数据类型的分类和判断

1. 基本(值)类型
   - Number ----- 任意数值 -------- typeof
   - String ----- 任意字符串 ------ typeof
   - Boolean ---- true/false ----- typeof
   - undefined --- undefined ----- typeof/===
   - null -------- null ---------- ===
2. 对象(引用)类型
   - Object -----任意对象 -----typeof/instanceof
   - Array ------一种特别的对象(数值下标，内部数据有序)-----instanceof
   - Function ---- 一种特别的对象(可以执行)-----typeof/instanceof
3. 判断方法
   - typeof：
     可以判断：undefined、数值、字符串、布尔值、function；
     不能判断：null与object、object与array
   - instanceof：
     判断对象的具体类型
   - ===：
     可以判断undefined、null

> ==做数据转换，===不做数据转换
>
> ```
> var a;
> typeof a==='undefined'
> a===undefined
> a=null;
> typeof a==='object'
> ```
>
> 注意：typeof返回的是字符串(小写的基本类型或function)
> 用法：A instanceof B

- undefined与null的区别?（常考）
  - undefined代表定义未赋值
  - nulll定义并赋值了, 只是值为null
- 什么时候给变量赋值为null呢?
  - 初始赋值, 表明将要赋值为对象
  - 结束前, 让对象成为垃圾对象(被垃圾回收器回收)
- 严格区别变量类型与数据类型?（了解）
  - 数据的类型
    - 基本类型
    - 对象类型
  - 变量的类型(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

数据,变量, 内存的理解

1. 什么是数据?
   - 在内存中可读的, 可传递的保存了特定信息的'东东'，本质上是0101...
   - 一切皆数据, 函数也是数据
   - 在内存中的所有操作的目标: 数据
     - 算术运算
     - 逻辑运算
     - 赋值运算
     - 运行函数
   - 数据的特点：可传递，可运算

2. 什么是变量?
    - 可变化的量，变量名+变量值
    - 一个变量对应一块小内存, 变量名用来查找对应的内存，变量值是内存中保存的数据
3. 什么是内存?
    - 内存条通电后产生的存储空间(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
    - 一块内存包含2个方面的数据
        - 内部存储的数据
        - 地址值数据
    - 内存空间的分类
        - 栈空间: 全局变量和局部变量
        - 堆空间: 对象
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

相关问题：

1.var a = xxx, a内存中到底保存的是什么？

- xxx是基本数据, 保存的就是这个数据
- xxx是对象, 保存的是对象的地址值
- xxx是一个变量, 保存的xxx的内存内容(可能是基本数据, 也可能是地址值)

2.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
- 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

3.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(基本/地址值)传递
- 理解2: 可能是值传递, 也可能是引用传递(地址值)

4. JS引擎如何管理内存？

- 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
- 释放内存
    - 局部变量: 函数执行完自动释放
    - 对象: 成为垃圾对象==>垃圾回收器回收

对象的理解和使用

1. 什么是对象?
    - 多个数据(属性)的集合
    - 用来保存多个数据(属性)的容器
2. 为什么要用对象?
    - 统一管理多个数据
3. 属性组成:
    - 属性名 : 字符串(标识)
    - 属性值 : 任意类型
4. 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

5. 特别的对象
    - 数组: 属性名是0,1,2,3之类的索引
    - 函数: 可以执行的
6. 如何操作内部属性(方法)
    - .属性名
    - ['属性名']: 属性名有特殊字符/属性名是一个变量

函数的理解和使用

1. 什么是函数?
    - 用来实现特定功能的, n条语句的封装体
    - 只有函数类型的数据是可以执行的, 其它的都不可以
2. 为什么要用函数?
    - 提高复用性
    - 便于阅读交流
3. 如何定义函数?
    - 函数声明
    - 表达式
4. 如何调用(执行)函数?
    - test(): 直接调用
    - obj.test(): 通过对象调用
    - new test(): new调用
    - test.call/apply(obj): 临时让test成为obj的方法进行调用
5. 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
```

6. 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
        - 不会污染外部(全局)命名空间
        - 用它来编码js模块
7. 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
        - 定时器回调函数
        - ajax请求回调函数(后面讲解)
        - 生命周期回调函数(后面讲解)
8. 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
9. 函数中的this
    - 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
    - 如何确定this的值?
        - test(): window
        - p.test(): p
        - new test(): 新创建的对象
        - p.call(obj): obj

原型与原型链

1. 函数的prototype属性(图)
    - 每个函数都有一个prototype属性, 它默认指向一个Object空对象(即称为: 原型对象)
    - 原型对象中有一个属性constructor, 它指向函数对象
2. 给原型对象添加属性(一般都是方法)
    - 作用: 函数的所有实例对象自动拥有原型中的属性(方法)
3. 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
4. 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
    - 对象的隐式原型的值为其对应构造函数的显式原型的值
    - 原型对象即为当前实例对象的父对象
    - 程序员能直接操作显式原型, 但不能直接操作隐式原型(ES6之前)
5. 原型链
    - 别名: 隐式原型链
    - 作用: 查找对象的属性(方法)
    - 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
6. 访问一个对象的属性时，
    - 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

> 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

- 原型链的属性问题
    - 读取对象的属性值时: 会自动到原型链中查找
    - 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
    - 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
- instanceof是如何判断的?
    - 表达式: A instanceof B
    - 如果B函数的显式原型对象在A对象的原型链上, 返回true, 否则返回false
- Function是通过new自己产生的实例

注意

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,Object的原型属性默认指向Object的原型对象)
   `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)`
3. Object的原型对象是原型链尽头
   `console.log(Object.prototype.__proto__) // null`

面试题

执行上下文与执行上下文栈

> 栈: 后进先出

1. 变量提升与函数提升
   - 变量提升: 在变量定义语句之前, 就可以访问到这个变量(undefined)
   - 函数提升: 在函数定义语句之前, 就执行该函数的定义
   - 先有变量提升, 再有函数提升（函数提升的优先级更高）
2. 理解
   - 执行上下文: 由js引擎自动创建的对象, 包含对应作用域中的所有变量属性
   - 执行上下文栈: 用来管理产生的多个执行上下文
3. 执行上下文的分类:
   - 全局: window
   - 函数: 对程序员来说是透明的
4. 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
5. 包含哪些属性:
   - 全局 :
     - 用var定义的全局变量 ==>undefined
     - 使用function声明的函数 ===>function
     - this ===>window
   - 函数
     - 用var定义的局部变量 ==>undefined
     - 使用function声明的函数 ===>function
     - this ===> 调用函数的对象, 如果没有指定就是window
     - 形参变量 ===>对应实参值
     - arguments ===>实参列表的伪数组
6. 执行上下文创建和初始化的过程
   - 全局:
     1. 在执行全局代码前将window确定为全局执行上下文
     2. 对全局数据进行预处理
        - var定义的全局变量==>undefined, 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)
     3. 开始执行全局代码
   - 函数:
     1. 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)
     2. 对局部数据进行预处理
        - 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)
     3. 开始执行函数体代码

面试题

作用域与作用域链

1. 作用域:
   - 一块代码区域, 在编码时就确定了（不是调用时）, 不会再变化，与函数的执行位置无关
   - 分类:
     - 全局
     - 函数
     - js没有块作用域(在ES6之前)

    - 作用: 隔离变量, 可以在不同作用域定义同名的变量不冲突
    - 个数：n+1（n为定义的函数的个数，1为全局作用域）
2. 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
    - 作用: 查找变量

- 区别作用域与执行上下文
    - 作用域: 静态的, 编码时就确定了(不是在运行时), 一旦确定就不会变化了
    - 执行上下文: 动态的, 执行代码时动态创建, 当执行结束消失
    - 联系: 执行上下文环境是在对应的作用域中的

面试题

闭包

1. 理解:
    - 当嵌套的内部函数引用了外部函数的变量(函数)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
2. 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
3. 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();
```

4. 常见的闭包
    - 将函数作为另一个函数的返回值
    - 将函数作为实参传递给另一个函数调用
5. 闭包的生命周期
    - 产生: 在嵌套内部函数定义执行完时就产生了(不是在调用)
    - 死亡: 在嵌套的内部函数成为垃圾对象时
6. 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
7. JS模块

    - 1.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)
    - 3.只向外暴露一个包信n个方法的对象或函数
    - 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

```
//MyModule.js
function myModule() {
//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)
return {
 doSomething: doSomething,
 doOtherthing: doOtherthing
}
//html中调用
<script type="text/javascript">
var module = myModule()
module.doSomething()
module.doOtherthing()
</script>
```

```
//MyModule2.js
(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
}
})()
//html中调用
<script type="text/javascript">
myModule2.doSomething()
myModule2.doOtherthing()
</script>
```

- 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`

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, 就会抛出内存溢出的错误
2. 内存泄露
   - 占用的内存没有及时释放
   - 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
     - 没有及时清理的计时器或回调函数
     - 闭包

面试题

对象的创建模式

1. Object构造函数模式

```
var obj = {};
obj.name = 'Tom'
obj.setName = function(name){this.name=name}
```

   - 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
2. 对象字面量模式

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

   - 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
3. 构造函数模式

```
function Person(name, age) {
 this.name = name;
 this.age = age;
 this.setName = function(name){this.name=name;};
}
new Person('tom', 12);
```

   - 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
4. 构造函数+原型的组合模式

```
function Person(name, age) {
 this.name = name;
 this.age = age;
}
Person.prototype.setName = function(name){this.name=name;};
new Person('tom', 12);
```

   - 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
5. 工厂模式

```
function createPerson(name, age) {
//返回一个对象的函数===>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)
```

   - 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

继承模式

1. 原型链继承 : 得到方法

```
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
Child.prototype.test2=function(){}
var child = new Child(); //有test()
```

   - 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
   - 关键
     - 子类型的原型为父类型的一个实例对象
2. 借用构造函数 : 得到属性

```
function Parent(xxx){this.xxx = xxx}
Parent.prototype.test = function(){};
function Child(xxx,yyy){
   Parent.call(this, xxx);//借用构造函数   this.Parent(xxx)
}
var child = new Child('a', 'b');  //child.xxx为'a', 但child没有test()
```

   - 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
   - 关键:
     - 在子类型构造函数中通用call()调用父类型构造函数
3. 原型链+借用构造函数的组合继承

```
function Parent(xxx){this.xxx = xxx}
Parent.prototype.test = function(){};
function Child(xxx,yyy){
   Parent.call(this, xxx);//借用构造函数   this.Parent(xxx)
}
Child.prototype = new Parent(); //得到test()
Child.prototype.constructor = Child
var child = new Child(); //child.xxx为'a', 也有test()
```

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

- 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)

线程与进程

1. 进程:
   - 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:

    - 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
3. 关系
    - 一个进程至少有一个线程(主)
    - 程序是在某个进程中的某个线程执行的
    - 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
4. 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
5. 浏览器运行是单线程还是多线程?
    - 都是多线程运行的

浏览器内核模块组成

1. 什么是浏览器内核?
    - 支持浏览器运行的最核心的程序
2. 不同的浏览器可能不太一样
    - Chrome, Safari: webkit
    - firefox: Gecko
    - IE: Trident
    - 360,搜狗等国内浏览器: Trident + webkit
3. 内核的组成模块
    - 主线程
        - 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)
    - 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
        - 网络请求模块 : 负责Ajax请求

js线程

1. js是单线程执行的(回调函数也是在主线程)
2. 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
3. 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。这决定了它只能是单线程，否则会带来很复杂的同步问题
4. H5提出了实现多线程的方案: Web Workers
5. 只能是主线程更新界面

定时器问题:

- 定时器并不真正完全定时
- 如果在主线程执行了一个长时间的操作, 可能导致延时才处理
- 定时器是依靠事件处理机制实现的

事件处理机制(图)

- 代码分类
  - 初始化执行代码: 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
- js引擎执行代码的基本流程:
  - 初始化代码===>回调代码
- 模型的2个重要组成部分:
  - 事件管理模块(定时器/DOM事件/Ajax模块)
  - 回调队列
- 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

H5 Web Workers

- 可以让js在分线程执行
- Worker

```
var worker = new Worker('worker.js');
worker.onMessage = function(event){event.data} : 用来接收另一个线程发送过来的数据的回调
worker.postMessage(data1) : 向另一个线程发送数据
```

- 问题:
  - worker内代码不能操作DOM更新UI（因为全局对象不是window了）
  - 不是每个浏览器都支持这个新特性
  - 不能跨域加载JS
  - 慢