React组件化

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）
6. 掌握第三方组件的使用

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
6. 使用umi、antD

资源

Context参考

HOC参考

Hooks参考

antD参考

umi参考

[antd-pro安装参考][https://pro.ant.design/docs/getting-started-cn#%E5%AE%89%E8%A3%85]

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

快速开始

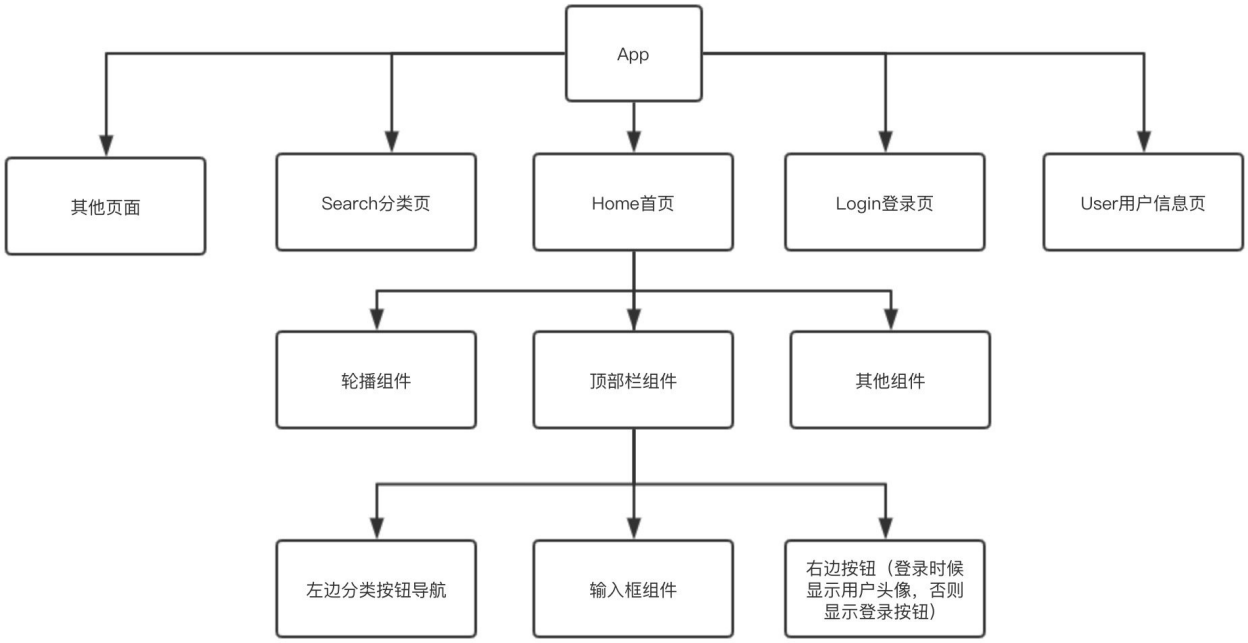
(https://www.html.cn/create-react-app/docs/getting-started/)

```
npx create-react-app my-app

cd my-app

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/t1/49973/2/8672/125419/5d679259Ecd46f8e7/0669f8801dff67e8.jpg!cr_1125x445_0_171!q70.jpg.dpg"
      }
    ]
  },
  user: {
    isLogin: true,
    userName: "true"
  }
}

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`，拖拽组件react-dnd，通过`Context`在组件中分发DOM的Drag和Drop事件，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

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 className="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>
    )
  }
}
```

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：高阶组件是一个工厂函数，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>Child</div>
}

const foo = Cmp => props => {
  return <Cmp {...props} />
}

/*const foo = (Cmp) => {
  return (props) => {
    return <Cmp {...props} />
  }
}*/

export default function HocPage(props) {
  const Foo = foo(Child)
  return (
    <div>
      HocPage
      <Foo />
    </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 {...props}></Cmp>
    }
  </Consumer>
}

export default function User(props) {

  const HandleConsumer = handleConsumer(UserCmp)
  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>
}

export default function HocPage() {
  const Foo = foo2(foo(Child))
  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>
  );
}
```

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());
  useEffect(() => {
    const timerId = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timerId);
  });
  return (
    <div>
      <h1>Home 页面</h1>
      <div>{date.toLocaleTimeString()}</div>
    </div>
  );
}
```

> 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";

export default function HookPage() {
  const [date, setDate] = useState(new Date());
  const [fruits, setFruits] = useState(["apple",
"banana", "berry"]);
```

```
  useEffect(() => {
    const timerId = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timerId);
  });
  const del = delIndex => {
    const tem = [...fruits];
    tem.splice(delIndex, 1);
    setFruits(tem);
  };
  return (
    <div>
      <h1>Home 页面</h1>
      <div>{date.toLocaleTimeString()}</div>
      <FruitList fruits={fruits} onSetFruits=
{del} />
    </div>
  );
}

function FruitList({ fruits, onSetFruits }) {
  return (
    <>
      <h2>点击下面水果删除当前</h2>
      <ul>
        {fruits.map((item, index) => {
          return (
            <li key={"fruit" + index} onClick={()
=> onSetFruits(index)}>
              {item}
            </li>
          );
```

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

- 用户输入处理

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

export default function HookPage() {
  const [date, setDate] = useState(new Date());
  const [fruits, setFruits] = useState(["apple",
"banana", "berry"]);
  useEffect(() => {
    const timerId = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timerId);
  });//副作用 , [date]);
  const del = delIndex => {
    const tem = [...fruits];
    tem.splice(delIndex, 1);
    setFruits(tem);
  };
  return (
    <div>
      <h1>Home 页面</h1>
      <div>{date.toLocaleTimeString()}</div>
      <FruitAdd onAdd={item =>
setFruits([...fruits, item])} />
```

```
      <FruitList fruits={fruits} onSetFruits=
{del} />
    </div>
  );
}

function FruitList({ fruits, onSetFruits }) {
  return (
    <>
      <h2>点击下面水果删除当前</h2>
      <ul>
        {fruits.map((item, index) => {
          return (
            <li key={"fruit" + index} onClick={()
=> onSetFruits(index)}>
              {item}
            </li>
          );
        })}
      </ul>
    </>
  );
}

function FruitAdd(props) {
  const [name, setName] = useState("");
  return (
    <div>
      <h2>增加水果</h2>
      <input type="text" value={name} onChange={e
=> setName(e.target.value)} />
      <button onClick={() =>
props.onAdd(name)}>add</button>
```

```
    </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);
})
```

> 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

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

> 组件卸载后会执行返回的清理函数

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 React, { useReducer, useEffect } from
"react";
import { FruitList, FruitAdd } from "./Fruit";

function fruitReducer(state, action) {
  switch (action.type) {
```

```
    case "init":
    case "replace":
      return action.payload;
    case "add":
      return [...state, action.payload];
    default:
      return state;
  }
}

export default function HookReducer() {
  const [fruits, dispatch] =
useReducer(fruitReducer, []);
  useEffect(() => {
    setTimeout(() => {
      dispatch({ type: "init", payload: ["apple",
"banana"] });
    }, 1000);
  }, []);
  return (
    <div>
      <h1>User 页面</h1>
      <FruitAdd onAdd={item => dispatch({ type:
"add", payload: item })} />
      <FruitList
        fruits={fruits}
        onSetFruits={cur => dispatch({ type:
"replace", payload: cur })}
      />
    </div>
  );
}
```

Fruit.js

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

export function FruitList({ fruits, onSetFruits }) {
  const delCur = delIndex => {
    const tem = [...fruits];
    tem.splice(delIndex, 1);
    onSetFruits(tem);
  };
  return (
    <>
      <h2>点击下面水果删除当前</h2>
      <ul>
        {fruits.map((item, index) => {
          return (
            <li key={"fruit" + index} onClick={() =>
delCur(index)}>
              {item}
            </li>
          );
        })}
      </ul>
    </>
  );
}

export function FruitAdd(props) {
  const [name, setName] = useState("");
  return (
    <div>
      <h2>增加水果</h2>
```

```
      <input type="text" value={name} onChange={e =>
setName(e.target.value)} />
      <button onClick={() =>
props.onAdd(name)}>add</button>
    </div>
  );
}
```

useContext

useContext用于在快速在函数组件中导入上下文。

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

const Context = React.createContext();
const Provider = Context.Provider;

export default function HookContext() {
  const store = {
    userName: "xiaoming",
  };
  return (
    <div>
      <h1>HookContext 页面</h1>
      <Provider value={store}>
        <Child />
      </Provider>
    </div>
  );
}
```

```
function Child(props) {
  const { userName } = useContext(Context);
  return (
    <div>
      Child
      <div>userName: {userName}</div>
    </div>
  );
}
```

Hook相关拓展

1. 基于useReducer的方式能否处理异步action
2. Hook规则
3. 自定义Hook
4. 一堆nb的实现

第三方库

```
antd-pro安装:
yarn create umi
选择 ant-design-pro
npm install
npm start
```

下节课内容

第三方组件使用