JS中基于原型实现继承

本文不会对于JS中的继承机制进行深入研究，只聊js中基于原型实现继承的方式，如果需要了解JS中的继承机制及其相关背景请移步javascript继承机制的设计思想;

涉及继承时不得不先了解原型这个概念，首先我们来看一段demo：

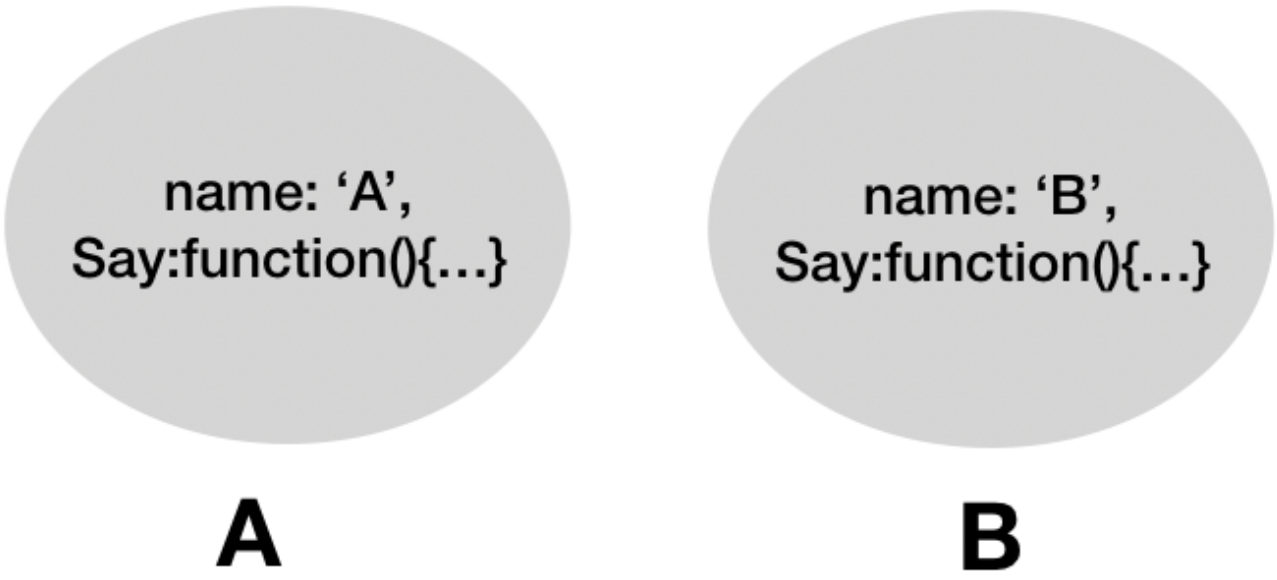
```
class Person {
  constructor(name) {
    this.name = name;
    this.say = function () {
      console.log(`my name is ${name}!`);
    }
  }
}
const A = new Person('A');
const B = new Person('B');

A.say();
console.log(A);
B.say();
console.log(B);
```

控制台输出:

```
my name is A!
▼Person {name: "A", say: f} i
    name: "A"
  ▶ say: f ()
  ▶ __proto__: Object
my name is B!
▼Person {name: "B", say: f} i
    name: "B"
  ▶ say: f ()
  ▶ __proto__: Object
> A.say === B.say;
< false
```

回顾一下这段代码：通过Person类，我们创造两个实例A、B，复杂数据类型比较比较的是其内存地址，很显然实例A的say方法与实例B的say方法指向的并不是同一个内存地址；A、B自身的属性和方法相互隔离；这是我们可以预见的结果，一切都很perfect！ 其内部存储如图所示：



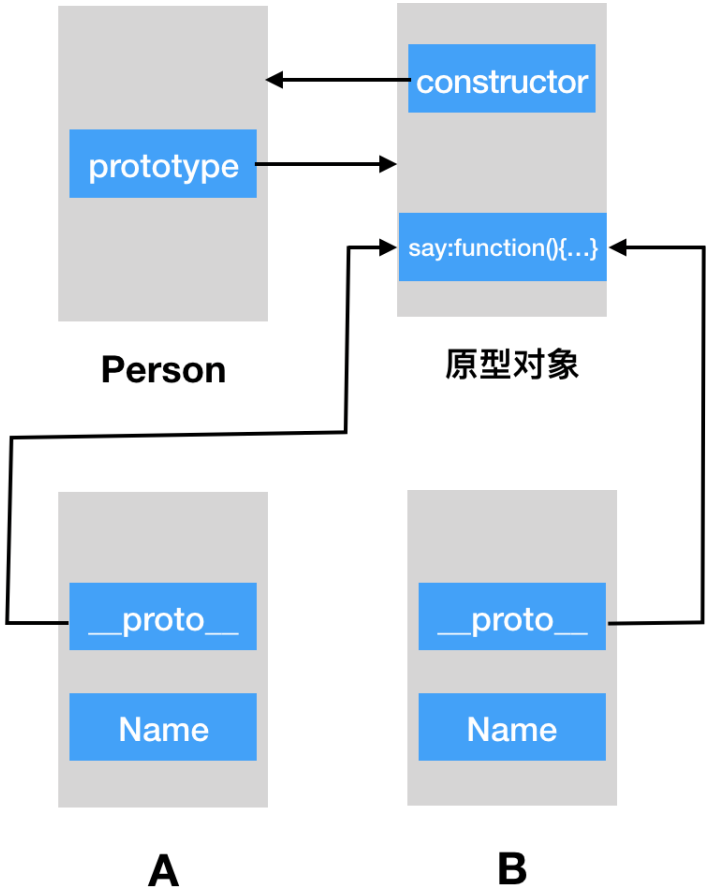
假如当我们创建批量实例时，与A,B这样类似的对象会被创建多个，say方法也会被copy多次，通过构造函数为实例对象定义属性，虽然很方便，但是有一个缺点。同一个构造函数的多个实例之间，无法共享属性，从而造成对系统资源的浪费，A与B都有相同的say，应该实现say方法共享，这样做不仅节省了内存，还体现了实例对象之间的联系！ 下面我们对demo略加改动：

```
// es6 class写法
class Person {
  constructor(name) {
    this.name = name;
  }
  say() {
    console.log(`my name is ${this.name}!`);
  }
}
const A = new Person('A');
const B = new Person('B');
A.say();
console.log(A);
B.say();
console.log(B);
// es5构造函数写法
const Person = function(name) {
    this.name = name;
}
Person.prototype.say = function() {
    console.log(`my name is ${this.name}!`);
}
```

控制台输出：

```
my name is A!
▼Person {name: "A"} i
    name: "A"
  ▶__proto__: Object
my name is B!
▼Person {name: "B"} i
    name: "B"
  ▶__proto__: Object
> A.say === B.say;
< true
```

由此可见此时A与B中的say方法引用的是同一个方法，其内存地址一致；其内部原理下图所示：




```
> A.__proto__ === Person.prototype
< true
```

我们简单回顾一下构造函数、原型和实例的关系:每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针__ proto__。本例中，我们在其原型对象上添加say方法，实例本身并没有say方法，但是当A.say()调用say方法时，其实是通过A.__ proto __.say来访问到say方法并调用之。


```
> A.say === B.say;
< true
```

在js中，某个对象能够访问到另一个对象的所有属性和方法，我们就认为二者之间存在继承关系，继承有利于代码的复用，上例中A和B除了能够访问自身的属性还能访问到Person.prototype对象中的属性和方法，这样就可以认为A就继承自Person.prototype。

基于原型实现继承主要有以下两种方式：

扩展原型对象

我们在上例中加入一行：

```
Person.prototype.sex = 'male';
```

上面代码中，向原型对象添加sex属性，两个实例对象都会具有值为male的sex属性。也就是说，当JavaScript引擎读取对象的某个属性时，JavaScript 引擎先寻找对象本身的属性，如果找不到，就到它的原型去找，如果还是找不到，就到原型的原型去找。如果直到最顶层的Object.prototype还是找不到，则返回undefined。如果对象自身和它的原型，都定义了一个同名属性，

那么优先读取对象自身的属性，这叫做“覆盖”。 控制台输出：

```
my name is A!
▼Person {name: "A"} i
    name: "A"
  ▼__proto__:
      sex: "male"
    ▶constructor: class Person
    ▶say: ƒ say()
    ▶__proto__: Object
my name is B!
▼Person {name: "B"} i
    name: "B"
  ▼__proto__:
      sex: "male"
    ▶constructor: class Person
    ▶say: ƒ say()
    ▶__proto__: Object
> A.sex
< "male"
> B.sex
< "male"
```

替换原型对象

所谓替换，顾名思义就是我们改变Person.prototype的指向,让其指向另一个对象。

```
function Person(name) {
  this.name = name;
}
Person.prototype = {
  constructor: Person,
  say: function(){
    console.log(`my name is ${this.name}!`);
  },
  run: function(){
    console.log(`I can run!`);
  }
}
const A = new Person('A');
const B = new Person('B');
```

在这里我们改变Person.prototype的指向，将其指向一个全新的原型对象，同时为保证构造函数与其原型对象结构关系的完整性，将其内部的constructor属性指向构造函数Person。 控制台输出：

```
▼Person {name: "A"} i
    name: "A"
  ▼__proto__:
    ▶constructor: ƒ Person(name)
    ▶run: ƒ ()
    ▶say: ƒ ()
    ▶__proto__: Object
▼Person {name: "B"} i
    name: "B"
  ▼__proto__:
    ▶constructor: ƒ Person(name)
    ▶run: ƒ ()
    ▶say: ƒ ()
    ▶__proto__: Object
> A.say();B.say();
  my name is A!
  my name is B!
< undefined
> A.run();B.run();
  I can run!
  I can run!
```

我们再来看下es6中基于Class实现的继承：

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  say() {
    console.log(`my name is ${this.name}!`);
  }
  run() {
    console.log('I can run!');
  }
}
class Person extends Animal {
  constructor(name){
    super(name);
  }
}
const A = new Person('A');
const B = new Person('B');
```

es6中子类继承父类以后，必须在constructor中调用时super方法，否则不能新建实例，因为子类没有属于自己的this对象，而是继承了父类的this对象对其进行加工，在这里我们让Person子类继承自Animal父类，这样Person的实例A和B就应该拥有Animal的全部属性和方法。 在Class中每一个对象都有__proto__属性，指向对应的构造函数的prototype属性。Class 作为构造函数的语法糖，同时有prototype属性和__proto__属性，因此同时存在两条继承链。

- 子类的__proto__属性，表示构造函数的继承，总是指向父类。

- 子类prototype属性的__proto__属性，表示实例方法的继承，总是指向父类的prototype属性。

```
▶ Person {name: "A"}
▶ Person {name: "B"}
> Person.__proto__ === Animal
< true
> Person.prototype.__proto__ === Animal.prototype
< true
```

子类实例的__proto__属性的__proto__属性，指向父类的prototype属性。也就是说，子类的原型的原型，是父类的原型。 控制台输出：

```
▼ Person {name: "A"} i
    name: "A"
  ▼ __proto__: Animal
    ▶ constructor: class Person
    ▼ __proto__:
      ▶ constructor: class Animal
      ▶ run: ƒ run()
      ▶ say: ƒ say()
      ▶ __proto__: Object
▼ Person {name: "B"} i
    name: "B"
  ▼ __proto__: Animal
    ▶ constructor: class Person
    ▼ __proto__:
      ▶ constructor: class Animal
      ▶ run: ƒ run()
      ▶ say: ƒ say()
      ▶ __proto__: Object
> A.__proto__.__proto__ === Animal.prototype
< true
> B.__proto__.__proto__ ===Animal.prototype
< true
```

js中除基于原型之外继承的实现方式常用的还有其他几种：

- 混入继承
- 经典继承
- 借用构造函数实现继承 有兴趣的同学可以自行研究，本文不再赘述。