1. **原始类型：string，number，boolean，null，undifined，symbol**
2. **Type of和instanceof**

**判断原始类型用type of比较好**

typeof 1 // 'number'

typeof '1' // 'string'

typeof undefined // 'undefined'

typeof true // 'boolean'

typeof Symbol() // 'symbol'

**判断对象类型用instanceof比较好**

const Person = function() {}

const p1 = new Person()

p1 instanceof Person // true

var str = 'hello world'

str instanceof String // false

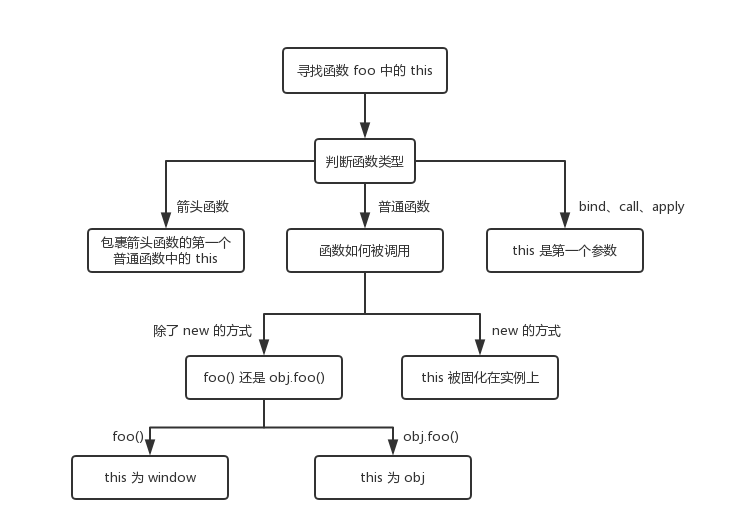
var str1 = new String('hello world')

str1 instanceof String // true

1. **类型转换**



1. **this的指向**



1. **闭包**

闭包不一定非要返回一个函数，只要是在A函数中定义了函数B，并且B中使用到了A中的变量，B就是个闭包。

为什么会有闭包？说到底是作用域链，由于闭包函数对自由变量的需求导致应该被销毁的上下文保留了下来，从而常驻内存，产生了闭包。**闭包所需求的上下文环境不会自动销毁，需要手动赋值闭包依赖变量所在函数为null才能清除对应的上下文。**

1. **==**

对于 == 来说，如果对比双方的类型****不一样****的话，就会进行****类型转换****

假如我们需要对比x和y是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否相同。相同的话就是比大小了
2. 类型不相同的话，那么就会进行类型转换
3. 会先判断是否在对比null和undefined，是的话就会返回true
4. 判断两者类型是否为string和number，是的话就会将字符串转换为number
5. 判断其中一方是否为boolean是的话就会把boolean转为number再进行判断
6. 判断其中一方是否为object且另一方为string、number或者symbol，是的话就会把object转为原始类型再进行判断
7. **深浅拷贝**

**浅拷贝：**

1. **object.assign()**

let a = {

  age: 1

}

let b = Object.assign({}, a)

a.age = 2

console.log(b.age) // 1

1. **ES6三点运算符**

let a = {

  age: 1

}

let b = { ...a }

a.age = 2

console.log(b.age) // 1

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，就需要深拷贝登场了

**深拷贝：**

**1.JSON.parse(JSON.stringify(object)) 有局限性**

**2.lodash的深拷贝函数**

**8.原型**

1.一切（引用类型）都是对象，对象是属性的集合

2.对象都是通过函数创建的

3.每个函数都有一个属性叫做prototype，这个prototype的属性值是一个对象（属性的集合，再次强调！），默认的只有一个叫做constructor的属性，指向这个函数本身。

4.每个函数function都有一个prototype，同时每个对象都有一个\_\_proto\_\_，且每个对象的\_\_proto\_\_属性，指向创建该对象的函数的prototype

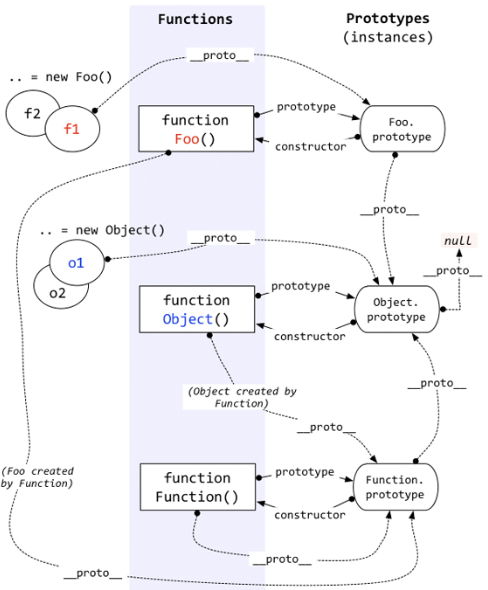
5.Object.prototype确实一个特例——它的\_\_proto\_\_指向的是null

6.函数也是一种对象：

例：var fn=new Function(“x”,”y”,”return x+y;”);

Console.log(fn(5,6));

7.Function的\_proto\_指向自身的prototype



**总结：**

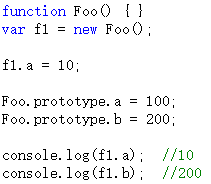
1. **\_proto\_一定是指向prototype的（Object.prototype的\_proto\_除外，指向null）**
2. **对象是没有prototype属性的，函数才有**
3. **对象的\_proto\_指向创建该对象的函数的prototype**
4. **所有函数（除Object之外）的prototype的\_proto\_指向Object的prototype**
5. **所有函数（包含Function）自身的\_proto\_指向Function的prototype**
6. **Instanceof的原理：**

**A instanceof B**

**A.\_proto\_.\_proto\_.\_proto\_...一直往下挖，直到发现其等于B.Prototype**

**返回true，否则返回false**

访问一个对象的属性（键值对和方法）时，先在基本属性中查找，如果没有，再沿着\_\_proto\_\_这条链向上找，这就是**原型链**。



<https://www.cnblogs.com/wangfupeng1988/p/3977987.html>

**9.var、let、const的区别**

1.函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶部

2.var存在提升，我们能在声明之前使用。let、const 因为暂时性死区的原因，不能在声明前使用

3.var在全局作用域下声明变量会导致变量挂载在 window 上，其他两者不会

4.let和const作用基本一致，但是后者声明的变量不能再次赋值

**10.原型继承和class继承**

**11.模块化**

**12.proxy**

**13.map,filter,reduce**

**14.generator**

**15.promise**

固定格式：

const promise = new Promise(function(resolve, reject) {

  // ... some code

  if (/\* 异步操作成功 \*/){

    resolve(value);

  } else {

    reject(error);

  }});

promise.then(function(value) {

   success},

//function(error) {

  // failure}

);

Then中的第二个参数是可以省略的，一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），总是使用catch方法

promise

.then(function(data) {})

.catch(function(err) {});

Promise使用步骤：

1. 声明一个promise对象，其中包含了一个函数，该函数具有
2. 在promise体中声明包含两个形参resolve和reject的函数，分别用以处理异步请求的成功和失败。
3. 调用promise的then和catch，分别向他们传入回调函数，处理异步操作后返回的成功和失败的结果

Promise的链式调用：

getJSON("/posts.json").then(function(json) {

return json.post;}).then(function(post) {

// ...});

注★：后面每一个then获得的resolve和reject参数都是在、上一个then中的resolve和reject函数的返回值

Promise.all()方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.all([p1, p2, p3]).then(x(),y());

只有当p1,p2,p3都成功才会调用x（），否则调用y（），且是将p1,p2,p3执行完后将他们各自resolve或者reject的参数组合成数组传递给x（）或者y（）。

比较不错的例子：<https://blog.csdn.net/u012045958/article/details/80635151>

**16.async和await**