

# 课程介绍

- ES6新特性
- ReactJS入门学习

## 1、ES6 新特性

现在使用主流的前端框架中，如ReactJS、Vue.js、angularjs等，都会使用到ES6的新特性，作为一名高级工程师而言，ES6也就成为了必修课，所以本套课程先以ES6的新特性开始。

说明：如果已经掌握ES6语法的同学，可以跳过这一节。

### 1.1、了解ES6

ES6，是ECMAScript 6的简称，它是 JavaScript 语言的下一代标准，已于 2015 年 6 月正式发布。

它的目标是使 JavaScript语言可以用于编写复杂的大型应用程序，成为企业级开发语言。

#### 1.1.1.什么是ECMAScript？

来看下前端的发展历程：

web1.0时代：

- 最初的网页以HTML为主，是纯静态的网页。网页是只读的，信息流只能从服务的到客户端单向流通。**开发人员也只关心页面的样式和内容即可。**

web2.0时代：

- 1995年，网景工程师Brendan Eich 花了10天时间设计了JavaScript语言。
- 1996年，微软发布了JScript，其实是JavaScript的逆向工程实现。
- 1997年，为了统一各种不同script脚本语言，ECMA（欧洲计算机制造商协会）以JavaScript为基础，制定了ECMAScript 标准规范。JavaScript和JScript都是 ECMAScript 的标准实现者，随后各大浏览器厂商纷纷实现了ECMAScript 标准。

所以，ECMAScript是浏览器脚本语言的规范，而各种我们熟知的js语言，如JavaScript则是规范的具体实现。

#### 1.1.2.ECMAScript的快速发展

而后，ECMAScript就进入了快速发展期。

- 1998年6月，ECMAScript 2.0 发布。
- 1999年12月，ECMAScript 3.0 发布。这时，ECMAScript 规范本身也相对比较完善和稳定了，但是接下来的事情，就比较悲剧了。
- 2007年10月。。。ECMAScript 4.0 草案发布。

这次的新规范，历时颇久，规范的新内容也有了很多争议。在制定ES4的时候，是分成了两个工作组同时工作的。

- 一边是以 Adobe, Mozilla, Opera 和 Google 为主的 ECMAScript 4 工作组。
- 一边是以 Microsoft 和 Yahoo 为主的 ECMAScript 3.1 工作组。

ECMAScript 4 的很多主张比较激进，改动较大。而 ECMAScript 3.1 则主张小幅更新。最终经过 TC39 的会议，决定将一部分不那么激进的改动保留发布为 ECMAScript 3.1，而 ES4 的内容，则延续到了后来的 ECMAScript 5 和 6 版本中

- 2009 年 12 月，ECMAScript 5 发布。
- 2011 年 6 月，ECMAScript 5.1 发布。
- 2015 年 6 月，ECMAScript 6，也就是 ECMAScript 2015 发布了。并且从 ECMAScript 6 开始，开始采用年号来做版本。即 ECMAScript 2015，就是 ECMAScript 6。
- 2016 年 6 月，小幅修订的《ECMAScript 2016 标准》(简称 ES2016) 如期发布，这个版本可以看作是 ES6.1 版，因为两者的差异非常小(只新增了数组实例的 includes 方法和指数运算符)，基本上可以认为是同一个标准。
- 2017 年 6 月发布了 ES2017 标准。

因此，ES6 既是一个历史名词，也是一个泛指，含义是 5.1 版本以后的 JavaScript 的下一代标准，涵盖了 ES2015、ES2016、ES2017 等，而 ES2015 则是正式名称，特指当年发布的正式版本的语言标准。

## 1.2、let 和 const 命令

var

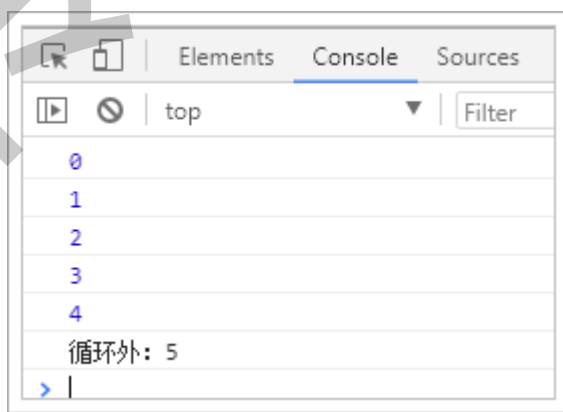
之前，我们写 js 定义变量的时候，只有一个关键字：var

var 有一个问题，就是定义的变量有时会莫名其妙的成为全局变量。

例如这样的一段代码：

```
1 for(var i = 0; i < 5; i++){
2     console.log(i);
3 }
4 console.log("循环外：" + i)
```

运行打印的结果是如下：



可以看出，在循环外部也可以获取到变量 i 的值，显然变量 i 的作用域范围太大了，在做复杂页面时，会带来很大的问题。

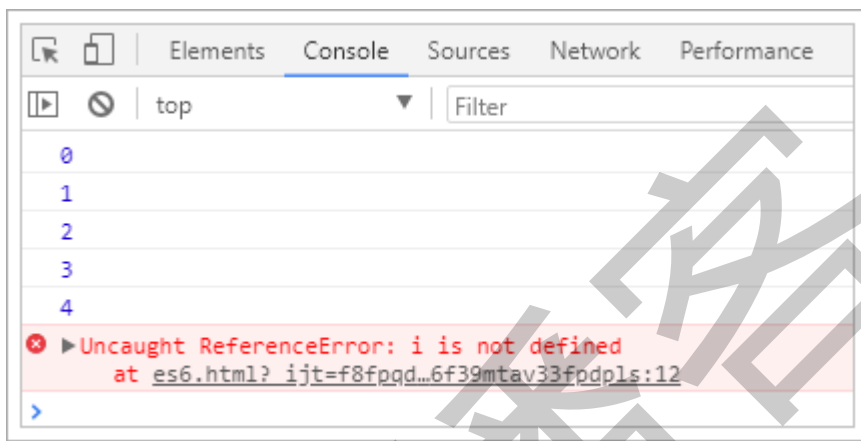
let

let 所声明的变量，只在 let 命令所在的代码块内有效。

我们把刚才的 var 改成 let 试试：

```
1 for(let i = 0; i < 5; i++){
2   console.log(i);
3 }
4 console.log("循环外：" + i)
```

结果：

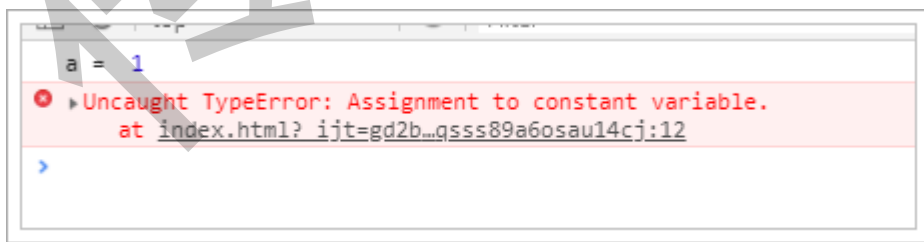


这样，就把变量的i的作用域控制在了循环内部。

const

const 声明的变量是常量，不能被修改，类似于java中final关键字。

```
1 const a = 1;
2 console.log("a = ", a);
3 //给a重新赋值
4 a = 2;
5 console.log("a = ", a);
```



可以看到，变量a的值是不能修改的。

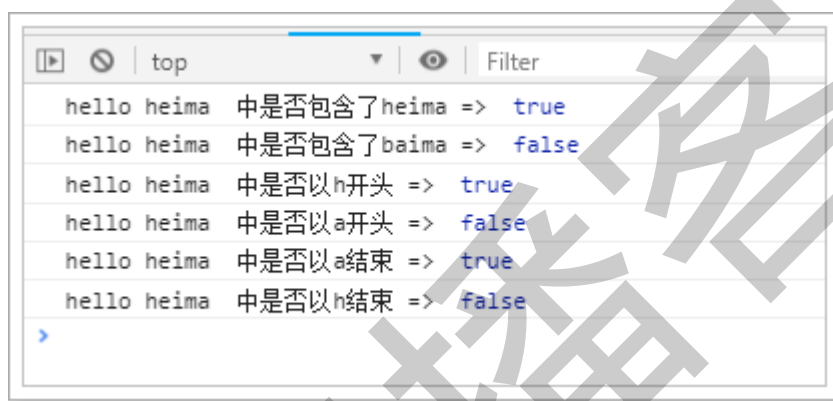
## 1.3、字符串扩展

在ES6中，为字符串扩展了几个新的API：

- includes()：返回布尔值，表示是否找到了参数字符串。
- startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
- endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。

实验一下：

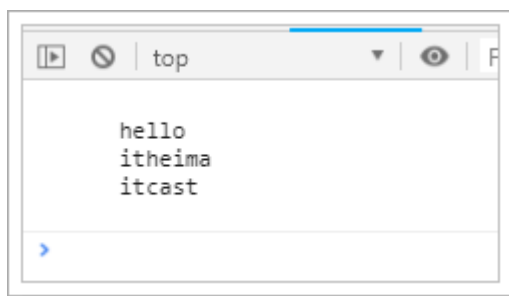
```
1 <script>
2   let str = "hello heima";
3   console.log(str, " 中是否包含了heima => ", str.includes("heima"));
4   console.log(str, " 中是否包含了baima => ", str.includes("baima"));
5
6   console.log(str, " 中是否以h开头 => ", str.startsWith("h"));
7   console.log(str, " 中是否以a开头 => ", str.startsWith("a"));
8
9   console.log(str, " 中是否以a结束 => ", str.endsWith("a"));
10  console.log(str, " 中是否以h结束 => ", str.endsWith("h"));
11 </script>
```



#### 字符串模板

ES6中提供了`来作为字符串模板标记。我们可以这么玩：

```
1 <script>
2   let str = `
3   hello
4   itheima
5   itcast
6   `;
7   console.log(str);
8
9 </script>
```



在两个`之间的部分都会被作为字符串的值，可以任意换行。

## 1.3、解构表达式

什么是解构？-- ES6中允许按照一定模式从数组和对象中提取值，然后对变量进行赋值，这被称为解构（Destructuring）。

### 1.3.1、数组解构

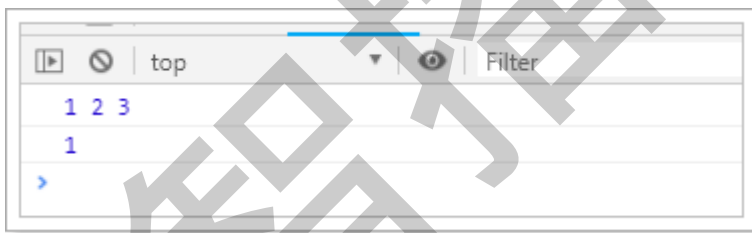
比如有一个数组：

```
1 let arr = [1,2,3]
```

之前，我想获取其中的值，只能通过角标。ES6可以这样：

```
1 let arr = [1,2,3]
2 const [x,y,z] = arr; // x,y,z将与arr中的每个位置对应来取值
3 // 然后打印
4 console.log(x,y,z);
5
6 const [a] = arr; //只匹配1个参数
7 console.log(a);
```

结果：



### 1.3.2、对象解构

例如有个person对象：

```
1 const person = {
2   name: "jack",
3   age: 21,
4   language: ['java', 'js', 'css']
5 }
```

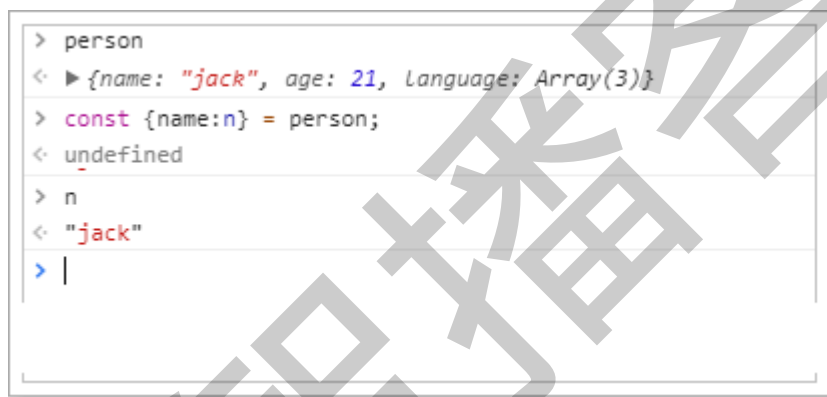
我们可以这么做：

```
1 // 解构表达式获取值
2 const {name,age,language} = person;
3 // 打印
4 console.log(name);
5 console.log(age);
6 console.log(language);
```

结果：



如过想要用其它变量接收，需要额外指定别名：



- `{name:n}`：name是person中的属性名，冒号后面的n是解构后要赋值给的变量。

## 1.4、函数优化

在ES6中，对函数的操作做了优化，使得我们在操作函数时更加的便捷。

### 1.4.1、函数参数默认值

在ES6以前，我们无法给一个函数参数设置默认值，只能采用变通写法：

```
1 function add(a, b) {  
2     // 判断b是否为空，为空就给默认值1  
3     b = b || 1;  
4     return a + b;  
5 }  
6 // 传一个参数  
7 console.log(add(10));
```

现在可以这么写：



```
1 function add(a , b = 1) {  
2     return a + b;  
3 }  
4 // 传一个参数  
5 console.log(add(10));
```

### 1.4.2、箭头函数

ES6中定义函数的简写方式：

一个参数时：

```
1 var print = function (obj) {  
2     console.log(obj);  
3 }  
4 // 简写为：  
5 var print2 = obj => console.log(obj);
```

多个参数：

```
1 // 两个参数的情况：  
2 var sum = function (a , b) {  
3     return a + b;  
4 }  
5 // 简写为：  
6 var sum2 = (a,b) => a+b;
```

没有参数：

```
1 // 没有参数时，需要通过()进行占位，代表参数部分  
2 let sayHello = () => console.log("hello!");  
3 sayHello();
```

代码不止一行，可以用 {} 括起来。

```
1 var sum3 = (a,b) => {  
2     return a + b;  
3 }  
4  
5 // 多行，没有返回值  
6 let sayHello = () => {  
7     console.log("hello!");  
8     console.log("world!");  
9 }  
10  
11 sayHello();
```

### 1.4.3、对象的函数属性简写

比如一个Person对象，里面有eat方法：

```
1 let person = {
2   name: "jack",
3   // 以前：
4   eat: function (food) {
5     console.log(this.name + "在吃" + food);
6   },
7   // 箭头函数版：
8   eat2: food => console.log(person.name + "在吃" + food), // 这里拿不到this
9   // 简写版：
10  eat3(food){
11    console.log(this.name + "在吃" + food);
12  }
13 }
```

### 1.4.4、箭头函数结合解构表达式

比如有一个函数：

```
1 const person = {
2   name: "jack",
3   age: 21,
4   language: ['java', 'js', 'css']
5 }
6
7 function hello(person) {
8   console.log("hello," + person.name)
9 }
```

如果用箭头函数和解构表达式

```
1 var hi = ({name}) => console.log("hello," + name);
2 hi(person)
```

## 1.5、map和reduce

ES6中，数组新增了map和reduce方法。

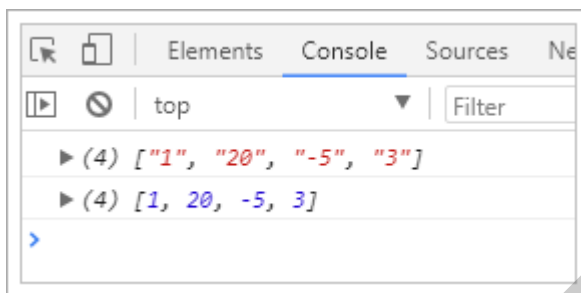
### 1.5.1、map

`map()`：接收一个函数，将原数组中的所有元素用这个函数处理后放入新数组返回。

举例：有一个字符串数组，我们希望转为int数组



```
1 let arr = ['1', '20', '-5', '3'];
2 console.log(arr)
3
4 let newArr = arr.map(s => parseInt(s));
5
6 console.log(newArr)
```



### 1.5.1、reduce

`reduce()`：接收一个函数（必须）和一个初始值（可选），该函数接收两个参数：

- 第一个参数是上一次reduce处理的结果
- 第二个参数是数组中要处理的下一个元素

`reduce()` 会从左到右依次把数组中的元素用reduce处理，并把处理的结果作为下次reduce的第一个参数。如果是第一次，会把前两个元素作为计算参数，或者把用户指定的初始值作为起始参数

举例：

```
1 const arr = [1, 20, -5, 3]
```

没有初始值：

```
> arr.reduce((a,b) => a+b)
< 19
> arr.reduce((a,b) => a*b)
< -300
```

指定初始值：

```
> arr.reduce((a,b) => a*b)
< -300
> arr.reduce((a,b) => a*b, 0)
< -0
> arr.reduce((a,b) => a*b, 1)
< -300
> arr.reduce((a,b) => a*b, -1)
< 300
>
```

## 1.6、扩展运算符

扩展运算符(spread)是三个点(...)，将一个数组转为用逗号分隔的参数序列。

用法：

```
1 console.log (...[1, 2, 3]); //1 2 3
2 console.log(1, ...[2, 3, 4], 5); // 1 2 3 4 5
3
4
5 function add(x, y) {
6     return x + y;
7 }
8 var numbers = [1, 2];
9 console.log(add(...numbers)); // 3
10
11 // 数组合并
12 let arr = [...[1,2,3],...[4,5,6]];
13 console.log(arr); //[1, 2, 3, 4, 5, 6]
14
15 // 与解构表达式结合
16 const [first, ...rest] = [1, 2, 3, 4, 5];
17 console.log(first, rest) //1 [2, 3, 4, 5]
18
19 //将字符串转成数组
20 console.log([... 'hello']) //["h", "e", "l", "l", "o"]
```

## 1.7、Promise

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

我们可以通过Promise的构造函数来创建Promise对象，并在内部封装一个异步执行的结果。

语法：

```
1 const promise = new Promise(function(resolve, reject) {
2     // ... 执行异步操作
3
4     if (/* 异步操作成功 */) {
5         resolve(value); // 调用resolve，代表Promise将返回成功的结果
6     } else {
7         reject(error); // 调用reject，代表Promise会返回失败结果
8     }
9 });
```

这样，在promise中就封装了一段异步执行的结果。

如果我们想要等待异步执行完成，做一些事情，我们可以通过promise的then方法来实现,语法：

```
1 promise.then(function(value){
2     // 异步执行成功后的回调
3 });
```

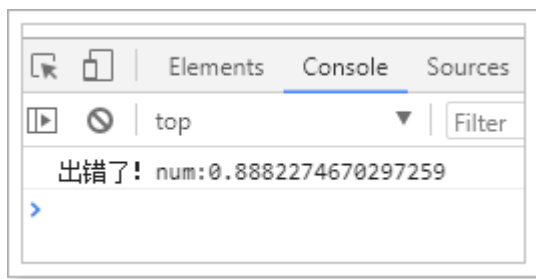
如果想要处理promise异步执行失败的事件，还可以跟上catch：

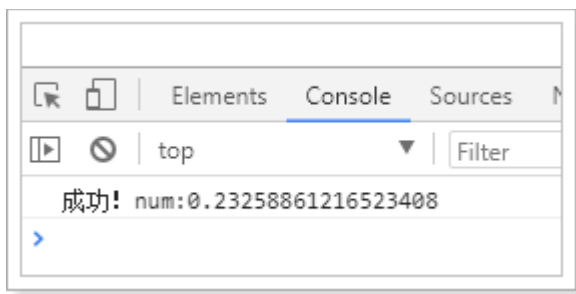
```
1 promise.then(function(value){
2     // 异步执行成功后的回调
3 }).catch(function(error){
4     // 异步执行失败后的回调
5 });
```

示例：

```
1 const p = new Promise(function (resolve, reject) {
2     // 这里我们用定时任务模拟异步
3     setTimeout(() => {
4         const num = Math.random();
5         // 随机返回成功或失败
6         if (num < 0.5) {
7             resolve("成功! num:" + num)
8         } else {
9             reject("出错了! num:" + num)
10        }
11    }, 300)
12 })
13
14 // 调用promise
15 p.then(function (msg) {
16     console.log(msg);
17 }).catch(function (msg) {
18     console.log(msg);
19 })
20
```

结果：





## 1.8、set和map

ES6提供了Set和Map的数据结构。

Set，本质与数组类似。不同在于Set中只能保存不同元素，如果元素相同会被忽略。和java中的Set集合非常相似。

构造函数：

```
1 // Set构造函数可以接收一个数组或空
2 let set = new Set();
3 set.add(1); // [1]
4 // 接收数组
5 let set2 = new Set([2,3,4,5,5]); // 得到[2,3,4,5]
```

方法：

```
1 set.add(1); // 添加
2 set.clear(); // 清空
3 set.delete(2); // 删除指定元素
4 set.has(2); // 判断是否存在
5 set.forEach(function() {}); // 遍历元素
6 set.size; // 元素个数。是属性，不是方法。
7
```

map，本质是与Object类似的结构。不同在于，Object强制规定key只能是字符串。而Map结构的key可以是任意对象。即：

- object是 <string,object>集合
- map是<object,object>集合

构造函数：



```
1 // map接收一个数组，数组中的元素是键值对数组
2 const map = new Map([
3   ['key1', 'value1'],
4   ['key2', 'value2'],
5 ])
6 // 或者接收一个set
7 const set = new Set([
8   ['key1', 'value1'],
9   ['key2', 'value2'],
10 ])
11 const map2 = new Map(set)
12 // 或者其它map
13 const map3 = new Map(map);
```

方法：

```
1 map.set(key, value); // 添加
2 map.clear(); // 清空
3 map.delete(key); // 删除指定元素
4 map.has(key); // 判断是否存在
5 map.forEach(function(key, value){}) // 遍历元素
6 map.size; // 元素个数。是属性，不是方法
7
8 map.values() // 获取value的迭代器
9 map.keys() // 获取key的迭代器
10 map.entries() // 获取entry的迭代器
11 用法：
12 for (let key of map.keys()) {
13   console.log(key);
14 }
15 或：
16 console.log(...map.values()); // 通过扩展运算符进行展开
```

## 1.9、class (类) 的基本语法

JavaScript 语言的传统方法是通过构造函数定义并生成新对象。ES6中引入了class的概念，通过class关键字自定义类。

基本用法：

```
1 <script>
2
3   class User{
4     constructor(name, age = 20){ // 构造方法
5       this.name = name; // 添加属性并且赋值
6       this.age = age;
7     }
8
9     sayHello(){ // 定义方法
10      return "hello";
11    }
12  }
```



```
13     static isAdult(age){ //静态方法
14         if(age >= 18){
15             return "成年人";
16         }
17         return "未成年人";
18     }
19 }
20
21 let user = new User("张三");
22
23 // 测试
24 console.log(user); // User {name: "张三", age: 20}
25 console.log(user.sayHello()); // hello
26 console.log(User.isAdult(20)); // 成年人
27
28
29 </script>
```

类的继承：

```
1 <script>
2
3     class User{
4         constructor(name, age = 20){ // 构造方法
5             this.name = name; // 添加属性并且赋值
6             this.age = age;
7         }
8
9         sayHello(){
10             return "hello"; // 定义方法
11         }
12
13         static isAdult(age){ //静态方法
14             if(age >= 18){
15                 return "成年人";
16             }
17             return "未成年人";
18         }
19     }
20
21     class ZhangSan extends User{
22         constructor(){
23             super("张三", 30); //如果父类中的构造方法有参数，那么子类必须通过super调用父类的构造
方法
24             this.address = "上海"; //设置子类中的属性，位置必须处于super下面
25         }
26     }
27
28     // 测试
29     let zs = new ZhangSan();
30     console.log(zs.name, zs.address);
31     console.log(zs.sayHello());
32     console.log(ZhangSan.isAdult(20));
```

```
33  
34  
35 </script>
```

## 1.10、Generator函数

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。

Generator函数有两个特征：一是 function命令与函数名 之间有一个星号；二是 函数体内部使用 yield语句定义不同的内部状态。

用法：

```
1 <script>  
2  
3     function* hello () {  
4         yield "hello";  
5         yield "world";  
6         return "done";  
7     }  
8  
9     let h = hello();  
10  
11     console.log(h.next()); //{value: "hello", done: false}  
12     console.log(h.next()); //{value: "world", done: false}  
13     console.log(h.next()); //{value: "done", done: true}  
14     console.log(h.next()); //{value: undefined, done: true}  
15  
16  
17 </script>
```

可以看到，通过hello()返回的h对象，每调用一次next()方法返回一个对象，该对象包含了value值和done状态。直到遇到return关键字或者函数执行完毕，这个时候返回的状态为ture，表示已经执行结束了。

### 1.10.1、for...of循环

通过for...of可以循环遍历Generator函数返回的迭代器。

用法：

```
1 <script>  
2  
3     function* hello () {  
4         yield "hello";  
5         yield "world";  
6         return "done";  
7     }  
8  
9     let h = hello();  
10  
11     for (let obj of h) {  
12         console.log(obj);  
13     }
```



```
14
15 </script>
16
17 // 输出：
18 hello
19 world
```

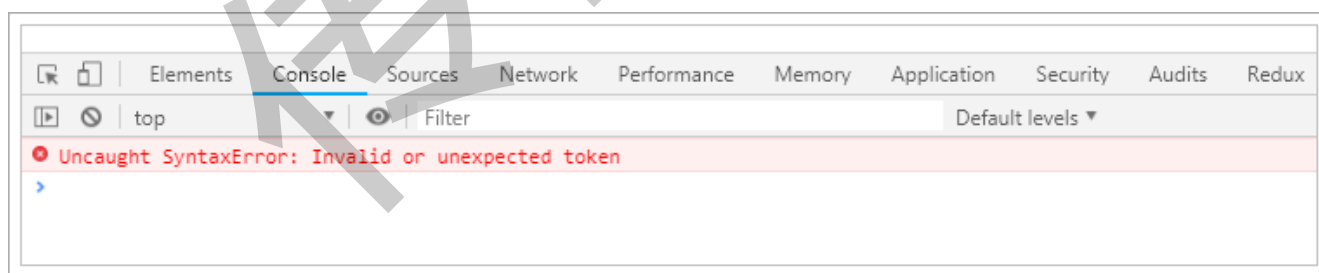
## 1.11、修饰器(Decorator)

修饰器(Decorator)是一个函数，用来修改类的行为。ES2017 引入了这项功能，目前 Babel 转码器已经支持。

使用：

```
1 <script>
2
3   @T //通过@符号进行引用该方法，类似java中的注解
4   class User {
5       constructor(name, age = 20){
6           this.name = name;
7           this.age = age;
8       }
9   }
10
11   function T(target) { //定义一个普通的方法
12       console.log(target); //target对象为修饰的目标对象，这里是User对象
13       target.country = "中国"; //为User类添加一个静态属性country
14   }
15
16   console.log(User.country); //打印出country属性值
17
18 </script>
```

运行报错：



原因是，在ES6中，并没有支持该用法，在ES2017中才有，所以我们不能直接运行了，需要进行编码后再运行。

转码的意思是：将ES6或ES2017转为ES5执行。类似这样：



```
1 //转码前
2 input .map(item => item + 1);
3
4 //转码后
5 input.map(function (item) {
6   return item + 1;
7 })
```

## 1.12、转码器

- Babel (babeljs.io) 是一个广为使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在浏览器或其他环境执行。
- Google 公司的 Traceur 转码器 (github.com/google/traceur-compiler)，也可以将 ES6 代码转为 ES5 的代码。

这两款都是非常优秀的转码工具，在本套课程中并不会直接使用，而是会使用阿里的开源企业级 react 框架：UmiJS。

### 1.12.1、了解UmiJS

官网：<https://umijs.org/zh/>



UmiJS 读音：( 乌米 )

特点：

- 插件化
  - umi 的整个生命周期都是插件化的，甚至其内部实现就是由大量插件组成，比如 pwa、按需加载、一键切换 preact、一键兼容 ie9 等等，都是由插件实现。
- 开箱即用
  - 你只需一个 umi 依赖就可启动开发，无需安装 react、preact、webpack、react-router、babel、jest 等等。
- 约定式路由
  - 类 next.js 的约定式路由，无需再维护一份冗余的路由配置，支持权限、动态路由、嵌套路由等等。

### 1.12.2、部署安装



```
1 #首先，需要安装Node.js
2 #在资料中，找到node-v8.12.0-x64.msi，一路下一步安装
3 #安装完成后，通过node -v 命令查看其版本号
4
5 F:\code\itcast-es6>node -v
6 v8.12.0
7
8 #接下来，开始安装yarn，其中tyarn使用的是npm.taobao.org的源，速度要快一些
9 #可以把yarn看做了优化了的npm
10 npm i yarn tyarn -g #-g 是指全局安装
11 tyarn -v #进行测试，如果能够正常输出版本信息则说明安装成功了
12 #如果安装失败，是由于将yarn添加到环境变量中导致，参见
13 http://www.easysb.cn/index.php/2017/06/04/11/
14
15 #下面开始安装umi
16 tyarn global add umi
17 umi #进行测试
```

### 1.12.3、快速入门

```
1 #通过初始化命令将生成package.json文件，它是 NodeJS 约定的用来存放项目的信息和配置等信息的文件。
2 tyarn init -y
3
4 #通过umi 命令创建index.js文件
5 umi g page index #可以看到在pages下创建好了index.js和index.css文件
6
7 #将下面内容拷贝到index.js文件中进行测试
8 @T //通过@符号进行引用该方法，类似java中的注解
9 class User {
10     constructor(name, age = 20){
11         this.name = name;
12         this.age = age;
13     }
14 }
15
16 function T(target) { //定义一个普通的方法
17     console.log(target); //target对象为修饰的目标对象，这里是User对象
18     target.country = "中国"; //为User类添加一个静态属性country
19 }
20
21 console.log(User.country); //打印出country属性值
22
23 #通过命令行启动umi的后台服务，用于本地开发
24 umi dev
25 #通过浏览器进行访问：http://localhost:8000/，查看效果
26 #值得注意的是，这里访问的是umi的后台服务，不是idea提供的服务
```

```
top Filter Default levels ▼  
  
class User {  
  constructor(name) {  
    var age = arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 20;  
    this.name = name;  
    this.age = age;  
  }  
}  
  
中国  
>
```

查看编码后的js文件：

```
var _class;  
  
var //通过@符号进行引用该方法，类似java中的注解  
User = T( class = class User {  
  constructor(name) {  
    var age = arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 20;  
    this.name = name;  
    this.age = age;  
  }  
}) || _class;  
  
function T(target) {  
  //定义一个普通的方法  
  console.log(target); //target对象为修饰的目标对象，这里是User对象  
  
  target.country = "中国"; //为用户类添加一个静态属性country  
}  
  
console.log (User.country); //打印出country属性值
```

可以看到，将我们写的代码进行的编码。

## 1.13、模块化

### 1.13.1.什么是模块化

模块化就是把代码进行拆分，方便重复利用。类似java中的导包：要使用一个包，必须先导包。

而JS中没有包的概念，换来的是 模块。

模块功能主要由两个命令构成：`export` 和 `import`。

- `export` 命令用于规定模块的对外接口，
- `import` 命令用于导入其他模块提供的功能。

### 1.13.2、export

比如我定义一个js文件:Util.js，里面有一个Util类：

```
1 class Util {
2     static sum = (a, b) => a + b;
3 }
4
5 //导出该类
6 export default Util;
```

### 1.13.3、import

使用 `export` 命令定义了模块的对外接口以后，其他 JS 文件就可以通过 `import` 命令加载这个模块。

例如我要使用上面导出的 Util：

```
1 //Index.js
2 //导入Util类
3 import Util from './Util'
4
5 //使用Util中的sum方法
6 console.log(Util.sum(1, 2));
```

通过 <http://localhost:8000/> 进行访问测试。

## 2、ReactJS入门

### 2.1、前端开发的演变

到目前为止，前端的开发经历了四个阶段，目前处于第四个阶段。这四个阶段分别是：

#### 阶段一：静态页面阶段

在第一个阶段中前端页面都是静态的，所有前端代码和前端数据都是后端生成的。前端只是纯粹的展示功能，js脚本的作用只是增加一些特殊效果，比如那时很流行用脚本控制页面上飞来飞去的广告。

那时的网站开发，采用的是后端 MVC 模式。

- Model（模型层）：提供/保存数据
- Controller（控制层）：数据处理，实现业务逻辑
- View（视图层）：展示数据，提供用户界面

前端只是后端 MVC 的 V。

#### 阶段二：ajax阶段

2004年，AJAX 技术诞生，改变了前端开发。Gmail 和 Google 地图这样革命性的产品出现，使得开发者发现，前端的作用不仅仅是展示页面，还可以管理数据并与用户互动。

就是从这个阶段开始，前端脚本开始变得复杂，不再仅仅是一些玩具性的功能。

#### 阶段三：前端MVC阶段

2010年，第一个前端 MVC 框架 Backbone.js 诞生。它基本上是把 MVC 模式搬到了前端，但是只有 M（读写数据）和 V（展示数据），没有 C（处理数据）。

有些框架提出了MVVM模式，用 View Model 代替 Controller。Model 拿到数据以后，View Model 将数据处理成视图层（View）需要的格式，在视图层展示出来。

#### 阶段四：SPA阶段

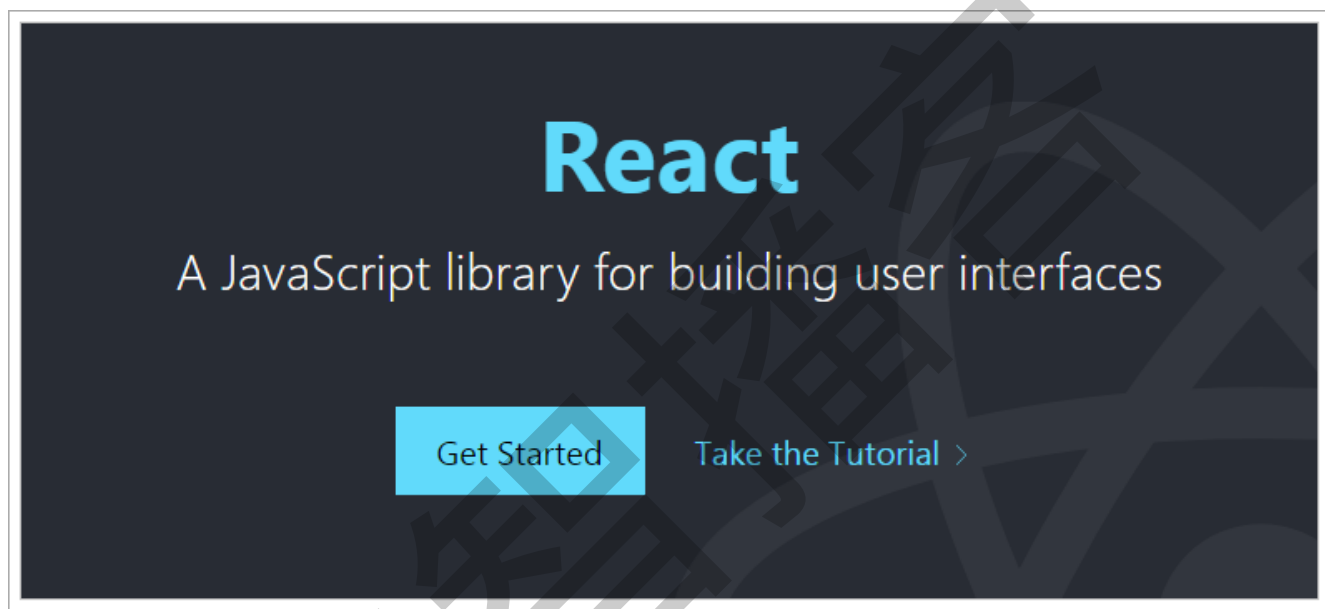
前端可以做到读写数据、切换视图、用户交互，这意味着，网页其实是一个应用程序，而不是信息的纯展示。这种单张网页的应用程序称为 SPA（single-page-application）。

2010年后，前端工程师从开发页面（切模板），逐渐变成了开发“前端应用”（跑在浏览器里面的应用程序）。

目前，最流行的前端框架 Vue、Angular、React 等等，都属于 SPA 开发框架。

## 2.2、ReactJS简介

官网：<https://reactjs.org/>



官方一句很简单的话，道出了什么是ReactJS，就是，一个用于构建用户界面的JavaScript框架，是Facebook开发的一款的JS框架。

ReactJS把复杂的页面，拆分成一个个的组件，将这些组件一个个的拼装起来，就会呈现多样的页面。ReactJS可以用于 MVC 架构，也可以用于 MVVM 架构，或者别的架构。

ReactJS圈内的一些框架简介：

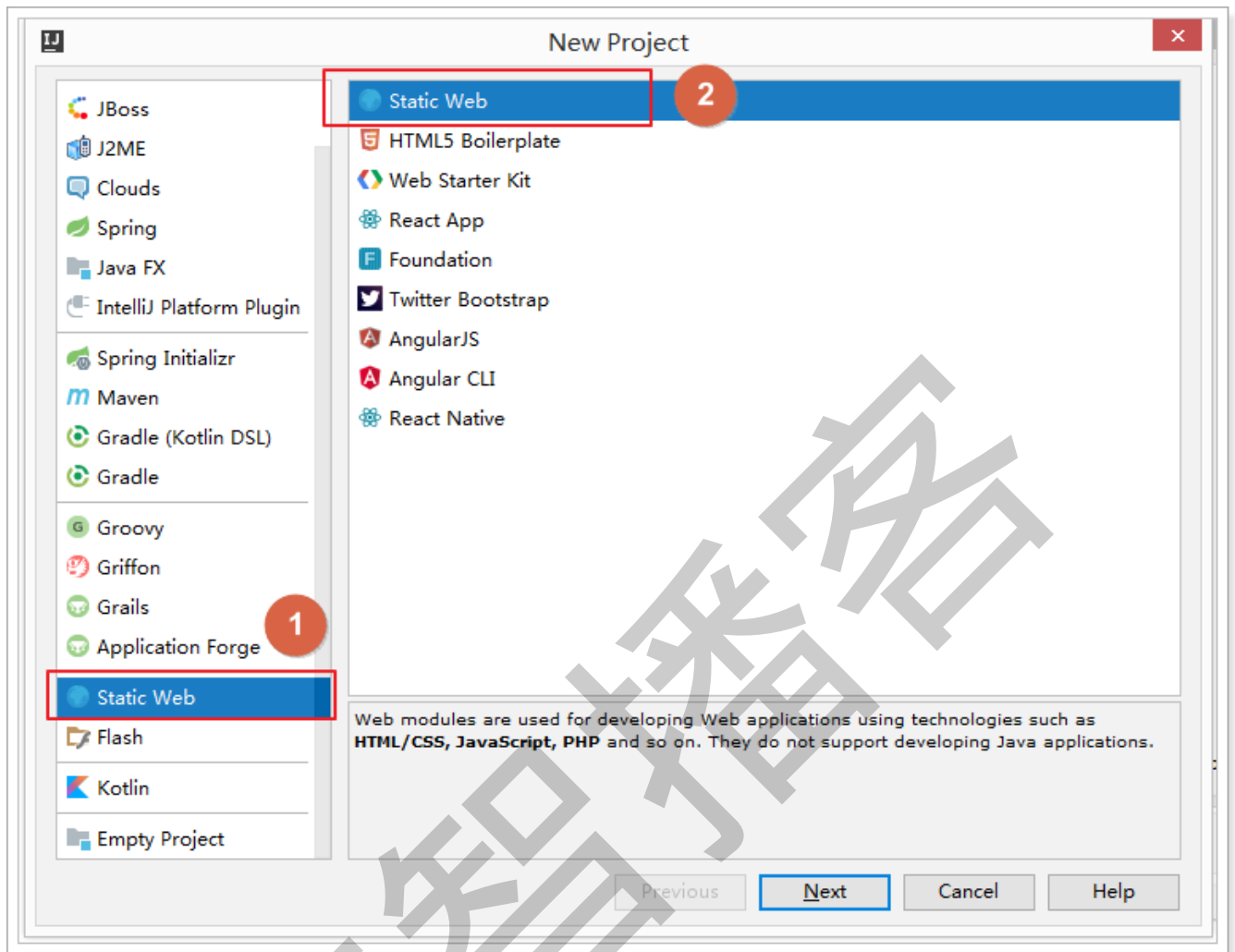
- Flux
  - Flux是Facebook用户建立客户端Web应用的前端架构，它通过利用一个单向的数据流补充了React的组合视图组件，这更是一种模式而非框架。
- Redux
  - Redux 是 JavaScript 状态容器，提供可预测化的状态管理。Redux可以让React组件状态共享变得简单。
- Ant Design of React
  - 阿里开源的基于React的企业级后台产品，其中集成了多种框架，包含了上面提到的Flux、Redux。
  - Ant Design提供了丰富的组件，包括：按钮、表单、表格、布局、分页、树组件、日历等。

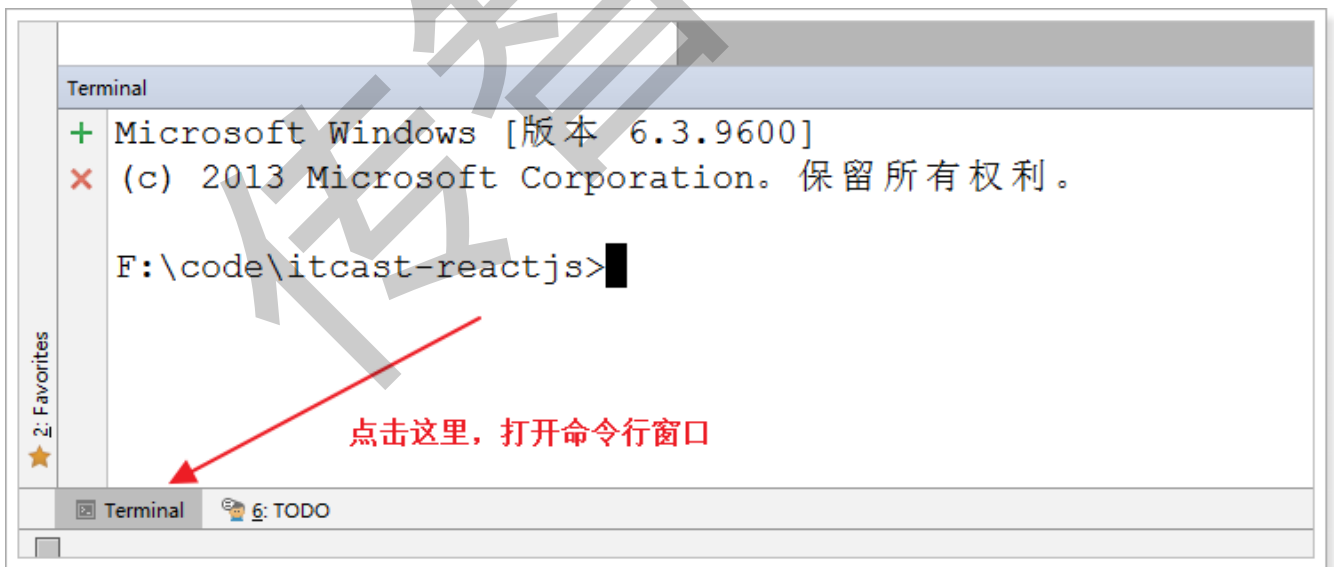
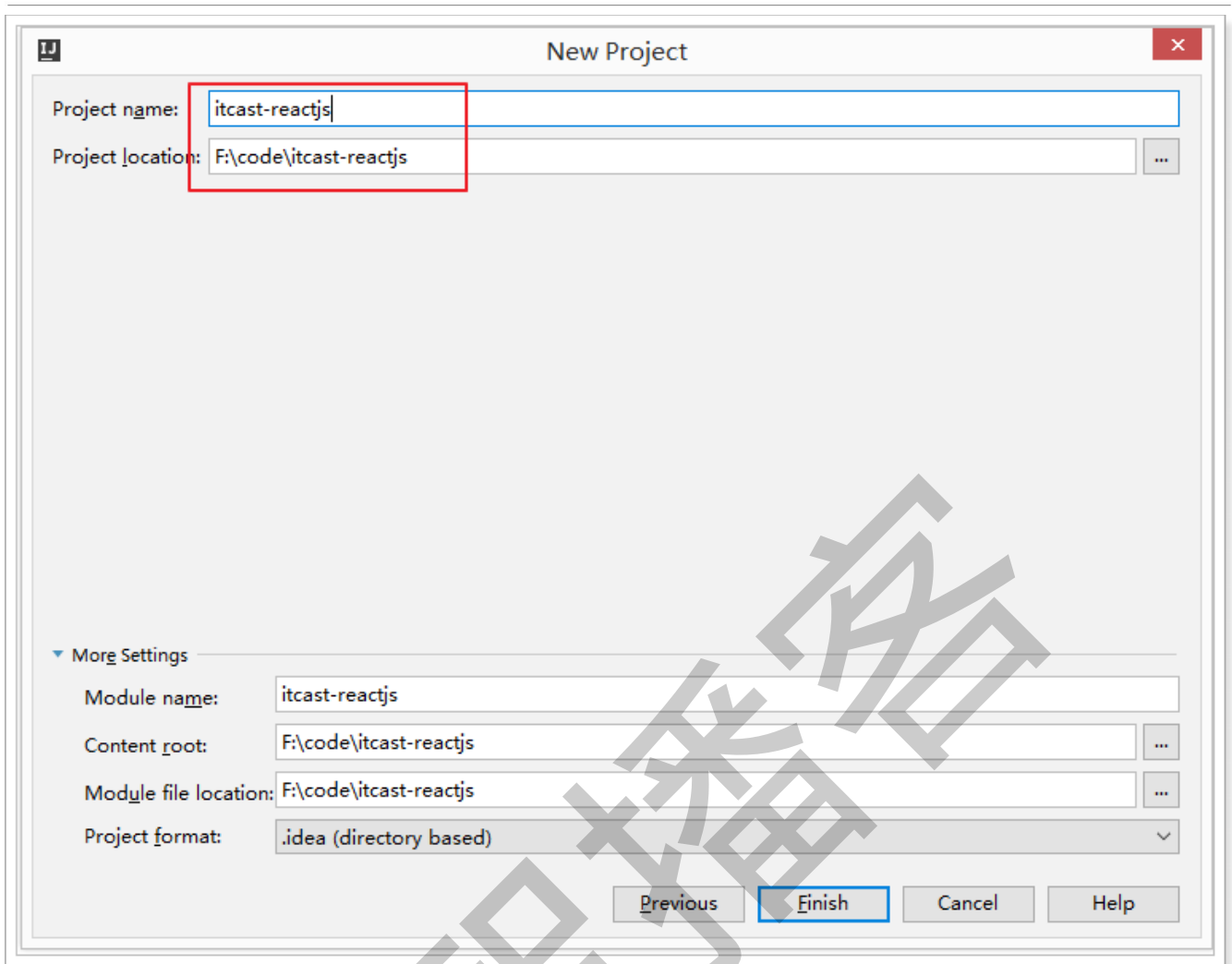
## 2.3、搭建环境

### 2.3.1、创建项目

我们依然选择使用Umijs作为构建工具。

创建工程：





输入命令，进行初始化：

```
1 | yarn init -y
```

初始化完成：







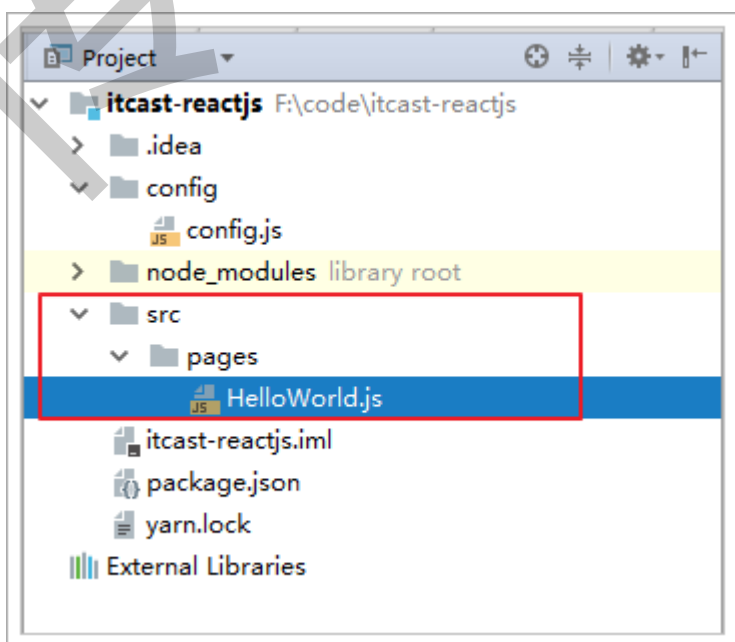
```
.
├─ dist/                // 默认的 build 输出目录
├─ mock/                // mock 文件所在目录，基于 express
├─ config/
│   └─ config.js        // umi 配置，同 .umirc.js，二选一
├─ src/                 // 源码目录，可选
│   └─ layouts/index.js // 全局布局
│   └─ pages/           // 页面目录，里面的文件即路由
│       ├── .umi/        // dev 临时目录，需添加到 .gitignore
│       ├── .umi-production/ // build 临时目录，会自动删除
│       ├── document.ejs // HTML 模板
│       ├── 404.js        // 404 页面
│       ├── page1.js      // 页面 1，任意命名，导出 react 组件
│       ├── page1.test.js // 用例文件，umi test 会匹配所有 .test.js 和 .e2e.js 结尾的文件
│       └─ page2.js       // 页面 2，任意命名
│   └─ global.css        // 约定的全局样式文件，自动引入，也可以用 global.less
│   └─ global.js         // 可以在这里加入 polyfill
├─ .umirc.js             // umi 配置，同 config/config.js，二选一
├─ .env                  // 环境变量
└─ package.json
```

在config.js文件中输入以下内存，以便后面使用：

```
1 //导出一个对象，暂时设置为空对象，后面再填充内容
2 export default {};
```

第二步，创建HelloWorld.js页面文件

在umi中，约定存放页面代码的文件夹是在src/pages，可以通过singular:false来设置单数的命名方式，我们采用默认即可。



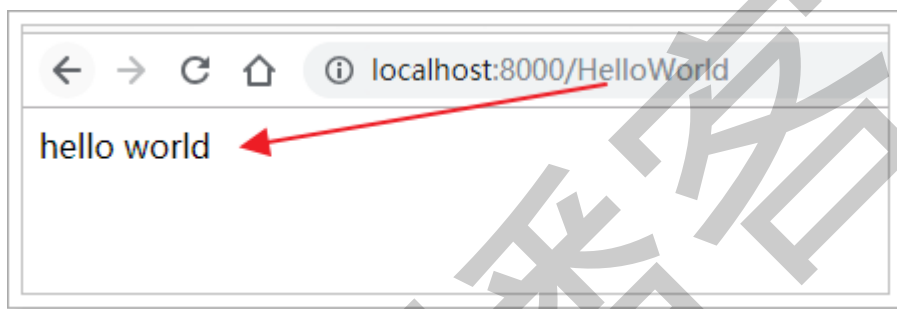
在HelloWorld.js文件中输入如下内容：

```
1 export default () => {  
2   return <div>hello world</div>;  
3 }
```

在这里，可以会比较奇怪，怎么可以在js文件中写html代码，其实，这是react自创的写法，叫SX，后面我们再细说。

第三步，启动服务查看页面效果

```
1 #启动服务  
2 umi dev
```



可以看到，通过/HelloWorld路径即可访问到刚刚写的HelloWorld.js文件。

在 umi 中，可以使用约定式的路由，在 pages 下面的 JS 文件都会按照文件名映射到一个路由，比如上面这个例子，访问 /helloworld 会对应到 HelloWorld.js。

当然了，也可以自定义路由，具体的路由配置在后面讲解。

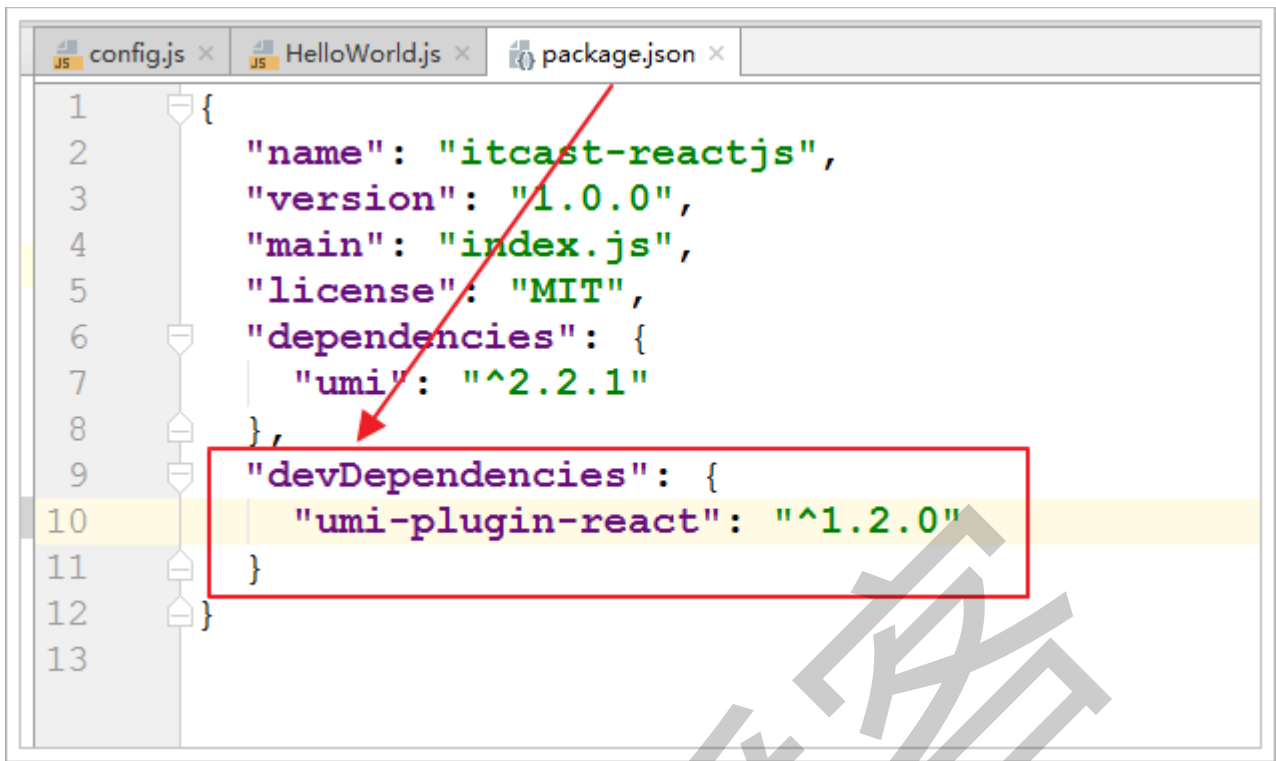
### 2.3.3、添加umi-plugin-react插件

umi-plugin-react插件是umi官方基于react封装的插件，包含了13个常用的进阶功能。

具体可查看：<https://umijs.org/zh/plugin/umi-plugin-react.html>

```
1 #添加插件  
2 yarn add umi-plugin-react --dev
```

添加成功：



The screenshot shows a code editor with three tabs: config.js, HelloWorld.js, and package.json. The package.json file is open, displaying the following JSON structure:

```
1 {
2   "name": "itcast-reactjs",
3   "version": "1.0.0",
4   "main": "index.js",
5   "license": "MIT",
6   "dependencies": {
7     "umi": "^2.2.1"
8   },
9   "devDependencies": {
10    "umi-plugin-react": "^1.2.0"
11  }
12 }
13
```

A red box highlights the `devDependencies` section, and a red arrow points from the `dependencies` section to it.

接下来，在config.js文件中引入该插件：

```
1 export default {
2   plugins: [
3     ['umi-plugin-react', {
4       //暂时不启用任何功能
5     }]
6   ]
7 };
```

#### 2.3.4、构建和部署

现在我们写的js，必须通过umi先转码后才能正常的执行，那么我们最终要发布的项目是普通的html、js、css，那么应该怎么操作呢？

其实，通过umi是可以进行转码生成文件的，具体操作如下：

```
1 | umi build
```

```
F:\code\itcast-reactjs>umi build
```

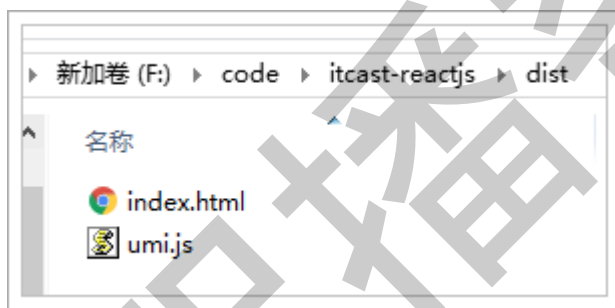
```
[16:10:44] webpack compiled in 5s 983ms
```

```
DONE Compiled successfully in 5992ms
```

```
File sizes after gzip:
```

```
77.24 KB dist\umi.js
```

```
F:\code\itcast-reactjs>
```



可以看到，已经生成了index.html和umi.js文件。我们打开umi.js文件看看。

```
ERROR (No such module: (Possibly not yet loaded) ...),function(){var e,r=/ /;e.cwd=function(){return r},t.chdir=function(t){e||(e=n("33yF")),r=e.resolve(t,r)}(),t.exit=t.kill=t.umask=t.dlopen=t.uptime=t.memoryUsage=t.uvCounters=function(){},t.features={},Q7cW:function(e,t,n){return"use strict";var r=n("TqRt");Object.defineProperty(t,"__esModule",{value:!0}),t.default=void 0;var o=r(n("q1tI")),i=()=>{return o.default.createElement("div",null,"hello world")};t.default=i,QCnb:function(e,t,n){return"use strict";e.exports=n("+wdc")},QLaP:function(e,t,n){return"use strict";var r=function(e,t,n,r,o,i,a,u){if(!e){var l;if(void 0===t)l=new Error("Minified exception occurred; use the non-minified dev environment for the full error message and additional helpful warnings.");else{var c=[n,r,o,i,a,u],s=0;l=new Error(t.replace(/%s/g,function(){return c[s++]})},l.name="Invariant Violation"}throw l.framesToPop=1,l}};e.exports=r},QaDb:function(e,t,n){return"use strict";var r=n("Kuth"),o=n("RjD/"),i=n("fyDq"),a={};n("Mukb")(a,n("K0xU")("iterator"),function(){return this}),e.exports=function(e,t,n){e.prototype=r(a,{next:o(1,n)}),i(e,t+"")
```

首先，看到的是umi.js文件是一个已经压缩过的文件，然后搜索“hello world”，可以找到，我们刚刚写的代码已经被转码了。

至此，开发环境搭建完毕。

## 2.4、React快速入门

### 2.4.1、JSX语法

JSX语法就是，可以在js文件中插入html片段，是React自创的一种语法。

JSX语法会被Babel等转码工具进行转码，得到正常的js代码再执行。

使用JSX语法，需要2点注意：

1. 所有的html标签必须是闭合的，如：



hello world

，写成这样是不可以的：

hello world

2. 在JSX语法中，只能有一个根标签，不能有多个。

```
1 const div1 = <div>hello world</div> //正确
2 const div2 = <div>hello</div> <div>world</div> //错误
```

在JSX语法中，如果想要在html标签中插入js脚本，需要通过{}插入js脚本。

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

formatName 是一个函数

## 2.4.2、组件

组件是React中最重要也是最核心的概念，一个网页，可以被拆分成一个个的组件，像这样：

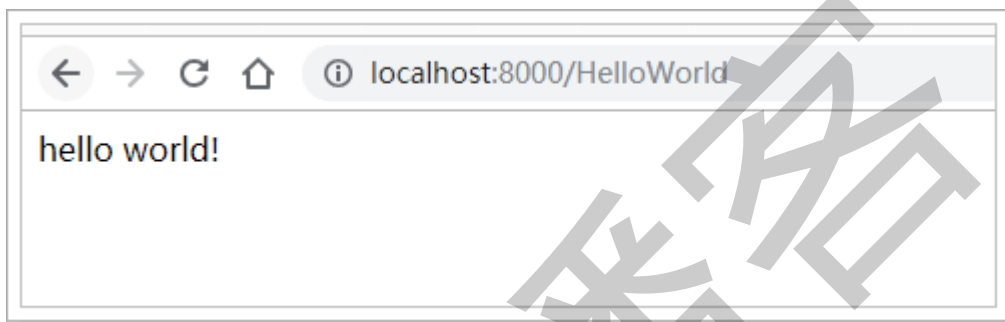


在React中，这样定义一个组件：



```
1 import React from 'react'; //第一步，导入React
2
3 class HelloWorld extends React.Component { //第二步，编写类并且继承 React.Component
4
5     render(){ //第三步，重写render()方法，用于渲染页面
6         return <div>hello world!</div> //JSX语法
7     }
8
9 }
10
11 export default HelloWorld; //第四步，导出该类
```

查看效果：

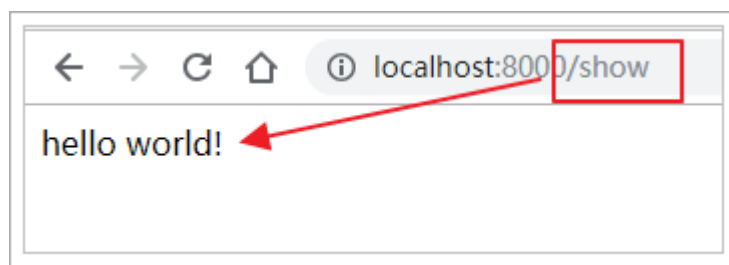


#### 2.4.2.1、导入自定义组件

创建Show.js文件，用于测试导入组件：

```
1 import React from 'react'
2 import HelloWorld from './HelloWorld' //导入HelloWorld组件
3
4 class Show extends React.Component{
5
6     render(){
7         return <HelloWorld/>; //使用HelloWorld组件
8     }
9
10 }
11
12 export default Show;
```

测试：



#### 2.4.2.2、组件参数



组件是可以传递参数的，有2种方式传递，分别是属性和标签包裹的内容传递，具体使用如下：

```
1 import React from 'react'
2 import HelloWorld from './HelloWorld' //导入HelloWorld组件
3
4 class Show extends React.Component{
5
6     render(){
7         return <HelloWorld name="zhangsan">shanghai</HelloWorld>; //使用HelloWorld组件
8     }
9
10 }
11
12 export default Show;
```

其中，name="zhangsan"就是属性传递，shanghai就是标签包裹的内容传递。

那么，在HelloWord.js组件中如何接收参数呢？

对应的也是2种方法：

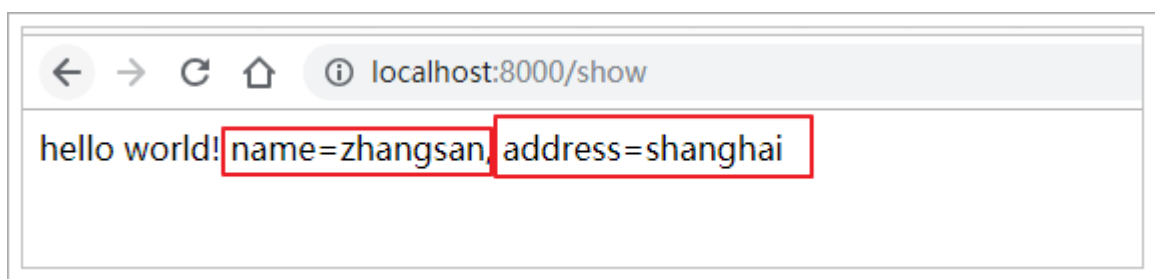
属性：this.props.name 接收；

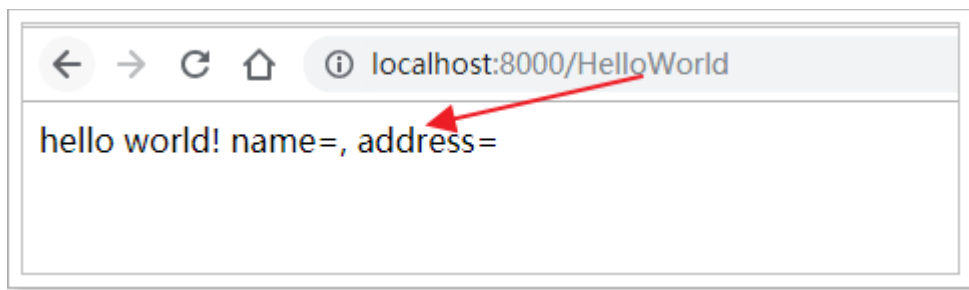
标签内容：this.props.children 接收；

使用如下：

```
1 import React from 'react'; //第一步，导入React
2
3 class HelloWorld extends React.Component { //第二步，编写类并且继承 React.Component
4
5     render(){ //第三步，编写render()方法，用于渲染页面
6         return <div>hello world! name={this.props.name}, address=
7         {this.props.children}</div> //JSX语法
8     }
9
10 }
11 export default HelloWorld; //第四步，导出该类
```

测试：





### 2.4.2.3、组件的状态

每一个组件都有一个状态，其保存在`this.state`中，当状态值发生变化时，React框架会自动调用`render()`方法，重新渲染页面。

其中，要注意两点：

一： `this.state`值的设置要在构造参数中完成；

二：要修改`this.state`的值，需要调用`this.setState()`完成，不能直接对`this.state`进行修改；

下面通过一个案例进行演示，这个案例将实现：通过点击按钮，不断的更新`this.state`，从而反应到页面中。

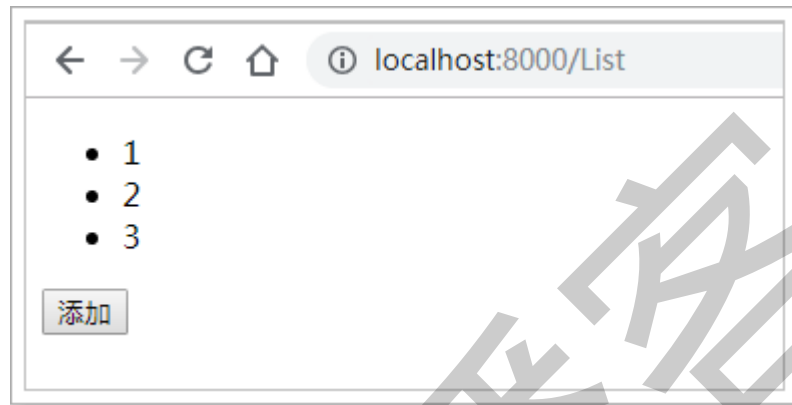
```
1  import React from 'react'
2
3  class List extends React.Component{
4
5      constructor(props){ // 构造参数中必须要props参数
6          super(props); // 调用父类的构造方法
7          this.state = { // 初始化this.state
8              dataList : [1,2,3],
9              maxNum : 3
10         };
11     }
12
13     render(){
14         return (
15             <div>
16                 <ul>
17                     {
18                         // 遍历值
19                         this.state.dataList.map((value,index) => {
20                             return <li key={index}>{value}</li>
21                         })
22                     }
23                 </ul>
24                 <button
25                     onClick={()=>{ //为按钮添加点击事件
26                         let maxNum = this.state.maxNum + 1;
27                         let list = [...this.state.dataList, maxNum];
28                         this.setState({ //更新状态值
29                             dataList : list,
30                             maxNum : maxNum
31                         });
32                     }>
33                     添加
```



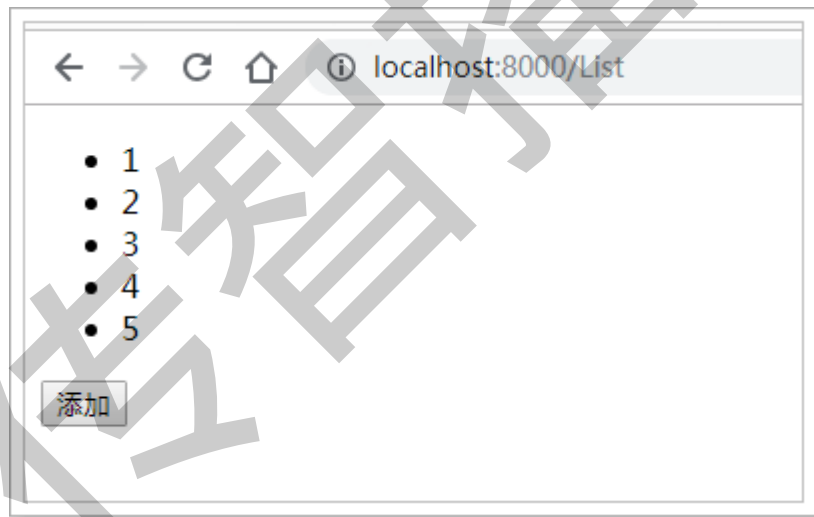


```
34         </button>
35     </div>
36     );
37
38 }
39 }
40
41 export default List;
```

初始状态：



当点击“添加”按钮：



过程分析：



```
super(props); // 调用父类的构造方法
this.state = { // 初始化this.state
  dataList: [1,2,3],
  maxNum: 3
};
render(){
  return (
    <div>
      <ul>
        {
          // 遍历值
          this.state.dataList.map((value,index) => {
            return <li key={index}>{value}</li>
          })
        }
      </ul>
      <button
        onClick={() => { // 为按钮添加点击事件
          let maxNum = this.state.maxNum + 1;
          let list = [...this.state.dataList, maxNum];
          this.setState({ // 更新状态值
            dataList: list,
            maxNum: maxNum
          });
        }}
      >添加
    </button>
  )
}
```

1、遍历初始的state的值，进行页面展现

2、点击按钮，触发该方法执行

3、更新state的值

4、触发render方法执行

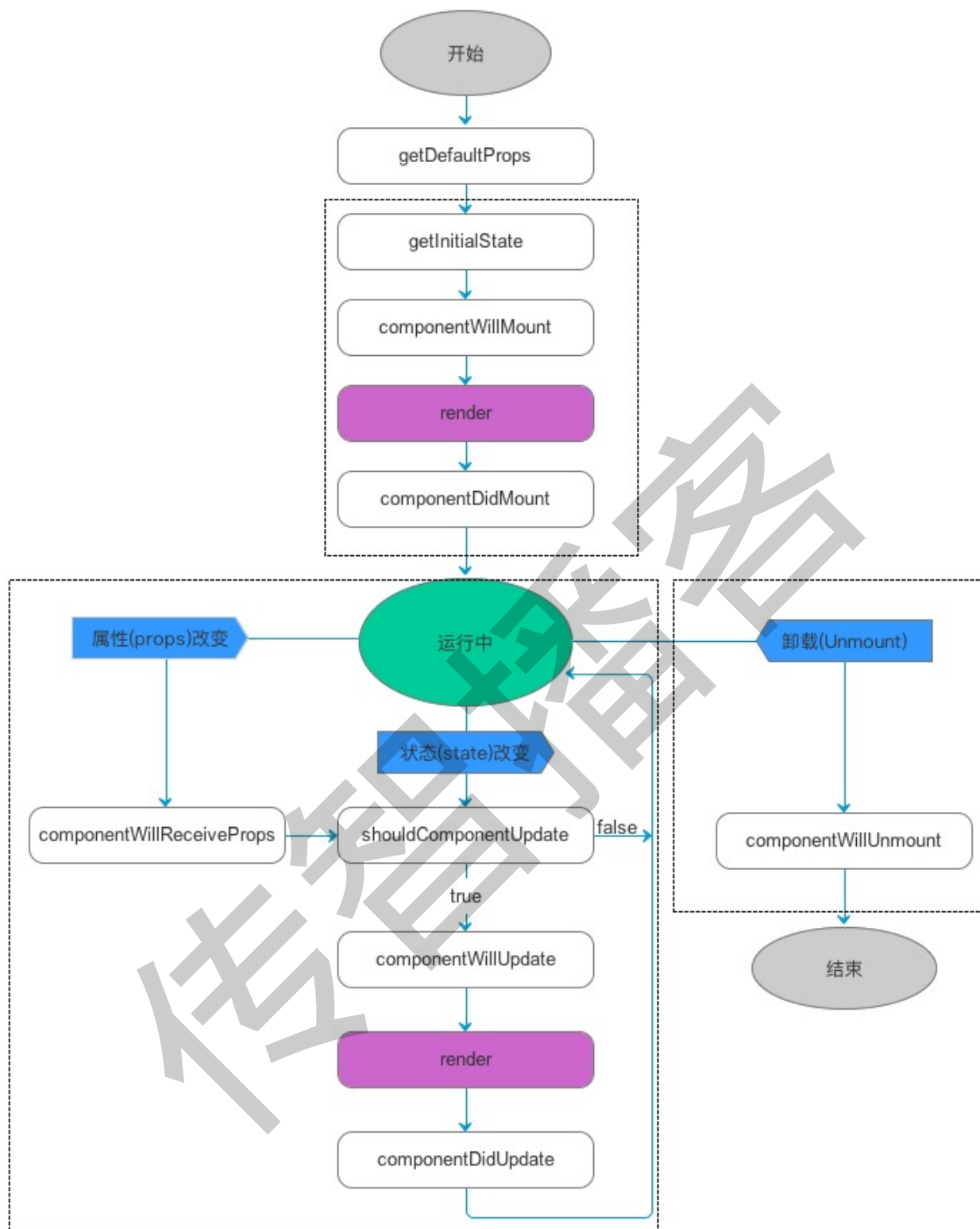
localhost:8000/List

- 1
- 2
- 3
- 4
- 5

添加

#### 2.4.2.4、生命周期

组件的运行过程中，存在不同的阶段。React 为这些阶段提供了钩子方法，允许开发者自定义每个阶段自动执行的函数。这些方法统称为生命周期方法（lifecycle methods）。



生命周期示例：

```
1 import React from 'react'; //第一步，导入React
2
3 class Lifecycle extends React.Component {
4
5     constructor(props) {
```



```
6     super(props);
7     //构造方法
8     console.log("constructor()");
9 }
10
11 componentDidMount() {
12     //组件挂载后调用
13     console.log("componentDidMount()");
14 }
15
16 componentWillUnmount() {
17     //在组件从 DOM 中移除之前立刻被调用。
18     console.log("componentWillUnmount()");
19 }
20
21 componentDidUpdate() {
22     //在组件完成更新后立即调用。在初始化时不会被调用。
23     console.log("componentDidUpdate()");
24 }
25
26 shouldComponentUpdate(nextProps, nextState){
27     // 每当this.props或this.state有变化，在render方法执行之前，就会调用这个方法。
28     // 该方法返回一个布尔值，表示是否应该继续执行render方法，即如果返回false，UI 就不会更新，
    默认返回true。
29     // 组件挂载时，render方法的第一次执行，不会调用这个方法。
30     console.log("shouldComponentUpdate()");
31 }
32
33 render() {
34     return (
35         <div>
36             <h1>React Life Cycle!</h1>
37         </div>
38     );
39 }
40 }
41
42 export default Lifecycle;
```

测试结果：

