# Java 性能分析工具 , 第 3 部分: Java Mission Control

<http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/>

在解决程序性能问题之前，性能分析人员需要了解应用程序内部的运行状况以及应用运行环境的状况。本文将介绍使用在 Java 7u40 之后加入的性能监控新功能 Java Mission Control 来对 Java 应用程序进行分析。

0http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#icomments)

[李 伟军](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#authorN10023), 高级软件工程师, IBM

[宋 翰瀛](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#authorN10033), 资深软件工程师, IBM

[杨 翔宇](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#authorN10043), 资深软件工程师, IBM

2016 年 8 月 23 日

* [expand](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-j-lo-performance-analysissy-tools3-_-sidebar)

引言

本文为 Java 性能分析工具系列文章第三篇，这里将介绍如何使用 Java 任务控制器 Java Mission Control 深入分析 Java 应用程序的性能，为程序开发人员在使用 Java 任务控制器的时候提供帮助。[第一篇：操作系统工具](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/index.html)，[第二篇：Java 内置监控工具](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/index.html)。

JMC 是在 JAVA 7u40 发布中加入的性能监控工具。使用过 JDK 6 中 JRockit JVM 的用户并不会陌生，因为它是 Java 7 中 JMC 功能的一部分。启动 JMC 后将会显示当前机器中的所运行的 JVM 进程信息，当然我们也可以选择添加更多的 JVM 进程进行监控。图 1 中展示了使用 JMC 监控 GlassFish 应用服务器的画面。图中展示了被监控程序的基本信息，其包括 CPU 使用率和内存堆的使用率，值得注意的是，JMC 监控图中显示的是当前机器的 CPU 的使用情况，可以看到的是 JMC 监控的是整个系统，而并非只是被选中的 JVM 对 CPU 的使用情况。通过自定义设置上方仪表盘中显示的信息，既可以查看 被监控 JVM 的详细信息，例如垃圾回收，类的加载，线程的使用，以及内存堆的使用率，等等。也可以查看指定操作系统信息，例如系统的 CPU 和内存的使用率，磁盘的交换信息，平均负载等相关信息。

图 1. Java Mission Control 监控图



[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#ibm-pcon)

JFR (Java Flight Recorder)

JFR 是 JMC 中一个非常关键的功能。它记录了 JVM 所有事件的历史数据，通过这些数据，程序性能分析人员可以结合以往的历史数据对 JVM 性能瓶颈进行分析诊断。

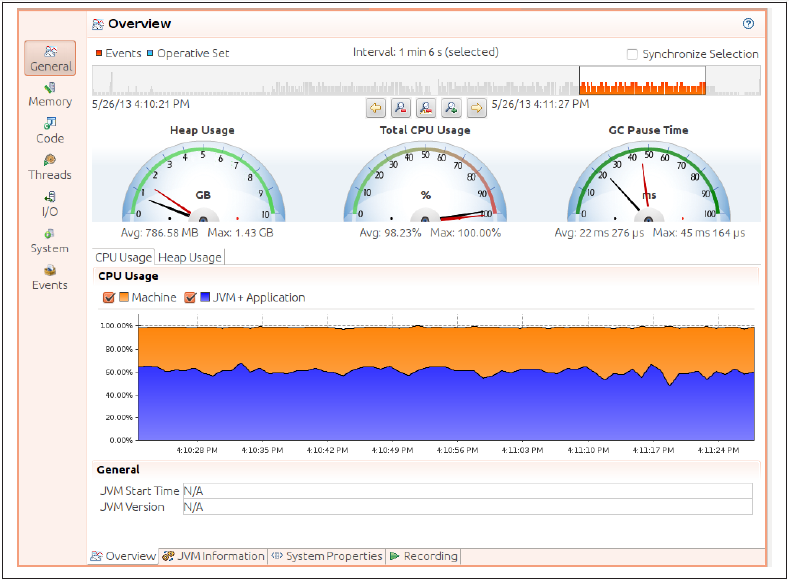
JFR 的基本操作是开启一系列的事件（例如，一个线程为了等待锁而被阻塞）。当某个事件发生时，这个事件的所有数据将会被保存至内存或者一个文件当中。数据流被保留在一个环形缓存中，所以只有最近发生的事件的数据才是可用的。JMC 可以从 JVM 或者文件中读取并展示这些事件数据，通过这些数据，可以进行性能分析。

通过 JVM 参数，JMC 用户界面以 jcmd 命令，可以指定上文中提到的事件类型、环形缓存的大小、数据存储的位置等信息。在默认设置下，JMC 对被监控应用的影响非常小。但是随着越来越多的事件被开启，以及触发事件的阀值的改变，JMC 的影响也在变化。接下来将展示 JFR 的用户界面。

JFR 概述

这个例子使用 JFR 记录 GlassFish 应用服务器的 6 分钟内的相关数据。图 2 展示了 JMC 将这些数据加载之后的到总体视图。

图 2. JMC 数据加载示例图



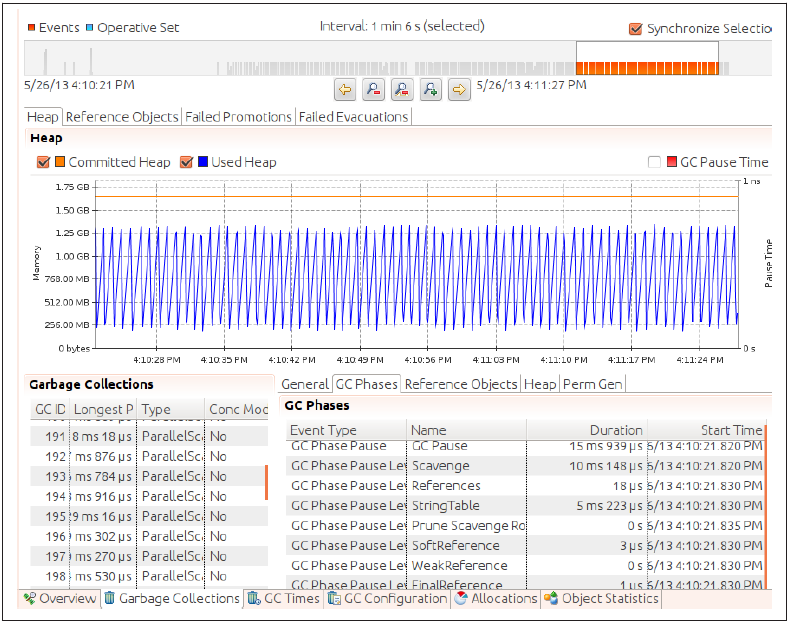
这张图上的信息类似于 JMC 基本信息监控所看到的画面。上面的仪表盘显示的是 CPU 使用率和内存堆使用率，在其上方有一条标示事件顺序的时间轴，垂直柱状体代表事件。可以放大时间轴中感兴趣的部分进行详细分析，就像图中所示，从 6 分钟的时间轴选出位于末端的 1 分 6 秒进行放大。

图 2 中 CPU 使用率曲线非常清楚的看到，GlassFish 服务器所在 JVM 平均大约 70%的使用率，而机器整体的 CPU 使用率达到 100%。在最下端，有一些标签可以选择，系统属性标签，JVM 信息标签等，在画面左侧的按钮提供更加详细的应用程序运行状况。

JFR 内存视图

JFR 内存视图收集的信息非常丰富，下图只展示了其中的一部分。从图 3 中可以看出，新生代（Young Generation) 占用的内存大小在周期性地进行有规律的波动。但有趣的是，由于没有对象需要移入老年代，应用整体的内存堆大小并没有增长。左下角面板中展示在监控期间所有类型的垃圾回收事件，在本例中一直为 ParallelScavenge。当选中一个事件时，右下角将更加细化的显示此事件，包括此事件的所有阶段以及每个阶段的时间消耗。

图 3. 新生代内存使用情况示例图

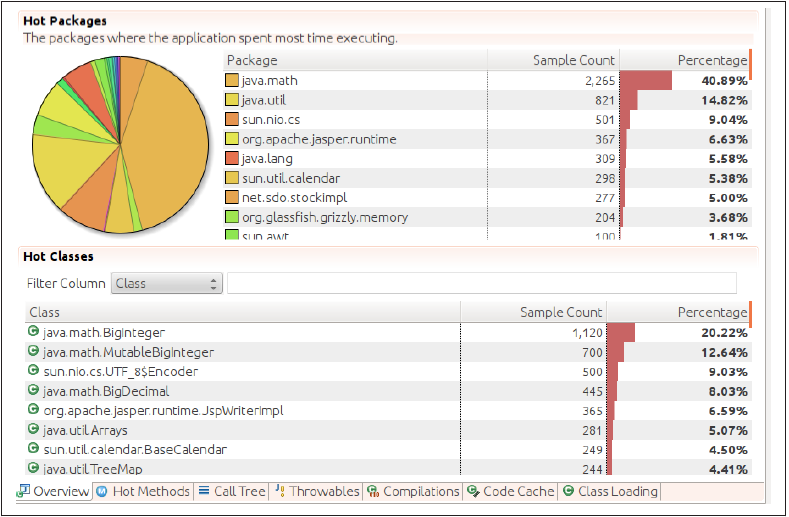


从图中各种各样的标签可以看到包含了丰富的信息，其中包括引用对象被清理的时间和数量，并发回收器中是否有 Promotion 失败或者 Evacuation 失败，垃圾回收器自身算法的配置，等等信息。

JFR 代码视图

JFR 代码视图展示了从记录的数据中获得的基础分析信息。如图 4 所示，第一个标签页显示了所有的包名，在其他分析工具我们并没见过类似的功能。可以看出图中使用 Java.Math 方法的执行占应用程序所有方法调用运行总时长的比重为 41%。传统的分析视角位于底部，例如热点方法和被分析代码的调用如下图所示。

图 4. Java Flight Recorder 代码视图



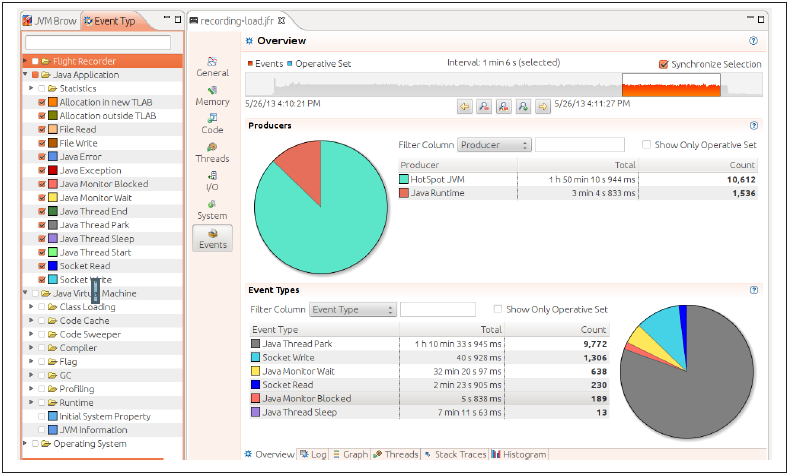
与其它分析软件不同，JFR 提供更多模式的代码可视化。Throwables 标签页展示应用中的异常处理，还有展示编译器工作信息的标签页。

JMC 还提供线程视图，I/O 视图，系统视图，所有的这些视图都是为了更好的分析 JFR 所记录的真实事件。

JFR 事件概述

JFR 记录并保存事件流，JMC 提供不同的视图来分析这些事件，但是 JFR 事件面板（如图 5 所示）才是分析事件最有效的途径。

图 5. Java Flight Recorder 事件视图



通过左侧面板的过滤器可以过滤显示的事件，在图中，只选择了应用级的事件。需要注意的是，只有在 JFR 记录保存事件中指定的事件类型才会出现在这里，在这里只能在此基础上进行过滤。

从图 5 中可以看出，在 66 秒内，应用程序产生 6 种类型的事件，其中 10612 个 JVM 事件和 1536 个 JDK 库事件。线程 Park 和 Monitor 事件消耗时间长的原因在前文中已有讨论，在此不再赘述。应用程序有多个线程共消耗 40 秒向套接字内写数据（Socket Write），这对于一个使用 4 核的应用来说是一个正常值，但仍然可以通过减少向套接字内写入的数据量来提高性能。

同样的，应用程序中多个线程共消耗 143 秒从套接字读取数据 (Socket Read)。这看起来并不正常，通过查看这些事件的处理记录可以发现，由多个线程使用阻塞式 I/O 读取并不连续的管理请求。这些管理请求的时间间隔通常很长，但这些线程却在 read() 方法内被阻塞，所以导致这些线程读取数据时消耗了过多的时间。

对于 Monitor 事件，

准性能分析器可以提供线程被阻塞之后调用 CPU 和操作系统代码等待解锁的相关信息，而 Java 本地分析器能够提供这些 CPU 和操作系统代码执行的记录，正如 JVM 直接将这些信息提供给 JFR。

JFR 事件是直接获得于 JVM 之内，通过这些事件能够非常详细的了解应用程序内部的运行状况，这是其他工具所不能比的。事件的类型非常之多，在不同版本的 JDK 之间也有所差异。下面列出一些常用的事件类型，第一行描述可以由其他工具获得的信息，第二行描述除了使用 JFR 无法获得信息。

* Classloading

被加载的类的数量和未被加载的类的数量

加载类的类加载器（classloader），加载一个类需要的时间

* Thread statistics

创建线程的数量、销毁线程的数量、线程快照（dump）

阻塞指定线程的锁以及被指定锁阻塞的线程

* Throwables

应用程序实用的异常类

应用程序抛出的异常和错误的数量以及创建异常和错误的栈记录

* TLAB allocation

内存堆中已分配的数量、TLAB（Thread-Local Allocation Buffers）的大小

在内存对中指定对象的内存分配以及为这些对象分配内存的栈记录

* File and socket I/O

进行 I/O 消耗的时间

每一次读写调用的时间消耗，读写操作消耗时间过长的文件或套接字

* Monitor blocked

等待监控器的线程

阻塞某个线程的监控器、线程被阻塞的时间

* Code cache

代码缓存的大小以及内容

从代码缓存中移除的方法、代码缓存的配置

* Code compilation

哪些方法被编译，OSR 编译，编译耗时

JFR 中没有额外的信息，但 JFR 总结了多个源文件的信息

* Garbage collection

GC 的次数，包括每一个阶段的次数、每个代的大小

JFR 中没有额外的信息，但 JFR 总结了来自多个工具的信息

* Profiling

检测分析和采样分析

JFR 并不能获得分析器获得的丰富信息，但 JFR 提供了更高层次的概述

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#ibm-pcon)

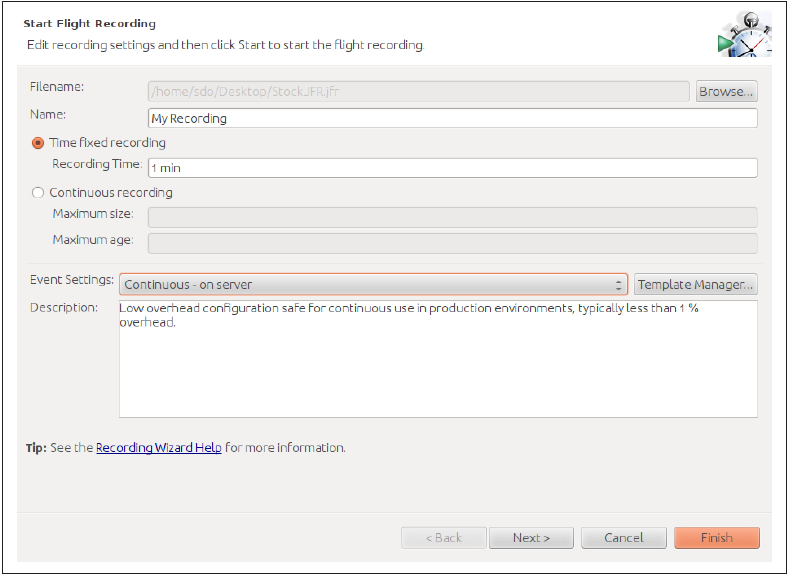
开启 JFR

Java Flight Recorder - JFR 默认被设置为关闭状态。通过在启动应用程序的命令中加入-XX:+UnlockCommercialFeatures –XX:+FlightRecorder 参数来开启 JFR，以及相关的一些功能。但是值得注意的是这个命令只是开启了 JFR 功能，但并没有开启记录进程各种事件，这就需要我们在开启了 JFR 功能之后，必须通过用户界面或者命令行来开启记录运行线程产生记录的功能。

通过 JMC 开启 JFR

最简单的方式是通过 JMC 来开启 JFR，当 JMC 启动后，界面上将显示当前系统中所有的 JVM 进程。JVM 进程以树形结构展示，展开 JVM 进程节点，通过 Flight Recorder 按钮开启 Flight Recorder 窗口，如图 6 所示。

图 6. Java Flight Recorder 记录窗口视图



Flight Recorder 有两种模式，固定时长模式（图中为 1 分钟）和持续模式，在持续模式中，使用环形缓存保存最近的事件。

在进行前瞻性分析（Proactive Analysis）时，例如在进行负载测试时开始记录事件，应该使用固定时长模式，这样可以很好的了解 JVM 在负责测试期间的表现。

持续模式适合反应（Reactive）分析，这样可以使 JVM 保存最近的事件并导出记录响应这些事件的数据。举例来说，Weblogic 应用服务中可以设置触发器，在应用服务器响应了一个反常事件之后导出相关数据。

通过命令行开启 JFR

在 JVM 启动参数中加入-XX:+FlightRecorderOption=参数启动记录事件。参数为一个以逗号间隔的键值对数据。下面将介绍这些选项：

* name=name

指定记录的名称

* defaultrecording=true/false

是否在初始化时启动记录事件，默认为 false，对于反应分析，应设置为 true

* settings=path

JFR 配置文件的路径

* delay=time

开始记录前的延时

* duration=time

记录持续时间

* filename=path

保存记录事件的文件路径

* compress=true/false

是否使用 gzip 压缩记录数据，默认为 false

* maxage=time

环形缓存中保存记录的数据的最长时间

* maxsize=size

环形缓存占用的最大空间

为了更灵活的使用 JFR，在运行过程中可以使用 jcmd 命令来动态改变这些选项的值。下面将介绍使用 jcmd 来设置 JFR 相关选项的命令。

% jcmd process\_id JFR.start [option\_list]

option\_list 是以逗号分隔的键值对，选项与上述选项一致。当使用持续模式记录事件时，可以使用下面的命令将环形缓存中的数据导出至文件中：

% jcm process\_id JFR.dump [option\_list]

选项包括：

* name=name

指定导出数据的事件记录名

* recording=n

JFR 事件记录号

* filename=path

导出文件的路径

一个进程下可能有多个 JFR 事件记录在运行，可以通过下面的命令查看当前所有的事件记录：

% jcmd process\_id JFR.check [verbose]

这些不同的事件记录以名字或者随机的记录号进行区分。使用下面的命令可以停止一个进程内正在运行的事件记录：

% jcmd process\_id JFR.stop [option\_list]

选项包括：

* name=name

需要停止的事件记录名

* recording=n

需要停止的事件记录号

* discard=Boolean

如果为 true，舍弃之前保存的数据

* filename=path

保存数据的文件路径

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#ibm-pcon)

选择 JFR 事件

JFR 目前支持 77 种事件类型，大多数类型的事件为周期性事件，每隔固定时间将会产生一次，其他类型的事件为触发型，只有当达到某种设定的阀值后被触发，也就是说才会被 JFR 发现并监控起来。

收集事件数据自然会影响应用程序性能，影响的大小由开启记录的事件的类型和数量有关。默认情况下，并没有开启所有事件类型的记录，将对性能影响较大的 6 种类型事件关闭，以此达到将 JFR 对应用程的影响降到 1%以内。

但有时开启一些对应用影响较大的事件类型是值得的，例如 TLAB 是默认关闭的事件类型，但是它能够帮助人们发现某些对象是否在分配内存时直接进入老年代（Old Generation）。同样的，Profiling 事件虽然默认开启，但是每 20 毫秒触发一次，这样很容以导致采样分析出现错误。

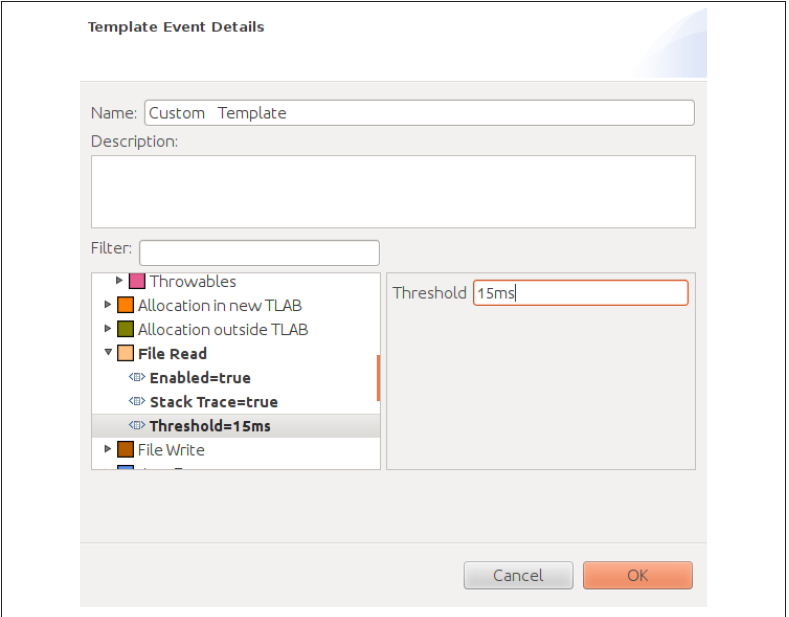
JFR 记录的事件定义在一个模版中，可以通过 settings 选项指定。JFR 运行时有两个模版：

1. 默认模版：事件对应用性能的影响在 1%以内
2. 分析模版：设置大部分基于阀值（Threshhold）的事件为 10 毫秒触发一次，分析模版对应用性能的影响大约为 2%。

这些模版由 JMC 模版管理器（Template Manager）管理，模版管理器通过 JMC 中相应的按钮启动。存储模版的目录有两个：$HOME/.JMC/<release>directory 和$JAVA\_HOME/jre/lib/jfr。在模版管理器中，可以选择一个全局模版、本地模版或者定义一个新模版。定义一个新模版是时，需要指定可用的事件，指定这些事件默认的状态为开启或者关闭，同时可以定义这些事件的阀值。

在图 7 中，File Read 事件被开启并且设置阀值为 15 毫秒，即当读取文件超过 15 毫秒时，此事件将触发。同时此事件还被设置为 File Read 事件生成栈记录，由于这将增加事件对应用性能的影响，所以将此选项设置为可选配置项。

图 7. Java Flight Recorder 事件模板示例图



事件模版文件为 xml 文件，所以通过读取 xml 文件很容易可以判断某个事件是否开启。也可以将本地模版文件拷贝至全局模版目录供其他用户和团队使用。

Java Flight Recorder（JFR）提供了深入 JVM 内部去看运行时的状态。这是由于 JFR 其本身就是 Java 的一部分内嵌其中的。如同其他工具，JFR 的使用某种程度上增加了对应用程序性能的开销。常规的使用 JFR 还是可以帮助我们收集大量的信息，而且对性能的开销是较低的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools3/#ibm-pcon)

总结

到此为止，本系列文章已经将常用的 Java 性能分析工具进行了介绍，我们都知道一个优秀的性能分析工具是我们在分析性能问题的关键，但是不能完全，甚至过分的依赖工具所能展现的表面，我们更应当铭记，没有最好的工具，只有是否适合的工具，正如我们讲过的工具 A 也许适合大多数的应用程序分析，但是它却不能够揭示工具 B 所能指出的问题，所以我们应当学会灵活自由的使用这些工具。

对于性能分析人员来说命令行工具也是非常有帮助，所以不要拒绝它所能带来的强大力量。它往往能在自动化的性能监控，或者性能测试中发挥巨大作用。

希望这些介绍能够给性能调优人员和开发人员的日常工作带来帮助。俗话说金无足赤，人无完人，工具亦如此。每一个工具都有其各自的优劣势，只有充分了解每个工具适合的场景，才能让这些工具扬长避短，更加充分地发挥其作用。

参考资料

* 访问 developerWorks [Java 专区](http://www.ibm.com/developerworks/cn/java)，了解关于信息管理的更多信息，获取技术文档、how-to 文章、培训、下载、产品信息以及其他资源。
* 加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/)。查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。

# Java 性能分析工具 , 第 2 部分：Java 内置监控工具

<http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/>

在解决程序性能问题之前，性能分析人员需要了解应用程序内部的运行状况以及应用运行环境的状况。本文将介绍使用 Java 内置监控工具将程序运行状况和 JVM 信息以直观的方式展现以便于分析，为性能分析和开发人员提供帮助。

0http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#icomments)

[李 伟军](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#authorN10023), 高级软件工程师, IBM

[宋 翰瀛](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#authorN10033), 资深软件工程师, IBM

[杨 翔宇](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#authorN10043), 资深软件工程师, IBM

2016 年 8 月 23 日

* [expand](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-j-lo-performance-analysissy-tools2-_-sidebar)

引言

本文为 Java 性能分析工具系列文章第二篇，[第一篇：操作系统工具](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/index.html)。在本文中将介绍如何使用 Java 内置监控工具更加深入的了解 Java 应用程序和 JVM 本身。在 JDK 中有许多内置的工具，其中包括：

* jcmd：打印一个 Java 进程的类，线程以及虚拟机信息。适合用在脚本中。使用 jcmd - h 来查看使用方法。
* jconsole：提供 JVM 活动的图形化展示，包括线程使用，类使用以及垃圾回收（GC）信息。
* jhat：帮助分析内存堆存储。
* jmap：提供 JVM 内存使用信息，适用于脚本中。
* jinfo：访问 JVM 系统属性，同时可以动态修改这些属性。
* jstack：提供 Java 进程内的线程堆栈信息。
* jstat：提供 Java 垃圾回收以及类加载信息。
* jvisualvm：监控 JVM 的可视化工具，剖析运行中的应用程序，分析 JVM 堆存储。

下面将根据功能划分来详细介绍这些工具。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#ibm-pcon)

VM 基本信息

JVM 工具能够提供一个运行中的 JVM 进程的基本信息，例如运行时间、使用中的 JVM 参数以及 JVM 系统属性。

* uptime

JVM 运行的时间，jcmd process\_id VM.uptime

* system properties

通过 System.getProperties() 可以得到的系统属性也可以通过下面的命令获得：

jcmd process\_id VM.system\_properties 或者 jinfo –sysprops process\_id

这些属性包括所有通过命令行-D 选项设置的属性、应用程序动态添加的属性和 JVM 的默认属性。

* JVM version

通过 jcmd process\_id VM.version 获得。

* JVM command line

JVM 命令行可以在 jconsole 中的 VM summary 中找到，或者通过 jcmd process\_id VM.command\_line 命令获得。

* JVM 调优参数

通过命令 jcmd process\_id VM.flags [-all] 命令或者所有生效的调优参数获得。

使用调优参数（Tuning Flags）

由于调优参数非常繁多，需要借助 JVM 命令行和 JVM 调优参数来使用。使用 command\_line 命令可以获得命令行中指定的调优参数，flags 命令可以获得通过命令设置的调优参数和 JVM 设置的调优参数。

通过 jcmd 命令可以获得一个运行中 JVM 内生效的调优参数。通过下面这条命令可以获得一个指定平台内生效的调优参数。

java other\_options –XX:+PrintFlagsFinal –version

我们需要把其他选项同时包含在这条命令中，尤其是设置了 GC 相关的调优参数。这条命令的部分输出如下所示，第一行中的冒号说明第一行的调优参数使用的不是默认值，而是以下三种方式设置：

1. 通过命令行设置

其他选项间接改变了此调优参数的值

1. JVM 计算出默认值

第二行由于没有包含冒号，说明此行的调优参数为当前 JVM 版本的默认值，最后一列的 product 说明此行的调优参数的值在不同平台相同，而 pd product 说明此行的调优参数的值依赖于平台。

uintx InitialHeapSize := 4169431040 {product}

intx InlineSmallCode = 2000 {pd product}

最后一列的其他选项：

manageable：此 flag 的值可以在运行时动态改变

c2 diagnostic：此 flag 提供帮助工程师理解的编译器如何工作的帮助信息。

jinfo 命令可以查看某个单一 flag 的值，通过下面的命令：

jinfo －flag PringGCDetails process\_id –XX:+PrintGCDetails

1. 通过下面的命令可以设置某个 flag 的 manageable 属性来控制其能否在运行时被改变：

jinfo －flag －printgcdetails process\_id # 关闭 PrintGCDetails 的 manageable 属性

尽管 jinfo 命令可以改变任何 flag 的值，但不能确定 JVM 会接受这些改变。例如很多影响垃圾回收算法执行的 flag 都会在 JVM 启动时被设置，在 JVM 运行过程中通过 jinfo 命令修改 flag 的值并不会影响算法执行。所有此命令只对那些 manageable 为真的 flag 起作用。

线程信息

jconsole 和 jvisualvm 命令可以帮助开发人员剖析应用程序运行过程中线程的相关信息。通过 jstack process\_id 命令可以查看线程的运行时栈信息，可以明确获得当前线程是否被阻塞。通过命令 jcmd process\_id Thread.print 可以获得相同结果。

类信息

通过 jconsole 和 jstat 命令可以获得应用程序运行过程中的所有类的相关信息，同时 jstat 命令也提供了类编译的相关信息。

垃圾回收信息

Jconsole 展示了 JVM 堆使用的情况，它所绘制的的动态图能够帮助开发人员了解堆的内部情况。jcmd 支持垃圾回收操作。jmap 提供堆信息总览。jstat 从不同的角度展示垃圾回收是如何工作的。

Heap Dump 文件的后序处理

通过 jvisualvm 用户界面可以得到 Heap Dump 文件，通过 jcmd 和 jmap 也可以获得。Heap Dump 文件是堆的快照，一般使用 jvisualvm 和 jhat 来分析这个快照。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#ibm-pcon)

性能分析工具

Java 提供的性能分析器是最重要的分析工具。它的种类繁多，各有所长，使用不同的分析器在分析同一个应用时可能会发现不同的问题。在使用过程中需要各取所长，这样才能对应用进行全面的分析。

基本上所有的 Java 性能分析器都是用 Java 实现的，通过套接字（socket）与被分析应用进行通信来获得被分析应用的运行信息。需要注意的是，在使用性能分析工具调优被分析应用的同时，需要关注性能分析器其自身的性能。假如当被分析的应用程序产生十分庞大的信息，而将其发送至性能分析器时，如果性能分析器没有空间充分管理高效的内存堆来处理这些信息时，分析将无法进行。采用并行垃圾回收算法进行内存管理是当前性能分析器比较流行的做法，这种算法能够最大程度地降低内存溢出的可能。

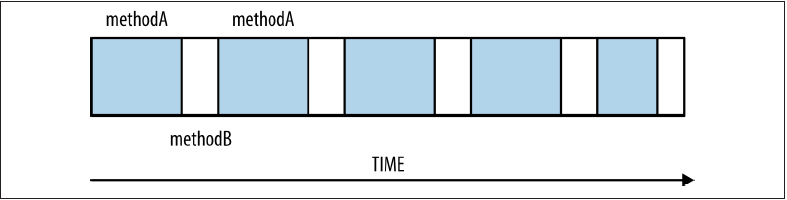
性能分析分为采样模式和检测模式。下面将分别介绍这两种模式。

采样分析

采样模式是性能分析中最常用的模式，因为其对被分析应用程序影响最小，这一点非常重要。只有当性能分析过程对应用程序的影响降到最低，才能获得有价值的性能分析结果。

在采样分析模式中，分析器被定时触发工作。在工作周期内，分析器依次检查每个线程并记录线程中正在运行的方法，在某些特定场景下，采样分析往往会带来错误的分析结果。例如，在图 1 中，某线程在一段时间内交替执行方法 A 和方法 B，每次当分析器被触发工作时，该线程都恰好在执行方法 B，那么分析器会认为该线程的所有时间都是在执行方法 B，但是事实并非如此，该线程执行方法 A 的时间远大于执行方法 B 的时间，只是并未被分析器采样到。

图 1. 某一时间段内线程交替执行方法 A 和 B 示例图



这是采样模式中最常见的错误，通过增加采样分析器的采样时间隔可以帮助我们有效的减少这类错误的发生，因为时间间隔太小往往会增加采样分析器对被分析的应用程序产生性能方面的影响，从而导致分析结果失真。所以时间间隔需要根据被分析应用的特点通过多次的试验以及经验来决定，权衡过大或过小的影响之后设定。

图 2. 采样模式分析示例图

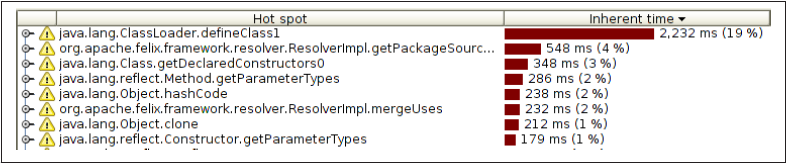
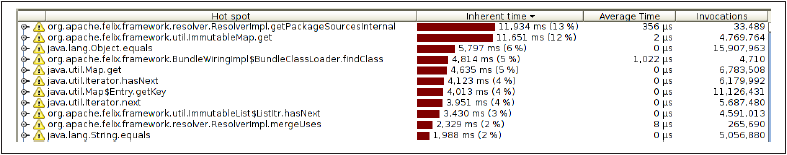


图 2 所示是使用采样模式分析一个应用服务器 GlassFish 启动过程的结果。从图中可以看到，方法 defineClass1() 使用了 19%的时间，接下来是方法 getPackageSourceInternal()，占用了 10%的时间。Java 应用程序中定义的类会影响应用程序启动过程中的性能表现，为了提高应用程序的启动速度，就必须通过提高类加载的速度，从而达到提升启动速度的目标。从图中我们可能会错误的认为要改善性能的方法是 defineClass1()，但是 defineClass1() 其实是 JDK 中的方法，我们不可能通过重写 JVM 来提高它的性能。即使重写此方法将其执行时间优化至原有时间的 60%，也只能减少 10%应用程序整体运行时间，这显然得不偿失。

检测分析

相比于采样模式，检测模式是要侵入被分析的应用程序内部，虽然这样做并不是高效、友好的，但它却可以获得非常有价值的信息。图 3 为使用相同分析工具的检测模式分析相同应用服务器 GlassFish 的结果。

图 3. 检测分析示例图



在图中有以下几点信息：

1. 最耗时的方法为 getPackageSourcesInternal(), 占用了 13%的时间，而并非在采样模式中得到的 4%；
2. 方法 defineClaass1() 并未出现在分析结果中。
3. 分析结果中包括每个方法执行的次数和平均耗时。

这些分析结果中的信息对于发现耗时多的代码是非常有帮助的。在本例中，尽管方法 ImmutableMap.get() 消耗 12%的时间，但是它被调用了四百七十万次之多。如果减少此方法的调用次数，应用的性能将会得到大幅度提升。

检测分析器在类被加载时通过改变其字节码顺序来获取应用运行数据，例如增加记录方法被调用次数的代码。相比于采样模式，这种方式会更大程度的影响应用本身的性能。例如，JVM 会根据方法的代码块大小，将方法体很小的方法内联化，这样在内联方法执行时就不会进行方法调用。在检测分析器在内联方法中加入其代码后，此方法因为方法体过大并未被 JVM 内联化，由此造成此方法的耗时被放大。内联化只是一个例子，当越来越多的代码被改变的时候，分析的结果失真的概率就会比较大。

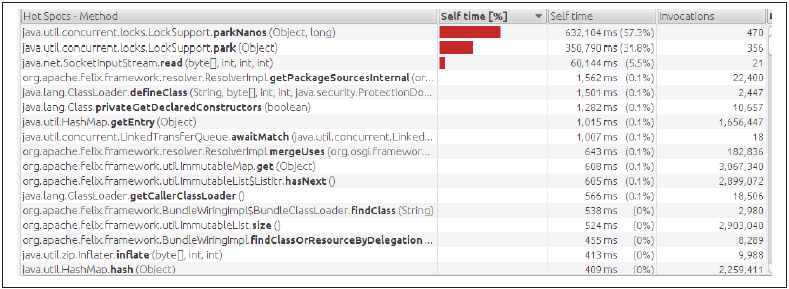
造成方法 ImmutableMap.get() 没有出现在采样模式分析结果中的原因是安全点（safepoint）的存在。只有当一个线程获得的内存大于安全点时，采样分析器才会对其进行分析。因为方法 ImmutableMap.get() 所在线程一直没有达到安全点，所以在结果中不会出现。当使用采样模式安全点过高时，会低估一些方法对性能的影响。

在本例中，无论是采样分析还是检测分析，都能发现应用的性能瓶颈在于类的加载和解析。但是在实际中，不同的分析器不可能得出完全相同的分析结果。分析器擅长估量，但也只是估量，一些误差甚至是错误不可避免，所以在性能分析过程中还需要我们更加灵活的使用分析器。

阻塞方法和线程的时间轴

如图 4 所示为使用 NetBeans Profiler（另一种检测分析器）分析上述应用服务器 GlassFish 启动过程的结果展示。在此结果中，方法 park()，parkNanos() 和 read() 占用了绝大多数的应用运行时间。这些方法都是被阻塞的方法，并不消耗 CPU，所以在计算应用的 CPU 使用率时这些时间不应计入。应用中的线程并没有使用 632 秒来执行 parkNanos() 方法，而是等待其他操作完成花费 632 秒。park() 和 read() 方法与此同理。

图 4. NetBeans 检测分析示例图



因此，大多数的分析器都不会将被阻塞的方法和闲置的线程计入结果。在 NetBeans 中，可以设置分析结果包含所有的方法，所以在本例中这些方法被计入结果。在本例中，执行 park() 方法的线程位于服务器线程池中，当服务器接收到请求时，这些线程处理请求。当没有请求时，这些线程处于阻塞状态，等到新的请求，并不占用 CPU。这是应用服务器的正常状态。

绝大多数的基于 Java 的分析器都可以提供过滤器功能来查看或者隐藏被阻塞方法调用的时间，如果需要可以使用该功能。通常情况下，查看线程的运行状况比查看被阻塞方法的阻塞时间更加有帮助。

图 5. Oracle Solaris Studio 中线程的运行示例图

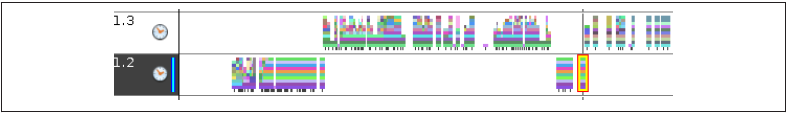


图 5 为在 Oracle Solaris Studio 中一个线程的运行情况。每一个水平区域代表一个不同的线程，所以上图中有两个线程（1.3 和 1.2）。不同颜色的柱子代表执行的不同方法；空白处代表该线程没有执行任何方法。综合来看，线程 1.2 先执行了一段代码然后等待线程 1.3 完成执行，线程 1.3 完成执行后等待线程 1.2 执行另一段代码。深入下去可以发现这些线程如何进行交互。

图中存在一些没有线程执行的空白区域，这是因为图中只展示了其中的两个线程，所以在那段空白区域是图中所示两个线程在等待其他线程执行完成。

本地分析器

本地分析器是用来分析 JVM 本身的工具。通过本地分析器可以观察到 JVM 正在进行的操作或者查看是否有应用程序包含了 JVM 的本地库，也可以观察到代码内部。任何本地分析器都可以分析使用 C 语言实现的 JVM（包括所有本地库），但是一些本地分析其不能分析使用 Java 和 C++实现的应用。

图 6. 本地分析器分析示例图

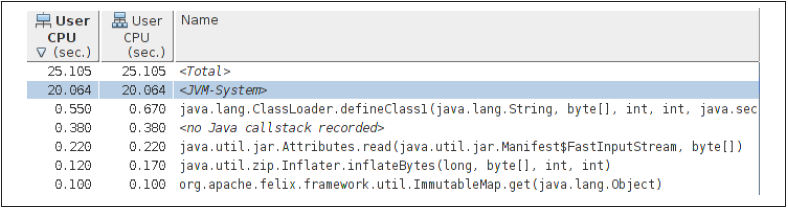
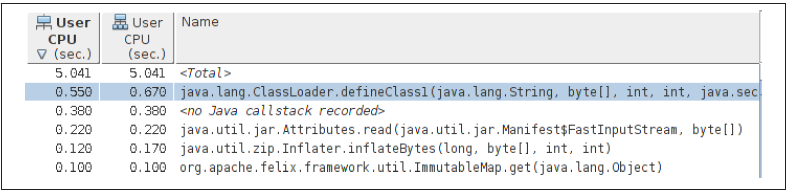


图 6 中展示了使用 Oracle Solaris Studio 分析器中分析 GlassFish 启动过程的结果。Oracle Solaris Studio 是一个可以分析 Java 和 C++的本地分析器。从图中可以发现，应用消耗的 CPU 时间为 25.1 秒。其中 JVM-System 消耗 20 秒，包括 JVM 编译器线程，垃圾回收线程以及一些辅助线程。由于在启动过程中需要编译非常多的代码，所以 JVM 编译器线程消耗了绝大多数时间，而垃圾回收线程只消耗了很少的时间。

通过本地分析器我们不仅可以分析优化 JVM 自身功能，更重要的是可以获得应用程序进行垃圾回收的时间。在 Java 分析工具中，垃圾回收线程的信息是无法得到的。

分析过 JVM 本地代码后，我们将对应用程序的启动过程进行分析。如图 7 所示，继在采样模式分析后，方法 defineclass1() 又一次被分析为最耗时的方法。值得关注的是，再次分析结果中，解压读取 jar 文件的方法耗时相对较多。类加载中会用到这些方法，所以证明优化的方向是正确的。由于 Java zip 库中引用的本地代码在其他分析工具中被作为阻塞方法调用，所以在上文各类工具中并没有发现此方法。

图 7. 采样模式分析示例图



无论使用何种性能分析工具，最重要的是熟悉每种工具的优势和劣势。这样才能取长补短，配合使用。开发人员必须学会如何使用性能分析器来找到性能瓶颈，找到需要优化的代码，而不是单纯的关注最耗时的个别方法。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools2/#ibm-pcon)

总结

基于采样的性能分析是最常见的一种，因为其相对能做到的分析是有限的，亦或者分析过程所能搜集到的信息是概述性的，往往并不能真实表现应用程序内部的运行情况，但是其分析过程中引入的工作量通常是较低的。不同的采样分析工具行为是不同的，充分利用其优势，做有针对性的分析才是最有意义的。

检测分析能够获得非常多的有关应用程序内部信息，但是前期准备工作往往是非常大的。检测分析方法应当尽量应用在一小节代码中，或者少数几个类、包中。这种方法其实一定程度上限制了对整体应用程序的性能分析，仅适合在程序单元中使用，点对点，针对性较强的分析，采用检测分析的时候，更多时间要求开发人员明确知道哪里有可能产生性能瓶颈。

线程阻塞不一定就是代码编写而产生的，发生线程阻塞时，更多的建议是去想，去看为什么会被阻塞，而不是直接查看代码。尽量采用线程执行时间轴的分析方法。

本地分析提供了既可以深入查看 JVM 内部，同时也可以查看应用程序代码执行的情况。

如果本地分析显示在 GC 过程中大量的使用 CPU 资源，那么调优收集器就是必要的。需要提醒大家的是，编译线程通常是不影响应用程序的性能。

参考资料

* 访问 developerWorks [Java 专区](http://www.ibm.com/developerworks/cn/java)，了解关于信息管理的更多信息，获取技术文档、how-to 文章、培训、下载、产品信息以及其他资源。
* 加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/)。查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。

# Java 性能分析工具 , 第 1 部分: 操作系统工具

https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/

的状况。本文将介绍使用操作系统中集成的程序监控工具将程序运行状况以直观的方式展现以便于分析，为性能分析和开发人员提供帮助。

0https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#icomments)

[李 伟军](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#authorN10018), 高级软件工程师, IBM

[宋 翰瀛](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#authorN10028), 资深软件工程师, IBM

[杨 翔宇](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#authorN10038), 资深软件工程师, IBM

2016 年 7 月 06 日

* [expand](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-j-lo-performance-analysissy-tools-_-sidebar)

引言

性能分析的前提是将应用程序内部的运行状况以及应用运行环境的状况以一种可视化的方式更加直接的展现出来，如何来达到这种可视化的展示呢？我们需要配合使用操作系统中集成的程序监控工具和 Java 中内置的监控分析工具来进行 Java 程序的性能分析。本文为系列文章，共三篇分别介绍这几类工具。在本文中将介绍操作系统中的性能监控工具。

操作系统中的程序性能监控工具并非只针对于 Java 程序，适用于所有运行其中的程序。在基于 UNIX 的操作系统中，有许多命令行工具可以用来监控程序的运行状况，例如 sar, vmstat, iostat，prstat 等等。在 Windows 操作系统中，既有图形化用户界面的资源监控器 Perfmon（Performance Monitor），也有如 typeperf 的命令行工具。

进行性能测试时，我们需要通过操作系统提供的工具收集操作系统中的各类资源监控数据，包括 CPU、内存和硬盘的使用数据，如果被测试程序使用了网络，还需要收集网络使用数据。只有收集的数据足够完整和充分，性能测试的结果才会更准确，性能分析也会更加容易进行。下面将首先介绍在 UNIX/类 UNIX 系统中各类资源的监控和分析的方法。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#ibm-pcon)

Linux 系统资源的监控

CPU 使用率

CPU 使用时间分为两类：用户时间（User Time）和系统时间（System Time），系统时间在 Windows 系统中被称为特权时间（Privileged Time）。用户时间为 CPU 执行应用程序代码的时间，而系统时间则为 CPU 执行操作系统内核代码的时间比例。系统时间与应用程序本身有关，例如当应用程序执行 I/O 操作时，操作系统内核将会执行从硬盘读取文件的代码，或者执行向网络数据缓存中写入数据的代码。应用程序中任何需要使用操作系统底层资源的行为都会导致应用程序占用更多的系统时间。

性能调优的终极目标是在单位时间内最大限度提高 CPU 使用率。CPU 使用率是在一个特定时间间隔内的平均值，这个时间间隔可以是 30 秒，也可以是 1 秒。比如，一个程序需要 10 分钟执行完成，在此期间该程序的 CPU 使用率为 50%。当对程序代码优化之后，CPU 使用率提高为 100%，那么该程序的性能将提升一倍，只需要 5 分钟执行完成。当该程序再次优化代码使用 2 个 CPU，CPU 的使用率依然为 100%，那么该程序将只需要 2.5 分钟执行结束。从这个例子可以看出，CPU 使用率能够反应程序使用 CPU 的效率，CPU 使用率越高程序性能越好，反之亦然。

在 Linux 操作系统中执行 vmstat 5 命令，将会得到如清单 1 中的数据（每 5 秒增加一行）。为了易于理解，此例中的程序只使用单线程运行，在多线程环境中同样适用。从示例数据中第一行数据可以知道，在 5 秒内，CPU 一共被使用了 2.25 秒（5\*（37%＋8%）），其中 37%的时间用于执行用户代码，8%的时间用户执行系统代码。剩余的 2.75 秒 CPU 处于闲置状态（idle）。

清单 1. vmstat 5 命令结果

procs -----------memory--------------- ----swap---- ---io--- -----system------ ----------CPU-------

r b swpd free buff cache si so bi bo in cs us sy id wa st

2 0 236456 2259632 200052 730348 0 0 1 6 1 1 37 8 55 0 0

2 0 236456 2259624 200052 730348 0 0 0 10 179 332 40 7 53 0 0

2 0 236456 2259624 200052 730348 0 0 0 20 180 356 56 7 37 0 0

以下三点原因会造成 CPU 闲置：

1. 应用程序被线程的同步操作阻塞，直到锁被释放；
2. 应用程在等待某些请求的响应，例如等待数据查询请求的响应；
3. 应用程序无事可做；

前两种情况比较容易理解，也有对应的调优方式。针对原因一，如果能够减少锁的竞争，或者调整数据库返回请求资源的性能，那么应用程序会运行的更快；对于原因二，优化请求响应方，提高响应速度；那么在其他条件不变的情况下，应用程序会运行的更快，CPU 使用率也会提高。

第三种情况当应用程序有事去做时，CPU 将利用 CPU 周期去执行应用程序的代码。这是一条通用的规则。当执行一段无限循环的代码（如下所示）时，它将会再消耗一个 CPU 100%的时间。如果 CPU 的使用率并没有达到 100%，意味着操作系统应该执行无限循环，但它并没有去做而是处于闲置状态。这种情况对于无限循环并没有多少影响，但是如果我们的程序是用来计算一个表达式的结果，那么这种情况将会导致计算的速度变慢。

清单 2. 无限循环示例

#!/bin/bash

while true

do

echo“In the loop…”

done

当在一台单核机器上运行清单 2 中的代码，绝大多数时间我们不会注意到它在运行。但是如果启动另一个程序或者监控另一个程序的性能时，这种影响就会体现出来。操作系统善于利用时间切片程序来竞争 CPU 周期，但是最新启动的程序只能获得极少的的可用 CPU 周期。有一种解决问题的方案，那就是留出一定比例的闲置 CPU 周期以防有其他程序需要使用 CPU。但是这种方案暴露出来的问题就是操作系统无法知晓下一步操作，操作系统只能去执行当前所有的操作而不会留出闲置的 CPU 周期。

Java 和单 CPU 使用率

我们回到 Java 应用程序，周期性的 CPU 闲置意味着什么呢？这取决于应用程序的类型。对于有固定作业量的批处理程序，除非全部作业完成，否则 CPU 不会有闲置时间。提高 CPU 的使用率可以使批处理程序更快地完成。如果 CPU 使用率已经达到 100%，我们可以在保持 CPU 使用率 100%的前提下从其他方面进行优化使程序完成地更快。

对于接收请求的服务器类型的应用程序，在没有请求到来的时候，CPU 会处于闲置状态。举例来说，当 Web 服务器处理完当前所有 HTTP 请求处于等待下一个请求的状态时，CPU 为闲置状态。从这里可以理解 CPU 使用率为何为一定时间间隔内的平均数值。上文 vmstat 示例中的数据采集自一个应用服务器的运行过程中，这个服务器每 5 秒接收到一个请求，花费 2.25 秒处理，这意味着在这 2.25 秒内 CPU 的使用率为 100%，而在剩下的 2.75 内使用率为 0。由此计算得出 CPU 使用率为 45%。

这种情况总是发生在非常短的时间间隔内，因此很难被发现，但是这种类似应用服务器的程序总是按照此方式运行。当我们降低时间间隔，上述应用服务每 2.5 秒接受一个请求同时花费 1.125 秒处理请求，剩余的 1.375 秒 CPU 处于闲置状态。平均下来，CPU 平均使用率依然为 45%，55%的时间处于闲置状态。

优化应用服务器之后，处理每个请求只需要 2 秒，CPU 使用率将降至 40%。降低 CPU 使用率是我们优化程序代码的目标。只有在单位时间内，没有外部资源约束的应用程序负载固定。从另一方面来讲，优化这种应用程序可以适当增加程序负载来提升 CPU 使用率。这样一来，可以看出这种优化策略依然遵循前文的规则，即在尽可能短的时间内使 CPU 使用率尽可能高。

Java 和多 CPU 使用率

调优多线程程序的目标依然是尽可能的提高每个 CPU 的使用率，使 CPU 尽可能少的被阻塞。在多核多线程环境中，当 CPU 处于闲置状态时需要多考虑的是即使应用程序有作业未完成，CPU 依然会处于闲置状态，因为该应用程序中没有可用的线程来处理作业。最典型的例子为一个拥有固定大小线程池的应用程序运行数量变化的任务。每个线程每次只能处理一个任务，如果此线程被某些操作阻塞，这个线程不能转而去处理另一个任务。在这种情况下就会出现没有可用线程来处理未完成的任务。因此会导致 CPU 处于闲置状态。对于此种情形应该考虑如何增加线程池的大小来完成更多的任务。

监控 CPU 使用率只是理解应用程序性能的第一步，这只能确定代码的 CPU 使用率是否达到开发人员的期望，或者找到代码中存在的同步问题和资源问题。

CPU Run Queue

在 Windows 和 UNIX 系统中都可以监控当前可以执行任务的线程数。UNIX 系统中称为 Run Queue，有很多工具可以查到此数据。例如前文中的 vmstat，每行的第一个数字即是 Run Queue 的长度。Windows 系统中称之为 Processor Queue，可以通过 typeperf 命令查到。

Windows 与 UNIX 的区别是：在 UNIX 中，Run Queue 长度为当前正在运行和可以运行的线程数，所以这个长度最小为 1；而在 Windows 中，Processor Queue 长度并不包括正在运行的线程数，因此 Processor Queue 长度最小值为 0。

当可用线程数大于可用 CPU 数量，性能就会下降。所以在 Windows 中 Processor Queue 长度为 0，在 UNIX 中 Run Queue 长度等于 CPU 数的情况性能达到最好。但这并不绝对，因为系统程序中会周期性运行导致此数值增大，单对应用程序的影响不大。如果 Run Queue 长度长时间远远大于 CPU 数，表示机器负载过大，应该适当减少当前机器的作业量。

硬盘使用率

监控硬盘使用率有两个重要的目标，一是应用程序本身，如果应用程序进行了非常多的硬盘 I/O 操作，很容易推断出应用程序的性能瓶颈在于 I/O。

需要进行详细的监控才能发现应用程序的性能瓶颈在于 I/O。当应用程序没有高效地使用缓存来进行硬盘写操作时，硬盘 I/O 的数据将会非常低。但是当应用程序进行的 I/O 操作数超出了硬盘能够处理的数量，硬盘 I/O 数据将会非常高。这两种情况都需要进行调优。

在 Linux 系统中执行 iostat -xm 5 命令可以得到清单 3 中的数据：

清单 3. iostat –xm 5 命令结果一

avg-CPU: %user %nice %system %iowait %steal %idle

18.20 0.00 40.20 0.00 0.00 51.60

Device: rrqm/s wrqm/s r/s w/s rMB/s wMB/s

sda 0.00 0.20 0.00 34.60 0.10 0.23

avgrq-sz avgqu-sz await svctm %util

8.35 0.00 5.04 0.04 2.02

应用程序在向硬盘 sda 中写入数据，看上去硬盘写入的时间情况还不错，每次写入等待时间（await）为 5.04 毫秒，硬盘使用率也仅为 2.02%。但仔细来看，系统执行内核代码用掉 40.2%的时间，这是意味着应用程序中存在低效的写操作。系统每秒进行 34.60（w/s）次写操作，但是只写入了 0.23MB(wMB/s) 数据。可以判断 I/O 是应用程序的性能瓶颈所在。下一步将分析应用程序如何进行写操作。

再看另一组数据（清单 4），硬盘使用率（％util）达到 100%，等待硬盘的时间占到了 49.81％（％iowait），应用程序每秒写入 60.45mb 数据，这些数据共同证明 I/O 是应用程序的性能瓶颈所在，必须减少如此大量的 I/O 操作。

清单 4. iostat –xm 5 命令结果二

avg-CPU: %user %nice %system %iowait %steal %idle

40.20 0.00 5.70 49.81 0.00 54.10

Device: rrqm/s wrqm/s r/s w/s rMB/s wMB/s

sda 0.00 0.20 0.00 134.60 0.10 60.45

avgrq-sz avgqu-sz await svctm %util

727.24 68.46 798.04 5.67 100

监控硬盘使用率的另一个作用是得知系统是否在进行交换（swapping），计算机拥有固定数量的物理内存，但是它可以运行使用内存远远大于其物理内存的一些应用程序。应用程序占有的内存往往多于它们真正需要的内存，在这种情况下，操作系统将这些没有被用到内存挪入硬盘，当需要的时候将它们通过换页换进物理内存中。对于大多数应用程序，这种内存管理方式是不错的，但是对于服务器类型的应用程序而言，这种方式就显得尤为糟糕，因为由于 Java 内存堆的存在，服务器类型的应用往往需要用到非常多的物理内存。

由于需要将硬盘中的数据与物理内存中的数据进行交换，会严重影响系统性能。vmstat 命令的结果中 si，so 两列数据表示了换入物理内存和换出物理内存的数据量。通过这些数据可以知道系统是否在进行交换。

网络使用率

如果应用程序运行过程中使用了网络，在进行性能监控时必须监控系统网络传输使用率。网络传输类似于硬盘传输，低效地使用网络传输会造成网络带宽不足；如果网络传输的数据量超过了其所能负载的上限同样会造成网络传输性能瓶颈。

操作系统内置的网络监控工具只能获得某个网络接口接收和发送的包数和字节数。通过这些信息还不足以确定网络负载正常还是负载过大。

在 UNIX 系统中，基本的网络监控工具是 netstat。

当然，还有非常多的第三方网络监控工具，nicstat 就是 UNIX 系统中使用很广泛的一个命令行工具，通过这个工具可以得到指定网络接口的使用率。

执行 nicstat 5 命令，得到清单 5 中的数据，从数据中可以看到，网络接口 e1000g1 为 1000MB 接口，该接口使用率只有 2.98%（％Util），每秒钟通过此接口读入 156.4Kb 数据，写入 256.9Kb 数据，通过这些数据，可以明确得出网络接口的带宽以及使用率。

清单 5. nicstat 5 命令结果

Time Int rKB/s wKB/s rPk/s wPk/s rAvs wAvs %Util Sat

17:05:17 e1000g1 156.4 256.9 875.0 909.5 215.4 175.3 2.98 0.00

如果只用 netstat 命令，可以获得每秒读写的数据量，但是必须知道网络带宽并且通过额外的脚本才能计算得出网络接口的使用率。在计算过程中需要注意，带宽单位为位每秒（bps），因此 1000Mb 带宽每秒可以传输 125MB 数据。而在 nicstat 已经帮我们做了类似的计算。

网络传输无法支撑 100%的使用率，在本地以太网网络，超过 40%的使用率就被认为接口饱和。使用其他媒介进行网络传输的饱和使用率需要咨询网络架构师。Java 程序只是使用操作系统的网络接口进行传输，并不能决定网络使用率的饱和值。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#ibm-pcon)

Windows 系统资源的监控

下面将主要介绍 Windows 系统的系统监控工具 Perfmon。Perfmon 是 Windows 系统自带的性能监控工具，可以监控包括上文所述的各类系统资源的使用情况，并提供图形化的用户展示界面。Perfmon 包括性能监视器、计数器日志、跟踪日志和警报四个部分。

1. 性能监视器

在 Windows 系统的命令行中运行 perfmon.msc 命令即可启动 Perfmon 的性能监视器的用户界面。通过性能监视器，可以对 CPU、硬盘、网络的资源进行实时监控。具体的分析方式与 Linux 系统中类似，在此不再赘述。使用下面将要介绍的计数器日志可以保存这些监控数据

性能监视器的另一个功能是将计数器日志保存的数据以图形化的形式展现给用户。通过“查看当前活动”或者“查看日志数据”功能来指定监控的资源。

2. 计数器日志

虽然性能监视器可以实时监控系统资源，但并不能保存监控数据，如果需要持续对系统的监控数据采样，必须使用 Perfmon 的计数器日志功能。用户可以使用系统监视器或者其他工具对计数器日志保存的数据的进行分析。

3. 跟踪日志

通过跟踪日志功能，用户可以跟踪某些重要的系统事件，可以跟踪指定的应用程序。跟踪日志默认被保存为扩展名为.etl 二进制文件，可以使用 tracert 命令对文件进行分析，并生成 CSV 格式的 Dump 文件。

目前必须通过编辑系统注册表的形式配置跟踪的应用程序以及保存日志文件的路径。

4. 警报

当某个计数器的性能监控数据达到预先设定的阀值时，将会触发 Perfmon 的警报，警报是指预先设定的动作，如发送电子邮件、运行指定的命令等动作。也可以将警报动作设置为将警报作为系统事件记录，这样就可以在事件查看器中查看警报的内容。针对不同的应用可以指定不同的警报策略。例如当 CPU 的闲置时间低于 80%时触发警报，将发送邮件给系统维护人员；或者当内存使用率高于 90％时出发警报，执行 typeperf 命令收集保存数据。

必须是管理员用户才能使用 Perfmon，Perfmon 有两种部署方式：本地监控模式和远程监控模式。

在本地监控模式中，日志文件被默认保存在 C:\perflogs 目录中，可以在“日志文件”下修改此目录。本地监控生成的日志文件既可以在本机使用性能监视器进行分析，也可以传输到其他分析平台中分析。

在远程监控模式中，在建立监控主机与被监控主机之间信任关系并打开远程访问控制的前提下，可以对局域网内的多台目标监控机器进行集中采样监控。但随之而来的是安全隐患，所以在访问控制比较严格的环境下，远程监控模式难以实施。

在部署过程中还需考虑日志文件存储问题，设置合适的采样间隔时间，如果设置过小，日志文件会快速递增，如果设置过大，监控数据会出现较大误差。

Perfmon 有两种管理方式：控制台管理和命令行管理，如前文所述，通过运行 perfmon.msc 可以打开控制台管理器，并根据监控策略管理控制台。通过命令 Logman 可以在命令行中创建、启动、停止日志 Session。主要包括 create, start, stop, delete, query, update 这些参数，具体的使用方法参考 Logman 帮助文档。除了 Logman 命令，Typeperf 命令也是 Windows 中常用的系统功能监控命令，通过此命令可以获得 Perfmon 中所有资源当前的性能数据，但不能生成日志和设置警报。我们将 Typeperf 的输出重定向至文本文件中，使用第三方工具进行分析。Typeperf 可以配合其他性能工具一起使用，通过定制计划任务执行此命令可以定时获得系统的性能数据。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-performance-analysissy-tools/#ibm-pcon)

总结

本文在性能监测和优化方面给出了多种方法，并结合 CPU 时间特性阐述了其性能影响的因素和原因。总体上讲，性能监测首先应当从应用程序运行时所消耗的 CPU 时间来入手，剖析其运行状态，以及资源消耗的瓶颈在哪里。其次，以 CPU 使用率的提升为目标来优化代码。

性能监测还可以通过监测硬盘的使用情况来获得，大量硬盘的读写将会产生应用程序的性能问题，这就要求我们的程序设计人员在设计程序的时候尽量降低减少磁盘的读写操作，以及读写的数据量，并采用缓存交换的数据的形式提升应用程序性能。

对基于网络的应用程序，监测其网络传输的开销，也是性能监测的一种方法。大量数据在网络层面上的交换也会带来性能上的开销。

最后，本文讲解了 Java 自带的性能分析工具的使用，在使用这些工具的时候请大家铭记一点，

没有一个完美的工具来帮您完全理解应用程序的整体性能问题，实际工作中我们可能需要结合多种工具来完成一个应用程序的性能分析。不同的工具有不同的方向，全方面的理解和使用才能够更好的完成分析任务。