05-进一步认识Hooks：如何正确理解函数组件的生命周期？

你好，我是王沛。今天我们来聊聊React组件的生命周期。

这一讲我会带你从 Hooks 的角度进一步理解 React 函数组件的生命周期。你可能会有疑问，前面几节课我们已经学习了 Hooks 的概念和内置 Hooks 的用法，那为什么还会有专门一讲来进一步介绍 Hooks 呢？

原因主要有两个。一方面，如果你已经熟悉了 Class 组件的生命周期，那么还需要一个转换编程思想到 Hooks 的过程，这样才能避免表面上用了 Hooks，却仍然用 Class 组件的方式思考问题。

另一方面，如果你是新接触 React，那么这一讲也能帮你了解原来 Class 组件的工作方式，从而如果看到或接手已有的 React 项目，也能从容应对。

忘掉 Class组件的生命周期

基于 Class 的组件作为 React 诞生之处就存在的机制，它的用法早已深入人心，甚至至今为止 React 的官方文档中仍然是以 Class 组件为基础的，而函数组件和 Hooks 则是作为新特性做了补充说明和解释。

这其实有两个原因：

- 一是 React 团队尽最大努力保持 API 的稳定，不希望给你造成一种 Class 组件将被废弃的感觉；
- 二是大量的存量应用其实还都是用 Class 组件实现的，无论是对于维护者还是加入者来说，了解 Class 组件都是很有必要的。

所以尽管函数组件和 Hooks 是未来发展的趋势，但你还是应该对 Class 组件的用法有一个了解，达到至少是能看懂的程度。

Class 组件和函数组件是两种实现 React 应用的方式，虽然它们是等价的，但是开发的思想有很大不同。如果你是从 Class 组件转换到 Hooks 的方式，那么很重要的一点就是，你要学会忘掉 Class 组件中的生命周期概念，千万不要将原来习惯的 Class 组件开发方式映射到函数组。

比如如何在函数组件中实现 componentDidMount, componentDidUpdate 这样的 Class 组件才有的生命周期方法，你应该通过理解 Hooks 的方式去思考业务需求应该如何实现。

为了理解函数组件的执行过程，我们不妨思考下 React 的本质：**从 Model 到 View 的映射**。假设状态永远不变，那么实际上函数组件就相当于是一个模板引擎，只执行一次。但是 React 本身正是为动态的状态变化而设计的，而可能引起状态变化的原因基本只有两个：

1. 用户操作产生的事件，比如点击了某个按钮。
2. 副作用产生的事件，比如发起某个请求正确返回了。

这两种事件本身并不会导致组件的重新渲染，但我们在这两种事件处理函数中，一定是因为改变了某个状态，这个状态可能是 State 或者 Context，从而导致了 UI 的重新渲染。

对于第一种情况，其实函数组件和 Class 组件的思路几乎完全一样：通过事件处理函数来改变某个状态；对于第二种情况，在函数组件中是通过 useEffect 这个 Hook 更加直观和语义化的方式来描述。对应到 Class 组件，则是通过手动判断 Props 或者 State 的变化来执行的。

比如对于在第三讲介绍的例子，一个用于显示博客文章的组件接收一个文章的 id 作为参数，然后根据这个 id 从服务器端获取文章的内容并显示出来。那么当 id 变化的时候，你就需要检测到这个变化，并重新发送请求，显示在界面上。在 Class 组件中，你通常要用如下的代码实现：

```
class BlogView extends React.Component {
  // ...
  componentDidMount() {
    // 组件第一次加载时去获取 Blog 数据
    fetchBlog(this.props.id);
  }
  componentDidUpdate(prevProps) {
    if (prevProps.id !== this.props.id) {
      // 当 Blog 的 id 发生变化时去获取博客文章
      fetchBlog(this.props.id);
    }
  }
  // ...
}
```

可以看到，在 Class 组件中，需要在两个生命周期方法中去实现副作用，一个是首次加载，另外一个则是每次 UI 更新后。而在函数组件中不再有生命周期的概念，而是提供了 useEffect 这样一个 Hook 专门用来执行副作用，因此，只需下面的代码即可实现同样的功能：

```
function BlogView({ id }) {
  useEffect(() => {
    // 当 id 变化时重新获取博客文章
    fetchBlog(id);
  }, [id]); // 定义了依赖项 id
}
```

可以看到，在函数组件中你要思考的方式永远是：**当某个状态发生变化时，我要做什么，**而不再是在 Class 组件中的某个生命周期方法中我要做什么。

所以如果你是从 Class 组件转型到函数组件，那么你要做的就是忘掉 Class 组件的生命周期机制，去逐渐习惯函数组件的思考方式。相信这样你就能够体会到函数组件带来的直观、简洁的好处了。

当然，要一下子彻底转变是很难的，你需要在实际开发中不断练习。那具体这样才能算是用 Hooks 方式思考呢？接下来我们就一起来对比看个例子，当一个需求来的时候，基于类组件的生命周期机制是如何做的，而在函数组件中又是如何做的。

重新思考组件的生命周期

在传统的类组件中，有专门定义的生命周期方法用于执行不同的逻辑，那么它们在函数组件的存在的形式又是什么样的呢？接下来我就带你一起看看在函数组件中，是如何思考组件的生命周期的。

构造函数

在类组件中有一个专门的方法叫 constructor，也就是构造函数，在里面我们会做一些初始化的事情，比如设置 State 的初始状态，或者定义一些类的实例的成员。

而现在，函数组件只是一个函数，没有所谓的对象，或者说类的实例这样的概念，那自然也就没有构造函数的说法了。

那么在函数组件中，我们应该如何去做一些初始化的事情呢？答案是：函数组件基本上没有统一的初始化需要，因为 Hooks 自己会负责自己的初始化。比如 useState 这个 Hook，接收的参数就是定义的 State 初始值。而在过去的类组件中，你通常需要在构造函数中直接设置 this.state ，也就是设置某个值来完成初始化。

但是要注意了，我提到的“基本上没有初始化需要”，也就是并不是完全没有。严格来说，虽然需求不多，但类组件中构造函数能做的不只是初始化 State，还可能有其它的逻辑。那么如果一定要在函数组件中实现构造函数应该怎么做呢？

这时候我们不妨思考下构造函数的本质，其实就是：在所以其它代码执行之前的一次性初始化工作。在函数组件中，因为没有生命周期的机制，那么转换一下思路，其实我们要实现的是：一次性的代码执行。

虽然没有直接的机制可以做到这一点，但是利用 useRef 这个 Hook，我们可以实现一个 useSingleton 这样的一次性执行某段代码的自定义 Hook，代码如下：

```
import { useRef } from 'react';

// 创建一个自定义 Hook 用于执行一次性代码
function useSingleton(callback) {
  // 用一个 called ref 标记 callback 是否执行过
  const called = useRef(false);
  // 如果已经执行过，则直接返回
  if (called.current) return;
  // 第一次调用时直接执行
  callBack();
  // 设置标记为已执行过
  called.current = true;
}
```

从而在一个函数组件中，可以调用这个自定义 Hook 来执行一些一次性的初始化逻辑：

```
import useSingleton from './useSingleton';

const MyComp = () => {
  // 使用自定义 Hook
  useSingleton(() => {
    console.log('这段代码只执行一次');
  });

  return (
    <div>My Component</div>
  );
};
```

代码中可以看到，useSingleton 这个 Hook 的核心逻辑就是定义只执行一次的代码。而是否在所有代码之前执行，则取决于在哪里调用，可以说，它的功能其实是包含了构造函数的功能的。

所以你在日常开发中，是无需去将功能映射到传统的生命周期的构造函数的概念，而是要从函数的角度出发，去思考功能如何去实现。比如在这个例子中，我们需要的其实就是抓住某段代码只需要执行一次这样一个本质的需求，从而能够更自然地用 Hooks 解决问题。

三种常用的生命周期方法

在类组件中，componentDidMount，componentWillUnmount，和componentDidUpdate这三个生命周期方法可以说是日常开发最常用的。业务逻辑通常要分散到不同的生命周期方法中，比如说在上面的 Blog 文章的例子中，你需要同时在 componentDidMount 和 componentDidUpdate 中去获取数据。

而在函数组件中，这几个生命周期方法可以统一到 useEffect 这个 Hook，正如useEffect 的字面含义，它的作用就是触发一个副作用，即在组件每次 render 之后去执行。

在第三讲中其实你已经看到了 useEffect 的用法，下面的代码演示了这三个生命周期方法是如何用 useEffect 实现的：

```
useEffect(() => {
  // componentDidMount + componentDidUpdate
  console.log('这里基本等价于 componentDidMount + componentDidUpdate');
  return () => {
    // componentWillUnmount
    console.log('这里基本等价于 componentWillUnmount');
  }
}, [deps])
```

可以看到，useEffect 接收的 callback 参数，可以返回一个用于清理资源的函数，从而在下一次同样的 Effect 被执行之前被调用。

你可能已经注意到了，在代码里我用了“基本等价于”这个说法，什么意思呢？指的就是这个写法并没有完全等价于传统的这几个生命周期方法。主要有两个原因。

一方面，useEffect(callback) 这个 Hook 接收的 callback，只有在依赖项变化时才被执行。而传统的 componentDidUpdate 则一定会执行。这样来看，Hook 的机制其实更具有语义化，因为过去在 componentDidUpdate 中，我们通常都需要手动判断某个状态是否发生变化，然后再执行特定的逻辑。

另一方面，callback 返回的函数（一般用于清理工作）在下一次依赖项发生变化以及组件销毁之前执行，而传统的 componentWillUnmount 只在组件销毁时才会执行。

第二点可能有点难理解，我们不妨继续看博客文章这个例子。假设当文章 id 发生变化时，我们不仅需要获取文章，同时还要监听某个事件，这样在有新的评论时获得通知，就能显示新的评论了。这时候的代码结构如下：

```
import React, { useEffect } from 'react';
import comments from './comments';

function BlogView({ id }) {
  const handleCommentsChange = useCallback(() => {
    // 处理评论变化的通知
  }, []);
  useEffect(() => {
    // 获取博客内容
    fetchBlog(id);
    // 监听指定 id 的博客文章的评论变化通知
    const listener = comments.addListener(id, handleCommentsChange);

    return () => {
      // 当 id 发生变化时，移除之前的监听
      comments.removeListener(listener);
    };
  }, [id, handleCommentsChange])
}
```

那么可以比较清楚地看到，useEffect 接收的返回值是一个回调函数，这个回调函数不只是会在组件销毁时执行，而且是每次 Effect 重新执行之前都会执行，用于清理上一次 Effect 的执行结果。

理解这一点非常重要。useEffect 中返回的回调函数，只是清理当前执行的 Effect 本身。这其实是更加语义化的，因此你不用将其映射到 componentWillUnmount，它也完全不等价于 componentWillUnmount。你只需记住它的作用就是用于清理上一次 Effect 的结果就行了，这样在实际的开发中才能够使用得更加自然和合理。

其它的生命周期方法

刚才我列举了几个 Class 组件中最为常用的生命周期方法，并介绍了对于同样的需求，在函数组件中应该如何去用 Hooks 的机制重新思考它们的实现。这已经能覆盖绝大多数的应用场景了。

但是 Class 组件中还有其它一些比较少用的方法，比如 getSnapshotBeforeUpdate, componentDidCatch, getDerivedStateFromError。比较遗憾的是目前 Hooks 还没法实现这些功能。因此如果必须用到，你的组件仍然需要用类组件去实现。

已有应用是否应该迁移到 Hooks?

说了这么多，你可能会觉得写 React 应用就一定非 Hooks 不可了，其实也并非绝对。比如说很多时候，你面临的可能并不是开始一个全新的项目，而是参与到一个已有的项目中。那么就很可能会遇到这样一个问题：对于已有项目中的 Class 组件，是否要重构到函数组件和 Hooks 呢?

答案其实很明确：完全没必要。

在 React 中，Class 组件和函数组件是完全可以共存的。对于新的功能，我会更推荐使用函数组件。而对于已有的功能，则维持现状就可以。除非要进行大的功能改变，可以顺便把相关的类组件进行重构，否则是没有必要进行迁移的。

因为终究来说，能正确工作的代码就是好代码。React 组件的两种写法本身就可以很好地一起工作了：

1. 类组件和函数组件可以互相引用；
2. Hooks 很容易就能转换成高阶组件，并供类组件使用。

总结来说，我们完全没必要为了迁移而迁移。

小结

这一讲我们主要学习了函数组件和类组件在思考方式上的区别，虽然 Hooks 在功能上基本可以映射到传统的 Class 组件的生命周期方法，但是它们却又不是完全等价的。在实现具体的业务功能的时候，都应该尽量从 Hooks 的语义角度出发去思考组件是如何展现和交互的，这样才能更加顺滑地切换到函数组件的开发方式。

思考题

今天这节课我们只描述了 componentWillUnmount 近似的实现：组件销毁和文章 id 变化时执行。那么在函数组件中如果要实现严格的 componentWillUnmount，也就是只在组件销毁时执行，应该如何实现？

欢迎把你的思考和想法分享在留言区，我会和你交流讨论。下节课再见！

精选留言：

- 毛毛开飞机 2021-06-03 10:48:43
  useEffect(() => {
  return () => {
  // 这里只会在组件销毁前（componentWillUnmount）执行一次
  }
  }, []) [3赞]

  作者回复2021-06-03 21:48:54
  正确~

- 海洋 2021-06-03 06:54:37
  useEffect 第二个参数传入空，就只在组件初始化时和销毁前分别执行一次 [1赞]

  作者回复2021-06-03 21:45:25
  没错~ 传入空数组就可以了。

- 大大小小 2021-06-03 23:45:43
  BlogView那个例子，为什么要把useCallback返回的函数作为useEffect的依赖呢？是要达到什么目的吗？

- Ada 2021-06-03 12:16:03
  如果想要实现只执行一次的功能，用useEffect，依赖项传空数组，不是可以实现吗？
  为什么要写一个自定义钩子？ [1赞]

  作者回复2021-06-03 21:46:05
  因为需要在函数体执行之前执行。useEffect 是在 render 完后执行。

- 浩然 2021-06-03 10:15:56
  useEffect(() => {
  return () => {

```
// 组件销毁时执行
}
}, [])
```

作者回复2021-06-03 21:45:35

正确！