Preliminary·笔者说

嗨咯，其实每一个部分我会先给出思维导图，这就是笔者的一个思路，对于这一部分的体系结构做出的规划。笔者还是一个菜鸟小白，但是在诸多年的学习中，已经深刻发现架建知识体系结构是多么的重要，不管什么学科，不管什么阶段的学习，搭建知识体系都是很重要。

我们开发软件，也会有软件的生命周期，这就是一个开发软件的必经之路，软件生命周期内有问题定义、可行性分析、总体描述、系统设计、编码、调试和测试、验收与运行、维护升级到废弃等阶段。这就是一个process，我们在其之中就可以更加明确高效的进行软件管理。那么，学习也是一样的，应该有一个学习路线。现在我引用下atguigu.com的前端路线。


前端学习路线图
www.atguigu.com
1 前端核心基础阶段
1-网页界面技术1: HTML
2-网页界面技术2: CSS
3-网页交互技术1: JavaScript
4-网页交互技术2: DOM
5-网页交互技术3: BOM
2 前端核心高级
1-网页界面新型技术1: HTML5
2-网页界面新型技术2: CSS3
3-移动端开发
4-前端样式库: BootStrap
5-前端绘图技术: Canvas
6-快速简洁图表可视化库: ECharts
7-流行图表可视化库: D3
8-使用最广JS函数库: jQuery
9-移动端类jQuery库: zepto
10-版本控制工具1: SVN
11-版本控制工具2: GIT&Github
3 JS后台技术
1-强大的后台JS: Node.js
2-高效的非关系型数据库: MongoDB
3-简洁的MongoDB操作库: Mongoose
4-后台Web开发框架: Express
5-后台模板技术: EJS
6-前端台实时通信库: socketIO
4 前后端交互阶段
1-AJAX
2-新型ajax请求方式: fetch
3-前后台ajax请求库: axios
4-REST API
5-跨域全面解决方案
6-mock后台数据的库: mockjs
7-接口调试神器: postman
1-JS作用域与作用链
2-JS原型与原型链
3-JS闭包
4-JS事件循环机制
5-ES6新特性
6-ES7新特性

这就是一个比较完整的前端的路线图，但是我想表达的并不是他的知识体系的完整。我更想通过这张图给大家分享我的看法，这里分了七个板块，每个板块在细分进去，它们上下之间是有联系在里面，说浅显点就是前面的是基础知识，后面的是高级知识，它们在知识点上是串联的，你需要的就是把这些知识串起来。开始形成自己的知识体系。把前端路线比作一栋高楼大厦，那么它一开始肯定是先把整个框架架建出来之后，再去加砖添瓦，再去粉刷精修。比如图中七个部分，每一部分就像高楼的每一层，每一层里面会有很多房间，它们既是独立的，也是有联系的。而你要做的是先学懂每个知识点，关键要知道你的知识点是属于哪个大楼？哪一层？哪一个房间？好的，确定下来之后，就把你学的这块知识丢进去。丢进去之后，房间也会很杂乱，所以你还是得整理，把衣服放在一堆，生活用品放在一起等等。这样子，你在学习中，就不会觉得很乱，或者学了一段时间后并不能自己独立去思考学了什么。因为你没有在平常学习中去培养这种高楼大厦的知识体系，那么你又怎么可能做到去对知识的熟能生巧和相互联系呢？

就如笔者在这一部分就会给出js高级的部分知识，那么你就要把这部分知识放在“前端知识”的高楼大厦中的第5层------JS高级知识。第5层有很多房间，比如有数据类型的相关知识，那么就把有关数据类型的相关知识放在一个房间里，当然这个房间也需要整理：数据类型的分类有哪些？有哪些手段可以对数据类型进行判断？然后又在数据类型分类的基础上，继而去划分出基本类型和对象类型，又在基本类型中划分number、string、Boolean、undefined、null、Symbol 类型。就是需要这种“一层一层拨开你的心”的感觉。

说了这么多，其实并不是废话。以后每一部分都会给出思维导图，那么你需要看之前整体把握思维导图，看完之后反复思考思维导图，在思维导图上细细思考内部的知识逻辑和联系。**必须要有构建个人知识体系的意识。**当然，笔者也做得很不好。暂时这种水平完全驾驭不了，但一点点来，总可以做好。

另外笔者有一些学习中的见解：

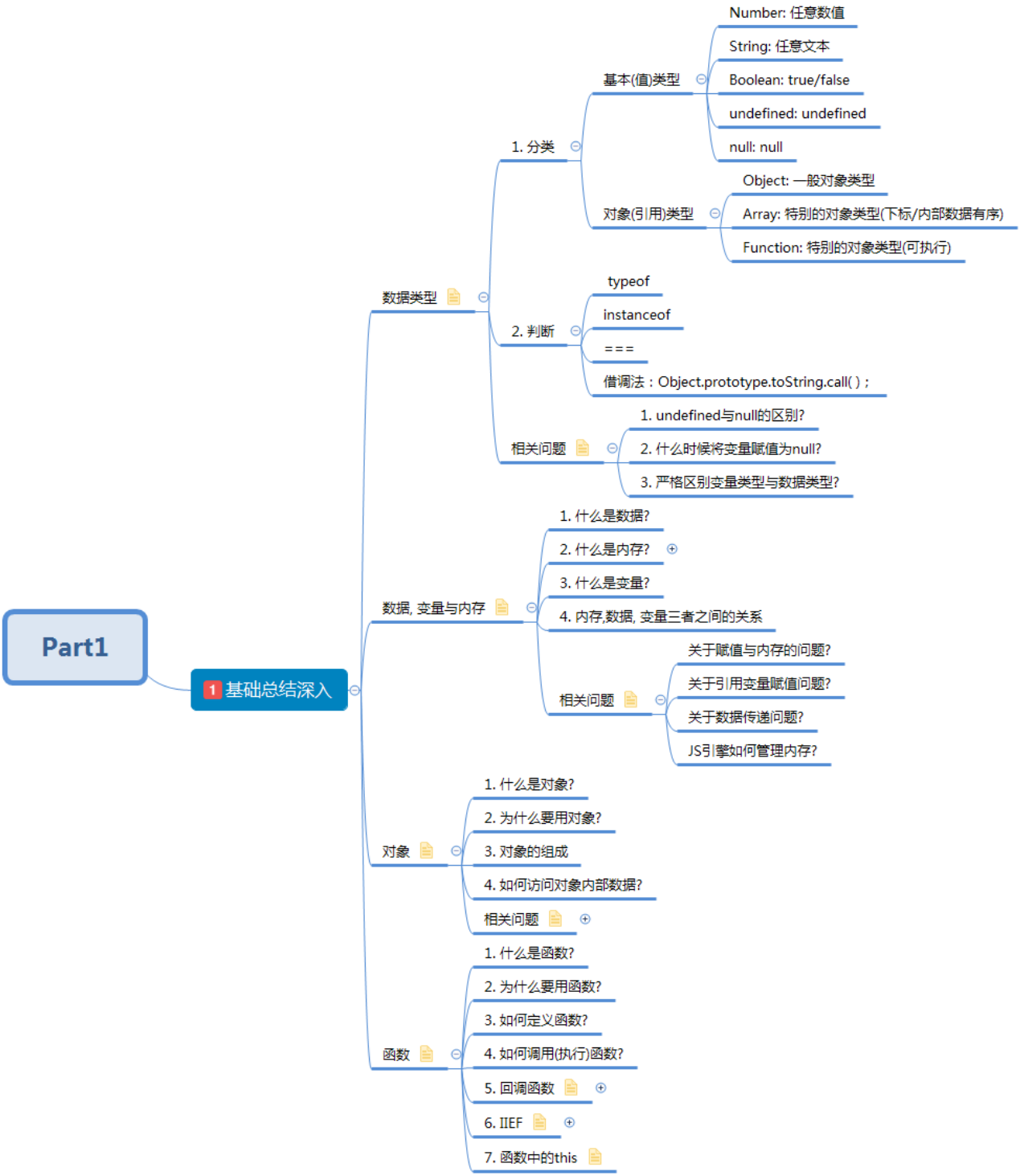
P1: 平常我们出现问题时，会去在网路上看一些博客，以帮助自己的理解。但是别忘记了还有两种手段，那就是我们人类最原始的学习方式，纸质书，像一些计算机的经典的书籍可以拿来看一看，有时候真的会很有帮助，网路上的碎片化的知识居多，但书籍整合的资料会更加完备。第二种手段就是原文标准.比如ECMA官方文件（ES6）我有时候会去看一下，就会发现，里面的规范规则实则就是很好的总结归纳，真得很详细。所以看一些原文标准文件是真的很不错。

P2: 一个非常不错的整合前端全部知识的文档

p3: ES567 GITBOOK

Part1：JS基础知识深入总结

> Part 1：思维导图



1.1 数据类型的分类和判断

1.1.1 数据类型的分类

- 基本(值)类型
  - Number ------- 可以任意数值 --------- 可以用typeof检测数据类型

    - String ----------可以任意字符串 ------- typeof
    - Boolean ------- 只有true/false --------- typeof
    - undefined ---- 只有undefined --------- typeof / ===
    - null ---------- 只有null---------- ===
    - 注：ES6新增Symbol。
- 对象(引用)类型
    - Object ----- 可以用typeof/instanceof检测数据类型

      描述：可以任意对象
    - Array ------ instanceof

      描述：一种特别的对象（有数值下标，而且内部数据是有序的。一般的对象内部的数据是无序的，比如你一个对象中有name和age，他们是无序的。）
    - Function ---- typeof

      描述：一种特别的对象（可以去执行的对象，内部包含可运行的代码。一个普通的对象可以执行吗？不能。）另外，对象是存储一些数据的，当然函数也是存储一些代码数据。

1.1.2 数据类型的判断

- 如何判断数据类型？
    - typeof 返回数据类型的字符串（小写）表达。特别注意：typeof不能去判断出 null与object object与array。

注1：用typeof来测试有以下七种输出结果：‘number’ ‘string’ ‘boolean’ 'object' 'function' 'symol' 'undefined‘。 因此typeof不能去判断出 null与object，因为用typeof去判断null会输出object。
注2：所有的任何对象，用typeof测试数据类型都是object。因此，typeof不能去判断出object与array。

    - === 全等 只可以判断undefined 和 null 因为这个类型只有一个值。
    - instanceof 比如：A instanceof B 翻译就是B的实例对象是A 吗？ 判断对象的具体类型（到底是对象类型中的Object Array Function的哪一个具体的类型），返回一个Boolean值。
    - 借调法：Object.prototype.toString.call( )；

      这种方法只可以检测出内置类型（引擎定义好的，自定义的不行），这种方法是相对而言更加安全。

      Object Date String Number RegExp Boolean Array Math Window等这些内置类型。

      例子见1.1.6习题与案例部分的借调法。

1.1.3 四个问题

- undefined与报错（not defined）的区别？
    - 对象.属性　属性不存在则返回undefined
    - 访问变量　变量不存在则报错，xx is not defined

```
 obj={
    name:'lvya'
  };
console.log(obj.age); //undefined
console.log(age);  //报错，age is not defined
```

```
 //从这个细节去看一个问题？
 function Person(name,age,price) {
    this.name = name
    this.age = age
    this.price=price
    setName=function (name) {
      this.name=name;
    }
  }
  var p1 = new Person('LV',18,'9kw')
  console.log(p1.price); // 9kw
//请问p1.price 先找啥？后找啥？通过啥来找？（问题问的不好，直接看答案吧）
//An:p1.price先找p1 后找 price 。 p1是一个全局变量哦，栈内容存着地址值，指向一个对象。先找p1是沿着作用域找的，后找price是沿着原型链找的。这就是联系，从另外一个方面细看问题。可能这样看问题，你就可以把原型链和作用域可以联系起来思考其他问题。
```

- undefined与null的区别？
  - undefined代表定义未赋值
  - nulll定义并赋值了，只是值为null

```
var a;
  console.log(a);  // undefined
  a = null;
  console.log(a); // null
```

使用Object.prototype.toString.call()形式可以具体打印类型。

如果值是undefined，返回“[object Undefined]”。

如果这个值为null，则返回“[object Null]”。

```
undefined实际上代表了不存在的值（non-existence of a value）。通常遇到当试图访问一个不存在的值时。在这种情况下，在JavaScript这种动态的弱类型语言中，只会默认返回一个undefined值，而不是上升为一个错误：
1、任何声明变量时没有提供一个初始值，都会有一个为undefined的默认值
2、当试图访问一个不存在的对象属性或数组项时，返回一个undefined值
3、如果省略了函数的返回语句,返回undefined
4、函数调用时未提供的实参值结果形参将为undefined/5、void操作符也可以返回一个undefined值。像Underscore的库使用它作为一个防御式的类型检查，因为它是不可变的，可以在任何上下文依赖返回undefined
5、undefined是一个预定义的全局变量(不像null关键字)初始化为undefined值
```

返回null的情况：
1、DOM，它是独立于语言的，不属于ECMAScript规范的范围。因为它是一个外部API，试图获取一个不存在的元素返回一个null值，而不是undefined。
2、通过分配null值，有效地清除引用，并假设对象没有引用其他代码，指定垃圾收集，确保回收内存。

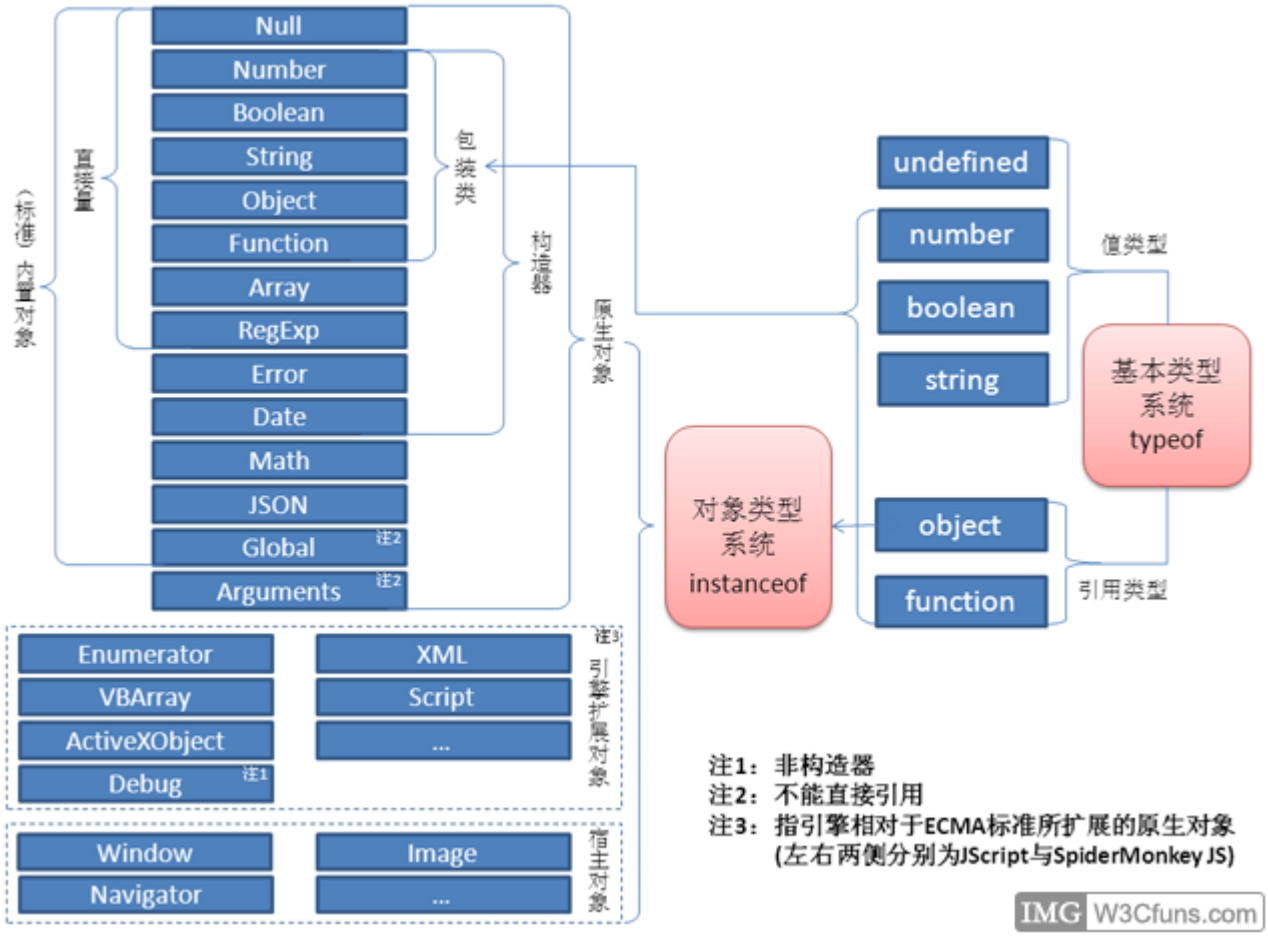
- 什么时候给变量赋值为null 呢?
  - 初始赋值, 表明这个变量我将要去赋值为对象
  - 结束前, 这个对象不再使用时，让对象成为垃圾对象(被垃圾回收器回收)

```
//起始
var b = null   // 初始赋值为null，表明变量b将要赋值为对象类型
//确定赋值为对象
b = ['lvya', 12]
//结束，当这个变量用不到时
b = null  // 让b指向的对象成为垃圾对象(被垃圾回收器回收)
// b = 2  //当然让b=2也可以，但不常使用
```

- 严格区别变量类型与数据类型？
  - 数据的类型
    - 基本类型
    - 对象类型
  - 变量的类型(实则是变量内存值的类型)
    - 基本类型: 保存就是基本类型的数据
    - 引用类型: 保存的是地址值

```
var c = function () {
  }
  console.log(typeof c) // 'function'

//理解：
//1：JavaScript是一种弱类型的语言，实则他的变量是没有类型的，都是var。但是我们一般说变量的类型是指变量内存值的类型。
//2： 这个案例中变量c去用typeof检测类型时为什么会知道是function呢？ 变量c是存储在内存中的栈内存，而对象是放在堆内存中的，在栈内存保存的是这个对象的地址值，所以，在检测typeof中找到变量c ，知道他保存了一个地址值，通过查找地址值的指向，才发现他是一个function。故变量c是属于引用类型。 再问？c本身是一个函数对象吗？--不是，c本身保存的是地址值。只是可以通过这个地址值找到一个对象。
//3：但一般我们都是对应起来说的，对象类型就是引用类型。
```

1.1.4 一张图看懂JavaScript各类型的关系



1.1.5 valueOf( ) 与 toString( )

- toString()和valueOf() 都是在Object.prototype里面定义.
- toString() :
  表示的含义是把这个对象表示成字符串形式, 并且返回这个字符串形式.
  在Object.prototype中的默认的实现是: "[object Object]"
  可以在自己的对象或者原型对 toString进行覆写(重写, override)

```
//1.首先看一下默认实现"[object Object]"
 var p={};
 console.log(p.toString());   //[object Object]   去Object.prototype的去找

 function Person(){
 }
 var p1=new Person();
 console.log(p1.toString());  //[object Object]  去Object.prototype的去找
```

```
  //2.再看一下可以在自己的对象或者原型对 toString进行覆写(重写, override)
   var p = {
   toString: function (){
     return "100";
     }
   };
   console.log(p.toString());  //100  这个时候就会在首先在P对象上找toString ( ) 方法，这个时候就是对toString方法的重写
```

```
  var date = new Date();
  console.log(date.toString());   //Fri Jun 08 2018 10:54:37 GMT+0800 (中国标准时间)
   /*从输出结果可知，Date这个构造函数的原型其实是有toString ( ) 方法的，说明JS引擎已经在Date原型对象中重写了toString ( ) 方法，故不会在Object.prototype中找*/
  console.log(Date.prototype);  //确实有toString ( )

  var n = new Number(1);
  console.log(n.toString());  //1 ( 字符串 )
```

- valueOf()
  应该返回这个对象表示的基本类型的值!
  在Object.prototype.valueOf 中找到， 默认返回的是this
  需要重写的时候, 应该返回一个基本类型的值.

```
  //1.先看默认返回的值
    function Person(){
    }
    var p1 = new Person();
    console.log(p1.valueOf() == p1);  //true  这个时候valueOf是在Object.prototype.valueOf找到的，返回值默认this。此时this就是p1的这个对象
```

```
   //2.再看重写valueOf后的情况
    var p = {
      toString: function (){
        return "100";
      },
      valueOf : function (){
        return 1;
      }
    };
    console.log(p.toString());  //100(字符串)   没有找Object.prototype.valueOf 而是找其本身的toString方法
    console.log(p.valueOf());  //1 ( number数据类型 ) 没有找Object.prototype.valueOf 而是找其本身的toString方法
```

```
    var n = new Number(100);
    console.log(n.valueOf());  //100 （number类型）

    var s = new String("abc");
    console.log(s.valueOf());  //abc (string类型)

    var regExp = /abc/gi;
    console.log(regExp.valueOf() === regExp);  //true 说明这个时候正则对象上没有valueOf,是在Object.prototype.valueOf找的，返回this，this指的就是regExp正则对象。

    /*结论：在js中，只有基本类型那几个包装类型进行了重写，返回的是具体的基本类型的值，其他的类型都没有重写.*/
```

- 了解完valueOf（ ）和toSting（ ）方法后，其实他们就是对象与基本数据类型的比较基础。我们数据类型，分为基本数据类型和对象两种，故在数据类型比较中，就会有三种情况：
  - 基本数据类型间的比较
  - 对象间的比较
  - 对象与基本数据类型的比较

```
//1.两个基本数据类型间的比较
  // 如果类型相同，则直接比较
  // 如果类型不同，都去转成number再去比较
  // 特殊:
  //   1. undefined == null
  //   2. 0和undefined，0和null都不等
  //   3. 如果有两个 NaN  参与比较，则总是不等的。
  var a=1;
  var b=1;
  console.log(a == b); //true  都是基本类型，而且都是基本类型中的number类型

//看些例子
//一下首先都是基本类型，但是其具体的类型不同,转为number类型
  console.log(true == "true");  //false  1与Nan比较
  console.log(0 == "0");   //true  0与0比较
  console.log(0 == "");   //true   0与0比较
  console.log("0" == "");   //false  都是相同的string类型，不用转，直接用字符串比较
  console.log(undefined == null);  //true  Nan与0比较
```

```
//2.两个对象间的比较
  // === 和 == 完全一样.
  // 其实比较是不是同一个对象. 比的就是他们的地址值是否一样。

  console.log({} === {});   //false  地址值不同
  console.log(new Number(1) == new Number(1));   //false  地址值不同
```

```
// 3. 基本类型和对象进行比较
  // 把对象转成基本类型的数据之后再比
  // 如何把对象转换成基本类型:
  //   1. 先调用这个对象(注意是对象)的valueOf方法, 如果这个方法返回的是一个基本类型的值, 则用这个基本类型去参与比较.
  //   2. 如果valueOf返回的不是基本类型, 则去调用toString() 然后用返回的字符串去参与比较.
//看例子1:
  var p = {};
    console.log(p == "[object Object]");  //true 对象与字符串比较
    console.log(p.valueOf());  //{}
    //步骤:step1:调用对象p的valueOf(),因为没重写,所以去Object.prototype去找,返回this,此时this是Object。
    //step2:因为返回的不是基本类型的值,又调用toString(),因为没重写,所以去Object.prototype去找,返回字符串"[object Object]"
```

```
//看例子2:
 var p1 = {
    valueOf : function (){
      return 'abc';
   },
    toString : function (){
      return {};
    }
  }

  console.log(p1 == "abc");  //true  对象与字符串比较

  //对象与基本数据类型比较,先把对象调用valueOf()

  //步骤:step1:调用对象p1的valueOf(),因为有重写,故调用重写的valueOf(),返回字符串'abc',此时返回的不是对象而是基本数据类型中的string类型,故直接比较。

  //现在,就是都是基本类型进行比较了,故相等。
```

1.1.6 习题与案例

```
//1. 基本数据类型部分
    // typeof返回数据类型的字符串(小写)表达
    var a;
    console.log(a, typeof a, typeof a === 'undefined', a === undefined)  // undefined 'undefined' true true
    console.log(undefined === 'undefined'); //false
    a = 4;
```

```
    console.log(typeof a === 'number');  //true
    a = 'lvya';
    console.log(typeof a === 'string');  //true
    a = true;
    console.log(typeof a === 'boolean');  //true
    a = null;
    console.log(typeof a, a === null); // 'object'  true

//2. 对象类型部分
    var b1 = {
        b2: [1, 'abc', console.log],
        b3: function () {
            console.log('b3');
            return function () {
                return 'ya Lv'
            }
        }
    };

    console.log(b1 instanceof Object, b1 instanceof Array); // true  false
    console.log(b1.b2 instanceof Array, b1.b2 instanceof Object) ;// true true
    console.log(b1.b3 instanceof Function, b1.b3 instanceof Object); // true true

    console.log(typeof b1.b2); // 'object'

    console.log(typeof b1.b3 === 'function');// true

    console.log(typeof b1.b2[2] === 'function'); // true
    b1.b2[2](4);  //4
    console.log(b1.b3()());  //ya Lv

    //instanceof一般测对象类型，那它去测基本数据类型会怎样？可知，1并不是Number类型的实例
    console.log(1 instanceof Number);  //false
    //new Number(1)的确是Number类型的实例
     console.log(new Number(1) instanceof Number);  //true


//3.借调法检测数据类型部分
console.log(Object.prototype.toString.call({}));  //[object Object]
console.log(Object.prototype.toString.call(new Date()));  //[object Date]
console.log(Object.prototype.toString.call(new Array()));  //[object Array]
console.log(Object.prototype.toString.call(window));  //[object Window]
console.log(Object.prototype.toString.call(document));  //[object HTMLDocument]
console.log(Object.prototype.toString.call(document.body)); //[object HTMLBodyElement]


//数据类型的面试题
//用typeof来测试有以下七种输出结果：'number'   'string'  'boolean'  'object'  'function'  'symol'
  'undefined'。
console.log(typeof "ab");    // string
console.log(String("ab"));   //'ab'  可以知道String("ab")就是var s='ab'的含义
console.log(typeof String("ab"));  // string

console.log(typeof  new String("ab")); // object
```

```
console.log(typeof /a/gi);    // object
console.log(typeof [0,'abc']);  // object
console.log(typeof function (){});  //function
var f = new Function("console.log('abc')");
f();  //'abc'  可以知道f就是一个函数
console.log(typeof f);   //function
console.log(typeof  new Function("var a = 10"));  //function
```

```
//面试：考点：加号的运用
//以及string Boolean undefined null 转number类型的结果
/*考点1：js加号有两种用法
case1：数学上的加法（只要没有字符串参与运算就一定是数学上的数字）：
case2:字符串连接符（只要有一个是字符串，那就是字符串链接）
*/

/*分别输出什么?*/
console.log(1 +  "2" + "2"); // 122
console.log(1 +  +"2" + "2"); // 32
console.log(1 +  -"1" + "2"); // 02
console.log(+"1" +  "1" + "2"); // 112
console.log( "A" - "B" + "2"); // NaN2
console.log( "A" - "B" + 2); // NaN

//见下表一是其他四种基本数据类型转为number类型（显性转换）
//另外其他类型转Boolean类型的情况，记住，只有五种转Boolean是false的。
//  0   NaN  undefined  null   ""   false  ===>  false
//其他情况都是true，比如"  "  "false"都是true
```

其他四种类型	number类型	其他四种类型	number类型	其他四种类型	number类型
undefined	Nan	'0'	0	false	0
'12a'	Nan	'' ' ' '\t'	0	true	1
'\'	Nan	null	0	‘abc’	Nan

在ECMA官方文档中总结的很好，很清晰。看下面的图，就不翻译了。所以，有时候大家真的可以去找这种官方文档去看一些知识点，它本身就总结的相当完备。

7.1.2 ToBoolean (argument)

The abstract operation ToBoolean converts *argument* to a value of type Boolean according to Table 10:

Table 10 — ToBoolean Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToBoolean(*argument*.[[value]]).
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return *argument*.
Number	Return **false** if *argument* is **+0**, **−0**, or **NaN**; otherwise return **true**.
String	Return **false** if *argument* is the empty String (its length is zero); otherwise return **true**.
Symbol	Return **true**.
Object	Return **true**.

7.1.3 ToNumber (argument)

The abstract operation ToNumber converts *argument* to a value of type Number according to Table 11:

Table 11 — ToNumber Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToNumber(*argument*.[[value]]).
Undefined	Return **NaN**.
Null	Return **+0**.
Boolean	Return **1** if *argument* is **true**. Return **+0** if *argument* is **false**.
Number	Return *argument* (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint Number). 2. Return ToNumber(*primValue*).

```
//面试题。考查valueOf和toString 以及 类型转换的知识。
console.log([] == ![]); //true
/*
*左边是对象，首先用valueOf 返回的是一个数组对象
*然后再调用toString 返回一个空的字符串，因为数组转字符串，就是去掉左右“中括号”，把值和逗号转为字符串
*console.log([].valueOf()); //[]
*console.log([].valueOf().toString()); //空的字符串
*故左边是一个空的字符串--空字符串（字符串）转number就是0
*右边是！[]是一个整体。由上一个面试案例总结的数据类型转换可知，世界上只有五种转为Boolean值是false的，故对象转为是一个true。加个！就是false了。false（Boolean值）转number就是0
*0==0 故true。
*/
```

```
//面试题：考点，&& ||在js中一个特别灵活的用法。如果第一个能最终决定结果的，那么结果就是第一个值，否则就是第二个。
/*
考察逻辑运算符
与和或
   优先级"与"高于"或"
*/
console.log(1 && 2 || 0); // 2
console.log((0 || 2 && 1)); //1  （注意，这里是先计算2 && 1，因为&&优先级高于||）
console.log(3 || 2 && 1); // 1   （注意，这里是先计算2 && 1，因为&&优先级高于||）
console.log(0 && 2 || 1); // 1
```

```
//面试题。考点，类型的转换
var bar = true;
console.log(bar + 0);  // 1 (+当做数字相加，两边都转number)
console.log(bar + "xyz"); // truexyz （+当做字符串连接）
console.log(bar + true); // 2  (+当做数字相加，两边都转number)
console.log(bar + false); // 1 (+当做数字相加，两边都转number)
console.log(bar + undefined); // NaN (+当做数字相加，两边都转number)
console.log(bar + null); // 1 (+当做数字相加，两边都转number)
console.log(undefined + null); // NaN (+当做数字相加，两边都转number。Nan与任何运算结果都是Nan)
console.log([1, 2] + {});  //1,2[object Object]

Array.prototype.valueOf = function () {
  return this[0];
};
console.log([1, 2] + [2]);  //3  重写了Array的valueOf方法，其重写后返回的是this[0],因为在这是number类型1，故直接用。

console.log([{}, 2] + [2]);  // [object Object],22   重写了Array的valueOf方法，其重写后返回的是this[0],因为在这是一个对象{}，故在对这个对象（[{},2]）调用toString（）返回‘[object Object],2’。这里要注意当调用toString是整个对象，而非重写valueOf后返回来的对象。  +右边的[2]是调用了valueOf之后返回的number类型2，所以直接用，因为左边是一个字符串，所以加号代表字符串拼接。返回最终结果[object Object],22
```

1.2 数据，变量， 内存的理解

1.2.1 什么是数据?

- 存储在内存中特定信息的'东东'，本质上是0101...的二进制
- 数据的特点: 可传递, 可运算

```
  var a = 3;
  var b = a;
//这里体现了数据的可传递。
//变量a是基本数据类型，保存的值是基本数据类型。在栈内存中存储。这两个语句传递的是变量a吗？-不是。传递的是数据3.实际上，是拿到变量a的内容3拷贝一份到b的小内存空间中。
//注意，每个小的内存空间都是有地址值的，在堆内存中，在栈内存空间中的每一块小内存中都会有地址值，只是要不要用的问题。那谁的地址值一般要用？--对象，对象的地址值一般会用到。其次，在进行这两条语句执行中，不是读取a的内存地址值，而是直接拿到内容拷贝给b。
```

- 一切皆数据, 函数也是数据

> 那么一切皆对象吗?万物皆对象？--我的理解是这句话不太准确，比如我们的数据类型分为基本数据类型和引用类型，那么基本数据类型是对象吗？不是吧。所以我一般不会说一切皆对象，但会说一切皆数据。
> 具体解释：基本数据类型中除了null都不是对象，一切引用类型和null都是对象，所以javascript中一切皆对象，确切说应该是一切引用类型都是对象。有些人会说：“好，如果基本类型不是对象，为什么我们可以调用他们的方法呢” 回答是包装对象。
> 当你尝试调用基本类型的方法，JavaScript在幕后做了一个巧妙的处理，将你的基本类型的值转换成临时对象用于构造函数，决定使用哪个构造函数取决于你尝试改变的基本类型的值，在String中调用.length会使用string()构造函数临时将基本类型转变成对象–允许你使用length方法而改变它，这个临时对象被称为包装对象。
> 有趣的是，null和undefined这两个基本类型不能调用这样的方法，否则会提示类型错误。
> 这一点也无关紧要。查了资料，说法不一。

- 在内存中的所有操作的目标: 数据
  - 算术运算
  - 逻辑运算
  - 赋值
  - 运行函数 eg：调用函数fn( ); 那么( )是否也可以看做是一种操作数据的方式呢，去执行代码块。

1.2.2 什么是变量?

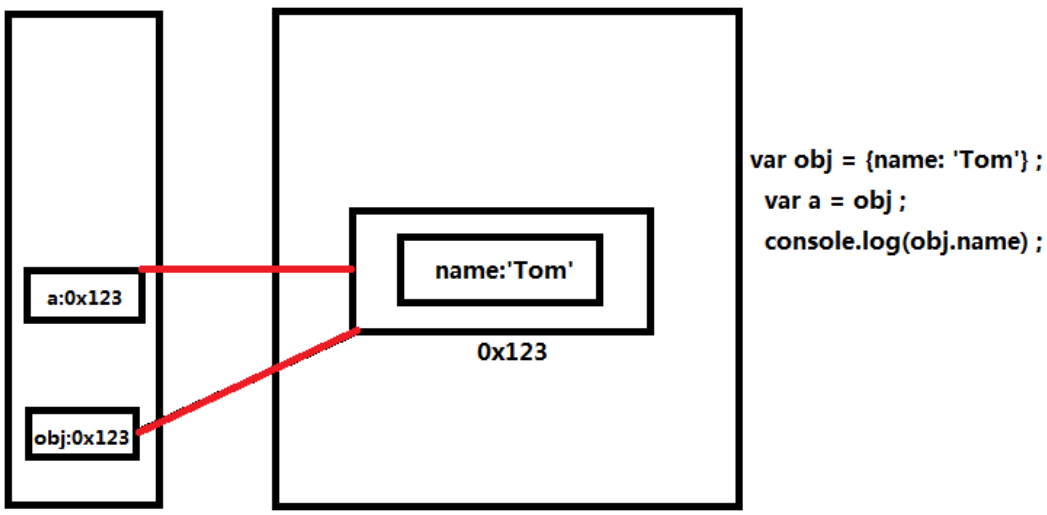
- 在程序运行过程中它的值是允许改变的量，由变量名和变量值组成
- 一个变量对应一块小内存，它的值保存在此内存中 。变量名用来查找对应的内存, 变量值就是内存中保存的数据。通过变量名先去找到对应的内存，然后再去操作变量值。

1.2.3 什么是内存?

- 内存条通电后产生的可存储数据的空间(临时的)

- 硬盘的数据是永久的，但其处理数据慢，内存快，但是临时的。
- 内存产生和死亡: 内存条(电路版)==>通电==>产生内存空间==>存储数据==>处理数据==>断电==>内存空间和数据都消失
- 一块小的内存包含2个方面的数据
  - 内部存储的数据（内容数据）
  - 地址值数据（只有一种情况读的是地址值数据，那就是将一个对象给一个变量时）

```
var obj = {name: 'Tom'} ;
var a = obj ;
console.log(obj.name) ;
//理解 ：
//var obj = {name: 'Tom'} ; 是将右边的这个对象的地址值给变量obj，变量obj读的就是对象的内存数据中的地址值数据。
//而var a = obj ; 首先右边不是一个对象，是一个变量（引用类型的变量），这时是将右边的这个变量的内存数据中的内部存储数据，也就是读取其内容数据0x123给变量a.而不是读取变量obj的内存数据中的地址值数据（每个小的内存都有自己的地址值，只是需不需要用而已。）简单来说就是，把obj的内容拷贝给a,而刚好obj的存储的内容是一个对象的地址值。（这句话必须理解透）
//console.log(obj.name) ;  这个读的是obj的内容值。
//总结：什么时候读的是地址值？只有把一个对象赋值给一个变量时才会读取他在自己的内存块中的内存数据的地址值数据。
```



- 内存空间的分类
  - 栈空间: 全局变量和局部变量【空间比较小】
  - 堆空间: 对象（指的是对象本身在堆空间里，而函数也是对象，其本身在堆内存中。但函数名在栈空间里）【空间比较大】

```
function fn () {
    var obj = {name: 'Tom'}
  }
  //obj在栈空间里  name是在堆空间里
```

1.2.4 内存，数据， 变量三者之间的关系

- 内存是容器, 用来存储不同数据
- 变量是内存的标识, 通过变量我们可以操作(读/写)内存中的数据

1.2.5 一些相关问题

问题一：var a = xxx, a内存中到底保存的是什么?

分类讨论:

- 当xxx是基本数据, 保存的就是这个数据

```
var a = 3;
//3是基本数据类型，保存的就是3.
```

- 当xxx是对象, 保存的是对象的地址值

```
a = function () {
  }
  //函数是对象，那么a保存的就是这个函数对象的地址值。
```

- 当xxx是一个变量, 保存的xxx的内存内容(这个内容可能是基本数据, 也可能是地址值)

```
//分类1：
var b = 'abc'
    a = b
    //b是一个变量，而b本身内存中的内容是一个基本数据类型。所以，a也是保存这个基本数据类型'abc'

//分类2：
    b = {}
    a = b
    //b是一个变量，而b本身内存中的内容是一个对象的地址值。所以，a也是保存这个对象的地址值'0x123'
```

问题二：关于引用变量赋值问题？

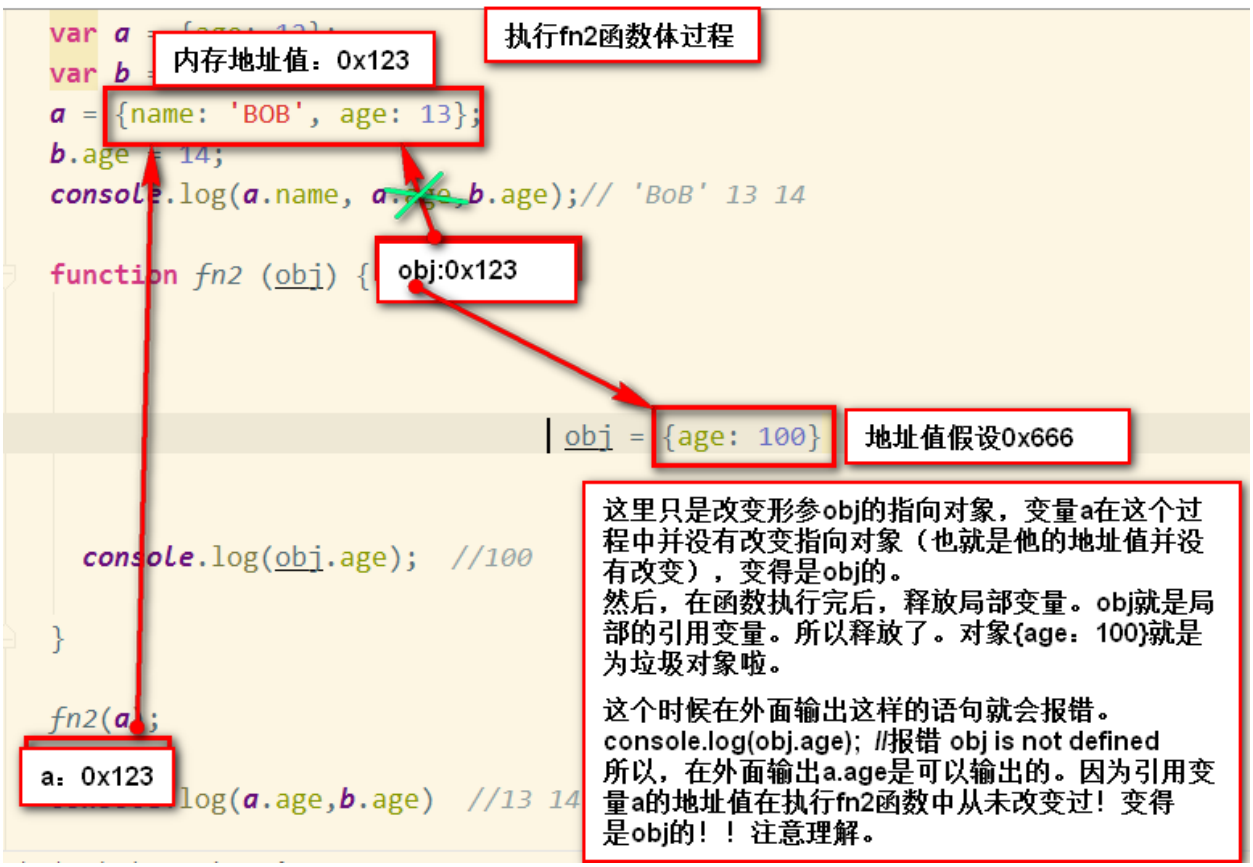
- 2个引用变量指向同一个对象, 通过一个变量修改对象内部数据, 另一个变量看到的是修改之后的数据

```
  var obj1 = {name: 'Tom'}
  var obj2 = obj1
  obj2.name = 'Git'

  console.log(obj1.name)  // 'Git'
```

```
  function fn (obj) {
    obj.name = 'A'
  }
  fn(obj1)
  console.log(obj2.name) //A

//理解：
//执行var obj2 = obj1  obj1是一个变量，而非对象。故把obj1的内容拷贝给obj2，只是刚好这个内容是一个对象的地址值。这个时候，obj1  obj2 这两个引用变量指向同一个对象{name: 'Tom'}。
//通过其中一个变量obj2 修改对象内部的数据。 obj2.name = 'Git'
//那么这个时候，另外一个对象看到的是修改后的结果。也就是说在堆内存中的这个对象name属性值就是被覆盖掉了，成为了'Git'.又因为两个变量obj1  obj2 指向同一个对象，所以输出的值都是最新修改的值。

//当然，后面的对fn(obj1)，也是一样的操作。语句fn(obj1) 中的实参obj1传递给函数的形参obj ，那么obj.name = 'A'赋完实参后就是 obj1.name = 'A'，这个一个赋值覆盖操作。同样因为因为两个变量obj1 obj2 指向同一个对象，故输出是修改后最新的值。
```

- 2个引用变量指向同一个对象, 让其中一个引用变量指向另一个对象, 另一引用变量依然指向前一个对象。

```
  var a = {age: 12};
  var b = a;
  a = {name: 'BOB', age: 13};
  b.age = 14;
  console.log(a.name, a.age,b.age);// 'BoB' 13 14

  function fn2 (obj) {
    obj = {age: 100}
    console.log(obj.age);  //100
  }
  fn2(a);   //函数执行完后会释放其局部变量

  console.log(a.age,b.age)  //13 14

  console.log(obj.age);  //报错 obj is not defined

  //理解：
  //一开始两个引用变量a b 都指向同一个对象，而后执行a = {name: 'BOB', age: 13};语句，就是让a指向另一个对象 {name: 'BOB', age: 13}，a中的内容的地址值变化了。而b还是指向之前a的那个对象{age: 12}。

 //fn2(a)分析见下图图文【初学者理解不深！】
```



问题三：在js调用函数时传递变量参数时，是值传递还是引用传递？

- 理解1: 都是值(数据类型的值/对象的地址值)传递
- 理解2: 可能是值传递, 也可能是引用传递(这个时候引用传递就是指把引用变量中的内容拷贝给形参，这个内容刚好是地址值)

```
var a = 3
function fn (a) {
  a = a +1
}
fn(a)
console.log(a)   //3
//理解：fn(a)中的a是一个实参。function fn (a)中的a是一个形参。 var a = 3中的a是一个变量，其内存中存储的内容是基本数据类型3.实参与形参的传递是把3传递（拷贝）给形参中的a的内存中的内容。传递的不是a!而是3.

function fn2 (obj) {
  console.log(obj.name)   //'Tom'
}
var obj = {name: 'Tom'}
fn2(obj)
//理解：实参形参obj。fn2(obj)中的实参obj（引用变量）把其内容（刚好是地址值）传递给形参中的内容。而不是指把{name: 'Tom'}整个对象赋值给形参obj。是把地址值拷贝给形参obj。也就是实参obj 形参obj这两个引用变量的内容一样（地址值一样）。
```

问题四：JS引擎如何管理内存?

1. 内存生命周期
   - 分配小内存空间, 得到它的使用权
   - 存储数据, 可以反复进行操作
   - 释放小内存空间
2. 释放内存
   - 局部变量: 函数执行完自动释放（全局变量不会释放内存空间）
   - 对象: 成为垃圾对象==>垃圾回收器回收（短暂间隔）

```
var a = 3
var obj = {}
//这个时候有三块小的内存空间：第一块 var a = 3  第二块 var obj  第三块 {} 在堆内存中
obj = undefined
//不管obj = null/undefined  这个时候内存空间还有两块。{}有垃圾回收器回收。

function fn () {
  var b = {}
  //局部变量  整个生命周期是在函数开始执行到结束。
}

fn() // 局部变量b是自动释放，b所指向的对象是在后面的某个时刻由垃圾回收器回收
```

1.3 对象的理解和使用

这一节主要是对对象的基本理解和使用作出阐述，一些基本的问题笔者会简单的罗列出来。具体的深入问题在Part 2中深入探讨。

- **什么是对象?**
  - 变量可以存数据，对象也可以存数据，那么他与变量功能就有差异。
  - 多个数据(属性)的集合
  - 用来保存多个数据(属性)的容器
  - 一个对象就是描述我们生活中的一个事物。
- **为什么要用对象?**
  - 统一管理多个数据。如果不这么做，那么就要引入很多的变量。

    比如我现在要建立一个对象Person,里面有name，age，gender等等，我可以用一个对象去
- **对象的分类？**
  - 内建对象---在任何ES实现中都可以使用（不需new，直接引用）

    Math/String/Function/Number/Data
  - 宿主对象---由JS运行环境提供对象（浏览器提供）

    所有的BOM和DOM对象都是宿主对象。

    console.log ( ) ; document.write ( ) ;
  - 自定义对象---由开发人员创建
- **对象的组成？**
  - 属性: 属性名(字符串)和属性值(任意)组成-----描述对象的状态数据
  - 方法: 一种特别的属性(属性值是函数)-----描述对象的行为数据
  - 它们之间的联系就是方法是特殊的属性。

在了解完对象之后，我们知道每个对象会去封装一些数据，用这个对象去映射某个事物。那么这些数据就是由属性来组成的，现在我们看看属性的一些相关知识。

- **属性组成？**
  - 属性名 : 字符串(标识)，本质上是字符串。本质上属性名是加引号的，也就是字符串。但一般实际操作都不加。
  - 属性值 : 任意类型

```
//属性名本质上是字符串类型
var obj={
    'name'='猪八戒';
    'gender'='男';
}
//一般我们会这么写，实际操作一般可以不加。但特殊情况要加。见下。
var obj={
    name='猪八戒';
    gender='男';
}
```

```
//面试题：对象的属性名问题
//考点:对象的属性名问题。
  var a = {},
      b = {key: 'b'},
      c = {key: 'c'};
  a[b] = 123;  // a["[object Object]"] = 123
  a[c] = 456;  // a["[object Object]"] =456
  console.log(a[b]); //456  求a["[object Object]"]=？

  //属性名本质上是字符串。ES6之前对象的属性名只能是字符串，不能是其他类型的数据！
  //如果你传入的是其他类型的数据作为属性名，则会把其他类型的数据转换成字符串,再做属性名.
  //若是对象，那么就调用toString（）
  //ES6 属性名可以是Symbol类型.

//我们看一段另外的code：
  var a = {
    "0" : "A",
    "1" : "B",
    length : 2
};
for(var i = 0; i < a.length; i++){
    console.log(a[i]);
}
//会输出A B

var a = {
  0 : "A",
  1 : "B",
  length : 2
};
for(var i = 0; i < a.length; i++){
  console.log(a[i]);
}
//也会输出A B

//注意看下面的code，深入理解：
var a = {};
a[[10,20]] = 2000; //首先把握好a是对象，a[]就是使用对象读其属性的语法，而不是数组。把a[]中[10,20]本质上是字符串，所以要转啊。[10,20]转字符串就是对象转字符串，调用toString（），变成“10,20”。这个转的字符串就是属性名。
console.log(a);  // output {10,20: 2000}
```

- 属性的分类？
  - 一般：属性值不是function 描述对象的状态
  - 方法：属性值为function的属性 描述对象的行为
- **特别的对象？**
  - 数组: 属性名是0,1,2,3之类的索引（有序）
  - 函数: 可以执行的

- **如何访问对象内部的数据？**
  - .属性名: 编码简单, 有时不能用。
  - ['属性名'] :编码麻烦, 能通用。

```
var p = {
    name: 'Tom',
    age: 12,
    setName: function (name) {
      this.name = name
    },
    setAge: function (age) {
      this.age = age
    }
  };

  p.setName('Bob')  //用.属性名的方式
  p['setAge'](23)   //用['属性名']语法
  console.log(p.name, p['age'])  //Bob 23
```

- **什么时候必须使用['属性名']的方式?**
  1. 属性名包含特殊字符: - 或 空格
  2. 属性名不确定时。

```
var p = {};
  //1. 给p对象添加一个属性: content type: text/json
  // p.content-type = 'text/json' //不能用
  p['content-type'] = 'text/json'
  console.log(p['content-type'])

  //2. 属性名不确定,用变量去存储这个值。
  var propName = 'myAge'
  var value = 18
  // p.propName = value //不能用
  p[propName] = value   //propName代表着的就是一个变量
  console.log(p[propName])   //18
```

1.4 函数的理解和使用

其实在JavaScript中笔者认为最复杂的数据类型，不是对象，而是函数。为什么函数是最复杂的数据类型呢？因为函数可以是对象，它本身就会有对象的复杂度。函数又可以执行，它有很多的执行调用的方式，这也决定了this是谁的问题。所以，这一小节我们就简单来说说函数的基本知识。在Part3会去更深入去介绍JS中的函数。

- **什么是函数?**
  - 用来实现特定功能的, n条语句的封装体，在需要的时执行功能函数。
  - 只有函数类型的数据是可以执行的, 其它的都不可以
- **为什么要用函数?**

- 提高复用性（封装代码）
- 便于阅读交流

```
function showInfo (age) {
    if(age<18) {
      console.log('未成年，再等等!')
    } else if(age>60) {
      console.log('算了吧!')
    } else {
      console.log('刚好!')
    }
  }
  //如果不用函数做，也可以，但要把中间的代码书写很多遍。
  //而函数就是抽象出共同的东西，把这些执行过程封装起来，给大家一起用。
  showInfo(17)  //未成年，再等等!
  showInfo(20)  //刚好!
  showInfo(65)  //算了吧!
```

- **如何定义函数？**
  - 函数声明
  - 表达式
  - 创建函数对象 var fun = new Function()； 一般不使用

```
function fn1 () { //函数声明
    console.log('fn1()')
  }
  var fn2 = function () { //表达式
    console.log('fn2()')
  }
  fn1();
  fn2();
```

- **如何调用(执行)函数?**
  - test(): 直接调用
  - obj.test(): 通过对象去调用
  - new test(): new调用
  - test.call/apply(obj): 临时让test成为obj对象的方法进行调用

```
var obj = {}  //一个对象
  function test2 () {  //一个函数
    this.xxx = 'lvya'
  }
  // obj.test2()  不能直接，根本obj对象中就没有这样的函数（方法）
  test2.call(obj) // 相当于obj.test2() ， 可以让一个函数成为指定任意对象的方法进行调用
  console.log(obj.xxx)  //lvya

  //这个借调是JS有的，其他语言做不到。借调就是假设一个对象中没有一个方法，那么就可以让这个方法成为想要调用这个方法的对象去使用的方式。也就是一个函数可以成为指定任意对象的方法进行调用 。
```

- **函数也是对象**
  - instanceof Object===true
  - 函数有属性: prototype
  - 函数有方法: call()/apply()
  - 可以添加新的属性/方法
- **函数的3种不同角色**
  - 一般函数 : 直接调用
  - 构造函数 : 通过new调用
  - 对象 : 通过.调用内部的属性/方法
- **this是什么?**
  - 任何函数本质上都是通过某个对象来调用的,如果没有直接指定就是window。
  - 所有函数内部都有一个变量this
  - 它的值是调用函数的当前对象
  - <具体this总结见2.1部分>
- **如何确定this的值?**
  - test(): window
  - p.test(): p
  - new test(): 新创建的对象
  - p.call(obj): obj
  - 回调函数: 看背后是通过谁来调用的: window/其它

```
<script type="text/javascript">
  function Person(color) {
    console.log(this)
    this.color = color;
    this.getColor = function () {
      console.log(this)
      return this.color;
    };
    this.setColor = function (color) {
      console.log(this)
      this.color = color;
    };
  }

  Person("red"); //this是谁? window

  var p = new Person("yello"); //this是谁? p ( Person )

  p.getColor(); //this是谁? p  ( Person )

  var obj = {};
  p.setColor.call(obj, "black"); //this是谁? obj  ( Object )

  var test = p.setColor;
  test(); //this是谁? window
```

```
  function fun1() {
    function fun2() {
      console.log(this);
    }

    fun2();
  }
  fun1();  //this是谁? window
</script>
```

- **匿名函数自调用:**

```
(function(w, obj){
  //实现代码
})(window, obj)
```

  - 专业术语为: IIFE (Immediately Invoked Function Expression) 立即调用函数表达式
  - 作用
    - 隐藏实现 (让外部的全局看不到里面)
    - 不会污染外部(全局)命名空间
    - 用它来编码js模块

```
;(function () { //匿名函数自调用
    var a = 1
    function test () {
      console.log(++a)
    }
    window.$ = function () { // 向外暴露一个全局函数
      return {
        test: test
      }
    }
  })()

  $().test() //需明白 1. $是一个函数 2. $执行后返回的是一个对象 3.然后对象.方法 ( ) 执行函数。
```

- **回调函数的理解**
  - 什么函数才是回调函数?
    - 你定义的
    - 你没有调用
    - 但它最终执行了(在一定条件下或某个时刻)
  - 常用的回调函数
    - dom事件回调函数====>this指的是发生事件的dom元素
    - 定时器回调函数====>this指的是window
    - ajax请求回调函数(后面讲解)
    - 生命周期回调函数(后面讲解)

```
<script type="text/javascript">
  document.getElementById('btn').onclick = function () { // dom事件回调函数
    alert(this.innerHTML)   //this就是指dom元素（btn）
  }

  //定时器
    // 超时(延迟)定时器  setTimeout()
    // 循环定时器 setInterval()
  setTimeout(function () { // 定时器回调函数
    alert('到点了'+this)     //this就是指的是window
  }, 2000);
  </script>
```

- **函数中的arguments**

  在调用函数时，浏览器每次都会传递两个隐含的参数：

  1. 函数的上下文对象 this
  2. 封装实参的对象 arguments（类数组对象）。这里的实参是重点，就是执行函数时实际传入的参数的集合。

```
function foo() {
   console.log(arguments);  //Arguments(3)返回一个带实参数据的类数组
   console.log(arguments.length);  //3  类数组的长度
   console.log(arguments[0]);   //ya LV  可以不传形参，可以访问到实参
   console.log(arguments.callee);  // ƒ foo() {...} 返回对应当前正在执行函数的对象
}
foo('ya LV',18,'male');
```

  - arguments一些妙用

    1.利用arguments实现方法的重载

    a.借用arguments.length属性来实现

```
function add() {
    var len = arguments.length,
        sum = 0;
    for(;len--;){
        sum += arguments[len];
    }
    return sum;
}

console.log( add(1,2,3) );   //6
console.log( add(1,3) );     //4
console.log( add(1,2,3,5,6,2,7) );   //26
```

    b.借用prototype属性来实现：

```
function add() {
  return Array.prototype.reduce.call(arguments, function(n1, n2) {
    return n1 + n2;
  });
};
add(1,2,3,6,8);   //20
//三个常用的数组的高阶函数：map（映射）filter（过滤）reduce（归纳）
//可以参见ES6函数新增特性之箭头函数进一步优化
```

2.利用arguments.callee实现递归

先来看看之前我们是怎么实现递归的，这是一个计算阶乘的函数:

```
function factorial(num) {
    if(num<=1) {
        return 1;
    }else {
        return num * factorial(num-1);
    }
}
```

但是当这个函数变成了一个匿名函数时，我们就可以利用callee来递归这个函数。

```
function factorial(num) {
    if(num<=1) {
        return 1;  //如果没有这个判断，就会内存溢出
    }else {
        return num * arguments.callee(num-1);
    }
}
 console.log(factorial(5));  //120
```

1.5 补充

1.5.1 分号问题

- js一条语句的后面可以不加分号,类似"可以加分号但是大家都不加" 的语言就有：Go, Scala, Ruby, Python, Swift, Groovy...
- 是否加分号是编码风格问题, 没有应该不应该，只有你自己喜欢不喜欢
- 在下面2种情况下不加分号会有问题
    - 小括号开头的前一条语句

```
  var a = 3
  ;(function () {
  })()

//如果不加分号就会这么错误解析：
//  var a = 3(function () {
//   })();
```

- 中方括号开头的前一条语句

```
  var b = 4
  ;[1, 3].forEach(function () {
  })

//  如果不加分号就会这么错误解析：
//  var b = 4[3].forEach(function () {
//   })
```

- 解决办法: 在行首加分号
- 强有力的例子: vue.js库。Vue.js 的代码全部不带分号。
- 有一个工具全自动帮你批量添加或者删除分号：https://github.com/yyx990803/semi

1.5.2 位运算符和移位在JS中的操作

像二进制，八进制，十进制，十六进制这些概念在JavaScript中很少被体现出来，可是我觉得这个计算机人的素养，所以我觉得有必要再去搞懂。另外一个就是原码反码补码的概念，在计算机硬件电路中有加法器，所以的运算都会转为加法运算，减法就是用加法实现。所以才引出原码反码补码的概念去解决这一问题。

那么笔者现在着重讲一下位运算符操作和移位操作。js中位运算符有四种：按位取反（~）、按位与（&）、按位或（|）、按位异或（^）。移位操作有四种：带符号向右移动（>>）、无符号向右移动（>>>）、带符号向左移动（<<）、无符号向左移动（<<<）.

示例1：如何快速判断一个数是不是奇数？

那么，取余是你先想到的，那么还有其他方法吗？就是用位运算符去解答。先思考奇数3（二进制：11），偶数4（二进制：100），可知偶数的最低位为0，奇数的最低位为1，那么我们只要通过某种方法得到一个数的二进制的最低位，判断它是不是为1，是1那这个数就是奇数。

现在的问题，转换为怎么得到一个二进制的最低位呢？那就是用按位与1（~1）去做。假设一个数的二进制为1111 1111 那么只要按位与1（1的二进制为0000 0001）是不是前面一排“与0”都变成0了，只剩最低位了，这样就得到了二进制的最低位。

```
var num = 57 ;
    if(num & 1){
        console.log(num + "是奇数");  //57是奇数
    }else{
        console.log(num + "是偶数");
    }
```

示例2：怎么交换两个number类型的变量值？

那么，新增一个变量来存储这种方式是你先想到的，那么另外一种就是通过按位异或操作去交换变量。

异或就是不同的为true（1），相同的为false（0）。

10^10=0 因为1010 ^ 1010 = 0000

11^0=11 因为1011 ^ 0000=1011

所以得到两个结论：

第一，两个相同的number数异或为0；第二，任何number数与0异或是其本身。

```
var a = 10;
    var b = 20;
    a = a ^ b;  //a=10 ^20
    b = a ^ b;  //b=10 ^20^20 =10 ^(20^20)=10^0=10
    a = a ^ b;  //a=10 ^20^ 10 =(10^10)^20=0^20=20
    console.log(a, b);  //20  10  -交换变量成功-

//但这种方法只适用于number数据类型。
```

示例3：如何计算出一个数字某个二进制位？

在回答这个问题前，我们先总结出一些结论供我们使用。移位都是当做32位来移动的，但我们这里就简单从操作，用8位来模拟。

先看带符号向右移位：

10 >> 1 翻译题目：10带符号向右移动一位是几？

0000 1010 >> 1

0000 0101 这个结果就是移位后的结果。我们可以知道0101就是十进制的5.

带符号向右移动就是整体向右移动一位，高位用符号位去补。正数用0补，负数用1补。

我们可以看出结论，带符号向右移动其实就是往右移动一位，相当于除以2.

现在再来看看带符号向左移位：

10 << 2 翻译题目： 10带符号向左移动2位是几？

0000 1010 << 2

0010 10**00** 低位用0补。这个 0010 1000 就是数就是40.

我们可以看出结论，带符号向左移动其实就是往左移动一位，相当于乘以2.移动2位，就是乘4.

现在回归题目，假设我要知道10（1010）的倒数第三位的0这个进制位。

首先往右移动两位变成0010 ， 然后进行 '&1' 操作 ， 0010 & 0001 =0000 =0 ,这个0就是10的二进制位的倒数第三位。所以是通过：（10 >> 2 & 1）的方式得到10的倒数第三位的进制位。

示例4：如何计算出2的6次方最快算法？

2B程序猿会用2 * 2 * 2 * 2 * 2 * 2 的方法吧。码农可能会用for循环去做或者用Math.pow(2,6)去写。

但是这些都不是最快的。我们来看看高级工程师会怎么写，哈哈。我们刚刚得到过2个结论，其中一个就是带符号向左移位其实就是往左移动一位，相当于乘以2。 移动2位，就是乘4。"左乘右除"。那么现在我是不是可以对 1 移动6位 不就可以了吗？所以就一行代码：1 << 6 。

由汇编知识我们知道，移位是最底层的计算。可以完全用加法器实现。而Math.pow(2,6)其实会有很多的汇编指令才可以实现这一条代码。但1 << 6 只需要一条，所以，性能是很好的。

1.5.3 内存溢出与内存泄露

1. 内存溢出
    - 一种程序运行出现的错误
    - 当程序运行需要的内存超过了剩余的内存时, 就出抛出内存溢出的错误。

```
// 1. 内存溢出
  var obj = {}
  for (var i = 0; i < 10000; i++) {
    obj[i] = new Array(10000000)
    console.log('-----')
  }
//直接崩掉了，需要的内存大于目前空闲的内存，直接报错误：内存不足。
//就如一个水杯，水倒满了就溢出，这就是内存溢出。
```

2. 内存泄露
    - 占用的内存没有及时释放，这时程序还是可以正常运行的
    - 内存泄露积累多了就容易导致内存溢出
    - 常见的内存泄露:
        - 意外的全局变量

```
 // 在意外的全局变量--在ES5的严格模式下就会报错。
  function fn() {
    a = new Array(10000000)
    console.log(a)
  }
  fn()
//a就是意外的全局变量，一直会占着内存，关键它还是指向一个数组非常大的对象。这块内存就一直占着。
```

- 没有及时清理的计时器或回调函数

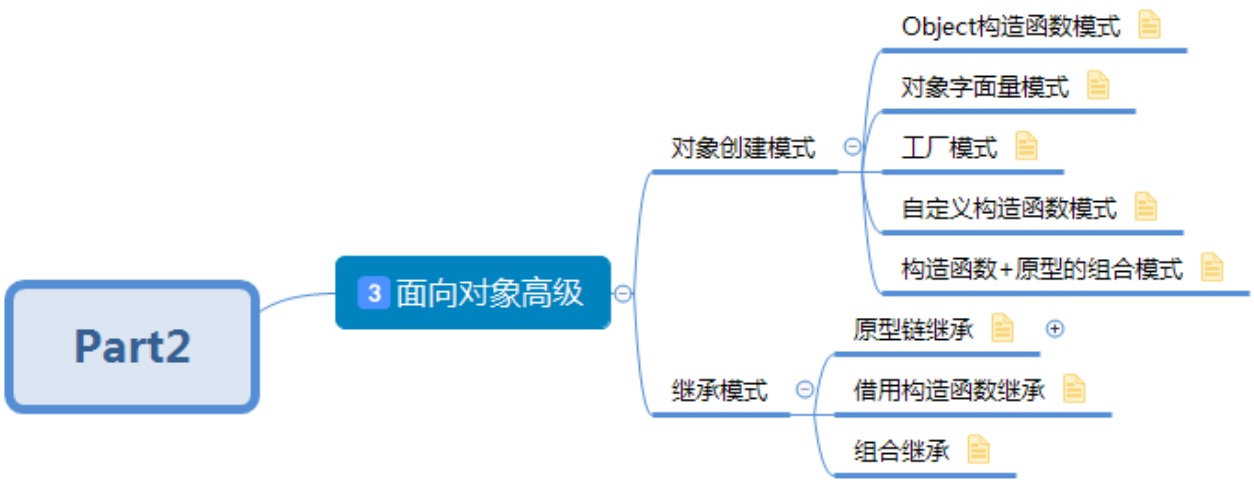
```
// 没有及时清理的计时器或回调函数
  var intervalId = setInterval(function () { //启动循环定时器后不清理
    console.log('----')
  }, 1000)

  // clearInterval(intervalId)
```

- 闭包

```
 // 闭包
function fn1() {
  var a = 4
  function fn2() {
    console.log(++a)
  }
  return fn2
}
var f = fn1()
f()
//f指向的fn2函数对象一直都在，设f指向空对象，进而让fn3成为垃圾对象，进而去回收闭包。
// f = null
```

Part2：对象高级

> Part2: 思维导图



2.1 对象的创建模式

- 方式一: Object构造函数模式

    - 套路: 先创建空Object对象, 再动态添加属性/方法
    - 适用场景: 起始时不确定对象内部数据
    - 问题: 语句太多

```
// 先创建空Object对象
  var p = new Object()
  p = {} //此时内部数据是不确定的
  // 再动态添加属性/方法
  p.name = 'Tom'
  p.age = 12
  p.setName = function (name) {
    this.name = name
  }

  //测试
  console.log(p.name, p.age)  //Tom 12
  p.setName('Bob')
  console.log(p.name, p.age)  //Bob 12
```

- 方式二: 对象字面量模式
    - 套路: 使用{}创建对象, 同时指定属性/方法
    - 适用场景: 起始时对象内部数据是确定的
    - 问题: 如果创建多个对象, 有重复代码

```
  var p = {
    name: 'Tom',
    age: 12,
    setName: function (name) {
      this.name = name
    }
  }

  //测试
  console.log(p.name, p.age)  //Tom 12
  p.setName('JACK')
  console.log(p.name, p.age)  //JACK 12

  var p2 = {  //如果创建多个对象代码很重复
    name: 'Bob',
    age: 13,
    setName: function (name) {
      this.name = name
    }
  }
```

- 方式三: 工厂模式（相对使用很少）
    - 套路: 通过工厂函数动态创建对象并返回
    - 适用场景: 需要创建多个对象
    - 问题: 对象没有一个具体的类型, 都是Object类型

```
function createPerson(name, age) { //返回一个对象的函数一般称为===>工厂函数
  var obj = {
    name: name,
    age: age,
    setName: function (name) {
      this.name = name
    }
  }
  return obj
}

// 创建2个人
var p1 = createPerson('Tom', 12)
var p2 = createPerson('Bob', 13)
// p1和p2是Object类型

function createStudent(name, price) {
  var obj = {
    name: name,
    price: price
  }
  return obj
}
var s = createStudent('张三', 12000)
// s也是Object
```

- 方式四: 自定义构造函数模式
  - 套路: 自定义构造函数, 通过new创建对象
  - 适用场景: 需要创建多个类型确定的对象
  - 问题: 每个对象都有相同的数据, 浪费内存。主要是指方法数据的重复，导致浪费。

```
//定义类型，Person类型
function Person(name, age) {
  this.name = name
  this.age = age
  this.setName = function (name) {
    this.name = name
  }
}
var p1 = new Person('Tom', 12)
p1.setName('Jack')
console.log(p1.name, p1.age) //Jack 12
console.log(p1 instanceof Person)  //true

function Student (name, price) {
  this.name = name
  this.price = price
}
var s = new Student('Bob', 13000)
console.log(s instanceof Student) //true
```

```
var p2 = new Person('JACK', 23)
console.log(p1, p2) //都返回Person类型对象
```

- 方式六: 构造函数+原型的组合模式
  - 套路: 自定义构造函数, 属性在函数中初始化, 方法添加到原型上
  - 适用场景: 需要创建多个类型确定的对象

```
function Person(name, age) { //在构造函数中只初始化属性
  this.name = name
  this.age = age
}
//方法放在原型对象里，就不会重复了。减少内存压力。
Person.prototype.setName = function (name) {
  this.name = name
}

var p1 = new Person('Tom', 23)
var p2 = new Person('Jack', 24)
console.log(p1, p2) //都返回一个Person类型对象
```

2.2 继承模式

- **方式1: 原型链继承**

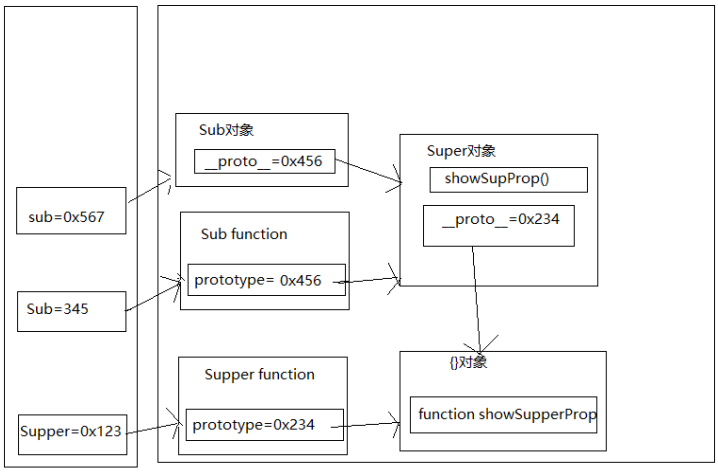
```
1. 套路
1. 定义父类型构造函数
2. 给父类型的原型添加方法
3. 定义子类型的构造函数
4. 创建父类型的对象赋值给子类型的原型
5. 将子类型原型的构造属性设置为子类型
6. 给子类型原型添加方法
7. 创建子类型的对象：可以调用父类型的方法
2. 关键
1. 子类型的原型为父类型的一个实例对象
```

```
//步骤思想
function Parent(){}
Parent.prototype.test = function(){};
function Child(){}
Child.prototype = new Parent(); // 子类型的原型指向父类型实例
Child.prototype.constructor = Child
var child = new Child(); //有test()
```

```
//案例分析
<script type="text/javascript">

  //父类型
```

```
  function Supper() {
    this.supProp = 'Supper property'
  }
  Supper.prototype.showSupperProp = function () {
    console.log(this.supProp)
  }

  //子类型
  function Sub() {
    this.subProp = 'Sub property'
  }

  // 子类型的原型为父类型的一个实例对象
  Sub.prototype = new Supper()
  // 让子类型的原型的constructor指向子类型（构造函数）
  Sub.prototype.constructor = Sub
  Sub.prototype.showSubProp = function () {
    console.log(this.subProp)
  }

  var sub = new Sub()
  sub.showSupperProp()  //Supper property
  sub.showSubProp()  //Sub property

  console.log(sub)  // 返回Sub类型对象
</script>
```

```
function Supper() { //父类型
  this.superProp = 'The super prop'
}
//原型的数据所有的实例对象都可见
Supper.prototype.showSupperProp = function () {
  console.log(this.superProp)
}

function Sub() { //子类型
  this.subProp = 'The sub prop'
}

Sub.prototype = new Supper()
Sub.prototype.showSubProp = function () {
  console.log(this.subProp)
}

var sub = new Sub()
sub.showSubProp()
sub.showSupperProp()
```



- **方式2: 借用构造函数继承(假的)**
  1. 套路:
     1. 定义父类型构造函数
     2. 定义子类型构造函数
     3. 在子类型构造函数中调用父类型构造
  2. 关键:
     1. 在子类型构造函数中通用call()调用父类型构造函数

```
<script type="text/javascript">
```

```
  function Person(name, age) {
    this.name = name
    this.age = age
  }
  function Student(name, age, price) {
    //为了获得父类型的Person的属性继承
    Person.call(this, name, age)  // 借调。相当于: this.Person(name, age)
    /*this.name = name
    this.age = age*/
    this.price = price
  }

  var s = new Student('Tom', 20, 14000)
  console.log(s.name, s.age, s.price) //Tom 20 14000

</script>
```

- **方式3: 原型链+借用构造函数的组合继承（一般都这么用）**

  1. 利用原型链实现对父类型对象的方法继承（继承方法）
  2. 利用super()借用父类型构建函数初始化相同属性（继承属性）

```
<script type="text/javascript">
  //类似于父函数
  function Person(name, age) {
    this.name = name
    this.age = age
  }
  Person.prototype.setName = function (name) {
    this.name = name
  }
 //类似于子函数
  function Student(name, age, price) {
    Person.call(this, name, age)  // 为了得到属性(属性的继承)
    this.price = price
  }
  Student.prototype = new Person() // 为了能看到父类型的方法（方法的继承--通过原型链）
  Student.prototype.constructor = Student //修正constructor属性
  Student.prototype.setPrice = function (price) {
    this.price = price
  }

  var s = new Student('Tom', 24, 15000)
  s.setName('Bob');
  s.setPrice(16000);
  console.log(s.name, s.age, s.price) //Bob 24 16000

</script>
```

- **ES6的class的继承（见ES6的讲解）**

2.3 new对象

new一个对象背后做了些什么?

- 创建一个空对象
- 给对象设置**proto**, 值为构造函数对象的prototype属性值 this.**proto** = Fn.prototype
- 执行构造函数体(给对象添加属性/方法)

要明白这个过程，很重要。

Part3 函数高级

3.1 this的使用总结

this是在函数执行的过程中自动创建的一个指向一个对象的内部指针。确切的说，this并不是一个对象，而是指向一个已经存在的对象的指针，也可以认为是this就是存储了某个对象的地址。

this的指向不是固定的，会根据调用的不同，而指向不同的地方。

> 此节的讲解思路是这样的：
>
> 1：首先在this总结之前先需要明白几个基础知识。比如，全局作用域中的变量和函数定义、构造函数与非构造函数的区别
>
> 2：然后从三大方向总结this。注意，这三个方向就是都是由我们调的情况。当然还有不是由我们调用的，那就是回调函数中的this，请参见1.4节回调函数有说。这样this的整个知识点的框架就有明白吧。
>
> 分两个层次，第一个层次就是回调函数中的this，有四种回调函数。第二个层次就是由我们调的函数中的this，分为以下三个方向。整个知识框架体系必须有一个“高度的认识”，我感觉这点很重要。掌握好大局知识体系，再慢慢的去塞你学的知识。
>
> 第一个方向：全局作用域中的this指向（也就是函数外的this）
>
> 第二个方向：函数中的this（包括构造函数和非构造函数的this指向）
>
> 第三个方向：JavaScript独特的方式，可以通过call / apply 改变this的指向
>
> 3：由this知识点，拓展出一个call、apply、bind的使用，放在这里总结。

3.1.1 搞懂this指向的基础知识

- **对在全局作用域中定义的变量和函数的进一步认识**

  永远记住：只要是在全局作用域声明的任何变量和函数默认都是作为window对象的属性而存在的.

  理解完以上的这句话，我们在来说明一下其中的区别，这是很多人没有关注过的。

```
  console.log(window.a);  //undefined
  console.log(a);   //报错！a is not defined
//也就是在未对变量（a）进行声明时，就会出现以上结果。
//首先明确一点，就是全局变量a 也是window的属性吧。没问题。所以，我们从这里就可以发现，什么时候undefined，什么时候报错呢？—那就是如果是访问一个对象的属性时，它没有声明赋值，那就是undefined；如果访问一个变量，它没有声明赋值，那就是报错。
```

好，现在回头过来，我们看全局作用域变量和函数的认识。

> 看下面的代码：

```
<script type="text/javascript">
    var num = 10;       //全局作用域声明的变量
    function sum () {  //全局作用域声明的函数
        alert("我是一个函数");
    }
    alert(window.num);  // 10
    window.sum();       // 我是一个函数
    // 在调用的时候window对象是可以省略的。
 </script>
```

- **构造函数和非构造函数的澄清**

在JavaScript中构造函数和非构造函数没有本质的区别。**唯一的区别只是调用方式的区别。**

- 使用new 就是构造函数
- 直接调用就是非构造函数

> 看下面的代码：

```
<script type="text/javascript">
    function Person () {
        this.age = 20;
        this.sex = "男";
    }
    //作为构造函数调用,创建一个对象。 这个时候其实是给p添加了两个属性
    var p = new Person();
    alert(p.age + "  " + p.sex);

    //作为普通函数传递，其实是给 window对象添加了两个属性
    //任何函数本质上都是通过某个对象来调用的,如果没有直接指定就是window，也就是Window可省略
    Person();
    alert(window.age + " " + window.sex);
</script>
```

3.1.2 第一个方向：全局作用域中的this指向

> 全局作用域中使用this，也就是说不在任何的函数内部使用this，那么这个时候this就是指的 **window**
>
> 看下面的代码：

```
<script type="text/javascript">
    //全局作用域中的this
    //向this对象指代的对象中添加一个属性 num， 并让属性的值为100
    this.num = 100;
    // 因为this就是window，所以这时是在修改属性num的值为200
    window.num = 200;
    alert(this === window);  // true this就是指向的window对象，所以是恒等
    alert(this.num);    //200
    alert(window.num);  //200
</script>
```

3.1.3 第二个方向：函数中的this

函数中this又可分为构造函数和非构造函数的this两个概念去理解。

- **非构造函数中的this指向**

  > 非构造函数中this指向的就是 **调用这个方法的那个对象**

```
<script type="text/javascript">
    function test() {
        alert(this == window);
        this.age = 20;
    }
    test();  //其实是 window.test();  所以这个时候test中的this指向window
</script>
```

> 看下面的代码：

```
<script type="text/javascript">
    var p = {
        age : 20,
        sex : "男",
        sayAge: function (argument) {
            alert(this.age);
        }
    }
    p.sayAge(); //调用对象p的方法sayAge()  所以这个时候this指的是 p 这个对象
</script>
```

> 再看下面的代码：

```
<script type="text/javascript">
    var p = {
        age : 20,
        sex : "男",
        sayAge: function (argument) {
            alert(this.age);
            alert(this === p);  //true
        }
    }
    var again = p.sayAge;   //声明一个变量(方法)，把p的方法复制给新的变量
  //调用新的方法： 其实是window.again().  所以 方法中的this指代的是window对象，这个时候age属性是
undefined
  // this和p也是不相等的。
    again();
</script>
```

> **综上：this的指代和代码出现的位置无关，只和调用这个方法的对象有关。**

- **构造方法中的this指向**

> 构造方法中的this指代的要未来要创建的那个对象。

```
<script type="text/javascript">
    function Person () {
        this.age = 20;
        return this;  //作为构造函数的时候，这个行代码默认会添加
    }
    var p1 = new Person();  //这个时候 Person中的this就是指的p1
    var p2 = new Person();  //这是时候 Person中的this就是知道p2
</script>
```

> 多了解一点：其实用new调用构造函数的时候，构造函数内部其实有个默认的 return this; 这就是为什么this指代那个要创建的对象了。

3.1.4 第三个方向： 改变this的指向（显示绑定）

> 在JavaScript中，允许更改this的指向。
>
> 通过call方法或apply方法
>
> 一个函数可以成为指定任意对象的方法进行调用 。这个函数就是函数对象，每个函数对象中都有一个方法call，通过call可以让你指定的对象去调用这个函数。
>
> ECMAScript 规范给所有函数都定义了 call 与 apply 两个方法。

```
<script type="text/javascript">
    var age = 20;
    function showPropertyValue (propertyName) {
        alert(this[propertyName]);
    }
    //使用call的时候，第一个参数表示showPropertyValue中的this的执行，后面的参数为向这个函数传的值。
    //注意一点：如果第一个参数是null，则this仍然是默认的指向。
    showPropertyValue.call(null, "age");
    showPropertyValue.call(this, "age");
    showPropertyValue.call({age:50}, "age")
</script>
```

3.1.5 call / apply / bind 的详解

在this的指向中有第三个方向就是通过call/apply去改变this的指向，这个JavaScript中一个独特的使用形式，其他语言并没有。那么，我们就在这里顺带讲一下call 、apply 以及 bind 的用法。

> 本小节将从三个方面讲解：
>
> 1：apply和call的区别
>
> 2：apply和call的用法
>
> 3：call和bind的区别

- **apply 和 call 的区别**

  ECMAScript 规范给所有函数都定义了 call 与 apply 两个方法，它们的应用非常广泛，它们的作用也是一模一样，只是传参的形式有区别而已。

简单来说，假设有一个函数A，我们调用函数A会直接去A（ ），那么如果是A（ ）这样直接调用的话，函数体A里面的this就是window了。而我们可以通过call（或apply）去调用，比如：A.call（ ）.这样子调用就可以指定A中的this到底是哪个对象。用call来做比对，里面有两个参数，参数一就是重新指定其中的this是谁，参数2是属性名。而事实上，call与apply也就是参数二的不同。

▹apply（ ）

apply 方法传入两个参数：一个是作为函数上下文的对象，简单来说，重新指定函数中的this是谁。另外一个是作为函数参数所组成的数组，是传入一个数组。

```
 var obj = {
    name : 'ya LV'
}

function func(firstName, lastName){
    console.log(firstName + ' ' + this.name + ' ' + lastName);
}

func.apply(obj, ['A', 'B']);    // A ya LV B
```

可以看到，obj 是作为函数上下文的对象，也就是说函数 func 中 this 指向了 obj 这个对象。本来如果直接调用 func（ ），那么函数体中的this就是指的是window。但是现在有了参数一，就是重新指定this，这个this就是参数一的obj这个对象。参数 A 和 B 是放在数组中传入 func 函数，分别对应 func 参数的列表元素。

▹call（ ）

call 方法第一个参数也是作为函数上下文的对象。与applyl没有任何区别。但是后面传入的是一个参数列表，而不是单个数组。

```
ya LVvar obj = {
    name: 'ya LV'
}

function func(firstName, lastName) {
    console.log(firstName + ' ' + this.name + ' ' + lastName);
}

func.call(obj, 'C', 'D');       // C ya LV D
```

对比 apply 我们可以看到区别，C 和 D 是作为单独的参数传给 func 函数，而不是放到数组中。

对于什么时候该用什么方法，其实不用纠结。如果你的参数本来就存在一个数组中，那自然就用 apply，如果参数比较散乱相互之间没什么关联，就用 call。

补充一个使用 `apply` 的例子。比如求一个数组的最大值？

明确JavaScript中没有返回一个数组中最大值的函数。但是，有一个函数Math.max可以返回任意多个数值类型的

参数中的最大值，Math.max函数入参并不支持数组，只能是将多个参数逐个传入，用逗号分隔。

这个时候如果我们非要传个数组，又可以用Math.max函数。我们自然而然会想到所有函数都定义了call和apply的方法，我们可以配合apply或call来实现，又因为call传参数并不是一个数组。所有我们就选择出Math.max函数加上apply就可以实现我们的目的。

原本只能这样用，并不能直接用数组。

```
let max = Math.max(1, 4, 8, 9, 0)
```

有了 `apply` ，就可以这么调用：

```
let arr = [1, 4, 8, 9, 0];
let max = Math.max.apply(null, arr);
//注：在调用apply的时候第一个参数给了一个null，这个是因为没有对象去调用这个方法，我们只需要用这个方法帮我们运算，得到返回的结果就行，所以就直接传递了一个null过去。
```

- **apply 和 call 的用法**

  apply和call的用法可以分为三个：改变this的指向，借用别的对象的方法，调用函数。

  ▹**1.改变 this 指向**

  ```
  var obj = {
      name: 'ya LV'
  }

  function func() {
      console.log(this.name);
  }

  func.call(obj);       // ya LV
  ```

  这个在前一小节有讲到，所以我们就简单的再来看看。所谓"熟能生巧"，一样东西，一个知识点，每看一次会有不同的体会，可能这次看的过程让你有更深刻的思考，这就是进步。call 方法的第一个参数是作为函数上下文的对象，这里把 obj 作为参数传给了 func，此时函数里的 this 便指向了 obj 对象。此处 func 函数里其实相当于：

  ```
  function func() {
      console.log(obj.name);
  }
  ```

  另外注意下call的一些特别用法，很奇葩的this指向。稍微注意下，有点印象就好。

  ```
  function func() {
      console.log(this);
    }
    func.call();   //window
    func.call(undefined);     //window
    func.call(null);    //window
    func.call(1);  //Number {1}   这种情况会自动转换为包装类Number 就相当于下面一行代码
    func.call(new Number(1));   //Number {1}
  ```

▹**2.借用别的对象的方法**

先看例子

```
var Person1  = function () {
    this.name = 'ya LV';
}

var Person2 = function () {
    this.getname = function () {
        console.log(this.name);
    }
    Person1.call(this);
}
var person = new Person2();
person.getname();       // ya LV
```

从上面我们看到，Person2 实例化出来的对象 person 通过 getname 方法拿到了 Person1 中的 name。因为在 Person2 中，Person1.call(this) 的作用就是使用 Person1 对象代替 this 对象，那么 Person2 就有了 Person1 中的所有属性和方法了，相当于 Person2 继承了 Person1 的属性和方法。不理解的话我们再来慢慢看，我们说A.call (参数一)这样的形式就是重新指定函数A中的this是'参数一'这个对象，那么我们来看看Person2函数体中的Person1.call(this)这条语句，其中这条语句的this是指Person2这个对象。现在就是把Person1函数的this重新指向为Person2，是不是有了Person2.name='ya LV'。

▹**3.调用函数**

apply、call 方法都会使函数立即执行，因此它们也可以用来调用函数。这个我们在这节的一开始就有说，比如A ( ) 和A.call ( ) 都是调用函数。

```
function func() {
    console.log('ya LV');
}
func.call();            // ya LV
```

- **call 和 bind 的区别**

  在 EcmaScript5 中扩展了叫 bind 的方法，在低版本的 IE 中不兼容。它和 call 很相似，接受的参数有两部分，第一个参数是是作为函数上下文的对象，第二部分参数是个列表，可以接受多个参数。 它们之间的区别有以下两点。

  - **1.bind 的返回值是函数**

```
 var name='HELLO'
var obj = {
  name: 'ya LV'
}

function func() {

  console.log(this.name);
```

```
  }

  //将func的代码拷贝一份，并且永远改变其拷贝出来的函数中的this，为bind第一个参数所指向的对象。把这
    份永远改变着this指向的函数返回给func1.
  var func1 = func.bind(obj);
 //bind方法不会立即执行，是返回一个改变上下文this的函数，要对这个函数调用才会执行。
  func1();  //ya LV
 //可以看到，现在这份改变this之后拷贝过来的函数，this的指向永远是bind（）绑定的那个，不管之后去
call    重新指向对象，func1 都不会改变this的指向。永远！可知，bind比call优先级还高。
  func1.call({name:'CALL'});   //ya LV

  //又从func重新拷贝一份永远改变this指向对象为{name:'LI SI'}这个对象的函数，返回给func2.
  var func2 = func.bind({name:'LI SI'});
  func2();   //LI SI

 //注意，这里是拷贝一份func2(而不是func)的代码，而func2之前已经绑定过去永远改变this的指向了，所以
这    里并不去改变！还是会输出原来的最先bind的this指向对象。
  var func3 = func2.bind({name:'ZHANG SAN'});
  func3();   //LI SI

 //上面对func最初的函数进行了多次绑定，绑定后原函数 func 中的 this 并没有被改变，依旧指向全局对象
    window。因为绑定bind的过程是拷贝代码的一个过程，而不是在其自身上修改。window.name = HELLO
  func();   //HELLO
```

bind 方法不会立即执行，而是返回一个改变了上下文 this 后的函数。而原函数 func 中的 this 并没有被改变，依旧指向全局对象 window。

- **2.参数的使用**

```
function func(a, b, c) {
    console.log(a, b, c);
}
var func1 = func.bind(null,'yaLV');

func('A', 'B', 'C');            // A B C
func1('A', 'B', 'C');           // yaLV A B
func1('B', 'C');                // yaLV B C
func.call(null, 'yaLV');       // yaLV undefined undefined
```

call 是把第二个及以后的参数作为 func 方法的实参传进去，而 func1 方法的实参实则是在 bind 中参数的基础上再往后排。也就是说，var func1 = func.bind(null,'yaLV'); bind现有两个参数，第一个是指向，第二个实参是'yaLV'，那么就是先让func中的a='yaLV',然后没排满就是让func1('A', 'B', 'C'); 这个参数依次排，现在b='A'，c='B'，形参已经排完了。也就是输出yaLV A B。

在低版本浏览器没有 bind 方法，我们也可以自己实现一个。

```
if (!Function.prototype.bind) {
        Function.prototype.bind = function () {
            var self = this,                        // 保存原函数
                context = [].shift.call(arguments), // 保存需要绑定的this上下文
                args = [].slice.call(arguments);    // 剩余的参数转为数组
            return function () {                    // 返回一个新函数
                self.apply(context,[].concat.call(args, [].slice.call(arguments)));
            }
        }
    }
```

3.1.6 习题与案例

```
//习题1
<script type="text/javascript">
     var name='window_dqs';
     var obj={
        name:'obj_dqs',
        showName:function(){
            console.log(this.name);
        }};

     function fn(){
        console.log(this);
     }
     function fn2(){
        this.name='fn_dqs';
     }

    //因为obj去调用，this就是obj
     obj.showName();    //obj_dqs
     //因为借调，而此时借调的对象是this，而this在全局作用域上就是指window,所以找window.name
     obj.showName.apply(this);   //window_dqs
     //因为借调的对象是一个函数对象，那么this就是指函数对象，this.name就是函数名
     obj.showName.apply(fn2);    //fn2
</script>
```

```
//习题2：
<script type="text/javascript">
 var name='window_dqs';
     function fn(){
        this.name='fn_dqs';
        this.showName=function(){
            console.log(this.name);
        }
        console.log(this);
     }

     function fn2(){

        this.name='fn_pps';
```

```
        this.showName=function(){
            console.log(this.name);
        }
        console.log(this);
    }

    var p=new fn();
    fn2.apply(p);
    p.showName();

    var obj={};
    fn2.apply(obj);
    obj.showName();
    </script>

    //结果：
    fn{name: "fn_pps"showName: ƒ ()__proto__: Object····}
    fn{name: "fn_pps"showName: ƒ ()__proto__: Object····}
    fn_pps
    Object{name: "fn_pps"showName: ƒ ()__proto__: Object··}
    fn_pps:
```

```
//习题3：
<script type="text/javascript">
var name='window_dqs';
    var obj={
        name:'json_dqs',
        showName:function(){
            console.log(this.name);
            return function(){
                console.log(this.name);
            }
        }
    }
   var p=obj.showName();
   obj.showName()();
   p.call(obj);
</script>

//结果：
json_dqs
json_dqs
window_dqs
json_dqs
```

```
//面试题：
//代码片段一
  var name = "The Window";
  var object = {
    name: "My Object",

    getNameFunc: function (){
```

```
      return function (){
        return this.name;
      };
    }
  };
  console.log(object.getNameFunc());     //f (){return this.name;}
  console.log(object.getNameFunc()());  //The Window
```
、//代码片段一没有闭包。有嵌套，但没有用外部函数的变量或函数。是使用this的。this与调用方式有关。

//理解：看object.getNameFunc()是对象.方法（） 返回的是一个函数，这个函数还未执行。js中this是动态的，所以函数没有执行，并不确定函数里的this是指的是谁？那么现在再对返回的函数加个（），也就是object.getNameFunc()()，调用执行，把最后一个括号和最后一个括号前当做两个部分，前面是函数名，后面一个括号是调用。相当于test（），这个时候this就是window。故这样调用的函数this就是指的window，故window.name=The Window.

```
  //代码片段二
  //对于片段一我们的本意是不是想输出My Object。那么怎么改造，通过that=this去操作，这样子
  var name2 = "The Window";
  var object2 = {
    name2: "My Object",
    getNameFunc: function () {
      var that = this;
      return function () {
        return that.name2;
      };
    }
  };
  console.log(object2.getNameFunc());    //f (){return that.name2;}
  console.log(object2.getNameFunc()()); //My Object
```
//代码片段二是有闭包的，有嵌套函数。内部函数有使用外部函数的变量that。外部和内部函数有执行。

//理解：首先还是看object2.getNameFunc()返回一个函数，注意这个函数中没有this，在调用object2.getNameFunc时，我们有执行一句var that = this;也就是把this给that，这个时候this是谁，是不是"对象.方法（）"这种类型的this问题，所以就是指的那个对象，也就是这个时候this指的是object2。在再次调用object2.getNameFunc()()时就是执行“object2.getNameFunc()返回来的函数”。that.name2=object2.name2;实质上是闭包，使用了外部函数的that变量。

```
  //代码片段三（对片段二的改造）
  var name3 = "The Window";
  var object3 = {
    name3: "My Object",
    getNameFunc: function () {
      return function () {
        return this.name3;
      }.bind(this);
    }
  };
  console.log(object3.getNameFunc());     //f (){return this.name3;}
  console.log(object3.getNameFunc()());   //My Object
```

//理解：与“代码片段二”一样，只是片段二是通过that=this去改变this的值，而片段三是通过bind绑定this的值。

看bind（this）这里的this就是指这条语句object3.getNameFunc()调用的对象object3.所以通过这个手段去把this指向了前面的对象object3.再去调用返回的函数时，那么this.name3=object3.name3。

```
//面试题：
<script>
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log(this.foo);  //bar
        console.log(self.foo);  //bar
        (function() {
          console.log(this.foo);  //undefined  此时的this是window
            console.log(self.foo);  //bar   闭包可以看到外部的局部变量
        }());  //匿名函数自执行，是window上调用这个函数。
    }
};
myObject.func();

//那么如何修改呢？使得在自执行函数中的this.foo就是我们想要的bar呢？提供两种方法：
//case1：用call去指向this是谁
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log(this.foo); // bar
        console.log(self.foo); // bar
        (function() {
            console.log(this.foo);  // bar
            console.log(self.foo); // bar
        }.call(this));  //myObject.func();这样调用func（）,那么func（）中的this就是前面的对象myObject。
    }
};
myObject.func();

//case2：用bind去绑定this，但要注意bind是返回一个函数，故要bind（this）（），后一个括号表示函数调用。把bind（this）将拷贝一份并改变this的指向的函数执行。

var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log(this.foo);  // bar
        console.log(self.foo);  // bar
        (function() {
            console.log(this.foo);  // bar
            console.log(self.foo);  // bar
        }.bind(this)());
    }
};

myObject.func();
```

```
</script>
```

```
//经典中的经典面试题：小白之前从来没有接触过的一个坑。
<script>
/*
考察this的指向:
难点:数组(类数组)中的元素当做函数调用时的this指向
也就是，如果是调用数组(类数组)中的元素，元素函数中的this是这个数组（类数组）.
 */
var length = 10;
function fn(){
    console.log(this.length);
}
var obj = {
    length: 5,
    method: function (fn){  // [fn, 1, 2]
        fn();  // 10
        arguments[0]();  // 3
    }
};
obj.method(fn, 1, 2);
/*obj.method(fn, 1, 2);传实参fn过去，此时fn拿到函数的地址值拷贝给形参fn，在执行fn（）这里调用是相当window调用fn，this指的是window。而不是obj不要感觉是在obj里面就是，迷惑大家的。this的指向永远跟调用方式有关。
*另外，arguments[0]();调用时，这个时候是类数组中的元素调用，那么这时的this是类数组本身，所以，数组.length是不是输出类数组的长度。
*如果是调用数组(类数组)中的元素，元素函数中的this是这个数组（类数组）.为什么呢？看以下两个例子：
* */

//例子1：
var obj = {
    age : 100,
    foo : function (){
        console.log(this);
    }
}
var ff = obj.foo;
ff();  //window
obj.foo(); //{age: 100, foo: ƒ}
obj["foo"]();  //{age: 100, foo: ƒ}
//上面的这个例子没有问题吧。很自然的。

//例子2：
var arr = [
    function (){
        console.log(this);
    },function (){

    }
];
var f = arr[0];

f();  //window
```

```
/*arr.0()--类似于这么写把，只是数组不允许这样的语法--*/
 arr[0]();  //输出数组本身：(2) [ƒ, ƒ] 。故验证一句话：如果调用数组（类数组）中的元素时，那么这时的this是数组（类数组）本身。

</script>
```

3.2 原型与原型链

3.2.1 五张图理解原型与原型链

【构造函数创建对象】我们先使用构造函数创建一个对象：

```
function Person() {

}

var person = new Person();
person.name = 'name';
console.log(person.name) // name
```

在这个例子中，Person就是一个构造函数，我们使用new创建了一个实例对象person。

很简单吧，接下来进入正题：**【prototype】**

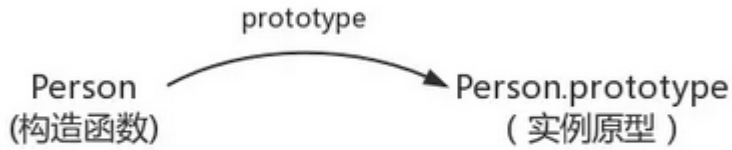
任何的函数都有一个属性 prototype，这个属性的值是一个对象，这个对象就称为这个函数的原型对象。但是一般情况,我们只关注构造函数的原型。比如：

```
function Person() {

}
// 虽然写在注释里，但是你要注意：prototype是函数才会有的属性
Person.prototype.name = 'name';

var person1 = new Person();  //person1是Person构造函数的实例
var person2 = new Person();  //person2是Person构造函数的实例
console.log(person1.name) // name
console.log(person2.name) // name
```

其实，函数的prototype属性指向了一个对象，这个对象正是调用该构造函数而创建的实例的原型,也就是这个例子中的person1和person2的原型。实例其实是通过一个不可见的属性[[proto]]指向的。

你可以这样理解：每一个JavaScript对象(null除外)在创建的时候就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型"继承"属性。

让我们用一张图表示构造函数和实例原型之间的关系：



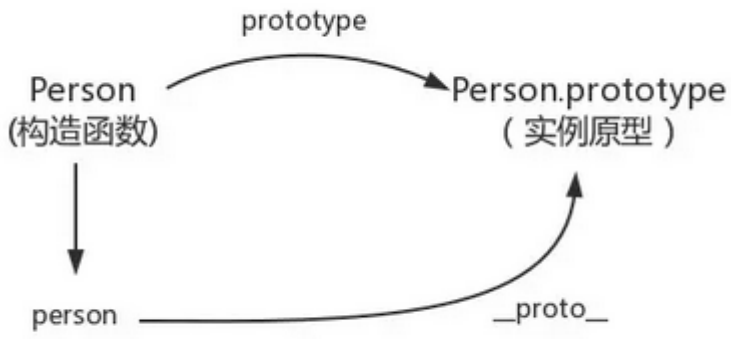
那么我们该怎么表示实例与实例原型，也就是person1 person2 和 Person.prototype之间的关系呢，这时候我们就要讲到第二个属性：**[[proto]]**

当使用构造函数创建对象的时候, 新创建的对象会有一个不可见的属性[[proto]], 他会指向构造函数的那个原型对象。事实上，每一个JavaScript对象(除了null)都具有的一个不可见属性，叫[[proto]]，这个属性会指向该对象的原型。

为了证明这一点,我们可以在火狐或者谷歌中输入：

```
function Person() {

}
var person1 = new Person();
console.log(person1.__proto__ === Person.prototype); //true
```

于是我们更新下关系图：

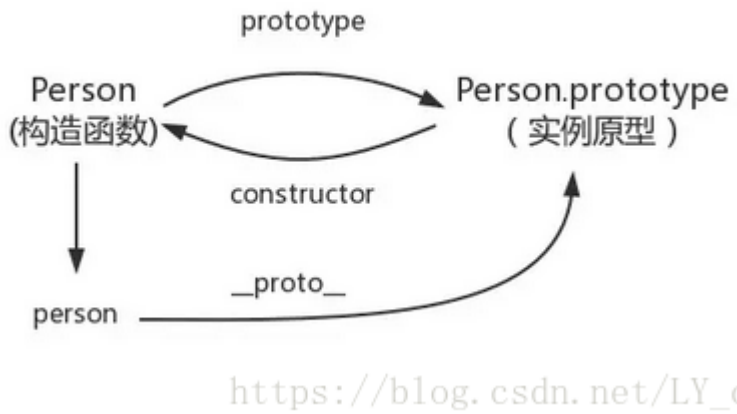


既然实例对象和构造函数都可以指向原型，那么原型是否有属性指向构造函数或者实例呢？指向实例倒是没有，因为一个构造函数可以生成多个实例，但是原型指向构造函数倒是有的，这就要讲到第三个属性：**【constructor】**，每个原型都有一个constructor属性指向关联的构造函数。

为了验证这一点，我们可以尝试：

```
function Person() {

}
console.log(Person === Person.prototype.constructor); //true
```

所以再更新下关系图：



综上我们已经得出：

```
function Person() {

}
var person1 = new Person();
//对象的__proto__属性：创建对象时自动添加的，默认值为构造函数的prototype属性值
console.log(person1.__proto__ == Person.prototype) // true
console.log(Person.prototype.constructor == Person) // true

// 顺便学习一个ES5的方法,可以获得对象的原型
console.log(Object.getPrototypeOf(person1) === Person.prototype) //true
```

了解了构造函数、实例原型、和实例之间的关系，接下来我们讲讲实例和原型的关系：**【实例与原型】**。当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层为止。

举个例子：

```
function Person() {

}
//往Person对象原型中添加一个属性
Person.prototype.name = 'name';
//创建一个person1实例对象
var person1 = new Person();
//给创建的实例对象person1添加一个属性
person1.name = 'name of this person1';
//查找person1.name，因为本身实例对象有，那么就找到了自身实例对象上的属性和属性值
console.log(person1.name) // name of this person1
```

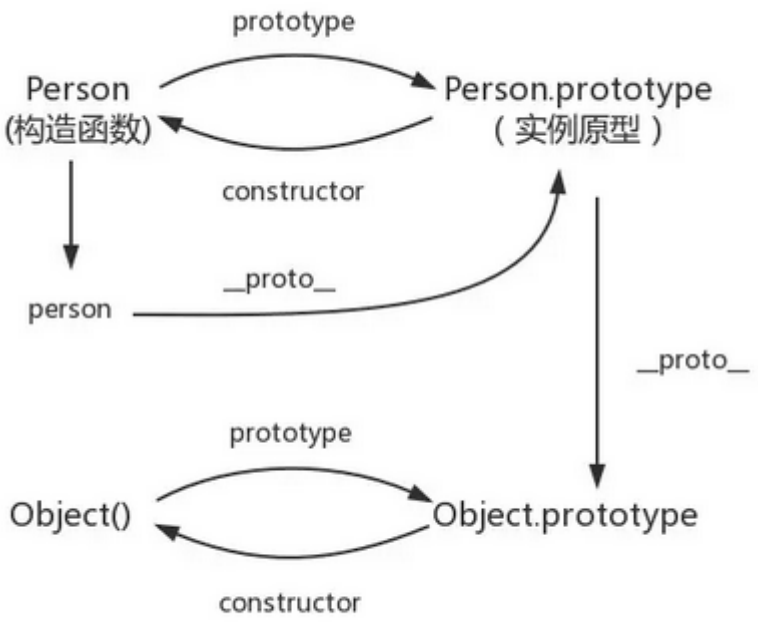
```
//删除实例对象的属性和属性值
delete person1.name;
//查找属性name，在实例对象自身上找不到，通过proto指向往原型链上找，在原型对象中找到
console.log(person1.name) // name
```

在这个例子中，我们设置了person1的name属性，所以我们可以读取到为'name of this person1'，当我们删除了person1的name属性时，读取person1.name，从person1中找不到就会从person的原型也就是person.**proto** == Person.prototype中查找，幸运的是我们找到了为'name'，但是万一还没有找到呢？原型的原型又是什么呢？

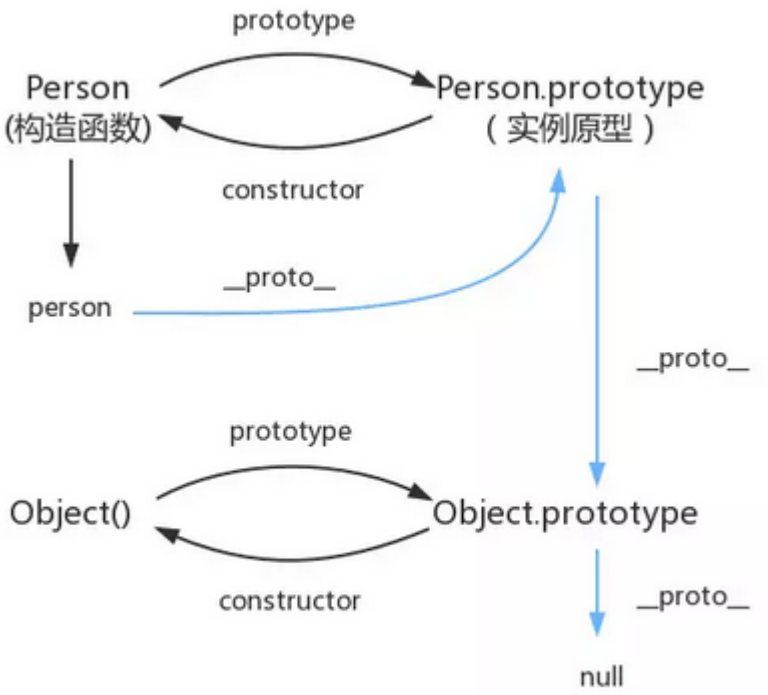
```
var obj = new Object();
obj.name = 'name'
console.log(obj.name) // name
```

所以原型对象是通过Object构造函数生成的，结合之前所讲的一句很重要的话，几乎就是涵盖原型与原型链知识的始终的一句话，那就是：实例对象的proto指向构造函数的prototype。也就是说，Person.prototype这个原型对象（实例原型）是通过Object这个构造函数new出来的，也就是Person.prototype这个原型对象是Object的实例，所以这个实例会有proto属性指向Object构造函数的原型对象Object.prototype。

这里呢插入一句总结出来的话，逆推顺推都是可行的，那就是实例通过proto这个属性指向其构造函数的原型对象。所以我们再更新下关系图：



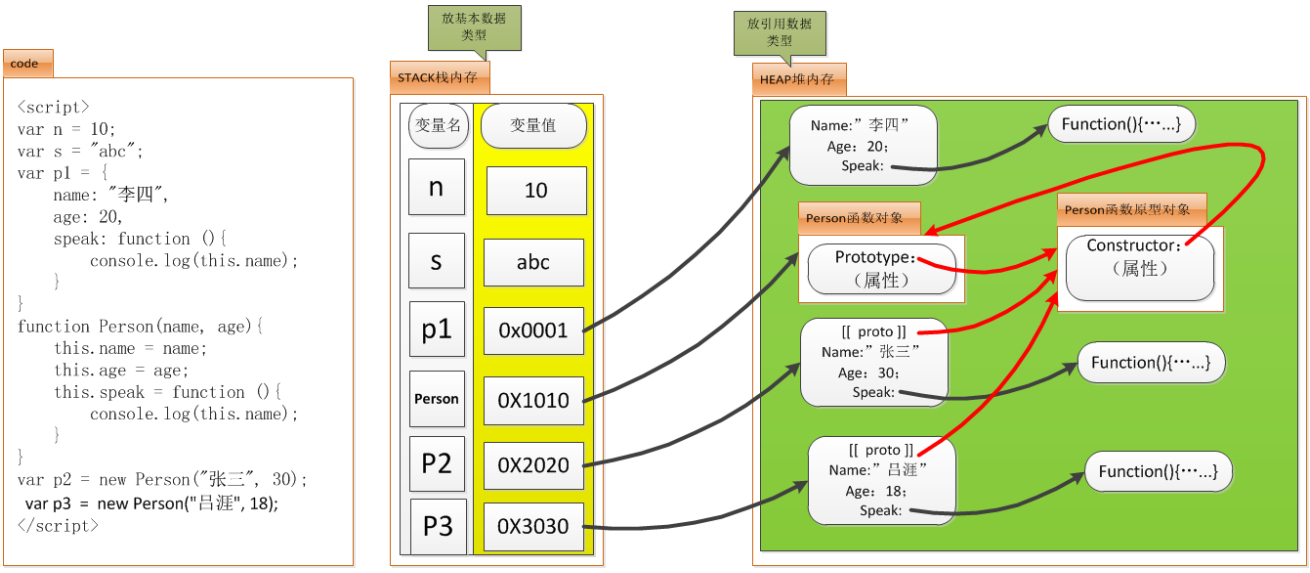
那Object.prototype的原型呢？null，嗯，就是null。所以查到Object.prototype就可以停止查找了。所以最后一张关系图就是：



那【**原型链**】是啥？那就是由proto这个属性进行查找的一个方向这就是一条原型链。图中由相互关联的原型组成的链状结构就是原型链，也就是蓝色的这条线，都是通过proto属性进行查找的。

那么访问一个对象的属性时，怎么通过原型链去查找属性或方法呢？----先在自身属性中查找，找到返回。如果没有, 再沿着proto这条链向上查找, 找到返回。如果最终没找到, 返回undefined。

3.2.2 从代码中看原型与原型链



解析上图：首先n ,s 都是全局变量，然后通过对象字面量的方法去创建了一个对象。然后有一个构造函数（之所以是构造函数，是因为后面有new他的新对象），这个构造函数就会有函数声明提前，当构造函数声明时，就会去在内存中创建一个person的函数对象，这个函数对象里**只有prototype属性**，去指向person的函数原型对象。要注意，现在还没有去执行里面的代码，只是函数声明时创建了一个person的函数对象。后面就是new的实例对象，新new出来的实例对象P2 P3 就会在内存中分配一块内存去把地址值给它，现在才会去执行构造函数中的代码。所以只有P2 P3才会 有name age speak 属性和方法。这些新new出来的实例对象就会有一个不可见的属性proto，去指向这个原型对象。而最终这个person的函数原型对象会有指向一个object的原型对象，再上去其实就是null。这就一层一层往上走就是原型链，因为原型链，我们才会有继承的特性。

注几点：

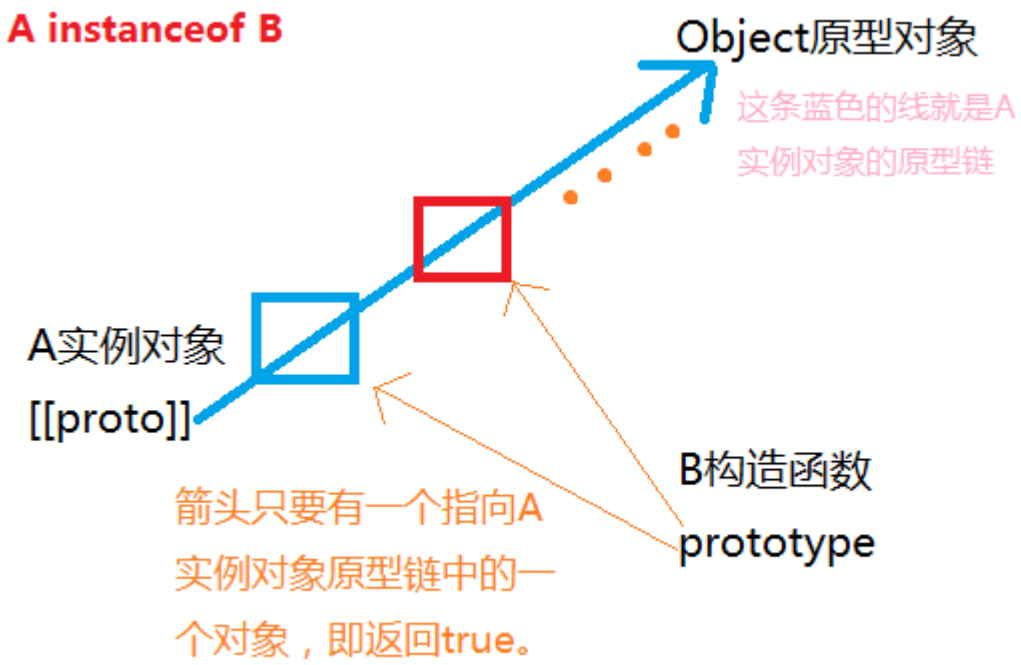
1. 从上图的图示中可以看到，创建 P2 P3 实例对象虽然使用的是 Person 构造函数，但是对象创建出来之后，这个P2 P3 实例对象其实已经与 Person 构造函数（函数对象）没有任何关系了，P2 P3 实例对象的 [[proto]] 属性指向的是 Person 构造函数的原型对象。
2. 如果使用 new Person() 创建多个对象，则多个对象都会同时指向 Person 构造函数的原型对象。
3. 我们可以手动给这个原型对象添加属性和方法，那么 P2 P3 ····这些实例对象就会共享这些在原型中添加的属性和方法。也就是说，原型对象相当于公共的区域，所有的同一类的实例都可以去访问到原型对象。
4. 如果我们访问P2 实例对象 中的一个属性 gender ，如果在P2 对象中找到，则直接返回。如果 P2 对象中没有找到，则直接去P2对象的 [[proto]] 属性指向的原型对象中查找，如果查找到则返回。(如果原型中也没有找到，则继续向上找原型的原型---原型链)。
5. 读取对象的属性值时: 会自动到原型链中查找
6. 设置对象的属性值时: 不会查找原型链, 如果当前对象中没有此属性, 直接添加此属性并设置其值。比如通过P2 对象只能读取原型中的属性 name 的值，并不能修改原型中的属性 name 的值。 P2.gender= "male" ；并不是修改了原型中的值，而是在 P2 对象中给添加了一个属性 gender。
7. 方法一般定义在原型中, 属性一般通过构造函数定义在对象本身上。

```
/*另外看看，原型与原型链的三点关注*/
/*
  1. 函数的显示原型指向的对象默认是空Object实例对象(但Object不满足)
   */
  console.log(Fn.prototype instanceof Object) // true
  console.log(Object.prototype instanceof Object) // false
  console.log(Function.prototype instanceof Object) // true
  /*
  2. 所有函数都是Function的实例(包含Function)
  */
  console.log(Function.__proto__===Function.prototype) //true
  /*
  3. Object的原型对象是原型链尽头
   */
  console.log(Object.prototype.__proto__) // null
```

3.2.3 探索instanceof

1. instanceof是如何判断的?
    - 表达式: A instanceof B
    - 如果B构造函数的原型对象（B.prototype）在A实例对象的原型链（A.proto.proto·····沿着原型链）上, 返回true, 否则返回false。(见下图)

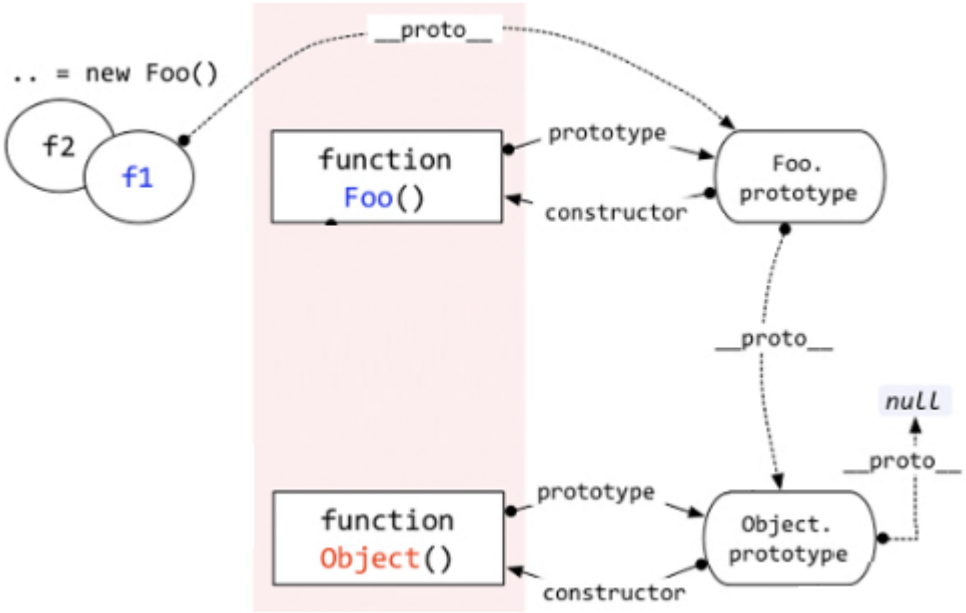
- 也就是说A实例对象的原型链上可能会有很多对象，只要B构造函数的原型对象有一个是在其原型链上的对象即可返回true。
- 反过来说也一样，实例对象A是否可以通过proto属性（沿着原型链,A.proto.proto……）找到B.prototype（B的原型对象）.找到返回true，没找到返回false.
- 注1：对实例对象的说明，事实上，实例对象有两种。一种是我们经常说的new 出来的实例对象（比如构造函数Person new出来p1 p2...,这些都是实例对象），另外一种就是函数，函数本身也是实例，是Function new出来的。但我们一般说的实例对象就是指new出来的类似于p1 p2这些的实例对象。



2. Function是通过new自己产生的实例（Function.proto===Function.prototype）

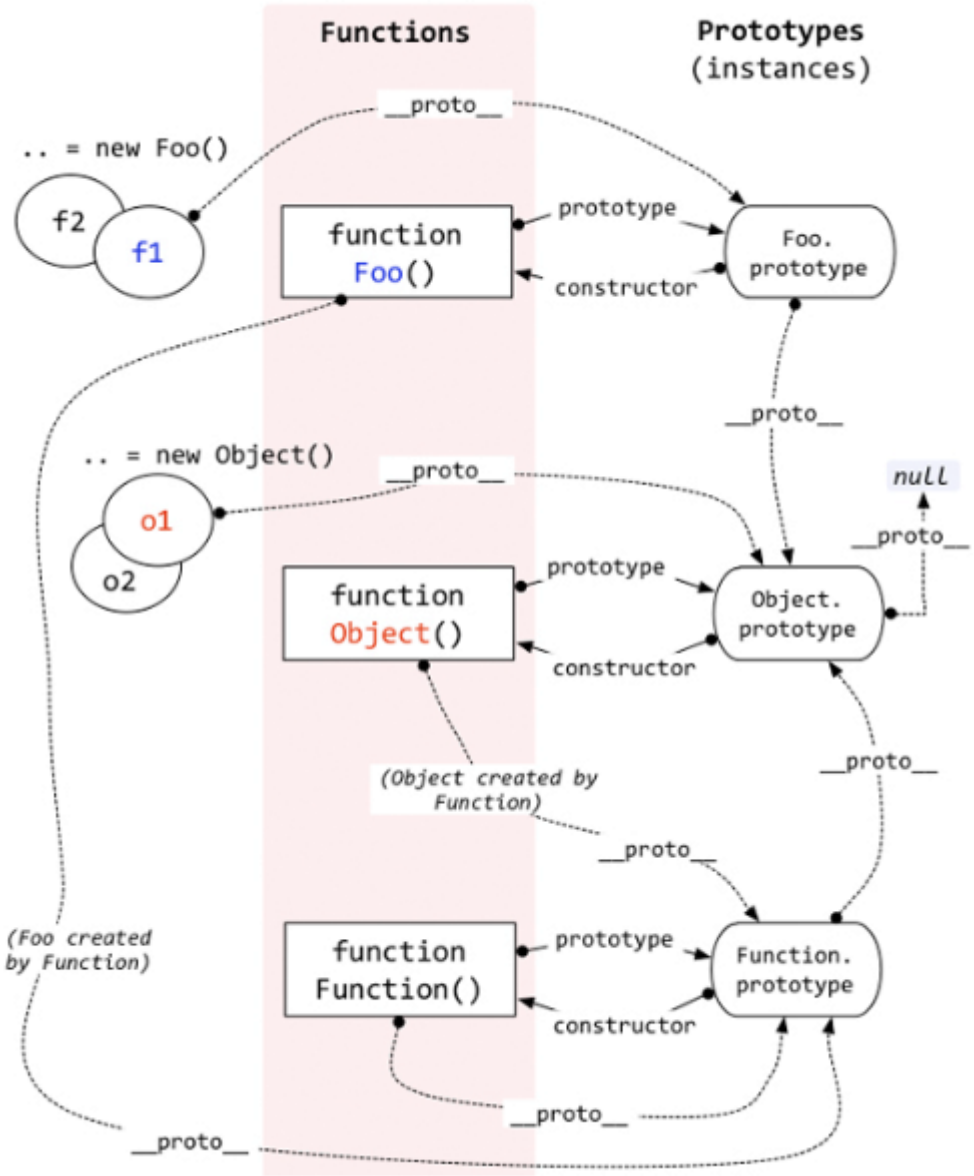
案例一：

```
/* 案例1 */
  //一个构造函数Foo
  function Foo() {  }
  //一个f1实例对象
  var f1 = new Foo()
  //翻译:f1是Foo的实例对象吗？
  //还记得我说过，一个实例对象通过proto指向其构造函数的原型对象上。
  //深入翻译：f1这个实例对象通过proto指向是否可以找到Foo.prototype上呢？
  console.log(f1 instanceof Foo) // true
  //这行代码可以得出，沿着proto只找了一层就找到了。
  console.log(f1.__proto__ === Foo.prototype);   // true

  //翻译：f1是Object的实例对象吗？
  //深入翻译：f1这个实例对象通过proto指向是否可以找到Object.prototype上呢？
  console.log(f1 instanceof Object) // true
  //这两行代码可以得出，沿着proto找了两层才找到。事实上，f1.__proto__找到了Foo.prototype（Foo构造函数原型上），再次去.__proto__，找到了Object的原型对象上。见下图。
  console.log(f1.__proto__ === Object.prototype);  // false
  console.log(f1.__proto__.__proto__ === Object.prototype);  // true
```



案例二：

```
/*案例2*/
//这个案例的实质还是那句话：一个实例对象通过proto属性指向其构造函数的原型对象上。
//翻译：实例对象Object是否可以通过proto属性（沿着原型链）找到Function.prototype（Function的原型对象）
  console.log(Object instanceof Function) // true
//以上结果的输出可以看到下图，Object.__proto__直接找到一层就是Function.prototype.（Object created by Function）可知Object构造函数是由Function创建出来的，也就是说，Object这个实例是new Function出来的。

  console.log(Object instanceof Object) // true
//很有意思。上面我们已经知道Object这个实例是new Function出来的。也就是Object.proto指向Function.prototype。有意思的是，Function的原型对象又是Object原型对象的一个实例，也就是Function.prototype.proto 指向  Object.prototype .很有意思吧，见下图很更清楚这个“走向”。


  console.log(Function instanceof Function) // true
//由这个可知，可以验证我们的结论：Function是通过new自己产生的实例。
Function.proto===Function.prototype

  console.log(Function instanceof Object) // true
//Function.proto.proto===Function.prototype（找了两层）

  //定义了一个Foo构造函数。由下图可知，Foo.proto.proto.proto===null
  function Foo() {}
  console.log(Object instanceof  Foo) // false
//这条语句要验证的是，Object是否可以通过其原型链找到Foo.prototype。
// Object.proto.proto.proto=null  并不会找到Foo.prototype。所以，返回FALSE。
```



注意：函数是对象。那你觉得函数包含的大？还是对象大呢？
看图，对象是由函数创造的。（Object created by Function）
也就是说，对象是new Function得到的。
继续翻译，对象是实例　Function是构造函数。
继续翻译，对象这个实例有不可见属性proto指向 Function构造函数的原型对象（Function.prototype）。
故，函数与对象的关系是：函数更大，它包含对象。
这个我个人觉得很重要，务必理解透。

3.2.4 一些概念的梳理

- 所有函数都有一个特别的属性:
  - `prototype`：显式原型属性
- 所有实例对象都有一个特别的属性:

    - `__proto__`: 隐式原型属性
- 显式原型与隐式原型的关系
    - 函数的prototype: 定义函数时被自动赋值, 值默认为{}, 即用为原型对象
    - 实例对象的**proto**: 在创建实例对象时被自动添加, 并赋值为构造函数的prototype值
    - 原型对象即为当前实例对象的父对象
- 原型链
    - 所有的实例对象都有**proto**属性, 它指向的就是原型对象
    - 这样通过**proto**属性就形成了一个链的结构---->原型链
    - 当查找对象内部的属性/方法时, js引擎自动沿着这个原型链查找
    - 当给对象属性赋值时不会使用原型链, 而只是在当前对象中进行操作

3.2.5 习题与案例

经典面试题1：

```
/*阿里面试题*/
①function Person(){
    ②getAge = function (){
        console.log(10)
    }
    ③return this;
}

④Person.getAge = function (){
    console.log(20);
}

⑤Person.prototype.getAge = function (){
    console.log(30);
}

⑥var getAge = function (){
    console.log(40)
}

⑦function getAge(){
    console.log(50)
}


Q1：Person.getAge() // 20
Q2：getAge() // 40
Q3：Person().getAge() // 10
Q4：getAge() // 10
Q5：new Person().getAge() // 30
Q6：new Person.getAge(); // 20


//解析：
整体代码块①定义了构造函数Person

②是在构造函数中有一个未声明的变量，这个变量是引用变量，内容为地址值。指向一个函数对象。又因为，未使用严格
```

模式下，在函数中不使用var声明的变狼都会成为全局变量。（注意这里不是属性，是全局变量）同时也要注意，这里②和③的语句在解析到这里后并没有执行。执行的话就要看有没有new（作为构造函数使用），或者有没有加（）调用（作为普通函数使用）。
③返回一个this。这个this是谁现在还不知道。需要明白js中的this是动态的，所以根据上一节this的总结才定位到this到底是谁。
④Person.getAge是典型的“对象.属性（方法）”的形式，所以它是给Person函数对象上添加一个getAge的方法。等同于：

```
  function Person.getAge(){
    console.log(20);
}
```

函数名其实就是变量名。
⑤在构造函数的原型中添加了getAge的方法
⑥这里也是给一个全局变量赋值一个地址值，使其指向一个函数对象。注意，这里var的变量会声明提前。与代码块②区别，这里当解析完后，getAge已经指向一个函数对象啦。可以看做：

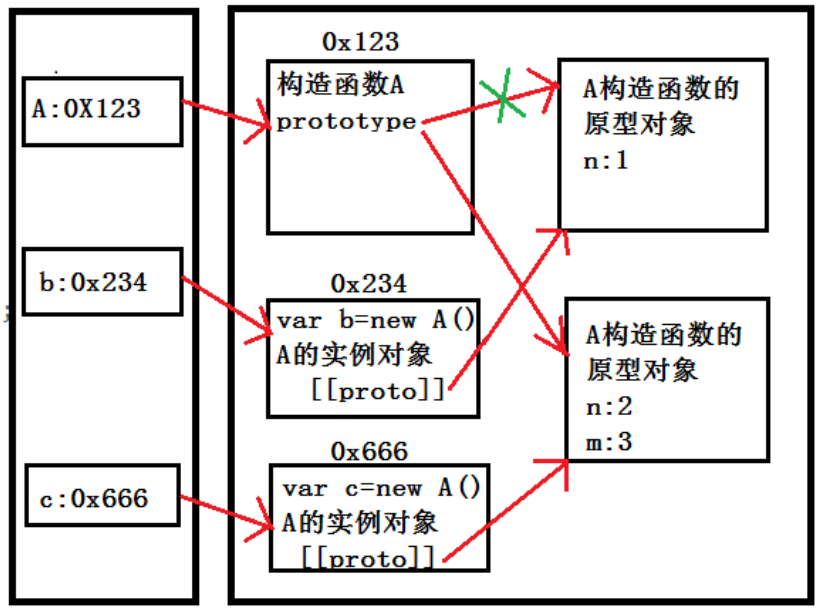
```
function getAge(){
    console.log(40)
}
```

⑦定义一个函数，函数也会声明提前。在栈内存有getAge，内容值为一个地址值，指向一个函数对象。

Q1：对象.属性方法（）。代码块④产生的结果。
Q2：调用函数，全局作用域里的。那只有代码块⑥产生结果。
Q3：Person().getAge()。先看前面一部分Person(),把Person当做一个普通函数调用，执行Person函数体对全局变量getAge进行定义并重新指向，也就是Person（）执行了代码块②而覆盖了代码块⑥的操作。又返回this，根据Person（）这种调用方式，可知this就是window。所以就是“window.gerAge（）”，因被覆盖了，所以这行代码执行结果是代码块②产生。
Q4：getAge()相当于window.getAge(); 还是上一个语句的结果，代码块②产生结果。
Q5：new Person()先看这部分，就是new出来一个实例，你可以想成p1，那么p1.getAge();p1是一个Person的实例，p1中有不可见的[[proto]]属性，指向Person的原型对象。那么p1.getAge (),现在p1本身找，找不到就沿着原型链（proto指向链）去找，好找到了原型对向中有，因为代码块⑤产生作用。
Q6：new Person.getAge(); 可以把Person.getAge看成一个对象，去new它，是不是类似于我们平常var p1=new Person（）；这样的操作，所以我们把Person.getAge看做一个构造函数去new它。由上面对代码块④的理解，可以看做那样的函数，所以结果就是代码块④产生的结果。

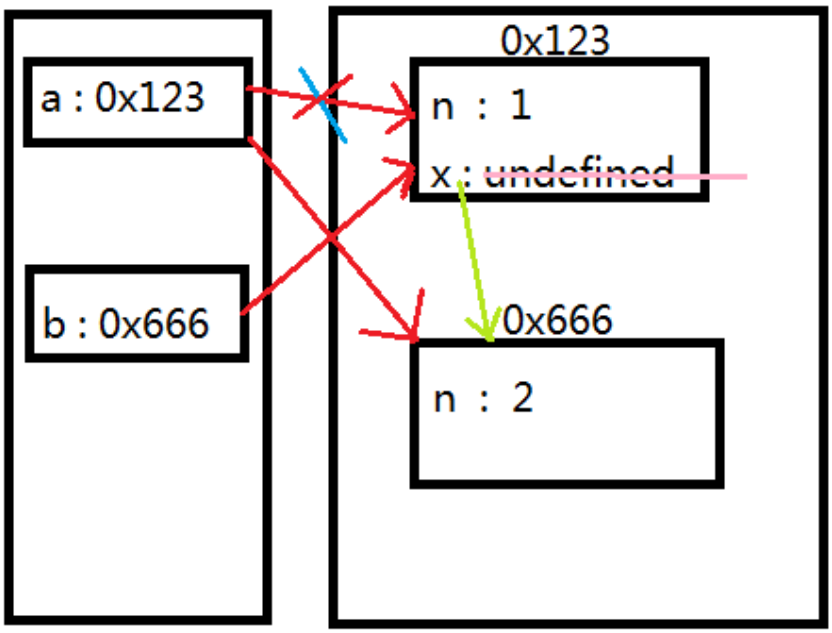
```
//面试题2：
function A () {

  }
  A.prototype.n = 1;
  var b = new A();
  A.prototype = {
    n: 2,
    m: 3
  };
  var c = new A();
  console.log(b.n, b.m, c.n, c.m);  //1 undefined 2 3
//见下图:
```

```
function A () {

}
A.prototype.n = 1;
var b = new A();
A.prototype = {
  n: 2,
  m: 3
};
var c = new A();
console.log(b.n, b.m, c.n, c.m);
//1 undefined 2 3
```




```
//面试题3：连续赋值问题
//与上题的区别在于如何理解a.x的执行顺序
<script>
var a = {n: 1};
var b = a;
a.x = a = {n: 2};  //先定义a.x再去从右往左赋值操作。
console.log(a.x);  // undefined  对象.属性 找不到 是返回undefined  变量找不到则报错！
console.log(b);  // {n :1, x : {n : 2}}
</script>
//见下图分析
```

```
var a = {n: 1};
var b = a;

a.x = a = {n: 2};

console.log(a.x);
// undefined
console.log(b);
// {n :1, x : {n :
2}}
```




```
//面试题4：
//构造函数F
  function F (){};
  Object.prototype.a = function(){
    console.log('a()')
  };
  Function.prototype.b = function(){
    console.log('b()')
  };
  //new一个实例对象f
  var f = new F();

  f.a();  //a()
  f.b();  //报错，找不到
  F.a();  //a()
  F.b();  //b()
```

2.3 执行上下文与执行上下文栈

2.3.1 变量提升与函数提升

1. 变量声明提升
    - 通过var定义(声明)的变量, 在定义语句之前就可以访问到
    - 值: undefined
    - 注：未使用var关键字声明变量时，该变量不会声明提升。

```
console.log(c); //报错，c is not defined.
console.log(b); //undefined
var b=0;
c=4;
console.log(c); //4 意外的全局变量->在ES5的严格模式下就会报错。
console.log(b); //0
```

2. 函数声明提升
   - 通过function声明的函数, 在之前就可以直接调用
   - 值: 函数定义(对象)
   - 注1：函数声明(Function Declaration)和函数表达式(Function Expression)是有微妙的区别，要明确他们两是Javascript两种类型的函数定义，两个概念上是并列的。定义函数的方式有两种：一种是函数声明，另一种就是函数表达式。
   - 注2：函数表达式并不会声明提升。

3. 先有变量提升, 再有函数提升

```
//案例一：
var a = 3;
  function fn () {
    console.log(a);  //undefined
    var a = 4
  }
  fn();

//上面这段代码相当于
 var a = 3;
  function fn () {
      var a;
      console.log(a);  //undefined
      a = 4
  }
  fn();
```

```
//案例二：
  console.log(b) //undefined  变量提升
  fn2() //可调用  函数提升
  fn3() //不能调用，会报错。  fn3是一个函数表达式，并不会函数提升，实际上他是变量提升。

  var b = 3
  function fn2() {
    console.log('fn2()')
  }
  var fn3 = function () {
    console.log('fn3()')
  }
```

4. 问题: 变量提升和函数提升是如何产生的?

An:因为存在全局执行上下文和函数执行上下文的预处理过程。所以我们就来学习下一节的执行上下文。

2.3.2 执行上下文

1. 代码分类(位置)
   - 全局代码
   - 函数(局部)代码
2. 执行上下文分为全局执行上下文和函数执行上下文
3. 全局执行上下文
   - 在执行全局代码前将window确定为全局执行上下文对象（虚拟的）
   - 对全局数据进行预处理（收集数据）
     - var定义的全局变量==>值为undefined, 并添加为window的属性
     - function声明的全局函数==>赋值(fun), 添加为window的方法
     - this==>赋值(window)
   - 开始执行全局代码

```
//全局执行上下文
console.log(a1);  //undefined
console.log(a2);  //undefined
a2();   //也会报错，a2不是一个函数
console.log(a3);  //ƒ a3() {console.log('a3()')}
console.log(a4)   //报错，a4没有定义
console.log(this); //window

var a1 = 3;
//函数表达式，实际上是变量提升。而不是函数提升。
var a2 = function () {
  console.log('a2()')
};
function a3() {
  console.log('a3()')
}
a4 = 4;
```

4. 函数执行上下文
   - 在调用函数, 准备执行函数体之前, 创建对应的函数执行上下文对象(虚拟的, 存在于栈中。栈会分为全局变量栈和局部变量栈，局部变量栈可以理解为是一个封闭的内存空间。虽然我们编写的代码无法访问这个对象，但解析器在处理数据时会在后台使用它。）
   - 对局部数据进行预处理（收集数据）
     - 形参变量==>赋值(实参)==>添加为执行上下文的属性
     - arguments==>赋值(实参列表), 添加为执行上下文的属性
     - var定义的局部变量==>undefined, 添加为执行上下文的属性
     - function声明的函数 ==>赋值(fun), 添加为执行上下文的方法
     - this==>赋值(调用函数的对象)
   - 开始执行函数体代码

```
//函数执行上下文
function fn(a1) {
   console.log(a1);  //2  实参对形参赋值
   console.log(a2);  //undefined  函数内部局部变量声明提升
   a3();     //a3()  可调用  函数提升
   console.log(arguments);  //类数组[2,3]
   console.log(this);  //window

   var a2=3;
   function a3() {
     console.log("a3()");
   }
 }
 fn(2,3);  //执行，不执行不会产生函数执行上下文
```

5.全局执行上下文和函数执行上下文的生命周期

全局 : 准备执行全局代码前产生, 当页面刷新/关闭页面时死亡

函数 : 调用函数时产生, 函数执行完时死亡

2.3.3 执行上下文栈

- 执行上下文栈流程理解

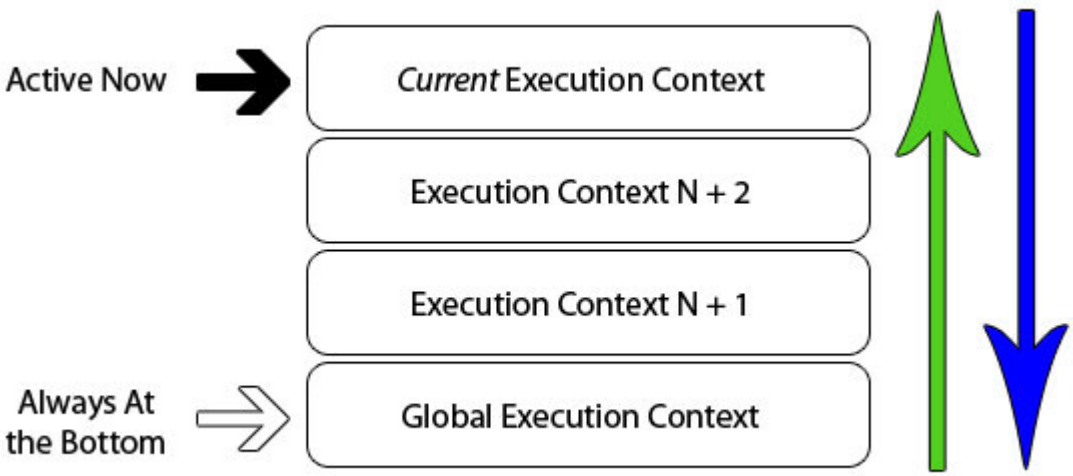
1. 在全局代码执行前, JS引擎就会创建一个栈来存储管理所有的执行上下文对象
2. 在全局执行上下文(window)确定后, 将其添加到栈中(压栈)
3. 在函数执行上下文创建后, 将其添加到栈中(压栈)
4. 在当前函数执行完后,将栈顶的对象移除(出栈)
5. 当所有的代码执行完后, 栈中只剩下window

```
<script type="text/javascript">
                            //1. 进入全局执行上下文
  var a = 10;
  var bar = function (x) {
    var b = 5;
    foo(x + b)              //3. 进入foo执行上下文
  };
  var foo = function (y) {
    var c = 5;
    console.log(a + c + y)
  };
  bar(10);   //2. 进入bar函数执行上下文(注：函数执行上下文对象在函数调用时产生，而不是函数声明时产生)

  /*以上这种情况整个过程产生了3个执行上下文
  * 调用一次函数产生一个执行上下文
  * 如果在上面代码最后一行的bar（10），再调用一次bar（10），那么就会产生5个上下文。
  * 因为第一个bar（10）产生一个函数上下文 在bar函数中调用foo，又产生一个函数执行上下文。
  * 那么现在又调用bar（10），与上面一个样会产生两个上下文，加起来4个函数执行上下文。

  * 最后加上window的全局变量上下文，一共五个执行上下文。
```

```
 *
 * */
</script>
```

图解：


```
<script type="text/javascript">
                                //1. 进入全局执行上下文
  var a = 10
  var bar = function (x) {
    var b = 5
    foo(x + b)                  //3. 进入foo执行上下文
  }
  var foo = function (y) {
    var c = 5
    console.log(a + c + y)
  }

  bar(10)                       //2. 进入bar函数执行上下文
</script>
```



注解1：这个过程有点像递归函数的回溯思想。在栈中，先有window的全局上下文，然后执行bar（）会把bar函数执行上下文压入栈中。bar中调用foo，把foo函数执行上下文压入栈中，foo函数执行完毕，释放，便会把foo函数执行上下文pop（推出来）。逐渐bar执行完毕，pop出bar函数执行上下文，最后只剩下window上下文。

注解2：假设一个情况：f1（）函数中会调用f2（）和f3（）函数。那么在当前时刻栈中可最多达到几个上下文？

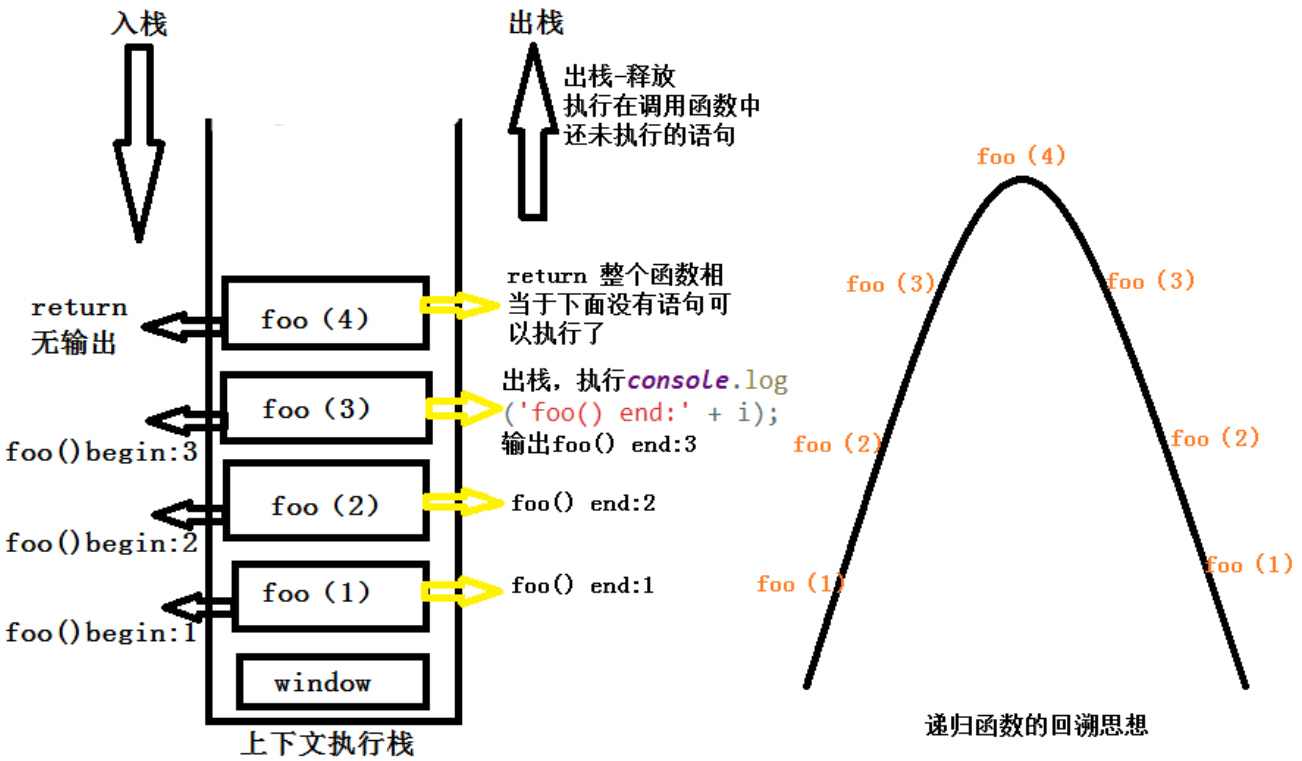
An: 当f1（）执行，会先调用f2（），调用完后，f2（）已经完成了使命，它的生命周期就结束了，所以栈 就会释放掉他，在执行f3（），所以栈中也就最多三个上下文。f3( ) f1( ) window.

假设另一个情况：f1（）函数中会调用f2（）,f2（）中会调用f3（）函数。那么在当前时刻栈中可最多达到几个上下文？

An: 当f1（）执行，会先调用f2（），执行f2（）时要调用f3（），所以，栈中可达到4个上下文。f3（） f2（） f1（） window .

2.3.4 习题与案例

```
//执行上下文栈的面试题1：
<!--
1. 依次输出什么?
2. 整个过程中产生了几个执行上下文?
-->
<script type="text/javascript">
  console.log('global begin: '+ i); //undefined 变量提升
  var i = 1;
  foo(1);
  function foo(i) {
    if (i == 4) {
      return;
    }
    console.log('foo() begin:' + i);
    foo(i + 1);
    console.log('foo() end:' + i);
  }
  console.log('global end: ' + i)  //1 全局变量i,其他的函数中的i当执行结束后就销毁了。
</script>

//执行结果
  global begin: undefined
  foo() begin:1
  foo() begin:2
  foo() begin:3
  foo() end:3
  foo() end:2
  foo() end:1
  global end: 1

  //一共产生5个上下文
  //分析见下图，我画的很清楚了。这张图画了12min。主要就是入栈出栈，在出栈前，回溯原来的那个函数，那个函数执行上下文还在，如果还有 没有执行完的语句 会在这个时候执行。当剩余的语句已经执行完了，那么这个函数的执行上下文生命周期结束，释放出栈。想想我们递归调用去求阶乘的例子，思想是一样的。
```




```
//一些综合关于执行上下文和执行上下文栈的面试题：
<script type="text/javascript">
  /*
  测试题1：[考查知识点]先预处理变量，后预处理函数
  */
  function a() {} //函数提升
  var a;  //变量提升
  //先预处理变量，后预处理函数。也就是，函数提升会覆盖变量提升。
  console.log(typeof a); //function

  /*
  测试题2:[考查知识点] 变量预处理，in操作符 (在window上能不能找到b,不管有没有值)
   */
  if (!(b in window)) {
    var b = 1;  //在ES6之前没有块级作用域，所以这个变量b 相当于window的全局变量
  }
  console.log(b); //undefined

  /*
  测试题3：[考查知识点]预处理，顺序执行
  这个题笔者认为出的相当好。混乱读者的视角。当然再次强调，面试题是专门命题出来考查的，实际开发上可能有些不会这么用。但主要作用就是深入理解。
   */
  var c = 1;
  function c(c) {
    console.log(c);
    var c = 3;

  }
```

```
  c(2); //报错。 c is not a function

  //这个题包含了变量和函数声明提升的问题，就是等价于一下的代码：
  var c;  //变量提升
  function c(c) {  //函数提升，覆盖变量提升
    console.log(c);
    var c = 3;   //函数内部的局部变量（在栈内存的封闭内存空间里，外面看不到）
  }
  c=1;
  console.log(c);
  c(2);  //c is not a function  c是一个变量，值为number类型的数值.怎么可以执行？

</script>
```

```
//变量提升和函数提升的面试题（执行上下文）
<script>
/*
考察点：声明提前
难点：函数优先

调用一开始，就会先创建一个局部变量a，（因为a是形参），然后把实参的值1赋值给a
a= 1
几乎同时时间节点上，那么一瞬间，开始处理函数的内其他变量的或者函数的声明，给他们提前。
a = function(){
}
以上这些过程都是函数执行上下文的预处理过程
正式执行内部函数的代码
console.log(a); //就是function源码  输出：ƒ a(){}
a = 2
*/
function fn(a){
    console.log(a);  // function的源码
    var a = 2;
    function a(){

    }
    console.log(a);  // 2
}
fn(1);
</script>
```

2.4 作用域与作用域链

2.4.1 作用域

1.理解:

- 作用域: 就是一块"地盘",一块代码区域, 在编码时就确定了, 不会再变化（见下图解）
- 作用域链: 多个嵌套的作用域形成的由内向外的结构, 用于查找变量（见下图解）

2.分类:

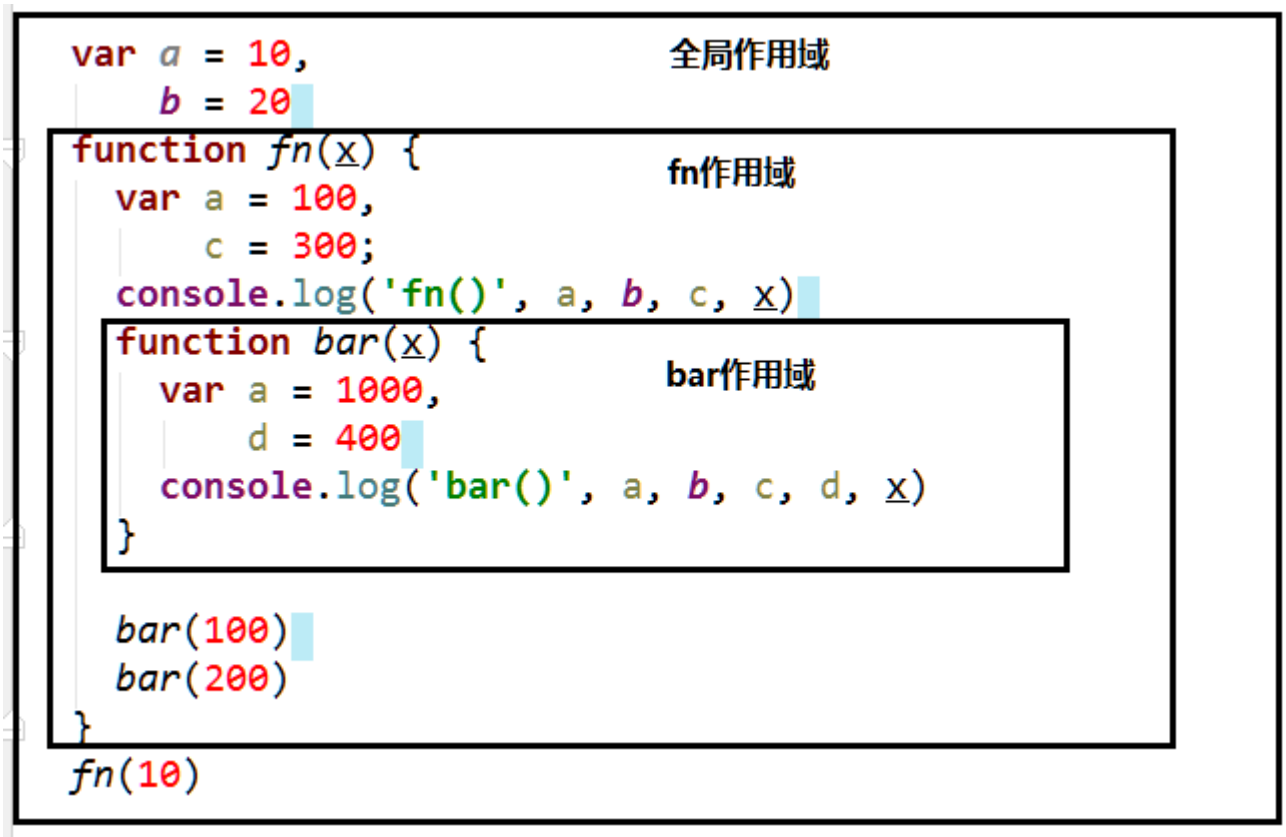
- 全局
- 函数
- js没有块作用域(但在ES6有了！)

3.作用

- 作用域: 隔离变量, 可以在不同作用域去定义同名的变量，不会造成冲突。不同作用域下同名变量不会有冲突。例如，在全局中有一个变量b，那么在函数体中能不能有变量b，当然可以，这就是分隔变量。
- 作用域链: 查找变量

```
//案例解析
var a = 10,
    b = 20
  function fn(x) {
    var a = 100,
      c = 300;
    console.log('fn()', a, b, c, x)
    function bar(x) {
      var a = 1000,
        d = 400
      console.log('bar()', a, b, c, d, x)
    }

    bar(100)
    bar(200)
  }
  fn(10);
//输出结果:
//fn() 100 20 300 10
//bar() 1000 20 300 400 100
//bar() 1000 20 300 400 200
```

4.作用域的图解如下：



2.4.2 作用域与执行上下文

1. 区别1
   - 全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时就已经确定了。而不是在函数调用时。
   - 全局执行上下文是在全局作用域确定之后， js代码马上执行之前创建。
   - 函数执行上下文是在调用函数时, 函数体代码执行之前创建。
2. 区别2
   - 作用域是静态的, 只要函数定义好了就一直存在, 且不会再变化。
   - 执行上下文是动态的, 调用函数时创建, 函数调用结束时就会自动释放(不是通过垃圾回收机制回收)。
3. 联系
   - 执行上下文(对象)是从属于所在的作用域
   - 全局上下文环境==>全局作用域
   - 函数上下文环境==>对应的函数作用域

4.作用域与执行上下文图解如下：

```
    ![](G:\尚硅谷JS高级\源码_课件\JSAdvance\work\02_函数高级\01_原型与原型链\作用域和执行上细纹.png)
```

2.4.3 作用域链

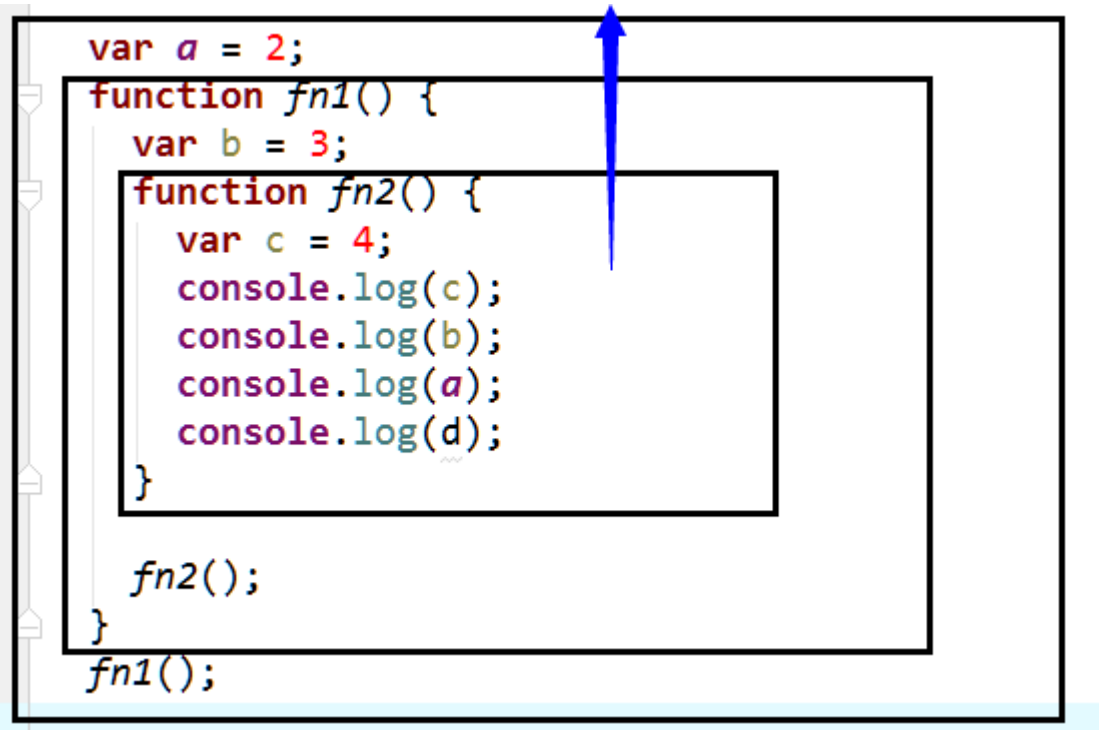
1. 理解
   - 多个上下级关系的作用域形成的链, 它的方向是从下向上的(从内到外)

- 查找变量时就是沿着作用域链来查找的

2. 查找一个变量的查找规则

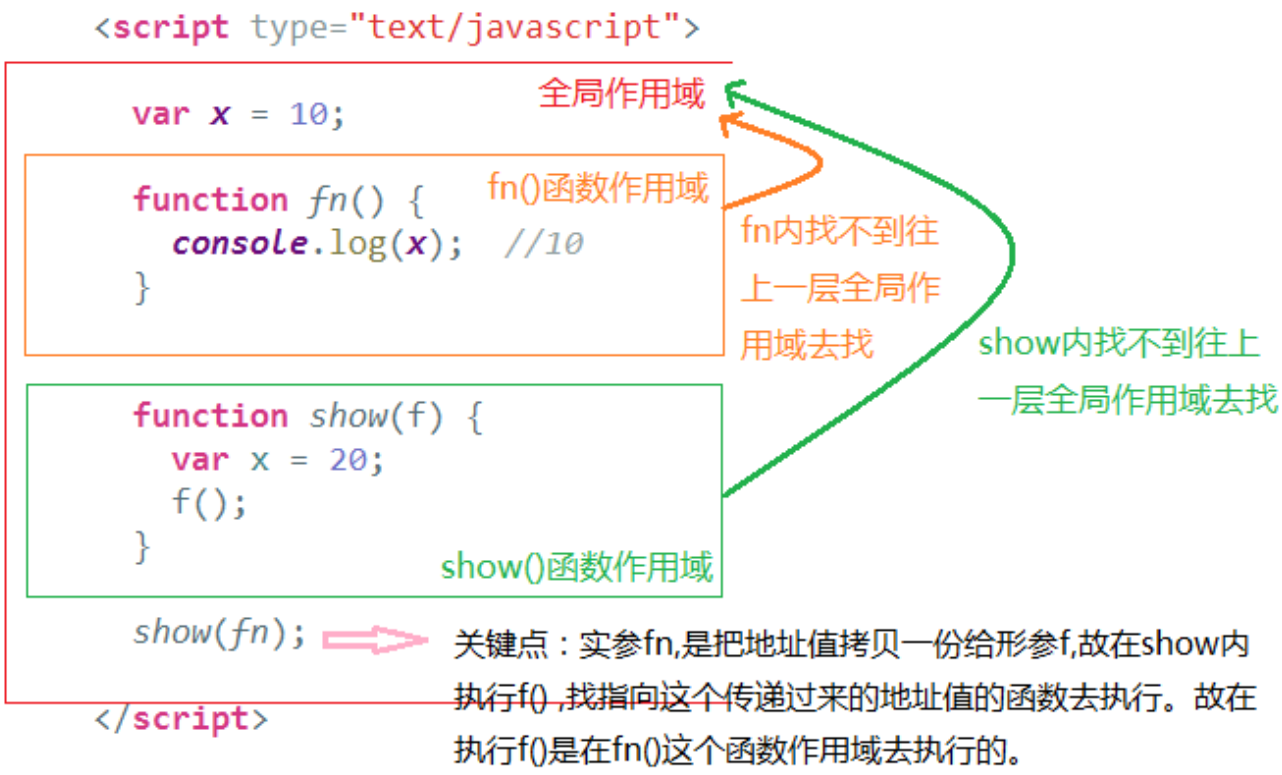
- a.在当前作用域下的执行上下文中查找对应的属性, 如果有直接返回, 否则进入b
- b.在上一级作用域的执行上下文中查找对应的属性, 如果有直接返回, 否则进入c
- c.再次执行2的相同操作, 直到全局作用域, 如果还找不到就抛出找不到的异常

3.作用域链的图解如下：



2.4.4 习题与案例

```
//面试题:作用域
<script type="text/javascript">
  var x = 10;
  function fn() {
    console.log(x);  //10
  }
  function show(f) {
    var x = 20;
    f();
  }
  show(fn);
</script>
//记住：作用域是代码一编写就确定下来了，不会改变。产生多少个作用域？n+1.n就是多少个函数，1就是指的是window。查找变量就是沿着作用域查找，而作用域是一开始就确定了，与哪里调用一点关系都没有。
//见图解：
```




```
//面试题。考察作用域与作用域链上的查找
<script type="text/javascript">
  var fn = function () {
    console.log(fn) //output: ƒ () {console.log(fn)}
  }
  fn()

  var obj = {
    fn2: function () {
     console.log(fn2) //报错，fn2 is not defined 因为首先在这个匿名函数作用域找，找不到去上一层全局找，没找到，报错。找fn2是沿着作用域查找的！
     console.log(this.fn2)//output：fn2这个函数对象  -如果要找到obj属性fn2，则用this.fn2(),让其用this这个指针指向obj，在obj这个对象中找fn2属性。
    }

  }
  obj.fn2()
</script>
```

```
//面试题。考察：连续赋值隐含的含义
(function(){
    var a = b = 3;
})();

console.log(a);  // 报错 a is not defined
console.log(b);  // 3

//理解：首先，赋值从右向左看。b = 3 因为没var 所以相当于在全局作用域中添加b,赋值为3。现在。看前面var a=的部分，a有var，那么a就是局部变量放在栈内存的封闭内存空间上。var a=b，b是变量，是基本数据类型的变量。它的内存中的内容值就是基本数据类型值，故拷贝一份给a。局部变量中a=3.
//匿名函数自执行，会有一个函数执行上下文对象。当函数执行完成，就会把执行上下文栈弹出这个上下文对象。就再也访问不到。
//所以，在函数自执行结束后，再执行输出a的语句。a压根就找不到，根本没定义。b因为是全局变量还是可以找到滴。

//注意扩展，这里只有在非严格模式下，才会把b当做全局变量。若在ES5中严格模式下，则会报错。
```

```
/*
考点：考虑返回值问题
*/
function foo1(){
    return {
        bar: "hello"
    }
}

function foo2(){
    return
    {
        bar: "hello"
    }
}

console.log(foo1()); // 返回一个对象 {bar:'hello'}
console.log(foo2());  // undefined
//因为foo2函数return 后面少了分号，在js引擎解析时，编译原理的知识可知，在词法分析会return后面默认加上分号，所以，后面那个对象压根不执行，压根不搭理。所以啊，当return ；时返回的就是undefined。
```

```
//考点：函数表达式的作用域范围
<script>
console.log(!eval(function f() {})); //false
var y = 1;
if (function f(){}){
    y += typeof f;
}
console.log(y);  // 1undefined
</script>
```

2.5 闭包

2.5.1 引入闭包

通过循环遍历加监听从而引入闭包概念。

```
//Code 1
<button>测试1</button>
<button>测试2</button>
<button>测试3</button>
<script type="text/javascript">
  var btns = document.getElementsByTagName('button')
  //遍历加监听
  for (var i = 0,length=btns.length; i < length; i++) {
    var btn = btns[i]
    btn.onclick = function () {
      alert('第'+(i+1)+'个')
    }
  }
</script>
//输出结果：不管点击哪个button，都是输出“第4个”。因为for循环一下就执行完了，可是btn.onclick是要等到用户事件触发的，故这个时候i是3.永远输出“第4个”。
//一些细节问题:在这个过程中，产生了多少个i？一个i,i是全局变量啊。
```

```
//Code 2  通过对象.属性保存i
<button>测试1</button>
<button>测试2</button>
<button>测试3</button>
<script type="text/javascript">
  var btns = document.getElementsByTagName('button')
  //遍历加监听
  for (var i = 0,length=btns.length; i < length; i++) {
    var btn = btns[i]
    //将btn所对应的下标保存在btn上（解决方式1）
    btn.index = i
    btn.onclick = function () {
      alert('第'+(this.index+1)+'个')
    }
  }
</script>
//这个时候就是我们想要的结果，点哪个ibutton 就输出第几个。
```

```
//Code 3  通过ES6的块级作用域 let
<button>测试1</button>
<button>测试2</button>

<button>测试3</button>
```

```
<script type="text/javascript">
  var btns = document.getElementsByTagName('button')
  //遍历加监听
  for (let i = 0,length=btns.length; i < length; i++) {  //(解决方式二)
    var btn = btns[i]
    btn.onclick = function () {
      alert('第'+(i+1)+'个')
    }
  }
</script>
//在ES6中引入块级作用域，使用let即可。
```

```
//Code 4  利用闭包解决
<button>测试1</button>
<button>测试2</button>
<button>测试3</button>
<script type="text/javascript">
  var btns = document.getElementsByTagName('button')
  //利用闭包
  for (var i = 0,length=btns.length; i < length; i++) {  //这里的i是全局变量
    (function (j) {  //这里的j是局部变量
      var btn = btns[j]
      btn.onclick = function () {
        alert('第'+(j+1)+'个')   //这里的j是局部变量
      }
    })(i); //这里的i是全局变量
</script>
//for循环里有两个函数，btn.click这个匿名函数就是一个闭包。它访问了外部函数的变量。
//产生几个闭包？3个闭包（外部函数执行几次就产生几个闭包）。每个闭包都有变量j,分别保存着j=0,j=1,j=2的值。故可以实现这样的效果。之前之所以出问题，是因为都是用着全局变量的i,同一个i值。
//闭包有没有被释放？没有，一直存在。我们知道闭包释放，那就是让指向内部函数的引用变量为null即可。但是此时btn.onclick一直引用这内部函数（匿名函数），故其闭包不会被释放。
//闭包应不应该被释放？不应该。因为一个页面的一个button是要一直存在的，页面显示过程中，button要一直关联着这个闭包。才能让每点击button1就alert（第1个）这样的结果。不可能让button点击了一次就失效吧。那么假设要释放这些闭包，那就让btn.onclick=null即可。
//闭包的作用？延长局部变量j的生命周期。
```

2.5.2 理解闭包（Closure）

1. 如何产生闭包?
   - 当一个嵌套的内部(子)函数引用了嵌套的外部(父)函数的变量(函数)时, 就产生了闭包。
   - 注1：若外部函数有变量b,而内部函数中没有引用b，则不会产生闭包。
2. 闭包到底是什么?

   —闭包是指有权访问另一个函数作用域中的变量的函数。
   - 使用chrome可以调试查看
   - 理解一: 闭包是嵌套的内部函数(绝大部分人)
   - 理解二: 包含被引用变量(函数)的对象(极少数人)

- 若理解二请注意: 闭包存在于嵌套的内部函数中。
- 口语：首先明白闭包的本质上是一个对象，保存在内部函数中的对象，这个对象保存着被引用的变量。
- 在后台执行环境中，闭包的作用域包含着它自己的作用域、包含函数的作用域和全局作用域。

3. 产生闭包的条件?

- 函数嵌套
- 内部函数引用了外部函数的数据(变量/函数)
- 执行外部函数（内部函数可以不执行）

```
function fn1 () {
    var a = 2
    var b = 'abc'
    function fn2 () { //执行函数定义就会产生闭包(不用调用内部函数)
      console.log(a)  //引用了外部函数变量，若里面没有引用任何的外部函数变量（函数）则不会产生闭包
    }
    // fn2() 内部函数可以不执行，也会产生闭包。只要执行了内部函数的定义就行。但若是函数表达式呢？
  }
  fn1(); //外部函数要执行哦，否则不会产生闭包
```

```
function fun1() {
    var a = 3
    var fun2 = function () {
      console.log(a)
    }
  }
  fun1()
//这样子通过函数表达式定义函数，若没有在里面调用内部函数，则不会产生闭包。
```

```
function fun1() {
    var a = 3
    var fun2 = function () {
      console.log(a)
    }
    fun2()
  }
  fun1()
//这样子通过函数表达式定义函数，但在里面调用了内部函数，那么这个情况是可以产生闭包的。
```

函数表达式不同于函数声明。函数声明要求有名字，但函数表达式不需要。没有名字的函数表达式也叫做匿名函数（anonymous function），匿名函数有时候也叫拉姆达函数，匿名函数的 name 属性是空字符串。

以上这些例子，只是辅助理解。并没有实际上的应用，下面就来说说闭包实际可以应用的地方。

2.5.3 常见的闭包

1. 将函数作为另一个函数的返回值
2. 将函数作为实参传递给另一个函数调用

```
// 1. 将函数作为另一个函数的返回值
```

```
  function fn1() {
    var a = 2
    function fn2() {
      a++
      console.log(a)
    }
    return fn2
  }
  var f = fn1()
  f() // 3
  f() // 4
//深入理解：
//问题一：有没有产生闭包？
/*An:条件一，函数的嵌套。外部函数fn1 ,内部函数fn2,条件一满足；条件二，内部函数引用了外部函数的数据(变量/函数)。a就是外部函数的数据变量。条件二满足。条件三，执行外部函数。var f = fn1() 其中的fn1()是不是执行了，外部函数执行了。赋值给f变量是因为外部函数fn1在执行后返回一个函数，用全局变量f来保存其地址值。条件三满足。综上所述，产生了闭包。*/
//问题二：产生了几个闭包？
/*产生了一个闭包。我们根据上一节的知识可知：执行函数定义就会产生闭包。那么执行函数定义是不是只要执行外部函数即可，因为外部函数一执行，就会有函数上下文对象，就会函数声明提前，也就是执行了函数定义。那么，这个时候执行了几次外部函数？是不是一次。执行了一次外部函数，也就是声明函数提前了一次，执行函数定义这个操作做了一次，故只产生了一个闭包。也可得出结论，外部函数执行几次，就产生几个闭包。跟内部函数执行几次没有关系（前提，在可以生成闭包的情况下）*/
//问题三：调用内部函数为什么可以读出a的最新值？
/*从结果可以知道，f(),f()是不是在调用了两次内部函数，从输出的结果看，a每次输出最新值。这就可以知道，在执行内部函数的时候，a并没有消失。记住这点，这就是闭包的本质作用。*/
/*问题四：那么如果我现在在以上代码最后（分别输出3,4语句后面）继续加入
  var h =fn1();
  h();
  f();
  这个时候会输出什么？*/
/*An:h()会输出3.f()会输出5. 因为：var h = fn1（）又执行了一次，h接收返回值函数对象（内部函数），也就是在这个时候产生了新的一个闭包。当调用内部函数时，h()，就会有新的函数上下文对象产生，a值就会从初始值开始记录。当调用f（）时，这个时候还是在上一个闭包的状态下，那个作用域并没有消失，故还在原先的基础上改变a值。*/
```

```
// 2. 将函数的实参传递给另一个函数调用
  function showDelay(msg, time) {
    setTimeout(function () {
      alert(msg)
    }, time)
  }
  showDelay('my name is ly', 2000)
//深入理解：
//问题一：有没有产生闭包？
```



```
/*An:条件一，函数的嵌套。外部函数showDelay ,内部函数定时器的回调函数,条件一满足；条件二，内部函数引用了
```

```
外部函数的数据(变量/函数)。msg就是外部函数的数据变量，而在回调函数中用了。注意，time不是哦，time还是在外部函数用的，在内部函数中并没有用到外部函数的time变量。是因为msg变量才满足条件二。条件三，执行外部函数。showDelay('my name is ly', 2000)执行了，外部函数执行了。但是注意回调函数没有声明提升，故还要等2000ms后触发进行调用回调函数。这个时候内部函数才执行。条件三满足（这个类似于函数表达式情况，如果是函数表达式，那么不仅仅要外部函数要执行，内部函数表达式定义的函数也要有执行，只有这样才会出现闭包。如果是函数声明定义的函数，那么就会有函数执行上下文去创建，函数提升，故在执行函数定义的时候就会出现闭包）。综上所述，产生了闭包。*/
```

2.5.4 闭包的作用

1. 使用函数内部的局部变量在函数执行完后, 仍然存活在内存中(延长了局部变量的生命周期)

本来，局部变量的生命周期是不是函数开始执行到执行完毕，局部变量就自动销毁啦。但是，通过闭包可以延长局部变量的生命周期，函数内部的局部变量可以在函数执行完成后继续存活在内存中。那就是通过闭包，具体怎样的内部机制见下。

2.让函数外部可以操作(读写)到函数内部的数据(变量/函数)

本来，函数内部是可以通过作用域链去由内向外去找数据（变量/函数），是可以访问到函数外部的。但是反过来是不行的，函数外部能访问到内部的数据（变量/函数）吗？

```
function fun1() {
   var a='hello world'
 }
 console.log(a); //报错 a is not defined
 //函数外部不能访问函数内部的数据
```

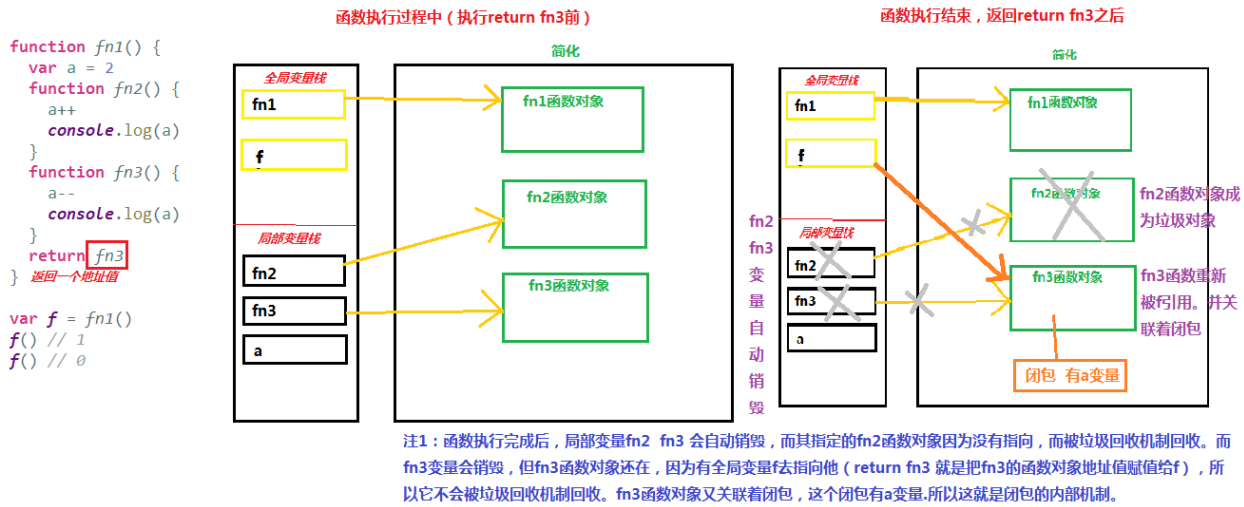
所以，函数外部不能访问函数内部的数据。但是，可以通过闭包去访问到函数内部的数据。具体的内部机制又是怎样的呢？见下。

```
//详解闭包的作用（重要）
function fn1() {
    var a = 2
    function fn2() {
      a++
      console.log(a)
    }
    function fn3() {
      a--
      console.log(a)
    }
    return fn3
  }

  var f = fn1()
  f() // 1
  f() // 0
```

问题:

1. **函数fn1（ ）执行完后, 函数内部声明的局部变量是否还存在?**

   An: 一般是不存在, 存在于闭包中的变量才可能存在。像fn2 fn3变量就自动销毁了。因为函数内的局部变量的生命周期就是函数开始执行到执行完毕的过程。那像fn2这个函数对象也会被垃圾回收机制回收，因为没有变量去引用（指向）fn2函数对象。但是fn3这个对象还在，根本原因是因为语句 var f = fn1(); fn( )执行完毕返回一个fn3的地址值并且赋值给全局变量f，那么全局变量f就会指向他，所以，这个fn3这个函数对象不会被垃圾回收机制回收。 但我在回答这个问题时，是说存在于闭包中的变量才可能存在。为什么可能呢？因为如果我把语句 var f = fn1(); 改成fn1()，这个时候还是没有变量去引用，所以这时还是会被回收的。见下图。



   那么我们发现，闭包一直会存在吗？这就是我们下一节要讲的闭包的生命周期。提前说一下，也就是fn3函数对象一直会有引用，闭包就会存在。这时我只要将f=null，这个时候fn3函数对象就没有被f引用，所以会被垃圾回收机制回收，故此时这个闭包死亡。

2. 在函数外部能直接访问函数内部的局部变量吗?**

   An: 不能, 但我们可以通过闭包让外部操作它.

2.5.5 闭包的生命周期

1. 产生: 在嵌套内部函数定义执行完时就产生了(不是在调用)--->针对的是用函数声明的内部嵌套函数。
2. 死亡: 在嵌套的内部函数成为垃圾对象时。

```
function fn1() {
    //此时闭包就已经产生了(函数提升，内部函数对象已经创建了)
    var a = 2
    function fn2 () {
      a++
      console.log(a)
    }
    return fn2
  }
  var f = fn1()
  f() // 3
  f() // 4
  f = null //闭包死亡(包含闭包的函数对象成为垃圾对象)
```

2.5.6 闭包的应用

闭包应用:

- 模块化: 封装一些数据以及操作数据的函数, 向外暴露一些行为。（本节具体讲）
- 循环遍历加监听（在2.5.1引入闭包章节有讲）
  JS框架(jQuery)大量使用了闭包

闭包的应用之一: 定义JS模块

1.要具有特定功能的js文件

2.将所有的数据和功能都封装在一个函数内部(私有的)（函数内部会有作用域与作用域链的概念，函数内部的数据就是私有的，外部访问不到。）

3.只向外暴露一个包含n个方法的对象（暴露多个行为）或函数（暴露一个行为）

4.模块的使用者, 只需要通过模块暴露的对象调用方法来实现对应的功能

自定义JS模块一：

```
//js文件
function myModule() {
  //私有数据
  var msg = 'Hello world';
  //操作数据的函数
  function doSomething() {
    console.log('doSomething() '+msg.toUpperCase())
  }
  function doOtherthing () {
    console.log('doOtherthing() '+msg.toLowerCase())
  }

  //向外暴露对象(给外部使用的方法)
  return {
    doSomething: doSomething,
    doOtherthing: doOtherthing
  }
}

//怎么使用？在html页面中
  var module = myModule()
  module.doSomething()
  module.doOtherthing()
```

自定义JS模块二：

```
//js文件
(function () {
  //私有数据
  var msg = 'My atguigu'
  //操作数据的函数

  function doSomething() {
```

```
    console.log('doSomething() '+msg.toUpperCase())
  }
  function doOtherthing () {
    console.log('doOtherthing() '+msg.toLowerCase())
  }

  //向外暴露对象(给外部使用的方法)
  window.myModule2 = {
    doSomething: doSomething,
    doOtherthing: doOtherthing
  }
})();

//怎么使用？在html页面中
  myModule2.doSomething()
  myModule2.doOtherthing()
//这种自定义模块相对而言更好，因为不需要先调用外部函数，直接使用 myModule2.doSomething()更加方便。
```

2.5.7 闭包的缺点和解决

1. 缺点
    - 函数执行完后, 函数内的局部变量没有释放, 占用内存时间会变长（是优点亦是缺点）
    - 容易造成内存泄露（注意和内存溢出的区别，见1.5.3节）
2. 解决
    - 能不用闭包就不用
    - 及时释放

```
<script type="text/javascript">
  function fn1() {
    var arr = new Array[100000]
    function fn2() {
      console.log(arr.length)
    }
    return fn2
  }
  var f = fn1()
  f()
  //这里是有闭包的，arr一直没有释放，很占内存。
  //如何解决？很简单。
  f = null //让内部函数成为垃圾对象-->回收闭包

</script>
```

- 理解:
    - 当嵌套的内部函数引用了外部函数的变量时就产生了闭包

    - 通过chrome工具得知: 闭包本质是内部函数中的一个对象, 这个对象中包含引用的变量属性
- 作用:
    - 延长局部变量的生命周期
    - 让函数外部能操作内部的局部变量
- 写一个闭包程序

```
function fn1() {
  var a = 2;
  function fn2() {
    a++;
    console.log(a);
  }
  return fn2;
}
var f = fn1();
f();
f();
```

- 闭包应用:
    - 模块化: 封装一些数据以及操作数据的函数, 向外暴露一些行为
    - 循环遍历加监听
    - JS框架(jQuery)大量使用了闭包
- 缺点:
    - 变量占用内存的时间可能会过长
    - 可能导致内存泄露
    - 解决:
        - 及时释放 : f = null; //让内部函数对象成为垃圾对象

2.6.9 习题与案例

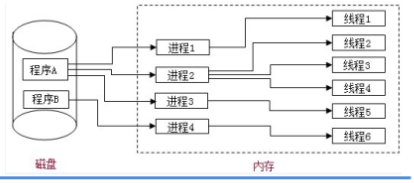
```
/*考察闭包*/
function foo(){
    var m = 1;
    return function (){
        m++;
        return m;
    }
}

var f = foo();  //这会形成一个闭包  （调用一次外部函数）
var f1 = foo();  //这会形成一个闭包 （调用一次外部函数）
/*不同闭包有不同作用域。同一个闭包可以访问其最新的值。--这句话知识一个表面现象，结合上面的案例去发现深入的步骤，这个过程是如何执行的？*/
console.log(f()); // 2
console.log(f1()); // 2
console.log(f()); // 3

console.log(f()); // 4
```

```
//考察闭包相关知识
<script>
function fun(n, o){
    console.log(o);  //实则就是输出闭包中的变量值，n是闭包引用的变量。延长的是n的变量生命周期。
    return {
        fun: function (m){
            return fun(m, n);
        }
    }
}
/*注意以上代码段是有闭包的，return fun(m,n)中的n是用到了外部函数的变量n*/

//测试一：
var a = fun(0);  // undefined
a.fun(1); // 0  执行这里实则是产生了新的闭包，但没有变量去指向这个内部函数产生的闭包故马上就消失啦。
a.fun(2); // 0
a.fun(3); // 0
/*最后三行语句一直用的闭包是fun(0)产生的闭包*/

//测试二：
var b = fun(0).fun(1).fun(2).fun(3); //undefined 0 1 2
/*产生了四个闭包，也就是外部函数fun(n，o)调用过4次。*/

// 测试三:
var c = fun(0).fun(1); // undefined 0
c.fun(2)  // 1
c.fun(3)  // 1
/*最后两行语句一直用的闭包是fun(0).fun(1)产生的闭包，故其语句*/

</script>
```

```
//考察闭包和作用域
// 写一个函数，使下面的两种调用方式都正确
console.log(sum(2,3));   // Outputs 5
console.log(sum(2)(3));  // Outputs 5
//答案：
<script>
function sum(){
    if(arguments.length == 2){
        return arguments[0] + arguments[1];
    }else if(arguments.length == 1){
        var first = arguments[0];
        return function (a){
            return first + a;
        }
    }
}
</script>
```

Part4: 事件对象与事件机制

- Part4
  - 4 线程机制与事件机制
    - 进程与线程
      - 进程(process)
        - 程序的一次执行, 它占有一片独有的内存空间
        - 可以通过windows任务管理器查看进程
      - 线程(thread)
        - 是进程内的一个独立执行单元
        - 是程序执行的一个完整流程
        - 是CPU的最小的调度单元
      - 图解

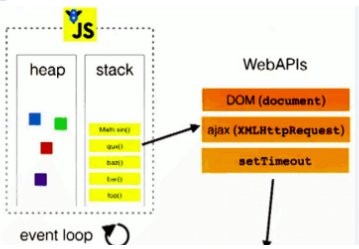


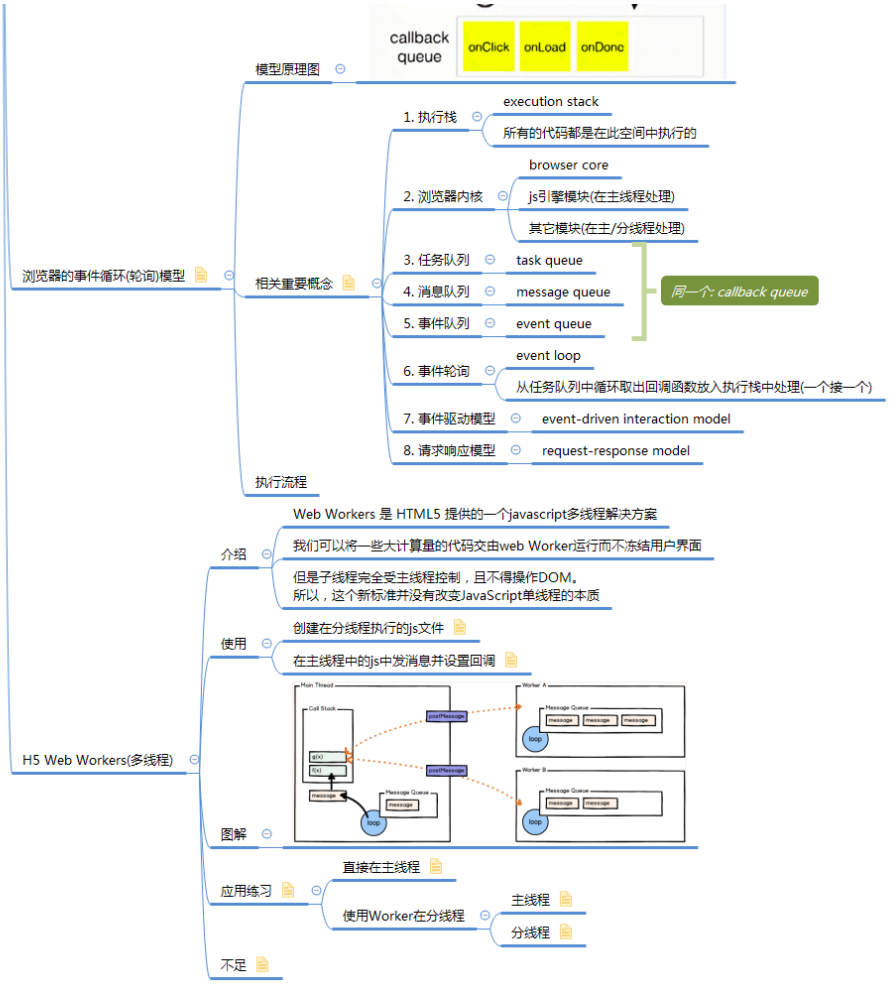
      - 相关知识
        - 应用程序必须运行在某个进程的某个线程上
        - 一个进程中至少有一个运行的线程: 主线程, 进程启动后自动创建
        - 一个进程中也可以同时运行多个线程, 我们会说程序是多线程运行的
        - 一个进程内的数据可以供其中的多个线程直接共享
        - 多个进程之间的数据是不能直接共享的
        - 线程池(thread pool): 保存多个线程对象的容器, 实现线程对象的反复利用
      - 相关问题
        - 何为多进程与多线程?
          - 多进程运行: 一应用程序可以同时启动多个实例运行
          - 多线程: 在一个进程内, 同时有多个线程运行
        - 比较单线程与多线程?
          - 多线程
            - 优点
              - 能有效提升CPU的利用率
            - 缺点
              - 创建多线程开销
              - 线程间切换开销
              - 死锁与状态同步问题
          - 单线程
            - 优点
              - 顺序编程简单易懂
            - 缺点
              - 效率低
        - JS是单线程还是多线程?
          - js是单线程运行的
          - 但使用H5中的 Web Workers可以多线程运行
        - 浏览器运行是单线程还是多线程?
          - 都是多线程运行的
        - 浏览器运行是单进程还是多进程?
          - 有的是单进程
            - firefox
            - 老版IE
          - 有的是多进程
            - chrome
            - 新版IE
          - 如何查看浏览器是否是多进程运行的呢?
            - 任务管理器-->进程
    - 浏览器内核
      - 支撑浏览器运行的最核心的程序
      - 不同的浏览器可能不一样
        - Chrome, Safari : webkit
        - firefox : Gecko
        - IE : Trident
        - 360,搜狗等国内浏览器: Trident + webkit
      - 内核由很多模块组成
        - js引擎模块 : 负责js程序的编译与运行
        - html,css文档解析模块 : 负责页面文本的解析
        - DOM/CSS模块 : 负责dom/css在内存中的相关处理
        - 布局和渲染模块 : 负责页面的布局和效果的绘制(内存中的对象)
        - ……
        - 定时器模块 : 负责定时器的管理
        - DOM事件响应模块 : 负责事件的管理
        - 网络请求模块 : 负责ajax请求

主线程

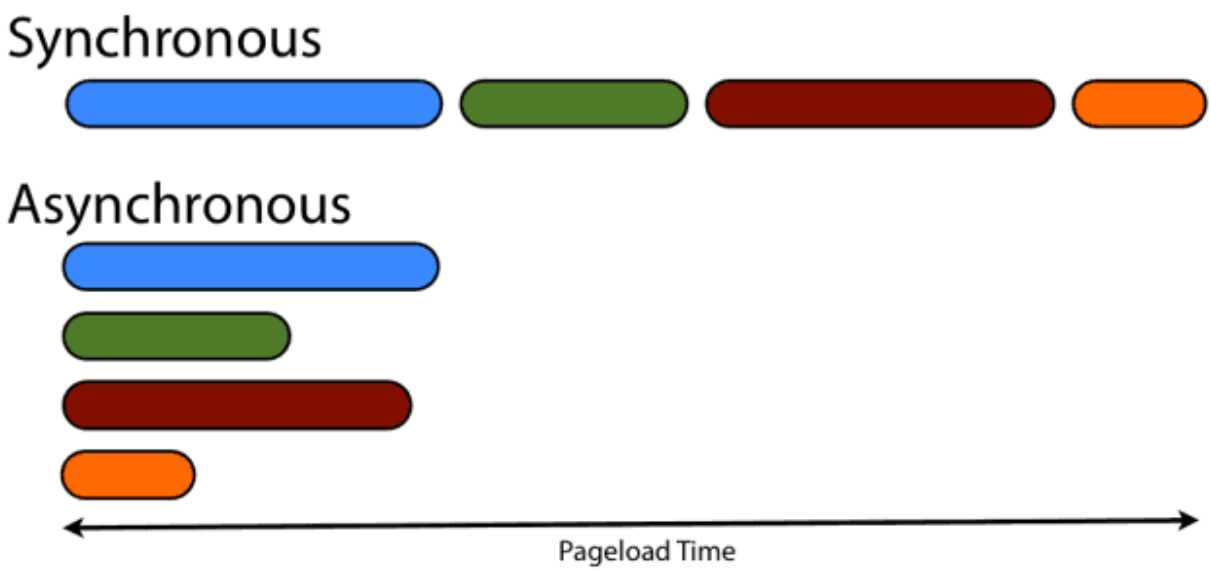
分线程

    - 定时器引发的思考
      1. 定时器真是定时执行的吗?
      2. 定时器回调函数是在哪个线程执行的?
      3. 定时器是如何实现的?
    - JS是单线程执行的
      1. 如何证明js执行是单线程的?
      2. 为什么js要用单线程模式, 而不用多线程模式?
      3. 代码的分类
      4. js引擎执行代码的基本流程





4.1 同步与异步



同步(Synchronous)：你在做一件事情，不能同时去做另外一件事。

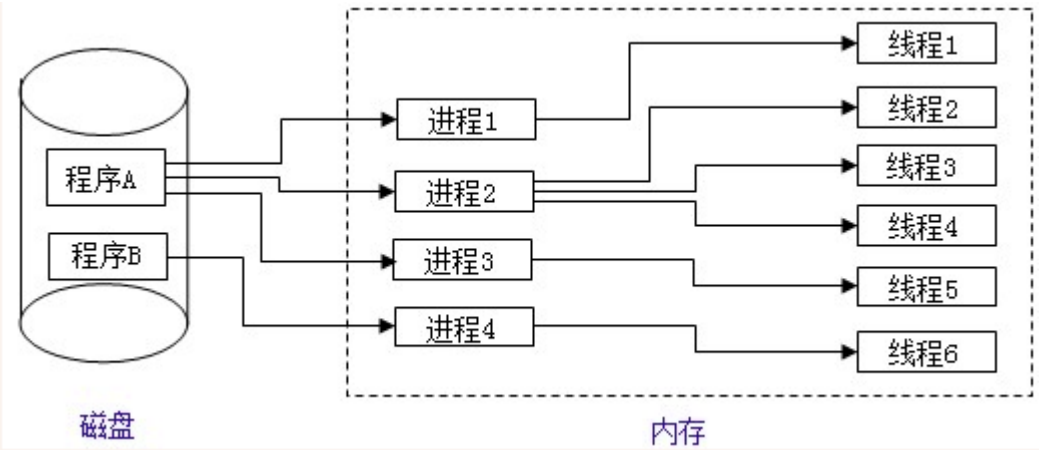
异步(Asynchronous)：你在做一件事情，这件事可能会耗时很久，而此时你可以在等待的过程中，去做另外一件事。

比如煮开水这件事吧..在这过程，你担心水沸了而不去做其它事情，就等到水沸腾，那就是同步。

而你觉得这过程耗时蛮久，可以先去做其它事情，比如去扫地，直到水沸腾。这就是异步。

4.1 线程与进程

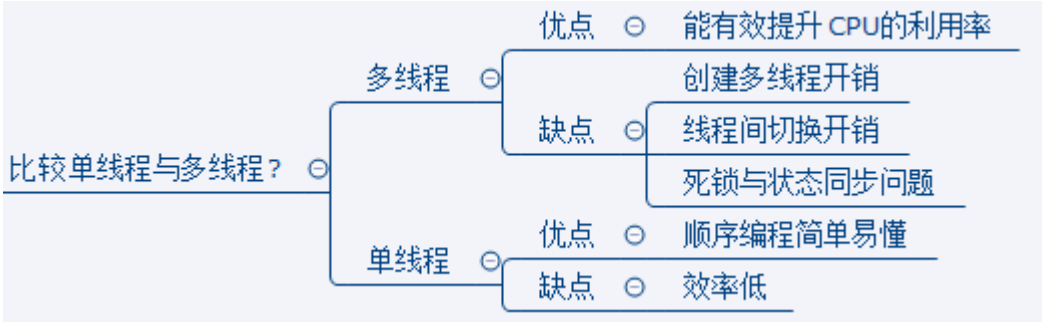
1. 进程(process)：程序的一次执行, 它占有一片独有的内存空间。可以通过windows任务管理器查看进程。
2. 线程(thread)： 是进程内的一个独立执行单元。 是CPU的最小的调度单元。 是程序执行的一个完整流程。
3. 图解：



一个程序A有多个进程，那么程序A 就是多进程的程序。程序B是只有一个进程，那么程序B就是单进程的程序。

如果一个进程有一个线程，那么这个程序就是单线程的。如果一个进程有多个线程，那么这个程序就是多线程的。单和多 线程是针对进程而言的。比如，我一个程序有两个进程，这两个进程分别有一个线程，那么这个程序还是单线程的程序。

4. 进程与线程
    - 应用程序必须运行在某个进程的某个线程上
    - 一个进程中至少有一个运行的线程: 主线程。进程启动后自动创建
    - 一个进程中也可以同时运行多个线程, 我们会说程序是多线程运行的
    - 一个进程内的数据可以供其中的多个线程直接共享
    - 多个进程之间的数据是不能直接共享的（因为进程是分配独立的内存空间给它）
    - 线程池(thread pool): 保存多个线程对象的容器, 实现线程对象的反复利用。
5. 何为多进程与多线程?
    - 多进程运行: 一个应用程序可以同时启动多个实例运行。
    - 多线程: 在一个进程内, 同时有多个线程运行
6. 比较单线程与多线程?



7. JS是单线程还是多线程?
    - js是单线程运行的

      在js设计的本意只是对一些简单的操作而已，比如提交表单用户名和密码之类的。当没有js时，那么这些数据就会提交到服务器中，那么这个数据处理将会特别大，首先假设有1000个人同时注册，那么这些请求就会到服务器上，服务器的加载量就会很大，而且，用户体验也不好，可能会延迟返回请求信息。这是如果这些操作在浏览器端来操作，那么就会简单很多。所以，js当时设计的初衷也就单线程了，因为不需要太多的操作。单线程足矣应付，而且不占太多的内存。当然后面会说道，因为他的功能（DOM操作等）也决定了它只能单线程。
    - 但使用H5中的 Web Workers可以多线程运行（主线程只有一个，要做其他的事可以启动分线程）
8. 浏览器运行是单进程还是多进程?
    - 有的是单进程
        - firefox（据Mozilla方面表示，FireFox 54版浏览器已经可以将全部打开的网页标签分为最多四个进程来运行，以此提升浏览器对PC硬件的利用率。）
        - 老版IE
    - 有的是多进程
        - chrome
        - 新版IE
9. 如何查看浏览器是否是多进程运行的呢？
    - 任务管理器==>进程
10. 浏览器运行是单线程还是多线程?
    - 都是多线程运行的

4.2 浏览器内核（Browser core）

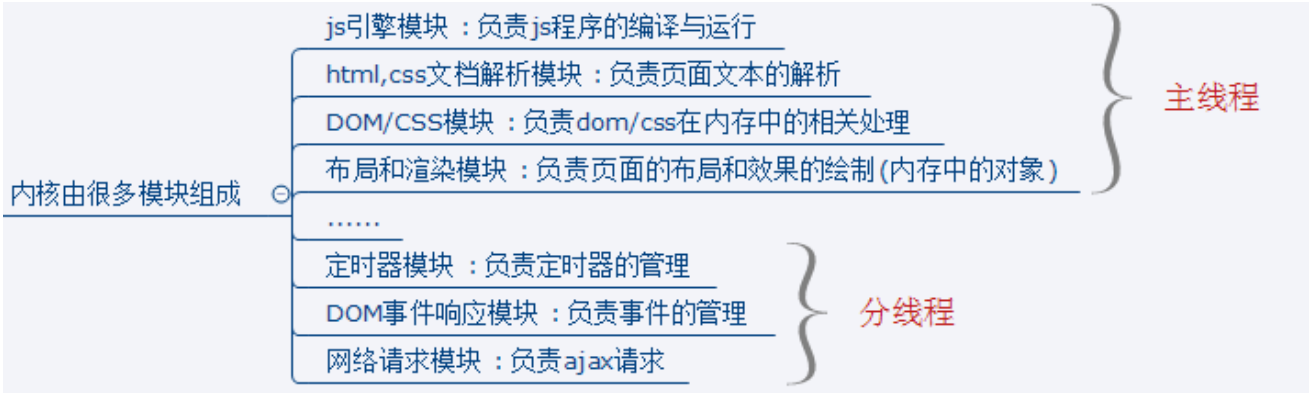
支撑浏览器运行的最核心的程序。

4.2.1 不同的浏览器可能有不同的内核

- Chrome, Safari : webkit
- firefox : Gecko
  IE : Trident
  360,搜狗等国内浏览器: Trident + webkit（双核，嘻嘻，给你一个眼神~）

4.2.2 浏览器内核由很多模块组成

- 主线程
  - js引擎模块 : 负责js程序的编译与运行（也是代码，是解释我们写的代码）
  - html,css文档解析模块 : 负责页面文本的解析（一开始是html和css文本信息）
  - DOM/CSS模块 : 负责dom/css在内存中的相关处理（把一些标签转为Dom树对象）
  - 布局和渲染模块 : 负责页面的布局和效果的绘制(参照内存中的对象数据进行布局与渲染)
- 分线程
  - 定时器模块 : 负责定时器的管理
  - DOM事件模块 : 负责事件的管理（onclick..）
  - 网络请求模块 : 负责Ajax请求



4.3 JS线程

1. 如何证明js执行是单线程的?
   - setTimeout()的回调函数是在主线程执行的
   - 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行
2. 为什么js要用单线程模式, 而不用多线程模式?
   - JavaScript的单线程，与它的用途有关。
   - 作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。
   - 如果在一个p对象（标签），假设是在多线程的环境下，那么就会有线程的切换，当一个是操作修改p的内容，另一个是删除p标签，这个时候就会有冲突。
   - 这决定了它只能是单线程，否则会带来很复杂的同步问题。
3. 代码的分类:
   - 初始化代码（同步代码）
   - 回调代码（异步代码）
4. js引擎执行代码的基本流程
   - 先执行初始化代码: 包含一些特别的代码
     - 设置定时器
     - 绑定事件监听
     - 发送ajax请求
   - 后面在某个时刻才会执行回调代码：回调函数(异步执行)

```
<script type="text/javascript">
```

```
  setTimeout(function () {
    console.log('timeout 2222')
  }, 2000)
  setTimeout(function () {
    console.log('timeout 1111')
  }, 1000)

  function fn() {
    console.log('fn()')
  }
  fn()
  console.log('alert()之前')
  alert('------') //暂停当前主线程的执行，同时暂停定时器的计时，点击确定后，恢复程序执行和计时。
  console.log('alert()之后')
</script>
```

- js是单线程执行的(回调函数也是在主线程)
- H5提出了实现多线程的方案: Web Workers
- 只能是主线程更新界面

4.4 定时器引出的问题

1. 定时器真是定时执行的吗?
    - 定时器并不能保证真正定时执行
    - 一般会延迟一丁点(可以接受), 也有可能延迟很长时间(不能接受)
2. 定时器回调函数是在分线程执行的吗?
    - 在主线程执行的, js是单线程的。不管是回调函数还是非回调函数都是在主线程执行的。
3. 定时器是如何实现的?
    - 事件循环模型(后面讲)

```
<button id="btn">启动定时器</button>
<script type="text/javascript">
  document.getElementById('btn').onclick = function () {
    var start = Date.now()
    console.log('启动定时器前...')
    setTimeout(function () {
      console.log('定时器执行了', Date.now()-start)
    }, 200)
    console.log('启动定时器后...')

    // 做一个长时间的工作
    for (var i = 0; i < 1000000000; i++) {
    }
  }
</script>
```

4.5 浏览器的事件循环(轮询)模型

4.5.1 一些简述

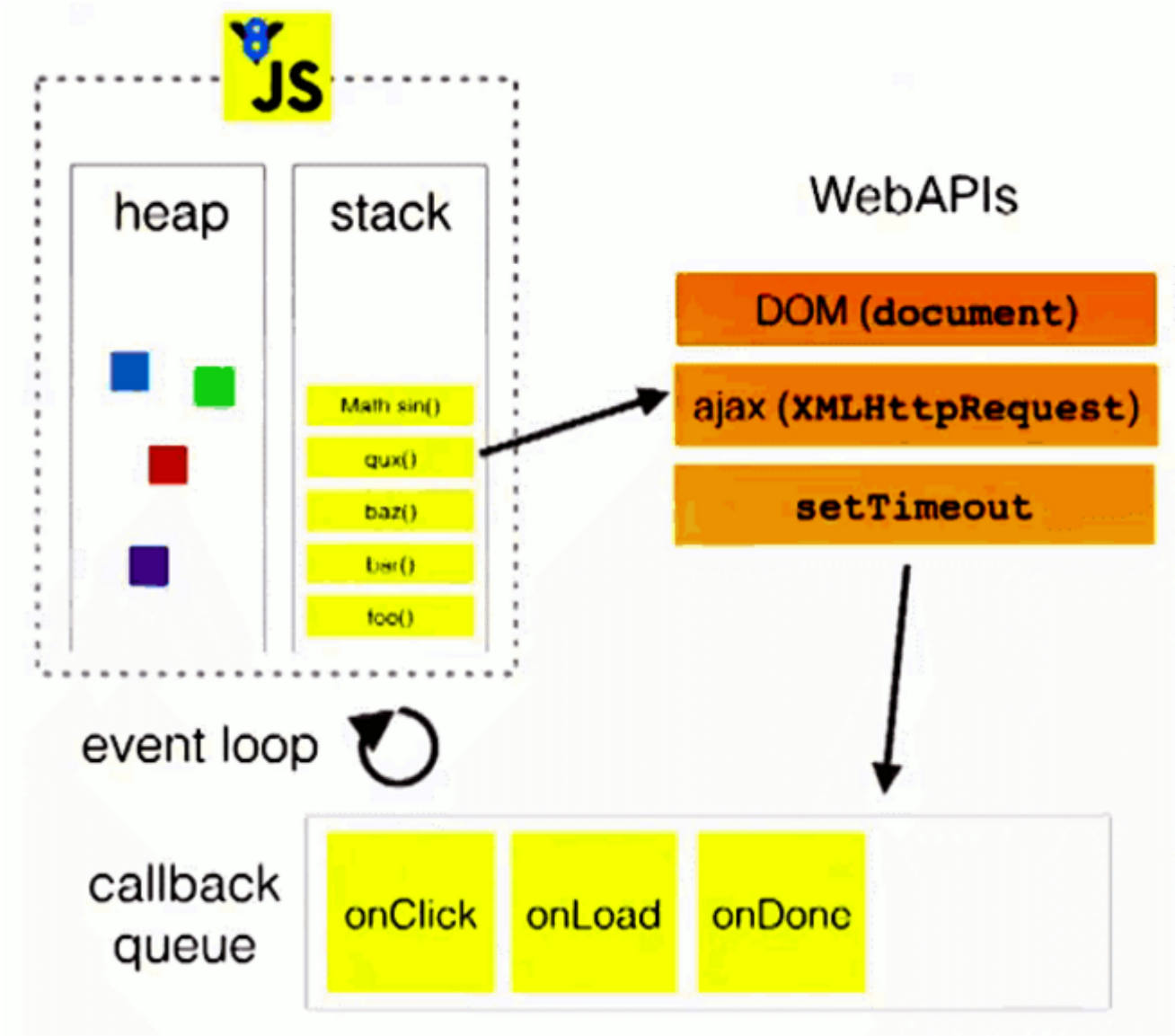
- 代码分类
  - 初始化执行代码:一些同步的代码， 包含绑定dom事件监听, 设置定时器, 发送ajax请求的代码
  - 回调执行代码: 处理回调逻辑，异步的代码（绑定dom事件监听, 设置定时器, 发送ajax请求的各自的回调函数）
- js引擎执行代码的基本流程:
  - 先执行初始化代码===>后执行回调代码
- 模型的2个重要组成部分:
  - 事件（定时器/DOM事件/AJAX）管理模块
  - 回调队列：等待去处理的回调函数。
- 模型的运转流程
  - 执行初始化代码, 将事件回调函数交给对应模块管理
  - 当事件发生时, 管理模块会将回调函数及其数据添加到回调列队中
  - 只有当初始化代码执行完后(可能要一定时间), 才会遍历读取回调队列中的回调函数执行

4.5.2 模型原理图

以下这张图就是event-driven interaction model（事件驱动模型）。

另外简单的说一下request-response model（事件响应模型），这个就相当于浏览器去服务器请求一些数据，服务器接收到这些请求，去处理这些请求，紧接着返回给浏览器的请求数据，浏览器接收到数据解析到页面上的一个过程。

现在我们主要是看一下event-driven interaction model：



首先，这个图分为三个部分:JS引擎等主线程、浏览器内核的分线程、任务队列。

在第一部分中：

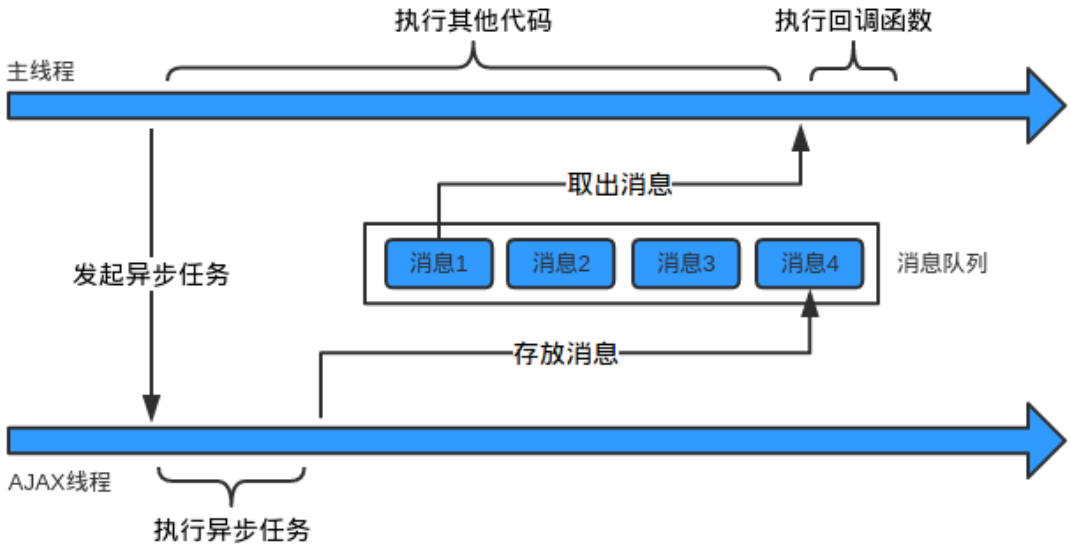
- 执行栈（execution stack）
  - 所有的代码都是在此空间中执行的
  - 各个任务按照文档定义的顺序一一推入"执行栈"中，当前一个任务执行完毕，才会开始执行下一个任务。
  - 实则是把上下文对象压栈和弹出，这里执行了一些初始化代码，包含绑定dom事件监听, 设置定时器, 发送ajax请求的代码。
  - 事件（定时器/DOM事件/AJAX）回调函数交给对应模块管理。用setTimeout做比较，他会把回调函数和延迟时间1000给WebAPIS的SetTimeout模块处理。这部分并不是在主进程中执行的，而是在浏览器分线程中执行。
- heap 用来存储声明的对象。

在第二部分中：

这一块主要是交给浏览器的分线程处理。以SetTimeout定时器为比较，他会拿到回调函数和延迟时间1000，当延迟时间过了之后，就会把回调函数推入队列中。

在第三部分中：

- 就会临时保存着回调函数，当执行栈为空时，就会依次将其回调函数压入执行栈中。
- 这个部分叫做callback queue。也叫任务队列（task queue）、消息队列（message queue）、事件队列（event queue）。指的都是同一个。
- 刚刚以定时器介绍了这个过程。我们再以AJAX为例看看是如何执行这些过程的？



上图以AJAX异步请求为例，发起异步任务后，由AJAX线程执行耗时的异步操作，而JS引擎线程继续执行堆中的其他同步任务，直到堆中的所有异步任务执行完毕。然后，从消息队列中依次按照顺序取出消息作为一个同步任务在JS引擎线程中执行，那么AJAX的回调函数就会在某一时刻被调用执行。

另外一点，我们看到事件机制模型图有事件轮询（event loop），就是从任务队列中循环取出回调函数放入执行栈中处理(一个接一个)。JS引擎线程用来执行栈中的同步任务（初始化代码），当所有同步任务（初始化代码）执行完毕后，栈被清空，然后读取消息队列中的一个待处理任务，并把相关回调函数压入栈中，单线程开始执行新的同步任务。JS引擎线程从消息队列中读取任务是不断循环的，每次栈被清空后，都会在消息队列中读取新的任务，如果没有新的任务，就会等待，直到有新的任务，这就叫事件轮询。

4.6 H5 Web Workers

4.6.1 介绍

Web Workers 是 HTML5 提供的一个javascript多线程解决方案，我们可以将一些大计算量的代码交由web Worker运行而不冻结用户界面，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

4.6.2 案例引入

实现一个斐波那契数列，在页面input中输入数列项的值，得到相应的数列值。

```
<input type="text" placeholder="数值" id="number">

<button id="btn">计算</button>
```

```
<script type="text/javascript">
  //斐波那契数列： 1 1 2 3 5 8    f(n) = f(n-1) + f(n-2)
  function fibonacci(n) {
    return n<=2 ? 1 : fibonacci(n-1) + fibonacci(n-2)  //递归调用（效率很低，时间复杂度很大）
  }
  var input = document.getElementById('number')
  document.getElementById('btn').onclick = function () {
    var number = input.value;
    var result = fibonacci(number); //主线程会一直在处理这个递归调用。导致冻结了用户界面，也就是不能操作界面了。
    alert(result)
  }
</script>
```

以上操作会在js引擎的主线程中，在计算的过程中，会冻结用户界面，达到不佳的用户体验。

4.6.3 使用Web Workers

- H5规范提供了js分线程的实现，取名为: Web Workers。它支持javascript多线程的操作。
- 相关API
  - Worker: 构造函数, 加载分线程执行的js文件
  - Worker.prototype.onmessage: 用于接收另一个线程的回调函数
  - Worker.prototype.postMessage: 向另一个线程发送消息
- 使用步骤
  - 创建在分线程执行的js文件

```
//workers.js文件

function fibonacci(n) {
  return n<=2 ? 1 : fibonacci(n-1) + fibonacci(n-2)  //递归调用
}

console.log(this)
//当接受到主线程的数据时
this.onmessage = function (event) {
  var number = event.data
  console.log('分线程接收到主线程发送的数据: '+number)
  //计算（目的：让复杂的、耗时的运算放在分线程中处理）
  var result = fibonacci(number)
  postMessage(result) //正因为可以直接使用这个方法，是因为在全局对象中有这个方法
  console.log('分线程向主线程返回数据: '+result)
  // alert(result)  alert是window的方法，在分线程不能调用。
  // 分线程中的全局对象不再是window，所以在分线程中不可能更新界面
}
```

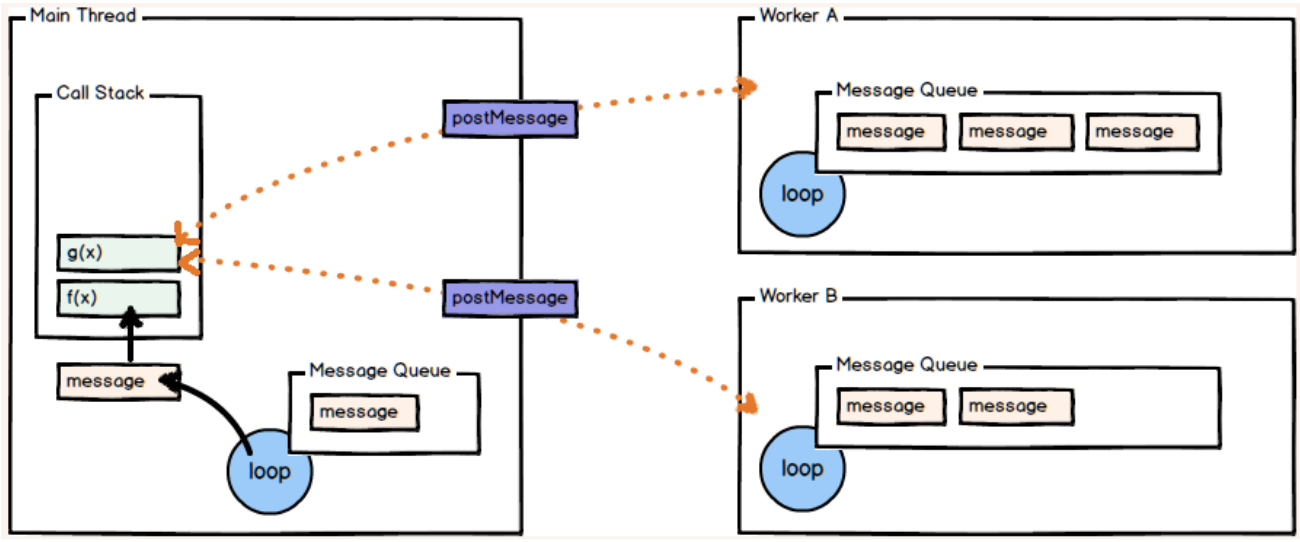
  - 在主线程中的js中发消息并设置回调

```
//主线程
```

```
<input type="text" placeholder="数值" id="number">
<button id="btn">计算</button>
<script type="text/javascript">
  var input = document.getElementById('number')
  document.getElementById('btn').onclick = function () {
    var number = input.value

    //创建一个Worker对象
    var worker = new Worker('worker.js')
    // 绑定接收消息的监听（这个位置与向分线程发送消息的代码位置可交换）
    worker.onmessage = function (event) {
      console.log('主线程接收分线程返回的数据: '+event.data)
      alert(event.data)
    }

    // 向分线程发送消息
    worker.postMessage(number)
    console.log('主线程向分线程发送数据: '+number)
  }
  // console.log(this) // window

</script>
```

回顾4.6.2的案例引入，我们可知，那个是完全在主线程中操作，带来的弊端就是冻结了用户界面。而使用Workers在分线程中处理耗时的运算，在主线程去接受计算好的数据，就可以解决直接使用主线程的冻结用户界面的弊端，这个时候不会冻结用户界面，但是子线程完全受主线程控制，且子线程不得操作DOM，因为其this不是window。

4.6.4 图解



4.6.5 不足点

1. 慢（本来在主线程肯定是更快的，现在在分线程肯定会慢点，是指这个层面上的"慢"）
2. 不能跨域加载JS

3. worker内代码不能访问DOM(更新UI)（因为分线程的this不是window）
4. 不是每个浏览器都支持这个新特性

4.7 习题与案例

```
//案例1：

console.log("1");

setTimeout(function(){
    console.log("2");
},1000);

console.log("3");

setTimeout(function(){
    console.log("4");
},0);
```

输出结果: 1->3->4->2.

分析：

1. 两个console.log()都是同步，按照文档的顺序将它们推入"执行栈"中。
2. 执行栈中的同步任务执行完毕。
3. 将两个异步任务（定时器）按照第二个参数（**延迟执行的时间**）顺序推入"任务队列"中。
4. 一一执行异步任务。

```
//案例2：

//同步code1
var t = true;

//异步code2
window.setTimeout(function (){
    t = false;
},1000);

//同步code2
while (t){}

//同步code3
alert('end');
```

分析：

1. 先执行同步code1->同步code2。
2. 此时while(true){}，进入死循环，说明这个时候栈中的同步代码永远不会执行完，也就栈永远不会清空出来，那么任务队列中的代码就不会执行。也就是任务队列中的异步的代码就无法执行。

```
//案例3：
//只有用户触发点击事件才会被推入队列中（如果点击时间小于定时器指定的时间，则先于定时器推入，否则反之）
document.querySelector("#box").onclick = function(){
  console.log("click");
};
//第一个推入队列中
setTimeout(function(){
  console.log("1");
},0);
//第三个推入队列中
setTimeout(function(){
 console.log("2");
},1000);
//第二个推入队列中
setTimeout(function(){
  console.log("3");
},0);
```

执行结果：如上面代码段中的分析。

分析：

以上都是异步代码。一定要分清哪些是异步的代码。异步代码中的回调函数都会定义在heap中，也就是在右边的堆分配一块内存给他们，这个时候根据他们指定的时候结束后，把他们的回调函数放到任务队列等待执行。

`setTimeout`的作用是在间隔一定的时间后，将回调函数插入消息队列中，等栈中的同步任务都执行完毕后，再执行。因为栈中的同步任务也会耗时，**所以间隔的时间一般会大于等于指定的时间**（指定的时间就是回调函数后面一个参数的毫秒值）。

`setTimeout(fn, 0)`的意思是，将回调函数fn**立刻插入消息队列，等待执行，而不是立即执行。**只有等待同步任务全部执行完，然后js引擎（js虚拟机）就回去从任务队列中拿出来他们去执行。

```
//案例4：
setTimeout(function() {
    console.log("a")
}, 0)

for(let i=0; i<10000; i++) {}
console.log("b")
```

执行结果：b(100) a

分析：

这个与案例3就差不多了。先执行for循环的同步代码。定时器是异步代码，先等线程的同步代码执行结束后在从任务队列中去拿这些异步代码段执行。

```
//案例5：
```

```
//执行下面这段代码，执行后，在 5s 内点击两下，过一段时间（>5s）后，再点击两下，整个过程的输出结果是什么？

//异步代码
setTimeout(function(){
    for(var i = 0; i < 100000000; i++){}
    console.log('timer a');
}, 0)
//同步代码
for(var j = 0; j < 5; j++){
    console.log(j);
}
//异步代码
setTimeout(function(){
    console.log('timer b');
}, 0)
//函数
function waitFiveSeconds(){
    var now = (new Date()).getTime();
    while(((new Date()).getTime() - now) < 5000){}
    console.log('finished waiting');
}
//异步代码
document.addEventListener('click', function(){
    console.log('click');
})
//同步代码
console.log('click begin');
//同步代码，调用函数，执行函数体
waitFiveSeconds();
```

首先，先执行同步任务。其中`waitFiveSeconds`是耗时操作，持续执行长达5s。

```
0
1
2
3
4
click begin
finished waiting
```

然后，在JS引擎线程执行的时候，'timer a'对应的定时器产生的回调、 'timer b'对应的定时器产生的回调和两次 click 对应的回调被先后放入消息队列。由于JS引擎线程空闲后，会**先查看是否有事件可执行**，接着再处理其他异步任务。因此会产生 下面的输出顺序。

```
click
click
timer a
timer b
```

最后，5s 后的两次 click 事件被放入消息队列，由于此时JS引擎线程空闲，便被立即执行了。

```
click
click
```

```
//案例6：
<script>
for (var i = 0; i < 5; i++){
    var btn = document.createElement('button');
    btn.appendChild(document.createTextNode('Button ' + i));
    btn.addEventListener('click', function (){
        console.log(i);
    });
    document.body.appendChild(btn);
}

// 1、点击 Button 4，会在控制台输出什么？ 5
/*An:不管点击哪个button都是输出5.*/
// 2. 给出一种预期的实现方式
/*  将for循环中的var 变成 let 或者 用对象.属性保存起来i的值 */
</script>
```

Part5: ES5

Part6: ES6(ES2015)

https://sagittarius-rev.gitbooks.io/understanding-ecmascript-6-zh-ver/content/

Part7: ES7(ES2016)