

prototype__prototype

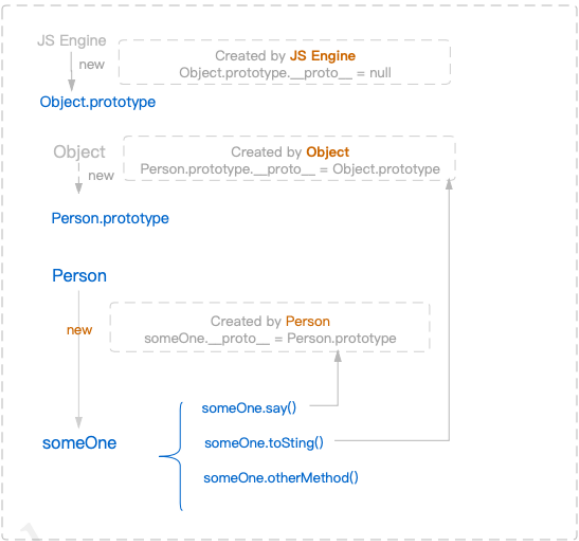
规则1、当一个对象被类（构造函数）实例化后，该实例可以访问该类的原型上的所有方法及属性。（实例的内部是通过__proto__指向类的原型来实现的）

```
function Person(name,age){
    this.name = name;
    this.age = age;
}

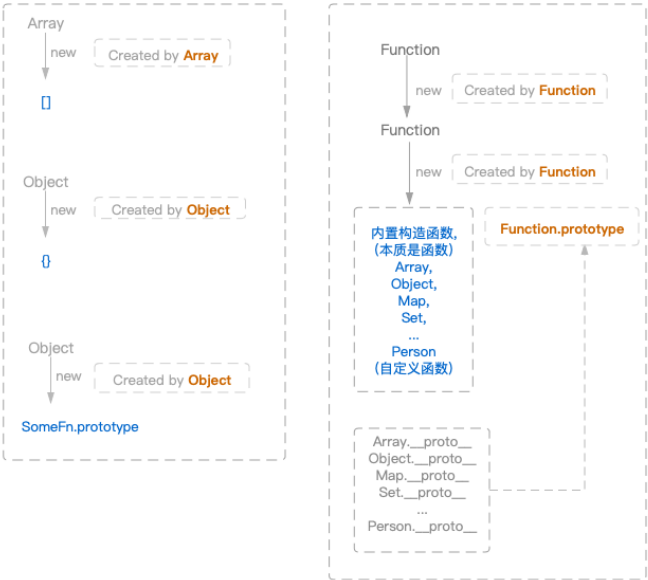
Person.prototype = {
    constructor: Person,
    say(){
        console.log(`Hello, I'm ${this.name}`);
    }
}

let someOne = new Person('someOne',20);

someOne;
▼ Person {name: "someOne", age: 20}
  age: 20
  name: "someOne"
  __proto__:
    ► constructor: f Person(name,age)
    ► say: f say()
    ► __proto__:
      ► constructor: f Object()
      ► hasOwnProperty: f hasOwnProperty()
      ► isPrototypeOf: f isPrototypeOf()
      ► propertyIsEnumerable: f propertyIsEnumerable()
      ► toLocaleString: f toLocaleString()
      ► toString: f toString()
```



规则2、根据一切皆对象的理论，我们可以认为任何对象(不包括值类型)都是由它的类（构造函数）实例化生成的。如果不是现实生成的，可以看成是系统隐式生成。除了Function\Function.prototype\Object.prototype，它们可认为是JS Engine 生成的。



规则3、Function\Function.prototype\Object.prototype 年轻人不讲武德，该3个年轻人不遵守以上2条规则

```
typeof Function.prototype;
"function"

Function.prototype instanceof Object;
true

Function.prototype instanceof Function;
false

Function instanceof Object;
true

Object instanceof Function;
true

Object.prototype instanceof Object;
false
```

灵魂拷问

既然Object,Array,CustomClass（等构造函数）是Function实例（隐式）化的，自定义对象又是t通过CustomClass实例化得到的，自定义对象是Function的实例么？为什么？

js new过程到底值执行了什么

问题：就上图的描述，new过程到底值执行了什么？

思考：..... 1、执行构造函数，得到一个新对象 2、将构造函数的原型，挂载到新对象上 3、得到新对象

instanceof 判断的原理（依据）

判断实例是不是某类对象的实例，去实例对象的原型链里寻找，如果原型链里的某一环原型与类对象的原型一致，就认为该实例是该类对象的实例。

所以，一个实例可以是多个类对象的实例。

isPrototypeOf

isPrototypeOf() 方法用来检测一个对象是否存在于另一个对象的原型链中，如果存在就返回 true，否则就返回 false。与 instanceof 有相似的检测

解读下一下执行结果：

```
var f = function () {}  
console.log(Object.prototype.isPrototypeOf(f));  
console.log(Function.prototype.isPrototypeOf(f));  
console.log(Function.prototype.isPrototypeOf(Object));  
console.log(Object.prototype.isPrototypeOf(Function));  
console.log(Object.prototype.isPrototypeOf(Object.prototype));  
console.log(Object.prototype.isPrototypeOf(Function.prototype));  
console.log(Function.prototype.isPrototypeOf(Function.prototype));  
console.log(Function.prototype.isPrototypeOf(Object.prototype));
```

```
function Person02(name,age){  
    this.name = name;  
    this.age = age;  
}  
Person02.prototype = {  
    constructor:Person02,  
    sayHello:function(){  
        console.log(`hello I am ${this.name} , ${this.age} ages old.`);  
    }  
}  
  
let one = new Person02('will',18);  
console.log(one instanceof Person02);  
console.log(one instanceof Object);
```

typeof 的实现原理

在 javascript 的最初版本中，使用的 32 位系统，为了性能考虑使用低位存储了变量的类型信息：

- 000: 对象
- 1: 整数
- 010: 浮点数
- 100: 字符串
- 110: 布尔

有 2 个值比较特殊：

- undefined: 用 $-(-2^{30})$ 表示。
- null: 对应机器码的 NULL 指针，一般是全零。

在第一版的 javascript 实现中，判断类型的代码是这么写的：

```
1 if (JSVAL_IS_VOID(v)) { // (1)
2     type = JSTYPE_VOID;
3 } else if (JSVAL_IS_OBJECT(v)) { // (2) obj = JSVAL_TO_OBJECT(v); if (obj && (ops = obj->map->ops, ops == &
```

- (1) : 判断是否为 undefined
- (2) : 如果不是 undefined，判断是否为对象
- (3) : 如果不是对象，判断是否为数字
- (4) : ...

这样一来，null 就出了一个 bug。根据 type tags 信息，低位是 000，因此 null 被判断成了一个对象。这就是为什么 `typeof null` 的返回值是 `object`。

关于 null 的类型在 MDN 文档中也有简单的描述：[typeof - javascript | MDN](#)

在 ES6 中曾有关于修复此 bug 的提议，提议中称应该让 `typeof null ===`

'null' http://wiki.ecmascript.org/doku.php?id=conventions:typeof_null 但是该提议被无情的否决了，自此 `typeof null` 终于不再是一个 bug，而是一个 feature，并且永远不会被修复。

```
if (JSVAL_IS_VOID(v)) { // (1)
    type = JSTYPE_VOID;
} else if (JSVAL_IS_OBJECT(v)) { // (2)
    obj = JSVAL_TO_OBJECT(v);
    if (obj &&
        (ops = obj->map->ops,
         ops == &js_ObjectOps
          ? (clasp = OBJ_GET_CLASS(cx, obj),
            clasp->call || clasp == &js_FunctionClass) // (3,4)
            : ops->call != 0)) { // (3)
        type = JSTYPE_FUNCTION;
    } else {
        type = JSTYPE_OBJECT;
    }
} else if (JSVAL_IS_NUMBER(v)) {
    type = JSTYPE_NUMBER;
} else if (JSVAL_IS_STRING(v)) {
    type = JSTYPE_STRING;
} else if (JSVAL_IS_BOOLEAN(v)) {
    type = JSTYPE_BOOLEAN;
}
```

- (1) : 判断是否为 undefined
- (2) : 如果不是 undefined，判断是否为对象
- (3) : 如果不是对象，判断是否为数字
- (4) :

17/ 000

这样一来，null 就出了一个 bug。根据 type tags 信息，低位是 000，因此 null 被判断成了一个对象。这就是为什么 typeof null 的返回值是 object。

关于 null 的类型在 MDN 文档中也有简单的描述：[typeof - javascript | MDN](#)

在 ES6 中曾有关于修复此 bug 的提议，提议中称应该让 typeof null === 'null'http://wiki.ecmascript.org/doku.php?id=conventions:typeof_null 但是该提议被无情的否决了，自此 typeof null 终于不再是一个 bug，而是一个 feature，并且永远不会被修复。

原理是这样的，不同的对象在底层都表示为二进制，在 JavaScript 中二进制前三位都为 0 的话会被判断为 object 类型，null 的二进制表示是全 0，自然前三位也是 0，所以执行 typeof 时会返回“object”。

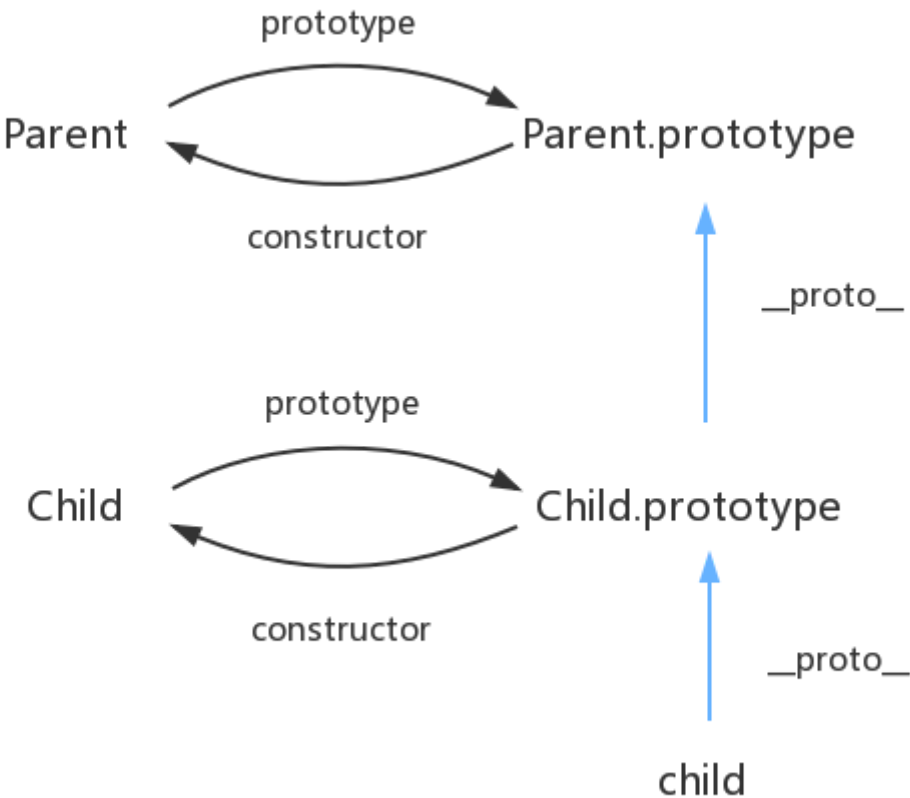
除了null以外的值类型及函数判断都是完美的，引用类型的判断最好别用typeof

提问：能聊一聊你理解的原型和原型链么？

继承的实现方式

关键词:构造函数继承/原型链继承/组合继承/原型式继承/寄生式继承/寄生组合式继承

何为继承？子类实例化过程中，父类、子类的构造函数都是得到顺序执行，实例化以后，子类实例得到父类实例的所有属性与方法，(相同的属性或方法，会通过子类覆盖父类处理)，并且（子类的实例 instanceof 父类/子类）都是true。能实现以上描述的父类、子类，我们称之为子类实现了对父类的继承。



看看ES6是如何实现继承的

```
//看下ES6的实现
class Father{
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
  sayHello(){
    console.log(`hello,I am ${this.name}, ${this.age} !`);
  }
}

class Child extends Father{
  constructor(name,age,gender,height){
    super(name,age);
    this.gender = gender;
    this.height = height;
  }

  goHome(){
    console.log(`I am ${this.name}, ${this.gender},${this.height},
    ${this.age} years old. I am going home!`);
  }
}

let one = new Child('will',18,'male',180);

console.log(one);
```

```

1 "use strict";
2
3 function _inheritsLoose(subClass, superClass) { subClass.prototype =
Object.create(superClass.prototype); subClass.prototype.constructor = subClass;
subClass.__proto__ = superClass; }
4
5 var Father = /*#__PURE__*/function () {
6   function Father(name, age) {
7     this.name = name;
8     this.age = age;
9   }
10
11   var _proto = Father.prototype;
12
13   _proto.sayHello = function sayHello() {
14     console.log("hello,I am " + this.name + ", " + this.age + " !");
15   };
16
17   return Father;
18 }();
19
20 var Child = /*#__PURE__*/function (_Father) {
21   _inheritsLoose(Child, _Father);
22
23   function Child(name, age, gender, height) {
24     var _this;
25
26     _this = _Father.call(this, name, age) || this;
27     _this.gender = gender;
28     _this.height = height;
29     return _this;
30   }
31
32   var _proto2 = Child.prototype;
33
34   _proto2.goHome = function goHome() {
35     console.log("I am " + this.name + ", " + this.gender + ", " + this.height + ", " +
this.age + " years old. I am going home!");
36   };
37
38   return Child;
39 }(Father);
40
41 var one = new Child('will', 18, 'male', 180);

```

▼ Child {name: "will", age: 18, gender: "male", height: 180} ⓘ

age: 18

gender: "male" 属性

height: 180

name: "will"

原型链上第1层是子类的原型，原型链的第2层是父类的原型，生成的结果是父类或子类的实例

▼ __proto__: Father

▶ constructor: class Child

▶ goHome: f goHome()


原型链的第1层，有自己类定义的方法 goHome

▼ __proto__:

▶ constructor: class Father

▶ sayHello: f sayHello()

原型链的第2层，有父类的方法 sayHello

 `___proto___: Object`

ES6继承转码：核心原理：0、闭包封装 1、在子类执行之前，通过Object.create挂载父类的原型，得到子类的原型对象，后面扩展子类的原型时直接在这个原型对象上扩展。2、在子类执行时构造函数前，先执行父类的构造函数。

缺点：有代码侵入

体会下 `subClass.prototype = superClass`; 对象关联扩展的好方法。

继承自我实现

```
function Father(name,age){
    this.name = name;
    this.age = age;
}
//如果显示定义怕prototype对象的化，请一定手动加上constructor属性
Father.prototype = {
    constructor:Father,
    sayHello:function(){
        console.log(`hello,I am ${this.name}, ${this.age} !`);
    }
}

function Child(gender,height){
    this.gender = gender;
    this.height = height;
}

Child.prototype = {
    constructor:Child,
    goHome(){
        console.log(`I am ${this.name}, ${this.gender},${this.height},
${this.age} years old. I am going home!`);
    }
}

//继承实现【原创】 __proto__ 兼容性有待考验
Function.prototype.myInherit = function(Father,FatherParams,selfParams){
    /*
    let obj = {};
    Father.apply(obj,FatherParams);
    this.apply(obj,selfParams);
    obj.__proto__ = this.prototype;
    this.prototype.__proto__ = Father.prototype;
    return obj;
    */

    /*
    let obj = {};
    Father.apply(obj,FatherParams);
    this.apply(obj,selfParams);
    */
}
```



```

    obj.__proto__ =
    Object.setPrototypeOf(this.prototype, Father.prototype);
    return obj;
    */

    this.prototype =
    Object.setPrototypeOf(this.prototype, Father.prototype);
    let newObj = new this(...selfParams);
    //父类构造函数后置
    Father.apply(newObj, FatherParams);
    return newObj;
}

let one = Child.myInherit(Father, ['will', 19], ['male', 180]);

```

通用的ES5寄生组合式继承

```

function Father(name, age) {
    this.name = name;
    this.age = age;
}
//如果显示定义怕prototype对象的化，请一定手动加上constructor属性
Father.prototype = {
    constructor: Father,
    sayHello: function() {
        console.log(`hello, I am ${this.name}, ${this.age} !`);
    }
}

function Child(gender, height, fatherParams) {
    Father.apply(this, fatherParams);
    this.gender = gender;
    this.height = height;
}
// 提问，
// 需要人有说出用Object.create的原因？ 这点很很很很重要！ 继承的关键！

/*
Child.prototype = Object.create(Father.prototype);
Child.prototype.constructor = Child;
Child.prototype.goHome = function() {
    console.log(`I am ${this.name}, ${this.gender}, ${this.height},
    ${this.age} years old. I am going home!`);
}
*/

Child.prototype = Object.setPrototypeOf({

```

```
    constructor:Child,
    goHome(){
        console.log(`I am ${this.name}, ${this.gender},${this.height},
        ${this.age} years old. I am going home!`);
    }
},Father.prototype);

let one = new Child('male',180,['will',18]);

console.log(one);
```

以上3个继承方法，使用方法不一样但却实现了完全同样的功能。666!

解读网上6继承方式

<https://blog.csdn.net/kingsleytong/article/details/68943675> 1、构造继承 继承补全 2、原型链继承 (实现了) 所有子类的实例的原型都共享同一个超类实例的属性和方法 3、组合继承 实际上子类上会拥有超类的两份属性，构造函数执行了2次 4、原型式继承 好没有实现继承吧 5、寄生式继承 静态的？ 6、寄生组合继承