



技术成就梦想 (HTTP://SPARKGIS.COM/)

一切皆有可能

[首页 \(HTTP://SPARKGIS.COM/\)](http://sparkgis.com/)
[联系方式 \(HTTP://SPARKGIS.COM/CONTACT/\)](http://sparkgis.com/contact/)

精读《useEffect 完全指南》

1. 引言

工具型文章要跳读，而文学经典就要反复研读。如果说 React **0.14** 版本带来的各种生命周期可以类比到工具型文章，那么 **16.7** 带来的 Hooks 就要像文学经典一样反复研读。

Hooks API 无论从简洁程度，还是使用深度角度来看，都大大优于之前生命周期的 API，所以必须反复理解，反复实践，否则只能停留在表面原地踏步。

相比 **useState** 或者自定义 Hooks 而言，最有理解难度的是 **useEffect** 这个工具，希望借着 [a-complete-guide-to-useeffect](#) 一文，深入理解 **useEffect**。

原文非常长，所以概述是笔者精简后的。作者是 [Dan Abramov](#)，React 核心开发者。

2. 概述

unLearning，也就是学会忘记。你之前的学习经验会阻碍你进一步学习。

想要理解好 **useEffect** 就必须先深入理解 Function Component 的渲染机制，Function Component 与 Class Component 功能上的不同在上一期精读 [精读《Function VS Class 组件》](#) 已经介绍，而他们还存在思维上的不同：

Function Component 是更彻底的状态驱动抽象，甚至没有 Class Component 生命周期的概念，只有一个状态，而 React 负责同步到 DOM。这是理解 Function Component 以及 **useEffect** 的关键，后面还会详细介绍。

由于原文非常非常的长，所以笔者精简下内容再重新整理一遍。原文非常长的另一个原因是采用了启发式思考与逐层递进的方式写作，笔者最大程度保留这个思维框架。

从几个疑问开始

假设读者有比较丰富的前端 & React 开发经验，并且写过一些 Hooks。那么你也许觉得 Function Component 很好用，但美中不足的是，总有一些疑惑萦绕在心中，比如：

- 🤔 如何用 **useEffect** 代替 **componentDidMount**？
- 🤔 如何用 **useEffect** 取数？参数 **[]** 代表什么？
- 🤔**useEffect** 的依赖可以是函数吗？是哪些函数？
- 🤔 为何有时候取数会触发死循环？
- 🤔 为什么有时候在 **useEffect** 中拿到的 state 或 props 是旧的？

第一个问题可能已经自问自答过无数次了，但下次写代码的时候还是会忘。笔者也一样，而且在三期不同的精读中都分别介绍过这个问题：

- 精读《React Hooks》
- 精读《怎么用 React Hooks 造轮子》
- 精读《Function VS Class 组件》

但第二天就忘记了，因为 用 Hooks 实现生命周期确实别扭。 讲真，如果想彻底解决这个问题，就请你忘掉 React、忘掉生命周期，重新理解一下 Function Component 的思维方式吧！

上面 5 个问题的解答就不赘述了，读者如果有疑惑可以去 原文 TLDR 查看。

要说清楚 **useEffect**，最好先从 Render 概念开始理解。

每次 Render 都有自己的 Props 与 State

可以认为每次 Render 的内容都会形成一个快照并保留下来，因此当状态变更而 Rerender 时，就形成了 N 个 Render 状态，而每个 Render 状态都拥有自己固定不变的 Props 与 State。

看下面的 **count**：

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

在每次点击时，**count** 只是一个不会变的常量，而且也不存在利用 **Proxy** 的双向绑定，只是一个常量存在于每次 Render 中。

初始状态下 **count** 值为 **0**，而随着按钮被点击，在每次 Render 过程中，**count** 的值都会被固化为 **1、2、3**：

```
// During first render
function Counter() {
  const count = 0; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>;
  // ...
}

// After a click, our function is called again
function Counter() {
  const count = 1; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>;
  // ...
}

// After another click, our function is called again
function Counter() {
  const count = 2; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>;
  // ...
}
```

其实不仅是对象，函数在每次渲染时也是独立的。这就是 **Capture Value** 特性，后面遇到这种情况就不会一一展开，只描述为 “此处拥有 **Capture Value** 特性”。

每次 Render 都有自己的事件处理

解释了为什么下面的代码会输出 5 而不是 3:

```
const App = () => {
  const [temp, setTemp] = React.useState(5);

  const log = () => {
    setTimeout(() => {
      console.log("3 秒前 temp = 5, 现在 temp =", temp);
    }, 3000);
  };

  return (
    <div
      onClick={() => {
        log();
        setTemp(3);
        // 3 秒前 temp = 5, 现在 temp = 5
      }}
    >
      xyz
    </div>
  );
};
```

在 **log** 函数执行的那个 Render 过程里，**temp** 的值可以看作常量 5，执行 **setTemp(3)** 时会交由一个全新的 Render 渲染，所以不会执行 **log** 函数。而 3 秒后执行的内容是由 **temp** 为 5 的那个 Render 发出的，所以结果自然为 5。

原因就是 **temp**、**log** 都拥有 Capture Value 特性。

每次 Render 都有自己的 Effects

useEffect 也一样具有 Capture Value 的特性。

useEffect 在实际 DOM 渲染完毕后执行，那 **useEffect** 拿到的值也遵循 Capture Value 的特性：

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

上面的 **useEffect** 在每次 `Render` 过程中，拿到的 **count** 都是固化下来的常量。

如何绕过 Capture Value

利用 **useRef** 就可以绕过 `Capture Value` 的特性。可以认为 **ref** 在所有 `Render` 过程中保持着唯一引用，因此所有对 **ref** 的赋值或取值，拿到的都只有一个最终状态，而不会在每个 `Render` 间存在隔离。

```
function Example() {
  const [count, setCount] = useState(0);
  const latestCount = useRef(count);

  useEffect(() => {
    // Set the mutable latest value
    latestCount.current = count;
    setTimeout(() => {
      // Read the mutable latest value
      console.log(`You clicked ${latestCount.current} times`);
    }, 3000);
  });
  // ...
}
```

也可以简洁的认为，**ref** 是 `Mutable` 的，而 **state** 是 `Immutable` 的。

回收机制

在组件被销毁时，通过 `useEffect` 注册的监听需要被销毁，这一点可以通过 `useEffect` 的返回值做到：

```
useEffect(() => {
  ChatAPI.subscribeToFriendStatus(props.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.id, handleStatusChange);
  };
});
```

在组件被销毁时，会执行返回值函数内回调函数。同样，由于 Capture Value 特性，每次 “注册” “回收” 拿到的都是成对的固定值。

用同步取代“生命周期”

Function Component 不存在生命周期，所以不要把 Class Component 的生命周期概念搬过来试图对号入座。Function Component 仅描述 UI 状态，React 会将其同步到 DOM，仅此而已。

既然是状态同步，那么每次渲染的状态都会固化下来，这包括 `state props useEffect` 以及写在 Function Component 中的所有函数。

然而舍弃了生命周期的同步会带来一些性能问题，所以我们需要告诉 React 如何比对 Effect。

告诉 React 如何对比 Effects

虽然 React 在 DOM 渲染时会 diff 内容，只对改变部分进行修改，而不是整体替换，但却做不到对 Effect 的增量修改识别。因此需要开发者通过 `useEffect` 的第二个参数告诉 React 用到了哪些外部变量：

```
useEffect(() => {
  document.title = "Hello, " + name;
}, [name]); // Our deps
```

直到 `name` 改变时的 Rerender，`useEffect` 才会再次执行。

然而手动维护比较麻烦而且可能遗漏，因此可以利用 `eslint` 插件自动提示 + FIX：

不要对 Dependencies 撒谎

如果你明明使用了某个变量，却没有申明在依赖中，你等于向 React 撒了谎，后果就是，当依赖的变量改变时，`useEffect` 也不会再次执行：

```
useEffect(() => {
  document.title = "Hello, " + name;
}, []); // Wrong: name is missing in dep
```

这看上去很蠢，但看看另一个例子呢？

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);

  return <h1>{count}</h1>;
}
```

setInterval 我们只想执行一次，所以我们自以为聪明的向 React 撒了谎，将依赖写成 []。

“组件初始化执行一次 **setInterval**，销毁时执行一次 **clearInterval**，这样的代码符合预期。” 你心里可能这么想。

但是你错了，由于 `useEffect` 符合 `Capture Value` 的特性，拿到的 **count** 值永远是初始化的 **0**。相当于 **setInterval** 永远在 **count** 为 **0** 的 `Scope` 中执行，你后续的 **setCount** 操作并不会产生任何作用。

诚实的代价

笔者稍稍修改了一下标题，因为诚实是要付出代价的：

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(count + 1);
  }, 1000);
  return () => clearInterval(id);
}, [count]);
```

你老实告诉 React “嘿，等 **count** 变化后再执行吧”，那么你会得到一个好消息和两个坏消息。

好消息是，代码可以正常运行了，拿到了最新的 **count**。

坏消息有：

- 1. 计时器不准了，因为每次 **count** 变化时都会销毁并重新计时。
- 2. 频繁 生成/销毁 定时器带来了一定性能负担。

怎么既诚实又高效呢？

上述例子使用了 **count**，然而这样的代码很别扭，因为你在一个只想执行一次的 `Effect` 里依赖了外部变量。

既然要诚实，那只好 想办法不依赖外部变量：

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(c => c + 1);
  }, 1000);
  return () => clearInterval(id);
}, []);
```

setCount 还有一种函数回调模式，你不需要关心当前值是什么，只要对 “旧的值” 进行修改即可。这样虽然代码永远运行在第一次 `Render` 中，但总是可以访问到最新的 **state**。

将更新与动作解耦

你可能发现了，上面投机取巧的方式并没有彻底解决所有场景的问题，比如同时依赖了两个 **state** 的情况：

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(c => c + step);
  }, 1000);
  return () => clearInterval(id);
}, [step]);
```

你会发现不得不依赖 **step** 这个变量，我们又回到了 “诚实的代价” 那一章。当然 `Dan` 一定会给我们解法的。

利用 **useEffect** 的兄弟 **useReducer** 函数，将更新与动作解耦就可以了：


```
const [state, dispatch] = useReducer(reducer, initialState);
const { count, step } = state;

useEffect(() => {
  const id = setInterval(() => {
    dispatch({ type: "tick" }); // Instead of setCount(c => c + step);
  }, 1000);
  return () => clearInterval(id);
}, [dispatch]);
```

这就是一个局部 “Redux”，由于更新变成了 `dispatch({ type: "tick" })` 所以不管更新时需要依赖多少变量，在调用更新的动作里都不需要依赖任何变量。 具体更新操作在 `reducer` 函数里写就可以了。[在线 Demo](#)。

Dan 也将 `useReducer` 比作 Hooks 的金手指模式，因为这充分绕过了 Diff 机制，不过确实能解决痛点！

将 Function 挪到 Effect 里

在 “告诉 React 如何对比 Diff” 一章介绍了依赖的重要性，以及对 React 要诚实。那么如果函数定义不在 `useEffect` 函数体内，不仅可能会遗漏依赖，而且 [eslint](#) 插件也无法帮助你自动收集依赖。

你的直觉会告诉你这样做会带来更多麻烦，比如如何复用函数？是的，只要不依赖 Function Component 内变量的函数都可以安全的抽出去：

```
// ✅ Not affected by the data flow
function getFetchUrl(query) {
  return "https://hn.algolia.com/api/v1/search?query=" + query;
}
```

但是依赖了变量的函数怎么办？

如果非要把 Function 写在 Effect 外面呢？

如果非要这么做，就用 `useCallback` 吧！

```
function Parent() {
  const [query, setQuery] = useState("react");

  // ✅ Preserves identity until query changes
  const fetchData = useCallback(() => {
    const url = "https://hn.algolia.com/api/v1/search?query=" + query;
    // ... Fetch data and return it ...
  }, [query]); // ✅ Callback deps are OK

  return <Child fetchData={fetchData} />;
}

function Child({ fetchData }) {
  let [data, setData] = useState(null);

  useEffect(() => {
    fetchData().then(setData);
  }, [fetchData]); // ✅ Effect deps are OK

  // ...
}
```

由于函数也具有 Capture Value 特性，经过 **useCallback** 包装过的函数可以当作普通变量作为 **useEffect** 的依赖。**useCallback** 做的事情，就是在其依赖变化时，返回一个新的函数引用，触发 **useEffect** 的依赖变化，并激活其重新执行。

useCallback 带来的好处

在 Class Component 的代码里，如果希望参数变化就重新取数，你不能直接比对取数函数的 Diff：

```
componentDidUpdate(prevProps) {
  // ❌ This condition will never be true
  if (this.props.fetchData !== prevProps.fetchData) {
    this.props.fetchData();
  }
}
```

反之，要比对的是取数参数是否变化：

```
componentDidUpdate(prevProps) {
  if (this.props.query !== prevProps.query) {
    this.props.fetchData();
  }
}
```

但这种代码不内聚，一旦取数参数发生变化，就会引发多处代码的维护危机。

反观 Function Component 中利用 **useCallback** 封装的取数函数，可以直接作为依赖传入 **useEffect**，**useEffect** 只要关心取数函数是否变化，而取数参数的变化在 **useCallback** 时关心，再配合 eslint 插件的扫描，能做到 依赖不丢、逻辑内聚，从而容易维护。

更更更内聚

除了函数依赖逻辑内聚之外，我们再看看取数的全过程：

一个 Class Component 的普通取数要考虑这些点：

- 1. 在 **didMount** 初始化发请求。
- 2. 在 **didUpdate** 判断取数参数是否变化，变化就调用取数函数重新取数。
- 3. 在 **unmount** 生命周期添加 flag，在 **didMount didUpdate** 两处做兼容，当组件销毁时取消取数。

你会觉得代码跳来跳去的，不仅同时关心取数函数与取数参数，还要在不同生命周期里维护多套逻辑。那么换成 Function Component 的思维是怎样的呢？

笔者利用 useCallback 对原 Demo 进行了改造。

```
function Article({ id }) {
  const [article, setArticle] = useState(null);

  // 取数函数：只关心依赖的 id
  const fetchArticle = useCallback(async () => {
    const article = await API.fetchArticle(id);
    if (!didCancel) {
      setArticle(article);
    }
  }, [id]);

  // 副作用，只关心依赖了取数函数
  useEffect(() => {
    // didCancel 赋值与变化的位置更内聚
    let didCancel = false;
    fetchArticle(didCancel);

    return () => {
      didCancel = true;
    };
  }, [fetchArticle]);

  // ...
}
```

当你真的理解了 Function Component 理念后，就可以理解 Dan 的这句话：虽然 **useEffect** 前期学习成本更高，但一旦你正确使用了它，就能比 Class Component 更好的处理边缘情况。

useEffect 只是底层 API，未来业务接触到的是更多封装后的上层 API，比如 **useFetch** 或者 **useTheme**，它们会更好用。

3. 精读

原文有 9000+ 单词，非常长。但同时也配合一些 GIF 动图生动解释了 Render 执行原理，如果你想用好 Function Component 或者 Hooks，这篇文章几乎是必读的，因为没有人能猜到什么是 Capture Value，然而不能理解这个概念，Function Component 也不能用的顺手。

重新捋一下这篇文章的思路：

1. 从介绍 Render 引出 Capture Value 的特性。
2. 拓展到 Function Component 一切均可 Capture，除了 Ref。
3. 从 Capture Value 角度介绍 useEffect 的 API。

- 4. 介绍了 Function Component 只关注渲染状态的事实。
- 5. 引发了如何提高 useEffect 性能的思考。
- 6. 介绍了不要对 Dependencies 撒谎的基本原则。
- 7. 从不得不撒谎的特例中介绍了如何用 Function Component 思维解决这些问题。
- 8. 当你学会用 Function Component 理念思考时，你逐渐发现它的一些优势。
- 9. 最后点出了逻辑内聚，高阶封装这两大特点，让你同时领悟到 Hooks 的强大与优雅。

可以看到，比写框架更高的境界是发现代码的美感，比如 Hooks 本是为增强 Function Component 能力而创造，但在抛出问题-解决问题的过程中，可以不断看到规则限制，换一个角度打破它，最后体会到整体的逻辑之美。

从这篇文章中也可以读到如何增强学习能力。作者告诉我们，学会忘记可以更好的理解。我们不要拿生命周期的固化思维往 Hooks 上套，因为那会阻碍我们理解 Hooks 的理念。

另补充一些零碎的内容。

useEffect 还有什么优势

useEffect 在渲染结束时执行，所以不会阻塞浏览器渲染进程，所以使用 Function Component 写的项目一般都有用更好的性能。

自然符合 React Fiber 的理念，因为 Fiber 会根据情况暂停或插队执行不同组件的 Render，如果代码遵循了 Capture Value 的特性，在 Fiber 环境下会保证值的安全访问，同时弱化生命周期也能解决中断执行时带来的问题。

useEffect 不会在服务端渲染时执行。

由于在 DOM 执行完毕后才执行，所以能保证拿到状态生效后的 DOM 属性。

4. 总结

最后，提两个最重要的点，来检验你有没有读懂这篇文章：

- 1. Capture Value 特性。
- 2. 一致性。将注意放在依赖上（**useEffect** 的第二个参数 []），而不是关注何时触发。

你对 “一致性” 有哪些更深的解读呢？欢迎留言回复。

讨论地址是：[精读《useEffect 完全指南》 · Issue #138 · dt-fe/weekly](#)




如果你想参与讨论，请 [点击这里](#)，每周都有新的主题，周末或周一发布。前端精读 - 帮你筛选靠谱的内容。

special Sponsors

- [DevOps 全流程平台](#)

版权声明： 自由转载-非商用-非衍生-保持署名（[创意共享 3.0 许可证](#)）

共享此文章：

-  (<http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-14/?share=twitter&nb=1>)
-  (<http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-14/?share=facebook&nb=1>)
-  (<http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-14/?share=google-plus-1&nb=1>)

相关

精读《useEffect 完全指南》 (http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-11/) 三月 25, 2019 在“框架”中	精读《useEffect 完全指南》 (http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-10/) 三月 25, 2019 在“框架”中	精读《useEffect 完全指南》 (http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-13/) 三月 25, 2019 在“框架”中
--	--	--

PUBLISHED ON 三月 25, 2019 由 [SPARK \(HTTP://SPARKGIS.COM/AUTHOR/SPARK/\)](http://sparkgis.com/author/spark/). 版面：[框架 \(HTTP://SPARKGIS.COM/CATEGORY/%E6%A1%86%E6%9E%B6/\)](http://sparkgis.com/category/%E6%A1%86%E6%9E%B6/).

[← 精读《useEffect 完全指南》](#)
(<http://sparkgis.com/2019/03/25/%e7%b2%be%e8%af%bb%e3%80%8auseeffect-%e5%ae%8c%e5%85%a8%e6%8c%87%e5%8d%97%e3%80%8b-13/>)

[微服务架构 – 解决Docker-Compose服务编排启动顺序问题 →](#)
(<http://sparkgis.com/2019/03/25/%e5%be%ae%e6%9c%8d%e5%8a%a1%e6%9e%b6%e6%9e%84-%e8%a7%a3%e5%86%b3docker-compose%e6%9c%8d%e5%8a%a1%e7%bc%96%e6%8e%92%e5%90%af%e5%8a%a8%e9%a1%ba%e5%ba%8f%e9%97%ae%e9%a2%98/>)