react知识点二

react知识点二

1.懒加载

React.lazy React.lazy 函数能让你像渲染常规组件一样处理动态引入（的组件）。

jsx 复制代码

```jsx
/// 使用之前：
import OtherComponent from "./OtherComponent";

/// 使用之后：
const OtherComponent = React.lazy(() => import("./OtherComponent"));

/// 完整的示例
/// fallback 属性接受任何在组件加载过程中你想展示的 React 元素。你可以将 Suspense 组件置于懒加载组件之上的
import React, { Suspense } from "react";
const OtherComponent = React.lazy(() => import("./OtherComponent"));
function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

避免兜底

jsx 复制代码

```jsx
import React, { Suspense } from "react";
import Tabs from "./Tabs";
import Glimmer from "./Glimmer";

const Comments = React.lazy(() => import("./Comments"));
const Photos = React.lazy(() => import("./Photos"));

function MyComponent() {
```

```jsx
  const [tab, setTab] = React.useState("photos");

  function handleTabSelect(tab) {
    setTab(tab);
  }

  return (
    <div>
      <Tabs onTabSelect={handleTabSelect} />
      <Suspense fallback={<Glimmer />}>
        {tab === "photos" ? <Photos /> : <Comments />}
      </Suspense>
    </div>
  );
}
```

在这个示例中，如果标签从 'photos' 切换为 'comments'，但 Comments 会暂停，用户会看到屏幕闪烁。这符合常理，因为用户不想看到 'photos'，而 Comments 组件还没有准备好渲染其内容，而 React 为了保证用户体验的一致性，只能显示上面的 Glimmer，别无选择。

然而，有时这种用户体验并不可取。特别是在准备新 UI 时，展示 “旧” 的 UI 会体验更好。你可以尝试使用新的 startTransition API 来让 React 实现这一点：

jsx 复制代码

```jsx
function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}
```

2.异常捕获边界（Error boundaries）

错误边界是一种 React 组件，这种组件可以捕获发生在其子组件树任何位置的 JavaScript 错误，并打印这些错误，同时展示降级 UI，而并不会渲染那些发生崩溃的子组件树。错误边界可以捕获发生在整个子组件树的渲染期间、生命周期方法以及构造函数中的错误。

> 注意 错误边界无法捕获以下场景中产生的错误：
>
> 1. 事件处理（了解更多）
> 2. 异步代码（例如 setTimeout 或 requestAnimationFrame 回调函数）
> 3. 服务端渲染

> 4. 它自身抛出来的错误（并非它的子组件）

使用

如果一个 class 组件中定义了 static getDerivedStateFromError() 或 componentDidCatch() 这两个生命周期方法中的任意一个（或两个）时，那么它就变成一个错误边界。当抛出错误后，请使用 static getDerivedStateFromError() 渲染备用 UI ，使用 componentDidCatch() 打印错误信息。

javascript 复制代码

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

3. 命名导出

命名导出（Named Exports） React.lazy 目前只支持默认导出（default exports）。如果你想被引入的模块使用命名导出（named exports），你可以创建一个中间模块，来重新导出为默认模块。这能保证 tree shaking 不会出错，并且不必引入不需要的组件。

javascript 复制代码

```javascript
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

4. Context

Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。

vbnet 复制代码

```vbnet
API
React.createContext
Context.Provider
Class.contextType
Context.Consumer
Context.displayName
```

javascript 复制代码

```javascript
// Context 可以让我们无须明确地传遍每一个组件，就能将值深入传递进组件树。
// 为当前的 theme 创建一个 context（“light”为默认值）。
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    // 使用一个 Provider 来将当前的 theme 传递给以下的组件树。
    // 无论多深，任何组件都能读取这个值。
    // 在这个例子中，我们将 “dark” 作为当前的值传递下去。
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// 中间的组件再也不必指明往下传递 theme 了。
function Toolbar() {
  return (
```

```
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // 指定 contextType 读取当前的 theme context。
  // React 会往上找到最近的 theme Provider，然后使用它的值。
  // 在这个例子中，当前的 theme 值为 “dark”。
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

你的组件并不限制于接收单个子组件。你可能会传递多个子组件，甚至会为这些子组件（children）封装多个单独的“接口（slots）”

javascript 复制代码

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}
```

React.createContext

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 Provider 中读取到当前的 context 值。

只有当组件所处的树中没有匹配到 Provider 时，其 defaultValue 参数才会生效。这有助于在不使用 Provider 包装组件的情况下对组件进行测试。注意：将 undefined 传递给 Provider 的

value 时，消费组件的 defaultValue 不会生效。

javascript 复制代码

```
const MyContext = React.createContext(defaultValue);
```

Context.Provider

javascript 复制代码

```
<MyContext.Provider value={/* 某个值 */}>
```

每个 Context 对象都会返回一个 Provider React 组件，它允许消费组件订阅 context 的变化。

Provider 接收一个 value 属性，传递给消费组件。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 value 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 shouldComponentUpdate 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

通过新旧值检测来确定变化，使用了与 Object.is 相同的算法。

Class.contextType

挂载在 class 上的 contextType 属性会被重赋值为一个由 React.createContext() 创建的 Context 对象。这能让你使用 this.context 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

javascript 复制代码

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* 在组件挂载完成后，使用 MyContext 组件的值来执行一些有副作用的操作 */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
```

```
    let value = this.context;
    /* 基于 MyContext 组件的值进行渲染 */
  }
}
MyClass.contextType = MyContext;
```

Context.Consumer

这种方法需要一个函数作为子元素（function as a child）。这个函数接收当前的 context 值，并返回一个 React 节点。传递给函数的 value 值等价于组件树上方离这个 context 最近的 Provider 提供的 value 值。如果没有对应的 Provider，value 参数等同于传递给 createContext() 的 defaultValue。

javascript 复制代码

```
<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染*/}
</MyContext.Consumer>
```

Context.displayName

context 对象接受一个名为 displayName 的 property，类型为字符串。React DevTools 使用该字符串来确定 context 要显示的内容。

javascript 复制代码

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" 在 DevTools 中
<MyContext.Consumer> // "MyDisplayName.Consumer" 在 DevTools 中
```

示例：

javascript 复制代码

```
import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// 一个使用 ThemedButton 的中间组件
function Toolbar(props) {
  return (
    <ThemedButton onClick={props.changeTheme}>
      Change Theme
    </ThemedButton>
  );
```

```
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light,
    };

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };
  }

  render() {
    // 在 ThemeProvider 内部的 ThemedButton 按钮组件使用 state 中的 theme 值，
    // 而外部的组件使用默认的 theme 值
    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
          <Toolbar changeTheme={this.toggleTheme} />
        </ThemeContext.Provider>
        <Section>
          <ThemedButton />
        </Section>
      </Page>
    );
  }
}

ReactDOM.render(<App />, document.root);
```