```
        return 0;
      }
    };
}
```

这里的 createComparisonFunction()函数返回一个匿名函数，这个匿名函数要么被赋值给一个变量，要么可以直接调用。但在 createComparisonFunction()内部，那个函数是匿名的。任何时候，只要函数被当作值来使用，它就是一个函数表达式。本章后面会介绍，这并不是使用函数表达式的唯一方式。

10.12　递归

递归函数通常的形式是一个函数通过名称调用自己，如下面的例子所示：

```
function factorial(num) {
  if (num <= 1) {
    return 1;
  } else {
    return num * factorial(num - 1);
  }
}
```

这是经典的递归阶乘函数。虽然这样写是可以的，但如果把这个函数赋值给其他变量，就会出问题：

```
let anotherFactorial = factorial;
factorial = null;
console.log(anotherFactorial(4));  // 报错
```

这里把 factorial()函数保存在了另一个变量 anotherFactorial 中，然后将 factorial 设置为 null，于是只保留了一个对原始函数的引用。而在调用 anotherFactorial()时，要递归调用 factorial()，但因为它已经不是函数了，所以会出错。在写递归函数时使用 arguments.callee 可以避免这个问题。

arguments.callee 就是一个指向正在执行的函数的指针，因此可以在函数内部递归调用，如下所示：

```
function factorial(num) {
  if (num <= 1) {
    return 1;
  } else {
    return num * arguments.callee(num - 1);
  }
}
```

像这里加粗的这一行一样，把函数名称替换成 arguments.callee，可以确保无论通过什么变量调用这个函数都不会出问题。因此在编写递归函数时，arguments.callee 是引用当前函数的首选。

不过，在严格模式下运行的代码是不能访问 arguments.callee 的，因为访问会出错。此时，可以使用命名函数表达式（named function expression）达到目的。比如：

```
const factorial = (function f(num) {
  if (num <= 1) {
    return 1;
  } else {
    return num * f(num - 1);
  }
});
```

这里创建了一个命名函数表达式 f()，然后将它赋值给了变量 factorial。即使把函数赋值给另一个变量，函数表达式的名称 f 也不变，因此递归调用不会有问题。这个模式在严格模式和非严格模式下都可以使用。

10.13 尾调用优化

ECMAScript 6 规范新增了一项内存管理优化机制，让 JavaScript 引擎在满足条件时可以重用栈帧。具体来说，这项优化非常适合“尾调用”，即外部函数的返回值是一个内部函数的返回值。比如：

```
function outerFunction() {
  return innerFunction(); // 尾调用
}
```

在 ES6 优化之前，执行这个例子会在内存中发生如下操作。

(1) 执行到 outerFunction 函数体，第一个栈帧被推到栈上。

(2) 执行 outerFunction 函数体，到 return 语句。计算返回值必须先计算 innerFunction。

(3) 执行到 innerFunction 函数体，第二个栈帧被推到栈上。

(4) 执行 innerFunction 函数体，计算其返回值。

(5) 将返回值传回 outerFunction，然后 outerFunction 再返回值。

(6) 将栈帧弹出栈外。

在 ES6 优化之后，执行这个例子会在内存中发生如下操作。

(1) 执行到 outerFunction 函数体，第一个栈帧被推到栈上。

(2) 执行 outerFunction 函数体，到达 return 语句。为求值返回语句，必须先求值 innerFunction。

(3) 引擎发现把第一个栈帧弹出栈外也没问题，因为 innerFunction 的返回值也是 outerFunction 的返回值。

(4) 弹出 outerFunction 的栈帧。

(5) 执行到 innerFunction 函数体，栈帧被推到栈上。

(6) 执行 innerFunction 函数体，计算其返回值。

(7) 将 innerFunction 的栈帧弹出栈外。

很明显，第一种情况下每多调用一次嵌套函数，就会多增加一个栈帧。而第二种情况下无论调用多少次嵌套函数，都只有一个栈帧。这就是 ES6 尾调用优化的关键：如果函数的逻辑允许基于尾调用将其销毁，则引擎就会那么做。

> **注意** 现在还没有办法测试尾调用优化是否起作用。不过，因为这是 ES6 规范所规定的，兼容的浏览器实现都能保证在代码满足条件的情况下应用这个优化。

10.13.1 尾调用优化的条件

尾调用优化的条件就是确定外部栈帧真的没有必要存在了。涉及的条件如下：

- 代码在严格模式下执行；
- 外部函数的返回值是对尾调用函数的调用；

- 尾调用函数返回后不需要执行额外的逻辑；
- 尾调用函数不是引用外部函数作用域中自由变量的闭包。

下面展示了几个违反上述条件的函数，因此都不符号尾调用优化的要求：

```
"use strict";

// 无优化：尾调用没有返回
function outerFunction() {
  innerFunction();
}

// 无优化：尾调用没有直接返回
function outerFunction() {
  let innerFunctionResult = innerFunction();
  return innerFunctionResult;
}

// 无优化：尾调用返回后必须转型为字符串
function outerFunction() {
  return innerFunction().toString();
}

// 无优化：尾调用是一个闭包
function outerFunction() {
  let foo = 'bar';
  function innerFunction() { return foo; }

  return innerFunction();
}
```

下面是几个符合尾调用优化条件的例子：

```
"use strict";

// 有优化：栈帧销毁前执行参数计算
function outerFunction(a, b) {
  return innerFunction(a + b);
}

// 有优化：初始返回值不涉及栈帧
function outerFunction(a, b) {
  if (a < b) {
    return a;
  }
  return innerFunction(a + b);
}

// 有优化：两个内部函数都在尾部
function outerFunction(condition) {
  return condition ? innerFunctionA() : innerFunctionB();
}
```

差异化尾调用和递归尾调用是容易让人混淆的地方。无论是递归尾调用还是非递归尾调用，都可以应用优化。引擎并不区分尾调用中调用的是函数自身还是其他函数。不过，这个优化在递归场景下的效果是最明显的，因为递归代码最容易在栈内存中迅速产生大量栈帧。

> **注意** 之所以要求严格模式，主要因为在非严格模式下函数调用中允许使用 f.arguments 和 f.caller，而它们都会引用外部函数的栈帧。显然，这意味着不能应用优化了。因此尾调用优化要求必须在严格模式下有效，以防止引用这些属性。

10.13.2 尾调用优化的代码

可以通过把简单的递归函数转换为待优化的代码来加深对尾调用优化的理解。下面是一个通过递归计算斐波纳契数列的函数：

```
function fib(n) {
  if (n < 2) {
    return n;
  }

  return fib(n - 1) + fib(n - 2);
}

console.log(fib(0));  // 0
console.log(fib(1));  // 1
console.log(fib(2));  // 1
console.log(fib(3));  // 2
console.log(fib(4));  // 3
console.log(fib(5));  // 5
console.log(fib(6));  // 8
```

显然这个函数不符合尾调用优化的条件，因为返回语句中有一个相加的操作。结果，fib(n)的栈帧数的内存复杂度是 $O(2^n)$。因此，即使这么一个简单的调用也可以给浏览器带来麻烦：

```
fib(1000);
```

当然，解决这个问题也有不同的策略，比如把递归改写成迭代循环形式。不过，也可以保持递归实现，但将其重构为满足优化条件的形式。为此可以使用两个嵌套的函数，外部函数作为基础框架，内部函数执行递归：

```
"use strict";

// 基础框架
function fib(n) {
  return fibImpl(0, 1, n);
}

// 执行递归
function fibImpl(a, b, n) {
  if (n === 0) {
    return a;
  }
  return fibImpl(b, a + b, n - 1);
}
```

这样重构之后，就可以满足尾调用优化的所有条件，再调用 fib(1000)就不会对浏览器造成威胁了。

10.14 闭包

匿名函数经常被人误认为是闭包（closure）。**闭包**指的是那些引用了另一个函数作用域中变量的函

数，通常是在嵌套函数中实现的。比如，下面是之前展示的 createComparisonFunction()函数，注意其中加粗的代码：


```
function createComparisonFunction(propertyName) {
  return function(object1, object2) {
    let value1 = object1[propertyName];
    let value2 = object2[propertyName];

    if (value1 < value2) {
      return -1;
    } else if (value1 > value2) {
      return 1;
    } else {
      return 0;
    }
  };
}
```

这里加粗的代码位于内部函数（匿名函数）中，其中引用了外部函数的变量 propertyName。在这个内部函数被返回并在其他地方被使用后，它仍然引用着那个变量。这是因为内部函数的作用域链包含 createComparisonFunction()函数的作用域。要理解为什么会这样，可以想想第一次调用这个函数时会发生什么。

本书在第 4 章曾介绍过作用域链的概念。理解作用域链创建和使用的细节对理解闭包非常重要。在调用一个函数时，会为这个函数调用创建一个执行上下文，并创建一个作用域链。然后用 arguments 和其他命名参数来初始化这个函数的活动对象。外部函数的活动对象是内部函数作用域链上的第二个对象。这个作用域链一直向外串起了所有包含函数的活动对象，直到全局执行上下文才终止。

在函数执行时，要从作用域链中查找变量，以便读、写值。来看下面的代码：

```
function compare(value1, value2) {
  if (value1 < value2) {
    return -1;
  } else if (value1 > value2) {
    return 1;
  } else {
    return 0;
  }
}

let result = compare(5, 10);
```

这里定义的 compare()函数是在全局上下文中调用的。第一次调用 compare()时，会为它创建一个包含 arguments、value1 和 value2 的活动对象，这个对象是其作用域链上的第一个对象。而全局上下文的变量对象则是 compare()作用域链上的第二个对象，其中包含 this、result 和 compare。图 10-1 展示了以上关系。

函数执行时，每个执行上下文中都会有一个包含其中变量的对象。全局上下文中的叫变量对象，它会在代码执行期间始终存在。而函数局部上下文中的叫活动对象，只在函数执行期间存在。在定义 compare()函数时，就会为它创建作用域链，预装载全局变量对象，并保存在内部的[[Scope]]中。在调用这个函数时，会创建相应的执行上下文，然后通过复制函数的[[Scope]]来创建其作用域链。接着会创建函数的活动对象（用作变量对象）并将其推入作用域链的前端。在这个例子中，这意味着 compare()函数执行上下文的作用域链中有两个变量对象：局部变量对象和全局变量对象。作用域链其实是一个包

含指针的列表，每个指针分别指向一个变量对象，但物理上并不会包含相应的对象。

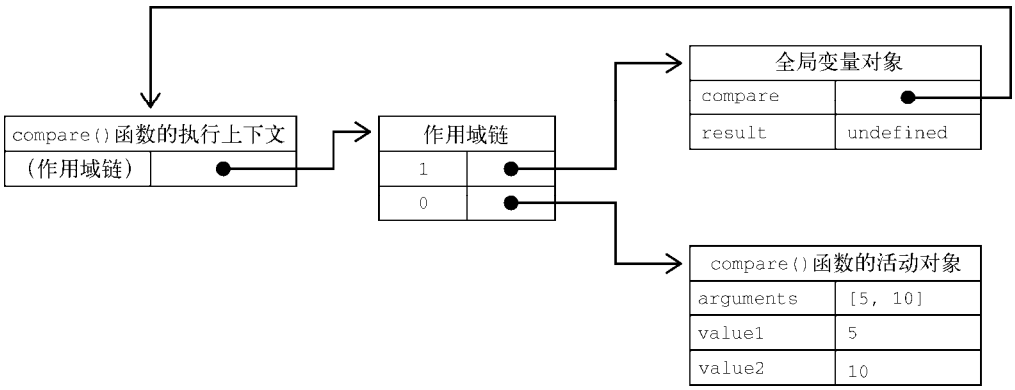

图 10-1

函数内部的代码在访问变量时，就会使用给定的名称从作用域链中查找变量。函数执行完毕后，局部活动对象会被销毁，内存中就只剩下全局作用域。不过，闭包就不一样了。

在一个函数内部定义的函数会把其包含函数的活动对象添加到自己的作用域链中。因此，在 `createComparisonFunction()` 函数中，匿名函数的作用域链中实际上包含 `createComparisonFunction()` 的活动对象。图 10-2 展示了以下代码执行后的结果。

```
let compare = createComparisonFunction('name');
let result = compare({ name: 'Nicholas' }, { name: 'Matt' });
```

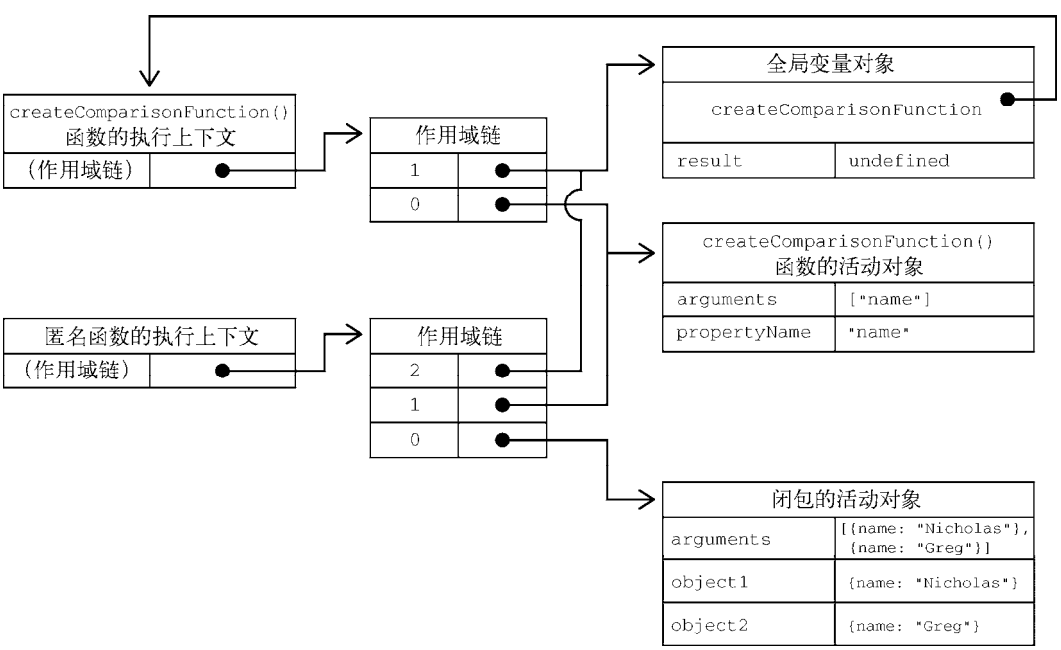

图 10-2

在 createComparisonFunction()返回匿名函数后，它的作用域链被初始化为包含 createComparisonFunction()的活动对象和全局变量对象。这样，匿名函数就可以访问到 createComparisonFunction()可以访问的所有变量。另一个有意思的副作用就是，createComparisonFunction()的活动对象并不能在它执行完毕后销毁，因为匿名函数的作用域链中仍然有对它的引用。在 createComparisonFunction()执行完毕后，其执行上下文的作用域链会销毁，但它的活动对象仍然会保留在内存中，直到匿名函数被销毁后才会被销毁：

```
// 创建比较函数
let compareNames = createComparisonFunction('name');

// 调用函数
let result = compareNames({ name: 'Nicholas' }, { name: 'Matt' });

// 解除对函数的引用，这样就可以释放内存了
compareNames = null;
```

这里，创建的比较函数被保存在变量 compareNames 中。把 compareNames 设置为等于 null 会解除对函数的引用，从而让垃圾回收程序可以将内存释放掉。作用域链也会被销毁，其他作用域（除全局作用域之外）也可以销毁。图 10-2 展示了调用 compareNames()之后作用域链之间的关系。

> **注意**　因为闭包会保留它们包含函数的作用域，所以比其他函数更占用内存。过度使用闭包可能导致内存过度占用，因此建议仅在十分必要时使用。V8 等优化的 JavaScript 引擎会努力回收被闭包困住的内存，不过我们还是建议在使用闭包时要谨慎。

10.14.1　this 对象

在闭包中使用 this 会让代码变复杂。如果内部函数没有使用箭头函数定义，则 this 对象会在运行时绑定到执行函数的上下文。如果在全局函数中调用，则 this 在非严格模式下等于 window，在严格模式下等于 undefined。如果作为某个对象的方法调用，则 this 等于这个对象。匿名函数在这种情况下不会绑定到某个对象，这就意味着 this 会指向 window，除非在严格模式下 this 是 undefined。不过，由于闭包的写法所致，这个事实有时候没有那么容易看出来。来看下面的例子：

```
window.identity = 'The Window';

let object = {
  identity: 'My Object',
  getIdentityFunc() {
    return function() {
      return this.identity;
    };
  }
};

console.log(object.getIdentityFunc()()); // 'The Window'
```

这里先创建了一个全局变量 identity，之后又创建一个包含 identity 属性的对象。这个对象还包含一个 getIdentityFunc()方法，返回一个匿名函数。这个匿名函数返回 this.identity。因为 getIdentityFunc()返回函数，所以 object.getIdentityFunc()()会立即调用这个返回的函数，从而得到一个字符串。可是，此时返回的字符串是"The Winodw"，即全局变量 identity 的值。为什

么匿名函数没有使用其包含作用域（getIdentityFunc()）的 this 对象呢?

前面介绍过，每个函数在被调用时都会自动创建两个特殊变量：this 和 arguments。内部函数永远不可能直接访问外部函数的这两个变量。但是，如果把 this 保存到闭包可以访问的另一个变量中，则是行得通的。比如：

```
window.identity = 'The Window';

let object = {
  identity: 'My Object',
  getIdentityFunc() {
    let that = this;
    return function() {
      return that.identity;
    };
  }
};

console.log(object.getIdentityFunc()()); // 'My Object'
```

这里加粗的代码展示了与前面那个例子的区别。在定义匿名函数之前，先把外部函数的 this 保存到变量 that 中。然后在定义闭包时，就可以让它访问 that，因为这是包含函数中名称没有任何冲突的一个变量。即使在外部函数返回之后，that 仍然指向 object，所以调用 object.getIdentityFunc()() 就会返回"My Object"。

> **注意** this 和 arguments 都是不能直接在内部函数中访问的。如果想访问包含作用域中的 arguments 对象，则同样需要将其引用先保存到闭包能访问的另一个变量中。

在一些特殊情况下，this 值可能并不是我们所期待的值。比如下面这个修改后的例子：

```
window.identity = 'The Window';
let object = {
  identity: 'My Object',
  getIdentity () {
    return this.identity;
  }
};
```

getIdentity()方法就是返回 this.identity 的值。以下是几种调用 object.getIdentity() 的方式及返回值：

```
object.getIdentity();                          // 'My Object'
(object.getIdentity)();                        // 'My Object'
(object.getIdentity = object.getIdentity)();   // 'The Window'
```

第一行调用 object.getIdentity()是正常调用，会返回"My Object"，因为 this.identity 就是 object.identity。第二行在调用时把 object.getIdentity 放在了括号里。虽然加了括号之后看起来是对一个函数的引用，但 this 值并没有变。这是因为按照规范，object.getIdentity 和 (object.getIdentity)是相等的。第三行执行了一次赋值，然后再调用赋值后的结果。因为赋值表达式的值是函数本身，this 值不再与任何对象绑定，所以返回的是"The Window"。

一般情况下，不大可能像第二行和第三行这样调用对象上的方法。但通过这个例子，我们可以知道，即使语法稍有不同，也可能影响 this 的值。

10.14.2　内存泄漏

由于 IE 在 IE9 之前对 JScript 对象和 COM 对象使用了不同的垃圾回收机制（第 4 章讨论过），所以闭包在这些旧版本 IE 中可能会导致问题。在这些版本的 IE 中，把 HTML 元素保存在某个闭包的作用域中，就相当于宣布该元素不能被销毁。来看下面的例子：

```
function assignHandler() {
  let element = document.getElementById('someElement');
  element.onclick = () => console.log(element.id);
}
```

以上代码创建了一个闭包，即 element 元素的事件处理程序（事件处理程序将在第 13 章讨论）。而这个处理程序又创建了一个循环引用。匿名函数引用着 assignHandler()的活动对象，阻止了对 element 的引用计数归零。只要这个匿名函数存在，element 的引用计数就至少等于 1。也就是说，内存不会被回收。其实只要这个例子稍加修改，就可以避免这种情况，比如：

```
function assignHandler() {
  let element = document.getElementById('someElement');
  let id = element.id;

  element.onclick = () => console.log(id);

  element = null;
}
```

在这个修改后的版本中，闭包改为引用一个保存着 element.id 的变量 id，从而消除了循环引用。不过，光有这一步还不足以解决内存问题。因为闭包还是会引用包含函数的活动对象，而其中包含 element。即使闭包没有直接引用 element，包含函数的活动对象上还是保存着对它的引用。因此，必须再把 element 设置为 null。这样就解除了对这个 COM 对象的引用，其引用计数也会减少，从而确保其内存可以在适当的时候被回收。

10.15　立即调用的函数表达式

立即调用的匿名函数又被称作**立即调用的函数表达式**（IIFE，Immediately Invoked Function Expression）。它类似于函数声明，但由于被包含在括号中，所以会被解释为函数表达式。紧跟在第一组括号后面的第二组括号会立即调用前面的函数表达式。下面是一个简单的例子：

```
(function() {
  // 块级作用域
})();
```

使用 IIFE 可以模拟块级作用域，即在一个函数表达式内部声明变量，然后立即调用这个函数。这样位于函数体作用域的变量就像是在块级作用域中一样。ECMAScript 5 尚未支持块级作用域，使用 IIFE 模拟块级作用域是相当普遍的。比如下面的例子：

```
// IIFE
(function () {
  for (var i = 0; i < count; i++) {
    console.log(i);
  }
})();

console.log(i);  // 抛出错误
```

前面的代码在执行到 IIFE 外部的 `console.log()`时会出错，因为它访问的变量是在 IIFE 内部定义的，在外部访问不到。在 ECMAScript 5.1 及以前，为了防止变量定义外泄，IIFE 是个非常有效的方式。这样也不会导致闭包相关的内存问题，因为不存在对这个匿名函数的引用。为此，只要函数执行完毕，其作用域链就可以被销毁。

在 ECMAScript 6 以后，IIFE 就没有那么必要了，因为块级作用域中的变量无须 IIFE 就可以实现同样的隔离。下面展示了两种不同的块级作用域形式：

```
// 内嵌块级作用域
{
  let i;
  for (i = 0; i < count; i++) {
    console.log(i);
  }
}
console.log(i); // 抛出错误

// 循环的块级作用域
for (let i = 0; i < count; i++) {
  console.log(i);
}

console.log(i); // 抛出错误
```

说明 IIFE 用途的一个实际的例子，就是可以用它锁定参数值。比如：

```
let divs = document.querySelectorAll('div');

// 达不到目的!
for (var i = 0; i < divs.length; ++i) {
  divs[i].addEventListener('click', function() {
    console.log(i);
  });
}
```

这里使用 `var` 关键字声明了循环迭代变量 `i`，但这个变量并不会被限制在 `for` 循环的块级作用域内。因此，渲染到页面上之后，点击每个 都会弹出元素总数。这是因为在执行单击处理程序时，迭代变量的值是循环结束时的最终值，即元素的个数。而且，这个变量 `i` 存在于循环体外部，随时可以访问。

以前，为了实现点击第几个 就显示相应的索引值，需要借助 IIFE 来执行一个函数表达式，传入每次循环的当前索引，从而“锁定”点击时应该显示的索引值：

```
let divs = document.querySelectorAll('div');

for (var i = 0; i < divs.length; ++i) {
  divs[i].addEventListener('click', (function(frozenCounter) {
    return function() {
      console.log(frozenCounter);
    };
  })(i));
}
```

而使用 ECMAScript 块级作用域变量，就不用这么大动干戈了：

```
let divs = document.querySelectorAll('div');

for (let i = 0; i < divs.length; ++i) {
  divs[i].addEventListener('click', function() {
```