

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了new、this等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了class关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从ES6以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: function() {
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();
```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用Object.create来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到new和prototype属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 [[class]]，语言标准为内置类型诸如Number、String、Date等指定了[[class]]属性，以表示它们的类。语言使用者唯一可以访问[[class]]属性的方式是Object.prototype.toString。

以下代码展示了所有具有内置class属性的对象：

```
var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));
```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始, `[[class]]` 私有属性被 `Symbol.toStringTag` 代替, `Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为, 以下代码展示了使用 `Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为:

```
var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");
```

这里创建了一个新对象, 并且给它唯一的一个属性 `Symbol.toStringTag`, 我们用字符串加法触发了 `Object.prototype.toString`的调用, 发现这个属性最终对 `Object.prototype.toString` 的结果产生了影响。

但是, 考虑到JavaScript语法中跟Java相似的部分, 我们对类的讨论不能用“`new`运算是针对构造器对象, 而不是类”来试图回避。

所以, 我们仍然要把`new`理解成JavaScript面向对象的一部分, 下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数, 实际上做了几件事:

- 以构造器的 `prototype` 属性 (注意与私有字段`[[prototype]]`的区分) 为原型, 创建新对象;
- 将 `this` 和调用参数传给构造器, 执行;
- 如果构造器返回的是对象, 则返回, 否则返回第一步创建的对象。

`new` 这样的行为, 试图让函数对象在语法上跟类变得相似, 但是, 它客观上提供了两种方式, 一是在构造器中添加属性, 二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法:

```
function c1(){
  this.p1 = 1;
  this.p2 = function(){
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改`this`, 给`this`添加属性。

第二种方法是修改构造器的`prototype`属性指向的对象, 它是从这个构造器构造出来的所有对象的原型。

没有`Object.create`、`Object.setPrototypeOf`的早期版本中, `new` 运算是唯一可以指定`[[prototype]]`的方法 (当时的`mozilla`提供了私有属性`__proto__`, 但是多数环境并不支持), 所以, 当时已经有人试图用它来代替后来的 `Object.create`, 我们甚至可以用它来实现一个`Object.create`的不完整的polyfill, 见以下代码:

```
Object.create = function(prototype){
  var cls = function(){}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类, 并把传入的原型挂在了它的`prototype`, 最后创建了一个它的实例, 根据`new`的行为, 这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的`Object.create`一致, 一个是不支持第二个参数, 另一个是不支持`null`作为原型, 所以放到今天意义已经不大。

ES6 中的类

好在ES6中加入了新特性`class`, `new`跟`function`搭配的怪异行为终于可以退休了 (虽然运行时没有改变), 在任何场景, 我都推荐使用ES6的语法来定义类, 而令`function`回归原本的函数语义。下面我们就来看一下ES6中的类。

ES6中引入了`class`关键字, 并且在标准中删除了所有`[[class]]`相关的私有属性描述, 类的概念正式从属性升级成语言的基础设施, 从此, 基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```

```

  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```

```

  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了**Animal**类，并且通过**extends**关键字让**Dog**继承了它，展示了最终调用子类的**speak**方法获取了父类的**name**。

比起早期的原型模拟方式，使用**extends**关键字自动设置了**constructor**，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用**class**来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，**class**关键字和箭头运算符可以完全替代旧的**function**关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用**class**还是仍然在用**function**来定义“类”？为什么这么做？如何把使用**function**定义类的代码改造到**class**的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了new、this等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了class关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从 ES6 以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- `Object.create` 根据指定的原型创建新对象，原型可以是`null`；
- `Object.getPrototypeOf` 获得一个对象的原型；
- `Object.setPrototypeOf` 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: function() {
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();
```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```
var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));
```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```
var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");
```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调

用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“new运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把new理解成JavaScript面向对象的一部分，下面我就来讲一下new操作具体做了哪些事情。

new 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

new 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```
function c1() {
  this.p1 = 1;
  this.p2 = function() {
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2() {
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function() {
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改`this`，给`this`添加属性。

第二种方法是修改构造器的`prototype`属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有`Object.create`、`Object.setPrototypeOf`的早期版本中，`new` 运算是唯一一个可以指定`[[prototype]]`的方法（当时的mozilla提供了私有属性`__proto__`，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的 `Object.create`，我们甚至可以用它来实现一个`Object.create`的不完整的polyfill，见以下代码：

```
Object.create = function(prototype) {
  var cls = function() {}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的`prototype`，最后创建了一个它的实例，根据new的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的`Object.create`一致，一个是不支持第二个参数，另一个是不支持`null`作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性`class`，`new`跟`function`搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令`function`回归原本的函数语义。下面我们来看一下ES6中的类。

ES6中引入了`class`关键字，并且在标准中删除了所有`[[class]]`相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
// Getter
get area() {
  return this.calcArea();
}
// Method
calcArea() {
  return this.height * this.width;
}
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```

```

  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```

```

  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了**Animal**类，并且通过**extends**关键字让**Dog**继承了它，展示了最终调用子类的**speak**方法获取了父类的**name**。

比起早期的原型模拟方式，使用**extends**关键字自动设置了**constructor**，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用**class**来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，**class**关键字和箭头运算符可以完全替代旧的**function**关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用**class**还是仍然在用**function**来定义“类”？为什么这么做？如何把使用**function**定义类的代码改造到**class**的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了new、this等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了class关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从 ES6 以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是null；

- `Object.getPrototypeOf` 获得一个对象的原型；
- `Object.setPrototypeOf` 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: function() {
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();
```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```
var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));
```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```
var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");
```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

`new` 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```
function c1() {
  this.p1 = 1;
  this.p2 = function() {
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2() {
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function() {
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改`this`，给`this`添加属性。

第二种方法是修改构造器的`prototype`属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有`Object.create`、`Object.setPrototypeOf`的早期版本中，`new` 运算是唯一一个可以指定`[[prototype]]`的方法（当时的`mozilla`提供了私有属性`__proto__`，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的 `Object.create`，我们甚至可以用它来实现一个`Object.create`的不完整的`polyfill`，见以下代码：

```
Object.create = function(prototype) {
  var cls = function() {}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的`prototype`，最后创建了一个它的实例，根据`new`的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的`Object.create`一致，一个是不支持第二个参数，另一个是不支持`null`作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性`class`，`new`跟`function`搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令`function`回归原本的函数语义。下面我们来看一下ES6中的类。

ES6中引入了`class`关键字，并且在标准中删除了所有`[[class]]`相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
```

```
    calcArea() {
      return this.height * this.width;
    }
  }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```

```
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```

```
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了**Animal**类，并且通过**extends**关键字让**Dog**继承了它，展示了最终调用子类的**speak**方法获取了父类的**name**。

比起早期的原型模拟方式，使用**extends**关键字自动设置了**constructor**，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用**class**来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，**class**关键字和箭头运算符可以完全替代旧的**function**关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用**class**还是仍然在用**function**来定义“类”？为什么这么做？如何把使用**function**定义类的代码改造到**class**的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了**new**、**this**等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了class关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从ES6以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```

var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: function() {
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();

```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```

var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));

```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```

var o = { [Symbol.toStringTag]: "MyObject" };
console.log(o + "");

```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；

- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

new 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 **prototype** 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```
function c1(){
  this.p1 = 1;
  this.p2 = function(){
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改**this**，给**this**添加属性。

第二种方法是修改构造器的**prototype**属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有**Object.create**、**Object.setPrototypeOf**的早期版本中，**new** 运算是唯一一个可以指定[[**prototype**]]的方法（当时的**mozilla**提供了私有属性**__proto__**，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的 **Object.create**，我们甚至可以用它来实现一个**Object.create**的不完整的**polyfill**，见以下代码：

```
Object.create = function(prototype){
  var cls = function(){}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的**prototype**，最后创建了一个它的实例，根据**new**的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的**Object.create**一致，一个是不支持第二个参数，另一个是不支持**null**作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性**class**，**new**跟**function**搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令**function**回归原本的函数语义。下面我们就来看一下ES6中的类。

ES6中引入了**class**关键字，并且在标准中删除了所有[[**class**]]相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了**JavaScript**的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过`get/set`关键字来创建`getter`，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```

```

  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```

```

  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了`Animal`类，并且通过`extends`关键字让`Dog`继承了它，展示了最终调用子类的`speak`方法获取了父类的`name`。

比起早期的原型模拟方式，使用`extends`关键字自动设置了`constructor`，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用`class`来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，`class`关键字和箭头运算符可以完全替代旧的`function`关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用`class`还是仍然在用`function`来定义“类”？为什么这么做？如何把使用`function`定义类的代码改造到`class`的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了`new`、`this`等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了`class`关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java 等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的 JavaScript 就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的 JavaScript 并非第一个使用原型的语言，在它之前，self、kevo 等语言已经开始使用原型来描述对象了。

事实上，Brendan 更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的 scheme 语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在 JavaScript 之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是 Brendan 选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript 显然选择了前一种方式。

JavaScript 的原型

如果我们抛开 JavaScript 用于模拟 Java 类的复杂语法设施（如 new、Function Object、函数的 prototype 属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在 ES 的各个历史版本中并没有很大改变，但从 ES6 以来，JavaScript 提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是 null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}
```

```

}

var tiger = Object.create(cat, {
  say:{
    writable:true,
    configurable:true,
    enumerable:true,
    value:function(){
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();

```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```

var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));

```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```

var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");

```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

`new` 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```
function c1(){
  this.p1 = 1;
  this.p2 = function(){
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改**this**，给**this**添加属性。

第二种方法是修改构造器的**prototype**属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有**Object.create**、**Object.setPrototypeOf**的早期版本中，**new**运算是唯一一个可以指定[[**prototype**]]的方法（当时的**mozilla**提供了私有属性**__proto__**，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的**Object.create**，我们甚至可以用它来实现一个**Object.create**的不完整的**polyfill**，见以下代码：

```
Object.create = function(prototype){
  var cls = function(){}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的**prototype**，最后创建了一个它的实例，根据**new**的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的**Object.create**一致，一个是不支持第二个参数，另一个是不支持**null**作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性**class**，**new**跟**function**搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令**function**回归原本的函数语义。下面我们来看一下ES6中的类。

ES6中引入了**class**关键字，并且在标准中删除了所有[[**class**]]相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```



```
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```



```
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了`Animal`类，并且通过`extends`关键字让`Dog`继承了它，展示了最终调用子类的`speak`方法获取了父类的`name`。

比起早期的原型模拟方式，使用`extends`关键字自动设置了`constructor`，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用`class`来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，`class`关键字和箭头运算符可以完全替代旧的`function`关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用`class`还是仍然在用`function`来定义“类”？为什么这么做？如何把使用`function`定义类的代码改造到`class`的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了`new`、`this`等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了`class`关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看

出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从ES6以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
```

```

        enumerable:true,
        value:function(){
            console.log("roar!");
        }
    }
})

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();

```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```

var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));

```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```

var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");

```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

`new` 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```

function c1(){
    this.p1 = 1;
    this.p2 = function(){
        console.log(this.p1);
    };
}

```

```

    }
}
var o1 = new c1;
o1.p2();

function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
    console.log(this.p1);
}

var o2 = new c2;
o2.p2();

```

第一种方法是直接在构造器中修改**this**，给**this**添加属性。

第二种方法是修改构造器的**prototype**属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有**Object.create**、**Object.setPrototypeOf**的早期版本中，**new** 运算是唯一一个可以指定[[**prototype**]]的方法（当时的**mozilla**提供了私有属性__**proto**__，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的 **Object.create**，我们甚至可以用它来实现一个**Object.create**的不完整的**polyfill**，见以下代码：

```

Object.create = function(prototype){
    var cls = function(){}
    cls.prototype = prototype;
    return new cls;
}

```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的**prototype**，最后创建了一个它的实例，根据**new**的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的**Object.create**一致，一个是不支持第二个参数，另一个是不支持**null**作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性**class**，**new**跟**function**搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令**function**回归原本的函数语义。下面我们就来看一下ES6中的类。

ES6中引入了**class**关键字，并且在标准中删除了所有[[**class**]]相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```

class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    // Getter
    get area() {
        return this.calcArea();
    }
    // Method
    calcArea() {
        return this.height * this.width;
    }
}

```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

```

class Animal {
    constructor(name) {
        this.name = name;
    }
}

```

```
    speak() {
      console.log(this.name + ' makes a noise.');
```

```
    }
  }

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```

```
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了`Animal`类，并且通过`extends`关键字让`Dog`继承了它，展示了最终调用子类的`speak`方法获取了父类的`name`。

比起早期的原型模拟方式，使用`extends`关键字自动设置了`constructor`，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用`class`来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，`class`关键字和箭头运算符可以完全替代旧的`function`关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用`class`还是仍然在用`function`来定义“类”？为什么这么做？如何把使用`function`定义类的代码改造到`class`的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了`new`、`this`等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了`class`关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java 等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的 JavaScript 就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的 JavaScript 并非第一个使用原型的语言，在它之前，self、kevo 等语言已经开始使用原型来描述对象了。

事实上，Brendan 更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的 scheme 语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在 JavaScript 之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是 Brendan 选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript 显然选择了前一种方式。

JavaScript 的原型

如果我们抛开 JavaScript 用于模拟 Java 类的复杂语法设施（如 new、Function Object、函数的 prototype 属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在 ES 的各个历史版本中并没有很大改变，但从 ES6 以来，JavaScript 提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是 null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: function() {
      console.log("roar!");
    }
  }
})
```

```

})

var anotherCat = Object.create(cat);

anotherCat.say();

var anotherTiger = Object.create(tiger);

anotherTiger.say();

```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```

var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));

```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```

var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");

```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

`new` 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```

function c1(){
  this.p1 = 1;
  this.p2 = function(){
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();

```

```
function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
  console.log(this.p1);
}

var o2 = new c2;
o2.p2();
```

第一种方法是直接在构造器中修改**this**，给**this**添加属性。

第二种方法是修改构造器的**prototype**属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有**Object.create**、**Object.setPrototypeOf**的早期版本中，**new** 运算是唯一一个可以指定[[**prototype**]]的方法（当时的**mozilla**提供了私有属性**__proto__**，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的 **Object.create**，我们甚至可以用它来实现一个**Object.create**的不完整的**polyfill**，见以下代码：

```
Object.create = function(prototype){
  var cls = function(){}
  cls.prototype = prototype;
  return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的**prototype**，最后创建了一个它的实例，根据**new**的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的**Object.create**一致，一个是不支持第二个参数，另一个是不支持**null**作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性**class**，**new**跟**function**搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令**function**回归原本的函数语义。下面我们就来看一下ES6中的类。

ES6中引入了**class**关键字，并且在标准中删除了所有[[**class**]]相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了**JavaScript**的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上**JavaScript**认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

  speak() {
    console.log(this.name + ' makes a noise.');
```

```
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
```



```
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

以上代码创造了**Animal**类，并且通过**extends**关键字让**Dog**继承了它，展示了最终调用子类的**speak**方法获取了父类的**name**。

比起早期的原型模拟方式，使用**extends**关键字自动设置了**constructor**，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用**class**来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，**class**关键字和箭头运算符可以完全替代旧的**function**关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用**class**还是仍然在用**function**来定义“类”？为什么这么做？如何把使用**function**定义类的代码改造到**class**的新语法？

欢迎给我留言，我们一起讨论。

早期的JavaScript程序员一般都有过使用JavaScript“模拟面向对象”的经历。

在上一篇文章我们已经讲到，JavaScript本身就是面向对象的，它并不需要模拟，只是它实现面向对象的方式和主流的流派不太一样，所以才让很多人产生了误会。

那么，随着我们理解的思路继续深入，这些“模拟面向对象”，实际上做的事情就是“模拟基于类的面向对象”。

尽管我认为，“类”并非面向对象的全部，但我们不应该责备社区出现这样的方案，事实上，因为一些公司的政治原因，JavaScript推出之时，管理层就要求它去模仿Java。

所以，JavaScript创始人Brendan Eich在“原型运行时”的基础上引入了**new**、**this**等语言特性，使之“看起来语法更像Java”，而Java正是基于类的面向对象的代表语言之一。

但是JavaScript这样的半吊子模拟，缺少了继承等关键特性，导致大家试图对它进行修补，进而产生了种种互不相容的解决方案。

庆幸的是，从ES6开始，JavaScript提供了**class**关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，但是它修正了之前的一些常见的“坑”，统一了社区的方案，这对语言的发展有着非常大的好处。

实际上，我认为“基于类”并非面向对象的唯一形态，如果我们把视线从“类”移开，Brendan当年选择的原型系统，就是一个非常优秀的抽象对象的形式。

我们从头讲起。

什么是原型？

原型是顺应人类自然思维的产物。中文中有个成语叫做“照猫画虎”，这里的猫看起来就是虎的原型，所以，由此我们可以看出，用原型来描述对象的方法可以说是古已有之。

我们在上一节讲解面向对象的时候提到了：在不同的编程语言中，设计者也利用各种不同的语言特性来抽象描述对象。

最为成功的流派是使用“类”的方式来描述对象，这诞生了诸如 C++、Java等流行的编程语言。这个流派叫做基于类的编程语言。

还有一种就是基于原型的编程语言，它们利用原型来描述对象。我们的JavaScript就是其中代表。

“基于类”的编程提倡使用一个关注分类和类之间关系开发模型。在这类语言中，总是先有类，再从类去实例化一个对象。类与类之间又可能会形成继承、组合等关系。类又往往与语言的类型系统整合，形成一定编译时的能力。

与此相对，“基于原型”的编程看起来更为提倡程序员去关注一系列对象实例的行为，而后才去关心如何将这些对象，划分到最近的使用方式相似的原型对象，而不是将它们分成类。

基于原型的面向对象系统通过“复制”的方式来创建新对象。一些语言的实现中，还允许复制一个空对象。这实际上就是创建一个全新的对象。

基于原型和基于类都能够满足基本的复用和抽象需求，但是适用的场景不太相同。

这就像专业人士可能喜欢在看到老虎的时候，喜欢用猫科豹属豹亚种来描述它，但是对一些不那么正式的场合，“大猫”可能更为接近直观的感受一些（插播一个冷知识：比起老虎来，美洲狮在历史上相当长时间都被划分为猫科猫属，所以性格也跟猫更相似，比较亲人）。

我们的JavaScript并非第一个使用原型的语言，在它之前，self、kevo等语言已经开始使用原型来描述对象了。

事实上，Brendan更是曾透露过，他最初的构想是一个拥有基于原型的面向对象能力的scheme语言（但是函数式的部分是另外的故事，这篇文章里，我暂时不做详细讲述）。

在JavaScript之前，原型系统就更多与高动态性语言配合，并且多数基于原型的语言提倡运行时的原型修改，我想，这应该是Brendan选择原型系统很重要的理由。

原型系统的“复制操作”有两种实现思路：

- 一个是并不真的去复制一个原型对象，而是使得新对象持有一个原型的引用；
- 另一个是切实地复制对象，从此两个对象再无关联。

历史上的基于原型语言因此产生了两个流派，显然，JavaScript显然选择了前一种方式。

JavaScript的原型

如果我们抛开JavaScript用于模拟Java类的复杂语法设施（如new、Function Object、函数的prototype属性等），原型系统可以说相当简单，我可以用两条概括：

- 如果所有对象都有私有字段[[prototype]]，就是对象的原型；
- 读一个属性，如果对象本身没有，则会继续访问对象的原型，直到原型为空或者找到为止。

这个模型在ES的各个历史版本中并没有很大改变，但从ES6以来，JavaScript提供了一系列内置函数，以便更为直接地访问操纵原型。三个方法分别为：

- Object.create 根据指定的原型创建新对象，原型可以是null；
- Object.getPrototypeOf 获得一个对象的原型；
- Object.setPrototypeOf 设置一个对象的原型。

利用这三个方法，我们可以完全抛开类的思维，利用原型来实现抽象和复用。我用下面的代码展示了用原型来抽象猫和虎的例子。

```
var cat = {
  say() {
    console.log("meow~");
  },
  jump() {
    console.log("jump");
  }
}

var tiger = Object.create(cat, {
  say:{
    writable:true,
    configurable:true,
    enumerable:true,
    value:function(){
      console.log("roar!");
    }
  }
});

var anotherCat = Object.create(cat);

anotherCat.say();
```

```
var anotherTiger = Object.create(tiger);

anotherTiger.say();
```

这段代码创建了一个“猫”对象，又根据猫做了一些修改创建了虎，之后我们完全可以用`Object.create`来创建另外的猫和虎对象，我们可以通过“原始猫对象”和“原始虎对象”来控制所有猫和虎的行为。

但是，在更早的版本中，程序员只能通过Java风格的类接口来操纵原型运行时，可以说非常别扭。

考虑到`new`和`prototype`属性等基础设施今天仍然有效，而且被很多代码使用，学习这些知识也有助于我们理解运行时的原型工作原理，下面我们试着回到过去，追溯一下早年的JavaScript中的原型和类。

早期版本中的类与原型

在早期版本的JavaScript中，“类”的定义是一个私有属性 `[[class]]`，语言标准为内置类型诸如`Number`、`String`、`Date`等指定了 `[[class]]` 属性，以表示它们的类。语言使用者唯一可以访问 `[[class]]` 属性的方式是 `Object.prototype.toString`。

以下代码展示了所有具有内置`class`属性的对象：

```
var o = new Object;
var n = new Number;
var s = new String;
var b = new Boolean;
var d = new Date;
var arg = function(){ return arguments }();
var r = new RegExp;
var f = new Function;
var arr = new Array;
var e = new Error;
console.log([o, n, s, b, d, arg, r, f, arr, e].map(v => Object.prototype.toString.call(v)));
```

因此，在ES3和之前的版本，JS中类的概念是相当弱的，它仅仅是运行时的一个字符串属性。

在ES5开始，`[[class]]` 私有属性被 `Symbol.toStringTag` 代替，`Object.prototype.toString` 的意义从命名上不再跟 `class` 相关。我们甚至可以自定义 `Object.prototype.toString` 的行为，以下代码展示了使用`Symbol.toStringTag`来自定义 `Object.prototype.toString` 的行为：

```
var o = { [Symbol.toStringTag]: "MyObject" }
console.log(o + "");
```

这里创建了一个新对象，并且给它唯一的一个属性 `Symbol.toStringTag`，我们用字符串加法触发了`Object.prototype.toString`的调用，发现这个属性最终对`Object.prototype.toString`的结果产生了影响。

但是，考虑到JavaScript语法中跟Java相似的部分，我们对类的讨论不能用“`new`运算是针对构造器对象，而不是类”来试图回避。

所以，我们仍然要把`new`理解成JavaScript面向对象的一部分，下面我就来讲一下`new`操作具体做了哪些事情。

`new` 运算接受一个构造器和一组调用参数，实际上做了几件事：

- 以构造器的 `prototype` 属性（注意与私有字段`[[prototype]]`的区分）为原型，创建新对象；
- 将 `this` 和调用参数传给构造器，执行；
- 如果构造器返回的是对象，则返回，否则返回第一步创建的对象。

`new` 这样的行为，试图让函数对象在语法上跟类变得相似，但是，它客观上提供了两种方式，一是在构造器中添加属性，二是在构造器的 `prototype` 属性上添加属性。

下面代码展示了用构造器模拟类的两种方法：

```
function c1(){
  this.p1 = 1;
  this.p2 = function(){
    console.log(this.p1);
  }
}
var o1 = new c1;
o1.p2();
```

```
function c2(){
}
c2.prototype.p1 = 1;
c2.prototype.p2 = function(){
```

```
        console.log(this.p1);
    }

    var o2 = new c2;
    o2.p2();
```

第一种方法是直接在构造器中修改**this**，给**this**添加属性。

第二种方法是修改构造器的**prototype**属性指向的对象，它是从这个构造器构造出来的所有对象的原型。

没有**Object.create**、**Object.setPrototypeOf**的早期版本中，**new**运算是唯一一个可以指定[[**prototype**]]的方法（当时的**mozilla**提供了私有属性__proto__，但是多数环境并不支持），所以，当时已经有人试图用它来代替后来的**Object.create**，我们甚至可以用它来实现一个**Object.create**的不完整的polyfill，见以下代码：

```
Object.create = function(prototype) {
    var cls = function() {}
    cls.prototype = prototype;
    return new cls;
}
```

这段代码创建了一个空函数作为类，并把传入的原型挂在了它的**prototype**，最后创建了一个它的实例，根据**new**的行为，这将产生一个以传入的第一个参数为原型的对象。

这个函数无法做到与原生的**Object.create**一致，一个是不支持第二个参数，另一个是不支持**null**作为原型，所以放到今天意义已经不大了。

ES6 中的类

好在ES6中加入了新特性**class**，**new**跟**function**搭配的怪异行为终于可以退休了（虽然运行时没有改变），在任何场景，我都推荐使用ES6的语法来定义类，而令**function**回归原本的函数语义。下面我们就来看一下ES6中的类。

ES6中引入了**class**关键字，并且在标准中删除了所有[[**class**]]相关的私有属性描述，类的概念正式从属性升级成语言的基础设施，从此，基于类的编程方式成为了JavaScript的官方编程范式。

我们先看下类的基本写法：

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    // Getter
    get area() {
        return this.calcArea();
    }
    // Method
    calcArea() {
        return this.height * this.width;
    }
}
```

在现有的类语法中，**getter/setter**和**method**是兼容性最好的。

我们通过**get/set**关键字来创建**getter**，通过括号和大括号来创建方法，数据型成员最好写在构造器里面。

类的写法实际上也是由原型运行时来承载的，逻辑上JavaScript认为每个类是有共同原型的一组对象，类中定义的方法和属性则会被写在原型对象之上。

此外，最重要的是，类提供了继承能力。我们来看一下下面的代码。

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

    speak() {
        console.log(this.name + ' makes a noise.');
```

```
class Dog extends Animal {
    constructor(name) {
        super(name); // call the super class constructor and pass in the name parameter
    }
}
```

```
    speak() {  
      console.log(this.name + ' barks.');    }  
  }  
}  
  
let d = new Dog('Mitzie');  
d.speak(); // Mitzie barks.
```

以上代码创造了`Animal`类，并且通过`extends`关键字让`Dog`继承了它，展示了最终调用子类的`speak`方法获取了父类的`name`。

比起早期的原型模拟方式，使用`extends`关键字自动设置了`constructor`，并且会自动调用父类的构造函数，这是一种更少坑的设计。

所以当我们使用类的思想来设计代码时，应该尽量使用`class`来声明类，而不是用旧语法，拿函数来模拟对象。

一些激进的观点认为，`class`关键字和箭头运算符可以完全替代旧的`function`关键字，它更明确地区分了定义函数和定义类两种意图，我认为这是有一定道理的。

总结

在新的ES版本中，我们不再需要模拟类了：我们有了光明正大的新语法。而原型体系同时作为一种编程范式和运行时机制存在。

我们可以自由选择原型或者类作为代码的抽象风格，但是无论我们选择哪种，理解运行时的原型系统都是很有必要的一件事。

在你的工作中，是使用`class`还是仍然在用`function`来定义“类”？为什么这么做？如何把使用`function`定义类的代码改造到`class`的新语法？

欢迎给我留言，我们一起讨论。