

## React组件化

@todo

课堂目标

知识要点

资源

起步

组件

试用 ant-design组件库

配置按需加载

容器组件 VS 展示组件

PureComponent

React.memo

高阶组件

高阶链式调用

高阶组件装饰器写法

组件通信--上下文

老版本的context

新版本上下文

React未来

Fiber

Suspense

拥抱Suspense

Hooks

useState

useEffect

回顾

## @todo

一个实际项目作为载体 设计一个组件

## 课堂目标

1. 学习react组件化
2. 掌握容器组件 VS 展示组件
3. 掌握高阶组件
4. PureComponent
5. 掌握render props
6. 了解异步渲染组件
7. 了解函数化组件Hooks

## 知识要点

1. 组件化

- 2. antd组件库使用
- 3. 组件通信
- 4. 组件的多个模式

## 资源

---

## 起步

---

### 组件

React没有vue那么多api，基本全部都是组件，React的开发模式，大体可以用一个公式表达

```
UI = F(state)
```

### 试用 ant-design组件库

<https://ant.design/docs/react/use-with-create-react-app-cn>

```
npm install antd --save 安装
```

试用button

```
import React, { Component } from 'react'
import Button from 'antd/lib/button'
import "antd/dist/antd.css"
class App extends Component {
  render() {
    return (
      <div className="App">
        <Button type="primary">Button</Button>
      </div>
    )
  }
}

export default App
```

### 配置按需加载

安装react-app-rewired取代react-scripts，可以扩展webpack的配置，类似vue.config.js

```
npm install react-app-rewired@2.0.2-next.0 babel-plugin-import --save
```

### 容器组件 VS 展示组件

基本原则：容器组件负责数据获取，展示组件负责根据props显示信息

优势

1. 如何工作和如何展示分离
2. 重用性高
3. 更高的可用性
4. 更易于测试

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

// 容器组件
export default class CommentList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      comments: []
    };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({
        comments: [
          { body: "react is very good", author: "facebook" },
          { body: "vue is very good", author: "youyuxi" }
        ]
      });
    }, 1000);
  }
  render() {
    return (
      <div>
        {this.state.comments.map((c, i) => (
          <Comment key={i} data={c} />
        ))}
      </div>
    );
  }
}

// 展示组件
function Comment({ data }) {
  return (
    <div>
      <p>{data.body}</p>
      <p>--- {data.author}</p>
    </div>
  );
}
```

## PureComponent

定制了shouldComponentUpdate后的Component (浅比较)

```
class Comp extends React.PureComponent {  
  
}
```

缺点是必须要用class形式

```
import shallowEqual from './shallowEqual'  
import Component from './Component'  
  
export default function PureComponent(props, context) {  
  Component.call(this, props, context)  
}  
  
PureComponent.prototype = Object.create(Component.prototype)  
PureComponent.prototype.constructor = PureComponent  
PureComponent.prototype.isPureReactComponent = true  
PureComponent.prototype.shouldComponentUpdate = shallowCompare  
  
function shallowCompare(nextProps, nextState) {  
  return !shallowEqual(this.props, nextProps) ||  
    !shallowEqual(this.state, nextState)  
}
```

```

export default function shallowEqual(objA, objB) {
  if (objA === objB) {
    return true
  }

  if (typeof objA !== 'object' || objA === null || typeof objB !== 'object' || objB === null) {
    return false
  }

  var keysA = Object.keys(objA)
  var keysB = Object.keys(objB)

  if (keysA.length !== keysB.length) {
    return false
  }

  // Test for A's keys different from B.
  for (var i = 0; i < keysA.length; i++) {
    if (!objB.hasOwnProperty(keysA[i]) || objA[keysA[i]] !== objB[keysA[i]]) {
      return false
    }
  }

  return true
}

```

## React.memo

React v16.6.0 之后的版本，可以使用一个新功能 `React.memo` 来完美实现 React 组件，让函数式的组件，也有了 `PureComponent` 的功能

```

const Joke = React.memo(() => (
  <div>
    {this.props.value || 'loading...'}
  </div>
));

```

## 高阶组件

提高组件复用率，首先想到的就是抽离相同逻辑，在 React 里就有了 HOC (Higher-Order Components) 的概念

高阶组件也是一个组件，但是他返回另外一个组件，产生新的组件可以对属性进行包装，也可以重写部分生命周期

```

const withKaikeba = (Component) => {
  const NewComponent = (props) => {
    return <Component {...props} name="开课吧高阶组件" />;
  };
  return NewComponent;
};

```

上面withKaikeba组件，其实就是代理了Component，只是多传递了一个name参数

## 高阶链式调用

高阶组件最巧妙的一点，是可以链式调用。

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

const withKaikeba = (Component) => {

  const NewComponent = (props) => {
    return <Component {...props} name="开课吧高阶组件" />;
  };
  return NewComponent;
};

const withLog = Component=>{
  class NewComponent extends React.Component{
    render(){
      return <Component {...this.props} />;
    }
    componentDidMount(){
      console.log('didMount',this.props)
    }
  }
  return NewComponent
}

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>hi,{this.props.name}</h2>
        <Button type="primary">Button</Button>
      </div>
    )
  }
}

export default withKaikeba(withLog(App))
```

## 高阶组件装饰器写法

这种链式写法略显蛋疼，逻辑比较绕，ES7中有一个优秀的语法—装饰器，专门用啦!处理这种问题

```
npm install --save-dev babel-plugin-transform-decorators-legacy
```

```
const { injectBabelPlugin } = require('react-app-rewired')
```

```

module.exports = function override(config) {
  config = injectBabelPlugin(
    ['import', { libraryName: 'antd', libraryDirectory: 'es', style: 'css' }],
    config,
  )

  config = injectBabelPlugin(
    ['@babel/plugin-proposal-decorators', { "legacy": true }],
    config,
  )

  return config
}

```

```

import React, { Component } from 'react'
import { Button } from 'antd'

const withKaikeba = (Component) => {

  const NewComponent = (props) => {
    return <Component {...props} name="开课吧高阶组件" />;
  };
  return NewComponent;
};

const withLog = Component=>{
  class NewComponent extends React.Component{
    render(){
      return <Component {...this.props} />;
    }
    componentDidMount(){
      console.log(Component.name, 'didMount', this.props)
    }
  }
  return NewComponent
}

@withKaikeba
@withLog
class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>hi,{this.props.name}</h2>
        <Button type="primary">Button</Button>
      </div>
    )
  }
}

```

```
export default App
```

## 组件通信--上下文

vuejs的provide&inject模式的来源---context

这种模式下有两个角色，一个是Provider 一个是Consumer，Provider在外城的组件，内部需要数据的，就用Consumer来读取

### 老版本的context

1. getChildContext用来返回数据
2. 定义 childContextTypes声明

### 新版本上下文

```
const FormContext = React.createContext()
const FormProvider = FormContext.Provider
const FormConsumer = FormContext.Consumer

let store = {
  name: '开课吧',
  sayHi() {
    console.log(this.name)
  }
}

let withForm = Component=>{
  const NewComponent = (props) => {
    return <FormProvider value={store}>
      <Component {...props} />
    </FormProvider>
  };
  return NewComponent;
}

@withForm
class App extends Component {
  render() {
    return <FormConsumer>
      {
        store=>{
          return <Button onClick={()=>store.sayHi()}>
            {store.name}
          </Button>
        }
      }
    </FormConsumer>
  }
}
```



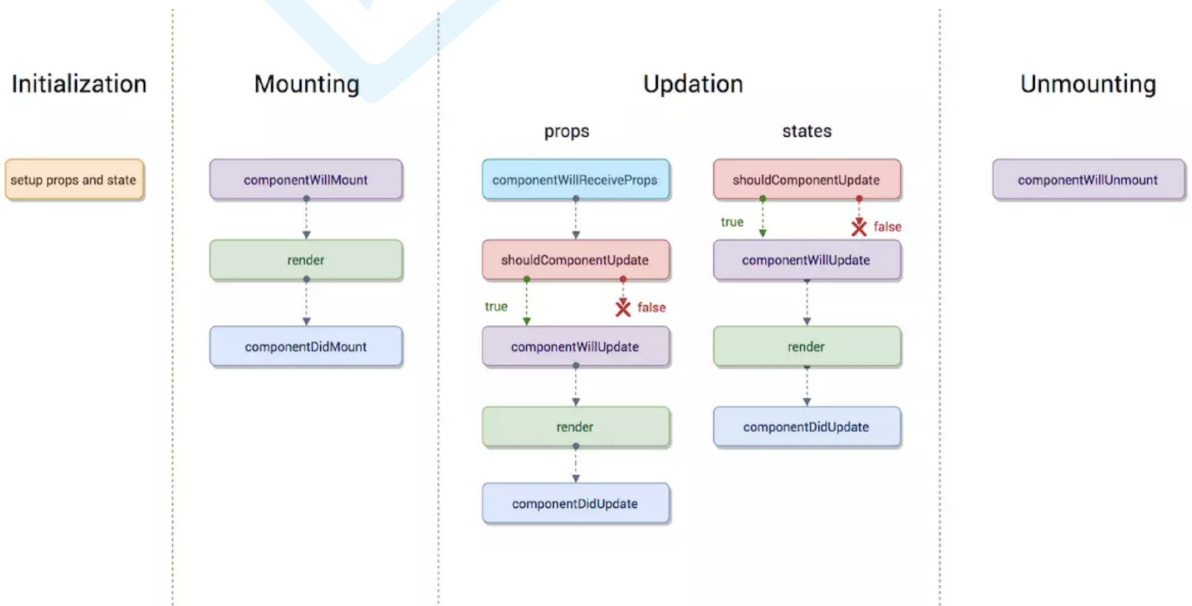
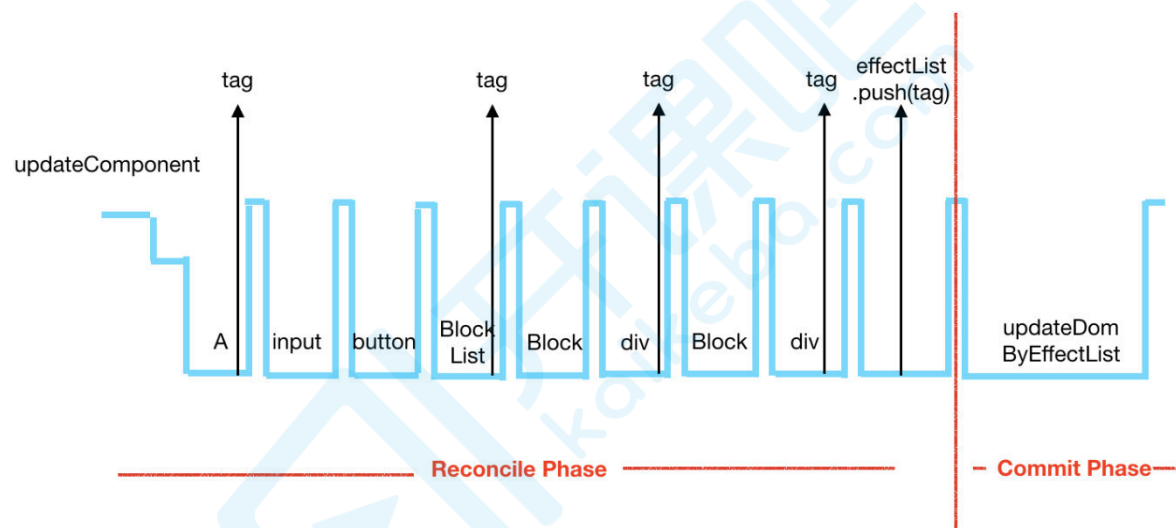
# React未来

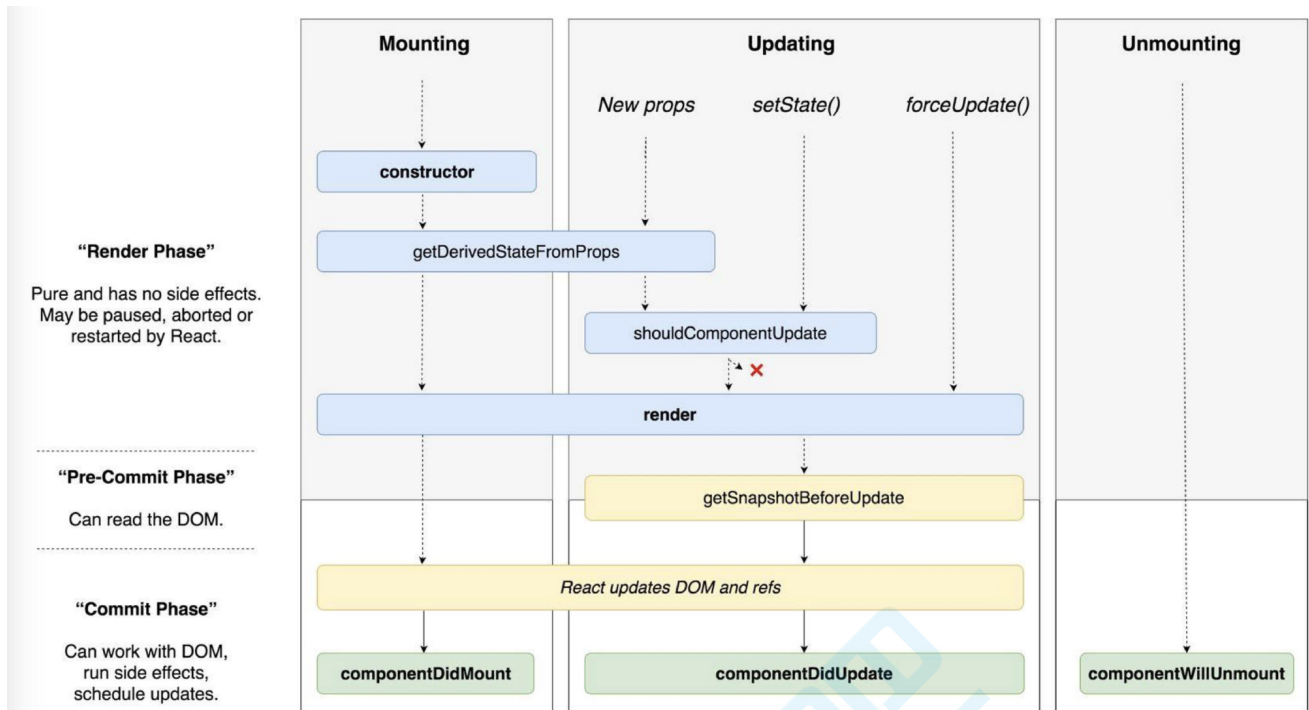
## Fiber

同步渲染 vs 异步渲染

React的新引擎Fiber的关键特性如下：

- 增量渲染（把渲染任务拆分成块，匀到多帧）
- 更新时能够暂停，终止，复用渲染任务
- 给不同类型的更新赋予优先级
- 并发方面新的基础能力





## Suspense

用同步的代码来实现异步操作

常见的异步操作代码

```
class Foo extends React.Component {
  state = {
    data: null
  }

  render() {
    if (!this.state.data) {
      return null;
    } else {
      return <div>this.state.data</div>;
    }
  }

  componentDidMount() {
    callAPI().then(result => {
      this.setState({data: result});
    });
  }
}
```

1. 组件必须要有自己的 `state` 和 `componentDidMount` 函数实现，也就不可能做成纯函数形式的组件。
2. 需要两次渲染过程，第一次是 `mount` 引发的渲染，由 `componentDidMount` 触发 AJAX 然后修改 `state`，然后第二次渲染才真的渲染出内容。

3. 代码啰嗦，十分啰嗦。

## 拥抱Suspense

```
const Foo = () => {  
  const data = createFetcher(callAJAX).read();  
  return <div>{data}</div>;  
}
```

createFetcher还没有正式发布

```
import React, {Suspense} from 'react';  
  
import {unstable_createResource as createResource} from 'react-cache';  
  
const getName = () => new Promise((resolve) => {  
  setTimeout(() => {  
    resolve('开课吧');  
  }, 1000);  
})  
  
const resource = createResource(getName);  
  
const Greeting = () => {  
  return <div>hello {resource.read()}</div>  
};  
  
const SuspenseDemo = () => {  
  return (  
    <Suspense fallback={<div>loading...</div>} >  
      <Greeting />  
    </Suspense>  
  );  
};
```

变革：函数组件可以做数据的获取，扩展了FP在react中的应用

## Hooks

这一节我们来介绍 Hooks，React v16.7.0-alpha 中第一次引入了 [Hooks](#) 的概念，因为这是一个 alpha 版本，不算正式发布，所以，将来正式发布时 API 可能会有变化

### useState

Hooks的目的,是开发者可以完全抛弃class,每天早晨起床,拥抱函数式,提供一个叫 `useState` 的方法,它开启了一扇新的定义 state 的门,对应 Counter 的代码可以这么写:

```
import { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <div>{count}</div>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

还可以设置多个

```
const [foo, updateFoo] = useState('foo');
```

## useEffect

除了 `useState`, React 还提供 `useEffect`, 用于支持组件中增加副作用的支持。对应class写法中的生命周期

```
componentDidMount() {
  document.title = `开课吧: ${this.state.count}`;
}

componentDidUpdate() {
  document.title = `开课吧: ${this.state.count}`;
}
```

```
import { useState, useEffect } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `开课吧: ${count}`;
  });

  return (
    <div>
      <div>{count}</div>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

```
};
```

`useEffect` 的参数是一个函数，组件每次渲染之后，都会调用这个函数参数，这样就达到了 `componentDidMount` 和 `componentDidUpdate` 一样的效果。

Hooks会大大减少react的代码

## 回顾

---

