你不知道的JavaScript（上卷）

Kyle Simpson、赵望野、梁杰

1.1 编译原理

2019-02-24 02:53:17

对于JavaScript来说，大部分情况下编译发生在代码执行前的几微秒（甚至更短！）的时间内。

1.2.1 演员表

2019-02-24 02:55:27

引擎

2019-02-24 02:55:32

编译器

2019-02-24 02:55:36

作用域

1.2.4 引擎和作用域的对话

2019-02-24 03:10:07

function foo(a) {

```
console.log( a ); // 2
}
foo( 2 );
```

2019-02-24 03:10:12

引擎：我说作用域，我需

2019-02-24 03:10:16

要为foo进行RHS引用。你见过它吗?
作用域：别说，我还真见过，编译器那小子刚刚声明了它。它是一个函数，给你。
引擎：哥们太够意思了！好吧，我来执行一下foo。
引擎：作用域，还有个事儿。我需要为a进行LHS引用，这个你见过吗?
作用域：这个也见过，编译器最近把它声名为foo的一个形式参数了，拿去吧。
引擎：大恩不言谢，你总是这么棒。现在我要把2赋值给a。

2019-02-24 03:10:22

引擎：哥们，不好意思又来打扰你。我要为console进行RHS引用，你见过它吗?
作用域：咱俩谁跟谁啊，再说我就是干这个的。这个我也有，console是个内置对象。给你。
引擎：么么哒。我得看看这里面是不是有log(..)。太好了，找到了，是一个函数。
引擎：哥们，能帮我再找一下对a的RHS引用吗？虽然我记得它，但想再确认一次。
作用域：放心吧，这个变量没有变动过，拿走，不谢。
引擎：真棒。我来把a的

2019-02-24 03:10:32

值，也就是2，传递进log(..)。
……

2.1 词法阶段

2019-02-24 09:52:06

它首先从最内部的作用域，也就是bar(..)函数的作用域气泡开始查找。引擎无法在这里找到a，因此会去上一级到所嵌套的foo(..)的作用域中继续查找。在这里找到了a，因此引擎使用了这个引用。对b来讲也是一样的。而对c来说，引擎在bar(..)中就找到了它。

2019-02-24 09:49:53

作用域查找会在找到第一个匹配的标识符时停止。在多层的嵌套作用域中可以定义同名的标识符，这叫作“遮蔽效应”（内部的标识符“遮蔽”了外部的标识符）。抛开遮蔽效应，作用域查找始终从运行时所处的最内部作用域开始，逐级向外或者说向上进行，直到遇见第一个匹配的标识符为止。

2019-02-24 09:53:31

无论函数在哪里被调用，也无论它如何被调用，它的词法作用域都只由函数被声明时所处的位置决定。

2019-02-24 09:55:16

词法作用域查找只会查找一级标识符，比如a、b和c。如果代码中引用了foo.bar.baz，词法作用域查找只会试图查找foo标识符，找到这个变量后，对象属性访问规则会分别接管对bar和baz属性的访问。

2.2.1 eval

2019-02-24 09:58:33

eval(..)调用中的"var b = 3;"这段代码会被当作本来就在那里一样来处理。由于那段代码声明了一个新的变量b，因此它对已经存在的foo(..)的词法作用域进行了修改。事实上，和前面提到

2019-02-24 09:58:37

的原理一样，这段代码实际上在foo(..)内部创建了一个变量b，并遮蔽了外部（全局）作用域中的同名变量。

2019-02-24 09:59:21

在严格模式的程序中，eval(..)在运行时有其自己的词法作用域，意味着其中的声明无法修改所在的作用域

2019-02-24 10:00:00

JavaScript中还有其他一些功能效果和eval(..)很相似。setTimeout(..)和setInterval(..)的第一个参数可以是字符串，字符串的内容可以被解释为一段动态生成的函数代码。这些功能已经过时且并不被提倡。不要使用它们！

2019-02-24 10:02:16

在程序中动态生成代码的使用场景非常罕见，因为它所带来的好处无法抵消性能上的损失。

2.2.2 with

2019-02-24 10:11:36

但是可以注意到一个奇怪的副作用，实际上a = 2赋值操作创建了

2019-02-24 10:11:45

一个全局的变量a。

2019-02-24 10:13:29

eval(..)函数如果接受了含有一个或多个声明的代码，就会修改其所处的词法作用域，而with声明实际上是根据你传递给它的对象凭空创建了一个全新的词法作用域。

2019-02-24 10:16:08

另外一个不推荐使用eval(..)和with的原因是会被严格模式所影响（限制）。with被完全禁止，而在保留核心功能的前提下，间接或非安全地使用eval(..)也被禁止

2019-02-24 10:16:12

了。

2.2.3 性能

2019-02-24 10:19:05

最悲观的情况是如果出现了eval(..)或with，所有的优化可能都是无意义的，因此最简单的做法

2019-02-24 10:19:12

就是完全不做任何优化。

2019-02-24 10:19:33

如果代码中大量使用eval(..)或with，那么运行起来一定会变得非常慢。无论引擎多聪明，试图将这些悲观情况的副作用限制在最小范围内，也无法避免如果没有这些优化，代码会运行得更慢这个事实。

2.3 小结

2019-02-24 10:20:32

词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。编译的词法分析阶段基本能够知道全部标识符在哪里以及是如何声明的，从而能够预测在执行过程中如何对它们进行查找。

3.1 函数中的作用域

2019-02-24 11:47:35

最常见的答案是JavaScript具有基于函数的作用域，意味着每声明一个函数都会为其自身创建一个气泡，而其他结构都不会创建作用域气泡。但事实上这并不完全正确，下面我们来看一下。

2019-02-24 11:52:28

函数作用域的含义是指，属于这个函数的全部变量都可以在整个函数的范围内使用及复用（事实上在嵌套的作用域中也可以使用）

3.2 隐藏内部实现

2019-02-24 11:58:41

可以把变量和函数包裹在一个函数的作用域中，然后用这个作用域来“隐藏”它们。

2019-02-24 11:58:56

有很多原因促成了这种基于作用域的隐藏方法。它们大都是从最小特权原则中引申出来的，也叫最小授权或最小暴露原则

规避冲突

2019-02-24 12:08:39

显而易见，这些工具并没有能够违反词法作用域规则的“神奇”功能。它们只是利用作用域的规则强制所有标识符都不能注入到共享作用域中，而是保持在私有、无冲突的作用域中，这样可以有效规避掉所有的意外冲突。

3.3 函数作用域

2019-02-24 12:10:57

```
var a = 2;
(function foo(){ // <-- 添加这一行
var a = 3;
console.log( a ); // 3
})(); // <-- 以及这一行
console.log( a ); // 2
```

2019-02-24 12:11:55

如果function是声明中的第一个词，那么就是一个函数声明，否则就

2019-02-24 12:12:00

2019-02-24 12:13:25

函数声明和函数表达式之间最重要的区别是它们的名称标识符将会绑定在何处。

2019-02-24 12:17:04

比较一下前面两个代码片段。第一个片段中foo被绑定在所在作用域中，可以直接通过foo()来调用它。第二个片段中foo被绑定在函数表达式自身的函数中而不是所在作用域中。

2019-02-24 12:17:31

foo变量名被隐藏在自身中意味着不会非必要地污染外部作用域。

3.3.1 匿名和具名

2019-02-24 12:18:10

函数表达式可以是匿名的

2019-02-24 12:20:10

1. 匿名函数在栈追踪中不会显示出有意义的函数名，使得调试很困难。

2019-02-24 12:20:26

如果没有函数名，当函数需要引用自身时只能使用已经过期的arguments.callee引用，比如在递归中。

2019-02-24 12:21:44

3. 匿名函数省略了对于代码可读性/可理解性很重要的函数名。一个描述性的名称可以让代码不言自明。

2019-02-24 12:35:56

始终给函数表达式命名是一个最佳实践

3.3.2 立即执行函数表达式

2019-02-24 12:37:02

由于函数被包含在一对( )括号

2019-02-24 12:37:10

内部，因此成为了一个表达式，通过在末尾加上另外一个( )可以立即执行这个函数，比如(function foo(){ .. })()。

3.4 块作用域

2019-02-24 12:51:47

我们在for循环的头部直接定义了变量i，通常是因为只想在for循环内部的上下文中使用i，而忽略了i会被绑定在外部作用域（函数或全局）中的事实。

3.4.2 try/catch

2019-02-24 14:24:37

非常少有人会注意到JavaScript的ES3规范中规定try/catch的catch分句会创建一个块作用域，其中声明的变量仅在catch内部有效。

2019-02-24 14:25:55

为了避免这个不必要的警告，很多开发者会将catch的参数命名为err1、err2等。也有开发者干脆关闭了静态检查工具对重复变量名的检查。

3.4.3 let

2019-02-24 14:31:37

但是使用let进行的声明不会在块作用域中进行提升。声明的代码被运行之前，声明并

不“存在”。

2019-02-24 14:33:52

但是，由于click函数形成了一个覆盖整个作用域的闭包，JavaScript引擎极有可能依然保存着这个结构（取决于具体实现）。
块作用域可以打消这种顾虑，可以让引擎清楚地知道没有必要继续保存someReallyBigData了

2019-02-24 14:52:42

for循环头部的let不仅将i绑定到了for循环的块中，事实上它将其重新绑定到了循环的每一个迭代中，确保使用上一个循环迭代结束时的值重新进行赋值。

3.5 小结

2019-02-24 14:57:20

函数是JavaScript中最常见的作用域单元。本质上，声明在一个函数内部的变量或函数会在所处的作用域中“隐藏”起来，这是有意为之的良好软件的设计原则。
但函数不是唯一的作用域单元。块作用域指的是变量和函数不仅可以属于所处的作用域，也可以属于某个代码块（通常指{ .. }内部）。

4.2 编译器再度来袭

2019-02-24 23:18:10

可以看到，函数声明会被提升，但是函数表达式却不会被提升。

2019-02-24 23:19:28

foo()由于对undefined值进行函数调用而导致非法操作，因此抛出TypeError异常。

2019-02-24 23:25:26

同时也要记住，即使是具名的函数表达式，名称标识符在赋值之前也无法在所在作用域中使用：

2019-02-24 23:25:37

这个代码片段经过提升后，实际上会被理解为以下形式：

4.3 函数优先

2019-02-24 23:27:03

有多个“重复”声明的代码中）是函数会首先被提升，然后才是变量。

2019-02-24 23:27:12

重复的var声明会被忽略掉，但出现在后面的函数声明还是可以覆盖前面的。

2019-02-24 23:27:31

虽然这些听起来都是些无用的学院理论，但是它说明了在同一个作用域中进行重复定义是非常糟糕的，而且经常会导致各种奇怪的问题。

2019-02-24 23:28:19

因此应该尽可能避免在块内部声明函数。

4.4 小结

2019-02-24 23:28:42

我们习惯将var a = 2;看作一个声明，而实际上JavaScript引擎并不这么认为。它将var a和a = 2当

2019-02-24 23:28:49

作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。

2019-02-24 23:30:09

所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提

升。

2019-02-24 23:30:17

声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升。

5.1 启示

2019-02-24 23:34:12

闭包是基于词法作用域书写代码时所产生的自然结果，你甚至不需要为了利用它们而有意识地创建闭包。闭包的创建和使用在你的代码中随处可见。你缺少的是根据你自己的意愿来识别、拥抱和影响闭包的思维环境。

5.2 实质问题

2019-02-24 23:37:21

当函数可以记住并访问所在的词法作用域时，就产生了闭包，即使函数是在当前词法作用域之外执行。

2019-02-24 23:39:37

```
function foo() {
var a = 2;
function bar() {
console.log( a );
}
return bar;
}
var baz = foo();
```

2019-02-24 23:39:42

baz(); // 2 ———— 朋友，这就是闭包的效果。

这个函数在定义时的词法作用

2019-02-24 23:42:18

域以外的地方被调用。闭包使得函数可以继续访问定义时的词法作用域。

2019-02-24 23:44:23

无论通过何种手段将内部函数传递到所在的词法作用域以外，它都会持有对原始定义作用域的引用，无论在何处执行这个函数都会使用闭包。

5.3 现在我懂了

2019-02-24 23:50:04

```
function setupBot(name, selector) {
$( selector ).click( function activator() {
console.log( "Activating:" + name );
} );
}
setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

2019-02-24 23:49:59

本质上无论何时何地，如果将函数（访问它们各自的词法作用域）当作第一级的值类型并到处传递，你就会看到闭包在这些函数中的应用。在定时器、事件监听器、Ajax请求、跨窗口通信、Web Workers或者任何其他的异步（或者同步）任务中，只要使用了回调函数，实际上就是在使用闭包！

2019-02-24 23:51:45

因为函数（示例代码中的IIFE）并

2019-02-24 23:52:06

不是在它本身的词法作用域以外执行的。它在定义时所在的作用域中执行（而外部作用

域，也就是全局作用域也持有a）。a是通过普通的词法作用域查找而非闭包被发现的。

2019-02-24 23:52:54

既非风动，亦非幡动，仁者心动耳。

2019-02-24 23:52:58

it's a tree falling in the forest with no one around to hear it,

5.4 循环和闭包

2019-02-24 23:55:37

仔细想一下，这好像又是显而易见的，延迟函数的回调会在循环结束时才执行。事实上，当定时器运行时即使每个迭代中执行的是setTimeout(.., 0)，所有的回调函数依然是在循环结束后才会被执行，因此会每次输出一个6出来。

2019-02-24 23:56:56

如果将延迟函数的回调重复定义五次，完全不使用

2019-02-24 23:57:01

循环，那它同这段代码是完全等价的。

2019-02-24 23:59:26

```
for (var i=1; i<=5; i++) {
(function(j) {
setTimeout( function timer() {
console.log( j );
}, j*1000 );
})( i );
}
```

```
for (let i=1; i<=5; i++) {
setTimeout( function timer() {
console.log( i );
}, i*1000 );
}
```

5.5 模块

2019-02-25 00:06:41

从方便观察的角度看，一个从函数调用所返回的，只有数据属性而没有闭包函数的对象并不是真正的模块。

2019-02-25 00:07:33

我们将模块函数转换成了IIFE（参见第3章），立即调用这个函数并将返回值直接赋值给单例的模块实例标识符foo。

5.6 小结

2019-02-25 00:16:15

模块有两个主要特征：(1)为创建内部作用域而调用了一个包装函数；(2)包装函数的返回值必须至少包括一个对内部函数的引用，这样

2019-02-25 00:16:22

就会创建涵盖整个包装函数内部作用域的闭包。

附录A 动态作用域

2019-02-25 12:59:59

词法作用域最重要的特征是它的定义过程发生在代码的书写阶段

2019-02-25 13:08:30

而动态作用域并不关心函数和作用域是如何声明以及在何处声明的，只关心它们从何处调用。

2019-02-25 13:10:11

词法作用域是在写代码或者说定义时确定的，而动态作用域是在运行时确定的

2019-02-25 13:10:32

词法作用域关注函数在何处声明，而动态作用域关注函数从何处调用。

附录C this词法

2019-02-25 13:20:42

它放弃了所有普通this绑定的规则，取而代之的是用当前的词法作用域覆盖了this本来的值。

1.2.1 指向自身

2019-02-25 13:37:05

执行foo.count = 0时，的确向函数对象foo添加了一个属性count。但是函数内部代码this.count中的this并不是指向那个函数对象，所以虽然属性名相同，根对象却并不相同，困惑随之产生

2019-02-25 13:39:41

从某种角度来说这个方法确实“解决”了问题，但可惜它忽略了真正的问题——无法理解this的含义和工作原理——而是返回舒适区

1.2.2 它的作用域

2019-02-25 18:00:41

this指向函数的作用域。这个问题有点复杂，因为在某种情况下它是正确的，但是在其他情况下它却是错误的。

2019-02-25 18:01:07

this在任何情况下都不指向函数的词法作用域

1.3 this到底是什么

2019-02-25 18:11:04

this的绑定和函数声明的位置没有任何关系，只取决于函数的调用方式。

1.4 小结

2019-02-25 18:12:18

this既不指向函数自身也不指向函数的词法作用域，你也许被这样的解释误导过，但其实它们都是错误的。

2019-02-25 18:12:28

this实际上是在函数被调用时发生的绑定，它指向什么完全取决于函数在哪里被调用。

2.1 调用位置

2019-02-25 20:53:49

你可以在工具中给foo()函数的第一行代码设置一个断点，或者直接在第一行代码之前插入一条debugger;语句。运行代码时，调试器会在那个位置暂停，同时会展示当前位置的函数调用列表，这就是你的调用栈

2.2.1 默认绑定

2019-02-25 21:03:35

声明在全局作用域中的变量（比如var a = 2）就是全局对象的一个同名属性。它们本质上就是同一个东西，并不是通过复制得到的，就像一个硬币的两面一样。

2019-02-25 21:05:13

如果使用严格模式（strict mode），那么全局对象将无法使用默认绑定，因此this会绑定到undefined：

2.2.2 隐式绑定

2019-02-25 21:09:36

当函数引用有上下文对象时，隐式绑定规则会把函数调用中的this绑定到这个上下文对象。

2.2.3 显式绑定

2019-02-25 22:43:51

它们的第一个参数是一个对象，它

2019-02-25 22:43:59

们会把这个对象绑定到this，接着在调用函数时指定这个this。

2019-02-25 22:44:37

如果你传入了一个原始值（字符串类型、布尔类型或者数字类型）来当作this的绑定对象，这个原始值会被转换成它的对象形式

2019-02-25 23:23:37

由于硬绑定是一种非常常用的模式，所以在ES5中提供了内置的方法Function.prototype.bind，

判断this

2019-02-26 01:38:51

1. 函数是否在new中调用（new绑定）？如果是的话this绑定的是新创建的对象。

2019-02-26 01:38:57

2. 函数是否通过call、apply（显式绑定）或者硬绑定调用？如果是的话，this绑定的是指定的对象。

2019-02-26 01:39:04

3. 函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this绑定的是那个上下文对象。

2019-02-26 01:39:08

4. 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到undefined，否则绑定到全局对象。

2.4.1 被忽略的this

2019-02-26 01:41:47

```
function foo(a,b) {
console.log( "a:" + a + ", b:" + b );
}
// 把数组“展开”成参数
foo.apply( null, [2, 3] ); // a:2, b:3
// 使用 bind(..) 进行柯里化
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

2019-02-26 01:43:33

比如第三方库中的一个函数），那默认绑定规则会把this绑定到全局对象（在浏览器中这个对象是window），这将导致不可预计的后果（比如修改全局对象）。

2019-02-26 01:45:02

Object.create(null)和{}很像，但是并不会创建Object.prototype这个委托，所以它比{}“更空”：

2.4.2 间接引用

2019-02-26 01:47:32

赋值表达式p.foo = o.foo的返回值是目标函数的引用，因此调用位置是foo()而不是p.foo()或者o.foo()。根据我们之前说过的，这里会应用默认绑定

2019-02-26 01:47:56

如果函数体处于严格模式，this会被绑定到undefined，否则this会被绑定到全局对象。

2.4.3 软绑定

2019-02-26 09:23:29

软绑定版本的foo()可以手动将this绑定到obj2或者obj3上，但如果应用默认绑定，则

2019-02-26 09:23:35

会将this绑定到obj。

2.5 this词法

2019-02-26 09:27:21

箭头函数的绑定无法被修改

2.6 小结

2019-02-26 09:34:21

如果要判断一个运行中函数的this绑定，就需要找到这个函数的直接调用位置。找到之后就可以顺序应用下面这四条规则来判断this的绑定对象。
1. 由new调用？绑定到新创建的对象。

2019-02-26 09:34:35

2. 由call或者apply（或者bind）调用？绑定到指定的对象

3. 由上下文对象调用？绑定到那个上下文对象。
4. 默认：在严格模式下绑定到undefined，否则绑定到全局对象。

2019-02-26 09:34:46

ES6中的箭头函数并不会使用四条标准的绑定规则，而是根据当前的词法作用域来决定this，具体

2019-02-26 09:34:57

来说，箭头函数会继承外层函数调用的this绑定（无论this绑定到什么）。这其实和ES6之前代码中的self = this机制一样。

3.2 类型

2019-02-26 09:38:49

不同的对象在底层都表示为二进制，在JavaScript中二进制前三位都为0的话会被判断为object类型，null的二进制表示是全0，自然前三位也是0，所以执行typeof时会返回“object”

2019-02-26 09:42:52

引擎自动把字面量转换成String对象，所以可以访问属性和方法。

3.3.2 属性与方法

2019-02-26 10:50:10

最保险的说法可能是，“函数”和“方法”在JavaScript中是可以互换的。

3.3.5 属性描述符

2019-02-26 10:59:23

但是从ES5开始，所有的属性都具备了属性描述符

2019-02-26 11:04:28

把configurable修改成false是单向操作，无法撤销！

2019-02-26 11:04:45

要注意有一个小小的例外：即便属性是configurable:false， 我们还是可以把writable的状态由true改为false，但是无法由false改为true。

2019-02-26 11:05:07

除了无法修改，configurable:false还会禁止删除这个属性

3.3.6 不变性

2019-02-26 11:08:26

```
var myObject = {};
Object.defineProperty( myObject, "FAVORITE_NUMBER", {
value: 42,
writable: false,
configurable: false
} );
```

3.3.10 存在性

2019-02-26 13:40:17

in操作符会检查属性是否在对

2019-02-26 13:40:25

象及其[[Prototype]]原型链中

2019-02-26 13:48:52

propertyIsEnumerable(..)会检查给定的属性名是否直接存在于对象中（而不是在原型链

上）并且满足enumerable:true

4.3.1 多态

2019-02-26 15:57:00

多态并不表示子类和父类有关联，子类得到的只是父类的一份副本。类的继承其实就是复制。

4.4.1 显式混入

2019-02-26 16:03:40

有一点需要注意，我们

2019-02-26 16:03:48

处理的已经不再是类了，因为在JavaScript中不存在类，Vehicle和Car都是对象，供我们分别进行复制和粘贴。

2019-02-27 23:23:56

由于两个对象引用的是同一个

2019-02-27 23:24:07

函数，因此这种复制（或者说混入）实际上并不能完全模拟面向类的语言中的复制。

2019-02-27 23:29:30

首先我们复制一份Vehicle父类（对象）的定义，然后混入子类（对象）的定义（如果需要的话保留到父类的特殊引用），然后用这个复合对象构建实例。

5.1 [[Prototype]]

2019-02-27 23:53:20

Object.create(..)的原理，现在只需要知道它会创建一个对象并把这个对象的[[Prototype]]关联到指定的对象。

2019-02-27 23:56:19

通过各种语法进行属性查找时都会查找[[Prototype]]链，直到找到属性或者查找完整条原型链。

5.1.2 属性设置和屏蔽

2019-02-28 00:39:56

1. 如果在[[Prototype]]链上层存在名为foo的普通数据访问属性（参见第3章）并且没有被标记为只读（writable:false），那就会直接在myObject中添加一个名为foo的新属性，它是屏蔽属性。

2019-02-28 00:40:00

2. 如果在[[Prototype]]链上层存在foo，但是它被标记为只读（writable:false），那么无法修

2019-02-28 00:40:05

改已有属性或者在myObject上创建屏蔽属性。如果运行在严格模式下，代码会抛出一个错误。否则，这条赋值语句会被忽略。总之，不会发生屏蔽。

2019-02-28 00:47:02

3. 如果在[[Prototype]]链上层存在foo并且它是一个setter（参见第3章），那就一定会调用这个setter。foo不会被添加到（或者说屏蔽于）myObject，也不会重新定义foo这个setter。

2019-02-28 00:58:58

如果你希望在第二种和第三种情况下也屏蔽foo，那就不能使用=操作符来赋值，而是使用Object.defineProperty(..)（参见第3章）来向myObject添加foo。

2019-02-28 01:02:57

myObject.a++; // 隐式屏蔽！

5.2 “类”

2019-02-28 01:04:31

在JavaScript中，类无法描述对象的行为，（因为根本就不存在类！）

5.2.1 “类”函数

2019-02-28 01:11:26

委托（参见第6章）这个术语可以更加准确地描述JavaScript中对象的关联机制

2019-02-28 01:13:36

我们只是定义了B的一些指定特性，其他没有定义的东西都变成了“洞”。而这些洞（或者说缺少定义的空白处）最终会被委托行为“填满”。

5.2.2 “构造函数”

2019-02-28 01:16:29

Foo.prototype默认（在代码中第一行声明时！）有一个公有并且不可枚举（参见第3章）的属性.constructor，这个属性引用的是对象关联的函数（本例中是Foo）

2019-02-28 01:19:37

函数不是构造函数，但是当且仅当使用new时，函数调用会变成“构造函数调用”。

5.2.3 技术

2019-02-28 01:25:17

a.constructor只是通过默认的[[Prototype]]委托指向Foo，这和“构造”毫无关系

2019-02-28 01:26:32

Foo.prototype的.constructor属性只是Foo函数在声明时的默认属性。如果你创建了一个新对象并替换了函数默认的.prototype对象引用，那么新对象并不会自动获得.constructor属性。

2019-02-28 01:26:39

```
function Foo() { /* .. */ }
Foo.prototype = { /* .. */ }; // 创建一个新原型对象
var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

2019-02-28 01:28:36

a1并没有.constructor属性，所以它会委托[[Prototype]]链上的Foo.prototype。但是这个对象也没有.constructor属性（不过默认的Foo.prototype对象有这个属性！），所以它会继续委托，这次会委托给委托链顶端的Object.prototype。这个对象有.constructor属性，指向内置的Object(..)函数。

2019-02-28 01:29:23

```
function Foo() { /* .. */ }
Foo.prototype = { /* .. */ }; // 创建一个新原型对象
// 需要在 `Foo.prototype` 上“修复”丢失的.constructor属性
// 新对象属性起到Foo.prototype的作用
// 关于defineProperty(..)，参见第3章
```

2019-02-28 01:29:18

```
Object.defineProperty( Foo.prototype, "constructor" , {
enumerable: false,
writable: true,
configurable: true,
value: Foo // 让.constructor指向Foo
```

```
} );
```

2019-02-28 01:30:14

.constructor并不是一个不可变属性。它是不可枚举（参见上面的代码）的，但是它的值是可写的（可以被修改）。此外，你可以给任意[[Prototype]]链中的任意对象添加一个名为constructor的属性或者对其进行修改，你可以任意对其赋值。

2019-02-28 01:30:49

a1.constructor是一个非常不可靠并且不安全的引用。通常来说要尽量避免使用这些引用。

5.3 （原型）继承

2019-02-28 08:37:41

Foo.call(this, name);

2019-02-28 08:37:50

我们创建了一个新的Bar.prototype对象并关联到Foo.prototype

2019-02-28 08:40:56

调用Object.create(..)会凭空创建一个“新”对象并把新对象内部的[[Prototype]]关联到你指定的对象

2019-02-28 08:41:25

声明function Bar() { .. }时，和其他函数一样，Bar会有一个.prototype关联到默认的对象，但是这个对象并不是我们想要的Foo.prototype。因此我们创建了一个新对象并把它关联到我们希望的对象上，直接把原始的关联对象抛弃掉。

2019-02-28 08:42:20

因此当你执行类似Bar.prototype.myLabel = ...的赋值语句时会直接修改Foo.proto

2019-02-28 08:42:25

type对象本身。

2019-02-28 08:43:19

Bar.prototype = new Foo()的确会创建一个关联到Bar.prototype的新对象。但是它使用了Foo(..)的“构造函数调用”，如果函数Foo有一些副作用（比如写日志、修改状态、注册到其他对象、给this添加数据属性，等等）的话，就会影响到Bar()的“后代”，后果不堪设想。

2019-02-28 08:46:29

在ES6之前，我们只能通过设置.__proto__属性来实现，但是这个方法并不是标准并且无法兼容所有浏览器。ES6添加了辅助函数Object.setPrototypeOf(..)，可以用标准并且可靠的方法来修改关联。

2019-02-28 08:47:07

```
// ES6之前需要抛弃默认的Bar.prototype
Bar.ptototype = Object.create( Foo.prototype );
// ES6开始可以直接修改现有的Bar.prototype
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

2019-02-28 08:53:41

这个方法只能处理对象（a）和函数（带.prototype引用的Foo）之间的关系。如果你想判断两个对象（比如a和b）之间是否通过[[Prototype]]链关联，只用instanceof无法实现

2019-02-28 08:54:53

如果使用内置的.bind(..)函数来生成一个硬绑定函数（参见第2章）的话，该函数是没有.prototype属性的。在这样的函数上使用instanceof的话，目标函数的.prototype会代替硬绑定函数的.prototype。

2019-02-28 09:04:48

下面是第二种判断[[Prototype]]反射的方法，它更加简洁：

2019-02-28 09:04:52

Foo.prototype.isPrototypeOf(a); // true

2019-02-28 09:10:58

如果你想直接查找（甚至可以通过.__proto__.__ptoto__...来遍

2019-02-28 09:11:03

历）原型链的话，这个方法非常有用。

2019-02-28 09:11:16

和我们之前说过的.constructor一样，.__proto__实际上并不存在于你正在使用的对象中（本例中是a）。实际上，它和其他的常用函数（.toString()、.isPrototypeOf(..)，等等）一样，存在于内置的Object.prototype中

2019-02-28 09:12:32

```
Object.defineProperty( Object.prototype, "__proto__", {
get: function() {
return Object.getPrototypeOf( this );
},
set: function(o) {
// ES6中的setPrototypeOf(..)
Object.setPrototypeOf( this, o );
return o;
```

2019-02-28 09:16:38

最好把[[Prototype]]对象关联看作是只读特性，从而增加代码的可读性。

2019-02-28 09:16:52

JavaScript社区中对于双下划线有一个非官方的称呼，他们会把类似__proto__的属性称为“笨蛋（dunder）”。所以，JavaScript潮人会把__proto__叫作“笨蛋proto”。

5.4.1 创建关联

2019-02-28 09:18:48

Object.create(..)会创建一个新对象（bar）并把它关联到我们指定的对象（foo），这样我们就可以充分发挥[[Prototype]]机制的

2019-02-28 09:18:52

威力（委托）并且避免不必要的麻烦（比如使用new的构造函数调用会生成.prototype和.constructor引用）。

2019-02-28 09:19:38

Object.create(null)会创建一个拥有空（或者说null）[[Prototype]]链接的对象，这个对象无法进行委托。由于这个对象没有原型链，所以instanceof操作符（之前解释过）无法进行判断，因此总是会返回false。这些特殊的空[[Prototype]]对象通常被称作“字典”，它们完全不会受到原型链的干扰，因

2019-02-28 09:19:45

此非常适合用来存储数据。

2019-02-28 09:23:12

```
if (!Object.create) {
Object.create = function(o) {
function F(){}
F.prototype = o;
return new F();
};
}
```

2019-02-28 09:27:27

```
var anotherObject = {
a:2
};
var myObject = Object.create( anotherObject, {
b: {
enumerable:
```

2019-02-28 09:27:31

```
false,
writable: true,
configurable: false,
value: 3
},
c: {
enumerable: true,
writable: false,
configurable: false,
value: 4
}
});
myObject.hasOwnProperty(
```

2019-02-28 09:27:36

```
"a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true
myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

5.4.2 关联关系是备用

2019-02-28 09:31:09

在ES6中有一个被称为“代理”（Proxy）的高端功能，它实现的就是“方法无法找到”时的行为。

2019-02-28 09:33:04

这里我们调用的myObject.doCool()是实际存在于myObject中的，这可以让我们的API设计更加清晰（不那么“神奇”）。从内部来说，我们的实现遵循的是委托设计模式

2019-02-28 09:33:20

内部委托比起直接委托可以让API接口设计更加清晰。

5.5 小结

2019-02-28 09:44:20

虽然这些JavaScript机制和传统面向类语言中的“类初始化”和“类继承”很相似，但是JavaScript中的机制有一个核心区别，那就是不会进行复制，对象之间是通过内部的[[Prototype]]链关联的。

6.1.2 委托理论

2019-02-28 10:01:25

我们真正关心的只是XYZ对象（和ABC对象）委托了Task对象。

2019-02-28 10:03:35

这时就可以找到setID(..)方法。此外，由于调用位置触发了this的隐式绑定规则（参见第2章），因此虽然setID(..)方法在Task中，运行时this仍然会绑定到XYZ，

2019-02-28 10:06:40

在API接口的设计中，委托最好在内部实现，不要直接暴露出去。在之前的例子中我们并没有让开发者通过API直接调用XYZ.setID()。（当然，可以这么做！）相反，我们把

委托隐藏在了API的内部，XYZ.prepareTask(..)会

2019-02-28 10:06:43

委托Task.setID(..)。更多细节参见5.4.2节。

2019-02-28 10:07:36

你无法在两个或两个以上互相（双向）委托的对象之间创建循环委托。如果你把B关联到A然后试着把A关联到B，就会出错。

2019-02-28 10:10:57

之所以有这种细微的差别，是因为Chrome会动态跟踪并把实际执行构造过程的函数名当作一个内置属性，但是其他浏览器并不会跟踪这些额外的信息。

2019-02-28 10:11:56

```
function Foo() {}
var a1 = new Foo();
Foo.prototype.constructor = function Gotcha(){};
a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"
a1; // Foo {}
```

2019-02-28 10:13:52

除了这个bug，Chrome内部跟踪（只用于调试输出）“构造函数名称”的方法是Chrome自身的一种扩展行为，并不包含在JavaScript的规范中。

2019-02-28 10:15:37

如果你并不是使用“构造函数”来生成对象，比如使用本章介绍的对象关联风格来编写代码，那Chrome就无法跟踪对象内部的“构造函数名称”，这样的对象输出是Object {}，意思是“Object()构造出的对象”。

6.1.3 比较思维模型

2019-02-28 10:29:30

```
Foo = {
init: function(who) {
this.me = who;
},
identify: function() {
return "I am " + this.me;
}
};
Bar = Object.create( Foo );
Bar.speak = function() {
```

2019-02-28 10:29:38

```
alert( "Hello, " + this.identify() + "." );
};
var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );
b1.speak();
b2.speak();
```

2019-02-28 10:21:13

但是非常重要的一点是，这段代码简洁了许多，我们只是把对象关联起来，并不需要那些既复杂又令人困惑的模仿类的行为（构造函数、原型以及new）

2019-02-28 10:29:50

通过比较可以看出，对象关联风格的代码显然更加简洁，因为这

2019-02-28 10:29:56

种代码只关注一件事：对象之间的关联关系。

6.2.1 控件“类”

2019-02-28 14:01:04

```
// 父类
function Widget(width,height) {
this.width = width || 50;
this.height = height || 50;
this.$elem = null;
}
```

2019-02-28 14:01:11

```
Widget.prototype.render = function($where){
if (this.$elem) {
this.$elem.css( {
width: this.width + "px",
height: this.height + "px"
} ).appendTo( $where );
}
};
// 子类
function Button(width,height,label) {
// 调用“super”构造函数
```

2019-02-28 14:01:16

```
Widget.call( this, width, height );
this.label = label || "Default";
this.$elem = $( "<button>" ).text( this.label );
}
// 让Button“继承”Widget
Button.prototype = Object.create( Widget.prototype );
// 重写render(..)
```

```
Button.prototype.render = function($where) {
```

2019-02-28 14:01:21

```
/ “super”调用
Widget.prototype.render.call( this, $where );
this.$elem.click( this.onClick.bind( this ) );
};
Button.prototype.onClick = function(evt) {
console.log( "Button '" + this.label + "' clicked!" );
};
$( document ).ready( function(){
```

2019-02-28 14:01:25

```
var $body = $( document.body );
var btn1 = new Button( 125, 30, "Hello" );
var btn2 = new Button( 150, 40, "World" );
btn1.render( $body );
btn2.render( $body );
} );
```

2019-02-28 14:04:48

class仍然是通过[[Prototype]]机制实现的

6.5 内省

2019-02-28 14:55:15

ES6的Promise就是典型的“鸭子类型”（之前解释过，本书并不会介绍Promise）。出于各种各样的原因，我们需要判断一个对象引用是否是Promise，但是判断的方法是检查对象是否有then()方法。换句话说，如果对象有then()方法，ES6的Promise就会认为这个对象是“可持续”（thenable）的

2019-02-28 14:57:27

我们认为JavaScript中对象关联比类风格的代码更加简洁（而且功能相同）。

6.6 小结

2019-02-28 14:57:56

行为委托认为对象之间是兄弟关系，互相委托，而不是父类和子类的关系。JavaScript的[[Prototype]]机制本质上就是行为委托机制。也就是说，我们可以选择在JavaScript中努力实现类机制（参见第4和第5章），也可以拥抱更自然的[[Prototype]]委托机制。

A.2 class陷阱

2019-02-28 15:04:48

class基本上只是现有[[Prototype]]（委托！）机制的一种语法糖。

2019-02-28 15:11:37

super并不像this一样是晚绑定（late bound， 或者说动态绑定）的，它在[[HomeObject]].[[Prototype]]上，[[HomeObject]]会在创建时静态绑定。

A.3 静态大于动态吗

2019-02-28 15:15:08

动态太难实现了，所以这可能不是个好主意。这里有一种看起来像静态的语法，所以编写静态代码吧。

《你不知道的JavaScript（上卷）》的笔记（作者：Kyle Simpson、赵望野、梁杰）

多看笔记 来自多看阅读 for iOS