```
let result = addTen(count);
console.log(count);  // 20，没有变化
console.log(result); // 30
```

这里，函数 addTen()有一个参数 num，它其实是一个局部变量。在调用时，变量 count 作为参数传入。count 的值是 20，这个值被复制到参数 num 以便在 addTen()内部使用。在函数内部，参数 num 的值被加上了 10，但这不会影响函数外部的原始变量 count。参数 num 和变量 count 互不干扰，它们只不过碰巧保存了一样的值。如果 num 是按引用传递的，那么 count 的值也会被修改为 30。这个事实在使用数值这样的原始值时是非常明显的。但是，如果变量中传递的是对象，就没那么清楚了。比如，再看这个例子：

```
function setName(obj) {
  obj.name = "Nicholas";
}

let person = new Object();
setName(person);
console.log(person.name);  // "Nicholas"
```

这一次，我们创建了一个对象并把它保存在变量 person 中。然后，这个对象被传给 setName()方法，并被复制到参数 obj 中。在函数内部，obj 和 person 都指向同一个对象。结果就是，即使对象是按值传进函数的，obj 也会通过引用访问对象。当函数内部给 obj 设置了 name 属性时，函数外部的对象也会反映这个变化，因为 obj 指向的对象保存在全局作用域的堆内存上。很多开发者错误地认为，当在局部作用域中修改对象而变化反映到全局时，就意味着参数是按引用传递的。为证明对象是按值传递的，我们再来看看下面这个修改后的例子：

```
function setName(obj) {
  obj.name = "Nicholas";
  obj = new Object();
  obj.name = "Greg";
}

let person = new Object();
setName(person);
console.log(person.name);  // "Nicholas"
```

这个例子前后唯一的变化就是 setName()中多了两行代码，将 obj 重新定义为一个有着不同 name 的新对象。当 person 传入 setName()时，其 name 属性被设置为"Nicholas"。然后变量 obj 被设置为一个新对象且 name 属性被设置为"Greg"。如果 person 是按引用传递的，那么 person 应该自动将指针改为指向 name 为"Greg"的对象。可是，当我们再次访问 person.name 时，它的值是"Nicholas"，这表明函数中参数的值改变之后，原始的引用仍然没变。当 obj 在函数内部被重写时，它变成了一个指向本地对象的指针。而那个本地对象在函数执行结束时就被销毁了。

> **注意**　ECMAScript 中函数的参数就是局部变量。

4.1.4　确定类型

前一章提到的 typeof 操作符最适合用来判断一个变量是否为原始类型。更确切地说，它是判断一个变量是否为字符串、数值、布尔值或 undefined 的最好方式。如果值是对象或 null，那么 typeof

返回"object"，如下面的例子所示：

```
let s = "Nicholas";
let b = true;
let i = 22;
let u;
let n = null;
let o = new Object();
console.log(typeof s); // string
console.log(typeof i); // number
console.log(typeof b); // boolean
console.log(typeof u); // undefined
console.log(typeof n); // object
console.log(typeof o); // object
```


typeof 虽然对原始值很有用，但它对引用值的用处不大。我们通常不关心一个值是不是对象，而是想知道它是什么类型的对象。为了解决这个问题，ECMAScript 提供了 instanceof 操作符，语法如下：

```
result = variable instanceof constructor
```

如果变量是给定引用类型（由其原型链决定，将在第 8 章详细介绍）的实例，则 instanceof 操作符返回 true。来看下面的例子：

```
console.log(person instanceof Object);  // 变量 person 是 Object 吗?
console.log(colors instanceof Array);   // 变量 colors 是 Array 吗?
console.log(pattern instanceof RegExp); // 变量 pattern 是 RegExp 吗?
```

按照定义，所有引用值都是 Object 的实例，因此通过 instanceof 操作符检测任何引用值和 Object 构造函数都会返回 true。类似地，如果用 instanceof 检测原始值，则始终会返回 false，因为原始值不是对象。

> **注意** typeof 操作符在用于检测函数时也会返回"function"。当在 Safari（直到 Safari 5）和 Chrome（直到 Chrome 7）中用于检测正则表达式时，由于实现细节的原因，typeof 也会返回"function"。ECMA-262 规定，任何实现内部[[Call]]方法的对象都应该在 typeof 检测时返回"function"。因为上述浏览器中的正则表达式实现了这个方法，所以 typeof 对正则表达式也返回"function"。在 IE 和 Firefox 中，typeof 对正则表达式返回"object"。

4.2 执行上下文与作用域

执行上下文（以下简称“上下文”）的概念在 JavaScript 中是颇为重要的。变量或函数的上下文决定了它们可以访问哪些数据，以及它们的行为。每个上下文都有一个关联的**变量对象**（variable object），而这个上下文中定义的所有变量和函数都存在于这个对象上。虽然无法通过代码访问变量对象，但后台处理数据会用到它。

全局上下文是最外层的上下文。根据 ECMAScript 实现的宿主环境，表示全局上下文的对象可能不一样。在浏览器中，全局上下文就是我们常说的 window 对象（第 12 章会详细介绍），因此所有通过 var 定义的全局变量和函数都会成为 window 对象的属性和方法。使用 let 和 const 的顶级声明不会定义在全

局上下文中，但在作用域链解析上效果是一样的。上下文在其所有代码都执行完毕后会被销毁，包括定义在它上面的所有变量和函数（全局上下文在应用程序退出前才会被销毁，比如关闭网页或退出浏览器）。

每个函数调用都有自己的上下文。当代码执行流进入函数时，函数的上下文被推到一个上下文栈上。在函数执行完之后，上下文栈会弹出该函数上下文，将控制权返还给之前的执行上下文。ECMAScript 程序的执行流就是通过这个上下文栈进行控制的。

上下文中的代码在执行的时候，会创建变量对象的一个**作用域链**（scope chain）。这个作用域链决定了各级上下文中的代码在访问变量和函数时的顺序。代码正在执行的上下文的变量对象始终位于作用域链的最前端。如果上下文是函数，则其**活动对象**（activation object）用作变量对象。活动对象最初只有一个定义变量：`arguments`。（全局上下文中没有这个变量。）作用域链中的下一个变量对象来自包含上下文，再下一个对象来自再下一个包含上下文。以此类推直至全局上下文；全局上下文的变量对象始终是作用域链的最后一个变量对象。

代码执行时的标识符解析是通过沿作用域链逐级搜索标识符名称完成的。搜索过程始终从作用域链的最前端开始，然后逐级往后，直到找到标识符。（如果没有找到标识符，那么通常会报错。）

看一看下面这个例子：

```
var color = "blue";

function changeColor() {
  if (color === "blue") {
    color = "red";
  } else {
    color = "blue";
  }
}

changeColor();
```

对这个例子而言，函数 `changeColor()`的作用域链包含两个对象：一个是它自己的变量对象（就是定义 `arguments` 对象的那个），另一个是全局上下文的变量对象。这个函数内部之所以能够访问变量 `color`，就是因为可以在作用域链中找到它。

此外，局部作用域中定义的变量可用于在局部上下文中替换全局变量。看一看下面这个例子：

```
var color = "blue";

function changeColor() {
  let anotherColor = "red";

  function swapColors() {
    let tempColor = anotherColor;
    anotherColor = color;
    color = tempColor;

    // 这里可以访问 color、anotherColor 和 tempColor
  }

  // 这里可以访问 color 和 anotherColor，但访问不到 tempColor
  swapColors();
}

// 这里只能访问 color
changeColor();
```

以上代码涉及 3 个上下文：全局上下文、changeColor()的局部上下文和 swapColors()的局部上下文。全局上下文中有一个变量 color 和一个函数 changeColor()。changeColor()的局部上下文中有一个变量 anotherColor 和一个函数 swapColors()，但在这里可以访问全局上下文中的变量 color。swapColors()的局部上下文中有一个变量 tempColor，只能在这个上下文中访问到。全局上下文和 changeColor()的局部上下文都无法访问到 tempColor。而在 swapColors()中则可以访问另外两个上下文中的变量，因为它们都是父上下文。图 4-3 展示了前面这个例子的作用域链。

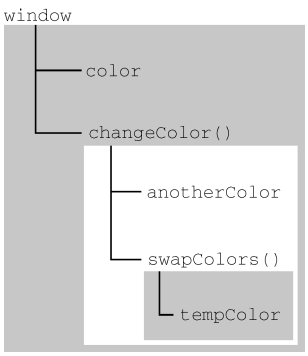

图 4-3

图 4-3 中的矩形表示不同的上下文。内部上下文可以通过作用域链访问外部上下文中的一切，但外部上下文无法访问内部上下文中的任何东西。上下文之间的连接是线性的、有序的。每个上下文都可以到上一级上下文中去搜索变量和函数，但任何上下文都不能到下一级上下文中去搜索。swapColors()局部上下文的作用域链中有 3 个对象：swapColors()的变量对象、changeColor()的变量对象和全局变量对象。swapColors()的局部上下文首先从自己的变量对象开始搜索变量和函数，搜不到就去搜索上一级变量对象。changeColor()上下文的作用域链中只有 2 个对象：它自己的变量对象和全局变量对象。因此，它不能访问 swapColors()的上下文。

> **注意** 函数参数被认为是当前上下文中的变量，因此也跟上下文中的其他变量遵循相同的访问规则。

4.2.1 作用域链增强

虽然执行上下文主要有全局上下文和函数上下文两种（eval()调用内部存在第三种上下文），但有其他方式来增强作用域链。某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在代码执行后会被删除。通常在两种情况下会出现这个现象，即代码执行到下面任意一种情况时：

- try/catch 语句的 catch 块
- with 语句

这两种情况下，都会在作用域链前端添加一个变量对象。对 with 语句来说，会向作用域链前端添加指定的对象；对 catch 语句而言，则会创建一个新的变量对象，这个变量对象会包含要抛出的错误对象的声明。看下面的例子：

```
function buildUrl() {
  let qs = "?debug=true";

  with(location){
    let url = href + qs;
  }

  return url;
}
```

这里，with 语句将 location 对象作为上下文，因此 location 会被添加到作用域链前端。buildUrl()函数中定义了一个变量 qs。当 with 语句中的代码引用变量 href 时，实际上引用的是 location.href，也就是自己变量对象的属性。在引用 qs 时，引用的则是定义在 buildUrl()中的那个变量，它定义在函数上下文的变量对象上。而在 with 语句中使用 var 声明的变量 url 会成为函数上下文的一部分，可以作为函数的值被返回；但像这里使用 let 声明的变量 url，因为被限制在块级作用域（稍后介绍），所以在 with 块之外没有定义。

> **注意**　IE 的实现在 IE8 之前是有偏差的，即它们会将 catch 语句中捕获的错误添加到执行上下文的变量对象上，而不是 catch 语句的变量对象上，导致在 catch 块外部都可以访问到错误。IE9 纠正了这个问题。

4.2.2　变量声明

ES6 之后，JavaScript 的变量声明经历了翻天覆地的变化。直到 ECMAScript 5.1，var 都是声明变量的唯一关键字。ES6 不仅增加了 let 和 const 两个关键字，而且还让这两个关键字压倒性地超越 var 成为首选。

1. 使用 var 的函数作用域声明

在使用 var 声明变量时，变量会被自动添加到最接近的上下文。在函数中，最接近的上下文就是函数的局部上下文。在 with 语句中，最接近的上下文也是函数上下文。如果变量未经声明就被初始化了，那么它就会自动被添加到全局上下文，如下面的例子所示：

```
function add(num1, num2) {
  var sum = num1 + num2;
  return sum;
}

let result = add(10, 20); // 30
console.log(sum);         // 报错：sum 在这里不是有效变量
```

这里，函数 add()定义了一个局部变量 sum，保存加法操作的结果。这个值作为函数的值被返回，但变量 sum 在函数外部是访问不到的。如果省略上面例子中的关键字 var，那么 sum 在 add()被调用之后就变成可以访问的了，如下所示：

```
function add(num1, num2) {
  sum = num1 + num2;
  return sum;
}

let result = add(10, 20); // 30
console.log(sum);         // 30
```

这一次，变量 sum 被用加法操作的结果初始化时并没有使用 var 声明。在调用 add()之后，sum 被添加到了全局上下文，在函数退出之后依然存在，从而在后面可以访问到。

> **注意** 未经声明而初始化变量是 JavaScript 编程中一个非常常见的错误，会导致很多问题。为此，读者在初始化变量之前一定要先声明变量。在严格模式下，未经声明就初始化变量会报错。

var 声明会被拿到函数或全局作用域的顶部，位于作用域中所有代码之前。这个现象叫作“提升”（hoisting）。提升让同一作用域中的代码不必考虑变量是否已经声明就可以直接使用。可是在实践中，提升也会导致合法却奇怪的现象，即在变量声明之前使用变量。下面的例子展示了在全局作用域中两段等价的代码：

```
var name = "Jake";

// 等价于：

name = 'Jake';
var name;
```

下面是两个等价的函数：

```
function fn1() {
  var name = 'Jake';
}

// 等价于：
function fn2() {
  var name;
  name = 'Jake';
}
```

通过在声明之前打印变量，可以验证变量会被提升。声明的提升意味着会输出 undefined 而不是 Reference Error：

```
console.log(name); // undefined
var name = 'Jake';

function() {
  console.log(name); // undefined
  var name = 'Jake';
}
```

2. 使用 let 的块级作用域声明

ES6 新增的 let 关键字跟 var 很相似，但它的作用域是块级的，这也是 JavaScript 中的新概念。块级作用域由最近的一对包含花括号{}界定。换句话说，if 块、while 块、function 块，甚至连单独的块也是 let 声明变量的作用域。

```
if (true) {
  let a;
}
console.log(a); // ReferenceError: a 没有定义

while (true) {
  let b;
}
```

```
console.log(b); // ReferenceError: b 没有定义

function foo() {
  let c;
}
console.log(c); // ReferenceError: c 没有定义
                // 这没什么可奇怪的
                // var 声明也会导致报错

// 这不是对象字面量，而是一个独立的块
// JavaScript 解释器会根据其中内容识别出它来
{
  let d;
}
console.log(d); // ReferenceError: d 没有定义
```

let 与 var 的另一个不同之处是在同一作用域内不能声明两次。重复的 var 声明会被忽略，而重复的 let 声明会抛出 SyntaxError。

```
var a;
var a;
// 不会报错

{
  let b;
  let b;
}
// SyntaxError: 标识符 b 已经声明过了
```

let 的行为非常适合在循环中声明迭代变量。使用 var 声明的迭代变量会泄漏到循环外部，这种情况应该避免。来看下面两个例子：

```
for (var i = 0; i < 10; ++i) {}
console.log(i); // 10

for (let j = 0; j < 10; ++j) {}
console.log(j); // ReferenceError: j 没有定义
```

严格来讲，let 在 JavaScript 运行时中也会被提升，但由于“暂时性死区”（temporal dead zone）的缘故，实际上不能在声明之前使用 let 变量。因此，从写 JavaScript 代码的角度说，let 的提升跟 var 是不一样的。

3. 使用 const 的常量声明

除了 let，ES6 同时还增加了 const 关键字。使用 const 声明的变量必须同时初始化为某个值。一经声明，在其生命周期的任何时候都不能再重新赋予新值。

```
const a; // SyntaxError: 常量声明时没有初始化

const b = 3;
console.log(b); // 3
b = 4; // TypeError: 给常量赋值
```

const 除了要遵循以上规则，其他方面与 let 声明是一样的：

```
if (true) {
  const a = 0;
}
console.log(a); // ReferenceError: a 没有定义
```

```
while (true) {
  const b = 1;
}
console.log(b); // ReferenceError: b 没有定义

function foo() {
  const c = 2;
}
console.log(c); // ReferenceError: c 没有定义

{
  const d = 3;
}
console.log(d); // ReferenceError: d 没有定义
```


const 声明只应用到顶级原语或者对象。换句话说，赋值为对象的 const 变量不能再被重新赋值为其他引用值，但对象的键则不受限制。

```
const o1 = {};
o1 = {}; // TypeError: 给常量赋值

const o2 = {};
o2.name = 'Jake';
console.log(o2.name); // 'Jake'
```

如果想让整个对象都不能修改，可以使用 Object.freeze()，这样再给属性赋值时虽然不会报错，但会静默失败：

```
const o3 = Object.freeze({});
o3.name = 'Jake';
console.log(o3.name); // undefined
```

由于 const 声明暗示变量的值是单一类型且不可修改，JavaScript 运行时编译器可以将其所有实例都替换成实际的值，而不会通过查询表进行变量查找。谷歌的 V8 引擎就执行这种优化。

> **注意** 开发实践表明，如果开发流程并不会因此而受很大影响，就应该尽可能地多使用 const 声明，除非确实需要一个将来会重新赋值的变量。这样可以从根本上保证提前发现重新赋值导致的 bug。

4. 标识符查找

当在特定上下文中为读取或写入而引用一个标识符时，必须通过搜索确定这个标识符表示什么。搜索开始于作用域链前端，以给定的名称搜索对应的标识符。如果在局部上下文中找到该标识符，则搜索停止，变量确定；如果没有找到变量名，则继续沿作用域链搜索。（注意，作用域链中的对象也有一个原型链，因此搜索可能涉及每个对象的原型链。）这个过程一直持续到搜索至全局上下文的变量对象。如果仍然没有找到标识符，则说明其未声明。

为更好地说明标识符查找，我们来看一个例子：

```
var color = 'blue';

function getColor() {
  return color;
}

console.log(getColor()); // 'blue'
```

在这个例子中，调用函数 getColor()时会引用变量 color。为确定 color 的值会进行两步搜索。第一步，搜索 getColor()的变量对象，查找名为 color 的标识符。结果没找到，于是继续搜索下一个变量对象（来自全局上下文），然后就找到了名为 color 的标识符。因为全局变量对象上有 color 的定义，所以搜索结束。

对这个搜索过程而言，引用局部变量会让搜索自动停止，而不继续搜索下一级变量对象。也就是说，如果局部上下文中有一个同名的标识符，那就不能在该上下文中引用父上下文中的同名标识符，如下面的例子所示：

```
var color = 'blue';

function getColor() {
  let color = 'red';
  return color;
}

console.log(getColor()); // 'red'
```

使用块级作用域声明并不会改变搜索流程，但可以给词法层级添加额外的层次：

```
var color = 'blue';

function getColor() {
  let color = 'red';
  {
    let color = 'green';
    return color;
  }
}

console.log(getColor()); // 'green'
```

在这个修改后的例子中，getColor()内部声明了一个名为 color 的局部变量。在调用这个函数时，变量会被声明。在执行到函数返回语句时，代码引用了变量 color。于是开始在局部上下文中搜索这个标识符，结果找到了值为'green'的变量 color。因为变量已找到，搜索随即停止，所以就使用这个局部变量。这意味着函数会返回'green'。在局部变量 color 声明之后的任何代码都无法访问全局变量 color，除非使用完全限定的写法 window.color。

> **注意**　标识符查找并非没有代价。访问局部变量比访问全局变量要快，因为不用切换作用域。不过，JavaScript 引擎在优化标识符查找上做了很多工作，将来这个差异可能就微不足道了。

4.3　垃圾回收

JavaScript 是使用垃圾回收的语言，也就是说执行环境负责在代码执行时管理内存。在 C 和 C++等语言中，跟踪内存使用对开发者来说是个很大的负担，也是很多问题的来源。JavaScript 为开发者卸下了这个负担，通过自动内存管理实现内存分配和闲置资源回收。基本思路很简单：确定哪个变量不会再使用，然后释放它占用的内存。这个过程是周期性的，即垃圾回收程序每隔一定时间（或者说在代码执行过程中某个预定的收集时间）就会自动运行。垃圾回收过程是一个近似且不完美的方案，因为某块内

存是否还有用，属于“不可判定的”问题，意味着靠算法是解决不了的。

我们以函数中局部变量的正常生命周期为例。函数中的局部变量会在函数执行时存在。此时，栈（或堆）内存会分配空间以保存相应的值。函数在内部使用了变量，然后退出。此时，就不再需要那个局部变量了，它占用的内存可以释放，供后面使用。这种情况下显然不再需要局部变量了，但并不是所有时候都会这么明显。垃圾回收程序必须跟踪记录哪个变量还会使用，以及哪个变量不会再使用，以便回收内存。如何标记未使用的变量也许有不同的实现方式。不过，在浏览器的发展史上，用到过两种主要的标记策略：标记清理和引用计数。

4.3.1 标记清理

JavaScript 最常用的垃圾回收策略是**标记清理**（mark-and-sweep）。当变量进入上下文，比如在函数内部声明一个变量时，这个变量会被加上存在于上下文中的标记。而在上下文中的变量，逻辑上讲，永远不应该释放它们的内存，因为只要上下文中的代码在运行，就有可能用到它们。当变量离开上下文时，也会被加上离开上下文的标记。

给变量加标记的方式有很多种。比如，当变量进入上下文时，反转某一位；或者可以维护“在上下文中”和“不在上下文中”两个变量列表，可以把变量从一个列表转移到另一个列表。标记过程的实现并不重要，关键是策略。

垃圾回收程序运行的时候，会标记内存中存储的所有变量（记住，标记方法有很多种）。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉。在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了。随后垃圾回收程序做一次**内存清理**，销毁带标记的所有值并收回它们的内存。

到了 2008 年，IE、Firefox、Opera、Chrome 和 Safari 都在自己的 JavaScript 实现中采用标记清理（或其变体），只是在运行垃圾回收的频率上有所差异。

4.3.2 引用计数

另一种没那么常用的垃圾回收策略是**引用计数**（reference counting）。其思路是对每个值都记录它被引用的次数。声明变量并给它赋一个引用值时，这个值的引用数为 1。如果同一个值又被赋给另一个变量，那么引用数加 1。类似地，如果保存对该值引用的变量被其他值给覆盖了，那么引用数减 1。当一个值的引用数为 0 时，就说明没办法再访问到这个值了，因此可以安全地收回其内存了。垃圾回收程序下次运行的时候就会释放引用数为 0 的值的内存。

引用计数最早由 Netscape Navigator 3.0 采用，但很快就遇到了严重的问题：循环引用。所谓**循环引用**，就是对象 A 有一个指针指向对象 B，而对象 B 也引用了对象 A。比如：

```
function problem() {
  let objectA = new Object();
  let objectB = new Object();

  objectA.someOtherObject = objectB;
  objectB.anotherObject = objectA;
}
```

在这个例子中，`objectA` 和 `objectB` 通过各自的属性相互引用，意味着它们的引用数都是 2。在标记清理策略下，这不是问题，因为在函数结束后，这两个对象都不在作用域中。而在引用计数策略下，