2. 面向对象编程/原型及原型链

1. 课程目标

💡
- 初级：
  - 掌握JavaScript基础使用；
- 中级：
  - 掌握JavaScript语法的常见面试题及使用技巧；
- 高级：
  - 掌握JavaScript的核心原理实现；

2. 课程大纲

- 参数按值传递
- 手写call 和 apply
- 手写 bind
- 手写模拟 new
- 类数组对象和 arguments
- 创建对象的多种方式&优缺点
- 继承的多种方式&优缺点

3. 参数按值传递

在《JavaScript高级程序设计》中提到传递参数：

> ECMAScript中所有函数的参数都是按值传递的。

什么是按值传递呢？

> 把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。

3.1 按值传递

举个简单的例子：

```
var value = 1;
function foo(v) {
    v = 2;
    console.log(v); //2
}
foo(value);
console.log(value) // 1
```

很好理解，当传递 value 到函数 foo 中，相当于拷贝了一份 value，假设拷贝的这份叫 _value，函数中修改的都是 _value 的值，而不会影响原来的 value 值。

3.2 共享传递

拷贝虽然很好理解，但是当值是一个复杂的数据结构的时候，拷贝会产生性能上的问题。

这里提及一种：按引用传递。

所谓按引用传递，就是传递对象的引用，函数内部对参数的任何改变都会影响该对象的值，因为两者引用的是同一个对象。

举个例子：

```
var obj = {
    value: 1
};
function foo(o) {
    o.value = 2;
    console.log(o.value); //2
}
foo(obj);
console.log(obj.value) // 2
```

为什么《JavaScript高级程序设计》都说了 ECMAScript 中所有函数的参数都是按值传递的，那为什么能按"引用传递"成功呢?

```
var obj = {
    value: 1
};
function foo(o) {
    o = 2;
    console.log(o); //2
}
foo(obj);
console.log(obj.value) // 1
```

如果 JavaScript 采用的是引用传递，外层的值也会被修改，那这里如何解释?

这就要讲到第二种传递方式，叫按共享传递。

而共享传递是指，在传递对象的时候，传递的是地址索引。

所以修改 o.value，可以通过引用找到原值，但是直接修改 o，并不会修改原值。所以第二个和第三个例子其实都是按共享传递。

最后，你可以这样理解：

参数如果是基本类型是按值传递，如果是引用类型按共享传递。

但是因为拷贝副本也是一种值的拷贝，所以在高程中也直接认为是按值传递了。

换句话说，函数传递参数 ，传递的是参数的拷贝：

1. 指针拷贝，拷贝的是地址索引；
2. 常规类型拷贝，拷贝的是值 ；

所以，一共是两种传递方式，按值传递和按共享传递。

3.3 总结

javascript中数据类型分为基本类型与引用类型：

1. 基本类型值存储于栈内存中，传递的就是当前值，修改不会影响原有变量的值；
2. 引用类型值其实也存于栈内存中，只是它的值是指向堆内存当中实际值的一个地址；索引引用传递传的值是栈内存当中的引用地址，当改变时，改变了堆内存当中的实际值；

所以针对上述的内容：

```
1   var value = 1;
2   function foo(v) {
3       v = 2;
4       console.log(v); //2
5   }
6   foo(value);
7   console.log(value) // 1
```

内存分布：

改变前：

栈内存		堆内存

value	1	
v	1	

改变后：

栈内存		堆内存
value	1	
v	2	

```
var obj = {
value: 1
};
function foo(o) {
    o.value = 2;
    console.log(o.value); //2
}
foo(obj);
console.log(obj.value) // 2
```

改变前：

栈内存		堆内存
obj	指针地址	{value: 1}
o	指针地址	{value: 1}

改变后：

栈内存		堆内存
obj	指针地址	{value: 2}
o	指针地址	{value: 2}

```
var obj = {
value: 1
```

```
};
function foo(o) {
    o = 2;
    console.log(o); //2
}
foo(obj);
console.log(obj.value) // 1
```

改变前：

栈内存		堆内存
obj	指针地址	{value: 1}
o	指针地址	{value: 1}

改变后：

栈内存		堆内存
obj	指针地址	{value: 1}
o	2	

4. 手写 call和 apply

4.1 手写call

> call()：在使用一个指定的 this 值和若干个指定的参数值的前提下调用某个函数或方法。

```
let foo = {
    value: 1
};

function bar() {
    console.log(this.value);
}

bar.call(foo); // 1
```

注意两点：

1. call 改变了 this 的指向，指向到 foo；
2. bar 函数执行了；

4.1.1 第一步

上述方式等同于：

```
let foo = {
    value: 1,
    bar: function() {
        console.log(this.value)
    }
};

foo.bar(); // 1
```

这个时候 this 就指向了 foo，但是这样却给 foo 对象本身添加了一个属性，所以们用 delete 再删除它即可。

所以我们模拟的步骤可以分为：

1. 将函数设为对象的属性；
2. 执行该函数；
3. 删除该函数；

以上个例子为例，就是：

```
// 第一步
// fn 是对象的属性名，反正最后也要删除它，所以起什么都可以。
foo.fn = bar
// 第二步
foo.fn()
// 第三步
delete foo.fn
```

根据上述思路，提供一版：

```
// 第一版
Function.prototype.call2 = function(context) {
    // 首先要获取调用call的函数，用this可以获取
    context.fn = this;
    context.fn();
    delete context.fn;
}

// 测试一下
 let foo = {
    value: 1
};

function bar() {
    console.log(this.value);
}

bar.call2(foo); // 1
```

4.1.2 第二步

call除了可以指定this，还可以指定参数

```
var foo = {
    value: 1
};

function bar(name, age) {
    console.log(name)
    console.log(age)
    console.log(this.value);
}

bar.call(foo, 'kevin', 18);
// kevin
// 18
// 1
```

可以从 Arguments 对象中取值，取出第二个到最后一个参数，然后放到一个数组里。

上述代码的Arguments中取第二个到最后一个的参数

```
// 以上个例子为例，此时的arguments为：
// arguments = {
```

```
//      0: foo,
//      1: 'kevin',
//      2: 18,
//      length: 3
// }
// 因为arguments是类数组对象，所以可以用for循环
var args = [];
for(var i = 1, len = arguments.length; i < len; i++) {
    args.push('arguments[' + i + ']');
}

// 执行后 args为 ["arguments[1]", "arguments[2]", "arguments[3]"]
```

接下来使用eval拼接成一个函数

```
eval('context.fn(' + args +')')
```

考虑到目前大部分浏览器在console中限制eval的执行，也可以使用rest

此处代码为：

```
// 第二版
Function.prototype.call2 = function(context) {
    context.fn = this;
    let arg = [...arguments].slice(1)
    context.fn(...arg)
    delete context.fn;
}

// 测试一下
var foo = {
    value: 1
};

function bar(name, age) {
    console.log(name)
    console.log(age)
    console.log(this.value);
}

bar.call2(foo, 'kevin', 18);
// kevin
// 18
```

```
// 1
```

4.1.3 第三步

1. this 参数可以传 null，当为 null 的时候，视为指向 window

举个例子：

```
var value = 1;

function bar() {
    console.log(this.value);
}

bar.call(null); // 1
```

2. 针对函数，可以实现返回值

```
var obj = {
    value: 1
}

function bar(name, age) {
    return {
        value: this.value,
        name: name,
        age: age
    }
}

console.log(bar.call(obj, 'kevin', 18));
// Object {
//    value: 1,
//    name: 'kevin',
//    age: 18
// }
```

这里

```
// 第三版
Function.prototype.call2 = function (context) {
        var context = context || window;
```

```
    context.fn = this;

    let arg = [...arguments].slice(1)
    let result = context.fn(...arg)

    delete context.fn
    return result
}

// 测试一下
var value = 2;

var obj = {
    value: 1
}

function bar(name, age) {
    console.log(this.value);
    return {
        value: this.value,
        name: name,
        age: age
    }
}

bar.call2(null); // 2

console.log(bar.call2(obj, 'kevin', 18));
// 1
// Object {
//    value: 1,
//    name: 'kevin',
//    age: 18
// }
```

这边给出最简化的写法：

```
Function.prototype.call2 = function(context, ...args) {
  // 判断是否是undefined和null
  if (typeof context === 'undefined' || context === null) {
    context = window
  }
  let fnSymbol = Symbol()
  context[fnSymbol] = this
  let fn = context[fnSymbol](...args)
```

```
    delete context[fnSymbol]
    return fn
}
```

4.2 手写apply

apply 的实现跟 call 类似，只是入参不一样，apply为数组

```
Function.prototype.apply = function (context, arr) {
    var context = Object(context) || window;
    context.fn = this;

    var result;
    if (!arr) {
        result = context.fn();
    }
    else {
                result = context.fn(...arr)
    }

    delete context.fn
    return result;
}
```

最简化版方式：

```
Function.prototype.apply2 = function(context, args) {
  // 判断是否是undefined和null
  if (typeof context === 'undefined' || context === null) {
    context = window
  }
  let fnSymbol = Symbol()
  context[fnSymbol] = this
  let fn = context[fnSymbol](...args)
  delete context[fnSymbol]
  return fn
}
```

5. 手写 bind

> bind() 方法会创建一个新函数。当这个新函数被调用时，bind() 的第一个参数将作为它运行时的 this，之后的一序列参数将会在传递的实参前传入作为它的参数。

由此我们可以首先得出 bind 函数的两个特点：

1. 返回一个函数；
2. 可以传入参数；

5.1 返回函数的模拟实现

```
var foo = {
    value: 1
};

function bar() {
    console.log(this.value);
}

// 返回了一个函数
var bindFoo = bar.bind(foo);

bindFoo(); // 1
```

关于指定 this 的指向，我们可以使用 call 或者 apply 实现

```
// 第一版
Function.prototype.bind2 = function (context) {
    var self = this;

    // 考到绑定函数可能是有返回值的，加上return
    return function () {
        return self.apply(context);
    }

}
```

5.2 传参的模拟实现

接下来，关于参数的传递：

```
var foo = {
    value: 1
```

```
};

function bar(name, age) {
    console.log(this.value);
    console.log(name);
    console.log(age);

}

var bindFoo = bar.bind(foo, 'daisy');
bindFoo('18');
// 1
// daisy
// 18
```

当需要传 name 和 age 两个参数时，可以在 bind 的时候，只传一个 name，在执行返回的函数的时候，再传另一个参数 age。

这里如果不适用rest，使用arguments进行处理：

```
// 第二版
Function.prototype.bind2 = function (context) {

    var self = this;
    // 获取bind2函数从第二个参数到最后一个参数
    var args = Array.prototype.slice.call(arguments, 1);

    return function () {
        // 这个时候的arguments是指bind返回的函数传入的参数
        var bindArgs = Array.prototype.slice.call(arguments);
        return self.apply(context, args.concat(bindArgs));
    }
}
```

5.3 构造函数效果的模拟实现

bind 还有一个特点，就是

> 一个绑定函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。提供的 this 值被忽略，同时调用时的参数被提供给模拟函数。

也就是说当 bind 返回的函数作为构造函数的时候，bind 时指定的 this 值会失效，但传入的参数依然生效。举个例子：

```
var value = 2;

var foo = {
    value: 1
};

function bar(name, age) {
    this.habit = 'shopping';
    console.log(this.value);
    console.log(name);
    console.log(age);
}

bar.prototype.friend = 'kevin';

var bindFoo = bar.bind(foo, 'daisy');

var obj = new bindFoo('18');
// undefined
// daisy
// 18
console.log(obj.habit);
console.log(obj.friend);
// shopping
// kevin
```

尽管在全局和 foo 中都声明了 value 值，最后依然返回了 undefind，说明绑定的 this 失效了

> 后文中new 的模拟实现，就会知道这个时候的 this 已经指向了 obj。

```
// 第三版
Function.prototype.bind2 = function (context) {
    var self = this;
    var args = Array.prototype.slice.call(arguments, 1);

    var fBound = function () {
        var bindArgs = Array.prototype.slice.call(arguments);
        // 当作为构造函数时，this 指向实例，此时结果为 true，将绑定函数的 this 指向该实例，可以让实例获得来自绑定函数的值
        // 以上面的是 demo 为例，如果改成 `this instanceof fBound ? null : context`，实例只是一个空对象，将 null 改成 this ，实例会具有 habit 属性
        // 当作为普通函数时，this 指向 window，此时结果为 false，将绑定函数的 this 指向 context
```

```
        return self.apply(this instanceof fBound ? this : context,
    args.concat(bindArgs));
    }
    // 修改返回函数的 prototype 为绑定函数的 prototype，实例就可以继承绑定函数的原型
    中的值
    fBound.prototype = this.prototype;
    return fBound;
}
```

5.4 构造函数效果的优化实现

但是在这个写法中，我们直接将 `fBound.prototype = this.prototype`，我们直接修改 fBound.prototype 的时候，也会直接修改绑定函数的 prototype。这个时候，我们可以通过一个空函数来进行中转：

```
// 第四版
Function.prototype.bind2 = function (context) {

    var self = this;
    var args = Array.prototype.slice.call(arguments, 1);

    var fNOP = function () {};

    var fBound = function () {
        var bindArgs = Array.prototype.slice.call(arguments);
        return self.apply(this instanceof fNOP ? this : context,
    args.concat(bindArgs));
    }

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();
    return fBound;
}
```

5.5 最终版

调用 bind 的不是函数时，提示错误：

```
if (typeof this !== "function") {
  throw new Error("Function.prototype.bind - what is trying to be bound is
    not callable");
}
```

最终代码为：

```
Function.prototype.bind2 = function (context) {

    if (typeof this !== "function") {
      throw new Error("Function.prototype.bind - what is trying to be bound is not callable");
    }

    var self = this;
    var args = Array.prototype.slice.call(arguments, 1);

    var fNOP = function () {};

    var fBound = function () {
        var bindArgs = Array.prototype.slice.call(arguments);
        return self.apply(this instanceof fNOP ? this : context, args.concat(bindArgs));
    }

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();
    return fBound;
}
```

最简化版：

```
Function.prototype.myBind = function(context) {
// 判断是否是undefined 和 null
    if (typeof context === "undefined" || context === null) {
        context = window;
    }
    self = this;
    return function(...args) {
        return self.apply(context, args);
    }
}
```

6. 手写模拟 new

new 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象类型之一

先看看 new 实现了哪些功能。

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

    this.habit = 'Games';
}

Person.prototype.strength = 80;

Person.prototype.sayYourName = function () {
    console.log('I am ' + this.name);
}

var person = new Person('Kevin', '18');

console.log(person.name) // Kevin
console.log(person.habit) // Games
console.log(person.strength) // 60

person.sayYourName(); // I am Kevin
```

我们可以看到，实例 person 可以：

1. 访问到 Otaku 构造函数里的属性；
2. 访问到 Otaku.prototype 中的属性；

接下来，我们可以尝试着模拟一下了。

因为 new 是关键字，所以无法像 bind 函数一样直接覆盖，所以我们写一个函数，命名为 objectFactory，来模拟 new 的效果。用的时候是这样的：

```
function Person () {
    ……
}

// 使用 new
var person = new Person(……);
// 使用 objectFactory
var person = objectFactory(Person, ……)
```

6.1 初步实现

因为 new 的结果是一个新对象，所以在模拟实现的时候，我们也要建立一个新对象，假设这个对象叫 obj，因为 obj 会具有 Person 构造函数里的属性，我们可以使用 `Person.apply(obj, arguments)` 来给 obj 添加新的属性。

然后，实例的 **proto** 属性会指向构造函数的 prototype，也正是因为建立起这样的关系，实例可以访问原型上的属性

```
// 第一版代码
function objectFactory() {
    var obj = new Object(),
    Constructor = [].shift.call(arguments);
    obj.__proto__ = Constructor.prototype;
    Constructor.apply(obj, arguments);

    return obj;

};
```

在这一版中，我们：

1. 用new Object() 的方式新建了一个对象 obj；
2. 取出第一个参数，就是我们要传入的构造函数。此外因为 shift 会修改原数组，所以 arguments 会被去除第一个参数；
3. 将 obj 的原型指向构造函数，这样 obj 就可以访问到构造函数原型中的属性；
4. 使用 apply，改变构造函数 this 的指向到新建的对象，这样 obj 就可以访问到构造函数中的属性；
5. 返回 obj；

测试下：

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

    this.habit = 'Games';
}

Person.prototype.strength = 60;

Person.prototype.sayYourName = function () {
    console.log('I am ' + this.name);
}
```

```
13
14  function objectFactory() {
15      var obj = new Object(),
16      Constructor = [].shift.call(arguments);
17      obj.__proto__ = Constructor.prototype;
18      Constructor.apply(obj, arguments);
19      return obj;
20  };
21
22  var person = objectFactory(Person, 'Kevin', '18')
23
24  console.log(person.name) // Kevin
25  console.log(person.habit) // Games
26  console.log(person.strength) // 60
27
28  person.sayYourName(); // I am Kevin
```

6.2 最终实现

假如构造函数有返回值

```
1   function Person (name, age) {
2       this.strength = 60;
3       this.age = age;
4
5       return {
6           name: name,
7           habit: 'Games'
8       }
9   }
10
11  var person = new Person('Kevin', '18');
12
13  console.log(person.name) // Kevin
14  console.log(person.habit) // Games
15  console.log(person.strength) // undefined
16  console.log(person.age) // undefined
```

在这个例子中，构造函数返回了一个对象，在实例 person 中只能访问返回的对象中的属性。

而且还要注意一点，在这里我们是返回了一个对象，假如我们只是返回一个基本类型的值呢？

再举个例子：

```
function Person (name, age) {
    this.strength = 60;
    this.age = age;

    return 'handsome boy';
}

var person = new Otaku('Kevin', '18');

console.log(person.name) // undefined
console.log(person.habit) // undefined
console.log(person.strength) // 60
console.log(person.age) // 18
```

这次尽管有返回值，但是相当于没有返回值进行处理。

所以我们还需要判断返回的值是不是一个对象，如果是一个对象，我们就返回这个对象，如果没有，我们该返回什么就返回什么。

```
// 最终版的代码
function objectFactory() {
    var obj = new Object(),
    Constructor = [].shift.call(arguments);
    obj.__proto__ = Constructor.prototype;
    var ret = Constructor.apply(obj, arguments);
    return typeof ret === 'object' ? ret : obj;

};
```

7. 类数组对象与arguments

7.1 类数组对象

所谓的类数组对象:

拥有一个 length 属性和若干索引属性的对象

举个例子:

```
var array = ['name', 'age', 'sex'];

var arrayLike = {
    0: 'name',
    1: 'age',
```

```
    2: 'sex',
    length: 3
}
```

7.1.1 读写

```
console.log(array[0]); // name
console.log(arrayLike[0]); // name

array[0] = 'new name';
arrayLike[0] = 'new name';
```

7.1.2 长度

```
console.log(array.length); // 3
console.log(arrayLike.length); // 3
```

7.1.3 遍历

```
for(var i = 0, len = array.length; i < len; i++) {
   ……
}
for(var i = 0, len = arrayLike.length; i < len; i++) {
    ……
}
```

但是调用原生的数组方法会报错，如push：

```
arrayLike.push is not a function
```

7.1.4 调用数组方法

只能通过 Function.call 间接调用

```
var arrayLike = {0: 'name', 1: 'age', 2: 'sex', length: 3 }

```

```
Array.prototype.join.call(arrayLike, '&'); // name&age&sex

Array.prototype.slice.call(arrayLike, 0); // ["name", "age", "sex"]
// slice可以做到类数组转数组

Array.prototype.map.call(arrayLike, function(item){
    return item.toUpperCase();
});
// ["NAME", "AGE", "SEX"]
```

7.1.5 类数组转数组

```
var arrayLike = {0: 'name', 1: 'age', 2: 'sex', length: 3 }
// 1. slice
Array.prototype.slice.call(arrayLike); // ["name", "age", "sex"]
// 2. splice
Array.prototype.splice.call(arrayLike, 0); // ["name", "age", "sex"]
// 3. ES6 Array.from
Array.from(arrayLike); // ["name", "age", "sex"]
// 4. apply
Array.prototype.concat.apply([], arrayLike)
```

7.2 Arguments对象

Arguments 对象只定义在函数体中，包括了函数的参数和其他属性。在函数体中，arguments 指代该函数的 Arguments 对象。

举个例子：

```
function foo(name, age, sex) {
    console.log(arguments);
}

foo('name', 'age', 'sex')
```

打印结果：

```
function foo(name, age, sex) {
    console.log(arguments);
}

foo('name', 'age', 'sex')
```

```
Arguments(3) ['name', 'age', 'sex', callee: ƒ, Symbol(Symbol.iterator): ƒ]
  0: "name"
  1: "age"
  2: "sex"
  callee: ƒ foo(name, age, sex)
  length: 3
  Symbol(Symbol.iterator): ƒ values()
  [[Prototype]]: Object
```

可以看到除了类数组的索引属性和length属性之外，还有一个callee属性

7.2.1 length属性

Arguments对象的length属性，表示实参的长度，举个例子：

```
function foo(b, c, d){
    console.log("实参的长度为：" + arguments.length)
}

console.log("形参的长度为：" + foo.length)

foo(1)

// 形参的长度为：3
// 实参的长度为：1
```

7.2.2 callee属性

Arguments 对象的 callee 属性，通过它可以调用函数自身。

讲个闭包经典面试题使用 callee 的解决方法：

```
var data = [];

for (var i = 0; i < 3; i++) {
    (data[i] = function () {
       console.log(arguments.callee.i)
    }).i = i;
}

data[0]();
data[1]();
```

```
data[2]();

// 0
// 1
// 2
```

7.2.3 arguments 和对应参数的绑定

```
function foo(name, age, sex, hobbit) {

    console.log(name, arguments[0]); // name name

    // 改变形参
    name = 'new name';

    console.log(name, arguments[0]); // new name new name

    // 改变arguments
    arguments[1] = 'new age';

    console.log(age, arguments[1]); // new age new age

    // 测试未传入的是否会绑定
    console.log(sex); // undefined

    sex = 'new sex';

    console.log(sex, arguments[2]); // new sex undefined

    arguments[3] = 'new hobbit';

    console.log(hobbit, arguments[3]); // undefined new hobbit

}

foo('name', 'age')
```

传入的参数，实参和 arguments 的值会共享，当没有传入时，实参与 arguments 值不会共享

7.2.4 传递参数

将参数从一个函数传递到另一个函数

```
// 使用 apply 将 foo 的参数传递给 bar
function foo() {
    bar.apply(this, arguments);
}
function bar(a, b, c) {
   console.log(a, b, c);
}

foo(1, 2, 3)
```

7.2.5 ES6

使用ES6的 ... 运算符，我们可以轻松转成数组。

```
function func(...arguments) {
    console.log(arguments); // [1, 2, 3]
}

func(1, 2, 3);
```

8. 创建对象的多种方式&优缺点

8.1 工厂模式

```
function createPerson(name) {
    var o = new Object();
    o.name = name;
    o.getName = function () {
        console.log(this.name);
    };

    return o;
}

var person1 = createPerson('kevin');
```

优点：简单；

缺点：对象无法识别，因为所有的实例都指向一个原型；

8.2 构造函数模式

```
function Person(name) {
    this.name = name;
    this.getName = function () {
        console.log(this.name);
    };
}

var person1 = new Person('kevin');
```

优点：实例可以识别为一个特定的类型；

缺点：每次创建实例时，每个方法都要被创建一次；

8.2.1 构造函数优化

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

function getName() {
    console.log(this.name);
}

var person1 = new Person('kevin');
```

解决了每个方法都要重新创建的问题

8.3 原型模式

```
function Person(name) {

}

Person.prototype.name = 'xianzao';
Person.prototype.getName = function () {
    console.log(this.name);
};

var person1 = new Person();
```

优点：方法不会重新创建；

缺点：

1. 所有的属性和方法都共享；
2. 不能初始化参数；

8.3.1 原型模式优化

```
function Person(name) {

}

Person.prototype = {
    name: 'xianzao',
    getName: function () {
        console.log(this.name);
    }
};

var person1 = new Person();
```

优点：封装清晰点；

缺点：重写了原型，丢失了constructor属性；

8.3.2 原型模式优化2

```
function Person(name) {

}

Person.prototype = {
    constructor: Person,
    name: 'kevin',
    getName: function () {
        console.log(this.name);
    }
};

var person1 = new Person();
```

优点：实例可以通过constructor属性找到所属构造函数；

缺点：

1. 所有的属性和方法都共享；
2. 不能初始化参数；

8.4 组合模式

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

Person.prototype = {
    constructor: Person,
    getName: function () {
        console.log(this.name);
    }
};

var person1 = new Person();
```

优点：该共享的共享，该私有的私有，使用最广泛的方式；

缺点：希望写在一个地方，即更好的封装性；

8.4.1 动态原型模式

```
function Person(name) {
    this.name = name;
    if (typeof this.getName != "function") {
        Person.prototype.getName = function () {
            console.log(this.name);
        }
    }
}

var person1 = new Person();
```

注意：使用动态原型模式时，不能用对象字面量重写原型

```
function Person(name) {
    this.name = name;
    if (typeof this.getName != "function") {
```

```
        Person.prototype = {
            constructor: Person,
            getName: function () {
                console.log(this.name);
            }
        }
    }

    var person1 = new Person('xianzao');
    var person2 = new Person('zaoxian');

    // 报错 并没有该方法
    person1.getName();

    // 注释掉上面的代码，这句是可以执行的。
    person2.getName();
```

开始执行 `var person1 = new Person('xianzao')`

我们回顾下 new 的实现步骤：

1. 首先新建一个对象；
2. 然后将对象的原型指向 `Person.prototype`；
3. 然后 `Person.apply(obj)`；
4. 返回这个对象；

注意这个时候，回顾下 apply 的实现步骤，会执行 obj.Person 方法，这个时候就会执行 if 语句里的内容，注意构造函数的 prototype 属性指向了实例的原型，使用字面量方式直接覆盖 `Person.prototype`，并不会更改实例的原型的值，person1 依然是指向了以前的原型，而不是 `Person.prototype`。而之前的原型是没有 getName 方法的，所以就报错了。

如果你就是想用字面量方式写代码，可以尝试下这种：

```
    function Person(name) {
        this.name = name;
        if (typeof this.getName != "function") {
            Person.prototype = {
                constructor: Person,
                getName: function () {
                    console.log(this.name);
                }
```

```
        }

        return new Person(name);
    }
}

var person1 = new Person('xianzao');
var person2 = new Person('zaoxian');

person1.getName(); // xianzao
person2.getName();  // zaoxian
```

9. 继承的多种方式&优缺点

9.1 原型链继承

```
function Parent () {
    this.name = 'xianzao';
}

Parent.prototype.getName = function () {
    console.log(this.name);
}

function Child () {

}

Child.prototype = new Parent();

var child1 = new Child();

console.log(child1.getName()) // xianzao
```

问题：引用类型的属性被所有实例共享，举个例子：

```
function Parent () {
    this.names = ['xianzao', 'zaoxian'];
}

function Child () {

```

```
}

Child.prototype = new Parent();

var child1 = new Child();

child1.names.push('test');

console.log(child1.names); // ["xianzao", "zaoxian", "test"]

var child2 = new Child();

console.log(child2.names); // ["xianzao", "zaoxian", "test"]
```

9.2 借用构造函数

```
function Parent () {
    this.names = ['xianzao', 'zaoxian'];
}

function Child () {
    Parent.call(this);
}

var child1 = new Child();

child1.names.push('test');

console.log(child1.names); // ["xianzao", "zaoxian", "test"]

var child2 = new Child();

console.log(child2.names); // ["xianzao", "zaoxian"]
```

优点：

1. 避免了引用类型的属性被所有实例共享；
2. 可以在 Child 中向 Parent 传参；

```
function Parent (name) {
    this.name = name;
}

```

```
function Child (name) {
    Parent.call(this, name);
}

var child1 = new Child('xianzao');

console.log(child1.name); // xianzao

var child2 = new Child('zaoxian');

console.log(child2.name); // zaoxian
```

缺点：

方法都在构造函数中定义，每次创建实例都会创建一遍方法。

9.3 组合继承

```
function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {

    Parent.call(this, name);

    this.age = age;

}

Child.prototype = new Parent();
Child.prototype.constructor = Child;

var child1 = new Child('kevin', '18');

child1.colors.push('black');

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]
```

```
28
29  var child2 = new Child('daisy', '20');
30
31  console.log(child2.name); // daisy
32  console.log(child2.age); // 20
33  console.log(child2.colors); // ["red", "blue", "green"]
```

优点：融合原型链继承和构造函数的优点，是 JavaScript 中最常用的继承模式。

9.4 原型继承

```
1   function createObj(o) {
2       function F(){}
3       F.prototype = o;
4       return new F();
5   }
```

缺点：

包含引用类型的属性值始终都会共享相应的值，这点跟原型链继承一样。

```
1   var person = {
2       name: 'kevin',
3       friends: ['daisy', 'kelly']
4   }
5
6   var person1 = createObj(person);
7   var person2 = createObj(person);
8
9   person1.name = 'person1';
10  console.log(person2.name); // kevin
11
12  person1.friends.push('taylor');
13  console.log(person2.friends); // ["daisy", "kelly", "taylor"]
```

9.5 寄生式继承

创建一个仅用于封装继承过程的函数，该函数在内部以某种形式来做增强对象，最后返回对象。

```
1   function createObj (o) {
2       var clone = Object.create(o);
```

```
    clone.sayName = function () {
        console.log('hi');
    }
    return clone;
}
```