**1 Introduction and Overview**   
 Java HotSpot™虚拟机是Sun公司针对Java平台实现的高性能虚拟机。 Java HotSpot为Java   
SE平台提供了基础，为快速开发和部署商业关键应用提供了首要的解决方案。Java SE技术  
在 Solaris、Linux、Windows和其他经过 Java技术授权的平台上均可使用。

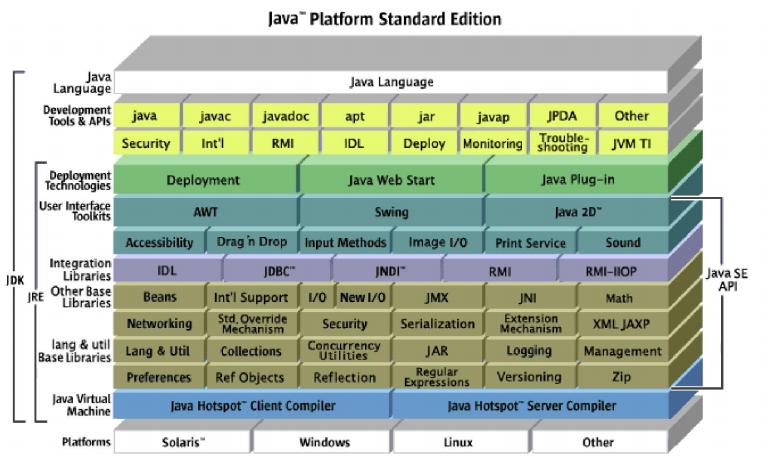


图 1 Java 标准版技术   
   
 Java平台已经成为了软件开发和部署的主要选择。有了成千上万的用户和开发者的支持，  
Java平台在很多方面都得到了迅速的发展：从信用卡到无线设备，从桌面应用到大型机设备  
等。在Web 方面，Java平台也称为了部署 Web 页面Applet、Web Service、大型商业应用等  
场景的基础。   
 Java HotSpot 虚拟机建立在Java的跨平台技术支持和健壮的安全模型的支持上，拥有良  
好的伸缩性、执行质量和执行性能。此外，Java HotSpot 虚拟机一直都是向后兼容的。   
 Java HotSpot 虚拟机几乎支持开发、部署和管理应用程序中的各个方面，在以下方面得  
到应用：   
λ IDE，包括 Java Studio Tools、NetBeans、Eclipse、IntelliJ IDEA等；   
λ 应用服务器供应商，包括 WebLogic、WebSphere、Tomcat 等。   
   
Sun公司也通过制定各种工业标准，开发内部标准来不断的提升执行性能。这些改进对  
客户端和服务器端的Java虚拟机环境都是适用的。   
   
Java标准版平台包含两种Java虚拟机的实现：   
λ Java HotSpot 客户端虚拟机：专为加速客户端应用程序启动，并减少内存占用而进行优  
化；   
λ Java HotSpot 服务器端虚拟机：运行在服务器环境中，专为最大化程序执行速度而优化。

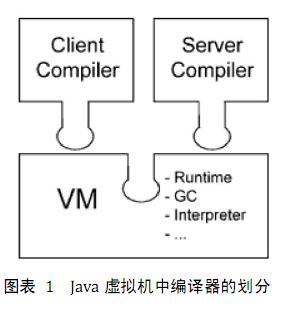
**1.1 Java HotSpot虚拟机——建立在坚实的基础上**   
Java HotSpot 虚拟机建立在一个坚实的基础之上，拥有许多特性。该架构支持动态的、  
面向对象的优化，具有世界级的执行性能。即使在需要进行大量计算的系统中，仍可通过高  
伸缩性来支持多线程运行。在执行企业级快速开发、内省和管理中，具有较高的可靠性、可  
用性和服务性。

**2  Java HotSpot虚拟机架构**

**2.1 概述**   
 Java HotSpot 虚拟机是 Sun公司为 Java平台实现的虚拟机，使用了很多技术来提升程序  
的执行性能，包括先进的内存模型、垃圾收集器和自适应的优化器等。这些都是以高级的、  
面向对象的形式实现的：   
λ 同一对象模型；   
λ 在同一个栈中混用解释帧、编译帧和本地帧；   
λ 基于本地线程的可抢占式多线程实现；   
λ 精确的分代压缩垃圾收集；   
λ 快速的线程同步机制；   
λ 动态的去优化和激进的编译器优化并存；   
λ 在虚拟机启动时产生系统指定的运行时例程；   
λ 支持并行编译的编译器接口；   
λ 运行时统计强制编译器只对“热点”方法进行编译。

JDK 中包含了两种Java虚拟机，分别是客户端虚拟机和专为服务器应用程序调优的服务  
器端虚拟机。这两种方案共享了同一个Java HotSpot虚拟机环境代码，但使用了适合于不同  
性能特点的编译器。这些区别包括编译器内联策略和堆的默认使用策略。   
在 JDK 的发行版中，包含了上述的两种虚拟机。程序员可以通过在命令行中指定“-client”  
或“-server”来启动不同的虚拟机。   
尽管服务器端虚拟机和客户端虚拟机是相似的，但服务器端虚拟机为提升最大执行速度  
做了调优。它适合于运行那些需要长时间运行、执行速度比启动速度或内存占用更重要的应  
用程序。   
客户端虚拟机的编译器作为 Classic 虚拟机和 JIT（Just-In-Time）编译器的升级版本，应  
用于各种版本的JDK 中。客户端虚拟机加快了应用程序和小程序（applet）的运行时性能。  
Java HotSpot 客户端虚拟机专门为加快启动速度和减少内存占用做了优化，使其更适用于客  
户端环境。一般来说，客户端虚拟机更适合于GUI应用程序。   
客户端虚拟机编译器并不会执行一些服务器端虚拟机所要作的优化操作，但这样可以用  
更少的时间来分析并编译一段代码。因此，客户端虚拟机有更快的启动速度和更少的内存占  
用。

服务器端虚拟机中有一个高级自适应编译器，可以支持很多类型的优化，例如调用虚方  
法时的激进的内联优化。相对于静态编译器，这具有较强的竞争力和性能优势，自适应优化  
技术在实现上非常灵活，优于静态分析和编译技术。



**2.2   内存模型**

2.2.1 无句柄对象   
 在之前的 Java 虚拟机版本中，例如 Classic 版本的虚拟机中，使用间接句柄来表示对象  
的引用。在执行垃圾回收时，这使得重定位对象比较容易，但这也是一个主要的性能瓶颈，  
因为Java编程语言访问实例变量需要两级转向。   
 在Java HotSpot 虚拟机中，Java代码没有使用句柄。对象引用是用直接指针实现的，这  
样访问实例变量的时候可以达到C语言一样的速度。在内存回收过程中，重定位对象时，垃  
圾收集器负责找到并更新所有对象的引用。

Java HotSpot 虚拟机使用了两个机器字长的对象头，而Classic 虚拟机则使用了3个字长  
的对象头。因为 Java 对象的平均大小比较小，这对内存空间的消耗具有重要影响，可节约  
大约8%的堆大小的内存。对象头的第1个字包含了诸如标识哈希码和GC状态等信息。第2  
个字是一个指向对象的类的引用。只有数组类型的对象有第3个字，用于存储数组的长度。

2.2.4 反射数据也是对象   
 类、方法和其他内部反射数据都是堆上的对象（尽管这些对象并不能由基于 Java 技术  
的程序来访问）。这不仅简化了虚拟机的内部对象模型，也使得这些类可以被其他Java对象  
所使用的同一个垃圾收集器回收掉。

2.2.5 本地线程支持，包括抢占式和多线程处理   
 每个线程的活动栈都会使用主机操作系统的栈和线程模型。 Java编程语言方法和本地方  
法共享同一个栈，实现了 C 语言和 Java 语言之间的快速调用。全抢占式 Java 线程使用了主  
机操作系统的线程调度机制。   
 使用本地操作系统线程及调度机制的优点是可以透明了利用本地操作系统的多线程处  
理支持。由于Java HotSpot 虚拟机被设计成对在执行Java代码时因抢占和/或多线程处理造  
成的竞争条件不敏感，所以 Java 线程会自动调整本地操作系统提供的调度和处理器分配策  
略。

**2.3 垃圾收集**   
 Java HotSpot 虚拟机系统中的分代管理使得针对不同类型应用程序选择不同的垃圾收集  
算法更具灵活性。Java HotSpot 虚拟机支持不同多种不同的垃圾收集算法来满足对不同的暂  
定时间和吞吐量的要求。

程序员们喜爱 Java 编程语言的一个主要原因是这是第 1 个内建支持自动内存管理，或  
称垃圾收集的编程语言。在传统的程序设计语言中，动态内存分配是显式的使用 allocate/free  
来完成的。在实际应用中，这经常导致内存溢出、程序 bug 甚至崩溃，往往也成为系统的瓶  
颈所在，难以实现模块化和代码重用。在 Java 程序设计语言中，垃圾回收是支持安全执行  
的一个重要组成部分，有力的支持的安全模型。   
 垃圾收集器会自动释放无用对象占用的内存，只有到程序再也不会用到这个对象时，才  
会被释放掉。这种机制解决了由自由指针引起的内存泄漏、程序崩溃，已经其他一些难以发  
现的 bug。   
 从传统上讲，相比于显式释放内存，垃圾收集被认为没有效率，并且会降低系统性能。  
事实上，现代垃圾收集技术已经取得长足的进步，系统的整体性能甚至略好于那些显式释放  
内存的系统。

2.3.2 Java HotSpot垃圾收集器   
 除了下面将要描述的一些特性外，内存系统的设计也是非常干净的、面向对象的，很容  
易使用新的垃圾回收算法来测量、实验和扩展。   
 Java HotSpot 垃圾收集器的主要特点将在下面叙述。总体上讲，这些特性既适合于那些  
对高性能有要求的应用程序，也可应用于那些需要长时间运行的、不允许出现内存溢出或较  
多内存碎片的应用程序。

2.3.2.1 精确性   
 Java HotSpot 垃圾收集器是一个非常精确的垃圾收集器。相比之下，其他的很多垃圾收  
集器都是保守的，或半精确的。有时，保守的垃圾收集器是比较有吸引力的，因为它可以很  
容易的被添加到一个没有垃圾收集器的系统中，但也有一些缺陷。保守的垃圾收集器可能会  
引发内存泄漏，使对象无法移动，也可能造成堆碎片。   
 保守的垃圾收集器无法准确定位所有的对象引用。所以，它会假设指向对象的内存字实  
际上都是对象引用。这样的话就说产生一些错误，例如将整形数字当成对象指针对待。看起  
来像指针的内存单元，被当作指针使用，垃圾收集器就无法做到准确收集。这样的做法还有  
一些其他的负面影响。首先，当发生这些错误时（实际上并不会频繁发生），可能会发生内  
存泄漏，而程序员几乎无法调试。其次，由于本身可能发生错误，保守的垃圾收集器可能会  
使用句柄来间接的引用对象（这样会降低性能），或者避免回收对象，因为回收无句柄对象  
要更新所有对象的句柄。无法回收对象会导致内存碎片的产生，更重要的是，无法使用将要  
在下面描述的高级的分代拷贝垃圾收集算法。   
 Java HotSpot 垃圾收集器是精确的，有一些保守垃圾收集器所没有的特性：   
（1） 所有不可达的对象都会被可靠的回收；   
（2） 所有对象都是可回收的，允许对象内存拷贝，这回减少内存碎片，增加内存紧  
凑型。   
精确的垃圾收集机制可以避免偶然的内存泄漏，移动对象可以防止对整个堆进行压缩  
Java HotSpot 虚拟机的垃圾收集机制适用于非常大的堆。

2.3.2.2 分代拷贝垃圾收集   
 Java HotSpot 虚拟机使用了业界领先的分代拷贝垃圾收集器，有以下两大主要特性：   
（1） 相比于非分代垃圾收集器，提升了垃圾回收速度和垃圾收集效率；   
（2） 相对减少了垃圾收集频率和因执行垃圾收集而造成的程序暂停的时间。   
分代垃圾收集器基于这样一个事实，在大多数程序中，大量对象（95%以上）的生命周  
期非常短（例如，只是临时存储数据使用）。通过将新近创建的对象隔离到一块专门的区域，  
分代垃圾收集器就可以做很多事了。首先，因为新创建的对象都被存放到一块类似于栈的空  
间中，分配速度是非常快的，因为只需要更新一个指针，检查一次这块区域是否有溢出。其  
次，当这块区域溢出后，区域的大多数对象都已经死了，垃圾收集器只需要简单的将或者的对象移到其他地方，就可以将全部区域回收，避免了对区域中已死对象的回收。

对象移到其他地方，就可以将全部区域回收，避免了对区域中已死对象的回收。

2.3.2.4 标记压缩老年对象垃圾收集器   
 尽管分代拷贝垃圾收集器能有效的回收已死亡对象，但那些存活时间较长的对象仍然会  
在 老年对象内存区不断积累。偶然情况下，由于内存较低或一些程序上的要求，必须使用  
老年对象垃圾收集。Java HotSpot 虚拟机默认使用了标准标记-压缩垃圾收集算法，该算法会  
从根对象开始遍历所有活动对象，然后执行清理，并压缩清理已死亡对象后留下的内存空隙。  
通过压缩堆中的空隙，而不是将它存入到一个 freelist 中，可以消除内存碎片，并可以实现  
老年对象分配的线性化，而无需再查找 freelist。

2.3.2.5 CMS 收集器   
 对于那些要求较大堆空间，默认的老年代标记清理垃圾收集器会引起应用程序线程的暂  
停。Java HotSpot 虚拟机实现了一个可选的并发垃圾收集器，可充分利用空闲处理器来对老  
年对象进行垃圾回收，并且只会使应用程序有很短的暂停。这是通过在应用程序执行过程中  
不断的跟踪和清理来完成的。在某些应用场景中，应用程序吞吐量的峰值会有一些下降，因

为处理器会并发执行垃圾回收。但是，平均和最差情况下垃圾收集暂停时间都可以通过 1  
个或 2个重要的命令来减少，当在一个大堆上执行默认的同步标记清理算法时，允许应用程  
序更加平稳的响应，而不会产生突发性响应的问题，

2.3.2.6 并行的老年代垃圾收集器   
 当前版本的Java HotSpot 虚拟机为老年代引入了并行标记清理垃圾收集器，专用于提升  
需要较大堆的应用程序的伸缩性。CMS 垃圾收集器专注于减少应用程序的暂停时间，老年  
代的并行垃圾收集器则通过在stop-the-world暂停期间，使用多线程同时执行垃圾收集来提  
高应用程序的吞吐量。并行老年代垃圾收集在内部使用了新的技术的数据结构，使其不仅保  
留了在垃圾收集期间的精确性和最小化开销的优点，还达到了较好的系统伸缩性。

2.4 超快的线程同步   
 Java编程语言支持应用程序的多线程执行，并在语言级提供了现场同步的机制，可以快  
速的建立具有较细粒度的锁的多线程应用程序。在此之前的同步实现，如 Classic 虚拟机中，  
相比于 Java编程语言中其他的宏观操作性能低下，使用细粒度的锁会成为性能的瓶颈。   
 Java HotSpot 虚拟机整合了竞争和非竞争的同步操作，这是提升同步性能的一大因素。  
非竞争的同步操作，包含了大量的同步操作，使用了超快的（ultra-fast）、常量时间的  
（constant-time）技术来实现。基于最新的优化技术，在最佳情况下，即使是在多处理器计  
算场景中，也可以无损耗的运行。竞争性同步操作，充分利用了自旋锁的优点，即使应用程  
序中有大量的竞争锁，仍可提升吞吐量。因此，大于大部分应用程序来说，同步性能已不再  
是关键的性能指标。

2.5 64位架构   
 Java HotSpot 虚拟机的早期版本会受到 4G内存空间的限制，即使是在如Solaris OE这样  
的 64 位操作系统上。现在，很多桌面系统都已经达到了 4G 内存的使用环境，而现代服务  
器都已经拥有了更大的内存。例如，Sun Fire E25K服务器的每个域都支持 1.15TB 的内存。  
使用 64位的 Java虚拟机，可以利用系统的整个内存空间。   
 基于 64 位地址空间，应用程序可以有很多种用途。例如，那些会在内存中保存大量数  
据集的应用程序。应用程序可以避免从磁盘中获取分页数据或从 RDBMS 中获取数据的性能  
损耗。这样，应用程序的性能得以大幅提升。   
 现在，64 位的 Java HotSpot 虚拟机是可以安全使用的。Server 模式下的虚拟机支持 32  
位和 64位两种操作。你可以使用命令行选项“-d32”或“-d64”来明确指定使用哪种模式。  
而 JNI的用户若想在64位虚拟机上运行程序，则需要重新编译代码。

2.6 对象包装   
 对象包装功能可以最小化不同数据类型大小之间的空间浪费。这是 64 位虚拟机的主要  
特性，即使在32位虚拟机上，对系统性能也有所提升。   
例如：   
public class Button {   
 char shape;   
 String label;   
 int xposition;   
 int yposition;   
 char color;   
 int joe;   
 object mike;   
 char armed;   
}

上面的代码中，在color和 joe 之间（浪费3个字节），以及 joe 和mike 之间（在64位  
系统上会浪费4个字节）会产生空间浪费。   
 现在，这些字段会被重新排序，如下所示：   
...   
 object mike;   
 int joe;   
 char color;   
 char armed;   
...   
   
 这样么就不会有内存空间的浪费了。

**3 Java HotSpot编译器**   
3.1 概述   
 大多数试图加速 Java编程语言的努力都集中在对为传入语言开发的编译器上。JIT 编译  
器本质上是一个快速的传统编译器，在 Java 程序运行过程中，将 Java 字节码编译成本地机  
器代码。运行在用户机上上的 JIT编译器实际上会执行字节码，并在第一次执行的时候编译  
每个方法。   
 但是，对 JIT 编译器还有几个问题要说。首先，因为编译器运行在执行代码的机器上，  
并且占用用户时间，所以它受 CPU 速度的影响很大。如果 CPU 的速度不快的话，在启动应  
用程序或应用程序的一部分时，用户会明显的感觉到延迟。这就会带来一种权衡，使得更难

以执行高级优化，因为高级优化通常会使编译性能大大降低。   
 其次，JIT 有足够的时间来执行全面的优化，相比于传统的编程语言，如 C/C++，Java  
编程语言的JIT执行一些的优化也不是那么有效率，原因如下：   
λ Java语言是动态安全的，即它保证了程序不会违反语义或直接访问位结构化的内存，  
这就要求必须经常执行动态类型检查（当类型扩展，或转换为 Object 时发生）。   
λ Java 语言中，所有的对象都是在堆中分配的。相比之下，C++中的很多对象是在栈  
中分配的。因此，Java 语言中需要更快的对象分配速度。此外，相比于 C++，Java  
中有着完全不同的内存分配开销（包括潜在的擦除和白色屏障开销）。   
λ 在 Java 语言中，大多数的方法调用都是虚方法调用（潜在的多态），这比 C++中使  
用的频率多很多。其中的含义不仅仅是方法调用的性能成了程序性能的主要因素，  
更说明了静态编译器优化（尤其是一些全局优化，如内联）更加难以对方法调用进  
行优化。许多传统的编译器优化对函数之间调用最为有效，减少函数调用间的距离，  
但在 Java语言中，这些优化就不那么有效了，因为Java中方法的代码段都比较小。   
λ 由于使用了动态类载入技术，基于 Java 技术的程序可以在运行时修改。这样执行  
全局优化就更加困难。借助于动态载入，编译器不仅仅可以检测到这些优化选项什  
么时候变为无效，还可以在程序运行期间取消或重做某些优化，即使它们涉及到栈  
上的活动方法。

结果是，任何想要在Java语言中实现基础优化的想法都必须对这些非传统的进行解答，  
而不是在传统编译器技术上提供 2进制兼容性。   
 Java虚拟机结构使用了自适应的优化技术来解决上述问题。

3.2 热点监测   
 自适应优化利用了一个有趣的程序属性来解决JIT编译的问题。几乎所有的程序都会花  
费大量的时间来执行很少的一段代码。因此，不再是一个一个方法进行编译，Java虚拟机会  
使用解释器立即执行程序，并在程序运行时监测关键热点。然后，它会让全局本地代码优化  
器（global native-code optimizer）对热点进行优化。这样，避免对一些不经常使用的方法进  
行编译（程序中的大部分方法都是这样的），Java HotSpot编译器可以将更多的精力放在影响  
性能的关键点上，而不会增加总体的编译时间。热点监测与程序运行同时执行，并收集一些  
其他信息，例如为虚方法调用手机调用者与被调用者的关系等。   
 使用这种方法有一个微妙但很重要的好处，通过将编译延后到程序运行了一段时间之后  
（这里的时间指的是机器时间，不是用户时间），可以对代码的执行信息进行收集，使用一  
些更好的优化方法。同时也可以降低内存消耗。此外，对程序运行中的热点收集信息的同时，  
也可以收集一些其他信息，如与在虚方法调用时的调用者与被调用者的关系相关的数据。

3.3 方法内联   
 Java 编程语言中对虚方法的频繁调用是一个很重要的优化瓶颈。在程序运行时，如果  
Java HotSpot 自适应优化器收集到了有关热点程序的信息，它不仅会将热点编译成本地代码，

还会在热点代码上执行有着不小性能损耗的方法内联。   
 内联的作用很大。它可以极大的方法调用的动态频率，这可以减少方法调用的时间。但  
更重要的是，方法内联会产生一些较大的代码块，方便优化器进行优化。这可以显著的提高  
传统编译优化的效率，克服了提升 Java编程语言性能的主要障碍。   
 内联与代码优化是协同作用的，因为内联使优化更有效率，随着Java HotSpot 编译器逐  
渐成熟，操作较大的、内联的代码块的能力会在将来打开通向更高级性能优化的大门。

3.4 动态去优化   
 尽管内联是一项重要的优化手段，但对于像 Java 这样的动态面向对象语言来说，执行  
起来很困难。此外，由于探测热点并对热点区域的方法调用进行内联操作也有相当的难度，  
所以也无法有效的提供 Java 语言的全部语义。这是因为 Java 语言不仅可以在运行时改变方  
法调用的模式，还可以动态载入 Java代码到程序中。   
 内联是一全局分析为基础的。动态载入使得内联的复杂度大大增加，因为它改变了程序  
中的全局关系。一个新的Java类可能包含了需要内联到适当位置的方法。因此， Java HotSpot  
虚拟机必须可以动态去优化（必要的话，还需要重优化）之前已经优化过的热点代码，甚至  
是正在热点代码。如果没有这种能力，一般意义上的内联操作就无法应用于基于 Java 技术  
的应用程序了。   
 Java HotSpot 虚拟机的客户端模式编译器和服务器模式编译器都提供了对动态去优化的  
完整支持。这样就可以使用一些激进乐观的优化方法和其他一些技术，例如全速调式  
（full-speed debugging）。

3.5 Java HotSpot客户端编译器   
 Java HotSpot 客户端编译器是一个简单、快速的三段式编译器。在第一阶段，一个平台  
无关的前端会从字节码中构建出高级中间表示（high-level intermediate representation， HIR）。  
HIR 使用静态单一分配（static single assignment，SSA）来表示值，这样就可以更容易使用某  
些优化策略，这些优化可以在 IR 创建中，或创建后执行。在第二个阶段，制定平台会从HIR  
中生成低级中间表示（low-level intermediate representation，LIR）。最后一个阶段中，会使  
用自定义的线性扫描算法在 LIR 上注册分配，在 LIR 上执行窥孔优化，并从 LIR 中生成机器  
码。   
 重点在于尽可能的从字节码中提取出更过的信息。客户端编译器着重于本地代码的质量，  
只做很少的全局优化，因为那些通常会需要较长的编译时间，开销较大。

3.6 Java HotSpot服务器端编译器   
 服务器端编译器用于典型的服务器端应用程序。Java HotSpot 服务器端编译器是一个高  
端全功能优化编译器。它使用了高级的基于 SSA 的 IR 优化策略。优化器会执行所有的经典

优化，包括剔除死代码（dead code elimination），提升循环不变量（loop invariant hoisting），  
剔除通用子表达式（common subexpression elimination），常量传播（constant propagation），  
全局数值计算（global value numbering）和全局代码移动（global code motion）。此外，还包  
括一些针对 Java技术的优化，例如Null检查和范围检查剔除，以及对异常抛出路径的优化。  
寄存器分配器是一个全局图像着色分配器，充分利用了常用于 RISC 微处理器中的寄存器集  
合。编译器具有较高的可移植性，依赖于描述了目标机器硬件各个方面的机器描述文件。虽  
然编译器比标准的JIT编译器慢一些，但仍然比传统的优化编译器快，通过降低编译代码的  
次数减少了提升代码质量所需的编译时间。

3.7 编译器优化   
 Java HotSpot 编译器可以使用很多高级优化来提升体传统面向过程语言和面向对象语言  
的执行性能。其中一些优化包括：   
λ 深度内联（deep inlining）和潜在虚调用内联（inlining of potentially virtual calls）：  
正如前文所述，客户端编译器和服务器端编译器都会使用方法内联、全局分析和动  
态去优化来启用深度内联，因此可以减少方法调用的性能损耗。   
λ 快速类型检查（Fast instanceof/checkcast）：Java编译语言为了类型安全的需要，会  
频繁的进行类型检查， Java HotSpot 虚拟机使用一项新技术来加速类型检查的过程。  
这可以减少面向对象语言的运行时消耗。   
λ 范围检查剔除（Range check elimination）：Java语言规范要求在访问数组时进行边  
界检查。如果编译器可以保证访问数组时所使用索引肯定不会超越边界，那么就可  
以将边界检查剔除。   
λ 循环展开（Loop unrolling）：以服务器模式运行的虚拟机对此提供了支持。循环展  
开是一种标准编译器优化，它可以加快循环的执行速度。循环展开在增加了循环体  
大小的同时，减少了迭代次数。循环展开也可以增加其他优化的效率。   
λ 直接反馈优化（Feedback-directed optimizations）：以服务器模式运行的虚拟机在将  
Java 字节码编译为本地代码前会对解释器中的程序进行一些比较耗费性能的监测  
操作。这些监测数据提供了与正在使用的数据类型、热点代码和其他一些属性相关  
的更多的信息。在特定情况下，编译器可以使用这些信息对代码执行一些更激进、  
更乐观的优化操作。如果代码在运行过程中违反了假定的属性，那么编译器会对代  
码进行去优化操作，并在稍后重新编译优化。

**4 Java HotSpot虚拟机的高级特性**   
 Java HotSpot 虚拟机拥有很多高级特性来提供对高扩展性、高性能和企业级可靠性、可  
用性和可服务性的支持。

4.1 扩展性   
 最近，Java HotSpot虚拟机添加了名为“人体工程学（Ergonomics）”的自适应机制。当  
前，人体工程学主要应用于两个方面。首先，根据目标机器的物理配置（例如，要考虑处理  
器的数量和可用物理内存的大小等），自动选择以客户端模式还是服务器端模式运行虚拟机。  
当物理硬件有大量的CPU，大量的内存时，会自动以服务器端模式运行虚拟机，并且会自动  
为服务器端应用程序选择一个适合的堆的大小值。第二，Java HotSpot 虚拟机中的垃圾收集  
算法也是会自动调整的，因此，已经不再需要显式的设置年轻代和老年代的相对大小了。垃  
圾收集器会自动进行调整，来提升应用程序的吞吐量，减少垃圾收集导致的程序暂停的时间。  
人体工程技术可提升服务器端应用程序的扩展性，在Java HotSpot 虚拟机将来的版本中，会  
对此作更多的开发工作。

4.2 性能   
 除了 Java HotSpot 虚拟机架构中使用的核心面向对象优化以外，虚拟机以及 Sun Java   
Runtime Environment（JRE）还提供了其他一些关键性能优化：   
λ 快速反射（Fast reflection）：现在， Java库会对那些经常使用的反射对象，如Method  
和 Constructor生成字节码存根。这项技术将反射调用暴露给Java HotSpot 编译器，  
提升了执行性能，在某些情况下，服务器端虚拟机可以完全剔除掉反射调用的性能  
损耗。可以明显提升某些大量使用反射操作的代码的执行效率，例如 RMI，序列化，  
和CORBA。   
λ 新 IO优化（New I/O optimizations）：Java HotSpot 编译器对新IO的 Buffer对象有专  
门的操作，为 get 方法和 put 方法调用生产保质量的机器代码。新 IO 提供了在网  
络和文件 IO 方面的高性能和高扩展性，也使得 Java 应用程序可以达到与 C/C++程  
序类似的吞吐量。新IO对 Buffer类优化也是用于其他类型的问题，如 3D图形，以  
及 Java平台之间大数据的传送。

4.3 可靠性、可用性和可服务性   
 当前版本的Java HotSpot 虚拟机是迄今为止最可靠的版本。基于对 Sun微系统公司大量  
应用程序的运行过程的跟踪，最近版本的虚拟机已经达到了企业级应用程序所需的可靠性和  
可用性的新纪录。   
 Java HotSpot 虚拟机包含了Java Virtual Machine Tools Interface（JVMTI）的参考实现。该  
接口允许一些工具来观察、控制JVM，如性能跟踪器、调试器和监视器等。JVMTI所包含的  
特性有：   
λ 全速调试（Full-speed debugging）：Java HotSpot 虚拟机利用动态去优化技术来支持  
对全速运行的应用程序的调试。在 Java虚拟机的早期实现中，当开启调试的时候，  
应用程序仅能在解释器中运行。在调试场景中启用 Java HotSpot 编译器可以极大的  
提升执行性能，并在在某些情况下，对某些可服务性有要求的应用程序来说，可以

始终开启调试功能。此外，启动调试器可能会抛出一些异常。   
λ 热替换（HotSwap）：Java HotSpot 虚拟机的面向对象架构提供了高级特性，例如类  
的运行时重定义，或称为热替换。该特性使应用程序可以在运行过程中通过调试  
API来改变代码。热替换为Java Platform Debugger Architecture（JPDA）添加的新功  
能，在调试器的控制下，可以在运行时修改类定义。此外，还可以在运行时插入一  
些性能监测代码来执行一些性能检测的操作。   
 Java HotSpot 虚拟机提供的一些额外特性提升了 Java应用程序的开发和可服务性。   
λ JNI错误检查（JNI error checking）：使用命令行选项“-Xcheck:jni”可以开启对JNI  
的额外检查。启用 JNI 错误检查后会在开发期间建超 JNI 参数的有效性，这样就可  
以在项目部署到正式环境之前发现问题。特别的，Java HotSpot 虚拟机会检查传给  
JNI 方法的参数的有效性，并在处理 JNI 请求前，检查运行时环境数据。如果在本  
地代码中发现了任何无效数据，虚拟机会自动终止。   
λ 错误报告（Error reporting）：如果JVM 检测到程序员编写的本地代码崩溃，或JVM  
本身崩溃，它会记录下相应的错误日志。一般来说，错误信息包括方法名称、库名  
称、源文件名称和发生错误的代码的行号。这样，程序员就可以更容易、更有效的  
调试程序。如果错误信息表示错误是由于 JVM 本身引起的，程序员也可以提交更  
准确、更有用的错误报告。   
λ 信号链（Signal-chaining facility）：信号链是Java平台能够更好的与拥有信号处理功  
能本地代码进行交互。该功能在Solaris和 Linux平台都可以正常工作。引入信号链  
是为了修复之前版本的Java HotSpot虚拟机中的问题。在1.4版本之前， Java HotSpot  
虚拟机中运行的应用程序无法对特定的信号进行相应，这些信号包括，SIGBUS、  
SIGSEGV 和 SIGILL。之所以有这个问题是因为这些信号处理与 Java HotSpot 虚拟机  
内部的信号处理有冲突。

**5 软件重用性的影响**   
 面向对象编程最主要的好处是，为了实现软件重用，它通过强有力的语言机制提升了软  
件开发的生产力。但实际上，软件重用很难实现。大量使用这种机制会显著降低执行性能，  
导致程序员在使用的时候也是持保守态度。Java HotSpot虚拟机的一个令人惊叹的优势是它  
显著的降低了这种性能损耗。Sun公司相信，如果各公司能够充分利用软件重用机制，并且  
无需向性能损耗妥协的话，那么这将多面向对象开发产生重要影响。   
 这种效果的案例很好找。一份对 Java 程序员的调查问卷显示，许多人避免使用虚方法  
调用（因此写了很多行数很大的方法），因为他们相信每个虚方法调用都会受到性能惩罚。  
普遍的、细粒度的虚方法使用，例如 Java 语言中的非静态方法或非 final 方法，对于构建高  
可用性的类是很重要的，因为每个这样的方法都扮演了“钩子”的角色，允许子类修改父类  
的行为。   
 因为 Java HotSpot 虚拟机可以自动对大部分虚方法调用做内联操作，所以性能惩罚会显  
著降低，很多情况下会完全剔除。   
 很难夸大这种影响的重要性。它潜在的改变了面向对象编程的方法，因为他显著的改变  
了使用重用机制的性能权衡。此外，随着面向对象编程的成熟，有一种趋势愈发明显，那就  
是编写更细粒度的对象和更细粒度的方法。这种趋势会增加虚方法的调用频率。随着这种更

高级的编码风格的流行，Java HotSpot 虚拟机技术的优势也更加突出。

**6 小结**   
 Java HotSpot 虚拟机致力于提升Java应用程序的性能，包括一些高级优化、垃圾回收和  
现场同步功能等。此外，为了提升基于 Java 技术的应用程序的整体的可靠性、可用性和可  
服务性，虚拟机还提供了调试功能。Java HotSpot 虚拟机分别为客户端环境可服务器端环境  
了提供了不同的编译器，这样应用程序可以根据自身部署环境而进行相应的优化。64 位的  
Java HotSpot 服务器端编译器也显著增强了应用程序的可扩展性。   
 在 Java HotSpot 虚拟机中，以客户端模式运行的应用程序启动更快速，占用更少的内存  
资源，以服务器端模式运行的应用程序则为长期运行提供了更好的性能支持。这两种方案都  
致力于提供可靠、安全和可维护的环境来满足企业客户的需要。

**7 使用**   
 Java HotSpot 虚拟机包含Java SE平台环境。在java.sun.com上可以找到对应如下环境的  
版本：   
λ SPARC平台上的Solaris操作环境；   
λ Intel平台上的Solaris操作环境；   
λ Linux操作系统，在包括 Red Hat Enterprise Linux、SuSE Linux和 Sun Java Desktop   
System的几个版本上都正式支持 Intel 平台， Java SE的最近几个版本也在其他 Linux  
平台。这里有与关于 Java SE相关的更多信息：   
λ Intel平台上的Microsoft Windows操作系统。   
 Java HotSpot 虚拟机已经成为苹果电脑公司的 Macintosh OS X 操作系统的一部分。Java   
HotSpot 服务器端虚拟机也包含在Hewlett-Packard公司为其PA-RISC硬件平台提供的Java SE  
技术的发行版中。