Class 的继承

1. 简介

Class 可以通过 `extends` 关键字实现继承，让子类继承父类的属性和方法。`extends` 的写法比 ES5 的原型链继承，要清晰和方便很多。

```
class Point { /* ... */ }
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // 调用父类的constructor(x, y)
    this.color = color;
  }
  toString() {
    return this.color + ' ' + super.toString(); // 调用父类的toString()
  }
}
```

`constructor()` 方法和 `toString()` 方法内部，都出现了 `super` 关键字。`super` 在这里表示父类的构造函数，用来新建一个父类的实例对象。

子类必须在 `constructor()` 方法中调用 `super()`，否则就会报错。因为**子类自己的 `this` 对象，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，然后再对其进行加工，添加子类自己的实例属性和方法。如果不调用 `super()` 方法，子类就得不到自己的 `this` 对象。**

```
class Point { /* ... */ }
class ColorPoint extends Point {
  constructor() {
  }
}
let cp = new ColorPoint(); // ReferenceError: Must call super constructor in
derived class before accessing 'this' or returning from derived constructor
```

`ColorPoint` 继承了父类 `Point`，但是它的构造函数没有调用 `super()`，导致新建实例时报错。

为什么子类的构造函数，一定要调用super()?

> - ES6 的继承机制，与 ES5 完全不同。
> - ES5 的继承机制，是先创造一个独立的子类的实例对象，然后再将父类的方法添加到这个对象上面，即“实例在前，继承在后”。
> - ES6 的继承机制，则是先将父类的属性和方法，加到一个空的对象上面，然后再将该对象作为子类的实例，即“继承在前，实例在后”。这就是为什么 ES6 的继承必须先调用 `super()` 方法，因为这一步会生成一个继承父类的 `this` 对象，没有这一步就无法继承父类。

这意味着新建子类实例时，父类的构造函数必定会先运行一次。

```
class Foo {
  constructor() {
    console.log(1);
  }
}
class Bar extends Foo {
  constructor() {
    super(); // 调用一次父类 Foo 的构造函数，输出 1
    console.log(2);
  }
}
const bar = new Bar();
// 1
// 2
```

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，必须先完成父类的继承，只有 `super()` 方法才能让子类实例继承父类。

```
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
  }
}
```

上面代码中，子类的 `constructor()` 方法没有调用 `super()` 之前，就使用 `this` 关键字，结果报错，而放在 `super()` 之后就是正确的。

如果子类没有定义 `constructor()` 方法，这个方法会默认添加，并且里面会调用 `super()`。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 ColorPoint extends Point {}
// 等同于
class ColorPoint extends Point {
  constructor(...args) {
    super(...args);
  }
}
```

有了子类的定义，就可以生成子类的实例了。

```
let cp = new ColorPoint(25, 8, 'green');

cp instanceof ColorPoint // true
cp instanceof Point // true
```

2. 私有属性和私有方法的继承

父类所有的属性和方法，都会被子类继承，除了私有的属性和方法。子类无法继承父类的私有属性，私有属性只能在定义它的 class 里面使用。

```
class Foo {
  #p = 1;
  #m() {
    console.log('hello');
  }
}
class Bar extends Foo {
  constructor() {
    super();
    console.log(this.#p); // Private field '#p' must be declared in an enclosing
class
    this.#m(); // Private field '#m' must be declared in an enclosing class
  }
}
```

上例中，子类 Bar 调用父类 Foo 的私有属性或私有方法，都会报错。

如果父类定义了私有属性的读写方法，子类就可以通过这些方法，读写私有属性。

```
class Foo {
  #p = 1;
  getP() {
    return this.#p;
  }
}
class Bar extends Foo {
  constructor() {
    super();
    console.log(this.getP()); // 1
  }
}
```

上例中，`getP()` 是父类用来读取私有属性的方法，通过该方法，子类就可以读到父类的私有属性。

3. 静态属性和静态方法的继承

父类的静态属性和静态方法，也会被子类继承。

```
class A {
  static hello() {
    console.log('hello world');
  }
}
class B extends A {}

B.hello()  // hello world
```

静态属性是通过浅拷贝实现继承的。

```
class A { static foo = 100; }
class B extends A {
  constructor() {
    super();
    B.foo--;
  }
}

const b = new B();
B.foo // 99
A.foo // 100
b.foo // undefined
```

上例中，foo 是 A 类的静态属性，B 类继承了 A 类，因此也继承了这个属性。但是，在 B 类内部操作 `B.foo` 这个静态属性，影响不到 `A.foo`，原因就是 B 类继承静态属性时，会采用浅拷贝，拷贝父类静态属性的值，因此A.foo和B.foo是两个彼此独立的属性。

但是，由于这种拷贝是浅拷贝，如果父类的静态属性的值是一个对象，那么子类的静态属性也会指向这个对象，因为浅拷贝只会拷贝对象的内存地址。

```
class A {
  static foo = { n: 100 };
}

class B extends A {
  constructor() {
    super();
    B.foo.n--;
  }
}

const b = new B();
B.foo.n // 99
A.foo.n // 99
```

上例中，`A.foo` 的值是一个对象，浅拷贝导致 `B.foo` 和 `A.foo` 指向同一个对象。所以，子类 B 修改这个对象的属性值，会影响到父类 A。

4. Object.getPrototypeOf()

`Object.getPrototypeOf()` 方法可以用来从子类上获取父类

```
class Point { /*...*/ }
class ColorPoint extends Point { /*...*/ }
Object.getPrototypeOf(ColorPoint) === Point; // true
```

可以使用这个方法判断，一个类是否继承了另一个类。