# 内置类型

ECMAScript 内置 6 种，Undefined, null ,boolean string number 和 object (symbol) 是 es6 新增， 其中 Object 是引用类型，其他是基本类型，划分依据是是否可以表示为固定长度。 typeof 判断类型，注意: `typeof null === 'object'` , `typeof function a(){} === 'function'`

```
function isNull(any){
    return !any && typeof any === 'object'
}
```

Array 是一种容器类型，可以容纳任何类型，包括自己

Number 类型，js 明确使用双精度格式，也就是 64 位二进制 0.1 + 0.2 !== 0.3 ,常见做法设置容差

```
Math.abs((0.1+0.2) - 0.3) < Number.EPSILON
```

## js 编译器步骤

- JS 源代码经过词法分析，转成 tokens
- tokens 经过语义分析，转化成 AST 抽象语法树
- 抽象语法树转化为字节码
- 字节码，机器运行

## new 的原理

https://github.com/azl397985856/leetcode

new 的原理

- 创建一个空对象 obj
- 将该对象 obj 的原型链 **proto** 指向构造函数的原型 prototype 并且在原型链**proto** 上设置 构造函数 constructor 为要实例化的 Fn
- 传入参数，并让构造函数 Fn 改变指向到 obj , 并执行
- 如果 Fn 执行后返回的是对象类型（null 除外），则返回该对象， 否则返回 obj

```
function New (Fn){
    let obj = {}
    let arg = Array.prototype.slice.call(arguments, 1)
    obj.__proto__ = Fn.prototype
    obj.__proto__.constructor = Fn
    let ret = Fn.apply(obj, arg) // 默认return this
    return typeof ret === 'object' ? ret || obj: obj
}

<!-- 例子 -->
function Parent(age, name){
```

```
    this.age = age
    this.name = name
    this.sayAge = function(){
        console.log('=======this.age======', this.age)
    }
}
Parent.prototype.sayName = function(){
    console.log('=======this.name======', this.name)
}

let son = new Parent(18,'xx')
let son2 = New(Parent, 18, 'zz')
```

## this 设计的目的就是指向函数运行时所在的环境

对象的属性可能是一个函数，当引擎遇到对象属性是函数的情况，会将函数单独保存在堆中，然后再将函数的地址赋值给对象属性；而 Javascript 是允许在函数体内引用当前环境的其他变量，那么问题来了，函数可以在不同的运行环境执行，所以我们就需要一种机制，能够在函数内获得当前运行环境，由此诞生了 this，它的设计目的就是指向函数运行时所在的环境。

let LRUcache = function(level){ this.level = level this.stack = [] this.secretKey ={} }

L.prototype.get = function(key){ // 使用 if(key in this.secretKey){ this.stack.splice(this.stack.indexOf(key), 1) this.stack.unshift(key) return this.secretKey[key] } return -1 } L.prototype.put = function(key){ if(key in this.secretKey){ this.stack.splice(this.stack.indexOf(key), 1) this.stack.unshift(key) this.secretKey[key] = value } else if (this.stack.length < this.level){ this.secretKey[key] = value this.stack.unshift(key) }else { delete this.secretKey[this.stack[this.level -1]] this.secretKey[key] = value this.stack.pop() this.stack.unshift(key) } }

class LRUcache { constructor(level){ this.cache = new Map() this.level = level } get(k){ if(!this.cache.has(k)) { return -1 } const v = this.cache.get(k) this.cache.delete(k) this.cache.set(k, v) return v } put(k, v){ if(this.cache.has(k)) this.cache.delete(k) this.cache.set(k, v) if(this.cache.size > this.level){ this.cache.delete(this.cache.keys().next().value) } } }

class l { constructor (level){ this.cache = new Map() this.level = level }, get(key){ if(!this.cache.has(k)){ return -1 } const v = this.cache.get(k) this.cache.delete(k) this.cache.set(k, v) return v }, put(k, v){ if(this.cache.has(k)) this.cache.delete(k) this.cache.set(k ,v) if(this.cache.size > this.level){ this.cache.delete(this.cache.keys().next().value) } }

}