React新特性实例详解（memo、lazy、suspense、hooks）

13266836563 19年1月

1.memo

其实react.memo的实现很简单，就几行代码。

```
export default function memo<Props>(
  type: React$ElementType,
  compare?: (oldProps: Props, newProps: Props) => boolean,
) {
  if (__DEV__) {
    if (!isValidElementType(type)) {
      warningWithoutStack(
        false,
        'memo: The first argument must be a component. Instead ' +
          'received: %s',
        type === null ? 'null' : typeof type,
      );
    }
  }
  return {
    $$typeof: REACT_MEMO_TYPE,
    type,
    compare: compare === undefined ? null : compare,
  };
}
```

可以看到，最终返回的是一个对象，这个对象带有一些标志属性，在react Fiber的过程中会做相应的处理。

在**ReactFiberBeginWork.js** 中可以看到：

```
if (updateExpirationTime < renderExpirationTime) {
    // This will be the props with resolved defaultProps,
    // unlike current.memoizedProps which will be the unresolved ones.
    const prevProps = currentChild.memoizedProps;
    // Default to shallow comparison
    let compare = Component.compare;
    compare = compare !== null ? compare : shallowEqual;
    if (compare(prevProps, nextProps) && current.ref === workInProgress.ref) {
      return bailoutOnAlreadyFinishedWork(
        current,
        workInProgress,
        renderExpirationTime,
      );
    }
  }
```

根据传入的`compare`函数比较`prevProps`和`nextProps`，最终决定生成对象，并影响渲染效果。

其实在这之前，早已经有一个生命周期函数实现了相同的功能。他就是`shouldComponentUpdate`。

之所以再增加这个`memo`，也是react团队一直在秉承的信念。那就是让一切变得更加函数式。

通过一个例子来看看memo如何使用。

先创建一个简单组件SubComponent。

```
const SubComponent = props =>
  <>
    i am {props.name}. hi~
  </>
```

调用React.memo创建memo组件

```
const Memo = React.memo(SubComponent, (prevProps, nextProps) =>
  prevProps.name === nextProps.name
);
```

在页面上调用memo

```
<div className="App">
  <Memo name={name} />
</div>
```

memo接收两个参数，一个是组件，一个是函数。这个函数就是定义了memo需不需要render的钩子。

比较前一次的props跟当前props，返回true表示不需要render。

也就是传给Memo的name不变时，不会触发SubComponent的render函数。

当前页面上的SubComponent还是之前的，并没有重新渲染。这也是为啥叫memo的原因吧。

2.lazy and suspence

React.lazy 用于做Code-Splitting，代码拆分。类似于按需加载，渲染的时候才加载代码。

用法如下：

```
import React, {lazy} from 'react';
const OtherComponent = lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

lazy(() => import('./OtherComponent'))使用es6的import()返回一个promise，类似于：

```
lazy(() => new Promise(resolve =>
  setTimeout(() =>
    resolve(
      // 模拟ES Module
      {
        // 模拟export default
        default: function render() {
          return <div>Other Component</div>
        }
      }
    ),
    3000
  )
));
```

`React.lazy`的提出是一种更优雅的条件渲染解决方案。

之所以说他更优雅，是因为他将条件渲染的优化提升到了框架层。

这里我们引出suspense。

当我们组件未渲染完成，需要loading时，可以这么写：

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

在我们的业务场景中，OtherComponent可以代表多个条件渲染组件，我们全部加载完成才取消loding。

只要promise没执行到resolve，suspense都会返回fallback中的loading。

代码简洁，loading可提升至祖先组件，易聚合。相当优雅的解决了条件渲染。

关于suspense的异步渲染原理有篇文章写的很好，感兴趣的在文末查看。

3.hooks（重点介绍useEffect）

hooks提出有一段时间了，dan也一直在推广，并且表示很快加入react正式版本。

关于一些介绍，直接看官网会更好。

hooks常用api有：`useState`、`useEffect`、`useContext`、`useReducer`、`useRef`等。

主要操作一下`useEffect`，用处很大。举一反三。

当`all is function`，没了component，自然也没了各种生命周期函数，此时`useEffect`登场。

下面通过一个组件实例来说明。

影像组件，功能有：前端加水印、实现拖拽。

大致实现如下：

```
class ImageModal extends Component {
  constructor(props) {
    ...
  }

  componentDidMount() {
    // 画水印、注册拖拽事件逻辑
    // 以及其他的image处理相关逻辑
  }

  componentDidUpdate(nextProps, prevProps) {
    if (nextProps.cur !== prevProps.cur) {
      // 切换时重置状态（比如 旋转角度、大小等）逻辑
      // image特有逻辑
    }
  }
```

```
  render() {
    return <>
      ...
      <img ... />
    </img>
  }
}
```

ImageModal负责渲染图片modal，现在有另一个modal用来渲染html模板。

命名为HtmlModal，HtmlModal接受后端返回的html，经过处理后内嵌在网页中。

同样要求加水印、拖拽的功能等。

也就是image跟html有部分逻辑相同有部分不相同。

基于这个考虑，再写一个组件。

同理实现如下：

```
class HtmlModal extends Component {
  constructor(props) {
    ...
  }

  componentDidMount() {
    // 画水印、注册拖拽事件逻辑
    // 以及其他的html处理相关逻辑
  }

  componentDidUpdate(nextProps, prevProps) {
    if (nextProps.cur !== prevProps.cur) {
      // 切换时重置状态（比如 旋转角度、大小等）逻辑
      // html特有逻辑
    }
  }

  render() {
    return <>
      ...
      <div dangerouslySetInnerHTML={{ __html: ... }}></div>
    </img>
  }
}
```

可以看到`HtmlModal`和`ImageModal`在`componentDidMount`和`componentDidUpdate`周期中有不少逻辑是相同的。

如果我们使用`useEffect`的话，可以怎么实现这个复用和分离呢？来看看。

```
export function useMoveEffect() {
  // 第二个参数传了固定值 []
  // 相当于 componentDidMount
  useEffect(() => {
    // 实现拖拽逻辑
  }, []);
}

export function useDrawMarkEffect(cur) {
  useEffect(() => {
    // 实现水印逻辑
```

```
  }, []);
}

export function useResetEffect(cur); {
  // 第二个参数传了固定值 [ cur ]
  // 相当于 componentDidUpdate 比较 cur
  useEffect(() => {
    // 实现重置逻辑
  }, [ cur ]);
}

function useOtherImageEffect(...) {
  useEffect(() => {
    // 实现image特有逻辑
  }, [ ... ]);
}

function ImageModal (props) {
  // 细分 Effect，方便复用
  useMoveEffect();
  useDrawMarkEffect();
  useResetEffect(props.cur);
  ...

  useOtherImageEffect(...);

  return <>
    ...
    <img ... />
  </img>

}
```

ok，有了上面的梳理和`useEffect`重构，我们来编写`HtmlModal`:

```
import { useMoveEffect, useDrawMarkEffect, useResetEffect } from './imageModal'

function useOtherHtmlEffect(...) {
  useEffect(() => {
    // 实现html特有逻辑
  }, [ ... ]);
}

function HtmlModal (props) {
  // 细分 Effect，方便复用
  useMoveEffect();
  useDrawMarkEffect();
  useResetEffect(props.cur);
  ...

  useOtherHtmlEffect(...);

  return <>
    ...
    <img ... />
  </img>

}
```

以上，实现了生命周期中重复逻辑的复用。以后无论新增什么modal，都可以复用逻辑，摆脱了 `ctr c/ ctr v`。

从而组件变得更小、代码变得简洁，提升编程体验。

参考资料：
***React v16.6.0: lazy, memo and contextType** (官网)*

***Hooks API Reference** (官网)*

***Making Sense of React Hooks** (dan)*

***React Suspense** (中文)*

觉得有帮助的点个赞，甚至可以关注一波哦~