# 原型链类

- 创建对象有几种方法
- 原型,构造函数,实例,原型链
- instanceof原理
- new 运算符

## 创建对象有几种方法

1. 字面量对象，new Object()创建

```
var o1 = {name:'o1'};//字面量对象（默认原型链指向Object）
var o11 = new Object({name:'o11'});//new Object()创建对象
```

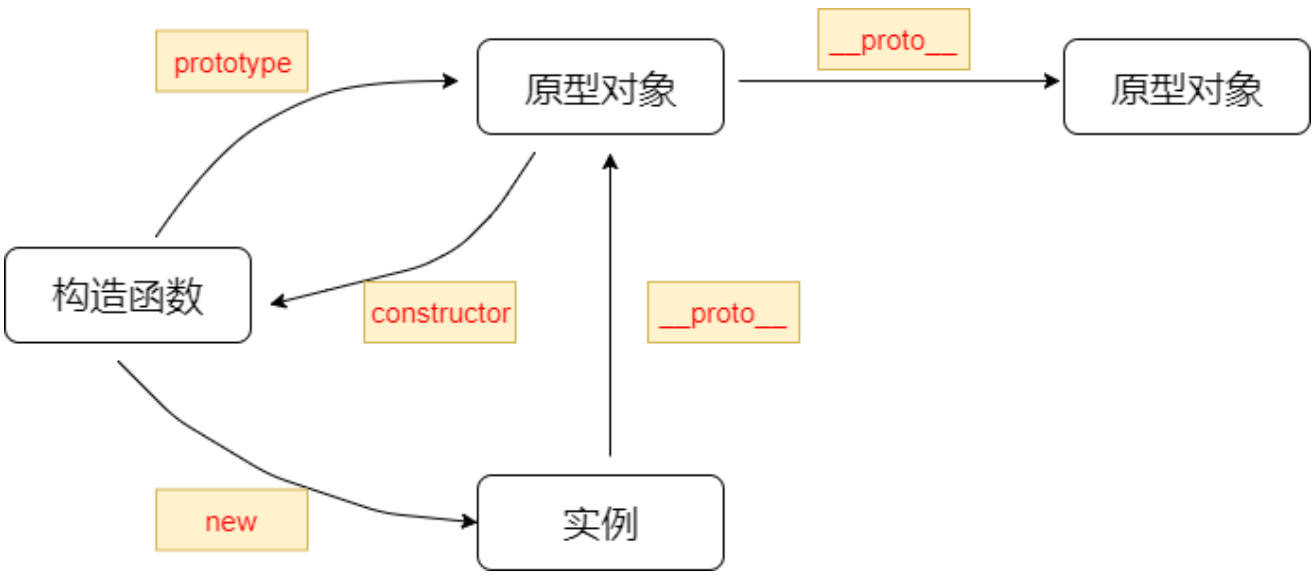
2. 显示构造函数创建对象

```
var M = function(){
    this.name='o2';
}
var o2 = new M();
```

3. Object.create()方法创建

```
var P = {name:'o3'};
var o3 = Object.create(P);
```

## 原型,构造函数,实例,原型链




1. 任何函数只要被`new`使用了就可以被称为构造函数，不用new的函数则是普通函数。

2. 当新建一个函数时，`js`引擎会为这个函数添加一个`prototype`属性，这个属性会创建一个空对象，也就是原型对象。
3. 原型对象中会有一个构造器`constructor`，默认指向你声明的那个函数

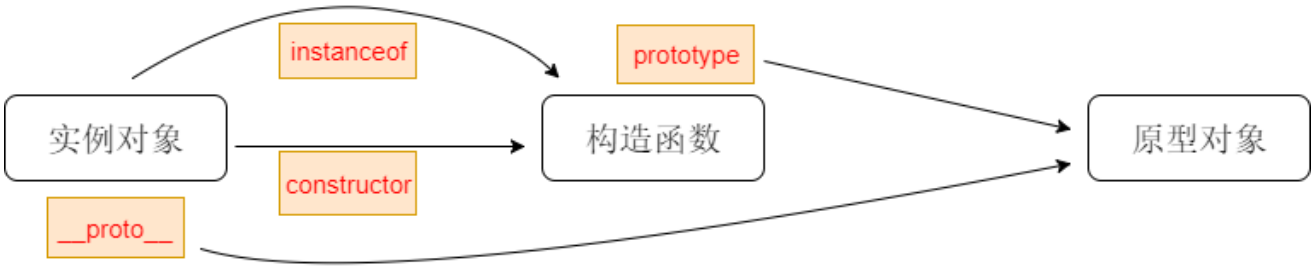
```
function M(name){
    this.name = name;
}
var o3 = new M('o3');
//print o3
M {name: "o3"}
-------------------------
//spread
name:"o3"
__proto__:Object
-------------------------
//spread more
name:"o3"
__proto__:
  constructor:ƒ M(name)
  __proto__:Object
-------------------------
//print M
ƒ M(name){
  this.name = name;
}
-------------------------
//print M.prototype
Object{
  constructor:ƒ M(name)
  __proto__:Object
}
-------------------------
M.prototype.constructor === M;//true
o3.__proto__ === M.prototype;//true
-------------------------
//原型链的顶端为 Object.prototype
-------------------------
M.prototype.say = function(){
   console.log('h1');
}
var o5 = new M('o5');
o3.say();//hi
o5.say();//hi
```

4. 在访问一个实例的时候，比如说在这个实例上有什么方法，在这个实例本身上没有找到这个方法和属性的话，它通过`__proto__`往它的原型对象上找（上一级原型对象），如果还没有找到，继续通过`__proto__`往再上一级原型对象找，找到对应的方法后立即原路返回，如果还没有找到则继续往再上一级原型对象找直到`Object.prototype`。
5. 只有实例对象有`__proto__`属性，函数既是函数也是对象，所以函数也有`__proto__`属性。
6.

```
M.__proto__ === Function.prototype;//true
```

`M`这个普通函数的构造函数是`Function`，也可以理解为`M`这个普通函数是`Function`这个函数的实例。

## instanceof 原理




1. `__proto__`实际关联的是构造函数的`prototype`属性，也就是构造函数的原型对象。
2. `instanceof`的运力是判断实例对象的`__proto__`属性和构造函数的`prototype`属性是不是同一个引用（原型对象）。
3. 在判断实例对象是否为该构造函数的实例的时候，实际上判断的是实例对象的`__proto__`和构造函数的`prototype`是不是引用同一个地址。

```
//print o3
M {name: "o3"}
-----------------------------
o3 instanceof M;//true
o3 instanceof Object;//true
//只要在原型链上的构造函数都会被instanceof看作是实例的构造函数
o3.__proto__ === M.prototype;//true
M.prototype.__proto__ === Object.prototype;//true
-----------------------------
o3.__proto__.constructor === M;//true
o3.__proto__.constructor === Object;//false
```

## new 运算符

### new 操作过程

1. 一个新对象（理解为空对象，字面量对象）被创建。它继承自`foo.prototype`（`foo`为构造函数）。
2. 构造函数`foo`被执行。执行的时候，相应的传参会被传入，同时上下文（`this`）会被指定为这个新实例。`new foo`等同于`new foo()`，只能用在不传递任何参数的情况下。
3. 如果构造函数返回了一个“对象”，那么这个对象会取代整个`new`出来的结果。如果构造函数没有返回值，那么`new`出来的结构为步骤1创建的对象。

```
//new 运算符工作原理
var new2 =function(func){
    var o = Object.create(func.prototype);
    var k = func.call(o);
    if(typeof k === 'Object'){
        return k
```

```
    }else{
        return o
    }
}

var o6 = new2(M);
o6 instanceof M;//true
o6 instanceof Object;//true
o6.__proto__.constructor === M;//true
M.prototype.walk = function(){
    console.log('walk');
}
o6.walk();//walk
----------------------------------
//Object.create()方法创建的一个对象连接原型对象是把参数中的对象作为原型对象赋给P的
var P = {name:'Mike'};
var o7 = Object.create(P);
o7.__proto__ === P;//true
```

## 原型链详解

```
function Foo(){
    //属性和方法
}
var f1 = new Foo();
var f2 = new Foo();
var o1 = new Object();
var o2 = new Object();
```

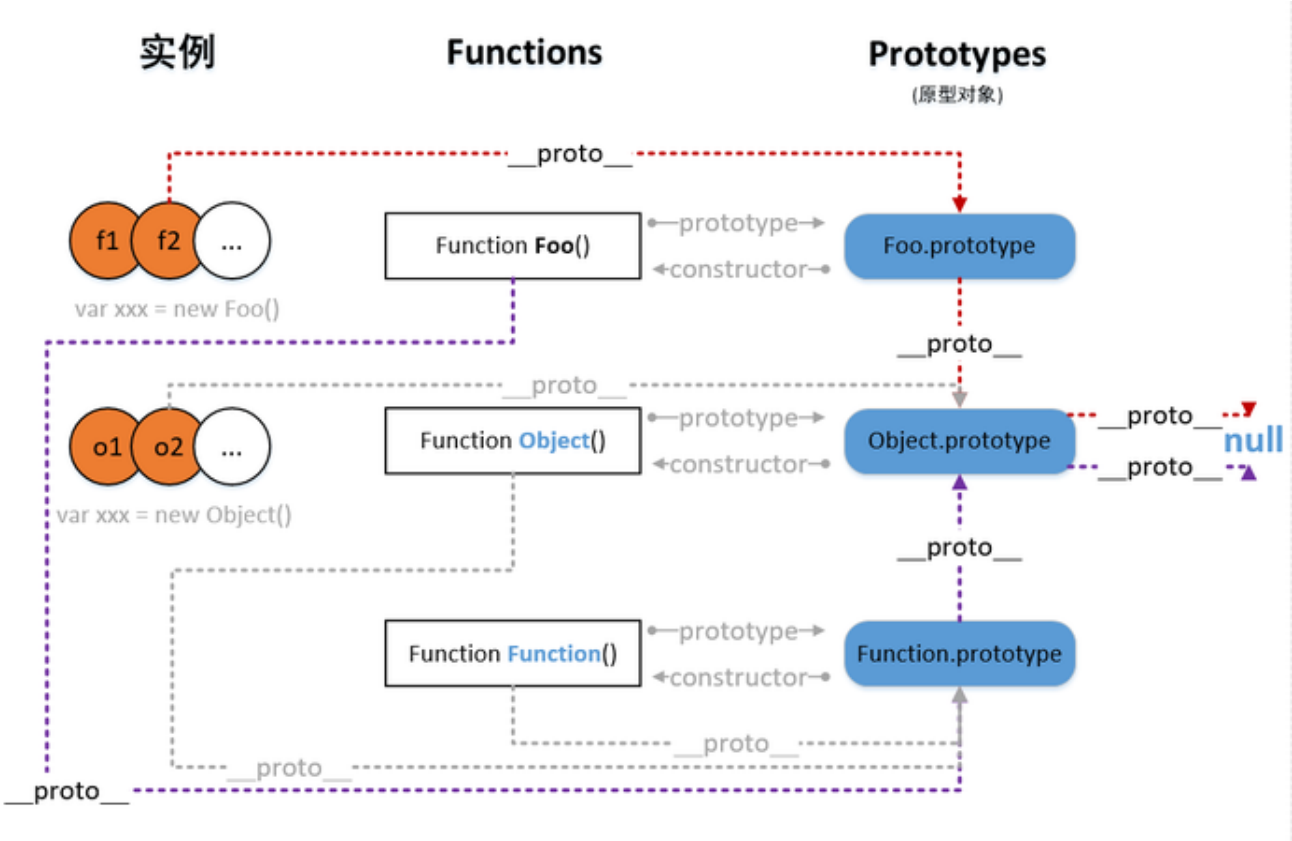
### 对象的分类

1. **普通对象** ，除了函数对象之外的对象都是，**包括new函数对象()产生的实例**，普通对象没有`prototype`。
2. **函数对象** ，包括两种：
    - 由`function`创造出来的函数：

      ```
      function f1() {} // 匿名函数
      var f2 = function() {}
      var f3 = new Function('x','console.log(x)');
      // 以上都是函数对象
      ```

    - 系统内置的函数对象：

      `Function、Object、Array、String、Number` ，**Function其实充当了函数对象的构造器**，比如`Object`对象的构造源码其实是`Function Object() {[native code]}`的形式，这一点对于理解原型链很重要。




上图从结构上分为**实例对象**、**Functions函数对象**、**prototype原型对象**三部分，图中`f1`、`f2`的原型链标成了红色，`Foo`的原型链为紫色。

简单的说，凡是使用`function`关键字或`Function`构造函数创建的对象都是函数对象。而且，只有函数对象才拥有`prototype`（原型）属性。

每个对象都有`__proto__`属性，用于储存继承得来的方法和属性；每个函数对象都有`prototype`属性，用于继承，将其中定义的属性和方法传递给**后代**（比如实例化）。

## 如何实现原型继承

`f1`为何有`Foo`、`Object`的原型方法，其实就是通过原型链继承。继承的过程可以表示为`f1.__proto__ = Foo.prototype`，即`对象.__proto__ = 构造器.prototype`。

`new`实例实现继承的过程其实与上面原理相同，`new`的过程可以拆解为下面几个步骤：

```
var temp = {};
temp.__proto__ = Foo.prototype; // 原型继承
var f1 = Foo.call(temp);
return f1;
```

### （1）`f1`的原型链（红色虚线）

1. `f1`为普通对象，它的构造器为`Foo`，以`Foo`为原型，原型链第一链为`f1.__proto__ == Foo.prototype`；
2. `Foo.prototype`（注意这边不是`Foo`）为`json`对象，即普通对象，构造器为`Object`，以`Object`为原型，得出原型链第二链`Foo.prototype.__proto__ == Object.prototype`；
3. `Object.prototype`以`Null`为原型，原型链第三链为`Object.prototype.__proto__ == null`；


f1原型链

```
f1.__proto__===Foo.prototype;//true
Foo.prototype.__proto__ === Object.prototype;//true
Object.prototype.__proto__ === null;//true
```

**（2） `Foo` 的原型链（紫色虚线）**

1. `Foo` 为函数对象，它的构造器为 `Function` ，以 `Function` 为原型，原型链第一链为 `Foo.__proto__ == Function.prototype` ;
2. `Function.prototype` 为 `json` 对象，即普通对象，构造器为 `Object` ，以 `Object` 为原型，原型链第二链为 `Function.prototype.__proto__ == Object.prototype` ;
3. 最后 `Object.prototype` 以 `Null` 为原型，原型链第三链为 `Object.prototype.__proto__ == null` ;


Foo原型链

```
Foo.__proto__ === Function.prototype;//true
Function.prototype.__proto__===Object.prototype;//true
Object.prototype.__proto__ === null;//true
```

## 小结

当 `js` 引擎执行对象的属性或方法时，先查找对象本身是否存在该方法，如果不存在则会在原型链上查找。 因而 `f1` 拥有 `Foo` 、 `Object` 的原型方法， `Foo` 拥有 `Function` 、 `Object` 的原型方法。

虽然 `f1` 原型链其中有一链是涉及到函数对象 `Foo` ，但 `f1` 并不拥有 `Function` 的原型方法，因为原型链并没有延伸到 `Function` 。

比如下例中 `bind` 是 `Function` 的原型方法， `f1` 并没有拥有。

```
f1.bind;//undefined
Foo.bind;//ƒ bind() { [native code] }
```

如何找出一个对象的原型链，只需要两步

1. 判断对象是普通对象还是函数对象，得出对象的构造器
2. 对象.`proto` = 构造器.`prototype`