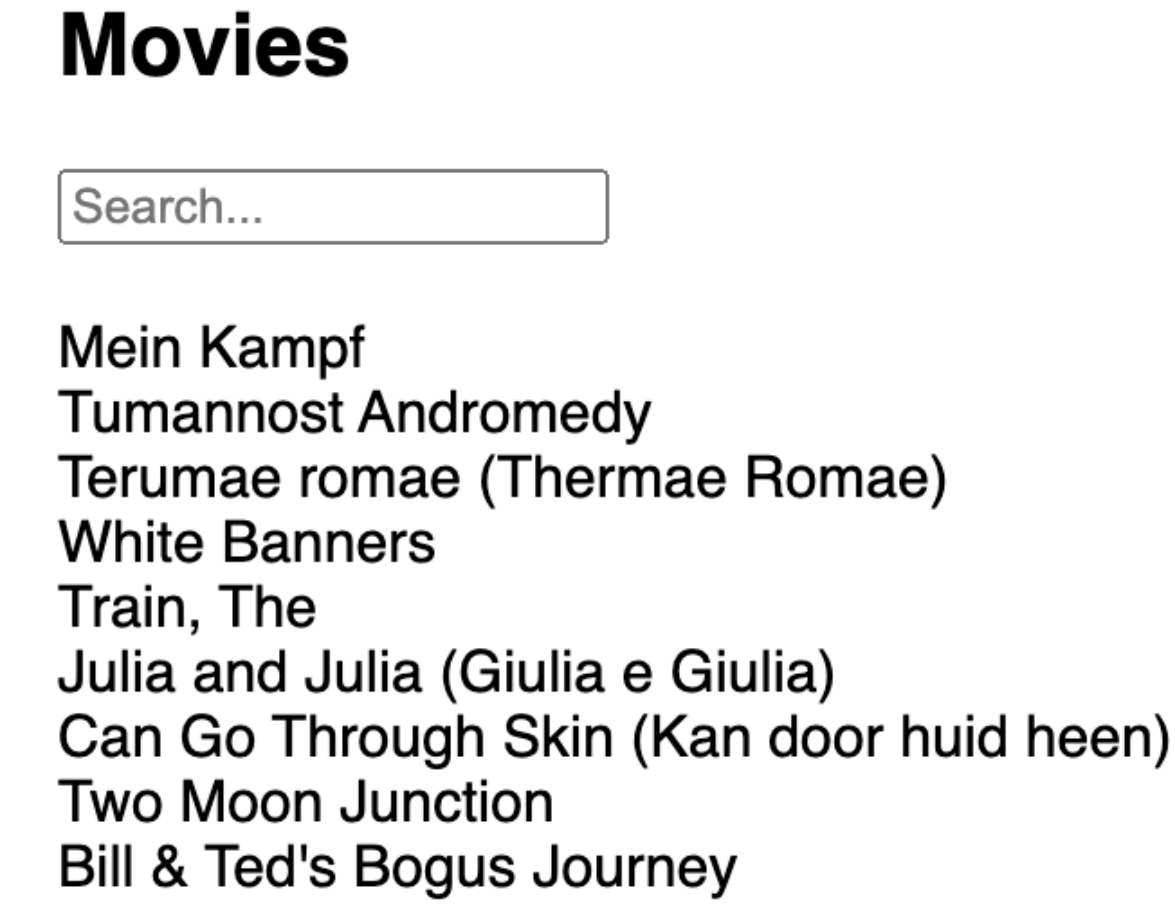
Hooks实战学习

Created and last modified by 王文磊 on Oct 13, 2022

一、状态一致性

1. 状态最小化原则

以一个简单的电影搜索列表为例



按照 React 的状态驱动 UI 的思想，第一步就是**要考虑整个功能有哪几个状态**。直观上来说，页面可能包含三个状态：

1. 电影列表的数据：可能来自某个 API 请求

2. 用户输入的关键字：来自用户的输入
3. 搜索的结果数据：来自原始数据结合关键字的过滤结果。

一般实现：

```
function FilterList({ data }) {
  // 设置关键字的 State
  const [searchKey, setSearchKey] = useState('');
  // 设置最终要展示的数据状态，并用原始数据作为初始值
  const [filtered, setFiltered] = useState(data);

  // 处理用户的搜索关键字
  const handleSearch = useCallback(evt => {
    setSearchKey(evt.target.value);
    setFiltered(data.filter(item => {
      return item.title.includes(evt.target.value)));
   }));
 }, [filtered])
  return (
    <div>
      <input value={searchKey} onChange={handleSearch} />
     {/* 根据 filtered 数据渲染 UI */}
    </div>
 );
}
```

其中隐藏的一致性的问题：展示的结果数据完全由原始数据和关键字决定，而现在却作为一个独立的状态去维护了。这意味着你始终要在原始数据、关键字和结果数据之间保证一致性。

问题：如果原始数据 data 属性变化了，最终的结果却没有使用新的数据。

在处理关键字变化的同时，再处理一下 data 属性变化的场景，这样不就可以保证三个状态的一致性了吗？比如再加上下面这段代码。

```
function FilterList({ data }) {
  // ...
  // 在 data 变化的时候，也重新生成最终数据
  useEffect(() => {
    setFiltered(data => {...})
 }, [data, searchKey])
  // ...
}
```

最终实现的代码略微复杂，根源在于没有遵循状态最小化的原则，而是设计了一个多余的状态：**过滤后的结果数据**，这个结果实际上完全由原始数据和过滤关键字决定，我们只用在需要的时候每次重新计算得出就可以了，再考虑到性能问题，使用useMemo实现：

```
import React, { useState, useMemo } from "react";

function FilterList({ data }) {
  const [searchKey, setSearchKey] = useState("");

  // 每当 searchKey 或者 data 变化的时候，重新计算最终结果
  const filtered = useMemo(() => {
    return data.filter((item) =>
   item.title.toLowerCase().includes(
      searchKey.toLowerCase()
    )
   );
 }, [searchKey, data]);

  return (
    <div className="08-filter-list">
      <h2>Movies</h2>
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={(evt) => setSearchKey(evt.target.value)}
      />
      <ul style={{ marginTop: 20 }}>
       {filtered.map((item) => (
```

```
            <li key={item.id}>{item.title}</li>
          ))}
        </ul>
      </div>
    );
  }
```

在实际开发的过程中，很多复杂场景之所以变得复杂，如果抽丝剥茧来看，你会发现它们都有**定义多余状态**现象的影子，而问题的根源就在于**它们没有遵循状态最小化的原则。**

总结：在保证State完整性的同时，也要保证它的最小化

2. 避免中间状态，确保唯一数据源原则

以1中例子为背景，需要实现一个分享的功能，即搜索后需要将关键字添加至url查询参数中：

```
// getQuery 函数用户获取 URL 的查询字符串
import getQuery from './getQuery';
// history 工具可以用于改变浏览器地址
import history from './history';

function SearchBox({ data }) {
  // 定义关键字这个状态，用 URL 上的查询参数作为初始值
  const [searchKey, setSearchKey] = useState(getQuery('key'));
  // 处理用户输入的关键字
  const handleSearchChange = useCallback(evt => {
    const key = evt.target.value;
    // 设置当前的查询关键状态
    setSearchKey(key);
    // 改变 URL 的查询参数
    history.push(`/movie-list?key=${key}`);
  })
  // ....
  return (
    <div className="08-search-box">
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={handleSearchChange}
      />
     {/* 其它渲染逻辑*/}
    </div>
  );
}
```

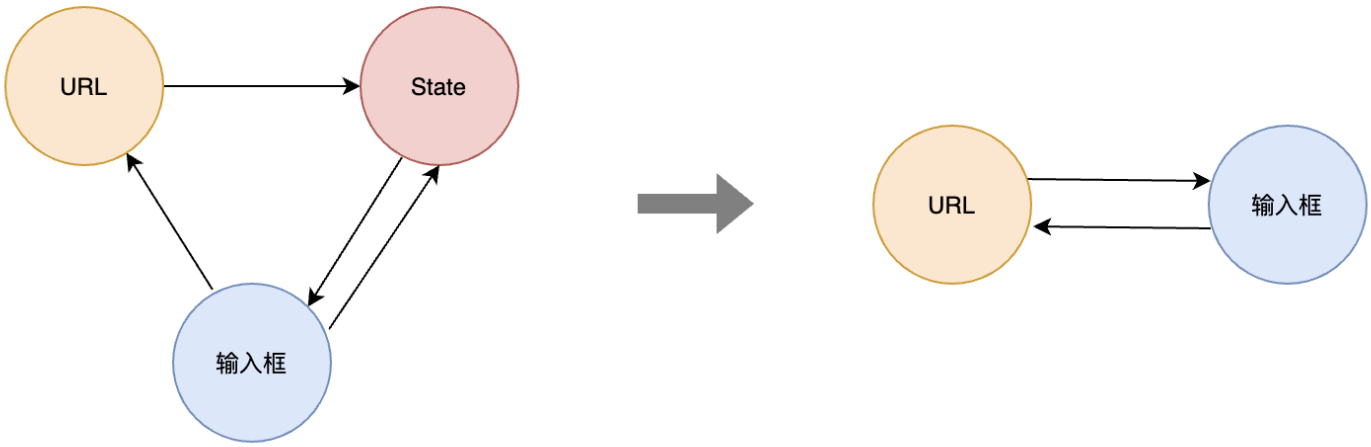
看上去似乎没有什么问题：保证了关键字状态，还有 URL 参数的一致性

问题：从 URL 参数到内部 State 的同步只有组件第一次渲染才会发生，而后面的同步则是由输入框的 onChange 事件保证的，一致性很容易被破坏

也就是说，如果 URL 不是由用户在组件内搜索栏去改变的，而是其它地方，比如说组件外的某个按钮去触发改变的，那么组件由于已经渲染过了，其实内部的 searchKey 这个 State 是不会被更新的，一致性就会被破坏。

要解决这个问题，一个比较容易想到的思路就是我们要有更加完善的机制，让在 URL 不管因为什么原因而发生变化的时候，都能同步查询参数到 searchKey 这个 State。

如果遵循唯一数据源这个原则，把URL上的查询关键字作为唯一数据源，逻辑就简单了。



通过对比可以看到，左边引入了一个多余的 State 作为关键字这个状态，而为了保证一致性，就需要很多复杂的同步逻辑，比如说以下几点：

- URL 变化时，同步查询关键字到 State
- State 变化时，同步查询关键字到输入框
- 用户在输入框输入的时候，同步关键字到 URL 和 State。

在去掉多余的 State 后，我们就只需要在输入框和 URL 的查询参数之间做一个同步。那么实现的代码可以简化如下：

```
import React, { useCallback, useMemo } from "react";
import { useSearchParam } from "react-use";

function SearchBox({ data }) {
  // 使用 useSearchParam 这个 Hook 用于监听查询参数变化
  const searchKey = useSearchParam("key") || "";
  const filtered = useMemo(() => {
    return data.filter((item) =>
      item.title.toLowerCase().includes(searchKey.toLowerCase())
    );
  }, [searchKey, data]);

  const handleSearch = useCallback((evt) => {
    // 当用户输入时，直接改变 URL
    window.history.pushState(
      {},
      "",
      `${window.location.pathname}?key=${evt.target.value}`
    );
  }, []);
  return (
    <div className="08-filter-list">
      <h2>Movies (Search key from URL)</h2>
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={handleSearch}
      />
      <ul style={{ marginTop: 20 }}>
       {filtered.map((item) => (
          <li key={item.id}>{item.title}</li>
       ))}
      </ul>
    </div>
  );
}
```

当用户输入参数的时候，我们是直接改变当前的 URL，而不是去改变一个内部的状态。所以当 URL 变化的时候，我们使用了 useSearchParam 这样一个第三方的 Hook 去绑定查询参数，并将其显示在输入框内，从而实现了输入框内容和查询关键字这个状态的同步。

总结：找到正确的数据来源并直接使用，避免中间状态

二、异步处理

1. 实现自己的API Client

基于 fetch / axios，创建项目级的 API Client，之后所有的请求都会通过这个Client发出去。实现这样一个Client之后，就可以对项目中所有需要连接服务端的请求做一些通用的配置和处理，比如 Token、URL、错误处理等等。

通常来说，会包括以下几个方面：

1. 一些通用的 Header。比如 Authorization Token。
2. 服务器地址的配置。前端在开发和运行时可能会连接不同的服务器，比如本地服务器或者测试服务器，此时这个 API Client 内部可以根据当前环境判断该连接哪个 URL。
3. 请求未认证的处理。比如如果 Token 过期了，需要有一个统一的地方进行处理，这时就会弹出对话框提示用户重新登录。

以axios 为例，提供一个示例实现：

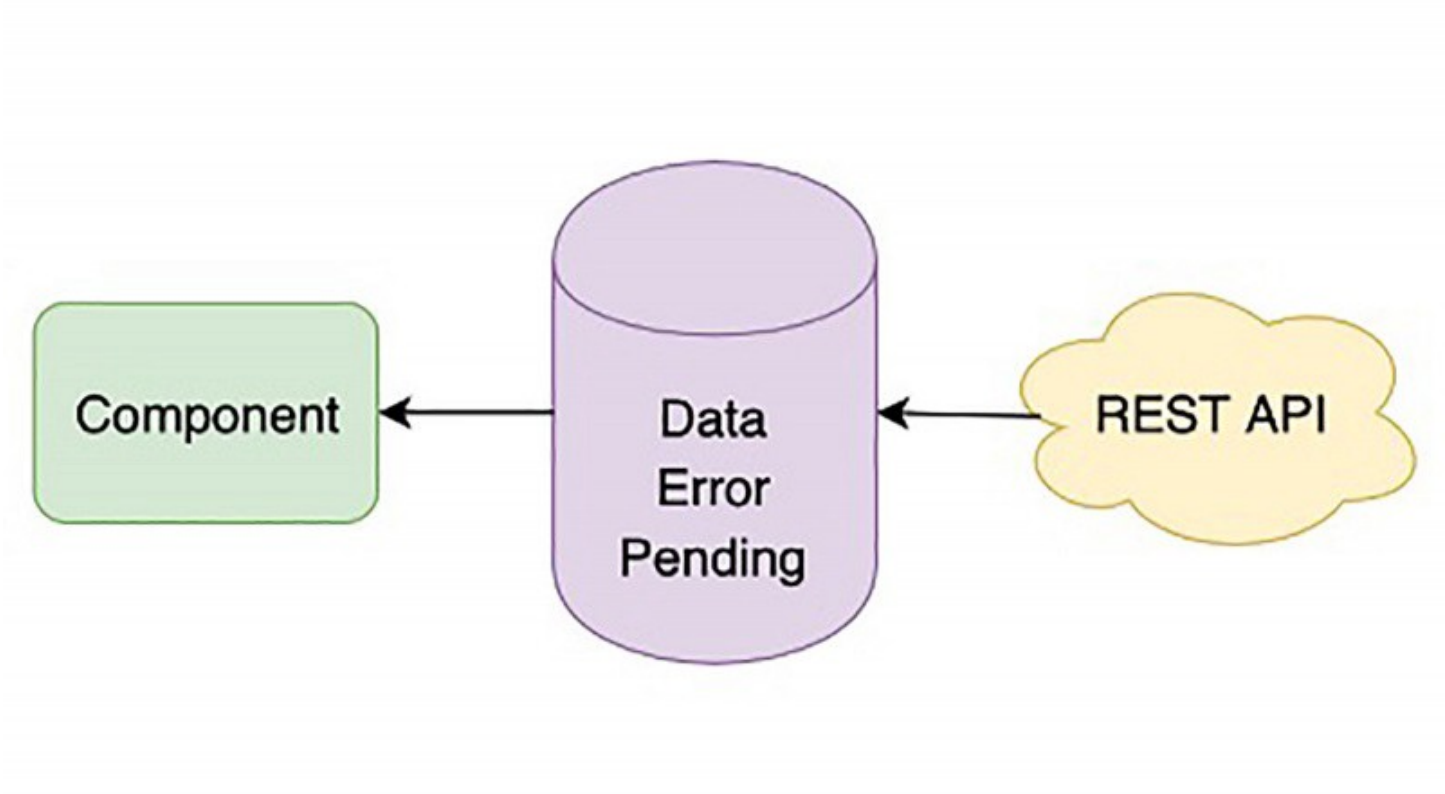
```
import axios from "axios";

// 定义相关的 endpoint
const endPoints = {
  test: "https://60b2643d62ab150017ae21de.mockapi.io/",
  prod: "https://prod.myapi.io/",
  staging: "https://staging.myapi.io/"
};

// 创建 axios 的实例
const instance = axios.create({
  // 实际项目中根据当前环境设置 baseURL
  baseURL: endPoints.test,
  timeout: 30000,
  // 为所有请求设置通用的 header
  headers: { Authorization: "Bear mytoken" }
});

// 通过 axios 定义拦截器预处理所有请求
instance.interceptors.response.use(
 (res) => {
    // 可以假如请求成功的逻辑，比如 log
    return res;
 },
 (err) => {
    if (err.response.status === 403) {
      // 统一处理未授权请求，跳转到登录界面
      document.location = '/login';
   }
    return Promise.reject(err);
 }
);

export default instance;
```

2. 使用Hooks思考异步请求：封装远程资源

从 Hooks 角度来说，我们可以认为一个 Get 请求就是一个远程数据源。那么把这个数据源封装成 Hooks 后，使用远程 API 将会非常方便。

对于一个 Get 类型的 API，我们完全可以将它看成一个远程的资源。只是和本地数据不同的地方在于，它有三个状态，分别是：

1. Data: 指的是请求成功后服务器返回的数据
2. Error: 请求失败的话，错误信息将放到 Error 状态里
3. Pending: 请求发出去，在返回之前会处于 Pending 状态。



有了这三个状态，我们就能够在 UI 上去显示 loading，error 或者获取成功的数据了。使用起来会非常方便。

比起在组件内部直接发请求，我们只是把代码换了个地方，也就是写到了 Hook 里面。下面是代码的实现：

```
import { useState, useEffect } from "react";
import apiClient from "./apiClient";

// 将获取文章的 API 封装成一个远程资源 Hook
const useArticle = (id) => {
  // 设置三个状态分别存储 data, error, loading
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  useEffect(() => {
    // 重新获取数据时重置三个状态
    setLoading(true);
    setData(null);
    setError(null);
    apiClient
     .get(`/posts/${id}`)
     .then((res) => {
        // 请求成功时设置返回数据到状态
        setLoading(false);
        setData(res.data);
     })
     .catch((err) => {
        // 请求失败时设置错误状态
        setLoading(false);
        setError(err);
     });
 }, [id]); // 当 id 变化时重新获取数据

  // 将三个状态作为 Hook 的返回值
  return {
    loading,
    error,
    data
 };
};
```

使用的时候，我们就可以把组件的表现层逻辑写得非常简洁：

```
import useArticle from "./useArticle";
```

```
const ArticleView = ({ id }) => {
  // 将 article 看成一个远程资源，有 data, loading, error 三个状态
  const { data, loading, error } = useArticle(id);
  if (error) return "Failed.";
  if (!data || loading) return "Loading...";
  return (
    <div className="exp-09-article-view">
      <h1>
       {id}. {data.title}
      </h1>
      <p>{data.content}</p>
    </div>
 );
};
```

通过这个hook的封装，将业务逻辑从React组件中抽离出来，组件只需要**把数据映射到JSX并显示出来**就可以了。

把每一个get请求做成一个hook，实现model和view的隔离

Q：为什么要给每个请求都定义一个 Hook 呢？我们直接提供一个通用的 Hook，比如 useRemoteData，然后把 API 地址传进去，难道不可以吗？

A：不是完全不可以，但这其实是为了**保证每个 Hook 自身足够简单**。一般来说，为了让服务器的返回数据满足 UI 上的展现要求，通常需要进一步处理。而这个对于每个请求的处理逻辑可能都不一样，通过一定的代码重复，能够避免产生太复杂的逻辑。同时呢，某个远程资源有可能是由多个请求组成的，那么 Hooks 中的逻辑就会不一样，因为要同时发出去多个请求，组成 UI 展现所需要的数据。所以，将每个 Get 请求都封装成一个 Hook ，也是为了让逻辑更清楚。

3. 多个 API 调用：如何处理并发或串行请求?

在实际的业务场景中，请求的控制往往更加复杂，以上述请求文章内容为例，如果还需要显示作者、作者头像，以及文章的评论列表，就需要发送**三个请求**：

1. 获取文章内容
2. 获取作者信息，包括名字和头像的地址
3. 获取文章的评论列表

这三个请求**同时包含了并发和串行的场景**：文章内容和评论列表是两个可以并发的请求，它们都通过 Article ID 来获取；用户的信息需要等文章内容返回，这样才能知道作者的 ID，从而根据用户的 ID 获取用户信息，这是一个串行的场景。

传统思路实现：

```
// 并发获取文章和评论列表
const [article, comments] = await Promise.all([
  fetchArticle(articleId),
  fetchComments(articleId)
]);
// 得到文章信息后，通过 userId 获取用户信息
const user = await fetchUser(article.userId);
```

但是 **React 函数组件是一个同步的函数**，没有办法直接使用 await 这样的同步方法，而是要**把请求通过副作用去触发**。

因此，我们需要考虑用状态去驱动UI（React的本质），这意味着我们可以**从状态变化的角度去组织异步调用**。函数组件的每一次 render，都提供给我们根据状态执行不同操作的机会。**利用这个机制，通过不同的状态组合，来实现异步请求的逻辑**。

那么刚才这个显示作者和评论列表的业务需求，主要的实现思路就包括下面这么四点：

1. 组件首次渲染，只有文章 ID 这个信息，产生两个副作用去获取文章内容和评论列表
2. 组件首次渲染，作者 ID 还不存在，因此不发送任何请求
3. 文章内容请求返回后，获得了作者 ID，然后发送请求获取用户信息
4. 展示用户信息。

可以看到，这里的任何一个副作用，也就是**异步请求，都是基于数据的状态去进行的**。

所以，在代码层面，我们首先需要对 useUser 这个 Hook 做一个改造，使得它在没有传入 ID 的情况下，就不发送请求。对比上面的 useArticle 这个 Hook，唯一的变化就是**在 useEffect 里加入了ID 是否存在的判断**：

```
import { useState, useEffect } from "react";
import apiClient from "./apiClient";

export default (id) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
```

```
    useEffect(() => {
      // 当 id 不存在，直接返回，不发送请求
      if (!id) return;
      setLoading(true);
      setData(null);
      setError(null);
      apiClient
       .get(`/users/${id}`)
       .then((res) => {
          setLoading(false);
          setData(res.data);
       })
       .catch((err) => {
          setLoading(false);
          setError(err);
       });
    }, [id]);
     return {
      loading,
      error,
      data
    };
  };
```

那么，在文章的展示页面，我们就可以使用下面的代码来实现：

```
  import { useState } from "react";
  import CommentList from "./CommentList";
  import useArticle from "./useArticle";
  import useUser from "./useUser";
  import useComments from "./useComments";

  const ArticleView = ({ id }) => {
    const { data: article, loading, error } = useArticle(id);
    const { data: comments } = useComments(id);
    const { data: user } = useUser(article?.userId);
    if (error) return "Failed.";
    if (!article || loading) return "Loading...";
    return (
      <div className="exp-09-article-view">
        <h1>
         {id}. {article.title}
        </h1>
       {user && (
          <div className="user-info">
            <img src={user.avatar} height="40px" alt="user" />
            <div>{user.name}</div>
            <div>{article.createdAt}</div>
          </div>
       )}
        <p>{article.content}</p>
        <CommentList data={comments || []} />
      </div>
    );
  };
```

这里，结合代码我们**再理一下其中并发和串行请求的思路**。

因为文章的 ID 已经传进来了，因此 useArticle 和 useComments 这两个 Hooks 会发出两个并发的请求，来分别获取信息。而 userUser 这个 Hook 则需要等 article 内容返回后，才能获得 userId 信息，所以这是一个串行的请求：需要等文章内容的请求完成之后才能发起。

三、函数组件设计模式：如何应对复杂条件渲染场景

> 所谓设计模式，就是**针对特定场景，提供一种公认的最佳实践**。

1. 容器模式：实现按条件执行Hooks

Hooks 必须在顶层作用域调用，而不能放在条件判断、循环等语句中，同时也不能在可能的 return 语句之后执行。换句话说，Hooks 必须按顺序被执行到。

这是因为 React需要在函数组件内部维护所用到的 Hooks 的状态

```
import { Modal } from "antd";
import useUser from "../09/useUser";

function UserInfoModal({ visible, userId, ...rest }) {
  // 当 visible 为 false 时，不渲染任何内容
  if (!visible) return null;
  // 这一行 Hook 在可能的 return 之后，会报错!
  const { data, loading, error } = useUser(userId);

  return (
    <Modal visible={visible} {...rest}>
     {/* 对话框的内容 */}
    </Modal>
 );
};
```

为了解决该规则带来的限制，需要用到一个间接的模式，即**容器模式**

具体做法是：**把条件判断的结果放到两个组件之中，确保真正 render UI 的组件收到的所有属性都是有值的。**

```
// 定义一个容器组件用于封装真正的 UserInfoModal
export default function UserInfoModalWrapper({
  visible,
  ...rest, // 使用 rest 获取除了 visible 之外的属性
}) {
  // 如果对话框不显示，则不 render 任何内容
  if (!visible) return null;
  // 否则真正执行对话框的组件逻辑
  return <UserInfoModal visible {...rest} />;
}
```

在容器模式中我们其实也可以看到，条件的隔离对象是多个子组件，这就意味着它通常用于一些比较大块逻辑的隔离。所以对于一些比较细节的控制，其实还有一种做法，就是把判断条件放到 Hooks 中去。

Hook不能放在条件语句中，但可以把条件语句自包含在Hook之中。

总体来说，通过这样一个容器模式，我们把原来需要条件运行的 Hooks 拆分成子组件，然后通过一个容器组件来进行实际的条件判断，从而渲染不同的组件，实现按条件渲染的目的。这在一些复杂的场景之下，也能达到拆分复杂度，让每个组件更加精简的目的。

2. render props 模式：重用 UI 逻辑

render props：把一个 render 函数作为属性传递给某个组件，由这个组件去执行这个函数从而 render 实际的内容。

Hooks的局限性：**只能用作数据逻辑的重用**

比如，我们需要显示一个列表，如果超过一定数量，则把多余的部分折叠起来，通过一个弹出框去显示。

对于这一类场景，**功能相同的部分是**：数据超过一定数量时，显示一个 “更多...”的文字；鼠标移上去则弹出一个框，用于显示其它的数据。

功能不同的部分是：每一个列表项如何渲染，是在使用的时候决定的。

```
import { Popover } from "antd";

function ListWithMore({ renderItem, data = [], max }) {
  const elements = data.map((item, index) => renderItem(item, index, data));
  const show = elements.slice(0, max);
  const hide = elements.slice(max);
  return (
    <span className="exp-10-list-with-more">
     {show}
     {hide.length > 0 && (
        <Popover content={<div style={{ maxWidth: 500 }}>{hide}</div>}>
          <span className="more-items-wrapper">
           and{" "}
            <span className="more-items-trigger"> {hide.length} more...</span>
          </span>
        </Popover>
```

```
        )}
      </span>
    );
```

可以看到，这个组件接收了三个参数，分别是：

1. renderItem：用于接收一个函数，由父组件决定如何渲染一个列表项
2. data：需要渲染的数据
3. max：最多显示几条数据。

```
// 这里用一个示例数据
import data from './data';

function ListWithMoreExample () => {
  return (
    <div className="exp-10-list-with-more">
      <h1>User Names</h1>
      <div className="user-names">
       Liked by:{" "}
        <ListWithMore
          renderItem={(user) => {
            return <span className="user-name">{user.name}</span>;
         }}
          data={data}
          max={3}
        />
      </div>
      <br />
      <br />
      <h1>User List</h1>
      <div className="user-list">
        <div className="user-list-row user-list-row-head">
          <span className="user-name-cell">Name</span>
          <span>City</span>
          <span>Job Title</span>
        </div>
        <ListWithMore
          renderItem={(user) => {
            return (
              <div className="user-list-row">
                <span className="user-name-cell">{user.name}</span>
                <span>{user.city}</span>
                <span>{user.job}</span>
              </div>
            );
         }}
          data={data}
          max={5}
        />
      </div>
    </div>
  );
};
```

可以看到，代码里使用了两个 ListWithMore 组件，通过 renderItem 这个属性，我们可以自主决定该如何渲染每一个列表项，从而把一部分 UI 逻辑抽象出来，形成一个可复用的逻辑，以简化不同场景的使用。

四、事件处理：创建自定义事件

1. React原生事件

回调函数是否需要 useCallback **和函数的复杂度没有必然关系**，而是和回调函数**绑定的目标组件**有关。

而对于原生的 DOM 节点，比如 button、input 等，我们是不用担心重新渲染的。所以呢，如果你的事件处理函数是传递给原生节点，那么不写 callback，也几乎不会有任何性能的影响。

但是**如果你使用的是自定义组件，或者一些 UI 框架的组件，那么回调函数还都应该用 useCallback 进行封装。**

React合成事件（Synthetic Events）

由于虚拟DOM的存在，导致在React中绑定一个事件到原生的 DOM 节点，事件也并不是绑定在对应的节点上，而是所有的事件都绑定在根节点上

- before React 17: 绑定在document
- after：绑定在App的根节点

原因如下：

1. 虚拟 DOM render 的时候， DOM 很可能还没有真实地 render 到页面上，所以无法绑定事件。
2. React 可以屏蔽底层事件的细节，避免浏览器的兼容性问题。同时呢，对于 React Native 这种不是通过浏览器 render 的运行时，也能提供一致的 API。

在浏览器的原生机制中，事件会从被触发的节点往父节点冒泡，然后沿着整个路径一直到根节点，所以根节点其实是可以收到所有的事件的。这也称之为浏览器事件的冒泡模型。

因此，无论事件在哪个节点被触发， React 都可以通过事件的 srcElement 这个属性，知道它是从哪个节点开始发出的，这样 React 就可以收集管理所有的事件，然后再以一致的 API 暴露出来。

2. 创建自定义事件

所谓的自定义事件，其实就是利用了属性传递回调函数给子组件，实现事件的触发。本质上，它和原生事件的机制是完全不一样的，原生事件是浏览器层面的事件，而自定义事件则是纯组件实现的一种机制。

看一个用 Hooks 封装键盘事件的例子：

```
import { useEffect, useState } from "react";

// 使用 document.body 作为默认的监听节点
const useKeyPress = (domNode = document.body) => {
  const [key, setKey] = useState(null);
  useEffect(() => {
    const handleKeyPress = (evt) => {
      setKey(evt.keyCode);
   };
    // 监听按键事件
    domNode.addEventListener("keypress", handleKeyPress);
    return () => {
      // 接触监听按键事件
      domNode.removeEventListener("keypress", handleKeyPress);
   };
 }, [domNode]);
  return key;
};
```

有了这个 Hook，我们在使用的时候就非常方便，无需做任何事件的绑定，而是只要把键盘按键看做是一个不断变化的数据源，这样，就可以去实时监听某个 DOM 节点上触发的键盘事件了。

例：

```
import useKeyPress from './useKeyPress';

function UseKeyPressExample() => {
  const key = useKeyPress();
  return (
    <div>
      <h1>UseKeyPress</h1>
      <label>Key pressed: {key || "N/A"}</label>
    </div>
 );
};
```

Like Be the first to like this