2. 对象(Object)

普通对象
- API:
- 键名会被转换为字符串
- 成员访问
- 假删除
- 真删除
- arguments
- 全局对象方法

标准特殊对象
- 数组(Array)or类数组
- 正则对象(RegExp)
- 日期对象(Date)
- 错误对象(Error)
- 数字对象(Math)
- (ArrayBuffer)
- DataView
- Set
- Map

非标准特殊对象
- Number
- Boolean
- String
- Symbol
- BigInt

函数(function)
- **函数创建过程**
- **函数执行过程**
- **this指向问题**
- **事件绑定**
- **回调函数**
- 函数形参默认值
- **数据类型检测**
- 1. typeof [value]
- 2. [value] instanceof [constructor]
- 3. [value].constructor === constructor
- 4. Object.prototype.toString.call([value])
- 数据类型相等判断:
- 需要注意的点:

普通对象

API:

Object.getOwnPropertySymbols(obj)

获取对象的Symbol属性值

Object.keys(obj)

Object.getOwnPropertyNames(obj)

获取对象的非Symbol私有属性

Object.assign([obj1],[obj2])

对象浅合并,返回合并后的参数1，参数2不变

键名会被转换为字符串

```
let user = {
    [{name: 'link'}] : 'chen'
  }
```

```
    // 输出 -> [object Object]: 'chen'
    // 即 ({name: 'link'}).toString()
```

成员访问

1. 对象.属性名
    1. 这种方式不适用于`数字类型`或`Symbol类型`

```
let user = {
    name: 'link'
    0: 1
  }
  console.log(user.0) // -> 报错
```

2.对象[属性名]

- 对于字符串键名 访问方式为
    - 需要指定好属性的类型:
    → obj[key] 和 obj['key'] 不是同一个情况

```
let user = {
    name: 'link'
  }
  console.log(user['name']) // -> link
```

变量与键是不同的

1. 变量仅仅代表的是一个数据的名称
2. 键本身是可以代表一种数据类型如(`number`,`string`,`symbol`), 也是一个具体的值, 是对当前特征的描述

```
var x = 10
      let obj = {
        x: 'lin',
        10: 100
      }
      console.log(obj['x']) // lin *
      console.log(obj.x) // lin
      console.log(obj[x]) //100 **
```

`obj['x']`指的是访问x属性下的值, 其中的'x'代表的是**键名**.

`obj[x]` 中的x表示的事变量x 即**全局下的 x** 该语句会通过x的值去寻找**值**

> obj[值]：则一定是成员访问, 而obj[变量] 则是将变量的值作为键进行成员访问

假删除

> 虽然访问值为undefined 但是该键值对仍然存在

```
let user = {
  name: 'link'
}
user.name = undefined
console.log(user) // -> user.name = undefined
```

真删除

```
let obj = {
        x: 'lin',
        10: 100,
        [xx]: '1000'
      }
      delete obj['x']
```

arguments

函数内置实参集合:

1. arguments是一个类数组
2. 当不确定函数使用时需要传递多少个实参, 就可以使用arguments来获取实参集合

```
function fn() {
        console.log(arguments)
        console.log(...arguments) //直接拿到实参
      }
      fn()
      fn(10)
      fn(10, 20)
      fn(10, 20, 30)
      // 剩余运算符
      function fn(...params) {
```

```
    console.log(params) // 这样输出的是一个数组

}
fn()
fn(10)
fn(10, 20)
fn(10, 20, 30)
```

全局对象方法

1. eval([String])

> 作用: 可以将字符串转换为有效的JS表达式
> 实参: String
> 返回: 表达式执行结果

标准特殊对象

数组(Array)or类数组

数组本身的工具类方法

console.dir(Array)

1. Array.from([value])
    1. 把一个类数组集合或者set结构转换为数组,如: 节点集合/元素集合/arguments/.
    2. 只有数组才能调用Array.prototype上的方法
2. Array.isArray([value])
    1. 检测当前值是否为一个数组

供数组调用的方法

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7f299893-a585-4e54-8d30-400deaa49456/ArrayMathStringAPI.xmind

😃所有常用的API都在这里

PDF:

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/40de39cc-0503-43ce-9a17-41f5629e87ff/.pdf

正则对象(RegExp)

日期对象(Date)

错误对象(Error)

数字对象(Math)

(ArrayBuffer)

DataView

Set

Map

非标准特殊对象

> 基于构造函数或者Object 创造出来的原始值对象类型的格式信息,类型属于对象类型

基于**new Object**产生的值:

```
const n = new Object(true)
      console.log(n) // Boolean {true}
      console.log(typeof n) // 'object'

      const n = new Number('a')
      console.log(n) // Number {NaN}
      console.log(typeof n) // 'object'
```

Number

Boolean

String

API:

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0a224b1f-75e8-447a-b812-4a6cb6c9a11f/API.pdf

Symbol

BigInt

函数(function)

函数创建过程

函数是一个特殊的引用类型, 但是本质上也是对象, 同样是在堆内存中开辟一块内存, 存放数据.

1. 开辟一个的堆内存, 有一个16进制的内存用于地址指向
2. 存储内容: 在代码执行前, 上下文的代码会以代码字符串的形式存储在对应的堆内存中. 当做普通对象,也可以存放一些键值对(name, prototype) 如下:
3. 创建 `[[scope]]` 声明作用域, (函数创建所在的执行上下文)
4. 把堆内存地址放到栈中, 供变量指向

函数执行过程

1. 形成自己的执行上下文EC(..), 进站执行 创建AO

2. 初始化作用域链
    初始化this指向
    初始化`arguments`实参集合
    形参赋值 (形参是函数的私有变量, 包括在该执行上下文中声明的变量也是私有变量)
    变量提升

3. 代码执行

4. 根据情况, 决定当前上下文中的私有上下文是否要出栈释放(闭包)

函数的每次执行都是相对独立的, 都有各自的执行上下文, 但是在一些特定的情况下, 也可以被关联在一起.

this指向问题

正常的普通函数执行：看函数执行前是否有“点”，有，“点”前面是谁this就是谁，没有“点”，this是window「严格模式下是undefined」

THIS: 指向的是函数的调用者(谁执行的函数)

何种情况下会初始化THIS指向

1. 事件绑定
2. 构造函数
3. 执行函数(普通函数, 成员访问, 匿名函数, 回调函数)
4. 箭头函数(generator函数)
5. 基于call/apply/bind 改变this指向

在全局下执行的函数, this指向window(`use strict` 模式下undefined)
块级上下文中没有this指向, 其中的this全部继承自上级上下文的this(箭头函数也是)

事件绑定

```
<button id="btn">click</button>

// Dom0 :
btn.onclick = function () {         //  (*)
  console.log(this) // 当前节点
}
// Dom2 :
btn.addEventListener('click',function() { //  (**)
  console.log(this) // 当前节点
})
btn.attachEvent('click', function () {      // (***)
  console.log(this) // window/undefined
})
```

当按钮被点击后, 函数执行时, `(*)`和`(**)` 都是指向当前节点对象, `(***)`而兼容IE678的方法 则指向window对象, 严格模式下指向为undefined

回调函数

```
var i = 1
function fn(callback) {
  var i = 2
  // callback -> 匿名函数
  callback()
}
fn(function () {
  console.log(this.i) // 1 this指向了window
})

arr.forEach(
  function (item, index) {
    console.log(this) //->forEach第二个参数「对象」  forEach内部做处理了
  },
  { xxx: 'xxx' }
)
```

函数形参默认值

总结: 尽量不要在函数中使用形参默认值, 容易造成各种各样的问题

作用机制: 如果当前函数使用了形参默认值(*无论是否生效*), 并且函数中有基于let/const/var声明的变量(*无论变量名是否与形参一致*),则在函数执行时除了生产一个私有的上下文,还会在`{}`块内生产一个块级上下文(*该块级作用域链指向函数的上下文*)

注意: (如果函数体中的变量名与形参一致,则最开始会把形参变量的值同步一份给同名的私有变量)

```
var x = 1;
function func(x, y = function anonymous1(){x = 2}){
    /*
        0x000
        [[scope]] : EC(func),EC(G)
        形参赋值:
```

```
        x : 5 => 2
        y = 0x001 [[scope]] EC(func)
    */
   /*
        0x002
        [[scope]] : EC(block),EC(func)
        变量提升:
        var x ;
        ------
        AO:
        x : 5 => 3
   */
  /*
     0x001
     [[scope]]: EC(y),EC(func)
     AO:
     代码执行
     x = 2

  */
    var x = 3;
    y();
    console.log(x); // => 3
}
func(5)
console.log(x); // 1
```

解决办法:

```
function foo (x) {
  if (typeof x === 'undefined') {
    ...  // 对形参做处理
  }
}
```

数据类型检测

1. typeof [value]

```
> typeof {}
"object"
> typeof {xx:'xx'}
"object"
> typeof [10,20]
"object"
> typeof /^$/
"object"
> typeof new Date()
"object"
> new Date()
Wed Sep 16 2020 11:01:17 GMT+0800 (中国标准时间)
> dir(new Date())
  ▶Wed Sep 16 2020 11:01:32 GMT+0800 (中国标准时间)
undefined
> typeof function(){}
"function"
```

1. typeof 返回的是一个字符串:
    1. 'number'/'string'/'boolean'/'bigInt'/'undefined'/'function'/'object'/'symbol'
2. 局限性:
    1. typeof null → 'object'
    2. 对于特殊对象无法检测 如: 正则/日期/ 数组... ⇒ 'object'
3. 注意点:
    1. 无论是构造函数, 生成器函数, 箭头函数, typeof都是返回 `function`
    2. typeof [value] 该变量未声明也不会报错,而是返回 `undefined`

即typeof 无法检测**细分对象**和**null**

typeof 检测原理:

1. 所有的数据类型在计算机中都是按二进制进行存储的, 而`typeof`是通过检测二进制值来区分数据类型的.
2. 由于`对象`的二进制值都是以000开头的, 而`null`因为全部位都为0,所以`对象`与`null`检测出来都为**object**
3. `function`作为一种特殊的对象, 有他自己的特殊的属性(实现了`call`方法), JavaScript设计者认为有必要区分`object`与`function`, 并无设计缺陷上的问题.

检测方式:

在底层有一个`C++`提供的办法 **getValue(val)** 它按照存储值去检测数据属于什么类型

typeof 检测的速度快且准确, 但只能用于检测原始值类型(除null)

1. 对象 000 → 实现call，则返回'function'，没实现call返回'object'
2. null 000000
3. undefined -2^30
4. 数字 → 整数1 浮点数010
5. 字符串 100
6. 布尔 110

2. [value] instanceof [constructor]

检测某个实例是否属于哪个类

instanceof **检测原理:**

1. 如构造函数存在[Symbol.hasInstanceof]()方法, 则返回该方法的检测结果.
2. 通过obj的原型链(__proto__),判断构造函数Object的原型是否在obj的原型链上, 如果存在,则返回true;

```
obj instanceof Object // ture
```

缺点: 无法识别原始值类型, 且由于prototype是可以被修改的, 所以这个方法也不一定准确

检测实例是否属于某个类

3. [value].constructor === constructor

获取构造函数

访问实例的构造函数, 看看是否与目标的函数相等.

缺点:

同样的,构造函数指向可以被修改.

4. Object.prototype.toString.call([value])

toString()方法正常情况下, 使用来字符化数值的, 但是在顶级对象Object的toString方法, 可以用来检测数据类型, 但不可否认,**这是最为准确,且没有弊端的方法**

1. 查看[value]上的[Symbol.toStringTag]属性, 如果存在, 则返回"[object ?]", 如果不存在,则返回构造函数的信息

```
let class2type = {}
  toString = {}.prototype.toString // 简写
```

也有一些特殊的检测方法如:

1. isNaN()
2. Array.isArray()
3. isFinite()
3. Object.is(,)

数据类型相等判断:

等号	名称	区别
`==`	相等	相当于'绝对相等', 浏览器在判断两者时,如数据类型不同会进行**隐式转换**(转为数字类型)
`===`	绝对相等	要求两边**值**和**数据类型**都相等

需要注意的点:

> 1. 尽量使用三等号进行判断
> 2. 字符串==对象时, 会把对象转换为字符串再进行比较
> 3. 在其他情况下,都将其他数据类型转换为数字进行比较

`null` 和 `undefined` 与任何值都不相等, 仅在双等号下与彼此相等

```
NaN == NaN  // => false 因为非有效数字可以是很多种值, 所以不相等
  null == undefined // => true
  null === undefined // => false
```