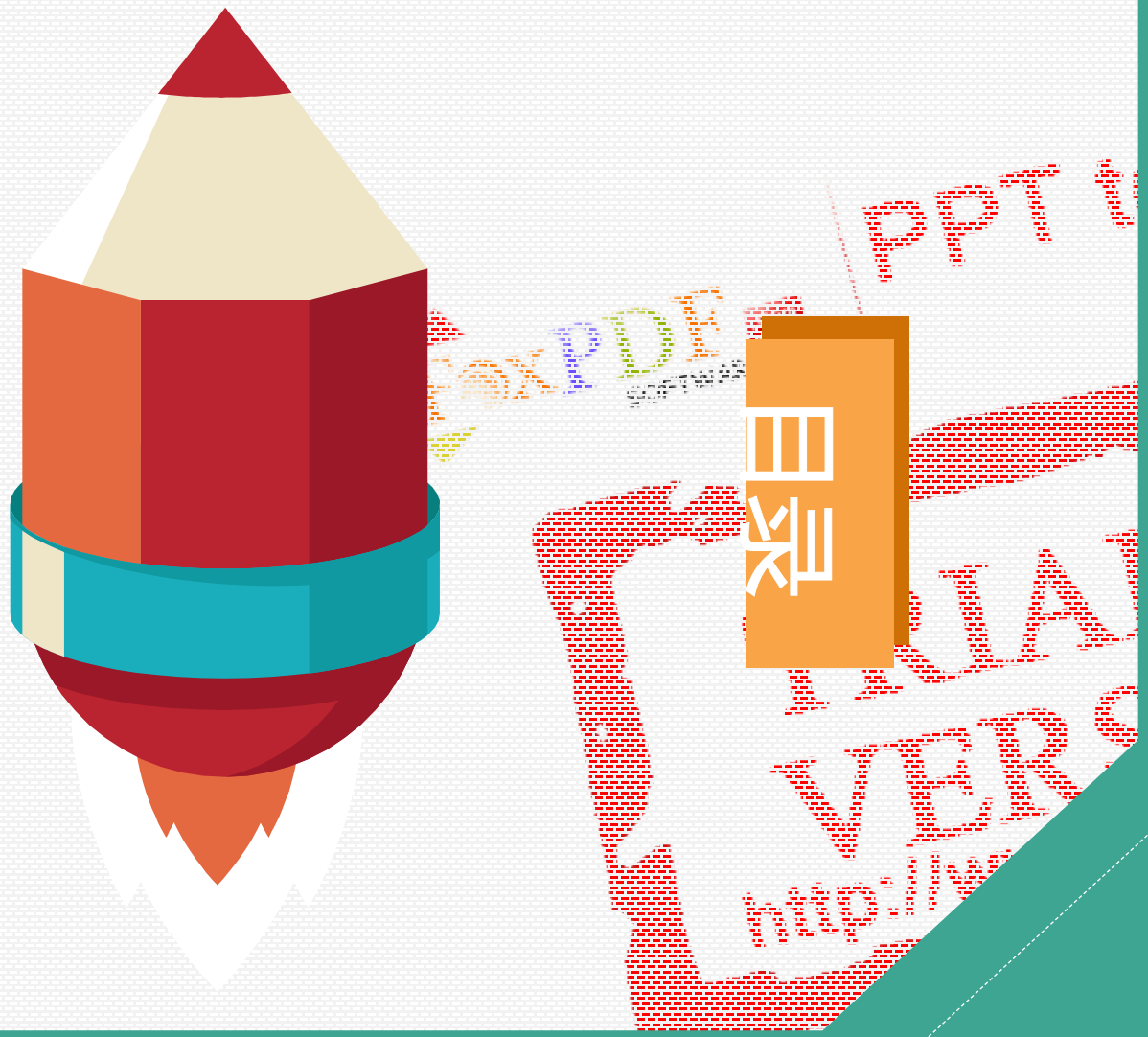

第35课：Class的继承
主讲老师：万章



- Class的extends继承
- super关键字详解
- Mixin模式的实现

Class的extends继承

Class的extends继承

```
class Point {
    constructor(x,y){
        this.x=x;
        this.y=y;
    }
    sayName(name){
        console.log(name);
    }
}

class ColorPoint extends Point {
    constructor(x,y,z){
        super(x,y);
        this.z=z;
    }
    sayHello(){
        console.log("hello world");
    }
}
```

```
> let o1=new Point(1,2);
< undefined
> let o2=new ColorPoint(1,2,3);
< undefined
> o1.sayName("万章");
  万章                                    VM1700:7
< undefined
> o2.sayName("万章");
  万章                                    VM1700:7
< undefined
```

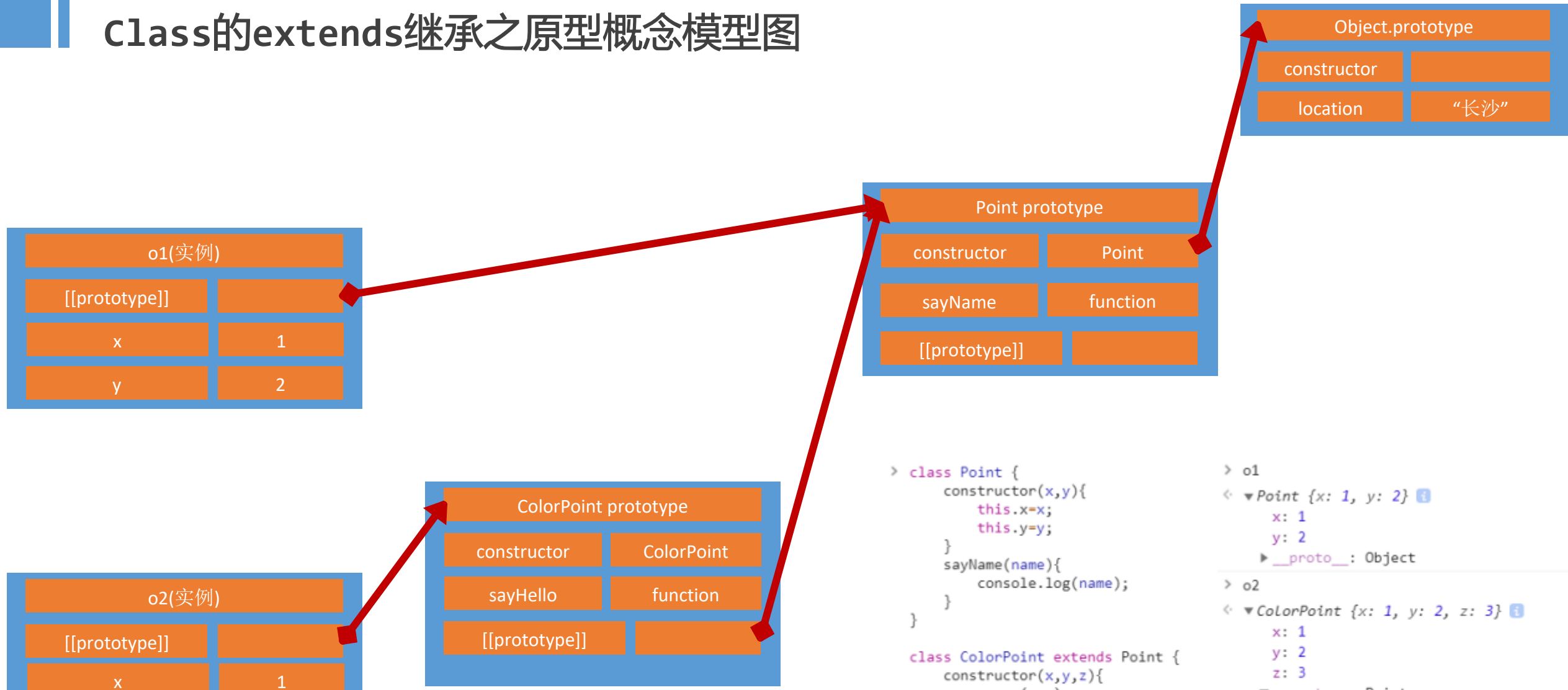
```
> o1
< ▼Point {x: 1, y: 2}
    x: 1
    y: 2
  ▶__proto__: Object
> o2
< ▼ColorPoint {x: 1, y: 2, z: 3}
    x: 1
    y: 2
    z: 3
  ▼__proto__: Point
    ▶constructor: class ColorPoint
    ▶sayHello: ƒ sayHello()
    ▼__proto__:
      ▶constructor: class Point
      ▶sayName: ƒ sayName(name)
      ▶__proto__: Object
```

o2._proto_._proto_=Point.prototype

o2._proto_=ColorPonit.prototype

上面代码定义了一个ColorPoint类，该类通过extends关键字，继承了Point类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个Point类

Class的extends继承之原型概念模型图




```
class Point {
    constructor(x,y){
        this.x=x;
        this.y=y;
    }
    sayName(name){
        console.log(name);
    }
}

class ColorPoint extends Point {
    constructor(x,y,z){
        super(x,y);
        this.z=z;
    }
    sayHello(){
        console.log("hello world");
    }
}
```

```
> o1
< ▼Point {x: 1, y: 2}
    x: 1
    y: 2
  ▶__proto__: Object
> o2
< ▼ColorPoint {x: 1, y: 2, z: 3}
    x: 1
    y: 2
    z: 3
  ▼__proto__: Point
    ▶constructor: class ColorPoint
    ▶sayHello: ƒ sayHello()
    ▼__proto__:
      ▶constructor: class Point
      ▶sayName: ƒ sayName(name)
      ▶__proto__: Object
```

Class的extends继承之原型概念

```
> class A {
  }

  class B extends A {
  }
< undefined
> B.__proto__
< class A {
  }
> B.prototype
< ▼A {constructor: ƒ}
    ▶constructor: class B
    ▶__proto__: Object
> B.prototype.__proto__
< ▼{constructor: ƒ}
    ▶constructor: class A
    ▶__proto__: Object
> A.prototype
< ▼{constructor: ƒ}
    ▶constructor: class A
    ▶__proto__: Object
>
```

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。

（1）子类的__proto__属性，表示构造函数的继承，总是指向父类。

（2）子类prototype属性的__proto__属性，表示方法的继承，总是指向父类的prototype属性。

Class的extends继承之原型概念

```
Object.setPrototypeOf = function (obj, proto) {
  obj.__proto__ = proto;
  return obj;
}
```

Object.setPrototypeOf方法的实现

```
class A {
}

class B {
}

// B 的实例继承 A 的实例
Object.setPrototypeOf(B.prototype, A.prototype);

// B 继承 A 的静态属性
Object.setPrototypeOf(B, A);

const b = new B();
```

类的继承是按照上面的模式实现的。

```
Object.setPrototypeOf(B.prototype, A.prototype);
// 等同于
B.prototype.__proto__ = A.prototype;

Object.setPrototypeOf(B, A);
// 等同于
B.__proto__ = A;
```

这两条继承链，可以这样理解：作为一个对象，子类（B）的原型（__proto__属性）是父类（A）；作为一个构造函数，子类（B）的原型对象（prototype属性）是父类的原型对象（prototype属性）的实例。

Class的extends继承之实例创造步骤

```
> class Point {
      constructor(x,y){
          this.x=x;
          this.y=y;
      }
      sayName(name){
          console.log(name);
      }
  }

  class ColorPoint extends Point {
      constructor(x,y,z){
          this.z=z;
      }
      sayHello(){
          console.log("hello world");
      }
  }
< undefined
> let o1=new Point(1,2);
< undefined
> let o2=new ColorPoint(1,2,3);
✕ ▶Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor    VM2052:13
      at new ColorPoint (<anonymous>:13:9)
      at <anonymous>:1:8
>
```

子类必须在constructor方法中调用super方法，否则新建实例时会报错。

这是因为子类自己的this对象：

1 必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法(也就是this是指父类的实例)；

2 然后再对其进行加工，加上子类自己的实例属性和方法。

如果不调用super方法，子类就得不到this对象。

Class的extends继承之实例创造中的super调用问题

```
> class Point { /* ... */ }

  class ColorPoint extends Point {
    constructor() {
    }
  }

  let cp = new ColorPoint(); // ReferenceError
```

```
▶ Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor    VM2120:4
    at new ColorPoint (<anonymous>:4:14)
    at <anonymous>:8:10
```

上面代码中，ColorPoint继承了父类Point，但是它的构造函数没有调用super方法，导致新建实例时报错。

```
> class Point { /* ... */ }

  class ColorPoint extends Point {
    constructor() {
      super();
    }
  }

  let cp = new ColorPoint(); // 这就没得问题了
undefined
> cp
▼ ColorPoint {}
  ▶ __proto__: Point
```

上面代码中，ColorPoint继承了父类Point，并且调用了super方法，成功创造实例

ES5 的继承，实质是**先创造子类的实例对象**this，然后再将父类的方法添加到this上面（Parent.apply(this)）。

ES6 的继承，实质是**先将父类实例对象的属性和方法，加到this上面**（所以必须先调用super方法），然后再用子类的构造函数修改this。

Class的extends继承之实例创造中的super调用问题

```
class ColorPoint extends Point {
}

// 等同于
class ColorPoint extends Point {
  constructor(...args) {
    super(...args);
  }
}
```

如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    this.color = color; // ReferenceError
    super(x, y);
    this.color = color; // 正确
  }
}
```

在子类的构造函数中，只有调用super之后，才可以使用this关键字，否则会报错。这是因为子类实例的构建，基于父类的实例这个，this是父类的实例，只有super方法才能调用父类实例。

注意：父类的静态方法，也会被子类继承。

super关键字详解

super关键字详解之作为函数

super这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

```
class A {}

class B extends A {
  constructor() {
    super();
  }
}
```

```
class A {
  constructor() {
    console.log(new.target.name);
  }
}
class B extends A {
  constructor() {
    super();
  }
}
new A() // A
new B() // B
```

super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。

注意，super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B的实例，因此super()在这里相当于A.prototype.constructor.call(this)。

作为函数时，super()只能用在子类的构造函数之中，用在其他地方就会报错。

super关键字详解之作为对象

super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

```
class A {
  p() {
    return 2;
  }
}

class B extends A {
  constructor() {
    super();
    console.log(super.p()); // 2
  }
}

let b = new B();
```

```
class A {
  constructor() {
    this.p = 2;
  }
}

class B extends A {
  get m() {
    return super.p;
  }
}

let b = new B();
b.m // undefined
```

```
class A {}
A.prototype.x = 2;

class B extends A {
  constructor() {
    super();
    console.log(super.x) // 2
  }
}

let b = new B();
```

上面代码中，子类B当中的super.p()，就是将super当作一个对象使用。这时，super在普通方法之中，指向A.prototype，所以super.p()就相当于A.prototype.p()。

这里需要注意，由于super指向父类的原型对象，所以**定义在父类实例上的方法或属性，是无法通过super调用的**。

super关键字详解之作为对象

```
class A {
  constructor() {
    this.x = 1;
  }
  print() {
    console.log(this.x);
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
  }
  m() {
    super.print();
  }
}

let b = new B();
b.m() // 2
```

ES6 规定，在子类普通方法中通过super调用父类的方法时，方法内部的**this指向当前的子类实例**。

```
class A {
  constructor() {
    this.x = 1;
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
    super.x = 3;
    console.log(super.x); // undefined
    console.log(this.x); // 3
  }
}

let b = new B();
```

由于this指向子类实例，所以如果通过super对某个属性赋值，这时super就是this，赋值的属性会变成子类实例的属性。

super关键字详解之作为静态方法中的对象

```
class Parent {
  static myMethod(msg) {
    console.log('static', msg);
  }

  myMethod(msg) {
    console.log('instance', msg);
  }
}

class Child extends Parent {
  static myMethod(msg) {
    super.myMethod(msg);
  }

  myMethod(msg) {
    super.myMethod(msg);
  }
}

Child.myMethod(1); // static 1

var child = new Child();
child.myMethod(2); // instance 2
```

如果super作为对象，用在静态方法之中，这时super将指向父类，而不是父类的原型对象。

```
class A {
  constructor() {
    this.x = 1;
  }
  static print() {
    console.log(this.x);
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
  }
  static m() {
    super.print();
  }
}

B.x = 3;
B.m() // 3
```

在子类的静态方法中通过super调用父类的方法时，方法内部的this指向当前的子类，而不是子类的实例。

super关键字详解之作为静态方法中的对象

注意，使用super的时候，必须显式指定是作为函数、还是作为对象使用，否则会报错。

```
class A {}

class B extends A {
  constructor() {
    super();
    console.log(super); // 报错
  }
}
```

上面代码中，console.log(super)当中的super，无法看出是作为函数使用，还是作为对象使用，所以 JavaScript 引擎解析代码的时候就会报错

```
class A {}

class B extends A {
  constructor() {
    super();
    console.log(super.valueOf() instanceof B); // true
  }
}

let b = new B();
```

上面代码中，super.valueOf()表明super是一个对象，因此就不会报错。同时，由于super使得this指向B的实例，所以super.valueOf()返回的是一个B的实例。

Mixin模式详解

Mixin模式详解

Mixin 指的是多个对象合成一个新的对象，新对象具有各个组成成员的接口。它的最简单实现如下。

```
const a = {
  a: 'a'
};
const b = {
  b: 'b'
};
const c = {...a, ...b}; // {a: 'a', b: 'b'}
```

上面代码中，c对象是a对象和b对象的合成，具有两者的接口。

Mixin模式详解

这是一个更完备的实现，将多个类的接口“混入”（mix in）另一个类。

代码中的mix函数，可以将多个对象合成为一个类。使用的时候，只要继承这个类即可。

```
class DistributedEdit extends mix(Loggable, Serializable) {
  // ...
}
```

```
function mix(...mixins) {
  class Mix {
    constructor() {
      for (let mixin of mixins) {
        copyProperties(this, new mixin()); // 拷贝实例属性
      }
    }
  }

  for (let mixin of mixins) {
    copyProperties(Mix, mixin); // 拷贝静态属性
    copyProperties(Mix.prototype, mixin.prototype); // 拷贝原型属性
  }

  return Mix;
}

function copyProperties(target, source) {
  for (let key of Reflect.ownKeys(source)) {
    if ( key !== 'constructor'
      && key !== 'prototype'
      && key !== 'name'
    ) {
      let desc = Object.getOwnPropertyDescriptor(source, key);
      Object.defineProperty(target, key, desc);
    }
  }
}
```