你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代

码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有"constructor()"方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法"constructor()"。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写"constructor()"方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个"constructor()"方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用`super()`调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过`return`来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的`this`；
- 返回通过`return`传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类`X`，却又需要`X.prototype`来为实例`this`设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建`this`之外，还需要调用一个为实例`x`置它的类原型`X.prototype`的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于`X.prototype`是子类通过`super()`传递来的，因此作为父类的`MyClass`中通常需要处理的代码，就变成了为`this`引用置`new.target.prototype`这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有`this`这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为`extends`这个声明置`null`值时，由于`extends`声明`MyClass`派生自`null`（也就是没有原型），那么在构造方法中也是不能调用`super()`的。并且由于没有原型，JavaScript引擎也不会缺省为这个`MyClass`创建`this`实例。所以，在这个“`constructor()`”构造方法中，既没有`this`也不能调用`super()`。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个`this`实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在`super()`的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
```

```
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有"constructor()"方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法"constructor()"。因为多数情况下，类主要是定义它的实例的那些

性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用`super()`。**至于原因，则是非常明显的，因为“创建`this`实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}
```

```
class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类x，却又需要x.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原

因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
```

```
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- `new.target`为什么称为元属性，它与`a.b`（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用`Object.create()`来创建的对象。

因为用到了`return`这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了`return`，它还用到了一个称为元属性（*meta property*）的东西，也就是`new.target`。

迄今为止，`new.target`是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码`new X`为例），对象`this`实际上是由new运算依据`X.prototype`来创建的。循此前例，ECMAScript 6中的类，在创建`this`对象时也需要这个`X.prototype`来作为原型。

但是，按照ECMAScript 6的设计，创建这个`this`对象的行为与权力，将通过`super()`被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道`new X`运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类`x`，却又需要`x.prototype`来为实例`this`设置原型。**

ECMAScript为此提出了`new.target`这个东西，它就指向上面的`X`，并且随着`super()`调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的`Date()`为例，它的构建过程必然包括“类似于”如下两行代码来处理`this`：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
```

```
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为`_newTarget_`的额外参数的原因。`new.target`这个元属性，事实上就是在构造过程中，在`super()`调用的参数界面上传递的。只不过你在构造方法中写`super()`的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是`super()`在帮助你传递这个`new.target`参数！**

那么，如果函数中没有调用`super()`呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
```

```
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己

任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- `new.target`为什么称为元属性，它与`a.b`（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用`Object.create()`来创建的对象。

因为用到了`return`这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了`return`，它还用到了一个称为元属性（*meta property*）的东西，也就是`new.target`。

迄今为止，`new.target`是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码`new X`为例），对象`this`实际上是由new运算依据`X.prototype`来创建的。循此前例，ECMAScript 6中的类，在创建`this`对象时也需要这个`X.prototype`来作为原型。

但是，按照ECMAScript 6的设计，创建这个`this`对象的行为与权力，将通过`super()`被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道`new X`运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类x，却又需要x.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这

个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这

个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类x，却又需要x.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
```

```
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码`new X`为例），对象`this`实际上是由new运算依据`X.prototype`来创建的。循此前例，ECMAScript 6中的类，在创建`this`对象时也需要这个`X.prototype`来作为原型。

但是，按照ECMAScript 6的设计，创建这个`this`对象的行为与权力，将通过`super()`被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道`new X`运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类x，却又需要x.prototype来为实例this设置原型。**

ECMAScript为此提出了`new.target`这个东西，它就指向上面的`X`，并且随着`super()`调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的`Date()`为例，它的构建过程必然包括“类似于”如下两行代码来处理`this`：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为`_newTarget_`的额外参数的原因。`new.target`这个元属性，事实上就是在构造过程中，在`super()`调用的参数界面上传递的。只不过你在构造方法中写`super()`的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用`super()`呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是

一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}
```

```
// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己

创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括“类似于”如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用`super()`。**至于原因，则是非常明显的，因为“创建`this`实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有`new.target`的。

然而为什么呢？`new.target`是需要用`super()`来传递的呀？！

是的，这两种函数与类的确不调用`super()`，但这只说明它不需要向父类传递`new.target`而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些`new.target`的。

所以，你所提出的问题还有第二个答案：

如果是不使用`super()`调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过`return`来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有"constructor()"方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法"constructor()"。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写"constructor()"方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个"constructor()"方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在"constructor()"方法内部无法访问它自身，不能写出类似"*constructor*===*MyClass*"这样的检

测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有"constructor()"方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法"constructor()"。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用**super()**的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用`super()`。**至于原因，则是非常明显的，因为“创建`this`实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有`new.target`的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
```

```
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用`super()`。**至于原因，则是非常明显的，因为“创建`this`实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。

二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- new.target为什么称为元属性，它与a.b（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用Object.create()来创建的对象。

因为用到了return这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了return，它还用到了一个称为元属性（*meta property*）的东西，也就是new.target。

迄今为止，new.target是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码new X为例），对象this实际上是由new运算依据X.prototype来创建的。循此前例，ECMAScript 6中的类，在创建this对象时也需要这个X.prototype来作为原型。

但是，按照ECMAScript 6的设计，创建这个this对象的行为与权力，将通过super()被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道new X运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类X，却又需要X.prototype来为实例this设置原型。**

ECMAScript为此提出了new.target这个东西，它就指向上面的X，并且随着super()调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的Date()为例，它的构建过程必然包括"类似于"如下两行代码来处理this：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是`super()`在帮助你传递这个`new.target`参数！**

那么，如果函数中没有调用`super()`呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor*===*MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”`extends`声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的`this`”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}

function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
```

```
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- `new.target`为什么称为元属性，它与`a.b`（例如super.xxx，或者'a'.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。

今天这一讲的标题呢，比较长。它是我这个专栏中最长的标题了。不过说起来，这个标题的意义还是很简单的，就是返回一个用`Object.create()`来创建的对象。

因为用到了`return`这个子句，所以它显然应该是一个函数中的退出代码，是不能在函数外单独使用的。

这个函数呢，必须是一个构造器。更准确地说，标题中的代码必须工作在构造过程之中。因为除了`return`，它还用到了一个称为元属性（*meta property*）的东西，也就是`new.target`。

迄今为止，`new.target`是JavaScript中唯一的一个元属性。

为什么需要定义自己的构建过程

通过之前的课程，你应该知道：JavaScript使用原型继承来搭建自己的面向对象的继承体系，在这个过程中诞生了两种方法：

1. 使用一般函数的构造器；
2. 使用ECMAScript 6之后的类。

从根底上来说，这两种方法的构建过程都是在JavaScript引擎中事先定义好了的，例如在旧式风格的构造器中（以代码`new X`为例），对象`this`实际上是由new运算依据`X.prototype`来创建的。循此前例，ECMAScript 6中的类，在创建`this`对象时也需要这个`X.prototype`来作为原型。

但是，按照ECMAScript 6的设计，创建这个`this`对象的行为与权力，将通过`super()`被层层转交，直到父类或祖先类中有能力创建该对象的那个构造器或类为止。而在这时，父类是不可能知道`new X`运算中的这个子类为何的，因为父类通常是更早先被声明出来的。既然它的代码一早就被决定了，那么对子类透明也就是正常的了。

于是真正的矛盾在这时候就出现了：**父类并不知道子类`x`，却又需要`x.prototype`来为实例`this`设置原型。**

ECMAScript为此提出了`new.target`这个东西，它就指向上面的`X`，并且随着`super()`调用一层层地向上传递，以便最终创建者类可以使用它。也就是说，以之前讨论过的`Date()`为例，它的构建过程必然包括“类似于”如下两行代码来处理`this`：

```
// 在JavaScript内置类Date()中可能的处理逻辑
function _Date() {
  this = Object.Create(Date.prototype, { _internal_slots });
  Object.setPrototypeOf(this, new.target.prototype);
  ...
}
```

1. 依据父类的原型，也就是Date.prototype来创建对象实例this，因为它是父类创建出来的；
2. 置this实例的原型为子类的prototype，也就是new.target.prototype，因为它最终是子类的实例。

这也就是为什么Proxy()类的construct句柄与Reflect.construct()方法中都需要传递一个称为_newTarget_的额外参数的原因。new.target这个元属性，事实上就是在构造过程中，在super()调用的参数界面上传递的。只不过你在构造方法中写super()的时候，是JavaScript引擎隐式地帮你传递了这个参数而已。

你可能已经发现了问题的关键：**是super()在帮助你传递这个new.target参数！**

那么，如果函数中没有调用super()呢？

先补个课：关于隐式的构造方法

在之前的课程中我提及过，当类声明中没有“constructor()”方法时，JavaScript会主动为它创建一个。关于这一点当时并没有展开来细讲，所以这里先补个课。

首先，你通常写一个类的时候，都不太会主动去声明构造方法“constructor()”。因为多数情况下，类主要是定义它的实例的那些性质，例如方法或属性存取器。极端的情况下，你也可能只写一个空的类，只是为了将父类做一次简单的派生。例如：

```
class MyClass extends Object {}
```

无论是哪种情况，总之**你就是没有写“constructor()”方法**。有趣的是，事实上JavaScript初始化出来的这个MyClass类，（它作为一个函数）就是指向那个“constructor()”方法的，两者是同一个东西。

不过，这一点不太容易证实。因为在“constructor()”方法内部无法访问它自身，不能写出类似“*constructor===MyClass*”这样的检测条件来。所以，你只能在ECMAScript的规范文档中去确认这一点。

那么，既然MyClass就是constructor()方法，而用户代码又没有声明这个方法。那么该怎么办呢？

ECMAScript规范就约定，在这种情况下，引擎需要向用户代码中插入一段硬代码。也就是帮你写一个缺省的构造方法，然后引擎为这个硬代码的代码文本动态地生成一个“构造方法”声明，最后再将它初始化为类MyClass()。这里的“硬代码”包括两个代码片断，分别对应于“有/没有”extends声明的情况。如下：

```
// 如果在class声明中有extends XXX
class MyClass extends XXX {
  // 自动插入的缺省构造方法
  constructor(...args) {
    super(...args);
  }
  ...
}

// 如果在class声明中没有声明extends
class MyClass {
  // 自动插入的缺省构造方法
  constructor() {}
  ...
}
```

在声明中如果有extends语法的话，缺省构造方法中就插入一个SuperCall()；而如果声明中没有extends，那么缺省构造方法就是一段空的代码，什么也没有。

所以，现在你看到了你所提出的问题的第一个答案：

如果没有声明构造方法（因此没有super()调用），那么就让引擎偷偷声明一个。

非派生类是不用调用super()的

另一种特殊情况就是上面的这种非派生类，也就在类声明中语法中没有“extends XXX”的这种情况。上面的硬代码中，JavaScript引擎为它生成的就是一个空的构造方法，目的呢，也就是为了创建类所对应的那个函数体。并且，貌似别无它用。

这种非派生类的声明非常特别，本质上来说，它是兼容旧的JavaScript构造器声明的一种语法。也就是说，如果“extends XXX”不声明，那么空的构造方法和空的函数一样；并且即使是声明了具体的构造方法，那么它的行为也与传统的构造函数一样。

为了这种一致性，当这种非派生类的构造方法返回无效值时，它和传统的构造函数也会发生相同的行为——“返回已创建的this”。例如：

```
class MyClass extends Object {
  constructor() {
    return 1;
  }
}
```

```
function MyConstructor() {
  return 1;
}
```

测试如下：

```
> new MyClass;
{}

> new MyConstructor;
{}
```

这样的相似性还包括一个重要的、与今天讨论的主题相关的特性：**非派生类也不需要调用super()。**至于原因，则是非常明显的，因为“创建this实例”的行为是由引擎隐式完成的，对于传统的构造器是这样，对于非派生类的构造方法，也是这样。二者的行为一致。

那么这种情况下还有没有“new.target”呢？事实是：

在传统的构造函数和非派生类的构造方法中，一样是有new.target的。

然而为什么呢？new.target是需要用super()来传递的呀？！

是的，这两种函数与类的确不调用super()，但这只说明它不需要向父类传递new.target而已。要知道，当它自己作为父类时，还是需要接受由它的子类传递来的那些new.target的。

所以，你所提出的问题还有第二个答案：

如果是不使用super()调用的类或构造器函数，那么可以让它做根类（祖先类）。

定制的构造方法

你应该还记得，上面这两种情况的类或构造器函数都是可以通过return来返回值的。之前的课程中也一再强调过：

- 在这样的类中返回非对象值，那么就默认替换成已创建的this；
- 返回通过return传出的对象（也就是一个用户定制的创建过程）。

所以如果是用户定制的创建过程，那么就回到了最开始的那个问题上：

父类并不知道子类X，却又需要X.prototype来为实例this设置原型。

因此事实上如果用户要在“根类/祖先类”的层级上实现一个定制过程，并且还需要返回一个子类所需要的实例，那么它除了自己创建this之外，还需要调用一个为实例x置它的类原型X.prototype的过程：

```
// 参见本讲开始的_Date()过程
Object.setPrototypeOf(x, X.prototype)
```

由于X.prototype是子类通过super()传递来的，因此作为父类的MyClass中通常需要处理的代码，就变成了为this引用置new.target.prototype这个原型。

```
// （也就是）
Object.setPrototypeOf(this, new.target.prototype);
```

然而还有一种更加特殊的情况：类的构造方法中也可能没有this这个引用。

```
class MyClass extends null {
  constructor() {
    ...
  }
}
```

例如，当你为extends这个声明置null值时，由于extends声明MyClass派生自null（也就是没有原型），那么在构造方法中也是不能调用super()的。并且由于没有原型，JavaScript引擎也不会缺省为这个MyClass创建this实例。所以，在这个“constructor()”构造方法中，既没有this也不能调用super()。

怎么办呢？

你必须确信这样的类只能用作根类（显然，它不是任何东西派生出来的子类）。因此，在语义上，它可以自己创建一个实例。也就是说，这样的根类之所以存在的目的，就是用来替代本讲前面讨论的所有过程，以为“它的子类创建一个this实例”为己任。因此，完整实现这一目的的最简单方式，就是本讲标题中的这一行代码：

```
class MyClass extends null {
  constructor() {
```

```
    return Object.create(new.target.prototype);
  }
}

// 测试
console.log(new MyClass);  // MyClass {}
console.log(new (class MyClassEx extends MyClass{})); // MyClassEx {}
```

所以，仅仅是这样的一行代码，就几乎已经穷尽了JavaScript类构建过程的全部秘密。

其他

当然如果父类并不关心子类实例的原型，那么它返回任何的对象都是可以的，子类在super()的返回中并不检查原型继承链的维护情况。也就是说，确实存在“子类创建出非该类的实例”的情况。例如：

```
class MyClass {
  constructor() { return new Date };
}

class MyClassEx extends MyClass {
  constructor() { super() }; // or default
  foo() {
    console.log('check only');
  }
}

var x = new MyClassEx;
console.log(x instanceof MyClassEx); // false
console.log('foo' in x); // fals
```

今天的内容就到这里。有关继承、原型与类的所有内容就暂时告一段落了。下一讲开始，我将侧重为你介绍对象的本质，以及它的应用。

思考题

当然，这一讲仍然会留有一个习题。仅仅一个而已：

- `new.target`为什么称为元属性，它与`a.b`（例如super.xxx，或者’a’.toString）有什么不同？

希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。