

# 大前端面试小册

作者：小鹿

首发于公众号：小鹿动画学编程

## 版权声明

本电子书由公众号「小鹿动画学编程」原创作者小鹿整理而成，未经允许授权，禁止任何形式发布其他平台。

## 更新日志

- 2020/5/25 发布第一版本 主要是 javascript 和 ES6 核心内容。

## 阅读说明

本电子书会对内容不定时进行更新，获取最新版本，关注小鹿公众号：[小鹿动画学编程](#)，回复：[「前端PDF」](#) 即可获取最新更新版本。



## 目录

### 大前端面试小册

版权声明

更新日志

阅读说明

目录

JavaScript

## 数据类型

知识线:

- 1、堆栈池内存
  - 大厂面试题
- 2、基本类型
- 3、类型转换
  - 3.1 数字类型转换(Number)
    - 3.1.1 `Number`
    - 3.1.2 `parseInt()/parseFloat()`
    - 3.1.3 大厂面试题:
  - 3.2 字符串类型转换(String)
  - 3.3 布尔类型转换 (Boolean)
  - 3.4 原始类型转换
- 4、数据类型检测
  - 4.1 `typeof`
  - 4.2 `instanceof`
  - 4.3 `constructor`
  - 4.4 `Object.prototype.toString.call()`
- 5、各种运算
  - 5.1 四则运算
    - 加法运算
    - 其他运算
  - 5.2 逻辑运算符(&& ||)
    - 条件判断
    - 赋值操作
  - 5.3、比较运算符
- 6、`i++/++i`
  - 6.1 面试题一
  - 6.2 面试题二

## this

知识线

- 1、什么是 this?
- 2、this 的指向?
- 3、如何改变 this?
- 4、call、apply、bind 实现
  - 4.1 手写call
  - 4.2 手写 apply
  - 4.3 手写 bind

## new

知识线

- 1、创建对象的几种方式?
  - 1.1 字面量
  - 1.2 `new`
    - 1.2.1 返回值
    - 1.2.2 `in` : 检测对象属性 (私有 + 共有)
    - 1.2.3 `hasOwnProperty` (私有)
  - 1.3 `Object.create(null)`
  - 1.4 三者创建对象的区别

## 闭包

知识线

- 1、作用域和作用域链
- 2、执行栈和执行上下文
  - 2.1 执行上下文
  - 2.2 执行栈
  - 2.3 小结
- 3、闭包
  - 3.1 什么是闭包?

### 3.2 闭包的作用

#### 3.2.1 保护

#### 3.2.2 保存

### 3.4 大厂经典面试题

## 原型和原型链

### 知识线

#### 1、原型的前世今生

#### 2、什么是原型/原型链?

##### 2.1 原型

##### 2.2 原型链

##### 2.3 大厂面试题

## 继承

### 知识线

#### 1、原型继承

#### 2、组合继承

#### 3、寄生组合继承

#### 4、ES6 的 extend 继承

## 垃圾回收机制

### 知识线

#### 1、什么是内存泄漏?

#### 2、为什么会发生内存泄漏?

#### 3、垃圾回收机制

##### 3.1 两种垃圾回收策略

##### 3.2 标记清除

##### 3.3 引用计数

##### 存在的缺陷

#### 4、如何管理内存

## 面向对象

### 知识线

#### 1、什么是面向对象?

#### 2、面向对象编程 (OOP)

##### 2.1 构造函数模式

##### 2.1.1 构造函数的 new

##### 2.1.2 Object 的创建方式

##### 2.2 单例模式

##### 2.2.1 由来

##### 2.2.2 高级单例模式

#### 3、封装、继承、多态

##### 3.1 封装

##### 3.2 继承

##### 3.3 多态

## 深拷贝和浅拷贝

### 知识线

#### 1、为什么进行拷贝?

#### 2、浅拷贝

##### 2.1 自己实现一个浅拷贝

##### 2.2 扩展运算符实现

##### 2.3 `Object.assign(target, source)` 实现

#### 3、深拷贝

##### 3.1 手动实现 (最简单实现)

##### 3.2 存在的问题

##### 3.2 `JSON.stringify()`

## 数组去重

### 知识线

#### 1、最 Low 的两种方案

##### 1.1 方案一

##### 1.2 方案二

#### 2、最优秀的两种方案

2.1 基于对象去重

2.2 基于 Set 去重

## 异步编程

### 知识线

1、JS 为什么是单线程？

2、单线程带来的问题？

3、如何实现异步编程？

3.1 回调函数

3.1.1 为什么不能捕获异常？

3.1.2 为什么不能 return？

3.2 解决回调地狱问题

4、Event Loop

4.1 执行上下文

4.2 执行栈

4.3 宏任务

4.4 微任务

4.5 定时器 `setTimeout`

## ES6 基础知识点

### var/let/const

#### 知识线

1、变量提升

1.1 为什么存在变量提升？

1.2 变量和函数提升的规则？

1.3 var、let、const

### map、filter、reduce

#### 知识线

1、map

2、filter

3、reduce

## 代理

### 知识线

3.1 字面量定义

3.2 ES6 中的 Class 定义

3.3 `Object.defineProperty()`

3.4 Proxy 代理

## ES6/7 的异步编程

### 知识线

1、Generator

2、Promise

为什么使用 Promise

2.1 Promise 简单使用

2.2 Promise 的状态

2.3 Promise 链式调用

2.3 Promise 并行请求

2.4 如何实现一个 Promise

3、async 及 await

3.1 特点

3.2 `async/await` 的实现原理

## 模块化

### 知识线

1、为什么使用模块化？

2、使用模块化的几种方式？

2.1 方式一：函数

2.2 方式二：立即执行函数

3、方式三：CommonJS 规范

3.1 暴露模块

3.2 引入模块

3.3 实现原理

- 3.4 `CommonJS` 的特点
- 4、方式四：AMD 和 CMD
  - 4.1 AMD
    - 4.1.1 使用方式
    - 4.1.2 导出模块
    - 4.1.3 引入模块
  - 4.2 CMD
    - 4.2.1 使用方式
    - 4.2.2 AMD 和 CMD 的区别
- 5、方式五：ES6 Module
  - 5.1 `export` 导出模块
  - 5.2 `import` 导入模块
  - 5.3 `ES6` 和 `CommonJS` 的区别

第二版正在书写中 ...

# JavaScript

## 数据类型

### 知识线：

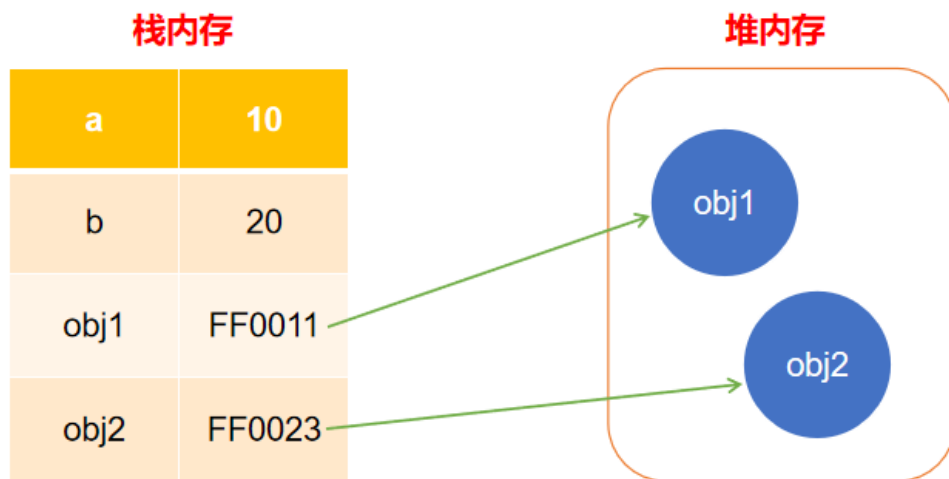
堆栈内存 => 原始类型/引用类型（6个） => 类型特点 / 类型转换 => 数据类型检测（4种） => 各种运算（四则运算、逻辑运算符、比较运算符、`i++/++i`区别） => 大厂笔试题解析

面试官：说说 JavaScript 中的基本类型有哪些？以及各个数据类型是如何存储的？

## 1、堆栈池内存

- 栈：存放变量
- 堆：存放复杂对象
- 池：存放常量

栈（Stack）和堆（Heap），是两种基本的数据结构。Stack 在内存中自动分配内存空间的；Heap 在内存中动态分配内存空间的，不一定会自动释放。一般我们在项目中将对象类型手动置为 `null` 原因，减少无用内存消耗。

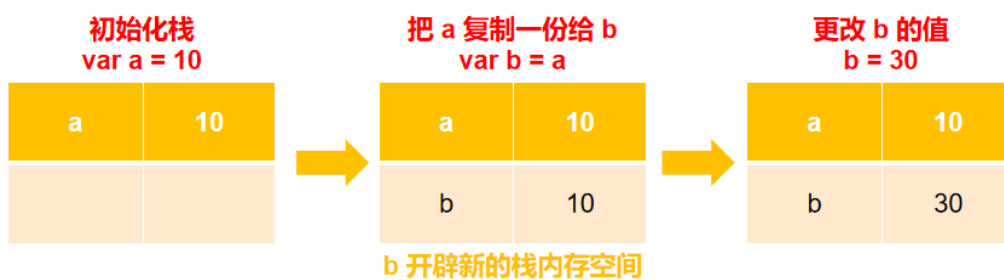


@小鹿动画学编程

原始类型是按值形式存放在**栈**中的数据段，内存空间可以自由分配，同时可以**按值直接访问**。

```
var a = 10;
var b = a;
b = 30;
console.log(a); // 10值
console.log(b); // 30值
```

过程图示：

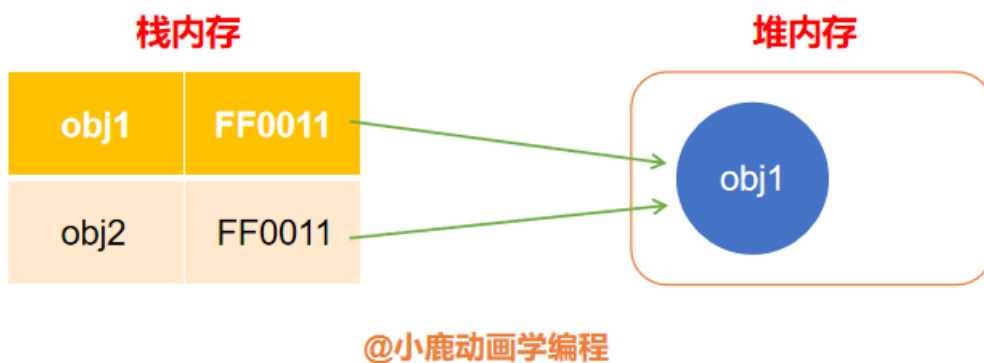


@小鹿动画学编程

引用类型是存放在**堆**内存中，每个对象在堆内存中有一个引用地址，就像是每个房间都有一个房间号一样。引用类型在栈中保存的就是这个对象在堆内存的引用地址，我们所说的“房间号”。通过“房间号”可以快速查找到保存在堆内存的对象。

```
var obj1 = new Object();
var obj2 = obj1;
obj2.name = "小鹿";
console.log(obj1.name); // 小鹿
```

过程图示：



## 大厂面试题

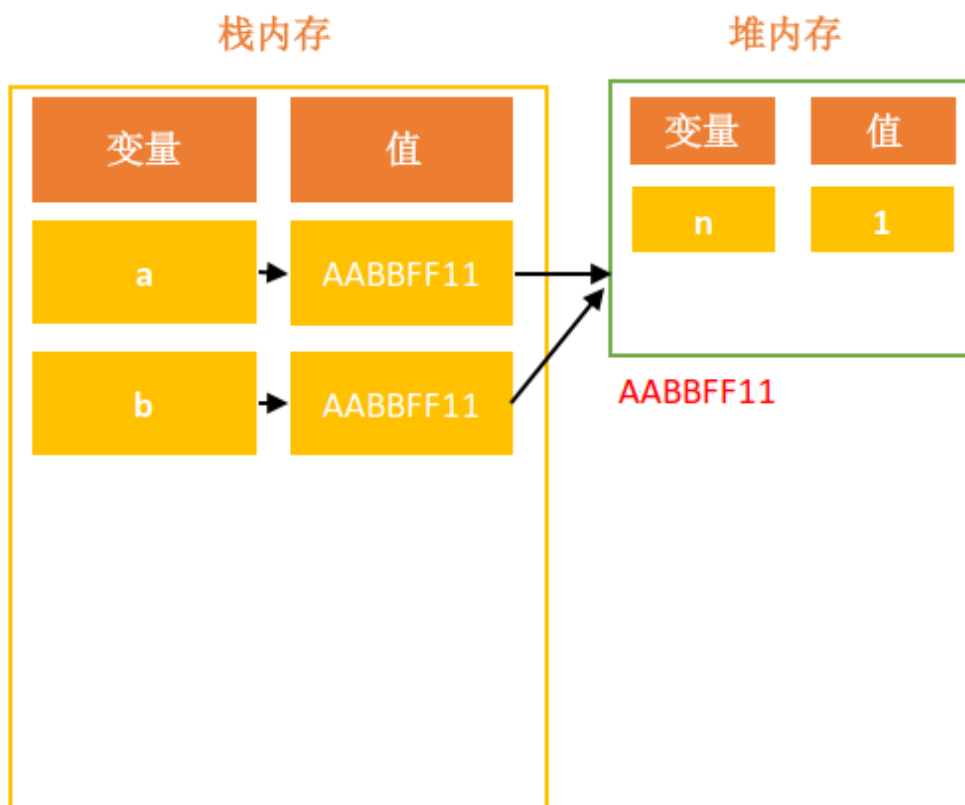
写出以下程序运行的结果。

```
let a = {  
  n:1  
}  
let b = a:  
a.x = a = {  
  n:2  
}  
console.log(a.x)  
console.log(b)
```

题目分析：

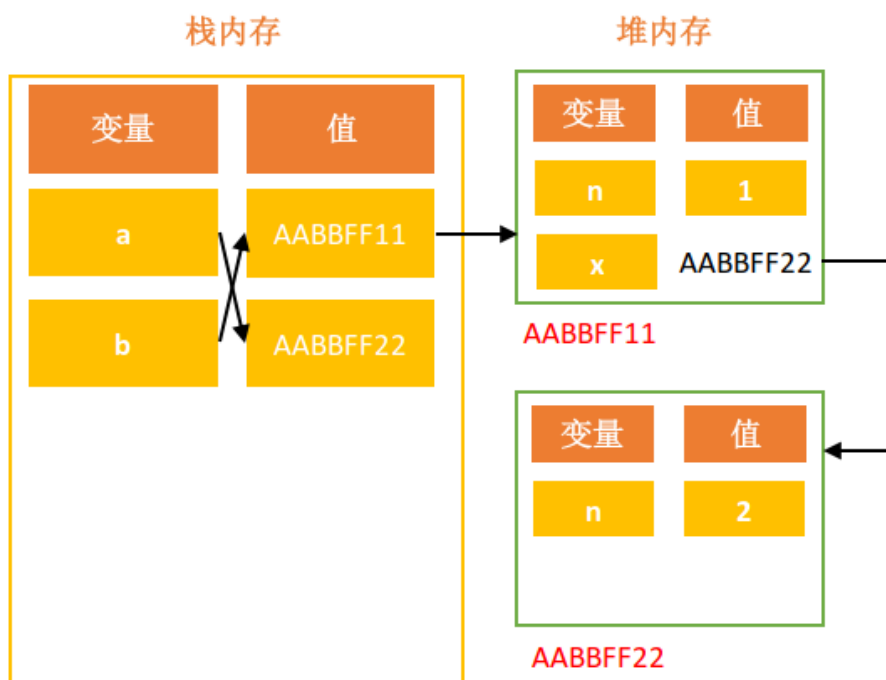
- 首先，代码执行，开辟栈内存，将变量和值分别存储到栈中(一一对应)
- 如果是存储的引用类型，开辟堆空间，栈中变量存储的是相对应的引用类型堆空间的地址
- 主线程自上而下依次执行进栈执行代码，执行完毕出栈

示意图：变化前



公众号：小鹿动画学编程

变化后：a.x = a = {n:2}



公众号：小鹿动画学编程



## 2、基本类型

`javascript` 的数据类型包括 **原始类型** 和 **引用类型(对象类型)**。

原始类型包括以下 6 个：

- `String`
- `Number`
- `Boolean`
- `null`
- `undefined`
- `Symbol`

引用类型统称为 `Object` 类型，如果细分的话，分为以下 5 个：

- `Object`
- `Array`
- `Date`
- `RegExp`
- `Function`

## 3、类型转换

类型转换通常在面试笔试中出现的比较多，对于类型转换的一些细节应聘者也是很容易忽略的，所以俺整理的尽量系统一些。`javascript` 是一种弱类型语言，变量不受类型限制，所以在特定情况下我们需要对类型进行转换。

「类型转换」分为 **显式类型转换** 和 **隐式类型转换**。每种转换又分为 **原始类型转换** 和 **对象类型转换**。

### 3.1 数字类型转换(Number)

笔试题：转化为数字类型！

- `Number`：类型转化走的是 V8 引擎最底层机制的转换规则。
- `parseFloat/parseInt`：是一个额外提供的方法的转化规则。

#### 3.1.1 Number

对于不同的类型有不同类型的底层转换规则。将引用类型（对象类型）转化为字符串，首先将它基于 `toString` 方法转化为字符串，然后将字符串转化为数字。

- 字符串 => 数字：非有效字符串转化为 NaN;
- 布尔 => 数字：1 或 0;
- null => 数字：0;
- undefined => 数字：NaN;
- '' => 数字：0;
- 对象 => 数字：先把【对象】转化为【字符串】，在转化为【数字】；

- [] => 数字: 0 (遵循以上对象转换的规则)。

举个栗子:

```
Number(10);           // 10
Number('10');          // 10
Number(null);          // 0
Number('');            // 0
Number(true);          // 1
Number(false);         // 0
Number([]);            // 0
Number([1,2]);         // NaN
Number('10a');         // NaN
Number(undefined);    // NaN
```

### 3.1.2 parseInt()/parseFloat()

parseInt()/parseFloat([val,[进制]]) 对于字符串来说, 它从左到右依次查找有效的数字字符, 直到遇到非有效数字字符, 停止查找 (不管后边是否还有数字, 都不在进行差最后啊, 把找到的当做数字返回)

如果当前转化不为字符串, 所以先将这个额外的方法主要先将转化的值为字符串类型, 然后进行检索数字直到遇到非数字结束。

举个栗子:

```
let str = '12.5px'
parseInt(str) // 12
parseFloat(str) // 12.5
parseFloat(true) // NaN
```

isNaN 判断数字类型:

如果当前为数字类型, 则返回 false, 否则返回 true。

注意: 如果当前传入的是字符串, isNaN() 将字符串通过 Number() 方法转化为数字, 然后判断当前是否为字符串类型。

### 3.1.3 大厂面试题:

写出以下程序运行的结果。

```

var str = 'abc123';
var num = parseInt(str);
if(num == NaN){
    alert(NaN);
}else if(num == 123){
    alert(123);
}else if(typeof num == 'number'){
    alert('number');
}else{
    alert('str');
}

```

根据上述所涉及到的 `parseInt` 转化规则，可知 `num` 的值为 `NAN`，`NAN` 是数字类型，所以输出 `number`。

## 3.2 字符串类型转换(String)

笔试题：其他数据类型转字符串类型！

对于原始类型来说，转字符串类型会默认调用 `toString()` 方法。之前长什么样子，然后在外层包一层引号。

- 数字 => 字符串：包裹一层引号。
- NaN => 字符串：'NaN'。
- true => 字符串：'true'。
- null => 字符串：'null'（浏览器会报错（禁止你使用）—— 通常可以进行转换）
- undefined => 字符串：'undefined'（浏览器会报错（禁止你使用）—— 通常可以进行转换）
- Object => 字符串：'[object,object]'。

普通对象转化的结果为“`[object,object]`”，因为 `Object.prototype.toString` 方法不是转化为字符串的，而是用来检测数据类型的。

举个栗子：

```

String(123);      // "123"
String(true);     // "true"
String(null);     // "null"（报错）
String(undefined); // "undefined"（报错）
String([1,2,3])   // "1,2,3"
String({});       // "[object Object]"

```

## 3.3 布尔类型转换 (Boolean)

笔试题：其他数据类型转布尔类型！

除了特殊的几个值 `''`、`undefined`、`NAN`、`null`、`false`、`0` 转化为 `Boolean` 为 `false` 之外，其他类型值都转化为 `true`。

```
Boolean('')           // false
Boolean(undefined)    // false
Boolean(null)         // false
Boolean(NaN)          // false
Boolean(false)        // false
Boolean(0)            // false
Boolean({})           // true
Boolean([])           // true
```

### 3.4 原始类型转换

笔试题：对象类型转原始类型！

转化原始类型分为两种情况：转化为 **字符串类型** 或 **其他原始类型**。

- 如果已经是原始类型，不需要再进行转化。
- 如果转字符串类型，就调用内置函数中的 `toString()` 方法。
- 如果是其他基本类型，则调用内置函数中的 `valueOf()` 方法。
- 如果返回的不是原始类型，则会继续调用 `toString()` 方法。
- 如果还没有返回原始类型，则报错。

## 4、数据类型检测

面试官：一共有几种数据类型检测？

一共有四种数据类型检测：

- `typeof` :
- `instanceof`
- `constructor`
- `Object.prototype.toString.call()`

### 4.1 `typeof`

`typeof` 是一元运算符，同样返回一个字符串类型。一般用来判断一个变量是否为空或者是什么类型。

除了 `null` 类型以及 `Object` 类型不能准确判断外，其他数据类型都可能返回正确的类型。

```
typeof undefined // 'undefined'
typeof '10'      // 'String'
typeof 10        // 'Number'
typeof false     // 'Boolean'
typeof Symbol()  // 'symbol'
typeof Function  // 'function'
typeof null      // 'object'
typeof []        // 'object'
typeof {}        // 'object'
```

面试官：为什么 `typeof null` 等于 `Object`?

不同的对象在底层原理的存储是用二进制表示的，在 `JavaScript` 中，如果二进制的前三位都为 0 的话，系统会判定为是 `Object` 类型。`null` 的存储二进制是 `000`，也是前三位，所以系统判定 `null` 为 `Object` 类型。

扩展：

这个 bug 个第一版的 `JavaScript` 留下来的。俺也进行扩展一下其他的几个类型标志位：

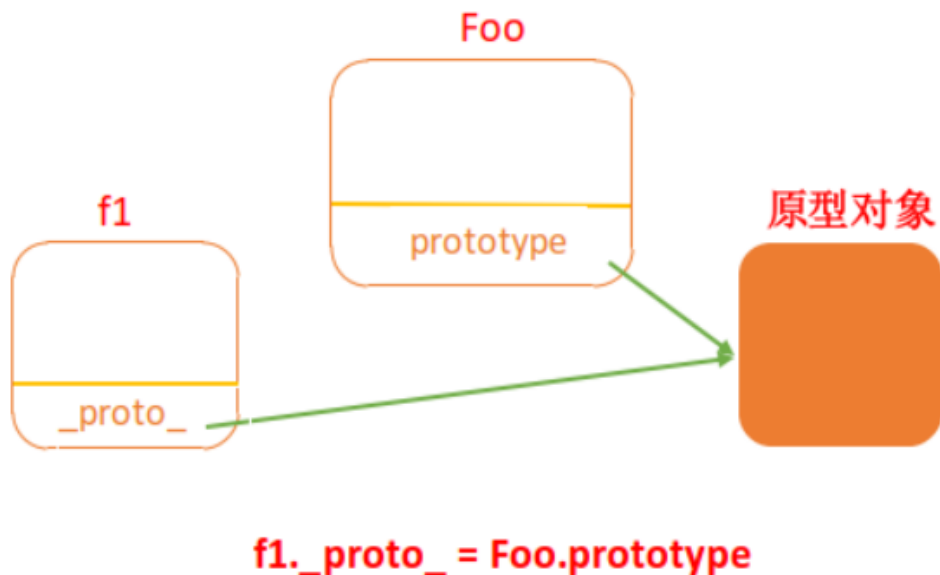
- `000`：对象类型。
- `1`：整型，数据是31位带符号整数。
- `010`：双精度类型，数据是双精度数字。
- `100`：字符串，数据是字符串。
- `110`：布尔类型，数据是布尔值。

## 4.2 `instanceof`

`instanceof` 运算符用来测试一个对象在其原型链中是否存在一个构造函数的 `prototype` 属性。我们可以使用 `instanceof` 来进行判断某个对象是不是另一个对象的实例。

`instanceof` 的原理是通过判断该对象的原型链中是否可以找到该构造类型的 `prototype` 类型。

```
function Foo(){}  
var f1 = new Foo();  
  
console.log(f1 instanceof Foo); // true
```



## 4.3 `constructor`

对于引用类型，除此之外，还可以使用 `xx.constructor.name` 构造函数来判断。

```
// constructor 构造来判断
console.log(fn.constructor.name) // Function
console.log(date.constructor.name) // Date
console.log(arr.constructor.name) // Array
console.log(reg.constructor.name) // RegExp
```

## 4.4 Object.prototype.toString.call()

一种最好的基本类型检测方式 `Object.prototype.toString.call()` ;它可以区分 `null`、`string`、`boolean`、`number`、`undefined`、`array`、`function`、`object`、`date`、`math` 数据类型。

```
// 判断基本类型
Object.prototype.toString.call(null); // "[object Null]"
Object.prototype.toString.call(undefined); // "[object Undefined]"
Object.prototype.toString.call("abc"); // "[object String]"
Object.prototype.toString.call(123); // "[object Number]"
Object.prototype.toString.call(true); // "[object Boolean]"

// 判断引用类型
function fn(){
  console.log("xiaolu");
}
var date = new Date();
var arr = [1,2,3];
var reg = /[hbc]at/gi;

Object.prototype.toString.call(fn); // "[object Function]"
Object.prototype.toString.call(date); // "[object Date]"
Object.prototype.toString.call(arr); // "[object Array]"
Object.prototype.toString.call(reg); // "[object RegExp]"
```

缺陷：不能细分是谁谁的实例。

## 5、各种运算

### 5.1 四则运算

四则运算法则中，除了加法之外，其余都是数学计算。在 JS 中只有【加法】可能存在字符串拼接（一旦遇到字符串，则不是数学运算，而是字符串拼接）

#### 加法运算

- 从左到后依次执行相加，如果是数学运算，都先将其转化为 `Number` 类型。
- 遇到字符串肯定不是【数学运算】而是【字符串拼接】。

```
console.log('10' + 10) // 1010
console.log('10' - 10) // 0
console.log('10px' - 10) // NaN
console.log(10 + null + true + [] + undefined + '小鹿' + null + [] + 10 + false )
```

```
true + true // 2
1 + true    // 2
[1] + 3     // '13'
```

## 其他运算

其他算术运算符（比如减法、除法和乘法）都不会发生重载。它们的规则是：所有运算符一律转为数值，再进行相应的数学运算。

```
1 * '2' // 2
1 * []  // 0
```

## 5.2 逻辑运算符(&& ||)

逻辑运算符包括两种情况，分别为 **条件判断** 和 **赋值操作**。

### 条件判断

- **&&**：所有条件为真，整体才为真。
- **||**：只有一个条件为真，整体就为真。

```
true && true // true
true && false // false
true || true // true
true || false // true
```

### 赋值操作

- **A && B**

首先看 **A** 的真假，**A** 为假，返回 **A** 的值，**A** 为真返回 **B** 的值。（不管 **B** 是啥）

```
console.log(0 && 1) // 0
console.log(1 && 2) // 2
```

- **A || B**

首先看 **A** 的真假，**A** 为真返回的是 **A** 的值，**A** 为假返回的是 **B** 的值（不管 **B** 是啥）

```
console.log(0 || 1) // 1
console.log(1 || 2) // 1
```

## 5.3、比较运算符

比较运算符在逻辑语句中使用，以判定变量或值是否相等。

面试官：== 和 === 的区别？

对于 `===` 来说，是严格意义上的相等，会比较两个操作符的类型和值。

- 如果 `X` 和 `Y` 的类型不同，返回 `false`；
- 如果 `X` 和 `Y` 的类型相同，比较值的大小。

而对于 `==` 来说，是非严格意义上的相等，先判断两个操作符的类型是否相等，如果类型不同，则先进行类型转换，然后再判断值是否相等。

- 如果 `X` 和 `Y` 的类型相同，返回 `X == Y` 的比较结果；
- 如果 `X` 和 `Y` 的类型不同，根据下方表格进一步判断；
  - `null == undefined` -> `true`
  - `String == Number` -> 字符串转数字，再比较
  - `Boolean == Number` -> 布尔转数字，再比较
  - `Object == String, Number, Symbol` -> `Object` 转化为原始类型

## 6、i++/++i

两者都是数学运算中的累加 1，区别的计算顺序。

- `i++`：先用再加
- `++i`：先加再用

### 6.1 面试题一

写出以下程序运行的结果。

```
let a = '10'  
a == 10 ? a++ : a--;  
console.log(a);
```

涉及到知识点：`i++ / ++i` 的区别；`if (==)` 和 `switch (===)` 的区别；四则运算规则。

### 6.2 面试题二

写出以下程序运行的结果。

```
let b = '10';  
switch(b){  
  case 10:  
    b++;  
    break;  
  default:  
    b--;  
}  
console.log(b);
```



涉及到知识点: i++ / ++i 的区别; if (==) 和 switch (===) 的区别; 四则运算规则。

# this

## 知识线

什么是 this(对象) => 不同情况下的 this 指向 (4 种) => 如何改变 this 指针 (3 种) => 三者的共同点与不同点 (传参、执行) => call、apply、bind 源码实现 => 大厂笔试题解析

面试官: 什么是 this 指针?以及各种情况下的 this 指向问题。

## 1、什么是 this?

`this` 就是一个对象。不同情况下 `this` 指向的不同, 有以下几种情况, (希望各位亲自测试一下, 这样会更容易弄懂)。

## 2、this 的指向?

- 对象调用, `this` 指向该对象 (前边谁调用 this 就指向谁)。

```
var obj = {
  name: '小鹿',
  age: '21',
  print: function(){
    console.log(this)
    console.log(this.name + ':' + this.age)
  }
}

// 通过对象的方式调用函数
obj.print();      // this 指向 obj
```

- 直接调用的函数, `this` 指向的是全局 `window` 对象。

```
function print(){
  console.log(this);
}

// 全局调用函数
print();    // this 指向 window
```

- 通过 `new` 的方式, `this` 永远指向新创建的对象。

```
function Person(name, age){
  this.name = name;
  this.age = age;
  console.log(this);
}

var xiaolu = new Person('小鹿',22); // this => xiaolu
```

- 箭头函数中的 `this`。

由于箭头函数没有单独的 `this` 值。箭头函数的 `this` 与声明所在的上下文相同。也就是说调用箭头函数的时候，不会隐士的调用 `this` 参数，而是从定义时的函数继承上下文。

```
const obj = {
  a:()=>{
    console.log(this);
  }
}
// 对象调用箭头函数
obj.a(); // window
```

### 3、如何改变 this?

面试官：如何改变 this 的指向？以及三者的共同点和不同点

我们可以通过调用函数的 `call`、`apply`、`bind` 来改变 `this` 的指向。

```
var obj = {
  name: '小鹿',
  age: '22',
  adress: '小鹿动画学编程'
}

function print(){
  console.log(this); // 打印 this 的指向
  console.log(arguments); // 打印传递的参数
}

// 通过 call 改变 this 指向
print.call(obj,1,2,3);

// 通过 apply 改变 this 指向
print.apply(obj,[1,2,3]);

// 通过 bind 改变 this 的指向
let fn = print.bind(obj,1,2,3);
fn();
```

再说一说这三者的共同点和不同点。

### 共同点:

- **功能角度:** 三者都能改变 `this` 指向, 且第一个传递的参数都是 `this` 指向的对象。
- **传参角度:** 三者都采用的后续传参的形式。

### 不同点:

- **传参方面:** `call` 的传参是单个传递的 (试了下数组, 也是可以的), 而 `apply` 后续传递的参数是数组形式 (传单个值会报错), 而 `bind` 没有规定, 传递值和数组都可以。
- **执行方面:** `call` 和 `apply` 函数的执行是直接执行的, 而 `bind` 函数会返回一个函数, 然后我们想要调用的时候才会执行。

面试官: 如果我们使用上边的方法改变箭头函数的 `this` 指针, 会发生什么情况呢? 能否进行改变呢?

由于箭头函数没有自己的 `this` 指针, 通过 `call()` 或 `apply()` 方法调用一个函数时, 只能传递参数 (不能绑定 `this`), 他们的第一个参数会被忽略。

## 4、call、apply、bind 实现

面试官: 分别说说 `call`、`apply`、`bind` 的内部实现?

### 4.1 手写call

- 首先 `context` 为可选参数, 如果不传的话默认上下文为 `window`;
- 接下来给 `context` 创建一个 `fn` 属性, 并将值设置为需要调用的函数;
- 因为 `call` 可以传入多个参数作为调用函数的参数, 所以需要将参数剥离出来;
- 然后调用函数并将对象上的函数删除。

```
// this 为调用的函数
// context 是参数对象
Function.prototype.myCall = function(context){
  // 判断调用者是否为函数
  if(typeof this !== 'function'){
    throw new TypeError('Error')
  }
  // 不传参默认为 window
  context = context || window
  // 新增 fn 属性, 将值设置为需要调用的函数
  context.fn = this
  // 将 arguments 转化为数组将 call 的传参提取出来 [...arguments]
  const args = Array.from(arguments).slice(1)
  // 传参调用函数
  const result = context.fn(...args)
  // 删除函数
  delete context.fn
  // 返回执行结果
  return result;
}
```

```
// 普通函数
function print(age){
    console.log(this.name+" "+age);
}
// 自定义对象
var obj = {
    name: '小鹿'
}
// 调用函数的 call 方法
print.myCall(obj,1,2,3)
```

## 4.2 手写 apply

- 首先 `context` 为可选参数，如果不传的话默认上下文为 `window`
- 接下来给 `context` 创建一个 `fn` 属性，并将值设置为需要调用的函数
- 因为 `apply` 传参是数组传参，所以取得数组，将其剥离为顺序参数进行函数调用
- 然后调用函数并将对象上的函数删除

```
// 手写一个 apply 方法
Function.prototype.myApply = function(context){
    // 判断调用者是否为函数
    if(typeof this !== 'function'){
        throw new TypeError('Error')
    }
    // 不传参默认为 window
    context = context || window
    // 新增 fn 属性,将值设置为需要调用的函数
    context.fn = this
    // 返回执行结果
    let result;
    // 判断是否有参数传入
    if(arguments[1]){
        result = context.fn(...arguments[1])
    }else{
        result = context.fn()
    }
    // 删除函数
    delete context.fn
    // 返回执行结果
    return result;
}

// 普通函数
function print(age,age2,age3){
    console.log(this.name+" "+ age + " "+ age2+" "+age3);
}
// 自定义对象
var obj = {
    name: '小鹿'
}
// 调用函数的 call 方法
print.myApply(obj,[1,2,3])
```

## 4.3 手写 bind

- 判断调用者是否为函数。
- 截取参数，注意：这里有两种形式传参。
- 返回一个函数，判断外部哪种方式调用了该函数（new | 直接调用）

```
// 手写一个 bind 函数
Function.prototype.myBind = function (context) {
  // 判断调用者是否为函数
  if(typeof this !== 'function'){
    throw new TypeError('Error')
  }
  // 截取传递的参数
  const args = Array.from(arguments).slice(1)
  // _this 指向调用的函数
  const _this = this;

  // 返回一个函数
  return function F(){
    // 因为返回了一个函数，我们可以 new F()，所以需要判断
    // 对于 new 的情况来说，不会被任何方式改变 this
    if(this instanceof F){
      return new _this(...args,...arguments)
    }else{
      return _this.apply(context,args.concat(...arguments))
    }
  }
}

// 普通函数
function print(){
  // new 的方式调用 bind 参数输出换做 [...arguments]
  console.log(this.name);
}

// 自定义对象
var obj = {
  name: '小鹿'
}

// 调用函数的 call 方法
let F = print.myBind(obj,1,2,3);
// 返回对象
let obj1 = new F();
console.log(obj1);
```

## new

### 知识线

创建对象的几种方式（3种） => 三者的区别是什么 => new 的内部发生了什么（四步） => Object.create() 的应用。

### 1、创建对象的几种方式？

面试官：有几种创建对象的方式，字面量相对于 new 创建对象有哪些优势？

- 字面量
- new
- Object.create()

## 1.1 字面量

对于字面量创建对象：

```
var obj = {  
  name: '小鹿动画学编程'  
}
```

- 代码量更少，更易读
- 对象字面量运行速度更快。它们可以在解析的时候被优化，他不会像 new 一个对象一样，解析器需要顺着作用域链从当前作用域开始查找，如果在当前作用域找到了名为 Object() 的函数就执行，如果没找到，就继续顺着作用域链往上照，直到找到全局 Object() 构造函数为止。
- Object() 构造函数可以接收参数，通过这个参数可以把对象实例的创建过程委托给另一个内置构造函数，并返回另外一个对象实例，而这往往不是你想要的。

## 1.2 new

面试官：new 内部发生了什么过程？可不可以手写实现一个 new 操作符？

对于 new 关键字，我们第一想到的就是在面向对象中 new 一个实例对象，但是在 JS 中的 new 和 Java 中的 new 的机制不一样。

一般 Java 中，声明一个构造函数，通过 new 类名() 来创建一个实例，而这个构造函数是一种特殊的函数。但是在 JS 中，只要 new 一个函数，就可以 new 一个对象，函数和构造函数没有任何的区别。

对于 new 创建对象：

```
var arr = new Array();
```

new 的过程包括以下四个阶段：

- 创建一个新对象。
- 这个新对象的 \_\_proto\_\_ 属性指向原函数的 prototype 属性。(即继承原函数的原型)
- 将这个新对象绑定到此函数的 this 上。
- 返回新对象，如果这个函数没有返回其他对象。

```
// new 生成对象的过程  
// 1、生成新对象  
// 2、链接到原型  
// 3、绑定 this  
// 4、返回新对象  
// 参数：
```

```

// 1、Con: 接收一个构造函数
// 2、args: 传入构造函数的参数
function create(Con, ...args){
  // 创建空对象
  let obj = {};
  // 设置空对象的原型(链接对象的原型)
  obj.__proto__ = Con.prototype;
  // 绑定 this 并执行构造函数(为对象设置属性)
  let result = Con.apply(obj,args)
  // 如果 result 没有其他选择的对象, 就返回 obj 对象
  return result instanceof Object ? result : obj;
}

// 构造函数
function Test(name, age) {
  this.name = name
  this.age = age
}
Test.prototype.sayName = function () {
  console.log(this.name)
}

// 实现一个 new 操作符
const a = create(Test, '小鹿', '23')
console.log(a.age)

```

### 1.2.1 返回值

构造函数执行, 不写 `return`, 浏览器会默认返回创建的实例, 但是如果我们自己写了 `return`。

- 基本值: `return` 是一个基本值, 返回的结果依然是类的实例, 没有收到影响。
- 引用值: 如果返回的是一个对象, 则将默认的返回的实例覆盖, 接收的结果就不再是当前类的实例。

### 1.2.2 `in` : 检测对象属性 (私有 + 共有)

检测当前对象是否存在某个属性 (不管当前这个属性是对象的私有属性还是公有属性, 只要有结果就是 `true`)

```

var obj = {
  name: '小鹿'
}
console.log('a' in obj) // true

```

### 1.2.3 `hasOwnProperty` (私有)

检测当前属性是否为对象的私有属性 (不仅要有这个属性, 而且必须还是私有的才可以)

```
let arr = new Array();
arr.a = 1
arr.__proto__.b = 1

arr.hasOwnProperty('1') // true
arr.hasOwnProperty('b') // false
```

### 1.3 Object.create(null)

对于 `Object.create()` 方式创建对象：

```
Object.create(proto, [propertiesObject]);
```

- `proto`: 新创建对象的原型对象。
- `propertiesObject`: (可选) 可为创建的新对象设置属性和值。

一般用于继承：

```
var People = function (name){
    this.name = name;
};

People.prototype.sayName = function (){
    console.log(this.name);
}

function Person(name, age){
    this.age = age;
    People.call(this, name); // 使用call, 实现了People 属性的继承
};

// 使用Object.create()方法, 实现People原型方法的继承, 并且修改了constructor指向
Person.prototype = Object.create(People.prototype, {
    constructor: {
        configurable: true,
        enumerable: true,
        value: Person,
        writable: true
    }
});

Person.prototype.sayAge = function (){
    console.log(this.age);
}

var p1 = new Person('person1', 25);

p1.sayName(); // 'person1'
p1.sayAge(); // 25
```

### 1.4 三者创建对象的区别



面试官：new/字面量 与 Object.create(null) 创建对象的区别？

- new 和 字面量创建的对象的原型指向 Object.prototype，会继承 Object 的属性和方法。
- 而通过 Object.create(null) 创建的对象，其原型指向 null，null 作为原型链的顶端，没有也不会继承任何属性和方法。

## 闭包

### 知识线

什么是作用域/作用域链（表面/堆栈） => 什么是执行栈和执行上下文（执行步骤） => 什么是闭包 => 闭包的作用（保护和保存） => 闭包的应用 => 闭包循环引用问题 => 大厂笔试题解析

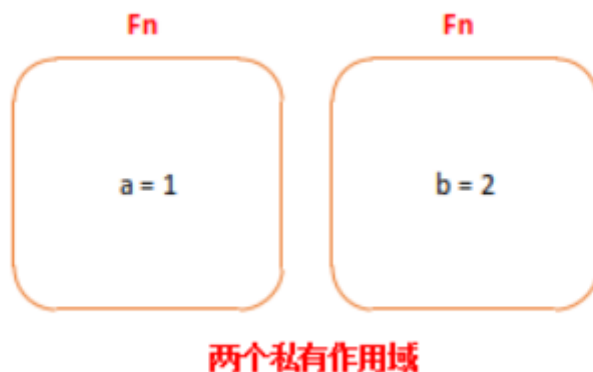
### 1、作用域和作用域链

面试官：什么是作用域？什么是作用域链？

规定 **变量和函数** 的可使用范围叫做作用域。只看定义，挺抽象的，举个例子：☹

```
function fn1() {  
  let a = 1;  
}  
  
function fn2() {  
  let b = 2;  
}
```

声明两个函数，分别创建量两个私有的作用域（可以理解为两个封闭容器），fn2 是不能直接访问私有作用域 fn1 的变量 a 的。同样的，在 fn1 中不能访问到 fn2 中的 b 变量的。一个函数就是一个作用域。



堆栈理解：声明两个函数，在堆内存中创建两个完全独立私有的内存空间，每个空间都是独立私有的。

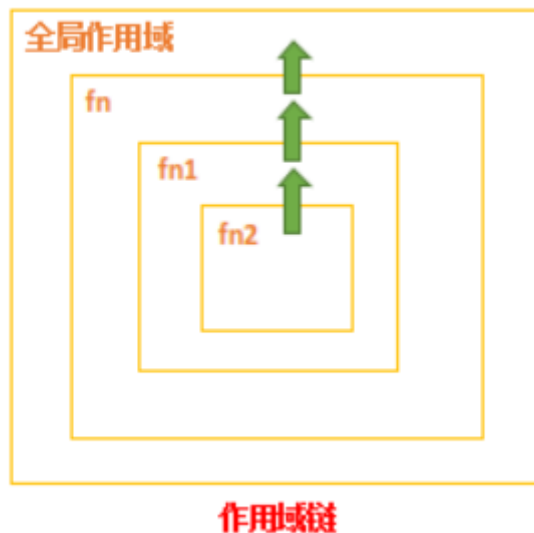
每个函数都会有一个作用域，查找变量或函数时，由局部作用域到全局作用域依次查找，**这些作用域的集合就称为作用域链**。如果还不是很好理解，俺再举个例子：☹

```

let a = 1
function fn() {
  function fn1() {
    function fn2() {
      let c = 3;
      console.log(a);
    }
    // 执行 fn2
    fn2();
  }
  // 执行 fn1
  fn1();
}
// 执行函数
fn();

```

虽然上边看起来嵌套有点复杂，我们前边说过，一个函数就是一个私有作用域，根据定义，在 `fn2` 作用域中打印 `a`，首先在自己所在作用域搜索，如果没有就向上级作用域搜索，直到搜索到全局作用域，`a = 1`，找到了打印出值。整个搜索的过程，就是基于作用域链搜索的。



## 2、执行栈和执行上下文

面试官：什么是执行栈和执行上下文？

### 2.1 执行上下文

- **全局执行上下文** — 这是默认或者说基础的上下文，任何不在函数内部的代码都在全局上下文中。它会执行两件事：创建一个全局的 `window` 对象（浏览器的情况下），并且设置 `this` 的值等于这个全局对象。一个程序中只会有一个全局执行上下文。
- **函数执行上下文** — 每当一个函数被调用时，都会为该函数创建一个新的上下文。每个函数都有它自己的执行上下文，不过是在函数被调用时创建的。函数上下文可以有任意多个。每当一个新的执行上下文被创建，它会按定义的顺序（将在后文讨论）执行一系列步骤。
- **Eval 函数执行上下文** — 执行在 `eval` 函数内部的代码也会有它属于自己的执行上下文，但由于 JavaScript 开发者并不经常使用 `eval`，

## 2.2 执行栈

执行栈，也就是在其它编程语言中所说的“调用栈”，是一种拥有 LIFO（后进先出）数据结构的栈，被用来存储代码运行时创建的所有执行上下文。

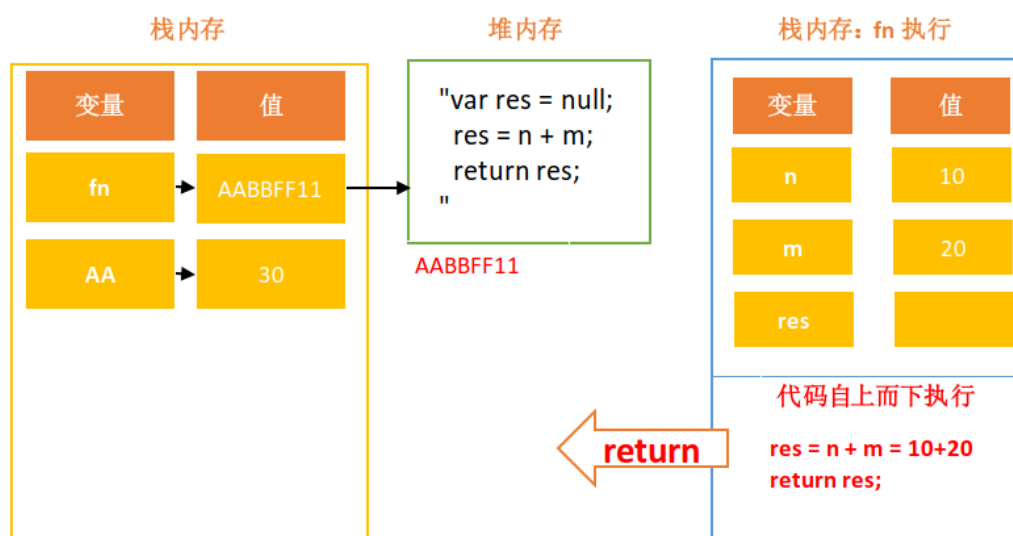
当 JavaScript 引擎第一次遇到你的脚本时，它会创建一个全局的执行上下文并且压入当前执行栈。每当引擎遇到一个函数调用，它会为该函数创建一个新的执行上下文并压入栈的顶部。

引擎会执行那些执行上下文位于栈顶的函数。当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

举个例子：

```
function fn(n, m){  
  var res = null;  
  res = n + m;  
  return res;  
}  
var AA = fn(10, 20);  
console.log(AA);
```

整个代码的执行过程如下：



公众号：小鹿动画学编程

- 执行环境（栈内存 —— 存储变量：存储值）
- 代码自上而下执行（主线程）
- 创建函数，开辟堆内存，存储的是函数体内字符串（之所以创建函数他不执行，就是一堆破字符串）。
- 将栈内存的地址指向该堆内存。
- 函数执行，每一次函数执行的目的是把函数体内的代码（先将字符串转化为代码）执行 => 此时会形成一个私有的栈内存。
- 开始拿出代码一行行执行，也是变量和值的存储形式，将执行后的代码得到的值返回。

## 2.3 小结

- 1、函数内的变量之所以不能在函数外边引用，因为在内存中形成的一个私有内存，不能在外边进行引用。
- 2、栈内存中存储值的过程为（VO阶段），而执行代码的阶段为（AO阶段）——JS 编译过程
- 3、函数创建时，不执行的原因
- 4、由于引用类型的值太大，栈内存放不下，只能存放在堆内存中。

## 3、闭包

面试官：什么是闭包？闭包的作用？闭包的应用？

### 3.1 什么是闭包？

函数执行，形成一个私有的执行上下文，保护里边的私有变量不受外界干扰，除了**保护**私有变量外，还可以**保存**一些内容，这样的模式叫做**闭包**。

```
function makeFunc() {  
    var name = "公众号：小鹿动画学编程";  
    function displayName() {  
        alert(name);  
    }  
    return displayName;  
}  
  
var myFunc = makeFunc();  
myFunc();
```

### 3.2 闭包的作用

闭包的作用有两个，**保护和保存**。

#### 3.2.1 保护

- 团队开发时，每个开发者把自己的代码放在一个私有的作用域中，防止相互之间的变量命名冲突；把需要提供给别人的方法，通过 `return` 或 `window.xxx` 的方式暴露在全局下。
- `jQuery` 的源码中也是利用了这种保护机制。
- 封装私有变量。

#### 3.2.2 保存

- 选项卡闭包的解决方案。

面试官：循环绑定事件引发的索引什么问题？怎么解决这种问题？

```
// 事件绑定引发的索引问题
var btnBox = document.getElementById('btnBox'),
    inputs = btnBox.getElementsByTagName('input')
var len = inputs.length;
for(var i = 0; i < len; i++){
    inputs[i].onclick = function () {
        alert(i)
    }
}
```

闭包剩余的部分，之前的文章已经总结过，俺就不复制过来了，直接传送过去~ [动画：什么是闭包？](#)

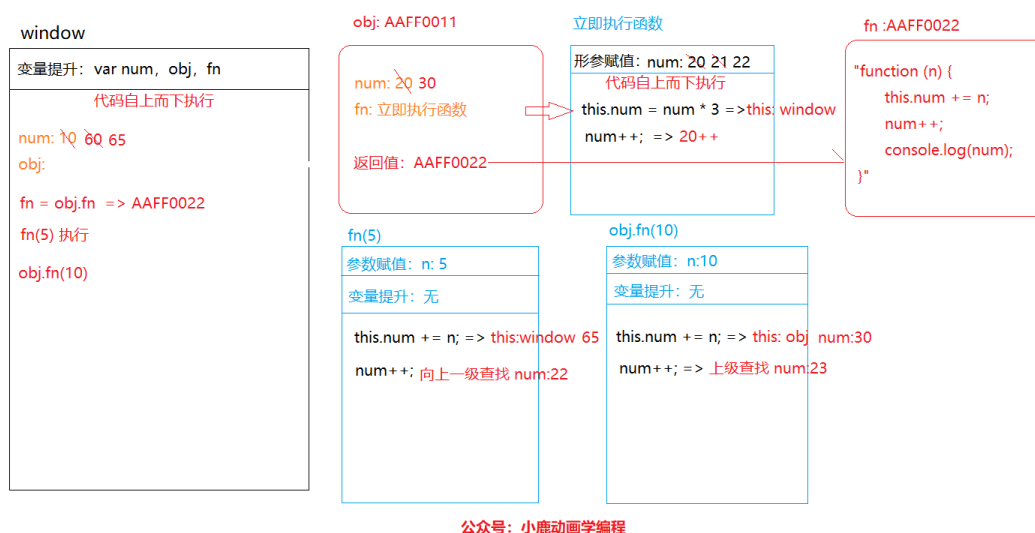
### 3.4 大厂经典面试题

```
var num = 10;
var obj = {num: 20};
obj.fn = (function (num){
    this.num = num * 3;
    num++;
    return function (n) {
        this.num += n;
        num++;
        console.log(num);
    }
})(obj.num)
var fn = obj.fn;
fn(5)
obj.fn(10)
console.log(num,obj.num)
// 22 23 65 30
```

- 1、首先会声明一个全局作用域（window）。
- 2、第一步：全局作用域下的变量提升（注意：函数表达式、箭头函数、立即执行函数不会进行变量提升）。
- 3、代码自上而下执行。
- 4、正常的（**数字/字符串**）变量或常量直接赋值。
  - 如果遇到**对象或者函数**赋值就会在堆内存开辟一个内存空间，将字符串存进去。
    - 存储属性名
    - 将地址赋值给变量
    - 代码继续自上而下执行
  - 如果遇到自执行函数，就将自执行函数的返回结果赋值给变量。
    - 函数执行，声明一个私有的作用域（变量都是私有的）。
    - 函形参赋值。
    - 变量提升。
    - 自执行函数代码自上而下继续执行。

- 如果自执行函数没有主体，则 `this` 指向的 `window`；注意：看改变的值改变的是私有变量还是其他变量。
- 如果自执行函数返回一个函数（`return`）。
  - 开辟一个堆内存空间（将字符串存进去）；
  - 将该堆内存的地址返回给 `return`
- 所以自执行函数执行返回的变量地址指向 `return` 的地址（也就是执行开辟的堆内存空间）。
- 此时函数内部的函数被外部的变量引用着，所以这个私有的作用域不会销毁。
- 遇到函数执行。
  - 函数执行，形成一个私有的作用域（变量是私有的）。
  - 形参赋值。
  - 变量提升。
  - 函数自上而下执行。
  - **`this` 的指向遵循三个原则，以及注意私有作用域变量的作用范围**
  - 函数执行完毕，进行作用域销毁

图解：



## 原型和原型链

### 知识线

JS 原型对象的由来 => 什么是原型/原型链 => 实例对象-构造函数-原型关系图 => 原型链中的各个关系？  
=> 大厂面试题

### 1、原型的前世今生

对于原型和原型链的前世今生，由于篇幅过大，俺的传送门~ [图解：告诉面试官什么是 JS 原型和原型链？](#)

PS：下面的看不懂，一定去看文章哦！

## 2、什么是原型/原型链？


面试官：什么是原型？什么是原型链？如何理解？

### 2.1 原型

**原型：**每个 JS 对象都有 `__proto__` 属性，这个属性指向了原型，跟俺去看看。

```
> console.log([]);
```

```
▼ [] ⓘ  
  length: 0  
  ▶ __proto__: Array(0)
```



再来一个，

```
> console.log({})
```

```
▼ {} ⓘ  
  ▶ __proto__: Object
```



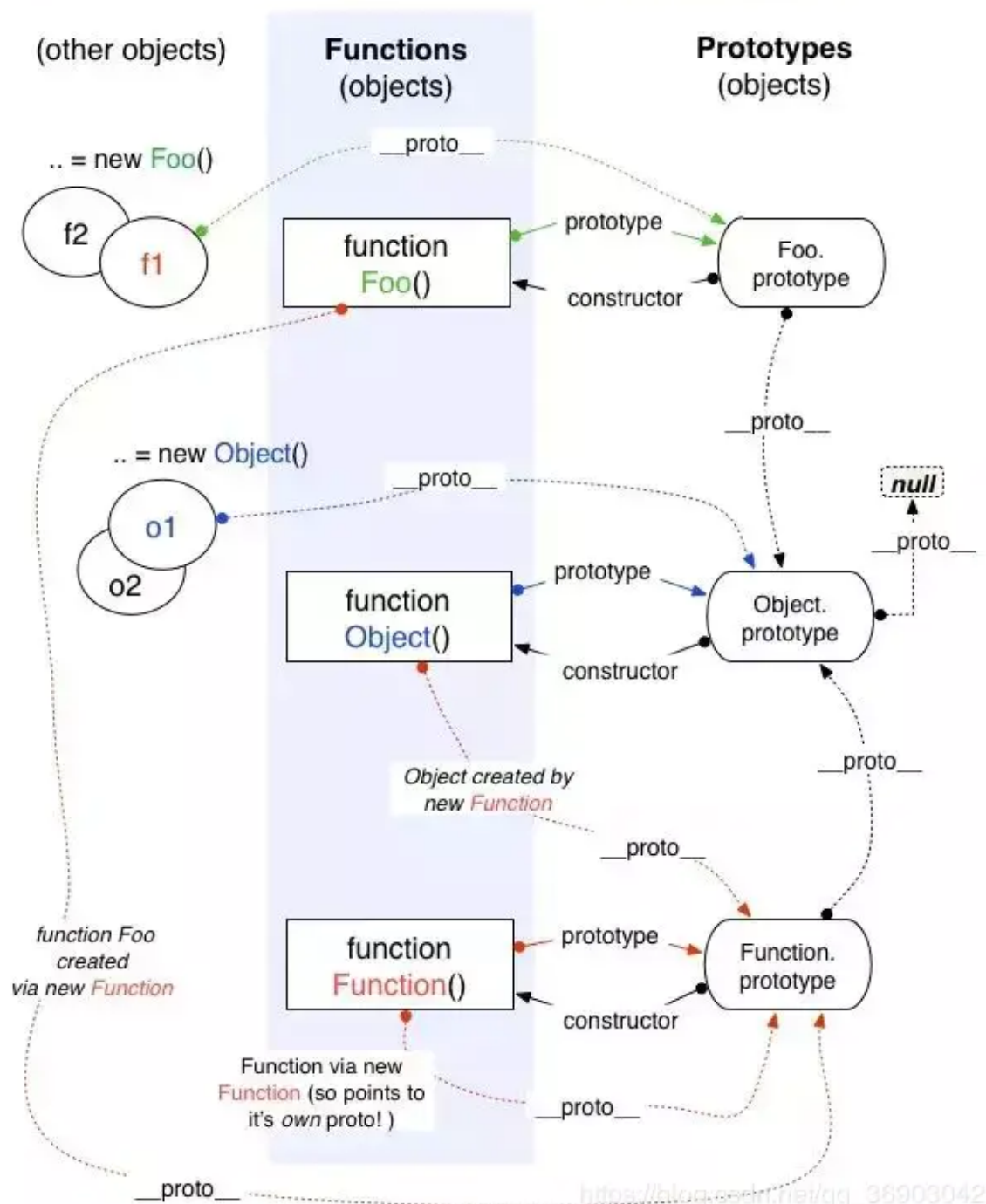
我们可以看到，只要是对象类型，都会有这个 `__proto__` 属性，这个属性指向的也是一个原型对象，原型对象也是对象呀，肯定也会存在一个 `__proto__` 属性。那么就形成了原型链，定义如下：

### 2.2 原型链

**原型链：**原型链就是多个对象通过 `__proto__` 的方式连接了起来形成一条链。

面试官：如果理清原型链中的关系呢？

再往深处看，他们之间存在复杂的关系，但是这些所谓的负责关系俺已经总结好了，小二上菜。



这张图看起来真复杂，但是通过下边总结的，再来分析这张图，试试看。

- 所有的实例的 `__proto__` 都指向该构造函数的原型对象（`prototype`）。
- 所有的函数（包括构造函数）是 `Function()` 的实例，所以所有函数的 `__proto__` 的都指向 `Function()` 的原型对象。
- 所有的原型对象（包括 `Function` 的原型对象）都是 `Object` 的实例，所以 `__proto__` 都指向 `Object`（构造函数）的原型对象。而 `Object` 构造函数的 `__proto__` 指向 `null`。
- `Function` 构造函数本身就是 `Function` 的实例，所以 `__proto__` 指向 `Function` 的原型对象。

## 2.3 大厂面试题

```
function Fn() {
  this.x = 100;
```



```

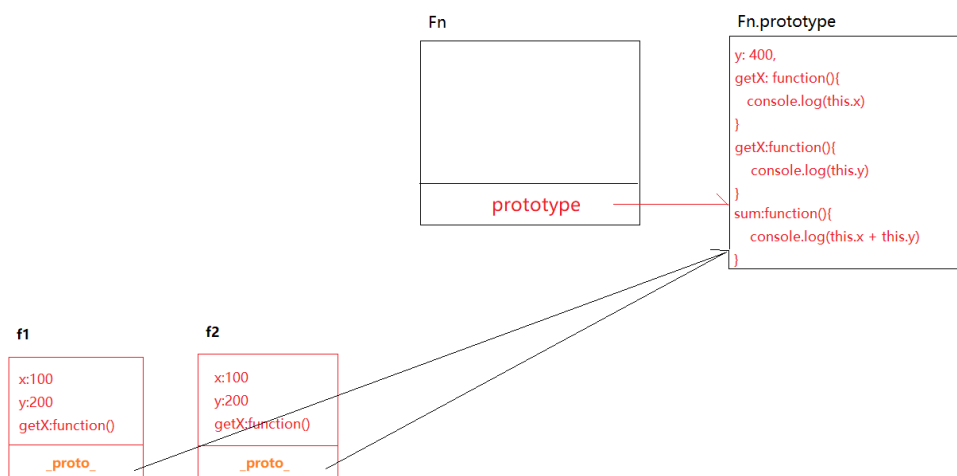
    this.y = 200;
    this.getX = function () {
        console.log(this.x)
    }
}

Fn.prototype = {
    y:400,
    getX: function () {
        console.log(this.x)
    },
    getY: function () {
        console.log(this.y)
    },
    sum: function () {
        console.log(this.x + this.y)
    }
}

var f1 = new Fn();
var f2 = new Fn();
console.log(f1.getX === f2.getX) // => 不相等
console.log(f1.getY === f2.getY) // => 相等
console.log(f1._proto_.getY === Fn.prototype.getY) // => 相等
console.log(f1._proto_.getX === f2.getX) // => 不相等
console.log(f1.getX === Fn.prototype.getX) // => 不相等
console.log(f1.constructor) // => Object
console.log(Fn.prototype._proto_.constructor) // Object
f1.getX(); // => 100
f1._proto_.getX(); // => Fn.prototype undefined
f2.getY(); // => 200
Fn.prototype.getY(); // => 400
f1.sum(); // => this:f1 f1.x + f1.y=>300
Fn.prototype.sum(); // => this:Fn.prototype Fn.prototype.x + Fn.prototype.y =
undefined + 400 = NaN

```

将上述简单画一个原型图，如下：



公众号：小鹿动画学编程

- 函数会开辟一个堆内存空间，存储执行函数里边的代码字符串。
- 所有的类函数，天生自带一个属性 `prototype`，这个属性是一个对象值，所以也会开辟一个对内存空间存储这个对象值（浏览器默认给开辟的）。
- 这个（每个）原型对象都会有一个天生自带属性 `constructor`，它指向 `Fn`（函数）本身。

- 而且**每个对象实例**天生会自带一个属性 `_proto_` 指向构造函数的（`prototype`）原型对象。
- 构造函数的原型改变时，会断开原型对象以及其他相关的连接。

## 继承

继承的核心思想就是，能够继承父类方法的同时，保证自己的私有属性和方法。

## 知识线

最常用的四种继承方式 => 原型继承 => 组合继承 => 寄生组合继承 => ES6 的 `extend` 继承 => 各个继承的优缺点？

面试官：说一说 JS 中的常用的继承方式有哪些？以及各个继承方式的优缺点。

最常用的继承有以下四个：

- 原型继承
- 组合继承
- 寄生组合继承
- ES6 的 `extend` 继承

### 1、原型继承

- **核心思想**：将父类的实例作为子类的原型。
- **优点**：方法复用
  - 由于方法定义在父类的原型上，复用了父类构造函数原型上的方法。
- **缺点**：
  - 创建的子类实例不能传参。
  - 子类实例共享了父类构造函数的引用属性（如：`arr`）。

```
var person = {  
  friends : ["Van","Louis","Nick"]  
};  
  
var p1 = Object.create(person);  
p1.friends.push("Rob");  
  
var p2 = Object.create(person);  
p2.friends.push("Style");  
  
console.log(person.friends); // "Van,Louis,Nick,Rob,Style"
```

### 2、组合继承

- **核心思想**：通过调用父类构造函数，继承父类的属性并保留传参的优点；然后将父类实例作为子类原型，实现函数复用。
- **优点**：
  - **可传参**：子类实例创建可以传递参数。
  - **方法复用**：同时所有子类可以复用父类的方法。
  - **不共享父类引用属性**：子类的改变不会引起父类引用类型的共享。
- **缺点**：
  - 组合继承调用了两次父类的构造函数，造成了不必要的消耗。

```
// 父类
function Father(name){
  this.name = name;
  this.colors = ["red","blue","green"];
}

// 方法定义在原型对象上（共享）
Father.prototype.sayName = function(){
  alert(this.name);
};

function Son(name,age){
  Father.call(this,name);    // 核心：继承实例属性，第一次调用 Father()
  this.age = age;
}

// 子类和父类共享的方法（实现了父类属性和方法的复用）
Son.prototype = new Father();    // 核心：继承实例属性，第二次调用 Father()
Son.prototype.constructor = Son; // 修复子类的 constructor 的指向

// 优点一：可传参
var instance1 = new Son("louis",5);
var instance2 = new Son("zhai",10);

// 优点二：共享父类方法
instance1.sayName(); // louis
instance2.sayName(); // zhai

// 优点三：不共享父类引用属性
instance1.colors.push("black");
console.log(instance1.colors); //"red,blue,green,black"
console.log(instance2.colors); //"red,blue,green"
```

### 3、寄生组合继承

- **核心思想**：组合继承 + 原型继承结合两者的优点。
- **优点**：完美！
- **缺点**：无！

```
// 父类
function Father(name){
  this.name = name;
```

```

    this.colors = ["red", "blue", "green"];
}

// 方法定义在原型对象上（共享）
Father.prototype.sayName = function(){
    alert(this.name);
};

function Son(name, age){
    Father.call(this, name);    // 核心
    this.age = age;
}

Son.prototype = Object.create(Father.prototype); // 核心：
Son.prototype.constructor = Son; // 修复子类的 constructor 的指向

```

## 4、ES6 的 extend 继承

ES6 的 `extend` 继承其实就是寄生组合式继承的语法糖。

- 核心思想：
  - `extends`：内部相当于设置了 `Son.prototype = Object.create(Father.prototype);`
  - `super()`：内部相当于调用了 `Father.call(this)`
- 小结：
  - 子类只要继承父类，可以不写 `constructor`，一旦写了，则在 `constructor` 中的第一句话必须是 `super`。
  - 把父类当做普通方法执行，给方法传递参数，让方法中的 `this` 是子类的实例。

```

class Son extends Father {    // Son.prototype.__proto__ = Father.prototype
    constructor(y) {
        super(200);           // super(200) => Father.call(this, 200)
        this.y = y
    }
}

```

## 垃圾回收机制

### 知识线

什么是内存泄漏？=> 是什么导致的内存泄漏？=> 怎么解决内存泄漏？=> 垃圾回收机制的策略（两种）=> 如何管理好内存？

面试官：什么是内存泄漏？为什么会发生内存泄漏？

### 1、什么是内存泄漏？

不再用到的内存，没有及时释放，就叫做内存泄漏。

## 2、为什么会导致内存泄漏？

内存泄漏是指我们已经无法再通过 js 代码来引用到某个对象，但垃圾回收器却认为这个对象还在被引用，因此在回收的时候不会释放它。导致了分配的这块内存永远也无法被释放出来。如果这样的情况越来越多，会导致内存不够用而系统崩溃。

## 3、垃圾回收机制

面试官：怎么解决内存泄漏？说一说 JS 垃圾回收机制的运行机制的原理？。

需要我们手动管理好内存，但是对于 JS 有自动垃圾回收机制，自动对内存进行管理。

### 3.1 两种垃圾回收策略

垃圾回收器主要的功能就是每隔一段时间，就去周期性的执行收集不再继续用到的内存，然后将其释放掉。

- 标记清除法
- 引用计数法

### 3.2 标记清除

它的实现原理就是通过判断一个变量是否在执行环境中被引用，来进行标记删除。

垃圾回收器给所有内存变量进行标记，然后去掉执行环境中未被引用的标记，剩余的标记变量是不会被用到的变量，会被垃圾回收器回收。

### 3.3 引用计数

引用计数的最基本含义就是跟踪记录每个值被引用的次数。

当声明变量并将一个引用类型的值赋值给该变量时，则这个值的引用次数加 1，同一值被赋予另一个变量，该值的引用计数加 1。当引用该值的变量被另一个值所取代，则引用计数减 1，当计数为 0 的时候，说明无法再访问这个值了，所有系统将会收回该值所占用的内存空间。

#### 存在的缺陷

两个对象的相互循环引用，在函数执行完成的时候，两个对象相互的引用计数并未归 0，而是依然占据内存，无法回收，当该函数执行多次时，内存占用就会变多，导致大量的内存得不到回收。

注意：最常见的就是在 IE BOM 和 DOM 中，使用的对象并不是 js 对象，所以垃圾回收是基于计数策略的。但是在 IE9 已经将 BOM 和 DOM 真正的转化为了 js 对象，所以循环引用的问题得到解决。

## 4、如何管理内存

JS 的内存自动管理一个最主要的问题就是 **分配给 Web 浏览器的可用内存数量通常比分配给桌面应用程序的少**。

为了能够让页面获得最好的性能，必须确保 js 变量占用最少的内存，最好的方式就是将不用的变量引用释放掉，也叫做**解除引用**。

- **局部变量**：函数执行完成离开环境变量，变量将自动解除。
- **全局变量**：对于全局变量我们需要进行手动解除。（注意：解除引用并不意味着被收回，而是将变量真正的脱离执行环境，下一次垃圾回收将其收回）

```
var a = 20; // 在栈内存中给数值变量分配空间
alert(a + 100); // 使用内存
var a = null; // 使用完毕之后，释放内存空间
```

补充：因为通过上边的垃圾回收机制的标记清除法的原理得知，只有与环境变量失去引用的变量才会被标记回收，所用上述例子通过将对象的引用设置为 null，此变量也就失去了引用，等待被垃圾回收器回收。

## 面向对象

### 知识线

什么是面向对象？ => 面向对象编程 => 面向对象编程的特点？(封装、继承、多态)

面试官：什么是面向对象？面向对象的特点有哪些？

### 1、什么是面向对象？

**面向对象是一种编程思想。**

JS 本身就是基于面向对象构建出来的。例如：JS 中很多内置的类，像 `Promise` 就是 ES6 新增的一个内置类，我们可以通过 `new Promise` 来创建一个实例，管理异步编程。

我们平时用到的 `vue/jquery` 都是基于面向对象构建出来的，他们都是类。平常开发的时候，就是创建他们的实例来操作的。

（**根据个人的情况回答**：我自己在真实的项目中，也封装过一些组件插件（颜色组件），也是基于面向对象开发的，这样可以创造出不同的实例，来管理私有的属性和公有方法，很方便.....）

### 2、面向对象编程（OOP）

面向对象编程主要涉及到两种最常用的设计模式，分别是构造函数模式和单例模式。

## 2.1 构造函数模式

### 2.1.1 构造函数的 new

- 通过 new 构造函数来创建该类的实例。
- 自己出创建类名，最好的一个单词首字母大写。
- 这种构造函数设计模式执行，主要用于组件、类库、插件、框架等封装。

### 2.1.2 Object 的创建方式

以下这两者都是 `Object` 类的实例，而实例之间是独立分开的，所以 `var xxx={}` 这种模式就是 JS 中的单例模式。

- 字面量表达式

```
var obj = {}
```

- 构造函数模式

```
var obj = new Object();
```

## 2.2 单例模式

### 2.2.1 由来

每个命名空间都是 JS 中 `Object` 这个内置基类的实例，而实例之间是相互独立互不干扰的，所以我们称它为【单例】——“单独的实例”。

### 2.2.2 高级单例模式

在给命名空间赋值的时候，不是直接赋值一个对象，而是先执行匿名函数，形成一个私有作用域 AA（不销毁的栈内存），在这个 AA 中创建一个堆内存，把堆内存地址赋值给命名空间。

```
var nameSpace = (function () {  
    var n = 12;  
  
    function fn () {  
        // ...  
    }  
  
    return {  
        fn: fn  
    }  
})();
```

完全可以在 AA 中创造很多的内容（变量 OR 函数），哪些需要供外面调取，我们就暴露在返回的对象中。

—— 模块化实现的一种思想。

## 3、封装、继承、多态

JS 中的面向对象，和其他编程语言还是略有不同的，**JS 中类和实例是基于原型和原型链机制来处理的**；

而且 JS 中关于类的重载、重写、继承也和其他语言不太一样。

### 3.1 封装

封装讲究低耦合、高内聚。

### 3.2 继承

回顾上述 JS 继承四种常用的继承。

### 3.3 多态

主要是有关方法的重载和重写。方法重载是让类以统一的方式处理不同类型数据的一种手段。

- **重载**：方法名相同，形参个数或者类型不同。

JS 本来就不具有类的概念，所以 JS 不存在真正意义上的重载，JS 的重载指的是同一个方法，根据传参不同，实现出不同的效果罢了。

- **重写**：在类的继承中，子类可以重写父类的方法。

## 深拷贝和浅拷贝

### 知识线

为什么进行拷贝？ => 有哪些浅拷贝？ => 浅拷贝存在的问题 => 有哪些深拷贝？ => 一般深拷贝存在的问题 =>

最佳深拷贝解决方案

面试官：什么是深拷贝？什么是浅拷贝？

## 1、为什么进行拷贝？

上边在 `JavaScript` 基本类型中我们说到，数据类型分为**基本类型**和**引用类型**。对基本类型的拷贝就是对值复制进行一次拷贝，而对于引用类型来说，拷贝的不是值，而是**值的地址**，最终两个变量的地址指向的是同一个值。还是以前的例子：



```
var a = 10;
var b = a;
b = 30;
console.log(a); // 10值
console.log(b); // 30值

var obj1 = new Object();
var obj2 = obj1;
obj2.name = "小鹿";
console.log(obj1.name); // 小鹿
```

要想将 `obj1` 和 `obj2` 的关系断开，也就是不让他指向同一个地址。根据不同层次的拷贝，分为深拷贝和浅拷贝。

- **浅拷贝**：只进行一层关系的拷贝。
- **深拷贝**：进行无限层次的拷贝。

## 2、浅拷贝

面试官：浅拷贝和深拷贝分别如何实现的？有哪几种实现方式？

### 2.1 自己实现一个浅拷贝

循环遍历对象，将对象的属性和属性值拷贝到另一个对象中，返回该对象。

```
// 实现浅克隆
function shallowClone(o){
  const obj = {};
  for(let i in o){
    obj[i] = o[i]
  }
  return obj;
}
```

### 2.2 扩展运算符实现

ES6 的新语法。

```
let a = {c: 1}
let b = {...a}
a.c = 2
console.log(b.c) // 1
```

### 2.3 `Object.assign(target, source)` 实现

- `target`：目标对象。
- `source`：原对象。

`Object.assign` 将 `source` 的值拷贝到 `target` 目标对象上，进行一层拷贝，如果对象中存在对象，则不能进行拷贝。

```
let a = {c: 1};
let b = Object.assign({}, a);
a.c = 2;
console.log(b.c); // 1
```

### 3、深拷贝

深拷贝，是在浅拷贝的基础上，进行多层遍历拷贝。

#### 3.1 手动实现（最简单实现）

对于深拷贝来说，在浅拷贝的基础上加上递归，我们改动上边自己实现的浅拷贝代码。

```
var a1 = {b: {c: {d: 1}}};
function clone(obj) {
  var target = {};
  for(var i in obj) {
    if (obj.hasOwnProperty(i)) {
      if (typeof obj[i] === 'object') {
        target[i] = clone(obj[i]); // 递归
      } else {
        target[i] = obj[i];
      }
    }
  }
  return target;
}
```

#### 3.2 存在的问题

- **参数没有做检验。** 传入的可能是以下类型数据，`null`、`Date`、`RegExp`、`Array` 等。

```
// 判断传入的参数
function deepClone(obj){
  if(!isObject(obj)) return obj; // 非对象返回自身
  if(obj == null) return null;
  if(obj instanceof Date) return new Date(obj);
  if(obj instanceof RegExp) return new RegExp(obj);
  if(Object.prototype.toString.call(x) === '[object Array]'){
    // 处理数组
  }

  .....
}
```

- **判断对象不够严谨。**

```
// 判断的最佳方案
function isObject(x) {
    return Object.prototype.toString.call(x) === '[object Object]';
}
```

- **没有考虑到数组，以及 ES6 的 set, map, weakset, weakmap 兼容性。**

```
function cloneDeep(obj) {
    if (!isObject(obj)) return source;
    // 考虑数组的兼容
    var target = Array.isArray(obj) ? [] : {};

    ...
}
```

- **循环引用问题（解决办法：暴力破解，循环检测）。**

以下就是循环引用问题，要想解决该问题需要进行循环检测。

```
var a = {};
a.a = a;
clone(a); // 会造成一个死循环
```

所谓的循环检测非常简单，我们设置一个数组或者哈希表用来存储已经拷贝过的对象，检测当前对象是否已经存在于哈希表中，如果有，取出该值，然后返回。

```
function cloneDeep(obj, hash = new Map()) {
    // 非对象返回自身
    if (!isObject(obj)) return obj;

    // 循环检测 -- 如果已存在，直接返回该值
    if (hash.has(obj)) return hash.get(obj);

    // 判断数组是否是对象
    var target = Array.isArray(obj) ? [] : {};

    // 每次都添加未有的对象
    hash.set(obj, target);

    // 开始循环遍历拷贝
    for (var key in obj) {
        if (obj.hasOwnProperty(key)) {
            if (isObject(obj[key])) {
                target[key] = cloneDeep(obj[key], hash); // 新增代码，传入哈希表
            } else {
                target[key] = obj[key];
            }
        }
    }

    return target;
}
```

```
}
```

- **最严重的问题就是递归容易爆栈（递归层次很深的时候）。**

以上都是基于递归实现的，都会容易存在递归爆栈的问题。

```
// RangeError: Maximum call stack size exceeded
```

我们将递归改为循环，这样就不会出现递归爆栈问题。

```
function cloneLoop(x) {
  const root = {};

  // 栈
  const loopList = [
    {
      parent: root,
      key: undefined,
      data: x,
    }
  ];

  while(loopList.length) {
    // 深度优先
    const node = loopList.pop();
    const parent = node.parent;
    const key = node.key;
    const data = node.data;

    // 初始化赋值目标，key 为 undefined 则拷贝到父元素，否则拷贝到子元素
    let res = parent;
    if (typeof key !== 'undefined') {
      res = parent[key] = {};
    }

    for(let k in data) {
      if (data.hasOwnProperty(k)) {
        if (typeof data[k] === 'object') {
          // 下一次循环
          loopList.push({
            parent: res,
            key: k,
            data: data[k],
          });
        } else {
          res[k] = data[k];
        }
      }
    }
  }

  return root;
}
```

## 3.2 JSON.stringify()

还有一个最简单的实现深拷贝的方式，那就是利用 `JSON.parse(JSON.stringify(object))`，但是也存在一定的局限性。

```
function cloneJSON(source) {  
  return JSON.parse(JSON.stringify(source));  
}
```

- 循环引用问题

```
let obj = { a: { c: 1 }, b: {} };  
obj.b = obj;  
console.log(JSON.parse(JSON.stringify(obj))) // 报错  
// Converting circular structure to JSON
```

- 递归爆栈的问题

## 数组去重

### 知识线

最 low 的两种数组去重方案 => 存在的问题和性能 => 如何解决该问题 => 最优秀的实现方案 => 其他数组去重的实现方案

面试官：如何给一个数组去重？

通过实现一个函数，将下列数组中重复的元素进行删除。

```
[1, 2, 1, 1, 1, 2, 3, 3, 3, 2]
```

结果：

```
[1, 2, 3]
```

## 1、最 Low 的两种方案

### 1.1 方案一

创建一个新数组，然后遍历老数组，每次检查当前元素是否在新数组中，如果存在，则不加入。

```

let ary = [1,2,1,1,1,2,3,3,3,2]
let newArray = []
for(let i = 0; i < ary.length; i++){
  let item = ary[i];
  if(!newArray.includes(item)){
    newArray.push(item)
  }
}

console.log(newArray)

```

## 1.2 方案二

每拿出一项数据，就与当前数组其他数据作对比，相同的数据进行删除，依次遍历所有数据。

```

let ary = [1,2,1,1,1,2,3,3,3,2]
for(let i = 0; i < ary.length; i++){
  var item = ary[i];
  for(let j = i + 1; j < ary.length; j++){
    var compare = ary[j];
    if(compare === item){
      ary.splice(j, 1);
      // 数组塌陷问题: j 后边每一项索引都提前了一位，下次要比较应该
      // 还是 j 这个索引
      j--;
    }
  }
}

```

但是上方代码存在一个问题，就是数组塌陷。当我们使用 `splice` 方法删除的时候，通常将后边的数据都往前移动。所以通过 `j--` 来达到正常效果。

## 2、最优秀的两种方案

### 2.1 基于对象去重

对象去重，最主要的是我们优化了上述谈到的数组塌陷问题。如果当前是重复的数据，就将数组的最后一个数据移动到当前数据进行替换，这样重复的数据删除了，而且数组塌陷问题也得到解决。

```

let obj = {}
for(let i = 0; i < ary.length; i++){
  let item = ary[i]
  if(obj[item]){
    // 将最后一项填补当前项
    ary[i] = ary[ary.length - 1]
    // 数组长度减一
    ary.length--;
    // 索引减一
    i--;
    continue;
  }
  obj[item] = item
}

```

```
}  
console.log(ary)
```

## 2.2 基于 Set 去重

Set 是 ES6 的新语法，可以去掉重复的元素。

```
ary = new Set(ary);  
console.log([...ary])
```

# 异步编程

## 知识线

JS 为什么是单线程？ => 单线程带来的问题？ => 如何实现异步编程？ (同步和异步) => 异步代码的执行顺序 => Event Loop 运行机制。

面试官：JS 为什么是单线程？ 又带来了哪些问题呢？

## 1、JS 为什么是单线程？

**JS 单线程的特点就是同一时刻只能执行一个任务。**

这是由一些与**用户的互动以及操作 DOM 等相关的操作**决定了 JS 要使用单线程，否则使用多线程会带来复杂的同步问题。如果是多线程，一个线程正在修改 DOM，另一个线程正在删除 DOM，那么以哪一个为准呢？

如果执行同步问题的话，多线程需要加锁，执行任务造成非常的繁琐。

注意：虽然 HTML5 标准规定，允许 JavaScript 脚本创建多个线程，但是子线程完全受主线程控制，且不得操作 DOM。

## 2、单线程带来的问题？

由于 JavaScript 是单线程的，单线程就意味着**阻塞问题**，当一个任务执行完成之后才能执行下一个任务。这样就会导致出现页面卡死的状态，页面无响应，影响用户的体验，所以不得不出现了**同步任务和异步任务**的解决方案。

- **同步任务**：在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务。
- **异步任务**：不进入主线程、而进入"任务队列"的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

## 3、如何实现异步编程？

为了解决上述问题，出现了异步任务执行，它的运行机制如下：

- 所有同步任务都在主线程上执行，形成一个**执行栈**。
- 主线程之外，还存在一个“任务队列”。只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。
- 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
- 主线程不断重复上面的第三步

**注意：**在主线程读取任务队列的时候，任务队列是一种“先进先出”的数据结构，排在前面的事件，优先被主线程读取。执行栈只要执行空，任务队列的第一个事件进入主线程。同时，如果存在定时器，主线程需要检查定时器执行的时间，到了时间才能给主线程执行。

面试官：JS 如何实现异步编程？

### 3.1 回调函数

回调函数就是作为异步任务的执行。异步任务必须指定回调函数。

回调函数不是直接调用，而是在特定的事件或条件发生时另一方调用的，用于对该事件或条件进行响应。比如 Ajax 回调：

```
// jQuery 中的 ajax
$.ajax({
  type : "post",
  url : 'test.json',
  dataType : 'json',
  success : function(res) {
    // 响应成功回调
  },
  fail: function(err){
    // 响应失败回调
  }
});
```

但是如果某个请求存在依赖性，如下：

```
$.ajax({
  type:"post",
  success: function(res){//成功回调
    //再次异步请求
    $.ajax({
      type:"post",
      url:"...?id=res.id",
      success:function(res){
        $.ajax({
          type:"post",
          url:"...?id=res.id",
          success:function(){
            // 往复循环
          }
        });
      }
    });
  }
});
```



```
}
  })
}
})
}
```

就会形成不断的循环嵌套，我们称之为 **回调地狱**。我们可以看出回调地狱有以下缺点：

- 嵌套函数存在耦合性，一旦有所改动，牵一发而动全身。
- 嵌套函数一多，就很难处理错误。
- 回调函数不能使用 `try catch` 捕获异常(异常的捕获只能在函数执行的时候才能捕获到)。
- 回调函数不能直接 `return`。

### 3.1.1 为什么不能捕获异常？

其实这跟 `js` 的运行机制相关，异步任务执行完成会加入任务队列，当执行栈中没有可执行任务了，主线程取出任务队列中的异步任务并入栈执行，**当异步任务执行的时候，捕获异常的函数已经在执行栈内退出了**，所以异常无法被捕获。

### 3.1.2 为什么不能 `return`？

`return` 只能终止回调的函数的执行，而不能终止外部代码的执行。

## 3.2 解决回调地狱问题

面试官：如何解决回调地狱问题呢？

滑到 ES6 部分异步编程解决方案。

## 4、Event Loop

面试官：说说异步代码的执行顺序？Event Loop 的运行机制是如何的运行的？

主线程从任务队列读取事件，这个过程是循环不断的，所以整个运行机制又称为 **Event Loop(事件循环)**。

在深入事件循环机制之前，需要弄懂一下几个概念：

- **执行上下文** (Execution context)
- **执行栈** (Execution stack)
- **微任务** (micro-task)
- **宏任务** (macro-task)

### 4.1 执行上下文

执行上下文是一个抽象的概念，可以理解为是代码执行的一个环境。JS 的执行上下文分为三种，**全局执行上下文、函数(局部)执行上下文、Eval 执行上下文**。

- **全局执行上下文**：全局执行上下文指的是全局 `this` 指向的 `window`，可以是外部加载的 JS 文件或者本地 `<script></script>` 标签中的代码。
- **函数执行上下文**：函数上下文也称为局部上下文，每个函数被调用的时候，都会创建一个新的局部上下文。
- **Eval 执行上下文**：这个不经常用，所以不多讨论。

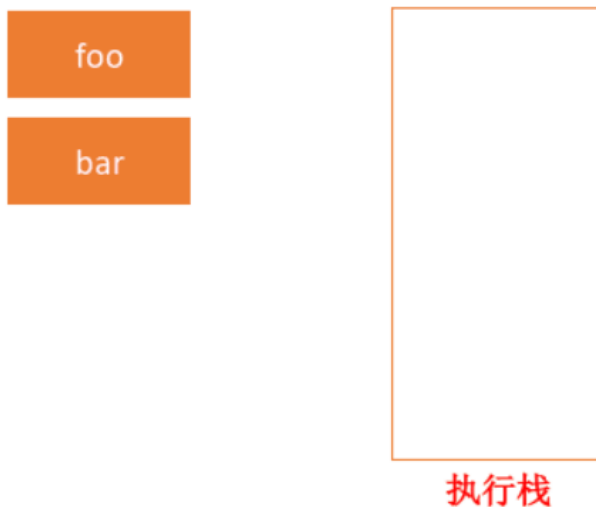
## 4.2 执行栈

执行栈，就是我们数据结构中的“栈”，它具有“先进后出”的特点，正是因为这种特点，在我们代码进行执行的时候，遇到一个执行上下文就将其依次压入执行栈中。

当代码执行的时候，先执行位于栈顶的执行上下文中的代码，当栈顶的执行上下文代码执行完毕就会出栈，继续执行下一个位于栈顶的执行上下文。

```
function foo() {  
  console.log('a');  
  bar();  
  console.log('b');  
}  
  
function bar() {  
  console.log('c');  
}  
  
foo();
```

- 初始化状态，执行栈任务为空。
- `foo` 函数执行，`foo` 进入执行栈，输出 `a`，碰到函数 `bar`。
- 然后 `bar` 再进入执行栈，开始执行 `bar` 函数，输出 `c`。
- `bar` 函数执行完出栈，继续执行执行栈顶端的函数 `foo`，最后输出 `b`。
- `foo` 出栈，所有执行栈内任务执行完毕。



## 4.3 宏任务

对于宏任务一般包括：

- 整体的 `script` 标签内的代码，
- `setTimeout`
- `setInterval`
- `setImmediate(Node)` —— 与 `setTimeout(fn,0)` 相似
- I/O

## 4.4 微任务

对于微任务一般包括：

- `Promise`
- `process.nextTick(Node)` —— 所有异步任务之前触发
- `MutationObserver`

注意：nextTick 队列会比 Promise 队列先执行。

以上概念弄明白之后，再来看循环机制是如何运行的呢？以下涉及到的任务执行顺序都是靠函数调用栈来实现的。

- 1) 首先，事件循环机制的是从 `<script>` 标签内的代码开始的，上边我们提到过，整个 `script` 标签作为一个宏任务处理的。
- 2) 在代码执行的过程中，如果遇到宏任务，如：`setTimeout`，就会将当前任务分发到对应的执行队列中去。
- 3) 当执行过程中，如果遇到微任务，如：`Promise`，在创建 `Promise` 实例对象时，代码顺序执行，如果到了执行 `then` 操作，该任务就会被分发到微任务队列中去。
- 4) `script` 标签内的代码执行完毕，同时执行过程中所涉及到的宏任务也和微任务也分配到相应的队列中去。
- 5) 此时宏任务执行完毕，然后去微任务队列执行所有的存在的微任务。
- 6) 微任务执行完毕，第一轮的消息循环执行完毕，页面进行一次渲染。
- 7) 然后开始第二轮的消息循环，从宏任务队列中取出任务执行。
- 8) 如果两个任务队列没有任务可执行了，此时所有任务执行完毕。

## 4.5 定时器 `setTimeout`

任务队列除此之外，还可以放定时器的回调函数，需要指定某些代码多少时间之后执行。

定时器主要包括两种，`setTimeout` 和 `setInterval` 两个函数。

当我们设置定时器的时间，执行某个特定的任务，如下：

```
// 1 秒后执行
setTimeout(function(){
    console.log(2);
},1000);
console.log(1)
```

上述的输出结果为 1, 2, 执行完同步代码后, 就会执行定时器中的任务事件。

```
// 同步执行完后立即执行
setTimeout(function(){
    console.log(2);
},0);
console.log(1)
```

当我们执行 `setTimeout(fn,0)` 定时器时, 会将这个定时任务回调放在任务队列的尾部, 代表的含义就是尽早的执行。

也就是等到主线程同步任务和"任务队列"现有的事件都处理完, 然后才会立即执行这个定时器的任务。

**上述的前提是, 等到同步任务和任务队列的代码执行完毕后, 如果当前代码执行很长时间, 定时器没办法保证一定在指定时间执行。**

注意: HTML5 标准规定了 `setTimeout()` 的第二个参数的最小值 (最短间隔), 不得低于4毫秒, 如果低于这个值, 就会自动增加。

如果涉及到页面的改动, 这个定时器任务通常不会立即执行, 而是 16 毫秒执行一次, 我们通常使用 `requestAnimationFrame()`。

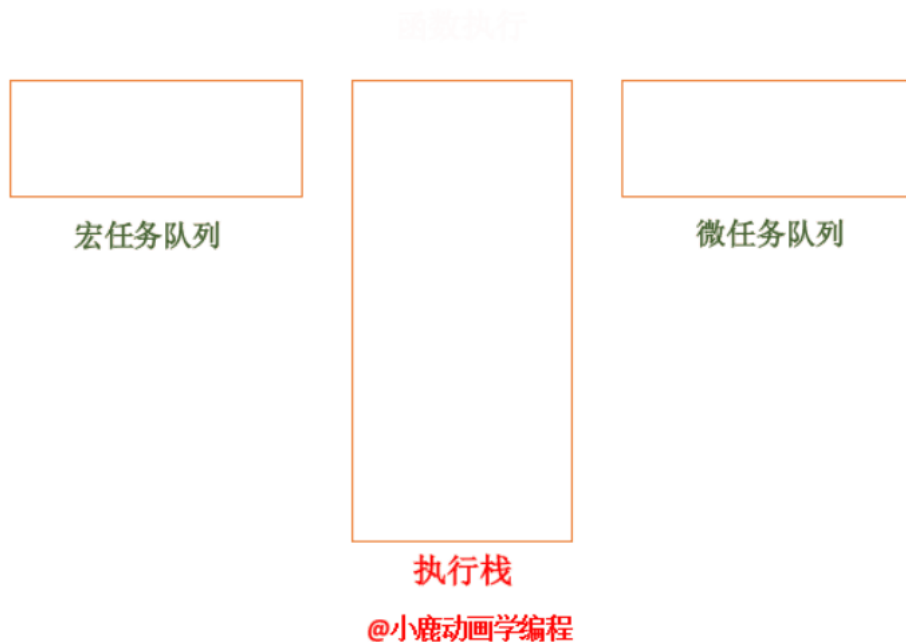
**实战一下:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>消息运行机制</title>
</head>
<body>

</body>
<script>
    console.log('1');
    setTimeout(() => {
        console.log('2')
    }, 1000);
    new Promise((resolve, reject) => {
        console.log('3');
        resolve();
        console.log('4');
    }).then(() => {
        console.log('5');
    });
```

```
console.log('6');// 1,3,4,6,5,2
</script>
</html>
```

- 初始化状态，执行栈为空。
- 首先执行 `<script>` 标签内的同步代码，此时全局的代码进入执行栈中，同步顺序执行代码，输出 1。
- 执行过程中遇到异步代码 `setTimeout`（宏任务），将其分配到宏任务异步队列中。
- 同步代码继续执行，遇到一个 `promise` 异步代码（微任务）。但是构造函数中的代码为同步代码，依次输出 3、4，则 `then` 之后的任务加入到微任务队列中去。
- 最后执行同步代码，输出 6。
- 因为 `script` 内的代码作为宏任务处理，所以此次循环进行到处理微任务队列中的所有异步任务，直达微任务队列中的所有任务执行完成为止，微任务队列中只有一个微任务，所以输出 5。
- 此时页面要进行一次页面渲染，渲染完成之后，进行下一次循环。
- 在宏任务队列中取出一个宏任务，也就是之前的 `setTimeout`，最后输出 2。
- 此时任务队列为空，执行栈中为空，整个程序执行完毕。



以上难免有些啰嗦，所以简化整理如下步骤：

- 一开始执行宏任务（`script` 中同步代码），执行完毕，调用栈为空。
- 然后检查微任务队列是否有可执行任务，执行完所有微任务。
- 进行页面渲染。
- 第二轮从宏任务队列取出一个宏任务执行，重复以上循环。

## ES6 基础知识点

### var/let/const

### 知识线

什么是变量提升？ => 为什么会存在变量提升 => 变量和函数如何进行提升？ => let、const、var 之间的区别 =>

## 1、变量提升

面试官：什么是变量提升？为什么存在变量提升？

变量提升就是该变量还未被声明，但是却可以使用未声明的变量。

### 1.1 为什么存在变量提升？

虽然描述为提升，但不是真正的将代码提升到顶部，而是在代码执行前，先在词法环境中进行了注册。

如果没有变量提升，下方代码就无法执行：

```
function fn1(){
  fn2();
}

function fn2(){
  fn1();
}

fn1();
```

变量提升的根本原因就是为了解决函数之间互相调用的情况。

### 1.2 变量和函数提升的规则？

面试官：变量和函数怎么进行提升的？优先级是怎么样的？

- **第一阶段：**对所有**函数声明**进行提升（**忽略表达式和箭头函数**），引用类型的赋值（函数的声明，函数的提升代表着可执行，因为提升之后保持着引用）分为三步：
  - 开辟堆空间
  - 存储内容
  - 将地址赋值给变量
- **第二阶段：**对**所有的变量**进行提升，全部赋值为 `undefined`（如果已经存在，不赋值为 `undefined`）。然后依次顺序执行代码。

PS：使用 `let` 和 `const` 时，我们不能在声明之前使用变量，这叫做暂时性死区。

### 1.3 var、let、const

面试官：var、let、const 三者的区别什么？

- `var` 存在变量提升，而 `let`、`const` 则不会。
- `var` 在浏览器环境下声明的变量会挂载到 `window` 上，而其他两者不会。
- `let` 和 `const` 的作用基本一致，后者声明的变量不能再次赋值（注意：但是能改变值）

## map、filter、reduce

### 知识线

map => filter => reduce => 三者的区别是什么？ => 在实际项目中如何使用？

面试官：说说 map、filter、reduce 三者的使用以及区别是什么？

### 1、map

`map` 的作用是 `map` 中传入一个函数，该函数会遍历该数组，对每一个元素做变换之后返回新数组。

- `element`：对应数组的每个元素。
- `index`：数组元素的下标。
- `arr`：原数组。

```
let arr = [2,3,4]
arr = arr.map(function(element,index,arr){
    return arr[index]+1;
}) // [3,4,5]
```

### 2、filter

`filter` 的作用也是生成一个数组，传入的函数返回值确实布尔类型，返回值为 `true` 的元素放入新数组，通常来筛选删除不需要的元素。

- `element`：对应数组的每个元素。
- `index`：数组元素的下标。
- `arr`：原数组。

```
let array = [1, 2, 4, 6]
let arr = array.filter(function(element){
    return element != 6;
})// [1,2,4]
```

### 3、reduce

`reduce` 可以将数组中的元素通过回调函数最终转换为一个值。

- `acc`：累计值(第一次的值代表初始化的值)。

- `element`: 当前元素。
- `index`: 当前索引。
- `arr`: 原数组。

```
let arr = [1,2,3]
let sum = arr.reduce(function(acc,element){
    return acc + element;
},0) // 6
```

## 代理

### 知识线

`getter/setter` 访问属性的好处 => `getter/setter` 有几种使用方式? (3 种) => `Proxy` 和 `definedProperty` 使用区别? => `Proxy` 的基本应用

对于 `Proxy` 代理我们从最基本的 `getter` 和 `setter` 说起~

面试官: 通过 `getter` 与 `setter` 访问属性值有什么好处? 代理与 `getter` 和 `setter` 的主要区别是什么?

我们为什么会使用 `getter` 和 `setter` 来进行控制对值的访问呢, 有以下三个作用:

- 避免意外的错误发生。
- 需要记录属性的变化: 比如属性值的访问日志记录。
- 数据绑定: 在 `vue` 中使用的数据的双向绑定。

对于有哪几种方式来定义 `getter` 和 `setter` 呢?

- 字面量定义;
- ES6 中的 `class` 定义;
- 使用 `Object.defineProperty` 方法;

### 3.1 字面量定义

对象访问属性通常隐式调用 `getter` 和 `setter` 方法, 属性自动关联 `getter` 和 `setter` 方法。这些都是标准方法, 在访问属性的时候都是立即执行的。

```
const collection = {
    name:'xiaolu',

    // 读取属性
    get name(){
        return this.name;
    }
    // 设置属性
    set name(value){
```



```

        this.name = value;
    }
}
collection.name           // 隐士调用 getter 方法
collection.name = "xiaolu" // 隐士调用 setter 方法

```

### 3.2 ES6 中的 Class 定义

```

// class 定义 setter 和 getter
class xiaolu {
    constructor(){
        this.name = 'xiaolu'
    }

    get firstxiaolu(){
        console.log('属性已访问')
        return this.name;
    }

    set firstxiaolu(value){
        this.name = value;
    }
}

const x = new xiaolu();
x.firstxiaolu

```

### 3.3 Object.defineProperty()

对象的字面量与类、getter 和 setter 方法不是在一同一作用域定义的，因此那些希望作为私有变量属性的标量是无法实现的。

```

// Object.defineProperty
function xiaolu(){
    let count = 0;

    Object.defineProperty(this, 'skillLevel', {
        get: () => {
            return count;
        },
        set: value => {
            count = value;
        }
    })
}

// 隐士的调用 get 方法
console.log(xiaolu.count)

```

通过 `Object.defineProperty()` 创建的 `get` 和 `set` 方法，与私有的变量处于相同的作用域中，`get` 和 `set` 方法分别创建了含有私有变量的闭包。

### 3.4 Proxy 代理

面试官：说说 ES6 中的 Proxy 和 getter、setter 的区别？

代理 proxy 是 ES6 新提出的。代理（proxy）使我们通过代理控制对另一个对象的访问。proxy 和 getter 以及 setter 的区别是，getter 和 setter 仅仅控制的是单个对象属性，而 proxy 代理的是对象交互的通用处理，包括对象的方法。

用法：

- target：要进行代理的对象。
- handler：是一个特定的对象，定义了特定行为时触发的函数。

```
var proxy = new Proxy(target, handler);
```

get 和 set 的参数：

- target：传入的对象。
- key：属性。
- value：要赋的值。

```
const obj = {name: 'xiaolu'}
const representtive = new Proxy(obj, {
  get: (target, key) => {
    return key in target ? target[key] : "不存在该值"
  },
  set: (target, key, value) => {
    target[key] = value;
  }
})
```

我们试图想使用代理访问对象时，此时会触发 get 方法。或者试图将代理的对象进行赋值时，会触发调用 set 方法。

面试官：说说 Proxy 的基本应用？

最基本的应用如下一个：

- 日志记录 —— 当访问属性时，可以在 get 和 set 中记录访问日志。
- 校验值 —— 有效的避免指定属性类型错误的发生。
- 定义如何计算属性值 —— 每次访问属性值，都会进行计算属性值。
- 数据的双向绑定（vue） —— 在 vue3.0 中将会通过 Proxy 来替换原本的 Object.defineProperty 来实现数据响应式。

```
// 校验值
function xiaolu(){
  let count = 0;

  Object.defineProperty(this, 'skillLevel', {
    get: () => {
      return count;
    }
  })
}
```

```

    },
    set: value => {
      if (!Number.isInteger(value)) {
        throw new TypeError("抛出错误")
      }
      count = value;
    }
  })
}

```

```

// 定义如何计算属性值
const collection = {
  name: 'xiaolu',
  age: '2',
  // 读取属性
  get getName(){
    return this.name + " "+this.age;
  },
  // 设置属性
  set setName(value){
    this.name = value;
  }
}

console.log(collection.getName)

```

## ES6/7 的异步编程

### 知识线

generator 的使用 => generator 内部实现 => Promise 的使用方式 => Promise 的使用原理 => Promise 源码实现

### 1、Generator

面试官：说说 Generator 是如何使用的？以及各个阶段的状态是如何变化的？

使用生成器函数可以生成一组值的序列，每个值的生成是基于每次请求的，并不同于标准函数立即生成。

调用生成器不会直接执行，而是通过叫做**迭代器**的对象控制生成器执行。

```
function* WeaponGenerator(){
  yield "1";
  yield "2";
  yield "3";
}

for(let item of weaponGenerator()){
  console.log(item);
}
//1
//2
//3
```

使用迭代器控制生成器。

- 通过调用生成器返回一个迭代器对象，用来控制生成器的执行。
- 调用迭代器的 `next` 方法向生成器请求一个值。
- 请求的结果返回一个对象，对象中包含一个 `value` 值和 `done` 布尔值，告诉我们生成器是否还会生成值。
- 如果没有可执行的代码，生成器就会返回一个 `undefined` 值，表示整个生成器已经完成。

```
function* WeaponGenerator(){
  yield "1";
  yield "2";
  yield "3";
}

let weapon = WeaponGenerator();
console.log(weapon.next());
console.log(weapon.next());
console.log(weapon.next());
```

状态变化如下：

- 每当代码执行到 `yield` 属性，就会生成一个中间值，返回一个对象。
- 每当生成一个值后，生成器就会非阻塞的挂起执行，等待下一次值的请求。
- 再次调用 `next` 方法，将生成器从挂起状态唤醒，中断执行的生成器从上次离开的位置继续执行。
- 直到遇到下一个 `yield`，生成器挂起。
- 当执行到没有可执行代码了，就会返回一个结果对象，`value` 的值为 `undefined`，`done` 的值为 `true`，生成器执行完成。

面试官：说说 Generator 内部结构实现？

生成器更像是一个状态运动的状态机。

- 挂起开始状态——创建一个生成器处于未执行状态。
- 执行状态——生成器的执行状态。
- 挂起让渡状态——生成器执行遇到第一个 `yield` 表达式。
- 完成状态——代码执行到 `return` 全部代码就会进入全部状态。

执行上下文跟踪生成器函数。

```
function* weaponGenerator(action){
  yield "1"+action;
  yield "2";
  yield "3";
}

let Iterator = weaponGenerator("xiao1u");
let result1 = Iterator.next()
let result2 = Iterator.next()
let result3 = Iterator.next()
```

- 在调用生成器之前的状态——只有全局执行上下文，全局环境中除了生成器变量的引用，其他的变量都为 `undefined`。
- 调用生成器并没有执行函数，而是返回一个 `Iterator` 迭代器对象并指向当前生成器的上下文。
- 一般函数调用完成上下文弹出栈，然后被摧毁。当生成器的函数调用完成之后，当前生成器的上下文出栈，但是在全局的迭代器对象还与保持着与生成器执行上下文引用，且生成器的词法环境还存在。
- 执行 `next` 方法，一般的函数会重新创建执行上下文。而生成器会重新激活对应的上下文并推入栈中（这也是为什么标准函数重复调用时，重新从头执行的原因所在。与标准函数相比较，生成器暂时会挂起并将将来恢复）。
- 当遇到 `yield` 关键字的时候，生成器上下文出栈，但是迭代器还是保持引用，处于非阻塞暂时挂起的状态。
- 如果遇到 `next` 指向方法继续在原位置继续执行，直到遇到 `return` 语句，并返回值结束生成器的执行，生成器进入结束状态。

## 2、Promise

面试官：说说 Promise 的原理？你是如何理解 Promise 的？

在深入 `Promise` 之前，我们先想想 Why（为什么）会有 `Promise`，`Promise` 的诞生解决了哪些问题呢？

小鹿总结到的原因有两方面，第一，由于 JS 的运行是单线程的，所以当执行耗时的任务时，就会造成 UI 渲染的阻塞。当前的解决方法是使用回调函数来解决这个问题，当任务执行完毕会，会调用回调方法。

### 为什么使用 Promise

第二就是回调函数存在以下几个缺点：

- 不能捕捉异常（错误处理困难）——回调函数的代码和开始任务代码不在同一事件循环中；
- 回调地域问题（嵌套回调）—— `promise` 链式调用；
- 处理并行任务棘手（请求之间互不依赖）—— `promise.all`；

### 2.1 Promise 简单使用

```
let promise = new Promise(function(resolve, reject) {
    resolve();
});

promise.then((res)=>{
    console.log("回调成功！")
},(err) =>{
    console.log("回调失败！")
});
```

- 通过内置的 `Promise` 构造函数可以创建一个 `Promise` 对象，构造函数中传入两个函数参数：`resolve`，`reject`。两个参数的作用是，在函数内手动调用 `resolve` 的时候，就说明回调成功了；调用 `reject` 说明调用失败。通常在 `promise` 中进行耗时的异步操作，响应是否成功，我们根据判断就可以调用对应的函数。
- 调用 `Promise` 对象内置的方法 `then`，传入两个函数，一个是成功回调的函数，一个失败回调的函数。当再 `promise` 内部调用 `resolve` 函数时，之后就会回调 `then` 方法里的第一个函数。当调用了 `reject` 方法时，就会调用 `then` 方法的第二个函数。
- `promise` 相当于是一个承诺，当承诺兑现的时候（调用了 `resolve` 函数），就会调用 `then` 中的第一个回调函数，在回调函数中做处理。当承诺出现未知的错误或异常的时候（调用了 `reject` 函数），就会调用 `then` 方法的第二个回调函数，提示开发者出现错误。

## 2.2 Promise 的状态

其实 `Promise` 对象用作异步任务的一个占位符，代表暂时还没有获得但在未来获得的值。

`Promise` 共有三种状态，完成状态和拒绝状态都是由等待状态转变的。一旦 `Promise` 进入了拒绝或完成状态，它的状态就不能切换了。

- 等待状态 (pending)
- 完成状态 (resolve)
- 拒绝状态 (reject)

`Promise` 共有两种拒绝状态：显示拒绝（直接调用 `reject`）和隐式拒绝（抛出异常）。

### 一个 `Promise` 实例：

客户端请求服务器是最常用的异步任务，下面实现一下。

```
function getJson(url){
    // 返回一个 promise 对象
    return new Promise((resolve, reject)=>{
        const request = new XMLHttpRequest();
        request.open("GET", url);

        // 服务器响应成功
        request.onload = function(){
            //try-catch 考虑到 JSON 可能解析会出现错误
            try{
                // 响应码正确且 JSON 解析正确
                if(this.status == 200){
                    // 调用成功承诺
                    resolve(JSON.parse(this.response))
                }else{
```

```

        // 调用失败承诺
        reject(this.status + " " +this.statusText);
    }
}catch(e){
    // 调用失败承诺
    reject(e.message)
}
}

// 服务器响应失败
request.onerror = function(){
    // 调用失败承诺
    reject(this.status + " " +this.statusText);
}

// 发送请求
request.send();

})
}

// 调用上述异步请求
getJSON("url").then((res)=>{
    console.log("解析的JSON数据为:"+res)
},(res)=>{
    console.log("错误信息: "+res)
})

```

## 2.3 Promise 链式调用

面试官：什么是 Promise 链？说说 Promise 如何使用的？

`promise` 可以实现链式调用，每次 `then` 之后返回的都是一个 `promise` 对象，可以紧接着使用 `then` 继续处理接下来的任务，这样就实现了链式调用。如果在 `then` 中使用了 `return`，那么 `return` 的值也会被 `Promise.resolve()` 包装。

```

Promise.resolve(1)
  .then(res => {
    console.log(res) // => 1
    return 2 // 包装成 Promise.resolve(2)
  })
  .then(res => {
    console.log(res) // => 2
  })

```

`Promise` 最常用的使用方式有以下几个方式：

**嵌套任务处理：**

```
// 链式回调
getJSON("url")
  .then(n => {getJSON(n[0].url)})
  .then(m => {getJSON(m[0].url)})
  .then(w => {getJSON(w[0].url)})
  .catch((error => {console.log("异常错误！")}))
```

- 因为 `then` 方法会返回一个 `promise` 对象，所以连续调用 `then` 方法可以进行链式调用 `promise`。
- 多个异步任务中可能出现错误，只需要调用一个 `catch` 方法并向其传入错误处理的回调函数。

## 2.3 Promise 并行请求

上述的链式调用主要处理的是多个异步任务之间存在依赖性的，如果同时执行多个异步任务，就是用 `promise` 中的 `all` 方法。

```
// 并行处理多个异步任务
Promise.all([getJSON(url),
             getJson(url),
             getJson(url)]).then(result => {
  // 如果三个请求都响应成功
  if(result[0] == 1 && result[1] == 1 && result[2] == 1){
    console.log("请求成功！")
  }
}).catch(error => {console.log("异常错误！")})
```

- 使用 `Promise.all()` 方法进行异步请求，将多个请求任务封装数组进行同步请求。
- 返回的结果值会打包成一个数组，可以通过数组的下标获取值对返回的结果进行判断。
- 只有全部请求成功才会进入成功的方法，否则就会调用 `catch` 抛出异常。
- 与 `Promise.race()` 方法不同的是，`race` 方法只要其中一个返回成功，就会调用成功的方法。

## 2.4 如何实现一个 Promise

面试官：说说怎么实现一个 Promise？

我们根据 `Promise` 的执行顺序，手动实现一个 `Promise`。

- 先执行 `MyPromise` 构造函数；
- 注册 `then` 函数；
- 此时的 `promise` 挂起，`UI` 非堵塞，执行其他的同步代码；
- 执行回调函数。

```
// 三种状态
const PENDING = "pending";
const RESOLVE = "resolve";
const REJECT = "reject";

// promise 函数
function MyPromise(fn){
  const that = this;          // 回调时用于保存正确的 this 对象
```



```

that.state = PENDING; // 初始化状态
that.value = null; // value 用于保存回调函数(resolve/reject 传递的参数值)

that.resolvedCallbacks = []; // 用于保存 then 中的回调
that.rejectedCallbacks = [];

// resolve 和 reject 函数
function resolve(value) {
  if(that.state === PENDING){
    that.state = RESOLVE;
    that.value = value;
    that.resolvedCallbacks.map(cb => cb(that.value));
  }
}

function reject(value) {
  if (that.state === PENDING) {
    that.state = REJECT
    that.value = value
    that.rejectedCallbacks.map(cb => cb(that.value))
  }
}

// 实现如何执行 Promise 中传入的函数
try {
  fn(resolve, reject)
} catch (e) {
  reject(e)
}

}

// 实现 then 函数
MyPromise.prototype.then = function(onResolved, onRejected) {
  const that = this;
  // 判断两个参数是否为函数类型(如果不是函数, 就创建一个函数赋值给对应的参数)
  onResolved = typeof onResolved === 'function' ? onResolved : v => v;
  onRejected = typeof onRejected === 'function' ? onRejected : r => {throw r}

  // 判断当前的状态
  if (that.state === 'pending') {
    that.resolvedCallbacks.push(onResolved)
    that.rejectedCallbacks.push(onRejected)
  }
  if (that.state === 'resolve') {
    onResolved(that.value)
  }
  if (that.state === 'reject') {
    onRejected(that.value)
  }
}

new MyPromise((resolve, reject) => {
  setTimeout(() => {
    resolve(1)
  }, 0)
}).then(value => {
  console.log(value)
})

```

### 3、async 及 await

面试官：async 及 await 和 Generator 以及 Promise 什么区别？它们的优点和缺点分别是什么？await 原理是什么？

其实 ES7 中的 `async` 及 `await` 就是 `Generator` 以及 `Promise` 的语法糖，内部的实现原理还是原来的，只不过是在写法上有所改变，这些实现一些异步任务写起来更像是执行同步任务。

#### 3.1 特点

一个函数前加上 `async` 关键字，就将该函数返回一个 `Promise`，`async` 直接将返回值使用 `Promise.resolve()` 进行包裹（与 `then` 处理效果相同）。

`await` 只能配套 `async` 使用，`await` 内部实现了 `generator`，`await` 就是 `generator` 加上 `Promise` 的语法糖，且内部实现了自动执行 `generator`。

- **内置执行器。** `Generator` 函数的执行必须靠执行器，所以才有了 `co` 函数库，而 `async` 函数自带执行器。也就是说，`async` 函数的执行，与普通函数一模一样，只要一行。
- **更好的语义。** `async` 和 `await`，比起星号和 `yield`，语义更清楚了。`async` 表示函数里有异步操作，`await` 表示紧跟在后面的表达式需要等待结果。
- **更广的适用性。** `co` 函数库约定，`yield` 命令后面只能是 `Thunk` 函数或 `Promise` 对象，而 `async` 函数的 `await` 命令后面，可以跟 `Promise` 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

缺点：

因为 `await` 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 `await` 会导致性能上的降低。

#### 3.2 `async/await` 的实现原理

`async` 函数的实现，就是将 `Generator` 函数和自动执行器，包装在一个函数里。

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    var gen = genF();
    function step(nextF) {
      try {
        var next = nextF();
      } catch(e) {
        return reject(e);
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        reject(e);
      });
    }
    step(function() {});
  });
}
```

```
    step(function() { return gen.throw(e); });
  });
}
step(function() { return gen.next(undefined); });
});
}
```

# 模块化

## 知识线

为什么要使用模块化？ => 有几种模块化的使用方式？ => 每个使用方式内部如何实现的？ => 模块化之间的区别是什么？

## 1、为什么使用模块化？

面试官：为什么要使用模块化？都有哪几种方式可以实现模块化，各有什么特点？

模块化解决了命名冲突问题，可以提高代码的复用率，提高代码的可维护性。

模块化的好处：

- 避免命名冲突(减少命名空间污染)
- 更好的分离, 按需加载
- 更高复用性
- 高可维护性

## 2、使用模块化的几种方式？

### 2.1 方式一：函数

最起初，实现模块化的方式使用函数进行封装。将不同功能的代码实现封装到不同的函数中。通常一个文件就是一个模块，有自己的作用域，只向外暴露特定的变量和函数。

```
function a(){
  // 功能二
}
function b(){
  // 功能一
}
```

**缺陷：**容易发生命名冲突或者数据的不安全性。

### 2.2 方式二：立即执行函数

立即执行函数中的匿名函数中有独立的词法作用域，避免了外界访问此作用域的变量。**通过函数作用域解决了命名冲突、污染全局作用域的问题。**

```
// module.js文件
(function(window) {
  let name = 'xiaolu'
  // 暴露的接口来访问数据
  function a() {
    console.log(`name:${name}`)
  }
  //暴露接口
  window.myModule = { a }
})(window)
```

```
<script type="text/javascript" src="module.js"></script>
<script type="text/javascript">
  myModule.name = 'xixi' // 无法访问
  myModule.foo() // name:xiaolu
</script>
```

**缺陷：**不能直接访问到内部的变量。

### 3、方式三：CommonJS 规范

CommonJS 的规范主要用在 Node.js 中，为模块提供了四个接口：module、exports、require、global，CommonJS 用同步的方式加载模块（服务器端），在浏览器端使用的是异步加载模块。

#### 3.1 暴露模块

主要通过两种方式：

- module.exports = {}
- exports.xxx = 'xxx'

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
// 对外暴露接口
module.exports = {
  counter: counter,
  incCounter: incCounter,
};
```

#### 3.2 引入模块

主要通过 require 的方式引入，但是 require 是 Node 的语法，在浏览器中无法识别。

```
// 加载外部模块
var mod = require('./lib');

console.log(mod.counter); // 3
mod.incCounter();
// 原始类型的值被缓存，所以就没有被改变（commonJS 不会随着执行而去模块随时调用）
console.log(mod.counter); // 3
```

除此之外，还有其它的三种模块标识符。

```
// 核心模块直接导入
const path = require('path');

// 路径模块
const m = require('./m.js');

// 自定义模块
const lodash = require('lodash');
```

- 核心模块，直接跳过路径分析和文件定位；
- 路径模块，直接得出相对路径就好了；
- 自定义模块，先在当前目录的 `node_modules` 里找这个模块，如果没有，它会往上一级目录查找，查找上一级的 `node_modules`，依次往上，直到根目录下都没有，就抛出错误。

### 3.3 实现原理

浏览器不兼容 `CommonJS` 的原因有是因为缺少 `Node` 环境变量。

- `module`
- `exports`
- `require`
- `global`

```
var module = {
  exports: {}
};

(function(module, exports) {
  exports.multiply = function (n) { return n * 1000 };
})(module, module.exports)

var f = module.exports.multiply;
f(5) // 5000
```

### 3.4 `CommonJS` 的特点

- `CommonJS` 模块的加载机制是，输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。
- 所有代码都运行在模块作用域，不会污染全局作用域。
- 模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。

- 模块加载的顺序，按照其在代码中出现的顺序。

## 4、方式四：AMD 和 CMD

上边有我们的 `CommonJS` 规范了，为什么还出 `AMD` 规范，因为 `CommonJS` 是同步加载代码的，我们在浏览器中会发生堵塞问题，造成页面的无响应。所以浏览器不太适合使用 `CommonJS` 来加载。

`CommonJS` 规范对浏览器和服务端的不同之处。

- 服务器端所有的模块都存放在本地硬盘中，可以同步加载完成，等待时间就是硬盘的读取时间。
- 浏览器，所有的模块都放在服务器端，等待时间取决于网速的快慢，可能要等很长时间，浏览器处于“假死”状态。

### 4.1 AMD

`AMD` (`Asynchronous Module Definition`)，即“异步模块定义”。它主要采用异步方式加载模块，模块的加载不影响它后边语句的运行。所加载的模块，都会定义在回调函数中，加载完成，再执行回调函数。

#### 4.1.1 使用方式

`defined` 是一个 `javascript` 库中的一个方法，使用之前需要安装一个库。

```
npm i requirejs
```

使用语法如下：

```
define(id, dependencies, factory)
```

- `id`: 一个字符串，表示模块的名称。
- `dependencies`: 一个数组，是我们当前定义的模块要依赖于哪些模块，数组中的每一项表示的是要依赖模块的相对路径。
- `factory`: 工厂方法，具体模块内容。

#### 4.1.2 导出模块

我们将 `add.js` 中的一个函数导出语法如下：

```
define(function () {  
  var add = function (a, b) {  
    return a + b;  
  }  
  return {  
    add: add  
  }  
})
```

### 4.1.3 引入模块

导入上述导出的模块。

```
var requirejs = require("requirejs"); //引入 requirejs 模块

requirejs(['add'],function(math) {
    console.log(add.add(1, 2));
})
```

## 4.2 CMD

CMD (Common Module Definition), 主要是 `seajs` 的规范。

AMD 和 CMD 最大的区别是对依赖模块的执行时机处理不同, 注意不是加载的时机或者方式不同, 二者皆为异步加载模块。

### 4.2.1 使用方式

```
// 所有模块都通过 define 来定义
define(function(require, exports, module) {

    // 通过 require 引入依赖
    var $ = require('jquery');
    var Spinning = require('./spinning');

    // 通过 exports 对外提供接口
    exports.doSomething = ...

    // 或者通过 module.exports 提供整个接口
    module.exports = ...
});
```

```
define(function (require, exports, module) {
    console.log('我比 m1 要早加载....')
    var m1 = require('m1');    // 用到时才加载
    var add = function (a, b) {
        return a + b;
    }
    var print = function () {
        console.log(m1.name)
    }
    module.exports = {
        add: add,
        print: print
    }
})
```

### 4.2.2 AMD 和 CMD 的区别

- AMD 依赖前置，js 很方便的就知道要加载的是哪个模块了，因为已经在 `define` 的 `dependencies` 参数中就定义好了，会立即加载它。
- CMD 是就近依赖，需要使用把模块变为字符串解析一遍才知道依赖了那些模块。只有在用到某个模块的时候再去 `require`。

## 5、方式五：ES6 Moudle

ES6 实现的模块非常简单，用于浏览器和服务端。 `import` 命令会被 JavaScript 引擎静态分析，在编译时就引入模块代码。

面试官：说说 ES6 中的模块化？CommonJS 和 ES6 模块化的区别？

### 5.1 export 导出模块

两种导出方式：

- 命名式导出
- 默认导出

```
// 方式一
const a = 1;
export { a };

// 方式二
export const a = 1;
export const b = 2;

// 方式三(as 重命名导出)
const a = 1;
export { a as A };
```

```
const a = 1;
export default a;

// 等价于
export { a as default };
```

### 5.2 import 导入模块

- 命名式导入
- 默认导入

```
// 默认导入
import { a } from './module';

// 重新命名
import { a as A } from './module';

// 只想要运行被加载的模块
```



```
import './module';

// 整体加载
import * as module from './module'

// default接口和具名接口
import module, { a } from './module'
```

### 5.3 ES6 和 CommonJS 的区别

- **CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。**
  - 所谓值的拷贝，原始类型的值被缓存，不随模块内部的变化而改变。
  - ES6 模块是**动态引用**，**不缓存值**，模块内外是**绑定的**，而且是**只读引用**，不能修改值。ES6 的 js 引擎对脚本静态分析的时候，遇到加载命令模块 `import`，就会生成一个只读引用，当真正用到模块里边的值的时候，就会去模块内部去取。
- **CommonJS 模块是运行时加载，ES6 模块是编译时加载输出接口。**
  - **运行时加载：**CommonJS 模块就是对象；是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。
  - **编译时加载：**ES6 模块不是对象，而是通过 `export` 命令「显式指定输出的代码」。`import` 时采用静态命令的形式，即在 `import` 指定「加载某个输出值」，而「不是加载整个模块」，这种加载称为“编译时加载”。
- **CommonJS 导入的模块路径可以是一个表达式，因为它使用的是 `require()` 方法；而ES6 Modules只能是字符串**
- **CommonJS `this` 指向当前模块，ES6 Modules `this` 指向 undefined**

## 第二版正在书写中 ...

---

