金渡教育 内部资料,请勿外传,违者必究！！！ QQ:2429462491

CSS\JS代码规范

参考文件: [css js规范]

JavaScript简介

1、计算机语言

计算机不能直接理解高级语言，只能直接理解机器语言，所以必须要把高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。 翻译的方式有两种，一个是编译，一个是解释。两种方式只是翻译的时间不同。

2、编译性语言

编译型语言写的程序执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如exe文件，以后要运行的话就不用重新翻译了，直接使用编译的结果就行了（exe文件），因为翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高。

3、解释性语言

解释则不同，解释性语言的程序不需要编译，省了道工序，解释性语言在运行程序的时候才翻译，比如解释性java语言，专门有一个解释器能够直接执行java程序，每个语句都是执行的时候才翻译。这样解释性语言每执行一次就要翻译一次，效率比较低

4、编译器与解释器的区别

编译型与解释型，两者各有利弊。前者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像C/C++等都是编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释性语言，如：JAVA javascript python等

5、JavaScript语言

JavaScript（简称"JS"） 是一种具有函数优先的轻量级，解释型或即时编译型的高级编程语言。虽然它是作为开发Web页面的脚本语言而出名的，但是它也被用到了很多非浏览器环境中，JavaScript 基于原型编程、多范式的动态脚本语言，并且支持面向对象、命令式和声明式（如函数式编程）风格。

6、什么是堆？什么是栈?

计算器语言有一个处理的过程，写的代码会进行解释或编译执行，这个过程是在内存中，内存的使用和分配，涉及到堆和栈
任何语言都有堆和栈 ，堆和栈都存放在内存中

栈

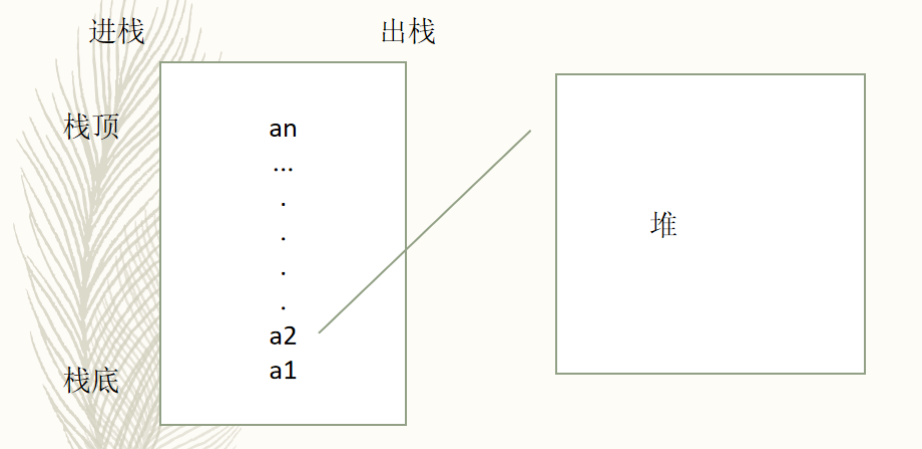
栈：javascript的基本类型就5种:Undefined、Null、Boolean、Number和String，它们都是直接按值存储在栈中，每种类型的数据占用的内存空间的大小是确定的

栈由系统自动分配，例如，声明在函数中一个局部变量var a; 系统自动在栈中为a开辟空间
只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出

堆

堆：javascript中其他类型的数据被称为引用类型的数据：如对象(Object)、数组(Array)、函数(Function)…，
它们是通过拷贝和new出来的，这样的数据存储于堆中
其实，说存储于堆中，也不太准确，因为，引用类型的数据的地址指针是存储于栈中的，当我们想要访问引用类型的值的时候，需要先从栈中获得对象的地址指针，然后，在通过地址指针找到堆中的所需要的数据。

图形说明



JS数据类型

类型定义及检测

```
var str = 'abc';
var num = 123;
var bool = true;
var und = undefined;
var n = null;
var arr=['x','y','z'];
var obj = {};
var fun = function() {};
console.log(typeof str);   //string
console.log(typeof num);   //number
console.log(typeof bool);  //boolean
console.log(typeof und);   //undefined
console.log(typeof n);     //object
console.log(typeof arr);   //object
console.log(typeof obj);   //object
console.log(typeof fun);   //function
```

基本类型与引用类型的区别

```
var a1 = 10;
var b1 = a1;
a1 = 9;
console.log(b1);  //10

var obj = {id:100};
var obj2 = obj;
obj.id = 99;
console.log(obj2.id); //99
```

数组的定义方式

1、使用Array构造函数

构造函数，是一种特殊的方法。主要用来在创建对象时初始化对象， 即为对象成员变量赋初始值，总与new运算符一起使用

```
var arr = new Array();
arr[0]='a';
arr[1]='b';
arr[2]='c';
```

简洁的写法：

```
var arr2 = new Array('red','blue','yellow');
```

2、字面量表示法

字面量是变量的字符串表示形式。它不是一种值，而是一种变量记法。

```
var arr3 = ['alice','angela','jack'];
```

对象的定义方式

1、使用Object构造函数

```
var person = new Object();
person.name = 'alice';
person['age'] = 20;
```

2、字面量表示法

2.1简单字面量

```
var person2 = {};
person2.name='jack';
person2.action = function(){console.log(this.name)};
person2.action();
```

2.2嵌套字面量

```
var person3 = {
    name:'angela',
    age:18,
    action:function(){
        console.log(this.age)
    }
};
person3.action();
```

函数的定义方式

1、函数声明

```
 function sum(x,y){
    return x+y
 };
```

2、函数表达式

```
var sum2 = function(x,y){
    return x+y
}
```

对象的属性和方法

属性的设置和获取

1、 设置

```
var obj = {};
obj.name='amy';
obj['age'] = 20;
```

2、 获取

```
obj.name;
obj['age'];
```

属性的删除

```
var o2= {
    name:'abc',
    age:18
};
delete o2.name;
console.log(o2);
```

检测属性

检测属性 该方法可以判断对象的自有属性是否存在

in 运算符 检测属性是否存在于某个对象中，自有属性和继承属性都返回true

```
var obj={
    name:'sonia',
    age:22
};
console.log('name' in obj);//自有属性
```

hasOwnProperty() 方法用于检测属性是否是自有属性，是则返回true,否则返回false

```
var obj = {
    name:'lily'
};
console.log(obj.hasOwnProperty("name"));
```

区别

```
function Init() {}
Init.prototype.name = 'xyz';  //原型对象

var init = new Init();
init.age = 18;

console.log("name" in init); // true
console.log("age" in init); // true
console.log(init.hasOwnProperty("name"));  //false
console.log(init.hasOwnProperty("age"));  //true
```

遍历属性

for ... in

```
var obj = {name:'a',age:20};
for(var key in obj){
    console.log(key)
    console.log(obj[key])
};
```

序列化

JSON.stringify() 方法用于将 JavaScript 值转换为 JSON 字符串

```
var obj = {name:'a',age:20};
console.log(typeof JSON.stringify(obj))
```

JSON.parse() 方法用于将一个 JSON 字符串转换为对象

```
var obj = {name:'a',age:20};
var str = JSON.stringify(obj);
console.log(typeof JSON.parse(str))
```

深拷贝和浅拷贝

在面试时经常会碰到面试官问:什么是深拷贝和浅拷贝，请举例说明？如何区分深拷贝与浅拷贝，简单来说，假设B复制了A，当修改A时，看B是否会发生变化，如果B也跟着变了，说明这是浅拷贝，如果B没变，那就是深拷贝；我们先看两个简单的案例

案例一

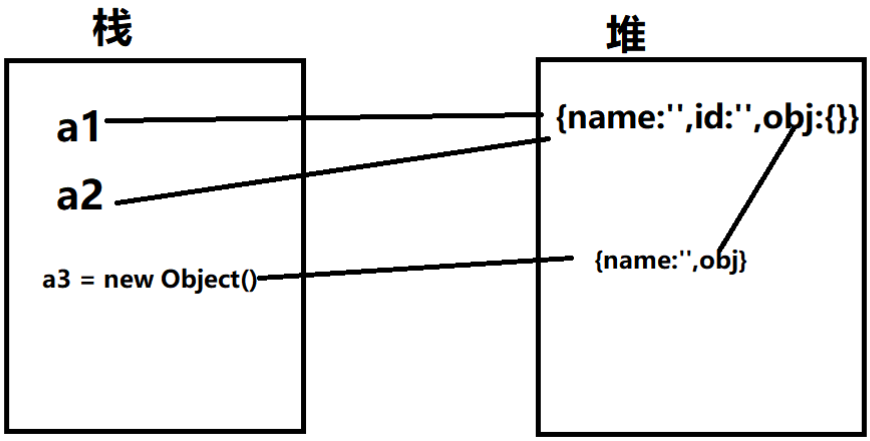
```
var a1 = 1, a2= a1;
console.log(a1) //1
console.log(a2) //1
a2 = 2; //修改 a2
console.log(a1) //1
console.log(a2) //2
```

案例二

```
var o1 = {x: 1, y: 2}, o2 = o1;
console.log(o1) //{x: 1, y: 2}
console.log(o2) //{x: 1, y: 2}
o2.x = 2; //修改o2.x
console.log(o1) //{x: 2, y: 2}
console.log(o2) //{x: 2, y: 2}
```

按照常规思维，o1应该和a1一样，不会因为另外一个值的改变而改变，而这里的o1 却随着o2的改变而改变了。同样是变量，为什么表现不一样呢?

参考文件 深入剖析javascript中的深拷贝和浅拷贝



预解析

定义

JavaScript"预解析"，可以理解为把变量或函数预先解析到它们被使用的环境中。

解析过程

第一步，会预先解析关键字var、function等

第二步，提前赋值：

1. var a = 10 提前解析 var a；(此时a的值为undefined)

2. 函数，在正式运行代码前，赋值为整个函数块

```
console.log(fn)   //function变量提升   function fn(){console.log("123")}
function fn(){
    console.log("123")
}
```

第三步，预解析结束后，浏览器再逐行解读代码;

```
//看看下面的代码输出结果
console.log(a)
var a = 1;
//解析过程
var a;
console.log(a); //var变量提升   undefined
a = 1;
```

解析原则

预解析过程中，当变量和函数同名时：只留下函数的值，不管谁前谁后，所以函数优先级更高;

```
console.log(fn)   //function变量提升   function fn(){console.log("123")}
var fn = 234;
function fn(){
    console.log("123")
}
```

经典预解析面试题

```
console.log(a)
var a=1;
function a(){console.log(2)}
console.log(a)
var a=3;
console.log(a)
function a(){console.log(4)}
console.log(a)
```

```
function fun ( n ) {
  console.log( n );
  var n = 456;
  console.log( n );
}
var n = 123;
fun( n );
答：123  456
 //解析过程
  //预解析
  var n;
  function fun(){};
  n = 123 //全局变量
  fun(n) ////当执行fun(n)，会执行函数体里的内容，此时fun函数会形成一个新的私有作用域
  //fun()内部解析过程
  //如果有形参，先给形参赋值
```

```
var n = 123;
//进行私有作用域中的预解析;
var n;
//私有作用域中的代码从上到下执行
console.log( n );  //123
n = 456;
console.log( n );  //456
```

作用域

定义

作用域:它是指对某一变量和方法具有访问权限的代码空间, 在JS中, 作用域是在函数中维护的。表示变量或函数起作用的区域,指代了它们在什么样的上下文中执行,亦即上下文执行环境。

ES5的作用域只有两种:全局作用域和局部作用域

全局作用域

```
var a=1;  //全局作用域
function fn1(){
    console.log(a)
};
fn1()
```

局部作用域

```
function fn1(){
    var a=1;    //局部作用域
};
fn1();
console.log(a);
```

全局变量和局部变量同名的坑

(1)在全局变量和局部变量不同名时，其作用域是整个程序。

(2)在全局变量和局部变量同名时，全局变量的作用域不包含同名局部变量的作用域。

```
var a=1;
function fn1(){
    console.log(a)
    var a = 2;
};
fn1();
console.log(a);
```

经典作用域面试题

```
var a = 10;
function f1(){
  var b = 2 * a;
  var a = 20;
  var c = a+1;
  console.log(b);
  console.log(c);
}
f1()
```

```
var a=10;
function test(){
    console.log(a);
    a=100;
    console.log(this.a);
    var a;
    console.log(a);
}
test();
```

函数式编程与面向对象编程的异同

函数式编程写法

```
window.onload = function(){
    var testA = document.querySelector(".testA");
    testA.onmousemove = function(){
        //移入处理
    };
    testA.onmouseout= function(){
        //移出处理
    };
}
```

面向对象编程写法

```
var web = {
     bind:function(){
        this.testA = document.querySelector(".testA");
        this.testA.onmousemove = function(){
            //移入处理
        };
        this.testA.onmouseout= function(){
            //移出处理
        };
    }
}
window.onload = function(){
    web.bind();
}
```

原型

定义

用原型实例指向创建对象的类，使用于创建新的对象的类的共享原型的属性与方法

原型是一个对象，其它对象可以通过它实现属性继承

JS对象分两种

普通对象object和函数对象function

```
prototype是函数才有的属性
__proto__是每个对象都有的属性
```

普通对象和函数对象区别

凡是通过new Function创建的对象都是函数对象，其他都是普通对象（通常通过Object创建），可以通过typeof来判断。

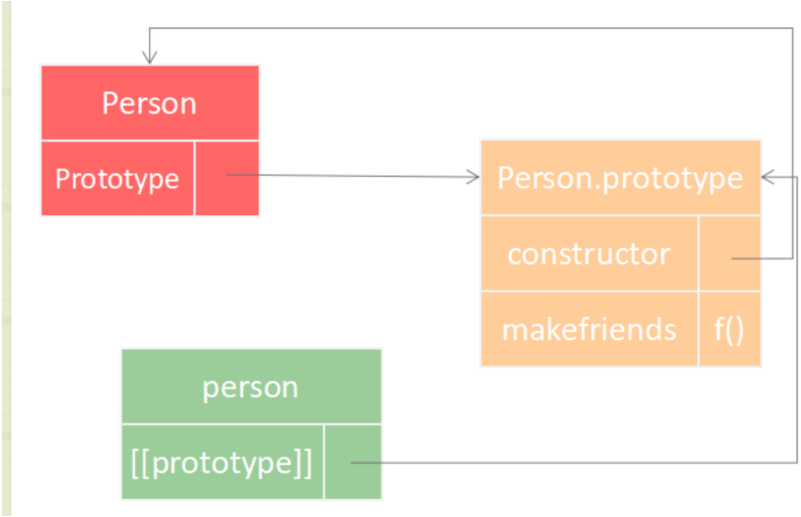
```
function f1(){};
typeof f1 //"function"

var o1 = new f1();  //函数实例
typeof o1 //"object"

var o2 = {};
typeof o2 //"object"
```

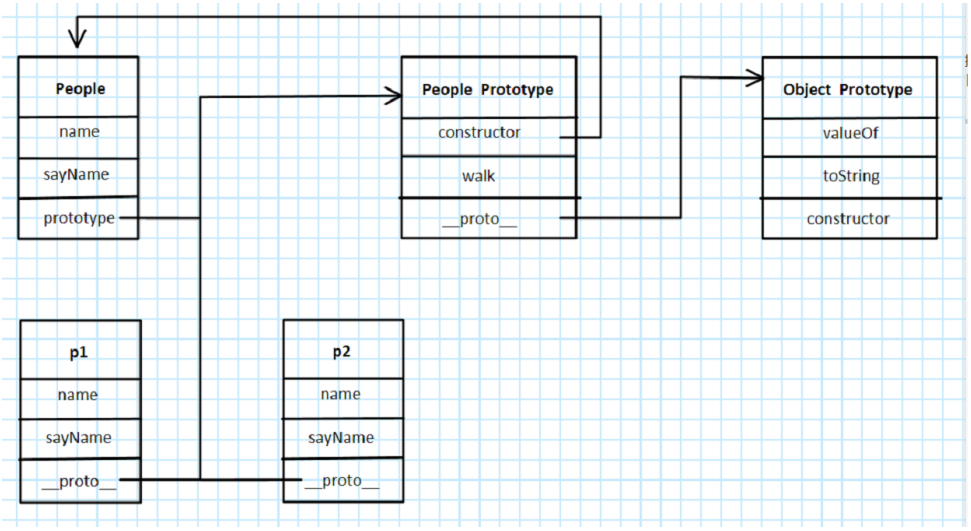
```
1、每一个函数对象都有一个prototype属性，但是普通对象是没有的；
   prototype下面又有个constructor，指向这个函数。
2、每个对象都有一个名为__proto__的内部属性，指向它所对应的构造函数的原型对象，原型链基于
__proto__;
```

原型的写法

```
function Person(){};  //定义一个函数对象
Person.prototype.name="abc";  //原型对象中添加属性
Person.prototype.age = 18;
var p1 = new Person();  //实例化
var p2 = new Person();
```

原型分解图





为什么使用function中的prototype? 为什么要继承

需求：生成多个实例

```
var cat1 = {};//创建一个空对象
cat1.name="大明";
cat1.color ="黄色";
var cat2 = {};//创建一个空对象
cat2.name="小明";
cat2.color ="白色";
//缺点，如果有几十个实例，写起来麻烦，且实例与原型没有关联
```

封装一个函数

```
function cat(name,color){
    return {
        name:name,
        color:color
    }
};
var cat1 = cat("大明","黄色");
var cat2 = cat("小明","白色");
```

构造函数

```
function Cat(name,color){   //构造函数
    this.name = name;
    this.color = color;
};
var cat1 = new Cat("大明","黄色");
var cat2 = new Cat("小明","白色");
```

如果当前cat函数中添加了eat()方法和type属性

```
function Cat(name,color){
    this.name = name;
    this.color = color;
    this.type='动物';
    this.eat = function(){console.log("吃老鼠")};
};
存在一个浪费内存的问题，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
既不环保，也缺乏效率
```

解决方法：

```
function Cat(name,color){   //构造函数
    this.name = name;
    this.color = color;
    //this.type='动物';
    //this.eat = function(){console.log("吃老鼠")};
};
Cat.prototype.type='动物';
Cat.prototype.eat = function(){console.log("吃老鼠")};
```

验证

```
console.log('name' in p1);  //in  不管自身的还是原型 都返回true
console.log('type' in p1);
console.log(p1.hasOwnProperty('name'));  //hasOwnProperty() 自身返回true  原型 返回
false
console.log(p1.hasOwnProperty('type'));
```

继承

继承的几种常见方式

原型继承

```
function Animal(){
    this.type = "动物"
};

function Cat(name,color){
    this.name = name;
    this.color = color;
};
Cat.prototype = new Animal();
var c1 = new Cat('x','白色');
var c2 = new Cat('t','花色');
c1.type
```

优点：同一个原型对象
缺点：不能修改原型对象，会影响所有实例

```
function Animal(){   //动物对象
    this.type = '动物'
};
function Cat(name,color){   //猫对象
    this.name = name;
    this.color = color;
    this.type='我是猫';
};
Cat.prototype = new Animal();  //猫的原型对象指向了动物函数
var cat1 = new Cat("大明","黄色");
var cat2 = new Cat("大明","黄色");
console.log(cat1.type); //获取的是当前构造器中的属性
console.log(cat2.type); //获取的是当前构造器中的属性
//想获取Animal成员值
console.log(cat1.__proto__.type);
console.log(cat2.__proto__.type);
//当我们访问一个原型对象的属性时，__proto__是一级级来获取，当继承关系很复杂，未知继承时
```

构造函数的继承

```
function Animal(){
    this.type = "动物"
};
function Cat(name,color){
    Animal.apply(this);   //将Animal对象的成员放到Cat对象上
    this.name = name;
    this.color = color;
};
var cat1 = new Cat("大明","黄色");
var cat2 = new Cat("小明","白色");
cat1.type = '我是黄猫';
cat2.__proto__.type = '我是动物';
console.log(cat1.type); //'我是黄猫'  cat1被修改
console.log(cat2.type); //"动物"
```

优点：不存在修改原型对象影响所有实例，各自拥有独立属性
缺点：父类的成员会被创建多次，存在冗余且不是同一个原型对象
通过apply/call只能拷贝成员，原型对象不会拷贝

组合继承

```
 function Animal(){
    this.type = '动物'
 };
Animal.prototype.eat = function(){console.log('吃')};
function Cat(name,color){
    this.name = name;
    this.color = color;
    Animal.call(this);
};
Cat.prototype = new Animal();
var cat1 = new Cat("大明","黄色");
var cat2 = new Cat("小明","白色");
cat1.type = '我是黄猫';  //修改当前构造器中的属性
```

```
cat2.__proto__.type = '我是动物';//修改了原型对象的值，但并不影响cat1,cat2的值
console.log(cat1.type); //'我是黄猫'  //原型对象的值变化，并不影响构造函数值
console.log(cat2.type); //'动物'
console.log(cat2.__proto__.type);  //'我是动物
cat1.eat(); //还可以调用原型对象中eat()方法
```

原型链

```
function F1(){
    this.name1 = 'f1'
};
F1.prototype.name = 'object';
function F2(){
    this.name2= 'f2'
};
function F3(){
    this.name3 = 'f3'
};
F2.prototype = new F1();  //f2的原型是f1
F3.prototype = new F2();  //f3的原型是f2
var f = new F3();   //实例化的处理
f.name1;
f.__proto__.__proto__.__proto__.name= '12414';
//修改
//f.__proto__.__proto__.__proto__.name= '12414';
//删除
delete f.__proto__.__proto__.__proto__.name;
```

公共组件的定义及封装

直接定义

```
<button class="btn btn-red">红色按钮</button>
<button class="btn btn-green">绿色按钮</button>
<button class="btn btn-blue">蓝色按钮</button>
```

动态定义

```
common.chooser("#red").innerHTML = '<button class="btn btn-red">红色按钮
</button>';
common.chooser("#green").innerHTML = '<button class="btn btn-green">绿色按钮
</button>'
```

面向对象的写法

```
var common = {
    chooser:function(name){
        return document.querySelector(name);
    },
    toBtn :function(title,style){
        return '<button class="btn '+'btn-'+style+'">'+title+'</button>'
    }
}
```

原型继承的方式

```
function Btn(title,style){
    this.title = title;
    this.style = style;
};
Btn.prototype.toHtml = function(){
    return '<button class="btn '+'btn-'+this.style+'">'+this.title+'</button>'
};
```

this指向

this是什么

this是Javascript语言的一个关键字。 它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用，随着函数使用场合的不同，this的值会发生变化，指向是不确定的，也就是说是可以动态改变的；但是有一个总的原则，那就是this指的是，调用函数的那个对象。

this有啥用？平时我们在哪里用到过呢

```
<ul>
    <li>1111</li>
    <li>2222</li>
    <li>3333</li>
    <li>4444</li>
  </ul>
    <script type="text/javascript">
      $("ul li").click(function() {
        $(this).css('color','#f00').siblings().css('color','#333');
      })
    </script>
```

这个是不是好熟悉，这是我们在用到Jquery时最多的处理方式，这里的this就是当前单击的li;这是我们接触到的最简单的this,用途相信很多同学都是理解的，但在实际的应用中this并不只是这么简单，下面跟大家说下其它的引用

1、在简单函数中的使用

```
function test(){
        console.log(this)  //window
};
test()
```

在这种情况下,因为代码不是运行在严格模式下， this 又必须是一个对象，所以他的值默认为全局对象。
因为严格模式下，this的值为undefined

```
function test(){
        "use strict";       //严格模式
        console.log(this)  //undefined
   };
test()
```

2、在对象的方法中使用

```
var obj = {}
    obj.a = 3;
     obj.fun = function(){
        return this.a;   //this表示当前o对象   当前的this.a 等价于 obj.a
```

```
    };
    console.log(obj.fun())
```

还有一些特殊的情况：

1）对象中调用外部函数

```
    var a = 1;
    function test() {
        return this.a;
    };
    var o = {}
    o.a = 3;
    o.b = test;
    console.log(o.b())  //结果为3，因为当前test()中的this表示的是o对象
```

2）字面量方式中的this

```
    var o = {
        name:'sonia',
        bind:function() {
          return this.name;   //当前this表示为o对象
        }
    };
    o.bind()
```

3、在构造函数中使用

所谓构造函数，就是通过这个函数生成一个新对象（object）。当一个函数作为构造器使用时(通过 new 关键字)，它的 this 值绑定到新创建的那个对象。如果没使用 new 关键字，那么他就只是一个普通的函数，this 将指向 window 对象。

```
function Fun(name,age) {
        this.name = name;
        this.age = age;
};
var fun = new Fun('lili',22);
console.log(fun.name);
```

在上面的示例中，有一个名为 Fun() 的构造函数。通过使用 new 操作符创建了一个全新的对象，名为 fun。同时还通传给构造函数参数，作为新对象的name、age属性。通过最后一行代码中可以看到这个字符串成功地打印出来了，因为 this 指向的是新创建的对象，而不是构造函数本身。

4、 如何改变this的指向

apply() 方法接收两个参数：第一个是要设置为 this 的那个对象，第二个参数是可选的，如果要传入参数，则封装为数组作为 apply() 的第二个参数即可。

call() 方法 和 apply() 基本上是一样的，除了后面的参数不是数组， 而是分散开一个一个地附加在后面。

```
var num = 10;
function test(){
    return this.num;
};
var obj ={
    num : 5,
    fun:test
}
console.log(obj.fun.call(this))       //返回的值是10，当前的this表示全局对象

var num = 10;
function test(){
    return this.num;
};
```

```
var obj ={
    num : 5,
    fun:test
}
console.log(obj.fun.call(obj))  //返回值为5，当前的this为obj对象
```

常见面试题

```
var number = 1;
var obj = {
    number:2,
    showNumber:function(){
        this.number = 3;
        (function(){
            console.log(this.number);
        })();
        console.log(this.number);
    }
};
obj.showNumber();

答案是  1    3
由于showNumber方法的拥有者是obj，所以this.number=3; this 指向的就是 obj 的属性 number。
   同理，第二个 console.log 打印的也是属性 number。
   为什么第二点说一般情况下this都是指向函数的拥有者，因为有特殊情况。函数自执行就是特殊情况，在函数自执行里，this 指向的是：window。所以第一个 console.log 打印的是 window 的属性 number。
   所以要加一点：  在函数自执行里，this 指向的是 window 对象。
```

```
测试题：
var length = 100;
function f1() {
  console.log( this.length )
}
var obj = {
  x: 10,
  f2: function( f1 ){
    f1();
    arguments[0]();
  }
}
obj.f2(f1,1);
```

事件冒泡及事件委托

事件冒泡和捕获

当事件发生后，这个事件就要开始传播(从里到外或者从外向里)。

为什么要传播呢？因为事件源本身（可能）并没有处理事件的能力，即处理事件的函数（方法）并未绑定在该事件源上。

例如我们点击一个按钮时，就会产生一个click事件，但这个按钮本身可能不能处理这个事件，事件必须从这个按钮传播出去，从而到达能够处理这个事件的代码中（例如我们给按钮的onclick属性赋一个函数的名字，就是让这个函数去处理该按钮的click事件）

```
document.getElementById("d1").onclick = function(e){
        console.log("d1");
}
```

事件委托

事件委托，通俗地来讲，就是把一个元素响应事件（click、keydown……）的函数委托到另一个元素；

一般来讲，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，当事件响应到需要绑定的元素上时，会通过事件冒泡机制从而触发它的外层元素的绑定事件上，然后在外层元素上去执行函数。

举个例子，比如一个宿舍的同学同时快递到了，一种方法就是他们都傻傻地一个个去领取，还有一种方法就是把这件事情委托给宿舍长，让一个人出去拿好所有快递，然后再根据收件人一一分发给每个宿舍同学；

在这里，取快递就是一个事件，每个同学指的是需要响应事件的 DOM 元素，而出去统一领取快递的宿舍长就是代理的元素，所以真正绑定事件的是这个元素，按照收件人分发快递的过程就是在事件执行中，需要判断当前响应的事件应该匹配到被代理元素中的哪一个或者哪几个。

举例说明：

1、傻傻地一个个去领取

```
var list = document.getElementById("list");
      var li = list.getElementsByTagName("li");
      for(var i=0; i<li.length; i++) {
        li[i].onclick = function(){
          this.style.color = 'red';
        }
      }
```

2、委托 将事件绑定到父节点，点击子节点通过事件冒泡到父并处理

```
var list = document.getElementById("list");
      list.onclick = function(e){
        if (e.target.nodeName === 'LI') {
          e.target.style.color = 'red';
        }
      }
```

递归

程序调用自身的编程技巧称为递归（ recursion）。递归做为一种算法在程序设计语言中广泛应用。 一个过程或函数在其定义或说明中有直接或间接调用自身的一种方法。

递归，就是在运行的过程中调用自己。

构成递归需具备的条件

1. 子问题须与原始问题为同样的事，且更为简单；
2. 不能无限制地调用本身，须有个出口，化简为非递归状况处理。

具体应用

```
function f1(){
   console.log("f1.......")
   f1();
}
f1();
自调，递归虽好，但运用不得当，会造成死循环
```

```
添加结束条件
var i=0;
function f1(){
   i++;
   console.log(i)
   if (i<5) {
       f1();
   }
}
f1();
```

```
用递归求 5 的阶乘
function fun(n){
    if (n == 1){
        return 1;
    }
    return n * fun(n-1);
}
console.log(fun(5))
解析过程
代码执行fun(5)函数,此时的x是5,执行的是5*fun(4),此时代码等待,
先执行fun(4),进入函数,执行的是4*fun(3),等待,
先执行的是fun(3),进入函数,执行3*fun(2),等待,
先执行fun(2),进入函数,执行 2*fun(1);等待,
先执行fun(1),执行的是x==1的判断,return 1,条件结束;
```

闭包

理解

很多同学在面试的时候都会被问到闭包是什么？举例说明下闭包的运用?

闭包（closure）是javascript的一大难点，也是它的特色。很多高级应用都要依靠闭包来实现。

要理解闭包，首先要理解javascript的全局变量和局部变量。

javascript语言的特别之处就在于：函数内部可以直接读取全局变量，但是在函数外部无法读取函数内部的局部变量。

```
function f1(){
       var a=10;
       function f2(){
          alert(a);
       }
   }
```

如何从外部读取函数内部的局部变量?

我们有时候需要获取到函数内部的局部变量，正常情况下，这是办不到的！只有通过变通的方法才能实现。那就是在函数内部，再定义一个函数。

闭包的概念

上面代码中的f2函数，就是闭包。
各种专业文献的闭包定义都非常抽象，我的理解是: 闭包就是能够读取其他函数内部变量的函数。
由于在javascript中，只有函数内部的子函数才能读取局部变量，所以说，闭包可以简单理解成"定义在一个函数内部的函数"。
所以，在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

```
能够读取其他函数内部变量的函数
      function f1(){
          var a=10;
          function f2(){
             alert(a);
          }
           f2()
       };
        f1();  //可以获取到局部变量a
```

其实也可以直接通过以下方式来获取局部变量

```
        function f1(){
            var a = 10;
            return a;
        }
        f1();
```

为什么需要闭包

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，不会在f1调用后被自动清除。
总结：局部变量无法共享和长久的保存，而全局变量可能造成变量污染，当我们希望有一种机制既可以长久的保存变量又不会造成全局污染。

```
既可以长久的保存变量又不会造成全局污染
     function f1() {
            var a=10;
            function f2(){
                a++;
                console.log(a);
            };
            return f2;
        };
        var f = f1();
        f();
```

闭包的实际应用

使用闭包，我们可以做很多事情，比如缓存局部变量，避免污染

```
function Person(){
      var name = "张三";
      return {
          getName: function(){
              return name;
          },
          setName: function(newName) {
              name = newName;
          }
      }
  };
  var p = Person();
  console.log(p.getName());
  p.setName("xxxxx");
  console.log(p.getName())
```

总结：闭包就是一个函数引用另外一个函数的变量，因为变量被引用着所以不会被回收，因此可以用来封装私有变量，这是优点也是缺点，不必要的闭包只会徒增内存消耗！

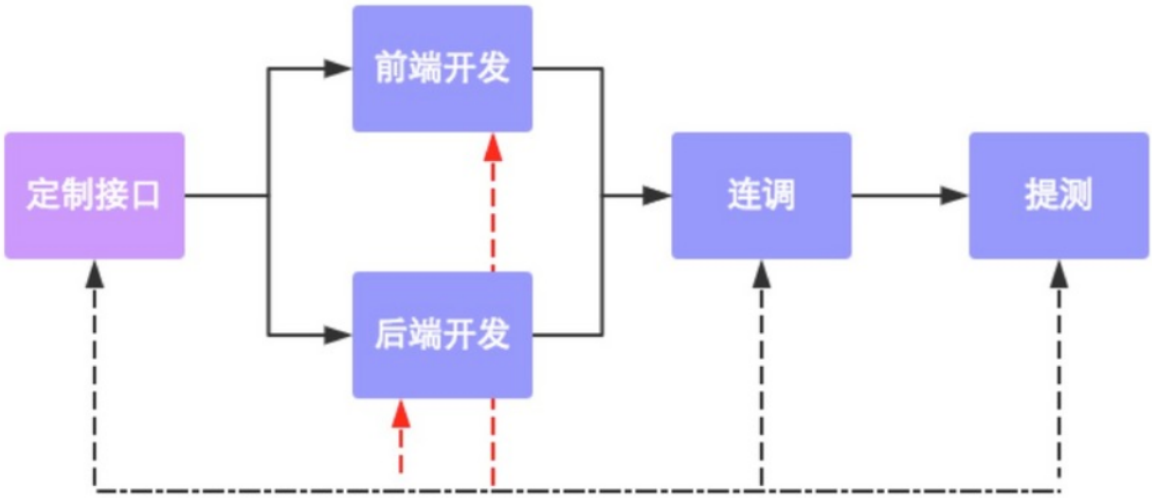
前后分离-数据交互

为什么要前后分离

前后分离——开发阶段，前后端工程师约定好数据交互接口，实现并行开发和测试；在运行阶段前后端分离模式需要对web应用进行分离部署，使用HTTP或者其他协议进行交互请求。

在以前传统的网站开发中，前端一般扮演的只是切图的工作，简单地将UI设计师提供的原型图实现成静态的HTML页面，而具体的页面交互逻辑，比如与后台的数据交互工作等，可能都是由后台的开发人员来实现的，这也就导致了前后端工作分配不均。这样做不仅仅开发效率慢，代码也难以维护。而前后端分离的话，则可以很好的解决前后端分工不均的问题，将更多的交互逻辑分配给前端来处理，而后端则可以专注于其本职工作，像提供API接口，进行权限控制以及进行运算工作。前端可以独立完成与用户交互的整一个过程，两者都可以同时开工，不互相依赖，开发效率更快，而且分工比较均衡。

数据接口规范流程



http协议

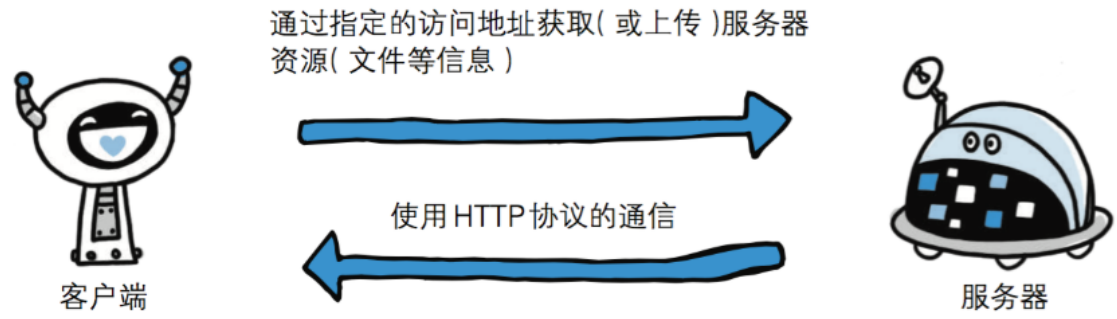
理解

HTTP是一个客户端终端（用户）和服务器端（网站）请求和应答的标准（TCP）。通过使用网页浏览器、网络爬虫或者其它的工具，客户端发起一个HTTP请求到服务器上指定端口（默认端口为80）。

通常，由HTTP客户端发起一个请求，创建一个到服务器指定端口（默认是80端口）的TCP连接。HTTP服务器则在那个端口监听客户端的请求。一旦收到请求，服务器会向客户端返回一个状态，比如"HTTP/1.1 200 OK"，以及返回的内容，如请求的文件、错误消息、或者其它信息。

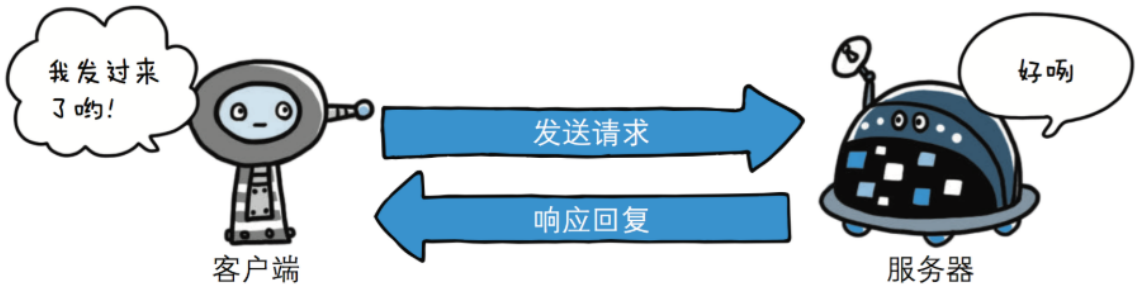
工作原理

HTTP协议定义Web客户端如何从Web服务器请求Web页面，以及服务器如何把Web页面传送给客户端。HTTP协议采用了请求/响应模型。客户端向服务器发送一个请求报文，请求报文包含请求的方法、URL、协议版本、请求头部和请求数据。服务器以一个状态行作为响应，响应的内容包括协议的版本、成功或者错误代码、服务器信息、响应头部和响应数据。



基于 请求-响应 的模式

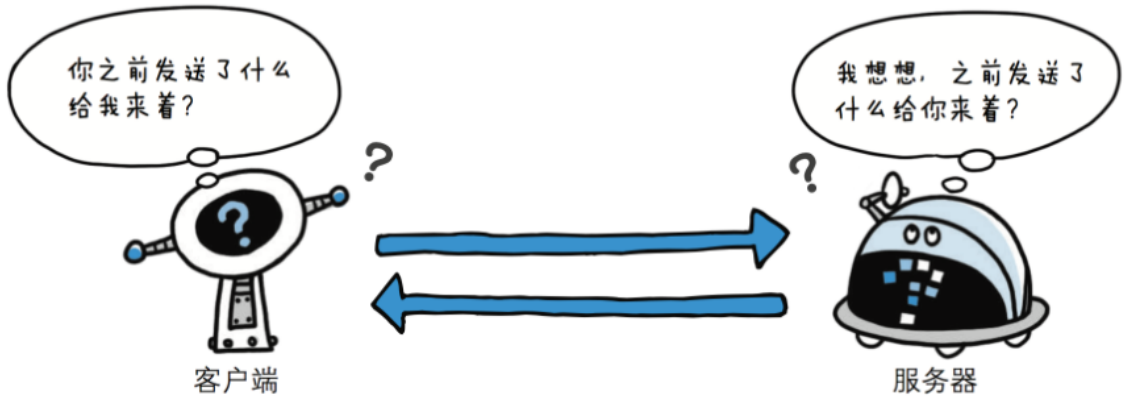
HTTP协议规定,请求从客户端发出,最后服务器端响应该请求并 返回。换句话说,肯定是先从客户端开始建立通信的,服务器端在没有 接收到请求之前不会发送响应



图：请求必定由客户端发出，而服务器端回复响应

无状态保存

HTTP是一种不保存状态,即无状态(stateless)协议。HTTP协议 自身不对请求和响应之间的通信状态进行保存。也就是说在HTTP这个 级别,协议对于发送过的请求或响应都不做持久化处理。



图：HTTP 协议自身不具备保存之前发送过的请求或响应的功能

HTTP请求方法

HTTP协议中共定义了八种方法（也叫“动作”）来以不同方式操作指定的资源：

GET

向指定的资源发出“显示”请求。使用GET方法应该只用在读取数据，而不应当被用于产生“副作用”的操作中，例如在Web Application中。其中一个原因是GET可能会被网络蜘蛛等随意访问。

HEAD

与GET方法一样，都是向服务器发出指定资源的请求。只不过服务器将不传回资源的本文部分。它的好处在于，使用这个方法可以在不必传输全部内容的情况下，就可以获取其中“关于该资源的信息”（元信息或称元数据）。

POST

向指定资源提交数据，请求服务器进行处理（例如提交表单或者上传文件）。数据被包含在请求本文中。这个请求可能会创建新的资源或修改现有资源，或二者皆有。

PUT

向指定资源位置上传其最新内容。

DELETE

请求服务器删除Request-URI所标识的资源。

TRACE

回显服务器收到的请求，主要用于测试或诊断。

OPTIONS

这个方法可使服务器传回该资源所支持的所有HTTP请求方法。用'*'来代替资源名称，向Web服务器发送OPTIONS请求，可以测试服务器功能是否正常运作。

CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。通常用于SSL加密服务器的链接（经由非加密的HTTP代理服务器）。

get与post请求的区别

- GET提交的数据会放在URL之后，也就是请求行里面，以?分割URL和传输数据，参数之间以&相连，如EditBook?name=test1&id=123456； POST方法是把提交的数据放在HTTP包的请求体中。因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变
- GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制.

HTTP状态码

所有HTTP响应的第一行都是状态行，依次是当前HTTP版本号，3位数字组成的状态代码，以及描述状态的短语，彼此由空格分隔。

状态代码的第一个数字代表当前响应的类型：

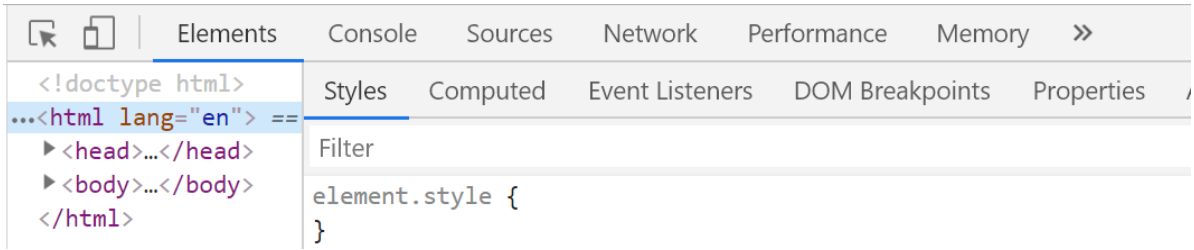
	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

常见状态码：

```
200 OK                          //客户端请求成功
400 Bad Request                 //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized                //请求未经授权，这个状态代码必须和WWW-Authenticate报头域
一起使用
403 Forbidden                   //服务器收到请求，但是拒绝提供服务
404 Not Found                   //请求资源不存在，eg：输入了错误的URL
500 Internal Server Error       //服务器发生不可预期的错误
503 Server Unavailable          //服务器当前不能处理客户端的请求，一段时间后可能恢复正常
```

浏览器调试工具

以chrome为例



1.箭头按钮

用于在页面选择一个元素来审查和查看它的相关信息，当我们在Elements这个按钮页面下点击某个Dom元素时，箭头按钮会变成选择状态

2.设备图标

可以切换到不同的终端进行开发模式，移动端和pc端的一个切换，可以选择不同的移动终端设备，同时可以选择不同的尺寸比例

3.Elements

该面板显示了渲染完毕后的全部HTML源代码，用来查看、修改页面上的元素，包括DOM标签，以及css样式，方便对静态网页进行调试。

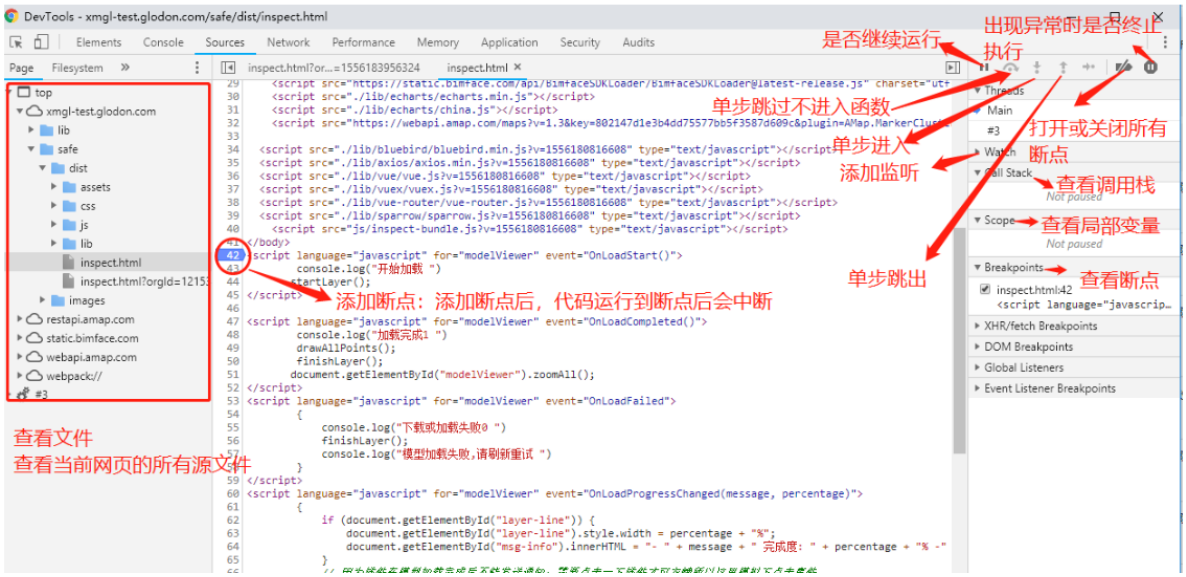
4.Console

该面板用来显示网页加载过程中的日志信息，包括打印，警告，错误及其他可显示的信息等，同时也是一个js交互控制台。

5.Sources

该面板以站点为分组，存放着请求下来的所有资源(html,css,jpg,gif,js等)。正是因为该面板存放了所有的资源，因此在调试js时，目标代码都在此处寻找，方便断点调试

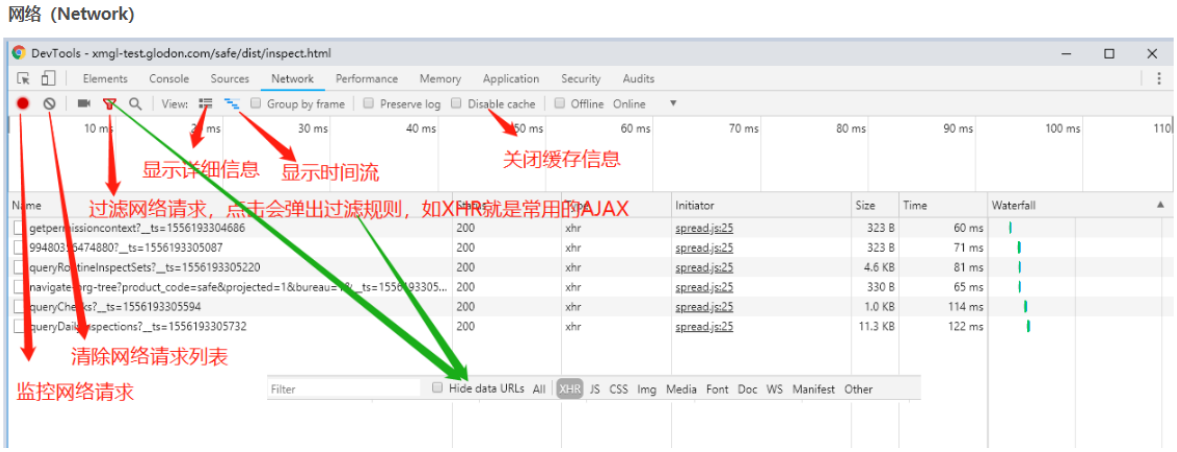
源代码（Sources）



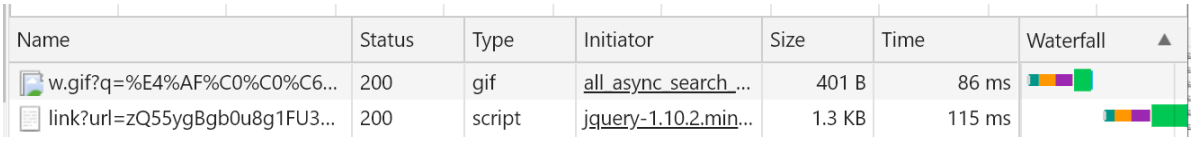
6.Network

网络请求标签页：可看到所有的资源请求，包括网络请求，图片资源，html,css，js文件等请求，可以根据需求筛选请求项，一般多用于网络请求的查看和分析，分析后端接口是否正确传输，获取的数据是否准确，请求头，请求参数的查看

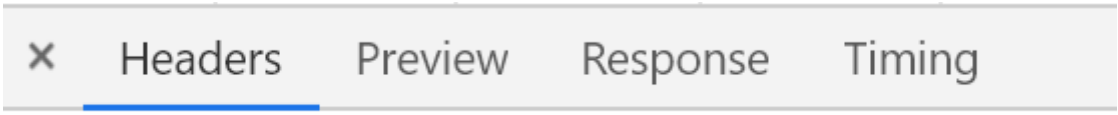
网络（Network）



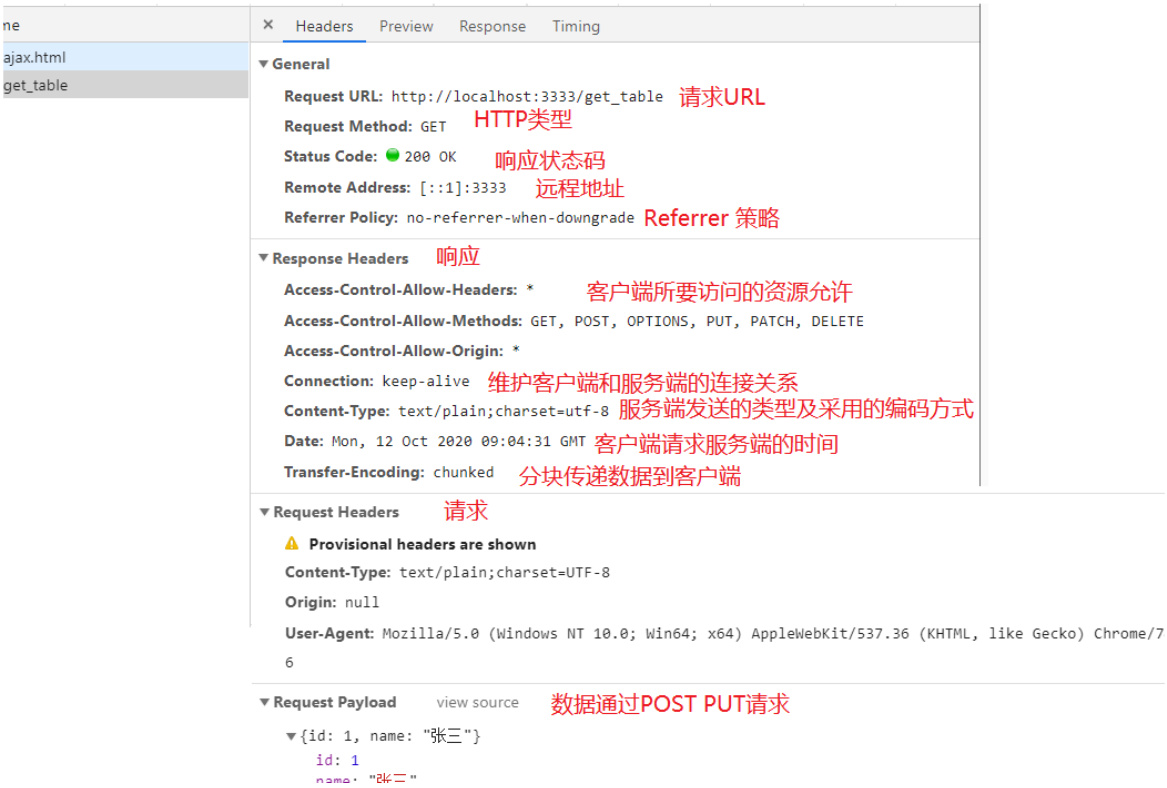
查看Network基本信息：URL，响应状态码，响应数据类型，响应数据大小，响应时间



请求文件的具体介绍：



- Header：面板列出资源的请求url、HTTP方法、响应状态码、请求头和响应头及它们各自的值、请求参数等等
- Preview：预览面板，用于资源的预览。
- Response：响应信息面板包含资源还未进行格式处理的内容
- Timing：资源请求的详细信息花费时间



7.Performance(旧版浏览器为Timeline)

时间表可以记录和运行分析应用程序所有的活动，为了使的记录页面的交互，打开时间轴面板，然后按开始录制录制按钮

8.Memory(旧版为Profiles):

可以查看CPU执行时间与内存占用

9.Application(旧版为Resources)

会列出所有的资源，以及HTML5的Database和LocalStore等，你可以对存储的内容编辑和删除

10.Security

可以告诉你这个网站的安全性，查看有效的证书等

11.Audits

可以帮你分析页面性能，有助于优化前端页面，分析后得到的报告

HTTP

HTTP协议即超文本传送协议(Hypertext Transfer Protocol)，是Web联网的基础，也是手机联网常用的协议之一，HTTP协议是建立在TCP协议之上的一种应用(`TCP是传输层，而http是应用层`)。

HTTP的交互流程简单来讲就是客户端与服务器端的通信，包括客户端对服务器端的请求以及服务器端对客户端的响应。

客户端与服务器端建立一个连接，三次握手经历完成之后才能建立一个稳定可靠的连接。

三次握手

第一次握手：客户端给服务器端发送一个syn的标志位；服务器端接收到syn后会返回一个ack(相当于一个回调的机制)，同时还有一个服务器端的syn；客户端接收服务器端发送的syn后会再次给服务器端发送一个ack，这样才算完成三次握手。

四次分手

客户端向服务器端发送断开连接的请求；服务器端接收到请求后，返回可以断开连接的请求，客户端断开连接并且释放资源；服务器端向客户端发送断开连接的信息；客户端向服务器端发送同意断开连接的信息，服务器端断开连接释放资源。

建立连接要3次，断开为什么要4次呢

因为tcp是全双工的，每个方向要单独断开，每个方向2次，所以4次。

协议规范

HTTP协议是一个规范。一定会限制请求的格式。
http协议的请求格式分为3个部分：请求行、请求头、请求体。

请求行

包括三个属性。描述对应的请求的时候，最精确的形式就是K-V键值对的格式。
请求头中也是一堆的K-V数据。包含头信息中的一些附加信息(比如客户端允许接收的信息格式)。

请求头

提供了关于请求,响应或者其他的发送实体的信息

Content-Type

- text/plain 纯文本
- application/x-www-form-urlencoded 指定内容类型为键值对
- application/octet-stream 二进制流数据
- application/json json数据格式
- multipart/form-data formdata表单数据格式

请求体

当发送某一个请求的时候，请求后面可以加一些用户定义的参数(比如表单)。以K=V的形式发送给后台。

服务启动

什么是Node?

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时。

开发框架

- Express 快速、开放、极简的 Web 开发框架。
- Koa 下一代 web 开发框架。
- Egg 为企业级框架和应用而生。

启动node服务

创建包文件：`npm init`

安装依赖：`npm i -S express body-parser`

启动：`node xxx.js`

停止：`ctrl+c`

数据交互

原生请求

- xhr

```
var xmlhttp = getXMLHttpRequest();
            // xmlhttp.open("get", "http://localhost:3000/info?name=jack",
true);
            // xmlhttp.send();

            // json
            // xmlhttp.open("post", "http://localhost:3000/info4", true);
            // xmlhttp.setRequestHeader("Content-Type","application/json");
            // xmlhttp.send(JSON.stringify({"name":"joho", "age":20}));

            //formdata
            // xmlhttp.open("post", "http://localhost:3000/info41", true);
            // var formdata = new FormData();
            // formdata.append("name", "joho");
            // formdata.append("age", 20)
            // xmlhttp.send(formdata);

            xmlhttp.onreadystatechange = function(){
                if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
                    console.log(xmlhttp.responseText);
                }
            }
```

- Fetch

```
//get
            fetch('http://localhost:3000/info?name=jack')
              .then(function(response) {
                return response.json();
              })
              .then(function(myJson) {
                console.log(myJson);
              });
            //post
            fetch('http://localhost:3000/info4', {
                body: JSON.stringify({"name":"joho", "age":20}),
                headers: {
                    'content-type': 'application/json'
                },
                method: 'POST',
            })
              .then(function(response) {
                return response.json();
              })
              .then(function(myJson) {
                console.log(myJson);
              });
            //formdata
            var formdata = new FormData();
            formdata.append("name", "joho");
            formdata.append("age", 200)
            fetch('http://localhost:3000/info41', {
                body: formdata,
                method: 'POST',
            })
              .then(function(response) {
                return response.json();
              })
              .then(function(myJson) {
                console.log(myJson);
              });
```

参数解析

获取请求很中的参数是每个web后台处理的必经之路，nodejs的 **express框架** 提供了3种方法来实现。

1. req.body

   包含了提交数据的键值对在**请求体**中，默认是underfined,
   你可以用body-parser或者multer来解析body

   对应前端传入参数及内容类型如下：

   `application/json`

   `{"name"="jack"}`

   `req.body.name`

2. req.query

   包含在**路由**中每个查询字符串参数属性的对象。如果没有，默认为{}

注：此方法多适用于GET请求，解析GET里的参数

`application/x-www-form-urlencoded`

```
// GET /search?q=tobi+ferret
req.query.q
```

3. req.params

   包含映射到指定的路线“参数”属性的对象。
   例如，如果你有route/user/：name，那么“name”属性可作为req.params.name。
   该对象默认为{}。

```
// GET /user/tj
req.params.name
// => "tj"
```

   多适用于restful风格url中的参数的解析

4. Formdata

   处理formdata需要下载依赖包multiparty。

```
npm i multiparty -S

router.post("/test", function(req, res) {
    var form = new multiparty.Form();
    form.parse(req, function(err, fields, files) {
           //fields：类似post 的一些字符串，
           //files 文件
    })
})
```

req.query与req.params区别
req.params包含路由参数（在URL的路径部分），而req.query包含URL的查询参数（在URL的？后的参数）。

跨域

什么是跨域

跨域是指从一个域名的网页去请求另一个域名的资源。比如从www.baidu.com 页面去请求 www.google.com 的资源。跨域的严格一点的定义是：只要**协议**，**域名**，**端口**有任何一个的不同，就被当作是**跨域**。

为什么浏览器要限制跨域访问呢?

原因就是安全问题：如果一个网页可以随意地访问另外一个网站的资源，那么就有可能在客户完全不知情的情况下出现安全问题。

为什么要跨域（从一个页面引用其它页面的资源）

既然有安全问题，那为什么又要跨域呢？有时公司内部有多个不同的子域，比如一个是location.company.com ,而应用是放在app.company.com , 这时想从 app.company.com去访问location.company.com 的资源就属于跨域

跨域解决方案

- cors
- nginx
- jsonp

cors

```
app.use(function (req, res, next) {
 res.setHeader('Access-Control-Allow-Origin', '*');
 res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT,
DELETE');
 res.setHeader('Access-Control-Allow-Headers', '*');
 next();
});
```

nginx

Windows: 在nginx目录查找nginx.conf文件并添加以下内容

Mac: open -t /usr/local/etc/nginx/nginx.conf

mac默认ng端口8080

```
server {
        listen       8089;
        server_name  localhost;
        root   html;    #根目录
        # cors
        add_header Access-Control-Allow-Origin $http_origin always;  # '*'
        add_header Access-Control-Allow-Credentials true always;
        add_header Access-Control-Allow-Methods 'GET, POST, OPTIONS' always;
        add_header Access-Control-Allow-Headers 'DNT,X-Mx-ReqToken,Keep-
Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-
Type,Authorization' always;

        #请求http://localhost:8080/api，将该请求转发到http://localhost:3000/api
        location /api {
            proxy_pass http://localhost:3000/api;
        }
    }
```

关于Mac下操作:

`brew services start nginx`

`brew services stop nginx`

jsonp

```
app.get("/api/jsonp", function(req, res){
    var data = {name: "jsonp数据"}
    data = JSON.stringify(data); //转字符串
    var callback = `${req.query.callback}(${data})`; //函数名+数据
    console.log(callback)
    res.send(callback);
})
function getJsonp() {
```

```
    $.ajax({
      url: "http://localhost:3000/api/jsonp",
      data: {name: "xxxx"},
      dataType: "jsonp",
      jsonpCallback: "handleResponse"
    })
};
function handleResponse(data) {
    console.log(data);
}
```

设计模式

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

1995 年，GoF（Gang of Four，四人组/四人帮）合作出版了《设计模式：可复用面向对象软件的基础》一书，共收录了 23 种设计模式，从此树立了软件设计模式领域的里程碑，人称「GoF设计模式」。

这些模式可以分为三大类：**创建型模式**（Creational Patterns）、**结构型模式**（Structural Patterns）、**行为型模式**（Behavioral Patterns）。

创建型模式

- 工厂模式（Factory Pattern）
- 抽象工厂模式（Abstract Factory Pattern）
- 单例模式（Singleton Pattern）
- 建造者模式（Builder Pattern）
- 原型模式（Prototype Pattern）

结构型模式

- 适配器模式（Adapter Pattern）
- 桥接模式（Bridge Pattern）
- 过滤器模式（Filter、Criteria Pattern）
- 组合模式（Composite Pattern）
- 装饰器模式（Decorator Pattern）
- 外观模式（Facade Pattern）
- 享元模式（Flyweight Pattern）
- 代理模式（Proxy Pattern）

行为型模式

- 责任链模式（Chain of Responsibility Pattern）
- 命令模式（Command Pattern）
- 解释器模式（Interpreter Pattern）
- 迭代器模式（Iterator Pattern）
- 中介者模式（Mediator Pattern）
- 备忘录模式（Memento Pattern）
- 观察者模式（Observer Pattern）
- 状态模式（State Pattern）
- 空对象模式（Null Object Pattern）
- 策略模式（Strategy Pattern）

- 模板模式（Template Pattern）
- 访问者模式（Visitor Pattern）

设计模式的六大原则

1. **开闭原则（Open Close Principle）**

   意思是：**对扩展开放，对修改关闭。**

2. **里氏代换原则（Liskov Substitution Principle）**

   里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。

3. **依赖倒转原则（Dependence Inversion Principle）**

   具体内容：针对接口编程，依赖于抽象而不依赖于具体

4. **接口隔离原则（Interface Segregation Principle）**

   使用多个隔离的接口，比使用单个接口要好，降低类之间的耦合度。。

5. **迪米特法则，又称最少知道原则（Demeter Principle）**

   一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

6. **合成复用原则（Composite Reuse Principle）**

   尽量使用合成/聚合的方式，而不是使用继承。

其实设计模式就是从**大型软件架构**出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

观察者模式

当一个对象数据发生变化时，通知其它一系列对象，让其响应这种变化

观察者设计模式中主要区分两个概念：

观察者：指观察者对象，也就是消息的订阅者；

被观察者：指要观察的目标对象，也就是消息的发布者。

```
class Subject{//发布者
  constructor(){
    this.subs = [];
  }
  addSub(sub){
    this.subs.push(sub);
  }
  notify(food){
    this.subs.forEach(sub=> {
      sub.update(food);
    });
  }
}
class Observer{//订阅者
  constructor(name, food){
    this.name = name;
    this.food = food;
  }
  update(food){
    if(food===this.food){
        console.log(this.name + "的外卖："+food);
    }
  }
```

```
}
var subject = new Subject();
var tom = new Observer("tom","地三鲜");
var jack = new Observer("jack","红烧肉");
//目标添加观察者了
subject.addSub(tom);
subject.addSub(jack);
//目标发布消息调用观察者的更新方法了
subject.notify("地三鲜");
subject.notify("红烧肉");
```

优点：

1. **观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者**角色所知道的只是一个具体**观察者列表**，每一个具体观察者都符合一个抽象观察者的接口。**被观察者并不认识任何一个具体观察者**，它只知道它们都有一个**共同的接口。**由于被观察者和观察者没有紧密地耦合在一起，因此它们可以**属于不同的抽象化层次。**如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。
2. **观察者模式支持广播通讯。**被观察者会向所有的登记过的观察者发出通知。

代理模式

代理是一个对象，跟本体对象**具有相同的接口**，以此达到对本体对象的访问控制。

代理，顾名思义，即**代替被请求者来处理相关事务。**代理对象一般会全权代理被请求者的全部只能，客户访问代理对象就像在访问被请求者一样，虽然代理对象最终还是可能会访问被请求者，但是其可以在请求之前或者请求之后进行一些额外的工作，或者说客户的请求不合法，直接拒绝客户的请求。

```
// 声明女孩对象
var girl = function (name) {
    this.name = name;
};
// 声明男孩对象
var boy = function (girl) {
    this.girl = girl;
    this.sendGift = function (gift) {
        alert("Hi " + girl.name + ", 男孩送你一个礼物：" + gift);
    }
};
// 声明代理对象
var proxyObj = function (girl) {
    this.girl = girl;
    this.sendGift = function (gift) {
        (new boy(girl)).sendGift(gift); // 替dudu送花咯
    }
};
var proxy = new proxyObj(new girl("花花"));
proxy.sendGift("999朵玫瑰");
```

优点：代理对象可以代替本体对象被实例化，此时本体对象未真正实例化，等到合适时机再实例化。

代理模式可以**延迟创建开销很大的本体对象**，他会把本体的实例化推迟到有方法被调用时。

工厂模式

最常用的设计模式之一，这种类型的设计模式属于创建型模式,它提供了一种创建对象的最佳方式。

```
// function
var MobileFactory = (function () {
    var Mobile = function (name,model){
        this.model = model;
        this.name = name;
    };
    Mobile.prototype.play=function(){
        console.log("Mobile:"+this.name+"-"+this.model)
    }
    return function (name,model) {
        return new Mobile(name,model);
    };
})();

var p6 = new MobileFactory("iphone", "6");
var px = new MobileFactory("iphone", "X");
p6.play()
px.play()
//Creator是个工厂，有个create函数，工厂通过create创建Product
class Product {
    constructor(name, model) {
        this.name = name
        this.model = model;
    }
    play(){
        console.log("Mobile:"+this.name+"-"+this.model)
    }
}

class FactoryCreator {
    create(name,model) {
        return new Product(name,model)
    }
}
let creator = new FactoryCreator()
// 通过工厂省城product的实例
let p = creator.create('iphone',"6")
p.play();
let p2 = creator.create('iphone',"7")
p2.play()
```

优点

1. 工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅"消费"产品。工厂模式通过这种做法实现了对责任的分割。
2. 当产品有复杂的多层等级结构时，工厂类只有自己，以不变应万变，就是模式的缺点。因为工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。

缺点

1. 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，有可能造成工厂逻辑过于复杂, **违背了"开放--封闭"原则(OCP)** .

单例模式

系统中被唯一使用，一个类只有一个实例

```
class Store {
    action() {
        console.log('vue store.')
    }
}
// 定义一个静态的方法，将方法挂载到class上面,无论SingleObject被new多少个，getInstance的方法只有一个
Store.getInstance = (function() {
    let instance
    return function() {
        if (!instance) {
            instance = new Store();
        }
        return instance
    }
})()

// js中的单例模式只能靠文档去约束，不能使用private关键字去约束
// 测试：注意这里只能使用静态函数getInstance,不能使用new Store()
let obj1 = Store.getInstance()
obj1.action()
let obj2 = Store.getInstance()
obj2.action()

// 单例模式(唯一的)，每次获取的都是一个东西，所以他两相等，否则就不是单例模式
console.log(obj1 === obj2) //true
```

优点：

1. 在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例
2. 避免对共享资源的多重占用。

适配器模式

将原本不适合的数据转换为适合的数据(计算属性)。

将原本不适合的接口转换为适合的接口。

```
class IPhoneX {
    constructor(){
        this.interface = "平接口type-c";
    }
}

class Adapter {
    constructor(oldInter, newInter) {
        this.phone = new IPhoneX() // 初始化实例
        this.oldInter = oldInter
        this.newInter = newInter
    }
    getOldInter() {
        return this.oldInter
```

```
    }
    translate(p, newInter){ //模拟转接头
        console.log(`${p.interface}---->>${newInter}`)
    }
    getInter() { // 覆盖
        this.translate(this.phone, this.newInter);
        return this.newInter
    }
}

let adapter = new Adapter('圆接口', '平接口')
let res = adapter.getInter()
console.log(res) // 圆接口

res = adapter.getOldInter()
console.log(res) // 平接口
```

优点：

1. 将**目标类**和**适配者类**解耦.
2. 增加了类的透明性和复用性，将具体的实现封装在适配者类中.

装饰器模式

允许向一个现有的对象添加新的功能，同时又不改变其结构。

比喻：

假如我有一个蛋糕，如果在上面加上奶油其他什么都不加，那么它就成了一个奶油蛋糕。如果再加上草莓，那就是草莓奶油蛋糕，如果再加上一块巧克力板，上面写上姓名，然后插上蜡烛，就变成了一块生日蛋糕。

不论是蛋糕，奶油蛋糕，草莓蛋糕还是生日蛋糕，它们的核心都是蛋糕。不过，在加上一系列装饰之后，它变得更加甜美了，目的也更加明确了。装饰器模式中的被装饰对象和蛋糕很相似。

```
class Circle {
    draw() {
        console.log('我要画一个圆')
    }
}

class Decorator {
    constructor(circle) {
        this.circle = circle
    }
    draw() {
        this.circle.draw()
        this.setBorder(circle)
    }
    setBorder(circle) {
        console.log('加上边框')
    }
}
// 想引用某个类的时候将他实例化，以参数的形式进行传递进去
let circle = new Circle()
circle.draw()
console.log('+++++')
let dec = new Decorator(circle)
```

```
dec.draw()
```

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式可以动态扩展一个实现类的功能。

装饰器模式和代理模式的区别：

装饰器模式关注于在一个对象上动态的添加方法，然而**代理模式关注于控制对对象的访问。**换句话说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模式的时候，我们通常的做法是**将原始对象作为一个参数传给装饰者的构造器。**

装饰器模式与适配器模式的区别

装饰器与适配器都有一个别名叫做**包装模式(Wrapper)**，它们看似都是起到包装一个类或对象的作用，但是使用它们的目的很不一一样。**适配器模式的意义是要将一个接口转变成另一个接口，它的目的是通过改变接口来达到重复使用的目的。**
而装饰器模式**不是要改变被装饰对象的接口**，而是恰恰要保持原有的接口，但是增强原有对象的功能，或者改变原有对象的处理方式而提升性能。所以这两个模式设计的目的是不同的。

设计模式分析

依赖具体：

老板要造一辆小车，找最好的工匠制作最好的零部件，如轮胎，底盘等，由于长时间使用一些零件开始损坏，如轮胎破了需要再做新的，再找工匠对比之前的轮胎制作。

依赖抽象：

老板要造一辆小车，先找最好的设计人员设计出详细的尺寸，规格等，并找工匠实现出原型产品，当原型产品损坏后，可以再照着设计图再制作轮胎，自己制作还可以扩展，如加上自己的签名，特别的花纹等。

组件封装

封账就是隐藏实现细节，仅对外提供访问的接口。内部的具体实现细节我不关心。

封装好处

- 模块化，分工明确
- 信息隐藏
- 代码重用

HttpAjax组件封装

```
// 封装原生XMLHttpRequest请求
function createXMLHttp(baseUrl) {
    // 低版本IE兼容
    var XmlHttp = window.XMLHttpRequest ? new XMLHttpRequest() :
ActiveXobject("Msxml12.XMLHTTP");
    // 默认请求 协议、IP、port
    var defaultURL = baseUrl ? baseUrl : "http://localhost:3000";
    return {
        sendRequest({method='GET', url, data=null, success, error}) {
            XmlHttp.open(method, defaultURL + url);
            if (method == "GET") {
                XmlHttp.send();
            } else {
                XmlHttp.setRequestHeader("Content-Type", "application/json")
```

```
                XmlHttp.send(data);
            }
            XmlHttp.onreadystatechange = function (resp) {
                if (XmlHttp.readyState == 4 && XmlHttp.status == 200) {
                    success(resp);
                }
            };
            XmlHttp.onerror = function (err) {
                error(err);
            }
        }
    }
}
```