第6条：消除过期的对象引用

当你从手工管理内存的语言（比如C或C++）转换到具有垃圾回收功能的语言的时候，程序员的工作会变得更加容易，因为当你用完了对象之后，它们会被自动回收。当你第一次经历对象回收功能的时候，会觉得这简直有点不可思议。这很容易给你留下这样的印象，认为自己不再需要考虑内存管理的事情了。其实不然。

考虑下面这个简单的栈实现的例子：

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这段程序（它的泛型版本请见第26条）中并没有很明显的错误。无论如何测试，它都会成功地通过每一项测试，但是这个程序中隐藏着一个问题。不严格地讲，这段程序有一个“内存泄漏”，随着垃圾回收器活动的增加，或者由于内存占用的不断增加，程序性能的降低会逐渐表现出来。在极端的情况下，这种内存泄漏会导致磁盘交换（Disk Paging），甚至导致程序失败（OutOfMemoryError错误），但是这种失败情形相对比较少见。

那么，程序中哪里发生了内存泄漏呢？如果一个栈先是增长，然后再收缩，那么，从栈中弹出来的对象将不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。这是因为，栈内部维护着对这些对象的过期引用（obsolete reference）。所谓的过期引

用，是指永远也不会再被解除的引用。在本例中，凡是在elements数组的“活动部分（active portion）”之外的任何引用都是过期的。活动部分是指elements中下标小于size的那些元素。

在支持垃圾回收的语言中，内存泄漏是很隐蔽的（称这类内存泄漏为“无意识的对象保持（unintentional object retention）”更为恰当）。如果一个对象引用被无意识地保留起来了，那么，垃圾回收机制不仅不会处理这个对象，而且也不会处理被这个对象所引用的所有其他对象。即使只有少量的几个对象引用被无意识地保留下来，也会有许许多多的对象被排除在垃圾回收机制之外，从而对性能造成潜在的重大影响。

这类问题的修复方法很简单：一旦对象引用已经过期，只需清空这些引用即可。对于上述例子中的Stack类而言，只要一个单元被弹出栈，指向它的引用就过期了。pop方法的修订版本如下所示：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

清空过期引用的另一个好处是，如果它们以后又被错误地解除引用，程序就会立即抛出NullPointerException异常，而不是悄悄地错误运行下去。尽快地检测出程序中的错误总是有益的。

当程序员第一次被类似这样的问题困扰的时候，他们往往会过分小心：对于每一个对象引用，一旦程序不再用到它，就把它清空。其实这样做既没必要，也不是我们所期望的，因为这样做会把程序代码弄得很乱。清空对象引用应该是一种例外，而不是一种规范行为。消除过期引用最好的方法是让包含该引用的变量结束其生命周期。如果你是在最紧凑的作用域范围内定义每一个变量（见第45条），这种情形就会自然而然地发生。

那么，何时应该清空引用呢？Stack类的哪方面特性使它易于遭受内存泄漏的影响呢？简而言之，问题在于，Stack类自己管理内存（manage its own memory）。存储池（storage pool）包含了elements数组（对象引用单元，而不是对象本身）的元素。数组活动区域（同前面的定义）中的元素是已分配的（allocated），而数组其余部分的元素则是自由的（free）。但是垃圾回收器并不知道这一点；对于垃圾回收器而言，elements数组中的所有对象引用都同等有效。只有程序员知道数组的非活动部分是不重要的。程序员可以把这个情况告知垃圾回收器，做法很简单：一旦数组元素变成了非活动部分的一部分，程序员就手工清空这些数组元素。

一般而言，只要类是自己管理内存，程序员就应该警惕内存泄漏问题。一旦元素被释放掉，则该元素中包含的任何对象引用都应该被清空。

内存泄漏的另一个常见来源是缓存。一旦你把对象引用放到缓存中，它就很容易被遗忘掉，从而使得它不再有用之后很长一段时间内仍然留在缓存中。对于这个问题，有几种可能的解决方案。如果你正好要实现这样的缓存：只要在缓存之外存在对某个项的键的引用，该项就有意义，那么就可以用WeakHashMap代表缓存；当缓存中的项过期之后，它们就会自动被删除。记住只有当所要的缓存项的生命周期是由该键的外部引用而不是由值决定时，WeakHashMap才有用处。

更为常见的情形则是，“缓存项的生命周期是否有意义”并不是很容易确定，随着时间的推移，其中的项会变得越来越没有价值。在这种情况下，缓存应该时不时地清除掉没用的项。这项清除工作可以由一个后台线程（可能是Timer或者ScheduledThreadPoolExecutor）来完成，或者也可以在给缓存添加新条目的时候顺便进行清理。LinkedHashMap类利用它的removeEldestEntry方法可以很容易地实现后一种方案。对于更加复杂的缓存，必须直接使用java.lang.ref。

内存泄漏的第三个常见来源是监听器和其他回调。如果你实现了一个API，客户端在这个API中注册回调，却没有显式地取消注册，那么除非你采取某些动作，否则它们就会积聚。确保回调立即被当作垃圾回收的最佳方法是只保存它们的弱引用（weak reference），例如，只将它们保存成WeakHashMap中的键。

由于内存泄漏通常不会表现成明显的失败，所以它们可以在一个系统中存在很多年。往往只有通过仔细检查代码，或者借助于Heap剖析工具（Heap Profiler）才能发现内存泄漏问题。因此，如果能够在内存泄漏发生之前就知道如何预测此类问题，并阻止它们发生，那是最好不过的了。

第7条：避免使用终结方法

*终结方法（finalizer）通常是不可预测的，也是很危险的，一般情况下是不必要的。*使用终结方法会导致行为不稳定、降低性能，以及可移植性问题。当然，终结方法也有其可用之处，我们将在本条目的最后再做介绍；但是根据经验，应该避免使用终结方法。

C++的程序员被告知“不要把终结方法当作是C++中析构器（destructors）的对应物”。在C++中，析构器是回收一个对象所占用资源的常规方法，是构造器所必需的对应物。在Java中，当一个对象变得不可到达的时候，垃圾回收器会回收与该对象相关联的存储空间，并不需要程序员做专门的工作。C++的析构器也可以被用来回收其他的非内存资源。而在Java中，一般用try-finally块来完成类似的工作。

终结方法的缺点在于不能保证会被及时地执行[JLS, 12.6]。从一个对象变得不可到达开始，到它的终结方法被执行，所花费的这段时间是任意长的。这意味着，注重时间（time-critical）的任务不应该由终结方法来完成。例如，用终结方法来关闭已经打开的文件，这是严重错误，因为打开文件的描述符是一种很有限的资源。由于JVM会延迟执行终结方法，所以大量的文件会保留在打开状态，当一个程序再不能打开文件的时候，它可能会运行失败。

及时地执行终结方法正是垃圾回收算法的一个主要功能，这种算法在不同的JVM实现中会大相径庭。如果程序依赖于终结方法被执行的时间点，那么这个程序的行为在不同的JVM中运行的表现可能就会截然不同。一个程序在你测试用的JVM平台上运行得非常好，而在你最重要顾客的JVM平台上却根本无法运行，这是完全有可能的。

延迟终结过程并不只是一个理论问题。在很少见的情况下，为类提供终结方法，可能会随意地延迟其实例的回收过程。一位同事最近在调试一个长期运行的GUI应用程序的时候，该应用程序莫名其妙地出现OutOfMemoryError错误而死掉。分析表明，该应用程序死掉的时候，其终结方法队列中有数千个图形对象正在等待被终结和回收。遗憾的是，终结方法线程的优先级比该应用程序的其他线程的要低得多，所以，图形对象的终结速度达不到它们进入队列的速度。Java语言规范并不保证哪个线程将会执行终结方法，所以，除了不使用终结方法之外，并没有很轻便的办法能够避免这样的问题。

Java语言规范不仅不保证终结方法会被及时地执行，而且根本就不保证它们会被执行。当一个程序终止的时候，某些已经无法访问的对象上的终结方法却根本没有被执行，这是完全有可能的。结论是：*不应该依赖终结方法来更新重要的持久状态。*例如，依赖终结方法来释放共享资源（比如数据库）上的永久锁，很容易让整个分布式系统垮掉。

不要被System.gc和System.runFinalization这两个方法所诱惑，它们确实增加了终结方法

被执行的机会，但是它们并不保证终结方法一定会被执行。唯一声称保证终结方法被执行的方法是System.runFinalizersOnExit，以及它臭名昭著的孪生兄弟Runtime.runFinalizersOnExit。这两个方法都有致命的缺陷，已经被废弃了[ThreadStop]。

当你并不确定是否应该避免使用终结方法的时候，这里还有一种值得考虑的情形：如果未被捕获的异常在终结过程中被抛出来，那么这种异常可以被忽略，并且该对象的终结过程也会终止[JLS, 12.6]。未被捕获的异常会使对象处于破坏的状态（a corrupt state），如果另一个线程企图使用这种被破坏的对象，则可能发生任何不确定的行为。正常情况下，未被捕获的异常将会使线程终止，并打印出栈轨迹（Stack Trace），但是，如果异常发生在终结方法之中，则不会如此，甚至连警告都不会打印出来。

还有一点：使用终结方法有一个非常严重的（**Severe**）性能损失。在我的机器上，创建和销毁一个简单对象的时间大约为5.6ns。增加一个终结方法使时间增加到了2 400ns。换句话说，用终结方法创建和销毁对象慢了大约430倍。

那么，如果类的对象中封装的资源（例如文件或者线程）确实需要终止，应该怎么做才能不用编写终结方法呢？只需提供一个显式的终止方法，并要求该类的客户端在每个实例不再有用的时候调用这个方法。值得提及的一个细节是，该实例必须记录下自己是否已经被终止了：显式的终止方法必须在一个私有域中记录下“该对象已经不再有效”。如果这些方法是在对象已经终止之后被调用，其他的方法就必须检查这个域，并抛出IllegalStateException异常。

显式终止方法的典型例子是InputStream、OutputStream和java.sql.Connection上的close方法。另一个例子是java.util.Timer上的cancel方法，它执行必要的状态改变，使得与Timer实例相关联的该线程温和地终止自己。java.awt中的例子还包括Graphics.dispose和Window.dispose。这些方法通常由于性能不好而不被人们关注。一个相关的方法是Image.flush，它会释放所有与Image实例相关联的资源，但是该实例仍然处于可用的状态，如果有必要的话，会重新分配资源。

显式的终止方法通常与**try-finally**结构结合起来使用，以确保及时终止。在finally子句内部调用显式的终止方法，可以保证即使在使用对象的时候有异常抛出，该终止方法也会执行：

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

那么终结方法有什么好处呢？它们有两种合法用途。第一种用途是，当对象的所有者忘记调用前面段落中建议的显式终止方法时，终结方法可以充当“安全网（safety net）”。虽然这

样做并不能保证终结方法会被及时地调用，但是在客户端无法通过调用显式的终止方法来正常结束操作的情况下（希望这种情形尽可能地少发生），迟一点释放关键资源总比永远不释放要好。但是如果终结方法发现资源还未被终止，则应该在日志中记录一条警告，因为这表示客户端代码中的一个Bug，应该得到修复。如果你正考虑编写这样的安全网终结方法，就要认真考虑清楚，这种额外的保护是否值得你付出这份额外的代价。

显式终止方法模式的示例中所示的四个类（FileInputStream、FileOutputStream、Timer和Connection），都具有终结方法，当它们的终止方法未能被调用的情况下，这些终结方法充当了安全网。

终结方法的第二种合理用途与对象的本地对等体（**native peer**）有关。本地对等体是一个本地对象（native object），普通对象通过本地方法（native method）委托给一个本地对象。因为本地对等体不是一个普通对象，所以垃圾回收器不会知道它，当它的Java对等体被回收的时候，它不会被回收。在本地对等体并不拥有关键资源的前提下，终结方法正是执行这项任务最合适的工具。如果本地对等体拥有必须被及时终止的资源，那么该类就应该具有一个显式的终止方法，如前所述。终止方法应该完成所有必要的工作以便释放关键的资源。终止方法可以是本地方法，或者它也可以调用本地方法。

值得注意的很重要一点是，“终结方法链（finalizer chaining）”并不会被自动执行。如果类（不是Object）有终结方法，并且子类覆盖了终结方法，子类的终结方法就必须手工调用超类的终结方法。你应该在一个try块中终结子类，并在相应的finally块中调用超类的终结方法。这样做可以保证：即使子类的终结过程抛出异常，超类的终结方法也会得到执行。反之亦然。代码示例如下。注意这个示例使用了Override注解（@Override），这是Java 1.5发行版本将它增加到Java平台中的。你现在可以不管Override注解，或者到第36条查阅一下它们表示什么意思：

```
// Manual finalizer chaining
@Override protected void finalize() throws Throwable {
    try {
        ... // Finalize subclass state
    } finally {
        super.finalize();
    }
}
```

如果子类实现者覆盖了超类的终结方法，但是忘了手工调用超类的终结方法（或者有意选择不调用超类的终结方法），那么超类的终结方法将永远也不会被调用到。要防范这样粗心大意或者恶意的子类是有可能的，代价就是为每个将被终结的对象创建一个附加的对象。不是把终结方法放在要求终结处理的类中，而是把终结方法放在一个匿名的类（见第22条）中，该匿名类的唯一用途就是终结它的外围实例（enclosing instance）。该匿名类的单个实例被称为终结方法守卫者（**finalizer guardian**），外围类的每个实例都会创建这样一个守卫者。外围实例在它的私有实例域中保存着一个对其终结方法守卫者的唯一引用，因此终结方法守卫者

与外围实例可以同时启动终结过程。当守卫者被终结的时候，它执行外围实例所期望的终结行为，就好像它的终结方法是外围对象上的一个方法一样：

```
// Finalizer Guardian idiom
public class Foo {
   // Sole purpose of this object is to finalize outer Foo object
   private final Object finalizerGuardian = new Object() {
      @Override protected void finalize() throws Throwable {
         ... // Finalize outer Foo object
      }
   };
   ...  // Remainder omitted
}
```

注意，公有类Foo并没有终结方法（除了它从Object中继承了一个无关紧要的之外），所以子类的终结方法是否调用super.finalize并不重要。对于每一个带有终结方法的非final公有类，都应该考虑使用这种方法。

总之，除非是作为安全网，或者是为了终止非关键的本地资源，否则请不要使用终结方法。在这些很少见的情况下，既然使用了终结方法，就要记住调用super.finalize。如果用终结方法作为安全网，要记得记录终结方法的非法用法。最后，如果需要把终结方法与公有的非final类关联起来，请考虑使用终结方法守卫者，以确保即使子类的终结方法未能调用super.finalize，该终结方法也会被执行。