ES6 Class 继承与 super

ssshooter 发布于 2018-07-09

原文：https://javascript.info/class...

在博客阅读：https://ssshooter.com/2021-01...

Class 继承与 super

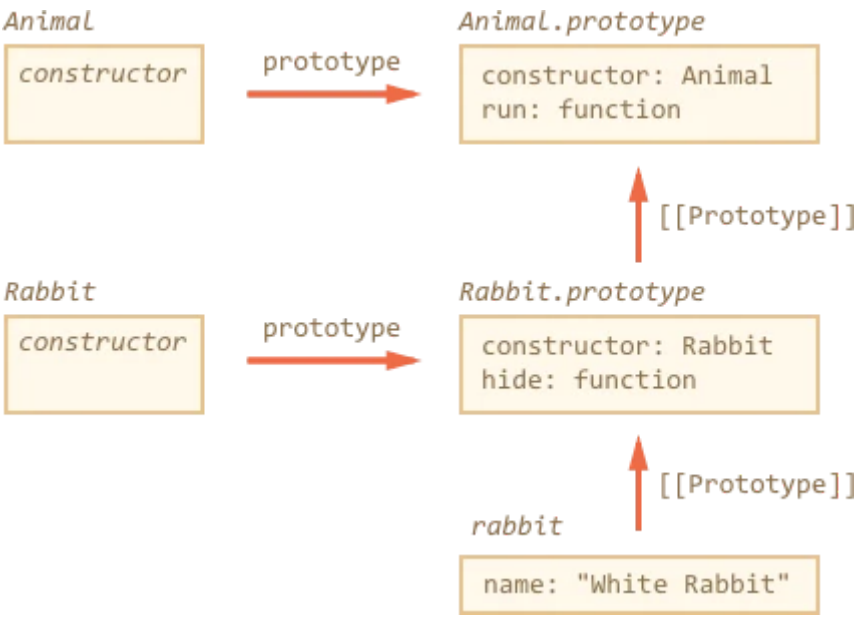
class 可以 extends 自另一个 class。这是一个不错的语法，技术上基于原型继承。

要继承一个对象，需要在 `{..}` 前指定 `extends` 和父对象。

这个 `Rabbit` 继承自 `Animal`：

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} stopped.`);
  }

}


// Inherit from Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}
```

如你所见，如你所想，`extend` 关键字实际上是在 `Rabbit.prototype` 添加 `[Prototype]]`，引用到 `Animal.prototype`。



所以现在 `rabbit` 既可以访问它自己的方法，也可以访问 `Animal` 的方法。

`extends` 后可跟表达式

Class 语法的 `extends` 后接的不限于指定一个类，更可以是表达式。

例如一个生成父类的函数：

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}


class User extends f("Hello") {}


new User().sayHi(); // Hello
```

例子中，`class User` 继承了 f('Hello')返回的结果。

对于高级编程模式，当我们使用的类是根据许多条件使用函数来生成时，这就很有用。

重写一个方法

现在让我们进入下一步，重写一个方法。到目前为止，`Rabbit` 从 `Animal` 继承了 `stop` 方法，`this.speed = 0`。

如果我们在 `Rabbit` 中指定了自己的 `stop`，那么会被优先使用：

```
class Rabbit extends Animal {
  stop() {
    // ...this will be used for rabbit.stop()
  }
}
```

.....但通常我们不想完全替代父方法，而是在父方法的基础上调整或扩展其功能。我们进行一些操作，让它之前/之后或在过程中调用父方法。

Class 为此提供 `super`关键字。

- 使用 `super.method(...)` 调用父方法。
- 使用 `super(...)` 调用父构造函数（仅在 constructor 函数中）。

例如，让兔子在 `stop` 时自动隐藏：

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} stopped.`);
  }

}

class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
```

现在，`Rabbit` 的 `stop` 方法通过 `super.stop()` 调用父类的方法。

箭头函数无 super

正如在 arrow-functions 一章中提到，箭头函数没有 `super`。

它会从外部函数中获取 `super`。例如：

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // call parent stop after 1sec
  }
}
```

箭头函数中的 `super` 与 `stop()` 中的相同，所以它按预期工作。如果我们在这里用普通函数，便会报错：

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

重写构造函数

对于构造函数来说，这有点棘手 tricky。

直到现在，`Rabbit` 都没有自己的 `constructor`。
Till now, `Rabbit` did not have its own `constructor`.

根据规范，如果一个类扩展了另一个类并且没有 `constructor`，那么会自动生成如下 `constructor`：

```
class Rabbit extends Animal {
  // generated for extending classes without own constructors

  constructor(...args) {
    super(...args);
  }

}
```

我们可以看到，它调用了父 `constructor` 传递所有参数。如果我们不自己写构造函数，就会发生这种情况。

现在我们将一个自定义构造函数添加到 `Rabbit` 中。除了`name`，我们还会设置 `earLength`：

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  // ...
}

class Rabbit extends Animal {


  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }


  // ...
}


// Doesn't work!
let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.
```

哎呦出错了！现在我们不能生成兔子了，为什么呢?

简单来说：继承类中的构造函数必须调用 `super(...)`，(!)并且在使用 `this` 之前执行它。

...但为什么？这是什么情况？嗯...这个要求看起来确实奇怪。

现在我们探讨细节，让你真正理解其中缘由 ——

在JavaScript中，继承了其他类的构造函数比较特殊。在继承类中，相应的构造函数被标记为特殊的内部属性 `[[ConstructorKind]]: “derived”`。

区别在于：

- 当一个普通的构造函数运行时，它会创建一个空对象作为 this，然后继续运行。

所以如果我们正在构造我们自己的构造函数，那么我们必须调用 `super`，否则具有 `this` 的对象将不被创建，并报错。

对于 `Rabbit` 来说，我们需要在使用 `this` 之前调用 `super()`，如下所示：

```
    constructor(name) {
      this.speed = 0;
      this.name = name;
    }

    // ...
  }

  class Rabbit extends Animal {

    constructor(name, earLength) {

      super(name);

      this.earLength = earLength;
    }

    // ...
  }


  // now fine
  let rabbit = new Rabbit("White Rabbit", 10);
  alert(rabbit.name); // White Rabbit
  alert(rabbit.earLength); // 10
```

Super 的实现与 [[HomeObject]]

让我们再深入理解 `super` 的底层实现，我们会看到一些有趣的事情。

首先要说的是，以我们迄今为止学到的知识来看，实现 super 是不可能的。

那么思考一下，这是什么原理？当一个对象方法运行时，它将当前对象作为 `this`。如果我们调用 `super.method()`，那么如何检索 `method`？很容易想到，我们需要从当前对象的原型中取出 `method`。从技术上讲，我们（或JavaScript引擎）可以做到这一点吗？

也许我们可以从 `this` 的 [[Prototype]] 中获得方法，`this.__proto__.method` 这样？对不起，这是行不通的。

在这里，`rabbit.eat()` 调用父对象的 `animal.eat()` 方法：

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {

    // that's how super.eat() could presumably work
    this.__proto__.eat.call(this); // (*)

  }
};

rabbit.eat(); // Rabbit eats.
```

在 `(*)` 这一行，我们从原型（`animal`）中取出 `eat`，并以当前对象的上下文中调用它。请注意，`.call(this)` 在这里很重要，因为只写 `this.__proto__.eat()` 的话 `eat` 的调用对象将会是 `animal`，而不是当前对象。

以上代码的 `alert` 是正确的。

但是现在让我们再添加一个对象到原型链中，就要出事了：

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...bounce around rabbit-style and call parent (animal) method
    this.__proto__.eat.call(this); // (*)
  }
};
```

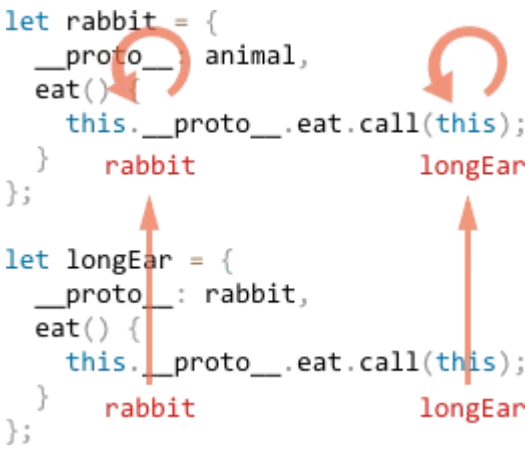
```
    __proto__: rabbit,
    eat() {
      // ...do something with long ears and call parent (rabbit) method
      this.__proto__.eat.call(this); // (**)
    }
  };


  longEar.eat(); // Error: Maximum call stack size exceeded
```

噢，完蛋！调用 `longEar.eat()` 报错了！

这原因一眼可能看不透，但如果我们跟踪 `longEar.eat()` 调用，大概就知道为什么了。在 `(*)` 和 `(**)` 两行中， `this` 的值是当前对象(`longEar`)。重点来了：所有方法都将当前对象作为 `this`，而不是原型或其他东西。

因此，在两行 `(*)` 和 `(**)` 中，`this.__proto__` 的值都是 `rabbit`。他们都调用了 `rabbit.eat`，于是就这么无限循环下去。

情况如图：



1.在 `longEar.eat()` 里面，`(**)` 行中调用了 `rabbit.eat`，并且`this = longEar`。

```
  // inside longEar.eat() we have this = longEar
  this.__proto__.eat.call(this) // (**)
  // becomes
  longEar.__proto__.eat.call(this)
  // that is
  rabbit.eat.call(this);
```

2.然后在`rabbit.eat`的 `(*)` 行中，我们希望传到原型链的下一层，但是 `this = longEar`，所以 `this .__ proto __.eat`又是 `rabbit.eat`!

```
// inside rabbit.eat() we also have this = longEar
this.__proto__.eat.call(this) // (*)
// becomes
longEar.__proto__.eat.call(this)
// or (again)
rabbit.eat.call(this);
```

1. ...因此 `rabbit.eat` 在无尽循环调动，无法进入下一层。

这个问题不能简单使用 `this` 解决。

[[HomeObject]]

为了提供解决方案，JavaScript 为函数添加了一个特殊的内部属性：`[[HomeObject]]`。

当函数被指定为类或对象方法时，其 `[[HomeObject]]` 属性为该对象。

这实际上违反了 unbind 函数的思想，因为方法记住了它们的对象。并且 `[[HomeObject]]` 不能被改变，所以这是永久 bind（绑定）。所以在 JavaScript 这是一个很大的变化。

但是这种改变是安全的。`[[HomeObject]]` 仅用于在 `super` 中获取下一层原型。所以它不会破坏兼容性。

让我们来看看它是如何在 `super` 中运作的：

```
let animal = {
  name: "Animal",
  eat() {         // [[HomeObject]] == animal
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {         // [[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Long Ear",
```

```
  }
};


longEar.eat();  // Long Ear eats.
```

每个方法都会在内部 `[[HomeObject]]` 属性中记住它的对象。然后 `super` 使用它来解析原型。

在类和普通对象中定义的方法中都定义了 `[[HomeObject]]`，但是对于对象，必须使用：`method()` 而不是 `"method: function()"`。

在下面的例子中，使用非方法语法（non-method syntax）进行比较。这么做没有设置 `[[HomeObject]]` 属性，继承也不起作用：

```
let animal = {
  eat: function() { // should be the short syntax: eat() {...}
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};


rabbit.eat();  // Error calling super (because there's no [[HomeObject]])
```

静态方法和继承

`class` 语法也支持静态属性的继承。

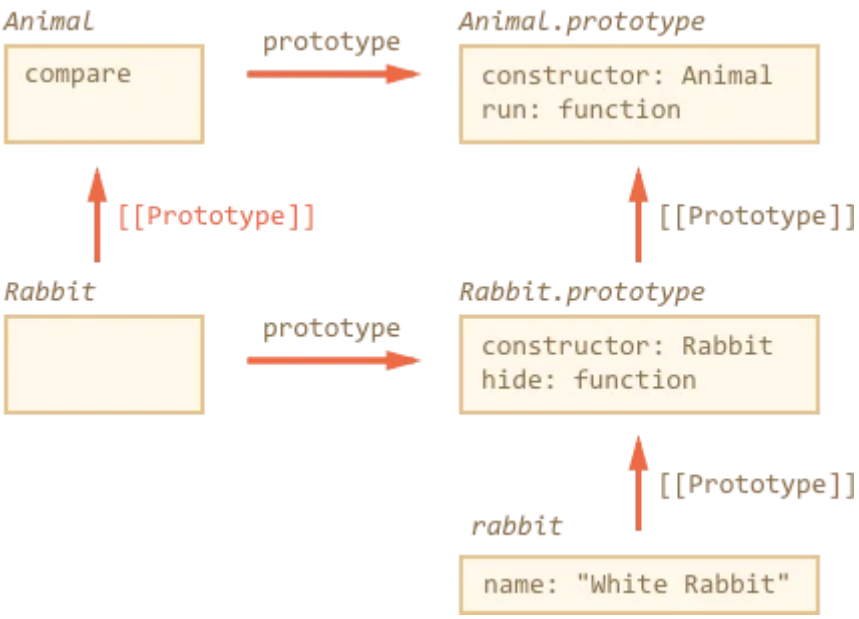
例如：

```
class Animal {

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }
```

```
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }

}

// Inherit from Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

let rabbits = [
```

现在我们可以调用 Rabbit.compare，假设继承的 Animal.compare 将被调用。

它是如何工作的？再次使用原型。正如你猜到的那样，extends 同样给 Rabbit 提供了引用到 Animal的 [Prototype]。



所以，Rabbit 函数现在继承 Animal 函数。Animal 自带引用到 Function.prototype 的 [[Prototype]]（因为它不 extend 其他类）。

看看这里：

```
class Animal {}
class Rabbit extends Animal {}
```

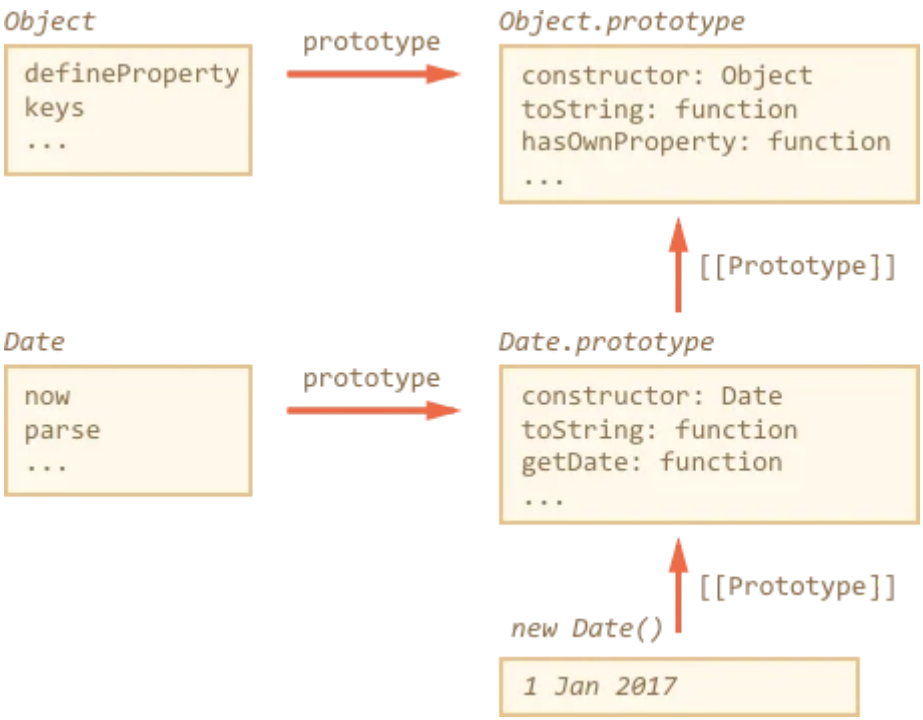
```
// and the next step is Function.prototype
alert(Animal.__proto__ === Function.prototype); // true

// that's in addition to the "normal" prototype chain for object methods
alert(Rabbit.prototype.__proto__ === Animal.prototype);
```

这样 Rabbit 可以访问 Animal 的所有静态方法。

在内置对象中没有静态继承

请注意，内置类没有静态 [[Prototype]] 引用。例如，Object 具有 Object.defineProperty，Object.keys等方法，但 Array，Date 不会继承它们。

Date 和 Object 的结构：



Date 和 Object 之间毫无关联，他们独立存在，不过 Date.prototype 继承于 Object.prototype，仅此而已。

造成这个情况是因为 JavaScript 在设计初期没有考虑使用 class 语法和继承静态方法。

原生拓展

Array，Map 等内置类也可以扩展。

举个例子，PowerArray 继承自原生 Array：

```
// add one more method to it (can do more)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

请注意一件非常有趣的事情。像 `filter`，`map` 和其他内置方法 - 返回新的继承类型的对象。他们依靠 `constructor` 属性来做到这一点。

在上面的例子中，

```
arr.constructor === PowerArray
```

所以当调用 `arr.filter()` 时，它自动创建新的结果数组，就像 `new PowerArray` 一样，于是我们可以继续使用 PowerArray 的方法。

我们甚至可以自定义这种行为。如果存在静态 getter `Symbol.species`，返回新建对象使用的 constructor。

下面的例子中，由于 `Symbol.species` 的存在，`map`，`filter`等内置方法将返回普通的数组：

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }


  // built-in methods will use this as the constructor
  static get [Symbol.species]() {
    return Array;
  }

}
```

```
// filter creates new array using arr.constructor[Symbol.species] as constructor
let filteredArr = arr.filter(item => item >= 10);


// filteredArr is not PowerArray, but Array

alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function
```

我们可以在其他 key 使用 `Symbol.species`，可以用于剥离结果值中的无用方法，或是增加其他方法。

javascript prototype class

本文系翻译，阅读原文
https://javascript.info/class-inheritance

阅读 26.4k · 更新于 2021-01-28