React组件化01

课堂目标

掌握组件化开发中多种实现技术

1. 了解组件化概念，能设计并实现自己需要的组件
2. 掌握使用跨层级通信-Context（新API在v>=16.3）
3. 组件复合 - Composition
4. 高阶组件 - HOC
5. Hooks（>=16.8）

知识要点

2. 运用Context
3. 运用组件复合 - Composition
4. 运用高阶组件 - HOC
5. Hooks使用

资源

Context参考

HOC参考

Hooks参考

antD参考

起步

组件化优点：

1. 增强代码重用性，提高开发效率
2. 简化调试步骤，提升整个项目的可维护性
3. 便于协同开发
4. 注意点：降低耦合性

快速开始

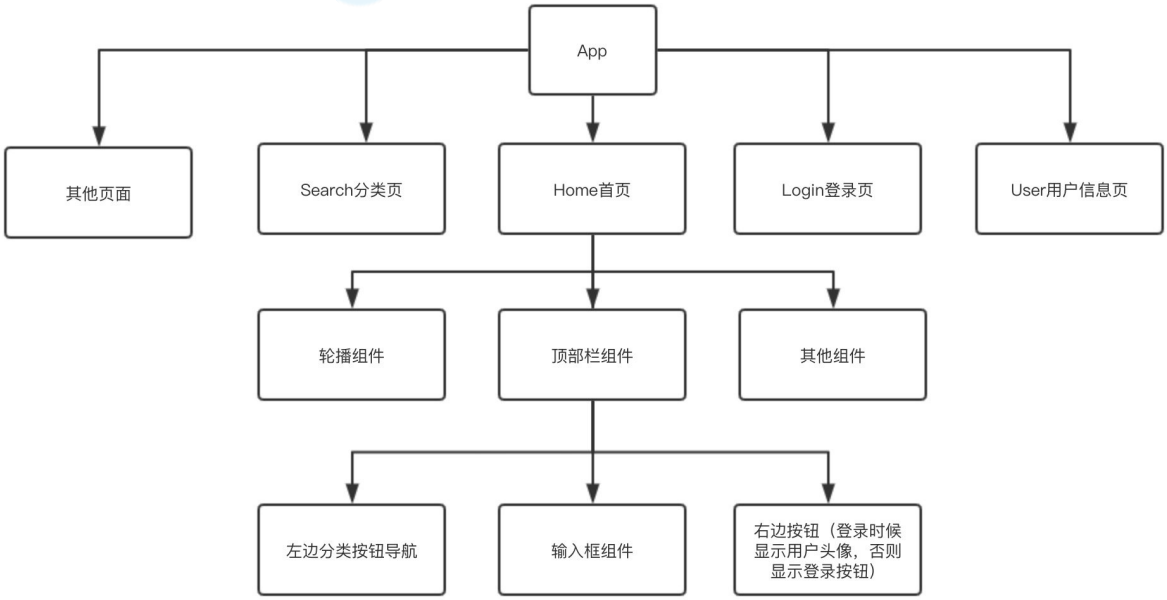
(https://www.html.cn/create-react-app/docs/getting-started/)

```
npx create-react-app lesson2

cd lesson2

npm start
```

组件跨层级通信 - Context



React中使用Context实现祖代组件向后代组件跨层级传值。Vue中的provide & inject来源于Context

在Context模式下有两个角色：

- Provider：外层提供数据的组件
- Consumer ： 内层获取数据的组件

使用Context

创建Context => 获取Provider和Consumer => Provider提供值 => Consumer消费值

范例：模拟redux存放全局状态，在组件间共享

```
//App.js
import React from 'react';
import Home from './pages/Home'
import User from './pages/User'

import { Provider } from './AppContext' //引入Context的Provider

const store = {
  home: {
    imgs: [
      {
        "src":
"//m.360buyimg.com/mobilecms/s700x280_jfs/t1/49973/2/8672/125419/5d679259Ecd46f8
e7/0669f8801dff67e8.jpg!cr_1125x445_0_171!q70.jpg.dpg"
      }
    ]
  },
  user: {
    isLogin: true,
    userName: "Rabbit"
  }
}

function App() {
  return (
    <div className="app">
      <Provider value={store}>
        <Home />
      </Provider>
    </div>
  );
}

export default App;
```

```
//AppContext.js
import React, { Component } from 'react'

export const Context = React.createContext()
export const Provider = Context.Provider
export const Consumer = Context.Consumer
```

```
// /pages/Home.js
import React, { Component } from 'react'
import { Consumer } from '../AppContext';


export default class Home extends Component {
  render() {
    return (
      <Consumer>
        {
          ctx => <HomeCmp {...ctx} />
        }
      </Consumer>

    )
  }
}

function HomeCmp(props) {
  const { home, user } = props
  const { isLogin, userName } = user
  return (
    <div>
      {
        isLogin ? userName : '登录'
      }
    </div>
  )
}
```

```
// /pages/User.js
import React, { Component } from 'react'
import { Consumer } from '../AppContext';
import TabBar from '../components/TabBar';


export default class User extends Component {
  render() {
    return (
      <>
        <Consumer>
          {
            ctx => <UserCmp {...ctx} />
          }
        </Consumer>
        <TabBar />
      </>
    )
  }
}

function UserCmp(props) {
  const { home, user } = props
  const { isLogin, userName } = user
  return (
    <div>
      {
```

```
        isLogin ? userName : '登录'
      }
    </div>
  )
}
```

```
// /components/TabBar
import React from 'react'
import { Consumer } from '../AppContext';

export default function TabBar() {
  return (
    <div>
      <Consumer>
        {
          ctx => <TabBarCmp {...ctx} />
        }
      </Consumer>
    </div>
  )
}


function TabBarCmp(props) {
  const { home, user } = props
  const { isLogin, userName } = user
  return (
    <div>
      {
        isLogin ? userName : '登录'
      }
    </div>
  )

}
```

在React的官方文档中，`Context`被归类为高级部分(Advanced)，属于React的高级API，但官方并不建议在稳定版的App中使用Context。

不过，这并非意味着我们不需要关注`Context`。事实上，很多优秀的React组件都通过Context来完成自己的功能，比如react-redux的`<Provider />`，就是通过`Context`提供一个全局态的`store`，路由组件react-router通过`Context`管理路由状态等等。在React组件开发中，如果用好`Context`，可以让你的组件变得强大，而且灵活。

> 函数组件中可以通过useContext引入上下文，后面hooks部分介绍

高阶组件-HOC

为了提高组件复用率，可测试性，就要保证组件功能单一性；但是若要满足复杂需求就要扩展功能单一的组件，在React里就有了HOC（Higher-Order Components）的概念，

定义：是一个函数，它接收一个组件并返回另一个组件。

基本使用

```
// HocPage.js
import React from 'react'

function Child(props) {
  return <div className="border">Child+{props.name}</div>;
}

//这里大写开头的Cmp是指function或者class组件
const foo = Cmp => props => {
  return <Cmp {...props} />
}

/*const foo = (Cmp) => {
  return (props) => {
    return <Cmp {...props} />
  }
}*/
const Foo = foo(Child)
export default function HocPage(props) {
  return (
    <div>
      HocPage
      <Foo name={"msg"} />
      {/* {foo(Child)({ name: "msg" })} */}
    </div>
  )
}
```

运用hoc改写前面的Context例子：

```
// /pages/User.js
import React from 'react'
import { Consumer } from '../AppContext';
import Layout from './Layout';

const handleConsumer = Cmp => props => {
  return <Consumer>
    {
      ctx => <Cmp {...ctx} {...props}></Cmp>
    }
  </Consumer>
}
const HandleConsumer = handleConsumer(UserCmp)
export default function User(props) {
  return (
    <Layout title="用户中心">
      <HandleConsumer />
    </Layout>
  )
}
function UserCmp(props) {
  console.log('user', props)
  return <div>
    User
  </div>
}
```

链式调用

```
import React from 'react'

function Child(props) {
  return <div>Child</div>
}

const foo = Cmp => props => {
  return <div style={{ background: 'red' }}>
    <Cmp {...props} />
  </div>
}

const foo2 = Cmp => props => {
  return <div style={{ border: 'solid 1px green' }}>
    <Cmp {...props} />
  </div>
}
const Foo = foo2(foo(Child))
export default function HocPage() {
  return (
    <div>
      HocPage
      <Foo />
    </div>
  )
}
```

装饰器写法

高阶组件本身是对装饰器模式的应用，自然可以利用ES7中出现的装饰器语法来更优雅的书写代码。

cra项目配置装饰器方法：

1. npm run eject （如果是直接down下来的代码，并且有改动，先commit本地代码）
2. 配置package.json（如果不会配置， 直接看提供的package.json代码）

```
"babel": {
    "presets": [
      "react-app"
    ],
    "plugins": [
      [
        "@babel/plugin-proposal-decorators",
        {
          "legacy": true
        }
      ]
    ]
  }
```

3. 安装装饰器插件 npm install @babel/plugin-proposal-decorators --save-dev
4. 如果介意vscode的warning，vscode设置里加上
   `javascript.implicitProjectConfig.experimentalDecorators": true`

   温馨提示：~~配置有点复杂，如果出现问题看log找办法解决下，或者用下节课的方式来配置。~~

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，cra版本高于2.1.0。

```
// 装饰器只能用在class上
// 执行顺序从下往上
const foo = Cmp => props => {
  return (
    <div className="border">
      <Cmp />
    </div>
  );
};

const foo2 = Cmp => props => {
  return (
    <div className="border" style={{ border: "green 1px solid" }}>
      <Cmp />
    </div>
  );
};

@foo2
@foo
class Child extends Component {
  render() {
    return <div className="border">Child</div>;
  }
}
// const Foo = foo2(foo(Child));
export default class HocPage extends Component {
  render() {
    return (
      <div>
        <h1>HocPage</h1>
        {/* <Foo /> */}
        <Child />
      </div>
    );
  }
}
```

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，这种方式更明确和安全。如果组件间有公用的非UI逻辑，将它们抽取为JS模块导入使用而不是继承它。

基本使用

不具名

```
// /pages/Layout.js
import React, { Component } from 'react'

export default class Layout extends Component {
  componentDidMount() {
    const { title = "商城" } = this.props
    document.title = title
  }
  render() {
    const { children, title = "商城" } = this.props
    return (
      <div style={{ background: 'yellow' }}>
```

```
        <p>{title}</p>
        {
          children.btns ? children.btns : children
        }
        <TabBar />
      </div>
    )
  }
}

function TabBar(props) {
  return <div>
    TabBar
</div>
}
```

```
// /pages/Home.js
import React, { Component } from 'react'
import { Consumer } from '../AppContext';
import Layout from './Layout';

export default class Home extends Component {
  render() {
    return (
      <Consumer>
        {
          ctx => <HomeCmp {...ctx} />
        }
      </Consumer>
    )
  }
}

function HomeCmp(props) {
  const { home, user } = props
  const { carsouel = [] } = home
  const { isLogin, userName } = user
  return (
    <Layout title="首页">
      <div>
        <div>{isLogin ? userName : '未登录'}</div>
        {
          carsouel.map((item, index) => {
            return <img key={'img' + index} src={item.img} />
          })
        }
      </div>
    </Layout>
  )
}
```

传个对象进去就是具名插槽

```
// /pages/User.js
import React, { Component } from 'react'
import { Consumer } from '../AppContext';
import Layout from './Layout';

export default class User extends Component {
  render() {
    return (
      <div>
        <p>用户中心</p>
        <Consumer>
          {
            ctx => <UserCmp {...ctx} />
          }
        </Consumer>
      </div>
    )
  }
}

function UserCmp(props) {
  const { home, user } = props
  const { carsouel = [] } = home
  const { isLogin, userName } = user
  return (
    <Layout title="用户中心">
      {
        {
          btns: <button>下载</button>
        }
      }

      {/* <div>
        <div>用户名： {isLogin ? userName : '未登录'}</div>
      </div> */}
    </Layout>
  )
}
```

实现一个简单的复合组件，如antD的Card

```
import React, { Component } from 'react'

function Card(props) {
  return <div xu="card">
    {
      props.children
    }
  </div>
}

function Formbutton(props) {
  return <div className="Formbutton">
    <button onClick={props.children.defaultBtns.searchClick}>默认查询</button>
    <button onClick={props.children.defaultBtns.resetClick}>默认重置</button>
    {
```

```
      props.children.btns.map((item, index) => {
        return <button key={'btn' + index} onClick={item.onClick}>{item.title}
</button>
      })
    }
  </div>
}

export default class CompositionPage extends Component {
  render() {
    return (
      <div>
        <Card>
          <p>我是内容</p>
        </Card>
        CompositionPage
        <Card>
          <p>我是内容2</p>
        </Card>
        <Formbutton>
          {{
            /* btns: (
              <>
                <button onClick={() => console.log('enn')}>查询</button>
                <button onClick={() => console.log('enn2')}>查询2</button>
              </>
            ) */
            defaultBtns: {
              searchClick: () => console.log('默认查询'),
              resetClick: () => console.log('默认重置')
            },
            btns: [
              {
                title: '查询',
                onClick: () => console.log('查询')
              }, {
                title: '重置',
                onClick: () => console.log('重置')
              }
            ]
          }}
        </Formbutton>
      </div>
    )
  }
}
```

Hooks

Hook是React16.8一个新增项，它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

Hooks的特点：

- 使你在无需修改组件结构的情况下复用状态逻辑

- 可将组件中相互关联的部分拆分成更小的函数，复杂组件将变得更容易理解
- 更简洁、更易理解的代码

状态钩子 State Hook

- 创建HookPage.js

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

export default function HookPage() {
  // const [date, setDate] = useState(new Date());
  const [counter, setCounter] = useState(0);
  return (
    <div>
      <h1>HookPage</h1>
      <p>{useClock().toLocaleTimeString()}</p>
      <p onClick={() => setCounter(counter + 1)}>{counter}</p>
    </div>
  );
}

//自定义 Hook 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook。
function useClock() {
  const [date, setDate] = useState(new Date());
  useEffect(() => {
    console.log("useEffect");
    const timer = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timer);
  }, []);
  return date;
}
```

> 更新函数类似setState，但它不会整合新旧状态

- 声明多个状态变量

```
import React, { useState, useEffect } from "react";
// import FruitAdd from "../components/FruitAdd";
import FruitList from "../components/FruitList";

export default function HooksPage() {
  const [counter, setCounter] = useState(0);
  const [fruits, setFruits] = useState(["apple", "banana"]);
  return (
    <div>
      <h1>HooksPage</h1>
      <p>{useDate().toLocaleTimeString()}</p>
      <p onClick={() => setCounter(counter + 1)}>{counter}</p>
      {/* <FruitAdd
        fruits={fruits}
        addFruit={name => setFruits([...fruits, name])}
      /> */}
      <FruitList fruits={fruits} setFruits={setFruits} />
    </div>
  );
```

```
}
```

```
import React from "react";

export default function FruitList({ fruits, setFruits }) {
  const delFruit = delIndex => {
    const tem = [...fruits];
    tem.splice(delIndex, 1);
    setFruits(tem);
  };
  return (
    <ul>
      {fruits.map((item, index) => (
        <li key={item} onClick={() => delFruit(index)}>
          {item}
        </li>
      ))}
    </ul>
  );
}
```

- 用户输入处理

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

export default function FruitAdd({ fruits, addFruit }) {
  const [name, setName] = useState("");
  return (
    <div>
      <input value={name} onChange={event => setName(event.target.value)} />
      <button onClick={() => addFruit(name)}>add</button>
    </div>
  );
}
```

副作用钩子 Effect Hook总结：

`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。

React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

- 异步数据获取，更新HooksTest.js

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

useEffect(()=>{
    setTimeout(() => {
        setFruits(['香蕉','西瓜'])
    }, 1000);
})
```

> 测试会发现副作用操作会被频繁调用

- 设置依赖

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

> 如果副作用操作对某状态有依赖，务必添加依赖选项
>
> ```
> import React, { useState, useEffect } from "react";
>
> export default function FruitAdd({ fruits, addFruit }) {
> const [name, setName] = useState("");
> useEffect(() => {
>  document.title = name;
> }, [name]);
> return (
>  
>    <input value={name} onChange={event => setName(event.target.value)}
> />
>    <button onClick={() => addFruit(name)}>add</button>
>  
> );
> }
> ```

- 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);
    }
  }, []);
```

> 组件卸载后会执行返回的清理函数

useReducer

reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

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。

- 水果列表状态维护

```jsx
import React, { useEffect, useReducer } from "react";
import FruitList from "../components/FruitList";
import FruitAdd from "../components/FruitAdd";

function fruitReducer(state = [], action) {
  switch (action.type) {
    case "replace":
    case "init":
      return [...action.payload];
    case "add":
      return [...state, action.payload];
    default:
      return state;
  }
}
export default function UseReducerPage() {
  const [fruits, dispatch] = useReducer(fruitReducer, []);
  useEffect(() => {
    setTimeout(() => {
      dispatch({ type: "init", payload: ["apple", "banana"] });
    }, 1000);
    return () => {};
  }, []);
  return (
    <div>
      <h1>UseReducerPage</h1>
      <FruitAdd
        fruits={fruits}
        addFruit={name => dispatch({ type: "add", payload: name })}
      />
      <FruitList
        fruits={fruits}
        setFruits={newList => dispatch({ type: "init", payload: newList })}
      />
    </div>
  );
}
```

useContext

useContext用于在快速在函数组件中导入上下文。

```
import React, { useContext } from "react";
import { Context } from "../AppContext";

export default function UseContextPage() {
  const ctx = useContext(Context);
  const { name } = ctx.user;
  return (
    <div>
      <h1>UseContextPage</h1>
      <p>{name}</p>
    </div>
  );
}
```

Hook规则

- 只在最顶层使用 Hook，不要在循环，条件或嵌套函数中调用 Hook。

  ```
  //下面这些用法都是错误的
  if (counter % 2) {
    const [counter, setCounter] = useState(0);
  }
  if (counter % 2) {
    useEffect(() => {
      setCounter(100);
    });
  }
  ```

  如果我们想要有条件地执行一个 effect，可以将判断放到 Hook 的*内部*

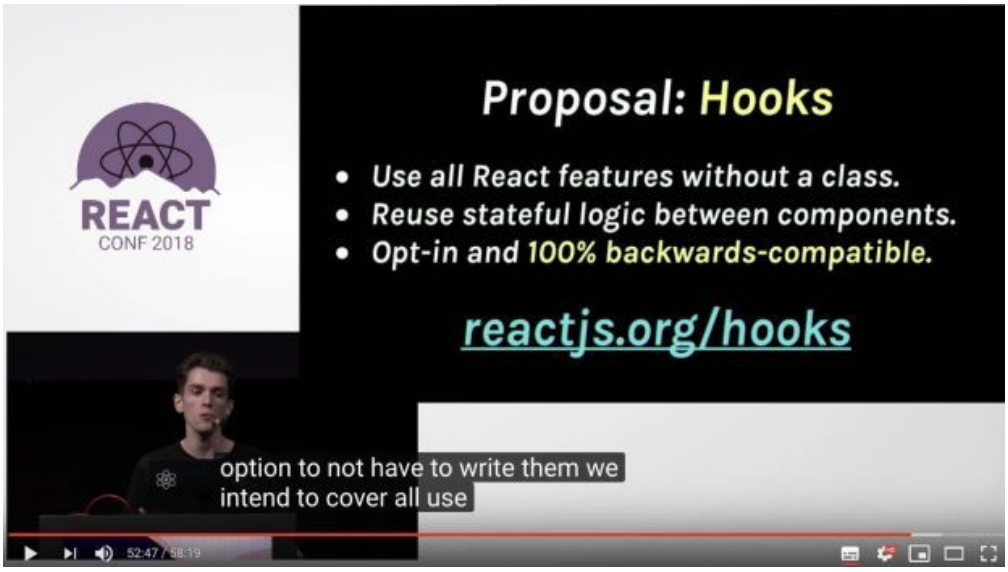
  ```
  useEffect(() => {
    //把条件判断放在effect内部
    if (counter % 2) {
      setCounter(100);
    }
  });
  ```

  我们在单个组件中可以使用多个state hook或者effect hook，那么 React 怎么知道哪个 state 对应哪个 `useState`？答案是 React 靠的是 Hook 调用的顺序。因为我们的示例中，Hook 的调用顺序在每次渲染中都是相同的，所以它能够正常工作。只要 Hook 的调用顺序在多次渲染之间保持一致，React 就能正确地将内部 state 和对应的 Hook 进行关联。

- 只在 React 函数中调用 Hook。不要在普通的 JavaScript 函数中调用 Hook： 要在 React 的函数组件中调用 Hook；在自定义 Hook 中调用其他 Hook。
- 谨记依赖。

Hook相关拓展

3. 更多hook api
4. React Conf Hooks 2018



回顾

下节课内容

组件化02：antd表单实现、弹窗类组件设计与实现、树组件、常见组件优化技术