不能中断，对于特别庞大的组件树，这个渲染过程会很耗时，而且，这种同步处理，也会导致我们的代码比较麻烦。

当我们开始渲染某个组件的时候，假设这个组件需要从服务器获取数据，那么，要么由这个组件的父组件想办法拿到服务器的数据，然后通过 props 传递进来，要么就要靠这个组件自力更生来获取数据，但是，没有办法通过一次渲染完成这个过程，因为渲染过程是同步的，不可能让 React 等待这个组件调用 AJAX 获取数据之后再继续渲染。

常用的做法，需要组件的 render 和 componentDidMount 函数配合。

1. 在 componentDidMount 中使用 AJAX，在 AJAX 成功之后，通过 setState 修改自身状态，这会引发一次新的渲染过程。
2. 在 render 函数中，如果 state 中没有需要的数据，就什么都不渲染或者渲染一个“正在装载”之类提示；如果 state 中已经有需要的数据，就可以正常渲染了，但这也必定是在 componentDidMount 修改了 state 之后，也就是只有在第二次渲染过程中才可以。

下面是代码实例：

```
class Foo extends React.Component {
  state = {
     data: null
  }

  render() {
     if (!this.state.data) {
        return null;
     } else {
        return <div>this.state.data</div>;
     }
  }

  componentDidMount() {
     callAPI().then(result => {
       this.setState({data: result});
     });
  }
}
```

这种方式虽然可行，我们也照这种套路写过不少代码，但它的缺点也是很明显的。

1. 组件必须要有自己的 state 和 componentDidMount 函数实现，也就不可能做成纯函数形式的组件。
2. 需要两次渲染过程，第一次是 mount 引发的渲染，由 componentDidMount 触发 AJAX 然后修改 state，然后第二次渲染才真的渲染出内容。
3. 代码啰嗦，十分啰嗦。

理想中的代码形式

而 Suspense 就是为了克服上述 React 的缺点。

在了解 Suspense 怎么解决这些问题之前，我们不妨自己想象一下，如果要利用 AJAX 获取数据，代码怎样写最简洁高效？

我先来说一说自己设想的最佳代码形式。首先，我不想写一个有状态的组件，因为通过 AJAX 获取的数据往往也就在渲染用一次，没必要存在 state 里；其次，想要使数据拿来就用，不需要经过 componentDidMount 走一圈。所以，代码最好是下面这样：

```
const Foo = () => {
  const data = callAPI();
  return <div>{data}</div>;
}
```

够简洁吧，可是目前的 React 版本做不到啊！

因为 `callAPI` 肯定是一个异步操作，不可能获得同步数据，无法在同步的 React 渲染过程中立足。

不过，现在做不到，不代表将来做不到，将来 React 会支持这样的代码形式，这也就是 Suspense。

Suspense

在 JsConf Iceland 2018 技术大会 上，React 的开发者展示了未来 React 会支持的新特性 Suspense，有了 Suspense，就可以在 React 中以同步的形式来写异步代码，代码形式类似下面：

```
const Foo = () => {
  const data = createFetcher(callAJAX).read();
  return <div>{data}</div>;
}
```

看到这里，你的第一反应很可能就是：怎么可能？

真的可能。

你还会想：这怎么做到的？怎么在同步的渲染过程中强行加入异步操作？

接下来，我们就介绍一下 Suspense 的原理。

在 React 推出 v16 的时候，就增加了一个新生命周期函数 componentDidCatch。如果某个组件定义了 componentDidCatch，那么这个组件中所有的子组件在渲染过程中抛出异常时，这个 componentDidCatch 函数就会被调用。

可以这么设想，componentDidCatch 就是 JavaScript 语法中的 `catch`，而对应的 `try` 覆盖所有的子组件，就像下面这样：

```
try {
  //渲染子组件
} catch (error) {
  // componentDidCatch被调用
}
```

Suspense 就是巧妙利用 componentDidCatch 来实现同步形式的异步处理。

Suspense 提供的 createFetcher 函数会封装异步操作，当尝试从 createFetcher 返回的结果读取数据时，有两种可能：一种是数据已经就绪，那就直接返回结果；还有一种可能是异步操作还没有结束，数据没有就绪，这时候 createFetcher 会抛出一个“异常”。

你可能会说，抛出异常，渲染过程不就中断了吗？

的确会中断，不过，createFetcher 抛出的这个“异常”比较特殊，这个“异常”实际上是一个 Promise 对象，这个 Promise 对象代表的就是异步操作，操作结束时，也是数据准备好的时候。当 componentDidCatch 捕获这个 Promise 类型的“异常”时，就可以根据这个 Promise 对象的状态改变来重新渲染对应组件，第二次渲染，肯定就能够成功。

下面是 createFetcher 的一个简单实现方式：

```
var NO_RESULT = {};

export const createFetcher = (task) => {
  let result = NO_RESULT;

  return () => {
    const p = task();

    p.then(res => {
      result = res;
    });


    if (result === NO_RESULT) {
      throw p;
    }

    return result;
  }
}
```

在上面的代码中，createFetcher 的参数 `task` 被调用应该返回一个 Promise 对象，这个对象在第一次调用时会被 throw 出去，但是，只要这个对象完结，那么 `result` 就有实际的值，不会再被 throw。

还需要一个和 createFetcher 配合的 Suspense，代码如下：

```
class Suspense extends React.Component {
  state = {
    pending: false
  }

  componentDidCatch(error) {
    // easy way to detect Promise type
    if (typeof error.then === 'function') {
      this.setState({pending: true});

      error.then(() => this.setState({
        pending: false
      }));
    }
  }

  render() {
    return this.state.pending ? null : this.props.children;
  }
}
```

上面的 Suspense 组件实现了 componentDidCatch，如果捕获的 `error` 是 Promise 类型，那就说明子组件用 createFetcher 获取异步数据了，就会等到它完结之后重设 state，引发一次新的渲染过程，因为 createFetcher 中会记录异步返回的结果，新的渲染就不会抛出异常了。

使用 createFetcher 和 Suspense 的示例代码如下：

```
const getName = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve('Morgan');
  }, 1000);
})

const fetcher = createFetcher(getName);

const Greeting = () => {
  return <div>Hello {fetcher()}</div>
};

const SuspenseDemo = () => {
  return (
    <Suspense>
      <Greeting />
    </Suspense>
  );
};
```

上面的 `getName` 利用 `setTimeout` 模拟了异步 AJAX 获取数据，第一次渲染 `Greeting` 组件时，会有 Promise 类型的异常抛出，被 `Suspense` 捕获。1 秒钟之后，当 `getName` 返回实际结果的时候，`Suspense` 会引发重新渲染，这一次 `Greeting` 会显示出 `hello Morgan`。

上面的 createFetcher 和 Suspense 是一个非常简陋的实现，主要用来让读者了解 Suspense 的工作原理，正式发布的 Suspense 肯定会具备更强大的功能。

React v16.6.0 对 Suspense 的支持

React 发布 v16.6.0 的时候，提供了 Suspense 组件，直接支持 Suspense 功能，但是还没有正式提供 createFetcher 的功能，只发布了一个独立但不稳定的 react-cache 包。这个包里的 unstable_createResource 相当于上面描述的 createFetcher。照这个命名来看，正式发布的时候这个 API 可能会叫做 createResource 而不是叫 createFetcher。

我们利用 React v16.6.0 和不稳定的 react-cache 来实现上述功能，代码如下：

```
import React, {Suspense} from 'react';

import {unstable_createResource as createResource} from 'react-cache';

const getName = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve('Morgan');
  }, 1000);
})

const resource = createResource(getName);

const Greeting = () => {
  return <div>hello {resource.read()}</div>
};

const SuspenseDemo = () => {
  return (
    <Suspense fallback={<div>loading...</div>} >
      <Greeting />
    </Suspense>
  );
};
```

在上面的代码中，我们使用 React 提供的 Suspense 组件，支持一个 `fallback` 属性，这个属性可以用于显示“装载中”界面。在上面的例子中，要等待 1 秒钟时间才得到模拟 API 的结果，这时候显示一个空白页面是肯定不合适的，在等待的这 1 秒钟里，显得就是一个“Loading...”字样。

很显然，需要一个最佳实践来控制 Suspense 的范围。如果我们只在组件树最顶层放一个 Suspense 组件，那么在 API 返回之前，整个页面只显示“装载中”，这样的用户体验并不好。正确的做法，是将每一个独立依赖某个 API 调用的组件用一个 Suspense 包住。

例如，一个页面中包括头部的 `Header`、左侧的导航栏 `LeftPanel` 和右侧的内容 `Content`，其中只有 `Header` 的渲染不依赖于 API，那么，JSX 可以这样写：

```
<div>
    <Header />
    <Suspense fallback={<LoadingSpin />}>
        <LeftPanel />
    </Suspense>
    <Suspense fallback={<LoadingSpin />}>
        <Content />
    </Suspense>
</div>
```

这样，网页首先显示 `Header`，然后无论 `LeftPanel` 还是 `Content` 中谁的 AJAX 首先返回结果，都可以立刻显示对应模块，而不用等待所有 AJAX 都返回才让用户看到更新。

Suspense 带来的 React 使用模式改变

Suspense 被推出之后，可以极大地减少异步操作代码的复杂度。

之前，只要有 AJAX 这样的异步操作，就必须要用两次渲染来显示 AJAX 结果，这就需要用组件的 state 来存储 AJAX 的结果，用 state 又意味着要把组件实现为一个 class。总之，我们需要做这些：

1. 实现一个 class；
2. class 中需要有 state；
3. 需要实现 componentDidMount 函数；
4. render 必须要根据 `this.state` 来渲染不同内容。

有了 Suspense 之后，不需要做上面这些杂事，只要一个函数形式组件就足够了。

在介绍 Redux 时，我们提到过在 Suspense 面前，Redux 的一切异步操作方案都显得繁琐，读者现在应该能够通过代码理解这一点了。

很可惜，目前 Suspense 还不支持服务器端渲染，当 Suspense 支持服务器端渲染的时候，那就真的会对 React 社区带来革命性影响。

小结

在这一小节中我们介绍了 Suspense 功能，读者应该可以了解到：

1. Suspense 解决异步操作的问题；
2. 有了 Supsense 之后，依赖 AJAX 的组件也可以是函数形式，不需要是 class。

留言

评论将在后台进行审核，审核通过后对所有人可见

jeffacode同学 前端开发 @ 不好意思无业游民
上一节说componentDidCatch发生在render之后，可这边的例子：
const Foo = () => {
 const data = createFetcher(callAJAX).read();
 return <div>{data}</div>;
}
明明都没执行到return就抛出异常了啊？

▲ 0 收起评论 19天前

程墨 Hulu
我没说过componentDidCatch发生在render之后，你是不是在某个地方看到componentDidMount在render之后？
19天前

评论审核通过后显示 评论

ITSheng
揪三个小错

引言第六段，「用同步的代码来实现异步操作」，末尾少了一个 * 号，导致 Markdown 被识别为斜体。

第一节，缺点里的第一条，「要由」应改为「要有」。

第二节第六段，「这个该组件中」，「这个」和「该」语义重复了

▲ 0 评论 28天前

position_柚子 前端
程墨老师，第一个例子是不是写反了：
render() {
  if (this.state.data) {
   return null;
  } else {
   return <div>this.state.data</div>;
  }
}
如果 data 有值的时候，return null？没有值的时候，return <... 展开全部

▲ 0 收起评论 1月前

程墨 Hulu
我少写了一个感叹号，改过来了
1月前

评论审核通过后显示 评论