计算机科学2002Vol.29№.4

Java程序内存空间优化策略的研究

Strategy on Optimizing Memory Size for Java Applications

徐 鹏 王克宏

(清华大学计算机系 北京100084)

Abstract A Java class can be interpreted by Java Virtual Machine directly. However, the size of class will impact the performance on program loading and running. As Java technology has been adopted by more and more application systems and the amount of data shoots up continuously, the strategy on optimizing memory size for Java applications has been considers as an important issue. In this paper, we propose the solutions in two ways, by reducing the size of class file and descending the memory overload. Additionally, we analyze the design pattern for avoiding memory consuming by object-oriented technology in the process of developing software, and provide the solutions on memory leaking.

Keywords Memory optimization, Obfuscation, Memory leaking, Java VM, Unique mode

一、引言

基于Java的应用程序在运行时由于受到语言本身的特点和虚拟机的限制,对于客户端硬件配置的要求相对于Visual C++和Visual Basic语言编写的程序要高。虽然Java 2对于JDK虚拟机解释器进行了优化处理,但是仍然不能够完全令用户满意。因此,Java程序的开发人员在对程序代码进行优化时,很自然地将关注焦点放在提高运行性能上。然而,Java程序还可以从另一方面得到优化,即对程序所占用内存空间进行优化。很多功能复杂的Java应用程序在运行时由于系统频繁针对内存进行存储交换操作,从而在很大程度上影响了软件运行的性能。程序中存在的内存漏洞造成了程序通过操作系统消耗内存,而没有及时进行对内存空间的复用和释放,这最终造成了程序消耗的内存空间逐渐加大。从软件工程的角度分析,对空间的优化处理主要以两种形式表现出来:降低类文件的大小和降低程序运行时对内存的占用量。本文将从这两方面提出相应的优化策略。这两方面的优化策略从实现角度来看可以划分在三个不同的级别上进行:①编译器级:通过类文件的编译优化处理实现;②程序员级:通过内存管理和程序优化处理实现;③优化工具级:通过现有的实用软件工具检测内存使用状况并优化。

二、类文件的编译优化处理

在Java开发过程中,Java源代码被编译成一种中间形式,称为字节代码(Java类文件)。从理论上讲,这种代码是具有平台独立性的,可以运行在任何具备Java虚拟机的计算平台上[1]。当然,Java源代码是以提高可读性为标准的方式编写的。Java编译器的工作就是获得源代码并产生相应的字节码。Java虚拟机将字节码转换成主机平台的本地代码。在Web环境下运行Applet程序之前,类文件首先必须被下载到客户端。类文件越大,程序下载时间越长。特别是对于网络并不发达的地区,由于网络的传输速度较慢,因此这种问题表现得更加突出。

从编译的角度入手,我们可以采用以下三种方式来降低类文件的大小:

1、使用Java编译器的优化选项 在缺省条件下,JDK提供的javac编译器生成的类文件包含程序行号信息(加入行号信息的作用是便于用户进行调试,同时在程序运行时如果出现例外,则解释器会利用行号信息显示出例外发生的行)。如果使用-g选项进行编译操作,那么类文件也包含变量调试信息。这些额外的数据在类文件中占据了一定比例。通过使用-O选项启动编译优化器进行编译,则可以强制javac编译器删除这些额外数据。这种优化选项也可以造成javac编译器删除一些冗余的代码,但是同时它也内嵌一些方法用于改进性能(这些方法也占据了类文件的一定空间)。与一个标准的C编译器不同,javac的-O选项在优化方面所做的工作还非常有限。

2. 以Java档案文件(JAR文件)的形式对类进行压缩处理 从Java 1.1开始,JDK就提供了生成Java档案文件(JAR文件)的功能。这种经过压缩处理的文件中包含了Java类,其压缩方法基于ZIP压缩格式。档案文件的使用显著地降低了类文件所占用的磁盘空间,例如标准的Java类库经过压缩处理后所占用的磁盘空间从8.63MB降低到5.03MB。这是一种最简单也是最有效的降低类文件所占空间的方法。此外,因为档案文件将多个类打包在一个文件里,所以在Applet下载过程中只涉及了一次FTP事务处理,因此下载时间大大加快了。开发人员可以使用JDK提供的jar工具来生成JAR文件。例如,jar -cf classes.jar *.class命令实现了将当前目录中所有类文件封装成JAR文件的功能。在HTML文件中的APPLET标签支持ARCHIVE参数,该参数的取值为当前Applet程序中所使用的JAR文件路径。如果我们在一个Java应用程序中引用JAR文件,则我们必须在系统环境变量CLASSPATH中加入JAR文件的路径。

3. 使用扰码处理技术降低空间占用量 Java解释器实现对类的动态链接,也就是说,解释器并不是一次将所有类进行加载,而是在运行时遇到类引用时才对其进行加载。正是由于这个原因,每个类文件必须包含标识符的名称。这就导致了可能由于标识符名称过于冗长而导致类文件所占磁盘空间的加大。当然,这并不是使用简短但含糊不清的标识符名称的理由。扰码处理使字节代码经过反编译后不可理解,此处理首先修改所有变量和方法的名称。目前适用的扰码处理器很多。扰码处理器实现了重命名技术,该技术称为"负载归纳",即在实

现上不是将每个现有名称重新命名，而是尽可能将方法重命名为相同的名称。重新命名的标识符简短且隐秘，这也是降低类文件大小的一种方法。

当然，在使用过程中我们要注意以下几点。首先，必须对 Applet 中使用的所有类进行扰码处理，否则虚拟机将不能够对类和方法标识符进行匹配。这意味着您不能够对希望提供外部访问功能的类进行扰码处理。此外，任何通过字符串引用一个标识符的程序将失败(因为标识符已经被重新命名，但是字符串仍然按照原始的命名设置)。例如，方法 getClass().getMethod("_方法名称_",null)将失败。然而，在许多应用程序中这种现象不会发生。

针对类文件采用以上不同的优化处理技术将对类文件所占空间起到不同的缩减作用。下表展示了不同策略对 java.lang.Vector 类文件的作用。同样的类文件经过不同方式的优化处理后文件大小最多可以减半，这样就可以使下载时间缩短一半。针对类文件的编译优化方式所能够达到的效果主要依赖于 Java 虚拟机的优化程度。

表1 编译方式对 Vector 类大小的影响

编译方式	类文件(字节)
标准 javac 编译并通过调试	5330
标准 javac 编译	4163
经过优化后的编译	3398
经过优化和扰码处理的编译	3107
经过标准编译，同时对 JAR 文件进行优化	2103
经过优化编译，同时在 JAR 文件中采用扰码处理	1934

三、运行时程序优化与内存管理

在讨论降低运行时内存消耗的多种方法之前，首先了解一下对象实例是以何种形式存储在内存中。对象实例不同于数据类型，它们包含了操作数据的方法。程序中的对象通常是以两种方式存储的。最简单的方法是存储每个对象内部方法代码的指针。这是一种快捷的方式，因为调用一个方法只需要虚拟机将一个指针指向方法代码。缺陷也是明显的，即方法越多，则对象实例变得越大。另外一种方式是使一个类的每个实例包含对类实例的引用，它包含了针对该类方法的指针。这第二种方式明显要慢，因为虚拟机必需跟踪两个指针(其一是指向方法表，其二是指向方法)。然而，其重要的优势在于方法数量的增加并没有增加实例的大小[2]。虽然 Java 规范没有定义如何在内存中存储对象，但是大多数主流的虚拟机使用了一种相同的方式。

Java 程序在运行时处理内存短缺问题方面的解决方案通常是使用 Java 语言提供的无用内存单元回收功能(又称为垃圾回收功能)。程序在运行过程中需要构造对象实例，这时程序解释器向操作系统发出内存申请，操作系统从内存堆中预留一块内存空间，并使用这个空间来存储对象实例变量的值[4]。在程序运行过程中对象、变量和方法的构造都会占用一定的内存空间，而某些变量或对象在完成了一次或几次调用后不会继续被程序所使用。"无用单元回收"这一概念的含义就是程序中一个实用线程或进程跟踪程序，对那些不再被使用的内存单元进行清理，并将释放后的内存单元归还操作系统或程序以便复用。在这种前提下，开发人员并不需要特别指定从内存中删除特定对象的时间。传统的面向对象程序设计语言在对象使用完成时，可以通过调用特定的方法来释放特定对象占用的内存空间。如果程序员在设计程序过程中忘记释放所有冗余的对象，则随着时间的延长，空闲内存的数量将逐渐减少。这就是所谓的"内存泄漏"现象。内存泄漏现象的出现不但影响了程序的运行性能，而且会造成运行时内存不足而导致系统运行意外中断等例外情况的发生。

Java 语言也是按照传统的方式为对象分配内存空间的，但是并不要求程序员在完成对特定对象的处理之后明确释放这个对象。取而代之的是 Java 解释器使用一个完成"无用单元自动回收"功能的专用进程对内存中保存的所有对象进行校验。如果发现存在不被任何变量引用的对象，则该对象占用的内存空间符合被回收的条件。Java 语言的自动无用单元回收功能能够对弥补内存使用方面的漏洞起关键作用，但是在一个 Java 程序中仍然或多或少存在这个问题。

在运行时 Java 程序中造成内存泄漏问题的三个主要原因包括：①未知或不必要的对象引用；②清除或释放本地系统资源失败；③在 JDK 或第三方类库中存在的错误。下面主要分析 Java 程序造成的内存泄漏现象产生的原因，并从软件设计的角度出发，提出相应的解决方案。

3.1 Java 语言垃圾回收功能的实现

自动垃圾收集的思想是在程序内部实现一个永动的实用线程或进程，用于清理所有与该程序相关的无用内存单元，将这些单元释放给操作系统或程序以实现对其复用。而开发人员不需要专门制定何时将对象从内存中删除。这样就解决了造成内存漏洞的主要诱因。

Java 语言提供的无用单元自动回收机制是通过 Java 虚拟机来实现的。目前，在无用单元精确回收方面，Java 虚拟机主要采用保守式回收，即采用 stop-and-copy 算法(标记和清除算法)[3]。stop-and-copy 算法要求每个对象包含一个称为标记位的域，或者要求当该算法运行时创建一个外部数组存放标记位。算法开始时扫描堆中所有已经分配的内存块并且复位该内存块的标记位。然后，检查引用堆中所有对象的所有域和变量，设置被引用的对象的标记位为真。最后扫描已分配的堆中的对象，找出任何没有被标记的对象。然后有两种选择：a)把这些未用的对象放到空闲列表中来释放空间；b)复制活动对象到末尾，然后把原来的空间释放回空闲列表中并且进行结合。stop-and-copy 算法简单、存储消耗低且不影响运行时的总体性能，但是存在一定的缺陷：它没有利用数据类型信息，不能够区分某对象域是基本域(例如 Integer)还是对某一有效对象的引用，因此它要遍历全部对象空间；此外还要考虑对象域的跨界组合问题，如果某一整数域的值恰巧为一个合法对象地址，即便该对象是无用对象也不能够回收。在 Java 虚拟机实现垃圾回收操作的过程中，通过一个低优先级的进程定期对内存进行搜索，而搜索的目标就是那些"游离"的对象。所谓游离对象就是指那些程序执行过程中不会再被引用的对象。在 Java 程序运行过程中存在一个或多个活动线程，而每个活动线程均与多个对象相关，而"游离对象"就是不被活动线程相关对象引用的对象。例如：

```
public static void main(String[] args) {
        TextObject text = TextObject("Text1");
        text.getProperties();
}
//TextObject 类的内部方法
public double getProperties() {
        PropertyUtility utility = new PropertyUtility();
        double value = utility.computeObjectSize(this);
        return value;
}
public void finalize() {
}
```

当程序调用方法 getProperties()并得到返回值之后，对于 utility 实例来说不再存在任何引用关系。此时这个对象就是一个游离对象，并符合垃圾回收的条件。如果程序中特定类中实现了 finalize()方法，那么在系统进行内存回收操作之前将调用此方法；这样您就可以很容易地发现对象实例被进行垃圾回收的时间。但这并不意味着 utility 对象能够立刻被系统回收。在上面所分析的一些情况下，此对象可能不会被回收。

不同的 Java 虚拟机使用了不同的算法来控制垃圾回收操作。而就标准的 Sun 虚拟机来说，从1.1.x 到2版并没有对垃圾回收机制从总体上有什么改进。

Java 虚拟机提供了对象实例的垃圾回收功能，而类也存在垃圾回收和卸载的操作。在前面的实例中存在类 TextObject 及其对象实例 text。对于类 TextObject 来说，它也是一个堆分配对象，每个类加载器只有唯一一个 TextObject 类实例，当程序第一次实例化 TextObject 对象时，Java 虚拟机必须通过使用类加载器来找到这个类。你可以用 Java 来编写一个自己的类加载器，而在 Java 虚拟机中有一个内置的类加载器，我们称这个加载器为原类加载器。类加载器的工作方式解释起来相对比较复杂，本文不对此技术进行过多的解释。重要的是要理解类实例在垃圾回收规则方面不同于常规的实例。这些规则包括：

·如果原类加载器加载了类实例，那么在程序运行过程中这些实例将常驻于内存中，这些实例不符合垃圾回收条件；

·如果加载一些类的类加载器不再被引用，那么通过用户定制的类加载器加载的类符合垃圾回收和卸载条件。

这些特定规则之所以成立的主要理由是类实例可以具有静态变量和本地方法，在程序生存期中它们均需维持自身的状态，因为虚拟机不能够判断它们是否不会再被使用。

3.2 “唯一模式”的运用

Java 虚拟机针对静态变量内存分配的基本规则是一旦通过数据(通常是对象)进行初始化，那么这个变量将占用一定的内存空间，而占用时间与定义它的类占用内存空间的时间长短一致。由于 Java 虚拟机的类加载器加载的大部分类将在程序生存期内一直占用内存，因此这些类中的静态变量也就由此会在程序生存期内占用内存。

静态变量的流行用法是实现“唯一”模式。“唯一”模式是一种面向对象的设计模式，这种模式用于确保一个类只有一个实例，并为这个实例提供一个全局入口指针。例如，如果我们希望构造一个用于生成 TextObject 对象的 ObjectFactory 类，那么我们可以使用“唯一”模式。我们在 ObjectFactory 类定义中生成一个静态变量，这个变量保存本身的一个实例。之后我们可以定义一个静态方法 getObjectFactory()，它可以被用于获得 objectFactory 实例，提供对其进行访问的全局指针。我们可以通过调用 getObject(String name)方法来从 ObjectFactory 实例中获得具备特定对象名的对象。我们从性能角度来分析一下，ObjectFactory 提供的向量可以对对象进行缓存以便复用。如果所需的对象没有在 ObjectFactory 对象中找到，则系统会使用一些资源来生成一个对象实例并将其加入缓存。

这种做法对于内存会造成什么影响？我们有一个引用 ObjectFactory 实例对象的原类加载器。ObjectFactory 类对象具有一个静态变量，这个变量引用了 ObjectFactory 的实例。最终，ObjectFactory 的实例引用了一个包含多个 Object 对象的向量。由于这个对象实例作用链中的第一个元素，即原类加载器不符合垃圾回收条件，因此整个链中的对象实例都不能够被进行垃圾回收处理。

“唯一”模式在实际应用中的优劣取决于具体的设计要求。如果这些对象需要保存在内存中以便能够快速地得到复用，则这种方式是好的；对于这种设计需求来说，程序运行速度是首要的。这种想法的不足之处是通过两方面暴露出来的。首先，内存是宝贵的资源，它的容量有限，因此那些不会被复用或很少被复用的对象会消耗空闲的内存空间；其次，开发人员不经意地通过 ObjectFactory 引用了一些对象，而这些被引用的对象按照常规是应当被回收的，但是由于 ObjectFactory 是常驻于内存的，因此这些本不应常驻内存的变量也无法被进行有效的回收了，这样就造成了内存漏洞。这种情况被称为多余或未知的引用。

3.3 解决未知或多余对象引用造成的内存漏洞

我们对垃圾回收的一般认识是实现在某一对象不再被引用时，通过程序从内存中删除对象的操作。然而，有很多时候我们认为已经对一个对象停止了引用，但是实际上此对象仍然存在不为我们所感知的引用关系。让我们通过下面的实例对这个问题进行解释。

```
public static void main (String[]args) {
        TextObject text = new TextObject("Object1");
        text.getProperties();
        //下面这行代码使得 text 对象不会被收集
        SecurityManager.getSecurityManager ( ) .addToNotifyList
          (text);
        // 撤销对 text 的引用
        text=null;
        TextObject text = new TextObject("Text2");
        text.getProperties();
}
public static ObjectManager getObjectManager() {
        if (objectManager==null)
              objectManager = new ObjectManager();
        return objectManager;
}
public void addToNotifyList(TextObject object) {
        getObjectManager().notifyList.addElement(object);
}
```

程序中存在一个 ObjectManager 对象，它是作为一个单一对象来实现的。ObjectManager 对象需要在对象处理策略发生变化时立刻通知所有的 TextObject 对象，因此它保存了对所有 TextObject 对象的引用。在这个实例中，我们将 text 变量设置为 null，希望垃圾收集功能能够将此对象收集起来。然而，因为 ObjectManager 对象中通过向量对所有 TextObject 对象进行了保存，所以 text 对象并不能够被回收。ObjectManager 类因为是一个单一对象，这个类通过一个静态变量引用 ObjectManager 对象，因此不符合垃圾回收的条件。

既然我们了解了未知或多余对象引用现象发生的原因，那么我们就可以通过避免由静态变量引用对象、在处理完对象之后清除引用关系或者使用弱引用等方式解决这个问题。

对于上面的实例而言，我们可以在 TextObject 对象中编写一个 cleanUp()方法，通过这个方法来通过外部调用的方式清除静态变量对该对象自身的引用，它从 ObjectManager 对象的列表中删除此对象。这种解决方案并不理想，因为开发人员需要在完成对一个对象的处理之时必须记住调用 cleanUp()方法。这种外部调用的方式并没有从根本上解决未知引用的问题。我们需要的是一种引用对象实例的方式，且不妨碍它被垃圾回收。JDK 1.2为我们提供了一种便捷的方式来通过弱引用的方式解决这个问题[5]。弱引用是一种典型的对象引用方式，它以一种不阻止垃圾回收的方式存在。垃圾回

收器在进行回收操作时首先考察对象，只有在弱引用的状态下，此对象才能够被从内存中清除。在 JDK 1.2中，弱引用通过几个包含在 java.lang.ref 包中的类实现。这是一种最佳的方式。

3.4 清除或释放本地系统资源操作

对于清理无用的对象实例来说，无用单元回收器能够较出色地完成工作。然而，Java 虚拟机在分配内存方面还有其他的方式，这些操作主要是针对除 Java 实例以外的对象，特别是以本地系统资源形式出现的对象。本地系统资源是指那些通过 Java 以外的函数分配的资源。这些分配操作通常是通过 Java 本地接口 JNI 来完成的，通常是以 C 或 C++ 语言实现的。一个简单且常见的实例是抽象窗口工具集 AWT 资源的使用。Frame、Dialog 和 Graphics 等类如果不再使用时，应当调用方法 dispose()来释放它们所占用的系统资源。

当开发人员使用 JNI 编写自己的本地方法或者访问第三方提供的方法时，情况更加复杂。在这些实例中，本地方法可能需要通过 Java 来实现强行的清理操作。如果没有进行这些调用，则会造成运行时的内存漏洞。

当处理一些可视化的本地资源时，通常会通过对窗体进行高速缓存的方式来提高性能和避免内存漏洞发生的可能。通过对每个窗体进行缓存和复用，您不必担心窗体及其相关的系统资源，因为目前只存在有限数量的窗体。然而，这种高缓/复用方式如果在程序具有40个以上不同的窗体时，则很难完成。在这些情况下，需要调用 dispose()或者混合使用多种缓存方案来实现回收。当使用 Frame 或 Window 时，可以通过关闭窗口来实现清除操作，因为在事件处理方法中调用了 dispose()方法。

另外一种方式可以被用于其他的系统资源，那就是在 Java 类的 finalize()方法中加入清除内存的代码。这些方法非常有用，但是时间的选择是重要的，我们很难预测该方法被调用的时间。正如前面所提，对于虚拟机来说并没有将回收未使用的引用回收，因此在方法 finalize()被调用之前可能会经过一段的等待时间，而等待时间的长度并不定。

一种降低内存负载的最简单也是最有效的方法就是在实例不再被使用时，将对象引用设置为 null。如果实例占用的空间很大，那么这种处理方法的效果就非常明显。一旦对对象实例的引用被删除，那么垃圾回收器回收它被占用的内存。但是在实用过程中，开发人员经常忘记了对对象实例的置空操作。例如，在一个数组中保存了一系列对象实例，同时使用一个整型变量 count 来计算实际的实例数目。我们删除了一个对象时，程序只是将 count 减一，而并没有将这个对象设置为 null 对象。这样系统就无法进行正确的垃圾回收。

许多标准的存储类提供了一些方法，通过这些方法可以将程序实用的内存数量降低到最小。例如，Vector 类可以存储数量可变的对象。在缺省情况下，每次超过容量时，Vector 就对空间进行加倍处理。很明显，这就会造成内存的浪费。控制这种问题有两种方法。首先，当你完成将元素加入 Vector 对象之后，调用 trimToSize()方法，这样就可以使 Vector 对象所占空间最小。另外一种方法是在构造函数中设置 Vector 对象最初的容量和每次加入新元素后的增量。这两种方法同样适用于 Hashtable 类。

四、监测内存使用状况

针对 Java 程序进行内存空间的优化处理之前，必须对被优化的目标进行有效分析。开发人员只有通过内存测试过程发现程序中哪部分代码需要进行优化，才能够针对实际情况选择相应的优化策略。有多种方式可以用于发现 Java 程序中内存泄漏现象。

第一种，也是最简单的方法就是使用一个操作系统进程监视器，它可以提供一个正在运行的进程所使用的内存数量。我们也可以使用 Java Runtime 类中提供的 totalMemory()和 freeMemory()等方法来得到虚拟机所控制的连续内存空间的容量以及在特定时刻未使用的内存容量，通过将两个方法捆绑在一起使用可以计算出当前运行的 Java 程序所使用的内存量。大多数商业用的 Java 集成开发环境并没有提供虚拟机级的控制，因此我们通常可以通过 JDK 来完成对内存使用状况的测试。最后，我们还可以使用一些内存优化工具，例如 OptimizeIt、JProbe 或 JInsight。这些工具可以帮助您诊断正在运行的程序，提供诸如实例个数在内的有用信息。

参考文献

1 Java Language Specification. 1.2. Sun Microsystems, 1999

2 Venners B. Inside the Java Virtual Machine. McGraw-Hill, 1997

3 丁宇新，程虎. Java 虚拟机中无用单元的精确回收. 计算机学报，1999.11

4 Agesen O. Finding Reference in Java Stacks. Proc of the OOPSLA'97, 1997

5 Jewell T. Greater Garbage Collector Access with Reference Objects. Java Report, 1999.6

（上接第44页）

5 Gossard D, Zuffante R, Sakurai H. representing dimensions, tolerances and features in MCAE systems. IEEEE Computer Graphics and Applications, 1988, 8(8)

6 Kramer G A. A geometric constraint engine. Artificial Intelligence, 1992, 58: 327～360

7 陈立平，周济，等. 几何约束系统推理研究. 华中理工大学学报，1995, 6:70～74

8 Hoffman C. Geometric Constraint Solving in R^2 and R^3, in Computing in Euclidean Geometry, eds D. Z and F. Huang, World Scientific, 1995. 266～298

9 Gao Xiao-Shan, Chou Shang-Ching. Solving Geometric Constraint Systems. I. A Global Propagation Approach, CAD, 1998, 30(1): 47～54

10 董金祥，等. 变参绘图系统中约束求解的新思路. 计算机辅助设计与图形学学报，1997, 9(6):513～519

11 Lee J Y, Kim K. A 2-D geometric constraint solver using DOF-based graph reduction. Computer-Aided Design, 1998, 30(11): 883～896

12 Ge Jian-Xin, Chou Shang-Ching, Gao Xiao-Shan. Geometric constraint satisfaction using optimization methods. CAD, 1999, 31: 867～879

13 Chen Liping, Peng Xiaobo. An Approach to A 2D/3D Geometric Constraint Solver. In: Proc. of ASME DETC'2000

14 Hoffmann C, Lomonosov A, Sitharam M. Finding Solvable Subsets of Constraint Graphs, Third International Conference on Principles and Practice of Constraint Programming, No. 1330, Lecture Notes in Computer Science, Springer Verlag, Schloss Hagenberg, Linz, Austria, October 29 - November 1, 1997

15 Hoffmann C, Lomonosov A, Sitharam M. Geometric constraint decomposition. Geometric Constraint Solving and Applications, Springer, Berlin, 1998. 171～195