06-自定义Hooks：四个典型的使用场景

你好，我是王沛。

我在开篇词就说过，要用好 React Hooks，很重要的一点，就是要能够从 Hooks 的角度去思考问题。要做到这一点其实也不难，就是在遇到一个功能开发的需求时，首先问自己一个问题：**这个功能中的哪些逻辑可以抽出来成为独立的 Hooks?**

这么问的目的，是为了让我们尽可能地把业务逻辑拆成独立的 Hooks ，这样有助于实现代码的模块化和解耦，同时也方便后面的维护。如果你基础篇的知识掌握得牢固的话，就会发现，这是因为 Hooks 有两个非常核心的优点：

- 一是方便进行逻辑复用；
- 二是帮助关注分离。

接下来我就通过一个案例，来带你认识什么是自定义Hooks，以及如何创建。然后，我们再通过其它3个典型案例，来看看自定义Hooks 具体有什么用，从而帮你掌握从 Hooks 角度去解决问题的思考方式。

如何创建自定义 Hooks?

自定义 Hooks 在形式上其实非常简单，就是**声明一个名字以 use 开头的函数**，比如 useCounter。这个函数在形式上和普通的 JavaScript 函数没有任何区别，你可以传递任意参数给这个 Hook，也可以返回任何值。

但是要注意，Hooks 和普通函数在语义上是有区别的，就在于**函数中有没有用到其它 Hooks。**

什么意思呢？就是说如果你创建了一个 useXXX 的函数，但是内部并没有用任何其它 Hooks，那么这个函数就不是一个 Hook，而只是一个普通的函数。但是如果用了其它 Hooks ，那么它就是一个 Hook。

举一个简单的例子，在第3讲中我们看到过一个简单计数器的实现，当时把业务逻辑都写在了函数组件内部，但其实是可以把业务逻辑提取出来成为一个 Hook。比如下面的代码：

```
import { useState, useCallback }from 'react';

function useCounter() {
  // 定义 count 这个 state 用于保存当前数值
  const [count, setCount] = useState(0);
  // 实现加 1 的操作
  const increment = useCallback(() => setCount(count + 1), [count]);
  // 实现减 1 的操作
  const decrement = useCallback(() => setCount(count - 1), [count]);
  // 重置计数器
  const reset = useCallback(() => setCount(0));

  // 将业务逻辑的操作 export 出去供调用者使用
  return { count, increment, decrement, reset };
}
```

有了这个 Hook，我们就可以在组件中使用它，比如下面的代码：

```
import React from 'react';

function Counter() {
  // 调用自定义 Hook
  const { count, increment, decrement, reset } = useCounter();

  // 渲染 UI
  return (
    <div>
      <button onClick={decrement}> - </button>
      <p>{count}</p>
      <button onClick={increment}> + </button>
      <button onClick={reset}> reset </button>
    </div>
  );
}
```

在这段代码中，我们把原来在函数组件中实现的逻辑提取了出来，成为一个单独的 Hook，**一方面能让这个逻辑得到重用，另外一方面也能让代码更加语义化，并且易于理解和维护。**

从这个例子，我们可以看到自定义 Hooks 的两个特点：

1. 名字一定是以 use 开头的函数，这样 React 才能够知道这个函数是一个 Hook；
2. 函数内部一定调用了其它的 Hooks，可以是内置的 Hooks，也可以是其它自定义 Hooks。这样才能够让组件刷新，或者去产生副作用。

当然，这只是一个非常简单的例子，实现了计数器业务逻辑的拆分和重用。不过通过这个例子，你也看到了创建自定义 Hook 是如此之简单，和过去的高阶组件设计模式相比，简直是天上地下的区别。也正因如此，Hooks 出现后就得到了迅速的普及。

那么，在日常开发的时候，除了解耦业务相关的逻辑，还有哪些场景需要去创建自定义 Hooks 呢？下面我就再给你介绍三个典型的业务场景。

封装通用逻辑：useAsync

在组件的开发过程中，有一些常用的通用逻辑。过去可能会因为逻辑重用比较繁琐，而经常在每个组件中去自己实现，造成维护的困难。但现在有了 Hooks，就可以将更多的通用逻辑通过 Hooks 的形式进行封装，方便被不同的组件重用。

比如说，在日常 UI 的开发中，有一个最常见的需求：**发起异步请求获取数据并显示在界面上**。在这个过程中，我们不仅要关心请求正确返回时，UI 会如何展现数据；还需要处理请求出错，以及关注 Loading 状态在 UI 上如何显示。

我们可以重新看下在第1讲中看到的异步请求的例子，从 Server 端获取用户列表，并显示在界面上：

```
import React from "react";
```

```
export default function UserList() {
  // 使用三个 state 分别保存用户列表，loading 状态和错误状态
  const [users, setUsers] = React.useState([]);
  const [loading, setLoading] = React.useState(false);
  const [error, setError] = React.useState(null);

  // 定义获取用户的回调函数
  const fetchUsers = async () => {
    setLoading(true);
    try {
      const res = await fetch("https://reqres.in/api/users/");
      const json = await res.json();
      // 请求成功后将用户数据放入 state
      setUsers(json.data);
    } catch (err) {
      // 请求失败将错误状态放入 state
      setError(err);
    }
    setLoading(false);
  };

  return (
    <div className="user-list">
      <button onClick={fetchUsers} disabled={loading}>
        {loading ? "Loading..." : "Show Users"}
      </button>
      {error &&
        <div style={{ color: "red" }}>Failed: {String(error)}</div>
      }
      <br />
      <ul>
        {users.length > 0 &&
          users.map((user) => {
            return <li key={user.id}>{user.first_name}</li>;
          })}
      </ul>
    </div>
  );
}
```

在这里，我们定义了 users、loading 和 error 三个状态。如果我们在异步请求的不同阶段去设置不同的状态，这样 UI 最终能够根据这些状态展现出来。在每个需要异步请求的组件中，其实都需要重复相同的逻辑。

事实上，在处理这类请求的时候，模式都是类似的，通常都会遵循下面步骤：

1. 创建 data，loading，error 这3个 state；
2. 请求发出后，设置 loading state 为 true；
3. 请求成功后，将返回的数据放到某个 state 中，并将 loading state 设为 false；
4. 请求失败后，设置 error state 为 true，并将 loading state 设为 false。

最后，基于 data、loading、error 这3个 state 的数据，UI 就可以正确地显示数据，或者 loading、error 这些反馈给用户了。

所以，通过创建一个自定义 Hook，可以很好地将这样的逻辑提取出来，成为一个可重用的模块。比如代码

可以这样实现：

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

const useAsync = (asyncFunction) => {
  // 设置三个异步逻辑相关的 state
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  // 定义一个 callback 用于执行异步逻辑
  const execute = useCallback(() => {
    // 请求开始时，设置 loading 为 true，清除已有数据和 error 状态
    setLoading(true);
    setData(null);
    setError(null);
    return asyncFunction()
      .then((response) => {
        // 请求成功时，将数据写进 state，设置 loading 为 false
        setData(response);
        setLoading(false);
      })
      .catch((error) => {
        // 请求失败时，设置 loading 为 false，并设置错误状态
        setError(error);
        setLoading(false);
      });
  }, [asyncFunction]);

  return { execute, loading, data, error };
};
```

那么有了这个 Hook，我们在组件中就只需要关心与业务逻辑相关的部分。比如代码可以简化成这样的形式：

```
import React from "react";
import useAsync from './useAsync';

export default function UserList() {
  // 通过 useAsync 这个函数，只需要提供异步逻辑的实现
  const {
    execute: fetchUsers,
    data: users,
    loading,
    error,
  } = useAsync(async () => {
    const res = await fetch("https://reqres.in/api/users/");
    const json = await res.json();
    return json.data;
  });

  return (
    // 根据状态渲染 UI...
  );
}
```

通过这个例子可以看到，我们**利用了 Hooks 能够管理 React 组件状态的能力，将一个组件中的某一部分状态独立出来，从而实现了通用逻辑的重用**。

不过在这里你可能会有一个疑问：这种类型的封装我写一个工具类不就可以了？为什么一定要通过 Hooks 进行封装呢？

答案很容易就能想到。因为在 Hooks 中，你可以管理当前组件的 state，从而将更多的逻辑写在可重用的 Hooks 中。但是要知道，在普通的工具类中是无法直接修改组件 state 的，那么也就无法在数据改变的时候触发组件的重新渲染。

监听浏览器状态：useScroll

虽然 React 组件基本上不需要关心太多的浏览器 API，但是有时候却是必须的：

- 界面需要根据在窗口大小变化重新布局；
- 在页面滚动时，需要根据滚动条位置，来决定是否显示一个“返回顶部”的按钮。

这都需要用到浏览器的 API 来监听这些状态的变化。那么我们就以滚动条位置的场景为例，来看看应该如何用 Hooks 优雅地监听浏览器状态。

正如 Hooks 的字面意思是“钩子”，它带来的一大好处就是：**可以让 React 的组件绑定在任何可能的数据源上。这样当数据源发生变化时，组件能够自动刷新**。把这个好处对应到滚动条位置这个场景就是：组件需要绑定到当前滚动条的位置数据上。

虽然这个逻辑在函数组件中可以直接实现，但是把这个逻辑实现为一个独立的Hooks，既可以达到逻辑重用，在语义上也更加清晰。这个和上面的 useAsync 的作用是非常类似的。

我们可以直接来看这个 Hooks 应该如何实现：

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

// 获取横向，纵向滚动条位置
const getPosition = () => {
  return {
    x: document.body.scrollLeft,
    y: document.body.scrollTop,
  }
}
const useScroll = () => {
  // 定一个 position 这个 state 保存滚动条位置
  const [position, setPosition] = useState(getPosition());
  const bodyNode = document.body;
  useEffect(() => {
 const handler = () => {
      setState(getPosition())
    };
    // 监听 scroll 事件，更新滚动条位置
    bodyNode.addEventListener('scroll', handler);
    return () => {
      // 组件销毁时，取消事件监听
      bodyNode.removeEventListener('scroll', handler);
    };
```

```
  }, []);

  return position;
};
```

有了这个 Hook，你就可以非常方便地监听当前浏览器窗口的滚动条位置了。比如下面的代码就展示了“返回顶部”这样一个功能的实现：

```
import React, { useCallback } from 'react';
import useScroll from './useScroll';

function ScrollTop() {
  const { y } = useScroll();

  const goTop = useCallback(() => {
    document.body.offsetTop = 0;
  });

  // 当滚动条位置纵向超过 300 时，显示返回顶部按钮
  if (y > 300) {
    return <button onClick={goTop}>返回顶部</button>
  }
  // 否则不 render 任何 UI
  return null;
}
```

通过这个例子，我们看到了如何将浏览器状态变成可被 React 组件绑定的数据源，从而在使用上更加便捷和直观。当然，除了窗口大小、滚动条位置这些状态，还有其它一些数据也可以这样操作，比如 cookies，localStorage, URL，等等。你都可以通过这样的方法来实现。

拆分复杂组件

函数组件虽然很容易上手，但是当某个组件功能越来越复杂的时候，我发现很多同学会出现一个问题，就是组件代码很容易变得特别长，比如超过500行，甚至1000行。这就变得非常难维护了。

设想当你接手某个项目，发现一个函数动辄就超过了500行，那会是什么感受？所以**“保持每个函数的短小”**这样通用的最佳实践，同样适用于函数组件。只有这样，才能让代码始终易于理解和维护。

那么现在的关键问题就是，怎么才能让函数组件不会太过冗长呢？做法很简单，就是**尽量将相关的逻辑做成独立的 Hooks，然后在函数组中使用这些 Hooks，通过参数传递和返回值让 Hooks 之间完成交互**。

这里可以注意一点，拆分逻辑的目的不一定是为了重用，而可以是仅仅为了业务逻辑的隔离。所以在这个场景下，我们不一定要把 Hooks 放到独立的文件中，而是可以和函数组件写在一个文件中。这么做的原因就在于，这些 Hooks 是和当前函数组件紧密相关的，所以写到一起，反而更容易阅读和理解。

为了让你对这一点有更直观的感受，我们来看一个例子。设想现在有这样一个需求：我们需要展示一个博客文章的列表，并且有一列要显示文章的分类。同时，我们还需要提供表格过滤功能，以便能够只显示某个分类的文章。

为了支持过滤功能，后端提供了两个 API：一个用于获取文章的列表，另一个用于获取所有的分类。这就需要我们在前端将文章列表返回的数据分类 ID 映射到分类的名字，以便显示在列表里。

这时候，如果按照直观的思路去实现，通常都会把逻辑都写在一个组件里，比如类似下面的代码：

```
function BlogList() {
  // 获取文章列表...
  // 获取分类列表...
  // 组合文章数据和分类数据...
  // 根据选择的分类过滤文章...

  // 渲染 UI ...
}
```

你可以想一下，如果你是在写一个其它的普通函数，会不会将其中一些逻辑写成单独的函数呢？相信答案是肯定的，因为这样做可以隔离业务逻辑，让代码更加清楚。

但我却发现很多同学在写函数组件时没有意识到 Hooks 就是普通的函数，所以通常不会这么去做隔离，而是习惯于一路写下来，这就会造成某个函数组件特别长。还是老生常谈的那句话，**改变这个状况的关键仍然在于开发思路的转变**。我们要真正**把 Hooks 就看成普通的函数，能隔离的尽量去做隔离**，从而让代码更加模块化，更易于理解和维护。

那么针对这样一个功能，我们甚至可以将其拆分成4个 Hooks，每一个 Hook 都尽量小，代码如下：

```
import React, { useMemo } from 'react';
import Table from './Table';
import useAsync from './useAsync';

const useArticles = () => {
  // 使用上面创建的 useAsync 获取文章列表
  const { data, loading, error } = useAsync(async () => {
    const res = await fetch('/api/articles');
    return await res.json();
  });
  // 返回语义化的数据结构
  return {
    articles: data,
    articlesLoading: loading,
    articlesError: error;
  };
};
const useCategories = () => {
   // 使用上面创建的 useAsync 获取分类列表
   const { data, loading, error } = useAsync(async () => {
    const res = await fetch('/api/categories');
    return await res.json();
  });
  // 返回语义化的数据结构
  return {
    categories: data,
    categoriesLoading: loading,
    categoriesError: error;
  }
```

```
};
const useCombinedArticles = (articles, categories) => {
  // 将文章数据和分类数据组合到一起
  return useMemo(() => {
    // 如果没有文章或者分类数据则返回 null
    if (!articles || !categories) return null;
    return articles.map(article => {
      return {
        ...article,
        category: categories.find(
          c => c.id === article.categoryId
        ),
      }
    })
  }, [articles, categories])
};
const useFilteredArticles = (articles, selectedCategory) => {
  // 实现按照分类过滤
  return useMemo(() => {
    if (!articles) return null;
    if (!selectedCategory) return articles;
    return articles.filter(article => {
      return article.categoryId === selectedCategory;
    });
  }, [articles, selectedCategory]);
};

function BlogList() {
  // 获取文章列表
  const { articles, articlesError } = useArticles();
  // 获取分类列表
  const { categories, categoriesError } = useCategories();
  // 组合数据
  const combined = useCombinedArticles(articles, categories);
  // 实现过滤
  const result = useFilteredArticles(combined, filter);

  if (articlesError || categoriesError) return 'Failed';
  // 如果没有结果，说明正在加载
  if (!result) return 'Loading...';

  return <Table data={result} />;
}
```

通过这样的方式，我们就把一个较为复杂的逻辑拆分成一个个独立的 Hook 了，不仅隔离了业务逻辑，也让代码在语义上更加明确。比如说有 useArticles、useCategories 这样与业务相关的名字，就非常易于理解。

虽然这个例子中抽取出来的 Hooks 都非常简单，甚至看上去没有必要。但是实际的开发场景一定是比这个复杂的，比如对于 API 返回的数据需要做一些数据的转换，进行数据的缓存，等等。那么这时就要避免把这些逻辑都放到一起，而是就要拆分到独立的 Hooks，以免产生过于复杂的组件。到时候你也就更能更体会到 Hooks 带给你的惊喜了。

小结

好了，这一讲我主要给你介绍了自定义 Hooks 的概念，以及典型的四个使用场景：

1. 抽取业务逻辑；

2. 封装通用逻辑；
3. 监听浏览器状态；
4. 拆分复杂组件。

其中，我通过四个案例来帮助你真正理解 Hooks ，并熟练掌握自定义 Hooks 的用法。应始终记得，要用 Hooks 的思路去解决问题，发挥 Hooks 的最大价值，就是要经常去思考哪些逻辑应该封装到一个独立的 Hook，保证每个 Hook 的短小精悍，从而让代码更加清晰，易于理解和维护。

思考题

在 useCounter 这个例子中，我们是固定让数字每次加一。假如要做一个改进，允许灵活配置点击加号时应该加几，比如说每次加10，那么应该如何实现？

欢迎在留言区分享你的思考和想法，我会和你交流讨论。我们下节课再见！

精选留言：

- 凡凡 2021-06-05 11:15:36

```
import { useState, useCallback } from 'react';

const useCounter = (step) => {
const [counter, setCounter] = useState(0);
const increment = useCallback(() => setCounter(counter + step), [counter, step]);
const decrement = useCallback(() => setCounter(counter - step), [counter, step]);
const reset = useCallback(() => setCounter(0), []);

return {counter, increment, decrement, reset};
}

export default useCounter;
```