创建React项目

npx create-react-app my-app

cd my-app

npm start

Jsx:

const element = <h1>Hello, world!</h1>;

**在 JSX 中嵌入表达式**

1.嵌入变量：

const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;

2. 嵌入方法：

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1>

Hello, {formatName(user)}! </h1>

);

### JSX 也是一个表达式

在编译之后，JSX 表达式会被转为普通 JavaScript 函数调用，并且对其取值后得到 JavaScript 对象。

也就是说，你可以在 if 语句和 for 循环的代码块中使用 JSX，将 JSX 赋值给变量，把 JSX 当作参数传入，以及从函数中返回 JSX：

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>; }

return <h1>Hello, Stranger.</h1>;

}

### JSX 特定属性

使用引号，来将属性值指定为字符串字面量：

const element = <div tabIndex="0"></div>;

也可以使用大括号，来在属性值中插入一个 JavaScript 表达式：

const element = <img src={user.avatarUrl}></img>;

在属性中嵌入 JavaScript 表达式时，不要在大括号外面加上引号。你应该仅使用引号（对于字符串值）或大括号（对于表达式）中的一个，对于同一属性不能同时使用这两种符号。

假如一个标签里面没有内容，你可以使用 /> 来闭合标签

const element = <img src={user.avatarUrl} />;

React Dom在渲染中会自动进行转义，不需要担心xss注入攻击。

Babel 会把 JSX 转译成一个名为 React.createElement() 函数调用。

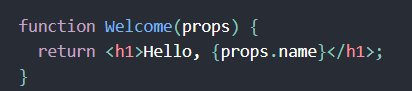


根节点下的所有节点都会被 ReactDom管理。 ReactDom.render渲染。

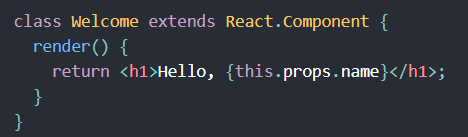
**组件 & Props**

## 函数组件与 class 组件

定义组件最简单的方式就是编写 JavaScript 函数：

该函数是一个有效的 React 组件，因为它接收唯一带有数据的 “props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。

你同时还可以使用 [ES6 的 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes" \t "_blank) 来定义组件：



以上2个是等效的。

React 元素也可以是用户自定义的组件：



当 React 元素为用户自定义组件时，它会将 JSX 所接收的属性（attributes）以及子组件（children）转换为单个对象传递给组件，这个对象被称之为 “props”。

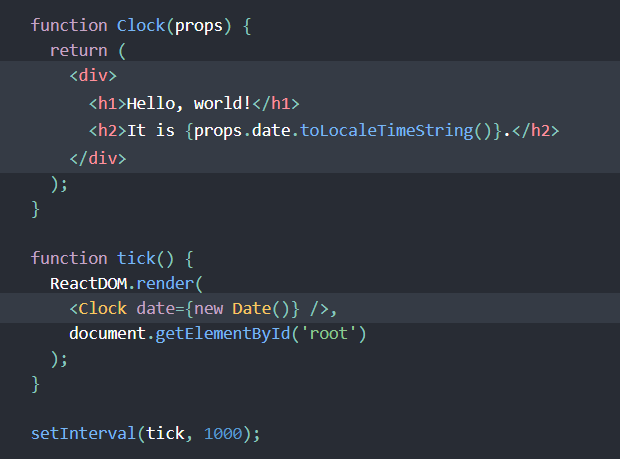


## Props 的只读性

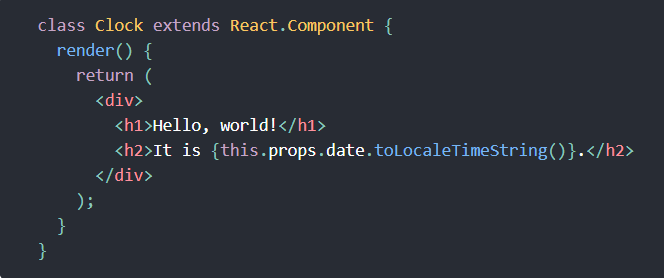
组件无论是使用[函数声明还是通过 class 声明](https://zh-hans.reactjs.org/docs/components-and-props.html" \l "function-and-class-components)，都决不能修改自身的 props。这样的函数被称为[“纯函数”](https://en.wikipedia.org/wiki/Pure_function" \t "_blank)，因为该函数不会尝试更改入参，且多次调用下相同的入参始终返回相同的结果

**所有 React 组件都必须像纯函数一样保护它们的 props 不被更改**

时钟实例：



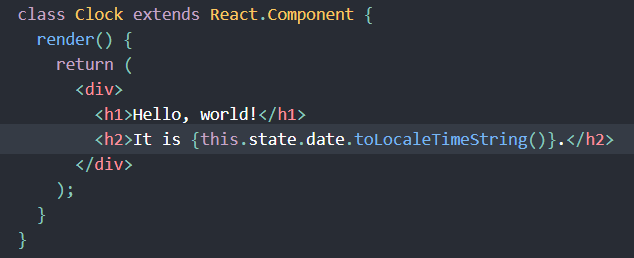
Class 组件：



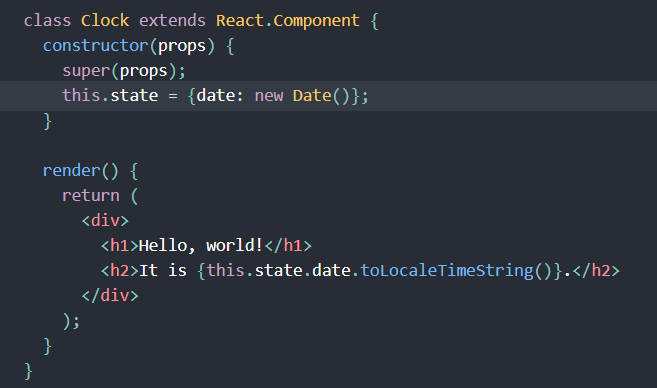
## 向 class 组件中添加局部的 state

我们通过以下三步将 date 从 props 移动到 state 中：

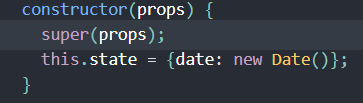
1. 把 render() 方法中的 this.props.date 替换成 this.state.date ：



1. 添加一个 [class 构造函数](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes" \l "Constructor" \t "_blank)，然后在该函数中为 this.state 赋初值：

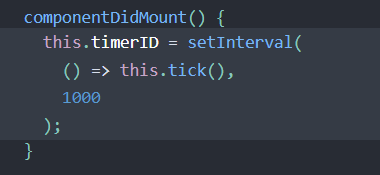


通过以下方式将 props 传递到父类的构造函数中：

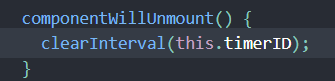


**生命周期：**

componentDidMount() 方法会在组件已经被渲染到 DOM 中后运行。



componentWillUnmount()方法会在组件销毁时运行



## 正确地使用 State

1．不要直接修改state

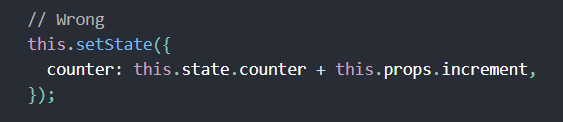


### State 的更新可能是异步的

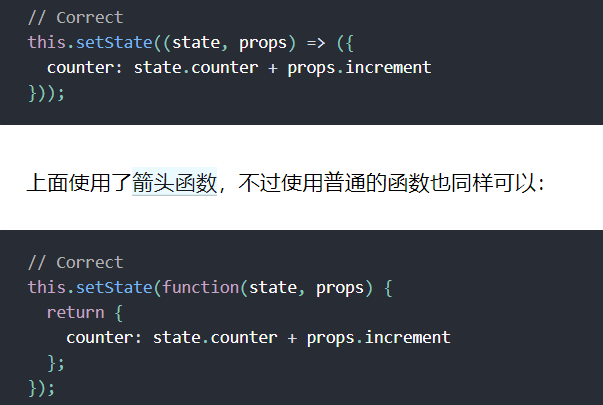
出于性能考虑，React 可能会把多个 setState() 调用合并成一个调用。

因为 this.props 和 this.state 可能会异步更新，所以你不要依赖他们的值来更新下一个状态。

例如，此代码可能会无法更新计数器：



要解决这个问题，可以让 setState() 接收一个函数而不是一个对象。这个函数用上一个 state 作为第一个参数，将此次更新被应用时的 props 做为第二个参数：

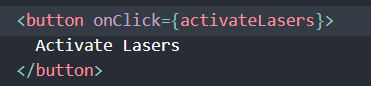


### State 的更新会被合并

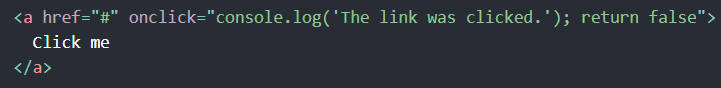
当你调用 setState() 的时候，React 会把你提供的对象合并到当前的 state。

**事件处理：**

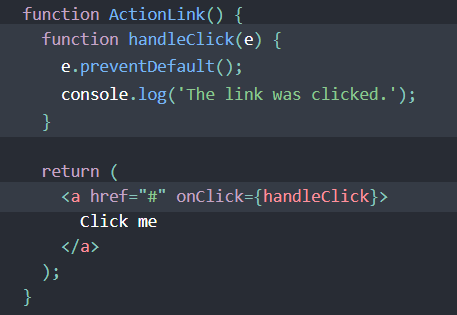
* React 事件的命名采用小驼峰式（camelCase），而不是纯小写。
* 使用 JSX 语法时你需要传入一个函数作为事件处理函数，而不是一个字符串。



在 React 中另一个不同点是你不能通过返回 false 的方式阻止默认行为。你必须显式的使用 preventDefault 。例如，传统的 HTML 中阻止链接默认打开一个新页面，你可以这样写：

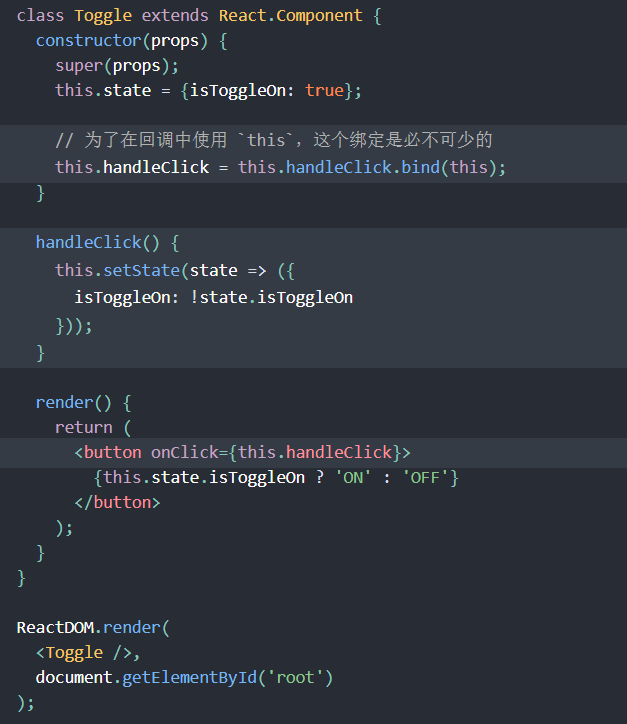


React中：



使用 React 时，你一般不需要使用 addEventListener 为已创建的 DOM 元素添加监听器。事实上，你只需要在该元素初始渲染的时候添加监听器即可。

当你使用 [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes" \t "_blank) 语法定义一个组件的时候，通常的做法是将事件处理函数声明为 class 中的方法。例如，下面的 Toggle 组件会渲染一个让用户切换开关状态的按钮：



如果觉得使用 bind 很麻烦，这里有两种方式可以解决。如果你正在使用实验性的 [public class fields 语法](https://babeljs.io/docs/plugins/transform-class-properties/" \t "_blank)，你可以使用 class fields 正确的绑定回调函数：



如果你没有使用 class fields 语法，你可以在回调中使用[箭头函数](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions" \t "_blank)：



此语法问题在于每次渲染 LoggingButton 时都会创建不同的回调函数。在大多数情况下，这没什么问题，但如果该回调函数作为 prop 传入子组件时，这些组件可能会进行额外的重新渲染。我们通常建议在构造器中绑定或使用 class fields 语法来避免这类性能问题。

## 向事件处理程序传递参数

在循环中，通常我们会为事件处理函数传递额外的参数。例如，若 id 是你要删除那一行的 ID，以下两种方式都可以向事件处理函数传递参数

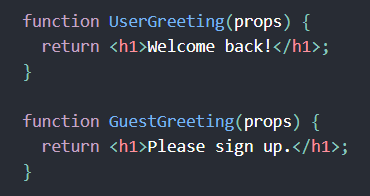


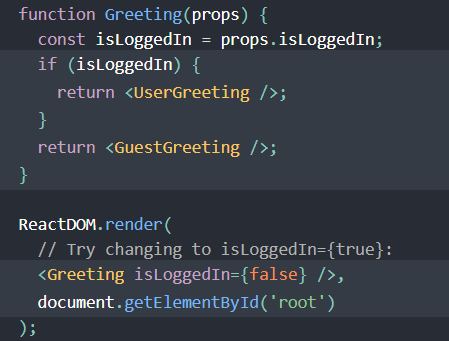
上述两种方式是等价的，分别通过[箭头函数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions" \t "_blank)和 [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind" \t "_blank) 来实现。

在这两种情况下，React 的事件对象 e 会被作为第二个参数传递。如果通过箭头函数的方式，事件对象必须显式的进行传递，而通过 bind 的方式，事件对象以及更多的参数将会被隐式的进行传递

# 条件渲染

React 中的条件渲染和 JavaScript 中的一样，使用 JavaScript 运算符 [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else" \t "_blank) 或者[条件运算符](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator" \t "_blank)去创建元素来表现当前的状态，然后让 React 根据它们来更新 UI。

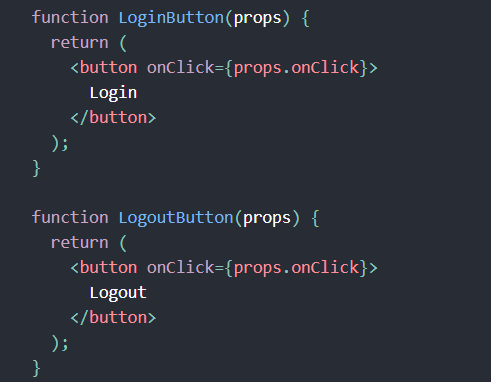




### 元素变量

你可以使用变量来储存元素。 它可以帮助你有条件地渲染组件的一部分，而其他的渲染部分并不会因此而改变。

观察这两个组件，它们分别代表了注销和登录按钮：

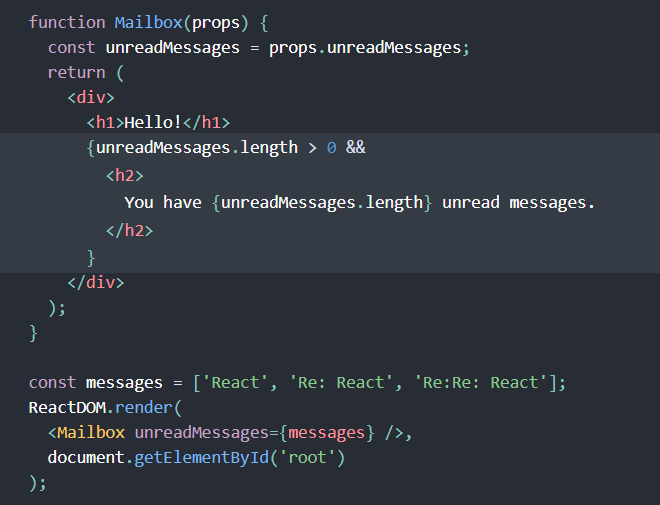


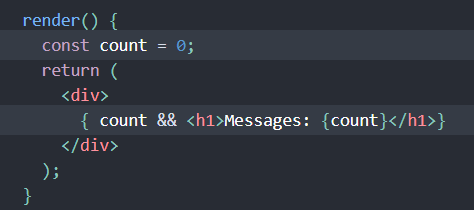
在下面的示例中，我们将创建一个名叫 LoginControl 的[有状态的组件](https://zh-hans.reactjs.org/docs/state-and-lifecycle.html" \l "adding-local-state-to-a-class)。

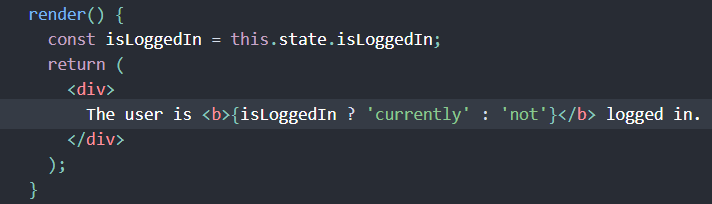
它将根据当前的状态来渲染 <LoginButton /> 或者 <LogoutButton />。同时它还会渲染上一个示例中的 <Greeting />。

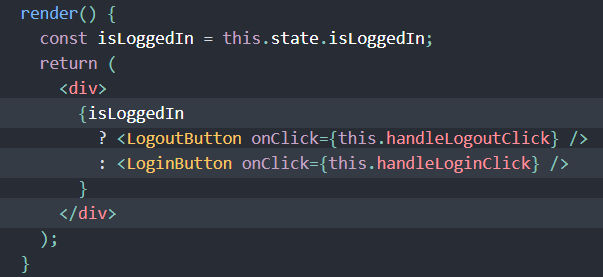


通过花括号包裹代码，你可以[在 JSX 中嵌入表达式](https://zh-hans.reactjs.org/docs/introducing-jsx.html" \l "embedding-expressions-in-jsx)。这也包括 JavaScript 中的逻辑与 (&&) 运算符。它可以很方便地进行元素的条件渲染：



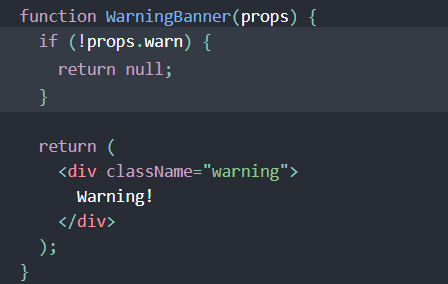




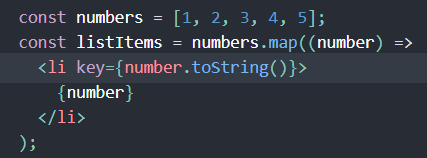


### 阻止组件渲染

在极少数情况下，你可能希望能隐藏组件，即使它已经被其他组件渲染。若要完成此操作，你可以让 render 方法直接返回 null，而不进行任何渲染。



key 帮助 React 识别哪些元素改变了，比如被添加或删除。因此你应当给数组中的每一个元素赋予一个确定的标识。



一个元素的 key 最好是这个元素在列表中拥有的一个独一无二的字符串。Key只是在兄弟节点之间必须唯一。

**表单：**

## 受控组件

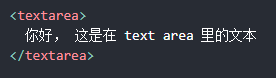
在 HTML 中，表单元素（如<input>、 <textarea> 和 <select>）通常自己维护 state，并根据用户输入进行更新。而在 React 中，可变状态（mutable state）通常保存在组件的 state 属性中，并且只能通过使用 [setState()](https://zh-hans.reactjs.org/docs/react-component.html" \l "setstate)来更新。我们可以把两者结合起来，使 React 的 state 成为“唯一数据源”。渲染表单的 React 组件还控制着用户输入过程中表单发生的操作。被 React 以这种方式控制取值的表单输入元素就叫做“受控组件”。

示例：

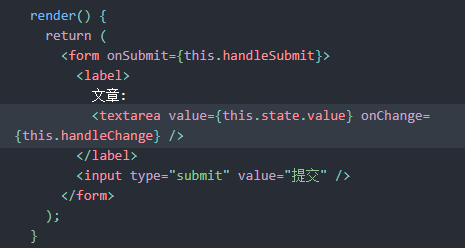


**Textarea标签:**

HTML 中, <textarea> 元素通过其子元素定义其文本,



而在 React 中，<textarea> 使用 value 属性代替。这样，可以使得使用 <textarea> 的表单和使用单行 input 的表单非常类似



**select标签:**

由于 selected 属性的缘故，椰子选项默认被选中。React 并不会使用 selected 属性，而是在根 select 标签上使用 value 属性.

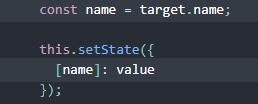


多个:



## 文件 input 标签

因为它的 value 只读，所以它是 React 中的一个**非受控**组件

 name为表单名单

## 受控输入空值

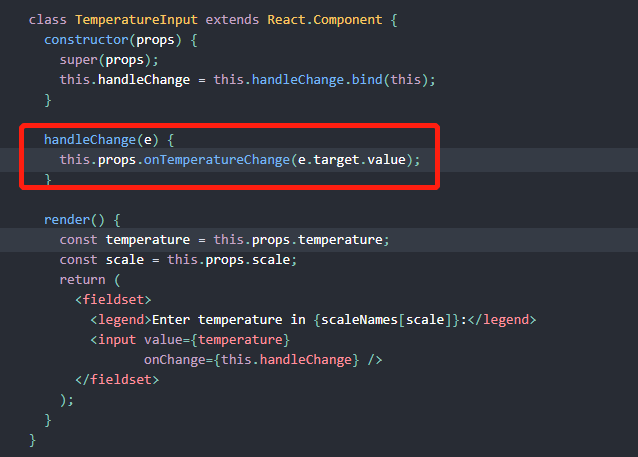
在[受控组件](https://zh-hans.reactjs.org/docs/forms.html" \l "controlled-components)上指定 value 的 prop 会阻止用户更改输入。如果你指定了 value，但输入仍可编辑，则可能是你意外地将value 设置为 undefined 或 null。

# 状态提升

通常，多个组件需要反映相同的变化数据，这时我们建议将共享状态提升到最近的共同父组件中去。让我们看看它是如何运作的。



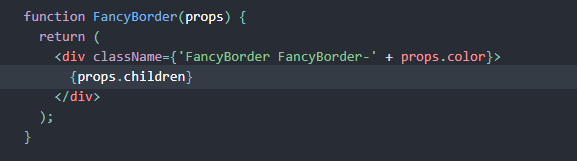
将方法以props的形式传给子组件，在子组件中调用，如下图



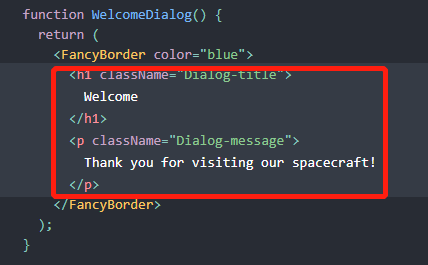
# 组合 vs 继承

React 有十分强大的组合模式。我们推荐使用组合而非继承来实现组件间的代码重用。

我们建议这些组件使用一个特殊的 children prop 来将他们的子组件传递到渲染结果中:



这使得别的组件可以通过 JSX 嵌套，将任意组件作为子组件传递给它们。

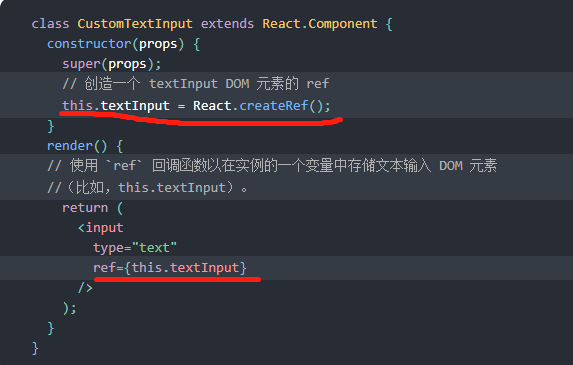


<FancyBorder> JSX 标签中的所有内容都会作为一个 children prop 传递给 FancyBorder 组件。因为 FancyBorder 将 {props.children} 渲染在一个 <div> 中，被传递的这些子组件最终都会出现在输出结果中。

组件可以接受任意 props，包括基本数据类型，React 元素以及函数。

如果你想要在组件间复用非 UI 的功能，我们建议将其提取为一个单独的 JavaScript 模块，如函数、对象或者类。组件可以直接引入（import）而无需通过 extend 继承它们。

创建ref来引用表单:



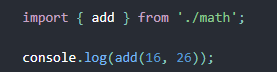
然后我们就可以在需要时于其他地方把焦点设置在这个组件上：



有时，父组件需要把焦点设置在其子组件的一个元素上。我们可以通过在子组件上设置一个特殊的 prop 来[对父组件暴露 DOM refs](https://zh-hans.reactjs.org/docs/refs-and-the-dom.html" \l "exposing-dom-refs-to-parent-components) 从而把父组件的 ref 传向子节点的 DOM 节点。



**import() 语法:**



React.lazy 函数能让你像渲染常规组件一样处理动态引入（的组件）。



## 命名导出（Named Exports）

React.lazy 目前只支持默认导出（default exports）。如果你想被引入的模块使用命名导出（named exports），你可以创建一个中间模块，来重新导出为默认模块。这能保证 tree shaking 不会出错，并且不必引入不需要的组件。

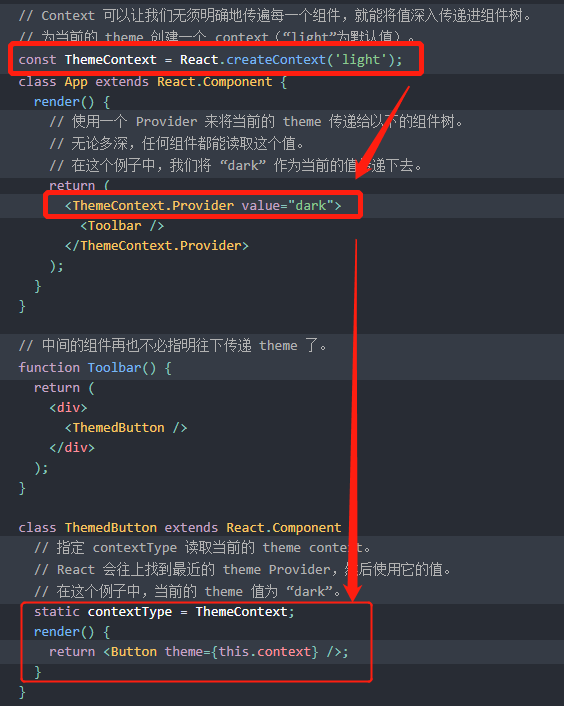


# Context

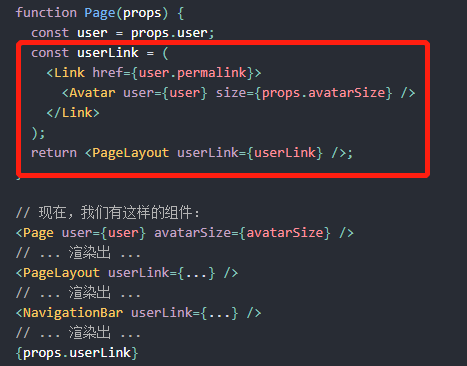
Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。

组件之间共享变量，可以通过context而不是一级一级向下传递。

缺点：复用性变差



一种**无需 context** 的解决方案是[将 Avatar 组件自身传递下去](https://zh-hans.reactjs.org/docs/composition-vs-inheritance.html" \l "containment)，因而中间组件无需知道 user 或者 avatarSize 等 props。

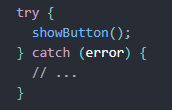


**错误边界：**

如果一个 class 组件中定义了 [static getDerivedStateFromError()](https://zh-hans.reactjs.org/docs/react-component.html" \l "static-getderivedstatefromerror) 或 [componentDidCatch()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentdidcatch) 这两个生命周期方法中的任意一个（或两个）时，那么它就变成一个错误边界。当抛出错误后，请使用 static getDerivedStateFromError() 渲染备用 UI ，使用 componentDidCatch() 打印错误信息。



try / catch 很棒但它仅能用于命令式代码（imperative code）：

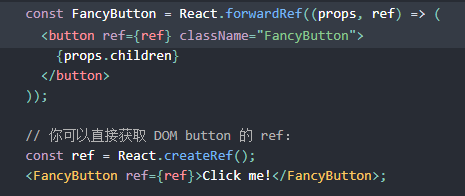


# Refs 转发

Ref 转发是一项将 [ref](https://zh-hans.reactjs.org/docs/refs-and-the-dom.html) 自动地通过组件传递到其一子组件的技巧。对于大多数应用中的组件来说，这通常不是必需的。但其对某些组件，尤其是可重用的组件库是很有用的。最常见的案例如下所述

**Ref 转发是一个可选特性，其允许某些组件接收 ref，并将其向下传递（换句话说，“转发”它）给子组件。**

在下面的示例中，FancyButton 使用 React.forwardRef 来获取传递给它的 ref，然后转发到它渲染的 DOM button：



这样，使用 FancyButton 的组件可以获取底层 DOM 节点 button 的 ref ，并在必要时访问，就像其直接使用 DOM button 一样。

# Fragments

React 中的一个常见模式是一个组件返回多个元素。Fragments 允许你将子列表分组，而无需向 DOM 添加额外节点。



# 高阶组件

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

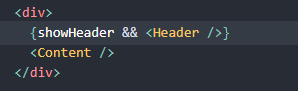
具体而言，**高阶组件是参数为组件，返回值为新组件的函数。**



如果你有一个自定义组件，你可以把回调函数作为 props.children 进行传递

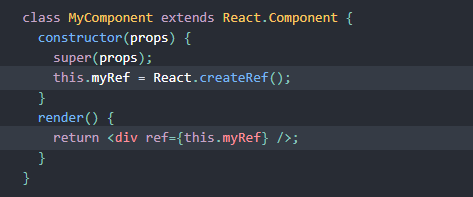


false, null, undefined, and true 是合法的子元素。但它们并不会被渲染。



### 创建 Refs

Refs 是使用 React.createRef() 创建的，并通过 ref 属性附加到 React 元素。在构造组件时，通常将 Refs 分配给实例属性，以便可以在整个组件中引用它们。



### 访问 Refs

当 ref 被传递给 render 中的元素时，对该节点的引用可以在 ref 的 current 属性中被访问。



ref 的值根据节点的类型而有所不同：

* 当 ref 属性用于 HTML 元素时，构造函数中使用 React.createRef() 创建的 ref 接收底层 DOM 元素作为其 current 属性。
* 当 ref 属性用于自定义 class 组件时，ref 对象接收组件的挂载实例作为其 current 属性。
* **你不能在函数组件上使用 ref 属性**，因为他们没有实例。

React 会在组件挂载时给 current 属性传入 DOM 元素，并在组件卸载时传入 null 值。ref 会在 componentDidMount 或 componentDidUpdate 生命周期钩子触发前更新。

Function Component 功能函数调用父类事件

const addTab = (e, props, title, id)=> {

    props.add(title, id); //调用父类的add函数

};

<Menu.Item key={item.id} onClick={(e) => addTab(e, props, item.name, item.id)} >{item.name}</Menu.Item>

<https://reactjs.org/docs/faq-functions.html>

传递事件到子组件：



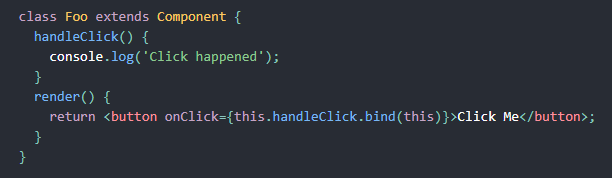
绑定一个方法到组件实例：



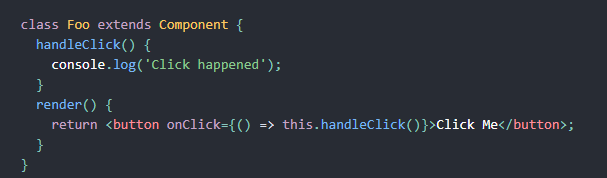
箭头函数绑定：



bind 方法

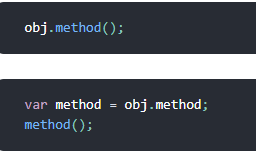


箭头函数在render里：

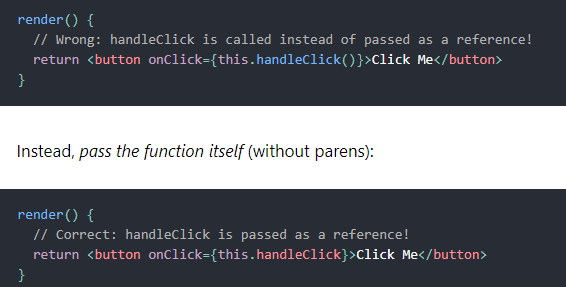


理解bind

在javascript中以下2个代码是不相等的。



Binding方法使第二个代码跟第一个表达的一样。在react中,你要绑定一个方法传递给其它组件,如<button onClick={this.handleClick}> 传递this.handleClick 。



怎样传参数给方法

* 1. 你可以使用箭头函数

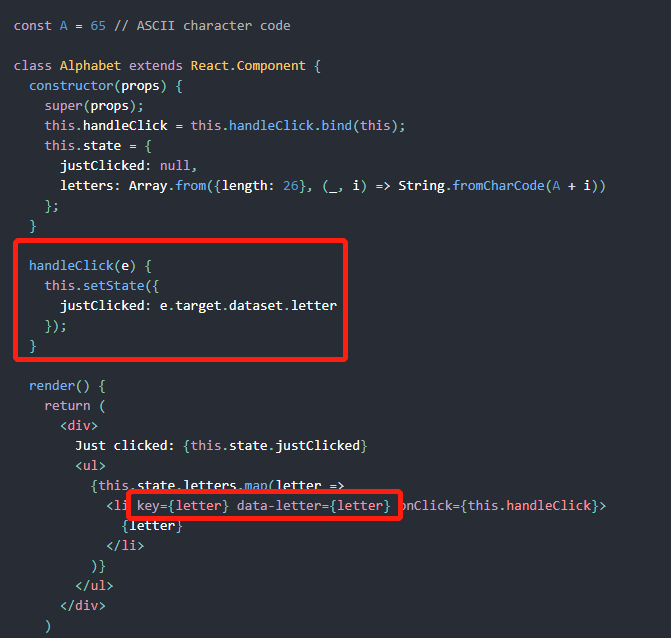


它等于 .bind





也可以使用dom属性传参



reactjs绑定事件传参，有两种方式，如下:

1. <button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
2. <button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
3. 上面两行是等价的，只是使用箭头函数和Function.prototype.bind的区别。
4. 在这两种情况下，e表示React事件的参数将作为ID之后的第二个参数传递。使用箭头函数，我们必须显式传递它，但bind任何进一步的参数都会自动转发。

JSX循环处理

const elements = [] //..some array

const items = []

for (const [index, value] of elements.entries()) {

items.push(<Element key={index} />)

}

用数组包装,先把元素放到数组中再渲染出来

render() {

const elements = ['one', 'two', 'three'];

const items = []

for (const [index, value] of elements.entries()) {

items.push(<li key={index}>{value}</li>)

}

return (

<div>

{items}

</div>

)

}

直接在jsx中使用map

render: function() {

const elements = ['one', 'two', 'three'];

return (

<ul>

{elements.map((value, index) => {

return <li key={index}>{value}</li>

})}

</ul>

)

}

返回多元素：

Const Pets =()=>{

return [<Dog/><Cat/>];

}

<Fragment> <Dog/><Cat/></Fragment>

(let item in items), (let item of items)

REACT引入html

f you want to include static html in ReactJS. You need to use html-loader plugin if you are using webpack to serve your react code.

First of all you need to install html-loader plugin. Open your terminal and run following command:

npm install --save-dev html-loader

Then open your webpack configuration file and add following code:

{

modules: {

loaders: [

{ test: /\.html$/, loader: 'html-loader' }

]

}

}

Now, in your react code do following changes:

var htmlContent = require('path/to/html/file.html');

// or you can also write

// import htmlContent from 'path/to/html/file.html');

export default function MyComponent() {

return (

<div dangerouslySetInnerHTML={ {\_\_html: htmlContent} } />

);

}

That is it. Now you can use static html files to load your html files in react.

功能函数父子调用子类方法:

子组件需要使用 useImperativeHandle 暴露 ref 自定义的实例值给父组件。这个需要用 forwardRef 包裹着。

function children(props, ref) {

useImperativeHandle(ref, () => ({

hello: () => console.log('hello world!')

}))

return <h1>children</h1>

}

export default forwardRef(children);

父组件调用：

const p = ()=>{

const childrenRef = useRef(null);

const something = () => childrenRef.current.hello();

return (

< children ref={ childrenRef } />

)

}

还是可以查看一下实现方法的关键点： useRef, useImperativeHandle, forwardRef。其实是利用了 ref 不变的性质，将子组件的方法保存着，父组件可以调用。可以说，ref 作为父组件和子组件的一座由父组件到达子组件的桥梁（单向的，父->子）。

NRM 可以管理npm地址包配置

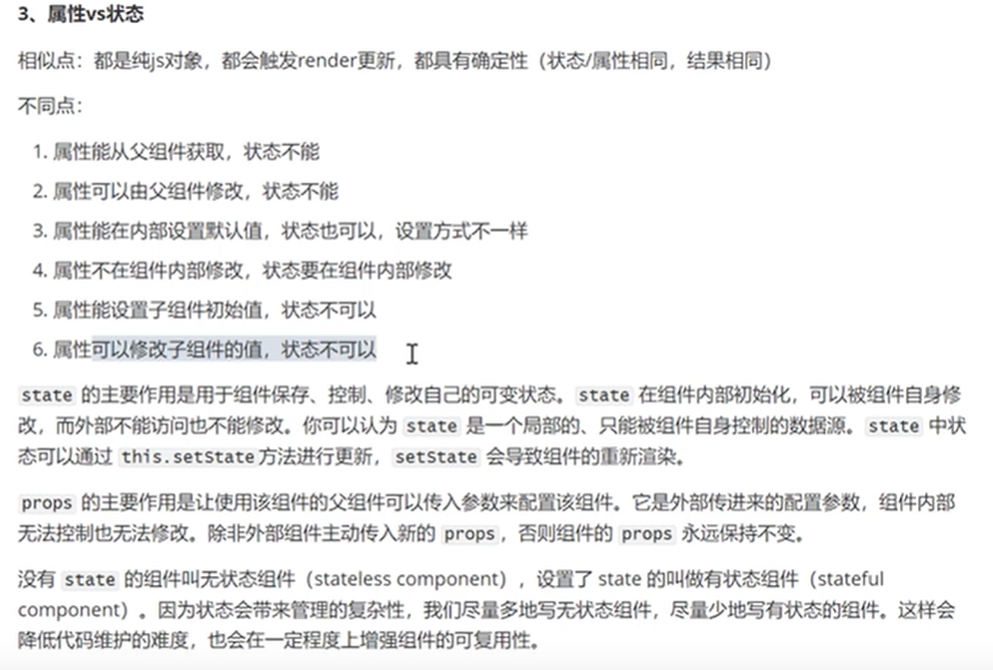






React并不会真正的绑定事件到每一个具体<>的元素上，而是采用事件代理的模式，绑定在根节点上？。







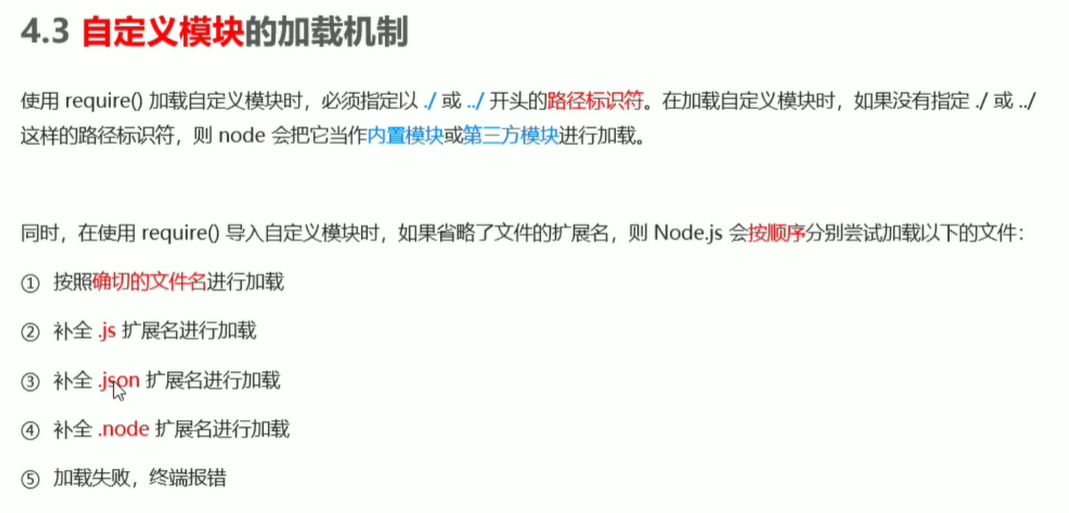
Ref的作用是获取实例，但是如果目标组件是一个自定义函数组件(Function Component),那么它是没有实例的，此时用ref去传递会报错，因为它无法获取到实例。React.forwardRef就是用来解决这个问题的。 React.forwardRef会创建一个React组件，这个组件能够将其接收的ref属性转发到自己的组件树。

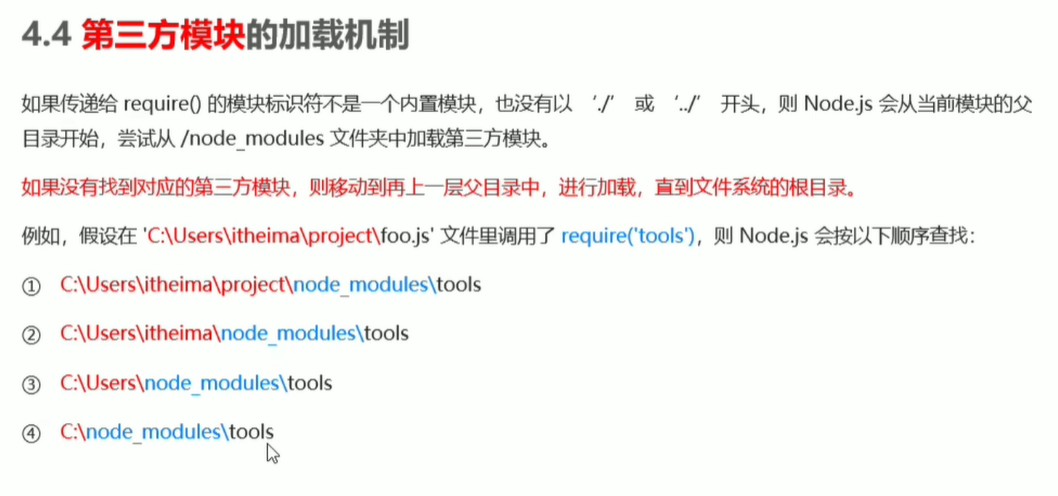
useCallback 和 useMemo的区别，useMemo会调用fn函数并返回其结果，而useCallback仅返回fn函数而不是调用它。 useCallback需要配合 React.memo使用。这是因为React.memo这个方法会对props做一个浅层比较，如果props没有发生改变，则不会重新渲染此组件。

**NodeJs**

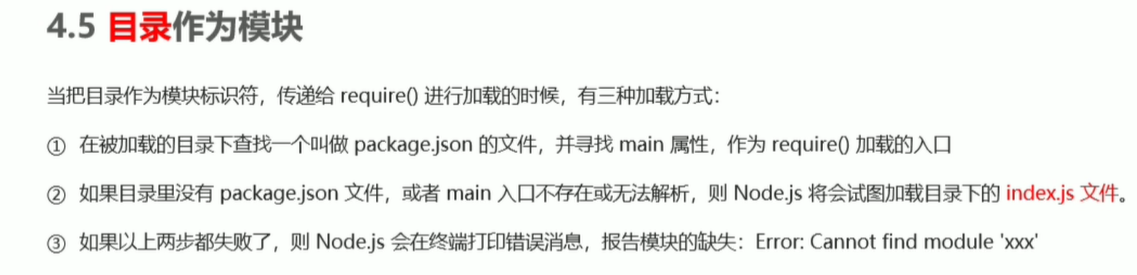
npm install nrm //nrm是用来对Npm环境进来配置的，如配置npm官方或淘宝Npm镜像

nodejs内置module.exports ={}





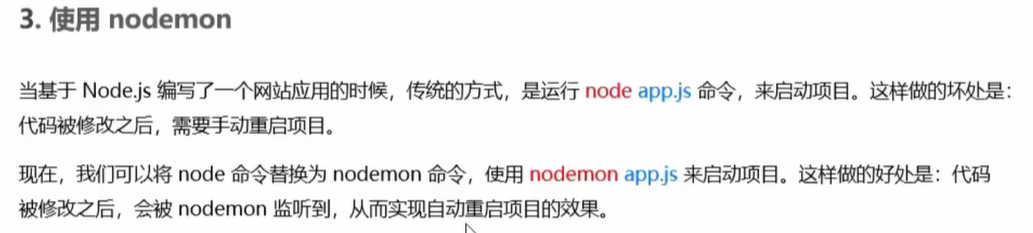
最后找不到就报错。



require(‘./目录’)

Json中只能使用双引号 {“A”:”1”} 规范

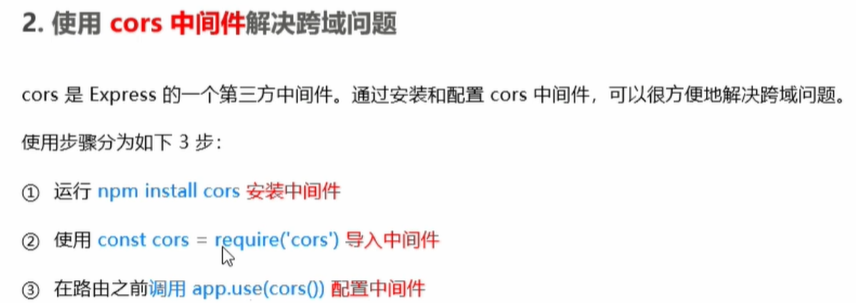




Express 跨域问题解决(cors跨域资源共享)

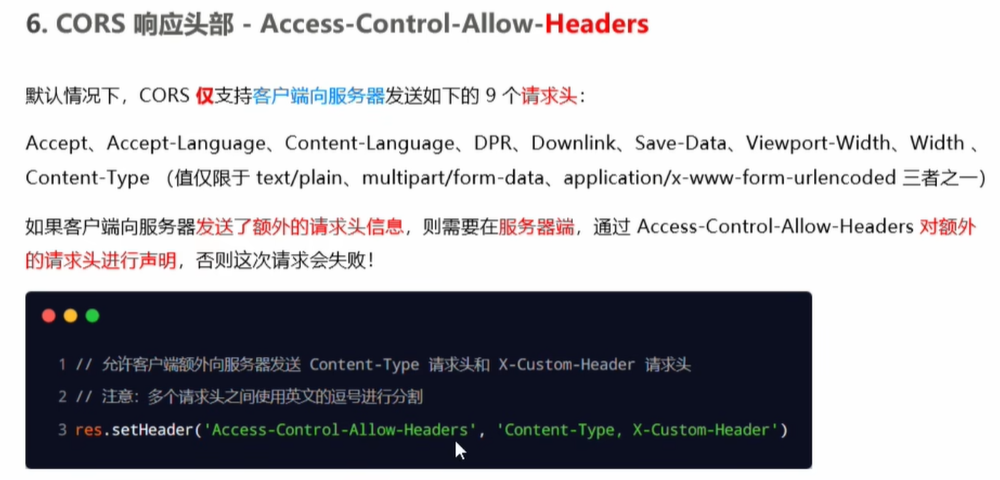
npm install cors

const cors = require(‘cors’); app.use(cors());

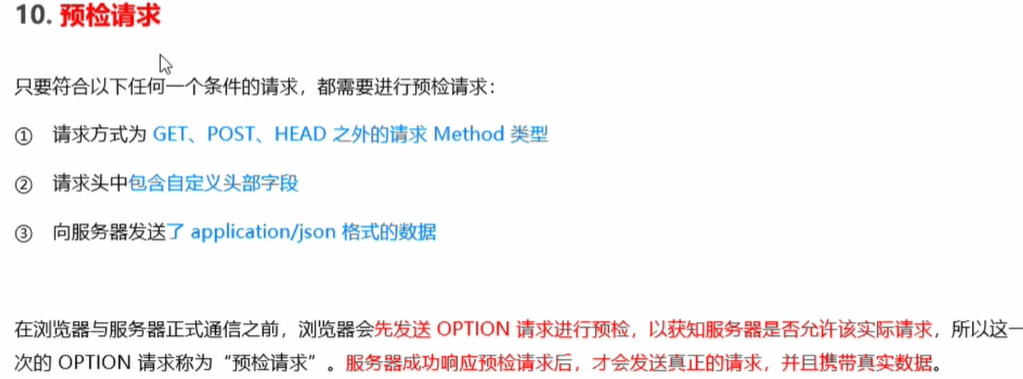




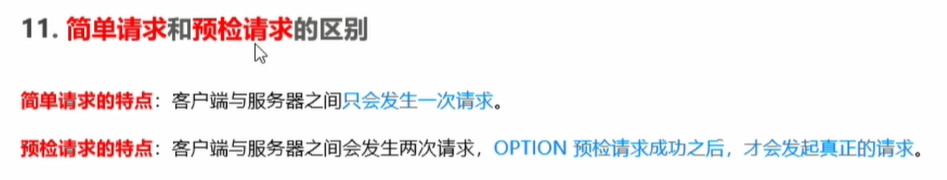




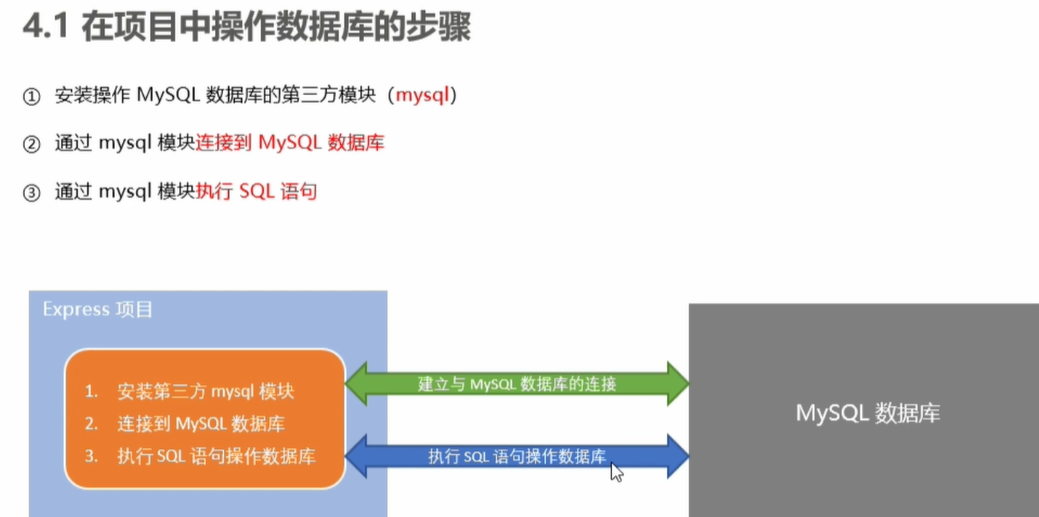


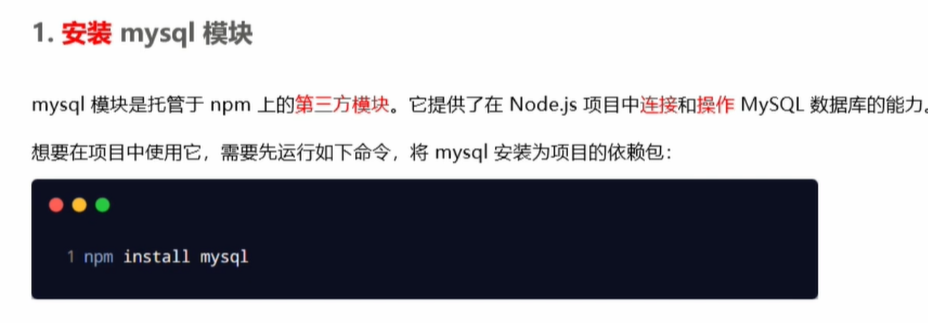


还有简单请求就是：get, post,put,head





















只要配置成功了express-jwt这个中间伯，就可以把解析出来的信息挂载在req中



全局的错误中间件

对密码进行加密处理：

导入包： bcryptjs

Npm install [bcryptjs@2.4.3](mailto:bcryptjs@2.4.3)





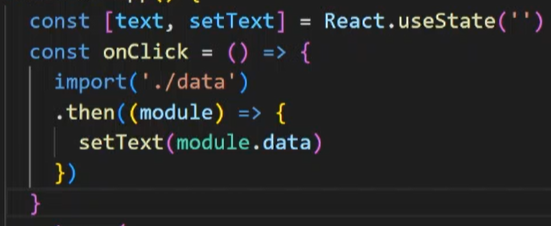


**Let maxwell = JSON.parse(jsonstring) as {name:string, age: number};**

Typescript中可以将函数返回值进行转化

Let x = abc() as string;

React异步加载：



如果export的是default,那么改成 .then((module)=>{ setText(module.default) }) };

js类中的私有变量加上 #,如 #size = 0; get size(){ return this.#size; }

Node导出 exports.a=function(){...}; 导入 const a = require("a");

Load事件是在所有文档的外部资源如图片都完全加载后触发。DOMContentLoaded是在HTML文档完全加载和解析后触发。

<script defer|asycn src=""/> defer 是文档全部加载完后执行（跟把js文件放在文档最下面一样）。asyn是同步执行，没有先后顺序。

Reflect, Proxy

页面中有iframe可能会有同源问题。同源跨域。

dataset <h2 id="title" data-name='age'/> document.querySelector('#title').dataset.name;

Worker对象； ESLint辅助发现代码中潜藏的缺陷和网络问题。

Prettier 用于以标准方式格式化javascript 代码；

Jest 一种编写javascript单元测试的一部式解决方案。

Object.definedProperty和Proxy经常用来实现对象的劫持

Object.definedProperty是vue2.x拦截对象属性变化的方法，再结合订阅-发布者模式和观察者模式实现对象属性的双向绑定更新

Proxy是vue3.0以后改进的双向绑定方案

Object.definedProperty

存在以下缺陷：

不能监听数组的变化

必须遍历对象的每个属性

必须深层遍历嵌套的对象

无法监听数组变化：vue无法对数组的变异方法（push、pop、shift、unshift、splice、sort、reserve）进行监听，只是对这些方法进行重载hack，达到监听变化的效果，这些重载需要一定的开销

必须遍历对象属性：vue对对象进行监听变化，需要使用Object.keys()搭配Object.definedProperty对对象的每个属性进行绑定，加大了开销

必须生成遍历对象: 针对多层嵌套的对象，则需要递归遍历进行绑定

Proxy

Proxy相对Object.definedProperty有以下优点:

Porxy监听的是整个对象而不是对象的某个属性

Proxy可以监听数组的变化

Proxy有多大13中的拦截方式，比Object.definedProperty功能更强大

Proxy结果返回的是一个新的对象，我们直接操作新的对象，而不是像Object.definedProperty遍历属性进行修改

Proxy有着更好的性能，浏览器厂商也在不对地进行维护和加强

但Proxy也着缺陷：

Proxy的对浏览器的兼容性不够，现阶段无法用polyfill进行兼容，这也是现在不推荐使用vue3.0进行重要项目开发的原因之一，需要等待后期功能的完善

淘宝适配包 flex.js

# Nodejs 学习笔记

1：**初始化项目**

$ npm init

$ npm install --save koa

--save 和 --save-dev 的区别.--save会将模块依赖写入dependencies节点，--save-dev会将模块依赖写入devDependencies节点.当运行npm install -production命令或者NODE\_ENV变量为production 时，安装dependencies下的依赖，不安装devDependencies下的依赖。

const Koa = require(‘koa’);

const app = new Koa();

app.use(async (ctx)=>{

ctx.body = ‘hello world’;

});

app.listen(4000);

app.use()是一个回调函数，它有2个参数，一个是ctx,g 一个是next()函数，可以理解为一个中间件，每个中间件以await next()函数为分界。

**路由：**

使用中间件 koa-router ,通过request对象的URL属性进行判断。

app.use(async (ctx)=>{

const url = ctx.request.url;

let content = ‘’;

switch(url){

case ‘/api/get/userinfo’:

content = ‘xxx’;

break;

}

});

安装koa-router

$ **npm install --save koa-router**

const Koa = require(‘koa’);

const app = new Koa();

**const Router = require(‘koa-router’);**

**const router = new Router();**

router.get(‘/api/get/userInfo’, async (ctx) => {

ctx.body = ‘200: xxxxxx’;

});

//加载路由中间件

app.use(**router.routes()**).use(async (ctx)=>{

ctx.body = ‘404:xxx’;

});

app.listen(4000, ()=>{.......});

**静态服务器 koa-static ：**

静态服务器功能可以利用Koa的中间件koa-static实现https://github.com/koajs/static进行了解。

$ npm install --save koa-static

const Koa = require(‘koa’);

const path = require(‘path’);

const static = require(‘koa-static’);

const app = new Koa();

//静态资源目录对于相对入口文件index.js的路径

const staticPath = ‘./static’;

app.use(**static(path.join(\_\_dirname, staticPath))**);

**如何实现一个静态服务器：**

不同的静态资源有不同的mimetype

const MIMES\_MAP = {

‘css’: ‘text/css’,

‘html’: ‘text/html’,

‘jpg’: ‘image/jpeg’

}

//解析类型资源

function parseMime(url){

let extName = path.extname(url);

extName = extName ? extName.slice(1) : ‘unknown’;

return MIMES\_MAP(extName);

}

let mime = parseMime(ctx.url)

if(mime) {

//输出静态资源内容

if(mime && mime.indexOf(‘image/’) >= 0){

ctx.type writeHead(200);

ctx.res.write(content, ‘binary’);

ctx.res.end();

}else{

//其他则输出文本

ctx.body = content;

}

**模板引擎 EJS：**

Koa中的模板功能实现需要用到中间件koa-views.

const Koa = require(‘koa’);

**const views = require(‘koa-views’);**

const path = require(‘path’);

const app = new Koa();

//加载模板引擎

app.use(views(path.join(\_\_dirname, ‘./view’), {

extension: ‘ejs’

}));

app.use(async (ctx)=>{

let title=’koa’;

await ctx.render(‘index’, {title,}

});

**Cookie 和 Session (koa-session):**

读取和设置cookie： ctx.cookies.get(name, [options]); ctx.cookies.set(name, value, [option]);

router.get(‘/setCookie’, async (ctx)=>{

ctx.cookies.set(...);

ctx.cookies.get(‘..’);

});

const sessionConfig = { key: xxx, maxAge: 86400000 /\*一天\*/, signed: false}

**app.use(session(sessionConfig, app));**

**const bodyParser = require(‘koa-bodyparser’)**

**app.use(router.routes);**

**//用来加载前端页面**

**router.get(‘/’, async (ctx)=>{**

**ctx.set({‘Content-Type’: ‘text/html’});**

**Ctx.body = fs.readFileSync(‘./index.html’);**

**});**

**const Redis = require(‘ioredis’);**

**文件上传：**

**文件上传需要一个中间件　koa-body 来实现。**

**const koaBody = require(‘koa-body’)**

**app.use(koaBody({**

**multipart: true,**

**Formidable: { maxFileSize: 200\*1024\*1024 //设置上传文件的限制，默认2M}**

**}));**

router.post(‘/upload’, async (ctx)=>{

//获取文件对象

**const file = ctx.request.files.file; //file为上传控制名字**

//读取文件内容

**const data = fs.readFilesSync(file.path);**

//保存到服务端

**fs.writeFileSync(path.join(\_\_dirname, file.name), data);**

ctx.body = {message: ‘success’}

});

**下载到客户端使用中间件 koa-send**

const send = require(‘koa-send’);

router.get(‘/download/:name’, async (ctx)=>{

//url参数

//ctx.request.query.name

const name = ctx.params.name;

const path = `${name}`;

ctx.attachment(path);

await send(ctx, path);

});

**Koa 解决同源策略的实质是校验请求头，提供了一个中间件 @koa/cors**

**const cors = require(‘@koa/cors’);**

**//加载cors中间件**

**app.use(cors({**

**origin: ‘\*’ //\*代表任何URL都可以进行跨域请求。**

**}));**

URL重写中间件 koa-rewrite

const rewrite = require(‘koa-rewrite’);

app.use(rewrite(‘A url’, ‘B url’));

app.use(router.routes());

//统一重写url， 根据正则表达式重写

app.use(rewrite(/^\/i(\w+)/, ‘/items/$1’);

**错误处理中间件koa-error**

**const error = require(‘koa-error’);**

app.use(error({

engine: ‘pug’,

Template: \_\_dirname + ‘/error.pug’

}));

**路由技巧：**

const Router = require(‘koa-router’)

const router = new Router();

router.get(‘/xxx’, async (ctx)=>{

ctx.body = ‘xxxxxx’;

});

app.use(router.routes());

路由分割：把所有路由按照类别进行划分，并分别维护在不同的文件里。

**//good.js**

const Router = require(‘koa-router’);

const router = new Router();

//设置货物路由前缀

router.prefix(‘/goods’);

route.get(‘/getInfo’, (ctx, next)=>{

ctx.body = “this is koa book”;

});

module.exports = router;

**//user.js**

const Router = require(‘koa-router’);

const router = new Router();

//设置用户路由前缀

router.prefix(‘/user);

route.get(‘/getInfo’, (ctx, next)=>{

ctx.body = “this is koa book”;

});

module.exports = router;

**合并中间件 koa-compose**

每个路由文件中都使用了一个路由前缀，这样方便分类。每个文件封装了不同类型的路由，接下来要做的就是对这些路由进行整合。使用到中间件koa-compose.

//routers/index.js

const compose = require(‘koa-compose’);

const glob = require(‘glob’);

const { resolve } = require(‘path’);

registerRouter = ()=> {

let routers = [];

//递归式获取当前文件夹下所有的.js文件

glob.sync(resolve(\_\_dirname, ‘./’, ‘\*\*/\*.js’))

//排除index.js文件，因为这个文件不是具体的路由文件

.filter(value=> (value.indexOf(‘index.js’)===-1))

.forEach(router=>{

routers.push(require(router).routes());

Routers.push(require(router).allowedMethods());

})

}

return compose(routers);

}

module.exports = registerRouter;

这里可以使用koa-compose对koa-router进行整合，这是因为koa-router里面的routers方法和allowedMethods方法和我们平时用的中间件回调方法是一样的。

//app.js

const Koa = require(‘koa’);

const registerRouter = require(‘./routers’);

const app = new Koa();

app.use(registerRouter);

app.listen(4000, ()=>{

....

});

匹配路由时　<http://127.0.0.1:4000/goods/getInfo> ,那么会匹配到actions目录下的goods/getInfo.js 文件，最终会执行getInfo.js里面的逻辑。

依据项目的文件目录就能了解就能了解项目包含哪些路由，不用查看路由文件，非常方便。

用文件路径来组织路由，对用户非常友好，且便于开发。

实现代码：

**actions/goods/getInfo.js**

module.exports = {

method: ‘GET’,

handler: (ctx) => { //业务逻辑

ctx.body = “...”

}

}

**actions/user/getInfo.js**

module.exports = {

method: ‘GET’,

handler: (ctx) => {

ctx.body = “....”

}

}

**glob 包来获取所有的文件，然后对路径再做相应的处理。**

const glob = require(‘glob’);

const path = require(‘path’);

const Koa = require(‘koa’);

const app = new Koa();

//actions的绝对路径

const basePath = path.resolve(\_\_dirname, ‘./actions’);

//获取actions目录下所有的.js文件，并返回其绝对路径

const filesList = glob.sync(path.resolve(\_\_dirname, ‘./actions’, ‘\*\*/\*.js’));

//文件路由映射表

let routerMap = {};

filesList.forEach(item=>{

//解构的方式获得当前文件导出对象中的method属性和handler属性

const { method, handler } = require(item);

//获取和actions目录的相对路径，例如： goods/getInfo.js

const relative = path.relative(basePath, item);

//获取文件后缀 .js

const extname = path.extname(item);

//剔除后缀 .js， 并在前面加一个“/”，例如：/goods/getInfo

const fileRouter = ‘/’ + relative.split(exname)[0];

//连接method，形成一个唯一请求，例如： \_GET\_/goods/getInfo

const key = ‘\_’ + method + ‘\_’ + fileRouter;

//保存在路由表里

routeMap[key] = handler;

});

app.use(async (ctx, next)=>{

const { path, method } = ctx;

//构建和文件路由匹配的形式为\_GET\_路由

const key = ‘\_’ + method + ‘\_’ + path;

//如果匹配到，就执行对应到handler方法

if(routerMap[key]){

routerMap[key](ctx);

}else{

ctx.body = ‘no this router’;

}

});

**Koa JWT:**

const { sign } = require(‘jsonwebtoken’);

const secret = ‘xxxxxx’;

const jwt = require(‘koa-jwt’)({ secret });

........

const token = sign({username}, secret, {expiresIn: ‘1h’});

JWT鉴权失败会返回401。

Mysql:

const mysql = require(‘mysql2’);

const connection = mysql.createConnection(

{

host : ‘localhost’,

user : ‘root’,

password: ‘123456’,

database: ‘koadb’

}

);

connection.connect();

const sql = “....”;

connection.query(sql, function(error, results, fields){

if(error) throw error;

});

**Mysql ORM: 第三方插件 sequelize**

安装依赖sequelize 和 mysql2

const { Sequelize, DataTypes } = require(‘sequelize’);

const sequelize = new Sequelize(db, username, password, {host: ‘localhost’, dialect: ‘mysql’});

<http://sequelize.org/>

**日志处理 log4js.**

const log4js = require(“log4js”);

const logger = log4js.getLogger();

logger.level = “debug”;

logger.debug(“xxxxxx”);

日志分类：

const log4js = require(“log4js”);

log4js.configure({

appenders: {

out: { type: ‘stdout’ },

app1: {type: ‘file’, filename: ‘application1.log’},

app2: {type: ‘file’, filename: ‘application2.log’},

},

categories: {

default: { appenders: [‘out’], level: ‘trace’ },

app1: {appenders: [‘app1’], level: ‘trace’},

app2: {appenders: [‘app2’], level: ‘trace’},

}

});

const app1Log = log4js.getLogger(‘app1’);

app1Log.trace(‘....’);

const app2Log = log4js.getLogger(‘app2’);

app2Log.trace(‘....’);

**日志分割：**

const log4js = require(‘log4js’);

log4j.configure({

appenders: {

app:{

type: ‘dateFile’,

filename: ‘application’,

alwaysIncludePattern: true,

pattern: ‘yyyy-MM-dd-hh.log’

}

},

categories: {

default: { appenders: [‘app’], level: ‘trace’},

app: {appenders: [‘app’], level: ‘track’}

}

});

const appLog = log4js.getLogger(‘app’);

applog.trace(‘....’);

**本地调试管理工具 lerna, 监控文件变化,能够自动重启服务端的工具nodemon. 通过 npm i -D nodemon命令安装依赖。**

环境变量： process.env.NODE\_ENV

在Nodejs项目中增加环境配置，通常使用.env 文件或使用环境变量。在项目根目录创建.env文件，定义环境变量

NODE\_ENV=development

使用dotenv包来加载.env文件， npm install dotenv

require(‘dotenv’).config();

const \_env = process.env.NODE\_ENV