你可能需要的React开发小技巧!

1. JSX 简写

如何将 true 值传递给给定的 props?

在下面的示例中，使用 prop `showTitle` 在导航栏组件中显示应用的标题：

```javascript 复制代码
export default function App() {
  return (
    <main>
    <Navbar showTitle={true} />
  </main>
);
}

function Navbar({ showTitle }) {
  return (
    <div>
      {showTitle && <h1>标题</h1>}
    </div>
  )
}
```

这里将 `showTitle` 显式设置为布尔值 `true`，其实这是没必要的，因为组件上提供的任何 `prop` 都具有默认值 `true`。因此只需要在调用组件时传递一个 `showTitle` 即可：

```javascript 复制代码
export default function App() {
  return (
    <main>
      <Navbar showTitle />
    </main>
  );
}

function Navbar({ showTitle }) {
  return (
    <div>
      {showTitle && <h1>标题</h1>}
    </div>
```

```
  )
}
```

另外，当需要传递一个字符串作为 `props` 时，无需使用花括号 `{}` 包裹，可以通过双引号包裹字符串内容并传递即可：

```javascript 复制代码
export default function App() {
  return (
    <main>
      <Navbar title="标题" />
    </main>
  );
}

function Navbar({ title }) {
  return (
    <div>
      <h1>{title}</h1>
    </div>
  )
}
```

2. 将不相关代码移动到单独的组件中

编写更简洁的 React 代码的最简单和最重要的方法就是善于将代码抽象为单独的 React 组件。

下面来看一个例子，应用中最上面会有一个导航栏，并遍历 `posts` 中的数据将文章标题渲染出来：

```javascript 复制代码
export default function App() {
  const posts = [
    {
      id: 1,
      title: "标题1"
    },
    {
      id: 2,
      title: "标题2"
    }
  ];

  return (
```

```
    <main>
      <Navbar title="大标题" />
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            {post.title}
          </li>
        ))}
      </ul>
    </main>
  );
}

function Navbar({ title }) {
  return (
    <div>
      <h1>{title}</h1>
    </div>
  );
}
```

那我们怎样才能让这段代码更加清洁呢？我们可以抽象循环中的代码（文章标题），将它们抽离到一个单独的组件中，称之为 `FeaturedPosts`。抽离后的代码如下：

javascript 复制代码

```
export default function App() {
 return (
    <main>
      <Navbar title="大标题" />
      <FeaturedPosts />
    </main>
  );
}

function Navbar({ title }) {
  return (
    <div>
      <h1>{title}</h1>
    </div>
  );
}

function FeaturedPosts() {
  const posts = [
    {
      id: 1,
      title: "标题1"
    },
```

```
    {
      id: 2,
      title: "标题2"
    }
  ];

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

如你所见，在 App 组件中，通过其中的组件名称： `Navbar` 和 `FeaturedPosts`，就可以快速地看到应用的作用。

3. 为每个组件创建单独的文件

在上面的例子中，我们将三个组件在一个文件中实现。如果组件逻辑较少，这些写还没啥问题，但是如果组件逻辑较为复杂，那这样写代码的可读性就很差了。为了使应用文件更具可读性，可以将每个组件放入一个单独的文件中。

这可以帮助我们在应用中分离关注点。 这意味着每个文件只负责一个组件，如果想在应用中重用它，就不会混淆组件的来源：

javascript 复制代码

```
// src/App.js
import Navbar from './components/Navbar.js';
import FeaturedPosts from './components/FeaturedPosts.js';

export default function App() {
  return (
    <main>
      <Navbar title="大标题" />
      <FeaturedPosts />
    </main>
  );
}
```

javascript 复制代码

```
// src/components/Navbar.js
export default function Navbar({ title }) {
```

```
  return (
    <div>
      <h1>{title}</h1>
    </div>
  );
}
```

javascript 复制代码

```
// src/components/FeaturedPosts.js
export default function FeaturedPosts() {
  const posts = [
    {
      id: 1,
      title: "标题1"
    },
    {
      id: 2,
      title: "标题2"
    }
  ];

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

此外，通过将每个单独的组件包含在其自己的文件中，可以避免一个文件变得过于臃肿。

4. 将共享函数移动到 React hook 中

在 FeaturedPosts 组件，假设想要从 API 获取文章数据，而不是使用假数据。可以使用 fetch API 来实现：

javascript 复制代码

```
import React from 'react';

export default function FeaturedPosts() {
  const [posts, setPosts] = React.useState([]);

  React.useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
```

```
      .then(res => res.json())
      .then(data => setPosts(data));
  }, []);

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

但是，如果想在多个组件执行这个数据请求怎么办?

假设除了 FeaturedPosts 组件之外，还又一个名为 Posts 的组件，其中包含相同的数据。 我们必须复制用于获取数据的逻辑并将其粘贴到该组件中。为了避免重复编写代码，可以定义一个新的 React hook，可以称之为 useFetchPosts：

javascript 复制代码

```
import React from 'react';

export default function useFetchPosts() {
  const [posts, setPosts] = React.useState([]);

  React.useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(res => res.json())
      .then(data => setPosts(data));
  }, []);

  return posts;
}
```

这样就可以在任何组件中重用它，包括 FeaturedPosts 组件：

javascript 复制代码

```
import useFetchPosts from '../hooks/useFetchPosts.js';

export default function FeaturedPosts() {
  const posts = useFetchPosts()

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
```

```
    </ul>
  );
}
```

5. 从 JSX 中删除JS

另一种简化组件的方式就是从 JSX 中删除尽可能多的 JavaScript。来看下面的例子：

```javascript 复制代码
import useFetchPosts from '../hooks/useFetchPosts.js';

export default function FeaturedPosts() {
  const posts = useFetchPosts()

  return (
    <ul>
      {posts.map((post) => (
        <li onClick={event => {
          console.log(event.target, 'clicked!');
        }} key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

这里我们尝试处理文章的点击事件，可以看到我们的 JSX 变得更加难以阅读。 鉴于函数是作为内联函数包含的，它掩盖了这个组件的用途，以及它的相关函数。

那该如何来解决这个问题？可以将包含 onClick 的内联函数提取到一个单独的处理函数中，给它一个名称 handlePostClick。这样 JSX 的可读性就变高了：

```javascript 复制代码
import useFetchPosts from '../hooks/useFetchPosts.js';

export default function FeaturedPosts() {
  const posts = useFetchPosts()

  function handlePostClick(event) {
    console.log(event.target, 'clicked!');
  }

  return (
    <ul>
      {posts.map((post) => (
        <li onClick={handlePostClick} key={post.id}>{post.title}</li>
```

```
      ))}
    </ul>
  );
}
```

6. 格式化内联样式

在 JSX 中编写过多的内联样式就会让代码更难阅读并且变得臃肿：

javascript 复制代码

```javascript
export default function App() {
  return (
    <main style={{ textAlign: 'center' }}>
      <Navbar title="大标题" />
    </main>
  );
}

function Navbar({ title }) {
  return (
    <div style={{ marginTop: '20px' }}>
      <h1 style={{ fontWeight: 'bold' }}>{title}</h1>
    </div>
  )
}
```

我们要尽可能地将内联样式移动到 CSS 样式表中。或者将它们组织成对象：

javascript 复制代码

```javascript
export default function App() {
  const styles = {
    main: { textAlign: "center" }
  };

  return (
    <main style={styles.main}>
      <Navbar title="大标题" />
    </main>
  );
}

function Navbar({ title }) {
  const styles = {
    div: { marginTop: "20px" },
    h1: { fontWeight: "bold" }
  };
```

```
  return (
    <div style={styles.div}>
      <h1 style={styles.h1}>{title}</h1>
    </div>
  );
}
```

一般情况下，最好将这些样式写在CSS样式表中，如果样式需要动态生成，可以将其定义在一个对象中。

7. 使用可选链运算符

在 JavaScript 中，我们需要首先确保对象存在，然后才能访问它的属性。如果对象的值为 undefined 或者 null，则会导致类型错误。

下面来看一个例子，用户可以在其中编辑他们发布的帖子。只有当isPostAuthor 为 true 时，也就是经过身份验证的用户的 id 与帖子作者的 id 相同时，才会显示该EditButton组件。

javascript 复制代码

```
export default function EditButton({ post }) {
  const user = useAuthUser();
  const isPostAuthor = post.author.userId !== user && user.userId;

  return isPostAuthor ? <EditButton /> : null;
}
```

这段代码的问题是 user 可能是 undefined. 这就是为什么我们必须在尝试获取 userId 属性之前使用 && 运算符来确保 user 是一个对象。如果我要访问一个对象中的另一个对象，就不得不再包含一个 && 条件。 这会导致代码变得复杂、难以理解。

JavaScript 可选链运算符（?.）允许我们在访问属性之前检查对象是否存在。用它来简化上面的代码：

javascript 复制代码

```
export default function EditButton({ post }) {
  const user = useAuthUser();
  const isPostAuthor = post.author.userId !== user?.userId;

  return isPostAuthor ? <EditButton /> : null;
}
```

这样将防止任何类型错误，并允许我们编写更清晰的条件逻辑。

8. 带括号的隐式返回

在 React 应用中可以使用 function 关键字的函数声明语法编写组件，也可以使用设置为变量的箭头函数。使用 function 关键字的组件必须在返回任何 JSX 之前使用 return 关键字。

```javascript
export default function App() {
  return (
    <Layout>
      <Routes />
    </Layout>
  );
}
```

通过将返回的代码包裹在一组括号中，可以通过隐式返回（不使用 return 关键字）从函数返回多行 JavaScript 代码。

对于使用箭头函数的组件，不需要包含 return 关键字，可以只返回带有一组括号的 JSX。

```javascript
const App = () => (
  <Layout>
    <Routes />
  </Layout>
);

export default App;
```

此外，当使用 `.map()` 迭代元素列表时，还可以跳过 `return` 关键字并仅在内部函数的主体中使用一组括号返回 JSX。

```javascript
function PostList() {
  const posts = usePostData();

  return posts.map(post => (
    <PostListItem key={post.id} post={post} />
  ))
}
```

9. 使用空值合并运算符

在 JavaScript 中，如果某个值是假值（如 null、undefined、0、''、NaN），可以使用 || 条件来提供一个备用值。

例如，在产品页面组件需要显示给定产品的价格，可以使用 || 来有条件地显示价格或显示文本“产品不可用”。

```javascript
export default function ProductPage({ product }) {
  return (
     <>
       <ProductDetails />
       <span>
          {product.price || "产品不可用"}
       </span>
     </>
  );
}
```

现有的代码存在一个问题，如果商品的价格为0，也不会显示产品的价格而显示"产品不可用"。如果左侧为 `null` 或者 `undefined`，而不是其他假值，就需要一个更精确的运算符来仅返回表达式的右侧。

这时就可以使用空值合并运算符，当左侧操作数为null或者 undefined 时，将返回右侧操作数。 否则它将返回其左侧操作数：

```javascript
null ?? 'callback';
// "callback"

0 ?? 42;
// 0
```

可以使用空值合并运算符来修复上面代码中的问题：

```javascript
export default function ProductPage({ product }) {
  return (
     <>
       <ProductDetails />
       <span>{product.price ?? "产品不可用"}
     </>
```

```
  );
}
```

10. 使用三元表达式

在 React 组件中编写条件时，三元表达式是必不可少的，经常用于显示或隐藏组件和元素。

当然，我们用可以使用三元表达式和模板字符串来给 React 元素动态添加或删除类名。

```javascript 复制代码
export default function App() {
  const { isDarkMode } = useDarkMode();

  return (
    <main className={`body ${isDarkMode ? "body-dark" : "body-light"}`}>
      <Routes />
    </main>
  );
}
```

这种条件逻辑也可以应用于任何 props：

```javascript 复制代码
export default function App() {
  const { isMobile } = useDeviceDetect();

  return (
    <Layout height={isMobile ? '100vh' : '80vh'}>
      <Routes />
    </Layout>
  );
}
```