1. 原型链和原型链的继承

函数有一个prototype属性，存储函数及其实例的所有公共方法和属性，prototype上还有一个指向其构造函数的指针constructor，每个实例对象上有一个__proro__属性，指向其父类的prototype

- 每个实例对象都有一个私有属性__proto__指向它的构造函数的 原型对象prototype，该原型对象也有一个__proto__,层层向上，直到一个对象的原型对象为null，null没有自己的原型对象，并作为原型链中的最后一个环节
- 只有函数有prototype，对象是没有的
- 函数也有__proto_-属性，因为函数也是对象，函数的__proto__指向Function.prototype 也就是说普通函数是Function这个构造函数的一个实例

我们查找一个实例的属性和方法，首先在实例上查找，如果没有通过__proto__去父类的prototype上查找，一直查找直到找到null则

Object.prototype.**proto** === null Object的原型对象作为实例的构造函数是null

- 创建对象的三种方式

1. 字面量创建 var a = {name: 'mimi'}或var a = new Object({name: 'mimi'})
2. 构造函数 function A(name){this.name = name} var a = new A('mimi') a => {name: 'mimi'}
3. Object.create: Object.create(prototype, propertiesObject)第一个参数是新创建对象的原型对象，第二个参数是可选的 var p = {name: 'p'} var a = Object.create('p', { age: { value:1, writable: true } })

2.instanceof 可以检测数据类型 但是只能检测new的基本类型，引用类型也可以，null和undefined不可以

原理：判断实例对象的__proto__是不是指向构造函数的prototype

3. es6之前使用prototype实现继承

1. 原型链继承(**思想：一个类的原型对象等于另一个类(被继承类)的实例**) function Super() {} Super.prototype.a = function(){} function Sub() {} Sub.prototype = new Super() 问题：原型对象上的所有属性都是实例共享的，如果原型对象中存储一个引用类型的属性如colors:[],那么一个实例修改这个属性，其他的实例也会受到影响

因此，我们将属性放在构造函数中而不是构造函数的原型对象上，例如function Super() { this.colors = ['black'] } 这样实例之间就不会相互影响了。还有个问题是创建子类型实例的时候，没法向超类型的构造函数中传参，会影响其他实例

2. 借用构造函数（**思想：子类型构造函数内部调用超类型构造函数**） function Super() { this.colors = [] this.name = name } function Sub() { Super.call(this, '222') } var sub = new Sub() 这种方式，每个子类型实例创建的时候都会执行Super并创建自己的属性，这样每个实例之间不会互相影响了，同时还可以实现在子类型构造函数中向超类型函数传参

问题：这种方法无法继承超类型prototype上的方法，函数必须在构造函数中调用，不能实现构造函数复用，每个实例都有自己的函数而不是公用

3. 组合继承（**思想：借用构造函数继承和原型链继承结合**） function Super() { this.name = 'ww' } Super.prototype.say = function() {}

function Sub(a) { Super.call(this,a) } Sub.prototype = new Super() Sub.prototype.constructor = Sub var sub = new Sub()

问题：会调用两次Super构造函数

4. 原型式继承（思想：Object.create()）

```
function create(o) {
  var F = function () {}
  F.prototype = o
  return new F()
}
```

问题：所有实例共用一套引用属性，实例之间会相互影响

myObject.foo = 'bar'

1. 如果myObject有foo，直接修改
2. 不在myObject，查找原型链，如果找不到，直接添加
3. 如果在原型链找到了且没有标记为只读，在myObject上添加
4. 如果在原型链找到了，标记为只读，那么无法修改或在myObject上创建，严格模式下会报错
5. 如果原型链上找到了且是一个setter，那么一定会调用这个setter，foo不会被添加到myObject，也不会重新定义foo这个setter