React组件化

资源

Context参考

HOC参考

Hooks参考

组件跨层级通信 - Context

范例：模拟redux存放全局状态，在组件间共享

```
import React from "react";

// 创建上下文
const Context = React.createContext();

// 获取Provider和Consumer
const Provider = Context.Provider;
const Consumer = Context.Consumer;

// Child显示计数器，并能修改它，多个Child之间需要共享数据
function Child(props) {
  return <div onClick={() => props.add()}>{props.counter}</div>;
}

export default class ContextTest extends React.Component {
  // state是要传递的数据
  state = {
    counter: 0
  };

  // add方法可以修改状态
  add = () => {
    this.setState(nextState => ({ counter: nextState.counter + 1 }));
  };

  // counter状态变更
  render() {
    return (
      <Provider value={{ counter: this.state.counter, add: this.add }}>
        {/* Consumer中内嵌函数，其参数是传递的数据，返回要渲染的组件 */}
        {/* 把value展开传递给Child */}
        <Consumer>{value => <Child {...value} />}</Consumer>
        <Consumer>{value => <Child {...value} />}</Consumer>
      </Provider>
```

```
    );
  }
}
```

高阶组件

范例：为展示组件添加获取数据能力

```
// Hoc.js
import React from "react";

// Lesson保证功能单一，它不关心数据来源，只负责显示
function Lesson(props) {
  return (
    <div>
      {props.stage} - {props.title}
    </div>
  );
}

// 模拟数据
const lessons = [
  { stage: "React", title: "核心API" },
  { stage: "React", title: "组件化1" },
  { stage: "React", title: "组件化2" }
];

// 高阶组件withContent负责包装传入组件Comp
// 包装后组件能够根据传入索引获取课程数据，真实案例中可以通过api查询得到
const withContent = Comp => props => {
  const content = lessons[props.idx];

  // {...props}将属性展开传递下去
  return <Comp {...content} />;
};

// LessonWithContent是包装后的组件
const LessonWithContent = withContent(Lesson);

export default function HocTest() {
  // HocTest渲染三个LessonWithContent组件
  return (
    <div>
      {[0,0,0].map((item, idx) => (
        <LessonWithContent idx={idx} key={idx} />
      ))}
    </div>
  );
}
```

范例：改造前面案例使上下文使用更优雅

```
// withConsumer是高阶组件工厂，它能根据配置返回一个高阶组件
function withConsumer(Consumer) {
  return Comp => props => {
    return <Consumer>{value => <Comp {...value} {...props} />}</Consumer>;
  };
}
// Child显示计数器，并能修改它，多个Child之间需要共享数据
// 新的Child是通过withConsumer(Consumer)返回的高阶组件包装所得
const Child = withConsumer(Consumer)(function (props) {
  return <div onClick={() => props.add()} title={props.name}>{props.counter}</div>;
});

export default class ContextTest extends React.Component {
  render() {
    return (
      <Provider value={{ counter: this.state.counter, add: this.add }}>
        {/* 改造过的Child可以自动从Consumer获取值，直接用就好了 */}
        <Child name="foo"/>
        <Child name="bar"/>
      </Provider>
    );
  }
}
```

链式调用

```
// 高阶组件withLog负责包装传入组件Comp
// 包装后组件在挂载时可以输出日志记录
const withLog = Comp => {
  // 返回组件需要生命周期，因此声明为class组件
  return class extends React.Component {
    render() {
      return <Comp {...this.props} />;
    }
    componentDidMount() {
      console.log("didMount", this.props);
    }
  };
};

// LessonWithContent是包装后的组件
const LessonWithContent = withLog(withContent(Lesson));
```

装饰器写法

CRA项目中默认不支持js代码使用装饰器语法，可修改后缀名为tsx则可以直接支持

```
// 装饰器只能用在class上
// 执行顺序从下往上
@withLog
@withContent
class Lesson2 extends React.Component {
  render() {
    return (
      <div>
        {this.props.stage} - {this.props.title}
      </div>
    );
  }
}

export default function HocTest() {
  // 这里使用Lesson2
  return (
    <div>
      {[0, 0, 0].map((item, idx) => (
        <Lesson2 idx={idx} key={idx} />
      ))}
    </div>
  );
}
```

> 注意修改App.js中引入部分，添加一个后缀名
>
> 要求cra版本高于2.1.0

组件复合 - Composition

复合组件给与你足够的敏捷去定义自定义组件的外观和行为

组件复合

范例：Dialog组件负责展示，内容从外部传入即可，components/Composition.js

```
import React from "react";

// Dialog定义组件外观和行为
function Dialog(props) {
  return <div style={{ border: "1px solid blue" }}>{props.children}</div>;
}

export default function Composition() {
```

```
  return (
    <div>
      {/* 传入显示内容 */}
      <Dialog>
        <h1>组件复合</h1>
        <p>复合组件给与你足够的敏捷去定义自定义组件的外观和行为</p>
      </Dialog>
    </div>
  );
}
```

范例：传个对象进去，key表示具名插槽

```
import React from "react";

// 获取相应部分内容展示在指定位置
function Dialog(props) {
  return (
    <div style={{ border: "1px solid blue" }}>
      {props.children.default}
      <div>{props.children.footer}</div>
    </div>
  );
}

export default function Composition() {
  return (
    <div>
      {/* 传入显示内容 */}
      <Dialog>
        {{
          default: (
            <>
              <h1>组件复合</h1>
              <p>复合组件给与你足够的敏捷去定义自定义组件的外观和行为</p>
            </>
          ),
          footer: <button onClick={() => alert("react确实好")}>确定</button>
        }}
      </Dialog>
    </div>
  );
}
```

如果传入的是函数，还可以实现作用域插槽的功能

```
function Dialog(props) {
```

```
  // 备选消息
  const messages = {
      "foo": {title: 'foo', content: 'foo~'},
      "bar": {title: 'bar', content: 'bar~'},
  }
  // 执行函数获得要显示的内容
  const {body, footer} = props.children(messages[props.msg]);

  return (
    <div style={{ border: "1px solid blue" }}>
      {/* 此处显示的内容是动态生成的 */}
      {body}
      <div>{footer}</div>
    </div>
  );
}

export default function Composition() {
  return (
    <div>
      {/* 执行显示消息的key */}
      <Dialog msg="foo">
        {/* 修改为函数形式，根据传入值生成最终内容 */}
        {({title, content}) => ({
          body: (
            <>
              <h1>{title}</h1>
              <p>{content}</p>
            </>
          ),
          footer: <button onClick={() => alert("react确实好")}>确定</button>
        })}
      </Dialog>
    </div>
  );
}
```

如果props.children是jsx，此时它是不能修改的

范例：实现RadioGroup和Radio组件，可通过RadioGroup设置Radio的name

```
function RadioGroup(props) {
  // 不可行，
  // React.Children.forEach(props.children, child => {
  // child.props.name = props.name;
  // });
  return (
    <div>
      {React.Children.map(props.children, child => {
        // 要修改child属性必须先克隆它
        return React.cloneElement(child, { name: props.name });
```

```
      })}
    </div>
  );
}
// Radio传入value,name和children，注意区分
function Radio({ children, ...rest }) {
  return (
    <label>
      <input type="radio" {...rest} />
      {children}
    </label>
  );
}

export default function Composition() {
  return (
    <div>
      {/* 执行显示消息的key */}
      <RadioGroup name="mvvm">
        <Radio value="vue">vue</Radio>
        <Radio value="react">react</Radio>
        <Radio value="ng">angular</Radio>
      </RadioGroup>
    </div>
  );
}
```

Hooks

准备工作

升级react、react-dom

```
npm i react react-dom -S
```

状态钩子 State Hook

- 创建HooksTest.js

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

export default function HooksTest() {
  // useState(initialState)，接收初始状态，返回一个由状态和其更新函数组成的数组
  const [fruit, setFruit] = useState("");
  return (
    <div>
      <p>{fruit === "" ? "请选择喜爱的水果：" : `您的选择是：${fruit}`}</p>
    </div>
  );
}
```

- 声明多个状态变量

```
// 声明列表组件
function FruitList({fruits, onSetFruit}) {
  return (
    <ul>
      {fruits.map(f => (
        <li key={f} onClick={() => onSetFruit(f)}>
          {f}
        </li>
      ))}
    </ul>
  );
}

export default function HooksTest() {
  // 声明数组状态
  const [fruits, setFruits] = useState(["香蕉", "西瓜"]);
  return (
    <div>
      {/*添加列表组件*/}
      <FruitList fruits={fruits} onSetFruit={setFruit}/>
    </div>
  );
}
```

- 用户输入处理

```
// 声明输入组件
function FruitAdd(props) {
  // 输入内容状态及设置内容状态的方法
  const [pname, setPname] = useState("");
  // 键盘事件处理
  const onAddFruit = e => {
    if (e.key === "Enter") {
      props.onAddFruit(pname);
      setPname("");
    }
```

```
  };
  return (
    <div>
      <input
        type="text"
        value={pname}
        onChange={e => setPname(e.target.value)}
        onKeyDown={onAddFruit}
      />
    </div>
  );
}
export default function HooksTest() {
  // ...
  return (
    <div>
      {/*添加水果组件*/}
      <FruitAdd onAddFruit={pname => setFruits([...fruits, pname])} />
    </div>
  );
}
```

副作用钩子 Effect Hook

`useEffect` 给函数组件增加了执行副作用操作的能力。

副作用（Side Effect）是指一个 function 做了和本身运算返回值无关的事，比如：修改了全局变量、修改了传入的参数、甚至是 console.log()，所以 ajax 操作，修改 dom 都是算作副作用。

- 异步数据获取，更新HooksTest.js

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

useEffect(()=>{
    setTimeout(() => {
        setFruits(['香蕉','西瓜'])
    }, 1000);
},[])// 设置空数组意为没有依赖，则副作用操作仅执行一次
```

> 如果副作用操作对某状态有依赖，务必添加依赖选项
>
> ```
> useEffect(() => {
>     document.title = fruit;
> }, [fruit]);
> ```

- 清除工作：有一些副作用是需要清除的，清除工作非常重要的，可以防止引起内存泄露

```
  useEffect(() => {
    const timer = setInterval(() => {
        console.log('msg');
    }, 1000);

    return function(){
        clearInterval(timer);
    }
  }, []);
```

useReducer

useReducer是useState的可选项，常用于组件有复杂状态逻辑时，类似于redux中reducer概念。

- 商品列表状态维护

```
import { useReducer } from "react";

// 添加fruit状态维护fruitReducer
function fruitReducer(state, action) {
  switch (action.type) {
    case "init":
      return action.payload;
    case "add":
      return [...state, action.payload];
    default:
      return state;
  }
}

export default function HooksTest() {
  // 组件内的状态不需要了
  // const [fruits, setFruits] = useState([]);

  // useReducer(reducer, initState)
  const [fruits, dispatch] = useReducer(fruitReducer, []);

  useEffect(() => {
    setTimeout(() => {
      // setFruits(["香蕉", "西瓜"]);
      // 变更状态，派发动作即可
      dispatch({ type: "init", payload: ["香蕉", "西瓜"] });
    }, 1000);
  }, []);

  return (
    <div>
      {/*此处修改为派发动作*/}
      <FruitAdd onAddFruit={pname => dispatch({type: 'add', payload: pname})} />
    </div>
  );
```

```
  }
```

useContext

useContext用于在快速在函数组件中导入上下文。

```
import React, { useContext } from "react";

// 创建上下文
const Context = React.createContext();

export default function HooksTest() {
  // ...
  return (
    {/* 提供上下文的值 */}
    <Context.Provider value={{fruits,dispatch}}>
      <div>
        {/* 这里不再需要给FruitAdd传递状态mutation函数，实现了解耦 */}
        <FruitAdd />
      </div>
    </Context.Provider>
  );
}

function FruitAdd(props) {
  // 使用useContext获取上下文
  const {dispatch} = useContext(Context)
  const onAddFruit = e => {
    if (e.key === "Enter") {
      // 直接派发动作修改状态
      dispatch({ type: "add", payload: pname })
      setPname("");
    }
  };
  // ...
}
```

Hooks相关拓展

1. Hooks规则
2. 自定义Hooks
3. 一堆nb的实现

作业练习

1.尝试利用hooks编写一个完整的购物应用

2.基于useReducer的方式能否处理异步action