

JS高频笔试题精讲——ES6专题

Web学科教学总监

张东 (东神)



高频笔试题包括







Q1: ES6知识点总结

1. 模板字符串 。

今后 只要拼接字符串 一律用 模板字符串

- \${}里:
 - 可以放:

变量、算术计算、三目、对象属性、创建对象、调用函数、访问数组元素

- ——有返回值的合法的is表达式
- 不能放:

没有返回值的js表达式,

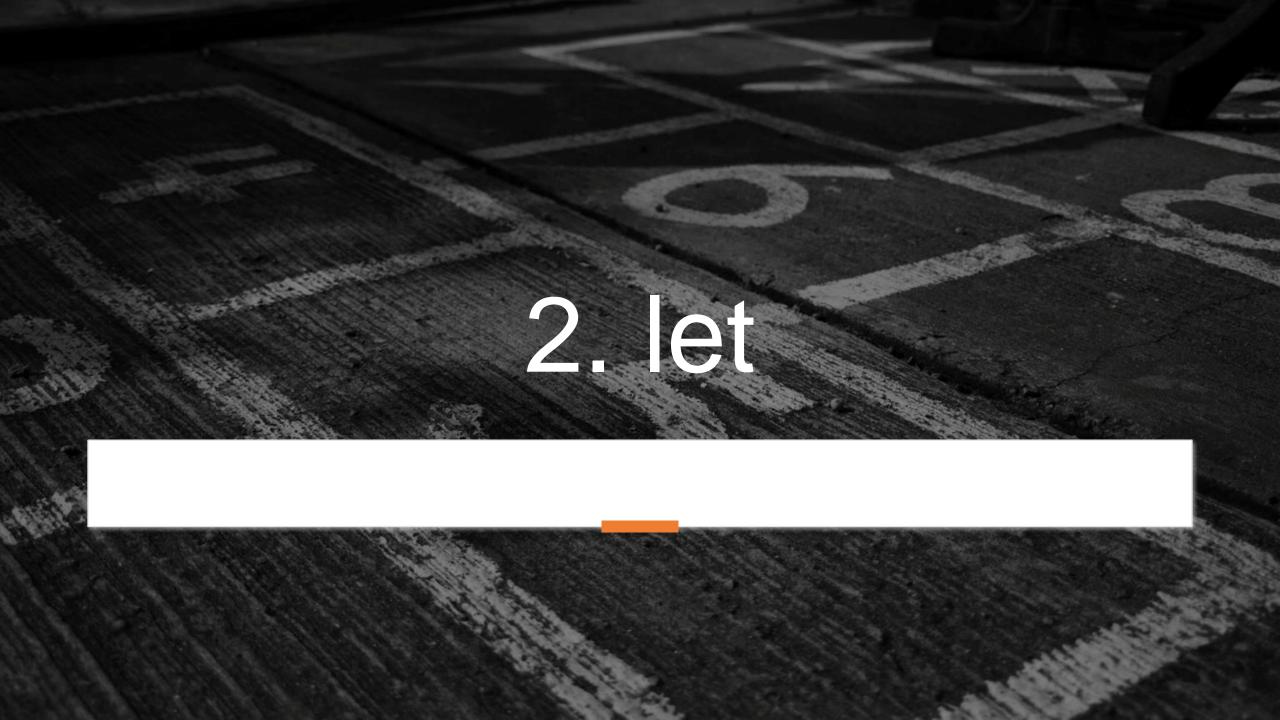
也不能放分支/判断、循环等程序结构。

比如: if else for while...等

\${}规则和今后各种框架中的绑定语法规则完全一样!



```
var uname="丁丁";
 console.log(`Welcome ${uname}`);
var sex=1;
                                                                                                                                                                                                                                         输出结果:
 console.log(`性别:${sex==1?"男":"女"}`);
var price=12.5;
                                                                                                                                                                                                                                       ■ Welcome 丁丁
var count=5;
                                                                                                                                                                                                                                           性别:男
console.log(`
                                                                                                                                                                                                                                          单价:¥12.50
         单价:\frac{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\parallel{\paral
                                                                                                                                                                                                                                          数量:5
         数量:${count}
                                                                                                                                                                                                                                          小计:¥62.50
         下单时间:2021/9/2 上午10:27:34
 //复习第一阶段Date部分
var orderTime=1630549654731;//new Date().getTime()
 console.log(`下单时间:${new Date(orderTime).toLocaleString()}`);
var arr=["日","一","二","三","四","五","六"];
//getDay()0 1 2 3 4 5
var day=new Date().getDay();
 console.log(`今天星期${arr[day]}`);
```



var的问题

- a. 会被声明提前
- ——打乱程序正常的执行顺序
- b. 没有块级作用域
- ——代码块内的变量会超出代码块范围,影响外部的变量

什么是块级作用域:

- (js中没有, 其它语言才有)
- · 指除了对象{}和function的{}之外,
- 其余if else、for等分支和循环结构的{}范围
- 在其他语言中称为块级作用域。
- 其他语言中,程序块{}内的变量,出了所在的程序块{},就不能使用

比如: 其他语言中

var i;

Js中:

```
for(var i=0;i<arr.length;i++){</li>...}
```

• console.log(i);//正常使用

再比如: 其他语言中

```
var i;
if(条件){
    var i=0;
    i=0;
}else{
    var i=1;
}
console.log(i); //报错: i未定义

var i;
if(条件){
    i=0;
    }else{
    i=1;
    }
    console.log(i); //正确
```

JS中:

```
if(条件){
  var i=0;
}else{
  var i=1;
}
console.log(i); //照样可用
```

示例: var的问题

var t=0; //全局变量t, 来记录程序的执行时间



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
console.log(`执行任务1, 耗时0.8s`);
t+=0.8; //想将fun1的执行时间累加到全局变量t上
}
function fun2(){//第二个函数, 执行, 耗时0.3s
console.log(`执行任务2, 耗时0.3s`)
t+=0.3; //将fun2的执行时间累加到全局变量t上
}
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`);//1.1 正确
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console log(`执行任务1, 耗时0.8s`);
 t+#0.8; / 想将fun1的执行时间累加到全局变量t上
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1 正确
```



后来

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console log(`执行任务1, 耗时0.8s`);
 t+#0.8; / 想将fun1的执行时间累加到全局变量t上
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1 正确
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   var t=new Date();
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`);//0.3 但是! 结果却少了0.8s
```



原因

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   var t=new Date();
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`);//0.3 但是! 结果却少了0.8s
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   <del>var t=new Date();</del> //被自动翻译为以下2句话
   //var t;//声明: 被提前到当前函数的顶部
   //t=new Date();//赋值: 留在原地
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //0.3 错了! 少了0.8s
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数,执行,耗时0.8s
 //var t;//undefined fun1中有了局部变量t
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   <del>var_t=new_Date();</del>//被自动翻译为以下2句话
  ■//var t;//声明,被提前到当前函数的顶部
   //t=new Date();//赋值, 留在原地
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //0.3 错了! 少了0.8s
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数,执行,耗时0.8s
 //var t;//undefined fun1中有了局部变量t
 //fun1函数内再操作t时,就不会去全局找了! 而是所有操作都作用在局部变量t上了
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   <del>var_t=new_Date();</del>//被自动翻译为以下2句话
 //var t;//声明,被提前到当前函数的顶部
   //t=new Date();//赋值, 留在原地
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数,执行,耗时0.8s
 //var t;//undefined fun1中有了局部变量t
 //funt函数内再操作t时,就不会去全局找了! 而是所有操作都作用在局部变量t上了。
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //因fun1中有局部变量t, 所以, 0.8被加到局部变量t, 没加到全局t。
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   <del>var_t=new_Date();</del>//被自动翻译为以下2句话
   //var t;//声明,被提前到当前函数的顶部
   //t=new Date();//赋值, 留在原地
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
```

解决:

• 今后,只要声明变量,都推荐使用let代替var。

优点: 2个:

- a. 不会被声明提前
- ——保证程序顺序执行
- b. 让程序块, 也变成了"块级作用域"。
- --保证块内的变量,不会影响块外的变量。

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数,执行,耗时0.8s
 //var t;//undefined fun1中有了局部变量t
 //fun1函数内再操作t时,就不会去全局找了! 而是所有操作都作用在局部变量t上了
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //因fun1中有局部变量t, 所以, 0.8被加到局部变量t, 没加到全局t。
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   <del>var_t=new_Date();</del>//被自动翻译为以下2句话
   //var t;//声明,被提前到当前函数的顶部
   //t=new Date();//赋值, 留在原地
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 //var t;//undefined fun1中有了局部变量t
 //fun1函数内再操作t时,就不会去全局找了!而是所有操作都作用在局部变量t上了。
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //因fun1中有局部变量t,所以,0.8被加到局部变量t,没加到全局t。
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //0.3 错了! 少了0.8s
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//不是作用域
   let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //0.3 错了! 少了0.8s
```



```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 t+=0.8; //想将fun1的执行时间累加到全局变量t上
  //因为fun1中暂时没有局部变量t,所以t+=0.8,给了全局的t变量。
 //后来,在fun1中添加如下代码
 //因为if后条件为false, 所以if内的代码根本没执行!
 if(false){//也成了一级作用域
   //局部t,只能在{}范围内使用,与{}外部的t,毫无关系!
   let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //0.3 错了! 少了0.8s
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 //将fun 的执行时间累加到全局变量t上
 t+=0.8; 从因为fun1中暂时没有局部变量t,所以t+=0.8,给了全局的t变量。
 // 后来,在fun1中添加如下代码
 //因为if后条件为false,所以if内的代码根本没执行!
 if(false){//也成了一级作用域
   //局部t,只能在(}范围内使用,与{}外部的t,毫无关系!
  let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 //将fun1的执行时间累加到全局变量t上
 t+=0.8; //因为fun1中暂时没有局部变量t,所以t+=0.8,给了全局的t变量。
 //后来,在fun1中添加如下代码
 //因为if后条件为true, 所以if内的代码会执行!
 if(true){//也成了一级作用域
   //局部t,只能在{}范围内使用,与{}外部的t,毫无关系!
   let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 //将fun 的执行时间累加到全局变量t上
 t+=0.8; 从因为fun1中暂时没有局部变量t,所以t+=0.8,给了全局的t变量。
 // 后来,在fun1中添加如下代码
 //因为if后条件为true,所以if内的代码会执行!
 <u>if(true)</u>{//也成了一级作用域
   //局部t,只能在{}范围内使用,与{}外部的t,毫无关系!
   let t<del>-new Date()</del>; //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数,执行,耗时0.3s
 ¢onsole.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1
```

let本质:

• 底层会被翻译为匿名函数自调: (function(){

})()

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 //将fun1的执行时间累加到全局变量t上
 t+=0.8; //因为fun1中暂时没有局部变量t,所以t+=0.8,给了全局的t变量。
 //后来,在fun1中添加如下代码
 //因为if后条件为true, 所以if内的代码会执行!
 if(true){//也成了一级作用域
   //局部t,只能在{}范围内使用,与{}外部的t,毫无关系!
   let t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
console.log(`共耗时:${t}s`); //1.1
```

```
var t=0; //全局变量t, 来记录程序的执行时间
function fun1(){//第一个函数, 执行, 耗时0.8s
 console.log(`执行任务1, 耗时0.8s`);
 //将fun1的执行时间累加到全局变量t上
 t+=0.8; //因为fun1中暂时没有局部变量t, 所以t+=0.8, 给了全局的t变量。
 //后来,在fun1中添加如下代码
 //因为if后条件为true, 所以if内的代码会执行!
 if(true){//也成了一级作用域
   (function(){ //let底层相当于自动添加匿名函数自调
   //局部t,只能在{}范围内使用,与{}外部的t,毫无关系!
   var t=new Date(); //不会被声明提前
   console.log(`上线时间:${t.toLocaleString()}`)
   })();
function fun2(){//第二个函数, 执行, 耗时0.3s
 console.log(`执行任务2, 耗时0.3s`)
 t+=0.3; //将fun2的执行时间累加到全局变量t上
fun1();
fun2();
```

let的三个小脾气:

• a. 因为不会声明提前, 所以不能在声明变量之前, 提前使用该变量。

var 变量可提前使用

```
console.log(a);//undefined var a=10; //var a;被声明提前 console.log(a);//10
```

var 变量可提前使用

var a;//undefined console.log(a);//undefined a=10; //var a;被声明提前 console.log(a);//10

改成let声明变量,就不能提前使用

console.log(b);

ReferenceError:

Cannot access 'b' before initialization

let b=10;//初始化: 第一次声明变量并赋值

console.log(b);

底层相当于

- console.log(b);
- (function(){
- var b=10;//初始化: 第一次声明变量并赋值
- console.log(b);
- **})()**

let的三个小脾气:

- a. 因为不会声明提前, 所以不能在声明变量之前, 提前使用该变量。
- ・b. 在相同作用域内,禁止声明两个同名的变量!

旧js中同一作用域中可重复声明变量

- var a=10;
- var a=100;
- console.log(a);//100

使用let代替var后

• 相同作用域中,不能重复声明两个同名变量

let的三个小脾气:

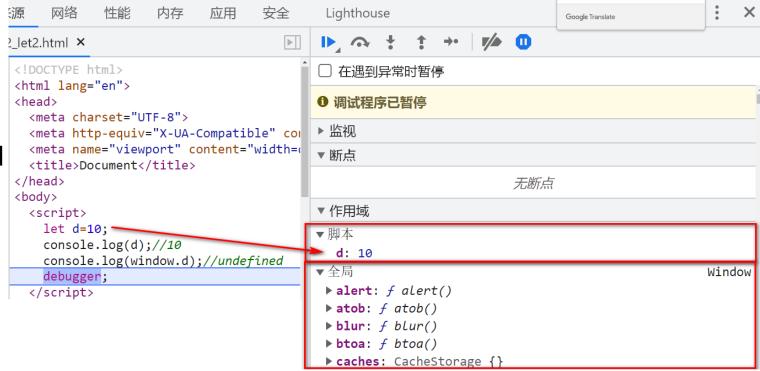
- a. 因为不会声明提前,所以不能在声明变量之前,提前使用该变量。
- b. 在相同作用域内,禁止声明两个同名的变量!
- ・c. 因为let底层相当于匿名函数自调,所以,即使在全局创建的let变量,在window中也找不到!

旧js中所有全局变量都保存在window

```
var a=10;
console.log(a); //10
console.log(window.a);//10
```

用let代替var后,

- let b=10;
- console.log(b); //10
- console.log(window.b);//undefined



原理

```
function(){
    let b=10;
    var b=10; //局部变量
    console.log(b); //10 局部变量
    console.log(window.b);//undefined
```

关于let底层相当于匿名函数自调,不接受反驳,自己看今天课前预习笔试题视频let_setTimeout let居然能形成闭包,所以let底层相当于什么……

```
})()
<script>
                                        ▼ 作用域
  (function(){
                                        ▼本地
    var d=10; d = 10
                                         ▶ this: Window
    console.log(d);//10
                                           d: 10
    console.log(window.d);//undefined
    debugger;
                                        ▼ 全局
                                                                                               Window
  })();
                                         ▶ alert: f alert()
</script>
                                         ▶ atob: f atob()
                                         ▶ blur: f blur()
                                         ▶ btoa: f btoa()
                                          ▶ caches: CacheStorage {}
```



赠: var, let, const区别

区别	var	let	const		
是否产生"块级作用域"	×	✓	✓		
是否会被声明提前	✓	×	×		
是否保存在window中	字在window中				
相同作用域中能否重复声明变量	作用域中能否重复声明变量				
是否能提前使用	✓	×	×		
是否必须设置初始值	×	×	✓		
能否修改实际保存在变量中的原始类型值或引用类型地址	✓	✓	×		

3. 箭头函数:

(上节已讲)



4. for of

问题:

• 如何遍历数字下标的数组或类数组对象

普通for循环:

```
需求1: 点名——可以实现
var arr=["涛涛","楠楠","东东"];
for(var i=0;i<arr.length;i++){
  console.log(`${arr[i]}-到!`)
}
```

需求2: 定义函数计算任意多个数的和——可实现

```
//定义一个函数,求任意多个数字的和
function add(
// arguments[
 var result=0;
 for(var i=0;i<arguments.length;i++){</pre>
  result+=arguments[i]
 return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```

普通for循环:

- 1). 优点: 既可遍历索引数组,又可以遍历类数组对象(arguments)——只要下标是数字
- 2). 缺点: 没有可简化的空间

forEach:

```
需求1: 点名——可以实现
var arr=["涛涛","楠楠","东东"];
arr.forEach(function(tname){
  console.log(`${tname} - 到!`)
})
```

需求1: 点名: 可更简化

```
var arr=["涛涛","楠楠","东东"];
arr.forEach(t=>console.log(`${t}-到!`));
```

需求2: 定义函数计算任意多个数的和 ——无法实现

```
//定义一个函数, 求任意多个数字的和
function add(
                ){
var result=0;
 arguments.forEach(function(n){
 result+=n;
                     Uncaught TypeError:
 })
                     arguments.forEach is not a function
 return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
因为
```

forEach是数组类型的函数,但arguments不是数组类型的对象,所以无权使用数组类型的函数

forEach:

- 1). 优点: 可以配合ES6的箭头函数,很简化
- 2). 缺点: 只能遍历数字下标的索引数组,无法遍历类数组对象

解决:

- 今后只要遍历数字下标的东西,
- 都可用for of代理普通for循环和forEach

如何:

```
for(var 元素值n of 索引数组/类数组对象){
属性值
//of会依次取出数组或类数组对象中每个属性值
//自动保存of前的变量中
}
```

需求1: 点名

```
var arr=["涛涛","楠楠","东东"];
for(var t of arr){
  console.log(`${t} - 到!`)
}
```

需求2: 定义函数计算任意个数的和

```
//定义一个函数,求任意多个数字的和
function add(
var result=0;
 for(var n of arguments){
  result+=n;
 return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```

for of的问题:

- a. 无法获得下标位置i, 只能获得元素值
- b. 无法控制遍历的顺序或步调,只能从头到尾,一个挨一个的顺序遍历
- c. 无法遍历下标名为自定义下标的对象和关联数组

但是:

- 因为绝大多数数组都是数字下标,
- 绝大多数循环都是从头到尾,一个挨一个遍历的,
- 且绝大多数循环不太关心下标位置,
- 只关心元素值,
- 所以for of将来用的还是非常多的!

比较: for, forEach, for of, for in

		普通for	forEach	for of	for in
数字下标	索引数组	٧	٧	٧	×
	类数组对象	V	×	٧	×
自定义下标	关联数组	×	×	×	٧
	对象	×	×	×	V

总结:

- 下标为数字,首选for of
- 下标为自定义字符串,首选for in

5. 参数增强:

(1). 参数默认值(default):

- 问题:
- 调用函数时,如果不传入实参值
- 虽然语法不报错,但是形参默认接住undefined
- undefined极容易造成程序错误!

解决: 参数默认值

- 今后,只要希望即使不传入实参值时,
- 形参变量也有默认值使用时
- 就为形参指定默认值

```
function 函数名(形参1=默认值1,形参2=默认值2,...){
//调用函数时,给形参传了实参值,
//则首选用户传入的实参值。
//如果没有给形参传是实参值,
// 则形参默认启用=右边的默认值。
}
```

实验: 定义函数——点肯德基套餐

```
function order(
zhushi="香辣鸡腿堡",xiaochi="辣翅",yinliao="可乐"
){ ... }
//第一个人
order("劲脆鸡腿堡","土豆泥","雪碧"); //都换,支持!
//第二个人
order(); //一个都不换, 也支持!
//第三个人: 只想换主食, 其它两个保持默认
order("巨无霸"); //只换开头, 也支持!
```

问题:

- 参数默认值只支持没有实参值和连续更换开头参数值的情况
- 不支持, 跳过开头形参, 只更换中间某个参数的情况。
- 比如:

```
function order(
zhushi="香辣鸡腿堡",xiaochi="辣翅",yinliao="可乐"
){ ... }
```

- //第四个人: 只想换小吃, 其它两个保持默认
- // order(,"鸡米花")//换中间,不支持
- // order(undefined,"鸡米花")//不好

解决: (暂时无法解决, 往后学...)

(2). 剩余参数(rest)

问题: 定义函数计算任意多个数的和

• 用arguments实现

```
//定义一个函数, 求任意多个数字的和
function add(
                   ){
// arguments[
 var result=0;
 for(var n of arguments){
  result+=n;
 return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```

问题: 定义函数计算任意多个数的和

• 改为箭头函数

```
//定义一个函数,求任意多个数字的和
var add= ( )=>{
//arguments[ ]
  var result=0;
  for(var n of arguments){
    result+=n;
  }
  return result;
}
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```

问题:箭头函数不支持arguments

• 如果箭头函数遇到参数个数不确定时, 怎么办?

解决: ES6剩余参数

• 今后,只要在ES6箭头函数中碰到参数个数不确定的情况,都要用剩余参数语法 来代替arguments

```
收集
\ ↓ /
var 函数名=(...数组名arr )=>{
```

//将来传入函数的所有实参值,都会被...收集起来,保存到...后指定的数组中。

}

```
B
```

```
//定义一个函数, 求任意多个数字的和
var add=
                           )=>{
//arguments[
  var result=0;
  for(var n of arguments){
    result+=n;
                      arguments is not defined
  return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```



```
//定义一个函数, 求任意多个数字的和
var add= ( ...arr)=>{
 //arr=[所有实参值]
 var result=0;
 for(var n of arr){
   result+=n;
 return result;
console.log(add(1,2,3));//6
console.log(add(1,2,3,4,5));//15
```

```
B
```

```
console.log(add(1,2,3));//6
var add= (
  //arr= [
 var result=0;
  for(var n of arr){
    result+=n;
  return result;
console.log(add(1,2,3,4,5));//15
```

```
B
```

```
console.log(add(1,2,3));//6
var add= ( ...arr/
 //arr= [1 , 2 , 3 ]
  var result=0;
  for(var n of arr){
    result+=n;
  return result;
console.log(add(1,2,3,4,5));//15
```

```
B
```

```
console.log(add(1,2,3,4,5));//15
var add= (
 //arr= [1 , 2 , 3 , 4 , 5]
  var result=0;
  for(var n of arr){
    result+=n;
  return result;
```

...arr返回的是纯正的数组

• 所以,arr可以使用数组类型的所有好用的函数

```
console.log(add(1,2,3,4,5));//15
var add=
 //arr= [1,2,3,4,5]
 return arr.reduce(
  (捐款箱box,当前参数值n)=>捐款箱box+当前参数值n
```

优点:

- 1). 支持箭头函数
- 2). 生成的数组是纯正的数组类型,所以使用数组家所有函数
- 3). 自定义数组名arr, 比arguments简单的多!

为什么叫rest剩余参数?

- ...可以和其它形参配合使用,
- 只获得其它形参不要的剩余参数。

```
var 函数名=(形参1, 形参2,...数组名arr)=>{
    //形参1接收实参1
    //形参2接收实参2
    //···数组名arr接收除实参值1、2之外其余实参
}
```

调用函数时:

• 调用时:
函数名(实参1, 实参2, 实参3, 实参4,.../
var 函数名=(形参1, 形参2,...数组名arr)=>{
//形参1接收实参1
//形参2接收实参2
//...数组名arr接收除实参值1、2之外其余实参
}

需求

- 定义计算总工资的函数
- 要求输入员工姓名和多项工资
- 每个人的工资组成各不相同



```
// 接收姓名 接收除姓名之外的其余
// ↓ ↓
function jisuan(ename, ...arr){
```

```
接收姓名 接收除姓名之外的其余
function jisuan(ename, ...arr){
 console.log(arr);
 var total=arr.reduce(
   function(捐款箱box,当前实参值n){
     return 捐款箱box+当前实参值n
 console.log(
    `${ename}的总工资是:${total}`);
jisuan("lilei",10000,1000,2000);
jisuan("hmm", 3000, 500, 1000, 2000, 3000)
```



```
jisuan("lilei",10000,1000,2000);
function jisuan(ename, ...arr)
 console.log(arr); //[10000,1000,2000]
 var total=arr.reduce(
   function(捐款箱box,当前实参值n){
     return 捐款箱box+当前实参值n
 console.log(
    `${ename}的总工资是:${total}`);
```



```
jisuan("hmm", 3000, 500, 1000, 2000, 3000);
function jisuan(ename, \...arr) ₹
  console.log(arr); //[10000,1000,2000,2000,3000]
 var total=arr.reduce(
   function(捐款箱box,当前实参值n){
      return 捐款箱box+当前实参值n
  console.log(
    `${ename}的总工资是:${total}`);
```

总结:

- 今后, 只要箭头函数中, 不确定实参值个数时,
- 都可用"...数组名"代替arguments接住所有或剩余实参值。

赠:不能使用箭头函数的地方/箭头函数缺点

- 构造函数不能用
- 对象的方法不能用
- 原型对象方法不能用
- DOM中事件处理函数不能用
- 箭头函数无法用call,apply,bind改变this
- 箭头函数不支持arguments
- 箭头函数没有prototype

(3). 展开运算符(spread)

a. 问题: 获取一个数组中的最大值

• 虽然:

```
Math.max(),可获得多个数中的最大值:
Math.max(1,7,2,5)返回7
```

• 但是,Math.max()支持数组吗?

```
var arr=[1,7,2,5];
```

Math.max(arr) //返回NaN

不支持获取数组中最大值!

a. 问题: 获取一个数组中的最大值

- 解决:先拆散数组,再传参
- 不好的解决: apply var arr=[1,7,2,5];Math.max.apply(替换this的对象, 要拆散的数组)
- 但是,本例中,与替换this无关,只想拆散数组
- 所以: 第一个实参值乱写,写什么都行
 Math.max.apply(?, arr)
 Math

arr

null

• 缺点: 不替换this, 也被迫传一个对象

好的解决: ES6 展开运算符"…"

- 今后, 只要希望单纯拆散数组,
- 都用...展开运算符
- · 只有即替换this, 又要拆散数组时, 才用apply。

函数名(...数组名arr);



- ... 先将数组拆散为多个实参值,
- 再依次分别传给函数的每个形参变量。

a. 所以: 获取一个数组中的最大值

```
var arr=[1,7,2,5];
Math.max(...arr);
```

问题: 矛盾

```
• 前边学...表示收集,
```

- add(1,2,3,4,5)
- 收↓集
- function add(...arr){ ... }
- 这里学...表示拆散
- Math.max(...arr);
- /↓\
- 拆散

总结:...

- 1). 定义函数时,形参列表中的...,表示收集
- 2). 调用函数时,实参列表中的...,反而表示拆散

h. 语法糖: (时髦的简写)

1). 复制一个数组

```
var arr1=[1,2,3];
//复制arr1中所有元素保存到新数组arr2中
var arr2=[...arr1];
```

```
var arr1=[1,2,3];
//复制arr1中所有元素保存到新数组arr2中
var arr2=[...arr1];
```

B

```
var arr2=[ ]//new Array()的简写
```

```
var arr1=[1,2,3];
//复制arr1中所有元素保存到新数组arr2中
var arr2=[...arr1];
         ...arr1
         1,2,3
var arr2=[1 ,2 ,3]//new Array()的简写
```

2). 合并多个数组和元素:

```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
```



```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
```



var arr3=[

]//new Array()的简写

```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
         ...arr1
        1,2,3
```

var arr3=[1, 2, 3]



]//new Array()的简写

```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
```

var arr3=[1,2,3,4,

B

]//new Array()的简写

```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
                  ...arr2
                  5,6,7
var arr3=[1 ,2 ,3,4,5 ,6 ,7, ]//new Array()的简写
```

```
var arr1=[1,2,3];
var arr2=[5,6,7];
//拼接多个数组和多个元素为一个新的数组
var arr3=[...arr1,4,...arr2,8];
```

var arr3=[1 ,2 ,3,4,5 ,6 ,7,8]//new Array()的简写



3). 克隆一个对象

```
var lilei={ sname:"Li Lei", sage:11 };
//克隆lilei
var lilei2={...lilei};
```



```
var lilei={ sname:"Li Lei", sage:11 };
//克隆lilei
var lilei2={ ...lilei };
```



```
var lilei={ sname:"Li Lei", sage:11 };
//克隆lilei
var lilei2={ ...lilei };
```

```
var lilei={ sname:"Li Lei", sage:11 };
//克隆lilei
                   ...lilei
var lilei2={
                    ...lilei
            sname: "Li Lei", sage: 11
var lilei2={sname:"Li Lei", sage:11 }//new Object()
```

4). 合并多个对象和属性:

```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3={...obj1, z:3, ...obj2, k:6};
```



```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3={...obj1, z:3, ...obj2, k:6};
```



```
var obj3={
```

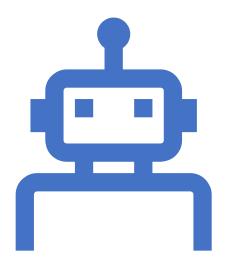
}//new Object()

```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3=\{...obj1, z:3, ...obj2, k:6\};
          ...obj1
          x:1, y:2
var obj3=\{x:1, y:2,
                                      }//new Object()
```

```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3={...obj1, z:3, ...obj2, k:6};
var obj3=\{x:1, y:2,z:3\}
                                      }//new Object()
```

```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3=\{...obj1, z:3, ...obj2, k:6\};
                        ...obj2
                       i:4, j:5
                                 }//new Object()
var obj3=\{x:1, y:2,z:3,i:4, j:5,
```

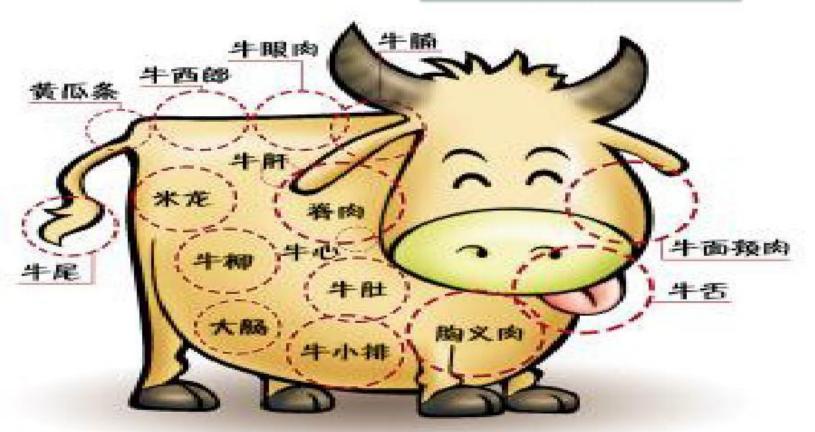
```
var obj1={ x:1, y:2 };
var obj2={ i:4, j:5 };
var obj3={...obj1, z:3, ...obj2, k:6};
var obj3=\{x:1, y:2,z:3,i:4, j:5, k:6\}//new Object()
```



6. 解构(destruct):

庖丁解牛

《庄子·养生主》





问题

• 旧js中,要想使用对象中的成员,或数组中的元素,都必须带着"对象名."或"数组名[]"前缀。

但是

- 实际开发中,对象或数组的嵌套结构可能很深。
- 前缀就可能写很长: "对象名.属性名.更子一级属性名...",
- 非常麻烦。比如:

```
var lilei={
    sname:"Li Lei",
    address:{
    street: "万寿路西街", //lilei.address.street
    city:"北京" //lilei.address.city
    }
}
```

解决:

- 今后, 遇到一个复杂的对象或数组时,
- 都可以通过解构方式,
- 来减少数组或对象的嵌套结构便于使用。

```
var lilei={
    sname:"Li Lei";
    sname:"Li Lei",
    address:{
    street: "万寿路西街",
    city:"北京"
    }
}
```

三种情况:

• 数组解构、对象解构、参数解构

数组解构:

• 从一个复杂的数组中只提取出需要的个别元素单独使用

结果:下标对下标

• var [变量1, 变量2, ...] =数组 //[元素值1, 元素值2]

• 0 1 0 1

- =右边数组中相同下标位置的元素值
- 会自动赋值给=左边相同下标位置的变量
- 变量1=数组[0]
- 变量2=数组[1]

```
var arr=[2021,6,7,9,33];
// 0 1 2 3 4
//仅提取出arr数组中年、月、日三个值单独使用
```

```
console.log(`今年是${ }年`);
console.log(`本月是${ }月`);
console.log(`今天是${ }号`);
```

```
var arr=[2021,6,7,9,33];
// 0 1 2 3 4
//仅提取出arr数组中年、月、日三个值单独使用
var [y,m,d]=arr;
0 1 2
```

```
console.log(`今年是${y}年`);
console.log(`本月是${m}月`);
console.log(`今天是${d}号`);
```

```
var arr=[2021,6,7,9,33];
//仅提取出arr数组中年、月、日三个值单独使用
var [y,m,d]=arr;
console.log(`今年是${y}年`); //2021
console.log(`本月是${m}月`); //6
console.log(`今天是${d}号`); //7
```

```
var arr=[2021, 6, 7, 9, 33];
       0 1 2 3 4
//如果不想要年,只想要月和日
var [ ]=arr;
// 012
console.log(`本月是${m}月`);
console.log(`今天是${d}号`);
```



```
var arr=[2021,6,7,9,33];
     不想要年,只想要月和日
var [m,d ]=arr; //错误
console.log(`本月是${m}月`);
console.log(`今天是${d}号`);
```

```
var arr=[2021,6,7,9,33];
//如果不想要年,只想要月和日
var [ ,m,d]=arr; //正确, 数组中允许空","
console.log(`本月是${m}月`); //6
console.log(`今天是$*{d}号`); //7
```

对象解构:

• 从一个大的对象中只提取出个别属性值单独使用

结果: 属性对属性

```
配对 接值 配对 接值
var { 属性名1:变量1, 属性名2:变量2,... }=对象//{
属性名1:属性值1,
属性名2:属性值2
```

- 变量1=对象.属性名1 //属性值1
- 变量2=对象.属性名2 //属性值2

```
B
```

```
var lilei={
    sname:"Li Lei",
    sage:11
}
//想提取出sname和sage单独使用
```

```
B
```

```
var lilei={
 sname:"Li Lei",
 sage:11
//想提取出sname和sage单独使用
// 属性名 变量名
// 配对儿 接值
var {sname : sname, sage:sage}=lilei;
```

```
B
```

```
var lilei={
  sname:"Li Lei",
  sage:11
//想提取出sname和sage单独使用
// 属性名 变量名
// 配对儿 接值
var {sname : sname, sage:sage}=lilei{sname:"Li Lei"
```

```
var lilei={
  sname:"Li Lei",
  sage:11
//想提取出sname和sage单独使用
// 属性名 变量名
// 配对儿 接值
var {sname : sname, sage:sage}=lilei{sname:"Li_Lei"}
console.log(`姓名:${sname}`) //Li Lei
```

console.log(`年龄:\${sage}`) //11

ES6简写:

- 问题:
 - 要解构的属性在原对象中的原属性名起的很好了
 - 解构时, 我也想用原属性名作为变量名, 单独用
 - 不想改名!
 - 但是, 却要:左右两边重复写两遍相同的属性名
- 每当对象中:左边的属性名和:右边的变量名刚好相同时,其实只写一个名字即可!
- 但是,一个名字两用,既当属性名配对,又当变量名接值

```
var lilei={
 sname:"Li Lei",
 sage:11
//想提取出sname和sage单独使用
// 属性名 变量名
// 配对儿 接值
var {sname : sname, sage:sage}=lilei{sname:"Li Lei"
```

console.log(`姓名:\${sname}`) //Li Lei console.log(`年龄:\${sage}`) //11

```
var lilei={
 sname:"Li Lei",
 sage:11
var {sname : sname, sage:sage}=lilei
var { sname, sage}=lilei;
//一个名字两用
//既当属性名配对儿
//又当变量名接值
```

console.log(`姓名:\${sname}`) //Li Lei console.log(`年龄:\${sage}`) //11

```
var lilei={
 sname:"Li Lei",
 sage:11
var {sname/ : sname, sage:sage}=lilei
var { sname,
            sage}=lilei;
//一个名字两用
//既当属性名配对儿
//又当变量名接值
console.log(`姓名:${sname}`) //Li Lei
console.log(`年龄:${sage}`) //11
```

解构失败:

- 数组解构或者对象解构中,
- 试图获取原数组或对象中不存在的位置或属性值,
- 就会解构失败
- 解构失败不报错! 而是返回undefined给=左边对应的变量

```
var lilei={
 sname:"Li Lei",
 sage:11
var {sname : sname, sage:sage}=lilei
var { sname, sage, className }=lilei;
//一个名字两用
//既当属性名配对儿
//又当变量名接值
console.log(`姓名:${sname}`) //Li Lei
console.log(`年龄:${sage}`) //11
console.log(`班级:${className}`) //undefined
```

参数解构

• 问题: 单靠参数默认值, 无法解决任意一个形参不确定有没有的情况。

//定义一个点套餐的函数: (暂不支持单点)

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐")
console.log(`
您点的套餐是:
主食:${zhushi},
小吃:${xiaochi},
饮料:${yinliao}
`)
}
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log()
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第一个人: 全换
order("劲脆鸡腿堡", "土豆泥", "雪碧");
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log(`"劲脆鸡腿堡"
                           ,"土豆泥"
 您点的套餐是:
   主食:${zhushi}
   小吃:${xiaochi},
   饮料:${yinliao}}
//第一个人: 全换
order("劲脆鸡腿堡", "土豆泥", "雪碧");
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order(<u>zhushi="香辣</u>鸡腿堡",<u>xiaochi="辣翅",yinliao=</u>"可乐"){
                             "土豆泥"
 console.log(`"劲脆鸡腿堡"
 您点的套餐是:
   主食:${zhushi}, //"劲脆鸡腿堡"
   小吃:${xiaochi}, // "土豆泥"
   饮料:${yinliao} // "雪碧"
//第一个人: 全换
order("劲脆鸡腿堡", "土豆泥", "雪碧"); //支持
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log()
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第二个人: 全不换
order();
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log(` undefined
                      undefined undefined
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第二个人: 全不换
```

order();

```
//定义一个点套餐的函数:
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐
 console.log(` undefined
                              undefined
                                          undefined
 您点的套餐是:
   主食:${zhushi}, //"香辣鸡腿堡"
   小吃:${xiaochi}, //"辣翅"
   饮料:${yinliao} //"可乐"
//第二个人: 全不换
```

order(); //支持

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log()
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第三个人: 只想换主食, 其它两个保持默认
order("巨无霸");
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log(`"巨无霸"
                             undefined undefined
 您点的套餐是:
   主食:${zhush;{},
   小吃:${xiaochi},
   饮料:${yin/liao}
//第三个人:/ 只想换主食, 其它两个保持默认
```

order("巨无霸");

```
//定义一个点套餐的函数: (暂不支持单点)
function order(<u>zhushi="香辣</u>鸡腿堡", xiaochi="辣翅"
            "巨无霸"
 console.log()
                              undefined
                                          undefined
 您点的套餐是:
   主食:${zhushi}, // "巨无霸"
   小吃:${xiaochi}, //"辣翅"
   饮料:${yinliao} //"可乐"
//第三个人: 只想换主食, 其它两个保持默认
```

order("巨无霸"); //支持

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log()
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第四个人: 只想换小吃, 其它两个保持默认
order("鸡米花")
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log(` "鸡米花"
                             undefined undefined
 您点的套餐是:
   主食:${zhush;{},
   小吃:${xiaochi},
   饮料:${yin/liao}
//第四个人:/ 只想换小吃, 其它两个保持默认
order("鸡米花")
```

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐")
console.log(` undefined undefined when the console.log(` undefined undefined undefined undefined when the console.log(` undefined undefined undefined undefined when the console.log(` undefined undefined undefined undefined undefined undefined undefined when the console.log(` undefined un
```

//第四个人: 只想换小吃, 其它两个保持默认

order("鸡米花")//传错位了,不是用户想要的

```
//定义一个点套餐的函数: (暂不支持单点)
function order( zhushi="香辣鸡腿堡", xiaochi="辣翅", yinliao="可乐"){
 console.log()
 您点的套餐是:
   主食:${zhushi},
   小吃:${xiaochi},
   饮料:${yinliao}
//第四个人: 只想换小吃, 其它两个保持默认
order( , "鸡米花") / / 语法错误! 不支持
```

解决

- 今后只要发现任意一个实参值都可能没有,
- 但是又要求实参值必须传给指定的形参,
- 顺序不能乱,
- 都可用参数解构:

如何: 2步

```
• i. 定义函数时:
function 函数名({
//配对儿:接实参值
 属性名1: 形参1,
 属性名2:形参2,
  ... .
}){
函数体
```

如何: 2步

```
ii. 调用函数时:
函数名({
//配对儿
属性名1: 实参值1,
属性名2: 实参值2,
...:...
```

结果:

```
function 函数名({
//配对儿:接实参值
属性名1: 形参1, 属性名2: 形参2, …: …
}){
函数体
函数名({
//配对儿
属性名1: 实参值1, 属性名2: 实参值2, ...:....
})
```

问题:

- 属性名与形参变量名通常不会起两个名字,绝大多数情况下同名。解决: ES6简写
- 定义函数时:
 function 函数名({
 //一个名字两用
 //既配对,
 //又接值
 属性名1,属性名2,xxx
 }){
 函数体

解决: ES6简写

```
function 函数名({
 //一个名字两用
 //既配对
 //又接值
 属性名1,属性名2,xxx
}){
 函数体
• ii. 调用时保持不变
函数名({
//配对儿
属性名1: 实参值1, 属性名2: 实参值2, ...:....
```

问题: 担心形参可能接不到实参值

```
• 解决: ES6简写+参数默认值
• i. 定义函数时:
function 函数名({
 //一个名字两用
 //既配对,
 //又接值
 属性名1=默认值1,属性名2=默认值2, xxx=xxx
}){
 函数体
```

解决: ES6简写+参数默认值

```
function 函数名({
 //一个名字两用
 //既配对,
 //又接值
 属性名1=默认值1,属性名2=默认值2, xxx=xxx
}){
 函数体
ii. 调用时保持不变
函数名({
 //配对儿
 属性名1: 实参值1, 属性名2: 实参值2, ...:...
```

本质:

- 因为定义函数时,
- 形参是对象结构,
- 调用函数时,
- 实参值也是对象结构,
- 所以,参数结构的本质,其实
- 还是对象结构——属性对属性

```
order({
function order({
 zhushi="香辣鸡腿堡"
 xiaochi="薯条",
 yinliao="可乐"
                          });
 console.log(`
  你点的套餐是:
   主食:zhushi="香辣鸡腿堡",
   小吃:xiaochi="薯条",
   饮料:yinliao="可乐"
```

```
zhushi:"奥尔良烤腿堡
xiaochi:"扭扭薯条",
yinliao:"雪碧"
```

```
order({
function order({
});
console.log(`
 你点的套餐是:
 主食:"奥尔良烤腿堡"
 小吃:"扭扭薯条",
 饮料:"雪碧"
```

```
order({
function order({
 zhushi="香辣鸡腿堡"
                             undefined
 xiaochi="薯条",
                             undefined
 yinliao="可乐"
                            undefined
}){
                           });
 console.log(`
  你点的套餐是:
   主食: zhushi="香辣鸡腿堡",
   小吃:xiaochi="薯条",
   饮料:yinliao="可乐"
```

```
order({
function order({
}){
           });
console.log(`
 你点的套餐是:
 主食:"香辣鸡腿堡"
 小吃:"薯条"
 饮料:"可乐"
```

```
order({
function order({
 zhushi="香辣鸡腿堡"
                             zhushi: "巨无霸"
 xiaochi="薯条",
                             undefined
 yinliao="可乐"
                            undefined
}){
                           });
 console.log(`
  你点的套餐是:
   主食: zhushi="香辣鸡腿堡",
   小吃: xiaochi="薯条",
   饮料: yinliao="可乐"
```

```
order({
function order({
}){
console.log(`
           });
 你点的套餐是:
 主食:"巨无霸"
 小吃:"薯条"
 饮料:"可乐"
```

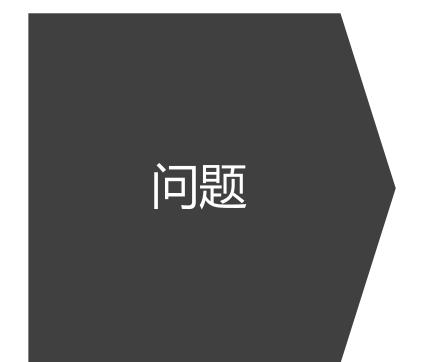
```
function order({
                           order({
 zhushi="香辣鸡腿堡"
                             undefined
 xiaochi="薯条",
                             xiaochi:"红豆派"
 yinliao="可乐"
                             undefined
}){
                           });
 console.log(`
  你点的套餐是:
   主食: zhushi="香辣鸡腿堡",
   小吃: xiaochi="薯条",
   饮料: yinliao="可乐"
```

```
function order({
              order({
yinliao="可乐" undefined
}){
              });
 console.log(`
 你点的套餐是:
 主食: "香辣鸡腿堡"
 小吃:"红豆派"
 饮料:"可乐"
```



Q2: Class & 继承





- 旧js中,构造函数和原型对象是分开定义的。
- 不符合"封装"概念

```
//想定义学生类型,描述所有学生的统一结构和功能
function Student(sname, sage){
 this.sname=sname;
 this.sage=sage;
Student.prototype.intr=function(){
 console.log(
   `I'm ${this.sname}, I'm ${this.sage}`);
//创建一个学生类型的对象
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```

解决:

- 今后,只要在es6中创建一种新的类型,
- 包含构造函数+原型对象方法,
- 都要用class来创建

什么是class

- 程序中专门集中保存
- 一种类型的所有子对象
- 的统一属性结构和方法定义
- 的程序结构。

如何定义 class: 3句话:

- i. 用class{}包裹原构造函数+原型对象方法
- ii. 原构造函数名升级为整个class的名字,所有构造函数统一更名为"constructor"
- iii.原型对象中的方法,不用再加prototype前缀,也不用 =function,直接简写为: **方法名(){}**

直接放在class{}内的方法定义,其实还是保存在原型对象中的。

```
//想定义学生类型,描述所有学生的统一结构和功能
function Student(sname, sage){
  this.sname=sname;
 this.sage=sage;
Student.prototype.intr=function(){
  console.log(
   `I'm ${this.sname}, I'm ${this.sage}`);
//创建一个学生类型的对象
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```

```
//想定义学生类型,描述所有学生的统一结构和功能
class Student{
 constructor(sname, sage){
  this.sname=sname;
  this.sage=sage;
        直接放在class{}内的方法定义,其实还是保存在原型对象中的。
  console.log(
   `I'm ${this.sname}, I'm ${this.sage}`);
//创建一个学生类型的对象
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```

如何使用 class:

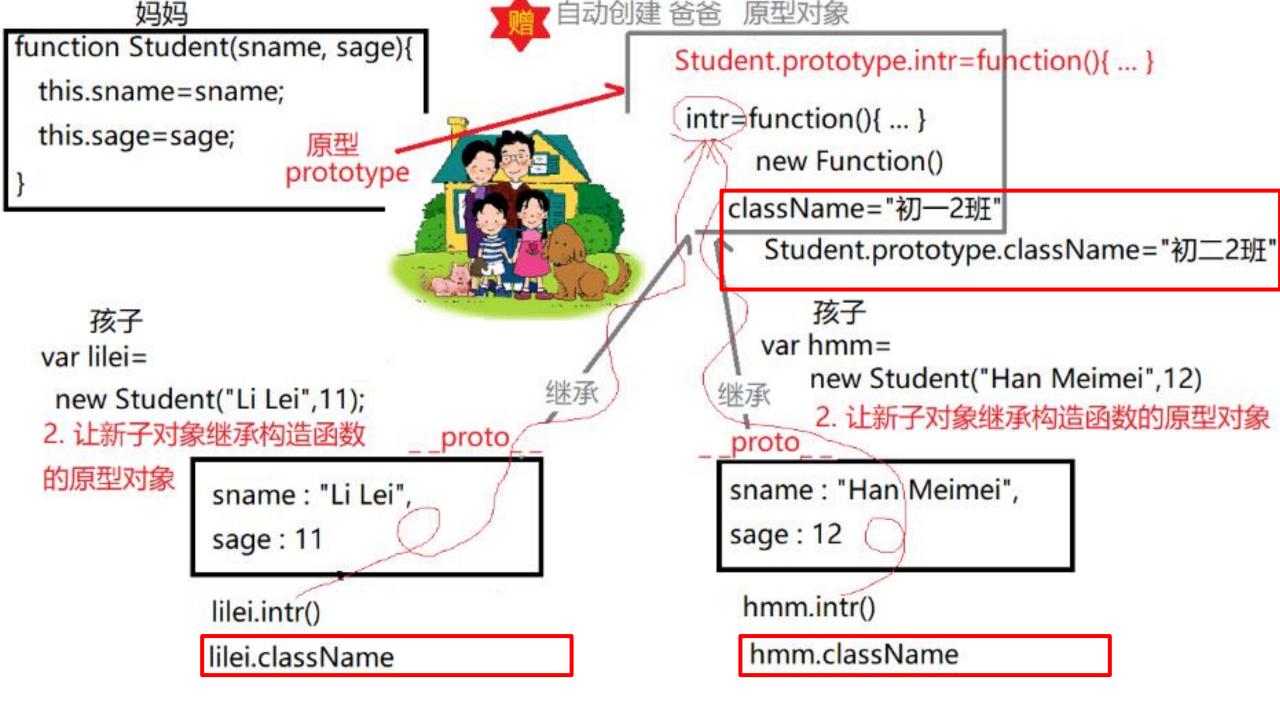
- 和使用旧的构造函数完全一样:
- var 对象名=**new class名(**属性值,...**)**;

本质:新瓶装旧酒,换汤不换药:

```
Student {sname: "Li Lei", sage
 sage: 11
 sname: "Li Lei"
 [[Prototype]]: Object
                                父对象原型对象
  ▶ constructor: class Student
  ▶ intr: f intr()
  ▶ [[Prototype]]: Object
```

问题:

- 如果多个子对象共用的相同的属性值,应该放在那里?
- 旧js中: 是和共有方法一起放在原型对象中

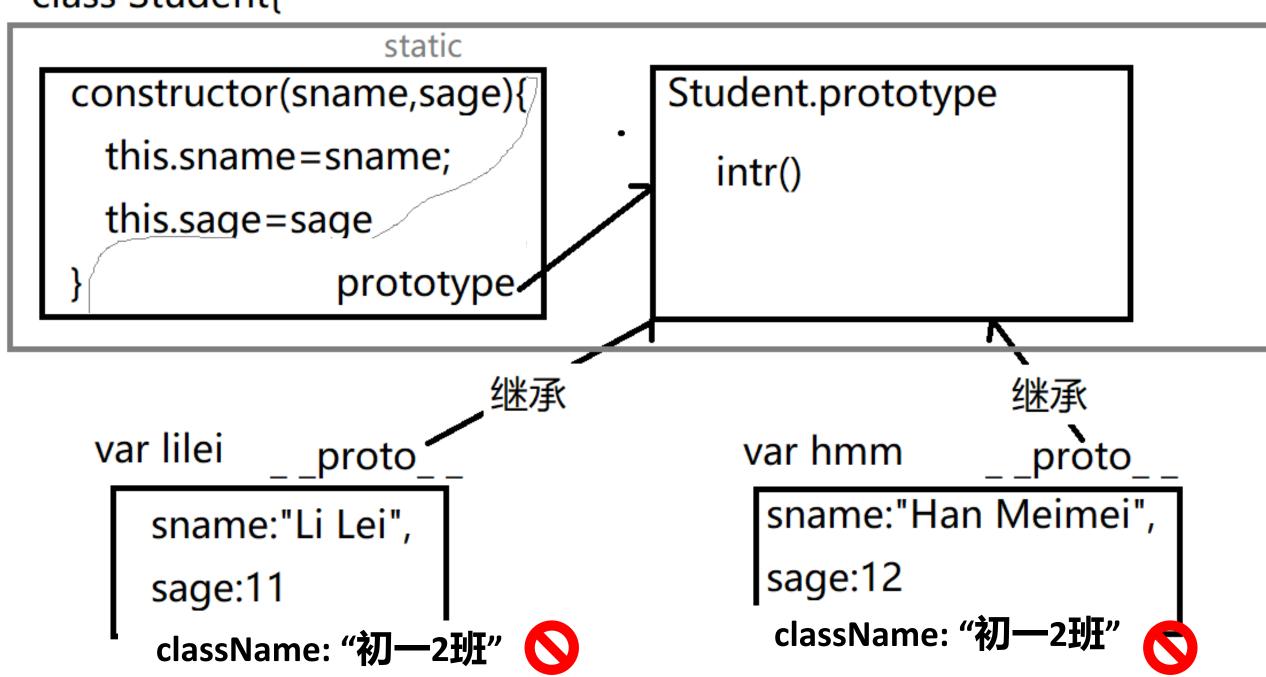


ES6 class中 问题:

- 虽然直接在class中定义的方法,
- 都默认保存在原型对象中。
- 但是直接在class中定义的属性,
- 却不会成为共有属性,不会保存在原型对象中。
- 而是成为每个子对象的自有属性。

```
class Student{
 constructor(sname, sage){
   this.sname=sname;
   this.sage=sage;
 className="初一2班"
 intr(){//自动放到原型对象
   console.log(
     `I'm ${this.sname}, I'm ${this.sage});
var lilei=new Student("Li Lei",11);
```

class Student{



解决:

- 为了和其它主流开发语言尽量一致,
- ES6的class放弃了在原型对象中保存共有属性的方式。
- 而是改为用静态属性保存!
- 什么是静态属性:
- 不需要创建子对象,
- 单靠类型名就可直接访问的属性,
- 就称为静态属性
- 今后,只要如果希望
- 所有子对象,都可使用一个共同的属性值时,
- 都要用静态属性代替原来的原型对象属性

如何定义静态属性

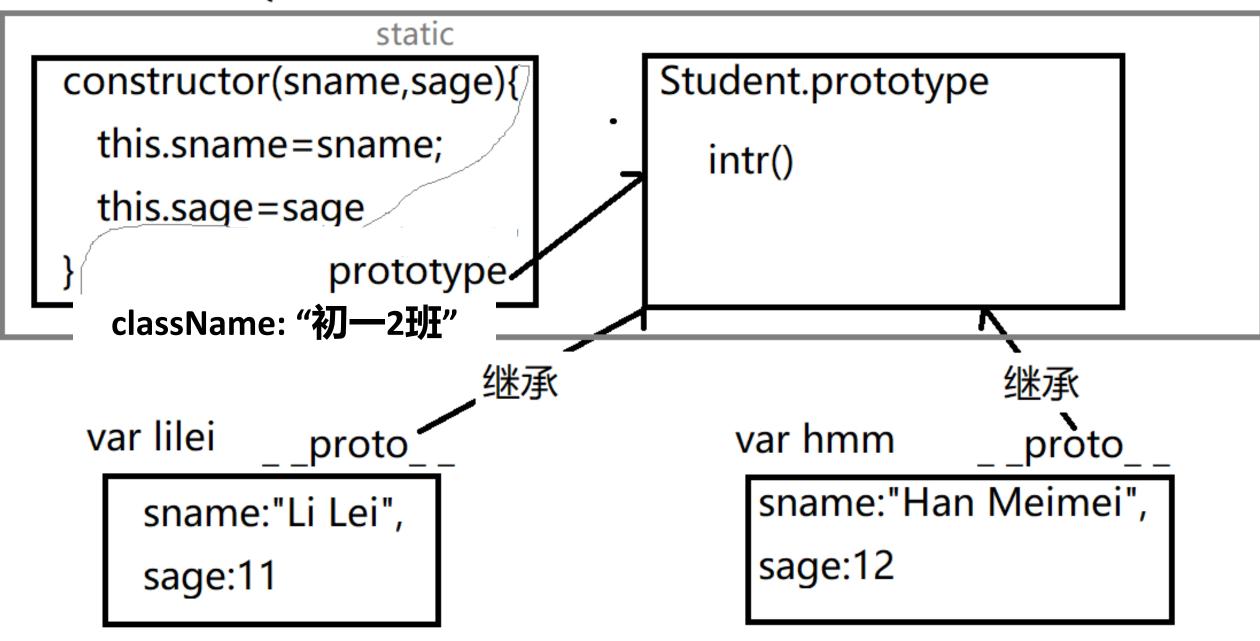
```
class 类型名{
static 共有属性名=属性值
...
...
```

原理:

- 标有static的静态属性,
- 都是保存在构造函数对象身上。
- 因为构造函数在程序中不会重复!
- 所以, 静态属性, 也不会重复!
- 任何时候,任何地点,访问一个类型的静态属性,永远访问的都是同一份!

```
class Student{
 constructor(sname, sage){
   this.sname=sname;
   this.sage=sage;
 static className="初一2班"
 intr(){//自动放到原型对象
   console.log(
     `I'm ${this.sname}, I'm ${this.sage});
var lilei=new Student("Li Lei",11);
```

class Student{

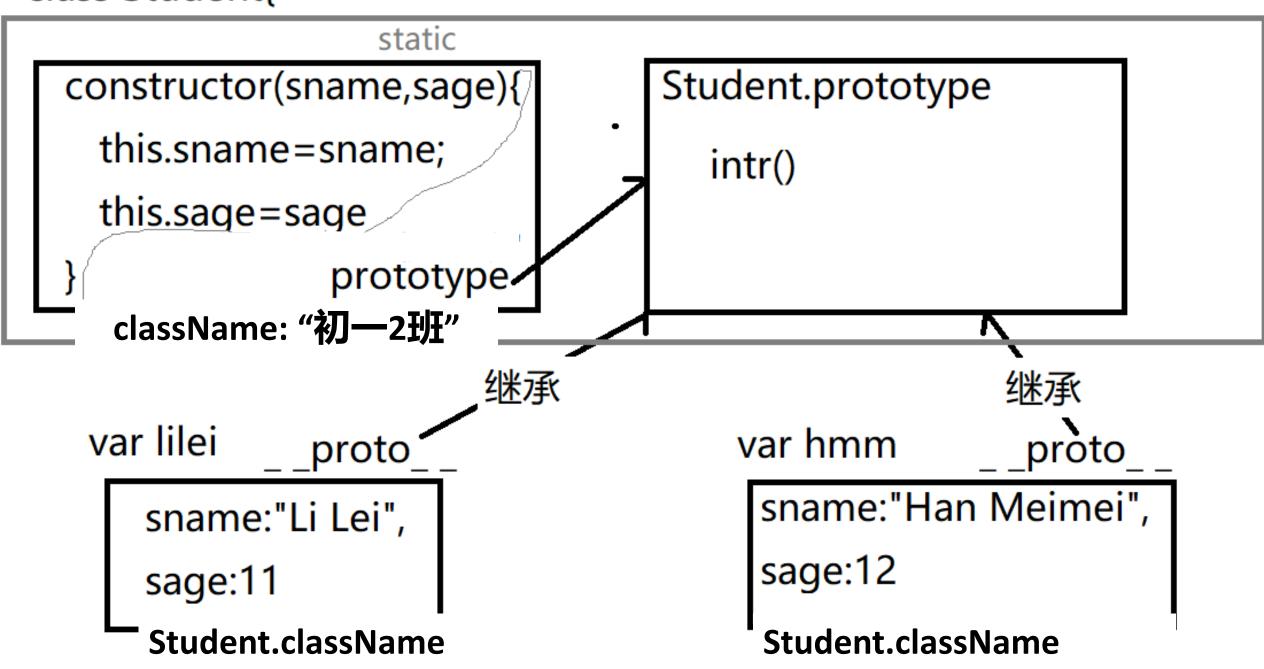


如何访问静态属性值

<u>◆ 坑: 错误: this.静态属性</u>

・正确: 类型名.静态属性

class Student{



```
class Student{
  constructor(sname, sage){
    this.sname=sname;
   this.sage=sage;
  static className="初一2班"
  intr(){//自动放到原型对象
    console.log(
     `I'm ${this.sname},I'm ${this.sage},I'm from ${Student.className})
var lilei=new Student("Li Lei",11);
var hmm=new Student("Han Meimei",12);
//过了一年,两个小朋友都生了一级
Student.className="初二2班"
lilei.intr(); //初二2班
Hmm.intr(); //初二2班
```

static className到底存哪儿了?

console.dir(Student); ▼ class Student i className: "初二2班" arguments: (...) caller: (...) length: 2 name: "Student" ▶ prototype: {constructor: f, intr: f} [[FunctionLocation]]: 2 class.html:15 ▶ [[Prototype]]: *f* () ▶ [[Scopes]]: Scopes[3]

class Student到底是什么?

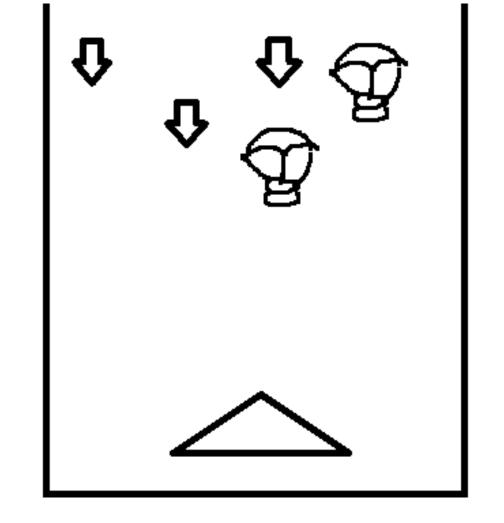
- console.log(typeof Student);
- //"function"
- ·说明class只是个外壳
- 其本质还是普通的构造函数function

```
▼ class Student i
className: "初二2班"
arguments: (...)
caller: (...)
length: 2
name: "Student"

▶ prototype: {constructor: f, intr: f}
[[FunctionLocation]]: 2 class.html:15
▶ [[Prototype]]: f ()
▶ [[Scopes]]: Scopes[3]
```

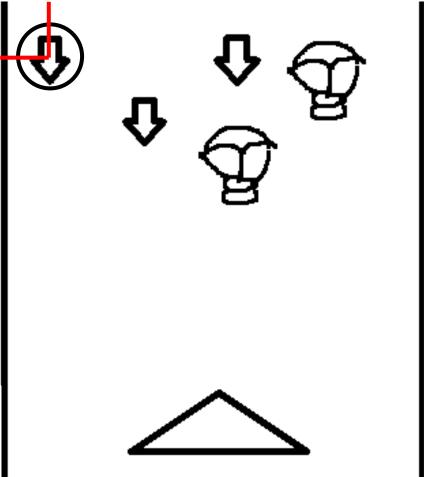
(9). 两种类型间的继承:

+



想做简易的飞机大战游戏

```
class Plane{
 constructor(x,y,score){
   this.x=x;
   this.y=y;
   this.score=score;
 fly(){
   console.log(`飞到xxx位置`)
 getScore(){
   console.log(`击落敌机得xx分`)
```



```
class Plane{
  constructor(x,y,score){
   this.x=x;
   this.y=y;
   this.score=score;
 fly(){
   console.log(`飞到xxx位置`)
 getScore(){
   console.log(`击落敌机得xx分`)
var p1=new Plane(50,100,5);
console.log(p1);
p1.fly();
p1.getScore();
```

```
class Plane{
  constructor(x,y,score){
   this.x=x;
   this.y=y;
   this.score=score;
  //Plane的原型对象
  fly(){
   console.log(`飞到xxx位置`)
  getScore(){
   console.log(`击落敌机得xx分`
                        继承
var p1=new Plane(50,100,5);
                      proto •
x:50
 y:100
 score:5
p1.fly();
p1.getScore();
```

```
class San{
class Plane{
  constructor(x,y,score){
                                                                 constructor(x,y,award){
   this.x=x;
                                                                   this.x=x;
                                                                   this.y=y;
   this.y=y;
                                                                   this.award=award;
   this.score=score;
  '/Plane的原型对象
                                                                 fly(){
                                                                   console.log(`飞到xxx位置`)
  fly(){
   console.log(`飞到xxx位置`)
                                                                 getAward(){
                                                                   console.log(`击落降落伞得xx`)
  getScore(){
   console.log(`击落敌机得xx分`
                        继承
var p1=new Plane(50,100,5);
                      proto •
x:50
 y:100
 score:5
p1.fly();
p1.getScore();
```

```
class San{
class Plane{
  constructor(x,y,score){
                                                                  constructor(x,y,award){
    this.x=x;
                                                                    this.x=x;
    this.y=y;
                                                                    this.y=y;
    this.score=score;
                                                                    this.award=award;
   '/Plane的原型对象
                                                                  fly(){
                                                                    console.log(`飞到xxx位置`)
  fly(){
    console.log(`飞到xxx位置`)
                                                                  getAward(){
                                                                    console.log(`击落降落伞得xx`)
  getScore(){
    console.log(`击落敌机得xx分`
                                                                var s1=new San(20,80,"1 life");
                                                                console.log(s1);
                         继承
                                                                |s1.fly();
var p1=new Plane(50,100,5);
                                                                s1.getAward();
                      proto •
 x:50
 y:100
 score:5
p1.fly();
p1.getScore();
```

```
class San{
class Plane{
  constructor(x,y,score){
                                                                 constructor(x,y,award){
   this.x=x;
                                                                   this.x=x;
   this.y=y;
                                                                   this.y=y;
   this.score=score;
                                                                   this.award=award;
  '/Plane的原型对象
                                                                  '/San的原型对象
  fly(){
                                                                 fly(){
    console.log(`飞到xxx位置`)
                                                                   console.log(`飞到xxx位置`)
  getScore(){
                                                                 getAward(){
    console.log(`击落敌机得xx分`
                                                                   console.log(`击落降落伞得xx`)
                        继承
                                                                                          继承
                                                                var s1=new San(20,80,"1 life")
var p1=new Plane(50,100,5);
                                                                                       proto
                      proto •
                                                                  x:20
 x:50
                                                                  y:80
 y:100
                                                                  award:"1 life"
 score:5
p1.fly();
                                                                s1.fly();
                                                                s1.getAward();
p1.getScore();
```

```
class San{
class Plane{
                                                               constructor(x,y,award){
  constructor(x,y,score){
   this.x=x;
                                                                 this.x=x;
                                                                 this.y=y;
   this.y=y;
   this.score=score;
                                                                 this.award=award;
                        问题: 两种class之间包含部分相同的属性结构和方法定义
  //Plane的原型对象
                                                                //San的原型对象
                                                               fly(){
 fly(){
   console.log(`飞到xxx位置`
                                                                 console.log(`飞到xxx位置`
 getScore(){
                                                               getAward(){
   console.log(`击落敌机得xx分`)
                                                                 console.log(`击落降落伞得xx`)
                        继承
                                                                                       继承
                                                              var s1=new San(20,80,"1 life")
var p1=new Plane(50,100,5);
                                                                                     proto
                      proto •
                                                                x:20
x:50
                                                                y:80
 y:100
                                                                award:"1 life"
 score:5
                                                               s1.fly();
p1.fly();
                                                               s1.getAward();
p1.getScore();
```

b. 解决: 两种class类型间继承: 2大步

1). 额外创建一个父级class, 2件事:

- i. 父级class的构造函数中包含子类型class中相同部分的属性结构定义
- ii. 父级class的原型对象中包含子类型class中相同部分的方法定义
- iii. 既然父级class中保存了相同的属性结构和方法定义,则子类型class中,就可以删除所有重复的属性结构和方法定义

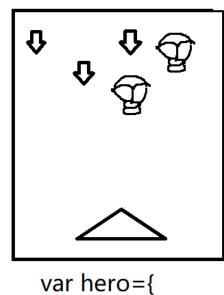
```
class San{
class Plane{
  constructor(x,y,score){
                                                                 constructor(x,y,award){
   this.x=x;
                                                                   this.x=x;
   this.y=y;
                                                                   this.y=y;
   this.score=score;
                                                                   this.award=award;
  //Plane的原型对象
                                                                  /San的原型对象
                                                                 fly(){
 fly(){
   console.log(`飞到xxx位置`
                                                                   console.log(`飞到xxx位置`
  getScore(){
                                                                 getAward(){
   console.log(`击落敌机得xx分`)
                                                                   console.log(`击落降落伞得xx`)
                        继承
                                                                                          继承
                                                                var s1=new San(20,80,"1 life")
var p1=new Plane(50,100,5);
                                                                                       proto
                      proto •
                                                                  x:20
x:50
                                                                  y:80
 y:100
                                                                  award:"1 life"
 score:5
                                                                s1.fly();
p1.fly();
                                                                s1.getAward();
p1.getScore();
```

子类型class的子对象缺少必要的属性,而且有些共有方法也无法使用了!

```
class Plane {
    constructor(x,y,score){

    this.score=score;
}
//plane的原型对象

getScore(){
    击落敌机获得this.score分
}
```



```
class San {
  constructor(x,y,award){

  this.award=award;
}

//San的原型对象

getAward(){
  击落降落伞得this.award奖励
```



2). 让子类型class继承父类型的class, 2步:

- i. 设置子类型的原型对象继承父类型的原型对象:
- class 子类型 extends 父类型{ ... }

```
class Enemy {
                                     constructor(x,y){ ;
                                       this.x=x;
                                        this.y=y;
                                     //父类型class的原型对象
只是设置子类型的原型对象继承父类型的原型对象,
只能保证孙子对象可以使用爷爷类型原型对象中的共有方法! fly(){飞到this.x, this.y位置}
暂时无法为孙子对象补全缺少的自有属性.
                                      继承
          class Plane extends Enemy
                                                      class San
                                      仚
               constructor(x,y,score){
                                                       constructor(x,y,award){
                                                         this.award=award;
                this.score=score;
                                                        //San的原型对象
               //plane的原型对象
                                                        getAward(){
               getScore(){
                                                         击落降落伞得this.award奖励
                击落敌机获得this.score分
                                        var hero={
```

• 原理:

```
class Enemy {
 constructor(x,y){ ;
   this.x=x;
   this.y=y;
                            像
 //父类型class的原型对象
                            Object.setPrototypeOf(
 子类型class的原型对象,
  继承
                              父类型class的原型对象
                 class San extends Enemy
  む
                 constructor(x,y,award){
                   this.award=award;
                  //San的原型对象
                  getAward(){
                   击落降落伞得this.award奖励
   var hero={
```

```
像
```

Object.setPrototypeOf(

子类型class的原型对象,

父类型class的原型对象

```
class Plane extends Enemy {
    constructor(x,y,score){

    this.score=score;
    }
    //plane的原型对象

    getScore(){
    击落敌机获得this.score分
```

```
继承
                                                                   继承
                                            var s1=new San(20,80,"1 life")
var p1=new Plane(50,100,5);
                   _proto
                                                                proto |
                            x:50
                                      x:20
                            y:100
                                      y:80
                                              award:"1 life"
score:5
                                            s1.fly(); //来自爷爷, 可用V
p1.fly(); //来自爷爷, 可用V
                                            s1.getAward(); //来自爸爸, 也可用V
p1.getScore(); //来自爸爸, 也可用V
```

问题: 孙子对象依然缺少x,y属性

- ii. 调用爷爷类型的构造函数。
- 使用super关键字
- super是专门指向父类型的关键字
- 调用super()等于调用爷爷类型的构造函数。
- 调用爷爷类型的构造函数,等效于执行爷爷类型的构造函数中共有的 "this.xx=xxx"语句。
- 可为孙子对象弥补缺少的共有属性结构。

```
class Enemy {
                            constructor(x,y){
                                this.x=x;
                                this.y=y;
                             //父类型class的原型对象
      像
                             fly(){ 飞到this.x, this.y位置 }
                                                              像
      Object.setPrototypeOf(
                                                              Object.setPrototypeOf(
       子类型class的原型对象,
                                                               子类型class的原型对象,
                              继承
                                                   继承
       父类型class的原型对象
                                                                父类型class的原型对象
           extends Enemy
class Plane
                                                class San extends Enemy
                               仚
    constructor(x,y,score){
                                                 constructor(x,y,award){
                                                   super(x,y)
       super(x, y)
                                                   this.award=award;
      this.score=score;
                                                  //San的原型对象
     //plane的原型对象
                                                 getAward(){
    getScore(){
                                                   击落降落伞得this.award奖励
      击落敌机获得this.score分
                                var hero={
```

s1.getAward(); //来自爸爸, 也可用V

```
• 结果:
```

x:50 //来自奶奶

y:100 //来自奶奶

score:5 //来自妈妈

var p1=new Plane(50,100,5);

p1.fly(); //来自爷爷, 可用V

p1.getScore(); //来自爸爸, 也可用V

- 1). 孙子对象可使用3处保存的方法: 自己的, 父级class的, 爷爷class的

继承

proto 🕨

- 2). 孙子对象中拥有两处规定的属性: 父类型构造函数+爷爷类型构造函数。

问题1: extends?

- inherit是继承的意思
- 但是,为什么用extends表示继承
- 答: 继承是为了更好的扩展
- 程序中的继承,都是为了在继承现有成员的基础上,进一步扩展出自己个性化的新成员!
 - 所以,程序中的继承,都用extends(扩展)

问题2: super?

- 为什么指向父类型构造函数的关键字, 称为super?
- 答: super在数学中指超集
- 所有的猫,是一个集合
- 所有的老虎, 也是一个集合
- 但是,我们还可以用一个更大的集合"猫科动物",来包含猫集合和老虎集合。
- · 这个可以包含猫集合和老虎集合的更大的集合"猫科动物",就称为超集,英文为 super



程序中:

- 敌机类型Plane是一个集合
- 降落伞类型San也是一个集合
- · 但是,可以用更大的一个集合敌人Enemy,来概括所有的敌机和降落伞。
- 所以,敌人Enemy集合,就称为敌机类型和降落伞类型的超集super,程序中, 也称为超类
- 所以: 程序中用super指父类型构造函数



Q3: Promise

a. 问题:

- 实际开发中, 经常需要让多个异步任务顺序执行
- 错误的解决: 单纯先后调用多个异步函数
- ran()
- tao()
- dong()

原因

- 异步函数几乎同时执行
- 各自执行各自的,
- 互不干扰,
- 互相之间也不会等待。

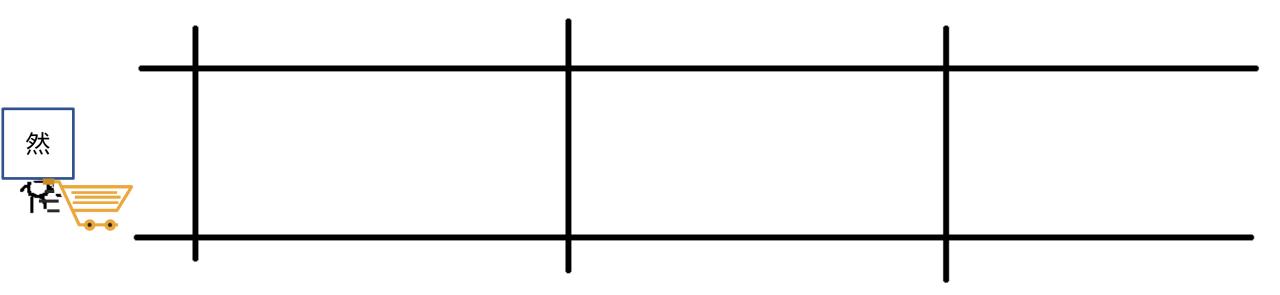
```
function ran(){
                                                console.log(`然起跑....`);
然
                                                setTimeout(function(){//异步
                                                  console.log(`然到达终点...`);
                                                },6000)
                                              function tao(){
                                                console.log(`涛起跑...`);
涛
                                                setTimeout(function(){//异步
                                                  console.log(`涛到达终点...`);
                                                },4000)
                                              function dong(){
                                                console.log(`东起跑...`);
```

```
function dong(){
   console.log(`东起跑...`);
   setTimeout(function(){//异步
    console.log(`东到达终点...`);
   },2000)
}
```

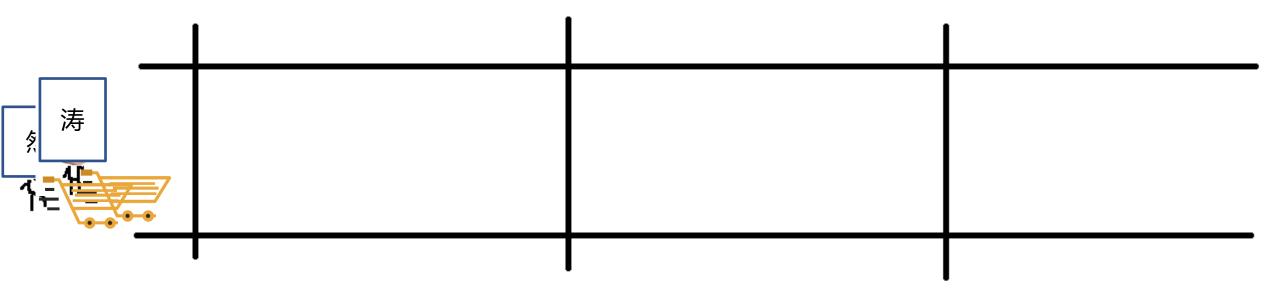


d. 不好的解决: 利用回调函数

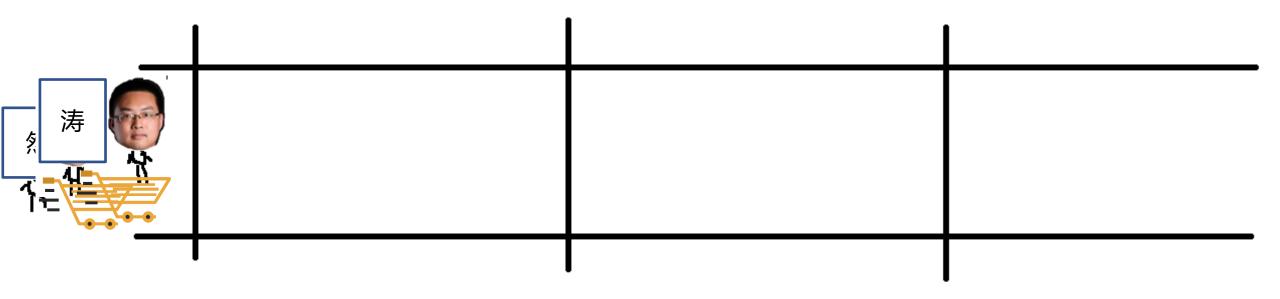




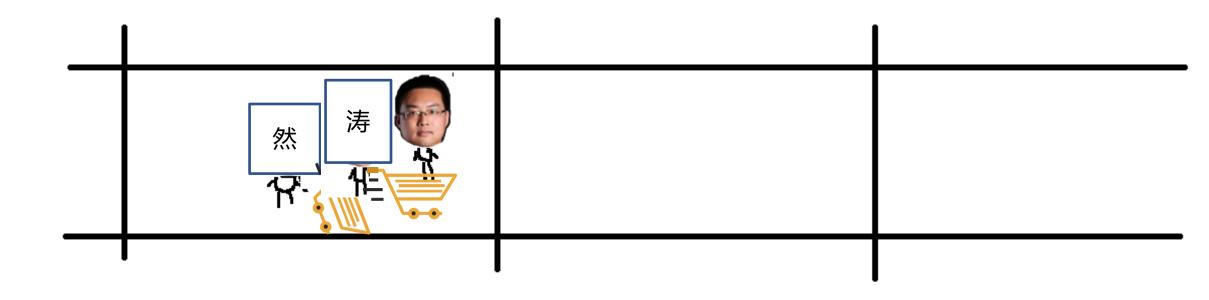




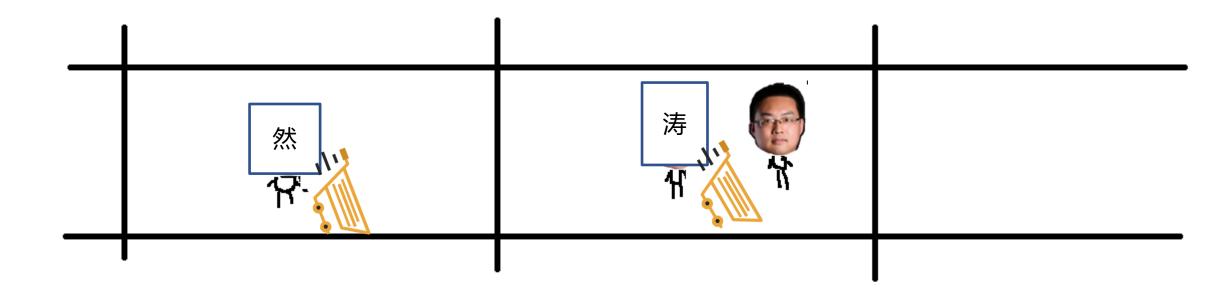












程序中:



i. 前一项任务:

下一项任务的函数

function 前一项任务的函数(购物车che){ 异步任务 异步任务最后一句话之后 购物车che()

ii. 调用前一项任务时:

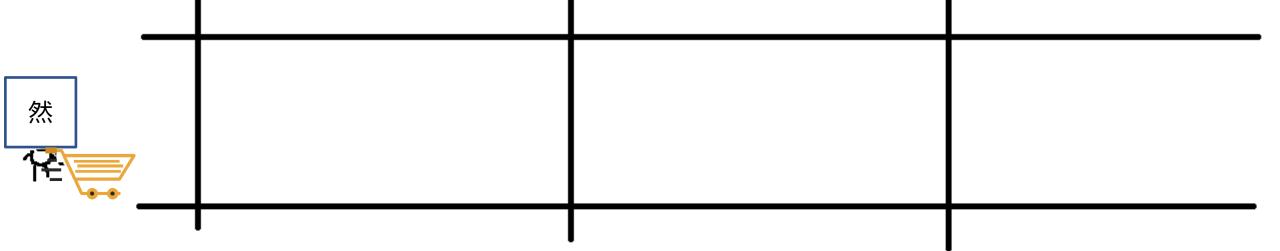
```
前一项任务的函数(
function(){ 下一项任务() }
)
```

```
function 前一项任务的函数(购物车che){
 异步任务
 异步任务最后
 购物车che()
前一项任务的函数(
function(){ 下一项任务() }
```

```
function ran(车){

console.log(`然起跑...`);
setTimeout(function(){//异步
console.log(`然到达终点...`);
},6000)
}
```





```
function ran(车){
  console.log(`然起跑...`);
  setTimeout(function(){//异步
    console.log(`然到达终点...`);
  },6000)
ran(
 function(){ tao(function(){ dong() }) }
);
   涛
```

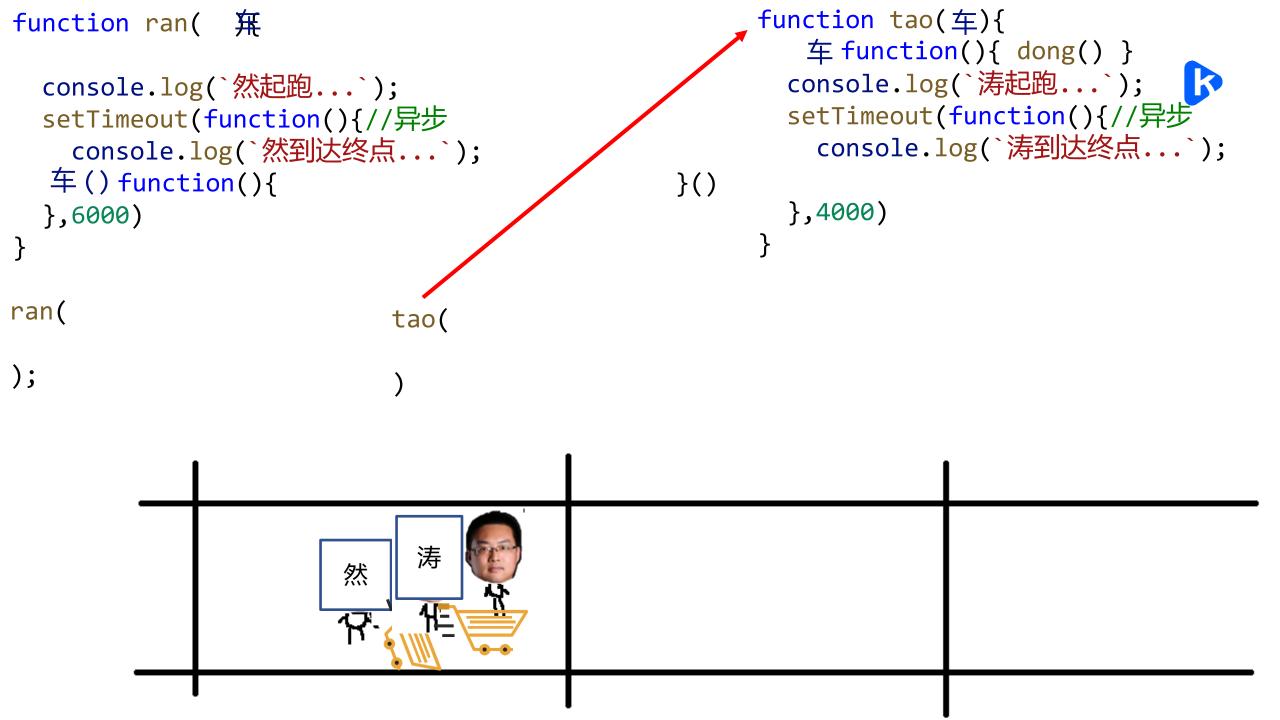
```
function ran(车){
  车 function(){ tao(function(){ dong() }) }
  console.log(`然起跑...`);
  setTimeout(function(){//异步
   console.log(`然到达终点...`);
  },6000)
ran(
   涛
```

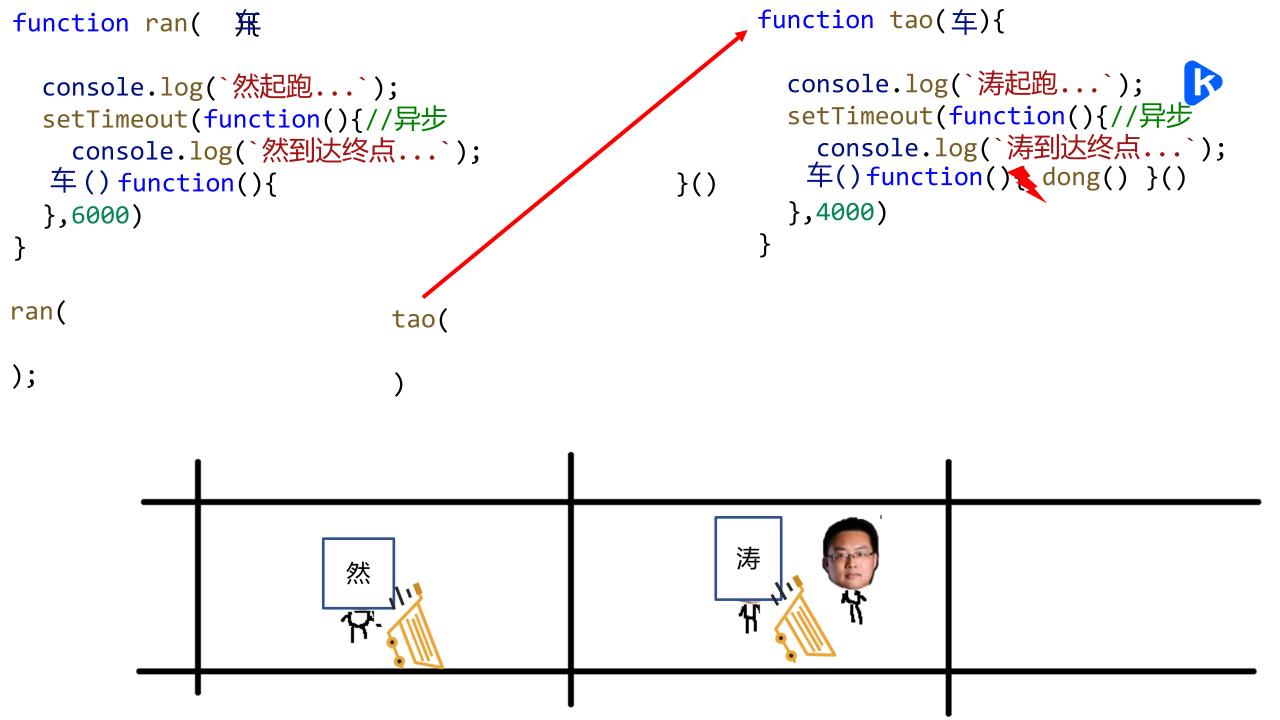
```
console.log(`然起跑...`);
  setTimeout(function(){//异步
   console.log(`然到达终点...`);
  车()function(){ tao(function(){ dong() })}()
  },6000)
ran(
                          涛
                     然
```

function ran(無

```
function tao(车){
console.log(`涛起跑...`);
 console.log(`然起跑...`);
                                               setTimeout(function(){//异步
  setTimeout(function(){//异步
                                                 console.log(`涛到达终点...`);
   console.log(`然到达终点...`);
  车()function(){
                                        }()
                                               },4000)
  },6000)
ran(
                       tao(
                         function(){ dong() }
                         涛
                    然
```

```
function tao(车){
console.log(`涛起跑...`);
 console.log(`然起跑...`);
                                               setTimeout(function(){//异步
  setTimeout(function(){//异步
                                                 console.log(`涛到达终点...`);
   console.log(`然到达终点...`);
  车()function(){
                                        }()
                                               },4000)
  },6000)
ran(
                       tao(
                        function(){ dong() }
                         涛
                    然
```





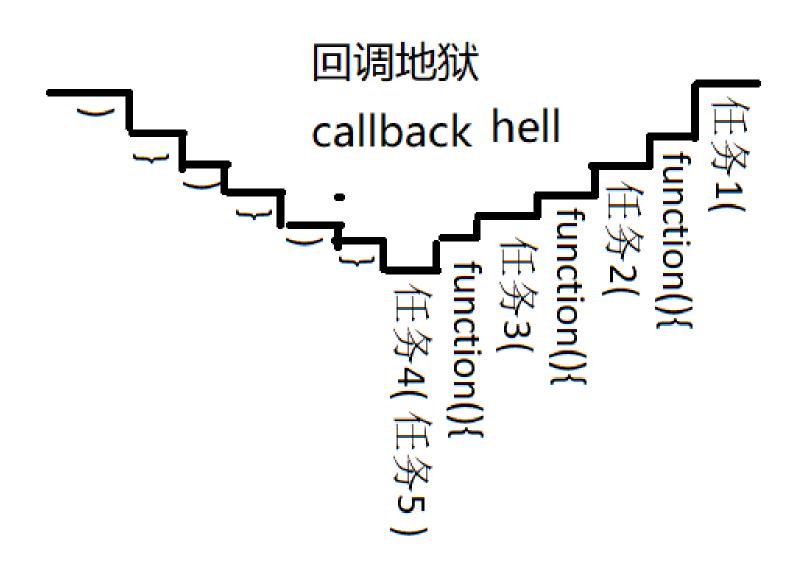
```
function tao(车){
console.log(`涛起跑...`);
 console.log(`然起跑...`);
                                               setTimeout(function(){//异步
  setTimeout(function(){//异步
                                                 console.log(`涛到达终点...`);
   console.log(`然到达终点...`);
                                                车()function(){ dong() }()
  车()function(){
                                        }()
                                               },4000)
  },6000)
ran(
                       tao(
                                            涛
```

2). 问题:

- 如果要先后执行的任务多了!
- 就会形成很深的嵌套结构
- ——回调地狱。
- 极其不优雅,极其不便于维护

```
任务1(
  function(){
    任务2(
     function(){
      任务3(
        function(){
         任务4(任务5)
```

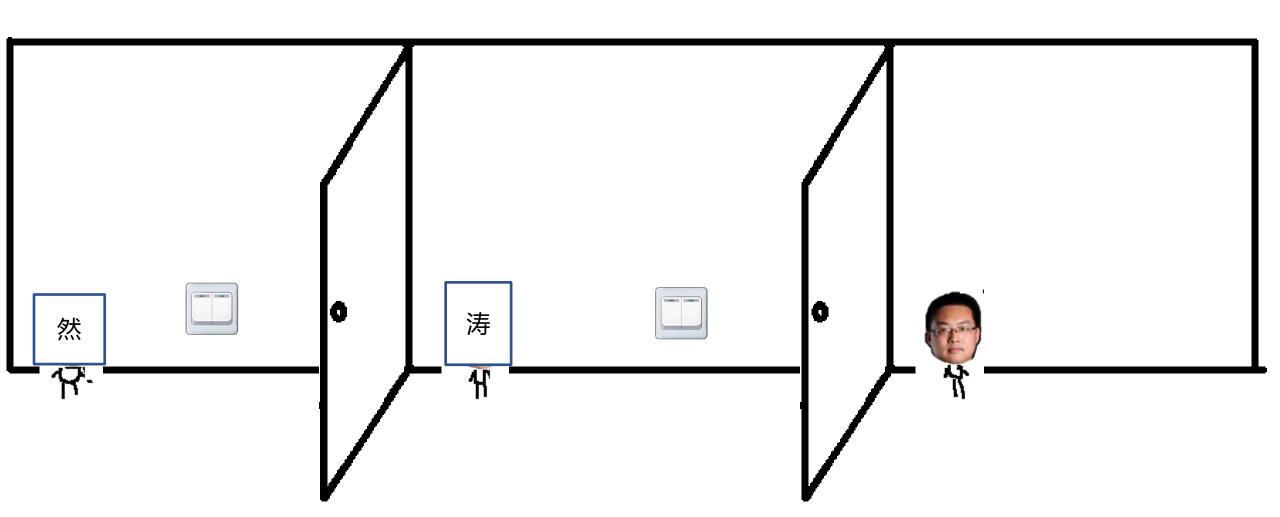




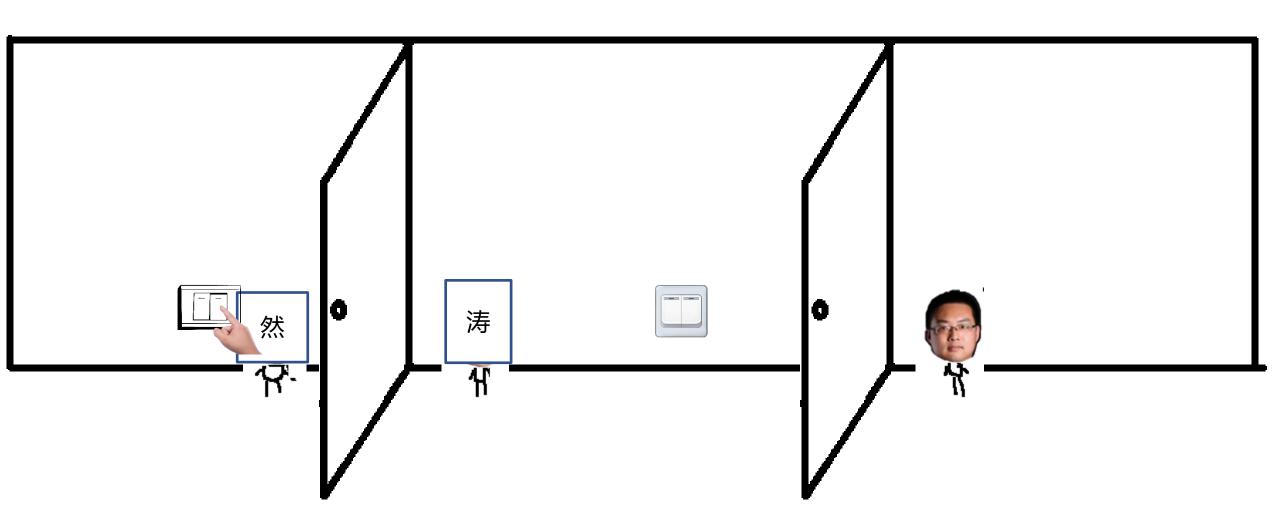
(3). 好的解决: Promise

- 今后,只要多个异步任务必须顺序执行时,
- 都要用promise技术来代替回调函数方式

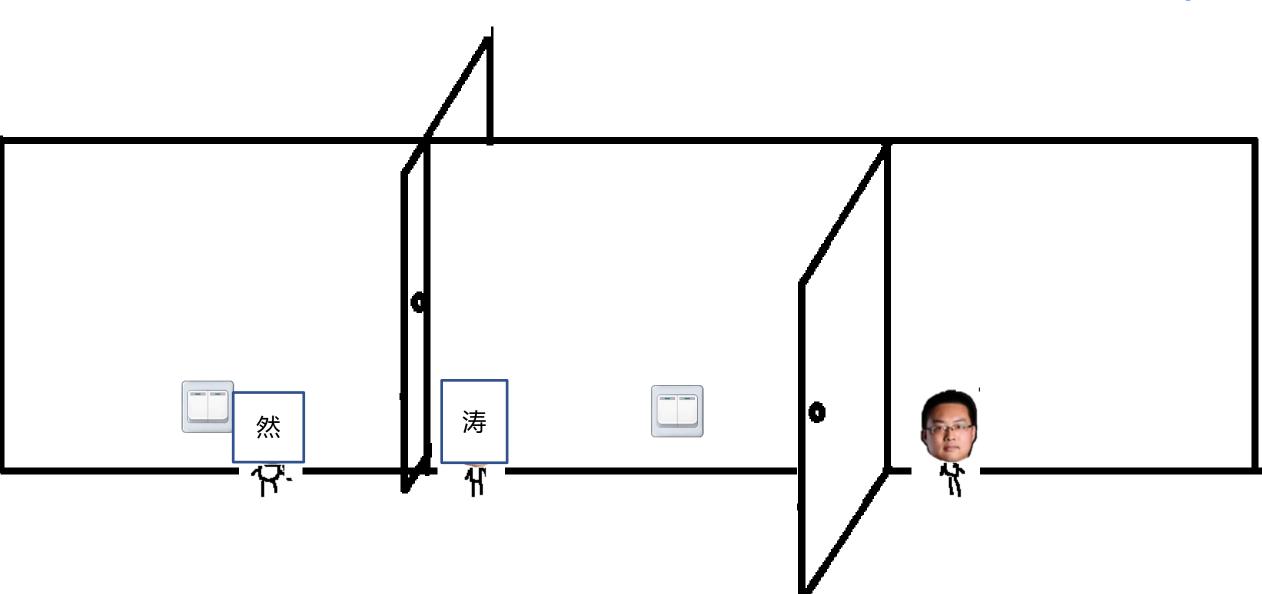




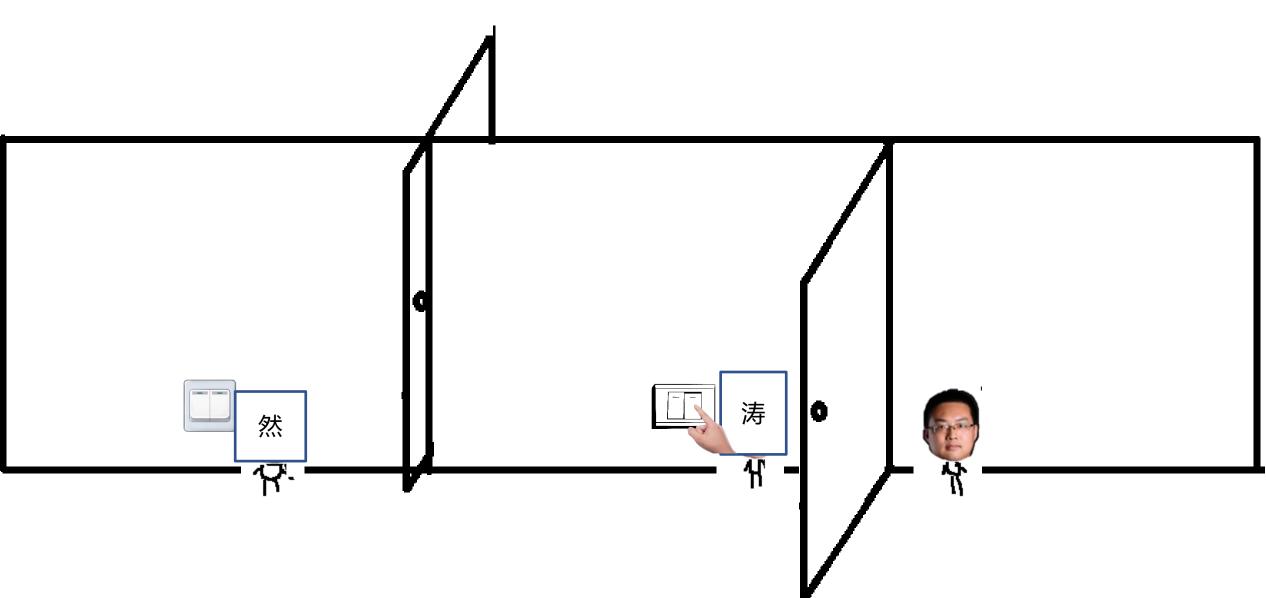


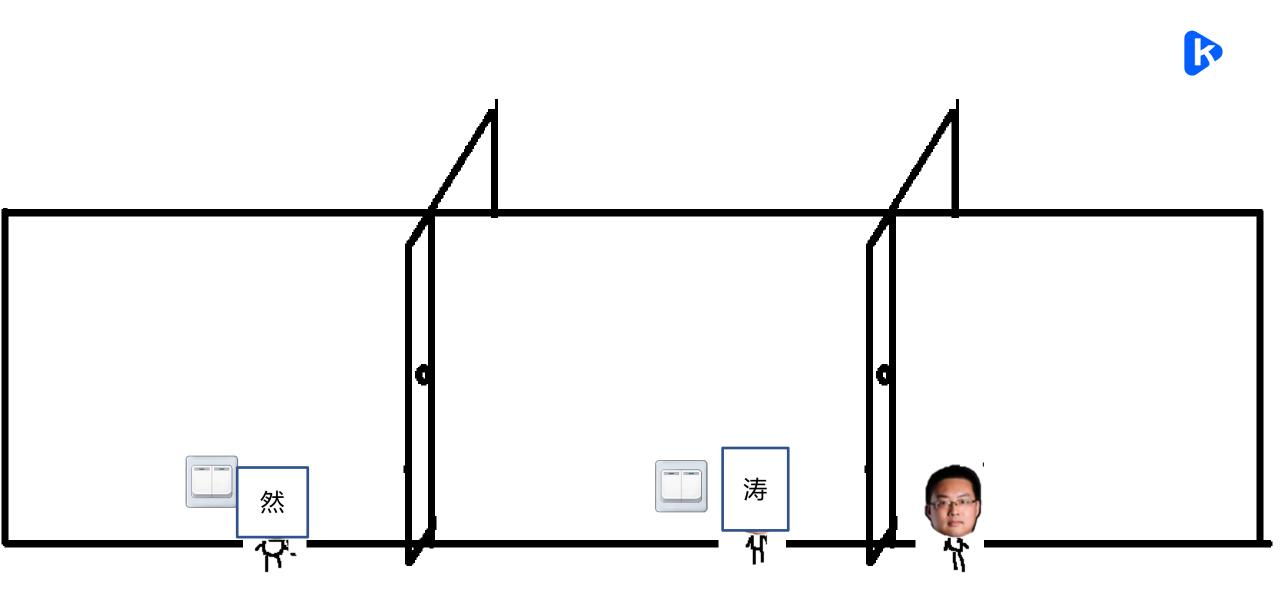












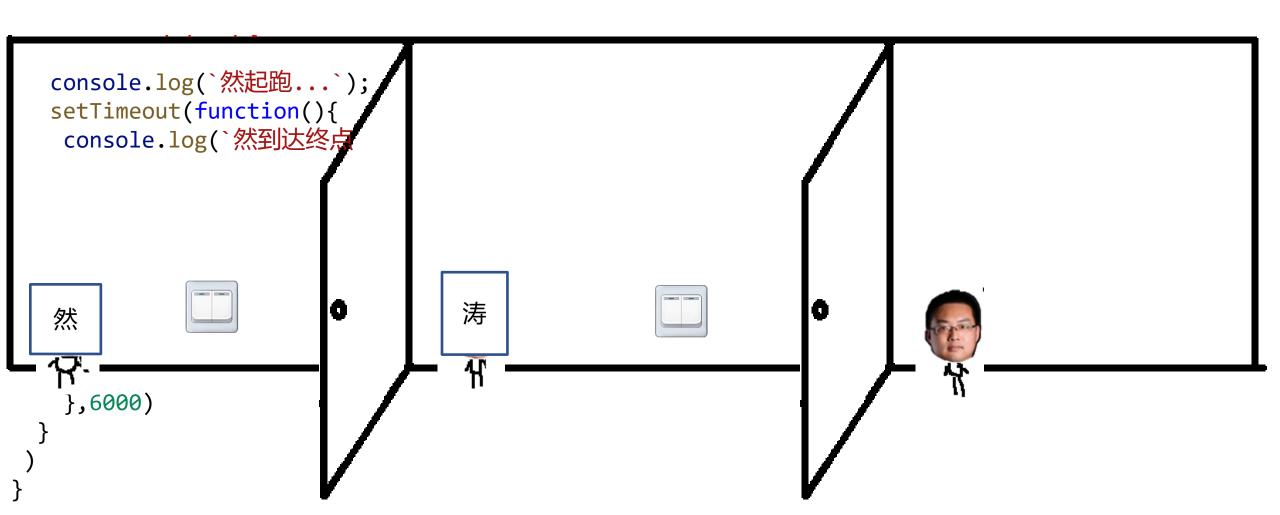
a. 如何: 2步:

1). 定义前一项任务时:

```
function 前一项任务(){
return new Promise(
 function(开关open){
  原异步任务
  异步任务最后一句话
  调用开关open()//->通向.then(),自动执行.then中的下一项任务
```

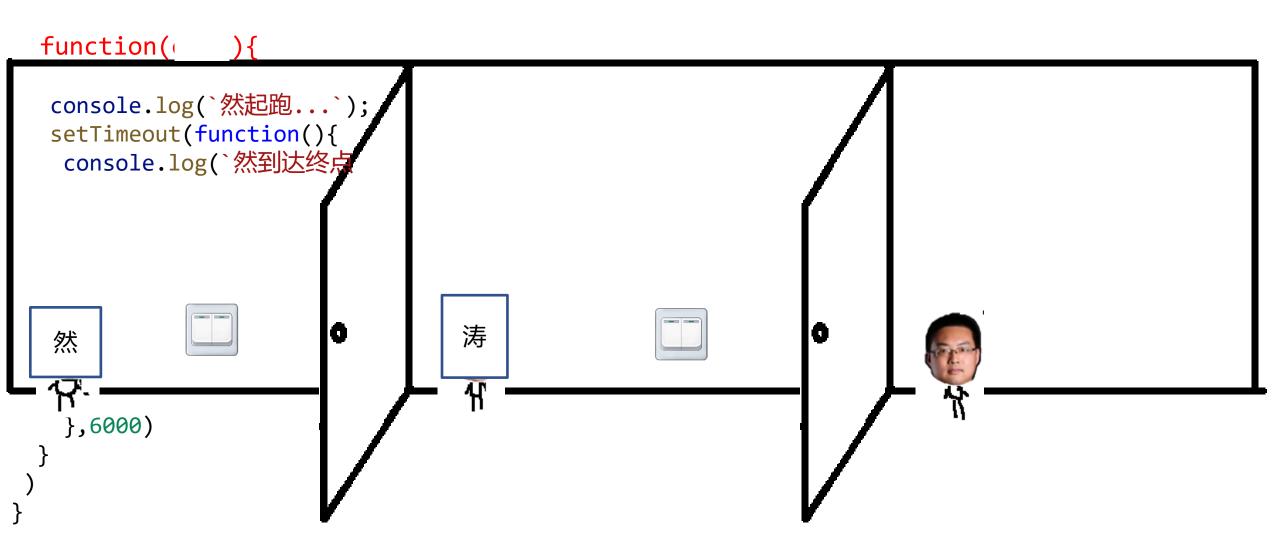
function ran(){





```
function ran(){
  return new Promise(
```

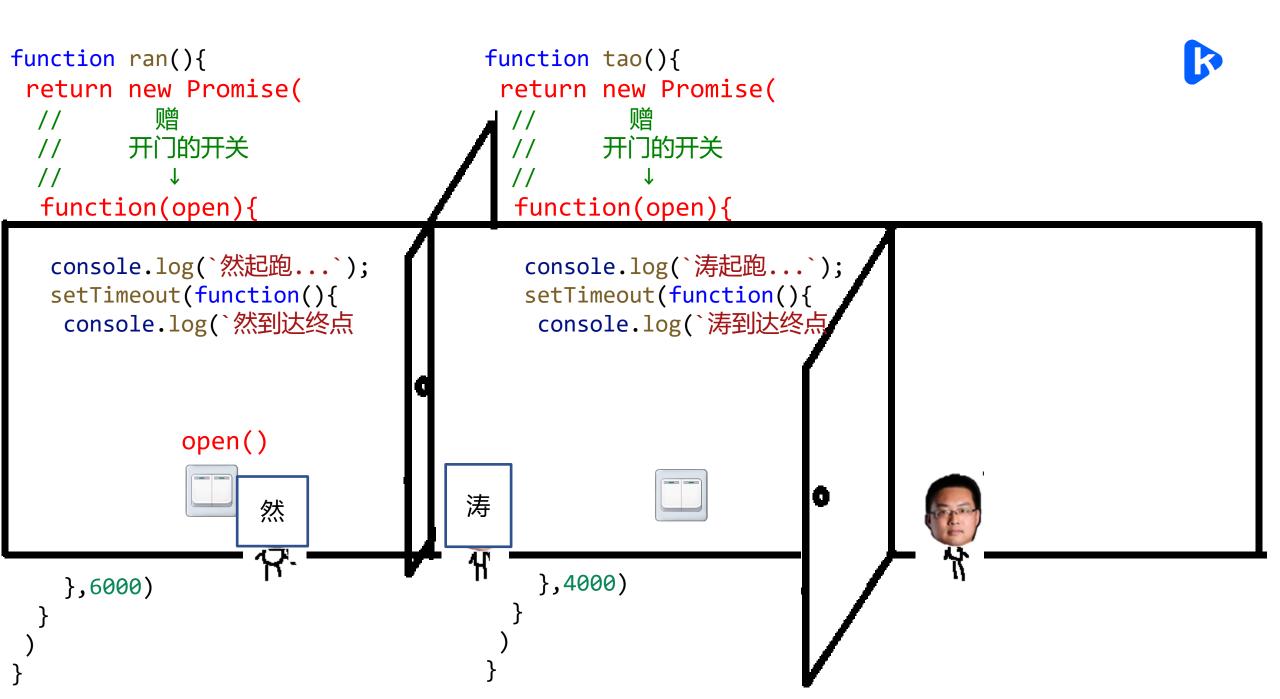




```
function ran(){
 return new Promise(
          赠
        开门的开关
  function(open){
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(`然到达终点
                               涛
   然
   },6000)
```

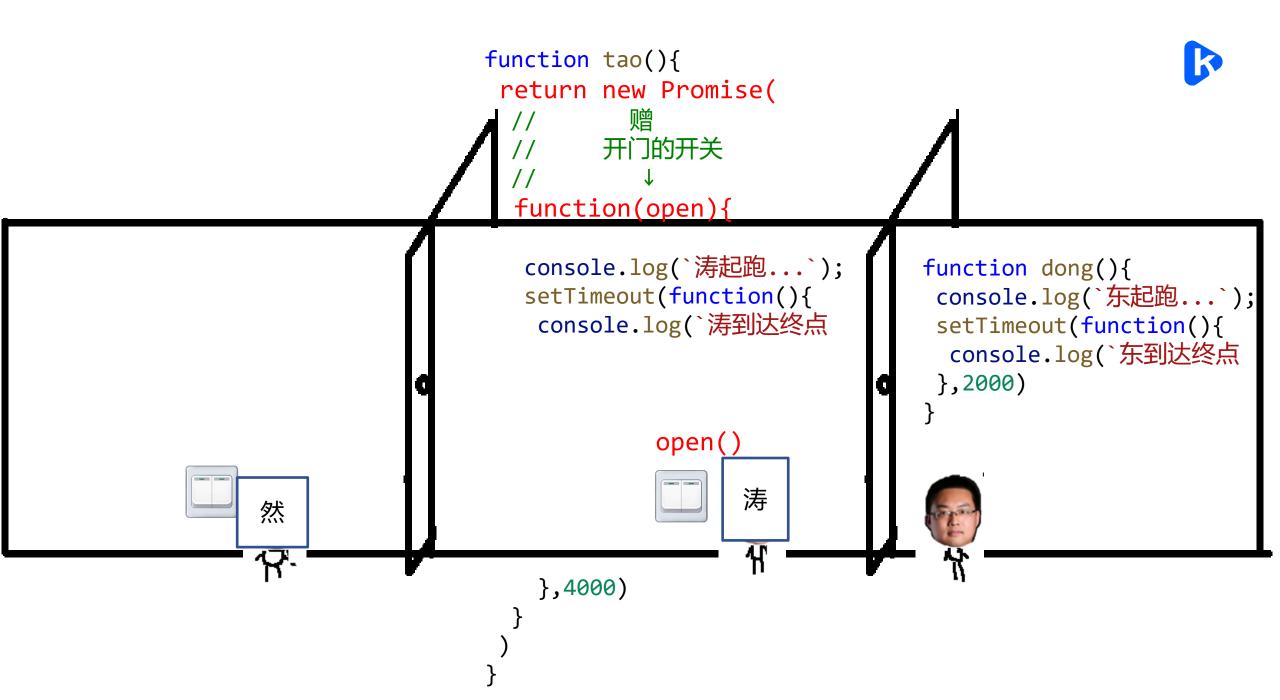
```
function ran(){
 return new Promise(
          赠
        开门的开关
  function(open){
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(`然到达终点
                               涛
   然
   },6000)
```

```
function ran(){
return new Promise(
          赠
        开门的开关
  function(open){
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(`然到达终点
           open()
                               涛
                                Ŧ
   },6000)
```

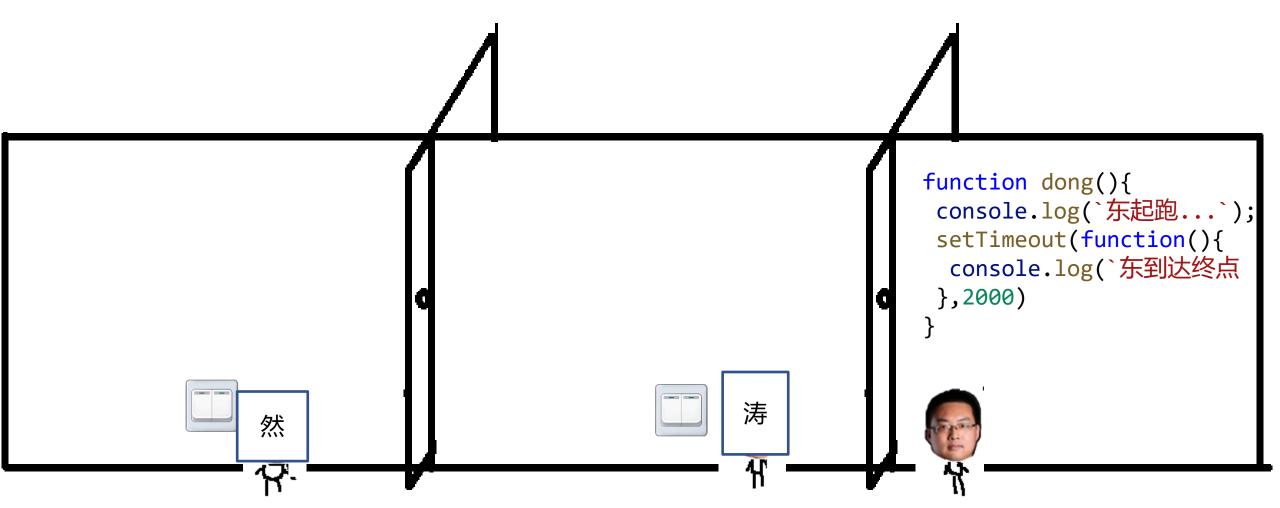


```
function tao(){
                return new Promise(
                         赠
                       开门的开关
                 function(open){
                  console.log(`涛起跑...`);
                  setTimeout(function(){
                   console.log(`涛到达终点
              涛
然
                   },4000)
```

```
function tao(){
                return new Promise(
                         赠
                       开门的开关
                 function(open){
                  console.log(`涛起跑...`);
                  setTimeout(function(){
                   console.log(`涛到达终点
                           open()
                                 涛
然
                   },4000)
```



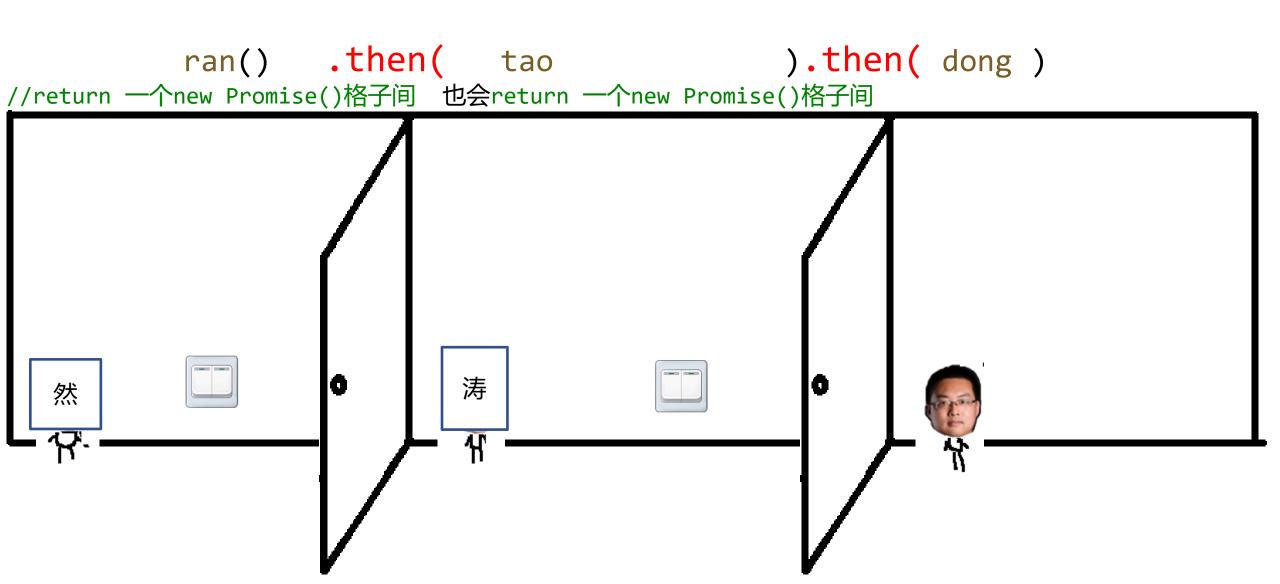




问题: 每个new Promise默认不相连

- 2). 连接前后两个Promise异步任务:
- 前一项任务().then(后一项任务)
- •强调:之后的任务一定不要加(),
- •加()表示立刻执行。
- 而我们不希望后边的任务立刻执行。
- 后续任务必须等待前一项任务执行完, 才能执行







```
然后
                                       然后
                                         ).then( dong )
         ran() .then(
                         tao
//return 一个new Promise()格子间 也会return 一个new Promise()格子间
                        涛
```

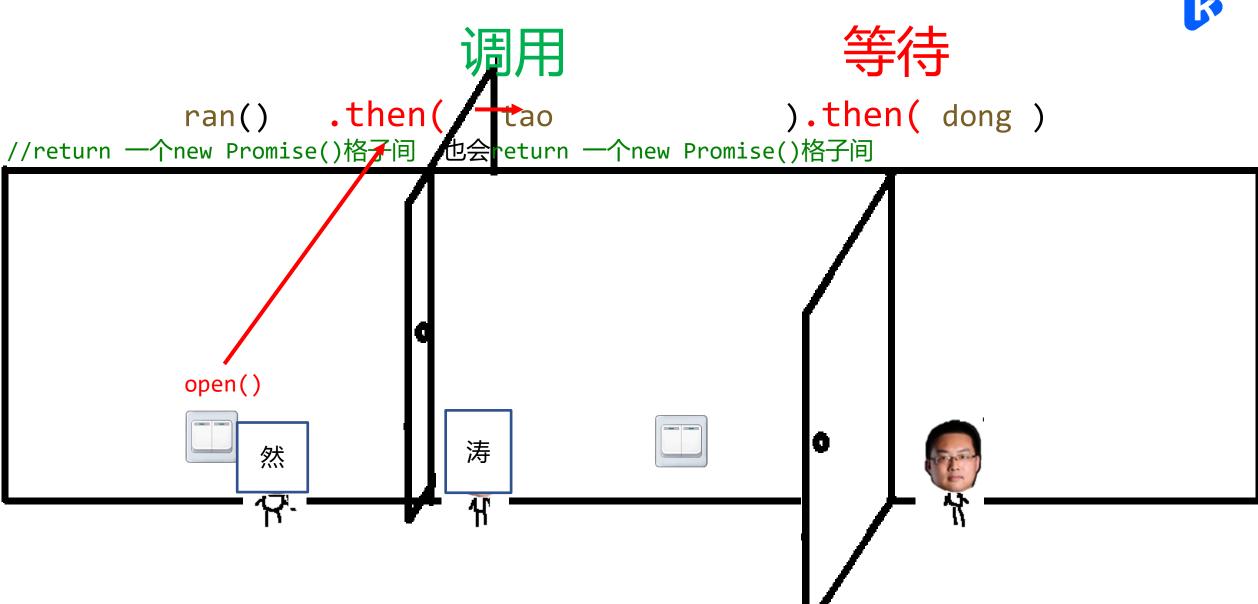


```
先调用
         ran() .then( tao
                                         ).then( dong )
//return 一个new Promise()格子间 也会return 一个new Promise()格子间
                        涛
   然
```

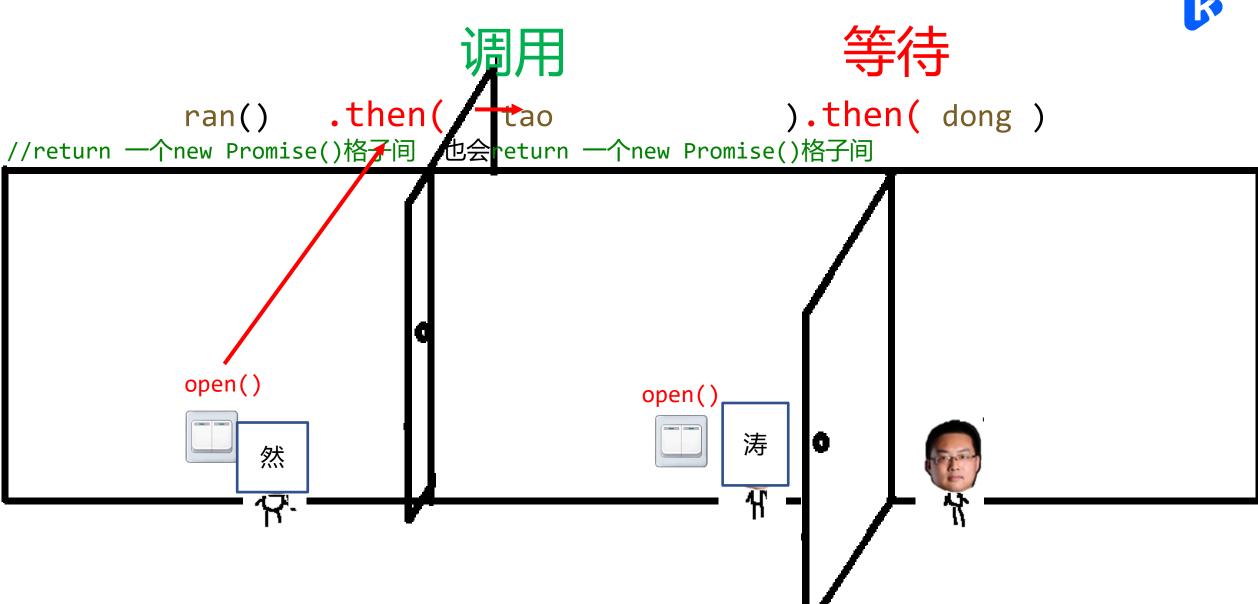


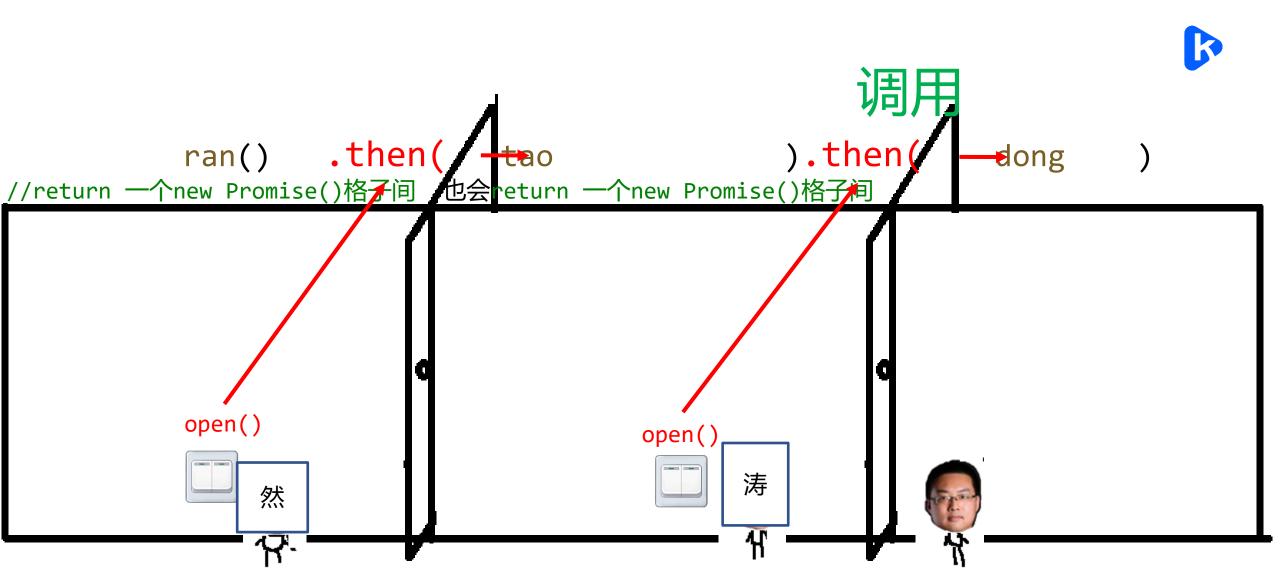
```
先调用
         ran() .then( tao
                                          ).then( dong )
//return 一个new Promise()格子间 也会return 一个new Promise()格子间
          open()
                         涛
              然
                         Ŧ
```











问题:

- 为什么会有要求下一项异步任务必须在前一项异步任务之后才能 执行?
- 往往因为下一项异步任务必须获得前一项异步任务的执行结果, 才能继续执行后续操作。
- 比如:田径接力比赛,前一个人跑完,必须把接力棒传给下一个人,下一个人才能开始跑。

所以:

前一项任务应该可以向下一项任务传值。

b. 前后两个任务之间 传参: 2步:

1). 前一项任务:

```
function 前一项任务(){
return new Promise(
    赠
 function(开关open){
  var 变量=值
  调用开关open(变量)//->通向.then(),自动执行.then中的下一项任务,并将open()中实参
值自动传给下一项任务函数
```

2). 后一项任务:

```
function 后一项任务(形参){

//形参=前一项任务中的变量值

//在后一项任务中使用形参,就是使用前一项任务传来的变量值
}
```

```
function ran(){
return new Promise(
          赠
        开门的开关
  function(open){
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然到达终点
          open()
                               涛
   然
   },6000)
```

```
function ran(){
return new Promise(
         赠
        开门的开关
  function(open){
   var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然到达终点
         open()
                               涛
   然
   },6000)
```

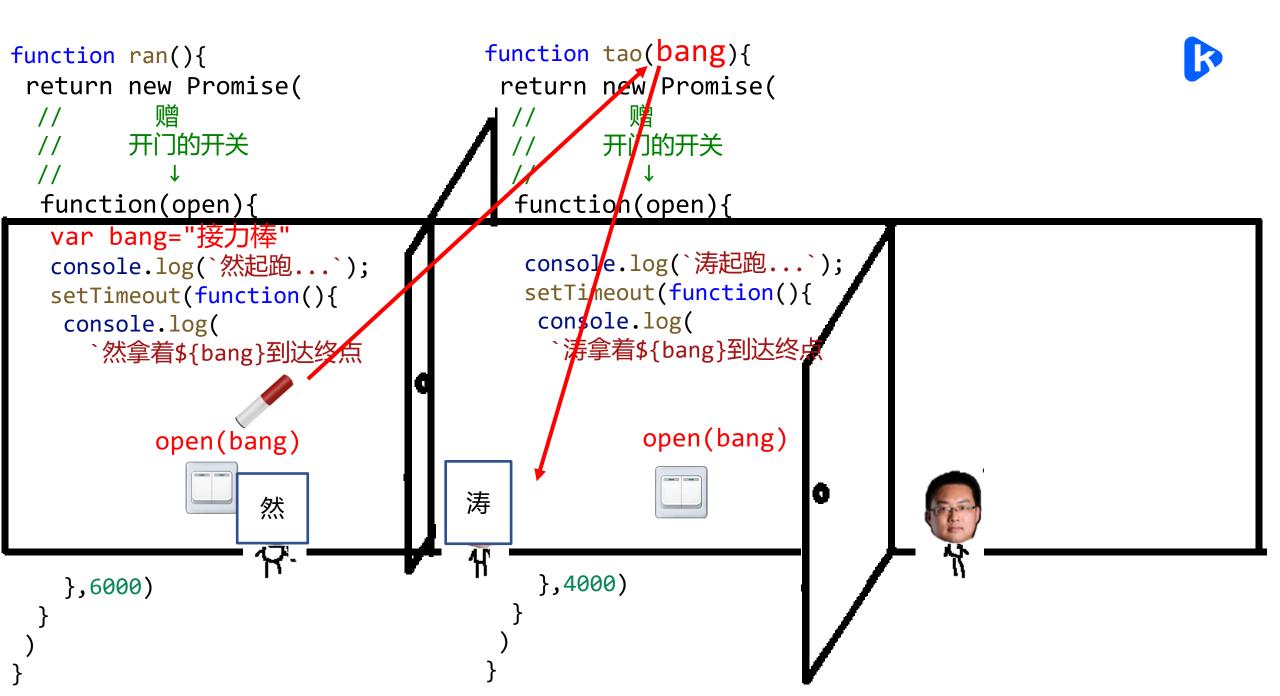
```
function ran(){
return new Promise(
         赠
       开门的开关
  function(open){
   var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然拿着${bang}到达缭点
         open()
                               涛
   然
   },6000)
```

```
function ran(){
return new Promise(
         赠
        开门的开关
  function(open){
  var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然拿着${bang}到达缭点
         open(bang)
                               涛
   然
   },6000)
```

```
function ran(){
return new Promise(
         赠
       开门的开关
  function(open){
  var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然拿着${bang}到达缭点
         open(bang)
                              涛
   然
   },6000)
```

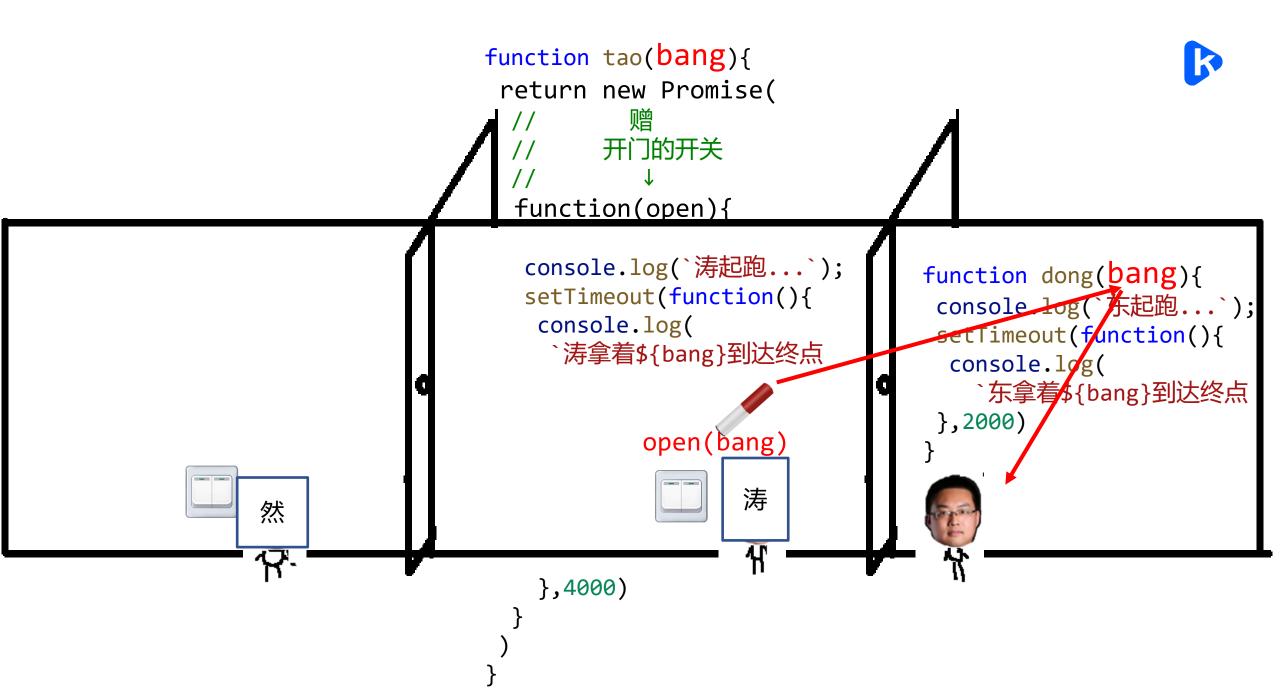
```
function ran(){
return new Promise(
         赠
        开门的开关
 function(open){
  var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然拿着${bang}到达缭点
         open(bang)
                               涛
   然
   },6000)
```

```
function ran(){
return new Promise(
          赠
        开门的开关
  function(open){
  var bang="接力棒"
  console.log(`然起跑...`);
  setTimeout(function(){
   console.log(
     `然拿着${bang}到达缭点
          open(bang)
                               涛
                               ዠ
   },6000)
```

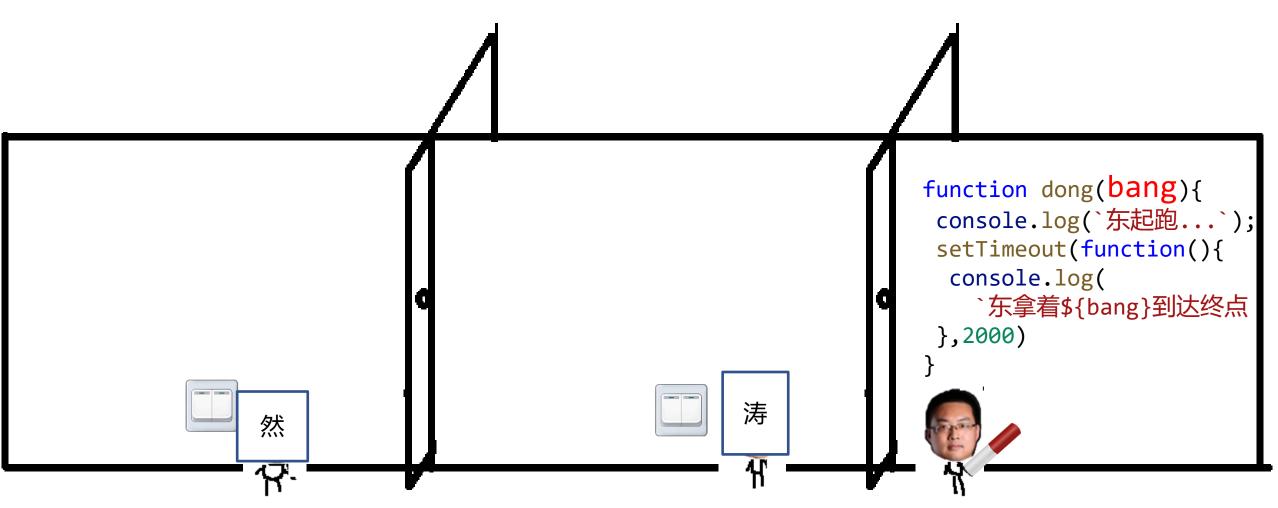


```
function tao(bang){
                return new Promise(
                         赠
                       开门的开关
                 function(open){
                  console.log(`涛起跑...`);
                  setTimeout(function(){
                   console.log(
                    `涛拿着${bang}到达终,
                          open(bang)
              涛
然
                   },4000)
```

```
function tao(bang){
                return new Promise(
                         赠
                       开门的开关
                 function(open){
                  console.log(`涛起跑...`);
                  setTimeout(function(){
                   console.log(
                    `涛拿着${bang}到达终原
                          open(bang)
                                 涛
然
                   },4000)
```





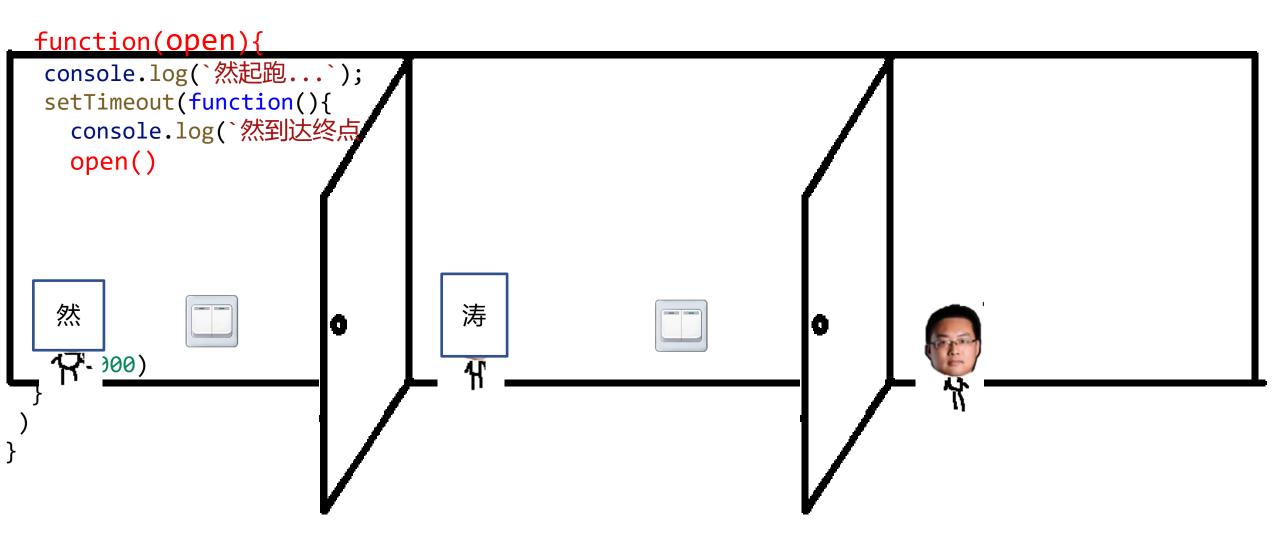


问题: 异步任务执行过程中很可能出错

- 一旦前一项任务出错
- 就没必要继续执行后续任务了
- 而要执行额外的错误处理代码

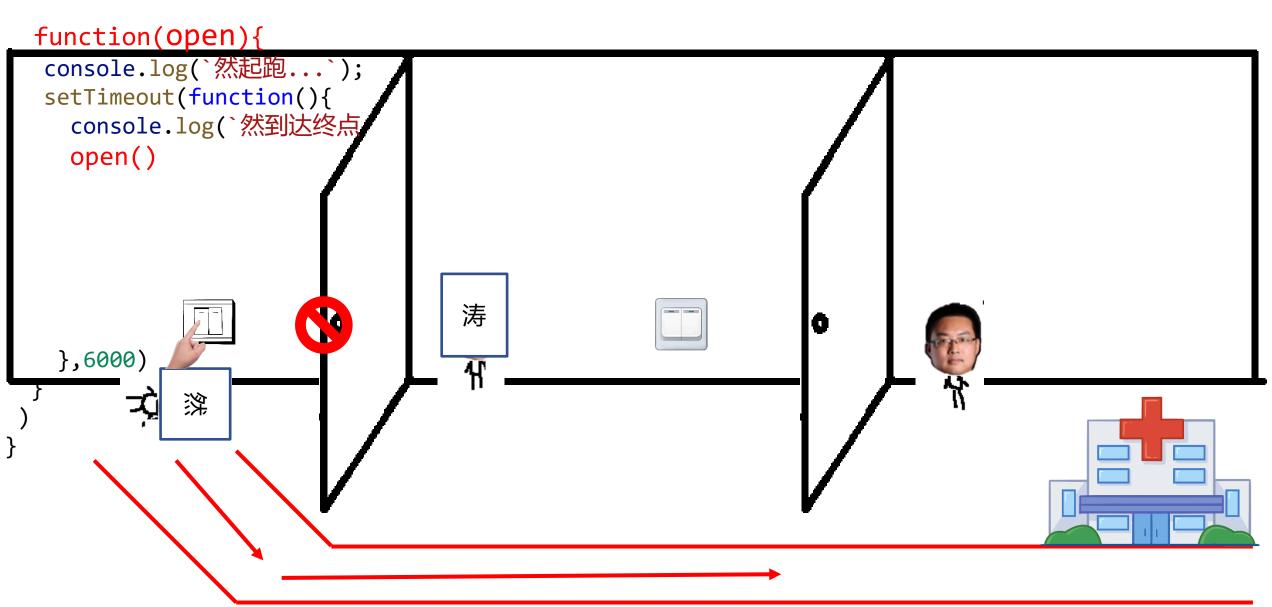
```
function ran(){
  return new Promise(
```





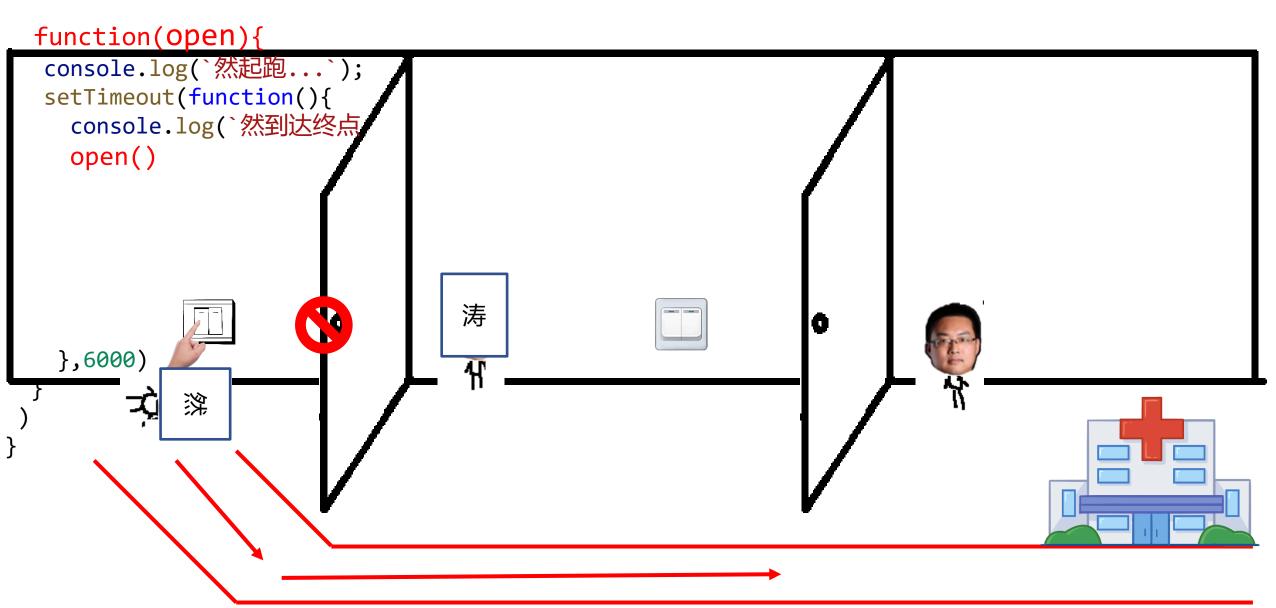
```
function ran(){
  return new Promise(
```





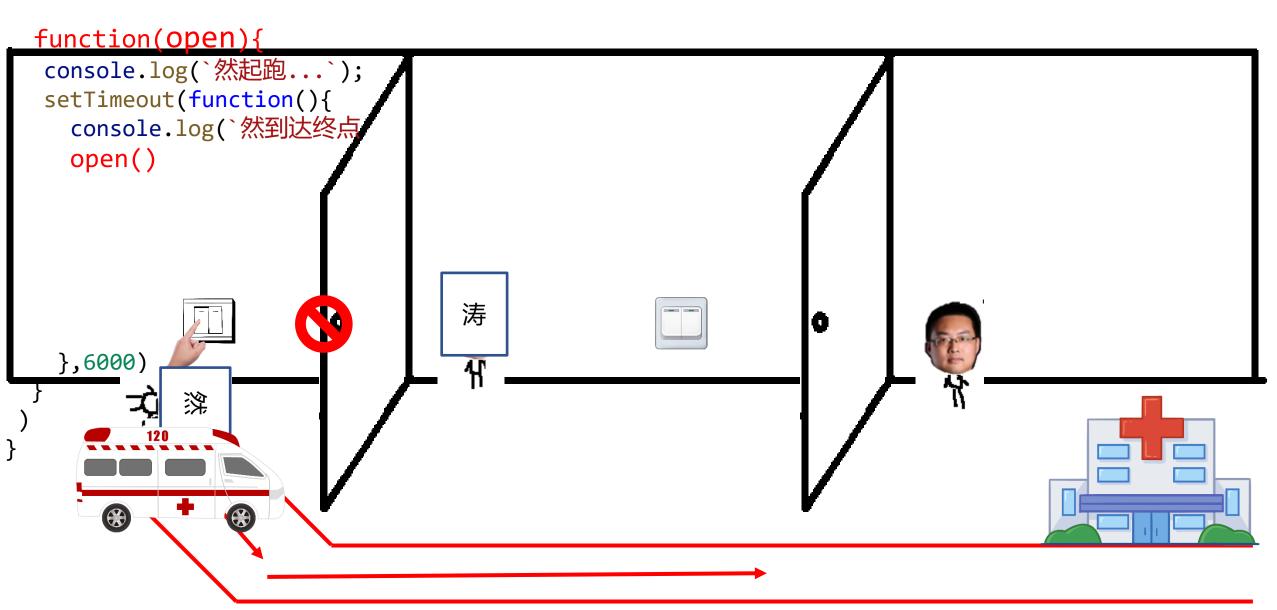
```
function ran(){
  return new Promise(
```





```
function ran(){
  return new Promise(
```





d. 错误处理: 2步

•1). 前一项任务:

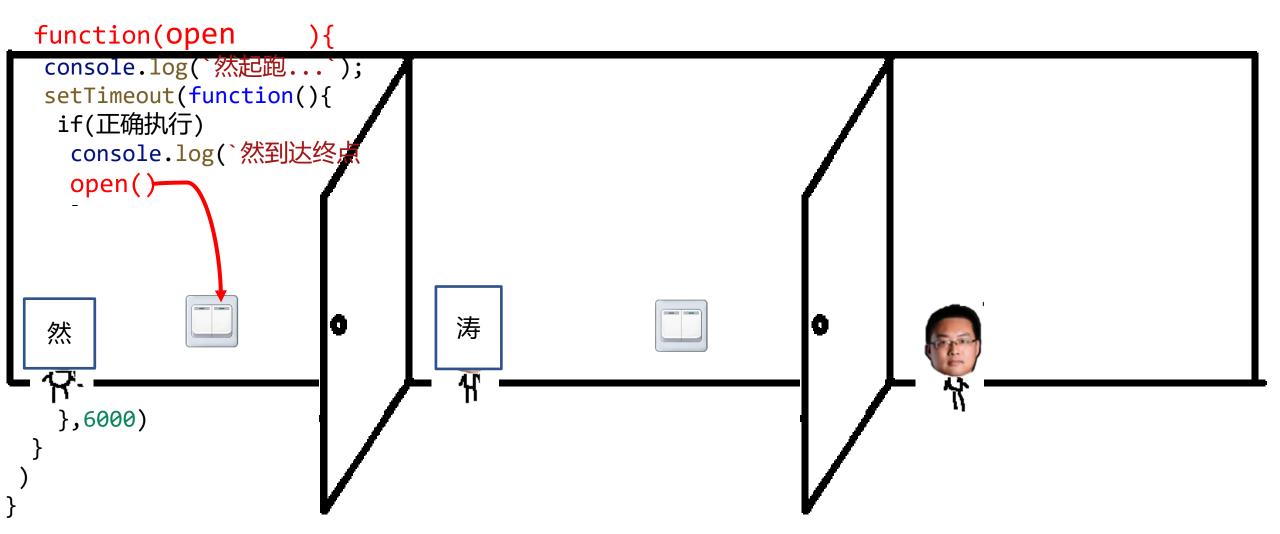
```
function 前一项任务(){
return new Promise(
 // 赠 两个开关
 function(成功的开关open, 失败的开关err){
  var 变量=值
  原异步任务
  异步任务最后一句话
  如果异步任务执行成功
   调用成功的开关open(变量)
       //->通向.then(), 正常自动执行.then中的下一项任务
  否则如果异步任务执行失败
   调用失败的开关err(错误提示信息)
       //-><mark>通向最后的.catch(/</mark>),不同.then(),后续.then()不再执行。
```

2). 调用时:

```
前一项任务()
.then(下一项任务)
.then(...)
.catch(
function(错误提示信息){ 错误处理代码 }
```

```
function ran(){
  return new Promise(
```



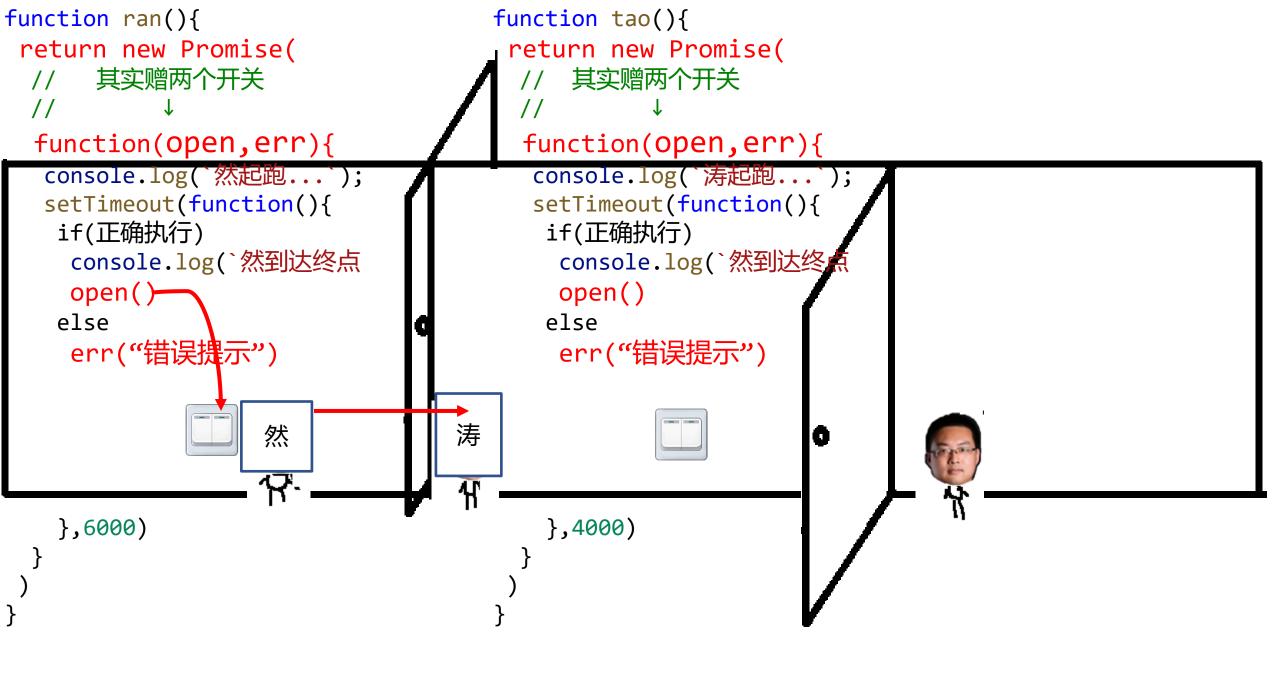


```
function ran(){
return new Promise(
      其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
                              涛
  然
   },6000)
```

```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终身
    open()
   else
    err("错误提示")
                              涛
  然
   },6000)
```

```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
   else
    err("错误提示")
                              涛
  然
   },6000)
```

```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终身
    open()
   else
    err("错误提示")
                              涛
                 然
   },6000)
```

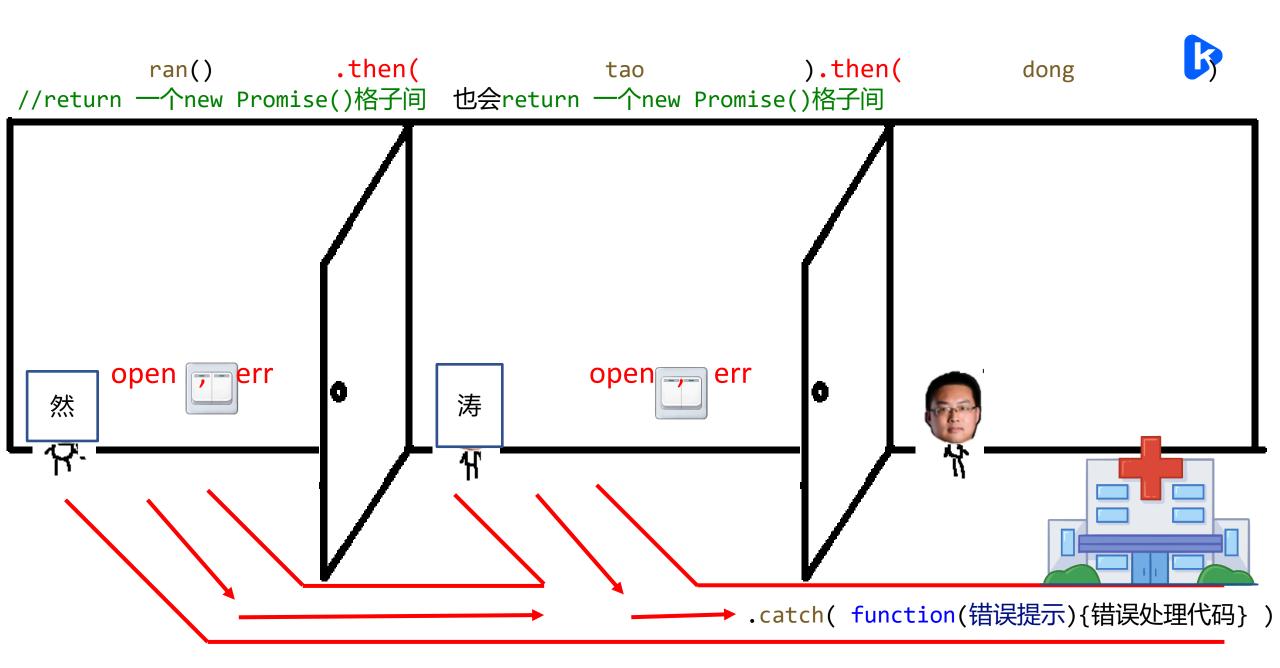


```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
   else
    err("错误提示")
   然
                              涛
   },6000)
```

```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
   else
    erṛ("错误提示")
                              涛
                              ዠ
           然
   },600
                                                    .catch( function(str){错误处理代码} )
```

```
function ran(){
return new Promise(
     其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
   else
    erṛ("错误提示")
                              涛
                              ዠ
           然
   },600
                                                    .catch( function(str){错误处理代码} )
```

```
function ran(){
return new Promise(
      其实赠两个开关
 function(open,err){
  console.log(`然起跑...`);
  setTimeout(function(){
   if(正确执行)
    console.log(`然到达终原
    open()
   else
    erṛ("错误提示")
                                涛
                                \overline{\mathbb{Y}}
            終
   },600
      .catch(function(错误提示){错误处理代码}
```



e. Promiss对象三大状态:

• 1). 当异步任务执行过程中,

• 整个new Promise()对象处于pending(挂起)状态

new Promise(){
function(resolve, reject){
异步任务
1. pending

完 (错误处理函数)

e. Promiss对象三大状态:

- 2). 当异步任务成功执行完,
- 调用成功的开关函数(resolve())时,
- 整个new Promise()对象切换为fulfilled(成功)状态,

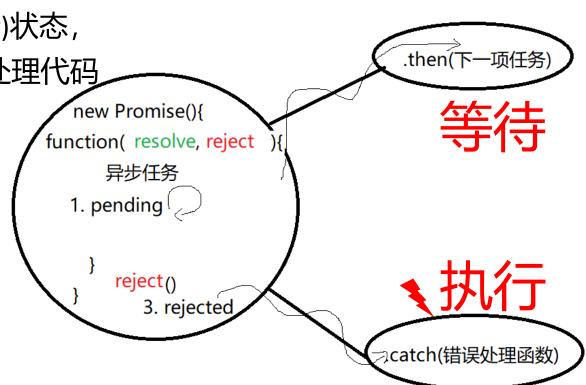
• new Promise()会自动调用.then()执行下一项任务

.then(下一项任务) new Promise(){ function(resolve, reject 异步任务 1. pending [resolve () catch(错误处理函数)

e. Promiss对象三大状态:

- 3). 当异步任务执行出错,
- 调用失败的开关(reject())函数时,
- 整个new Promise()对象切换为rejected(出错)状态,

• new Promise()会自动调用.catch()执行错误处理代码



e. Promiss对象三大状态:

- 1). pending(挂起)
- 2). fulfilled(成功)
- 3). rejected(出错)



Q4: async & await

async和await 其实就是promise中.then()的简写 目的是彻底消除嵌套



promise中.then的问题

- promise中.then()的问题是依然没有彻底消除嵌套
- 比如:

```
前一项任务函数().then(
function(){
    下一项任务要做的事儿。
}
```

今后,调用 promise函数时 的.then都可简 写为async和 await的组合。

```
(async function(){
var 返回值=await 第一项任务();
await 第二项任务(上一项任务的返回值)
第二项任务结束后才执行的操作;
})()
```

- 1. 只有基于Promise的函数,才支持async和await
- 12. await必须用在被async标记的函数内
- 3. 外层函数必须用async标记。目的是告诉主程序,这段函数内的 比代码整体是异步执行的。不影响主程序的执行。
- 13. await必须写在前一项任务之前
- 4. await的作用等效于.then(),用来通知程序必须等待前一项任务执行完,才能继续执行后续任务。
- 5. 一旦使用了await,前一项任务的resolve(返回值),可以像普通函数一样用=接住。后续代码可继续使用该变量里获得的返回值。
- 6. await和.then()一样,可多次使用。控制多个异步任务顺序执行。



比如:

```
ran().then(tao).then(dong)
可改为:
(async ()=>{
 var bang=await ran();
 bang=await tao(bang);
 dong(bang);
})()
```

所以,async和await的组合 虽然相对于主程序,整体是异步的 但async内部的多个await 是同步顺序执行的。



错误处理也和程序的错误处理一致了

• 用try catch组合代替了.catch(function(){ ... })函数嵌套调用,可自动接住 promise函数中reject()抛出的异常信息。

```
比如:
• (async ()=>{
 try{
   var bang=await ran();
     bang=await tao(bang);
     dong(bang);
  }catch(err){ //
   错误处理代码
```



Q5: 手写Promise

状态机

```
var PENDING = 0; //挂起状态
var FULFILLED = 1; //执行成功状态
var REJECTED = 2; //执行失败状态
//定义Promise构造函数,将来用new创建Promise对象
function Promise() {
// state变量存储当前Promise对象的执行状态
var state = PENDING;
// value变量存储执行成功后的返回值,或执行失败后的错误提示信息
var value = null;
// handlers变量是一个数组,存储将来用.then()函数传入的一个或多个后续任务函数。
var handlers = [];
```

状态变迁——定义两个底层专门修改状态的函数



```
// ...
function Promise() {
 // ...
function fulfill(result) { //执行成功后,把状态改为成功状态,并把执行结果返回值,保存在变量value中
 state = FULFILLED;
 value = result;
function reject(error) { //执行失败后,把状态改为失败状态,并把错误提示信息,保存在变量value中
 state = REJECTED;
 value = error;
```

fulfill和reject方法较为底层, 通常会对外开放一个更高级的resolve方法。



```
// ...
function Promise() {
// ...
function resolve(result) { //如果当前任务成功执行完成,使用者调用了resolve(返回值)
 try {
  var then = getThen(result); //就要收集当前Promise对象身上后续的.then()函数中传入的内容
  if (then) { //如果有.then
   //就调用核心doResolve函数,执行.then()中的函数,并传入两个状态切换函数。
   doResolve(then.bind(result), resolve, reject) //resolve和doResolve之间的递归用来处理promise的层层嵌套
  return
  //如果没有.then,就直接切换当前Promise对象的状态,并返回执行结果,结束当前Promise对象的执行
  fulfill(result);
 } catch (e) {
  //如果调用过程中出错,就调用reject()函数,将当前Promise状态切换为失败,并返回错误提示信息
  reject(e);
```

实现getThen()函数



```
/**
*.then()函数中传入的内容有两种情况: 可能传入的是下一个Promise对象, 也可能直接传入匿名函数
* 如果调用resolve时传入的是下一个Promise对象,
*就返回这个Promise对象的.then()函数.
* 如果调用resolve时传入的是下一个函数
* 就直接返回这个函数即可
* 如果调用resolve时什么都没传,就返回null
function getThen(value) {
var t = typeof value;
if (value && (t === 'object' || t === 'function')) {
 var then = value.then;
 if (typeof then === 'function') {
  return then;
return null;
```

实现doResolve()函数——核心



```
*调用传入的.then()函数,并传入执行成功和执行失败两个修改状态的回调函数*/
function doResolve(fn, onFulfilled, onRejected) {
var done = false; //默认暂时未执行成功
try {
 fn( //调用当前任务函数
function (value) { //传入执行成功后,请使用者主动调用的res函数
  if (done) return //如果done被标记为true,说明当前异步任务执行完
  done = true //否则如果done暂时未被标记为true,就标记为true,让当前异步任务状态变为完成状态
  onFulfilled(value) //调用传入的第一步定义的改变当前Promise状态的函数,把当前Promise对象标记为执行成功,并保存返回值
              //这里调用了resolve函数。
  function (reason) {//传入执行失败后,请使用者主动调用的resolve函数
  if (done) return //如果done被标记为true,说明当前异步任务执行完
  done = true //否则如果done暂时未被标记为true,就标记为true,让当前异步任务状态变为完成状态
  onRejected(reason) //调用传入的第一步定义的改变当前Promise状态的函数,把当前Promise对象标记为执行失败,并保存错误提示
} catch (ex) { //如果出现异常
 if (done) return //如果done被标记为true,说明当前异步任务执行完,就退出当前任务的执行
 done = true //否则如果done暂时未被标记为true,就标记为true。
 onRejected(ex) //调用传入的改变状态函数,把当前Promise对象标记为执行失败
```

调用核心函数doResolve(), 在new时就执行当前任务。

```
// ...
function Promise(fn) {
    // ...
    doResolve(fn, resolve, reject); //调用上一步的doResolve(...)

/*
    fn: function(res, rej){ 异步任务; 执行成功res(返回值); 执行失败rej(错误提示) }
    resolve是上一步定义的resolve函数
    reject是第一步定义的出错后,将当前Promise对象状态修改为失败的回调函数
    */
}
```

定义函数handler,用来根据当前Promise对象的状态 决定调用handlers数组中的哪种处理函数

```
// ...
function Promise(fn) {
 // ...
 function handle(handler) {
  if (state === PENDING) {
   handlers.push(handler);
  } else {
   if (state === FULFILLED &&
    typeof handler.onFulfilled === 'function') {
    handler.onFulfilled(value);
   if (state === REJECTED &&
    typeof handler.onRejected === 'function') {
    handler.onRejected(value);
```

为当前Promise对象添加.done()方法



```
function Promise(fn) {
// ...
this.done = function (onFulfilled, onRejected) {
 // 使用定时器,确保当前任务一定是异步执行的
  setTimeout(function () {
   handle({ //传入修改状态的两个回调函数
   onFulfilled: onFulfilled,
   onRejected: onRejected
  });
 }, 0);
```

为当前Promise对象添加.done()方法



```
// ...
function Promise(fn) {
 // ...
 this.then = function (onFulfilled, onRejected) {
   var self = this; //保存当前Promise对象
   return new Promise(function (resolve, reject) { //创建并返回下一个Promise对象
    return self.done(
     function (result) { //调用当前对象的done()函数
      if (typeof onFulfilled === 'function') { //如果.then()中传入的是一个函数
       try {
        return resolve(onFulfilled(result)); //就调用resolve,传入下一项任务的函数,执行。
       } catch (ex) { return reject(ex);}
      } else {
       return resolve(result); //否则就传入下一个Promise对象,继续等待。
     function (error) {
      if (typeof onRejected === 'function') {
       try {
        return resolve(onRejected(error));
       } catch (ex) {return reject(ex);}
      } else {
       return reject(error);
```



Q6: 作业 ——promise相关笔试题



我们的口号是: 早学早面试卷别人晚学晚面试被人卷