09-异步处理：如何向服务器端发送请求？

你好，我是王沛。这节课我们来学习如何在 React Hooks 中处理 Rest API 请求。

我遇到很多 React 的初学者，刚把Hello World跑起来，问的第一个问题就是：我该怎么发请求拿数据呢？

可见，这是最为常见也是最为重要的一个需求。毕竟90%以上的前端 App 都是和服务器端打交道，然后通过各种 API 完成各种功能。

虽然发请求拿数据有很多种做法，但基本上都会遵循一定的规律，而这正是咱们这节课要介绍的内容。

其实在第一讲，我们就已经看到了异步请求的一个实例，当时我们在一个组件内直接发起了一个请求，并处理了返回结果。但这个简单的例子只是演示了组件中发请求的基本流程，在实际的项目中我们很少会直接这么写，因为**还需要考虑更多的逻辑**。

比如说，如何给所有请求都带上一个 Token 供服务器端验证？如何发起并发请求？等等。此外，咱们整个课程都是从 Hooks 角度去思考问题，那么 Hooks 又能给异步逻辑处理带来怎样的优化？这些都是接下来要具体介绍的内容。

实现自己的 API Client

无论大小项目，在开始实现第一个请求的时候，通常我们要做的第一件事应该都是**创建一个自己的 API Client，之后所有的请求都会通过这个 Client 发出去**。而不是上来就用 fetch 或者是 axios 去直接发起请求，因为那会造成大量的重复代码。

实现这样一个 Client 之后，你就有了一个统一的地方，去对你需要连接的服务端做一些通用的配置和处理，比如 Token、URL、错误处理等等。

这个步骤呢，虽然跟 React 和 Hooks 没有直接的关系，但是我经常看到很多同学是在多个组件内分别实现了异步请求处理，写了很多重复代码之后，才开始考虑把这个功能提取出来，增加了额外工作量。所以这里我就先简单介绍下，实现这样一个 API Client 需要考虑哪些因素。

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。

从我的经验来看，我更推荐把 axios 作为基础来实现这个功能。原因就在于，axios 比起 fetch，提供了更为方便，也更加语义化的 API，比如请求拦截。此外，还很容易创建多个实例，让代码逻辑更简洁。所以我就以 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" }
});

// 听过 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;
```

这样我们就定义了一个简单的 API Client，之后所有的请求都可以通过 Client 连接到指定的服务器，从而不再需要单独设置 Header，或者处理未授权的请求了。当然，虽然例子中用的是 axios，但如果你更倾向于原生的 fetch API，那么使用 fetch 也是完全可以的。这两种方式没有绝对的优劣，你可以按照自己的喜好进行选择。

有了这个 Client，那接下来的例子中，我们都会使用这个 Client ，来连接指定的 server。

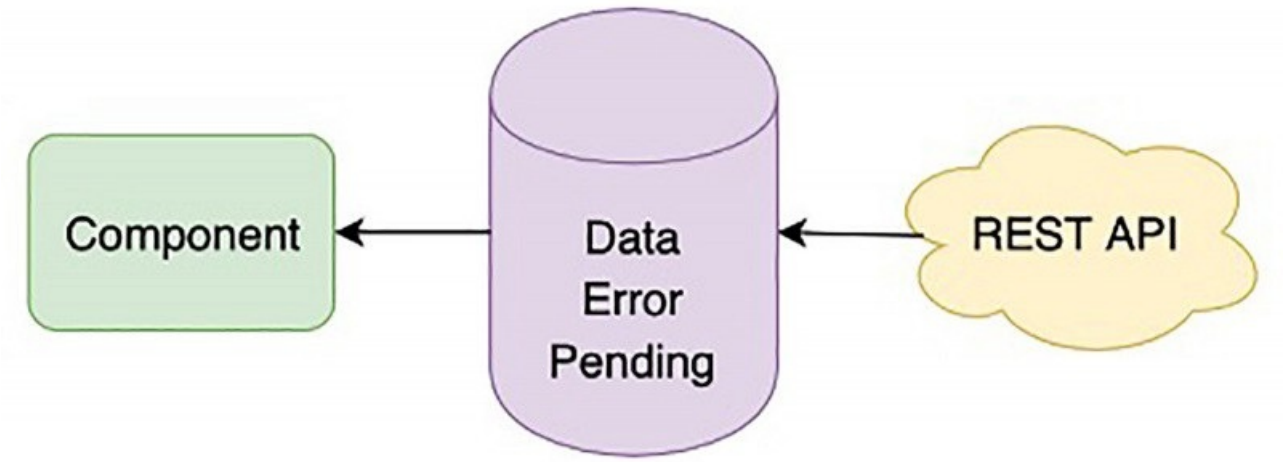
使用 Hooks 思考异步请求：封装远程资源

在第2讲我曾经介绍过 Hooks 的一个本质特性，那就是万物皆可钩，意思是说我们可以把任何一个数据源变成 React 组件中可以绑定的一个数据源。

这个特性对于远程的 Get 类型的请求非常有用。因为 Get 请求，通常就是用来获取数据的。所以从 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。

下面这张图可以比较直观地描述这种模式。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。

要实现这样一个 Hook 其实也并不难，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
  };
};
```

那么要显示一篇文章的时候，你的脑子里就不再是一个具体的 API 调用，而可以把它看作一个数据，只不过是个远程数据，于是很自然就有加载状态或者错误状态这些数据了。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>
  );
};
```

可以看到，有了这样一个 Hook，React 的函数组件几乎不需要有任何业务的逻辑，而只是**把数据映射到 JSX并显示出来**就可以了，在使用的时候非常方便。

事实上，在我们的项目中，可以把每一个 Get 请求都做成这样一个 Hook。数据请求和处理逻辑都放到 Hooks 中，从而实现 Model 和 View 的隔离，不仅代码更加模块化，而且更易于测试和维护。

这里有同学可能会问，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？

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。

当然，这个模式**仅适用于Get 请求的逻辑**，对于其它类型，我在第6讲中有提到过。当时举的例子是 useAsync 这样一个自定义的 Hook，同样也是用 Hook 的思想，把请求的不同状态封装成了一个数据源供组件使用。

多个 API 调用：如何处理并发或串行请求？

在刚才讲的文章显示的例子中，我们只是简单显示了文章的内容，要知道，实际场景肯定比这个更复杂。比如，还需要显示作者、作者头像，以及文章的评论列表。那么，作为一个完整的页面，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。

如果用传统的思路去实现，可能会写下这样一段代码，或者类似的代码：

```
// 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);
```

但是我们知道，**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**。因此如果按照上面这种传统的思维，是很难把逻辑理顺的。

这时候我们就要回到 React 的本质，那就是状态驱动 UI。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，其实都提供了我们根据状态变化执行不同操作的机会，我们思考的路径，就是**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**。只有当文章的数据返回之后，我们才能得到作者 ID，从而再发送另外一个请求来获取作者信息。这样基于一个数据状态的变化，我们就实现了串行发送请求这个功能。

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：

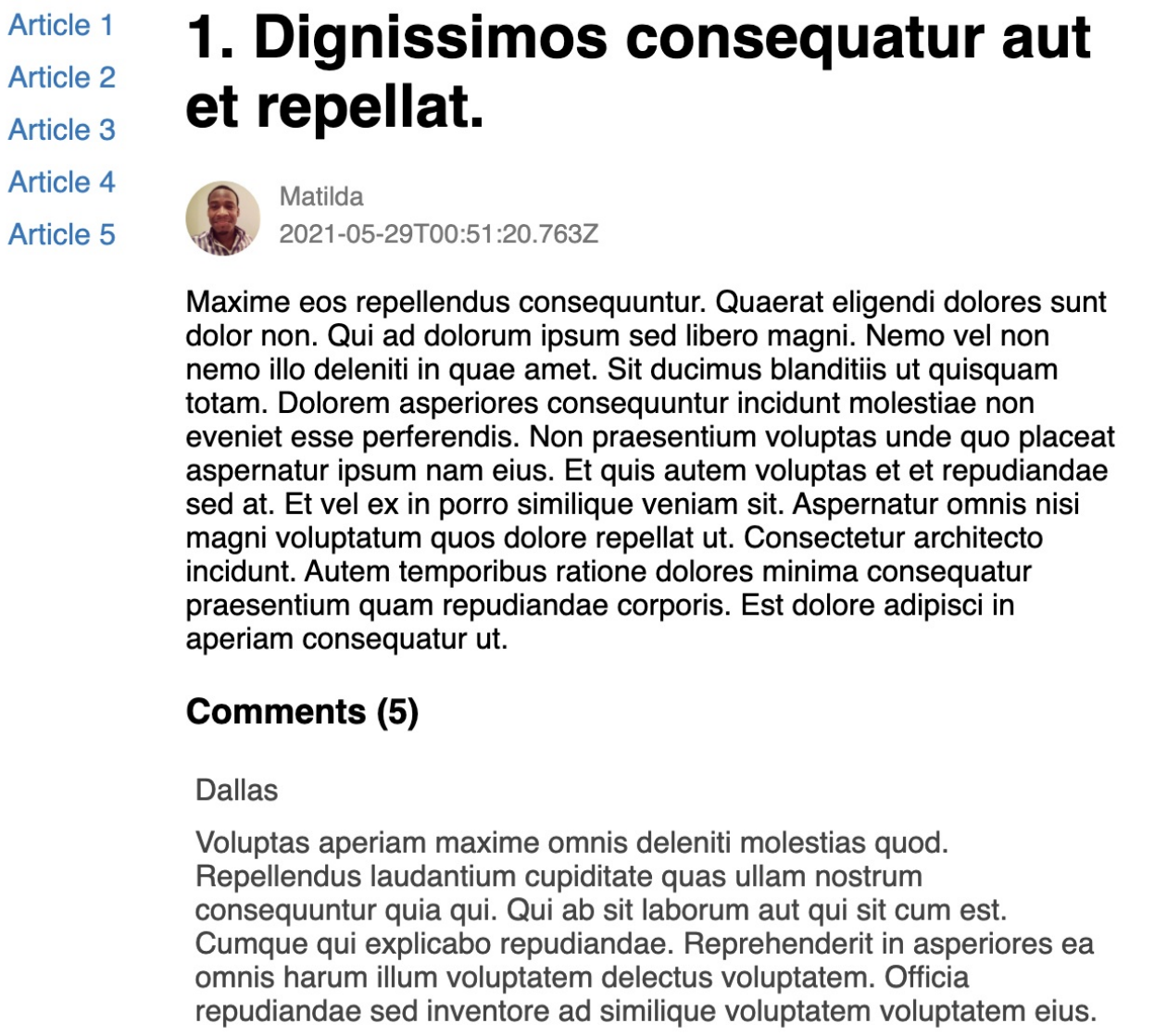
```
import { useState } from "react";
import CommentList from "./CommentList";
import useArticle from "./useArticle";
import useUser from "./useUser";
import useComments from "./useComments";

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。

那么，最终完整的页面显示的效果如下图所示：



实际的运行效果你也可以通过文末的代码链接来查看。

小结

在这一讲，我们看到了怎么从 Hooks 的角度去组织异步请求。

首先，我们**要定义一个自己的 API Client**，封装整个应用中异步请求的一些通过设置，以及统一处理，方便

在 Hooks 中使用。

然后，我们需要充分利用 Hooks 能让数据源变得可绑定的能力，**让一个远程 API 对应的数据成为一个语义化的数据源**，既可以把业务逻辑和UI 展现很好地分开，也有利于测试和维护。

最后呢，我们学习了针对多请求的处理，怎么**利用状态的组合变化来实现并发和串行请求**。

学到这里，你会发现，这节课的内容可能和传统的思考方式有较大区别，所以你一定要结合课程代码，仔细思考和体会，才能将这种思考方式熟记于心，并熟练运用到实际的开发当中。

课程中的例子地址是：https://codesandbox.io/s/react-hooks-course-20vzg。你可以在线查看代码和运行效果。

思考题

1. 在课程的例子中，每次调用 useArticle 这个 Hook 的时候都会触发副作用去获取数据。但是有时候，我们希望在有些组件自动获取，但有的组件中需要点击某个按钮才去获取数据，那么你会如何设计这个 Hook？（可能这道题有一点难度。）
2. 课程中的 Hook 都是使用的 useState 保存了状态数据，也就意味着状态的范围限定在组件内部，组件销毁后，数据就没了。那么如果希望数据直接缓存到全局状态，应该如何做呢？

欢迎把你的思考和想法分享在留言区，我会把其中不错的回答进行置顶，供同学们交流讨论。期待你的分享！

精选留言：

- L 2021-06-15 17:33:20
  1. hooks提供一个函数出来用来主动调用
  2. redux或者context等内容，对接口调用进行一次改造，先判断是否有缓存，再来决定是否调用接口 [2赞]

- Free fall 2021-06-15 21:34:13
  ```
  const defaultResult = {
  error: null,
  data: null,
  loading: false,
  }
  const useArticle = ({ id, isAutoFetch = true }) => {
  const [result, setResult] = useState(defaultResult)

  const fetch = (id) => {
  apiClient
  .get(`/posts/${id}`)
  .then((res) => {
  setResult({ ...defaultResult, loading: false, data: res.data })
  })
  .catch((err) => {
  setResult({ ...defaultResult, loading: false, error: err })
  })
  ```

```
}

useEffect(() => {
if (!id) return
if (!isAutoFetch) return

setResult({ ...defaultResult, loading: true })

fetch(id)
}, [id, isAutoFetch])

return isAutoFetch ? result : { ...result, fetch }
} [1赞]
```

- Isaac 2021-06-15 10:51:18
  思考题：
  1. useArticle Hook 可以提供一个参数，用来标记本地调用是否默认触发副作用去获取数据；对于点击按钮才触发请求的功能，可以在 Hook 中将获取数据的方法 retrn 出去，供外部自由调用。
  2. 可以借助 redux，配合 useContext 等 api ，将状态数据存储至全局中。 [1赞]

- 琪琪 2021-06-17 14:04:31
  老师，请问需要封装一个需要发送post请求的自定义hooks应该如何封装呢？

- 小畅叙 2021-06-17 00:42:52
  老师，对于 loading 和错误处理，我们项目是在全局处理的，且 loading 是通过 redux 管理的，想咨询一下，跟您的方法相比，应该用哪种方法呢？

- 王雪 2021-06-15 23:31:14
  问题1. 是否可以例如class组件reveiverProps的思想，在apiHook里面接受一个值假设是id，外层调用的时候控制这个id的变化，如果btn点击想要触发，可以将id设置为时间戳每次点击传的id不一样就驱动副作用了
  问题2. 可以使用useRef这个hook

- Free fall 2021-06-15 21:16:03

```
useEffect(() => {
if (!id) return
if (!isAutoFetch) return

setLoading(true)
setError(null)
setData(null)

apiClient
.get(`/posts/${id}`)
.then((res) => {
setLoading(false)
setData(res.data)
})
.catch((err) => {
setLoading(false)
```

```
setError(err)
})
}, [id, isAutoFetch])

return {
loading,
error,
data,
setIsAutoFetch,
}
}
```

- Geek_71adef 2021-06-15 20:04:36
  1,点击事件可以考虑到中间件，返回一个函数
  2 全局可以用redux或者usecontext 成为全局变量
  请问老师 userReducer是同步吧