objectA 和 objectB 在函数结束后还会存在，因为它们的引用数永远不会变成 0。如果函数被多次调用，则会导致大量内存永远不会被释放。为此，Netscape 在 4.0 版放弃了引用计数，转而采用标记清理。事实上，引用计数策略的问题还不止于此。

在 IE8 及更早版本的 IE 中，并非所有对象都是原生 JavaScript 对象。BOM 和 DOM 中的对象是 C++ 实现的组件对象模型（COM，Component Object Model）对象，而 COM 对象使用引用计数实现垃圾回收。因此，即使这些版本 IE 的 JavaScript 引擎使用标记清理，JavaScript 存取的 COM 对象依旧使用引用计数。换句话说，只要涉及 COM 对象，就无法避开循环引用问题。下面这个简单的例子展示了涉及 COM 对象的循环引用问题：

```
let element = document.getElementById("some_element");
let myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

这个例子在一个 DOM 对象（element）和一个原生 JavaScript 对象（myObject）之间制造了循环引用。myObject 变量有一个名为 element 的属性指向 DOM 对象 element，而 element 对象有一个 someObject 属性指回 myObject 对象。由于存在循环引用，因此 DOM 元素的内存永远不会被回收，即使它已经被从页面上删除了也是如此。

为避免类似的循环引用问题，应该在确保不使用的情况下切断原生 JavaScript 对象与 DOM 元素之间的连接。比如，通过以下代码可以清除前面的例子中建立的循环引用：

```
myObject.element = null;
element.someObject = null;
```

把变量设置为 null 实际上会切断变量与其之前引用值之间的关系。当下次垃圾回收程序运行时，这些值就会被删除，内存也会被回收。

为了补救这一点，IE9 把 BOM 和 DOM 对象都改成了 JavaScript 对象，这同时也避免了由于存在两套垃圾回收算法而导致的问题，还消除了常见的内存泄漏现象。

> **注意**　还有其他一些可能导致循环引用的情形，本书后面会介绍到。

4.3.3　性能

垃圾回收程序会周期性运行，如果内存中分配了很多变量，则可能造成性能损失，因此垃圾回收的时间调度很重要。尤其是在内存有限的移动设备上，垃圾回收有可能会明显拖慢渲染的速度和帧速率。开发者不知道什么时候运行时会收集垃圾，因此最好的办法是在写代码时就要做到：无论什么时候开始收集垃圾，都能让它尽快结束工作。

现代垃圾回收程序会基于对 JavaScript 运行时环境的探测来决定何时运行。探测机制因引擎而异，但基本上都是根据已分配对象的大小和数量来判断的。比如，根据 V8 团队 2016 年的一篇博文的说法：“在一次完整的垃圾回收之后，V8 的堆增长策略会根据活跃对象的数量外加一些余量来确定何时再次垃圾回收。”

由于调度垃圾回收程序方面的问题会导致性能下降，IE 曾饱受诟病。它的策略是根据分配数，比如分配了 256 个变量、4096 个对象/数组字面量和数组槽位（slot），或者 64KB 字符串。只要满足其中某个条件，垃圾回收程序就会运行。这样实现的问题在于，分配那么多变量的脚本，很可能在其整个生命周

期内始终需要那么多变量，结果就会导致垃圾回收程序过于频繁地运行。由于对性能的严重影响，IE7最终更新了垃圾回收程序。

IE7发布后，JavaScript引擎的垃圾回收程序被调优为动态改变分配变量、字面量或数组槽位等会触发垃圾回收的阈值。IE7的起始阈值都与IE6的相同。如果垃圾回收程序回收的内存不到已分配的15%，这些变量、字面量或数组槽位的阈值就会翻倍。如果有一次回收的内存达到已分配的85%，则阈值重置为默认值。这么一个简单的修改，极大地提升了重度依赖JavaScript的网页在浏览器中的性能。

> **警告** 在某些浏览器中是有可能（但不推荐）主动触发垃圾回收的。在IE中，`window.CollectGarbage()`方法会立即触发垃圾回收。在Opera 7及更高版本中，调用`window.opera.collect()`也会启动垃圾回收程序。

4.3.4 内存管理

在使用垃圾回收的编程环境中，开发者通常无须关心内存管理。不过，JavaScript运行在一个内存管理与垃圾回收都很特殊的环境。分配给浏览器的内存通常比分配给桌面软件的要少很多，分配给移动浏览器的就更少了。这更多出于安全考虑而不是别的，就是为了避免运行大量JavaScript的网页耗尽系统内存而导致操作系统崩溃。这个内存限制不仅影响变量分配，也影响调用栈以及能够同时在一个线程中执行的语句数量。

将内存占用量保持在一个较小的值可以让页面性能更好。优化内存占用的最佳手段就是保证在执行代码时只保存必要的数据。如果数据不再必要，那么把它设置为`null`，从而释放其引用。这也可以叫作**解除引用**。这个建议最适合全局变量和全局对象的属性。局部变量在超出作用域后会被自动解除引用，如下面的例子所示：

```
function createPerson(name){
  let localPerson = new Object();
  localPerson.name = name;
  return localPerson;
}

let globalPerson = createPerson("Nicholas");

// 解除 globalPerson 对值的引用

globalPerson = null;
```

在上面的代码中，变量`globalPerson`保存着`createPerson()`函数调用返回的值。在`createPerson()`内部，`localPerson`创建了一个对象并给它添加了一个`name`属性。然后，`localPerson`作为函数值被返回，并被赋值给`globalPerson`。`localPerson`在`createPerson()`执行完成超出上下文后会自动被解除引用，不需要显式处理。但`globalPerson`是一个全局变量，应该在不再需要时手动解除其引用，最后一行就是这么做的。

不过要注意，解除对一个值的引用并不会自动导致相关内存被回收。解除引用的关键在于确保相关的值已经不在上下文里了，因此它在下次垃圾回收时会被回收。

1. 通过 const 和 let 声明提升性能

ES6增加这两个关键字不仅有助于改善代码风格，而且同样有助于改进垃圾回收的过程。因为`const`

和 let 都以块（而非函数）为作用域，所以相比于使用 var，使用这两个新关键字可能会更早地让垃圾回收程序介入，尽早回收应该回收的内存。在块作用域比函数作用域更早终止的情况下，这就有可能发生。

2. 隐藏类和删除操作

根据 JavaScript 所在的运行环境，有时候需要根据浏览器使用的 JavaScript 引擎来采取不同的性能优化策略。截至 2017 年，Chrome 是最流行的浏览器，使用 V8 JavaScript 引擎。V8 在将解释后的 JavaScript 代码编译为实际的机器码时会利用“隐藏类”。如果你的代码非常注重性能，那么这一点可能对你很重要。

运行期间，V8 会将创建的对象与隐藏类关联起来，以跟踪它们的属性特征。能够共享相同隐藏类的对象性能会更好，V8 会针对这种情况进行优化，但不一定总能够做到。比如下面的代码：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
}

let a1 = new Article();
let a2 = new Article();
```

V8 会在后台配置，让这两个类实例共享相同的隐藏类，因为这两个实例共享同一个构造函数和原型。假设之后又添加了下面这行代码：

```
a2.author = 'Jake';
```

此时两个 Article 实例就会对应两个不同的隐藏类。根据这种操作的频率和隐藏类的大小，这有可能对性能产生明显影响。

当然，解决方案就是避免 JavaScript 的“先创建再补充”（ready-fire-aim）式的动态属性赋值，并在构造函数中一次性声明所有属性，如下所示：

```
function Article(opt_author) {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = opt_author;
}

let a1 = new Article();
let a2 = new Article('Jake');
```

这样，两个实例基本上就一样了（不考虑 hasOwnProperty 的返回值），因此可以共享一个隐藏类，从而带来潜在的性能提升。不过要记住，使用 delete 关键字会导致生成相同的隐藏类片段。看一下这个例子：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}

let a1 = new Article();
let a2 = new Article();

delete a1.author;
```

在代码结束后，即使两个实例使用了同一个构造函数，它们也不再共享一个隐藏类。动态删除属性与动态添加属性导致的后果一样。最佳实践是把不想要的属性设置为 null。这样可以保持隐藏类不变和继续共享，同时也能达到删除引用值供垃圾回收程序回收的效果。比如：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}

let a1 = new Article();
let a2 = new Article();

a1.author = null;
```

3. 内存泄漏

写得不好的 JavaScript 可能出现难以察觉且有害的内存泄漏问题。在内存有限的设备上，或者在函数会被调用很多次的情况下，内存泄漏可能是个大问题。JavaScript 中的内存泄漏大部分是由不合理的引用导致的。

意外声明全局变量是最常见但也最容易修复的内存泄漏问题。下面的代码没有使用任何关键字声明变量：

```
function setName() {
  name = 'Jake';
}
```

此时，解释器会把变量 name 当作 window 的属性来创建（相当于 window.name = 'Jake'）。可想而知，在 window 对象上创建的属性，只要 window 本身不被清理就不会消失。这个问题很容易解决，只要在变量声明前头加上 var、let 或 const 关键字即可，这样变量就会在函数执行完毕后离开作用域。

定时器也可能会悄悄地导致内存泄漏。下面的代码中，定时器的回调通过闭包引用了外部变量：

```
let name = 'Jake';
setInterval(() => {
  console.log(name);
}, 100);
```

只要定时器一直运行，回调函数中引用的 name 就会一直占用内存。垃圾回收程序当然知道这一点，因而就不会清理外部变量。

使用 JavaScript 闭包很容易在不知不觉间造成内存泄漏。请看下面的例子：

```
let outer = function() {
  let name = 'Jake';
  return function() {
    return name;
  };
};
```

调用 outer()会导致分配给 name 的内存被泄漏。以上代码执行后创建了一个内部闭包，只要返回的函数存在就不能清理 name，因为闭包一直在引用着它。假如 name 的内容很大（不止是一个小字符串），那可能就是个大问题了。

4. 静态分配与对象池

为了提升 JavaScript 性能，最后要考虑的一点往往就是压榨浏览器了。此时，一个关键问题就是如何减少浏览器执行垃圾回收的次数。开发者无法直接控制什么时候开始收集垃圾，但可以间接控制触发垃圾回收的条件。理论上，如果能够合理使用分配的内存，同时避免多余的垃圾回收，那就可以保住因释放内存而损失的性能。

浏览器决定何时运行垃圾回收程序的一个标准就是对象更替的速度。如果有很多对象被初始化，然后一下子又都超出了作用域，那么浏览器就会采用更激进的方式调度垃圾回收程序运行，这样当然会影响性能。看一看下面的例子，这是一个计算二维矢量加法的函数：

```
function addVector(a, b) {
  let resultant = new Vector();
  resultant.x = a.x + b.x;
  resultant.y = a.y + b.y;
  return resultant;
}
```

调用这个函数时，会在堆上创建一个新对象，然后修改它，最后再把它返回给调用者。如果这个矢量对象的生命周期很短，那么它会很快失去所有对它的引用，成为可以被回收的值。假如这个矢量加法函数频繁被调用，那么垃圾回收调度程序会发现这里对象更替的速度很快，从而会更频繁地安排垃圾回收。

该问题的解决方案是不要动态创建矢量对象，比如可以修改上面的函数，让它使用一个已有的矢量对象：

```
function addVector(a, b, resultant) {
  resultant.x = a.x + b.x;
  resultant.y = a.y + b.y;
  return resultant;
}
```

当然，这需要在其他地方实例化矢量参数 `resultant`，但这个函数的行为没有变。那么在哪里创建矢量可以不让垃圾回收调度程序盯上呢？

一个策略是使用对象池。在初始化的某一时刻，可以创建一个对象池，用来管理一组可回收的对象。应用程序可以向这个对象池请求一个对象、设置其属性、使用它，然后在操作完成后再把它还给对象池。由于没发生对象初始化，垃圾回收探测就不会发现有对象更替，因此垃圾回收程序就不会那么频繁地运行。下面是一个对象池的伪实现：

```
// vectorPool 是已有的对象池
let v1 = vectorPool.allocate();
let v2 = vectorPool.allocate();
let v3 = vectorPool.allocate();

v1.x = 10;
v1.y = 5;
v2.x = -3;
v2.y = -6;

addVector(v1, v2, v3);

console.log([v3.x, v3.y]); // [7, -1]

vectorPool.free(v1);
vectorPool.free(v2);
vectorPool.free(v3);

// 如果对象有属性引用了其他对象
// 则这里也需要把这些属性设置为 null
v1 = null;
v2 = null;
v3 = null;
```

如果对象池只按需分配矢量（在对象不存在时创建新的，在对象存在时则复用存在的），那么这个实现本质上是一种贪婪算法，有单调增长但为静态的内存。这个对象池必须使用某种结构维护所有对象，数组是比较好的选择。不过，使用数组来实现，必须留意不要招致额外的垃圾回收。比如下面这个例子：

```
let vectorList = new Array(100);
let vector = new Vector();
vectorList.push(vector);
```

由于 JavaScript 数组的大小是动态可变的，引擎会删除大小为 100 的数组，再创建一个新的大小为 200 的数组。垃圾回收程序会看到这个删除操作，说不定因此很快就会跑来收一次垃圾。要避免这种动态分配操作，可以在初始化时就创建一个大小够用的数组，从而避免上述先删除再创建的操作。不过，必须事先想好这个数组有多大。

> **注意** 静态分配是优化的一种极端形式。如果你的应用程序被垃圾回收严重地拖了后腿，可以利用它提升性能。但这种情况并不多见。大多数情况下，这都属于过早优化，因此不用考虑。

4.4 小结

JavaScript 变量可以保存两种类型的值：原始值和引用值。原始值可能是以下 6 种原始数据类型之一：`Undefined`、`Null`、`Boolean`、`Number`、`String` 和 `Symbol`。原始值和引用值有以下特点。

- 原始值大小固定，因此保存在栈内存上。
- 从一个变量到另一个变量复制原始值会创建该值的第二个副本。
- 引用值是对象，存储在堆内存上。
- 包含引用值的变量实际上只包含指向相应对象的一个指针，而不是对象本身。
- 从一个变量到另一个变量复制引用值只会复制指针，因此结果是两个变量都指向同一个对象。
- `typeof` 操作符可以确定值的原始类型，而 `instanceof` 操作符用于确保值的引用类型。

任何变量（不管包含的是原始值还是引用值）都存在于某个执行上下文中（也称为作用域）。这个上下文（作用域）决定了变量的生命周期，以及它们可以访问代码的哪些部分。执行上下文可以总结如下。

- 执行上下文分全局上下文、函数上下文和块级上下文。
- 代码执行流每进入一个新上下文，都会创建一个作用域链，用于搜索变量和函数。
- 函数或块的局部上下文不仅可以访问自己作用域内的变量，而且也可以访问任何包含上下文乃至全局上下文中的变量。
- 全局上下文只能访问全局上下文中的变量和函数，不能直接访问局部上下文中的任何数据。
- 变量的执行上下文用于确定什么时候释放内存。

JavaScript 是使用垃圾回收的编程语言，开发者不需要操心内存分配和回收。JavaScript 的垃圾回收程序可以总结如下。

- 离开作用域的值会被自动标记为可回收，然后在垃圾回收期间被删除。
- 主流的垃圾回收算法是标记清理，即先给当前不使用的值加上标记，再回来回收它们的内存。

- 引用计数是另一种垃圾回收策略，需要记录值被引用了多少次。JavaScript 引擎不再使用这种算法，但某些旧版本的 IE 仍然会受这种算法的影响，原因是 JavaScript 会访问非原生 JavaScript 对象（如 DOM 元素）。
- 引用计数在代码中存在循环引用时会出现问题。
- 解除变量的引用不仅可以消除循环引用，而且对垃圾回收也有帮助。为促进内存回收，全局对象、全局对象的属性和循环引用都应该在不需要时解除引用。

第5章 基本引用类型

本章内容

- 理解对象
- 基本 JavaScript 数据类型
- 原始值与原始值包装类型

引用值（或者对象）是某个特定**引用类型**的实例。在 ECMAScript 中，引用类型是把数据和功能组织到一起的结构，经常被人错误地称作“类”。虽然从技术上讲 JavaScript 是一门面向对象语言，但 ECMAScript 缺少传统的面向对象编程语言所具备的某些基本结构，包括类和接口。引用类型有时候也被称为**对象定义**，因为它们描述了自己的对象应有的属性和方法。

> **注意** 引用类型虽然有点像类，但跟类并不是一个概念。为避免混淆，本章后面不会使用术语“类”。

对象被认为是某个特定引用类型的**实例**。新对象通过使用 new 操作符后跟一个**构造函数**（constructor）来创建。构造函数就是用来创建新对象的函数，比如下面这行代码：

```
let now = new Date();
```

这行代码创建了引用类型 Date 的一个新实例，并将它保存在变量 now 中。Date()在这里就是构造函数，它负责创建一个只有默认属性和方法的简单对象。ECMAScript 提供了很多像 Date 这样的原生引用类型，帮助开发者实现常见的任务。

> **注意** 函数也是一种引用类型，但有关函数的内容太多了，一章放不下，所以本书专门用第 10 章来介绍函数。

5.1 Date

ECMAScript 的 Date 类型参考了 Java 早期版本中的 java.util.Date。为此，Date 类型将日期保存为自协调世界时（UTC，Universal Time Coordinated）时间 1970 年 1 月 1 日午夜（零时）至今所经过的毫秒数。使用这种存储格式，Date 类型可以精确表示 1970 年 1 月 1 日之前及之后 285 616 年的日期。

要创建日期对象，就使用 new 操作符来调用 Date 构造函数：

```
let now = new Date();
```

在不给 Date 构造函数传参数的情况下，创建的对象将保存当前日期和时间。要基于其他日期和时间创建日期对象，必须传入其毫秒表示（UNIX 纪元 1970 年 1 月 1 日午夜之后的毫秒数）。ECMAScript 为此提供了两个辅助方法：Date.parse()和 Date.UTC()。

Date.parse()方法接收一个表示日期的字符串参数，尝试将这个字符串转换为表示该日期的毫秒数。ECMA-262 第 5 版定义了 Date.parse()应该支持的日期格式，填充了第 3 版遗留的空白。所有实现都必须支持下列日期格式：

- “月/日/年”，如"5/23/2019"；
- “月名 日, 年”，如"May 23, 2019"；
- “周几 月名 日 年 时:分:秒 时区”，如"Tue May 23 2019 00:00:00 GMT-0700"；
- ISO 8601 扩展格式“YYYY-MM-DDTHH:mm:ss.sssZ”，如 2019-05-23T00:00:00（只适用于兼容 ES5 的实现）。

比如，要创建一个表示“2019 年 5 月 23 日”的日期对象，可以使用以下代码：

```
let someDate = new Date(Date.parse("May 23, 2019"));
```

如果传给 Date.parse()的字符串并不表示日期，则该方法会返回 NaN。如果直接把表示日期的字符串传给 Date 构造函数，那么 Date 会在后台调用 Date.parse()。换句话说，下面这行代码跟前面那行代码是等价的：

```
let someDate = new Date("May 23, 2019");
```

这两行代码得到的日期对象相同。

> **注意**　不同的浏览器对 Date 类型的实现有很多问题。比如，很多浏览器会选择用当前日期替代越界的日期，因此有些浏览器会将"January 32, 2019"解释为"February 1, 2019"。Opera 则会插入当前月的当前日，返回"January 当前日, 2019"。就是说，如果是在 9 月 21 日运行代码，会返回"January 21, 2019"。

Date.UTC()方法也返回日期的毫秒表示，但使用的是跟 Date.parse()不同的信息来生成这个值。传给 Date.UTC()的参数是年、零起点月数（1 月是 0，2 月是 1，以此类推）、日（1~31）、时（0~23）、分、秒和毫秒。这些参数中，只有前两个（年和月）是必需的。如果不提供日，那么默认为 1 日。其他参数的默认值都是 0。下面是使用 Date.UTC()的两个例子：

```
// GMT 时间 2000 年 1 月 1 日零点
let y2k = new Date(Date.UTC(2000, 0));

// GMT 时间 2005 年 5 月 5 日下午 5 点 55 分 55 秒
let allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

这个例子创建了两个日期 。第一个日期是 2000 年 1 月 1 日零点（GMT），2000 代表年，0 代表月（1 月）。因为没有其他参数（日取 1，其他取 0），所以结果就是该月第 1 天零点。第二个日期表示 2005 年 5 月 5 日下午 5 点 55 分 55 秒（GMT）。虽然日期里面涉及的都是 5，但月数必须用 4，因为月数是零起点的。小时也必须是 17，因为这里采用的是 24 小时制，即取值范围是 0~23。其他参数就都很直观了。

与 Date.parse()一样，Date.UTC()也会被 Date 构造函数隐式调用，但有一个区别：这种情况下创建的是本地日期，不是 GMT 日期。不过 Date 构造函数跟 Date.UTC()接收的参数是一样的。因此，如果第一个参数是数值，则构造函数假设它是日期中的年，第二个参数就是月，以此类推。前面的

例子也可以这样来写：

```
// 本地时间 2000 年 1 月 1 日零点
let y2k = new Date(2000, 0);

// 本地时间 2005 年 5 月 5 日下午 5 点 55 分 55 秒
let allFives = new Date(2005, 4, 5, 17, 55, 55);
```

以上代码创建了与前面例子中相同的两个日期，但这次的两个日期是（由于系统设置决定的）本地时区的日期。

ECMAScript 还提供了 `Date.now()`方法，返回表示方法执行时日期和时间的毫秒数。这个方法可以方便地用在代码分析中：

```
// 起始时间
let start = Date.now();

// 调用函数
doSomething();

// 结束时间
let stop = Date.now(),
result = stop - start;
```


5.1.1 继承的方法

与其他类型一样，`Date` 类型重写了 `toLocaleString()`、`toString()`和 `valueOf()`方法。但与其他类型不同，重写后这些方法的返回值不一样。`Date` 类型的 `toLocaleString()`方法返回与浏览器运行的本地环境一致的日期和时间。这通常意味着格式中包含针对时间的 AM（上午）或 PM（下午），但不包含时区信息（具体格式可能因浏览器而不同）。`toString()`方法通常返回带时区信息的日期和时间，而时间也是以 24 小时制（0~23）表示的。下面给出了 `toLocaleString()`和 `toString()`返回的 2019 年 2 月 1 日零点的示例（地区为`"en-US"`的 PST，即 Pacific Standard Time，太平洋标准时间）：

```
toLocaleString() - 2/1/2019 12:00:00 AM

toString() - Thu Feb 1 2019 00:00:00 GMT-0800 (Pacific Standard Time)
```

现代浏览器在这两个方法的输出上已经趋于一致。在比较老的浏览器上，每个方法返回的结果可能在每个浏览器上都是不同的。这些差异意味着 `toLocaleString()`和 `toString()`可能只对调试有用，不能用于显示。

`Date` 类型的 `valueOf()`方法根本就不返回字符串，这个方法被重写后返回的是日期的毫秒表示。因此，操作符（如小于号和大于号）可以直接使用它返回的值。比如下面的例子：

```
let date1 = new Date(2019, 0, 1);    // 2019 年 1 月 1 日
let date2 = new Date(2019, 1, 1);    // 2019 年 2 月 1 日

console.log(date1 < date2); // true
console.log(date1 > date2); // false
```

日期 2019 年 1 月 1 日在 2019 年 2 月 1 日之前，所以说前者小于后者没问题。因为 2019 年 1 月 1 日的毫秒表示小于 2019 年 2 月 1 日的毫秒表示，所以用小于号比较这两个日期时会返回 `true`。这也是确保日期先后的一个简单方式。