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ECMAScript5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ECMAScript6。
- 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

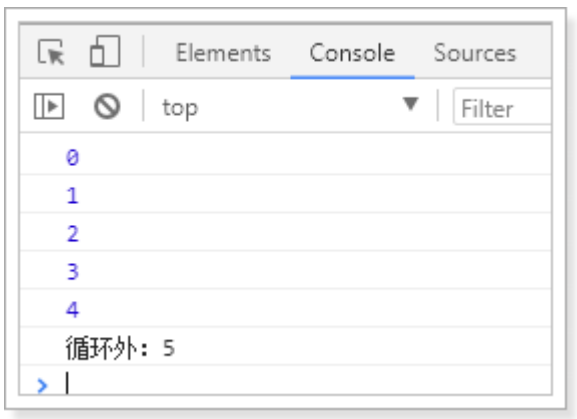
之前，我们写js定义变量的时候，只有一个关键字：`var`

`var`有一个问题，就是定义的变量有时会莫名奇妙的成为全局变量。

例如这样的一段代码：

```
for(var i = 0; i < 5; i++){
    console.log(i);
}
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。

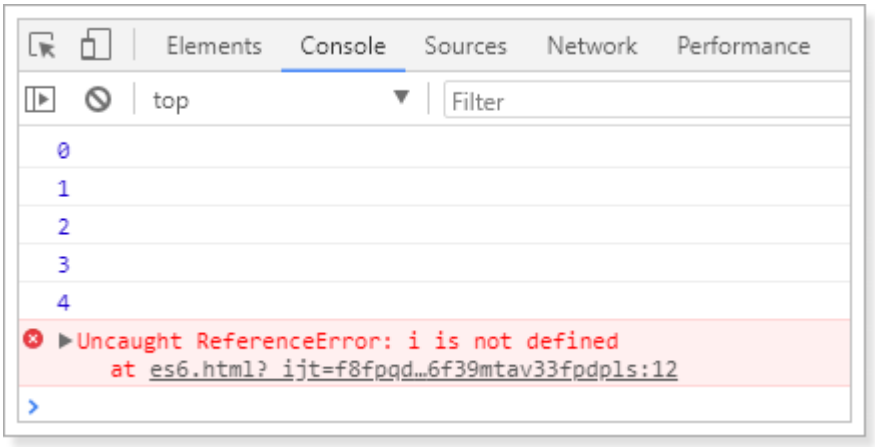
> let

`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：

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

结果：

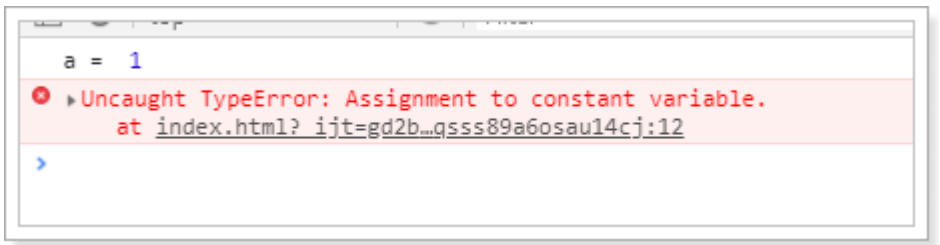


这样，就把变量的i的作用域控制在了循环内部。

> const

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

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



可以看到，变量a的值是不能修改的。

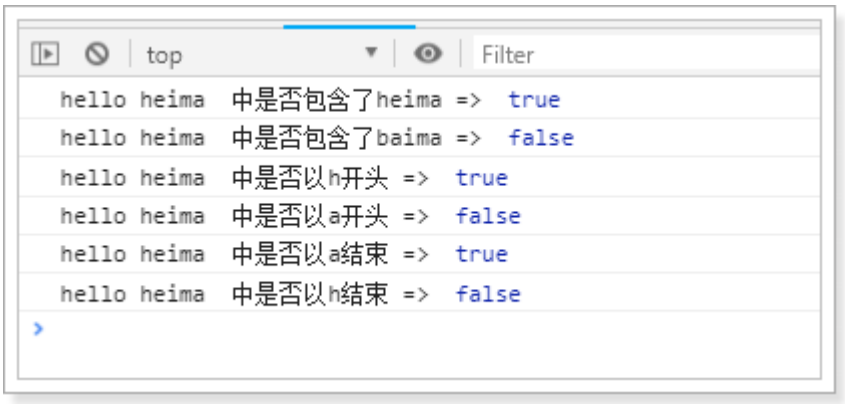
1.3、字符串扩展

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：

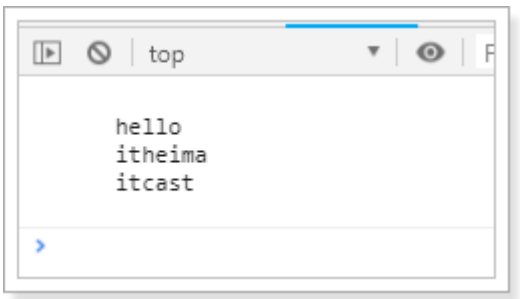
```
<script>
    let str = "hello heima";
    console.log(str, " 中是否包含了heima => ", str.includes("heima"));
    console.log(str, " 中是否包含了baima => ", str.includes("baima"));

    console.log(str, " 中是否以h开头 => ", str.startsWith("h"));
    console.log(str, " 中是否以a开头 => ", str.startsWith("a"));

    console.log(str, " 中是否以a结束 => ", str.endsWith("a"));
    console.log(str, " 中是否以h结束 => ", str.endsWith("h"));
</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：

```
<script>
    let str = `
    hello
    itheima
    itcast
    `;
    console.log(str);

</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

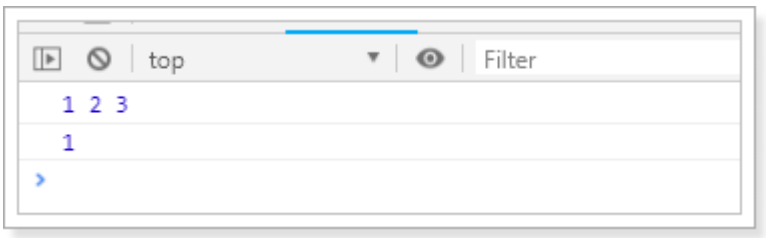
比如有一个数组：

```
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：

```
    let arr = [1,2,3]
    const [x,y,z] = arr;// x，y，z将与arr中的每个位置对应来取值
    // 然后打印
    console.log(x,y,z);

    const [a] = arr; //只匹配1个参数
    console.log(a);
```

结果：



1.3.2、对象解构

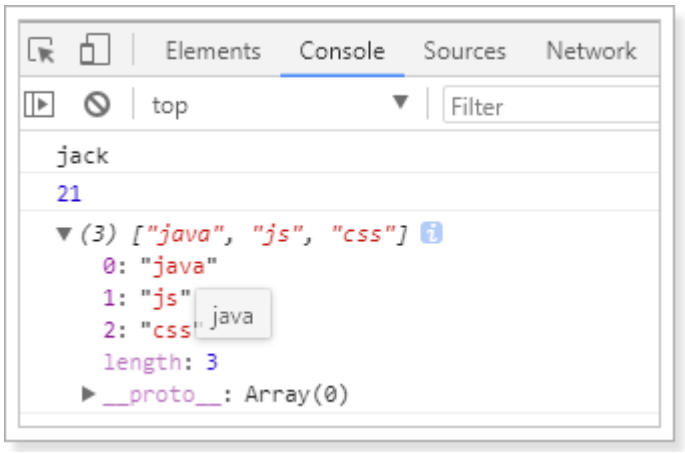
例如有个person对象：

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

我们可以这么做：

```
// 解构表达式获取值
const {name,age,language} = person;
// 打印
console.log(name);
console.log(age);
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：

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

现在可以这么写：

```
function add(a , b = 1) {
    return a + b;
}
// 传一个参数
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：

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

多个参数：

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

没有参数：

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

代码不止一行，可以用`{}`括起来。

```
var sum3 = (a,b) => {
    return a + b;
}

// 多行，没有返回值
let sayHello = () => {
    console.log("hello!");
    console.log("world!");
}

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：

```
let person = {
    name: "jack",
    // 以前：
    eat: function (food) {
        console.log(this.name + "在吃" + food);
    },
    // 箭头函数版：
    eat2: food => console.log(person.name + "在吃" + food),// 这里拿不到this
    // 简写版：
    eat3(food){
        console.log(this.name + "在吃" + food);
    }
}
```

1.4.4、箭头函数结合解构表达式

比如有一个函数：

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

function hello(person) {
    console.log("hello," + person.name)
}
```

如果用箭头函数和解构表达式

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

1.5、map和reduce

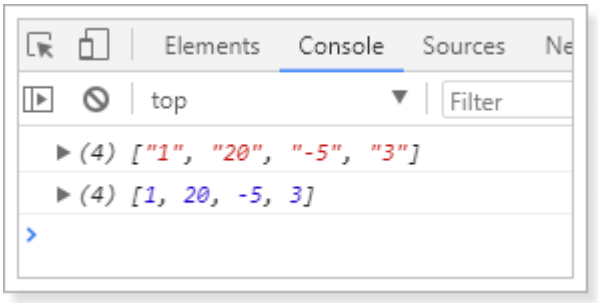
ES6中，数组新增了map和reduce方法。

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

```
let arr = ['1','20','-5','3'];
console.log(arr)

let newArr = arr.map(s => parseInt(s));

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：

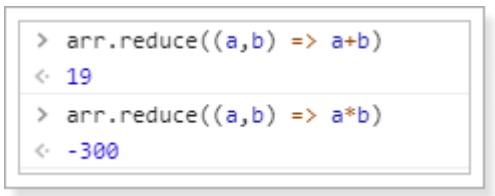
- 第一个参数是上一次reduce处理的结果
- 第二个参数是数组中要处理的下一个元素

`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

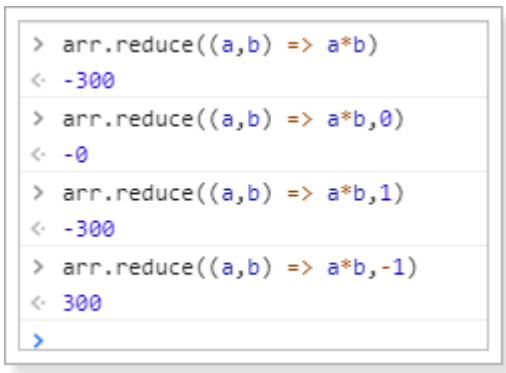
举例：

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

没有初始值：



指定初始值：



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：

```
    console.log (...[1, 2, 3]); //1 2 3
    console.log(1, ...[2, 3, 4], 5); // 1 2 3 4 5


    function add(x, y) {
        return x + y;
    }
    var numbers = [1, 2];
    console.log(add(...numbers)); // 3

    // 数组合并
    let arr = [...[1,2,3],...[4,5,6]];
    console.log(arr); //[1, 2, 3, 4, 5, 6]

    // 与解构表达式结合
    const [first, ...rest] = [1, 2, 3, 4, 5];
    console.log(first, rest) //1  [2, 3, 4, 5]

    //将字符串转成数组
    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：

```
const promise = new Promise(function(resolve, reject) {
  // ... 执行异步操作

  if (/* 异步操作成功 */){
    resolve(value);// 调用resolve，代表Promise将返回成功的结果
  } else {
    reject(error);// 调用reject，代表Promise会返回失败结果
  }
});
```

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：

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

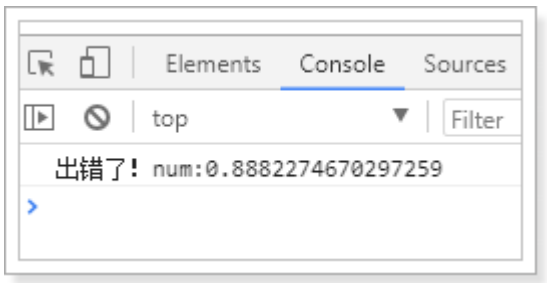
如果想要处理promise异步执行失败的事件，还可以跟上catch：

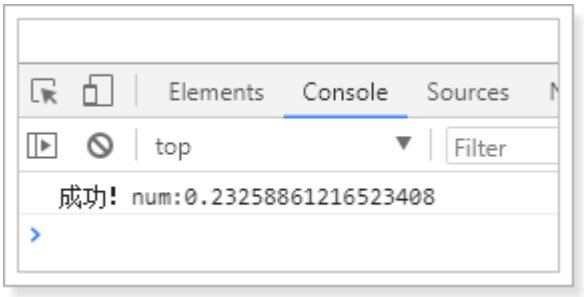
```
promise.then(function(value){
    // 异步执行成功后的回调
}).catch(function(error){
    // 异步执行失败后的回调
})
```

示例：

```
const p = new Promise(function (resolve, reject) {
    // 这里我们用定时任务模拟异步
    setTimeout(() => {
        const num = Math.random();
        // 随机返回成功或失败
        if (num < 0.5) {
            resolve("成功！num:" + num)
        } else {
            reject("出错了！num:" + num)
        }
    }, 300)
})

// 调用promise
p.then(function (msg) {
    console.log(msg);
}).catch(function (msg) {
    console.log(msg);
})
```

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：

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

方法：

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

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：

```
// map接收一个数组，数组中的元素是键值对数组
const map = new Map([
    ['key1','value1'],
    ['key2','value2'],
])
// 或者接收一个set
const set = new Set([
    ['key1','value1'],
    ['key2','value2'],
])
const map2 = new Map(set)
// 或者其它map
const map3 = new Map(map);
```

方法：

```
map.set(key, value);// 添加
map.clear();// 清空
map.delete(key);// 删除指定元素
map.has(key); // 判断是否存在
map.forEach(function(key,value){})//遍历元素
map.size; // 元素个数。是属性，不是方法

map.values() //获取value的迭代器
map.keys() //获取key的迭代器
map.entries() //获取entry的迭代器
用法：
for (let key of map.keys()) {
    console.log(key);
}
或：
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：

```
<script>

    class User{
        constructor(name, age = 20){ // 构造方法
            this.name = name; // 添加属性并且赋值
            this.age = age;
        }

        sayHello(){ // 定义方法
            return "hello";
        }
```

```
        static isAdult(age){ //静态方法
            if(age >= 18){
                return "成年人";
            }
            return "未成年人";
        }
    }

    let user = new User("张三");

    // 测试
    console.log(user); // User {name: "张三", age: 20}
    console.log(user.sayHello()); // hello
    console.log(User.isAdult(20)); // 成年人


</script>
```

类的继承：

```
<script>

    class User{
        constructor(name, age = 20){ // 构造方法
            this.name = name; // 添加属性并且赋值
            this.age = age;
        }

        sayHello(){
            return "hello"; // 定义方法
        }

        static isAdult(age){ //静态方法
            if(age >= 18){
                return "成年人";
            }
            return "未成年人";
        }
    }

    class ZhangSan extends User{
        constructor(){
            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
            this.address = "上海";//设置子类中的属性，位置必须处于super下面
        }
    }

    // 测试
    let zs = new ZhangSan();
    console.log(zs.name, zs.address);
    console.log(zs.sayHello());
    console.log(ZhangSan.isAdult(20));
```

```
</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吾句定义不同的内部状态。

用法：

```
<script>

    function* hello () {
        yield "hello";
        yield "world";
        return "done";
    }

    let h = hello();

    console.log(h.next()); //{value: "hello", done: false}
    console.log(h.next()); //{value: "world", done: false}
    console.log(h.next()); //{value: "done", done: true}
    console.log(h.next()); //{value: undefined, done: true}


</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：

```
<script>

    function* hello () {
        yield "hello";
        yield "world";
        return "done";
    }

    let h = hello();

    for (let obj of h) {
        console.log(obj);
    }
```

```
</script>

// 输出：
hello
world
```

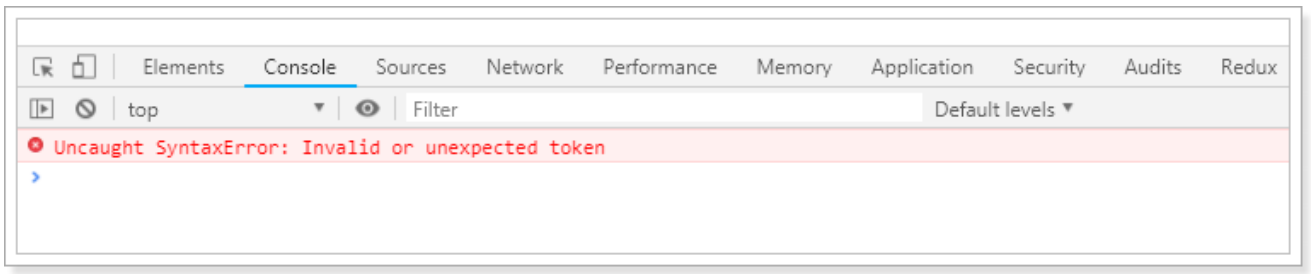
1.11、修饰器(Decorator)

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：

```
<script>

    @T //通过@符号进行引用该方法，类似java中的注解
    class User {
        constructor(name, age = 20){
            this.name = name;
            this.age = age;
        }
    }

    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User类添加一个静态属性country
    }

    console.log(User.country); //打印出country属性值

</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：

```
//转码前
input .map(item =>item + 1);

//转码后
input.map(function (item) {
    return item + 1;
})
```

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 Cgithub.com/google/traceur-compiler)， 也可 以将 ES6 代码转为ES5的代码。

这2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：

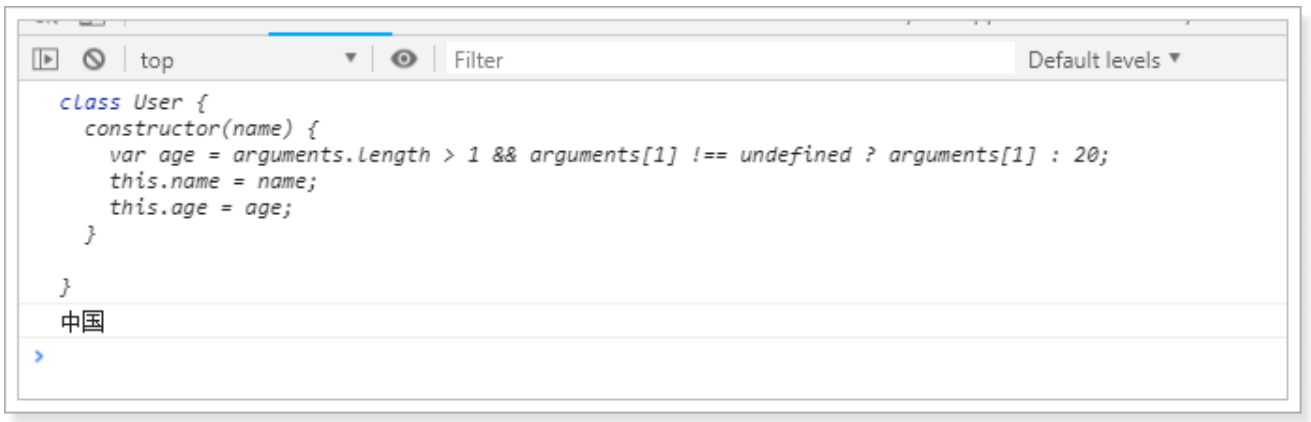
- 插件化
  - umi 的整个生命周期都是插件化的，甚至其内部实现就是由大量插件组成，比如 pwa、按需加载、一键切换 preact、一键兼容 ie9 等等，都是由插件实现。
- 开箱即用
  - 你只需一个 umi 依赖就可启动开发，无需安装 react、preact、webpack、react-router、babel、jest 等等。
- 约定式路由
  - 类 next.js 的约定式路由，无需再维护一份冗余的路由配置，支持权限、动态路由、嵌套路由等等。

1.12.2、部署安装

```
#首先，需要安装Node.js
#在资料中，找到node-v8.12.0-x64.msi，一路下一步安装
#安装完成后，通过node -v 命令查看其版本号

F:\code\itcast-es6>node -v
v8.12.0

#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
#可以把yarn看做了优化了的npm
npm i yarn tyarn -g  #-g 是指全局安装
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
#如果安装失败，是由于将yarn添加到环境变量中导致，参见
http://www.easysb.cn/index.php/2017/06/04/11/

#下面开始安装umi
tyarn global add umi
umi #进行测试
```

1.12.3、快速入门

```
#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。
tyarn init -y

#通过umi命令创建index.js文件
umi g page index  #可以看到在pages下创建好了index.js和index.css文件

#将下面内存拷贝到index.js文件中进行测试
    @T //通过@符号进行引用该方法，类似java中的注解
    class User {
        constructor(name, age = 20){
            this.name = name;
            this.age = age;
        }
    }

    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User类添加一个静态属性country
    }

    console.log(User.country); //打印出country属性值

#通过命令行启动umi的后台服务,用于本地开发
umi dev
#通过浏览器进行访问：http://localhost:8000/，查看效果
#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
```



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User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：

```
class Util {
    static sum = (a, b) => a + b;
}

//导出该类
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：

```
//Index.js
//导入Util类
import Util from './Util'

//使用Util中的sum方法
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

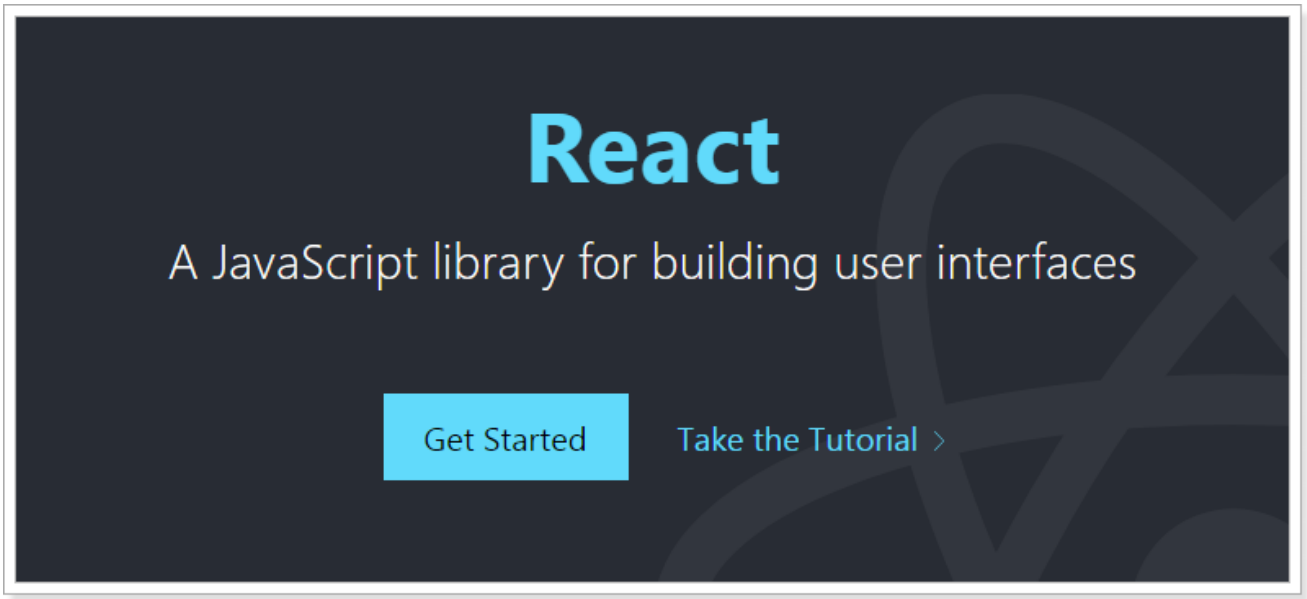
前端可以做到读写数据、切换视图、用户交互，这意味着，网页其实是一个应用程序，而不是信息的纯展示。这种单张网页的应用程序称为 SPA（single-page-application）。

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：

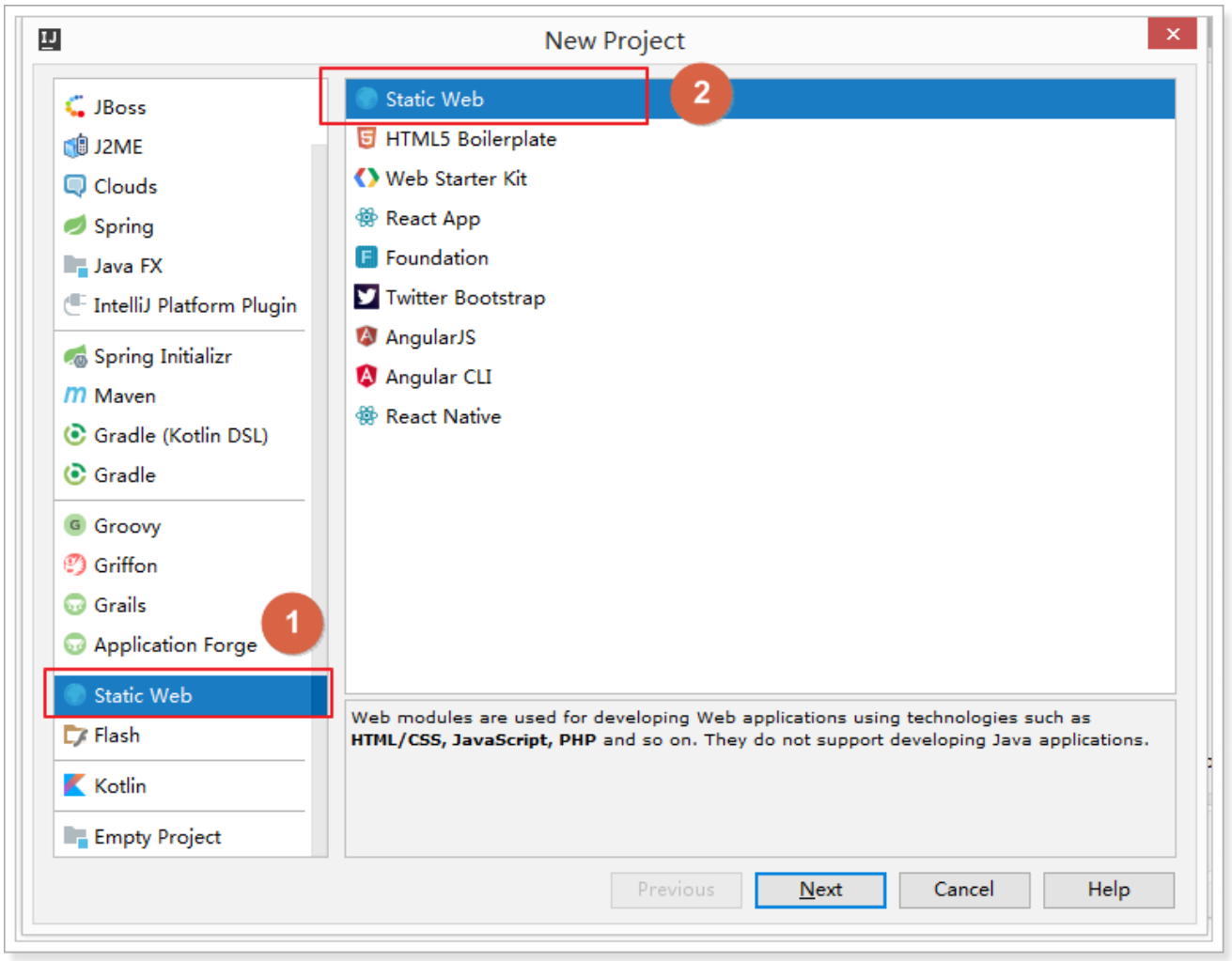
- Flux
  - Flux是Facebook用户建立客户端Web应用的前端架构， 它通过利用一个单向的数据流补充了React的组合视图组件，这更是一种模式而非框架。
- Redux
  - Redux 是 JavaScript 状态容器，提供可预测化的状态管理。Redux可以让React组件状态共享变得简单。
- Ant Design of React
  - 阿里开源的基于React的企业级后台产品，其中集成了多种框架，包含了上面提到的Flux、Redux。
  - Ant Design提供了丰富的组件，包括：按钮、表单、表格、布局、分页、树组件、日历等。

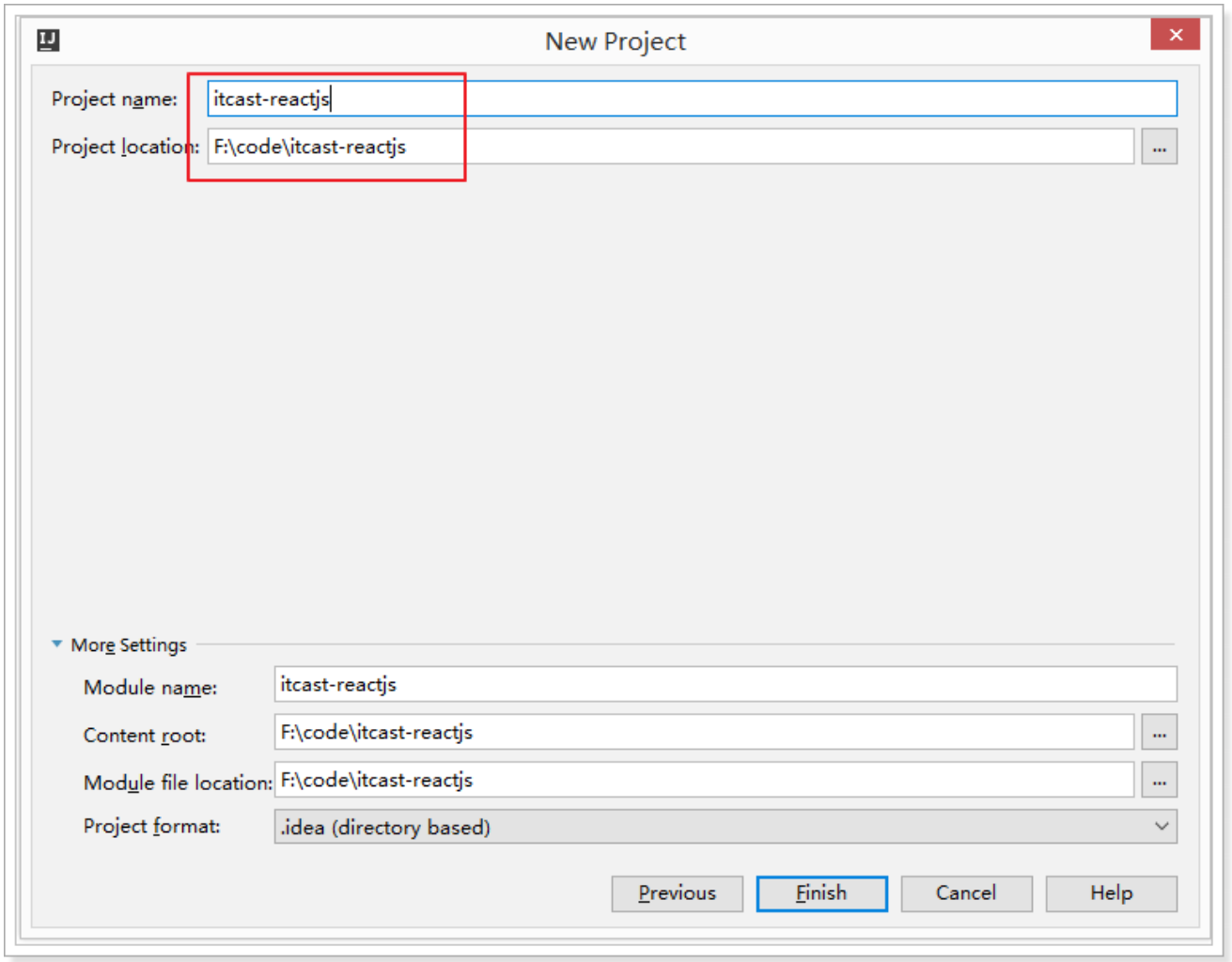
2.3、搭建环境

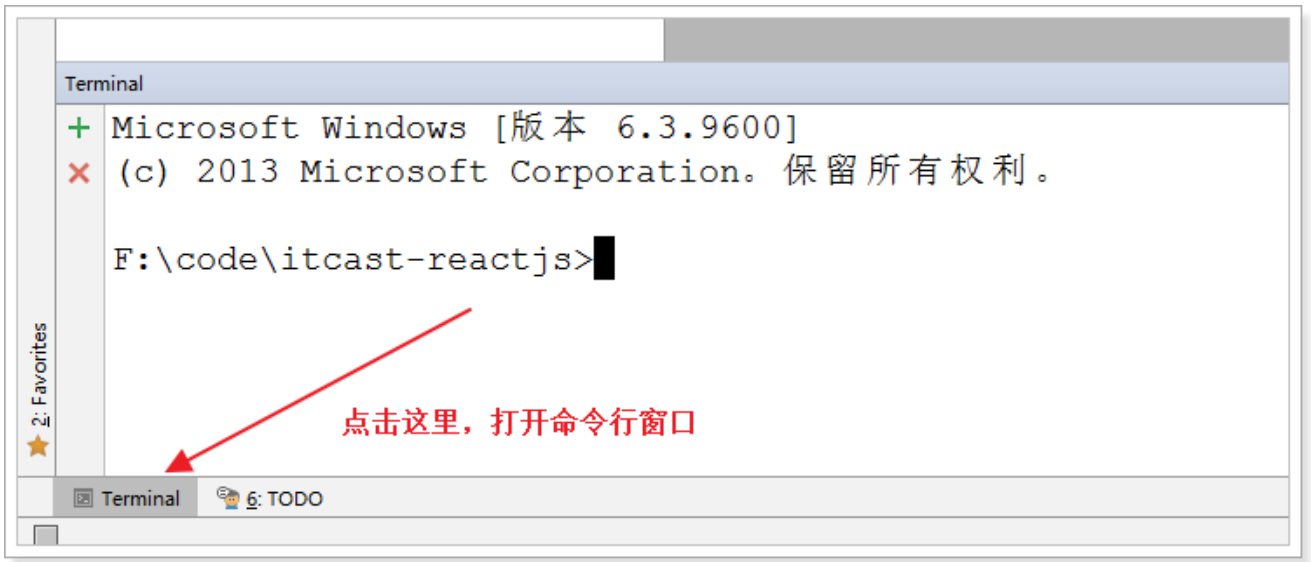
2.3.1、创建项目

我们依然选择使用UmiJS作为构建工具。

创建工程：



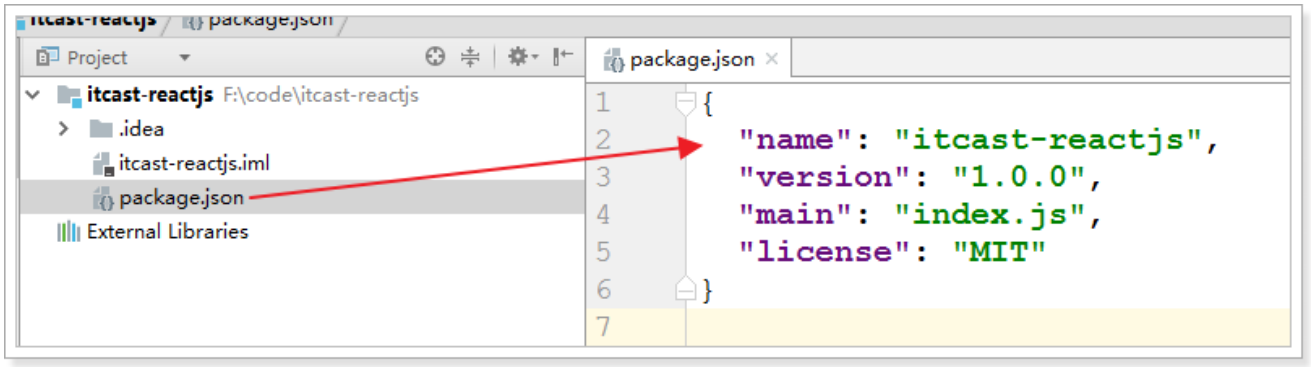




输入命令，进行初始化：

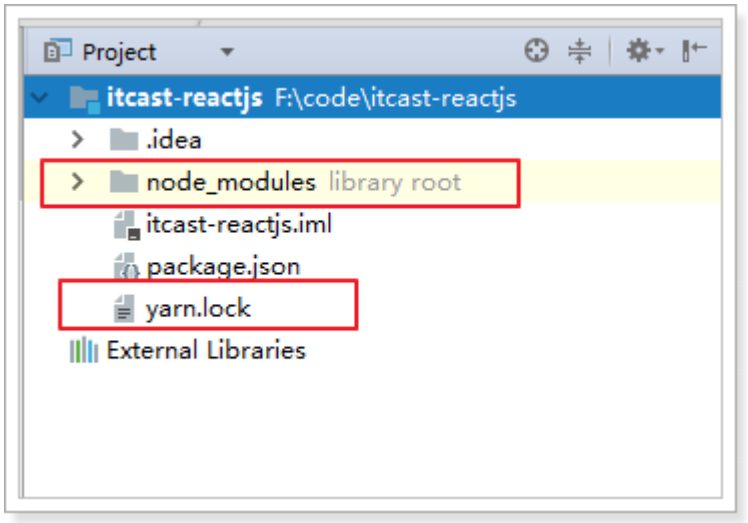
```
tyarn init -y
```

初始化完成：



在命令输入如下命令：

```
tyarn add umi --dev  #项目中添加umi的依赖
```

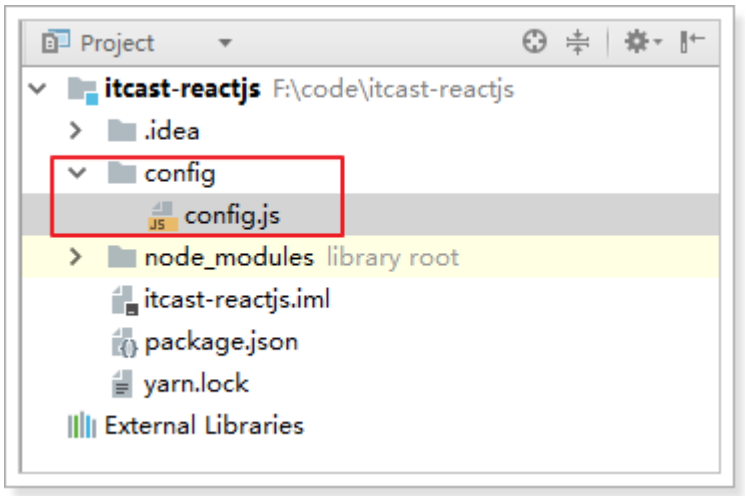


可以看到，相关的依赖已经导入进来了。

2.3.2、编写HelloWorld程序

> 第一步，在工程的根目录下创建config目录，在config目录下创建config.js文件。

在UmiJS的约定中，config/config.js将作为UmiJS的全局配置文件。



> 在umi中，约定的目录结构如下：

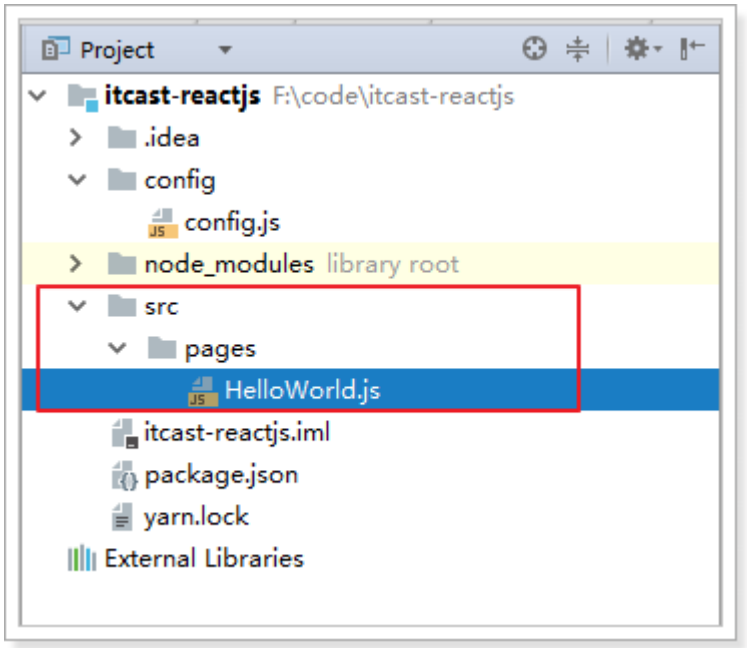
```
.
├── dist/                       // 默认的 build 输出目录
├── mock/                       // mock 文件所在目录，基于 express
├── config/
    ├── config.js               // umi 配置，同 .umirc.js，二选一
└── src/                        // 源码目录，可选
    ├── layouts/index.js        // 全局布局
    ├── pages/                  // 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：

```
//导出一个对象，暂时设置为空对象，后面再填充内容
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。



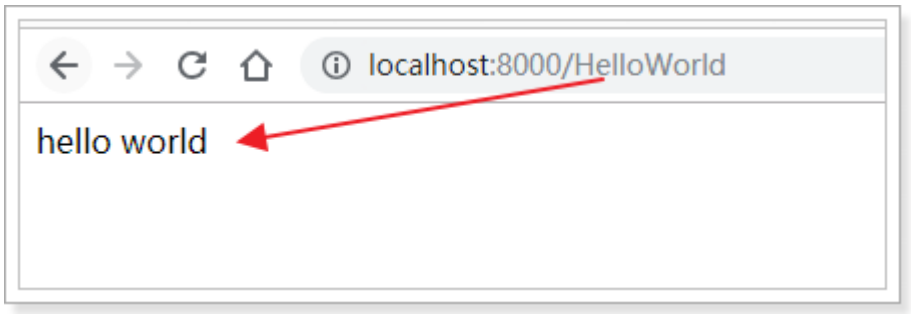
在HelloWorld.js文件中输入如下内容：

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

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JSX，后面我们再细说。

第三步，启动服务查看页面效果

```
#启动服务
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。

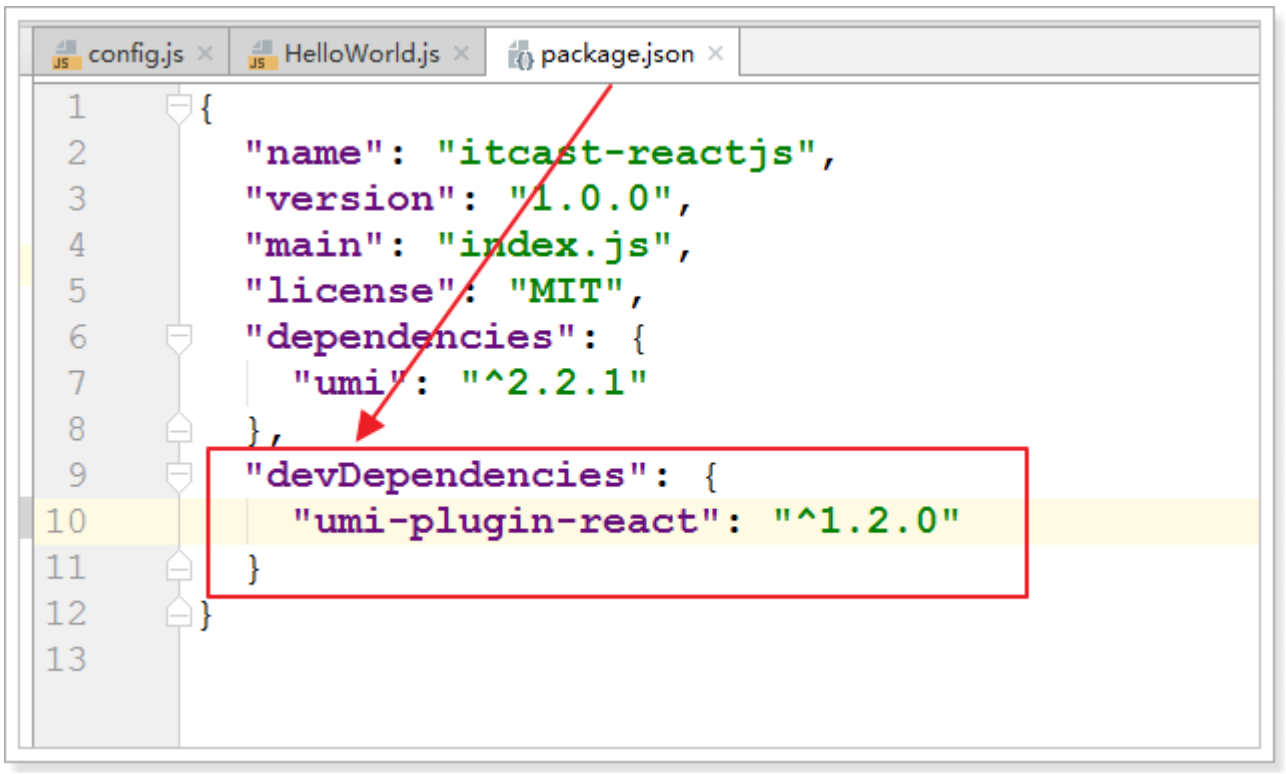
2.3.3、添加umi-plugin-react插件

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

```
#添加插件
tyarn add umi-plugin-react --dev
```

添加成功：



接下来，在config.js文件中引入该插件：

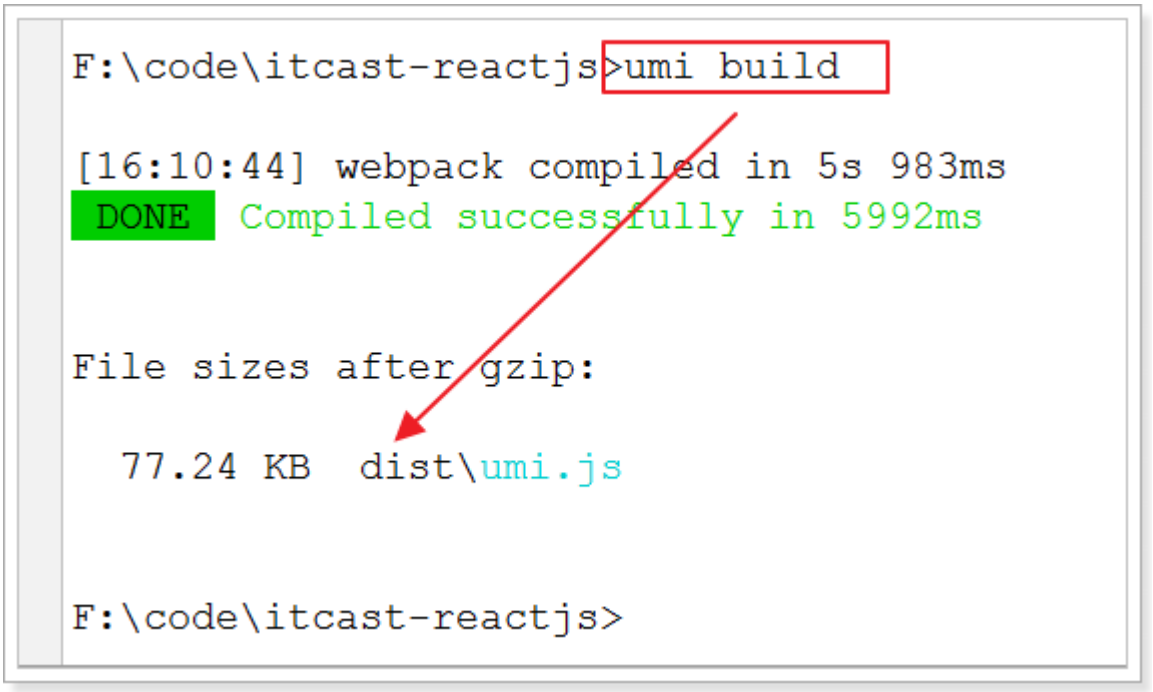
```
export default {
    plugins: [
        ['umi-plugin-react', {
            //暂时不启用任何功能
        }]
    ]
};
```

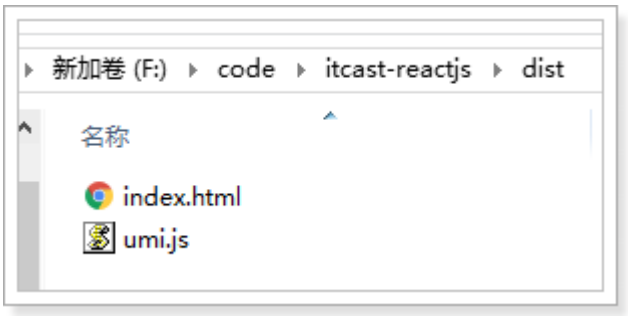
2.3.4、构建和部署

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？

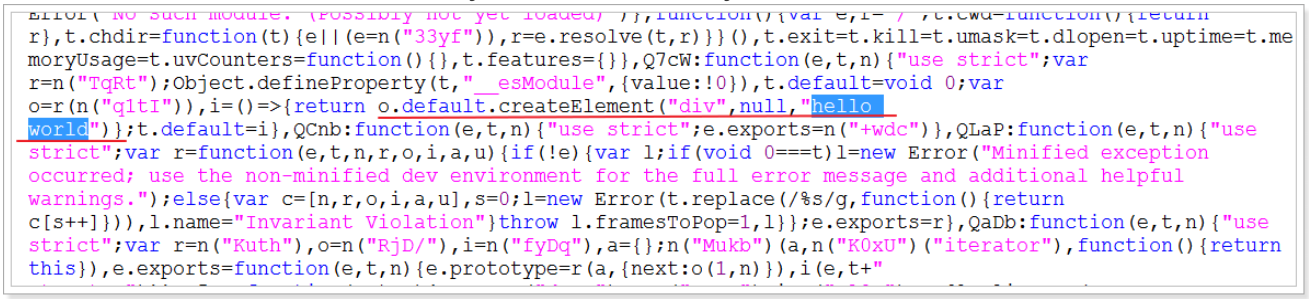
其实，通过umi是可以进行转码生成文件的，具体操作如下：

```
umi build
```





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。



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：

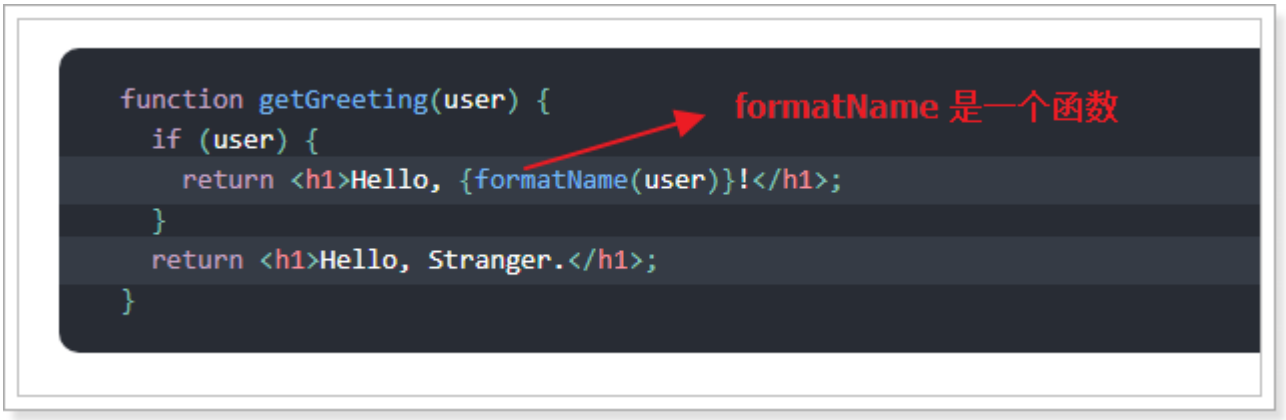
1. 所有的html标签必须是闭合的，如：

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。

```
const div1 = <div>hello world</div>  //正确
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。



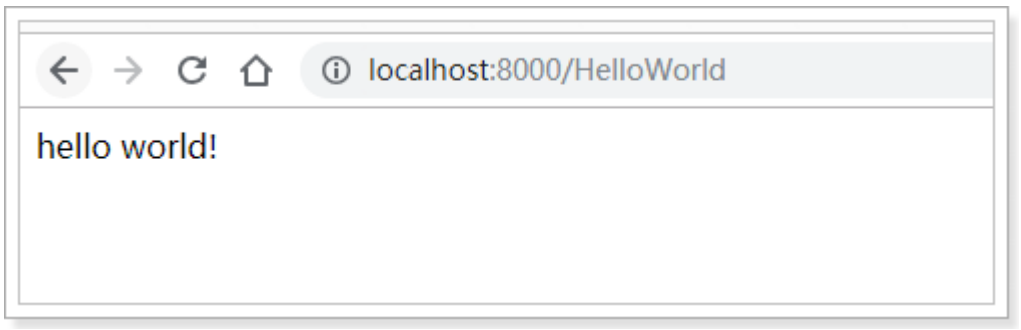
2.4.2、组件

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：

```
import React from 'react'; //第一步，导入React

class HelloWorld extends React.Component { //第二步，编写类并且继承 React.Component

    render(){ //第三步，重写render()方法，用于渲染页面
        return <div>hello world!</div> //JSX语法
    }

}

export default HelloWorld; //第四步，导出该类
```

查看效果：

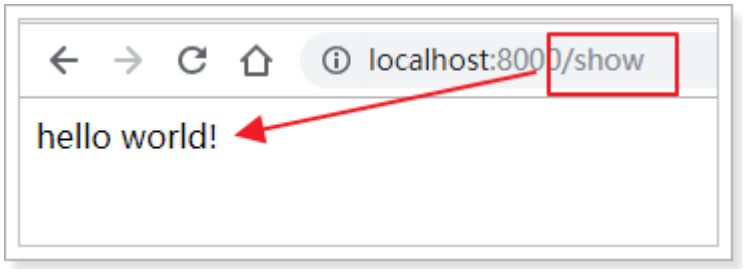


2.4.2.1、导入自定义组件

创建Show.js文件，用于测试导入组件：

```
import React from 'react'
import HelloWorld from './HelloWorld' //导入HelloWorld组件

class Show extends React.Component{

    render(){
        return <HelloWorld/>; //使用HelloWorld组件
    }

}

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：

```
import React from 'react'
import HelloWorld from './HelloWorld' //导入HelloWorld组件

class Show extends React.Component{

    render(){
        return <HelloWorld name="zhangsan">shanghai</HelloWorld>; //使用HelloWorld组件
    }

}

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：

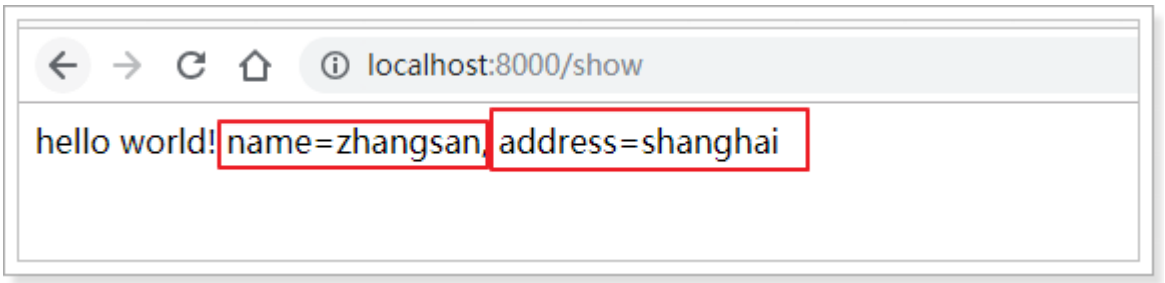
属性：this.props.name 接收；

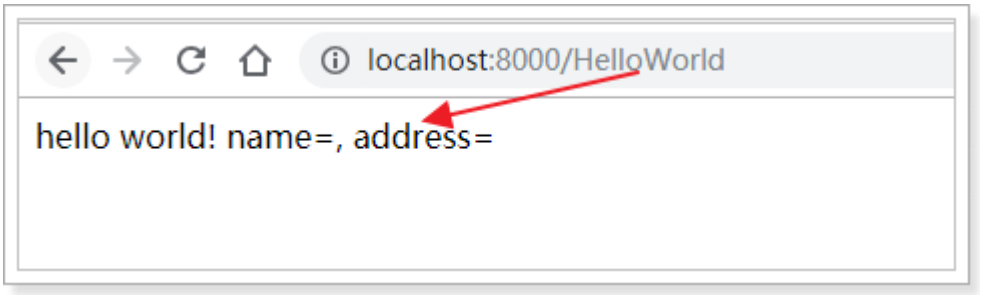
标签内容：this.props.children 接收；

使用如下：

```
import React from 'react'; //第一步，导入React

class HelloWorld extends React.Component { //第二步，编写类并且继承 React.Component

    render(){ //第三步，编写render()方法，用于渲染页面
        return <div>hello world! name={this.props.name}, address={this.props.children}
</div> //JSX语法
    }

}

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；

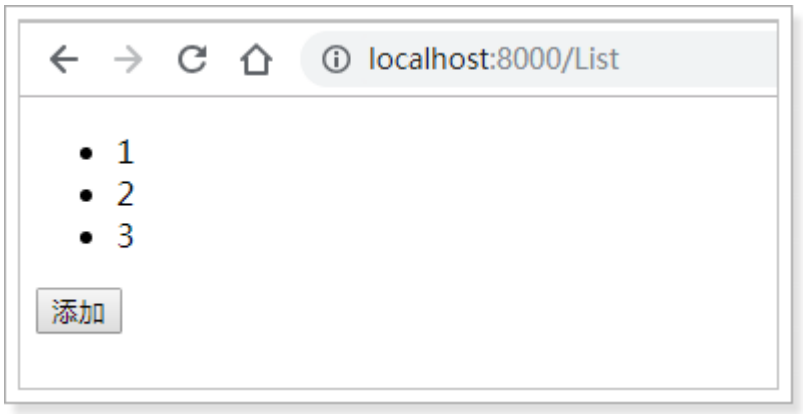
二：要修改this.state的值，需要调用this.setState()完成，不能直接对this.state进行修改；

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。

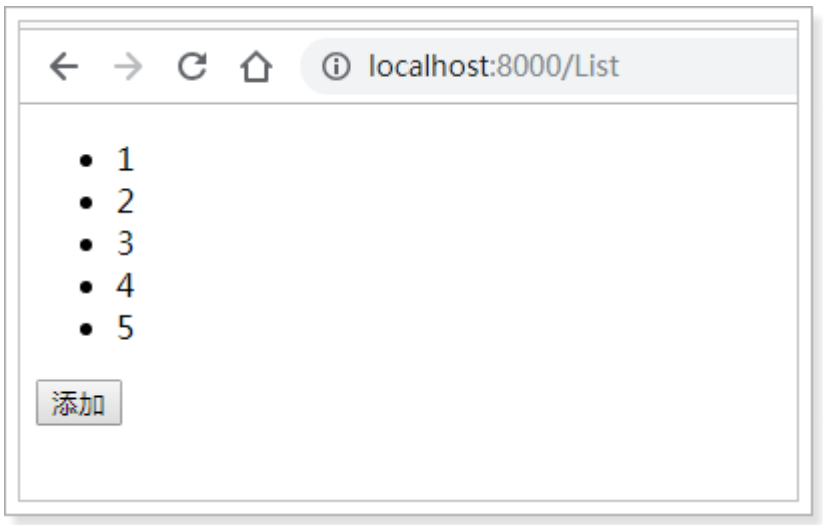
```
import React from 'react'

class List extends React.Component{

    constructor(props){ // 构造参数中必须要props参数
        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>
            </div>
        );

    }
}

export default List;
```

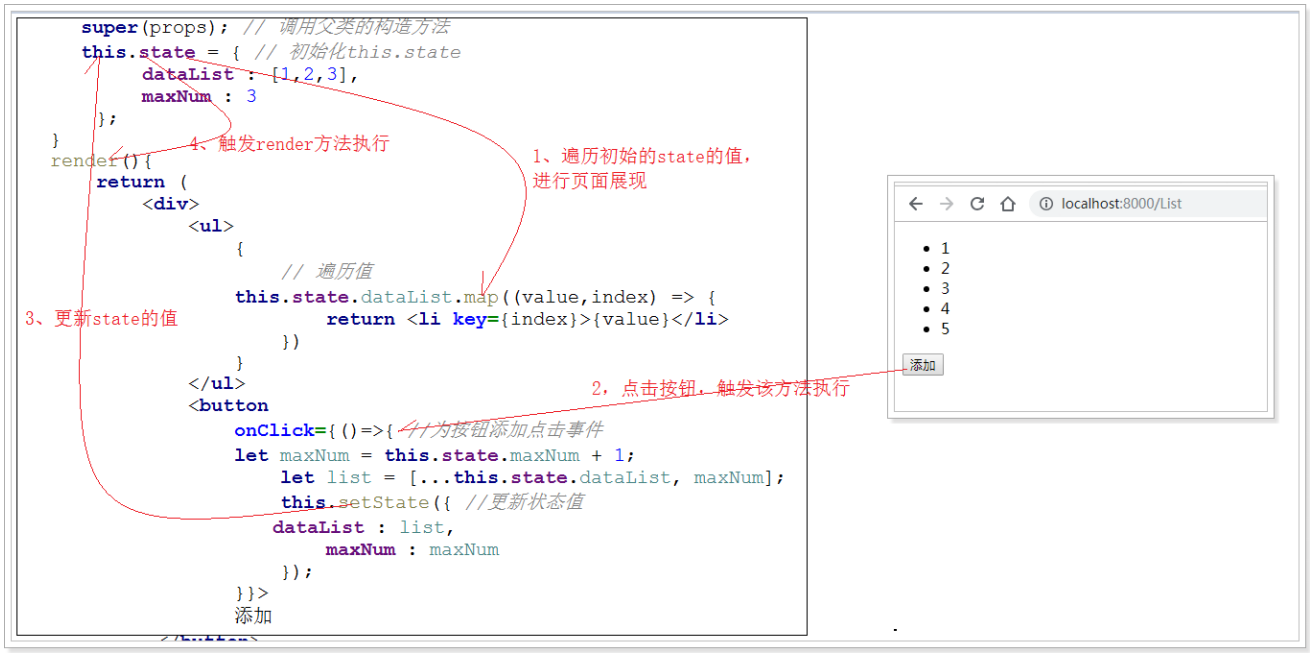
初始状态：



当点击“添加”按钮：



过程分析：



2.4.2.4、生命周期

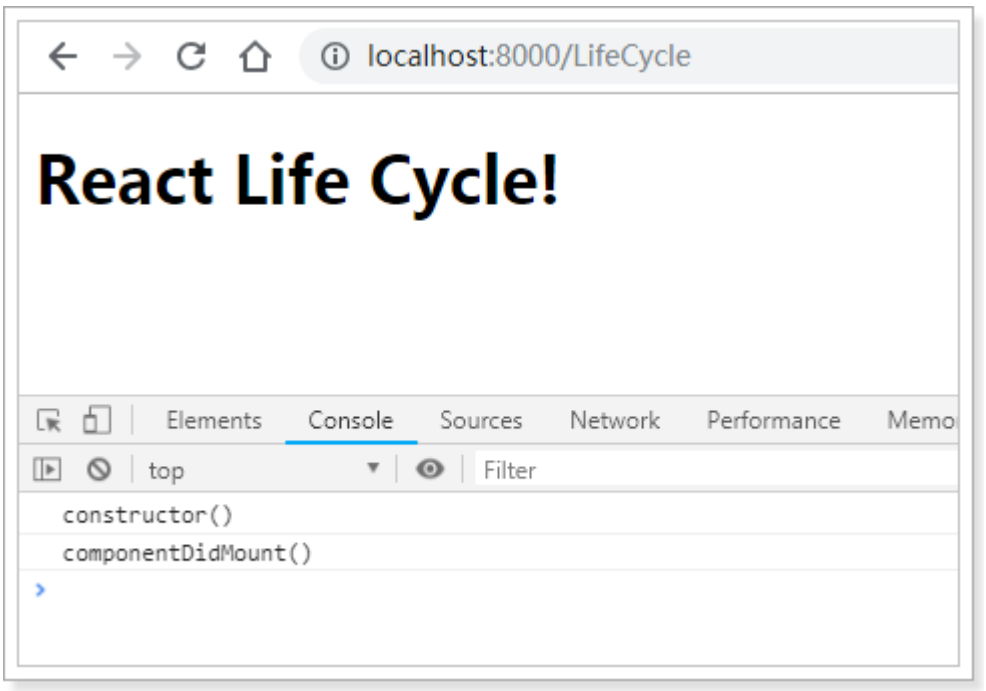
组件的运行过程中，存在不同的阶段。React 为这些阶段提供了钩子方法，允许开发者自定义每个阶段自动执行的函数。这些方法统称为生命周期方法（lifecycle methods）。

生命周期示例：

```
import React from 'react'; //第一步，导入React

class LifeCycle extends React.Component {

    constructor(props) {
        super(props);
```

```
        //构造方法
        console.log("constructor()");
    }

    componentDidMount() {
        //组件挂载后调用
        console.log("componentDidMount()");
    }

    componentWillUnmount() {
        //在组件从 DOM 中移除之前立刻被调用。
        console.log("componentWillUnmount()");
    }

    componentDidUpdate() {
        //在组件完成更新后立即调用。在初始化时不会被调用。
        console.log("componentDidUpdate()");
    }

    shouldComponentUpdate(nextProps, nextState){
        // 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。
        // 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。
        // 组件挂载时，render方法的第一次执行，不会调用这个方法。
        console.log("shouldComponentUpdate()");
    }

    render() {
        return (
            <div>
                <h1>React Life Cycle!</h1>
            </div>
        );
    }
}

export default LifeCycle;
```

测试结果：


localhost:8000/LifeCycle
React Life Cycle!
Elements
Console
Sources
Network
Performance
Memo
top
Filter
constructor()
componentDidMount()