

# 数据类型

在JS中，基本数值类型包括：

类型	值	构造函数
null	null	无
undefined	undefined	无
string	"hello world"	String()
number	123	Number()
bigint [ES10]	123n	BigInt()
boolean	true false	Boolean()
symbol [ES6]		无
object	{}	Object()

JS中的5大简单数据类型其实就是**值类型**，而object是**引用类型**。

## boolean

布尔类型只有true和false，跟其他语言一模一样。

## number

### 数字的存储

### 特殊的number常量

几个特殊的常量

- NaN：  
NaN即非数值（Not a Number），是一个特殊的数值；**在js中任何数值和非数值运算都会返回NaN，不会影响其他代码执行。**  
**任何涉及NaN的操作都会返回NaN；除了NaN+字符串。**  
在js中，只有一个东西不等于自己，那就是NaN。

```
1 NaN==NaN
2 //false
```

判断NaN的时候，可以使用官方提供的API `Number.isNaN()`，原理如下。

```

1 function isNaN(n){
2     return typeof n == 'number' && n!=n
3 }

```

- `Number.MAX_VALUE`  
最大值
- `Number.MIN_VALUE`  
可表示的最小正数
- `Infinity` 或者 `Number.MIN_VALUE*2`:  
无穷大
- `-Infinity` 或者 `-Number.MAX_VALUE*2`  
无穷小
- `Number.MAX_SAFE_INTEGER`  
可以安全表示的最大整数
- `Number.MIN_SAFE_INTEGER`  
可以安全表示的最小整数

```

1 var intNum = 77; //十进制整数;
2
3 var octalNum = 070; //八进制56
4
5 var hexNum = 0xA //十六进制10
6
7 var floatNum = 10.1 //浮点数
8 document.write("数值型的变量数据类型是"+typeof(floatNum));

```

判断非数值类型:

如果非数值就会返回true。

```

1 |

```

## string

```

1 var string ="我叫string数据类型是";
2 document.write(string);
3 document.write(typeof(string));

```

## undefined

声明但是未定义的变量，都是undefined

```
1 var a;  
2 document.write("我是a, 我是没有定义值, 所以数据类型是"+typeof(a));  
3 a=null;  
4 document.write("我是a, 我是没有定义值, 所以数据类型是"+typeof(a));
```

## object

object是所有对象的父类, 所有引用变量都是object类型。

object是引用数据类型, 里面操作的都是对象的引用 (指针)

```
1 var a=[1,2,3];  
2 document.write(typeof(a));  
3 var b={  
4   name:123  
5 };  
6 document.write(typeof(b));  
7 var c=function(){};  
8 document.write(typeof(c));
```

## null

null也是一种object类型的数据, 但是这个代表空对象。

实际上这个玩意就是JS设计时的bug, 理论上来说null 的类型就是null, 但是因为这个bug存在的时间已经非常长了, 大家早就习惯了, 而且修理起来很困难, 现在我们一般把这种没法修的bug叫特性。

JavaScript的特性之一就是 `typeof null === 'object'`。

而 `null instanceof Object == false`

所以我们为了准确判断null, 可以使用以下的代码。

```
1 var a = null;  
2 (!a && typeof a === "object");//true
```

因为null是唯一一个false并且typeof会返回object的类型。

这时候就可以理解, 为什么空对象返回是true, 如果其他对象返回值也是false, 那么就无法来判断null了。

## bigint [ES10]

在JS中, 可以安全表示的最大整数为 `Number.MAX_SAFE_INTEGER`, 也就是9007199254740991, 如果超出了这个值, 运算会静默失败。

```
1 const maxInt = Number.MAX_SAFE_INTEGER;  
2 console.log(maxInt) //9007199254740991  
3 console.log(maxInt+1) //9007199254740992  
4 console.log(maxInt+100) //9007199254741092  
5 console.log(maxInt+1000) //9007199254741992
```

可以看到，值已经无法再增加了。

对于这种情况，我们可以定义bigint类型，来计算这种超大的数。

```
1 | const maxInt = BigInt( Number.MAX_SAFE_INTEGER);
2 | console.log(maxInt) //9007199254740991n
3 | console.log(maxInt+1n)//9007199254740992n
4 | console.log(maxInt+100n)//9007199254741091n
5 | console.log(maxInt+1000n)//9007199254741991n
```

**注意！ bigint不可以和number的数进行直接运算。**

比如：

```
1 | 1000n/10n //100n
2 | 1000n/10 //Uncaught TypeError: Cannot mix BigInt and other
   | types, use explicit conversions
```

同样的，无法用+运算符转型为number

```
1 | +100n //Uncaught TypeError: Cannot convert a BigInt value to a
   | number
```

## symbol [ES6]

- symbol类型没有构造函数，只能用Symbol()函数来构建。

```
1 | let sym = Symbol('sym')
```

- 获取Symbol的时候，只能使用 `Object.getOwnPropertySymbols` 方法

```
1 | var age = Symbol('age')
2 | girl={}
3 | girl[age] = 14
4 | console.log( Object.getOwnPropertySymbols(girl))
```

symbol类型是专门为对象服务的，symbol可以定义对象的私有属性。

**定义symbol的时候，必须要用中括号！**

```

1 | let girl = {
2 |   name: "莲华",
3 |   [Symbol('age')]: 14
4 | }
5 |
6 | console.log(girl.name) //'莲华'
7 | console.log(girl.age) //undefined

```

同时，symbol对于Object.entries、Object.keys和for.....in不可见。

除此之外，symbol属性是唯一的，多个对象生成的symbol是不一样的。

```

1 | Symbol("test") == Symbol("test") //false

```

如果有特殊的需要，可以使用symbol的全局注册表

```

1 | let sym1 = Symbol.for("test")
2 | let sym2 = Symbol.for("test")
3 | console.log(sym1 == sym2) //true

```

## 类型转换

### 显式类型转换

#### 数值

正常情况下，如果字符串里面是纯数字，那么这个方法就会返回字符串里面的值

```

1 | 123 === Number("123")

```

如果字符串里面有非数值型数据，那么返回NaN

```

1 | isNaN( Number("123a") ) //true

```

注意！有一些特殊的数值转换

```

1 | Number(null) === 0
2 | Number(true) === 1
3 | isNaN(Number(undefined)) //true

```

也可以用parseInt()方法，强制转化成整数。而且会去掉末尾的单位。

**注意：只能去掉末尾的。**

```

1 | console.log(parseInt("100px")) //100

```

同样的，也有 `parseFloat` 方法。

## 字符串

- `toString()`

这个方法不会改变原来数据的值。

```
1 | var a = 1;  
2 | a.toString();
```

- 使用构造函数

```
1 | var b = 1;  
2 | String(b);
```

- 如果是数组的话，还可以使用 `join` 方法，来填充分隔符。

```
1 | var a = ["物理", "化学", "生物"]  
2 | console.log(a.join("和")+"都不会")  
3 | //物理和化学和生物都不会
```

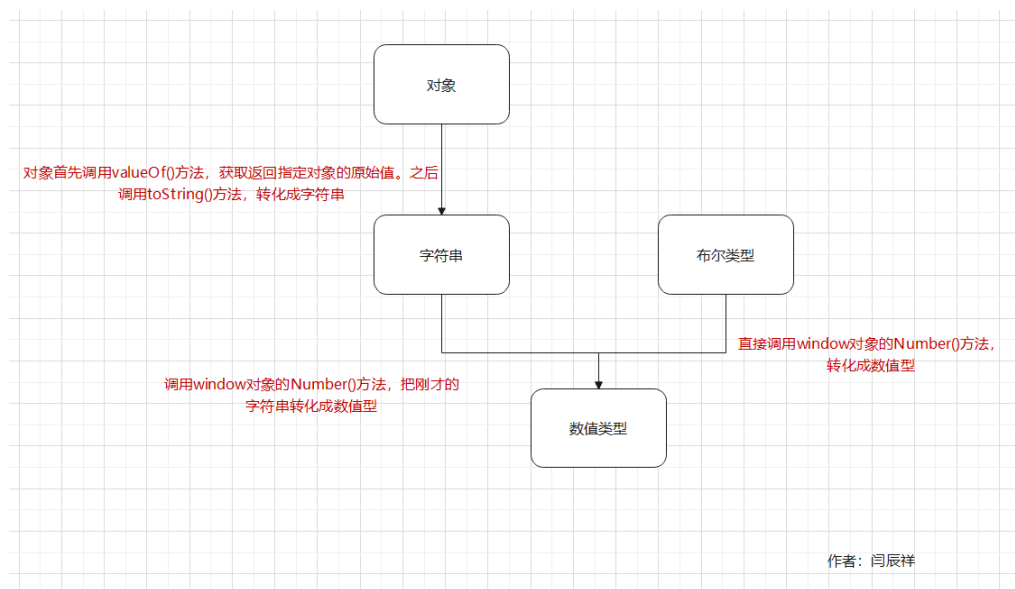
## 布尔

JS中只有5种数强转后会变成false。剩下的都是true。

```
1 | console.log(Boolean(0));  
2 | console.log(Boolean(undefined));  
3 | console.log(Boolean(null));  
4 | console.log(Boolean(NaN));  
5 | console.log(Boolean(''));
```

## 隐式类型转换

在JavaScript中，隐式类型转换实际上是隐式调用了显示类型转换的方法。数值型是转换的最后结果，具体过程如下：



```
1 //隐式转换的话就是
2 [1] == true
3
4 //首先转换成字符串
5 [1].valueOf().toString() === '1'
6 //然后转化成数值
7 Number([1].valueOf().toString()) === 1
8 //bool类型直接强转
9 Number(true) === 1
10 //最后
11 1 === 1
```

在转换的过程中，如果有一边出现了null或者undefined，则仅当两边都是null或者undefined之一时，才会判定true。

## 布尔

- `!!` 运算符可以快速隐式转型为bool

```
1 | !!114514 //true
```

- 只有以下几种类型转型为false

```
1 | !!undefined === false
2 | !!null === false
3 | !!NaN === false
4 | !!'' === false
5 | !!0 === false
6 | !!-0 === false
```

除此之外的数据都为true!!!

- 布尔值进行运算时，会转型为number

```
1 | 0 == false
2 | 1 == true
```

如果使用了`!`运算符，那么数据将会直接转换为bool类型。需要记住。

## 数值

- 进行四则运算的时候，会隐式转换成数值类型。

```
1 | console.log("12"-1); //11
2 | console.log("12"*1); //12, 这个可以用于快速地转换成数值。
```

- `number` 和 `bigint` 可以隐式转换。

```
1 | 10 == 10n
```

- 数值和`null`相加会变成数值本身，因为`null`会隐式转换成0。而和`undefined`相加会变成NaN。

```
1 | console.log(1+null) //1
2 | console.log(1+undefined) //NaN
```

- 对一个数据使用`+`运算符，会自动转型到`number`

```
1 | var str = "string"
2 | str = +str; //转型到number
3 | console.log(str)
```

## 字符串

```
1 | "123" == 123
```

同样的，模板字符串也可以转换

```
1 | var age = 10
2 | `${age}` == 10
```

特别的，空字符串将转换为0

```
1 | '' == 0 == false
```

字符串拼接会发生隐式转换，任何和字符串拼接的变量都会变成字符串。除了对象。



```
1 console.log("hello "+null)
2 console.log("hello "+undefined)
3 console.log("hello "+NaN)
4 console.log("hello "+false)
5 console.log("hello "+1)
6 console.log("hello "+{age:123})
7
8 /*
9 hello null
10 hello undefined
11 hello NaN
12 hello false
13 hello 1
14 hello [object Object]
15 */
```

## 数组

数组也是对象，所以数组第一步会转化成字符串。如果数组里面有字符串的话，会自动拼接。

```
1 [1,2,3] == '1,2,3'
2 ['name','age'] == 'name,age'
3 [1] == '1'
```

如果数组只有一个数值，那么转换成的字符串还能转换成数值型

```
1 [1] == '1' == 1
```

特别的，如果数组为空，将转换为空串

```
1 [] == '' == 0 == false
```

## 特殊转换

另外还有一些特殊类型的隐式转换

- NaN和任何数据都不相等，包括他自己。

```
1 NaN != NaN
```

- undefined 可以隐式转换为 null

```
1 undefined == null
```

## 包装类

## 基本类型与对象的转换

基本数据类型有时候也会转换成引用类型。

首先看这个例子。

```
1 var name = "もみじ";
2 console.log(name.length);
3 //3
```

这时你一定会奇怪，为什么我一个string还能有属性？而且还能正常打印出来？实际上这就是包装类。当你对一个基本数据类型进行属性方法等操作时，实际上内部会先 `new String(name)`。然后调用方法 `new String(name).length`。之后立刻销毁对象。

也就是说虽然看上去对一个数值类型进行了对象的操作，实际上确确实实是生成了一个临时对象。

要格外注意，临时对象使用完后立刻会销毁。

```
1 var name = "もみじ";
2 name.color = "紅色";
3 console.log(name.color);
4 //undefined
```

本例中，首先生成一个临时对象并且给这个临时对象赋予属性color，之后对象被销毁。之后再次生成一个临时对象，此时这个对象是没有color属性的，所以返回值为undefined。

## js中内置的包装对象

- String
- Number
- Boolean
- Object
- Function
- Array
- Date
- RegExp
- Error

## 数据类型的判断

### null与object

为什么typeof null是object?

实际上这是一个历史问题，在最早JavaScript版本中，数据有标记位和数据位组成。

000: object

001: integer

010: double

100: string

110: boolean

而null则是数据位和标记位都为0，因此会被判断成object

在ES6阶段，也有人提案修复这个bug，但是没有通过

## 语法基础

---

本文采用了最新的es6语法,可以放心食用.

## 专业术语

---

为了防止在文中反复解释一些术语,我将一些常用术语的解释放在了这里.这意味着你并不需要直接看这里.等之后遇到了这些术语再翻上来看才是正道.

## ECMAScript

ECMA本来是指欧洲计算机制造商协会，不过这个组织现在没事就喜欢制定一些标准，其中ECMAScript就是其中一个标准。ECMA-262这个标准的语言，一般被叫做JavaScript。但是实际上JavaScript是对ECMA进行了扩充。

node.js也是对ECMAScript标准的一个扩充。

实际上JavaScript包含三部分

- ECMAScript
- DOM
- BOM

## 字面量

字面量就是代码意义上的常量,说白了就是可以放到赋值号右边的都可以叫字面量.这样子这个赋值表达式的值就如字面上一样,你赋值号右边写的是啥,值就是啥,非常容易理解.

**注意,我下面代码中赋值语句的右边是字面量,我写这个语句只是方便理解,别理解错了.**

```
1 var 字符串字面量 = "hello world!";
2 var 数值字面量 = 996;
3 var 数组字面量 = ["java", "c++", "JavaScript"];
4 var 函数字面量 = function(){};
5 var 对象字面量 = {};
6 var 布尔字面量 = true;
```

## 函数与方法

理论上对象的函数就叫做方法,但是JavaScript是纯面向对象的语言,万物皆对象.所以函数和方法并没有C++那种半面向对象语言那种严格.一般来讲>window对象的方法叫做函数,其他对象的函数叫做方法,不过也没有那么较真就是了.

## 弱类型（动态类型）

JavaScript是一种动态类型的语言，这意味着一个变量可以存放任意类型的数据。这样可以提高程序的灵活性。但是也会降低代码的正确性。

## 注释

JavaScript提供了单行注释和多行注释

```
1 //单行注释
2
3 /*
4 多行注释
5 */
```

虽然JavaScript的标准没有提供文档注释，不过有些强大的编译器仍然可以识别并解析JavaScript的文档注释。

```
1 /**
2  * 文档注释
3  */
```

## 关于结尾的分号

JavaScript并没有强制要求你加上分号,也没有要求一定不加,一般情况下看自己喜好就行.

不过以`(`和`[开头的代码，前头必须加分号

```
1 function test(){
2     console.log("hello world")
3 }
4 test()
5
6 //1.立即执行函数前面加分号
7 ;(function(){
8     console.log("匿名函数前面要写分号")
9 })()
10
11 //2.数组前加分号
12 ;[1,2,3].forEach(element => {
13     console.log(element)
14 })
15
16 //3.模板字符串前面加分号
17 ;`test`.toString()
```

如果代码前是一个函数调用，那么编译器就会把前面的小括号和后面的解析到一起，所以必须加分号。

## 双引号与单引号

JavaScript不区分单引号和双引号,字符串可以随意用这两种引号,不像其他语言严格区分大小写。

不过ESlint标准，要求字符串必须是单引号。

## 输入与输出

输出有两种常用方式

```
1 alert("hello world");//弹窗
2 console.log("Hello world");//控制台输出
```

输入可以使用prompt方法，这个会直接弹出一个弹窗，也就是说需要浏览器环境

```
1 var age = prompt("输入年龄")
2 console.log(age)
```

## 严格模式和非严格模式

在脚本或函数开头写下这句声明就可以开启严格模式，之后会添加很多限制，比如不允许使用未声明的变量什么的。

```
1 'use strict';
```

**注意：这句代码只允许写在脚本或者函数的开头！**

## let与const [es6]

## Hello World

浏览器环境下，直接打开浏览器，F12找到console就能看到。

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title></title>
6     <script type="text/javascript">
7       console.log("Hello world");
8     </script>
9   </head>
10  <body>
11  </body>
12 </html>
13
```

node环境下，用node命令执行语句。

```
1 | console.log("Hello World");
```

## 运算符

### 除法

跟其他的语言不太一样，JavaScript的除法是保留小数的。

```
1 | 3/2==1.5
```

如果想只取整数，可以使用数学函数来截取。

```
1 | Math.floor(3/2) == 1
```

### 等于与严格等于

等于运算符 `==` 允许JavaScript的解释器进行隐式转换，而严格等于不允许。

```
1 | console.log("520"==520)//true
2 | console.log("520"===520)//false
```

### 不等和严格不等

同样的，`==` 允许隐式转型，而 `!==` 不允许。

```
1 | console.log("520"!=520)//false
2 | console.log("520"!==520)//true
```

## void

void在其他语言一般都用于代表方法的返回值类型为空，但是在JavaScript里面却是一种运算符。void运算符执行的表达式，返回值永远是undefined。

```
1 | console.log(void (3==2));//undefined
2 | void (console.log("hello world"));//执行了方法，但是返回值永远是
   | undefined
```

void经常用于HTML中a标签。用来表示点击不发生跳转。

```
1 | <a href="javascript:void(0)">单击此处什么也不会发生</a>
```

除此之外，`void`还用于箭头函数，防止内存泄漏。箭头函数标准中，允许在函数体不使用括号来直接返回值。如果右侧调用了原本没有返回值的函数，其返回值改变后，则会导致非预期的副作用。安全起见，当函数返回值是一个不会被使用到的时候，应该使用 `void` 运算符，来确保返回 `undefined`（如un下方示例）。这样，当 API 改变时，并不会影响箭头函数的行为。

```
1 button.onclick = () => void doSomething();
```

## !!

!!可以把其他类型的数据快速转成`bool`类型。因为一个!就是把数据转化为`bool`并且取反，而两个就是把数据取反后再变回来，相当于直接把数据转换成了`bool`类型。

```
1 //下面都是false
2 !!undefined
3 !!null
4 !!0
5 !!false
6
7 //下面都是true
8 !!true
9 !!"1"
10 !!1,
11 !!{}
12 !![]
```

## instanceof

我们可以用 `instanceof` 运算符来判断两个对象间的关系。如果 `A instanceof B == true` 代表A是B的实例对象。

```
1 Student instanceof Object == true
```

## typeof

可以来查看数据类型。

```
1 var a = 1;
2 console.log(typeof(a));
```

# 字符串

## 创建

```
1 //用字面量创建
2 var dream = "愿天下没有996!";
```

## 模板字符串 [es6]

模板字符串是超级无敌强化之后的字符串,用**间隔符**(ESC下面那个符号)来引用.

下面简述它的两个主要功能

1. 可以在字符串内直接使用变量,并计算
2. 可以在字符串内保留回车,不用自己拼接回车了.

当一个字符串拼接的过长时,原来那种+号的写法过于繁琐,所以可以使用模板字符串,直接把变量放在里面

```
1 //注意,return中的是间隔符,并不是引号.
2 function KillerQueen(name,age){
3     return `我叫${name},今年${age}岁`
4 }
5 console.log(KillerQueen("吉良吉影",33));
```

不仅仅是这样,模板字符串里面的变量也可以进行运算,并且保留回车.

```
1 function Real_GDP_per_capita(money){
2     return `
3         赵家有钱${money}万,隔壁9个穷光蛋;
4         平均起来算一算,各个都是赵${money/(9+1)}万`;
5 }
6 console.log(Real_GDP_per_capita(1000));
```

## 字符串的解构

可以把字符串结构成一个数组,每一个数组的元素代表字符串的一位。

```
1 var string = "123456789"
2 var [a,b,c] = string
3 console.log(a,b,c)
4 //输出:
5 //1 2 3
```

## 数组

### 数组与对象

在JavaScript中,数组是对象的一个子类。实际上数组就是一个特殊的对象,数组的key就是其下标,value就是数组元素。

注意数组和一般的对象有一个不一样的地方,就是它会把 `a['3']=1` 自动转型成 `a[3]=1`。



```
1 var a = [];  
2 a['3'] = 4;  
3 console.log(a)  
4 //[ <3 empty items>, 4 ]  
5  
6 var b = {};  
7 b['3'] = 4  
8 console.log(b)  
9 //{ '3': 4 }
```

## 数组的创建

### 1. 使用字面量

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

### 2. new一个

```
1 var a = new Array(1);//创建长度为1的数组  
2 var a = new Array(1,2);//创建一个数组，其元素为[1,2]
```

## 数组的解构 [es6]

指的是可以把数组里的内容一次批量导出并赋值。

```
1 var lol1=["86","波莱特","日日姬"]  
2 var [lol11,lol12,lol13]=lol1  
3 console.log(lol11,lol12,lol13)  
4 //输出:  
5 //86 波莱特 日日姬
```

## 数组元素增加

### 手动改length

修改完之后，末尾会出现几个空的元素。默认用empty填充。

```
1 var a = [1,2,3];  
2 a.length=5;  
3 console.log(a);  
4 //[ 1, 2, 3, <2 empty items> ]
```

## 用下标动态添加

```
1 var a = [1,2,3];
2 a[3]=4;
3 console.log(a);
4 //[ 1, 2, 3, 4 ]
```

同样的，如果修改的下标过大，中间就会有很多地方元素为empty。

```
1 var a = [1,2,3];
2 a[10]=4;
3 console.log(a);
4 //[ 1, 2, 3, <7 empty items>, 4 ]
```

## push

在数组尾部添加元素。

```
1 var a = [1,2,3];
2 a.push(4)
3 console.log(a);
4 //[ 1, 2, 3, 4 ]
```

## unshift

在数组最头头添加元素。

```
1 var a = [1,2,3];
2 a.unshift(4)
3 console.log(a);
4 //[ 4, 1, 2, 3 ]
```

## 数组元素的删除

### delete

```
1 var arr = [1, 2, 3, 4];
2 delete arr[0];
3
4 console.log(arr);    //[undefined, 2, 3, 4]
```

可以看出来，delete删除之后数组长度不变，只是被删除元素被置为undefined了。

## pop

删除最后一项

```
1 var colors = ["red", "blue", "grey"];
2 var item = colors.pop();
3 console.log(item);           //"grey"
4 console.log(colors.length);  //2
```

## shift

删除第一项

```
1 var colors = ["red", "blue", "grey"];
2 var item = colors.shift();
3 console.log(item);           //"red"
4 console.log(colors.length);  //2
```

## 数组的遍历

### forEach

forEach可以使用**回调函数**来操作数组的数据。回调函数里面的参数，就是数组里面的元素。

```
1 [1,2,3,4,5].forEach(
2   function(num){
3     console.log(num)
4   }
5 );
6 //1
7 //2
8 //3
9 //4
10 //5
```

## 判断数组相等

```
1 JSON.stringify(data1) == JSON.stringify(data2)
```

## 函数

### 简介

JavaScript的函数就是对象,万物皆对象.但是JavaScript是弱数据类型的,所以在使用函数的时候和别的强数据类型语言还是有差别.

比如说,函数不用写返回值类型,没有返回值也不用 `return void;` ,并且定义时需要显式定义.

## 函数的创建方式

### 赋值表达式创建

这种表达式的右边是一个匿名函数。

```
1 var print = function(){
2   console.log("hello world!")
3 }
4 print()
```

### 直接创建

```
1 function print(){
2   console.log("hello world!")
3 }
4 print()
```

### 用Function构造函数创建

不推荐使用。构造函数需要用Function来创建函数，Function里面的参数就是函数体，需要用引号括起来，但是里面没有提示，非常不好用。

```
1 var print=new Function(
2   `console.log("hello world!") `
3 )
4 print()
```

另外用构造函数创建函数时，`new Function()` 的new可以省略。

### 箭头函数 [es6]

这个东西看着复杂,实际上就是一个简化版匿名函数,

我们可以用匿名函数来参考着看.

```
1 var fun1 = function (id){
2   console.log("匿名函数"+id);
3 }
4 var fun2 = (id) => {
5   console.log("箭头函数"+id);
6 }
7 fun1(0);
8 fun2(1);
```

箭头函数还可以进一步简写：

- 若函数只有一个参数，可以省略小括号，但是没有参数的话，必须加一个空括号()。

- 函数只有一行代码的话，可以省略大括号。**注意！省略大括号的时候，那么编译器默认会给前面加上return，返回该语句。**

```
1 var fun1 = function(){1+1}
2
3 var fun2 = ()=> 1+1 //箭头函数省略了大括号会默认有返回值。
4
5 console.log(fun1(),fun2())//undefined 2
```

为了出现不必要的bug，可以使用**void运算符**来强行让返回值为undefined。

```
1 var fun2 = ()=> void (1+1)
2 console.log(fun2())//undefined
```

## 函数参数的特性

### 函数参数匹配

JavaScript语法非常自由，允许调用参数时，参数不用匹配。

```
1 function add(a, b) {
2   return a + b;
3 }
4
5 console.log(add(1, 2)); //3
6 console.log(add(1)); //NaN
7 console.log(add(1, 2, 3)); //3
```

可以看到，如果参数传的太多，那么JS不会去接受多出来的实参。如果太少，那么就把没有赋值的形参默认为undefined。所以 `1+undefined===NaN`。

### arguments

JS的函数默认有一个重要的参数，那就是arguments。在函数调用的时候，**所有的实参都会传到arguments里面**。这个arguments对外隐藏，跟this一样。

```
1 function show() {
2   console.log(arguments)
3 }
4 show("萝莉",12);
5 //["arguments"] { '0': '萝莉', '1': 12 }
```

可以看到arguments把实参全都接收了，而且数据的格式很像一个数组。**其实arguments就是一种伪数组**。它可以通过下标进行访问，并且拥有length属性。但是没有数组的方法。

## ...rest和...spread

rest和spread就像是反函数和函数。rest的意思的剩余的，spread则是展开、摊开的意思。这两个语法都会用到**扩展运算符**，这种特殊的运算符。

在函数传参的时候，有时候函数的参数数量是不确定的，这时候就可以使用rest参数。rest参数会统一获取剩下所有参数。

```
1 function Lolis(loli1,loli2,...lolis){
2     console.log(loli1)
3     console.log(loli2)
4     console.log(lolis)
5 }
6 Lolis("雷姆","伊莉雅","波莱特","巧克力");
7
8
9 //输出
10 雷姆
11 伊莉雅
12 ['波莱特', '巧克力']
```

可以看到，其实rest参数会把传进来的所有参数封装成数组。相当于收尾工作，所以这个参数只能写在最后面。

spread和rest刚好相反，rest是来打包参数的，而spread可以把一个数组拆开成为一堆零散的数据。

```
1 var lolis=["雷姆","伊莉雅","波莱特","巧克力"]
2
3 function Lolis(loli1,loli2,loli3){
4     console.log(loli1)
5     console.log(loli2)
6     console.log(loli3)
7 }
8 Lolis(...lolis,"小丛雨")
9
10 //输出
11 雷姆
12 伊莉雅
13 波莱特
```

这时候，函数接受了几个就能用几个，多余的参数只能去arguments里面找。

而且spread仅仅是解构数组，所以任何位置都可以写。

## 默认参数(ES5)

在ES5中，我们可以通过以下方式模拟默认值。

```
1 function test(a,b){
2     a=a||1
3     b=b||2
4     console.log(a,b)
5 }
6
7 test()//1, 2
```

但是这个写法也是有着bug存在的。

```
1 function test(a,b){
2     a=a||1
3     b=b||2
4     console.log(a,b)
5 }
6
7 test(0,0)//1, 2
```

我现在需要给a,b传入0, 0的值。但是0被隐式转换为了false。最终还是变成了默认值。

我们不妨做出以下改进。

```
1 function test(a, b) {
2     a = (typeof a !== "undefined") ? a : 1
3     b = (typeof b !== "undefined") ? a : 1
4     console.log(a, b)
5 }
6
7 test(0, 0)
```

## 默认参数 (ES6)

ES6中，可以直接给参数赋值。

```
1 function test(a = 1, b = 2) {
2     console.log(a, b)
3 }
4
5 test() //1, 2
```

并且和java不一样，js的默认参数之后可以跟普通参数。调用时，如果想使用默认参数，需要写undefined。

```
1 function test(a = 1, b) {
2     console.log(a, b)
3 }
4
5 test(undefined,0) //1, 0
```

## 默认参数对arguments的影响

```
1 function test(a, b=2) {
2     console.log(arguments.length)//1
3     console.log(a===arguments[0])//true
4     console.log(b===arguments[1])//false
5
6
7     a=3
8     b=4
9
10    console.log(a===arguments[0])//false
11    console.log(b===arguments[1])//false
12 }
13
14 test(1)
```

## 数组添加元素

1. `Array.push()`
2. `Array.unshift()`
3. `Array.length`

## 函数返回值

如果你没有自己写的话，默认为undefined。

## 立即执行函数

把一个匿名函数前后都用小括号括起来就变成了立即执行函数,这种函数只能用一次,一旦程序运行到这里立刻执行.

因为没有名字,之后无法被其他程序调用.

```
1 (function(){
2     document.write("我是匿名函数")
3 })();
```



## 函数的length

函数的length就是其参数的个数。几个参数length就是几。

```
1 function test(a,b,c){}
2 console.log(test.length)//3
```

## 回调函数

回调函数指的是那些由我们自己定义，但是由系统自动调用的函数。

```
1 function callBack(fun){
2     fun()
3 }
4 callBack(function(){
5     console.log("hello world")
6 })
```

上面这个例子中，我们定义了一个匿名函数，但是这个函数自动被callBack函数调用执行了，这种被自动调用的函数就是回调函数。

## eval函数

eval简直太强大了，js的这个函数，可以让你输入一个字符串，并且直接运行你字符串里面的代码。

```
1 function DIY(str){
2     eval(str);
3 }
4 DIY("console.log('hello world!')");
5 //hello world!
```

## name属性

```
1 function test1() {
2
3 }
4 var test2 = function (params) {
5
6 }
7
8 var obj = {
9     get test3(){
10         return ""
11     }
12 }
13
14 console.log(test1.name)//test1
```

```
15 console.log(test2.name)//tes2
16 console.log(test1.bind().name)//bound test1
17 console.log((new Function()).name)//anonymous
```

## apply, call和bind方法

bind方法就是把所执行的函数做了一个打包，而不是直接运行。

```
1 function show(){
2   console.log(this)
3 }
4 var obj={name:"我是obj啦"}
5 show.bind(obj)()
6
7 var show2 = show.bind(obj)
8 show2()
9 //输出结果都是
10 //{ name: '我是obj啦' }
```

可以看到，实际上bind就是把 `obj.show()` 这个函数进行了一个保存，不需要每次都去apply和call了，直接可以用一个新的函数去执行。

apply最多可以接收两个参数：this对象和参数数组

call则可以接受多个参数：this对象，和一连串参数。

## [[Call]]和[[Construct]]

js函数的内部有两个不同的方法。当通过new关键字调用的时候，执行[[Construct]]方法。如果没有new则调用[[Call]]方法。注意，不是所有的函数都有[[Construct]]方法，比如箭头函数就没有。

通过这两个属性，我们就可以判断一个函数是否是被new出来。

```
1 function maigo(namae) {
2   if(typeof new.target !== "undefined"){
3     this.namae = namae
4     console.log(new.target)//[Function: maigo]
5   }else{
6     console.log("你哈呐~如果想要和我在一起，请来new我哦~")
7   }
8 }
9 maigo()//当成普通函数使用
10 var maigo1 = new maigo("迷子")
11 console.log(maigo1.namae)
```

其中如果你使用new来创建对象，new.target就会被赋值为其new的构造函数。如果没有用new，那么就会赋值为undefined。

这个new.target被称为元属性。元属性是指非对象的属性。new就不是一个对象，但是它仍然拥有target属性。

# 块级函数

在ES5中, 如果在

## 高阶函数

如果一个函数的参数或者返回值也是一个函数, 那么这就是一个高阶函数。

最有代表性的高阶函数是数组的 `filter`、`map`和`reduce`

```
1  const nums = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024];
2  //filter用来过滤想要的数
3  //filter函数会调用一个回调函数
4  //回调函数的参数n会遍历nums数组的元素
5  //回调函数必须返回一个boolean值,
6  //若为true则把数据压入新数组,false则过滤掉
7  var newNums = nums.filter(
8    function (item) {
9      if (item > 100)
10         return true;
11    }
12  );
13  console.log("newNums:"+newNums);
14
15  //map用来对原数据进行一次映射
16  //对数组进行一次映射,遍历数组每一项,把数据存入n中,
17  //return的值存入新的数组
18  var newNums2 = newNums.map(
19    function (item) {
20      return item * 10
21    }
22  );
23  console.log("newNums2:"+newNums2);
24
25  //用于集合中的所有数据,可以进行如全部加一遍的操作
26  //reduce一共传参两个,第一个是回调函数,第二个是给oldValue初始化的值
27  //回调函数中,oldValue除第一次外,每次值被赋值为函数return的值
28  //item依旧遍历之前的数组
29  var newNums3 = newNums2.reduce(function (oldValue, item) {
30    return oldValue + item;
31  }, 0);
32  console.log("newNums3:"+newNums3);
33
34  var totle = nums.filter(function (item) {
35    if (item > 100)
36      return true;
37  }
38    ).map(function (item) {
39      return item * 10;
40    }
41    ).reduce(function (oldValue, item)
42    {
43      return oldValue+item;
44    }, 0);
44  console.log("totle:"+totle);
45  //箭头函数写法
```

```
46 | var
    |   total2=nums.filter(item=>item>100).map(item=>item*10).reduce((pre,item)=>item+pre);
47 | console.log("total2:"+total2);
```

# 对象

## 万物皆对象

JavaScript只有对象，甚至没有类的概念，可以说是真真正正面向对象的语言。但是很快大家就发现了，虽然说着没有类，但是没有又不行，只能用各种方法间接实现。最终在ES6中引入了类的概念。

记得我在开头说过的吗？JavaScript中不区分函数和方法，因为所有的一切都是对象，方法也是函数对象。

## 对象的分类

## 对象的创建方式

### 字面量创建

```
1 | var girl={
2 |         name: '雷姆',
3 |         age: 18,
4 |         loveStyle: function(){
5 |             console.log("最喜欢你啦！抱抱！")
6 |         }
7 | }
```

### 用Object创建

一般不怎么用这种

```
1 | var girl = new Object();
2 | girl.name = '雷姆';
3 | girl.age = 18;
4 | girl.loveStyle = function() {
5 |     console.log("最喜欢你啦！抱抱！")
6 | }
```

## 构造函数

构造函数用this来添加属性，其实原理和Object一样的。首先new的时候，this就指向了new出来的内存空间。之后给内存赋值，最后把这个内存起始地址返回给LeiMu，这样就通过this，间接完成了属性的添加。

```
1 function Girls(){
2   this.name = '雷姆';
3   this.age = 18;
4   this.loveStyle = function() {
5     console.log("最喜欢你啦! 抱抱!")
6   }
7 }
8 var LeiMu = new Girls();
```

## 工厂模式

一种设计模式，这样写出来不用加new

```
1 function createGirls(name, age) {
2   var o = new Object();
3   o.name = name;
4   o.age = age;
5   o.loveStyle = function () {
6     console.log("最喜欢你啦! 抱抱!")
7   }
8   return o;
9 }
10 var congYU = createGirls("小丛雨", 513);
11 congYU.loveStyle()
```

## 实例化类

ES6特有。

```
1 class Lolli
2 {
3   constructor(age){
4     this.age = age || 14;
5   }
6 }
7 var lolli = new Lolli();
8 console.log(lolli.age)
```

## 对象属性和方法简写 [es6]

如果在对象中的参数已经在外界被定义过了，那么可以直接写变量名，而不用再去写键值对。

```

1  var a=16
2  var b=2
3  var show={()=>{
4      console.log("hello world")
5  }}
6  var obj={
7      a,b,show
8  }
9  obj.show()
10 //输出
11 //hello world

```

如果不想用这个变量名，可以按照原本的语法来写。

```

1  var a=12
2  var b=2
3  var show={()=>{
4      console.log("hello world")
5  }}
6  var obj={
7      age:a,
8      b,show
9  }
10 obj.show()
11 console.log(obj.age)
12 //输出
13 //hello world
14 //12

```

而方法简写

## 对象的解构 [es6]

指的是可以把对象的键批量导出的一种语法。在export中用得很多。

```

1  var lolli={
2      name:"小可爱86酱",
3      age:70
4  }
5  var {name,age}=lolli
6  console.log(name,age)
7  //输出
8  //小可爱86酱 70

```

其实这里面就用到了属性的简写，实际上的代码应该是这样。

```

1  var loli={
2      name:"小可爱86酱",
3      age:70
4  }
5  var {name:name,age:age}=loli
6  console.log(name,age)
7  //输出
8  //小可爱86酱 70

```

首先var了一个对象，这个对象有两个属性，也就是左边的name和age，而这两个属性又解构的loli被传入了两个参数，分别是右边的name和age。

如果不是很理解可以看这个代码：

```

1  var loli={
2      name:"小可爱86酱",
3      age:70
4  }
5  var {name:MYname,age:MYage}=loli
6  console.log(MYname,MYage)
7  //输出
8  //小可爱86酱 70

```

也就是说右边的值是自定义的，也是需要被传入数据的。

## 对象的展开

对象的展开就是指把一个对象展开成一个个的键值对，一般用于把两个对象直接合并。看一下例子就明白了。

```

1  var c = { hasPointer: true }
2  var java = { isOOP: true }
3  var CPP = {...c,...java}
4  console.log(CPP)
5  //结果
6  //{ hasPointer: true, isOOP: true }

```

可以看到，CPP通过 `...` 的语法，可以把c和java的属性直接解析出来并且把结果传入CPP对象中。

如果不用这种展开的语法会怎么样呢？

```

1  var c = { hasPointer: true }
2  var java = { isOOP: true }
3  var CPP = { c, java }
4  console.log(CPP)
5  //结果
6  //{ c: { hasPointer: true }, java: { isOOP: true } }

```

可以看到，这个其实直接把指针传进去了，而不是传的值。

这种语法其实可以理解作为一种多继承，在本例中C++继承了c的指针和java面向对象的思想。所以以后使用多继承可以考虑这种对象的展开语法。

## 对象的数据的访问与添加

对象的数据可以用点运算符来访问，也可以用下标运算符来指定访问的key。

```
1 var loli = {
2   name:"小丛雨",
3   age:13
4 }
5 console.log(loli.name); //小丛雨
6 console.log(loli["age"]); //13
```

也可以用下标运算符修改指定key的value。如果这个key不存在，那么对象会自动添加这个键值对。

```
1 var obj = {}
2 obj["name"]="loli"
3 console.log(obj)
4 //name: "loli"
```

也可以用成员运算符来添加key

```
1 var obj = {}
2 obj.name="loli"
3 console.log(obj)
4 //name: "loli"
```

`.name` 的语法被叫做**属性访问**，`["name"]` 的语法被叫做**键访问**。

## with

with可以用来批量修改对象的属性或方法

使用with前：

```
1 var loli = {
2   name:"小丛雨",
3   age:13
4 }
5 loli.name="宁宁";
6 loli.age = 15;
7 console.log(loli)
8 //{name: "宁宁", age: 15}
```

使用with后：



```

1  var loli = {
2      name:"小丛雨",
3      age:13
4  }
5  with(loli){
6      name="宁宁",
7      age=15
8  }
9  console.log(loli)
10 //{name: "宁宁", age: 15}

```

但是不推荐使用with，因为with可能会导致很多bug。比如下面这个

```

1  var loli = {
2      name:"小丛雨",
3      age:13
4  }
5  var book = {
6      price:30
7  }
8
9  function change (obj){
10     with(obj){
11         name="宁宁",
12         age=15
13     }
14 }
15
16 change(loli)
17 change(book)
18
19 console.log(loli)//{name: "宁宁", age: 15}
20 console.log(book)//{price: 30}

```

可以看到，我明明把loli和book对象传了进去，但是最后只有loli对象正常输出了，这是为什么？其实当book对象进入with代码块之后，发现直接对一个未声明的属性赋值，按照编译器的规定，**一切未声明的属性全都归属window对象**。所以这个name和age实际上被添加到了window对象上面去。

```

1  //a没有用var声明，默认添加到了window对象上
2  a=13
3  console.log(a)

```

## 删除对象属性

```

1  delete obj.a

```

## get与set

跟c#的一模一样，很普通的语法。

```
1  var rennge = {
2      _age : 14,
3      get age(){
4          return this._age
5      },
6
7      set age(num){
8          this._age = num
9      }
10
11 }
12
13 console.log(rennge.age)
14 rennge.age=15
15 console.log(rennge.age)
```

## 属性描述符

### 获取与修改

其实JavaScript中，每个对象的属性都有很多属性，包括是否只读，是否可枚举等。

```
1  //查看某个属性
2  var obj = {a:1}
3  var result = Object.getOwnPropertyDescriptor(obj,"a");
4  console.log(result)
5  //{ value: 1, writable: true, enumerable: true, configurable:
   true }
```

```
1  //添加或修改属性
2  var obj = {a:1}
3  Object.defineProperty(obj,"a",{
4      value:2,
5      writable:true,
6      configurable:true,
7      enumerable:true
8  });
```

## 数据描述符

### writable

writable决定了该属性是否可以被修改，如果设置为false，那么该属性就无法被修改。

```

1  var obj = {a:1}
2  Object.defineProperty(obj,"a",{
3      writable:false,
4  });
5
6  obj.a = 2;
7  console.log(obj.a) //1

```

可以看到，虽然代码没有报错，但是该值没有修改成功。

如果是在严格模式下，代码会报错。

## configurable

这个玩意决定了你是否能使用 `defineProperty` 函数，如果为 `false`，那么对该属性使用 `defineProperty` 会不起效果。而且，如果你再写 `configurable:true` 会直接报错。也就是说如果把 `configurable:false` 设置了之后，那么操作不可逆，不能再设置回来。

但是这个操作还有两个特殊的地方

1. 即使把属性设置了 `configurable:false`，我们还是可以把 `writable` 设置为 `false`，但是无法设置为 `true`。

```

1  var obj = {a:1}
2  //把属性设置为不可修改配置
3  Object.defineProperty(obj,"a",{
4      writable:true,
5      configurable:false
6  });
7
8  //但是仍然可以把writable设置为false
9  Object.defineProperty(obj,"a",{
10     writable:false,
11 });
12
13 //可以看到属性赋值失败了，因为该属性是只读的
14 obj.a = 2;
15 console.log(obj.a) //1

```

这个属性也决定了该属性是否可以被 `delete` 删除

## enumerable

可否枚举，也就是说在 `for.....in` 的时候，是否可以被打印出来。在循环与迭代一章会详细讲解。

## 访问描述符

set与get

如果给某个对象定义了 `get` 或者 `set` 属性，那么这个属性就会变成**访问描述符**，JavaScript 会忽略它们的 `value` 和 `writable` 属性。

有两种方法可以来设定getter和setter。

```
1  var girl = {
2      _name_: "蓮華",
3      //可以直接在对象内部进行getter和setter的设定
4      get name() {
5          return this._name_;
6      },
7      set name(newName) {
8          //no!!!!!!
9          //永远喜欢莲华!
10     }
11 }
12 //也可以通过外界属性描述符设定
13 Object.defineProperty(girl, "age", {
14     get: function () {
15         return this._age_;
16     },
17     set: function (age) {
18         if (age < 25 && age > 8)
19             this._age_ = age;
20         else
21             this._age_ = 14;
22     }
23 })
24
25
26 console.log(girl.age);
27 girl.age = 5;
28 console.log(girl.age);
```

可以看到，实际上getter和setter都是通过创建一个新的变量来完成对私有数据的封装的，但是实际上这个变量并不是私有的，还是可以通过 `girl._age_` 和 `girl._name_` 来进行访问的。

但是这样子心理上似乎能感觉好一点，至少大部分人都知道下划线开头的变量是私有变量，不应该随意访问。

## 类 ES5

### 创建

虽然我很不喜欢用ES5，但是RPGMakerMV，人家用的是ES5的语法，而且一眼望过去，成片成片的原型链，各种骚操作。太难了。

话不多说，ES5没有直接的类继承什么的，必须依靠显示原型来间接完成类的继承与创建。

```
1  //构造函数
2  function Girl(){
3      //在构造之后，会调用原型的初始化函数
4      this.initialize.apply(this,arguments)
```

```

5   }
6
7   //将构造函数指向其对象
8   Girl.prototype.constructor=Girl;
9   //初始化函数
10  Girl.prototype.initialize=function(){
11      this._age=14;//私有变量
12  }
13
14  Girl.prototype.play = function(){
15      console.log("ㄋ~~~~~")
16  }
17
18  var girl = new Girl();
19  girl.play();
20

```

## 复制继承并重写父类的某一个方法

```

1  var newFun = Father.prototype.fun;
2  Father.prototype.fun = function() {
3      newFun.call(this);
4      //代码
5  };

```

## 继承

```

1  function Scene_Base() {
2      this.initialize.apply(this, arguments);
3  }
4
5  Scene_Base.prototype = Object.create(Stage.prototype);
6  Scene_Base.prototype.constructor = Scene_Base;
7
8  Scene_Base.prototype.initialize = function() {
9  };
10

```

## 静态类

核心的设计思想就是直接在构造函数的对象上面添加属性和方法。

```

1 function DataManager() {
2     throw new Error('This is a static class');
3 }
4
5 DataManager._globalId = 'RPGMV';
6
7 DataManager.isItem = function(item) {
8     return item && $dataItems.contains(item);
9 };

```

## 类 ES6

### 特性

- ES5的构造函数可以提升，但是class不能提升。在执行声明语句之前，一直在临时死区里面。
- 类中的代码自动运行在严格模式下，且无法强行脱离。
- 类中，所有的方法都是不可枚举的。
- 用new以外的方法调用构造函数会报错。
- 在类中，无法修改类名。

### 类的创建方式

ES6中，可以用类来创建对象，因为ES6真的很香，跟java他们用法差不多，现在你就能发现ES6有多香了。ES5学起来简直难受的一批，浑身不舒服。

```

1 class Lolli {
2     constructor(age) {
3         this.age = age;
4     }
5     gu_lu_gu_lu()
6     {
7         console.log("咕噜咕噜~~~");
8     }
9     showAge(){
10         console.log(`人家今年${this.age}岁`);
11     }
12     //静态方法
13     static ne()
14     {
15         console.log("呐呐呐呐呐呐呐呐呐呐呐呐呐呐");
16     }
17 }
18 var 巧克力 = new Lolli(14);
19 巧克力.gu_lu_gu_lu();
20 巧克力.showAge();
21 Lolli.ne();

```

可以看到,程序正常运行:

咕噜咕噜~~~  
人家今年14岁  
呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐

## 类的继承

用extends表示继承哪个类,之后用super来指定继承的属性,默认继承全部的方法.

```
1 class 天然呆萝莉 extends Lolli {
2     constructor(age,name) {
3         //用super来直接继承属性
4         super(age);
5         this.name=name;
6     }
7     //方法默认全部继承过来,无论是普通方法,还是静态方法.
8     showInfo(){
9         console.log(`嘟嘟噜~~,${this.name}です,今年${this.age}岁`)
10    }
11 }
12 var 椎名真由理 = new 天然呆萝莉(14,"椎名真由理");
13 椎名真由理.showInfo();
14 椎名真由理.gulu_gulu();
15 天然呆萝莉.ne();
```

可以看到程序正常运行:

嘟嘟噜~~,椎名真由理です,今年14岁  
咕噜咕噜~~~  
呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐呐

## 重载

JavaScript没有重载,只有重写。

## 实例方法和实例属性

```
1 class A{
2     //实例属性直接写就行,只能通过对象调用
3     message = "hello world"
4
5     show(){
6         console.log(this.message)
7     }
8 }
9 var a = new A()
10 a.show()
```

## 静态方法和静态属性

在前面加上static就行了,只能通过类来调用。

```

1  class A{
2      static message = "hello world"
3
4      static show(){
5          console.log(this.message)
6      }
7  }
8  A.show()
9  console.log(A.message)
10
11 // var a = new A()
12 // a.show()
13 //这两个会报错，JavaScript中静态属性只能由类来访问。

```

## 静态类

```

1  /**定义静态类**/
2  var Core = {};
3  Core.StaticClass = (function(){
4      var Return = {
5          Property: "Test Static Property",    //公有属性
6          Method: function(){    //公有方法
7              alert(_Field);    //调用私用字段
8              privateMethod();    //调用私用方法
9          }
10     };    //定义返回的公有对象
11
12     var _Field = "Test Static Field";    //私有字段
13     var privateMethod = function(){    //私有方法
14         alert(Return.Property);    //调用属性
15     }
16
17     return Return;    //生成公有静态元素
18 })();

```

## 词法作用域和执行上下文

### 编译基本过程

V8引擎在处理js代码的时候需要用到3个组件：

1. 解析器 (parser)：用于把js代码解析成抽象语法树AST。
2. 解释器 (interpreter)：将AST解释成字节码 (bytecode)
3. 编译器 (compiler)：将字节码转换成机器码



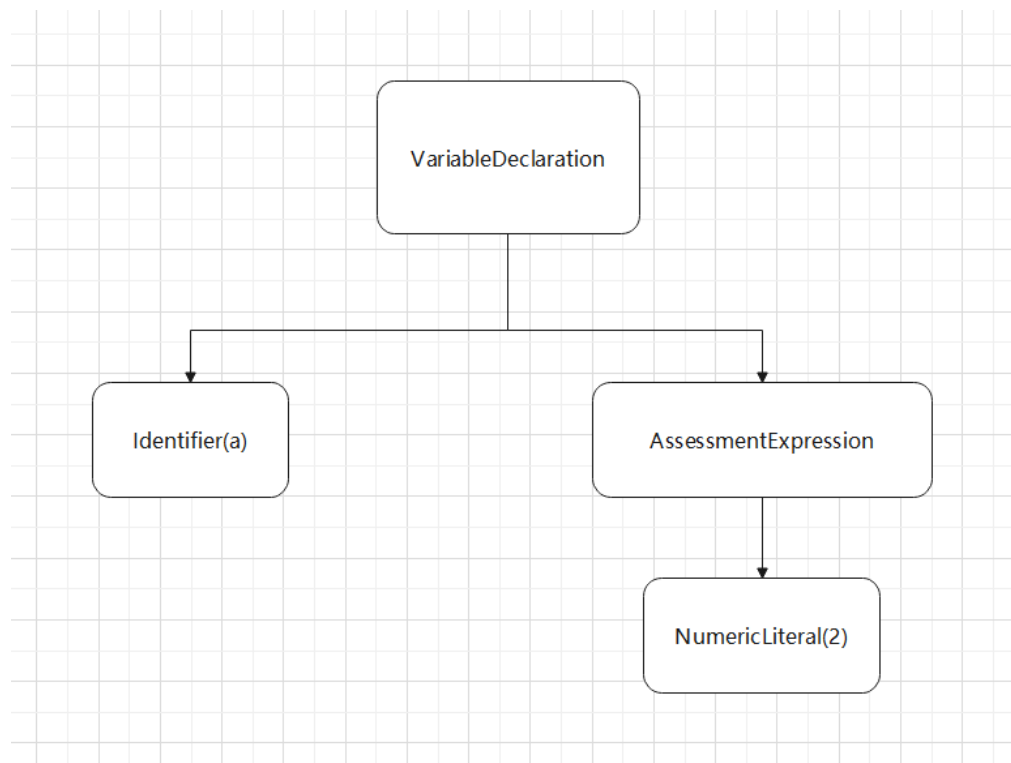
## 分词/词法分析

在这个过程中，解释器会把代码分成一个一个有实际意义的代码块，也就是所谓的**词法单元**。

比如说 `var a = 2;` 就会变成 `var` `a` `=` `2` 4个单独的词法单元。

## 语法分析

这个阶段，会把刚才的词法单元转化成一棵**抽象语法树**。



## 解释执行

将高级语言转化为机器语言

## LHS和RHS

这个就是左值和右值的关系，其他语言虽然也有，但是在编译原理这里意思略有不同。

LHS代表左值查询，例如 `a=1`，这种带赋值语句的表达式，就会调用左值查询，因为这种赋值语句会实实在在更改内存数据，所以左值查询实际上是去查找`a`所在内存。

RHS虽说是右值查询，实际上代表的是所有非左值查询，比如 `console.log(a)` `function f(a){}`。右值查询需要去查找`a`这个变量的值究竟是多少，而不关心内存什么的，他的任务就是查到`a`的值多少。

比如下面这个代码

```
1 function f(a){
2   var b=a;
3   return a+b;
4 }
5 var c = f(2);
```

这个代码一共包括3个LHS和4个RHS

1. `f(2)` 执行, 进行实参传递, 隐式赋值 `a=2`, 执行1次LHS
2. `var b=a`, 首先要对a进行RHS, 找出其值。之后执行赋值语句LHS。此时执行2次LHS, 1次RHS
3. `return a+b` 首先要去查找a和b的值, 调用2次RHS, 然后return。此时执行2次LHS, 3次RHS
4. `var c = f(2)` 首先去查找f(2)return的值, 调用一次RHS, 最后赋值调用LHS。此时执行3次LHS, 4次RHS

## 执行上下文

### 产生

执行上下文储存着程序运行所需要的各种变量。

四种情况会生成执行上下文：

- 开始执行全局代码
- 进入function的函数体
- 进入eval函数
- 进入module代码

### 执行过程

执行上下文分为**全局执行上下文**和**函数执行上下文**。说白了这个就是**预编译的作用域**。

在预编译前, 首先会把window确定为全局执行上下文对象, 之后进行预编译。对于函数也是如此, 在执行函数前, 首先会把要执行的函数确定为函数执行上下文对象, 然后进行预编译。

**注意！定义函数并不会产生执行上下文对象！只有调用才会产生。**

```
1 //1.全局执行上下文对象产生
2 function outer(){
3     //3.即将调用inner方法, 再次产生一个函数执行上下文
4     inner()
5 }
6 function inner(){}
7 //虽然定义了, 但是没有被调用, 不会产生执行上下文
8 function free(){}
9 //2.即将调用outer方法, outer函数执行上下文对象产生
10 outer()
```

也就是说在js执行中, 每调用一次方法, 就会产生一个执行上下文对象。那么js是如何管理这些执行上下文的呢? 实际上, js会用一种**执行上下文栈**来储存管理这些对象。

在代码即将执行时, 首先就会把全局执行上下文入栈, 之后每进行一个方法调用就会把当前正在执行的函数执行上下文入栈, 执行完毕时出栈。如果函数嵌套调用的话, 就按照调用顺序依次入栈。

inner函数执行上下文 <-栈顶

outer函数执行上下文

## 变量提升与方法提升

JavaScript为了提高性能，会把代码直接预编译，之后再解释。预编译分为全局预编译和局部预编译，全局预编译发生在页面加载完成时执行，而局部预编译发生在函数执行的前一刻。对应着全局执行上下文和函数执行上下文。

所谓的全局预处理实际上干了这么三件事：

1. 把var定义的全局变量添加为window的属性，并把值设置为undefined。
2. 把function声明的全局函数添加为window的方法，并且把这个函数对象指向对应的堆内存。
3. 把this指向window对象

这就是所谓的**变量提升**和**方法提升**。

局部预编译略有不同：

在执行函数体之前会进行函数的局部预编译。

1. 给形参赋值，并把形参添加为函数执行上下文对象的属性。
2. 给arguments赋值，建立一个伪数组，并把这个添加为上下文对象的属性
3. 把var声明的变量添加为属性，并且把值设置为undefined。
4. 把function声明的方法，添加为上下文对象的方法
5. 把this指向调用当前函数的对象。

理论的东西虽然很重要，但是不举个例子实在是过于抽象。

看看下面这两个代码，它的执行结果是什么？

```
1 //代码1
2 a = 2;
3 var a;
4 console.log(a);
5 //代码2
6 console.log(a);
7 var a = 2;
```

答案是：第一个是2，第二个是undefined。

这个看上去虽然很抽象，但是结合上面的理论来看的话，就能够理解了。

首先把var声明的变量提到最头头，并且赋值为undefined。然后顺序执行代码。

也就是说，实际上代码执行起来应该是这个样子。

```

1 //代码1
2 var a = undefined;
3 a = 2;
4 console.log(a);
5 //代码2
6 var a = undefined;
7 console.log(a);
8 a = 2;

```

其次，方法也是会提升的。注意下面两个代码。

```

1 //代码1
2 test()
3 function test(){
4     console.log("hello world! ")
5 }
6 //代码2
7 test()
8 var test = function (){
9     console.log("hello world! ")
10 }

```

第一个会正常打印结果，但是第二个则会报错。

实际上函数提升只限于function声明的，函数赋值表达式本质上还是一个var声明的变量，所以不会提升。实际上他们执行的代码如下：

```

1 //代码1
2 function test(){
3     console.log("hello world! ");
4 }
5 test();
6 //代码2
7 var test = undefined;
8 test();
9 test = function (){
10     console.log("hello world! ");
11 }

```

另外要注意！**先执行变量提升，后执行方法提升。**也就是说，如果变量名和方法名重名，那么方法会把变量覆盖掉。

```

1 function a(){}
2 var a;
3 console.log(typeof a)//function

```

如果对变量进行了赋值操作，那么就会重新把变量覆盖回来。

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

还要注意，JS中var会无视块级作用域。

```
1 console.log(fn)
2 if(false){
3     var fn = 1;
4 }
5 //undefined
```

## let、const与临时死区

要注意，let是不会变量提升的，如果对let提前引用的话，会报错。

```
1 console.log(a)
2 let a = 1;
3 //报错!!!!
```

这是为什么呢？实际上JS引擎有一个被称为临时死区的東西。

JavaScript引擎在进行预编译的时候，如果遇到var 声明的变量，就提升至作用域顶部。如果遇到let和const则放到临时死区（temporal dead zone）里面。访问TDZ中的变量会触发错误。只有执行了变量的声明之后，变量才会从TDZ中移除。

```
1 console.log(a) //undefined
2 if(false){
3     let a = 1
4 }
```

如果访问的变量不在TDZ中，则显示undefined。

## 全局作用域的绑定

var还有一个致命的特点就是会覆盖全局作用域的变量。

```
1 var a = 1
2 var a = 2
3 console.log(a)//2
```

就是这样，如果把一个变量声明两遍，那么之前那个会被顶替掉

```
1 let a = 1
2 let a = 2
3 console.log(a)//报错，a已经被声明过了。
4
5 var a = 1
6 let a = 2
7 console.log(a)//报错，a已经被声明过了。
8
```

let声明的变量则直接报错。就算之前那个是var也照样报错。

```
1 var a = 1
2 function test(){
3
4     let a = 2
5     console.log(a)
6 }
7 test()
8 console.log(a)
```

如果在局部区域用let定义了同名变量，那么只会覆盖原变量，而不会替换。跟C语言差不多。

## 词法作用域

词法作用域其实就是指得作用域。作用域有三种：

- 全局作用域
- 函数作用域
- 块作用域

## 全局作用域和函数作用域

顾名思义，在js文件中，整个都是全局作用域。在函数内部的是函数作用域。

在函数作用域的变量，不会泄露到外界，可以有效封装。

```
1 var a = "全局作用域";
2 function show(){
3     var a = "函数作用域";
4     console.log(a);
5 }
6 show();
7 console.log(a)
```

## 块作用域 [es6]

JavaScript本来没有块作用域，但是ES6之后新增了块作用域。可以用let声明局部变量。但是var声明的变量还是会忽视块作用域。

**特别注意：java，C++之类的语言是有块级作用域的，别到时候学了JavaScript不会写java了。**

```
1 {var a=1}
2 console.log(a)//1
3
4 {let b=1}
5 console.log(b)//报错
```

## 作用域链

查找一个变量的时候，是先查找当前作用域，之后查找上一级，这样的查找顺序很像一个链表，因此叫做作用域链。

**不过要注意，作用域链的查找是按照函数作用域来进行的，而不是块作用域，最终会抵达全局作用域。**

```
1 var a = "我是全局作用域"
2 var block ={
3   a:"我是块作用域",
4   show:function(){
5     console.log(a)
6   }
7 }
8 block.show()//我是全局作用域
```

可以看到，虽然块作用域也定义了a变量，但是直接被忽视了。

```
1 var a = "我是全局作用域"
2 var block ={
3   a:"我是块作用域",
4   show:function(){
5     var a = "我是函数作用域"
6     console.log(a)
7   }
8 }
9 block.show()//我是函数作用域
```

如果是函数作用域就不会被忽视。

## 访问未定义变量

如果所使用的变量未被定义，有可能触发ReferenceError异常。

是否触发跟LHS和RHS有关系。

如果你直接对一个未声明的变量使用RHS，那么查询会一直抵达全局作用域，如果还没有找到，那么就会抛出异常。

```
1 console.log(a)
```

如果你对一个未声明的变量使用LHS，那么在“非严格模式”下，引擎就会在全局作用域下创建这个变量。

```
1 function addA(){
2   a=1;
3 }
4 addA();
```

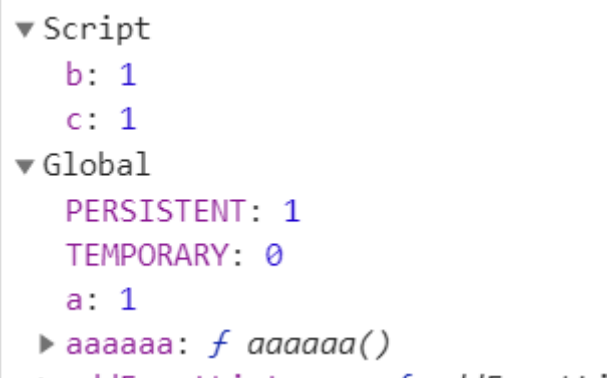
编译器会沿着作用域链一直查找a的定义，结果直到全局作用域都没有找到，所以最后在全局作用域创建了一个a。此时这个a就是全局作用域的变量了。

## 作用域与执行上下文

### 预编译

1. 创建全局执行上下文。并把全局上下文压入执行上下文栈。
2. 找到所有全局作用域中var声明的变量。并放到**全局对象**中。
3. 找到所有的函数声明。并放到**全局对象**中。
4. 找到let、const和class声明的变量。并放到**全局scope**中。

```
1 var a = 1 ;
2 function aaaaaa(){}
3
4 let b = 1 ;
5 const c = 1;
```



```
▼ Script
  b: 1
  c: 1
▼ Global
  PERSISTENT: 1
  TEMPORARY: 0
  a: 1
  ▶ aaaaaa: f aaaaaa()
  ...
```

这是我在chrome掐断点的结果，可以看到chrome中把全局对象叫Global，而把scope叫做Script。

注意在访问变量时，**首先会访问全局scope，之后再访问全局对象。也就是说先访问let和const的再访问var的。**



#### 5. 做名字重复的处理

彼此间名字不能重复，**而且var和function重名的话，function优先。**

6. 登记并把var声明的变量初始化为undefined。

7. 把全局作用域的function登记，并初始化为函数对象。

```
1 console.log(fn)
2 function fn() {}
3 //f fn() {}
```

8. 把块级作用域的function登记，但是初始化为undefined。

```
1 console.log(fn)
2 if(false){
3   function fn() {}
4 }
5 //undefined
```

9. 登记let、const和class的变量，但是**不进行初始化**。

```
1 console.log(a);
2 let a = 1;
3 //Uncaught ReferenceError: Cannot access 'a' before
  initialization at xxxxxxxx
```

我们把这种let提升后的变量叫做临时死区(temporal dead zone)。因为虽然这些变量进行了变量提升，但是还是不可使用。

10. 执行语句。

## 语句执行

我们以下面的代码为例：

```
1 |
```

## 函数原型

这个是函数最重要的地方

## 函数，对象，函数对象和实例对象

函数对象其实就是函数指针，保存着这个函数在内存中的地址。

```
1 function Student() {}
2 Student()
```

上面这个代码中，Student就是函数对象，保存着Student这个函数在堆中的地址值。

而 `Student()` 是函数，代表编译器去访问并执行 `Student` 指针所指向的内容。

```
1 | var xiaoMing=new Student()
```

如果用这个函数去构造对象，那么这个 `xiaoMing` 就是实例对象。其实就是类和对象的关系。

其次，要注意，函数对象是可以 `new` 的，也就是可以有子类的。而有一些静态对象是不能 `new` 的，那些静态对象就不是函数对象。

## prototype（显式原型）

每个函数对象都拥有 `prototype` 属性，这个实际上是一个指针，指向一个被称为原型对象的对象。`prototype` 就是显式原型的指针，可以直接通过属性的方式去访问。

如果这个函数对象是新定义的，那么这个 `prototype` 默认指向一个空对象。

```
1 | function Student(){  
2 |   console.log(Student.prototype)
```

这个空对象是在定义函数时就创建的，相当于下面这个代码

```
1 | this.prototype= new Object()
```

显式原型默认是一个空对象，但是这个空对象并不是完全空，它里面也有两个默认值 `constructor` 和 `__proto__`。

`constructor` 非常特殊，因为 `constructor` 指向 `Student` 函数对象，而函数对象的 `prototype` 又指向原型对象。刚好是一个双链表的结构。



我们可以来直接操作显式原型，为里面添加方法和属性。并且，在显式原型中添加的方法和属性可以直接在实例对象中使用。

```
1 | var 小明=new Student()  
2 | Student.prototype.study=>{  
3 |   console.log("学习使我快乐")  
4 | }  
5 | 小明.study()
```

## \_\_proto\_\_ (隐式原型)

所有函数都有隐式原型和显式原型属性，但是实际上这两个都是指针类型，都指向函数的原型对象。但是函数对象更强调显式原型的属性，而实例对象更强调隐式原型属性。

一旦一个实例对象被创建，其隐式原型就会自动指向构造函数的显式原型。这就导致实际上Student构造函数的prototype和实例对象小明的\_\_proto\_\_，实际上都指向了Student的原型对象。

```
1 function Student() { }
2 var 小明=new Student()
3 console.log(小明.__proto__==Student.prototype)
4 //结果为true
```

实际上在实例对象被创建时，编译器会自动执行这句代码

```
1 this.__proto__=Student.prototype
```

## 隐式原型链

在拥有了以上基础的情况下，我们就来思考这样一个问题。在刚才为显式原型添加了方法后，为什么可以直接就用实例对象调用了？究竟是如何调用的？其实这个就是原型链。

首先，实例对象在调用方法或属性时，首先会去查看自己有没有这个方法。如果有就直接用了。

```
1 function Student() {
2     this.study=()=>{
3         console.log("构造函数添加的方法")
4     }
5 }
6
7 Student.prototype.study=()=>{
8     console.log("原型对象的方法")
9 }
10
11 var 小明=new Student()
12 小明.study=()=>{
13     console.log("我自己手动添加的方法")
14 }
15 小明.study()
16 //结果为:
17 //我自己手动添加的方法
```

如果自己没有手动定义方法（其实就是重写），那么就回去调用构造函数提供的方法。

```
1 function Student() {
2     this.study=()=>{
3         console.log("构造函数添加的方法")
4     }
5 }
6
7 Student.prototype.study=()=>{
8     console.log("原型对象的方法")
9 }
10
11 var 小明=new Student()
12 小明.study()
```

```
13 //结果为:
14 //构造函数添加的方法
```

其次，如果自己没定义，构造函数也没有，那么紧接着就回去查看 `__proto__` 属性所指向的原型对象，调用原型对象的方法。

```
1 function Student() {}
2
3 Student.prototype.study={()=>{
4   console.log("原型对象的方法")
5 }}
6
7 var 小明=new Student()
8 小明.study()
9 //结果为:
10 //原型对象的方法
```

如果Student的原型对象也没有，那么就会去原型对象的对象找。直到抵达原型链的尽头，如果到了尽头还没有找到，就会报错说not defined。

因为这条原型链一直是依据 `__proto__` 来查找原型对象的，所以也叫做**隐式原型链**。

记住 实例对象的隐式原型指向其构造函数对象的显示原型对象。

## 隐式屏蔽

刚才我们说到，编译器会沿着原型链去查找数据，但是如果修改这个值会如何呢？

实际上，如果这个值在原型上的话，就会发生**隐式屏蔽**的bug。

```
1 function Student() {}
2 //可以看到这个a在原型对象上
3 Student.prototype.a=1
4
5 var 小明=new Student()
6 //修改的这个a并不是原型对象的值
7 小明.a+=1
8 console.log(小明.a)
9 console.log(Student.prototype.a)
10 //结果为
11 //2
12 //1
```

这种bug原理其实很简单，编译器在处理 `小明.a+=1` 的时候，首先会通过原型链找到 `Student.prototype.a`，之后进行 `a+=1`。最后在小明这个对象中新建一个属性a，把刚才赋值的结果再赋值给 `小明.a`。

总结一下：

- 读取属性时，会直接去原型链查找结果
- 修改时，先去找结果，之后把这个修改的结果加入本对象中

如果想避免这种情况，请务必直接写 `student.prototype.a+=1!!!`

## 原型链的结构

刚才我们只看了原型链调用数据的过程，现在我们来仔细看一下整个原型链的整体结构。

### Function链

首先我们先定义了Student函数，而这个定义的过程 `function Student() {}` 相当于 `var Student=new Function()`，也就是说我这个Student函数对象本身就是一个实例对象，所以同时拥有 `prototype` 和 `__proto__` 属性，`prototype` 默认指向空对象（下文会说到），而 `__proto__` 则指向了其构造函数 `Function` 的原型对象。

```
1 | Student.__proto__==Function.prototype
```

Function是什么？Function本身就是用来构造其他函数的函数对象，`Function()` 就是构造其他函数的构造函数。那你自己的也是一个函数啊？那你这个函数是谁构造的呢？

答案是 `Function` 的构造函数就是自身，也就是说：

```
1 | Function.prototype==Function.__proto__
```

没错，`Function` 非常特殊，它自己就是自己的构造函数，他自己就是自己的实例对象，所以他的两个原型属性相等，指向了 `Function` 的原型对象。

那么 `Function` 的原型对象是什么？`Function` 的原型对象就是一个对象，所以它由 `new Object` 产生，这就代表，`Function` 的隐式原型指向构造函数 `Object` 的原型对象。

```
1 | Function.prototype.__proto__==Object.prototype
```

而 `Function.prototype` 没有 `prototype` 了，因为他并不是一个函数对象。

这样，这条链就完了。

### Object链

刚刚说到，创建一个新的函数对象 `Student` 时，其 `prototype` 默认指向空对象。这个空对象的构建过程是什么呢？首先需要 `new` 一个 `Object`，那么既然是 `new` 出来的，那么这个 `Object()` 就是一个构造函数了，那么刚才我们说过，一切函数都是由 `Function` 构造函数创建的，所以

```
1 | Function.prototype==Object.__proto__
```

之后，`Object` 本身肯定也有一个 `prototype` 属性，它指向 `Object` 的原型对象。这个原型对象非常重要，里面定义了 `toString`，`valueOf` 等重要方法。而

`Object.prototype.prototype==undefined`，很简单，因为 `Object` 的原型对象本身并不是一个函数，所以没有 `prototype`。但是所有对象都有隐式原型，这就麻烦了。你 `Object` 的原型对象也是一个对象，如果让他也默认指向一个空对象，那么那个空对象也会有一个隐式原型，就会指向下一个空对象，导致无限递归。

为了避免这种情况，我们规定：

```
1 | Object.prototype.__proto__===null
```

这样这条链就完结了。

## 实例对象链

首先我要强调，只有函数对象才有prototype，普通的实例对象的prototype===null。

```
1 | 小明.prototype===null
```

小明作为实例对象，拥有\_\_proto\_\_属性，指向其构造函数的prototype。

```
1 | 小明.__proto__===Student.prototype
```

还记得我刚刚说过的吗？Student的prototype在一开始就被创建了，相当于new了一个空对象。那么既然是被new出来的空对象，那么也就是一个实例对象喽，既然是实例对象，那么这个Student的原型对象也拥有一个\_\_proto\_\_，指向Object构造函数的原型对象。

也就是说：

```
1 | Student.prototype.__proto__===Object.prototype
```

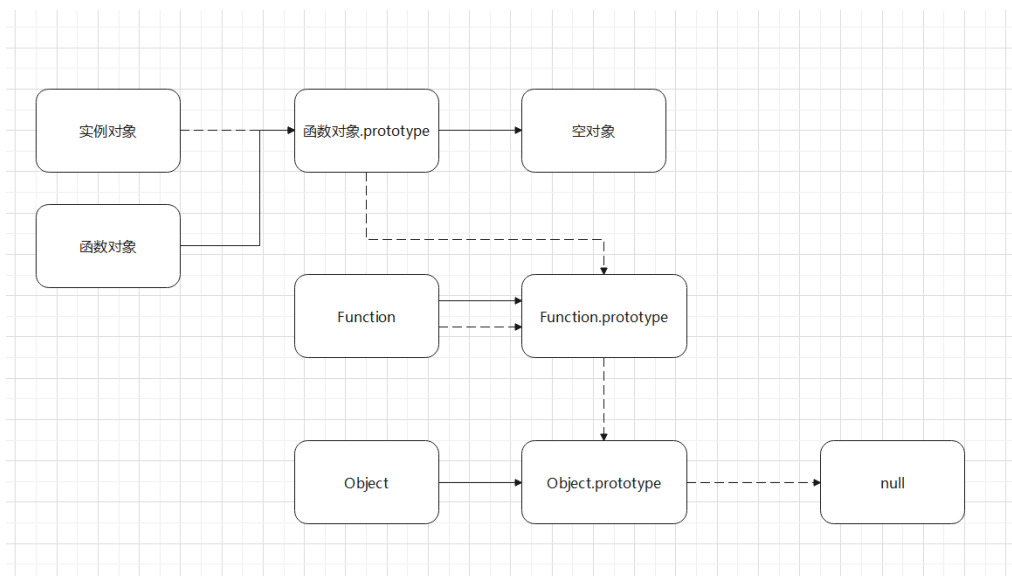
而Student.prototype是一个普通的实例对象，所以没有prototype。

```
1 | null===Object.prototype.prototype
```

这样这条线也完了。

## 总结

- 被谁new，隐式原型就指向谁的显示原型。
- 能new谁，谁就有显示原型。而且隐式原型指向Function.prototype



# this

精通this是你从JS萌新变成JS巨佬的必经之路,但是想熟练掌握这个东西,并不是一件轻松的工作..

this的指向由其调用函数的上下文决定的。

## 下马威

不要说你很懂this, 在java或者其他高级语言里面, this的指向或许比较简单, 一般都是指向类的实例对象。但是在JavaScript中, this真的是要人老命。

首先来看看这个代码, 看看结果和你想的是否一样。

```
1 function add(){
2   this.num++;
3 }
4 add.num = 0;
5 add();
6 console.log(add.num);
```

答案是0。add的num根本就没有增加!

原因其实也不是很简单, 首先 `add.num = 0;` 这句代码确实确实为add增加了一个num属性, 但是我们要注意, `add()` 这句代码实际上是缩写的, 完整版应该是 `window.add()`。也就是说, 此时this又变成了window, 而window没有num属性, 所以说 `this.num++;` 的时候, window对象动态生成了一个num属性, 并且++。

总结一下, 在执行add()之前, 只有add有num属性。而在执行add()的时候, this指向发生了变化, 指向了window。而window对象没有num属性, 所以动态生成了一个num属性, 并且++。

当然解决这种bug的方法也很简单, 第一种是在函数内部用函数名去访问属性。

```
1 function add(){
2   add.num++;
3 }
4 add.num = 0;
5 add();
6 console.log(add.num);
```

第二种就是让this强行指向add。

```

1 function add(){
2   this.num++;
3 }
4 add.num = 0;
5 add.call(add);
6 console.log(add.num);

```

如果你不想犯这种错误，建议平时就不要用this，或者说仔细看看我下面的讲解。

## this与函数调用栈

this其实和全局执行上下文是紧密相关的。其实就是函数的调用栈。this永远指向栈顶元素。

如果以函数的形式调用this，那么this就是调用对象。

因为这个调用栈为：小丛雨->show

然后从show开始，沿着栈去查找上一层指针，也就是小丛雨。

```

1 var 小丛雨={
2   age:18,
3   name:"小丛雨",
4   show:function(){
5     console.log(this)
6   }
7 }
8 小丛雨.show()
9 //输出
10 //{age: 18, name: "小丛雨", show: f}

```

可以看到，this就是调用该方法的对象。

比较特殊的是，所有全局函数都是定义在window对象上的，所以全局函数的this都是window

```

1 function Student() {
2   console.log(this)
3 }
4 Student()
5 //输出
6 //window
7 //实际上相当于window.Student()

```

this当然不可能这么简单，来看点难的吧！在回调函数中，this将会怎么样呢？

```

1 function recall(fn){
2   this.data =1;
3   fn();
4 }
5 var data = 0;

```



```

6
7 recall(
8   function(){
9     console.log(this.data);
10  }
11 )
12 //调用栈为
13 //window->recall->fn
14 //然后打印data，沿着调用栈去找，最后在recall里面找到1
15 //打印1

```

## 在构造函数中

首先来看看执行构造函数的流程：

1. 创建一个新的对象
2. 链接原型
3. 绑定this
4. 如果函数没有return语句，那么在new表达式中，会自动return this

```

1 function Girls() {
2   this.name = '雷姆';
3   this.age = 18;
4   this.show= function() {
5     console.log(this)
6   }
7 }
8 var LeiMu = new Girls();
9 LeiMu.show();
10 //输出
11 //Girls {name: "雷姆", age: 18, show: f}

```

也就是说实际上代码应该是这个样子的

```

1 function Girls() {
2   var temp = new Object();
3   temp.name = '雷姆';
4   temp.age = 18;
5   temp.show = function () {
6     console.log(this);
7   }
8   //注意，实际上是给temp赋值的，我这只是为了说明而已
9   this = temp;
10  return this;
11 }
12 var LeiMu = new Girls();
13 LeiMu.show();

```

另外，在调试的时候可以发现，在创建这个对象的时候，首先会开辟一片内存空间，this会一直指向这个内存空间。之后Girls把this直接return了，赋值给了LeiMu，所以之后LeiMu也指向了这个空间，这样就完成了地址的接力棒。

## 调用call和apply时

call方法里面是谁，this就指向谁，这个最简单。

```
1 function show(){
2   console.log(this)
3 }
4 var obj={name:"我是obj啦"}
5 show.apply(obj)
6 //输出
7 //{ name: '我是obj啦' }
```

## super的指向

super指向其父类的显式原型。

## let与this

let声明的变量并不依附于this

```
1 console.group("1")
2 let a = 1;
3 console.log(a) //1
4 console.log(this.a) //undefined
5 console.groupEnd();
6
7 console.group("2")
8 var b = 2;
9 console.log(b) //2
10 console.log(this.b) //2
11 console.groupEnd();
```

注意！在node环境下，this.b 也是undefined

## 全局环境下的this

默认指向window

```
1 function fun1(params) {
2   console.log(this)
3 }
4 fun1() //window
```

在严格模式下，undefined

```

1 function fun1(params) {
2     'use strict'
3     console.log(this)
4 }
5 fun1() //undefined

```

```

1 var obj = {
2     fun:function(){
3         console.log(this);
4     }
5 }
6 var temp = obj.fun
7 temp()//window

```

## 存在上下文时

```

1 const obj1 = {
2     text: "1",
3     fun: function () {
4         return this.text
5     }
6 }
7 const obj2 = {
8     text: "2",
9     fun: function () {
10        return obj1.fun()
11    }
12 }
13 const obj3 = {
14     text: "3",
15     fun: function () {
16         var fn = obj1.fun
17         return fn()
18     }
19 }
20
21 console.log(obj1.fun());//1
22 console.log(obj2.fun());//1
23 console.log(obj3.fun());//undefined

```

因为第三个return fn() 的时候，这个fn()其实是window调用的，obj3调用了fun，而fn由window调用。因此是undefined

现在问题来了，如果想要obj2返回2该怎么办？

如果不用apply之类的话，可以

```
1  const obj2 = {  
2    text: "2",  
3    fun: obj1.fun  
4  }
```

## 构造函数与this

构造函数中，this分为两种情况

- 显示返回一个对象

此时this绑定到了显示的对象上面

```
1  function fun1() {  
2    this.data = 1  
3    var obj = {}  
4    return obj  
5  }  
6  var obj1 = new fun1()  
7  console.log(obj1.data); //undefined
```

- 返回一个值

此时this绑定到了new的对象上面

```
1  function fun2(){  
2    this.data = 1  
3    return "返回值"  
4  }  
5  var obj2 = new fun2()  
6  console.log(obj2.data); //1
```

## 箭头函数的this

箭头函数的this会根据上下文决定，并不一定是window

```
1  var obj = {  
2    func: function () {  
3      setTimeout(()=>{  
4        console.log(this)  
5      })  
6    }  
7  }  
8  }  
9  
10 obj.func() // {func: f}
```

## this的优先级

- 用call和apply显示绑定优先级高

```

1  function foo(a){
2      this.a =a
3  }
4  const obj1 = {
5      a:1,
6      foo:foo
7  }
8
9  const obj2 = {
10     a:2,
11     foo:foo
12 }
13
14 obj1.foo.call(obj2)//2
15 obj2.foo.call(obj1)//1

```

- bind也是

```

1  function foo(a){
2      this.a =a
3  }
4  const obj1 = {}
5  var bar = foo.bind(obj1)
6  bar(2)
7  console.log(obj1.a)//2

```

- 但是作为构造函数就不一样了

```

1  function foo(a){
2      this.a =a
3  }
4  const obj1 = {}
5  var bar = foo.bind(obj1)
6  bar(2)
7  console.log(obj1.a)//2
8
9  var baz = new bar(3)
10 console.log(baz.a)//3

```

此时构造函数的this已经断掉了obj1

## 闭包

### 闭包的创建

当一个内部函数使用外部函数的数据时，就产生了闭包。

```

1 function outer(){
2     var a=1;
3     function inner(){
4         console.log(a)
5     }
6 }
7 outer()

```

下面这种情况就不会产生闭包，因为函数的定义并没有被执行，此时必须要调用inner才能产生闭包。

说白了，就是函数的私有数据被外界访问了，那么就会形成闭包。

```

1 function outer(){
2     var a=1;
3     var inner = function(){
4         console.log(a)
5     }
6     //inner() 调用了inner()就会产生闭包
7 }
8 outer()

```

闭包实际上是一种内存泄漏，因为一旦outer函数的数值被占用，就无法被垃圾回收，所以其实就是内存泄漏。

## 闭包的作用

### 实现静态局部变量

闭包可以让外部函数的局部变量不会被释放，相当于续命（膜）。利用这个特性，可以来间接实现静态变量。

```

1 function outer(){
2     var a=1
3     var add = function(){
4         a++
5         console.log(a)
6     }
7     return add
8 }
9 var out = outer()
10 out()//2
11 out()//3
12 out()//4

```

可以看到，a这个局部变量没有被释放，一直在内存中。间接实现了静态变量。

并不是所有的变量都不会被释放，只有被内部函数使用的数据才会保留，剩下的数据都会被释放，包括add这个变量。之所以还能执行add方法，是因为out保留了add方法的指针，这才使add的内存没有被回收。

## 回调函数

回调函数经常会使用闭包，原理就是回调函数会调用外部数据。

**只要使用了回调函数，就使用了闭包！**

```
1 function showDealy(msg){
2     //内部箭头函数调用了外部的msg数据，产生闭包
3     setTimeout(()=>{
4         console.log(msg)
5     })
6 }
7 showDealy("hello world")
```

## 模块化（对象导出）

使用闭包可以实现模块化。

```
1 //module.js
2 function module (){
3     function say(){
4         console.log("永远喜欢波莱特")
5     }
6     function say2(){
7         console.log("永远喜欢爱丽丝")
8     }
9     return{
10         say:say,
11         say2:say2
12     }
13 }
```

```
1 <!--index.html-->
2 <!DOCTYPE html>
3 <html>
4     <head>
5         <meta charset="utf-8">
6         <title></title>
7     </head>
8     <body>
9         <!--引入外部模块-->
10        <script type="text/javascript" src="module.js">
11    </script>
12        <script type="text/javascript">
13            //引用
14            var loveModule = module()
15            loveModule.say()
16            loveModule.say2()
17            //或者使用js对象解构赋值
18            var {say,say2} = module()
19            say()
20            say2()
21        </script>
22    </body>
23 </html>
```

## 模块化（添加window属性）

```
1 //module.js
2 (function module (){
3     function say(){
4         console.log("永远喜欢波莱特")
5     }
6     function say2(){
7         console.log("永远喜欢爱丽丝")
8     }
9     window.module={
10         say:say,
11         say2:say2
12     }
13 })()
```

```
1 <!--index.html-->
2 <!DOCTYPE html>
3 <html>
4     <head>
5         <meta charset="utf-8">
6         <title></title>
7     </head>
8     <body>
9         <script type="text/javascript" src="module.js">
10     </script>
11         <script type="text/javascript">
12             //添加window对象的属性之后可以直接调用module对象
13             module.say()
14             module.say2()
15         </script>
16     </body>
17 </html>
```

## for循环与闭包

首先看这段代码，猜猜看执行结果是什么。

```
1 for (var i = 0; i < 5; i++) {
2     setTimeout(function () {
3         console.log(i);
4     }, i*1000);
5 }
```

for执行5次，每隔1s打印一个数字，5秒刚好打印完5个。你是这么想的吗？

实际上打印结果将是5个5。

其实很好理解，for循环终止的时候，i刚好等于5。然后回调函数过了1秒以后又回来取得i的值，此时i已经是5了，所以不论过了几秒，都会打印5。

这个代码之所以没有按照我们的思路来执行，是因为实际上他们共享了一个作用域，而我们希望他们各自的i都有自己的作用域。解决方案如下：



1. 使用立即执行函数来构造一个独立的作用域，
2. 给这个作用域创立一个独立的变量来储存共有变量的值。

```
1  for (var i = 0; i < 5; i++) {
2      (function(j){
3          setTimeout(function () {
4              console.log(j);
5          }, j*1000);
6      })(i)
7  }
```

首先我们用一个新的变量 `j` 来储存 `i` 的值，此时这个 `j` 就是独立于全局作用域的，仅仅在函数内部。之后，全局作用域把 `i` 的值传了过来，这样就能保证每个作用域的数据独立了。

## 闭包的生命周期

闭包的创建只有一个要求，**内部函数使用了外部函数的数据**，而且内部函数不需要调用，只需要执行定义就可以。所以一旦满足这个要求，闭包就会产生。

另外要注意变量提升和方法提升。

```
1  function outer(){
2      //此时闭包已经产生，因为有变量提升和方法提升
3      var a=1
4      function add(){
5          a++
6          console.log(a)
7      }
8      return add
9  }
10 var out = outer()
```

```
1  function outer(){
2
3      var a=1
4      var add = function (){
5          //在赋值语句执行完毕之后闭包产生，因为此时才去执行函数的定义。
6          a++
7          console.log(a)
8      }
9      return add
10 }
11 var out = outer()
```

闭包的死亡其实很简单，只需要把外部函数的数据释放就可以了

```
1 function outer(){
2     var a=1
3     function add(){
4         a++
5         console.log(a)
6     }
7     return add
8 }
9 var out = outer()
10 //一旦out指向了null，那么垃圾回收器就会把outer的数据回收，闭包消失
11 out=null
```

## 异常处理

注意：JavaScript是解释型语言，所以一旦遇到异常之后，剩下的代码就不会执行了。

## 异常类型

### ReferenceError

引用错误，一般指引用的变量不存在

1. 访问了不存在的变量

```
1 console.log(a)
2 //Uncaught ReferenceError: a is not defined
```

2. 在初始化前访问变量

注意，只有var声明的变量会进行变量提升，所以let和const声明的变量，必须先声明后使用。

```
1 console.log(a)
2 let a =1;
3 //Uncaught ReferenceError: Cannot access 'a' before
  initialization
```

### TypeError

类型错误，指的是数据类型和使用的不一样

1. 把普通变量当成函数

```
1 var a =1
2 a()
3 //Uncaught TypeError: a is not a function
```

## RangeError

范围错误，一般是指堆栈溢出

### 1. 递归导致栈溢出

```
1 function fn(){
2   fn()
3 }
4 fn()
5 //test.html:9 Uncaught RangeError: Maximum call stack size
  exceeded
```

## SyntaxError

语法错误

```
1 var a = ?????
2 console.log(a)
3 //Uncaught SyntaxError: Invalid or unexpected token '??'
```

## try catch

使用了捕获之后，程序就可以一直运行下去

```
1 try{
2   var a =1
3   a()
4 }catch(error){
5   console.log("error message:"+error.message)
6   console.log("error stack:"+error.stack)
7 }
```

## throw

```
1 throw new Error("对异常的描述信息")
```

## 自定义异常

继承Error类，可以用于自定义异常

```
1 function MyError(message) {
2   this.name = 'MyError';
3   this.message = message || 'Default Message';
4   this.stack = (new Error()).stack;
5 }
6 MyError.prototype = Object.create(Error.prototype);
7 MyError.prototype.constructor = MyError;
```

## XML与JSON

## JSON

JavaScript Object Notation (JavaScript对象表示法)，简称JSON，其实就是一种更加规范的JavaScript对象。JSON的用途非常广泛，主要用于配置文件，还有数据的传递。

## 实例

```
1  {
2    "girls":[
3      {
4        "name":"まゆり",
5        "age":18
6      },
7      {
8        "name":"よしの",
9        "age":18
10     }
11   ]
12 }
```

## 字符串转JSON

```
1 | var jsonObject= JSON.parse(jsonstr);
```

## JSON转字符串

```
1 | var jsonstr =JSON.stringify(jsonObject);
```

## 对象转JSON

```
1 | JSON.stringify(studentInfo);
```

## 遍历JSON对象

如果find到了数据，会返回true

```
1  const fs = require('fs')
2  fs.readFile('./data.json', 'utf8', (err, data) => {
3
4    const db = JSON.parse(data);
5    const flag = db.girls.find((i) =>
6      i.name == "まゆり"
7    ) //要注意这个箭头函数不能带大括号，虽然我也不知道为什么
8    if (flag)
9      console.log('ok')
10   else
11     console.log('no')
12 })
```

## 访问本地JSON

```
1 DataManager.loadDataFile = function(name, src) {
2     var xhr = new XMLHttpRequest();
3     var url = 'data/' + src;
4     xhr.open('GET', url);
5     xhr.overrideMimeType('application/json');
6     xhr.onload = function() {
7         if (xhr.status < 400) {
8             window[name] = JSON.parse(xhr.responseText);
9             DataManager.onLoad(window[name]);
10        }
11    };
12    xhr.onerror = this._mapLoader || function() {
13        DataManager._errorUrl = DataManager._errorUrl || url;
14    };
15    window[name] = null;
16    xhr.send();
17 };
```

## XML

Extensible Markup Language

### 获取XML文件并解析

```
1 //打开XML文件
2 var xhr = new XMLHttpRequest();
3 xhr.open("GET", "data.xml");
4 xhr.send(null);
5
6 setTimeout(() => {
7     if (xhr.readyState == 4) {
8         if (xhr.status >= 200 && xhr.status < 300) {
9
10            //获取XML的字符串
11            var xmlString = xhr.response;
12
13            //把字符串转成对象
14            var ObjectParser = new DOMParser();
15            var data = ObjectParser.parseFromString(xmlString,
16            "text/xml");
17
18            console.log(data); //测试结果
19        }
20        else {
21            console.log("服务器状态错误");
22        }
23    } else {
24        console.log("没有服务器")
25    }
26 }, 2000);
```

## XML DOM

```
1 data.getElementsByTagName("name")[0] //获取name节点
2 data.getElementsByTagName("name")[0].nodeName //获取name节点的节点名，还是name
3 data.getElementsByTagName("name")[0].innerHTML //获取内容
```

## 正则表达式

<https://regex101.com/r/6aa35p/1/>

以 / / 来打开正则表达式的范围。

```
1 /^1[34567]\d{9}$/
2 //1开头，第二个数字为34567中的一个，之后有9个数字，并且以9个数字结束
```

## 限定符

限定符	说明	代码	匹配结果
?	前一个字符可有可无	abc?	ab 和 abc
*	前一个字符可以有多个或没有	ab*c	abbbc 和 abc 和 ac
+	前面的字符必须出现1次以上	ab+c	abc 和 abbbc
{出现次数}	前面字符出现的次数	ab{2}c	abbc
{出现最小次数, 最大次数}	前面字符出现的次数范围	ab{2,3}c	abbc 和 abbbc
{出现最小次数, }	字符至少出现多少次, 最高不限	ab{2,}c	abbc 和 abbbbc
	或运算符	a(b c)	ab 和ac
[0-9]	所有数字		

## 元字符

字符	说明	
\d	所有数字	
\w	所有字母	
\s	空白符，包括tab	
\D	所有非数字	
\W	非单词字符	
\S	非空白符	
.	任意字符（不包括换行符）	
^a	以a开头	
b\$	以b结尾	

## Promise

异步编程在程序开发中用的非常非常广泛，数据库访问、文件读写都需要用到。在ES5中，我们通常使用回调函数来解决，但是在ES6中，我们拥有了promise这种强大的工具来处理异步操作。

```
1 //文件读写
2 require('fs').readFile('/test.txt',(err,data)=>{})
3
4 //ajax
5 $.get('/server',(data)=>{})
6
7 //定时器
8 setTimeout(()=>{},3000)
```

## 为什么需要promise

### 1. 可以解决回调地狱

什么是回调地狱？简单来说，就是回调函数嵌套了好几层，可读性太差,不便于异常处理。

### 2. 指定回调函数的方式更加灵活

## 基本用法

- promise的构造函数内需要传递一个函数，第一个值为resolve，第二个为reject，都是promise内部定义的。
- then方法，需要传递两个函数，第一个是onResolved成功回调，第二个是onRejected失败回调
- catch方法，只能接收失败的回调

```

1  var name = "莲华"
2
3  var promise = new Promise((resolve, reject) => {
4      setTimeout(() => {
5          if (name == "莲华")
6              resolve("结婚");
7          else
8              reject("全部图图")
9      },1000)
10 })
11
12
13 promise.then(result => void console.log(result))
14 .catch(result => void console.log(result))
15 .finally(() => {
16     console.log("永远喜欢莲华")
17 })

```

## 实例

### ajax封装

```

1  function myAjax(url) {
2      var promise = new Promise((resolve, reject) => {
3          var xhr = new XMLHttpRequest()
4          xhr.responseType = 'json'
5          xhr.open('GET', url)
6          xhr.send()
7          xhr.onreadystatechange = function () {
8              if (xhr.readyState == 4) {
9                  if (xhr.status >= 200 && xhr.status < 300
10 || xhr.status === 304) {
11                      resolve('ok')
12                  }
13                  reject('no')
14              }
15          })
16          return promise
17      }
18      myAjax('https://api.apiopen.top/getJok').then(data=>{
19          console.log(data)
20      })

```



## 延时调用

下面是一个promise的实例，首先会生成一个随机数，如果该值小于0.5就会成功，如果大于0.5就会失败。

promise成功时，用resolve来表示，失败时用reject来表示。这两个方法里面都可以传参数，这个参数会传递到then里面去。

then有两个回调函数作为参数，第一个是resolve后调用，代表成功，第二个是reject后调用，代表发生错误。这个是系统自己调用的。

```
1  const promise = new Promise((resolve, reject) => {
2    setTimeout(function() {
3      var data = Math.random();
4      if (data < 0.5)
5        resolve(data);
6      else
7        reject(data);
8    }, 1000)
9  })
10
11 promise.then(value => {
12   console.log("成功, 值为" + value)
13 }, reason => {
14   console.log("失败, 原因为" + reason)
15 })
```

换句话说，promise的构造函数里面写触发成功或失败的条件，而then里面写成功或失败之后的事件。

## 文件读取

```
1  const promise = new Promise((resolve, reject) => {
2    require('fs').readFile('/test.txt', (err, data) => {
3      if (err)
4        reject(err);
5      resolve(data);
6    })
7  })
8
9  promise.then(value => {
10   console.log("成功, 值为" + value.toString())
11 }, reason => {
12   console.log("失败, 值为" + reason)
13 })
```

我们把Promise里面的那个函数叫做**执行器函数**。执行器函数里面的代码是**同步执行**的。

## util.promisify

这个东西可以直接把一个异步的方法封装成promise对象。注意这个最好封装官方提供的异步方法。

```
1  const fs = require('fs')
2  const util = require('util')
3  const myReadFile = util.promisify(fs.readFile);
4  myReadFile('data.json').then(value=>{
5    console.log(value.toString())
6  })
```

```
1  const util = require('util')
2  const myReadFile = util.promisify(setTimeout);
3  myReadFile(1000).then(()=>{
4    console.log('hello world')
5  })
```

## 链式调用

```
1  let b = new Promise((resolve, reject) => {
2    resolve(1)
3  })
4  .then((r) => {
5    console.log(r);
6    return 3
7  })
8  .then((r) => {
9    console.log(r)
10 })
```

## promise的属性

### *PromiseState*

这个属性决定了promise在then的时候，到底是进入第一个回调函数，还是进入第二个。

PromiseState有三个属性

- pending：默认值
- resolved 或者 fulfilled：表示成功
- rejected：表示失败

promise在创建的时候默认都是pending，如果调用了resolve或者reject之后，这个参数就会发生变化，而且一个promise对象这个属性只能改变一次。

在promise中throw一个异常，也会导致状态变为reject。

## PromiseResult

这个属性保存了你在promise时，传进来的参数。

```
1  const promise = new Promise((resolve, reject) => {
2    setTimeout(function() {
3      var data = Math.random();
4      if (data < 0.5)
5        resolve(data); // 这些data就会被传入到PromiseResult中去，以便
    then时来调用。
6      else
7        reject(data);
8    }, 1000)
9  })
```

## Promise.resolve()

这个方法比较特别，它的参数可以接收一个任意类型的数据，然后快速生成一个promise对象。

```
1  var promise = Promise.resolve([1, 2, 3])
```

之后它会返回一个promise对象，其PromiseState为fulfilled，PromiseResult为刚才传入的值。

如果传入的参数为一个Promise对象，那么这个Promise对象触发了什么，就会传入什么属性。

```
1  const promise = Promise.resolve( new Promise((resolve, reject) => {
2    resolve("data")
3  }))
4
5  promise.then(data => void console.log(data))
```

## Promise.reject()

可以快速创建一个失败的promise对象

这个要注意，就算里面传入的对象是resolve，仍然会失败

```
1  const promise = Promise.reject( new Promise((resolve, reject) => {
2    resolve("data")
3  }))
4
5  promise.catch(data => void console.log(data))
6  // catch之后的值是fulfilled，因为之前resolve了
```

## Promise.all()

当全部的promise都resolve的时候

```
1 let promise1 = new Promise((resolve, reject)=>{
2   resolve(1)
3 })
4
5 let promise2 = Promise.resolve(2)
6 let promise3 = Promise.resolve(3)
7
8 var result = Promise.all([promise1, promise2, promise3])
9 console.log(result)
10 //状态为: fulfilled
11 //PromiseResult为一个数组, 里面包含各个promise的返回值 [1, 2, 3]
```

如果有任何一个出现了reject, 那么就会返回reject, 并且result为第一个reject的值。

```
1 let promise1 = new Promise((resolve, reject)=>{
2   resolve(1)
3 })
4
5 let promise2 = Promise.reject(2)
6 let promise3 = Promise.reject(3)
7
8 var result = Promise.all([promise1, promise2, promise3])
9 console.log(result)
10 //如果有任何一个
11 //状态为: rejected
12 //PromiseResult 2
```

## 多回调函数

当PromiseState发生改变的时候, 所有的then都会被执行; 同样的, 如果没有状态的变化, then里面的方法就不会执行。

## async和await

### 基础

如果对一个函数使用async修饰, 那么就会返回一个promise对象, 其值为函数的返回值, 状态为fulfilled。

```
1 var a = async function test(){
2   return 520;
3 }()
4 console.log(a)
5 //Promise { 520 }
```

如果该函数的返回值是一个promise对象, 那么这个生成的promise对象将和返回的一样。

await必须写在async函数中，但是async函数可以没有await。

**await一般修饰一个promise对象**，如果promise对象的状态为成功，那么就会返回**值**。

```
1  async function test(){
2      var p = new Promise((resolve,reject)=>{
3          resolve("ok")
4      })
5      var message = await p; //p状态成功，会直接返回"ok"
6      console.log(message)
7  }
8  test()
```

如果await修饰的promise状态为失败，那么会抛出一个异常，其参数为reject的值。

```
1  async function test(){
2      var p = new Promise((resolve,reject)=>{
3          reject("no")
4      })
5      try{
6          var message = await p;
7      }catch(err){
8          console.log(err) //捕获异常，打印reject的值。
9      }
10 }
11 test()
```

## 用法

如果有一堆异步的操作，但是又需要他们同步执行的时候，await就用到了。比如说我现在需要依次访问3个文件，并且打印他们的内容，如果单纯写3个readfile绝对会出大问题，因为异步操作并不确定哪个文件先被访问，但是如果对他们用await修饰的话，就能同步执行了。

```
1  const fs = require('fs')
2  const util = require('util')
3  var filereader = util.promisify(fs.readFile);
4
5  async function test (){
6      try{
7          var data1 = await filereader('test1.txt');
8          var data2 = await filereader('test2.txt');
9          var data3 = await filereader('test3.txt');
10         console.log(data1+data2+data3);
11     }catch(err){
12         console.error(err);
13     }
14 }
15
16 test();
```

# 手写Promise

```
1  class myPromise {
2      // 声明3个状态，分别为pending等待/resolve解决/reject拒绝
3      static PENDING = 'pending'
4      static FULFILLED = 'fulfilled'
5      static REJECT = 'rejected'
6      constructor(executor) {
7          // executor为执行者
8          // 一开始是等待状态
9          this.status = myPromise.PENDING
10         this.value = null
11         // 用于保存需要执行的函数
12         this.callbacks = []
13         // 这里使用try..catch的原因是如果在promise中出现了错误信息
            的情况，就直接丢给reject来处理
14         try {
15             // class内this遵循严格模式，此处的this是指向的
window
16             // 使用bind认为干预this指向
17             executor(this.resolve.bind(this),
this.reject.bind(this));
18         } catch (error) {
19             // 把错误信息给reject来处理
20             this.reject(error)
21         }
22     }
23     // 类方法
24     resolve(value) {
25         // 这里需要增加一个判断，如果当前Promise的状态为pending的
            时候，才能进行状态更改和处理
26         if (this.status === myPromise.PENDING) {
27             // 执行类方法的时候就要改变Promise的状态和值
28             this.status = myPromise.FULFILLED
29             this.value = value
30             setTimeout(() => {
31                 // 这里还是要处理为异步任务，如果promise内出现异
步处理的函数内还有同步任务
32                 // 那么需要先解决同步任务，再进行Promise的状态改
变和处理
33                 // 可以结合后文的图片来理解[图：Pending状态下的
异步处理]
34                 this.callbacks.map(callback => {
35                     // 这里应该这样理解，当状态异步改变的时候，先
把then里面的方法保存起来，然后等状态改变了
36                     // 就从之前保存的函数中，拿到then里面的对应
方法执行
37                     callback.onFulfilled(value)
38                 })
39             })
        }
```

```

40     }
41   }
42   reject(reason) {
43     if (this.status === myPromise.PENDING) {
44       this.status = myPromise.REJECT
45       this.value = reason
46       setTimeout(() => {
47         this.callbacks.map(callback => {
48           // 这里用map应该是存入的时候使用了数组push，
           这里map会对每个数组数据进行处理，这里数组只有1个数据
49           // 所以是执行1次，存入的又是一个对象所以可以
           用callback.onRejected获取对应函数
50           callback.onRejected(reason)
51         })
52       })
53     }
54   }
55   // then方法的实现
56   then(onFulfilled, onRejected) {
57     // 两个函数不是必须传入的，例如会出现
           then(null,onRejected)的情况
58     if (typeof onFulfilled !== 'function') {
59       // 自己封装一个空函数上去
60       // 同时返回value的值，由此来实现穿透
61       onFulfilled = () => { return this.value }
62     }
63     if (typeof onRejected !== 'function') {
64       // 自己封装一个空函数上去
65       onRejected = () => { return this.value }
66     }
67     // 这里是返回一个全新的promise，是为了让then支持链式调用
68     // 将返回的Promise进行保存用于之后返回判断
69     let promise = new myPromise((resolve, reject) => {
70       if (this.status === myPromise.PENDING) {
71         // 这里是等待状态，在promise内部的函数是异步执行的，
           例如过多少秒之后才解决，
72         // 那么我们手写的Promise就应该在pending状态下，
           保存需要执行的函数。
73         // 异步改变状态的时候，会先来到这里保存函数
74         this.callbacks.push({
75           // 这里将保存的函数进行了一次包装，如果异步
           then中出现了错误，也会全部交给onRejected来处理
76           onFulfilled: value => {
77             this.parse(promise,
onFulfilled(value), resolve, reject)
78             // try {
79             //   // 这里的更改主要是针对Promise的
           内部函数的异步处理
80             //   let result =
onFulfilled(value)
81             //   // 通过instanceof来判断result
           是否是通过myPromise实现的，从而确定是否是返回的Promise
82             //   if (result instanceof
myPromise) {
83               //   // 如果是Promise的话，目的还是
           要改变Promise的状态，并且返回值
84             //   // 此时 result 是一个
           Promise

```

```

85 // // 这里相当于重新执行了一次返回
    回的Promise，递归
86 //
    result.then(resolve, reject)
87 // } else {
88 // // 普通值直接返回
89 // resolve(result)
90 // }
91 // } catch (error) {
92 // // then的异步处理中出现error就
    交给onRejected处理
93 // // 同时这里的处理函数从
    onRejected改为reject，相当于把错误代码交给了
94 // // 最后一个then来处理
95 // reject(error)
96 // }
97 },
98 onRejected: reason => {
99     this.parse(promise,
    onRejected(reason), resolve, reject)
100 },
101 })
102 }
103 // 这里判断状态，必须是解决状态的情况才会执行
    onFulfilled函数，否则会出现状态没改变也执行的情况
104 if (this.status === myPromise.FULFILLED) {
105     // 使用setTimeout将then中的处理变成异步方法，这
    样才能符合正常promise的运行顺序
106     // 这时这些代码不会立即执行，而是进入下一个loop中
107     // 这里有点小问题，是setTimeout是宏任务，和自带
    promise相比，这个会比自带的then晚执行
108     setTimeout(() => {
109         // 因为链式调用把之前的值保存，然后传递给下一
    个then
110         // 函数要是运行了就是把值传递，没有就是传递函
    数
111         this.parse(promise,
    onFulfilled(this.value), resolve, reject)
112     }, 0)
113     }
114     if (this.status === myPromise.REJECT) {
115         setTimeout(() => {
116             this.parse(promise,
    onRejected(this.value), resolve, reject)
117         }, 0)
118         }
119     })
120     return promise
121 }
122 // 代码复用
123 parse(promise, result, resolve, reject) {
124     if (promise === result) {
125         // 这里的判断用于限制Promise.then不能返回自身
126         throw new TypeError('aaa')
127     }
128     // 使用try..catch也是为了捕获then中出现的错误，只有写了
    catch才会有错误信息输出
129     try {

```



```

130         // 这里的更改主要是针对Promise的内部函数的异步处理
131         // 通过instanceof来判断result是否是通过myPromise实
    现的，从而确定是否是返回的Promise
132         if (result instanceof myPromise) {
133             // 如果是Promise的话，目的还是要改变Promise的状
    态，并且返回值
134             // 此时 result 是一个Promise
135             // 这里相当于重新执行了一次返回的Promise，递归
136             result.then(resolve, reject)
137         } else {
138             // 普通值直接返回
139             resolve(result)
140         }
141     } catch (error) {
142         // then的异步处理中出现error就交给onRejected处理
143         // 同时这里的处理函数从onRejected改为reject，相当于
    把错误代码交给了
144         // 最后一个then来处理
145         reject(error)
146     }
147 }
148 // resolve静态方法
149 static resolve(value) {
150     // 主要还是返回一个Promise
151     return new myPromise((resolve, reject) => {
152         //也是需要判断传入的是不是Promise
153         if (value instanceof myPromise) {
154             // 如果传入的是Promise，那么状态就和传入的
    Promise一样
155             value.then(resolve, reject)
156         } else {
157             // 这种是原生状态，普通值直接返回
158             resolve(value)
159         }
160     })
161 }
162 // reject静态方法实现
163 static reject(value) {
164     // 主要还是返回一个Promise
165     return new myPromise((resolve, reject) => {
166         reject(value)
167     })
168 }
169 // all静态方法，都成功才成功，返回的还是Promise
170 static all(promises) {
171     return new myPromise((resolve, reject) => {
172         const values = []
173         promises.forEach(promise => {
174             promise.then(
175                 value => {
176                     // 每次成功就保存一次值
177                     values.push(value)
178                     if (values.length ===
    promises.length) {
179                         // 如果存入结果的数组长度等于传入
    promise的数组长度，那么就相当于所有的Promise都是成功的
180                         resolve(values)
181                     }

```

```

182         },
183         reason => {
184             // 任何一个传入的Promise的状态为reject
的话就改变返回的all的promise的状态
185             // 上面的resolve就不会继续走了
186             reject(reason)
187         })
188     })
189 })
190
191 }
192 // race静态方法，谁快用谁，不管成功还是失败，返回的也是Promise
193 static race(promises) {
194     return new myPromise((resolve, reject) => {
195         promises.map(promise => {
196             promise.then(
197                 value => {
198                     // 状态只改变一次，所以直接使用就可以
199                     resolve(value)
200                 },
201                 reason => {
202                     reject(reason)
203                 }
204             )
205         })
206     })
207 }
208 }

```

## 题目

```

1  const promise = new Promise((resolve, reject) => {
2      console.log(1)
3      resolve()
4      console.log(2)
5  })
6  promise.then(() => {
7      console.log(3)
8  })
9  console.log(4)
10 //1243

```

```

1  let p1 = new Promise((resolve, reject) => {
2      setTimeout(() => {
3          resolve('success')
4      }, 1000)
5  })
6  let p2 = p1.then(() => {
7      throw new Error('error!!!')
8  })

```

```

9 console.log('promise1', p1)
10 console.log('promise2', p2)
11 setTimeout(() => {
12     console.log('promise1', p1)
13     console.log('promise2', p2)
14 }, 4000)
15 //promise1 Promise {<pending>}
16 //promise2 Promise {<pending>}
17 //Uncaught (in promise) Error: error!!! 此时p1进行完了resolve,
    开始进行then结果报错
18 //promise1 Promise {<fulfilled>: 'success'}
19 //promise2 Promise {<rejected>: Error: error!!! 此时p2已经报错了

```

```

1 let p = new Promise((resolve, reject) => {
2     resolve('success1')
3     reject('error')
4     resolve('success2')
5 })
6
7 p.then((res) => {
8     console.log('then: ', res)
9 }).catch((err) => {
10    console.log('catch: ', err)
11 })
12
13 // then success1

```

```

1 Promise.resolve(1).then((res) => {
2     console.log(res)
3     return 2
4 }).catch((err) => {
5     return 3
6 }).then((res) => {
7     console.log(res)
8 })
9 //1, 2

```

```

1 let p = new Promise((resolve, reject) => {
2     setTimeout(() => {
3         console.log('once')
4         resolve('success')
5     }, 1000)
6 })

```

```

7   let start = Date.now()
8   p.then((res) => {
9       console.log(res, Date.now() - start)
10  })
11  p.then((res) => {
12      console.log(res, Date.now() - start)
13  })
14  //once
15  //1002
16  //1002
17  //Promise.then()可以调用多次，同时所有的then都能获取到res的值，并且输出。

```

```

1   Promise.resolve()
2     .then(() => {
3       return new Error('error!!!')
4     })
5     .then((res) => {
6       console.log('then: ', res)
7     })
8     .catch((err) => {
9       console.log('catch: ', err)
10    })
11  //then: Error: error!!!
12  // return new Error('error!!!') 等价于 return
    Promise.resolve(new Error('error!!!'))

```

```

1   Promise.resolve(1)
2     .then(2)
3     .then(Promise.resolve(3))
4     .then(console.log())
5   //1
6   //then内部期望传入函数，如果 不是函数，就发生值穿透，最后console.log是输出的1
7
8
9   Promise.resolve(1).then(value => {return 2})
10    .then(Promise.resolve(3))
11    .then(console.log)
12  //2

```

```

1   const first = () => (new Promise((resolve, reject) => {
2       console.log(3);
3       let p = new Promise((resolve, reject) => {
4           console.log(7);
5           setTimeout(() => {

```

```

6         console.log(5);
7         resolve(6);
8     }, 0)
9     resolve(1);
10    });
11    resolve(2);
12    p.then((arg) => {
13        console.log(arg);
14    });
15
16    });
17
18    first().then((arg) => {
19        console.log(arg);
20    });
21    console.log(4);
22    //3 7 4 1 2 5

```

```

1    // 定义3个亮灯函数
2    function red() { console.log('red'); }
3    function green() { console.log('green'); }
4    function yellow() { console.log('yellow'); }
5    // 亮灯函数
6    let myLight = (timer, cb) => {
7        // 返回一个Promise
8        return new Promise((resolve) => {
9            // 定时timer毫秒
10           setTimeout(() => {
11               // 定时结束后执行cb函数
12               cb();
13               // 并且更改当前Promise的状态为resolve
14               resolve();
15           }, timer);
16       });
17   };
18   let myStep = () => {
19       Promise.resolve().then(() => {
20           // 因为myLight也是返回一个Promise
21           // 执行之后传入的是red函数和3000ms的定时
22           return myLight(3000, red);
23       }).then(() => {
24           // 这里的then承接的是上面then返回的Promise，在上面
25           // Promise执行完成之后
26           // 才会进入这个then开始执行
27           return myLight(2000, green);
28       }).then(() => {
29           return myLight(1000, yellow);
30       }).then(() => {
31           myStep();
32       })
33   };
34   // 执行函数
35   myStep();

```

```
35  
36 //red  
37 //green  
38 //yellow  
39 //无限递归
```

```
1  const timeout = ms => new Promise((resolve, reject) => {  
2      setTimeout(() => {  
3          resolve();  
4      }, ms);  
5  });  
6  const ajax1 = () => timeout(2000).then(() => {  
7      console.log('1');  
8      return 1;  
9  });  
10 const ajax2 = () => timeout(1000).then(() => {  
11     console.log('2');  
12     return 2;  
13 });  
14 const ajax3 = () => timeout(2000).then(() => {  
15     console.log('3');  
16     return 3;  
17 });  
18 const mergePromise = ajaxArray => {  
19     // 在这里实现你的代码  
20 };  
21 mergePromise([ajax1, ajax2, ajax3]).then(data => {  
22     console.log('done');  
23     console.log(data); // data 为 [1, 2, 3]  
24 });  
25  
26 // 要求分别输出  
27 // 1  
28 // 2  
29 // 3  
30 // done  
31 // [1, 2, 3]
```

请实现一个mergePromise函数，把传进去的数组按顺序先后执行，并且把返回的数据先后放到数组data中。

## 网络请求

### HTTP

HyperText Transfer Protocol（超文本传输协议）简称HTTP，

# HTTP的请求过程

1. DNS解析，把域名解析成ip地址
2. 建立TCP连接，三次握手
3. 服务器接收数据，并处理，返回
4. 客户端接收数据，进行页面渲染什么的

## 请求报文

由四部分组成，

1. 请求行
2. 请求头
3. 空行
4. 请求体

下面就是一个http请求的一个例子

```
1 POST /test.html HTTP/1.1
2 Host: 127.0.0.1:8848
3 Connection: keep-alive
4
5 username=admin&password=null
```

## 请求头

格式如下

```
1 请求方法 URL HTTP版本
```

比如

```
1 POST /test.html HTTP/1.1
```

请求的方法为POST方法，访问的URL为/test.html，使用的协议版本为HTTP/1.1

## 请求行

## 请求体

## 响应报文

由四部分组成，

1. 响应行
2. 响应头
3. 空行
4. 响应体

```
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 x-timestamp: 1603966982798
```

```
4 x-sent: true
5 Accept-Ranges: bytes
6 Cache-Control: public, max-age=0
7 Last-Modified: Thu, 29 Oct 2020 10:07:19 GMT
8 Date: Thu, 29 Oct 2020 10:23:02 GMT
9 ETag: W/"146-17573d375e3"
10 Content-Type: text/html; charset=UTF-8
11 content-length: 615
12
13 <!DOCTYPE html>
14 <html>
15     <head>
16         <style>
17             #out{
18                 perspective: 500px;
19             }
20             #in{
21                 width: 100px;
22                 height: 100px;
23                 background-color: #BC8F8F;
24                 transform: translate3d(0,0,60px);
25             }
26         </style>
27     </head>
28     <body>
29         <div id="out">
30             <div id="in">
31                 123
32             </div>
33         </div>
34         <script>document.write('<script src="//' + (location.host
|| 'localhost').split(':')[0] + ':35929/livereload.js?
snipver=1"></' + 'script>')</script>
<script>document.addEventListener('LiveReloadDisconnect',
function() { setTimeout(function() { window.location.reload();
}, 500); })</script></body>
35 </html>
```

可以看到，响应体直接就是一个网页源码，浏览器就是用这个来获取网页的。

## 响应头

格式如下

1 | HTTP版本 状态码 原因

## 响应行

## 响应体

## URL的结构

## 跨域



- 浏览器有同源策略
- 所谓的域，就是说协议，ip和端口号都要一致。

但是这时候问题就来了，比如 `login\hat-soft.top` 和 `index\hat-soft.top` 不同源，但是域名是一样的

## Ajax

Asynchronous JavaScript and XML

用于前端页面向后台请求数据的

### 基础结构

//设置响应类型

`xhr.responseType='json'`

```
1 //1. 创建xhr对象
2 const xhr = new XMLHttpRequest()
3 //2. 初始化，设置请求方式和url
4 xhr.open('POST', 'http://127.0.0.1:3000/')
5 //3. 请求体内容
6 xhr.send()
7 //4. 返回后端结果
8 xhr.onreadystatechange=()=>{
9   //一共右4个state，直到4时，代表服务器把结果全部返回了
10   if(xhr.readyState===4){
11     if(xhr.status>=200 && xhr.status<300)
12       console.log(reponse);
13   }
14 }
```

### GET/POST请求

```
1 xhr.open('GET', 'http://127.0.0.1:3000/')
2 xhr.open('POST', 'http://127.0.0.1:3000/')
```

### 设置响应类型

```
1 const xhr = new XMLHttpRequest()
2 xhr.open('GET', 'http://127.0.0.1:3000/')
3 xhr.send()
4 //设置响应类型
5 xhr.responseType='json'
6 xhr.onreadystatechange=()=>{
7   if(xhr.readyState===4){
8     if(xhr.status>=200 && xhr.status<300)
9       //可以直接用对象的方式去访问json数据
10       ctn.innerText=xhr.response.name
11   }
12 }
```

## 网络请求超时

```

1  const xhr = new XMLHttpRequest()
2  //超时设置，单位为毫秒
3  //服务器如果超过这个时间没有响应，那么请求自动取消
4  xhr.timeout=2000;
5  //超时回调函数，如果服务器超时响应超时则调用。
6  xhr.ontimeout=function(){
7      alert("网络异常!!!")
8  }
9  xhr.open('GET', 'http://127.0.0.1:3000/')
10 xhr.send()
11 xhr.onreadystatechange=()=>{
12     if(xhr.readyState==4){
13         if(xhr.status>=200 && xhr.status<300)
14             ctn.innerText=xhr.response
15     }
16 }

```

## 取消网络请求

### 常用来防止请求重复发送

```

1 <!DOCTYPE html>
2 <html lang="ch">
3
4 <head>
5   <meta charset="UTF-8">
6   <title>Ajax测试</title>
7   <style>
8     #content {
9       margin-top: 10px;
10      width: 300px;
11      height: 200px;
12      border: 1px black solid;
13    }
14  </style>
15  <script>
16    window.onload = function () {
17      const start = document.getElementById("start")
18      const content = document.getElementById("content")
19      let xhr = new XMLHttpRequest()
20      let isSending = false
21      //正常申请页面
22      start.onclick = () => {
23        if(isSending==true)
24          xhr.abort()
25        xhr.open('GET', 'http://127.0.0.1:3000/')
26        xhr.send()
27        xhr.onreadystatechange = () => {
28          if (xhr.readyState == 4) {
29            isSending=false
30            if (xhr.status >= 200 && xhr.status < 300)
31              content.innerText = xhr.response

```

```

32         }
33     }
34 }
35 }
36 </script>
37 </head>
38
39 <body>
40   <button id="start">加载页面</button>
41   <div id="content"></div>
42 </body>
43
44 </html>

```

## Fetch

### GET

```

1 fetch('https://dog.ceo/api/breeds/image/random')
2   .then(response => response.json())
3   .then(data => void console.log("收到的数据URL
  为",data.message))

```

### POST

```

1 fetch('https://jsonplaceholder.typicode.com/users',{
2   method : 'POST',
3   headers:{
4     'Content-Type': 'application/json'
5   },
6   body:JSON.stringify({name:"kamren"})
7 })
8 .then(response=>response.json())
9 .then(data=>console.log(data))

```

## <http://httpbin.org/>

这个网站可以用来模拟各种响应

## Axios

### 安装

```

1 npm install axios --save

```

```
1 axios({
2   url: '',
3   method: 'post',
4   params: {}
5   //请求体参数
6   data: {}
7   //请求头
8   headers: {}
9 }).then(response => {
10   console.log('请求结果: ', response);
11 });
```

## 事件循环（Event Loop）

JavaScript中有调用栈（Call Stack）、消息队列（Message Queue）和微任务队列（Microtask Queue）组成，

在js中，函数调用会压入栈，如果遇到异步操作如`setTimeout`，会将里面的内容输入到消息队列，变成消息；而遇到`Promise`，则会压入到微任务队列。**只有调用栈被清空，才能去执行微任务队列的内容；只有执行完了微任务，才会执行消息队列。**这就是为什么`setTimeout`的时间只是最小延迟。

所以我们可以将JavaScript代码分为三种：

- 消耗很多资源的**宏任务**，比如 `setTimeout`
- 消耗较少资源的**微任务**，比如 `Promise.then`，注意!!! 是`promise.then`是微任务，`promise`本身不是微任务
- 可以立即执行的**同步任务**

js在事件循环的时候，首先会以此扫过代码，如果是同步任务就立刻执行，如果是微任务就放到微任务队列中，如果是宏任务就放到消息队列中。

然后执行微任务队列。执行完毕之后，进行浏览器的渲染，进行数据更新。最后才进行宏任务的执行。

## 循环与迭代

### label循环

`label`不仅仅可以用于循环，也可以用于任何一个地方。只要是块就可以用，通过`break`来结束

比如：

```

1 test: {
2     console.log('test1')
3     break test;
4     console.log('test2')
5 }
6 //test1

```

代码运行到 `break test` 的时候，会跳转到test，然后把整个代码块都跳过

在循环中，label可以发挥强大的力量

```

1 var num = 0;
2 test: {
3     for (let i = 0; i < 10; i++) {
4         for (let j = 0; j < 10; j++) {
5             if (i == 5 && j == 5) {
6                 break test; //不加标签的话就是95
7             }
8             num++
9         }
10    }
11 }
12 console.log(num) //55
13
14

```

本来break只能break里面那一层循环，但是一旦使用了label，就会把整个循环全部跳出去

同样的

```

1 var num = 0;
2 test:
3 for (let i = 0; i < 10; i++) {
4     for (let j = 0; j < 10; j++) {
5         if (i == 5 && j == 5) {
6             continue test; //不加标签的话就是99
7         }
8         num++
9     }
10 }
11
12 console.log(num) //95

```

## for.....in (对象)

这种语法可以遍历对象的键。

```
1 var test = {age:10,name:"lol"};
2 for (let key in test){
3   console.log(key+":"+test[key]);
4 }
5 //age:10
6 //name:lol
```

如果对象是数组，那么就会返回数组的键。

```
1 var a = [1,2,3,4,5];
2 for(let i in a)
3   console.log(i)
4 //0
5 //1
6 //2
7 //3
8 //4
```

这是为什么？为什么for.....in可以遍历对象的键？

## for.....of (数组)

首先来看看这个代码

```
1 var a = [1,2,3,4,5,"你好"]
2 for(let i of a)
3   console.log(i)
```

结果会直接输出a数组的元素。这看上去平淡无奇，但是我们对一个对象使用for.....of会怎么样呢？

```
1 var test = {age:10,name:"lol"};
2 for(let i of test)
3   console.log(i)
4 //Uncaught TypeError: test is not iterable
```

一切的一切都是因为**迭代器 (iterator)** 的缘故。

for.....of遍历的本质就是 使用数组内部的迭代器来遍历其数据，而普通的对象之所以无法使用for.....of就是因为普通的对象没有迭代器。

我们简单来看一下for.....of的原理

```

1 | var array = [1,2,3]
2 | var it = array[Symbol.iterator]() //数组自带Symbol.iterator属性，并且该属性还是一个函数
3 | console.log(it.next()) //{ value: 1, done: false }
4 | console.log(it.next()) //{ value: 2, done: false }
5 | console.log(it.next()) //{ value: 3, done: false }
6 | console.log(it.next()) //{ value: undefined, done: true }

```

value是当前的值，done用来表示遍历是否结束。

现在我们既然明白了原理，也不妨给对象也写一个类的迭代器。

核心就是需要给对象设定一个 `Symbol.iterator` 属性。

```

1 | var loli = {
2 |   name: '86',
3 |   age: 14
4 | }
5 | Object.defineProperty(loli, Symbol.iterator, {
6 |   enumerable: false,
7 |   writable: false,
8 |   configurable: true,
9 |   value: function () {
10 |     var o = this;
11 |     var index = 0;
12 |     var keys = Object.keys(o);
13 |     return {
14 |       next: function () {
15 |         return {
16 |           value: o[keys[index++]],
17 |           done: (index > keys.length)
18 |         };
19 |       }
20 |     };
21 |   }
22 | })
23 |
24 | for (let i of loli)
25 |   console.log(i)

```

也可以使用生成器，

```

1  var lolli = {
2      name: '86',
3      age: 14,
4      *[Symbol.iterator]() {
5          for(let key in lolli)
6              yield "key:"+key + " value:"+lolli[key]
7      }
8  }
9
10
11  for (let i of lolli)
12      console.log(i)

```

可以看到，代码简单了不少，这就是语法糖的奥妙。

## 迭代器

首先来自制一个迭代器吧。

```

1  function myIterator(arr) {
2      var i = 0;
3      return {
4          next: function () {
5              return {
6                  value: arr[i++],
7                  done: i >= arr.length ? true : false
8              }
9          }
10     }
11 }
12
13 var it = myIterator([1, 2, 3, 4, 5])
14 console.log(it.next())
15 console.log(it.next())
16 console.log(it.next())
17 console.log(it.next())
18 console.log(it.next())
19 console.log(it.next())

```

## 生成器

如果在函数名前面添加星号，就可以创建生成器。

生成器里面会使用 yield 关键字。

```

1  function *createIterator(array) {
2      for(let i =0 ;i<array.length;i++){
3          yield array[i]
4      }
5  }
6
7  let it = createIterator([1,3])
8  console.log(it.next())
9  console.log(it.next())
10 console.log(it.next())

```



yield其实就是语法糖，引擎会自动帮你封装一个迭代器。只需要每次yield一个值即可。

**注意！yield只能在生成器内部使用！**就算是生成器内部的函数也不行。

```
1 function *createIterator(array) {
2     array.forEach(element => {
3         yield element
4     });
5 }
6
7 let it = createIterator([1,3])
8 console.log(it.next())
9 console.log(it.next())
10 console.log(it.next())
11 //SyntaxError
```

生成器不需要返回值，如果写了的话，将会被视为最后一个值。

```
1 function * myGenerator(){
2     yield 1
3     yield {value: undefined, done: true}
4     yield 3
5     return '最后一个值'
6 }
7 var it = myGenerator()
8 console.log(it.next()) //{value: 1, done: false}
9 console.log(it.next()) //{value: 2, done: false}
10 console.log(it.next()) //{value: 3, done: false}
11 console.log(it.next()) //{value: '最后一个值', done: true}
12 console.log(it.next()) //{value: undefined, done: true}
```

## forEach

对象

```
1 var test = {age:10,name:"loli"};
2 Object.keys(test).forEach(key => {
3     console.log(key)
4 });
```

数组

```
1 var a =[1,2,3,4,5]
2 //里面传一个回调函数
3 a.forEach((num)=>{
4     console.log(num)
5 })
6 //1 2 3 4 5
```

# 控制台调试

## 打印输出

### 占位符与样式

- `%s` 字符串

```
1 let str = 'hello '  
2 console.log("%s world",str);
```

- `%d` 或 `%i` 整数
- `%f` 浮点数
- `%o` 对象的链接

```
1 let loli = {  
2   name : 'yoshino'  
3 }  
4 console.log("小萝莉%o",loli.name);
```

- `%c` CSS格式字符串 如果方法的第一个参数中使用了占位符，那么就依次使用后面的参数进行替换。

```
1 console.log("%c我是红色","color:red");  
2 console.log("我是%c绿色","color:#bbffcc");
```

在ES6之后，可以通过模板字符串来代替一些占位符

### 打印的方法

```
1 console.log("hello world")  
2 console.info("hello world")  
3 console.warn("hello world")  
4 console.error("hello world")
```

### 清除控制台

```
1 console.clear()
```

## 断言输出

接收2个参数，第一个为断言条件，第二个为断言信息，只有当断言条件为false时，才会输出断言信息。

```
1 console.assert(true, 'true')
2 console.assert(false, 'false')
```

客户端的 `console.assert()` 打印断言，并不会阻塞后续代码的执行，只是在断言的表达式为false的时候，向控制台打印你的内容。

而在 `node.js` 中，值为假的断言将会导致一个 `AssertionError` 被抛出，使得代码执行被打断。这两者是有区别的。

## 打印计数

有时，我们需要统计一段代码的执行次数，那么就可以使用这个方法。

```
1 for (let i = 0; i < 30; i++) {
2   console.count("index");
3 }
4 console.countReset("index");//重启计数器
5
6 console.count("index2");//根据不同的字符串来标记不同的计数器。
```

如果不传参的话默认输出default

```
1 console.count();
```

## 计时

这个方法可以跟踪并输出执行代码所需要的时间。

```
1 console.time('test');
2 //代码
3 console.timeEnd('test');
```

- 页面中最多能同时运行10,000个计时器
- 该方法并不会将结算结果返回到js中，而只是能打印在控制台上。所以不能使用此方法在js中来作为普通计时器使用或者进行性能收集器的一部分。
- 没有传参的话，默认是default

## 以对象形式输出

可以查看某个对象或者变量的具体信息，实际上在控制台直接输出这个对象也可以。

```
1 let loli = {name:"莲华",age:14};  
2 console.dir(loli)
```

这个方法基本上和 `console.log` 差不多，但是在打印DOM元素的时候，`console.log` 会打印DOM元素本身，而 `console.dir()`

会打印其DOM对象

```
1 <body>  
2   <div id="test"></div>  
3  
4 </body>  
5 <script>  
6   var test = document.getElementById("test")  
7   console.log(test)  
8   console.dir(test)  
9 </script>
```

---

```
<div id="test"></div>
```

---

```
▼ div#test ⓘ  
  accessKey: ""  
  align: ""  
  ariaAtomic: null  
  ariaAutoComplete: null  
  ariaBusy: null  
  ariaChecked: null  
  ariaColCount: null  
  ariaColIndex: null  
  ariaColSpan: null  
  ariaCurrent: null  
  ariaDescription: null  
  ariaDisabled: null  
  ariaExpanded: null  
  ariaHasPopup: null  
  ariaHidden: null  
  ariaKeyShortcuts: null  
  ariaLabel: null  
  ariaLevel: null  
  ariaLive: null  
  ariaModel: null
```

## 分组输出

---

只要把代码包在group里面，就可以对控制台的信息进行分组管理。

```
1 console.group('1')
2   console.log("a")
3   console.log("b")
4   console.log("c")
5 console.groupEnd()
```

## 表格输出

---

把一个对象以表格的方式输出。

```
1 console.table(loli)
```

## 打印函数调用栈

---

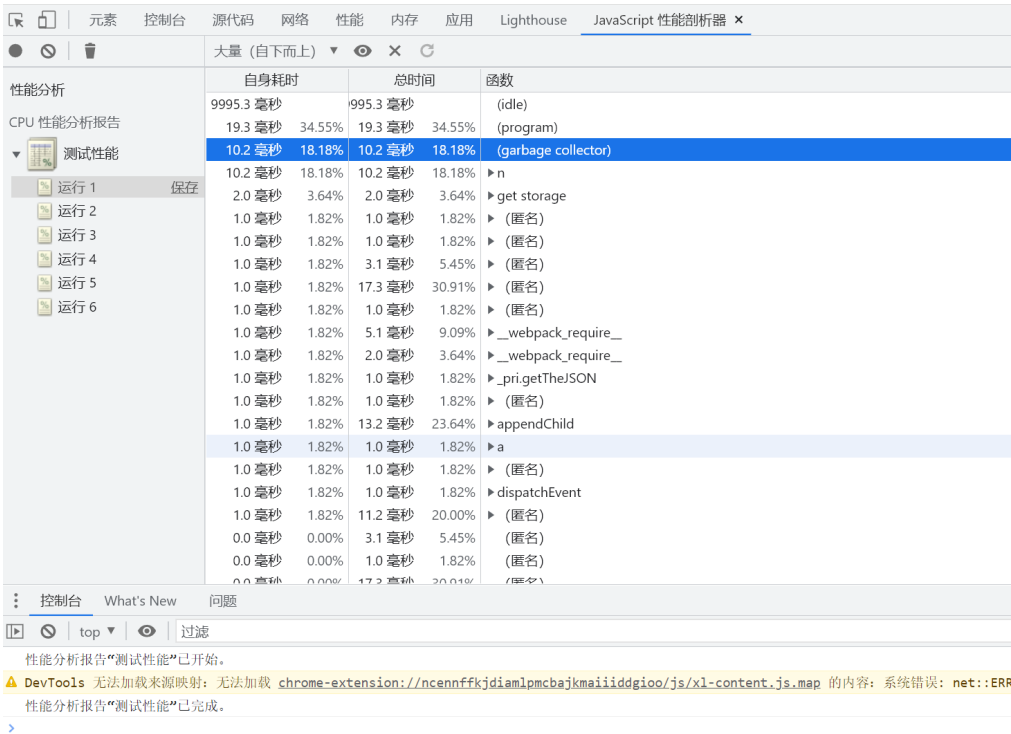
console.trace()

```
1 function test() {
2   test2()
3 }
4
5 function test2(){
6   console.trace()
7 }
8
9
10 test()
```

## 性能分析

---

console.profile()



## 命令菜单

1. 按F12，打开控制台
2. 按ctrl+shift+p，打开命令菜单

命令		
screenshot	截图	

点击一个元素，HTML右边会出现\$0 的符号。然后在console中输入\$0，可以打印这个DOM元素。

## 复制JSON

```
1 let lolli = {name:"莲华",age:14};
2 copy(lolli);
```

此时这个JSON对象就被复制到了你的复制粘贴缓存区中了，可以直接把这个JSON文本粘贴到任意地方，非常方便。

## 强制刷新

按住shift+F5

## copy函数

---

```
1 | let a = 1
2 | c
```

## 附录1

---

### 弹窗

---

```
1 | function showDialog(content) {
2 |     var layer = document.createElement("div");
3 |     layer.id = "layer";
4 |     var style = {
5 |         background: "#ccc",
6 |         position: "absolute",
7 |         zIndex: 10,
8 |         width: "auto",
9 |         height: "40px",
10 |        left: "50%",
11 |        top: "0%",
12 |        marginLeft: "-100px",
13 |        marginTop: "10px",
14 |        padding: "4px 10px"
15 |     }
16 |     for (var i in style) {
17 |         layer.style[i] = style[i];
18 |     }
19 |     if (document.getElementById("layer") == null) {
20 |         document.body.appendChild(layer);
21 |         layer.innerHTML = content;
22 |         layer.style.textAlign = "center";
23 |         layer.style.lineHeight = "40px";
24 |         setTimeout("document.body.removeChild(layer)", 3000)
25 |     }
26 | }
27 |
```

## 字符串常用方法

---

### 字符串分割为数组 split()

这个函数会把字符串直接按照指定字符截断，并存入数组之中。

```

1 | var string = "1 2 3 4 5";
2 | var array = string.split(" ");
3 | array.forEach(
4 |     function(num){
5 |         console.log(num)
6 |     }
7 | );
8 | //1
9 | //2
10 | //3
11 | //4
12 | //5

```

es6 新增字符串方法

- includes()
- startWith()

```

1 | //字符串开头字母之后的字符串是否为h
2 | str.startsWith("h");
3 | //字符串开头第2个字母之后的字符串是否为ello
4 | str.startsWith("ello",1);

```

- endWith()
- toUpperCase()/toLowerCase()

```

1 | //可以把str里面的字母全部变成大写/小写，但是不会改变a本身的数据
2 | str.toUpperCase()
3 | str.toLowerCase()

```

## 替换字符串

```

1 | String replace(char oldChar, char newChar)

```

# 附录2——JavaScript对象方法速查

## Array

- splice  
从数组中添加或删除元素。
- sort  
排序



a-b是从小到大

```
1 var a = [1,123,1232,3345345,412,5].sort((a,b)=>a-b);
2 console.log(a) //[1, 5, 123, 412, 1232, 3345345]
```

- 把数组的内容拼接成一个字符串

默认情况下，会把拼接的字符串用逗号间隔

```
1 var fruits = ["Banana", "Orange", "Apple", "Mango"];
2 var energy = fruits.join();
3 //'Banana,Orange,Apple,Mango'
```

也可以指定分隔符

```
1 var fruits = ["Banana", "Orange", "Apple", "Mango"];
2 var energy = fruits.join('和');
3 //'Banana和Orange和Apple和Mango'
```

- 拼接两个数组

```
1 var loli1 = ["雷姆","伊莉雅","波莱特","巧克力"];
2 var loli2 = ["86","香子兰","牛顿"];
3 console.log(loli1.concat(loli2));
```

- 数组元素反转

```
1 var a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
2 a.reverse();
3 console.log(a)
```

## 排序

```
1 var a = [1,3,5,2,-1,34,566,0,23];
2 a.sort(function(a,b){
3     return a-b; //升序排序
4 });
5 console.log(a);
```

## every

每一项都要符合

```
1 var a = [0,1,2,3,4,5,6,7]
2 var result = a.every(value=> value<10)
3 console.log(result) //true
```

## some

```
1 var a = [9000,100,200,300,400,500,600,7]
2 var result = a.some(value=> value<10)
3 console.log(result) //true
```

## flat [ES10]

扁平化数组

```
1 var test = [
2   1,2,3,
3   [1,2,3],
4   [[1],[2,3]],
5   [[[1,2,3]]]
6 ]
7 console.log(test.flat())
8 console.log(test.flat().flat())
9 console.log(test.flat(Infinity))
```

## flatMap [ES10]

flat是把一个多维数组扁平化，而flatMap 是把对数组处理的结果扁平化

```
1 var test = [1,2,3,4,5]
2 console.log(test.map(value=>[value,value*2]))
3 //0: (2) [1, 2]
4 //1: (2) [2, 4]
5 //2: (2) [3, 6]
6 //3: (2) [4, 8]
7 //4: (2) [5, 10]
```

如果想要对结果扁平化的话

```
1 var test = [1,2,3,4,5]
2 console.log(test.flatMap(value=>[value,value*2]))
3 //(10) [1, 2, 2, 4, 3, 6, 4, 8, 5, 10]
```

## Object

---

# Object.keys()

- 返回可枚举的属性

```
1 var obj = {
2   a:1,
3   b:2
4 }
5 Object.defineProperty(obj, 'c', {
6   value:3,
7   enumerable:false
8 })
9 console.log(Object.keys(obj)) //a,b
```

- 继承属性也可以返回

```
1 class A {
2   constructor() {
3     this.a = 1
4   }
5 }
6 class B extends A {
7   constructor(){
8     super()
9     this.b =2
10  }
11 }
12
13 var b = new B();
14 console.log(Object.keys(b)) //['a', 'b']
```

- 返回的顺序

大于0的数整数，排序

负数和小数不排序

字符串按照给定的顺序排序

Symbol不显示

```
1 var obj = {
2   1:1,
3   3:3,
4   "s":"s",
5   2:2,
6   [Symbol(123)]:123,
7   98:98,
8   6:6,
9   "G":"'G'"
10 }
11 console.log(Object.keys(obj))
12 // ['1', '2', '3', '6', '98', 's', 'G']
```

- 传入null或者undefined报错。因为这两个默认转型成对象会报错

```
1 | Object.keys(null)
2 | Object.keys(undefined)
3 | //VM140:1 Uncaught TypeError: Cannot convert undefined
   | or null to object
```

- 传入数字

数字会默认进行包装类的转换，变成Number对象，而Number对象没有key，所以为空数组

```
1 | Object.keys(123)//[]
```

- 传入字符串

同样进行包装类的转换，而字符串的键是下标

```
1 | Object.keys('1231asd')// ['0', '1', '2', '3', '4', '5',
   | '6']
```

## Date

- 现在是几几年

```
1 | (new Date()).getFullYear()
```

- 现在是几月份

从0到11。

```
1 | (new Date()).getMonth()
```

- 今天是几号

从1到31

```
1 | (new Date()).getDate()
```

- 现在是礼拜几

周日为0，周一是1

```
1 | (new Date()).getDay()
```

- 现在是几点

使用24小时制

```
1 | (new Date()).getHours()
```

- 当前是几分钟

```
1 | (new Date()).getMinutes()
```

- 现在是几秒

```
1 | (new Date()).getSeconds()
```

- 现在是几毫秒

```
1 | (new Date()).getMilliseconds()
```

## String

- 字符串拼接

```
1 | str1.concat(str2);
```

- 字符串切片

```
1 | var str="Hello world!";  
2 | var n=str.slice(1,5);  
3 | //n  
4 | //ello
```

- 将字符串分割为字符串数组

```
1 | var str="How are you doing today?";  
2 | var n=str.split(" ");  
3 |  
4 | //['How', 'are', 'you', 'doing', 'today?']
```

- 去除字符串两边的空白

```
1 | var str = "      Runoob      ";  
2 | str.trim();  
3 |  
4 | //Runoob
```