# Web前端

# HTML

HTML 不是一门编程语言，而是一种用于**定义内容结构的标记语言**。

<img src="images/firefox-icon.png" alt="测试图片">

alt： alternative 可替代的页面加载过程：(HTML，JS,CSS,图片，文本，接口等)

URL解析--->DNS查询--->资源请求—>浏览器解析

URL解析：解析URL提取里面的信息

协议+域名+端口号+ 路径+参数+哈希

DNS查询（DNS缓存+DNS服务器）：拿到上一步的域名去DNS拿到对应的IP（域名转为IP地址）

DNS优化： dns-prefetch页面一加载就去做DNS查询并缓存起来，提高页面加载数据的速度

资源请求：带着url信息去IP地址请求资源，从服务器把请求的资源下载下来

浏览器（request header （cookie）+ 参数(url或者参数)）

服务器返回（） status+ response-header + body

浏览器解析：最先加载HTML，遇到一个节点就放到树里，js优先级高是因为js可以操作DOM

DOM------>DOM---->渲染树-（DOM树和渲染树结合）--->布局（根据元素的大小和位置计算出实际位置）------>绘制

HTTP无状态的请求

cookie

访问：document.cookie

修改：document.cookie=””

session

跨域：

跨域，是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对JavaScript实施的安全限制。

# CSS

CSS思想： 一切皆盒子

## 加载CSS的三种方式

外部样式表(External style sheet) <link rel="stylesheet" type="text/css" href="mystyle.css">

内部样式表(Internal style sheet)

<style>

hr {color:sienna;}

p {margin-left:20px;}

body {background-image:url("images/back40.gif");}

</style>

内联样式(Inline style) <p style="color:sienna;margin-left:20px">这是一个段落。</p>

优先级： **内联样式）Inline style > （内部样式）Internal style sheet >（外部样式）External style sheet > 浏览器默认样式**

***注意：****如果外部样式放在内部样式的后面，则外部样式将覆盖内部样式。*

选择器的类型

元素选择器

ID选择器

类选择器

属性选择器

伪类选择器

## 块级元素和内联元素

块级元素：独占一行；宽度和高度，内边距，外边距都可以控制

内联元素：和相邻元素在同一行，宽度(width)、高度(height)、内边距的top/bottom(padding-top/padding-bottom)和外边距的top/bottom(margin-top/margin-bottom)都不可改变，就是里面文字或图片的大小;

display:inline-block -- 显示为内联块元素，表现为同行显示并可修改宽高内外边距等属性

我们常将<ul>元素加上display:inline-block样式，原本垂直的列表就可以水平显示了。

## 定位

Position属性的5个值

* [static](https://www.runoob.com/css/css-positioning.html#position-static)： HTML 元素的默认值，即没有定位，遵循正常的文档流对象。
* [relative](https://www.runoob.com/css/css-positioning.html#position-relative)： 相对定位元素的定位是相对其正常位置。
* [fixed](https://www.runoob.com/css/css-positioning.html#position-fixed)： 元素的位置**相对于浏览器窗口**是固定位置。
* [absolute](https://www.runoob.com/css/css-positioning.html#position-absolute)：绝对定位的元素的位置相对于**最近的已定位父元素**，如果元素没有已定位的父元素，那么它的位置相对于<html>
* [sticky](https://www.runoob.com/css/css-positioning.html#position-sticky)： 基于用户的滚动位置来定位

z-index属性指定了一个元素的堆叠顺序（哪个元素应该放在前面，或后面）**注意：** 如果两个定位元素重叠，没有指定z - index，最后定位在HTML代码中的元素将被显示在最前面。

## BFC

<https://juejin.im/entry/59c3713a518825396f4f6969>

<https://juejin.im/post/59b73d5bf265da064618731d>

### BFC的约束规则

* 内部的Box会在垂直方向上一个接一个的放置
* 垂直方向上的距离由margin决定。（完整的说法是：属于同一个BFC的两个相邻Box的margin会发生重叠（塌陷），与方向无关。）
* 每个元素的左外边距与包含块的左边界相接触（从左向右），即使浮动元素也是如此。（这说明BFC中子元素不会超出他的包含块，而position为absolute的元素可以超出他的包含块边界）
* BFC的区域不会与float的元素区域重叠
* 计算BFC的高度时，浮动子元素也参与计算
* BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面元素，反之亦然

**一个元素不能同时存在于两个BFC中**

#### BFC的创建方法

* **根元素**或其它包含它的元素；
* **浮动** (元素的float不为none)；
* **绝对定位元素** (元素的position为absolute或fixed)；
* **行内块**inline-blocks(元素的 display: inline-block)；
* **表格单元格**(元素的display: table-cell，HTML表格单元格默认属性)；
* overflow的值不为visible的元素；
* **弹性盒 flex boxes** (元素的display: flex或inline-flex)；

作者：考拉海购前端团队  
链接：https://juejin.im/post/59b73d5bf265da064618731d  
来源：掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 水平垂直居中

<https://juejin.im/post/5854e137128fe100698e6271>

水平居中：

* 块级元素
  + margin:auto; 同时需要设置width属性为特点值或者100%
  + 图片元素的对齐： img{display:block; margin:auto;width:40%;}

## 清除浮动的方式

（1）父级div定义 height  
原理：父级div手动定义height，就解决了父级div无法自动获取到高度的问题。   
优点：简单、代码少、容易掌握   
缺点：只适合高度固定的布局，要给出精确的高度，如果高度和父级div不一样时，会产生问题

（2）父级div定义 overflow:hidden  
原理：必须定义width或zoom:1，同时不能定义height，使用overflow:hidden时，浏览器会自动检查浮动区域的高度   
优点：简单、代码少、浏览器支持好

（3） 结尾处加空div标签 clear:both  
原理：添加一个空div，利用css提高的clear:both清除浮动，让父级div能自动获取到高度   
优点：简单、代码少、浏览器支持好、不容易出现怪问题  
缺点：不少初学者不理解原理；如果页面浮动布局多，就要增加很多空div，让人感觉很不好

## 弹性盒子：

提供一种更加有效的方式来对一个容器中的子元素进行排列、对齐和分配空白空间

是适应不同的屏幕大小以及设备类型时确保元素拥有恰当的行为的布局方式

# JavaScript

## null和undefined的区别

* null 是关键字， undefined不是关键字
* null是对象但是值为空，undefined的本质是全局对象的属性，这个属性的值叫做“未定义”
* null是已经初始化的，undefined是未初始化的
* 使用typeof返回一个是object, 一个是undefined

typeof 返回值(number, string, Boolean, undefined,object, function）

isNaN()，返回布尔类型，**用途是判断参数是否可以转换成数字**

## 变量提升

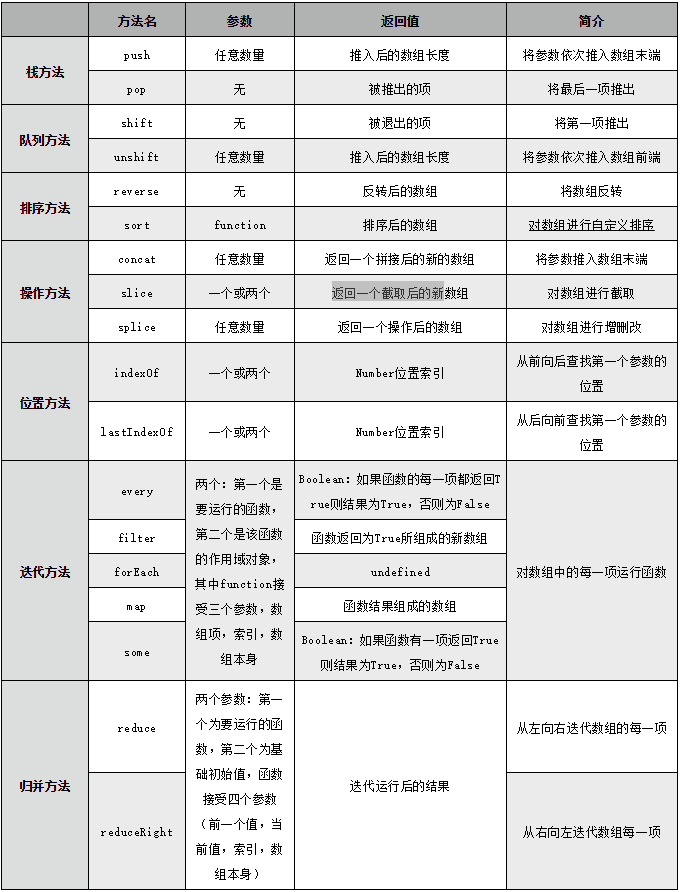
Let 和const 存在的原因

* 解决变量提升带来的缺陷

## 数组操作

检测数组的方法：

1. arr instanceof Array
2. vaue.isArray()
3. Object.prototype.toString.call(value)==’[object Array]’



1. concat 以及用ES6实现

var arr = [1,2,3,4,5,6]

var arr2 = [1,2,3]

arr.concat(arr2)

(9) [1, 2, 3, 4, 5, 6, 1, 2, 3]

[...arr,...arr2]

(9) [1, 2, 3, 4, 5, 6, 1, 2, 3]

2）slice: [索引1，索引2) 长度是索引2-索引1

arr.slice(2,4)

(2) [3, 4]

3）splice (start, ?deletecount, …items)

应用●

删除：splice(0,2)   //删除前两项

插入：spice(2,0,’red’)   //在第二项后面插入’red’

替换：spice(2,1,’red’)   //从第二项开始删除一个，即删除第二项，并在该位置添加’red’

**数组去重**

方式一： 思想: 创建一个空的数组

var a =[1,2,2,2,2]

function distinct(arr){

//两层for循环

for(let i=0, len=arr.length;i<len;i++){

//从数组的当前位置的后一个开始比较，一旦有重复的，直接删除

for(let j=i+1;j<len;j++){

if(arr[i]===arr[j]){

arr.splice(j,1);

//删除数组以后数组的长度改变了，因此j和len都需要改变

j--;

len--;

}

}

}

return arr

}

distinct(a)

(2) [1, 2]

方式二： 思想：利用filter和indexOf 检测元素在数组中第一次出现的位置和元素现在的位置是否相等

var a =[1,2,2,3,3]

function distinc(a){

let arr = a;

//

**return** a.filter((item,index)=>{

return arr.indexOf(item) === index

})

}

distinc(a)

(3) [1, 2, 3]

## 面向对象

### 3.2.1 作用域 作用域链 闭包

<https://time.geekbang.org/column/article/126339>

**执行JS代码经历两个阶段：**

1. 编译阶段
   1. 生成执行上下文（创建条件：1.执行全局代码 2.调用函数 3.使用eval函数）

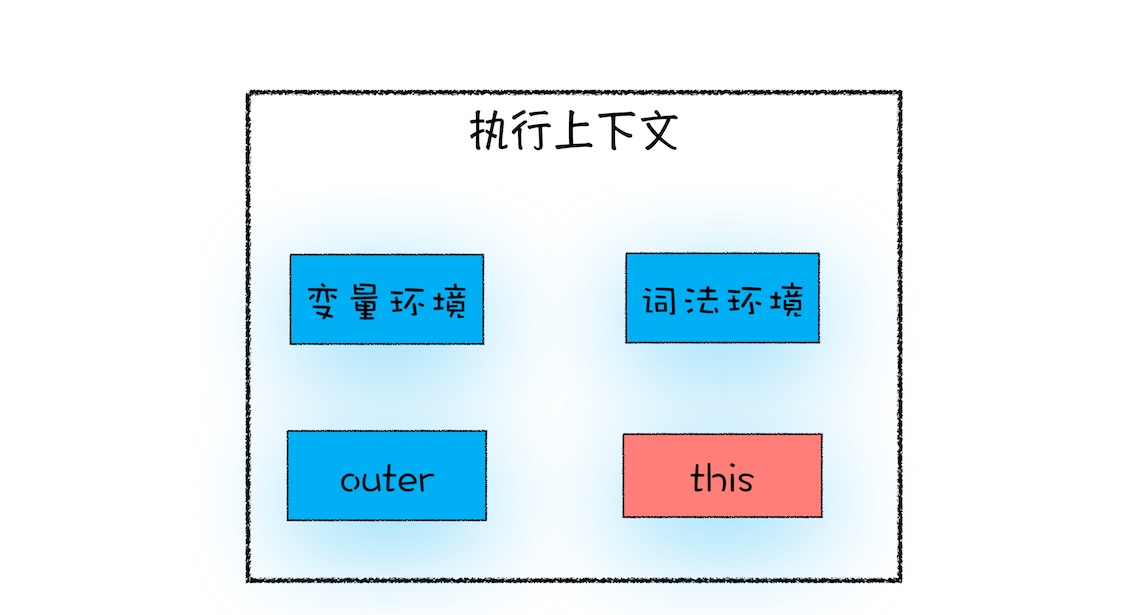
【全局执行上下文只有一个】

* 1. 生成可执行的代码

1. 执行阶段

块级作用域就是通过词法环境的栈结构来实现的，而变量提升是通过变量环境来实现

【拓展】  
var的创建和初始化被提升，赋值不会被提升。  
let的创建被提升，初始化和赋值不会被提升。  
function的创建、初始化和赋值均会被提升



执行上下文栈（调用栈）： 管理执行上下文的栈称为执行上下文栈

作用域： 作用域是指在程序中定义变量的区域，该位置决定了变量的生命周期。通俗地理解，作用域就是变量与函数的可访问范围，即作用域控制着变量和函数的可见性和生命周期。

**作用域链：**

function bar() {

console.log(myName)

}

function foo() {

var myName = "极客邦"

bar()

}

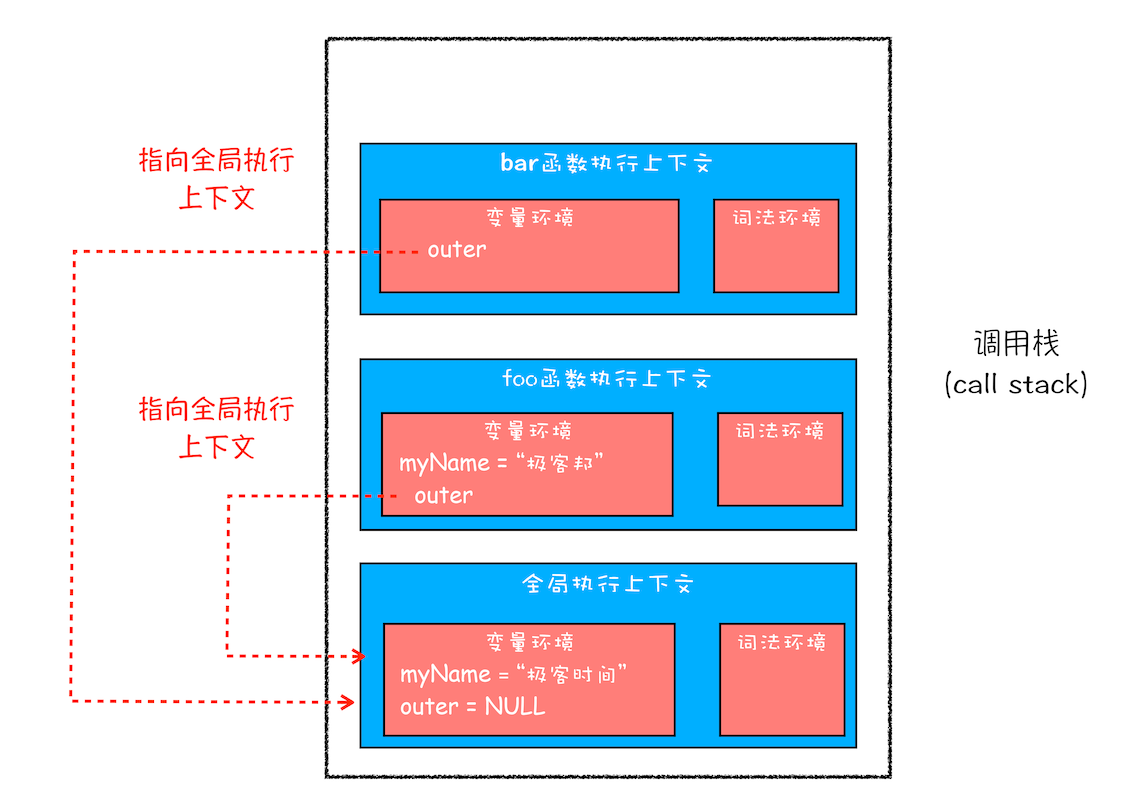
var myName = "极客时间"

foo()

其实在每个执行上下文的变量环境中，都**包含了一个外部引用，用来指向外部的执行上下文**，我们把这个外部引用称为 outer。

foo 函数调用的 bar 函数，那为什么 bar 函数的外部引用是全局执行上下文，而不是 foo 函数的执行上下文？要回答这个问题，你还需要知道什么是**词法作用域**。这是因为在 JavaScript 执行过程中，其作用域链是由词法作用域决定的。

**词法作用域**就是指作用域是由代码中函数声明的位置来决定的，所以词法作用域是**静态的作用域**，通过它就能够预测代码在执行过程中如何查找标识符。



闭包：

**new 运算符**创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。**new** 关键字会进行如下的操作：

1. 创建一个空的简单JavaScript对象（即**{}**）；
2. 链接该对象（即设置该对象的构造函数）到另一个对象 ；
3. 将步骤1新创建的对象作为**this**的上下文 ；
4. 如果该函数没有返回对象，则返回**this**。

This的设计缺陷：

1. 嵌套函数中的 this 不会从外层函数中继承

第一种是把 this 保存为一个 self 变量，再利用变量的作用域机制传递给嵌套函数。

第二种是继续使用 this，但是要把嵌套函数改为箭头函数，因为箭头函数没有自己的执行上下文，所以它会继承调用函数中的 this。

2. 普通函数中的 this 默认指向全局对象 window

This的解决方案

1. bind, apply, call
2. 箭头函数，因为箭头函数没有自己的执行上下文

以嵌套函数不会从调用它的函数中继承 this，这样会造成很多不符合直觉的代码。要解决这个问题，你可以有两种思路闭包

**闭包：**闭包就是能够读取其他函数内部变量的函数。（阮一峰理解）

在 JavaScript 中，根据词法作用域的规则，内部函数总是可以访问其外部函数中声明的变量，当通过调用一个外部函数返回一个内部函数后，即使该外部函数已经执行结束了，但是内部函数引用外部函数的变量依然保存在内存中，我们就把这些变量的集合称为闭包。比如外部函数是 foo，那么这些变量的集合就称为 foo 函数的闭包。（极客时间定义）

<http://www.ruanyifeng.com/blog/2009/08/learning_javascript_closures.html>

<https://www.cnblogs.com/huanghuali/p/9851453.html>

出现背景： 在函数外部访问函数里面的局部变量

　　function f1(){  
　　　　var n=999;  
　　}

alert(n); // error

解决方案：在函数内部再定义一个函数

　function f1(){

　　　　var n=999;

　　　　function f2(){  
　　　　　　alert(n);  
　　　　}

　　　　return f2;

　　}

　　var result=f1();

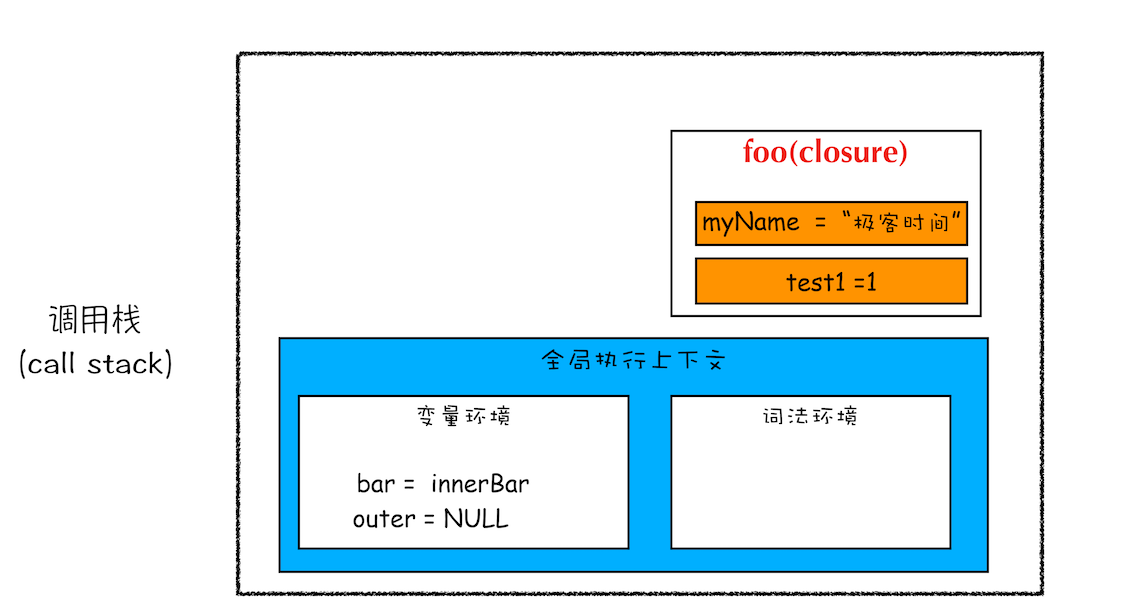
　　result(); // 999

闭包用途：

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

示例：

极客时间： <https://time.geekbang.org/column/article/127495>



闭包的产生过程从上图可以看出，foo 函数执行完成之后，其执行上下文从栈顶弹出了，但是由于返回的 setName 和 getName 方法中使用了 foo 函数内部的变量 myName 和 test1，所以这两个变量依然保存在内存中

【面试题】考点：作用域

var fullname='H';

var obj={

FullName:'Co',

prop:{

fullname:'inner',

getFullName: function(){

return this.fullname

}

}

}

console.log(obj.prop.getFullName()) //inner

var test = obj.prop.getFullName;

test() //H

【面试题】考点：闭包

function Foo(){

var i = 0;

return function(){console.log(i++)}

}

var f1 = Foo();

f2=Foo();

f1(); //0

f1(); //1

f2(); //0

【面试题】考点：传参数

var bb=1;

function aa(bb){

bb=2;

alert(bb);

}

aa(bb) //2

alert(bb); //1

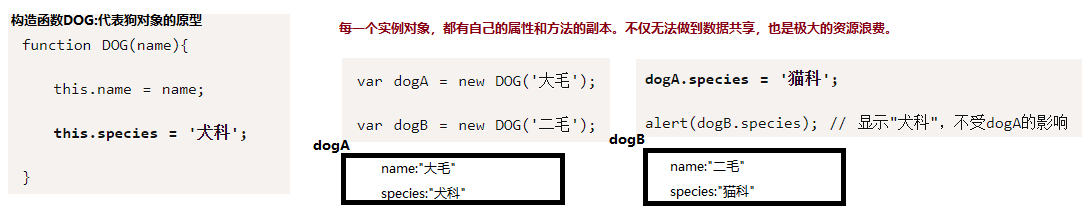
### 3.2.2 原型链与类

**原型模式**

借助原型对象来保存实例的公用方法和属性。

#### JS中如何实现继承？

1. new 构造函数（）



1. prototype属性设计的初衷是为了共享属性和方法



如何在JS中定义类: 因为它的语法中没有class（类）。

那么，如果我们要把"属性"（property）和"方法"（method），封装成一个对象，甚至要从原型对象生成一个实例对象，我们应该怎么做呢？

1. 原始模式：

　　function Cat(name,color) {

//返回一个对象，该对象包含name 和 color属性

　　　return {

　　　　　　name:name,

　　　　　　color:color

　　　　}

　　}

然后生成实例对象，就等于是在调用函数：

　　var cat1 = Cat("大毛","黄色");

　　var cat2 = Cat("二毛","黑色");

两个缺点，

一是如果多生成几个实例，写起来就非常麻烦；

二是实例与原型之间，没有任何办法，可以看出有什么联系。

1. 构造函数模式

　function Cat(name,color){

　　　　this.name = name;

　　　　this.color = color;

　　　　this.type = "猫科动物";

　　　　this.eat = function(){alert("吃老鼠");};

　　}

还是采用同样的方法，生成实例：

　　var cat1 = new Cat("大毛","黄色");

　　var cat2 = new Cat ("二毛","黑色");

　　alert(cat1.type); // 猫科动物

　　cat1.eat(); // 吃老鼠

那就是对于每一个实例对象，type属性和eat()方法都是一模一样的内容，每一次生成一个实例，都必须为重复的内容，多占用一些内存。这样既不环保，也缺乏效率。

改进： 让type属性和eat()方法在内存中只生成一次----🡪 **Prototype模式**

1. **Prototype模式**

function Cat(name,color){

this.name = name

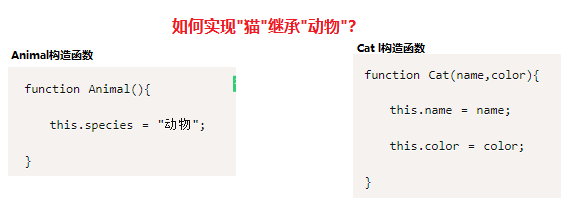
this.color = color

}

Cat.prototype.type=“猫科动物”；

Cat.prototype.eat = function(){alert(“吃老鼠”)}

如何在JS中继承类



1. 构造函数绑定：即在Cat构造函数中,用apply实现继承

　　function Cat(name,color){

　　　　Animal.apply(this, arguments);

　　　　this.name = name;

　　　　this.color = color;

　　}

1. prototype: 如果"猫"的prototype对象，指向一个Animal的实例，那么所有"猫"的实例，就能继承Animal了。

Cat.prototype = new Animal();

　　Cat.prototype.constructor = Cat;

代码的第一行，我们将Cat的prototype对象指向一个Animal的实例。它相当于完全删除了prototype 对象原先的值，然后赋予一个新值

代码第二行, 任何一个prototype对象都有一个constructor属性，指向它的构造函数。如果没有"Cat.prototype = new Animal();"这一行，Cat.prototype.constructor是指向Cat的；加了这一行以后，Cat.prototype.constructor指向Animal。

更重要的是，每一个实例也有一个constructor属性，默认调用prototype对象的constructor属性。

1. **直接继承prototype**

function Animal(){ }

　　Animal.prototype.species = "动物";

然后，将Cat的prototype对象，然后指向Animal的prototype对象，这样就完成了继承。

　　Cat.prototype = Animal.prototype;

　　Cat.prototype.constructor = Cat;

**理解原型对象●**

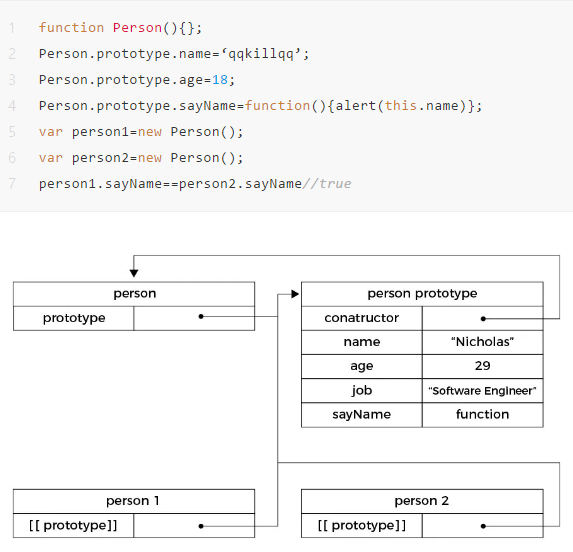
每一个**对象**都有一个**constructor属性**指向了该对象的构造函数

每一个**函数**都有一个**prototype属性**，它是一个对象，称之为**原型对象**

原型对象上只有一个属性constructor指向了原型对象所在的函数

**原型模式的实现 ●**

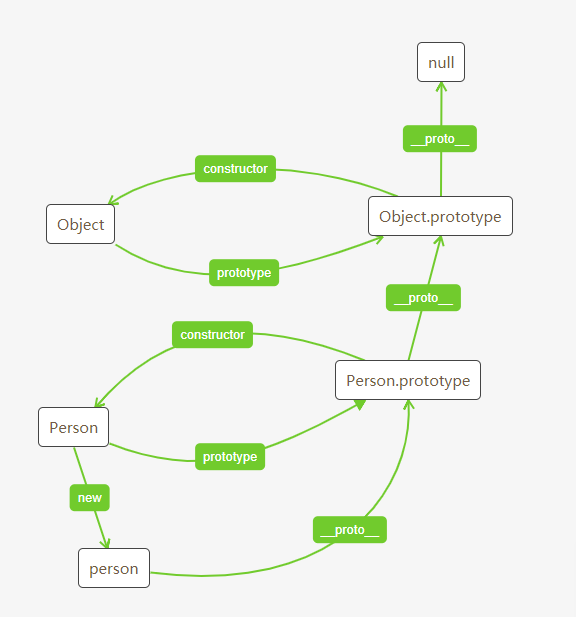
每一个对象的内部都有一个内部指针，指向了该对象的构造函数所对应的原型对象



构造函数person，实例化函数 Person1和Person2 的都指向原型对象person prototype

任何一个prototype对象都有一个constructor属性，指向它的构造函数

原型对象person prototype上的constructor指向了该原型对象的构造函数



**原型链查找●**

当访问对象的属性的时候，如果属性存在，直接返回属性所对应的值，如果属性不存在，则沿着内部指针查找原型对象，如果在原型对象上存在该属性则返回对应的值，如果不存在则沿着**原型对象的内部指针继续查找**。直到查找到Object的原型对象。

类：

子类访问父类： super

【面试题】考点： new操作符干了啥

var p=[];

var A = new Function();

A.prototype = p;

var a = new A();

a.push(1);

console.log(a.length); //1

console.log(p.length) //0

MDN 解释：

<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new>

1. 创建一个空的简单JavaScript对象（即**{}**）；
2. 链接该对象（即设置该对象的构造函数）到另一个对象 ；
3. 将步骤1新创建的对象作为**this**的上下文 ；
4. 如果该函数没有返回对象，则返回**this**。

this的不足之处： 就是无法共享属性和方法

1. var obj={} 创建一个空的对象

2.obj.\_\_proto\_ = A.prototype

3.A.apply(obj) 将obj作为this的上下文

4.return obj 返回对象，如果没有对象则返回this

【扩展this的用法】

<http://www.ruanyifeng.com/blog/2010/04/using_this_keyword_in_javascript.html>

* this是什么？this是JS的关键字，是**函数运行时**，在函数体内部自动生成的一个对象，只能在函数体内部使用
* 函数的不同使用场合
  1. 纯粹的函数调用：属于全局性调用，因此this就代表全局对象

var x = 1;

function test() {

console.log(this.x);

}

test(); // 1

* 1. 作为对象方法的调用：这时this就指这个上级对象。

function test() {

console.log(this.x);

}

var obj = {};

obj.x = 1;

obj.m = test;

obj.m(); // 1

* 1. **作为构造函数调用:** 所谓构造函数，就是通过这个函数，可以生成一个新对象。这时，this就指这个新对象

function test() {

　this.x = 1;

}

var obj = new test();

obj.x // 1

* 1. apply调用

apply()是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时this指的就是这第一个参数。

apply()的参数为空时，默认调用全局对象。因此，这时的运行结果为0，证明this指的是全局对象。

var x = 0;

function test() {

　console.log(this.x);

}

var obj = {};

obj.x = 1;

obj.m = test;

obj.m.apply() // 0

obj.m.apply(obj); //1

【面试题】如何定义一个类，例如一个animal动物类有一个color代表自身的颜色，一个方法walk代表可以走路。

【考点】Javascript中如何定义一个类

# ES6

## 异步编程

* 回调函数

**所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。**

fs.readFile('/etc/passwd', function (err, data) {

if (err) throw err;

console.log(data);

});

* Promise

如果依次读取多个文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团

fs.readFile(fileA, function (err, data) {

fs.readFile(fileB, function (err, data) {

// ...

});

});

Promise实现

var readFile = require('fs-readfile-promise');

readFile(fileA)

.then(function(data){

console.log(data.toString());

})

.then(function(){

return readFile(fileB);

})

.then(function(data){

console.log(data.toString());

})

.catch(function(err) {

console.log(err);

});

## Promise

/promise 异步解决方案,更易于维护

//以前：事件监听或者回调函数（缺点可能嵌套比较多）

//\*\*\*\*\*\*\*\*\*\*回调函数:耗时1秒，1秒完成以后就会调用回调函数cd\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        //  function func(cb){

        //      setTimeout(() => {

        //         cb && cb()

        //      }, 1000);

        //  }

        //  func(()=>{

        //      console.log(1);

        //      func(()=>{

        //          console.log(2)

        //          func(()=>{

        //              console.log(3)

        //          })

        //      })

        //      })

//\*\*\*\*\*\*\*\*\*\*Promise函数:耗时1秒，1秒完成以后就会调用回调函数cd\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1.func2返回一个Promise对象，该对象有.then方法, 1s后执行resolve（）方法 .catch捕捉错误

2.接受两个参数resolve和reject

3. promise的方法 promise.all

\*/

        // function func2(){

        //     return new Promise((resolve,reject)=>{

        //         setTimeout(() => {

        //             resolve()

        //         }, 1000);

        //     })

        // }

        // func2().then(()=>{

        //     console.log(2)

        //     return func2(); //返回promise，又有.then方法

        // })

        // .then(()=>{

        //     console.log(3)

        //     return func2();

        // })

        // .then(()=>{

        //     console.log(4)

        //     return func2();

        // })

        //async await 在遇到yield的时候会停止，知道遇到下一个Next()才会触发

        //async和await就是genertor函数的语法糖,捕捉错误try catch

        // function\* a(v){

        //     yield console.log(v+1)

        //     yield console.log(v+2)

        // }

        // const b = a(5)

        // setInterval(()=>{

        //     b.next()

        // },2000)

        async function a(v){

            await console.log(v+1)

        }

        const b = a(5)

        setInterval(()=>{

            b.next()

        },2000)

## 箭头函数

# Webpack

# React

### 基本概念

1. 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：

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

});

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：

// Correct

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

上面使用了[箭头函数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)，不过使用普通的函数也同样可以：

// Correct

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

1. React事件 处理

React 元素的事件处理和 DOM 元素的很相似，但是有一点语法上的不同:

* 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。

参考链接：<https://react.docschina.org/docs/handling-events.html>

<**div** id="outer">

<p id="inner">Click **me**!</p>

</**div**>

1. 事件冒泡&&事件捕获&&事件代理

**事件冒泡(event bubbling)：**事件冒泡可以形象地比喻为把一颗石头投入水中，泡泡会一直从水底冒出水面. 事件会从最内层的元素开始发生，一直向上传播 p🡪div🡪body🡪html🡪document

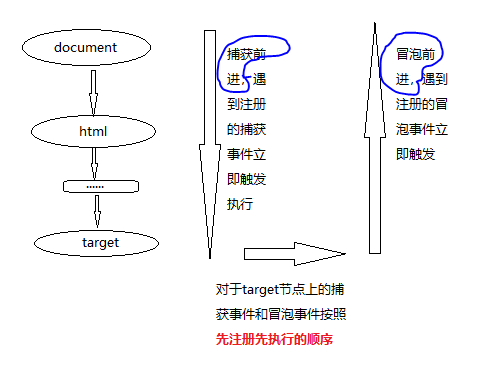
**事件捕获（event capturing）：**与事件冒泡相反，事件会从最外层开始发生，直到最具体的元素。document🡪html🡪body🡪div🡪p

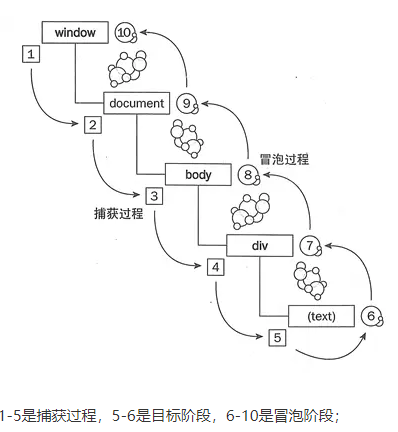
**addEventListener的第三个参数**

element.addEventListener(event, function, useCapture)

第一个参数是需要绑定的事件  
第二个参数是触发事件后要执行的函数  
第三个参数默认值是false，表示在**事件冒泡阶段**调用事件处理函数;如果参数为true，则表示在**事件捕获阶段**调用处理函数

事件捕获和事件冒泡一起存在的情况，事件又是如何触发呢？

****



**事件代理：**

由于事件冒泡机制，点击了 li 后会冒泡到 ul ，此时就会触发绑定在 ul 上的点击事件，再利用 target 找到事件实际发生的元素，就可以达到预期的效果。

使用事件代理的好处不仅在于将多个事件处理函数减为一个，而且对于不同的元素可以有不同的处理方法。假如上述列表元素当中添加了其他的元素节点（如：a、span等），我们不必再一次循环给每一个元素绑定事件，直接修改事件代理的事件处理函数即可。

（1）toLowerCase() 方法用于把字符串转换为小写。语法：stringObject.toLowerCase()

返回值：一个新的字符串，在其中 stringObject 的所有大写字符全部被转换为了小写字符。

（2）nodeName 属性指定节点的节点名称。如果节点是元素节点，则 nodeName 属性返回标签名。如果节点是属性节点，则 nodeName 属性返回属性的名称。对于其他节点类型，nodeName 属性返回不同节点类型的不同名称。

所有主流浏览器均支持 nodeName 属性。

案例：[file:///D:/Study/GitHub/DeepStudyForReact/Shine\_JSDemo/Event/eventAgent.html](file:///D:\Study\GitHub\DeepStudyForReact\Shine_JSDemo\Event\eventAgent.html)

链接：https://juejin.im/post/5cc941436fb9a03236394027  
来源：掘金

1. 合成事件与原生事件

**目的：**

避免DOM事件滥用，

同时屏蔽底层不同浏览器之间的事件系统差异，

实现了一个中间层——SyntheticEvent。

**合成事件是什么**

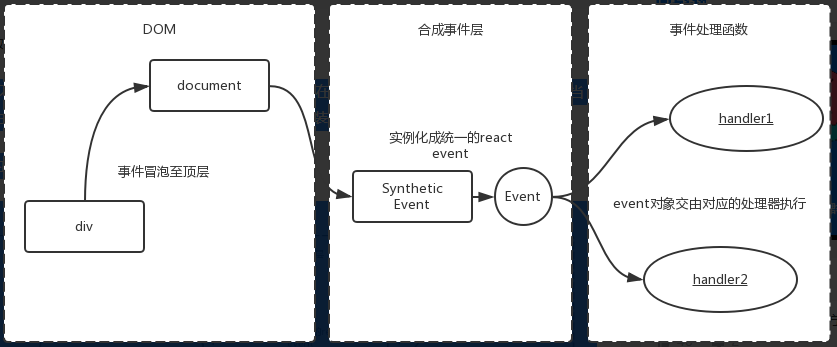
* 1. React事件处理函数接受到的是合成事件的实例.
  2. 合成事件是浏览器的原生事件的跨浏览器包装器
  3. 与原生事件具有相同的接口 包括stopPropagation()和preventDefault()



**原理：**

React并不是将click事件绑在该div的真实DOM上，而是在document处监听所有支持的事件，**当事件发生并冒泡至document处时，React将事件内容封装并交由真正的处理函数运行**。

以上面的代码为例，整个事件生命周期示意如下：



其中，由于event对象是复用的，事件处理函数执行完后，属性会被清空，所以event的属性无法被异步访问，详情请查阅[event-pooling](https://reactjs.org/docs/events.html#event-pooling)。

作者：Nekron  
链接：https://juejin.im/post/59db6e7af265da431f4a02ef

**如何在React中使用原生事件**

由于原生事件需要绑定在真实DOM上，所以一般是在componentDidMount阶段/ref的函数执行阶段进行绑定操作，在componentWillUnmount阶段进行解绑操作以避免内存泄漏。

合成事件与原生事件混合使用：

<https://juejin.im/post/59db6e7af265da431f4a02ef>‘’

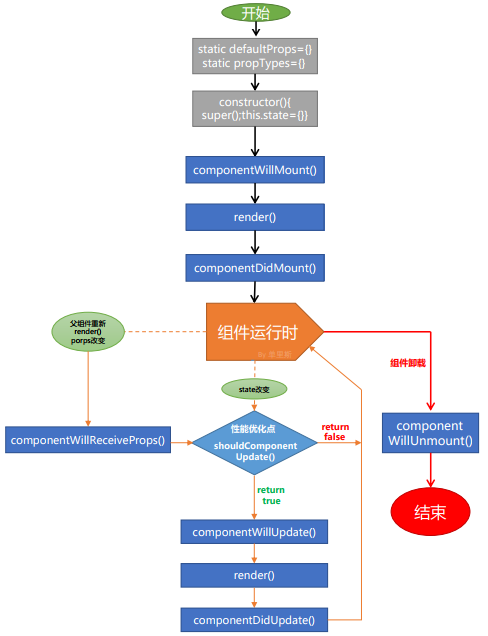
响应顺序：

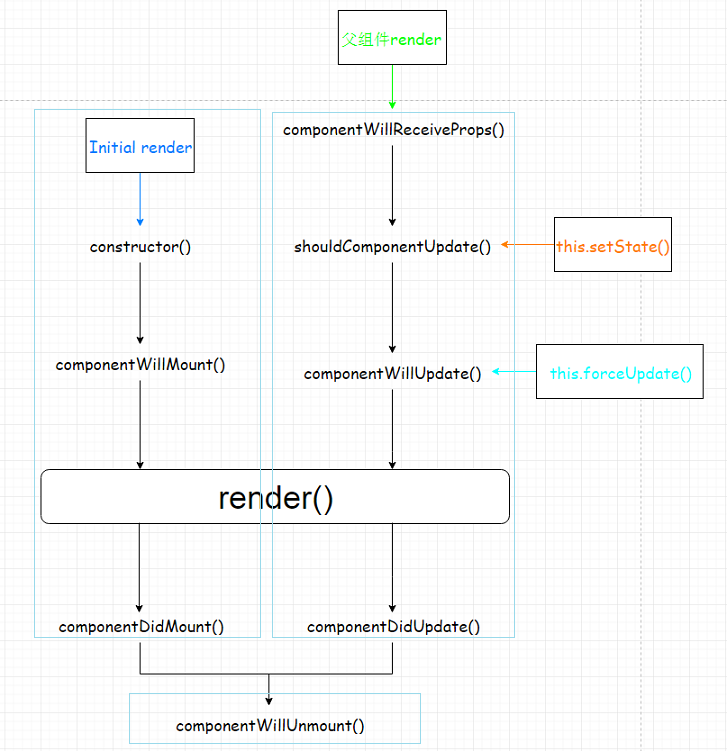
* 合成事件的监听器是统一注册在document上的，且仅有冒泡阶段。所以原生事件的监听器响应总是比合成事件的监听器早
* 阻止原生事件的冒泡后，会阻止合成事件的监听器执行
* 合成事件的nativeEvent在本文场景中，没毛用

1. Key的作用

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。

### 6.2 组件生命周期





三个阶段： 挂载组件---🡪更新组件--🡪卸载组件

1. 挂载（插入真实的DOM）

Constructor()

componentWillMount()

render()

componentDidMount(获取数据一般在这里发送ajax请求)

* 1. 组件已经渲染完毕

1. 更新

Componentwillreceiveprops)\_

shouldcomponentUpdate(性能优化点)

componentwillupdate()

render

componentDidUpdate()

1. 卸载

componentWillUnmount()

### 6.3 React-Router

路由器router

两个方法： get post

参数（path，func（req,res））

注册路由

route 路由其实就是一个映射关系， key(path) value(回调函数)

路由组件一般放在views或者pages文件夹中

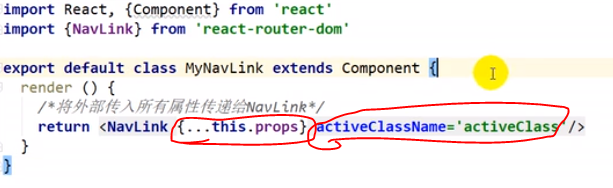
* 在index.js中将App用<BrowserRouter>或者<HashRouter>包裹起来，表示用它来管理路由
* 在App.js中，之所以用Switch包裹是因为每次只能显示一个



初始加载的时候默认是about,使用重定向



如何包装一个现有的组件?



使用

如何编写路由效果？

1. 编写路由组件
2. 在父路由组件中指定
   1. 路由链接<NavLink>
   2. <Route>



:id既是占位符也是参数



上述有个小错误：



路由链接和非路由链接的区分：看点击以后有没有发送请求，说白了有没有页面跳转

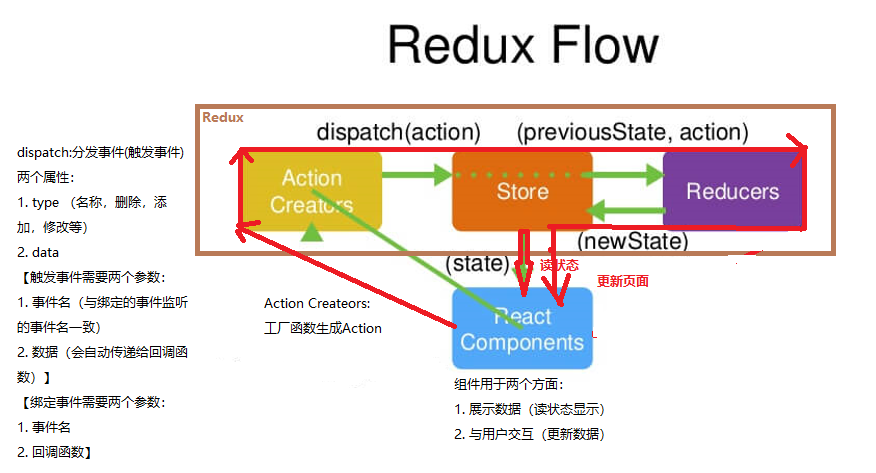
路由链接：不发送请求

非路由链接：发送请求

### 6.4 Redux

Redux是干嘛的？做状态管理的JS库

作用：集中式管理react应用中多个组件共享的状态



Demo: D:\Study\GitHub\DeepStudyForReact\ShineDemo\my-app

设计思想:

（1）Web 应用是一个状态机，视图与状态是一一对应的。

（2）所有的状态，保存在一个对象里面。



工作流程:

1. 用户发出Action store.dispatch(action)
2. store自动调用Reducer,并传入两个参数当前state和Action,然后返回新的state

let nextState = todoApp(previousState, action)

1. 一旦state有变化，store会调用监听函数 store.subscribe(listener)
2. listener通过store.getState()得到当前状态

function listerner(){

let newState = store.getState();

Component.setState(newStae)

}

如何生成store ?

Redux 提供**createStore**这个函数，用来生成 Store。

import { createStore } from 'redux';

const store = createStore(fn);

上面代码中，createStore函数接受另一个函数作为参数，返回新生成的 Store 对象。

如何操作store？

* store.getState()： 用来获取 store 的状态
* store.dispatch()： 用来修改store的状态；
* store.subscribe()：用来监听 store 的状态

Action是什么？

* Action 就是 View 发出的通知，表示 State 应该要发生变化了。
* Action 是一个对象。其中的type属性是必须的

const action = {

type:”ADD\_TODO”

}

* Action 描述当前发生的事情。改变 State 的唯一办法，就是使用 Action。它会运送数据到 Store。

Reducer是什么？

在store接受到action以后计算新的 State 的计算过程就叫做 Reducer，它是一个函数，入参是Action和State

const reducer = function(state, action){

//……

return new state

}

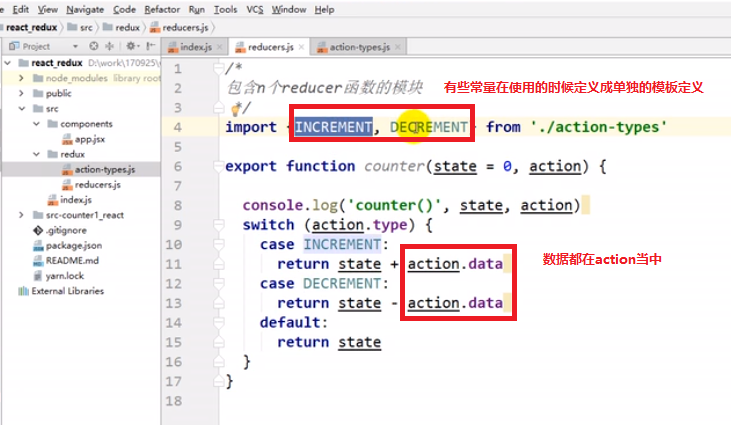
如何创建在store.dispatch的时候触发reducer?

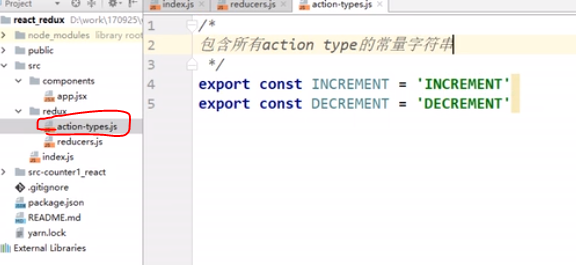
import {createStore} from ‘redux’

const store = createStore(reducer)

步骤【推荐：专门创建一个文件夹redux 】：

1. 在index.js中生成一个store对象





React-Redux

connect函数接收一个组件

**steps:**

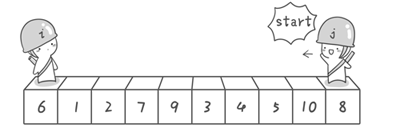
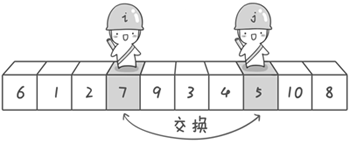
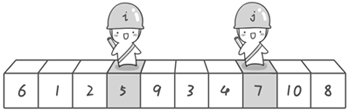
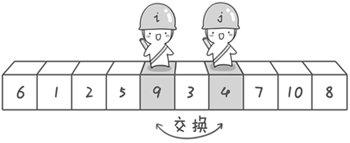
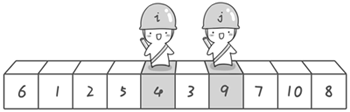
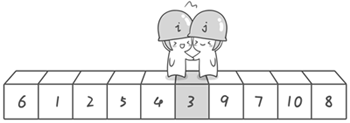
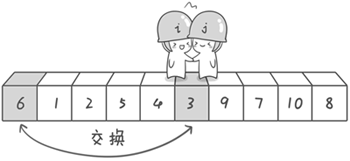
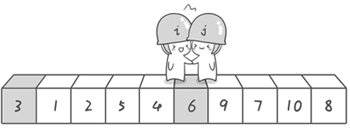
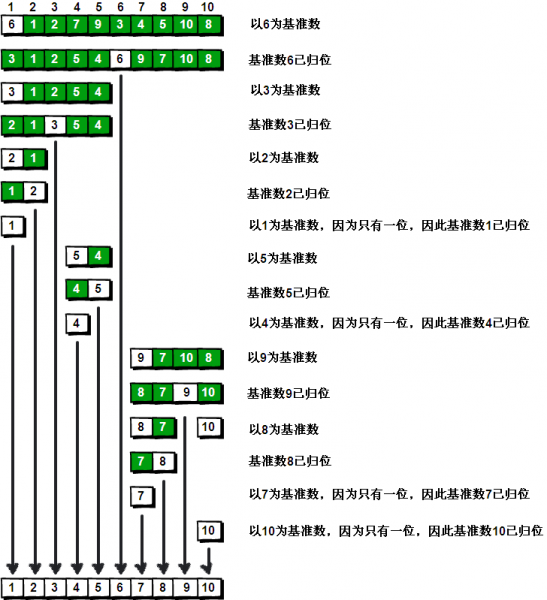
1. 创建文件夹：
   * redux,
   * components(UI组件) ,
   * containers（容器组件）
2. 在redux文件夹下面创建几个文件：
   * action-types:存放包含所有action的type名称常量（常量字符串）
   * actions: 包含了所有的action creator(action的工厂函数)
   * reducers:包含所有的reducer函数（根据老的state和action返回一个新的state）
   * store: Redu最核心的管理对象

# 算法

1. 快速排序

<https://blog.csdn.net/adusts/article/details/80882649>

平均时间复杂度 nlogn 最坏的情况是O(N²)

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9.   
   #include <stdio.h>
10. int a[101],n;//定义全局变量，这两个变量需要在子函数中使用
11. void quicksort(int left, int right) {
12. int i, j, t, temp;
13. if(left > right)
14. return;
15. temp = a[left]; //temp中存的就是基准数
16. i = left;
17. j = right;
18. while(i != j) { //顺序很重要，要先从右边开始找
19. while(a[j] >= temp && i < j)
20. j--;
21. while(a[i] <= temp && i < j)//再找右边的
22. i++;
23. if(i < j)//交换两个数在数组中的位置
24. {
25. t = a[i];
26. a[i] = a[j];
27. a[j] = t;
28. }
29. }
30. //最终将基准数归位
31. a[left] = a[i];
32. a[i] = temp;
33. quicksort(left, i-1);//继续处理左边的，这里是一个递归的过程
34. quicksort(i+1, right);//继续处理右边的 ，这里是一个递归的过程
35. }
36. int main() {
37. int i;
38. //读入数据
39. scanf("%d", &n);
40. for(i = 1; i <= n; i++)
41. scanf("%d", &a[i]);
42. quicksort(1, n); //快速排序调用
43. //输出排序后的结果
44. for(i = 1; i < n; i++)
45. printf("%d ", a[i]);

不可变数据：

不可以直接修改，需要复制一份，产生新的对象的方式，包含修改部分

性能优化： store变化通知组件需要更新，新的state和旧的state是两个完全不同的对象

如何操作不可变数据

{…},Object.assgin

const newState = {…state,todos:[…state.todos, ‘Learn Redux’]}