

原型、原型链、__proto__、prototype

在以类为中心的面向对象编程语言中，类和对象的关系可以想象成铸模和铸件的关系，对象总是从类中创建而来。而在原型编程的思想中，类并不是必需的，对象未必需要从类中创建而来，一个对象是通过克隆另外一个对象所得到的。

也就是说，在js中，我们是可以通过克隆创造世界，看下面的代码，首先我们创建一个构造函数

```
let User = function(name, age) {  
  this.name = name,  
  this.age = age,  
  this.getName = function() {  
    console.log(this.name)  
  }  
}
```

在这里，我们声明了一个构造函数，暂且将它想象成类，然后我们来new一个对象

```
let admin = new User('Admin', 18)
```

我们创造了一个admin对象，接着，如果有一个admin和上面所创造的对象同名同年龄时，我们该怎么创建呢，有人说：这还不简单嘛，再new一个不就得了，确实，在这里new一个难度不大，但当我们构造同一个对象需要的参数很多时，那又该怎么办，按着原来的参数继续new吗

```
let User = function(name, age, height, weight) {  
  this.name = name,  
  this.age = age,  
  this.height = height,  
  this.weight = weight,  
  //... 还有其它很多的属性  
}  
  
// 已经创建了一个对象  
let admin = new User('Admin', 18, 177, 110, ...)  
// 要再创建同个对象
```

这样的参数要我们一个一个填进去，但我们已经新建了一个完全一样的对象，而且这样子做还要防止参数一不小心填错导致对象不同

所以我们采用克隆的方式来创建一个一模一样的对象，ECMAScript 5提供了Object.create 方法，可以用来克隆对象

```
var cloneAdmin = Object.create( admin);
```

这，便是js的克隆，在这时候，cloneAdmin.__proto__会指向admin，__proto__是啥，后面我们会提到，这里你只需要知道，这时候的cloneAdmin已经继承admin几乎所有属性包括方法了

原型和原型链

我们再了解下原型和原型链，什么是原型，就像上面举的例子，cloneAdmin是通过admin克隆而来的，那么它的原型便是admin，如果现在有一个对象clone2，克隆自cloneAdmin，那么clone2的原型便是cloneAdmin，而从它到admin这一条线上的所有对象便是原型链

js的继承：基于原型链的委托机制就是原型继承的本质

如何理解这句话，我们继续上面的例子，现在cloneAdmin克隆自admin，那么它是可以使用admin的所有属性和方法的，前提是它没有覆盖掉原型的方法，这类似于我们学习其它语言的继承特性

所以我们也就可以知道：**当对象无法响应某个请求时，会把该请求委托给它自己的原型**

所以，通过以上，可以得出：原型编程范型至少包括以下基本规则

- 所有的数据都是对象
- 要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它
- 对象会记住它的原型
- 如果对象无法响应某个请求，它会把这个请求委托给它自己的原型

所有的数据都是对象

JavaScript 中的根对象是 Object.prototype 对象。Object.prototype 对象是一个空的对象。我们在 JavaScript 遇到的每个对象，实际上都是从 Object.prototype 对象克隆而来的，Object.prototype 对象就是它们的原型。

在 JavaScript 语言里，我们并不需要关心克隆的细节，因为这是引擎内部负责实现的。

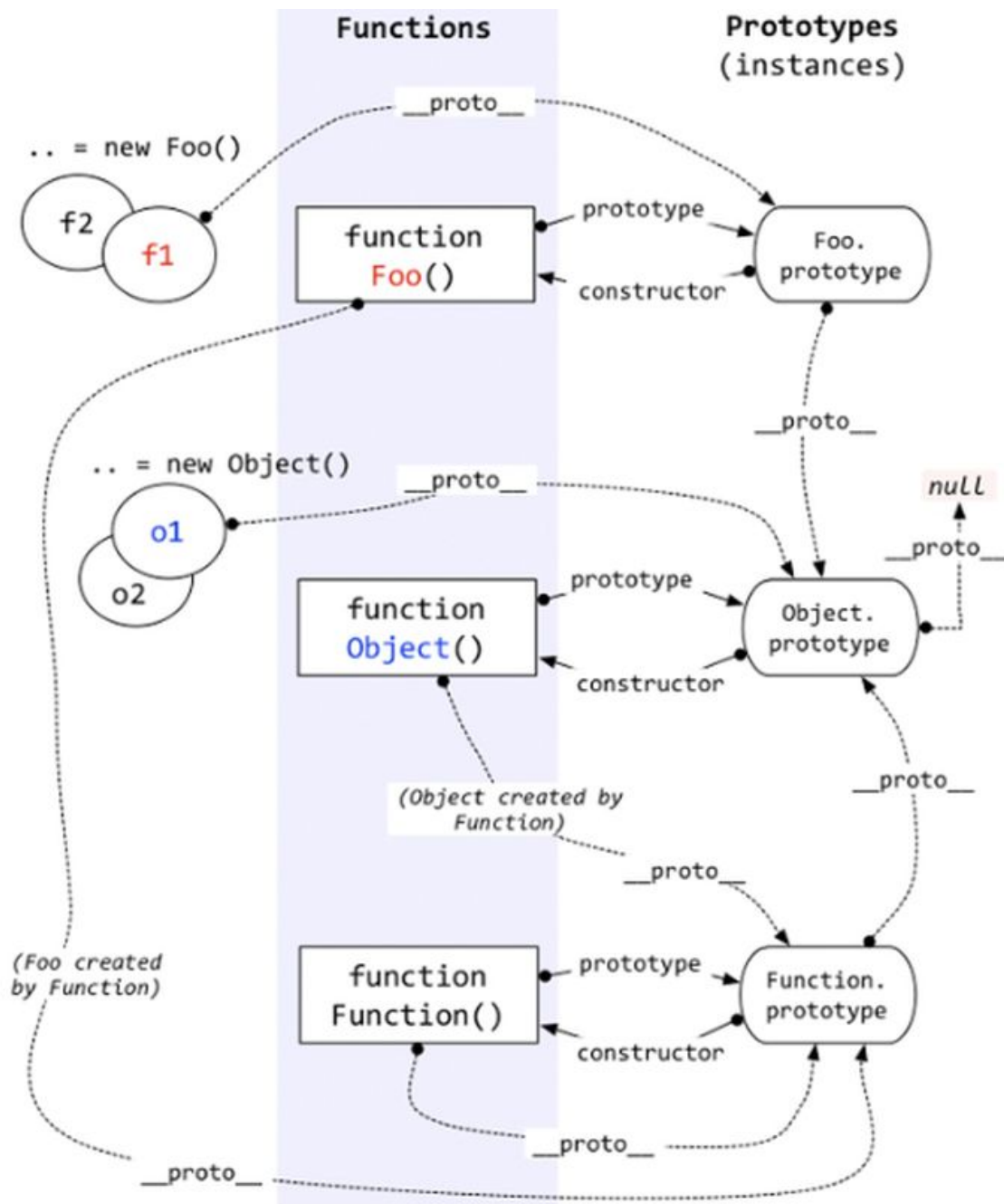
JS的克隆

JavaScript 的函数既可以作为普通函数被调用，也可以作为构造器被调用。当使用 new 运算符来调用函数时，此时的函数就是一个构造器。用 new 运算符来创建对象的过程，实际上也只是先克隆 Object.prototype 对象，再进行一些其他额外操作的过程。

对象会记住它的原型

JavaScript 给对象提供了一个名为__proto__的隐藏属性，某个对象的__proto__属性默认会指向它的构造器的原型对象，即{Constructor}.prototype。

__proto__就是对象跟“对象构造器的原型”联系起来的纽带。（__proto__：指向该对象的构造函数的原型对象，prototype指向该构造函数的原型对象——来自[知乎](#)，强烈建议阅读该文章）



对象委托

JavaScript 的对象最初都是由 `Object.prototype` 对象克隆而来的，但对象构造器的原型并不仅限于 `Object.prototype` 上，而是可以动态指向其他对象。

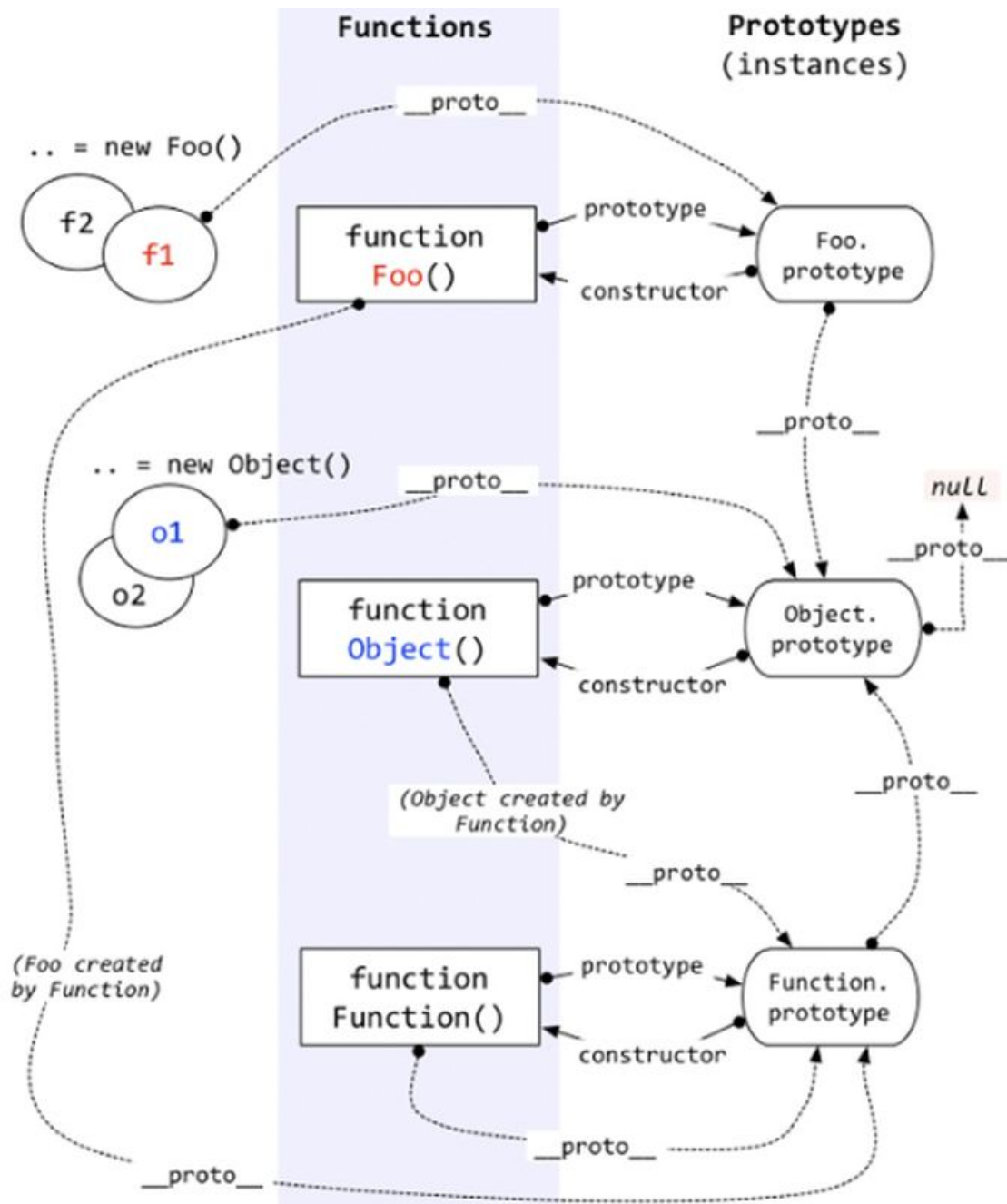
如：我们可以创建一个无原型的对象

```
// 第一参数为原型指向，第二个为对象属性编辑
let a = Object.create(null, {
  name: {
    value: 'hhh'
  }
})
```

原型链并不是无限长的，当对象通过原型链找某个属性找到根节点而找不到时，则会返回 undefined（Object.prototype 的原型是 null）

基于原型的继承以及constructor

通过上面，我们可以知道继承是可以通过原型实现，通过上面的继承原型图，我们进行相应的练习



首先创建构造函数User

```
function User (name, age) {
  this.name = name
  this.age = age
  this.getDetail = function() {
    return `我叫${this.name}, 今年${this.age}岁了`
  }
}
```

通过构造函数，我们新建一个对象zhangsan

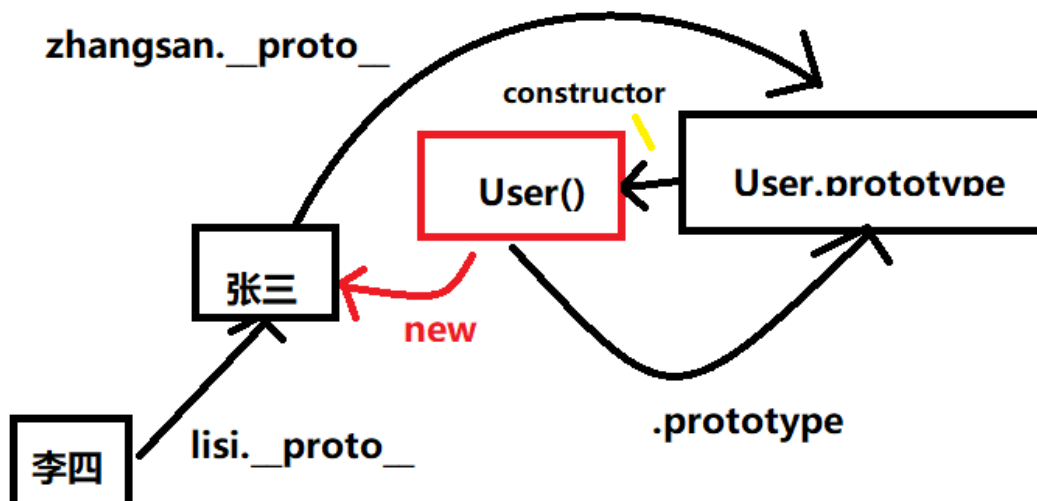
```
let zhangsan = new User('张三', 18)
```

现在我们想新建另一个对象李四，让它直接继承于张三（也就是张三的儿子），我们可以这样

```
let zhangsan = new User('张三', 18)
let lisi = {}
Object.setPrototypeOf(lisi, zhangsan)
lisi.name = '李四'
```

ok，这样我们就简单的实现了继承，然后，提出一个问题，张三、李四以及构造函数User之间的关系是怎样的（通过上图理解）

弄清楚之后，你便会会对__proto__、prototype、原型继承、原型和原型链有了更深刻的理解(下面是答案，记住一点，prototype是构造函数独有的，对象并没有这个属性，且这个属性指向构造函数原型)



通过这张图，我们也可以知道

```
User.prototype.constructor() === User()
```

也就是说，你也可以通过 `new User.prototype.constructor()` 创建对象，当然正常人不会这样做（麻烦）

构造函数/对象的原型检测

我们想要对某一个对象的原型进行判断，有两种方法，一种是通过判断构造函数，另一种则是直接判断对象

首先我们声明三个类、定义它们间的关系以及实例化出各自的对象

```
function A() {}  
function B() {}  
function C() {}  
  
let a = new A()  
B.prototype = a  
let b = new B()  
C.prototype = b  
let c = new C()
```

1. instanceof 方法（构造函数）

```
console.log(a instanceof A)  
console.log(b instanceof B)  
console.log(b instanceof A)  
console.log(c instanceof A)  
console.log(c instanceof B)  
console.log(c instanceof C)
```

结果都为真

2. `Object.prototype.isPrototypeOf`(对象)（对象）

```
console.log(a.isPrototypeOf(b))  
console.log(b.isPrototypeOf(c))  
console.log(a.isPrototypeOf(c))
```

结果也都为真

两个方法都会循着原型链向上找，唯一不同的便是instanceof需要传入构造函数，而另一个方法则是传入对象

原型链中的对象遍历

首先胡乱设置两个对象，同时这两个对象有继承关系

```
let a = {
  name: 'a',
  A() {
    return "我叫" + this.name
  }
}
let b = {
  age: 16
}

b.__proto__ = a
```

好的，也就是说a是b的爸爸，然后我们来遍历一下b对象

```
for (const key in b) {
  console.log(key)
}
```

结果是什么(name、age、A())

由此我们可以得出：in遍历是会遍历出包含原型链上其它原型的属性方法，那么如果我们只是想要遍历b独有的方法呢

这时候可以利用hasOwnProperty()方法去判断方法属性是否属于当前调用对象独有的，使用方式如下

```
被检测对象.hasOwnProperty('key')
```

所以我们就可以使用for_in循环遍历出当前对象的属性方法了

```
for (const key in b) {
  // 判断当前属性是否为当前对象独有
  if (b.hasOwnProperty( key)) {
    console.log(key)
  }
}
```

得出的结果就是b自己独有的属性方法啦

借用其他原型链方法

现在有个数组arr，当有个需求要我们求出这个数组的最大值时，你会咋做嘞

有些人可能会这样做

```
let res = arr.sort((a,b) => b - a)[0]
```

也有可能这样

```
let res = arr.reduce((v, c) => {  
  return c > v ? c : v  
}, 0)
```

但是嘞，这些都是基于数组原型的方法实现的，声明的数组类型本来就是指向该原型的，那如果这时候我想调用Math里面的方法去实现这一需求，Math.max()方法不需要上面那么复杂的逻辑实现，更简单有木有，但是要如何做呢

这时候我们就要用到apply或call方法啦（两个函数都可以改变方法的this指向）

```
let b = Math.max.apply(null, arr)
```

这样得出来的结果也是一样的，更简单明了是不，他的原理便是改变原函数max的指向，使得我们可以调用不处于同一条原型链的其它对象的方法

再来一个例子，我们利用数组的过滤来对Dom进行操作，需求是这样的：给出两个input控件，我们想要获得指定的控件

```
<input type="text" v-model="text" name="a" value="我是a">  
<input type="text" v-model="text" name="b" value="我是b">
```

我们想要获得name=a的元素，怎么做呢，很简单：1.获取所有input控件 2.调用数组的filter方法进行过滤筛选

```
let ts = document.querySelectorAll('[v-model]')  
let res = Array.prototype.filter.call(ts, e => e.getAttribute('name') === 'a');  
//下面的方法也行  
// let res = [].filter.call(ts, e => e.getAttribute('name') === 'a')  
console.log(res[0].value)
```

构造函数原型方法和构造函数内的方法

从前面我们可以知道，用构造函数实例化出来的对象，他的原型是构造函数的prototype属性，那么当我们给构造函数的prototype对象添加方法时，这个实例化出来的对象也可以使用该方法，那要是将一个对象的__proto__指向该构造函数的prototype时，它是否可以使用构造函数内部定义的方法呢，让我们看一下

首先定义一个构造函数

```
let User = function(name, age= 18) {  
  this.name = name  
  this.age = age  
  this.getName = function() {  
    console.log('我叫' + this.name)  
  }  
}
```

给它的原型新增方法

```
User.prototype.getAge = function() {  
  return this.age  
}
```

实例化一个对象，并分别调用构造函数内的方法和构造函数prototype的方法

```
let user1 = new User('user1', 19)  
  
console.log(user1.getAge())  
user1.getName()
```

结果都可以正常显示

接着我们直接定义另一个对象，并强制将其原型指向User.prototype

```
let user3 = {}  
user3.__proto__ = User.prototype
```

调用getAge()方法

```
console.log(user3.getAge())
```

正常显示，调用getName()方法

```
user3.getName()
```

无法执行，程序报错，显示该错误：user3.getName is not a function

所以由此我们可以得出，构造函数内部定义的方法，只有在由他实例化出来的对象才会得到，而将一个自定义对象原型指向该构造函数的prototype对象时，是无法使用该构造函数内部的方法的，但是毫无疑问的，它可以使用该构造函数的prototype对象内的方法

设置和获得原型

上面我们可以知道，设置对象的原型可以直接使用obj.__proto__属性强制绑定，下面是Object自带的两个方法进行原型设置以及获得当前对象的原型

```
// 设置原型
Object.setPrototypeOf(son, dad)
```

该方法和son.__proto__ = dad一致

```
// 获得原型
Object.getPrototypeOf(son)
```

使用该方法即可获得他爹了

__proto__原理 (get、set)

不知道你们有没有试过，将对象的__proto__属性定义成出对象外的其它值类型数据是不可行的

```
let obj = {}
obj.__proto__ = 3
```

这样是没有任何效果的，为什么呢，因为__proto__实际上是访问器(get/set)构造而成的，他会对设置的值进行过滤，只有符合对象类型的数据，它才会赋值，让我们来仿造一个，了解其实质

```
let Obj = {
  obj: {
    name: 'obj'
  },
  get _proto_() {
    return this.obj
  },
  set _proto_(e) {
    if(e instanceof Object) {
      this.obj = e
    }
  }
}
```

```
Obj._proto_ = 3
console.log(Obj._proto_);
```

核心代码便是 `e instanceof Object`，即判断当前传入类型是否为对象，是则赋值成功，否则忽略该赋值

基于原型面向对象的多态

js也可以多态，你没骗我吧，没有的事呢，那就让我们来看看js的多态究竟是个啥

假设现在有个情况，你家里来客人了，你爸叫你和你妹跟客人打招呼，你说了句“叔叔好，我是我爸的儿子jie（假设你叫jie）”，你妹说：“叔叔好，我是我爸的女儿hua（假设你妹叫hua）”

看到没有，同样的一个行为，哥哥和妹妹的表现是不同的，这便是多态。让我们用代码实现一下

首先定义爸爸构造函数，然后 给爸爸原型定义一个介绍的方法（毕竟指令由爹地发出的）

```
function Father() {}
Father.prototype.introduction = function() {
  console.log(this.show());
};
```

接着构造儿子，并继承于老爸，然后定义儿子的介绍方法show()

```
// 使得Son.prototype.__proto__指向Father.prototype
Son.prototype = Object.create(Father.prototype);
Son.prototype.show = function() {
  return '我是儿子' + this.name;
};
```

再来就是女儿，和儿子一样，但是其show()方法和儿子有所不同

```
function Daughter(name) {
  this.name = name;
}
Daughter.prototype = Object.create(Father.prototype);
Daughter.prototype.show = function() {
  return '我是女儿' + this.name;
};
```

然后就是实例化一个儿子和女儿

```
let jie = new Son('jie')
let hua = new Daughter('hua')
```

假设此时他们接收到了父亲的指令，于是乎分别调用introduction方法并介绍自己

```
jie.introduction() //我是儿子jie
hua.introduction() //我是女儿hua
```

这便是js的多态实现

方法重写

没啥好说的，就是继承父类的方法，但是由自己的实现方式，需要重新定义，这便是重写，直接理解代码

```
// 定义一个father构造函数
function Father() {
  this.hi = function () {
    console.log('father hi');
  };
}

// 创建son对象
let son = {};

// 实例化爹地
let father = new Father();
// 让son的原型指向爹地对象
son = Object.create(father);
son.hi()

// 重写son方法
son.hi = function () {
  console.log('son hi');
};

son.hi();
```

禁止自定义函数原型的constructor被遍历

之前我们已经了解过constructor，它是构造函数原型里面指向构造函数的一个属性，如下，两者是等价的

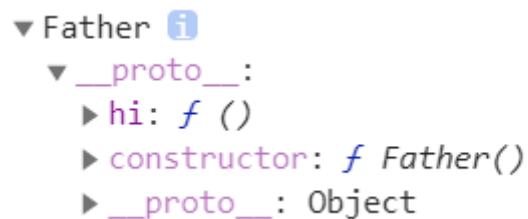
```
User.prototype.constructor === User
```

但当我们想让一个构造函数的原型继承于另一个构造函数时，会发生一点意外

```
function Father() {}
Father.prototype.hi = function () {
  console.log('father hi');
};

function Son() {
  this.name = 'son'
}
Son.prototype = Object.create(Father.prototype);
```

在这里面，我们让Son的prototype原型指向Father.prototype，看起来好像一切都正常，也不影响继承，但是仔细观察你会发现，里面的constructor不见了，通过打印 `console.dir(Son.prototype)`，我们可以发现它的constructor确实不见了



```
▼ Father ⓘ
  ▼ __proto__:
    ► hi: f ()
    ► constructor: f Father()
    ► __proto__: Object
```

这会造成什么问题，你会无法通过下列方法实例化对象

```
let son = new Son.prototype.constructor();
```

所以，为了避免它原先的constructor丢失造成的问题，我们需要把它纠正回来

```
Son.prototype.constructor = Son;
```

OK啦，只要在每次继承之后加上这句代码，就可以防止constructor丢失了，但其实在这里，还会有一个问题，让我们来遍历一下这个原型对象

```
for (const key in Son.prototype) {
  console.log(key);
}
```

我们会得到什么结果：hi以及constructor，我们希望constructor出现吗，并不会，我们希望这些原本对象自带的属性是隐藏且不可遍历的，但是在这里，我们将原本丢失的自带构造器添加上去，导致其暴露并可以遍历出来，所以我们要将其可枚举的属性设置为false，如下

```
// * 禁止自定义函数原型的constructor被遍历
Object.defineProperty(Son.prototype, 'constructor', {
  value: Son,
  enumerable: false,
});
```

所以，最后，当我们需要构造函数继承时，需要以下三个步骤

```
//继承
Son.prototype = Object.create(Father.prototype);

//恢复原有构造函数
Son.prototype.constructor = Son;

// * 禁止自定义函数原型的constructor被遍历
Object.defineProperty(Son.prototype, 'constructor', {
  value: Son,
  enumerable: false,
});
```

父类构造函数初始化属性

```
function User(name, age) {
  this.name = name;
  this.age = age;
}
User.prototype.show = function () {
  console.log(`我叫${this.name}, 我今年${this.age}岁了`);
};
```

上面是我们定义的一个父类构造函数，也就是说，有一个构造函数会继承它

```
function Admin() {}
```

如何正确继承，哎对了，三步走实现继承

```
Admin.prototype = Object.create(User.prototype);

Admin.prototype.constructor = Admin;

Object.defineProperty(Admin.prototype, 'constructor', {
  value: Admin,
  enumerable: false,
});
```

然后，我们现在想让Admin不用自定义的同时也可以同User一样初始化对象（name、age），怎么实现呢

前面有讲过的，改变User的this指向不就可以实现了吗

```
function Admin(...args) {
  User.apply(this, args);
}
```

通过args传值同时改变User实例化的this指向，从而实现Admin同User一样初始化属性

```
let mayun = new Admin('mayun', 22);
mayun.show();
```

正确输出啦

原型工厂封装属性

每次对构造函数原型进行继承，都要三步走，是不是有点麻烦，要是多个构造函数原型都要继承，岂不是要写很多次，所以我们把它给封装了吧，这样每次就可以直接调用啦

```
function extend(child, dad) {
  child.prototype = Object.create(dad.prototype);
  child.prototype.constructor = child;
  Object.defineProperty(child, 'constructor', {
    value: child,
    enumerable: false,
  });
}
```

封装后的函数也称为原型工厂，我们来看看它的使用方法把

```
extend(Son, Father);
```

OK，这样子每次继承就不会很麻烦了，直接调用该函数即可

```
extend(Daughter, Father);
```

对象工厂派生对象实现继承

我们之前已经了解过Object.create()可以实现对象继承，现在让我们来封装一个函数，来理解下构造函数实例化对象的原理，从而实现继承吧

```
function User(name, age) {  
  this.name = name;  
  this.age = age;  
}  
User.prototype.show = function () {  
  return this.name + ' is ' + this.age;  
}
```

首先我们有这么一个构造函数，我们想用它创造出许多子对象（类似于构造函数初始化），如何实现呢

第一步便是克隆出构造函数原型对象（creat）

第二步初始化对象(call、apply（前面刚讲过））

第三步便是返回这个对象啦

```
// 对象工厂造对象  
function admin(name, age) {  
  let instance = Object.create(User.prototype);  
  User.call(instance, name, age);  
  return instance;  
}
```

来实例化看看吧

```
let dy = admin('dy', 18);  
console.log(dy.show())
```

成功！

使用mixin实现多继承

在上面，我们了解到了可以借用其他原型链上的方法满足需求，但有没有其它方式可以使用其它原型链的方法呢。

"这个我知道，继承那个原型不就好嘛"，哎哟，好像有点道理哦，但这样却可能会导致继承的混乱。

我们知道，原型链为线形的，也就是说，你只能单向继承，大白话就是你只能有一个亲生爸爸，而如果这时候，跟上面的需求一样，arr数组想用Math对象中的方法时怎么办，

有小朋友提到了，我可以Math当Array的爸爸嘛，这样顺着一条原型链上去就能实现方法了嘛，是的没错，但你有没有想过在这个过程中，只是你一个数组需要用到Math中的方法，如果其他数组不需要呢，那么它们也没办法，依然会被绑定到这条原型链上，这样就会造成这条原型链混乱且复杂。

我们更加希望，事物是有序的，一条继承链下来，对象之间都是有需求的，都是相关联且合理的，所以，这个方法并不可取

那么有什么方法可以实现这一需求呢：mixin思想，我们使用混入来实现类似多继承的效果，从而使得原原型链（主干）不受影响

这里要用到Object的一个方法实现类似mixin的思想

```
Object.assign(obj1, obj2);
```

首先创建一个事物构造函数还有其原型的一个方法show

```
function Thing(name) {  
  this.name = name;  
}  
Thing.prototype.show = function () {  
  console.log(`我叫` + this.name);  
};
```

然后创造两个事物构造函数分别继承它，extend详见上面的原型工厂继承

```
function Human(name) {  
  Thing.call(this, name);  
}  
extend(Human, Thing);  
  
function Car(name) {  
  Thing.call(this, name);  
}  
extend(Car, Thing);
```

然后实例化对象并验证下是否继承到了

```
let xiaohong = new Human('xiaohong');
xiaohong.show();
let benchi = new Car('banchi');
benchi.show();
```

很幸运，可以使用父类方法展示自己（我叫xiaohong，我叫banchi）

然后我们创建一个行为类action

```
let action = {
  move() {
    console.log(this.name + '给我跑');
  },
  call() {
    console.log(this.name + '给我叫');
  },
};
```

我们想让benchi和xiaohong都能使用这个类中的方法，但是他们已经有爹地了，这时候我们用mixin思想实现

```
Object.assign(benchi, action);
Object.assign(xiaohong, action);
```

然后就执行方法吧

```
benchi.move(); //banchi给我跑
xiaohong.call(); //xiaohong给我叫
```

成功！我们实现了类似多继承

mixin的内部继承以及super

还是依照上面的例子，我们想让mixin内部的对象实现继承且，banchi和xiaohong可以调用得到，如何实现嘞

假设有个todo类，想让它当action的爸爸

```
let todo = {
  doing() {
    return '在 ';
  },
};
```

其实还是和前面一样的继承，让action的__proto__继承todo即可

```
action.__proto__ = todo;
```

这样便成功继承，然后我们稍微修改下action里面的方法，使得它可以调用父类todo的方法，这里我们用到super关键字

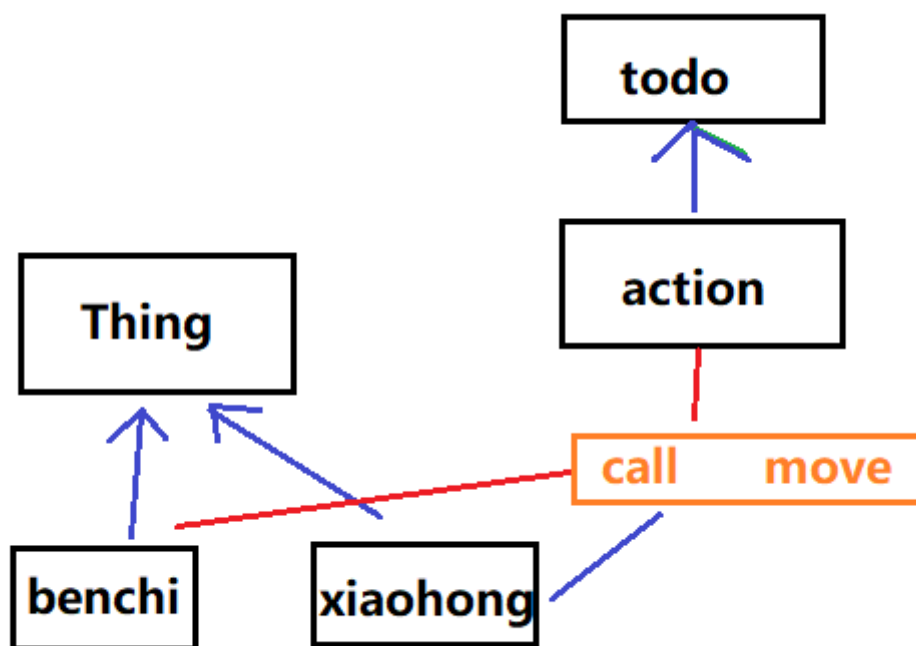
```
super === this(action).__proto__ (this指当前定义的对象)
```

```
let action = {  
  move() {  
    return this.show() + ', 我' + super.doing() + '跑';  
  },  
  call() {  
    return this.show() + ', 我' + super.doing() + '叫';  
  },  
};
```

然后同上面一样运行一下，结果也是正常显示啦

```
console.log(benchi.move());  
console.log(xiaohong.call());
```

我们来看一下他们之间的继承关系（可以多画画这种继承图帮助理解）



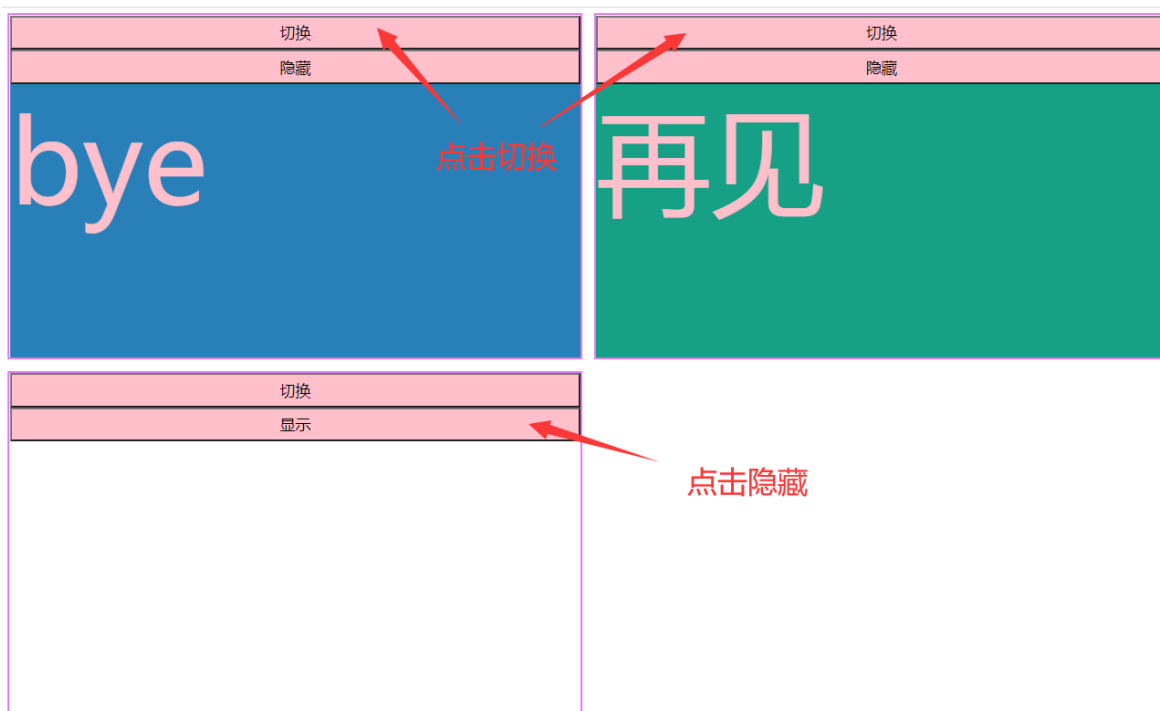
好啦，这一部分我们就了解了mixin内部继承以及super代替原型（son.__proto__）这两个重要的知识点啦

继承操作Dom（综合案例）

现在有这么一个需求，我们有三个几乎一样的组件，我们想让他们功能一致，只是样式稍微不同而已，如下图所示



三者都是点击切换按钮切换状态语且改变背景颜色，点击隐藏，下面的状态语模块消失，同时按钮提示语改变，开始动手吧



样式不说，直接上核心代码

首先创造动作构造函数，并且定义三个方法——隐藏、显示、背景颜色以及提示语改变

```
function Animation() {}
Animation.prototype.hide = function () {
  this.style.display = 'none';
};
Animation.prototype.show = function () {
  this.style.display = 'block';
};
Animation.prototype.change = function (color, value) {
  this.style.backgroundColor = color;
  this.innerHTML = value;
};
```

创造APP构造函数用于创建展示模块

```
function App(id, data={}) {
  this.div = document.querySelector(id);
  this.btnS = this.div.querySelector('[name="switch"]');
  this.btnH = this.div.querySelector('[name="hide"]');
  this.sec = this.div.querySelector('[name="sec"]');
  this.data = Object.assign({
    color: ['#8e44ad', '#16a085'],
    value: ['你好', '再见'],
    tog: true, //用于记录切换状态
  }, data)
}
```

让它继承于动作构造函数

```
extend(App, Animation);
```

接着为两个按钮注册点击事件，同时调用父类的显示隐藏以及切换背景方法

显示隐藏切换

```
App.prototype.changeIf = function () {
  this.btnH.addEventListener('click', () => {
    if (this.btnH.value === '隐藏') {
      this.hide.call(this.sec);
      this.btnH.value = '显示';
    } else {
      this.show.call(this.sec);
      this.btnH.value = '隐藏';
    }
  });
};
```

状态语切换

```
App.prototype.toggle = function () {
  this.btnS.addEventListener('click', () => {
    if (this.data.tog) {
      this.change.call(this.sec, this.data.color[1], this.data.value[1]);
      this.data.tog = !this.data.tog;
    } else {
      this.change.call(this.sec, this.data.color[0], this.data.value[0]);
      this.data.tog = !this.data.tog;
    }
  });
};
```

以及首次打开页面的初始化方法

```
App.prototype.init = function () {
  this.show.call(this.sec)
  this.change.call(this.sec, this.data.color[0], this.data.value[0])
}
```

最后就是执行这三个方法的启动函数

```
App.prototype.run = function () {
  this.toggle();
  this.changeIf();
  this.init()
};
```

然后就可以实例化对象，调用run方法啦

```
let div1 = new App('#app1', {
  color: ['#27ae60', '#2980b9'],
  value: ['hello', 'bye']
});
div1.run();

let div2 = new App('#app2');
div2.run();

let div3 = new App('#app3', {
  color: ['#d35400', '#f39c12']
})
div3.run()
```

就算你有其它需要类似功能的模块，你也可以直接实例化该对象并执行run方法，而不用每个同功能的模块书写多次相同冗余的代码了

[全部代码](#)

```

    }
  </style>
</head>
<body>
  <div id="app1">
    <input type="button" value="切换" name="switch" />
    <input type="button" value="隐藏" name="hide" />
    <section name="sec"></section>
  </div>
  <div id="app2">
    <input type="button" value="切换" name="switch" />
    <input type="button" value="隐藏" name="hide" />
    <section name="sec"></section>
  </div>
  <div id="app3">
    <input type="button" value="切换" name="switch" />
    <input type="button" value="隐藏" name="hide" />
    <nav name="sec"></nav>
  </div>
<script>
  function Animation() {}
  Animation.prototype.hide = function () {
    this.style.display = 'none';
  };
  Animation.prototype.show = function () {
    this.style.display = 'block';
  };
  Animation.prototype.change = function (color, value) {
    this.style.backgroundColor = color;
    this.innerHTML = value;
  };

  // * 原型继承工厂函数
  function extend(child, dad) {
    child.prototype = Object.create(dad.prototype);
    child.prototype.constructor = child;
    Object.defineProperty(child, 'constructor', {
      value: child,
      enumerable: false,
    });
  }

  function App(id, data={}) {
    this.div = document.querySelector(id);
    this.btnS = this.div.querySelector('[name="switch"]');
    this.btnH = this.div.querySelector('[name="hide"]');
    this.sec = this.div.querySelector('[name="sec"]');
    this.data = Object.assign({
      color: ['#8e44ad', '#16a085'],
      value: ['你好', '再见'],
      tog: true, //用于记录切换状态
    }, data)
  }

```



```

    }

    extend(App, Animation);

    App.prototype.changeIf = function () {
        this.btnH.addEventListener('click', () => {
            if (this.btnH.value === '隐藏') {
                this.hide.call(this.sec);
                this.btnH.value = '显示';
            } else {
                this.show.call(this.sec);
                this.btnH.value = '隐藏';
            }
        });
    };

    App.prototype.toggle = function () {
        this.btnS.addEventListener('click', () => {
            if (this.data.tog) {
                this.change.call(this.sec, this.data.color[1], this.data.value[1]);
                this.data.tog = !this.data.tog;
            } else {
                this.change.call(this.sec, this.data.color[0], this.data.value[0]);
                this.data.tog = !this.data.tog;
            }
        });
    };

    App.prototype.init = function () {
        this.show.call(this.sec)
        this.change.call(this.sec, this.data.color[0], this.data.value[0])
    }

    App.prototype.run = function () {
        this.toggle();
        this.changeIf();
        this.init()
    };

    let div1 = new App('#app1', {
        color: ['#27ae60', '#2980b9'],
        value: ['hello', 'bye']
    });
    div1.run();

    let div2 = new App('#app2');
    div2.run();

    let div3 = new App('#app3', {
        color: ['#d35400', '#f39c12']
    })

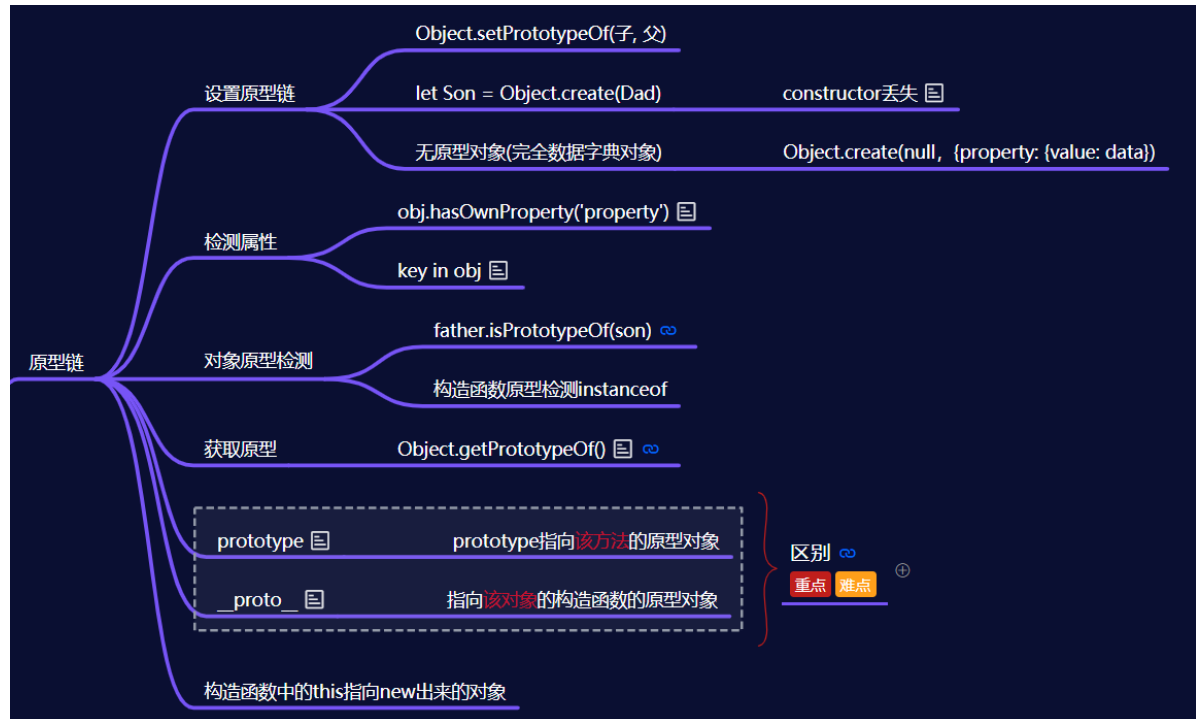
```

```

    div3.run()
  </script>
</body>
</html>

```

思维导图总结



这便是对以上全部知识点综合练习的案例了，不足之处还望批评指出，请多多指教