# Java 运行时监控，第 3 部分: 监控应用程序生态系统的性能与可用性

https://www.ibm.com/developerworks/cn/java/j-rtm3/

本文为 Java™ 应用程序运行时监控 [系列](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) 的第三部分，也是最后一部分，主要介绍在监视应用程序支持和依赖服务的性能和可用性时应使用哪些策略与技巧。所谓支持和依赖服务包括底层主机操作系统、运行数据库以及通信基础设施。文章结尾针对性能数据管理问题以及数据的报告和可视化做了论述。

[查看本系列更多内容](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) | 1https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/java/j-rtm3/#icomments)

[Nicholas Whitehead](https://www.ibm.com/developerworks/cn/java/j-rtm3/#authorN10032), 高级技术架构师, ADP

2008 年 9 月 02 日

* [expand](https://www.ibm.com/developerworks/cn/java/j-rtm3/#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-rtm3-_-sidebar)

在本系列（共三篇文章）的 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 和 [第 2 部分](http://www.ibm.com/developerworks/cn/java/j-rtm2/index.html) 中，我介绍了监控 Java 应用程序的技巧和模式，在这两部分中我把重点放在了 JVM 和应用程序类上。在这最后一期中，我将介绍从应用程序的依赖项（诸如底层操作系统、网络或者应用程序的后端数据库）收集性能与可用性数据的技巧。在文章结尾我将论述管理收集数据的模式以及报告和可视化数据的方法。

基于 Spring 的收集器

在 [第 2 部分](http://www.ibm.com/developerworks/cn/java/j-rtm2/index.html) 中，我实现了一个用于管理监控服务的基本的基于 Spring 的组件模型。该模型的基本原理及益处有：

* 使用基于 XML 的配置，使得管理大量用于配置更复杂性能数据收集器的参数集变得更加容易。
* 采用关注点分离 的结构，这样就可以使用更简单的组件，这些组件之间的相互交互可以通过注入 Spring 的依赖项来实现。
* Spring 给简单的收集 bean 提供了一个生命周期，该周期由初始化、启动 和停止 操作组成，还提供了将 Java 管理扩展（Java Management Extension，JMX）管理接口公开给 bean 的选项，这样就可以在运行时进行控制、监控和故障排除。

下面我将在本文的每个小节中介绍有关基于 Spring 的收集器的更多细节。

监控主机和操作系统

Java 应用程序总是运行于底层硬件和支持 JVM 的操作系统之上。一个全面的监控基础设施中最关键的组成就是从硬件和 OS — 通常是通过 OS 收集 — 那里收集性能、健康状况和可用性指标的能力。本节就涵盖了一些通过在 [第 1 部分](https://www.ibm.com/developerworks/cn/java/j-rtm3/URL%20HERE) 中介绍的 ITracer 类获取这类数据并一直跟踪到应用程序性能管理系统（application performance management，APM）的技巧。

典型的 OS 性能指标

下面这份摘要列出了典型指标，这些指标跨域操作系统的多个部分相关。虽然数据收集的细节迥异，而且数据的解释也必须在给定的 OS 上下文中进行，但是这些指标在大多数标准主机上基本都是等效的：

* **CPU 使用**：表示特定主机上的 CPU 的占用情况。单位通常为百分比的使用率，在较低的级别将 CPU 忙碌时间表示为消逝的时钟时间的某个特定时期的百分比。主机可以有多个 CPU，而 CPU 又可以包含多个内核，但多个内核通常都被 OS 抽象出来代表一个 CPU。例如，一个带有两个双核 CPU 的主机会被说成有四个 CPU。指标通常可以按照每个 CPU 收集或者作为总资源利用率收集，后者表示所有处理器的总体使用情况。到底是要分别监控每一个 CPU 还是监控总体 CPU，通常要取决于软件的本质及其内部架构。标准的多线程 Java 应用程序通常默认平衡所有 CPU 上的负载，所以监控总体较合适。但在某些情况下，个别 OS 进程是 “特定于” 特定 CPU 的，这时总体指标可能无法捕获到适当级别的粒度。

CPU 的使用通常被拆分成四个范畴：

* + **系统**：执行系统的或者 OS 内核级的活动耗费的处理器时间
  + **用户**：执行用户活动耗费的处理器时间
  + **I/O 等待**：处于空闲状态等待完成某个 I/O 请求耗费的处理器时间
  + **空闲**：暗指没有进行任何处理器活动

另外两个相关指标为运行队列长度（即等候 CPU 时间的请求的待处理事项）和上下文转换（即将处理器时间分配从一个进程转换到另一个进程的实例）。

* **内存**：最简单的内存指标为可用或使用中的物理内存的百分比。其他需要考虑的有虚拟内存、内存分配率和重新分配率以及表明内存有哪些区域被使用的更细粒度的指标。
* **磁盘与 I/O**：磁盘指标为每一个逻辑或物理磁盘设备的可用或使用中的磁盘空间的简单（但是至关重要的）报告，还有这些设备的读取和写入速率。
* **网络**：指网络接口上的数据传输速率和错误发生率，它通常被分为高级的网络协议范畴，如 TCP 和 IP。
* **进程与进程组**：可以说前面所述的指标都是特定主机的总活动。它们也可以划分为相同的指标，但是代表个别进程或相关进程组的消耗或活动。监控进程对资源的使用情况有助于解释主机上的每一个应用程序或者服务消耗的资源比例。有些应用程序只可以实例化一个进程；在其他情况下，一个诸如 Apache 2 Web Server 这样的服务可以实例化代表一个逻辑服务的一群进程。

代理与无代理

不同的 OS 有着不同的性能数据获取机制。我将呈现的收集数据的方式很多，但是在监控领域您可能经常要区别的是基于代理的 和无代理的 监控。也就是说在某些情况下，无需在目标主机上安装其他特定的软件也可以收集数据。但显然监控通常都会涉及到某种代理，因为监控总是需要一个接口，数据要通过它来读取。所以这里真正区别的是是使用通常出现在给定 OS 中的代理 — 诸如 Linux® 服务器上的 SSH — 还是安装其他专用于监控和使收集的数据对外部收集器可用的软件。两种方法都涉及到如下的权衡标准：

* 代理需要安装其他的软件并可能需要应用定期的维护补丁。在带有大量主机的环境中，管理软件工作不利于使用代理。
* 如果代理实际上是与应用程序相同的进程的一部分的话，哪怕它是一个单独的进程，代理进程的故障也将会蒙蔽监控。虽然主机本身仍在运行且健康状况良好，但是 APM 一定会因为无法到达代理而假定主机已停机。
* 安装在主机上的代理可能要比无代理远程监控器的数据收集能力和事件监听能力强得多。而且，报告总体指标可能需要收集好几个原始底层指标，远程收集这些指标的效率会很低。而本地的代理则能够快速地收集数据，再合计数据，然后将合计的数据提供给远程监控器。

归根结底，最佳的解决方案可能就是既实现无代理的监控又实现基于代理的监控：本地代理负责收集大多数指标，而远程监控器负责检查诸如服务器的运行情况和本地代理的状态这样的基本内容。

代理也可以有不同的选项。自治 代理按照自己的计划收集数据，反之，响应 代理按请求递送数据。此外，有些代理只将数据提供给请求程序，而有些则直接或间接地跟踪数据一直到 APM 系统。

接下来我将呈现监控带有 Linux 和 UNIX® OS 的主机的技巧。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

监控 Linux 和 UNIX 主机

监控代理可以用来实现专门的本机库以从 Linux 和 UNIX OS 收集性能数据。但是 Linux 和大多数 UNIX 变体都有很多内置数据收集工具，这些工具使得数据可以通过称为 /proc 的虚拟文件系统进行访问。该文件看起来像是普通文件系统目录里面的普通文本文件，但其实它们是常驻内存型数据结构，是通过文本文件抽取的。由于这种数据可以很容易地通过大量标准命令行的实用工具或自定义的工具来读取和解析，所以这些文件较易于使用，而且它们的输出既可以是通用的也可以是专用的。而且它们的性能也非常好，因为本质上它们是直接来源于内存的数据。

常见的用于从 /proc 中抽取性能数据的工具是 ps、sar、iostat 和 vmstat（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources) 查阅有关这些工具的参考文献）。因此，一个有效地监控 Linux 和 UNIX 主机的方法就是执行 shell 命令并解析响应。类似的监控器可以用于很多种 Linux 和 UNIX 实现；虽然它们之间都有着些许差异，但是，使用一种可以完全重用收集过程的方式格式化数据是很简单的。相反，专用的本机库可能要根据每一个 Linux 和 UNIX 发行版而进行重编码或重构（但它们正在读取的 /proc 数据有可能相同）。而编写专用于监控某一特定情况或可以标准化返回数据的格式这样的自定义 shell 命令很容易，并且开销较低。

现在我将展示几种调用 shell 命令和跟踪返回数据的方法。

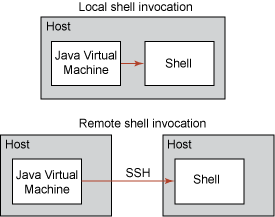
Shell 命令的执行

要在一个 Linux 主机上执行数据收集监控，就一定要调用一个 shell。它可以是 bash、csh、ksh 或其他任何允许调用目标脚本或命令并可以检索输出的、受支持的 shell。最通常的选择包括：

* **本地 shell**：如果目标主机上运行着 JVM 的话，那么线程可以通过调用 java.lang.Process 来访问这种 shell。
* **远程 Telnet 或 rsh**：这两个服务都允许调用 shell 和 shell 命令，但由于它们的安全性相对较低，所以很少使用它们。它们在大多数现代发行版上的默认状态为禁用。
* **安全 Shell（SSH）**：SSH 是最为常用的远程 shell。它提供了对 Linux shell 的完全访问，而且被公认是安全的。在文中基于 Shell 的例子里，我将主要使用该机制。大多数 OS 都提供 SSH 服务，包括所有 UNIX 系列、Microsoft® Windows®、OS/400 及 z/OS。

图 1 展示了本地 shell 与远程 shell 的基本差异：

图 1. 本地 shell 与远程 shell



要用服务器启动一个无人值守的对话需要进行一些设置。首先必须要创建一个由私钥和公钥组成的 SSH 密匙对。然后将公钥置于目标服务器，私钥置于远程监控服务器 —— 数据收集器可以在此获取该私钥。完成上述操作之后，数据收集器便能够提供私钥及其密码短语（passphrase），并能够访问目标服务器上的安全远程 shell 了。使用了密匙对之后，目标帐户的密码就是多余的了，根本不需要它。具体设置步骤如下：

1. 确保目标主机在本地的已知主机的文件中有入口。这个文件列出了已知 IP 地址或名称以及为每一个已知 IP 地址或名称验证的相关 SSH 公钥。在用户级别，该文件通常为用户主目录中的 ~/.ssh/known\_hosts 文件。
2. 用监控帐户（例如，monitoruser）连接到目标服务器。
3. 在主目录中创建一个名为　.ssh 的子目录。
4. 将目录改为 e .ssh 目录并发布 ssh-keygen -t dsa 命令。该命令提示密钥名和密码短语。然后会生成两个叫做 monitoruser\_dsa（私钥）和 monitoruser.\_dsa.pub（公钥）的文件。
5. 将私钥复制到一个安全的可访问的位置，数据收集器将从这个位置运行。
6. 用 cat monitoruser\_dsa.pub >> authorized\_keys 命令将私钥内容追加到 .ssh 目录中名为 authorized\_keys 的文件中。

清单 1 展示了我刚才所描述的过程：

清单 1. 创建一个 SSH 密匙对

whitehen@whitehen-desktop:~$ mkdir .ssh

whitehen@whitehen-desktop:~$ cd .ssh

whitehen@whitehen-desktop:~/.ssh$ ssh-keygen -t dsa

Generating public/private dsa key pair.

Enter file in which to save the key (/home/whitehen/.ssh/id\_dsa): whitehen\_dsa

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in whitehen\_dsa.

Your public key has been saved in whitehen\_dsa.pub.

The key fingerprint is:

46:cd:d4:e4:b1:28:d0:41:f3:ea:3b:8a:74:cb:57:e5 whitehen@whitehen-desktop

whitehen@whitehen-desktop:~/.ssh$ cat whitehen\_dsa.pub >> authorized\_keys

whitehen@whitehen-desktop:~/.ssh$

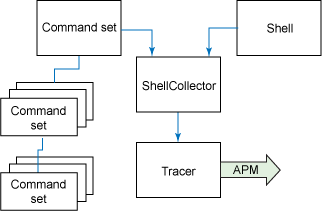
现在数据收集器已经能够使用 SSH 连接到目标 Linux 主机，该 SSH 连接名为 whitehen-desktop，它运行着 Ubuntu Linux。

这个例子的数据收集器将使用一个名为 org.runtimemonitoring.spring.collectors.shell.ShellCollector 的通用收集器类来实现。该类的一个实例将以 UbuntuDesktopRemoteShellCollector 这个名称部署在一个 Spring 上下文中。但要完成整个过程还需要一些其他的依赖项：

* 需要有一个调度器来每分钟调用一次收集器。该调度器由 java.util.concurrent.ScheduledThreadPoolExeutor 的一个实例来实现，它既可以提供一个有计划的回调机制，又可以提供一个线程池。这个调度器将以 CollectionScheduler 这个名称部署于 Spring。
* SSH shell 实现对服务器调用命令并返回结果。这个可以通过 org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell的一个实例来实现。这个类是一个名为 org.runtimemonitoring.spring.collectors.shell.IRemoteShell 的 Shell 接口的实现，它将以UbuntuDesktopRemoteShell 这个名称部署于 Spring。
* 该收集器不会硬编码一组命令及其相关解析例程，而是使用 org.runtimemonitoring.spring.collectors.shell.commands.CommandSet的一个实例，它将以 UbuntuDesktopCommandSet 这个名称部署于 Spring 中。命令集从一个 XML 文档载入，该文档表述了：
  + 将要用来执行 shell 的目标平台
  + 将要执行的命令
  + 将如何解析返回数据并将其映射到 APM 跟踪命名空间

稍候我将提供有关这些定义的更多细节。图 2 大致解释了收集器、shell 和命令集三者之间的关系：

图 2. 收集器、shell 和命令集



下面我将专门介绍一些关于专用于生成性能数据的命令以及它们的配置方法的简短示例。一个经典的例子就是 sar 命令。Linux 手册（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）对 sar 的定义是收集、报告或者保存系统活动信息。该命令非常灵活，它有超过 20 个参数，这些参数可以结合起来使用。一个简单的选择就是调用 sar -u 1 3，它报告了在三个时间间隔内（一个时间间隔为一秒）度量的 CPU 使用。清单 2 展示了它的输出：

清单 2. sar 命令的输出

whitehen@whitehen-desktop:~$ sar -u 1 3

Linux 2.6.22-14-generic (whitehen-desktop) 06/02/2008

06:53:24 PM CPU %user %nice %system %iowait %steal %idle

06:53:25 PM all 0.00 0.00 0.00 0.00 0.00 100.00

06:53:26 PM all 0.00 35.71 0.00 0.00 0.00 64.29

06:53:27 PM all 0.00 20.79 0.99 0.00 0.00 78.22

Average: all 0.00 18.73 0.33 0.00 0.00 80.94

该输出可以划分成开头、标题、三个时间间隔的数据读数和一个读数汇总平均值。这里的目标是要执行该 shell 命令、捕获输出，然后解析输出并跟踪到 APM 系统。输出数据的格式是够简单的，但却可能根据具体的版本而不同（轻微或显著的不同），而且其他 sar 选项也会返回完全不同的数据（更不用说其他的命令了，它们当然会返回不同的数据格式）。例如，清单 3 展示了一个显示活动的 socket 行为的 sar 执行：

清单 3. 显示 socket 行为的 sar

whitehen@whitehen-desktop:~$ sar -n SOCK 1

Linux 2.6.22-14-generic (whitehen-desktop) 06/02/2008

06:55:10 PM totsck tcpsck udpsck rawsck ip-frag

06:55:11 PM 453 7 9 0 0

Average: 453 7 9 0 0

因此，现在所需要的是一个解决方案：怎样在不重新编码收集器的情况下快速配置不同的数据。还可以将诸如 totsck 这样的含糊词语翻译成像 Total Used Sockets 这样的更易读的短语，以免收集到的记录会干扰 APM 系统。

在某些情况下，您可以选择以 XML 格式获取这个数据。例如，SysStat 包（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）中的 sadf 命令会以 XML 格式生成很多被经常收集的 Linux 监控数据。XML 格式增加了数据的可预测性和结构，并真正排除了分析数据、将数据映射到跟踪名称空间和解码模糊词语的需要。然而，这些工具对于您想监控的可以访问 shell 的系统可能是不可用的，因此需要一种灵活的文本解析和映射解决方案。

承接上面两个关于 sar 的应用的例子，接下来我将呈现一个设置 Spring bean 定义以监控这些数据的例子。所有引用的例子都包含在本文的示例代码中（参见 [下载](https://www.ibm.com/developerworks/cn/java/j-rtm3/#download)）。

首先，SpringCollector 实现的主要入口点为 org.runtimemonitoring.spring.collectors.SpringCollector。它采用了一个参数：Spring bean 配置文件所在的目录的名称。SpringCollector 载入了任何带有 .xml 扩展名的文件，并将他们当作 bean 描述符。该目录为位于项目根目录中的 ./spring-collectors 目录（稍后我将在本文中概述此目录中的所有文件。有多个文件可以选择，而且可以将所有的定义捆绑成一个，但要用虚构的函数单独隔开，以保持一定的顺序）。这个例子中的三个 bean 定义代表 shell 收集器、shell 和命令集。清单 4 展示了它们的描述符：

清单 4. shell 收集器、shell 与命令集的 Bean 描述符

<!-- The Collector -->

<bean id="UbuntuDesktopRemoteShellCollector"

class="org.runtimemonitoring.spring.collectors.shell.ShellCollector"

init-method="springStart">

<property name="shell" ref="UbuntuDesktopRemoteShell"/>

<property name="commandSet" ref="UbuntuDesktopCommandSet"/>

<property name="scheduler" ref="CollectionScheduler"/>

<property name="tracingNameSpace" value="Hosts,Linux,whitehen-desktop"/>

<property name="frequency" value="5000"/>

</bean>

<!-- The Shell -->

<bean id="UbuntuDesktopRemoteShell"

class="org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell"

init-method="init"

destroy-method="close">

<property name="userName" value="whitehen"/>

<property name="hostName" value="whitehen-desktop"/>

<property name="port" value="22"/>

<property name="knownHostsFile"

value="C:/Documents and Settings/whitehen/.ssh/known\_hosts"/>

<property name="privateKey"

value="C:/keys/whitehen/ubuntu-desktop/whitehen\_dsa"/>

<property name="passphrase" value="Hello World"/>

</bean>

<!-- The CommandSet -->

<bean id="UbuntuDesktopCommandSet"

class="org.runtimemonitoring.spring.collectors.shell.commands.CommandSet">

<constructor-arg type="java.net.URL"

value="file:///C:/projects//RuntimeMonitoring/commands/xml/UbuntuDesktop.xml"/>

</bean>

清单 4 中的 CommandSet 只有一个 id（UbuntuDesktopCommandSet）和另一个 XML 文件的 URL。这是因为命令集太大，我不想因为它们而使 Spring 文件显得很混乱。稍后我将描述 CommandSet。

[清单 3](https://www.ibm.com/developerworks/cn/java/j-rtm3/#listing3) 中的第一个 bean 为 UbuntuDesktopRemoteShellCollector。它的 bean id 值可以是任意的描述性的值，但是当从另一个 bean 引用该 bean 时需要保持一致。这个例子中的类为 org.runtimemonitoring.spring.collectors.shell.ShellCollector，它是一个通过类似于 Shell 的接口来收集数据的通用类。其他重要属性有：

* **shell**：收集器用来从 shell 命令调用和检索数据的 shell 类的实例。Spring 用 UbuntuDesktopCommandSet 的 bean id 来注入该 Shell 的实例。
* **commandSet**：代表一组命令和相关解析、跟踪名称空间映射指令的 CommandSet 实例。Spring 用 UbuntuDesktopRemoteShell 的 bean id注入该命令集的示例。
* **scheduler**：一个调度线程池的引用，该线程池管理数据收集的调度，将这项工作具体分配给一个线程来完成。
* **tracingNameSpace**：跟踪名称空间的前缀，它控制着这些指标将被跟踪到 APM 树中的哪个位置。
* **frequency**：数据收集频率，以毫秒为单位。

[清单 4](https://www.ibm.com/developerworks/cn/java/j-rtm3/#listing4) 中的第二个 bean 为 shell，它是一个名为 org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell 的 SSH shell 的实现。该类使用从 JCraft.com（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）下载的 JSch 来实现。它的其他重要属性有：

* **userName**：用户用来连接到 Linux 服务器的名称
* **hostName**：连接到的 Linux 服务器的名称（或 IP 地址）。
* **port**：Linux 服务器端口，sshd 在这个端口上监听。
* **knownHostFile**：一个包含主机名称和 SSH 服务器的 SSH 证书的文件，该 SSH 服务器对于运行 SSH 客户机的本地主机是 “已知的”（有趣的是，SSH 中的这个安全机制恰好颠倒了传统的安全结构，使用这种机制，除非主机是 “已知的” 并可以给出匹配的证书，否则客户机不会信任主机，并拒绝连接）。
* **privateKey**：用来验证 SSH 服务器的 SSH 私钥文件。
* **passPhrase**：用来解锁私钥的密码短语。它的外表与密码类似，只是它没有被传送到服务器，而且它只用于本地解密私钥。

清单 5 展示了 CommandSet 的内部细节：

清单 5. CommandSet 内部细节

<CommandSet name="UbuntuDesktop">

<Commands>

<Command>

<shellCommand>sar -u 1</shellCommand>

<paragraphSplitter>\n\n</paragraphSplitter>

<Extractors>

<CommandResultExtract>

<paragraph id="1" name="CPU Utilization"/>

<columns entryName="1" values="2-7" offset="1">

<remove>PM</remove>

</columns>

<tracers default="SINT"/>

<filterLine>Average:.\*</filterLine>

<lineSplit>\n</lineSplit>

</CommandResultExtract>

</Extractors>

</Command>

<Command>

<shellCommand>sar -n SOCK 1</shellCommand>

<paragraphSplitter>\n\n</paragraphSplitter>

<Extractors>

<CommandResultExtract>

<paragraph id="1" name="Socket Activity"/>

<columns values="1-5" offset="1">

<remove>PM</remove>

<namemapping from="ip-frag" to="IP Fragments"/>

<namemapping from="rawsck" to="Raw Sockets"/>

<namemapping from="tcpsck" to="TCP Sockets"/>

<namemapping from="totsck" to="Total Sockets"/>

<namemapping from="udpsck" to="UDP Sockets"/>

</columns>

<tracers default="SINT"/>

<filterLine>Average:.\*</filterLine>

<lineSplit>\n</lineSplit>

</CommandResultExtract>

</Extractors>

</Command>

</Commands>

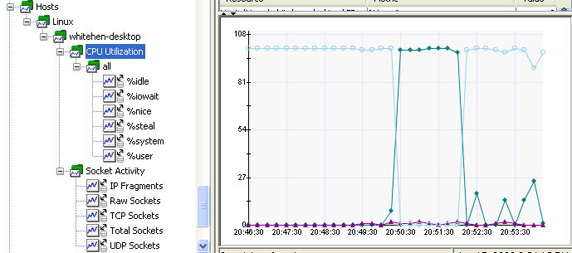
</CommandSet>

CommandSet 负责管理 shell 命令和解析指令。由于每一个 Linux 或 UNIX 系统的输出 —— 即便是对相同命令的输出 —— 都会有些不同，因此每一种类型的受监控的主机通常都会有一个对应的 CommandSet。要使用 XML 详述 CommandSet 背后的每一个选项，可能要占用很长的篇幅，因为它一直在不断演变，而且会根据情况的不同而改变，因此我只简短的概述一下其中一些标记，内容如下：

* **<shellCommand>**：定义将要传给 shell 的实际命令。
* **<paragraphSplitter>**：有些命令，或组成多个命令链的命令，可能会返回多个文本片段。这些文本片段被称为段落。正则表达式（regex）在这里指定了划分段落的标准。命令对象将结果分成多个段落，并将所需段落传递到底层提取器。
* **<Extractors>** 和其中包含的 **<CommandResultExtract>** 标记：这些结构定义解析和映射。
* **<paragraph>**：提取器使用 id 属性中的基于零的索引来定义它想要从结果中抽取的段落，所有从该段落中跟踪的指标都被归入到以该段落名定义的跟踪名称空间中。
* **<columns>**：如果定义了 entryName 的话，那么每一行中编入索引的列都会被添加到跟踪名称空间。这是针对左侧列包含一个指标分界的情况而言的。例如，sar 的一个选项将会分别为每一个 CPU 报告 CPU 使用，CPU 编号列于第二列中。在 [清单 5](https://www.ibm.com/developerworks/cn/java/j-rtm3/#listing5) 中，entryName 提取出 all 限定符，该限定符表明报告为所有 CPU 的总体汇总。values 属性代表每一行中需要被跟踪的行，而 offset 负责保持数据行中的列数和相应标题间的平衡。
* **<tracers>**：它定义默认跟踪类型，并允许为与指定的标题或 entryName 有关的值定义不同的跟踪器类型。
* **<filterLine>**：如果定义了它，regex 会忽略整行文本不匹配的数据行。
* **<lineSplit>**：它定义用于解析每一个段落中各行的分隔 regex。

图 3 展示了这个例子的 APM 树：

图 3. Ubuntu Desktop 监控的 APM 树



如果不喜欢这个树的外观的话，您还有其他的选择。发送到服务器的命令很容易修改，可以修改它们使其传递一连串的 grep、awk 和 sed 命令，以此来将数据重新格式化为很少需要解析的格式。例如，参见清单 6：

清单 6. 格式化命令内部的命令输出

whitehen@whitehen-desktop:~$ sar -u 1 | grep Average | \

awk '{print "SINT/User:"$3"/System:"$5"/IOWait:"$6}'

SINT/User:34.00/System:66.00/IOWait:0.00

另外一个可以提供最佳配置、灵活性和性能的选择就是使用动态脚本，这种方法在其他格式化工具不可用或输出格式极其笨拙的情况下尤为适用。在接下来的例子中，我配置了一个 Telnet shell 用以从 Cisco CSS 负载均衡器中收集负载均衡状态数据。输出格式和内容对于任何种类的标准化解析来说都是很重要的问题，而这个 shell 支持的命令有限。清单 7 展示了命令的输出：

清单 7. CSS Telnet 命令的输出

Service Name State Conn Weight Avg State

Load Transitions

ecommerce1\_ssl Alive 0 1 255 0

ecommerce2\_ssl Down 0 1 255 0

admin1\_ssl Alive 0 1 2 2982

admin2\_ssl Down 0 1 255 0

clientweb\_ssl Alive 0 1 255 0

XML 元素与属性

有些应用程序明显偏好使用 XML 元素而非属性，而另一些则偏好使用属性。我没有什么强烈的偏好，但我尽量通过如下两个经验法则保持一致性：

* 简短的、固定的命名值易于读取。
* 在值文本很长或包含必须转义的无效字符（诸如 & 或"）的情况下，我使用元素，因为数据可以包装在CDATA 块中，该块告诉 XML 解析器不要解析这样的内容。

清单 8 展示了用于执行和解析命令的命令集。注意 <preFormatter beanName="FormatCSSServiceResult"/> 标记。它引用了一个包含几行 Groovy 脚本的 Spring bean。Telnet shell 命令的原始输出被传给 Groovy 脚本，然后返回值以一种更友好的格式被传给命令数据提取器。还要注意的是，为了标记为 Status 的列中的值，跟踪器类型被覆盖成了 STRING 类型。眼光尖锐的读者将会注意到这个列不存在，但是 Groovy 脚本的一部分工作就是解决两个列均使用 State 名的问题（您知道这其中的原委），所以 Groovy 脚本将第一个列重命名为 Status。

清单 8. CSS CommandSet

<CommandSet name="CiscoCSS">

<Commands>

<Command>

<shellCommand>show service summary</shellCommand>

<paragraphSplitter>\n\n\n\n</paragraphSplitter>

<preFormatter beanName="FormatCSSServiceResult"/>

<Extractors>

<CommandResultExtract>

<paragraph id="0" name="Service Summary" header="true"/>

<columns entryName="0" values="1-5" offset="0"/>

<tracers default="SINT">

<tracer type="STRING">Status</tracer>

</tracers>

<lineSplit>\n</lineSplit>

</CommandResultExtract>

</Extractors>

</Command>

</Commands>

</CommandSet>

Groovy bean 的益处有很多。首先，它的脚本是动态配置的，所以可以在运行时更改它。其次，该 bean 可以检测出源发生了变更，并会在下一次调用它时调用 Groovy 编译器，所以它的性能是足够好的。此外，此种语言含有丰富的解析功能，且容易编写。 清单 9 展示了包含内联源代码文本的 Groovy bean：

清单 9. Groovy 格式化 bean

<bean id="FormatCSSServiceResult"

class="org.runtimemonitoring.spring.groovy.GroovyScriptManager"

init-method="init" lazy-init="false">

<property name="sourceCode"><value><![CDATA[

String[] lines = formatTarget.toString().split("\r\r\n");

StringBuffer buff = new StringBuffer();

lines.each() {

if(!(

it.contains("Load Transitions") ||

it.contains("show service summary") ||

it.trim().length() < 1)) {

buff.append(it).append('\n');

}

}

return buff.toString()

.replaceFirst("State", "Status")

.replaceFirst("Service Name", "ServiceName")

.replace("State", "Transitions");

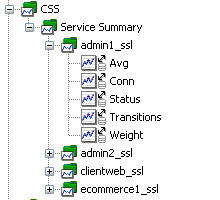
]]></value>

</property>

</bean>

图 4 展示了 CSS 监控的 APM 指标树

图 4. CSS 监控的 APM 树



SSH 连接

最后一个要考虑的有关 Linux/UNIX shell 收集的问题就是 SSH 连接的问题了。所有 Shell 类的基本接口为org.runtimemonitoring.spring.collectors.shell.IShell。它定义了一个名为 issueOSCommand() 的方法的两种变体，命令在这个方法中被作为参数传递并且返回结果。在我的使用远程 SSH 类 org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell 的例子中，底层 shell 调用建立在 Apache Ant 中 SSHEXEC 任务的实现的基础上（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。这种方法的优点在于它很简单，但是它有一个不可避免的缺点：要为每一个发出的命令创建一个新的连接。这显然会降低效率。一个远程的 shell 只可以每分钟轮询一次，但每一个轮询周期可以执行几个命令来获取监控数据的适当范围。问题是要在监控时窗期间（跨多个轮询周期）保持开放会话是很难的。它需要更详细地检查和解析返回数据，提供不同的 shell 类型以及不断显示 shell 提示，当然 shell 提示不包括在预期返回值中。

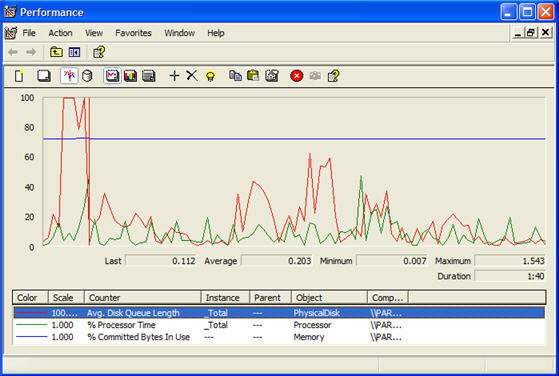
目前为止，我一直努力处理长期存活的会话 shell 实现。另外一个选择就是折衷：即为每一个轮询周期模式保留一个连接，但同时试着用一个命令捕获所有数据。这个可以通过追加命令或者（在某些情况下）通过对一个命令使用多个选项来实现。例如，我的 SuSE Linux 服务器上的 sar版本拥有一个 -A 选项，该选项返回一个包含 sar 支持的所有指标的示例；该命令与 sar -bBcdqrRuvwWy -I SUM -n FULL -P ALL 等效。返回的数据将拥有多个段落，但是用一个命令集来解析它应该没有问题。要查看这样的例子，请参见本文中名为 Suse9LinuxEnterpriseServer.xml 的示例代码中的命令集定义（参见 [下载](https://www.ibm.com/developerworks/cn/java/j-rtm3/#download)）。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

监控时窗

Microsoft Windows 与 Linux/UNIX 之间的本质差别必然导致性能数据收集也迥然各异。Windows 本身几乎没有可以提供丰富的性能报告数据的命令行工具。性能数据也无法通过像相对简单的 /proc 文件系统这样的东西来获取。Windows Performance Manager（WPM）— 也称为 SysMon、System Monitor 或者 Performance Monitor — 是从 Windows 主机获取性能度量的标准接口。它的功能很强大而且拥有大量有用的指标。此外，很多基于 Windows 的软件包都通过 WPM 发布了它们自己的指标。Windows 还通过 WPM 提供了制图、报告和警报设施. 图 5 展示了一个 WPM 实例的屏幕截图：

图 5. Windows Performance Manager



WPM 管理一组性能计数器，即引用了某一具体指标的复合命名的对象。组成复合名称的有：

* **性能对象**：性能指标的广义范畴，如：处理器 或内存。
* **实例**：当有多个可能成员时，某些性能对象要用一个实例来划分。例如，处理器既有代表单个 CPU 的实例，又有总实例。相反，内存则是一个 “纯” 性能对象，这是因为内存只有一个表现形式。
* **计数器**：实例（如果可用的话）以及性能对象内部的指标的粒度名。例如，处理器实例 0 有一个名为 % Idle Time 的计数器。

根据这些名称片段，表达这些对象的命名约定和语法为：

* **带实例：** \性能对象（实例名）\计数器名
* **不带实例：** \性能对象\计数器名

WPM 最大的缺点就是它访问这个数据有些困难，尤其是远程获取，如果是非 Windows 平台的话，那么困难更大。我将呈现很多使用基于ITracer 的收集器捕获 WPM 数据的方法。下面总结了一些主要方法：

* **读取日志文件**：可以配置 WPM 以将所有收集的指标记录在日志文件中，然后就可以读取、解析和跟踪该日志文件。
* **数据库查询**：可以配置 WPM 以将所有收集的指标记录在 SQL Server 数据库中，收集到的指标可以在这个数据库中被读取、解析和跟踪。
* **Win32 API**：使用 Win32 API（.NET、C++、Visual Basic 等）编写的客户机可以使用 WPM 的 API 直接连接到 WPM。
* **自定义代理**：自定义代理可以安装在目标 Windows 服务器上，该目标 Windows 服务器要能够为外部请求 —— 请求非 Windows 客户机中的 WPM 数据 —— 充当代理服务器。
* **简单网络管理协议（Simple Network Management Protocol，SNMP）**：SNMP 是一个代理的一个实例，该代理更加强调它对设备、主机等的监控能力。稍后我将在本文中论述 SNMP。
* **WinRM**：WinRM 是 WS-Management 规范的 Windows 实现，它概述了如何使用 Web 服务来管理系统。由于 Web 服务是独立于语言和平台的，因此 WinRM 必然会给非 Windows 客户机提供 WPM 指标访问。虽然 WinRM 可以被当作是代理的另一种形式，但它将成为 Windows 2008 的标准配置，这将把它推向无代理解决方案的舞台。最有趣的是，Java Specification Request 262（Web Services Connector for JMX Agent）承诺要与基于 Windows 的、WS-Management 服务直接交互。

在接下来的例子中，我将使用本地 Windows shell 和代理实现呈现一个理论上的概念验证。

本地 Windows shell

作为一个简单的概念验证，我已经创建了一个可使用 C# 执行的名为 winsar.exe 的 Windows 命令行。它的用途是提供一些与 Linux/UNIX sar命令相同的对性能统计的命令行访问。使用该命令行的语法很简单：winsar.exe CategoryCounterRawInstance.

实例名是强制使用的，除非计数器不是实例计数器，非实例计数器名称可以全是（\*）。计数器名也是强制使用的，但也可以全是（\*）。Raw是 true 或者 false。清单 10 展示了使用基于实例的计数器和非基于实例的计数器的例子：

清单 10. 使用了非基于实例的和基于实例的计数器的 winsar

C:\NetProjects\WinSar\bin\Debug>winsar Memory "% Committed Bytes In Use" false

%-Committed-Bytes-In-Use

79.57401

C:\NetProjects\WinSar\bin\Debug>winsar LogicalDisk "Current Disk Queue Length" false C:

Current-Disk-Queue-Length

C: 2

因为我的目的是重建某种类似 sar 的东西，数据是以杂乱无章的（非格式化的）表格形式输出的，因此可以使用标准 shell 命令集来解析这些数据输出。对于基于实例的计数器来说，实例位于数据行的第一列，计数器名位于标题行。而对于非基于实例的计数器来说，数据行的第一个字段中无名称。为了能够清晰解析，任何带有空格的名称都填充 “-” 字符。这样做虽然会很难看但却可以使解析比较容易。

为这些统计信息（为进行演示而进行了删减）设置一个收集器是相当简单的。shell 实现是一个org.runtimemonitoring.spring.collectors.shell.local.WindowsShell，命令集引用了 winsar.exe 和参数。shell 还可以被实现为使用 SSH 的远程 shell，要使用 SSH 需要在目标 Windows 主机上安装一个 SSH 服务器。然而，这个解决方案的效率非常低，这主要的原因是由于该实现是基于 .NET 的；在如此短暂的时间内反复启动 Common Language Runtime（CLR）是一种低效的实践。

另一个可能的解决方案是用原生 C++ 重写 winsar。这个就交给 Windows 编程专家来实现吧。.NET 解决方案的效率是可以提高的，但是程序必须要作为后台进程而一直运行，还要以其他某种方式提供对 WPM 数据的请求，并不会在每一个请求结束后终止。为了达到这个目标，我实现了 winsar 中的第二个选项，在这个选项中由 -service 参数来启动程序、读入一个名为 winsar.exe.config 的配置文件并通过 Java Message Service（JMS）监听请求。除了个别项外，文件的内容非常清楚明了。jmsAssembly 项指一个 .NET 汇编的名称，该 .NET 汇编包含着 JBoss 4.2.2 客户机库（它提供 JMS 功能）的一个 .NET 版本。这个汇编是使用 IKVM（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）创建的。respondTopic 引用了公共主题的名称，响应是在该公共主题中发布的，而不是使用私有主题来发布，因此其他监听器也可以接收到数据。commandSet 是一个命令集的引用，该命令集可供一般接收者用来解析和跟踪数据。清单 11 展示了 winsar.exe.config 文件：

清单 11. winsar.exe.config 文件

<configuration>

<appSettings>

<add key="java.naming.factory.initial"

value="org.jnp.interfaces.NamingContextFactory"/>

<add key="java.naming.factory.url.pkgs"

value="org.jboss.naming:org.jnp.interfaces"/>

<add key="java.naming.provider.url" value="10.19.38.128:1099"/>

<add key="connectionFactory" value="ConnectionFactory"/>

<add key="listenTopic" value="topic/StatRequest"/>

<add key="respondTopic" value="topic/StatResponse"/>

<add key="jmsAssembly" value="JBossClient422g" />

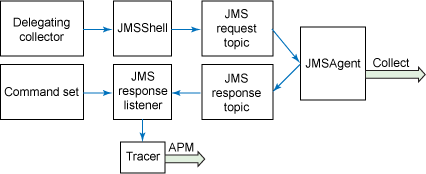
<add key="commandSet" value="WindowsServiceCommandSet" />

</appSettings>

</configuration>

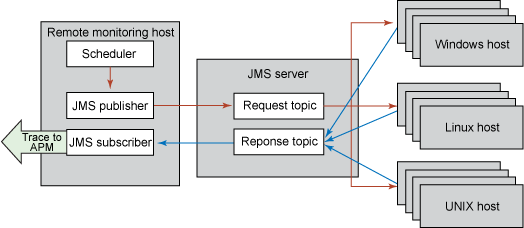
在 Spring 中实现这个收集器以使用该服务在概念上和设置 shell 是一回事。事实上，收集器本身就是org.runtimemonitoring.spring.collectors.shell.ShellCollector 的一个名为org.runtimemonitoring.spring.collectors.shell.DelegatingShellCollector 的扩展。不同的是这个 shell 扮演着一个普通收集器的角色，并发出对数据的请求，但是数据是通过 JMS 接收的，通过另一个组件解析和跟踪的。被实现的名为org.runtimemonitoring.spring.collectors.shell.jms.RemoteJMSShell 的 Shell 在行为上类似 shell，但它通过 JMS 分配命令，如图 6 所示：

图 6. 委托收集器



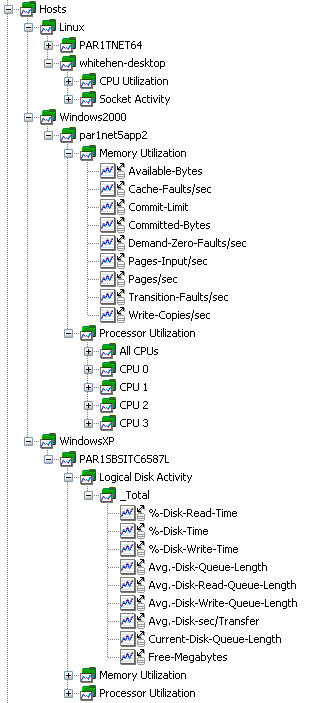
由于这种策略看起来非常适合全面部署代理，所以相同的基于 JMS 的代理是用 Java 代码实现的，它可以部署于任何支持 JVM 的 OS 上。图 7 展示了一个 JMS 发布/订阅性能数据收集系统：

图 7. JMS 发布/订阅监控



另外 JMS 代理发挥功能的方式也不同。本例中解释的模式展示的是目标主机上的一个请求监听 代理，因为这些代理在启动后不会执行任何操作，直到它们从中央监控系统接收到一个请求才执行。然而，这些代理可以按照自己的计划自主地 收集数据并将它发布到相同的 JMS 服务器。监听代理的优势有两个：第一，可以配置收集参数并保存在一个中央位置，而不用发送到每一个目标主机。第二（虽然在本例中没有实现），由于中央请求监听器发出请求，所以监听器可以在特定的已知服务器没有响应的情况下触发一个报警条件。图 8 显示了组合服务器的 APM 树：

图 8. Windows 和 Linux 服务器的 APM 树



实现 NSClient

NSClient 和 NC\_Net 使用几乎相同的套接字协议来从 Windows 服务器请求数据。nsclient4j（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）是一个 Java API，它包装了这个协议并使 WPM 指标对于任何带有 JVM 的系统可用。本文的示例代码中有一个 nsclient4j Spring 收集器的例子，它位于 /conf/spring-collectors/nsclientj.xml 目录中。

winsar 是一个简单的、早期的原型，它有几个缺点，包括：

* 通过编程访问一些 WPM 计数器（如处理器对象）会生成空的或原始的指标。所以诸如 CPU % Utilization 这样的指标是可以直接读取的。所需要解决的是如何在规定的时间段内多次读取这些指标，然后就可以计算出 CPU 使用了。winsar 没有这个功能，但是诸如 NSClient 和 NC\_Net（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）这样的类似代理却可以提供这个功能：
* 诚然，将 JMS 作为远程代理的传输机制有它的妙处，但是也有局限。NSClient 和 NC\_Net both 使用低级但简单的套接字协议来请求和接收数据。这些服务的本来用途之一就是要将 Windows 数据提供给 Nagios，这是一个几乎排斥 Linux 平台的网络监控系统，因此来自客户端的图像中实际上可能没有 Win32 API。

最后，就像我在前面提到的，SpringCollector 应用程序用一个参数引导，它是一个含有配置 XML 文件的目录。这个目录为 /conf/spring-collectors，它位于示例代码包的根目录中。前面的例子中使用的具体文件为：

* **shell-collectors.jmx**：它包含所有 shell 收集器的定义。
* **management.xml**：它包含 JMX 管理 bean 和收集调度器。
* **commandsets.xml**：它包含 shell 收集器的命令集的定义。这些定义引用了 /commands 中的外部 XML 文件。
* **shells.xml**：它包含所有 shell 的定义。
* **jms.xml**：它包含 JMS 连接工厂和主题的定义，还有 Java Naming 和 Directory Interface（JNDI）上下文。
* **groovy.xml**：它包含 Groovy 格式化程序 bean。

我对 OS 监控的论述就到此为止了。接下来，我将介绍数据库系统的监控。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

用 JDBC 监控数据库系统

我经常遇到这种情况，就是认为 DBA 以及他们的工具和应用程序的职责仅仅是监控数据库。然而，要为性能和可用性数据实现非孤立的、集中式 APM 储存库，需要在 DBA 的工作中再补充 一些监视整合的 APM 的工作。我将展示一些用名为 JDBCCollector 的 Spring 收集器来收集数据（这些数据在某种情况下很可能没有受到监控，而且是非常有用的指标）的技术。

您应该考虑的数据收集的广义范畴为：

* **简单的可用性和响应时间**：这是一个简单的机制，可以用它来定期连接到数据库、发布一个简单的查询并跟踪响应时间，或者在连接失败时跟踪服务器停机指标。连接失败不一定表明数据库正在经历硬停机，但它至少可以肯定应用程序一端的通信有问题。孤立的数据库监控可能永远无法指示出数据库连通性问题，但是记住这个还是很有用的，因为可以从那里 连接到一个服务并不意味着也可以从这里 连接。
* **上下文数据**：回顾一下 [第 1 部分](https://www.ibm.com/developerworks/cn/java/j-rtm3/URL%20HERE) 中的上下文跟踪 的概念，您可以从应用程序数据的周期性采样获得一些有用的信息。在很多情况下，数据库中的数据活动模式和应用程序设施的行为或性能有着很大的联系。
* **数据库性能表**：很多数据库以表或视图的形式，抑或是通过储存过程来公开内部性能和载入指标。同样，数据可以轻松地通过 JDBC 访问。这个领域显然覆盖了传统 DBA 监控，但是数据库性能通常与应用程序性能的联系非常紧密，以至于如果收集两组指标却不通过一个整合系统来关联它们的话，这简直就是一种巨大的浪费。

JDBCCollector 非常简单。基本上，它由一个查询和一串映射语句组成，它定义了将查询的结果映射到跟踪名称空间的方式。思考一下清单 12 中的 SQL：

清单 12. 一个简单的 SQL 查询

SELECT schemaname, relname,

SUM(seq\_scan) as seq\_scan,

SUM(seq\_tup\_read) as seq\_tup\_read

FROM pg\_stat\_all\_tables

where schemaname = 'public'

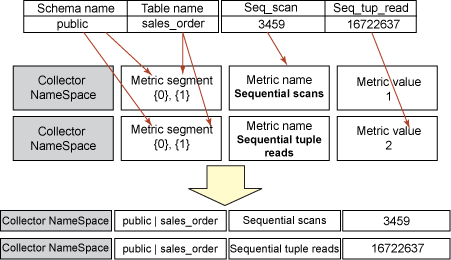
and seq\_scan > 0

group by schemaname, relname

order by seq\_tup\_read desc

该查询选择一个表格中的四个列。我想将每一个从查询返回的行映射到一个跟踪名称空间，这个名称空间由每行的一部分数据组成。切记名称空间是由片断名称加上指标名称组成的。使用字母值和/或行标记来指定这些值也就定义了映射。行标记代表编号列中的值，如 {2}。处理片断和指标名称时，字母值保持原样不变，而标记在查询结果中使用当前行的各个列值动态替换，如图 9 所示：

图 9. JDBCCollector 映射



在图 9 中，我呈现的是对查询的一行响应，但每为每一个返回的行定义一次映射，映射过程就会发生一次。片断的值是 {1}、{2}，所以跟踪名称空间的片段的部分是 {"public", "sales\_order"}。指标名是字母值，因而保持不变，指标值在第一次映射时被定义为 1，在第二次映射时被定义为 2，它们分别代表着 3459 和 16722637。具体的实现该过程便可以进一步澄清它。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

使用 JDBC 的上下文跟踪

运行的数据库中的应用程序数据也许包含有用的且有趣的上下文数据。应用程序数据本身并不一定是与性能相关的，但如果抽样这些数据，把它与表示 Java 类的性能、JVM 的健康状况和服务器性能统计信息的历史指标联系起来的话，就可以清楚地了解系统在某一具体的时间段内到底在做些什么。假想您现在正在监控一个极其繁忙的电子商务 Web 站点。您的客户所发出的订单被记录在一个名为 sales\_order 的表中，该表还记录了一个惟一的 ID 和发出订单的时间戳。你可以通过抽样在最后 n 分钟内输入的记录数来得出提交订单的速率。

这里是 ITracer 的增量（delta）功能的又一个用武之地，因为我可以设置 JDBCCollector 来从某个特定时间点开始查询行数，并将这个值作为一个增量而跟踪它。结果会得到一个可以描述您的网站有多繁忙的指标（或许还有很多其他指标）。这个指标也会成为一个颇有价值的历史参考。例如，如果知道输入的订单数达到每个周期 50 个时数据库的速度就会减慢的话，那么将会很有用。硬性的、特定的经验数据可以简化容量和增长情况的规划。

接下来我将要实现这个例子。JDBCCollector 使用与前面的例子相同的调度器 bean，它还有一个与我在 [第 2 部分](https://www.ibm.com/developerworks/cn/java/j-rtm3/URL%20HERE) 中涉及到的完全相同的 JDBC DataSource。这些数据库收集器是在 /conf/spring-collectors/jdbc-collectors.xml 文件中定义的。清单 13 展示了第一个收集器：

清单 13. 订单执行速率收集器

<bean name="OrderFulfilmentRateLast5s"

class="org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector"

init-method="springStart">

<property name="dataSource" ref="RuntimeDataSource" />

<property name="scheduler" ref="CollectionScheduler" />

<property name="query">

<value><![CDATA[

select count(\*) from sales\_order

where order\_date > ? and order\_date < ?

]]></value>

</property>

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="Database Context" />

<property name="objectName"

value="org.runtime.db:name=OrderFulfilmentRateLast5s,type=JDBCCollector" />

<property name="frequency" value="5000" />

<property name="binds">

<map>

<entry><key><value>1</value></key><ref bean="RelativeTime"/></entry>

<entry><key><value>2</value></key><ref bean="CurrentTime"/></entry>

</map>

</property>

<property name="queryMaps">

<set>

<bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">

<property name="valueColumn" value="0"/>

<property name="segments" value="Sales Order Activity"/>

<property name="metricName" value="Order Rate"/>

<property name="metricType" value="SINT"/>

</bean>

</set>

</property>

</bean>

<bean name="RelativeTime"

class="org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider">

<property name="period" value="-5000" />

</bean>

<bean name="CurrentTime"

class="org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider">

<property name="period" value="10" />

</bean>

这个例子中的收集器 bean 名为 OrderFulfilmentRateLast5s，类为 org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector。标准 scheduler 收集器被注入，它被当作 JDBC DataSource 的一个引用，RuntimeDataSource。query 定义要执行的 SQL。SQL 查询既可以把字母值用作参数也可以使用绑定变量（就像在这个例子中一样）。这个例子是人为构造的，因为 order\_date 的两个值很容易用 SQL 语法来表示，但是通常情况下，只有需要提供某些外部值时才会用到绑定变量。

为了提供绑定外部值的能力，我需要先实现 org.runtimemonitoring.spring.collectors.jdbc.IBindVariableProvider 接口，然后再将该类作为一个 Spring 管理的 bean 来实现。在这个例子中，我使用了org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider 的两个实例，还有一个通过被传递的 period 属性来提供当前时间戳偏移量的 bean。这些 bean 是 RelativeTime，它返回当前时间减去 5 秒得出的值，还有 CurrentTime，它返回 “现在” 时间加上 10 毫秒的值。这些 bean 的引用通过 binds 属性（一个映射）注入到收集器 bean。映射中的每一个入口值都要与要使用该映射的 SQL 语句中的绑定变量相匹配，这一点至关重要，如若不然就会发生错误或意外的结果。

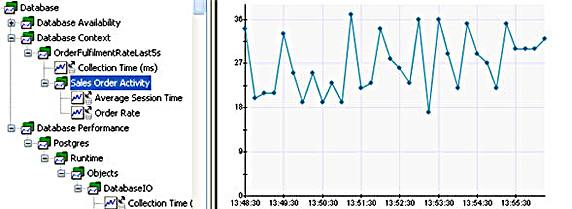
实际上，我利用了这些绑定变量来获取在大约最后 5 秒钟输入系统的销售订单数。这需要对生产表格执行大量查询，因此要适当调节收集的频率和时间窗口（即，Relative 的时间段）以避免在数据库上实现不恰当的负载。为了帮助正确地调节这些设置，收集器跟踪收集时间一直到 APM 系统，所以运行时间可以用来度量查询开销。更高级的收集器实现会减缓收集的频率，因为监控查询的运行时间增加了。

上面所呈现的映射是通过 queryMaps 属性定义的，该定义使用了一个 org.runtimemonitoring.spring.collectors.jdbc.QueryMap 类型的内部 bean。它有四个简单的属性：

* **valueColumn**：应该作为跟踪值而绑定的每一行中的列的索引，该索引是基于零的。在这个例子中，我绑定了 count(\*) 的值。
* **segments**：跟踪名称空间片断，它被定义为一个单个字母。
* **metricName**：指标名称的跟踪名称空间，同样被定义成一个字母。
* **metricType**：ITracer 指标类型，它被定义为一个 sticky int。

如果想从每一个执行的查询跟踪多个值，收集器允许为每一个收集器定义多个 queryMap。接下来我将向您展示的例子会使用 rowToken 把返回数据的值注入跟踪名称空间，但是现在的例子使用的是字母值。但是，要设计一个使用相同查询的例子的话，我会将查询改为 select count(\*), 'Sales Order Activity', 'Order Rate' from sales\_order where order\_date > ? and order\_date < ?。这使得我所期望的片断和指标名称返回于查询中。要想映射它们，我可以将 segments 配置为 {1}，将 metricName 设置为 {2}。在某些扩展的情况中，metricType 甚至可能来源于数据库，而且值也可以用 rowToken 来表示。 图 10 显示了这些收集的指标的 APM 树：

图 10. 销售订单速率监控



数据库性能监控

JDBCCollector 可以使用相同的进程来从数据库性能视图中获取和跟踪性能数据。在这个使用了 PostgreSQL 的例子中，这些表 — 称为统计视图— 的名称的前缀为 pg\_stat。很多其他的数据库也都拥有类似的视图，并可以使用 JDBC 来访问。在这个例子中，我将使用同一个繁忙的电子商务网站并设置一个 JDBCCollector 来监控最为繁忙的 5 个表上的表格和索引活动。具体的 SQL 展示在清单 14 中：

清单 14. 表格与索引活动监控器

<bean name="DatabaseIO"

class="org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector"

init-method="springStart">

<property name="dataSource" ref="RuntimeDataSource" />

<property name="scheduler" ref="CollectionScheduler" />

<property name="availabilityNameSpace"

value="Database,Database Availability,Postgres,Runtime" />

<property name="query">

<value><![CDATA[

SELECT schemaname, relname, SUM(seq\_scan) as seq\_scan,

SUM(seq\_tup\_read) as seq\_tup\_read,

SUM(idx\_scan) as idx\_scan, SUM(idx\_tup\_fetch) as idx\_tup\_fetch,

COALESCE(idx\_tup\_fetch,0) + seq\_tup\_read

+ seq\_scan + COALESCE(idx\_scan, 0) as total

FROM pg\_stat\_all\_tables

where schemaname = 'public'

and (COALESCE(idx\_tup\_fetch,0) + seq\_tup\_read

+ seq\_scan + COALESCE(idx\_scan, 0)) > 0

group by schemaname, relname, idx\_tup\_fetch,

seq\_tup\_read, seq\_scan, idx\_scan

order by total desc

LIMIT 5 ]]>

</value>

</property>

<property name="tracingNameSpace"

value="Database,Database Performance,Postgres,Runtime,Objects,{beanName}"

/>

<property name="frequency" value="20000" />

<property name="queryMaps">

<set>

<bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">

<property name="valueColumn" value="2"/>

<property name="segments" value="{0},{1}"/>

<property name="metricName" value="SequentialScans"/>

<property name="metricType" value="SDLONG"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">

<property name="valueColumn" value="3"/>

<property name="segments" value="{0},{1}"/>

<property name="metricName" value="SequentialTupleReads"/>

<property name="metricType" value="SDLONG"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">

<property name="valueColumn" value="4"/>

<property name="segments" value="{0},{1}"/>

<property name="metricName" value="IndexScans"/>

<property name="metricType" value="SDLONG"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">

<property name="valueColumn" value="5"/>

<property name="segments" value="{0},{1}"/>

<property name="metricName" value="IndexTupleReads"/>

<property name="metricType" value="SDLONG"/>

</bean>

</set>

</property>

</bean>

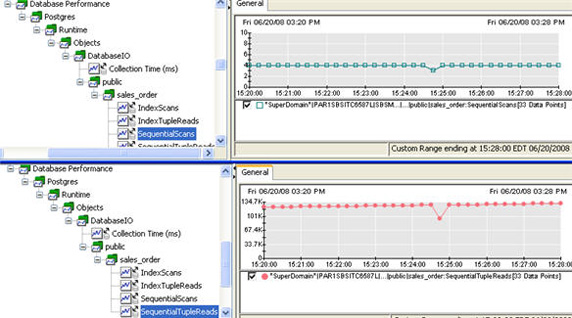
查询每隔 20 分钟为最繁忙的 5 个表检索如下的值：

* 数据库模式的名称
* 表的名称
* 顺序扫描的总数
* 顺序扫描检索出的元组总数
* 索引扫描的总数
* 索引扫描检索出的元组总数

后四列是持续增长的值，所以我使用的是 SDLONG 类型的指标，它是一个 sticky 增量 long。注意，在 [清单 14](https://www.ibm.com/developerworks/cn/java/j-rtm3/#listing14) 中，我配置了四个 QueryMap 来将四列值映射到跟踪名称空间。

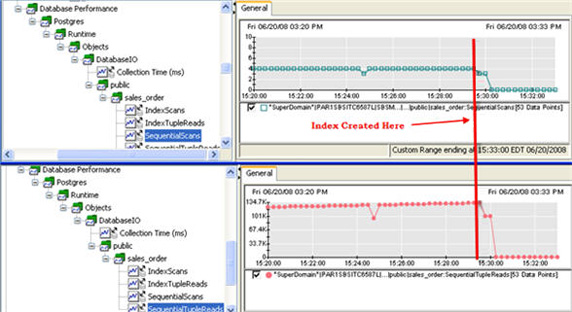
在这个场景中，我构造了一个有用的例子，而没有在 sales\_order 表上创建索引。因此，监控将会揭示更多的顺序扫描（按照数据库用语来说为表扫描），它是一个低效的检索数据的机制，这是因为它要读取表中的每一行。顺序元组读取— 主要指使用顺序扫描读取的行的数量 — 也一样。行和元组之间有一个很大的差别，但是在这里没有关系。要弄清楚这个差异，您可以查看 PostgreSQL 文档网站（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。看一看 APM 显示的这些统计信息，很明显我的数据库遗漏了一个索引。如图 11 所示：

图 11. 顺序读取



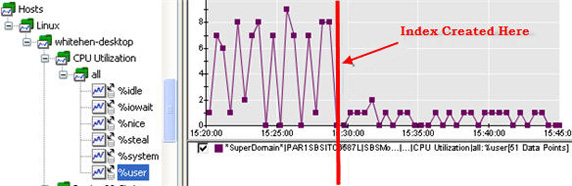
注意到了这一点后，我赶紧触发了两个 SQL 语句来索引表格。随后的结果是两个顺序操作都降到了零，而本来为零的索引操作现在却运行起来。如图 12 所示：

图 12. 索引后



这个索引的创建在整个系统中发生了连锁反应。另一个明显稳定下来的指标是数据库主机上的 User CPU %。如图 13 所示：

图 13. 索引后的 CPU



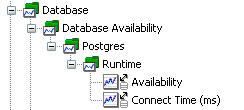
数据库可用性

我要介绍的有关 JDBC 的最后一个方面是数据库可用性。数据库可用性的最简单的形式是选择标准 JDBCCollector。如果收集器被配置了一个availabilityNameSpace 值，那么收集器将两个指标跟踪到配置的名称空间：

* **可用性**：可以连接到数据库的话，值为 1，无法连接到数据库的话，值为 0
* **连接时间**：获取连接消耗的运行时间

当用数据源或连接池来获取连接时，连接时间通常都是很快的。但大多数 JDBC 连接池系统都能在分发连接前执行一个可配置的 SQL 语句，所以测试是合法的。在重负载的情况下，连接的获取会有一个非零的运行时间。此外，可以为用于测试可用性的 JDBCCollector 设置单独的数据源。这个单独的数据源可以配置为不共享连接，所以每一个轮询周期都会启动一个新的连接。图 14 显示了我的 PostgreSQL 运行时数据库的可用性检查 APM 树。请参考 [清单 14](https://www.ibm.com/developerworks/cn/java/j-rtm3/#listing14)，查看使用 availabilityNameSpace 属性的例子。

图 14. 运行时数据库可用性检查



我见过需要多个连锁查询来决定一个特定状态的情况。例如，一个最终的状态需要对 Database A 进行查询，但它所需的参数只有对 Database B 进行查询才可以确定。这种情况可以用两个 JDBCCollector 来解决，但要特殊考虑如下几点：

* 按时间先后排列的第一个查询（针对 Database B）被配置为惰性的，因为它没有进行调度（收集的频率为零就意味着无调度）。JDBCCollector 的实例同样实现了 IBindVariableProvider，这意味着它也能够给另一个收集器提供绑定变量和绑定。
* 第二个收集器把第一个收集器定义为一个绑定，该绑定将会在第一次查询后实现。

我对数据库监控的论述就到此为止了。我必须补充说明一下，这一节重点介绍的是数据库监控，尤其是通过 JDBC 接口的数据库监控。完成一个典型的数据库监控还要监控数据库所在的 OS、单个数据库进程或数据库进程组、还有一些有必要访问数据库服务的相关的网络资源。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

监控 JMS 和消息传递系统

本节将描述监控消息传递服务的健康状况和性能的技巧。消息传递服务，如实现 JMS — 同样指面向消息的中间件（message-oriented middleware，MOM）— 的服务，在很多应用程序中都起着至关重要的作用。它们和其他应用程序依赖项一样也需要监控。通常，消息传递服务提供同步的或者说是 “即发即弃（fire-and-forget）” 调用点。监控这些点的难度要大一些，因为从很多角度来看，该服务看起来都运行得很好，服务的调用被频繁地分配，并且运行的时间也很短。仍然保持神秘的是上游瓶颈，消息要经过这里被转发到下一个目标，但消息在这里的传输速度很慢或者根本无法通过。

由于大多数消息传递服务都存在于 JVM 中，抑或是作为一个或多个本机进程而存在于一个主机（或一组主机）上，监控点包括一些与目标服务有关的相同的点。这些目标服务可能包括标准 JVM JMX 属性、支持主机上的监控资源、网络响应，以及诸如内存大小和 CPU 使用这样的服务进程特征。

我将概述消息传递服务监控的四个范畴，它们当中的三个都是专用于 JMS 的，另一个则涉及到专有的 API：

* 为了度量一个服务的吞吐性能，收集器会定时向该服务发送一组综合的 测试消息，然后等待它们的返回。发送、接收和整个往返过程的总运行时间都被度量、跟踪，同时被度量和跟踪的还有所有失败或超时事件。
* 很多基于 Java 的 JMS 产品都通过 JMX 公开指标和监控点，所以我将简短地回顾一下如何使用 Spring 收集器实现 JMX 监控。
* 有些消息传递服务为通信代理的管理提供了一个私有的 API。这些 API 通常包含提取运行中服务的性能指标的能力。
* 如果缺少上述选项中的任意一个的话，都可以使用诸如 javax.jms.QueueBrowser 这样的标准 JMS 构造来检索有用的指标。

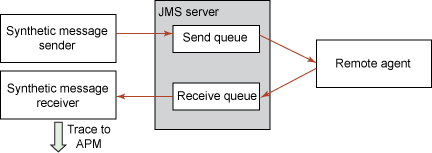
通过综合测试消息来监控消息传递服务

综合消息的前提是将测试消息的发送和接收安排到目标消息传递服务并度量消息的发送、接收和整个往返过程的运行时间。为了设计消息的返回并准确地度量从远程位置发送消息的响应时间，最佳的解决方案就是部署远程代理，它的任务是：

1. 监听中央监控器的测试信息
2. 检索它们
3. 给每一个收到的消息添加一个时间戳
4. 重新将它们发送回消息传递服务，目的是返回到中央监控器

然后中央监控器可以分析返回的消息，得出进程中的每一个跳跃点（hop）的运行时间并跟踪到 APM 系统。如图 15 所示：

图 15. 综合消息



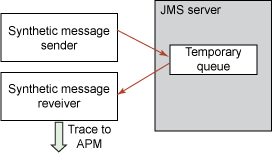
虽然这个方法涉及到了监控的大多方面，但它还是有一些缺点：

* 它需要部署和管理一个远程代理。
* 它需要在消息传递服务上为测试消息传输创建其他的队列。
* 有些消息传递服务允许自动动态创建一级队列，但很多消息传递服务都需要通过管理接口或管理 API 来手动创建队列。

这里介绍的另一个专用于 JMS（但可能在其他消息传递系统中有等效体）的选择是使用临时队列或主题。临时队列可以由标准 JMS API 自动创建，所以不需要介入任何管理。这些临时构造有很多额外的优势，但只有最初的创建者可以看到，其他所有 JMS 参与者都无法看到。

在这个场景中，我将使用 JMSCollector，它可以在启动时自动创建一个临时队列。当调度器提示时，它会将很多测试消息发送给目标 JMS 服务上的临时队列，然后再收回它们。这就有效地测试了 JMS 服务器的吞吐量，而无需创建具体队列或部署远程代理。如图 16 所示：

图 16. 带有临时队列的综合消息



这个场景的 Spring 收集器类为 org.runtimemonitoring.spring.collectors.jms.JMSCollector。配置依赖项相当的简单明了，而且大多数依赖项已经在前面的例子中设置过了。JMS 连通性需要一个 JMS javax.jms.ConnectionFactory。我用来获取它的 Spring bean 与在 Windows WPM 收集的例子中用来获取 JMS 连接工厂的 Spring bean 相同。在这里重新修改一下，它需要org.springframework.jndi.JndiTemplate 类型的 Spring bean 的一个实例，该类型的 Spring bean 可以给我的目标 JMS 服务提供一个 JNDI 连接；它还需要 org.springframework.jndi.JndiObjectFactoryBean 类型的 Spring bean 的一个实例，该类型的 Spring bean 可以使用 JNDI 连接来查找 JMS 连接工厂。

为了使综合消息负载的组成灵活一些，JMSCollector 被配置了名为org.runtimemonitoring.spring.collectors.jms.ISyntheticMessageFactory 的接口的一组实现。实现这个接口的对象提供了一组测试消息。收集器会调用每一个配置的工厂，并使用所提供的消息来执行往返测试。用这种方式，我就能够测试我的 JMS 服务器上的吞吐量，因为负载会随着消息大小和消息计数的变化而变化。

每一个 ISyntheticMessageFactory 都有一个可配置的、任意的名称，JMSCollector 把它添加到跟踪名称空间。清单 15 展示了完全的配置：

清单 15. 综合消息 JMSCollector

<!-- The JNDI Provider -->

<bean id="jbossJndiTemplate" class="org.springframework.jndi.JndiTemplate">

<property name="environment"><props>

<prop key="java.naming.factory.initial">

org.jnp.interfaces.NamingContextFactory

</prop>

<prop key="java.naming.provider.url">

localhost:1099

</prop>

<prop key="java.naming.factory.url.pkgs">

org.jboss.naming:org.jnp.interfaces

</prop>

</props></property>

</bean>

<!-- The JMS Connection Factory Provider -->

<bean id="RealJMSConnectionFactory"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiTemplate" ref="jbossJndiTemplate" />

<property name="jndiName" value="ConnectionFactory" />

</bean>

<!-- A Set of Synthetic Message Factories -->

<bean id="MessageFactories" class="java.util.HashSet">

<constructor-arg><set>

<bean

class="org.runtimemonitoring.spring.collectors.jms.SimpleSyntheticMessageFactory">

<property name="name" value="MapMessage"/>

<property name="messageCount" value="10"/>

</bean>

<bean

class="org.runtimemonitoring.spring.collectors.jms.ByteArraySyntheticMessageFactory">

<constructor-arg type="java.net.URL"

value="file:///C:/projects3.3/RuntimeMonitoring/lib/jta26.jar"/>

<property name="name" value="ByteMessage"/>

<property name="messageCount" value="1"/>

</bean></set>

</constructor-arg>

</bean>

<!-- The JMS Collector -->

<bean id="LocalJMSSyntheticMessageCollector"

class="org.runtimemonitoring.spring.collectors.jms.JMSCollector"

init-method="springStart">

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="JMS,Local,Synthetic Messages" />

<property name="frequency" value="5000" />

<property name="messageTimeOut" value="10000" />

<property name="initialDelay" value="3000" />

<property name="messageFactories" ref="MessageFactories"/>

<property name="queueConnectionFactory" ref="RealJMSConnectionFactory"/>

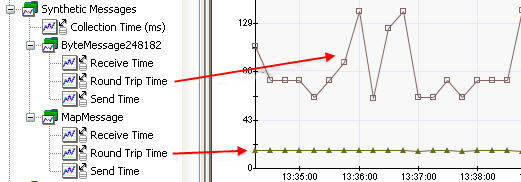
</bean>

清单 15 实现了两个消息工厂，它们是：

* javax.jms.MapMessage 工厂，它将当前 JVM 的属性载入到每一条消息的负载中，并被配置为每个周期发送 10 条消息
* javax.jms.ByteMessage 工厂，它将 JAR 文件中的字节载入到每一条消息的负载中，并被配置为每个周期发送 10 条消息

图 17 显示了综合消息监控的 APM 树。注意字节负载大小被附加到 javax.jms.ByteMessage 消息工厂名称的末尾。

图 17. 带有临时队列的综合消息的 APM 树



通过 JMX 监控消息传递服务

诸如 JBossMQ 和 ActiveMQ 这样的消息传递服务通过 JMX 公开了一个管理接口。我曾在 [第 1 部分](https://www.ibm.com/developerworks/cn/java/j-rtm3/URL%20HERE) 中介绍过基于 JMX 的监控。下面我将简短地回顾一下这种监控，然后我将介绍基于 org.runtimemonitoring.spring.collectors.jmx.JMXCollector 类的 Spring 收集器以及使用它来监控 JBossMQ 实例的方法。由于 JMX 是一个恒定的标准，所以可以使用同一过程来监控任何公开 JMX 的指标，而且它的应用范围很广。

JMXCollector 有两个依赖项：

* 一个名为 LocalRMIAdaptor 的 bean 为本地 JBossMQ 提供了一个 javax.management.MBeanServerConnection。在这种情况下，连接是通过发出一个对 JBoss org.jboss.jmx.adaptor.rmi.RMIAdaptor 的 JNDI 查找来获取的。假定可以提供任何可用的验证凭证，并且 Springorg.springframework.jmx.support 包提供了很多工厂 bean 来获取 MBeanServerConnection（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）的不同实现，这样的话其他的提供者通常很容易获取。
* JMX 收集属性的一个配置文件，它打包在包含 org.runtimemonitoring.spring.collectors.jmx.JMXCollection 的实例的收集 bean 中。这些是对 JMXCollector 发出的关于要收集哪些属性的指令。

JMXCollection 类展示了一些常见于 JMX 监控器的属性。基本的配置属性有：

* **targetObjectName**：它是旨在收集的 MBean 的完整 JMX ObjectName 名称，但它也可以是一个通配符。收集器为所有与通配符模式相匹配的 MBeans 查询 JMX 代理，然后从每一个代理中收集数据。
* **segments**：它是 APM 跟踪名称空间的一个片断，收集到的指标被跟踪到了这个 APM 跟踪名称空间。
* **metricNames**：它可以是一组指标名称，每一个 MBean 属性都要被映射到该指标名称；也可以是一个 \* 字符，它指导收集器使用 MBean 提供的属性名。
* **attributeNames**：它是一组必须从每一个目标 MBean 中收集的 MBean 属性名。
* **metricTypes** 或者 **defaultMetricType**：前者是必须用于每一个属性的一组指标类型，后者是必须应用于所有属性的某个指标类型。

MBean ObjectName 通配符功能非常强大，因为它可以有效地发现监控目标，而无需为每一个目标配置监控器。对于 JMS 对列，JBossMQ 为每一个队列创建了一个单独的 MBean，所以如果我想监控每一个队列中的消息数（称为队列长度），我只需要指定诸如jboss.mq.destination:service=Queue,\* 这样的通用通配符，JMS 队列 MBean 的所有实例都是从其中收集到的。但是，由于这些对象是自动发现的，所以如何动态确定队列名就成了另一个难题。在本例中，我知道被发现的 MBean 的 ObjectName name 属性值就是队列的名称。例如，一个被发现的 MBean 的对象名可能是 jboss.mq.destination:service=Queue,**name=MyQueue**。因此，我需要一种将被发现的对象的属性映射到跟踪名称空间的方式，这样就可以从每一个源中划分出被跟踪的指标。方法就是以与 JDBCCollector 中的 rowToken 相类似的形式来使用标记。JMXCollector 支持的标记有：

* **{target-property:name}**：该标记用源自目标 MBean 的 ObjectName 的命名属性来取代。例子：{target-property:name}。
* **{this-property:name}**：该标记用源自收集器的 ObjectName 的命名属性来取代。例子：{this-property:type}。
* **{target-domain:index}**：该标记用目标 MBean 的 ObjectName 域的索引片断来取代。例子：{target-domain:2}。
* **{this-domain:index}**：该标记用收集器的 ObjectName 域的索引片断来取代。例子：{target-domain:0}。

清单 16 展示了经过删减的 JBossMQ JMXCollector XML 配置：

清单 16. 本地 JBossMQ JMXCollector

<!-- The JBoss RMI MBeanServerConnection Provider -->

<bean id="LocalRMIAdaptor"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiTemplate" ref="jbossJmxJndiTemplate" />

<property name="jndiName" value="jmx/invoker/RMIAdaptor" />

</bean>

<!-- The JBossMQ JMXCollection Profile -->

<bean id="StandardJBossJMSProfile"

class="org.runtimemonitoring.spring.collectors.collections.InitableHashSet"

init-method="init" >

<constructor-arg><set>

<bean class="org.runtimemonitoring.spring.collectors.jmx.JMXCollection">

<property name="targetObjectName" value="\*:service=Queue,\*"/>

<property name="segments" value="Destinations,Queues,{target-property:name}"/>

<property name="metricNames" value="\*"/>

<property name="attributeNames"

value="QueueDepth,ScheduledMessageCount,InProcessMessageCount,ReceiversCount"/>

<property name="defaultMetricType" value="SINT"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.jmx.JMXCollection">

<property name="targetObjectName" value="jboss.mq:service=DestinationManager"/>

<property name="segments" value="Destniations,{target-property:service}"/>

<property name="metricNames" value="\*"/>

<property name="attributeNames" value="ClientCount"/>

<property name="defaultMetricType" value="SINT"/>

</bean>

<!-- MBeans Also Included: Topics, ThreadPool, MessageCache -->

</set>

</constructor>

</bean>

<!-- The JMXCollector for local JBoss MQ Server -->

<bean id="LocalJBossCollector"

class="org.runtimemonitoring.spring.collectors.jmx.JMXCollector"

init-method="springStart">

<property name="server" ref="LocalRMIAdaptor" />

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="JMS,Local" />

<property name="objectName"

value="org.runtime.jms:name=JMSQueueMonitor,type=JMXCollector" />

<property name="frequency" value="10000" />

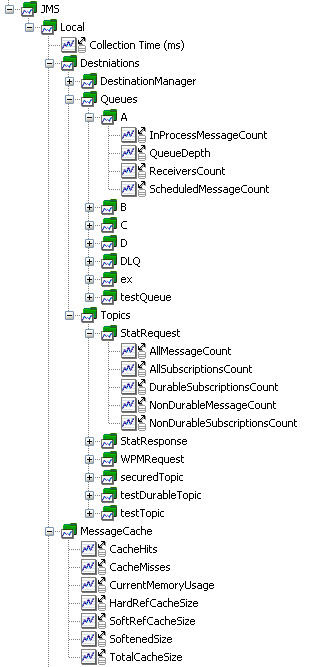
<property name="initialDelay" value="10" />

<property name="jmxCollections" ref="StandardJBossJMSProfile"/>

</bean>

图 18 显示了用 JMXCollector 监控的 JBossMQ 服务器的 JMS 队列的 APM 树：

图 18. JBossMQ 队列的 JMX 监控的 APM 树



用队列浏览器监控 JMS 队列

如果缺少足够用于监控 JMS 队列的管理 API 的话，可以使用 javax.jms.QueueBrowser。队列浏览器的行为与 javax.jms.QueueReceiver 的行为类似，不同的是获取的信息不会从队列中移除，而且在被浏览器检索之后仍然可以发送。队列深度通常都是一种重要的指标。据观察在很多消息传递系统中，消息生产者要多于消息消费者。这种严重的不平衡从代理队列中消息的数量可以看出来。因此，如果无法用其他的方式访问队列深度的话，那么最后一招就是使用队列浏览器了。该技巧有很多的缺点。为了计数队列中的消息数，队列浏览器一定要检索队列中的每一条信息（然后删除它们）。这样做的效率是很低的，而且要耗费比使用管理 API 多得多的收集时间 — 而且可能在 JMS 服务器的资源上开销更高。队列浏览的另一个问题是，对于繁忙的系统来说，计数在浏览完成的时候很可能就已经是错误的了。虽说如此，要是为了监控的话，近似值还是可以接受的；更何况在高负载的系统中，即便是对队列深度在给定瞬间的高精确度的度量在下一个瞬间也会是无用的了。

队列浏览有一个好处：在浏览队列的信息的过程中，可以确定最老的消息的年龄。这是一个很难获取的指标，即便是使用最好的 JMS 管理 API 也很难获取，而且在某些情况下，它可能是一个至关重要的监控点。思考一下用于传输重要信息的 JMS 队列。消息生产者和消息消费者有着明显的区别，而流量的模式是这样的：对队列深度执行一次标准的轮询通常显示一到两条信息。通常情况下，导致这个问题的原因是存在一定的延迟，而轮询的频率却是一分钟，队列中的消息不同于轮询到轮询间的消息。它们相同么？它们可能不是相同的消息，在这种情况下，上述的情况就是很正常的。但也可能是消息生产者和消息消费者同时遭遇失败，因此队列中被观察的这对消息在每个轮询中是相同的消息。在这种场景中，在监控队列深度的同时监控最老的消息的年龄就会使情况变得很清晰了：正常情况下，消息的年龄要少于几秒，但如果生产者和消费者同时失败的话，从 APM 出现明显的数据只会占用两个轮询周期间的时间。

这个功能在 Spring 收集器的 org.runtimemonitoring.spring.collectors.jmx.JMSBrowserCollector 中有所展示。它的另外两个配置属性有 javax.jms.ConnectionFactory，它与 JMSCollector 类似，还有要浏览的队列的集合。清单 17 展示了该收集器的配置：

清单 17. 本地 JBossMQ JMSBrowserCollector

<!-- A collection of Queues to browse -->

<bean id="BrowserMonitorQueues" class="java.util.HashSet">

<constructor-arg>

<set>

<bean id="QueueA"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiTemplate" ref="jbossJndiTemplate" />

<property name="jndiName" value="queue/A" />

</bean>

<bean id="QueueB"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiTemplate" ref="jbossJndiTemplate" />

<property name="jndiName" value="queue/B" />

</bean>

</set>

</constructor-arg>

</bean>

<!-- the JMS Queue Browser -->

<bean id="LocalQueueBrowserCollector"

class="org.runtimemonitoring.spring.collectors.jms.JMSBrowserCollector"

init-method="springStart">

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="JMS,Local,Queue Browsers" />

<property name="frequency" value="5000" />

<property name="initialDelay" value="3000" />

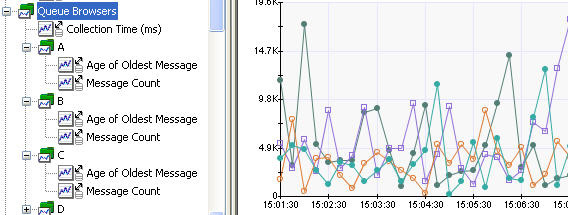
<property name="queueConnectionFactory" ref="RealJMSConnectionFactory"/>

<property name="queues" ref="BrowserMonitorQueues"/>

</bean>

图 19 展示了该收集器的 APM 树：

图 19. JMSBrowserCollector 的 APM 树



作为一个测试机制，一个载入脚本开始循环，在循环中向每一个队列发送了几百条消息。在每一个循环中，被清除的队列是随机选取的。因此，每一个队列中的消息年龄的上限都会随着时间的推移而随意变化。

使用私有 API 监控消息传递系统

有些消息传递系统拥有实现诸如监控这样的管理功能的私有 API。一些消息传递系统使用请求/响应 的模式来用它们自己的消息传递系统提交管理请求。ActiveMQ（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）提供了一个 JMS 通信管理 API 以及一个 JMX 管理 API。实现一个私有的管理 API 需要一个自定义的收集器。在这个小节中，我将呈现 WebSphere® MQ（原来称为 MQ Series）的收集器。该收集器结合使用了两种 API：

* **MS0B: WebSphere MQ Java classes for PCF**：PCF API 是 WebSphere MQ 的一个管理 API。
* **The core WebSphere MQ Java classes**：原来称为 MA88 的 API 已经被合并到了核心 WebSphere MQ Java 类库中（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。

其实使用两个 API 是多余的，但是毕竟展示了两种不同的私有 API 的使用方法。

Spring 收集器实现是一个名为 org.runtimemonitoring.spring.collectors.mq.MQCollector 的类。它监控 WebSphere MQ 服务器上的所有队列，收集每一个队列的队列深度以及当前打开的输入/输出处理的数量。清单 18 展示了org.runtimemonitoring.spring.collectors.mq.MQCollector 的配置：

清单 18. WebSphere MQ 收集器

<bean id="MQPCFAgentCollector"

class="org.runtimemonitoring.spring.collectors.mq.MQCollector"

init-method="springStart">

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="MQ, Queues" />

<property name="frequency" value="5000" />

<property name="initialDelay" value="3000" />

<property name="channel" value="SERVER1.QM2"/>

<property name="host" value="192.168.7.32"/>

<property name="port" value="50002"/>

</bean>

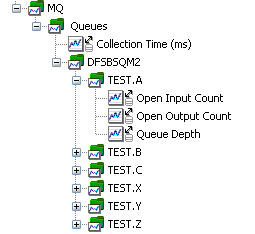
这里独特的配置属性有：

* **host**WebSphere MQ 服务器的主机名称的 IP 地址
* **port**：WebSphere MQ 进程在该端口监听连接
* **channel**：要连接到的 WebSphere MQ 通道

注意这个例子不包含任何的验证方面。

图 20 展示了为 org.runtimemonitoring.spring.collectors.mq.MQCollector 生成的 APM 树：

图 20. MQCollector 的 APM 树



我对消息传递服务监控的论述就到此为止。正如我在前面提到的，接下来我将介绍如何用 SNMP 进行监控。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

使用 SNMP 进行监控

SNMP 原本是作为应用层协议而创建的，该协议用来在诸如路由器、防火墙和交换机这样的网络设备间交换信息。这也许仍然是它最常用的功能，但是它也可以充当监控性能和可用性的灵活的标准化协议。SNMP 的整个主题以及它作为一个监控工具的实现超出了本文的讨论范围。但是，SNMP 在监控领域已经非常普及，因此难免会有所遗漏。

SNMP 中的核心结构之一就是代理，它负责代理针对某一个设备的 SNMP 请求。SNMP 相对简单且较低级，所以 SNMP 代理可以简单且快捷地嵌套到多种硬件设备和软件服务中。因此 SNMP 就是一个在应用程序生态系统中监控大量服务的标准化协议。另外，SNMP 被广泛用于通过扫描一系列的 IP 地址和端口来执行发现。从监控的角度看，SNMP 的这个应用可以节省一些管理时间，因为它能够自动填充和更新一个监控目标的集中式清单。SNMP 在很多方面都与 JMX 极其相似。虽然两者之间还是有一些明显的差别，但还是可以在两者间找出一些相同的内容，而且 JMX 和 SNMP 间的互操作性被广泛地支持与实现。表 1 总结了一些两者之间的等价内容：

表 1. SNMP 和 JMX 对比

| **SNMP 结构** | **等效 JMX 结构** |
| --- | --- |
| 代理 | 代理或者 MBeanServer |
| 管理程序 | 客户机、MBeanServerConnection、协议适配器 |
| MIB | MBean、ObjectName、MBeanInfo |
| OID | ObjectName 和 ObjectName+ Attribute 名 |
| 陷阱 | JMX Notification |
| GET、SET | getAttribute、setAttribute |
| BULKGET | getAttributes |

单从监控角度考虑，发布一个 SNMP 查询时，我需要了解的重要因素有：

* **主机地址**：目标 SNMP 代理所在的 IP 地址或主机名。
* **端口**：目标 SNMP 代理在其上进行监听的端口。由于一个网络地址可能在处理很多个 SNMP 代理，每一个代理都需要在不同的端口上进行监听。
* **协议版本**：SNMP 协议已经经历过多次的修订了，而支持级别会根据不同的代理而变化。可选的版本有：1、2c 和 3。
* **社区**：SNMP 社区是一个松散定义的管理域。如果社区不是已知的话，SNMP 客户机就无法凭代理发出请求，所以发挥着松散形式的验证作用。
* **OID**：这是一个指标或一组指标的惟一标示符。其格式为一系列用点分隔的整数。例如，Linux 主机的一分钟载入的 SNMP OID 为.1.3.6.1.4.1.2021.10.1.3.1，由 1、5 和 15 分钟载入的指标子集的 OID 为 .1.3.6.1.4.1.2021.10.1.3。

除了社区以外，有些代理还可以定义其他层面的验证。

在我深入论述 SNMP API 之前，要注意 SNMP 指标可以使用两个普通的命令行实用程序来检索，它们是： snmpget，它检索一个 OID 的值；snmpwalk，它检索 OID 值的子集。记住了这点，我就能够经常扩展我的 ShellCollector CommandSet 来跟踪 SNMP OID 值了。清单 19 示范了一个 snmpwalk 的例子，它带有在 Linux 主机上检索 1、5 和 15 分钟载入的原始的输出。我使用的是该协议的 2c 版本和公共社区。

清单 19. snmpwalk 示例

$> snmpwalk -v 2c -c public 127.0.0.1 .1.3.6.1.4.1.2021.10.1.3

UCD-SNMP-MIB::laLoad.1 = STRING: 0.22

UCD-SNMP-MIB::laLoad.2 = STRING: 0.27

UCD-SNMP-MIB::laLoad.3 = STRING: 0.26

$> snmpwalk -v 2c -c public 127.0.0.1 .1.3.6.1.4.1.2021.10.1.3 \

| awk '{ split($1, name, "::"); print name[2] " " $4}'

laLoad.1 0.32

laLoad.2 0.23

laLoad.3 0.22

第二条命令可以轻松地在我的 Linux 命令集中表现出来，如清单 20 所示：

清单 20. 处理 snmpwalk 命令的 CommandSet

<CommandSet name="LinuxSNMP">

<Commands><Command>

<shellCommand><![CDATA[snmpwalk -v 2c -c public 127.0.0.1

.1.3.6.1.4.1.2021.10.1.3 | awk '{ split($1, name, "::"); print name[2] "

" $4}']]></shellCommand>

<Extractors><CommandResultExtract>

<paragraph id="0" name="System Load Summary" header="false"/>

<columns entryName="0" values="1" offset="0">

<namemapping from="laLoad.1" to="1 Minute Load"/>

<namemapping from="laLoad.2" to="5 Minute Load"/>

<namemapping from="laLoad.3" to="15 Minute Load"/>

</columns>

<tracers default="SINT"/>

<lineSplit>\n</lineSplit>

</CommandResultExtract></Extractors>

</Command></Commands>

</CommandSet>

商用的、开源 SNMP Java API 有好多个。我实现了一个基本的 Spring 收集器，它的名称为org.runtimemonitoring.spring.collectors.snmp.SNMPCollector，该收集器的实现使用到了一个名为 joeSNMP 的开源 API（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。该收集器拥有如下的重要配置属性：

* **hostName**：目标主机的主机名或 IP 地址。
* **port**：目标 SNMP 代理正在其上进行监听的端口的数量（默认为 161）。
* **targets**：一组由 org.runtimemonitoring.spring.collectors.snmp.SNMPCollection 的实例组成的 SNMP OID 目标。该 bean 的配置属性有：
  + **nameSpace**：跟踪名称空间的后缀。
  + **oid**：跟踪名称空间的后缀。
* **protocol**：SNMP 协议， 0 表示 v1，1 表示 v2（默认为 v1）。
* **community**：SNMP 社区（默认为 public）。
* **retries**：尝试操作的次数（默认为 1）。
* **timeOut**：以 ms 为单位的 SNMP 调用的超时时间（默认为 5000）。

清单 21 展示了 SNMPCollector 设置的示例配置，该配置的目的是监控本地 JBoss 应用服务器：

清单 21. SNMPCollector 的配置

<!-- Defines the SNMP OIDs I want to collect and the mapped name -->

<bean id="JBossSNMPProfile" class="java.util.HashSet">

<constructor-arg><set>

<bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">

<property name="nameSpace" value="Free Memory"/>

<property name="oid" value=".1.2.3.4.1.2"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">

<property name="nameSpace" value="Max Memory"/>

<property name="oid" value=".1.2.3.4.1.3"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">

<property name="nameSpace" value="Thread Pool Queue Size"/>

<property name="oid" value=".1.2.3.4.1.4"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">

<property name="nameSpace" value="TX Manager, Rollback Count"/>

<property name="oid" value=".1.2.3.4.1.7"/>

</bean>

<bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">

<property name="nameSpace" value="TX Manager, Current Count"/>

<property name="oid" value=".1.2.3.4.1.8"/>

</bean>

</set></constructor-arg>

</bean>

<!-- Configures an SNMP collector for my local JBoss Server -->

<bean id="LocalJBossSNMP"

class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollector"

init-method="springStart">

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="Local,JBoss,SNMP" />

<property name="frequency" value="5000" />

<property name="initialDelay" value="3000" />

<property name="hostName" value="localhost"/>

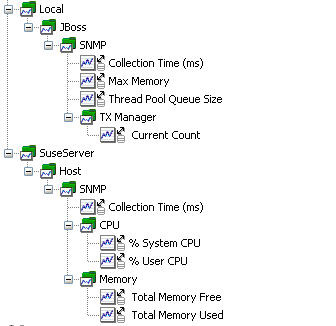
<property name="port" value="1161"/>

<property name="targets" ref="JBossSNMPProfile"/>

</bean>

该收集器的确是有些缺点，毕竟配置有些长，而且由于它要为每一个 OID 做出一个调用而非批量收集，导致运行时效率低下。示例代码中的 snmp-collectors.xml 文件（参见 [下载](https://www.ibm.com/developerworks/cn/java/j-rtm3/#download)）也包含了一个监控 Linux 服务器的 SNMP 收集器的配置示例。图 21 显示了 APM 系统指标树：

图 21. SNMPCollector 的 APM 树



此时，您可能了解到如何创建收集器了。要覆盖整个环境，可能需要集中不同类型的收集器，如果您对此感兴趣的话，您可以参见本文的示例代码，那里包含有其他监控目标的收集器示例。它们都位于 org.runtimemonitoring.spring.collectors 包中。表 2 是这些收集器的一个概述：

表 2. 其他收集器示例

| **收集目标** | **类** |
| --- | --- |
| Web 服务：检查安全 Web 服务的响应时间和可用性 | webservice.MutualAuthWSClient |
| Web 服务和 URL 检查 | webservice.NoAuthWSClient |
| Apache Web Server、性能和可用性 | apacheweb.ApacheModStatusCollector |
| Ping | network.PingCollector |
| NTop：一个收集详细网络统计信息的实用程序 | network.NTopHostTrafficCollector |

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

数据管理

收集繁忙的可用性和性能数据的系统所面临的一个最复杂的难题就是：如何有效地将收集到的数据持久化到普通指标数据库中。对于要使用的数据库和持久化机制要考虑的问题有：

* 指标数据库一定要支持对历史指标的适当快速的、简单的查询，以实现数据可视化、数据报告和数据分析。
* 指标数据库一定要保留数据的历史和粒度，从而支持报告的时窗、准确性和所需精确度。
* 持久化机制一定要足够好地、并行地运行，避免影响前端监控的活跃性。
* 指标数据的检索和储存要能够并行运行，且两者都不会对方有任何负面的影响。
* 从数据库中请求数据的请求要能够支持一段时间内的聚合。
* 数据库中的数据要以适当的方式存储，允许使用一种时间序列模式检索数据或使用某种机制保证多个数据点能够在有效时间段相同的情况下相互关联。

鉴于这些考虑，您需要：

* 一个性能良好的、可伸缩的、带有很多磁盘空间的数据库。
* 一个带有有效搜索算法的数据库。本质上讲，由于指标将会通过复合名称来储存，所以解决方案就是将复合名称储存为一个字符串并使用某种形式的模式匹配来指定目标查询指标。很多关系数据库都支持内置于 SQL 语法的正则表达式，这是一个通过复合名称查询的理想方法，但是这个方法比较慢，这是因为它通常排除使用索引。但是很多关系数据库同样支持功能索引，该索引能够在使用模式匹配搜索时用来加快查询的速度。另外一个选择是完全标准化 数据并分别取出复合名称的单个片断（参见下面的 [标准化数据库结构和扁平数据库结构](https://www.ibm.com/developerworks/cn/java/j-rtm3/#NormalizedVSFlat) ）。
* 限制写入数和写入数据库的总数据量的策略就是实现一连串的分层聚合和合并。这可以在数据被写入数据库前通过保持一个总体缓冲器来完成。在指定的时间段内，将所有标记为储存的指标写入一个累计缓冲器，它可以保留指标有效启动时间和终止时间，以及在该段时间内的平均的、最小的和最大的指标值。这样，指标值就可以在写入数据库前被聚合和合并。例如，如果一个收集器的频率是 15 秒，而聚合时间是 1 分钟的话， 4 个单个指标将会被合并成为 1 个。这里假设可以允许持久化的数据丢失粒度，所以要在低数据量的低粒度和高数据量的高粒度之间做出权衡。同时，指标的实时可视化可以通过从聚合前的常驻内存循环缓冲器呈现图形来实现，这样就只有持久化的数据被聚合了。当然，您有权选择哪些指标需要持久化。
* 另外一个减少储存数据量的策略是实现一个采样频率，在这个采样频率中只有每 y 行中的 x 个指标会被储存。这也会降低持久化数据的粒度，但会占用比维护聚合缓冲器更少的资源（尤其是内存）。
* 持久化的指标数据还可以在指标数据库中累积并清除。同样，在容忍的粒度丢失范围内，可以将数据库中几个时间段内的数据分小时、天、周、月等累计到汇总表格中；你也可以清除原始数据；仍然保留适当的有用的指标数据。
* 为了避免数据持久化过程的活动影响到数据收集器本身，有必要实现一个非侵入式的后台线程，它能够将离散的指标数据的收集刷新到数据储存进程。
* 对于多个拥有不同有效时窗的数据集，创建一个类似时间序列的报告有些困难。思考一下图形中表示时间的 x 轴，和表示特定指标的值和多个序列（直线或曲线）（代表来源于不同资源集的相同指标）的 y 轴。如果读取每一个序列的有效时间戳有很大差别的话，就一定要传输数据以保持图形表示有效。这个可以通过将数据聚合到所有绘制的序列中最不一致的时窗来实现，同样粒度也会丢失。更简单的解决方案是维护一个在所有指标值中都一致的、有效的时间戳。时间序列数据库有这个功能。常用的是 [RRDTool](https://www.ibm.com/developerworks/cn/java/j-rtm3/#rrdtool)，它跨序列中所有不同的数据值有效地实施一致的、均衡分布的时间戳。为了使有效的时间戳在关系数据库中保持一致，一个绝佳的策略就是用一个单一的、统一的调度程序对所有指标取样。例如，一个计时器可能每两分钟触发一次，结果导致在那一刻捕获了所有指标的一个单一的 “交换”，而且所有的指标都用相同的时间戳标记，然后放入队列中等待持久化。

显然，要以一种最佳的方式来解决上述的每一个问题是很难的，所以必须要考虑折衷的办法。图 22 展示了一个数据收集器在概念上的数据持久化流程：

图 22. 数据管理流程

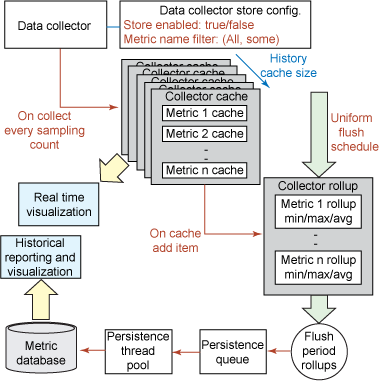


图 22 表示概念数据收集器（像本文中呈现的 Spring 收集器）的数据流。目的是通过一些层推动单个指标跟踪度量，直到它们被持久化。具体的过程为：

1. 数据收集器是随意配置的，为了实现带有如下属性的指标持久化：
   * **启用持久化**：True 或 false。
   * **指标名称过滤器**：一个正则表达式。匹配正则表达的指标名会被标记为持久化。
   * **采样计数**：会跳过持久化的指标收集的数量，也就是说 0 意味着全部持久化。
   * **历史缓存大小**：必须保存在缓存中的被跟踪的指标的数量。为了实现实时可视化，必须为历史缓存启用大多数指标，即使这些指标未被标记为持久化，这样它们可以呈现为实时图表。
2. 收集器缓存就是很多的离散指标读数，该读数与收集器生成的各种指标的数量相等。缓存遵循先入先出（first in, first out，FIFO）原则，所以当历史缓存变满时，最老的指标会被清除，为最新的指标腾出空间。缓存支持注册缓存事件监听器，它们可以被告知有哪些指标被从缓存中移除，又有哪些被存入。
3. 收集器总计是一系列由收集器生成的指标实例的两个或更多循环缓存。由于循环缓冲器中的每一个缓存实例变为活动状态，它就会为历史缓存注册新的跟踪事件并聚合每一个输入的新值，有效地合并给定时间段内的数据流。该总计包含：
   * 该时间段的启动和终止时间（终止时间在时间段结束前是未定的）。
   * 该时间段内的最小、最大和平均读数。
   * 指标类型。
4. 中央计时器会在每个时间段末尾触发一个刷新事件。当计时器触发后，收集器总计循环缓冲器索引是递增的，而历史缓存事件被传递给缓冲器中的下一个缓存。然后 “关闭的” 缓冲器中的每一个聚合的总计都会有一个公用的时间戳应用于该总计，并被写入待存入数据库的持久化队列。

注意当循环缓冲器递增时，下一个缓冲元素的最小的、最大的和平均值将会是零，除了 sticky 指标类型。

1. 持久化线程池中的一池线程从持久化队列中读取总计项并将它们写入数据库。

相同收集器总计被合并的总计时间段的长度要取决于中央刷新计时器的频率。该时间段越长，写入数据库的数据就越少，代价是数据粒度将会丢失。

标准化数据库结构与扁平数据库结构

RRDTool：集全部监视功能为一体的工具

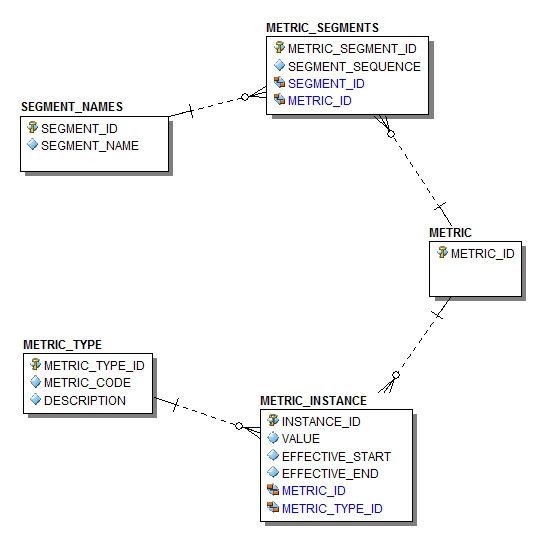
RRDTool（Round Robin Database Tool）是一个强大的工具，它在一个单独的可执行文件中提供了大量功能（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。RRDTool 不会为您收集任何实际数据，但是它通常被与 snmpget、snmpwalk 和其他命令行工具结合使用来创建一种综合监控解决方案。RRDTool 使您能够创建循环（round-robin）的时间序列数据库文件（循环特性使文件大小保持不变，避免文件不受控制地增长），以及针对多个聚合调度的一些选项。然后，它可以从文件查询并导出数据，生成丰富的图表以实现数据可视化和动态报告。添加一个简单的 Web 服务器来持续刷新包含图表和交叉表格数据的页面，从而得到一个功能强大的监控解决方案。

一些专用数据库（比如 [RRDTool](https://www.ibm.com/developerworks/cn/java/j-rtm3/#rrdtool)）可以用于指标存储，但是通常使用的是关系数据库。在关系数据库中执行指标存储时，需要考虑两个一般选项：标准化数据或保持一种更加扁平的结构。这主要关系到如何存储复合指标名称；根据数据建模最佳实践，应该对所有其他引用数据进行标准化。

完全标准化的数据库结构的好处在于，通过将指标复合名称分解为独立的片段，实际上任何数据库都可以利用索引功能来加速查询。它们存储的冗余数据更少，从而使数据库更小，数据密度更高，性能也更好。其缺点在于复杂性和查询的大小：即使相对简单的复合指标名称或指标名称的模式（例如主机 11 到 93 之间的 % User CPU）都要求 SQL 包含一些连接和大量谓词。这个缺点可以通过使用数据库视图和硬存储常用指标名解码来减轻。每个单独指标的存储需要分解复合名称来定位指标引用数据项（或创建一个新的引用数据项），但是这种性能开销可以通过在持久过程中缓存所有引用数据来减轻。

图 23 展示一种用于在标准化结构中存储复合指标的模型：

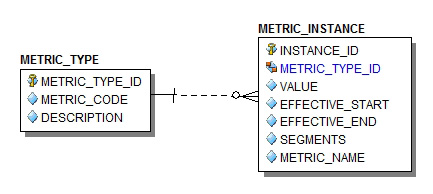
图 23. 一种用于指标存储的标准化模型



在图 23 中，为每个惟一的复合指标分配了一个惟一的 METRIC\_ID，为每个惟一的片段分配了一个 SEGMENT\_ID。然后在一个包含METRIC\_ID、SEGMENT\_ID 和片段出现顺序的关联实体中构建复合名称。指标类型存储在引用表 METRIC\_TYPE 中，而指标值本身存储在METRIC\_INSTANCE 中（包含值、时间戳的开始和结束属性，以及对指标类型和惟一 METRIC\_ID 的引用）。

另一方面，扁平模型结构非常简单，如图 24 所示：

图 24. 用于指标存储的扁平模型



在本例中，我将指标名称从复合名称中分离开来，剩余的片段仍然保持其在 segments 列中的原有模式。同样，如果实现的数据库引擎能够使用基于模式的谓词（比如正则表达式）执行适合较宽的文本列的查询，那么这种模型非常易于查询。这种优点应当重视。使用简化的查询结构，能够显著地简化数据分析、可视化和报告工具，而且在紧急的分类会话期间，加快查询的编写速度可能是最重要的方面！

如果需要持久化数据存储，挑选正确的数据库是非常关键的一步。使用关系数据库是一种可行的方法，因为关系数据库能够很好地执行，并且可以从中提取数据并针对您的需要进行格式化。时间序列驱动的数据的生成比较复杂，但是通过正确地分组和聚合 —— 以及使用其他工具，比如 JFreeChart（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）—— 可以生成出色的典型报告和图表。如果想要实现一种更加专门化的数据库（比如 RRDTool），那么请做好心理准备，因为这种数据库的提取和报告需要很长时间。如果数据库不支持 ODBC、JDBC 等标准，那么将不能使用常用的标准化报告工具。

对数据管理的讨论到此为止。本文最后一节将提供一些用于实时可视化数据的技术。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

可视化和报告

在某些情况下，将实现插装和性能数据收集器，数据将会流入您的 APM 系统中。那么下一个逻辑步骤就是实时地查看数据的可视化表示。我在这里宽泛地使用实时 这个词，其含义是可视化地表示最近收集的数据。

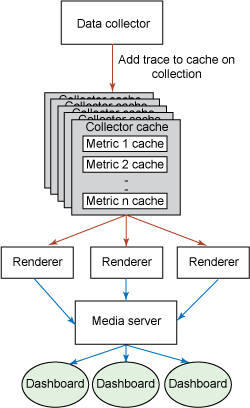
常用的数据可视化术语是指示板。指示板能够可视化地呈现您所考虑的数据模式或活动的所有方面，它们惟一的限制就是数据的质量和数量。本质上，指示板告诉您生态系统中正在发生什么。指示板在 APM 系统中的一个强大之处是，它能够通过一种统一的显示形式表示各种各样的数据（即从不同来源收集的数据）。例如，一种显示形式可以在指示板上同时显示数据库中 CPU 的最近使用情况和当前趋势、应用服务器之间的网络活动，以及当前登录到您的应用程序的用户数量。在本节中，我将提供一些不同的数据可视化样式，以及我在本文前面提供的 Spring 收集器的一个可视化层实现示例。

Spring 收集器可视化层的前提条件包括：

* 将一个缓存类实例部署为一个 Spring bean。可以配置缓存类实例来保存任何数量的 ITracer 历史跟踪，但它的大小是固定的并且遵循先进先出的规则。一个缓存可以配置为保存 50 条历史记录，这意味着一旦缓存被填满，它将保存最近的 50 条历史记录。
* Spring 收集器在 Spring XML 配置文件中使用 cacheConfiguration 进行配置，使用一个缓存进程封装 ITracer 实例。配置还将收集器与定义的缓存实例相关联。收集的跟踪记录像平常一样进行处理，但是添加了与收集器相关联的缓存实例。我们看一下前面的示例，如果缓存的历史记录大小为 50，并且收集器每隔 30 秒收集一次，那么当缓存被填满时，它将保存收集器在最近 25 分钟内收集的历史跟踪记录。
* Spring 收集器实例部署了大量呈现类。呈现器（Renderer）是一些用于实现 org.runtimemonitoring.spring.rendering.IRenderer 接口的类。它们的工作是从缓存获取一系列数据并呈现这些数据的某种可视化形式。从呈现器定期检索可视化媒体将生成最新的表示，或者与缓存数据一样新的表示。
* 呈现内容然后可以在一种指示板上下文（比如一个 Web 浏览器）中传递到一个客户机或某种形式的富客户机。

图 25 列出了这一过程：

图 25. 缓存和呈现



本例中的缓存实现是 org.runtimemonitoring.spring.collectors.cache.TraceCacheStore。可以将其他对象注册为缓存事件监听器，这样，在其他事件中，呈现器可以监听表示新值被添加到缓存的新缓存项 事件。通过这种方式，呈现器能够实际地缓存它们生成的内容，并在新数据可用时使缓存无效。来自呈现器的内容通过一个称为 org.runtimemonitoring.spring.rendering.MediaServlet 的 servlet 传递到客户机指示板。该 servlet 解析它接收到的请求，定位呈现器，并请求内容（所有内容都以一个字节数组的形式呈现和传递）和内容的 MIME 类型。字节数组和 MIME 类型然后流动到客户机，以供客户机解释。处理来自基于 URL 的服务的图形内容是最理想的，因为它们可以被 Web 浏览器、富客户机及它们之间的所有内容使用。当呈现器从媒体服务器接收到一个内容请求时，内容将从缓存传递，除非该缓存被一个缓存事件标记为 dirty。通过这种方式，呈现器不需要对每个请求重新生成它们的内容。

以字节数组格式生成、缓存和传递可视化的媒体非常有用，因为这种格式占用的空间最小，并且大多数客户机都可以在提供了 MIME 类型的情况下重新建立内容。由于这种实现在内存中缓存生成的内容，所以我使用了一种压缩方案。总体内存消耗随着缓存内容的增多而显著增加；同样，如果内容附带提供了压缩算法标记，那么大多数客户机都能够解压缩。例如，时下流行的大多数浏览器都支持 gzip 解压缩。然而，合理的压缩级别不一定要非常高（对于较大的图像来说，30-40% 的压缩率就不错了），因此，呈现实现可以缓存到磁盘，或者如果磁盘访问开销更高，动态地重新生成内容可能会需要更少的资源。

在这里，使用一个具体示例将有助于理解。我设置了两个 Apache Web Server 收集器来监控大量繁忙的 worker 线程。每个收集器都分配有一个缓存，而且我设置了少量的呈现器，用以提供图表来显示每个服务器上繁忙的 worker 线程。在本例中，呈现器生成一个 PNG 文件来显示两个服务器的时间序列线图及其序列。一个服务器的收集器和缓存设置如清单 22 所示：

清单 22. 一个 Apache Web Server 收集器和缓存

<!-- The Apache Collector -->

<bean id="Apache2-AP02"

class="org.runtimemonitoring.spring.collectors.apacheweb.ApacheModStatusCollector"

init-method="springStart">

<property name="scheduler" ref="CollectionScheduler" />

<property name="logErrors" value="true" />

<property name="tracingNameSpace" value="WebServers,Apache" />

<property name="frequency" value="15000" />

<property name="initialDelay" value="3000" />

<property name="modStatusURL" value="http://WebAP02/server-status?auto" />

<property name="name" value="Apache2-AP02" />

<property name="cacheConfiguration">

<bean

class="org.runtimemonitoring.spring.collectors.cache.CacheConfiguration">

<property name="cacheStores" ref="Apache2-AP02-Cache"/>

</bean>

</property>

</bean>

<!-- The Apache Collector Cache -->

<bean id="Apache2-AP02-Cache"

class="org.runtimemonitoring.spring.collectors.cache.TraceCacheStore">

<constructor-arg value="50"/>

</bean>

注意收集器中的 cacheConfiguration 属性，以及它如何引用称为 Apache2-AP02-Cache 的缓存对象。

我还设置了一个呈现器，它是 org.runtimemonitoring.spring.rendering.GroovyRenderer 的一个实例。这个呈现器将所有呈现任务委派给文件系统的一个底层 Groovy 脚本。这是最理想的方法，因为我能够在运行时对其进行调整，从而调整生成的图的细节。这个呈现器的一般属性包括：

* **groovyRenderer**：一个对 org.runtimemonitoring.spring.groovy.GroovyScriptManager 的引用，它被配置为从一个目录载入 Groovy 脚本。这也是我用于将从 Telnet 会话返回的数据传递到我的 Cisco CSS 的类。
* **dataCaches**：一组缓存，呈现器从中请求数据并呈现出来。呈现器也注册为当缓存添加新项时从缓存接收事件。当它这样做时，它将其内容标记为 dirty，并对下一个请求重新生成内容。
* **renderingProperties**：传递给呈现器的默认属性，指定生成的图形的具体细节，比如图像的默认大小。您将会在下面看到，这些属性可以被客户机请求覆盖。
* **metricLocatorFilters**：一个收集器缓存，包含收集器生成的每个指标的缓存跟踪。这个属性允许指定一个正则表达式数组来筛选您想要的指标。

缓存设置如清单 23 所示：

清单 23. 用于 Apache Web Server 繁忙 worker 线程监控的图形呈现器

<bean id="Apache2-All-BusyWorkers-Line"

class="org.runtimemonitoring.spring.rendering.GroovyRenderer"

init-method="init"

lazy-init="false">

<property name="groovyRenderer">

<bean class="org.runtimemonitoring.spring.groovy.GroovyScriptManager">

<property name="sourceUrl" value="file:///groovy/rendering/multiLine.groovy"/>

</bean>

</property>

<property name="dataCaches">

<set>

<ref bean="Apache2-AP01-Cache"/>

<ref bean="Apache2-AP02-Cache"/>

</set>

</property>

<property name="renderingProperties">

<value>

xSize=700

ySize=300

title=Apache Servers Busy Workers

xAxisName=Time

yAxisName=# of Workers Busy

</value>

</property>

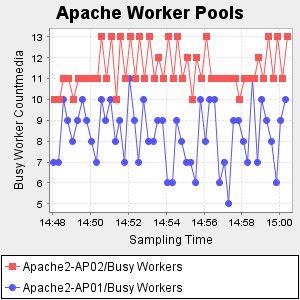
<property name="metricLocatorFilters" value=".\*/Busy Workers"/>

</bean>

呈现器非常容易实现，但是我发现，我经常需要对它们进行调整，因此这里列出的 Groovy 方法非常适用于对新的图表类型或者新的图形包进行快速原型化。编译 Groovy 代码时，性能非常好，实现出色的内容缓存也不是问题。动态热更新功能和功能强大的 Groovy 语法使动态更新变得非常轻松。随后，当我确定了呈现器要做的工作以及它应该支持哪些选项时，我将它们移植到 Java 代码。

指标名称由 org.runtimemonitoring.tracing.Trace 类生成。这个类的每个实例表示一个 ITracer 读操作，所以它封装了跟踪的值、时间戳和完整的名称空间。指标的名称就是完整的名称空间。在本例中，我所显示的指标是 WebServers/Apache/Apache2-AP01/Busy Workers，因此，在清单 23 的呈现器中定义的过滤器使用该指标来呈现。生成的 JPG 如图 26 所示：

图 26. 呈现的 Apache 繁忙 worker 线程

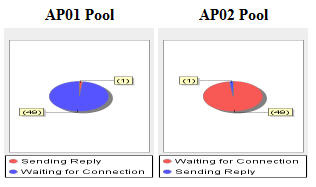


不同的客户机可能需要不同的呈现图。例如，一个客户机可能需要一个更小的图像。能够动态调整大小的图像通常比较合适。另一个客户机可能也需要一个更小的图像，但不想要标题（在其自己的 UI 中提供标题）。MediaServlet 允许在内容请求期间实现其他选项。这些选项被附加到内容请求的 URL 之后，并以 REST 格式进行处理。基本格式是**媒体** servlet 路径（可以进行配置）后接缓存名称，或 **/media/Apache2-All-BusyWorkers-Line**。每个呈现器能够支持不同的选项。对于我们上面使用的呈现器，以下选项就是一个很好的示例：

* **默认 URI**：/media/Apache2-All-BusyWorkers-Line
* **缩小到 300 X 300**：/media/Apache2-All-BusyWorkers-Line/300/300
* **缩小到 300 X 300，带有最小的标题和轴名称**：/media/Apache2-All-BusyWorkers-Line/300/300/BusyWorkers/Time/#Workers

图 27 展示了两个没有标题的缩小的饼图，使用了 URI Apache2-AP02-WorkerStatus-Pie/150/150/ /：

图 27. 缩小的 Apache Server worker 线程池图像



呈现器可以生成请求内容的客户机能够显示的几乎任何格式的内容。图像格式可以是 JPG、PNG 或 GIF。也支持其他图像格式，但是对于定位于 Web 浏览器客户机的静态图像，PNG 和 GIF 可能最适合。其他格式选择包括基于文本的标记，比如 HTML。浏览器和富客户机都能够呈现 HTML 片段，HTML 是显示独立数据字段和交叉表格的理想选择。纯文本也非常有用。例如，一个 Web 浏览器客户机可以从呈现器检索表示系统生成的事件消息的文本，并将其插入到文本框和列表框中。其他类型的标记也非常实用。许多呈现适用于浏览器的数据包的富客户机和客户端读入定义图形的 XML 文档，然后这些图形可以在客户端生成，这样可以优化性能。

客户端呈现能够提供额外的优化。如果客户机能够呈现自己的可视化，那么它就可以将缓存更新直接传递到客户机，绕过呈现器，除非需要使用呈现器来添加标记。通过这种方式，客户机可以订阅缓存更新事件，并接受这些事件，更新其自己的可视化。可以通过多种方式将数据传递到客户机。在浏览器客户机中，一种简单的 Ajax 风格的轮询程序能够周期性地检查服务器的更新，并实现一个能够将任何更新插入到数据结构中的处理程序来处理浏览器中的呈现。其他选项稍微复杂，涉及到使用 Comet 模式的实际数据流，要求一个到服务器的连接始终保持打开，而且当服务器写入数据时，这些数据就会被客户机读取（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。对于富客户机，使用一个消息传递系统是最理想的方法，客户机可以在其中订阅数据更新提要。ActiveMQ 既能够与 Jetty Web 服务器结合使用，又具有其 Comet 功能，因此可以创建一个基于浏览器的 JavaScript JMS 客户机并订阅队列和主题。

客户端上丰富的呈现选择还提供了扁平图像不具有的功能，比如单击元素以向下钻取的功能 — APM 指示板的一种常见需求，向下钻取用于导航或更详细地查看图表中的特定项。一个示例就是 Visifire，它是一种开源绘图工具，与 Silverlight（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）结合使用。清单 24 展示一个 XML 片段，该片段生成一个显示各个数据库服务器的 CPU 使用情况的条形图：

清单 24. 数据库平均 CPU 使用情况的图形呈现器

<vc:Chart xmlns:vc="clr-namespace:Visifire.Charts;assembly=Visifire.Charts"

Theme="Theme3">

<vc:Title Text="Average CPU Utilization on Database Servers"/>

<vc:AxisY Prefix="%" Title="Utilization"/>

<vc:DataSeries Name="Utilization" RenderAs="Column">

<vc:DataPoint AxisLabel="DB01" YValue="13"/>

<vc:DataPoint AxisLabel="DB02" YValue="57"/>

<vc:DataPoint AxisLabel="DB03" YValue="41"/>

<vc:DataPoint AxisLabel="DB04" YValue="10"/>

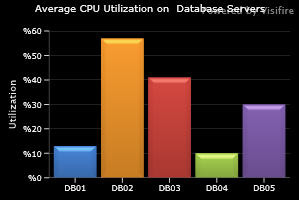
<vc:DataPoint AxisLabel="DB05" YValue="30"/>

</vc:DataSeries>

</vc:Chart>

这段 XML 非常普通，所以很容易为其创建一个呈现器，并且外观非常不错。客户端呈现器也可以为可视化添加动画效果，这对 APM 系统显示没什么价值，但是在一些情况下可能非常有帮助。图 28 展示了在启用了 Silverlight 客户机的浏览器中生成的图形：

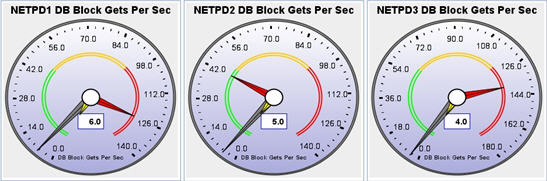
图 28. 一个 VisiFire Silverlight 呈现的图表



APM 指示板中包含所有标准图标类型。最常见的是折线图、多线图、条形图和饼图。一些图表通常是组合而成的，比如将条形图和折线图组合以显示两种不同类型数据的重叠情况。在另外一些情形中，一个折线图具有两条 y 轴，以便在同一个图表中表示数量差距较大的数据序列。绘制一个百分比值和一个标量值时通常属于这种情况，比如路由器上的 % CPU 利用率与传递的字节数。

在一些场景下，可以创建专门的小部件来以一种自定义方式表示数据，或者以一种直观的方式显示数据。例如，枚举的符号显示一个与监控目标状态一致的特定图标。因为状态通常由有限的值表示，使用图形表示有些大材小用了，所以可以采用一些类似交通灯的显示方式来表示，比如红色表示下降、黄色表示警告、绿色表示良好。另一个流行的小部件是刻度盘（通常表示为一种速度计）。我认为使用刻度盘浪费了屏幕空间，因为它们仅显示一个数据矢量，不包含历史记录。折线图既能显示相同的数据，还能够显示历史趋势。一个例外是，多指针的刻度盘能够显示高/低/当前状态等范围。但是在很大程度上，它们只能提供一种视觉上的吸引力，比如图 29 中的刻度盘，通过 “high/low/current” 显示了在过去一小时内每秒获得的数据库块缓冲区情况：

图 29. 示例刻度盘小部件



依我看来，可视化的好处在于信息密度高。屏幕空间是有限的，但我想要查看尽可能多的数据。可以通过多种方式来提高数据密度，但自定义图形表示是一种有趣的方式，它将多种维度的数据组合到一个小的图片之中。图 30 展示了一种不错的示例，该示例来自一个已经退役的数据库监控产品：

图 30. Savant 缓存命中率显示

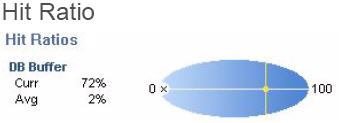
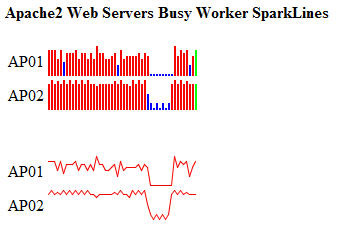


图 30 显示了与数据库缓冲命中率相关的一些数据矢量：

* 水平轴表示在缓存中找到的数据的百分比。
* 垂直轴表示当前的命中率。
* X 表示命中率趋势。
* 灰色的圆圈（在这个图像中几乎看不见）表示标准偏差。灰色圆圈的直径越大，缓存性能的偏差就越大。
* 黄色的球表示过去一小时内的缓存性能。

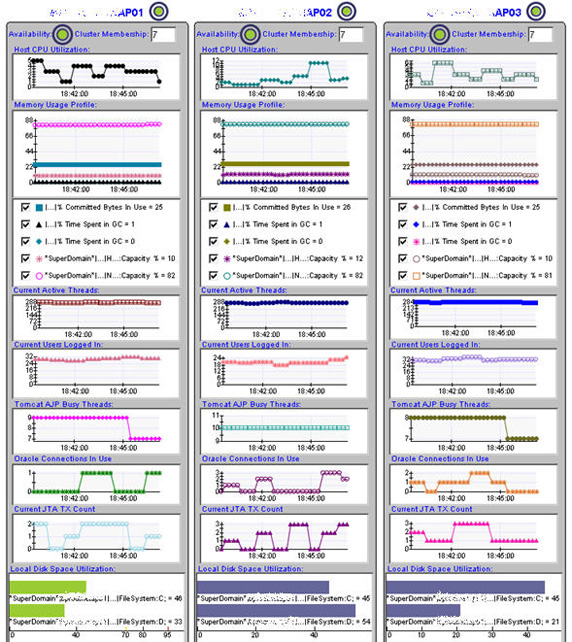
提高数据密度的第二种方法是 Sparkline，这是一种要求更低的方法。这个词是由数据可视化专家（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）根据 “嵌入到文字、数字和图像中的小型、高分辨率图形” 得出的。它通常用于显示大量的财务统计信息。尽管缺乏上下文，但它们的用途是显示跨许多指标的相对趋势。Sparkline 呈现器由 org.runtimemonitoring.spring.rendering.SparkLineRenderer 类实现，它为 Java 库实现开源的 Sparklines（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm3/#resources)）。图 31 中演示了两个 Sparklines，展示了基于条和线的显示效果：

图 31. 显示 Apache 2 繁忙 worker 线程的 Sparklines



此处和附加的代码中列出的示例都非常简单，但一个 APM 系统显然需要更高级和详细的指示板。而且，大多数用户都不希望从头创建指示板。APM 系统通常拥有某种指示板生成器，允许用户查看或搜索可用指标的存储库，并挑选要嵌入到指示板中的指标以及应该显示的格式。图 32 显示了我使用 APM 系统创建的指示板的一部分：

图 32. 指示板示例



[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

结束语

现在，[本系列](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) 已经全部结束了。我提供了一些指导、一些常见的性能监控技术，以及您能够实现的特定开发模式，可以使用这些模式增强或构建您自己的监控系统。收集和分析好的数据可以显著提高应用程序的正常运行时间和性能。我鼓励开发人员参与到监控生产应用程序的过程中来：要确定您编写的软件在负荷状态下运行时实际发生了什么，这是最好的信息来源。作为改进的一部分，这种反馈的价值不可估量。希望您的监控过程充满乐趣！

致谢

非常感谢 Sandeep Malhotra 在 Web 服务收集器方面提供的帮助。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm3/#ibm-pcon)

下载

| **描述** | **名字** | **大小** |
| --- | --- | --- |
| 本文的示例代码 | [j-rtm3.zip](http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=334508&filename=j-rtm3.zip&method=http&locale=zh_CN) | 316KB |

参考资料

学习

* 您可以参阅本文在 developerWorks 全球网站上的 [英文原文](http://www.ibm.com/developerworks/java/library/j-rtm3/)。
* 阅读整个 [Java 运行时监控](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) 系列。
* [Helios System Monitor](https://helios.dev.java.net/index.html)：本文中的示例代码和概念被融合到 Helios 开源监控解决方案中。
* [Linux 手册页](http://linux.die.net/man/)：查看 Linux 手册页，了解 ps、sar、iostat、vmstat 命令和更多信息。
* [Sysstat 实用工具主页](http://pagesperso-orange.fr/sebastien.godard/documentation.html)：查找常用 Linux 性能工具的文档和代码。
* [PostgreSQL 文档](http://www.postgresql.org/docs/)：阅读 PostgreSQL 文档。
* [Spring Framework JavaDoc](http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jmx/support/package-summary.html)：查看 JavaDoc，获得 Spring Framework 对 JMX 连接的支持。
* [MS0B: WebSphere MQ Java classes for PCF](http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg24000668&loc=en_US&cs=utf-8&lang=en)：一种用于在 WebSphere MQ 队列管理器上执行管理任务的 Java API。
* [Edward Tufte 的 Web 站点](http://www.edwardtufte.com/tufte/)：Tufte 是许多关于数据可视化的书籍的作者。
* “[面向 Java 开发人员的 Ajax: 使用 Jetty 和 Direct Web Remoting 编写可扩展的 Comet 应用程序](http://www.ibm.com/developerworks/cn/java/j-jettydwr/)”（Philip McCarthy，developerWorks，2007 年 7 月）：使用 Jetty 和 Direct Web Remoting 编写可伸缩的 Comet 应用程序。
* 在 [技术书店](http://www.ibm.com/developerworks/apps/SendTo?bookstore=safari) 中浏览关于这些主题和其他技术主题的图书。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：数百篇关于 Java 编程各个方面的文章。

获得产品和技术

* [JCraft](http://www.jcraft.com/)：访问 JCraft 开源 SSH Java 库的站点。
* [Apache Ant](http://ant.apache.org/)：Ant 是一种基于 Java 和 XML 的构建脚本工具，包含通过 SSH 执行远程 shell 命令的任务。
* [Nagios](http://www.nagios.org/)：Nagios 是一种开源网络监控程序。
* [IKVM](http://www.ikvm.net/)：IKVM 是一个编译器，它将 Java 字节码转换为 .NET CLR 字节码。
* [NSClient](http://nsclient.ready2run.nl/) 和 [NC\_Net](http://shatterit.com/nc_net/)：两个 Windows 服务，通过套接字连接提供 WPM 指标。
* [nsclient4j](https://nsclient4j.dev.java.net/)：NSClient 和 NC\_Net 的一种 Java 客户机 API。
* [Apache ActiveMQ](http://activemq.apache.org/)：来自 Apache Software Foundation 的一种开源 JMS 实现。
* [WebSphere MQ](http://www-306.ibm.com/software/integration/wmq/)：IBM® 的一种 JMS 兼容的消息传递服务。
* [OpenNMS](http://www.opennms.org/)：OpenNMS 是 joeSNMP Java API 的所有者。
* [RRDtool](http://oss.oetiker.ch/rrdtool/)：一种循环的时间序列数据存储工具。
* [MRTG](http://oss.oetiker.ch/mrtg/)：Multi Router Traffic Grapher 工具。
* [JFreeChart](http://www.jfree.org/jfreechart/)：一种用于从原始数据生成图表的工具。
* [Visifire](http://visifire.com/)：一种用于在 Silverlight 中呈现图表的工具。
* [Microsoft Silverlight](http://silverlight.net/)：一种基于标记的富客户机。
* [Sparklines for Java](http://www.representqueens.com/spark/)：一个关于使用 Java 代码创建 Sparkline 图形的开源库。
* [CA/Wily Introscope](http://www.wilytech.com/solutions/products/Introscope.html)：一个商用 Java 和 Web 应用程序性能管理解决方案。
* 访问 IBM Tivoli®[Monitoring for Transaction Performance](http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/index.jsp?toc=/com.ibm.itmtp.doc/toc.xml) 信息中心。

讨论

* 参与 [developerWorks blog](http://www.ibm.com/developerworks/blogs/) 并加入 [developerWorks 社区](http://www.ibm.com/developerworks/community/)。

# Java 运行时监控，第 2 部分: 编译后插装和性能监控

https://www.ibm.com/developerworks/cn/java/j-rtm2/

本系列包括三部分，介绍有关 Java™ 应用程序的运行时监控知识，[第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 重点介绍了 JVM 的健康状况和插装源代码以捕获性能指标的方法。本篇为第二部分，介绍了在无需改变原始源代码的情况下来插装 Java 类和构造的技巧。

[查看本系列更多内容](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) | 0https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/java/j-rtm2/#icomments)

[Nicholas Whitehead](https://www.ibm.com/developerworks/cn/java/j-rtm2/#authorN10030), 高级技术架构师, ADP

2008 年 8 月 22 日

* [expand](https://www.ibm.com/developerworks/cn/java/j-rtm2/#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-rtm2-_-sidebar)

简介

正如您在本系列（共三篇文章）的 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 中所了解到的，监控 Java 应用程序的可用性和性能及其生产中的依赖性，这对于确保问题检测和加速问题诊断和修复至关重要。需要进行监视的类的源代码级插装具有 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 所论述过的那些优势，但是这种方法通常都不可取或者不切实际。例如，很多您所感兴趣的监控点可能位于第三方组件中，而第三方组件的源代码您是不得而知的。在第 2 部分中，我着重介绍了无需修改原始源代码而插装 Java 类和资源的方法。

可选择的在源代码外编排插装的方法有：

* 截取
* 类包装
* 字节码插装

本文使用了 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 中呈现的 ITracer 接口来实现性能数据跟踪，依次举例阐明了这些技巧。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm2/#ibm-pcon)

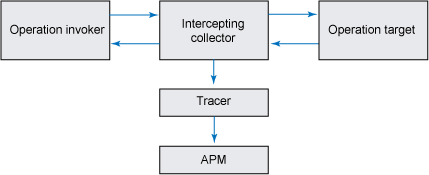
通过截取进行 Java 插装

截取 的基本前提是通过一个截取构造和收集传入的入站与出站调用信息，对特定的调用模式进行转换。一个基本的截取程序的实现会：

1. 获取对入站调用请求的当前时间。
2. 取回出站响应的当前时间。
3. 将运行时间作为两次度量的增量计算出来。
4. 将调用的运行时间提交给应用程序性能管理（APM）系统。

图 1 展示了该流程：

图 1. 性能数据收集截取程序的基本流程



清晰的界限

变更管理的爱好者可能会对通过源代码实现变更和通过配置实现变更之间的差异持有争议。诚然，“代码”、XML 和 “脚本” 之间的界限变得有些模糊了。但是下面两个变更之间还存在明显的界限：

* 需要改变源代码的变更，接着还要编译、打包，有时还会涉及到一系列看起来无休止的预部署过程
* 对（未改变的）二进制代码外部的资源所作的变更

这两种变更之间最主要的差异是实现前滚（roll-forward）和后滚（roll-back）的简单性。在某些情况下，这种差异可能在理论上说不通，或者可能低估了某些环境的复杂度或变更过程的严格性。

很多诸如 Java Platform 和 Enterprise Edition（Java EE）这样的 Java 框架都包括对截取栈的核心支持，服务的调用可以在截取栈中通过一系列预处理和后处理组件来进行传递。有了这些栈就可以很好地将插装注入到执行路径中，这样做的好处有二：第一，无需修改目标类的源代码；第二，只要将截取程序类插入到 JVM 的类路径中并修改组件的部署描述符，这样就把插装截取程序插入到了执行流程中。

截取的核心指标

截取程序所收集的一个典型的指标就是运行时间。其他的指标同样适合截取模式。我将介绍支持这些指标的 ITracer 接口的两个新的方面，所以在这里我要转下话题，先简要论述一下这些指标。

使用截取程序时需要收集的典型指标有：

* **运行时间**：完成一个执行的平均时钟时间。
* **每个时间间隔内的调用**：调用目标的次数。
* **每个时间间隔内的响应**：目标响应调用的次数。
* **每个时间间隔内的异常l**：目标调用导致异常的次数。
* **并发性**：并发执行目标的线程数。

还有两个 ThreadMXBean 指标可以选择，但它们的作用有限，而且收集成本会高一些：

* **运行 CPU 时间**：这是线程在执行期间消耗的 CPU 时间，以纳秒为单位。CPU 的利用情况在起初时似乎有用，但其实也就是作为一种趋势模式，其他的用处不大。或者，如果收集成本特别高的话，可以计算线程在执行时占用 CPU 资源的百分比的近似值。
* **阻塞/等待计数和时间**：等待表示由具体线程调度导致的同步或者等待。阻塞常见于执行等待资源时，如响应来自远程数据库的 Java 数据库连接（Java Database Connectivity，JDBC）调用（至于这些指标的用处，请参见本文的 [JDBC 插装](https://www.ibm.com/developerworks/cn/java/j-rtm2/#JDBCInstrumentation) 部分）。

为了澄清 ThreadMXBean 指标的收集方法，清单 1 快速回顾了基于源代码的插装。在这个例子中，我针对 heavilyInstrumentedMethod 方法实现了大量插装。

清单 1. 实现大量插装的方法

protected static AtomicInteger concurrency = new AtomicInteger();

.

.

for(int x = 0; x < loops; x++) {

tracer.startThreadInfoCapture(CPU+BLOCK+WAIT);

int c = concurrency.incrementAndGet();

tracer.trace(c, "Source Instrumentation", "heavilyInstrumentedMethod",

"Concurrent Invocations");

try {

// ===================================

// Here is the method

// ===================================

heavilyInstrumentedMethod(factor);

// ===================================

tracer.traceIncident("Source Instrumentation",

"heavilyInstrumentedMethod", "Responses");

} catch (Exception e) {

tracer.traceIncident("Source Instrumentation",

"heavilyInstrumentedMethod", "Exceptions");

} finally {

tracer.endThreadInfoCapture("Source Instrumentation",

"heavilyInstrumentedMethod");

c = concurrency.decrementAndGet();

tracer.trace(c, "Source Instrumentation",

"heavilyInstrumentedMethod", "Concurrent Invocations");

tracer.traceIncident("Source Instrumentation",

"heavilyInstrumentedMethod", "Invocations");

}

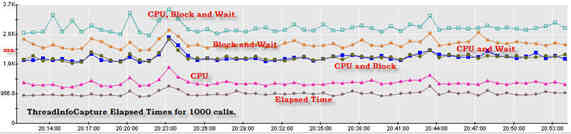
try { Thread.sleep(200); } catch (InterruptedException e) { }

}

[清单 1](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing1) 引入了两个新的构造：

* **ThreadInfoCapture 方法**：ThreadInfoCapture 方法对于获取运行时间和目标调用前后的 ThreadMXBean 指标增量都很有帮助。startThreadInfoCapture 为当前线程捕获基准，而 endThreadInfoCapture 计算出增量和趋势。由于这些指标永远都是递增的，所以必须事先确定一个基准，再根据它计算出之后的差量。但这个场景不适用于跟踪程序的增量功能，这是因为每一个线程的绝对值都是不同的，而且运行中的 JVM 中的线程也不是保持不变的。另外还要注意跟踪程序使用了一个栈来保存基准，所以您可以（小心地）嵌套调用。要收集这个数据可是要费一番力气。图 2 展示了收集各种 ThreadMXBean 指标所需要的相对运行时间：

图 2. 收集 ThreadMXBean 指标所需的相对成本

  
虽然如果小心使用调用的话，收集这些指标的总开销不会很大，但是仍然需要遵循在记录日志时需要考虑的一些事项，例如不要在紧凑循环（tight loop）内进行。

* **并发性**：要跟踪在特定时间内通过这个代码的线程数，需要创建一个计数器，该计数器既要是线程安全的又要对目标类的所有实例可用 — 在本例为 AtomicInteger 类。此种情况比较麻烦，因为有时可能多个类加载器都载入了该类，致使计数器无法精确计数，从而导致度量错误。解决这个问题的办法为：将并发计数器保存在 JVM 的一个特定的受保护位置中，诸如平台代理中的 MBean。

并发性只有在插装目标是多线程的或者共用的情况下可用，但是它是非常重要的指标，这一点我将在稍后介绍 Enterprise JavaBean（EJB）截取程序时进一步阐述。EJB 截取程序是我接下来要论述的几个基于截取的插装示例的第一个，借鉴了 [清单 1](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing1) 中查看的跟踪方法。

EJB 3 截取程序

发布了 EJB 3 后，截取程序就成了 Java EE 架构中的标准功能（有些 Java 应用服务器支持了 EJB 截取程序一段时间）。大多数 Java EE 应用服务器的确提供了性能指标，报告有关诸如 EJB 这样的主要组件，但是仍然需要实现自己的性能指标，因为：

* 您需要基于上下文的或者基于范围/阈值的跟踪。
* 应用服务器指标固然不错，但是您希望指标位于 APM 系统中，而不是应用服务器中。
* 应用服务器指标无法满足您的要求。

虽然如此，根据您的 APM 系统和应用服务器实现的不同，有些工作可能不用您再亲历亲为了。例如，WebSphere® PMI 通过 Java 管理扩展（Java Management Extensions，JMX）公开了服务器指标（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）。即使您的 APM 供应商没有提供自动读取这个数据的功能，读完本篇文章之后您也会知道如何自行读取。

在下一个例子中，我将向一个称为 org.aa4h.ejb.HibernateService 的无状态会话的上下文 bean 中注入一个截取程序。EJB 3 截取程序的要求和依赖性都是相当小的：

* **接口**：javax.interceptor.InvocationContext
* **注释**：javax.interceptor.AroundInvoke
* **目标方法**：任何一个名称里面有 public Object anyName(InvocationContext ic) 的方法

清单 2 展示了样例 EJB 的截取方法：

清单 2. EJB 3 截取程序方法

@AroundInvoke

public Object trace(InvocationContext ctx) throws Exception {

Object returnValue = null;

int concur = concurrency.incrementAndGet();

tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getClass()

.getName(), ctx.getMethod().getName(),

"Concurrent Invocations");

try {

tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);

// ===================================

// This is the target.

// ===================================

returnValue = ctx.proceed();

// ===================================

tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()

.getName(), ctx.getMethod().getName(), "Responses");

concur = concurrency.decrementAndGet();

tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getClass()

.getName(), ctx.getMethod().getName(),

"Concurrent Invocations");

return returnValue;

} catch (Exception e) {

tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()

.getName(), ctx.getMethod().getName(), "Exceptions");

throw e;

} finally {

tracer.endThreadInfoCapture("EJB Interceptors", ctx.getTarget()

.getClass().getName(), ctx.getMethod().getName());

tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()

.getName(), ctx.getMethod().getName(), "Invocations");

}

}

如 [清单 1](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing1) 一样，[清单 2](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing2) 包含一个大的插装集，一般不推荐使用，此处仅作为一个例子使用。清单 2 中有以下几点需要注意：

* @AroundInvoke 注释通过封装 EJB 调用而将方法标记为一个截取程序。
* 方法调用一直沿着栈传递调用，可能传递到最终目标，或到下一个截取程序。因此，要在调用该方法前确定度量基准，在调用后跟踪它。
* 传入跟踪方法的 InvocationContext 为截取程序提供全部有关调用的元数据，包括：
  + 目标对象
  + 目标方法名
  + 所传递的参数

注意到这点是很重要的，因为该截取程序可以应用于很多不同的 EJB，因此要截取什么类型的调用是无法预知的。拥有一个可以从截取程序内部访问的元数据源是至关重要的：没有这个源的话，只能得到很少的关于被截取调用的信息；您的指标可以展示出很多有趣的趋势，但却无法明确它们所指的是哪个操作。

从插装的角度看，这些截取程序最有用之处在于您可以通过修改部署描述符而将它们应用于 EJB。清单 3 展示了样例 EJB 的 ejb-jar.xml 部署描述符：

清单 3. EJB 3 截取程序部署描述符

<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/ejb-jar\_3\_0.xsd" version="3.0">

<interceptors>

<interceptor>

<interceptor-class>

org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor

</interceptor-class>

<around-invoke>

<method-name>trace</method-name>

</around-invoke>

</interceptor>

</interceptors>

<assembly-descriptor>

<interceptor-binding>

<ejb-name>AA4H-HibernateService</ejb-name>

<interceptor-class>

org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor

</interceptor-class>

</interceptor-binding>

</assembly-descriptor>

</ejb-jar>

正如我在前面所提到过的，插装截取程序对于基于上下文或者基于范围/阈值的跟踪是有用的。而 InvocationContext 中的 EJB 调用参数值是可用的，这加强了插装截取程序的作用。这些值可以用于跟踪范围或其他上下文的复合名称。考虑一下包含有issueRemoteOperation（String region、Command command） 方法的 org.myco.regional.RemoteManagement 类中的 EJB 调用。EJB 接受一个命令，然后远程调用根据域识别的服务器。在这个场景中，区域服务器遍布于一个广泛的地理区域，每一个区域服务都有自己的 WAN 特性。这里呈现的模式与 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-rtm1/) 中的 payroll-processing 例子类似，这是因为如果没有明确命令到底被分配到哪一个区域的话，确定一个 EJB 调用的运行时间是很困难的。您可能已经预料到，从距离一个洲远的区域调用的运行时间要比从隔壁调用的运行时间要长的多。但是您是可以从 InvocationContext 参数确定区域的，因此您只需将区域代码添加到跟踪复合名称并按区域划分性能数据，如清单 4 所示：

清单 4. EJB 3 截取程序实现上下文跟踪

String[] prefix = null;

if(ctx.getTarget().getClass().getName()

.equals("org.myco.regional.RemoteManagement") &&

ctx.getMethod().getName().equals("issueRemoteOperation")) {

prefix = new String[]{"RemoteManagement",

ctx.getParameters()[0].toString(),

"issueRemoteOperation"};

}

// Now add prefix to the tracing compound name

Servlet 过滤器截取程序

Java Servlet API 提供了一个叫做过滤器（filter）的构造，它与 EJB 3 截取程序非常类似，含有无需源代码的注入和元数据可用性。清单 5 展示了一个过滤器的 doFilter 方法，带有缩略了的插装。指标的复合名由过滤器类名和请求的统一资源标识符（Uniform Resource Identifier，URI）构建：

清单 5. servlet 过滤器截取程序方法

public void doFilter(ServletRequest req, ServletResponse resp,

FilterChain filterChain) throws IOException, ServletException {

String uri = null;

try {

uri = ((HttpServletRequest)req).getRequestURI();

tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);

// ===================================

// This is the target.

// ===================================

filterChain.doFilter(req, resp);

// ===================================

} catch (Exception e) {

} finally {

tracer.endThreadInfoCapture("Servlets", getClass().getName(), uri);

}

}

清单 6 展示了清单 5 的过滤器的 web.xml 部署描述符的相关片断：

清单 6. servlet 过滤器部署描述符

<web-app >

<filter>

<filter-name>ITraceFilter</filter-name>

<display-name>ITraceFilter</display-name>

<filter-class>org.myco.http.ITraceFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>ITraceFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

</web-app>

EJB 客户端截取程序与上下文传递

前面的例子侧重于服务器端组件，但一些诸如客户端截取这样的插装实现方法也是存在的。Ajax 客户机可以注册度量 XMLHttpRequest 运行时间的性能监听器，并可以在下一个请求的参数列表末尾承载请求的 URI（对于复合名称）和运行时间。有些 Java EE 服务器，如 JBoss，允许使用客户端的截取程序，本质上这些截取程序与 EJB 3 截取程序所作的工作相同，并且它们也能够承载下一个请求中的度量提交。

监控中的客户端通常都会被忽视。所以下次听到用户抱怨您的应用程序太慢时，不要因为服务器端的监控确保服务器端是良好的就无视这些抱怨。客户端的插装可以确保您所度量的正是用户所体验的，它们可能不会总是与服务器端的指标一致。

一些 Java EE 实现支持的客户端截取程序被实例化并绑定在 EJB 的客户端。这意味着如果一个远程客户机通过远程方法调用（Remote Method Invocation，RMI）协议调用服务器上的 EJB，则也可以从远程客户机无缝收集到性能数据。在远程调用的任一端实现截取程序类都会实现在两者间传递上下文的能力，从而获取额外的性能数据。

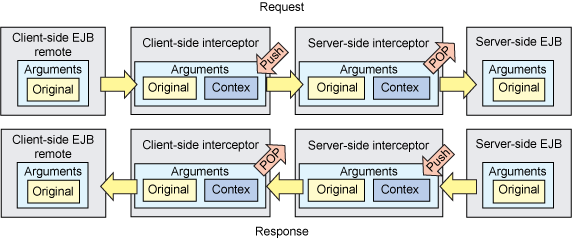
下面的例子展示了一对截取程序，它们共享数据并获得传输时间（传送请求和响应的运行时间）以及客户机方面对服务器的远程请求的响应时间。该例子使用了 JBoss 应用服务器的客户端和服务器端的 EJB 3 截取程序专有的实现。

这对截取程序通过在相同负载内承载上下文数据，将上下文数据作为 EJB 调用传递到同一个调用。上下文数据包括：

* **客户端发出请求的时间**：EJB 客户机截取程序发出请求时的请求的时间戳
* **服务器端接收请求的时间**：EJB 服务器端截取程序接收请求时的请求的时间戳
* **服务器端发送响应的时间**：EJB 服务器端截取程序将响应回送给客户机时的响应的时间戳

调用参数被当作一个栈结构，上下文数据通过这个结构进出参数。上下文数据由客户端截取程序放入该调用中，再由服务器端截取程序取出，然后传入到 EJB 服务器 stub。数据返回时则按此过程的逆向过程传递。图 3 展示了这个流程：

图3. 客户机和服务器 EJB 截取程序的数据流



为这个例子构建截取程序需要为客户机和服务器实现 org.jboss.aop.advice.Interceptor 接口。该接口有一个重要的方法：

public abstract java.lang.Object invoke(

org.jboss.aop.joinpoint.Invocation invocation) throws java.lang.Throwable

这个方法引入了调用封装 的理念，根据这个理念，一个方法的执行被封装成为一个独立对象，它表示以下内容：

* 目标类
* 要调用的方法名
* 由作为实参传入目标方法的参数组成的负载

接着这个对象可以被继续传递，直至传递到调用方，调用方解组调用对象并针对端点目标对象实现动态执行。

客户端截取程序将当前请求时间添加到调用上下文，而服务器端截取程序则负责添加接收请求的时间戳和发送响应的时间戳。或者，服务器可以获得客户机请求，由客户机计算出请求和来回传输的总运行时间。每种情况的计算方法为：

* **客户端，向上传输时间**等于 ServerSideReceivedTime 减去 ClientSideRequestTime
* **客户端，向下传输时间**等于 ClientSideReceivedTime 减去 ServerSideRespondTime
* **服务器端，向上传输时间**等于 ServerSideReceivedTime 减去 ClientSideRequestTime

清单 7 展示了客户端截取程序的 invoke 方法：

清单 7. 客户端截取程序的 invoke 方法

/\*\*

\* The interception invocation point.

\* @param invocation The encapsulated invocation.

\* @return The return value of the invocation.

\* @throws Throwable

\* @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)

\*/

public Object invoke(Invocation invocation) throws Throwable {

if(invocation instanceof MethodInvocation) {

getInvocationContext().put(CLIENT\_REQUEST\_TIME, System.currentTimeMillis());

Object returnValue = clientInvoke((MethodInvocation)invocation);

long clientResponseTime = System.currentTimeMillis();

Map<String, Serializable> context = getInvocationContext();

long clientRequestTime = (Long)context.get(CLIENT\_REQUEST\_TIME);

long serverReceiveTime = (Long)context.get(SERVER\_RECEIVED\_TIME);

long serverResponseTime = (Long)context.get(SERVER\_RESPOND\_TIME);

long transportUp = serverReceiveTime-clientRequestTime;

long transportDown = serverResponseTime-clientResponseTime;

long totalElapsed = clientResponseTime-clientRequestTime;

String methodName = ((MethodInvocation)invocation).getActualMethod().getName();

String className = ((MethodInvocation)invocation).getActualMethod()

.getDeclaringClass().getSimpleName();

ITracer tracer = TracerFactory.getInstance();

tracer.trace(transportUp, "EJB Client", className, methodName,

"Transport Up", transportUp);

tracer.trace(transportDown, "EJB Client", className, methodName,

"Transport Down", transportDown);

tracer.trace(totalElapsed, "EJB Client", className, methodName,

"Total Elapsed", totalElapsed);

return returnValue;

} else {

return invocation.invokeNext();

}

}

JBoss EJB 3 截取程序

JBoss 的 EJB 2 截取程序架构内置了传递任意负载的能力；它的目标是在 EJB 3 中交付，但效果不是很好。所以我实现了截取程序，从而将上下文负载作为请求的附加调用参数来传递。并且将响应对象编组为一个Object[2] 数组；第一项是 “real” 结果，第二项为上下文。在这两种情况下，被编组的对象都被对应的截取程序解组，所以请求方和服务端点都能获得它们所需要的类型。

服务器端截取程序在概念上是类似的，不同的是为了避免使例子过于复杂，它使用了本地线程来检查 reentrancy — 相同的请求处理线程在同一远程调用中不只一次调用相同的 EJB（和截取程序）。该截取程序忽略了除第一个请求之外的所有请求的跟踪和上下文处理。清单 8 展示了服务器端截取程序的 invoke 方法：

清单 8. 服务器端截取程序的 invoke 方法

/\*\*

\* The interception invocation point.

\* @param invocation The encapsulated invocation.

\* @return The return value of the invocation.

\* @throws Throwable

\* @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)

\*/

public Object invoke(Invocation invocation) throws Throwable {

Boolean reentrant = reentrancy.get();

if((reentrant==null || reentrant==false)

&& invocation instanceof MethodInvocation) {

try {

long currentTime = System.currentTimeMillis();

MethodInvocation mi = (MethodInvocation)invocation;

reentrancy.set(true);

Map<String, Serializable> context = getInvocationContext(mi);

context.put(SERVER\_RECEIVED\_TIME, currentTime);

Object returnValue = serverInvoke((MethodInvocation)mi);

context.put(SERVER\_RESPOND\_TIME, System.currentTimeMillis());

return addContextReturnValue(returnValue);

} finally {

reentrancy.set(false);

}

} else {

return invocation.invokeNext();

}

}

JBoss 通过面向方面的编程（aspect-oriented programming，AOP）（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）技术来应用截取程序，该技术读取名为 ejb3-interceptors-aop.xml 的指令文件并根据其中定义的指令应用截取程序。JBoss 使用这种 AOP 技术在运行时将 Java EE 核心规则应用于 EJB 3 类。因此，除了性能监控截取程序之外，该文件还包含了关于事务管理、安全性和持久性这样的指令。客户端指令则相当简单明了。它们被简单地定义为包含一系列截取程序类名的 stack name XML 元素。每一个在此定义的类名同时都有资格作为 PER\_VM 或 PER\_INSTANCE 截取程序，这表明每一个 EJB 实例都应该共享一个截取程序实例或者具有各自的非共享实例。针对性能监控截取程序的目标，则应该确定此项配置，无论截取程序代码是否是线程安全的。如果截取程序代码能够安全地并行处理多个线程，那么使用 PER\_VM 策略更有效，而对于线程安全但是效率较低的策略，则可以使用 PER\_INSTANCE。

服务器端的截取程序的配置要相对复杂一些。截取程序要依照一组语法模式和用 XML 定义的过滤器来应用。如果所关注的特定的 EJB 方法与定义的模式相符的话，那么为该模式定义的截取程序就会被应用。服务器端截取程序能够通过进一步细化定义来将部署的 EJB 的特定子集定为目标。对于客户端截取程序，您可以通过创建一个新的特定于目标 bean 的 stack name 来实现自定义栈。而在服务器端，自定义栈可以在一个新的 domain 中进行定义。个别 EJB 的关联客户机 stack name 和服务器栈 domain 可以在 EJB 的注释中指定。或者，如果您不能或是不想修改源代码的话，这些信息可以在 EJB 的部署描述符中指定或者跳过。清单 9 展示了一个删减的用于此例的 ejb3-interceptors-aop.xml 文件：

清单 9. 经过删减的 EJB 3 AOP 配置

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE aop PUBLIC

"-//JBoss//DTD JBOSS AOP 1.0//EN"

"http://labs.jboss.com/portal/jbossaop/dtd/jboss-aop\_1\_0.dtd">

<aop>

.

.

<interceptor

class="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"

scope="PER\_VM"/>

.

.

<stack name="StatelessSessionClientInterceptors">

<interceptor-ref

name="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"/>

.

.

</stack>

.

.

<interceptor

class="org.runtimemonitoring.ejb.interceptors.ServerContextualInterceptor"

scope="PER\_VM"/>

.

.

<domain name="Stateless Bean">

<bind pointcut="execution(public \* \*->\*(..))">

<interceptor-ref name="org.aa4h.ejb.interceptors.ServerContextualInterceptor"/>

.

.

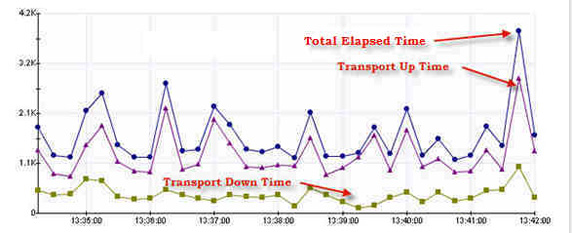
</bind>

</domain>

</aop>

这种性能数据收集方法可以一箭双标。首先，它可以告诉您从客户机的角度看，一个 EJB 目前的性能是多少。再者，如果性能滞后的话，传输时间可以很好地指明是否是由客户机和服务器间的网络连接速度缓慢而导致的。图 4 展示了总运行时间和上/下传输指标，该指标是从客户机度量的，度量方法是在客户机和服务器之间使用一个人为减缓的网络连接来突出显示传输时间：

图 4. 上下文客户机截取程序性能指标



使用客户端截取程序时，客户机截取程序类本身必须处于客户机应用程序的类路径中。或者一定要启用从服务器载入的远程类，这样才能够在启动时将客户端截取程序及其依赖项下载到客户机上。如果您的客户机系统时钟不是完全 与服务器系统时钟同步的话，您就会得到与两个时钟的时间差大小成正比的特殊结果。

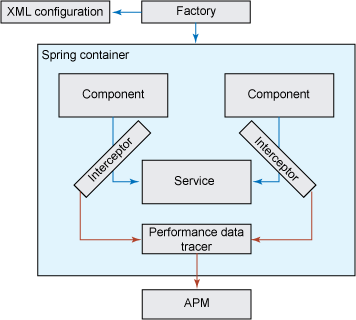
Spring 中的截取程序

尽管 Java EE 提供丰富的正交无缝截取方法，但很多流行的非 Java EE 容器同样支持隐式的和显式的截取。我之所以使用容器 这个词是想暗指某种使用或鼓励使用松散耦合的框架。只要不使用紧密耦合，就能够实现截取。这种类型的框架通常称为依赖注入 或者 Inversion of Control（IoC） 架构。它们让您能够在外部定义个别组件如何 “粘合” 在一起，而不是硬编码组件，从而实现组件间的之间通信。我将使用流行的 IoC 框架 Spring Framework（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）中的跟踪截取程序来查看性能数据的收集，以此结束对截取的讨论。

Spring Framework 让您能够使用普通初始 Java 对象（Plain Old Java Object，POJO）来构建应用程序。POJO 仅包含业务逻辑，而 Spring 框架添加了构建企业应用程序所需的内容。如果在最初编写 Java 应用程序时没有考虑插装的话，Spring 的分层架构是很有用处的。虽然将应用程序架构备份到 Spring 并非一无是处，但除一系列的 Java EE 和 AOP 集成外，还有 Spring 的 POJO 管理特性足以将普通硬连接的 Java 类委托给 Spring 的容器管理功能。您可以通过截取添加性能插装，无需修改目标类的源代码。

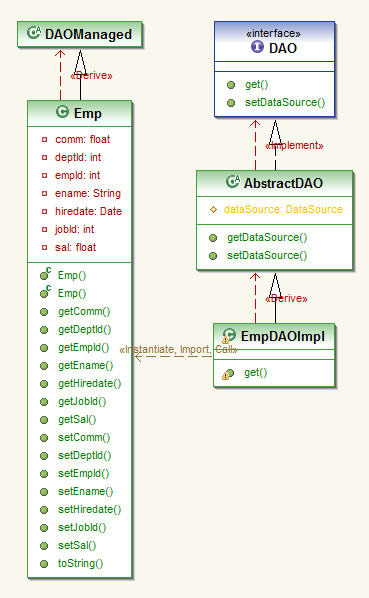
Spring 通常被描述为 IoC 容器，这是因为它颠倒了 Java 应用程序的传统拓扑结构。在传统的拓扑中，会有一个中心的程序或控制线程按照程序载入全部需要的组件和依赖项。容器用 IoC 载入几个组件，并依照外部配置管理组件间的依赖项。此处的依赖项管理称为依赖项注入，因为依赖项（如 JDBC DataSource）是通过容器注入组件的；组件无需寻找到它们自己的依赖项。为了进行插装，容器的配置可以轻易修改，从而将截取程序插入到这些组件间的 “结缔组织” 中。图 5 解释了该概念：

图 5. Spring 和截取概观



现在我将要展示一个简单的用 Spring 截取的例子。它涉及一个 EmpDAOImpl 类，该类是一个基本的数据访问对象（data access object，DAO）模式类，它实现了一个定义了名为 public Map<Integer, ? extends DAOManaged> get(Integer...pks) 的方法的 DAO 接口。该接口要求我传入一组主键作为完整的对象，DAO 实现将返回对象的 Map。这个代码中的依赖项的列表太长了，无法在此详细说明。可以肯定地说，它没有提供插装的供应，并且不使用任何种类的对象关系映射（object-relational mapping，ORM）框架。图 6 描述出了该类结构的轮廓。参见 [下载](https://www.ibm.com/developerworks/cn/java/j-rtm2/#download)，获取此处提及的工件的完整源代码和文本文件。

图 6. EmpDAO 类



EmpDAOImpl 在由 spring.xml 文件配置时被部署到 Spring 容器，清单 10 中展示了该文件的一小部分：

清单 10. Spring 例子的基本容器配置

<beans>

<bean id="tracingInterceptor"

class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">

<property name="interceptorName" value="Intercepted DAO"/>

</bean>

<bean id="tracingOptimizedInterceptor"

class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">

<property name="interceptorName" value="Optimized Intercepted DAO"/>

</bean>

<bean id="DataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:url="jdbc:postgresql://DBSERVER:5432/runtime"

p:driverClassName="org.postgresql.Driver"

p:username="scott"

p:password="tiger"

p:initial-size="2"

p:max-active="5"

p:pool-prepared-statements="true"

p:validation-query="SELECT CURRENT\_TIMESTAMP"

p:test-on-borrow="false"

p:test-while-idle="false"/>

<bean id="EmployeeDAO" class="org.runtimemonitoring.spring.EmpDAOImpl"

p:dataSource-ref="DataSource"/>

<bean id="empDao" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="org.runtimemonitoring.spring.DAO"/>

<property name="target" ref="EmployeeDAO"/>

<property name="interceptorNames">

<list>

<idref local="tracingInterceptor"/>

</list>

</property>

</bean>

<bean id="empDaoOptimized"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="EmployeeDAO"/>

<property name="optimize">

<value>true</value>

</property>

<property name="proxyTargetClass">

<value>true</value>

</property>

<property name="interceptorNames">

<list>

<idref local="tracingOptimizedInterceptor"/>

</list>

</property>

</bean>

</beans>

被部署的还有其他几个对象。这些组件通过引用它们的 Spring bean id 来描述，这些 bean id 在清单 10 中的每一个 bean 元素中都可以看得见：

* tracingInterceptor 和 tracingOptimizedInterceptor：两个 SpringTracingInterceptor 类型的截取程序。这个类包含了将收集到的数据跟踪到 APM 系统的 ITracer 调用。
* DataSource：一个将 JDBC 连接汇合到名为 runtime 的样例数据库的 JDBC DataSource，该样例数据库将会被注入到 EmpDAOImpl。
* EmployeeDAO：我将调用的 EmpDAOImpl 将作为例子的一部分调用。
* empDao 和 empDaoOptimized：spring.xml 文件中定义的最后两个 bean 为 Spring ProxyFactoryBean。它们本质上是 EmpDAOImpl 的代理，且每一个都各自引用一个截取程序。虽然 EmpDAOImpl 可以直接访问，但是使用代理会调用截取程序并生成性能指标。清单 10 中的两个代理和截取程序说明了一些差别和配置考虑。参见 [优化的截取程序](https://www.ibm.com/developerworks/cn/java/j-rtm2/##OptimizedInterceptors) 侧边栏。

优化的截取程序

标准截取程序和 [清单 10](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing10) 中的优化的截取程序之间的差异就在于优化的代理的附加的 <property name="optimize"><value>true</value></property> 属性。没有这个属性，代理使用java.lang.reflect.Proxy 的反射来调用截取程序。使用了这个选项，Spring 就可以使用名为 CGLIB 的字节码插装库为一个直接（非反射的）调用者创建字节码。在这样的情况下，优化了字节码的解决方案通常会比动态代理的性能好，但是由于最近的 JVM 中的 Java 反射有了相当大的进步，因此情况大为不同。

Spring 容器是从 SpringRunner 类中引导出来的。它还会启动一个测试循环，针对四个目标调用 DAO.get：

* EmployeeDAO Spring bean，它代表一个未用 Spring 插装的托管 DAO。
* empDao Spring bean，它代表一个用 Spring 插装的托管的带有标准截取程序的 DAO。
* empDaoOptimized Spring bean，它代表一个用 Spring 插装的托管的带有优化截取程序的 DAO。
* 一个非 Spring 管理的 EmpDAOImpl，与 Spring 管理的 bean 相对。

Spring 通过一个名为 org.aopalliance.intercept.MethodInterceptor 的接口实现这些类型的截取程序。要实现的方法只有一个：public Object invoke（MethodInvocation invocation）throws Throwable。MethodInvocation 对象提供了两个关键项：带有某种上下文（即正在被截取的方法名）的跟踪程序和proceed 方法，该方法将调用向前引领到指定目标。

清单 11 展示了 SpringTracingInterceptor 类的 invoke 方法。在这种情况下是不需要 interceptorName 属性的，但是我还是添加了这个属性，目的是为这个例子提供辅助的上下文。对于一个多用途的截取程序实现，跟踪程序通常都会将类名添加到跟踪上下文，这样所有被截取的类中的所有方法都会被跟踪到单独的 APM 名称空间中。

清单 11. SpringTracingInterceptor 类的 invoke 方法

public Object invoke(MethodInvocation invocation) throws Throwable {

String methodName = invocation.getMethod().getName();

tracer.startThreadInfoCapture(WAIT+BLOCK);

Object returnValue = invocation.proceed();

tracer.endThreadInfoCapture("Spring", "DAO",

interceptorName, methodName);

tracer.traceIncident("Spring", "DAO", interceptorName,

methodName, "Responses Per Interval");

return returnValue;

}

SpringRunner 类是这个例子的主入口点。它初始化 Spring bean 工厂，然后开始一个长的循环，从而将负载置于每一个 bean 中。清单 12 展示了该循环的代码。注意由于 daoNoInterceptor 和 daoDirect 不是通过 Spring 的截取程序插装的，所以我在 SpringRunner 循环中手动添加了插装。

清单 12. 缩略的 SpringRunner 循环

Map<Integer, ? extends DAOManaged> emps = null;

DAO daoIntercepted = (DAO) bf.getBean("empDao");

DAO daoOptimizedIntercepted = (DAO) bf.getBean("empDaoOptimized");

DAO daoNoInterceptor = (DAO) bf.getBean("EmployeeDAO");

DataSource dataSource = (DataSource) bf.getBean("DataSource");

DAO daoDirect = new EmpDAOImpl();

// Not Spring Managed, so dependency is set manually

daoDirect.setDataSource(dataSource);

for(int i = 0; i < 100000; i++) {

emps = daoIntercepted.get(empIds);

log("(Interceptor) Acquired ", emps.size(), " Employees");

emps = daoOptimizedIntercepted.get(empIds);

log("(Optimized Interceptor) Acquired ", emps.size(), "

Employees");

tracer.startThreadInfoCapture(WAIT+BLOCK);

emps = daoNoInterceptor.get(empIds);

log("(Non Intercepted) Acquired ", emps.size(), " Employees");

tracer.endThreadInfoCapture("Spring", "DAO",

"No Interceptor DAO", "get");

tracer.traceIncident("Spring", "DAO",

"No Interceptor DAO", "get", "Responses Per Interval");

tracer.startThreadInfoCapture(WAIT+BLOCK);

emps = daoDirect.get(empIds);

log("(Direct) Acquired ", emps.size(), " Employees");

tracer.endThreadInfoCapture("Spring", "DAO",

"Direct", "get");

tracer.traceIncident("Spring", "DAO", "Direct",

"get", "Responses Per Interval");

}

由 APM 系统报告的结果展示出了几个类似的项。表 1 表明了来自每一个 Spring bean 的调用在测试运行中的平均运行时间：

表 1. Spring 截取程序测试运行结果

| **Spring bean** | **平均运行时间（ms）** | **最小运行时间（ms）** | **最大运行时间（ms）** | **计数** |
| --- | --- | --- | --- | --- |
| **直接** | 145 | 124 | 906 | 5110 |
| **优化的截取程序** | 145 | 125 | 906 | 5110 |
| **无截取程序** | 145 | 124 | 891 | 5110 |
| **截取程序** | 155 | 125 | 952 | 5110 |

图 7 显示了在 APM 中为测试用例创建的指标树。

图 7. Spring 截取程序在测试运行中的 APM 指标树

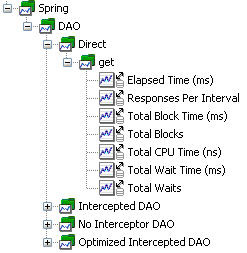
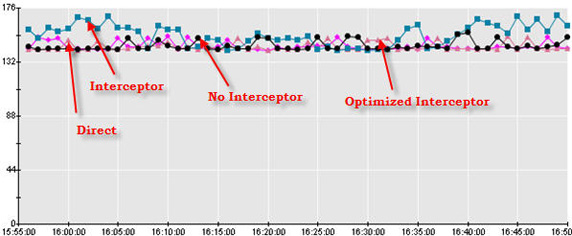


图 8 以图表的形式显示了该数据：

图 8. Spring 截取程序测试运行结果



很明显，这些结果相当紧密地聚集在了一起，但有一些模式显现了出来。优化的截取程序的确稍微胜过了未优化的截取程序。然而，在这个测试运行中只运行了一个线程，所以比较分析的用处并不大。在下一节中，我将详述这个测验用例并实现多个线程。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm2/#ibm-pcon)

通过类包装实现的 JDBC 插装

我发现造成大多数典型企业 Java 应用程序的慢性性能问题的根本原因在于数据库接口。通过 JDBC 的数据库调用或许是最普通的从 JVM 到外部服务的调用，目的是获取 JVM 中在本地不可用的数据集或资源。所以问题的起因在于数据库接口也不足为奇。逻辑上，在这种场景中可能出现问题的是数据库客户机、数据库本身或者两者兼有。然而，很多数据库的面向客户机的应用程序被许多性能反模式所困扰，包括：

* 逻辑上正确但执行很差的 SQL。
* 请求不够具体，致使检索到的数据要比所需的数据多得多。
* 频繁地检索相同的冗余数据。
* 请求基数小，导致大量数据库请求为一个逻辑结构检索数据，而不是少数的请求有效地检索同一数据集（本人的数据库访问原则是宁可一个请求返回很多行和列，也不要多个请求检索较短、较窄的数据集）。这个模式经常用于嵌套类结构，试图应用正统的封装概念（规定每一个对象管理它自己的数据检索，而不可以委托给一个公用的统一的数据请求者）的开发人员也会使用到。

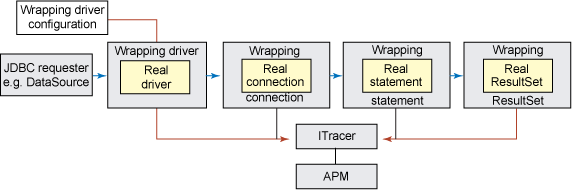
我当然不会违背每一个实例中的应用程序代码和设计，在本系列的第 3 部分中，我将展示监控数据库以进行性能统计的方法。但是基本上最有效的解决方案往往在客户机一边。因此，要监控 Java 应用程序中的数据库接口性能，最好的监控目标就是 JDBC。

我将展示如何使用类包装 的概念插装 JDBC 客户机。类包装背后的理念是：目标类可以包装在一层插装代码中，后者具有与被包装的类相同的外部行为。这些场景的难题就在于怎样可以在不改变依赖结构的情况下，无缝地引入被包装的类。

在这个例子中，我利用了 JDBC 首先是一个完全由接口定义的 API 这一事实：规范包括的具体类很少，而且 JDBC 的架构排除了直接紧密耦合到特定于数据库供应商提供的类的必要性。JDBC 的具体实现是被隐式加载的，而且源代码很少直接引用这些具体类。正因为如此，您可以定义一个全新的无实现的 JDBC 驱动程序，无需将所有针对它的调用全部委托给下面的 “真正的” 驱动程序，并在过程中收集性能数据。

我构建了一个名为 WrappingJDBCDriver 的实现，它足可以展示性能数据收集和支持前面的 [Spring 例子](https://www.ibm.com/developerworks/cn/java/j-rtm2/#spring) 中的 EmployeeDAO 测试用例。图 9 展示了 WrappingJDBCDriver 的总体工作方式：

图 9. WrappingJDBCDriver 概览



载入 JDBC 驱动程序的标准过程需要两项：驱动程序的类名和连接的目标数据库的 JDBC URL。驱动程序的加载程序载入驱动程序类（可能是通过调用 Class.forName(jdbcDriverClassName)）。大多数 JDBC 驱动程序会在类加载时用 JDBC java.sql.DriverManager 注册自己。然后驱动程序加载程序将 JDBC URL 传入 JDBC 驱动程序的一个实例中，以测试驱动程序是否接受该 URL。假定 URL 被接受，加载程序就能够对驱动程序调用 connect 并返回一个 java.sql.Connection。

包装的驱动程序的类名为 org.runtimemonitoring.jdbc.WrappingJDBCDriver。当被实例化时，它会从类路径中载入一个名为 wrapped-driver.xml 的配置文件。该文件包含插装配置项，配置项使用与目标驱动程序相关的形象化（figurative）名称索引：

* **<Figurative Name>.driver.prefix**：JDBC 驱动程序的真正的 JDBC URL 前缀 — 例如，jdbc.postgresql。
* **<Figurative Name>.driver.class**：JDBC 驱动程序的类名 — 例如，org.postgresql.Driver。
* **<Figurative Name>.driver.class.path**：一连串由逗号隔开的通往 JDBC 驱动程序位置的类路径入口。该项为可选项；如果不包括此项，WrappingJDBCDriver 会使用自己的类加载程序来定位驱动程序类。
* **<Figurative Name>.tracer.pattern.<Zero Based Index>**：一连串的正则表达模式，用于为特定目标数据库提取跟踪类别。索引必须以 0 开始，由序列来定义跟踪类别的层次结构。

WrappingJDBCDriver 的基本前提是配置 JDBC 客户机应用程序，让它使用 “被转换的（munged）” JDBC URL，其他任何的 JDBC 驱动程序（包括以插装为目标的）都无法识别这个 JDBC URL，因此除了 WrappingJDBCDriver 以外，不接受其他的 JDBC 驱动程序。WrappingJDBCDriver 将会识别被转换的 URL、内部载入目标驱动程序并将其与被转换的 URL 关联。此时，被转换的 URL 被 “解除转换”，并会被委托给内部驱动程序以获取与目标数据库的真正的连接。然后这个真正的连接被包装在 WrappingJDBCConnection 中，返回给请求应用程序。munge 算法是很基本的算法，只要它能够使目标 “真正的” JDBC 驱动程序完全无法识别 JDBC URL。否则的话，真正的驱动程序可能会绕过 WrappingJDBCDriver。在这个例子中，我将 jdbc:postgresql://DBSERVER:5432/runtime 真正的 JDBC URL 转换为jdbc:!itracer!wrapped:postgresql://DBSERVER:5432/runtime。

“真正的” 驱动程序的类名和可选类路径配置项的作用是允许 WrappingJDBCDriver 查找和载入驱动程序类，这样它就能够被包装和委托了。跟踪程序模式配置项是一组正则表达式，它们指导 WrappingJDBCDriver 如何为目标数据库确定跟踪名称空间。这些表达式被应用于 “真正的” JDBC URL，并被利用，这样跟踪程序就能够给按目标数据库划分的 APM 系统提供性能指标。由于 WrappingJDBCDriver 用于多个（可能是不同的）数据库，因此按目标系统库进行划分是很重要的，这样收集的指标就可以按目标数据库进行分组了。例如，一个jdbc:postgresql://DBSERVER:5432/runtime 的 JDBC URL 可能会生成一个 postgresql, runtime 的名称空间。

清单 13 展示了一个样例 wrapped-driver.xml 文件，它使用了映射到 PostgreSQL 8.3 JDBC Driver 的 postgres 的形象化的名称：

清单 13. 样例 wrapped-driver.xml 文件

<properties>

<entry key="postgres.driver.prefix">jdbc:postgresql:</entry>

<entry key="postgres.driver.class">org.postgresql.Driver</entry>

<entry key="postgres.driver.class.path">

C:\Postgres\psqlJDBC\postgresql-8.3-603.jdbc3.jar

</entry>

<entry key="postgres.tracer.pattern.0">:([a-zA-Z0-9]+):</entry>

<entry key="postgres.tracer.pattern.1">.\*\/\/.\*\/([\S]+)</entry>

</properties>

该部分实现受到了一个名为 P6Spy 的开源产品的启发（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）。

为了展示 WrappingJDBCDriver 的使用方法，我创建了一个新的 EmpDAO Spring 测试用例的加强版。新的 Spring 配置文件是 spring-jdbc-tracing.xml，新的入口点类是 SpringRunnerJDBC。该测试用例包含几个其他的对比测试点，所以为了更明确一些，一些命名约定被更新了。我还加强了测试用例使其成为多线程的，这样就会在收集的指标中创建各种有趣的行为。而且，为了使之富于变化，DAO 的参数可以随机化。

我为这个新测试用例添加了如下的跟踪加强：

* 定义了两个数据源。一个使用直接的 JDBC 驱动程序，另一个使用插装的 JDBC 驱动程序。
* 这两个数据源可以通过 Spring 代理随意访问，插装这个代理的目的是监控获取连接的运行时间。
* DAO 截取程序被加强了，它能够监控经过截取程序的并发线程的数量。
* 另外还衍生了另一个后台线程，用以轮询数据源中统计信息的使用情况。
* 所有的 WrappingJDBC 类都通过基类 WrappingJDBCCore 调用了它们的大部分 跟踪程序调用。该基类除了向它的 ITracer 发出一个直接传递（passthrough）外，还在数据库实例级发出了 rollup 级别的跟踪。这展示了 APM 系统中的一个常见的特性，凭此特性可以多次将低级的和特定指标跟踪到更高级的名称空间，提供汇总的指标。例如，任何对象类型中的全部 JDBC 调用都上升到数据库级别，从而为所有的数据库调用汇总平均运行时间和请求量。

清单 14 显示了 spring-jdbc-tracing.xml 文件中的新 bean 定义的实例。注意在 InstrumentedJDBC.DataSource bean 中定义的 JDBC URL 使用了 munged 约定。

清单 14. spring-jdbc-tracing.xml 片段

<!-- A DataSource Interceptor -->

<bean id="InstrumentedJDBCDataSourceInterceptor"

class="org.runtimemonitoring.spring.interceptors.SpringDataSourceInterceptor">

<property name="interceptorName" value="InstrumentedJDBC.DataSource"/>

</bean>

<!-- A DataSource for Instrumented JDBC -->

<bean id="InstrumentedJDBC.DataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:url="jdbc:!itracer!wrapped:postgresql://DBSERVER:5432/runtime"

p:driverClassName="org.runtimemonitoring.jdbc.WrappingJDBCDriver"

p:username="scott"

p:password="tiger"

p:initial-size="2"

p:max-active="10"

p:pool-prepared-statements="true"

p:validation-query="SELECT CURRENT\_TIMESTAMP"

p:test-on-borrow="false"

p:test-while-idle="false"/>

<!-- The Spring proxy for the DataSource -->

<bean id="InstrumentedJDBC.DataSource.Proxy"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="InstrumentedJDBC.DataSource"/>

<property name="optimize"><value>true</value></property>

<property name="proxyTargetClass"><value>true</value></property>

<property name="interceptorNames">

<list>

<idref local="InstrumentedJDBCDataSourceInterceptor"/>

</list>

</property>

</bean>

<!--

The Spring proxy for the DataSource which is injected into

the DAO bean instead of the DataSource bean itself.

-->

<bean id="InstrumentedJDBC.DataSource.Proxy"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="InstrumentedJDBC.DataSource"/>

<property name="optimize"><value>true</value></property>

<property name="proxyTargetClass"><value>true</value></property>

<property name="interceptorNames">

<list>

<idref local="InstrumentedJDBCDataSourceInterceptor"/>

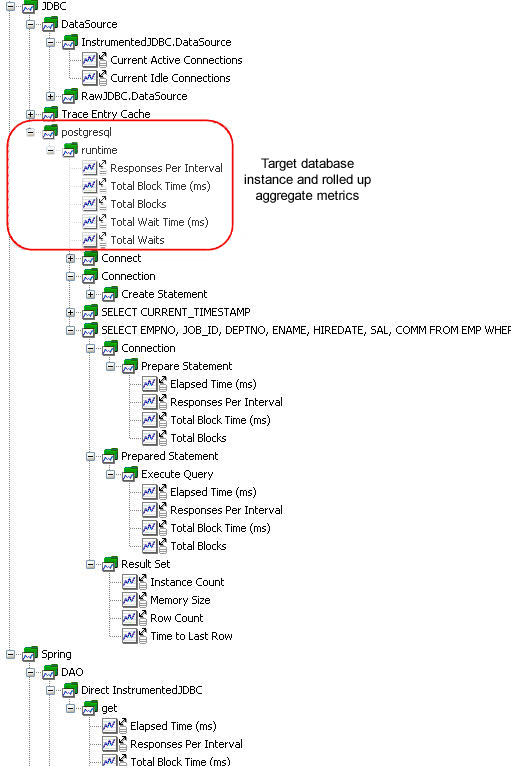
</list>

</property>

</bean>

图 10 显示了这个测试用例的 APM 指标树：

图 10. 插装的 JDBC 指标树



有了这个例子中的大量数据，就可以使用一些具体例子展示线程 BLOCK 和 WAIT 的起因。SpringRunnerJDBC 在每一个循环末尾的一个简单的语句 Thread.currentThread().join(100) 周围添加了一个 ThreadInfoCapture(WAIT+BLOCK) 跟踪。依照 APM 系统，这显示为一个平均为 103 ms 的线程等待。所以把一个线程置于等待某事发生的等待状态时，它会导致一段等待时间。相反，当线程试图从 DataSource 获取连接时，它在访问一个紧密同步的资源，而随着竞争连接的线程数的增加，DAO.get 方法会明确显示出增加了的线程阻塞数。

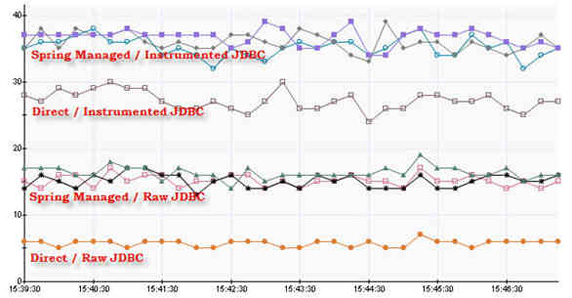
这个测试用例显示了由于添加了插装的和非插装的数据源而导致的另外几个 DAO.get bean 实例。表 2 展示了更新了的对比场景和数值结果的列表：

表 2. 插装的 JDBC 测试运行结果

| **测试用例** | **平均运行时间（ms）** | **最小运行时间（ms）** | **最大运行时间（ms）** | **计数** |
| --- | --- | --- | --- | --- |
| **直接访问，原始 JDBC** | 5 | 0 | 78 | 12187 |
| **直接访问，插装的　JDBC** | 27 | 0 | 281 | 8509 |
| **无截取程序 Spring bean，原始 JDBC** | 15 | 0 | 125 | 12187 |
| **无截取程序 Spring bean，插装的 JDBC** | 35 | 0 | 157 | 8511 |
| **插装的 Spring bean，原始 JDBC** | 16 | 0 | 125 | 12189 |
| **插装的 Spring bean，插装的 JDBC** | 36 | 0 | 250 | 8511 |
| **优化的插装 Spring bean，原始 JDBC** | 15 | 0 | 203 | 12188 |
| **优化的插装 Spring bean，插装的 JDBC** | 35 | 0 | 187 | 8511 |

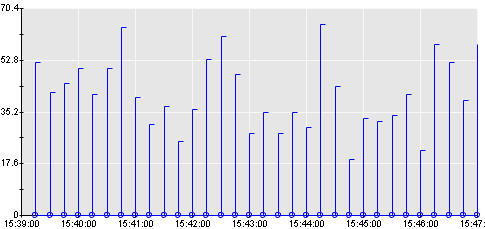
这些结果显示出了一些有趣的模式，但有一点很明了：插装的 JDBC 显然要比原始 JDBC 慢。这一点告诫我们一定要竭尽所能改进和调试插装。在这个基本的 JDBC 插装示例中，造成性能差异的原因是使用了插入式跟踪、较长的代码路径以及创建了大量额外的对象（用来执行一系列查询）。如果我想在高性能环境中使用这个方法，则需要对这个代码库执行更多的工作！使用插装的 DAO.get bean 会有另外一个明显但不是很严重的影响。这还是要归因于反射调用中的额外开销、较长的代码路径和跟踪活动。跟踪适配器看起来好像也能使用一些调优，但事实是所有的插装都会导致某种程度的开销。图 11 显示了此测试的运行时间结果：

图 11. 插装的 JDBC 结果



本节最后将介绍上升到数据库级的线程阻塞时间。这些数据库级的统计数字代表所有收集到的每个时间间隔内数据库调用指标的总计值。运行时间为平均值，但是计数（每个时间间隔内的响应、阻塞和等待）为每个时间间隔内的总数。在这个测试用例中，总计的每个时间间隔内的平均阻塞时间为零，但是在图 12 中，您可以观察到一些 APM 可视化工具的一个特性。虽然平均值是零，但是每一个时间间隔都有一个最大（和最小）读数。在这个图中，我的 APM 显示了一个空白的零行，它既表明了平均值也表明了最大值：

图 12. JDBC 总计阻塞时间



在本文的最后一节中，我将介绍最后一个不改变源代码插装 Java 类的技巧：字节码插装。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm2/#ibm-pcon)

字节码插装

到此为止，我向您展示的不基于源代码的插装都涉及到添加对象并经常延长代码执行路径，使它比跟踪代码本身的执行还要长。在字节码插装（BCI）技巧中，字节码被直接插入到一个 Java 类中，从而获得类最初不支持的功能。对于希望修改类而不触及源代码，或者希望在运行时动态修改类定义的开发人员，这个过程可以实现多种用途。我将向您展示如何使用 BCI 来将性能监控插装注入到类中。

不同的 BCI 框架可以以不同的方式达到这个目的。有一个简单的可以在方法级实现插装的技巧：重新命名目标方法，并使用包含跟踪指令并调用初始（重命名的）方法的原始签名插入一个新方法。一个名为 JRat 的开源 BCI 工具演示了一个技巧，该技巧专门为方法执行收集运行时间，因此要比通用的 BCI AOP 工具（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）简短。我将一个 JRat 项目的例子压缩成了清单 15 所示的内容：

清单 15. 使用 BCI 的插装方法示例

//////////////////////////////////////////////////////////////

// The Original Method

//////////////////////////////////////////////////////////////

public class MyClass {

public Object doSomething() {

// do something

}

}

//////////////////////////////////////////////////////////////

// The New and Old Method

//////////////////////////////////////////////////////////////

public class MyClass {

private static final MethodHandler handler = HandlerFactory.getHandler(...);

// The instrumented method

public Object doSomething() {

handler.onMethodStart(this);

long startTime = Clock.getTime();

try {

Object result = real\_renamed\_doSomething(); // call your method

handler.onMethodFinish(this, Clock.getTime() - startTime, null);

} catch(Throwable e) {

handler.onMethodFinish(this, Clock.getTime() - startTime, e);

throw e;

}

}

// The renamed original method

public Object real\_renamed\_doSomething() {

// do something

}

}

实现 BCI 的两个常用策略为：

* **静态**：Java 类或者类库被插装，插装的类被保存在原始类或类库的副本中。然后这些副本被部署到一个应用程序，这个应用程序对插装的类和其他的类一视同仁。
* **动态**：在类载入过程中，Java 类在运行时被插装。插装的类仅暂存于内存中；JVM 结束后，它们就会消失。

动态 BCI 的优势之一就在于提供了灵活性。动态 BCI 通常都是依照一组被配置的指令（通常位于一个文件中）执行。虽然它支持热交换，但修改插装只需要升级该文件和 JVM 周期就可以了。尽管动态 BCI 很简单，但我还是要先分析静态插装过程。

静态 BCI

在这个例子中，我将使用静态 BCI 来插装 EmpDAOImpl 类。我将使用 JBoss AOP，一个开源 BCI 框架（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）。

第一步：定义我要用来收集方法调用性能数据的截取程序，因为这个类将会被静态编入 EmpDAOImpl 类的字节码中。在这种情况下，JBoss 接口与我为 Spring 定义的截取程序是相同的，不同的是导入的类名。这个例子使用的截取程序是org.runtimemonitoring.aop.ITracerInterceptor。第二步：使用与定义 EJB 3 截取程序的 jboss-aop.xml 相同的语法定义 jboss-aop.xml 文件。清单 16 显示了该文件：

清单 16. 静态 BCI jboss-aop.xml 文件

<aop>

<interceptor class="org.runtimemonitoring.aop.ITracerInterceptor" scope="PER\_VM"/>

<bind

pointcut="execution(public \* $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">

<interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>

</bind>

</aop>

第三步：使用 JBoss 提供的名为 Aop Compiler（aopc）的工具来执行静态插装过程。用 Ant 脚本来完成这个过程是最简单的。清单 17 展示了 Ant 任务和编译器输出的代码片断，该片断表明我定义的切入点与目标类相匹配：

清单 17. aopc Ant 任务和输出

<target name="staticBCI" depends="compileSource">

<taskdef name="aopc" classname="org.jboss.aop.ant.AopC"

classpathref="aop.lib.classpath"/>

<path id="instrument.target.path">

<path location="${classes.dir}"/>

</path>

<aopc compilerclasspathref="aop.class.path" verbose="true">

<classpath path="instrument.target.path"/>

<src path="${classes.dir}"/>

<aoppath path="${conf.dir}/jboss-aop/jboss-aop.xml"/>

</aopc>

</target>

Output:

[aopc] [trying to transform] org.runtimemonitoring.spring.EmpDAOImpl

[aopc] [debug] javassist.CtMethod@955a8255[public transient get

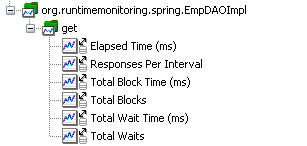
([Ljava/lang/Integer;)Ljava/util/Map;] matches pointcut:

execution(public \* $instanceof{org.runtimemonitoring.spring.DAO}->get(..))

定义于 jboss-aop.xml 文件的切入点和 [清单 16](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing16) 中定义的切入点一样实现了一个专用于 AOP 的语法，实现该语法的目的是为了提供一个表达力强的通配符语言来笼统地或是明确地定义切入点目标。实质上一个方法的任一标识属性都可以从类和包名映射到注释并返回类型。在 [清单 17](https://www.ibm.com/developerworks/cn/java/j-rtm2/#listing17)中，我指定 org.runtimemonitoring.spring.DAO 的任何实例中的任何名为 get 的公共方法都应被作为目标。因此，由于org.runtimemonitoring.spring.EmpDAOImpl 是惟一符合这个标准的具体类，所以只有这个类被插装了。

到此为止，插装就结束了。要运行启用了这个插装的 SpringRunner 测试用例，就必须在启动 JVM 时用诸如 -Djboss.aop.path=[directory]/jboss-aop.xml 这样的 JVM 参数把 jboss-aop.xml 文件的位置定义在系统属性中。这样做的前提是您可以获得一些灵活性，因为 jboss-aop.xml 首先在构建时的静态插装中使用，然后再在运行时使用，这是由于您一开始可以插装任意一个类，但在运行时却仅能激活特定类。为 SpringRunner 测试用例生成的 APM 系统指标树现在包含了 EmpDAOImpl 的指标。图 13 展示了这个树：

图 13. 静态 BCI 指标树



虽然静态插装的确可以提供某种灵活性，但是若非静态处理这些类（这很费力），就无法为插装激活它们，这一点终究是有限制性的。而且，一旦类被静态插装，它们就只能为插装时定义的截取程序激活。在下面的例子中，我将用动态 BCI 重复这个测试用例。

动态 BCI

完成动态 BCI 的方法很多，但是使用 Java 1.5 javaagent 接口有着一个很明显的优势。在此我将在更高的层面简要描述这个接口；想要深入了解关于这个主题的知识，请参见 Andrew Wilcox 所著的文章 “构建自己的分析工具”（参见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm2/#resources)）。

javaagent 通过两个结构启用运行时动态 BCI。首先，当用 -javaagent:a JAR file（这里的命名的 JAR 文件包含一个 javaagent 实现）启动 JVM 时，JVM 调用了在一个特殊清单条目中定义的类的一个 public static void premain(String args, Instrumentation inst) 方法。正如名称 premain 所暗示的，这个方法是在主 Java 应用程序入口点前被调用的，该入口点允许调用的类优先访问它，从而开始修改载入的类。关于这点它是通过注册 ClassTransformer（第二个结构）实例来实现的。ClassTransformer 接口负责从类加载程序有效截取调用并动态重写载入类的字节码。ClassTransformer 的单个方法 — transform — 被传入要重定义的类和包含类的字节码的字节数组。然后transform 方法实现各种修改，并返回一个包含修改的（或插装的）类的字节码的新字节数组。这种模型允许快速有效地传输类，并且与前面的一些方法不同，它不需要本地组件参与工作。

实现 SpringRunner 测试用例中的动态 BCI 有两步：首先，必须重新编译 org.runtimemonitoring.spring.EmpDAOImpl 类，将上面的测试用例中的静态 BCI 移除。其次，JVM 启动选项需要保留 -Djboss.aop.path=[directory]/jboss-aop.xml 选项，并且要按如下的方式添加javaagent 选项：

-javaagent:[directory name]/jboss-aop-jdk50.jar

清单 18 展示了一个稍微修改过的 jboss-aop.xml 文件，它说明了动态 BCI 的优势：

清单 18. 缩减的动态 BCI jboss-aop.xml 文件

<interceptor class="org.runtimemonitoring.aop.ITracerInterceptor"

scope="PER\_VM"/>

<interceptor class="org.runtimemonitoring.aop.PreparedStatementInterceptor"

scope="PER\_VM"/>

<bind

pointcut="execution(public \* $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">

<interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>

</bind>

<bind

pointcut="execution(public \* $instanceof{java.sql.Connection}->prepareStatement(..))">

<interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>

</bind>

pointcut="execution(public \* $instanceof{java.sql.PreparedStatement}->executeQuery(..))">

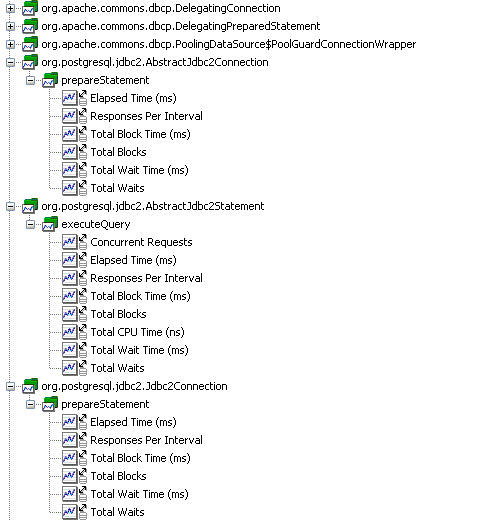
<interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>

</bind>

动态 BCI 的好处之一就是可以插装任何类，包括第三方库，所以清单 18 展示了 java.sql.Connection 所有实例的插装。然而它更强大的能力是可以把任何（但可用的）截取程序应用到定义的切入点。例如，org.runtimemonitoring.aop.PreparedStatementInterceptor 是一个普通的但却与 ITracerInterceptor 有些不同的截取程序。截取程序的整个库（在 AOP 用语中常指方面（aspects））都可以被开发，并可以通过开源提供商获得。这些方面库可以提供广泛的透视图，根据您想要应用的插装类型、要插装的 API 的不同，或者两者均不同，这些透视图的用途也不一样。

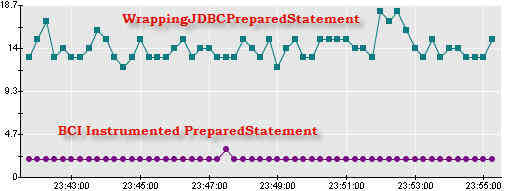
图 14 展示了其他指标的指标树。注意通过使用 Spring 中的 Jakarta Commons DataSource 提供者，有几个类实现了 java.sql 接口。

图 14. 动态 BCI 指标树



对比一下 WrappingJDBC 插装技术和使用 BCI 插装的驱动程序的性能差异，BCI 方法最大的优点就很明了了。这点在清单 15 中有所展示，清单 15 展示了 PreparedStatement.executeQuery 的对比运行时间：

图 15. BCI 对比包装性能



第 2 部分结束语

在这篇文章中我介绍了很多种插装 Java 应用程序的方式，目的是为了跟踪 APM 系统的性能监控数据。我所展现的这些技巧都不需要修改原始源代码。到底哪一个方法更合适要视情况而定，但是可以确定的是 BCI 已经成为了主流。APM 系统是内部开发的、开源的、商用的系统，可以用它来为 Java 性能管理实现 BCI，要想实现性能良好且高度可用的系统，APM 系统必不可少。

本系列的第三部分也是最后一部分将介绍监控 JVM 外部资源的方式，包括主机和它们的操作系统以及诸如数据库和通信系统这样的远程服务。它还总结了应用程序性能管理的其他问题，诸如数据管理、数据可视化、报告和警报。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm2/#ibm-pcon)

下载

| **描述** | **名字** | **大小** |
| --- | --- | --- |
| 本文的示例代码 | [j-rtm2.zip](http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=333057&filename=j-rtm2.zip&method=http&locale=zh_CN) | 316KB |

参考资料

学习

* 您可以参阅本文在 developerWorks 全球网站上的 [英文原文](http://www.ibm.com/developerworks/java/library/j-rtm2/)。
* 阅读整个 [Java 运行时监控](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) 系列。
* “[使用 JMX 接口来编写 PMI 应用程序](http://www.ibm.com/developerworks/cn/websphere/techjournal/0402_qiao/0402_qiao.html)”（Wenjian Qiao 和 Srini Rangaswamy，developerWorks，2004 年 2 月）：解释了 WebSphere Performance Monitoring Infrastructure、JMX MBeans 和 Java EE Performance Data Framework 之间的关系。
* “[Spring Framework 2.5 简介](http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25)”（Rod Johnson，TheServerSide，2007 年 10 月）：Spring 的创建者介绍了该框架。
* [P6Spy](http://www.p6spy.com/)：JDBC 分析器。
* [JRat](http://jrat.sourceforge.net/)：一个字节码插装工具。
* [JBoss AOP](http://www.jboss.org/jbossaop/)：一个针对 Java 应用程序的 AOP 框架。
* “[构建自己的分析工具](http://www.ibm.com/developerworks/cn/java/j-jip/)”（Andrew Wilcox，developerWorks，2006 年 3 月）：使用 Java 5 代理接口和 AOP 创建一个分析工具。
* 在 [技术书店](http://www.ibm.com/developerworks/apps/SendTo?bookstore=safari) 浏览关于这些主题和其他技术主题的图书。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：数百篇关于 Java 编程各个方面的文章。

获得产品和技术

* [CA/Wily Introscope](http://www.wilytech.com/solutions/products/Introscope.html)：一个商用 Java 和 Web 应用程序性能管理解决方案。
* [JINSPIRED JXInsight](http://www.wilytech.com/solutions/products/Introscope.html)：一个商用的 Java 性能监控、问题诊断、事务分析和应用程序管理解决方案。
* [PerformaSure](http://www.quest.com/performasure/)：一个商用的 Java 事务分析和性能诊断解决方案。
* 访问 IBM® Tivoli®[Monitoring for Transaction Performance](http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/index.jsp?toc=/com.ibm.itmtp.doc/toc.xml) 信息中心。

讨论

* 参与 [developerWorks blog](http://www.ibm.com/developerworks/blogs/) 并加入 [developerWorks 社区](http://www.ibm.com/developerworks/community)。

# Java 运行时监控，第 1 部分: Java 系统运行时性能和可用性监控

https://www.ibm.com/developerworks/cn/java/j-rtm1/

运行时性能监控对于实现和维护性能优异的系统至关重要。本文是系列文章的第 1 部分（共 3 部分），Nicholas Whitehead 将阐述如何有效地对 Java™ 性能执行低级粒度的监控。您生成的数据可以很好的反映系统操作的内在状态，并揭示影响环境稳定性及性能的约束和因素。

[查看本系列更多内容](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=Java+%E8%BF%90%E8%A1%8C%E6%97%B6%E7%9B%91%E6%8E%A7) | 1https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/java/j-rtm1/#icomments)

[Nicholas Whitehead](https://www.ibm.com/developerworks/cn/java/j-rtm1/#authorN1002E), 高级技术架构师, ADP

2008 年 8 月 19 日

* [expand](https://www.ibm.com/developerworks/cn/java/j-rtm1/#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-rtm1-_-sidebar)

简介

当今的许多 Java 应用程序都依赖于一组复杂的分布式依赖关系和移动部件。很多外部因素都可能对应用程序的性能和可用性造成影响。这些影响基本上都无法完全消除或解决，且难以在预生成环境中准确模拟。Stuff happens。但是，您可以创建并维护一个全面的系统来监控应用程序的整个生态系统，从而显著降低这些事件的严重性和持续时间。

本系列文章给出了实现此类系统的一些模式和技巧。模式，以及我将使用的一些术语，都表示泛指。通过结合示例代码和插图，它们将帮助您理解应用程序性能监控的概念。这种理解强调解决方案的必要性，并能帮助您选择商业或开源的解决方案。您可以扩展和定制一个解决方案，或者根据需要将其作为设计解决方案的蓝图。

第 1 部分：

* 探究应用程序性能管理（APM）系统的属性
* 描述系统监控的常见反面模式
* 列举监控 JVM 性能的方法
* 提供有效插装应用程序源代码的方法

第 2 部分将重点介绍插装 Java 类及资源而无需修改原始源代码的方法。第 3 部分将论述监控 JVM 外部资源的方法，包括主机及其操作系统以及数据库和消息传递系统等远程服务。它还将总结并归纳其他的 APM 问题，如数据管理、数据虚拟化、报告和报警。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

APM 系统：模式和反面模式

为让大家正确入门，应当强调，虽然此处介绍的多数与 Java 相关的内容看上去与应用程序和代码性能分析的流程类似，但其实并非 如此。性能分析是一个极具价值的生产前流程，它可以确认您的 Java 代码是否可扩展、高效、快速和足够出色。但是，根据 stuff happens 公理，当您在生产中遇到无法说明的问题时，优秀的开发阶段代码性能分析可能无用武之地。

我的意思是，在生产中实现性能分析的一些方面，并从运行中的应用程序收集一些相同的实时数据及其所有外部依赖关系。该数据由一系列遍及目标的定量测量指标组成，它们为整个系统的健康状况提供细粒度和详细的表示。此外，通过保留这些指标的历史库，您可以捕获准确的基线，以帮助您确认环境仍然健康，或查明特定缺陷的根源和规模。

监控反面模式

完全没有监控资源的应用程序微乎其微，但仍然需要考虑这些反面模式，它们经常出现在运行环境中：

孤立监控的缺陷

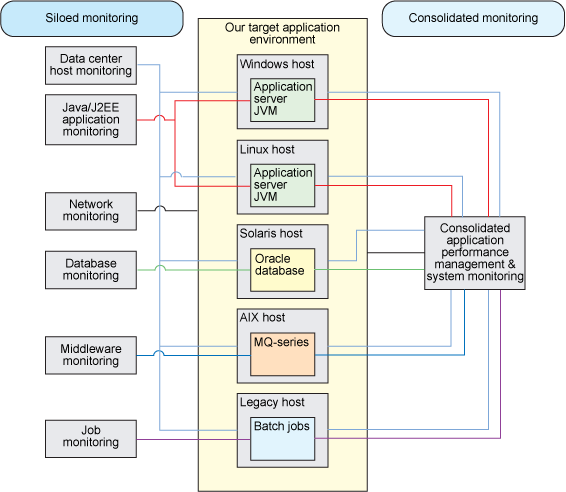
当系统视图无法反映整体情况时，便会出现孤立 监控。最为复杂和难以诊断的问题通常涉及多个参与和相关组件。请考虑一个简单的例子，托管在 Java 应用服务器上的应用程序（所有者未知）实现了一个有故障的 JDBC 连接池类（泄漏连接）。

当应用程序所有者查看管理接口时，他们声称自己的服务器与数据库保持了 100 个连接。相反，数据库管理员（DBA）查看数据库管理控制台却能看到相同的主机实际维持了 120 个连接，并且其数量正在迅速增加。在经过整合的 APM 系统中，创建一个显示这两种指标的曲线图应该说是微不足道的。当这两个数字彼此背离时，看到该图的任何人都可以立即清楚地看到真实数字，并能准确判断出问题所在。

* **盲点：**某些系统依赖关系未受监控，或者监控数据不可访问。运行中的数据库可以覆盖所有监控范围，但如果受支持的网络无法全面覆盖，则诊断小组在分析数据库性能和应用服务器症状时将无法看到网络中的故障。
* **黑盒：**核心应用程序或者它的某个依赖关系对于其内部可能不具有监控透明性。JVM 是一个不折不扣的黑盒。举例来说，诊断小组正在调查 JVM 中的莫名延时问题，并且只拥有支持操作系统的统计数据（如 CPU 利用率和进程需要的内存大小），则他们可能无法诊断垃圾收集或线程同步问题。
* **脱节和断开的监控系统：**应用程序可以由大型共享数据中心托管，其中，依赖关系由一系列共享资源组成，比如说数据库、存储区网络（SAN）库、消息传递及中间件服务。组织有时高度孤立，各小组只负责管理自己的监控和 APM 系统（请参阅 [孤立监控的缺陷](https://www.ibm.com/developerworks/cn/java/j-rtm1/#pitfalls) 侧栏）。没有各依赖关系的整合视图，各组件所有者只能管中窥豹，只见一斑。

图 1 对比了孤立和整合的 APM 系统：

图 1. 孤立和整合 APM 系统的对比



* **事后报告和相关性：**为尝试解决孤立监控的问题，运营支持小组可以运行定期进程获取各来源的数据，将这些数据整合到一个地方，然后再生成汇总报表。这种方法有时效率低下且不切实际，因为它需要按照指定频率严格执行，而缺乏实时数据也会对诊断小组当场发现问题的能力产生负面影响。此外，事后聚合有时缺乏足够的粒度，从而导致重要模式隐藏在数据中不被发觉。举例来说，某个报告可能显示某特定服务调用昨天平均耗时 200 毫秒，但却隐藏了它在下午 1:00 到 1:45 间平均耗时 3500 毫秒。
* **定期或随需应变的监控：**由于某些工具强制占用较高的资源开销，因此不能（或不应）经常使用它们。结果，它们很少收集数据，或者只在检测到问题后才收集数据。因此，APM 系统只能执行最低基线，而无法在问题恶化前提前报警，并且可能会自己加剧势态的严重性。
* **非持久化监控：**许多工具都提供了有用的性能和可用性指标实时显示功能，但它们并不支持持久化指标供长期或短期比较和分析的功能。常见的一种情况是，如果缺少历史上下文，则性能指标将毫无价值，因为没有判断指标优劣的基准。举例来说，当前的 CPU 利用率是 45%。如果不知道历史利用率的情况，则不好判断当前 CPU 利用率负荷的轻重程度。但是，如果知道历史的典型值为百分之 x，可接受的用户性能上限是百分之 y，则情况就大有改观了。
* **对生产前模型的依赖：**假设所有潜在问题都可在生产部署之前从环境中清除，则完全依赖生产前监控和系统模型的实践经常会导致运行时监控不够全面。这些假设无法解决不可预测事件和依赖性故障，因此，诊断小组在遇到此类事件时将没有工具和数据可用。

整合 APM 的实现并不排除监控和诊断工具，如 DBA 管理工具集、低级网络分析应用程序和数据中心管理解决方案。这些工具仍然是无价的资源，但如果它们依赖于整合视图的专有性，则难以克服孤立效果的影响。

理想 APM 系统的属性

与刚才讨论的反面模式相反，本系列文章介绍的理想 APM 系统拥有以下属性：

* **渗透力**：它监控所有应用程序组件和依赖关系。
* **粒度化**：它可以监控层次极低的函数。
* **整合性**：收集的所有指标将被发送到支持整合视图的同一逻辑 APM 中。
* **恒定**：一周 7 天，一天 24 小时不间断监控。
* **高效**：性能数据收集不会对监控目标造成不利影响。
* **实时**：可以实时显示、报告和警告监控的资源指标。
* **历史**：监控的资源指标将持久化存储在一个数据库中，因此可以查看、比较和报告历史数据。

在深入研究此系统的实现细节之前，了解 APM 系统的一些基本概念是有帮助的。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

APM 系统概念

所有 APM 系统都能访问性能数据源 并提供数据收集 和跟踪 实用工具。注意，这些是我自己选择的用于描述一般类别的通用术语。它们并非特定于任何 APM 系统，不同 APM 系统可以使用其他术语表示相同的概念。在本文的其余部分中，我所使用的术语定义如下。

性能数据源

性能数据源（PDS）是性能或可用性数据的来源，这些数据对于反映组件的相对健康状况非常有用。例如，Java Management Extensions (JMX) 服务通常可以提供关于 JVM 健康状况的丰富数据。大多数关系数据库通过 SQL 接口发布性能数据。这两种 PDS 都是直接 源的例子，即可以直接提供性能数据。相反，推断 源测定有意和偶然操作，并且产生性能数据。例如，测试消息可以定期发送，并随后从 Java Message Service (JMS) 服务器中取回，这个往返时间将作为该服务性能的推断测量。

推断源（它的实例被称作综合事务）有时极为有用，因为它们可以通过遍历与实际活动相同的路径来有效测定多个组件或分层调用。综合事务还在监控连续性方面发挥着重要作用，当直接源不能胜任时，它们可以确认系统在相对空闲期的健康状况。

收集和收集器

收集是从 PDS 获取性能或可用性数据的流程。对于直接 PDS，收集器 通常实现一些 API 来访问该数据。要从网络路由器读取统计数据，收集器可以使用简单网络管理协议（Simple Network Management Protocol，SNMP）或 Telnet。对于推断 PDS，收集器用于执行和测定底层操作。

跟踪和跟踪程序

跟踪是收集器向核心 APM 系统交付测量数据的流程。许多商业和开源 APM 系统都提供了一些用于此目的的 API。对于本文中的示例，我实现了一个通用的 Java 跟踪程序接口，将在下节详细讨论。

通常，大多数 APM 系统将跟踪程序提交的数据组织到某种分类的层次结构中。图 2 展示了该数据捕获的一般流程：

图 2. 收集、跟踪和 APM 系统

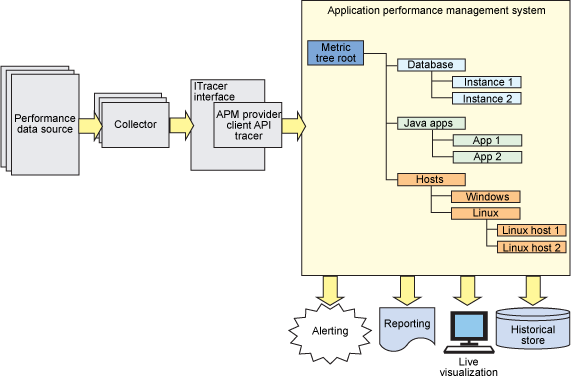


图 2 还展示了 APM 系统提供的常用服务：

* **实时显示**：近乎实时显示选定指标的图表。
* **报告**：生成的指标活动报告。这通常包括一系列固定报告和自定义报告，并能导出数据供用户在别处使用。
* **历史库**：包含原始或汇总指标的历史数据库，从而能够查看特定时间范围内的图表和报告。
* **报警：**将收集指标确定的具体条件通知相关个体或组的功能。典型的报警方法是电子邮件和某种类型的自定义钩子接口，可以允许运营小组将事件传播给事件处理系统。

公共跟踪 API 在 APM 的目标环境中的实现和应用提供了一些一致性。此外，自定义收集器的目的是让开发人员能够专心获取性能数据，而不必担心跟踪的问题。下一节将介绍解决此问题的 APM 跟踪接口。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

ITracer：跟踪程序接口

Java 语言可以很好地充当收集器的实现语言，因为：

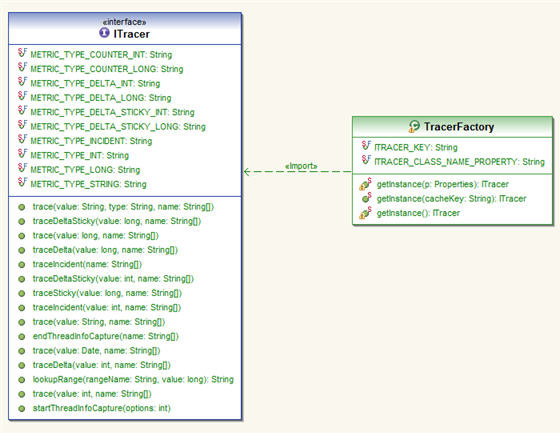
* 广泛的平台支持。Java 收集器类可以在大多数目标平台上运行，而无需修改。这使监控架构可以在本地灵活地使用 PDS 合并收集器进程，而不需要使用远程收集。
* 出色的性能（但是会随可用资源而变化）。
* 健壮的并发和同步执行支持。
* 支持一组丰富的通信协议。
* 受第三方 API 的广泛支持，比如 JDBC 实现、SNMP 和专用 Java 接口，因而能支持多种收集器库。
* 受活跃开源社区的支持，它提供了额外的工具和接口，使语言能访问或获取大量来源的数据。

但是，有一点需要注意，您的 Java 收集器必须能够与目标 APM 系统提供的跟踪 API 相结合。如果您的 APM 跟踪机制未提供 Java 接口，则它的一些模式将仍然适用。但是，如果目标 PDS 只基于 Java（如 JMX），而应用程序平台并不基于 Java，则需要一个桥接接口（如 IKVM）和一个 Java-to-.NET 编译器（请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm1/#resources)）。

当缺少官方标准时，不同 APM 产品提供的跟踪 API 也全然不同。因此，我通过实现一个通用的跟踪 Java 接口（名称为org.runtimemonitoring.tracing.ITracer）抽象了此问题。ITracer 接口是针对专用跟踪 API 的一个通用包装器。此技巧将确保源代码库不会因版本或 API 提供程序而有所不同，并且还支持实现包装 API 中不可用的额外功能。本文中的大多数其余示例都实现了 ITracer 接口和它所支持的一般底层概念。

图 3 是 org.runtimemonitoring.tracing.ITracer 接口的 UML 类图：

图 3. ITracer 接口和工厂类



跟踪类别和名称

ITracer 的基本前提是向中央 APM 系统提交一个度量和相关的名称。此活动由 trace 方法实现，该方法因提交的度量而有所不同。各 trace 方法都接受一个 String[] name 参数，其中包含复合名称的上下文组件，其结构特定于 APM 系统。复合名称向 APM 系统指示提交的名称空间和实际的指标名称；因此，复合名称中通常至少包括根类别和度量说明。底层 ITracer 实现应该知道如何通过传递的 String[] 构建复合名称。表 1 演示了复合命名约定的两个示例：

表 1. 示例复合名称

| **名称结构** | **复合名称** |
| --- | --- |
| 简单斜杠分隔 | Hosts/SalesDatabaseServer/CPU Utilization/CPU3 |
| JMX MBeanObjectName | com.myco.datacenter.apm:type=Hosts,service=SalesDatabaseServer,group=CPU Utilization,instance=CPU3 |

清单 1 是使用此 API 跟踪调用的简短示例：

清单 1. 跟踪 API 调用示例

ITracer simpleTracer = TracerFactory.getInstance(sprops);

ITracer jmxTracer = TracerFactory.getInstance(jprops);

.

.

simpleTracer.trace(37, "Hosts", "SalesDatabaseServer",

"CPU Utilization", "CPU3", "Current Utilization %");

jmxTracer.trace(37,

"com.myco.datacenter.apm",

"type=Hosts",

"service=SalesDatabaseServer",

"group=CPU Utilization",

"instance=CPU3", "Current Utilization %");

);

跟踪程序度量数据类型

在此接口中，度量数据可以是以下类型：

* int
* long
* java.util.Date
* String

APM 系统提供商可能支持其他数据类型的收集度量数据。

跟踪程序类型

选定了具体的度量数据类型（如 long）之后，可以根据 APM 系统支持的类型来选择解释特定值的方式。还需记住，各 APM 实现可以使用不同的术语来表示本质相同的类型，并且 ITracer 使用了一些通用的命名规则。

时间间隔

讨论跟踪程序类型需要了解时间间隔 的概念。请考虑这样一个概念：进程正在收集操作已经运行的时间并将它发送给 APM 系统。您每分钟都可以看到成百上千的调用。传输并存储每个度量的细节并不是切实有效的方法，而且也不应该在任何性能报告或图表中反应每一个度量，因为即便非常罕见的调用也可以歪曲整个信息表示。同时，由于长时期过度捕获，您将丧失粒度化的概念，因为这段时间内的合法峰值可能相当多。

解决此问题的一种模式是选取一个时间时隔来表示您希望聚合的最低粒度。1 个小时可能太长，而 200 毫秒又太短，因此您可以设定 30 秒的间隔。现在，您仍然可以对假设操作的每次调用都调用 trace，但是，保留在各时间间隔末尾的数据是：

* 时间间隔平均耗时
* 时间间隔最长耗时
* 时间间隔最短耗时
* 时间间隔内的调用次数
* 时间时隔开始时刻的时间戳

这是过度聚合度量和存储所有单独度量的一种有效的折衷方法。

ITracer 中表示的跟踪程序类型：

* **平均时间间隔**：trace(long value, String[] name) 和 trace(int value, String[] name) 方法将发出时间间隔平均值的跟踪（请参阅 [时间间隔](https://www.ibm.com/developerworks/cn/java/j-rtm1/#intervals) 侧栏）。这表示每个提交将被转化为当前时间间隔的聚合值。当新时间间隔开始时，聚合值计数器将重置为零。
* **粘附：**traceSticky(value long, String[] name) 和 traceSticky(value int, String[] name) 方法发出粘附值跟踪。这表示，与时间间隔平均指标相反，聚合将它们的值保留在时间间隔中。如果现在跟踪值 5，而此后不再执行跟踪直到第二天某个时刻，则该指标将保持为 5，直到提供了新值。
* **增量**：增量跟踪将传递一个数值，但提供给 APM 系统（或由 APM 系统解释）的实际值是此度量与前一度量之间的增量。它们有时被称作 rate 类型，用于反映自己的性能优势。请考虑事务管理程序的提交总数度量值。该数字始终在增加，并且其绝对值几乎没有用处。该数字有用的地方是它增加的速率，因此定期收集它的绝对值并跟踪每次读取数据之间的增量可以反映事务提交的速率。增量跟踪比平均时间间隔和粘附方式的跟踪更为常用，但仍有些用例采用了平均时间间隔。增量跟踪必须能够区分只能增加的度量和同时能增减的度量。小于前值的提交度量应被忽略或造成底层增量重置。
* **事件**：这种类型是一种简单的非聚合指标，它表示特定事件在时间间隔内发生的次数的增量计算。由于收集器和跟踪程序都不期望知道特定时刻的运行总数，因此基本的traceIncident(String[] name) 调用没有指定任何值，并且隐式只增加一次事件增量。当需要计算多次增量时，除了在循环中多次调用该方法之外，另一种较好的方法是通过traceIncident(int value, String[] name) 方法根据 value 来计算合值。
* **智能**：智能跟踪程序是一个参数化的类型，它与跟踪程序中的某种其他类型相映射。度量值和跟踪类型将作为 String 传递，并且可将可用类型作为常量定义在接口中。当收集器不知道正在收集的数据的类型或跟踪程序类型时，这是一个非常方便的方法，但是也可以直接将收集值和配置的类型名称传递给跟踪程序。

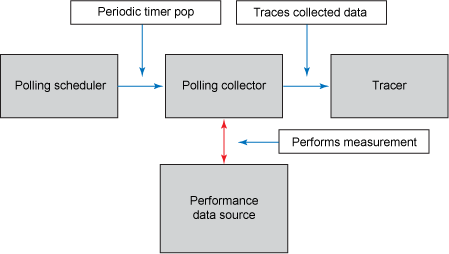
TracerFactory 是一个普通的工厂类，用于根据传递的配置属性创建新 ITracer 实例，或者从缓存中引用已创建的 ITracer。

收集器模式

收集通常有三种可选模式，这影响到应该使用的跟踪程序类型：

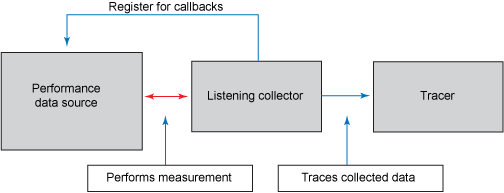
* **轮询：**按固定频繁调用收集器，它将检索和跟踪 PDS 中的指标或指标集的当前值。例如，可以每分钟调用一次收集器来读取主机的 CPU 利用率，或通过 JXM 接口从事务管理器读取提交事务的总数。轮询模式的前提是对目标指标的定期采样。因此，对于轮询事件，指标的值将提供给 APM 系统，但是，假定中间时期的值不变。因而，轮询收集器通常使用粘附跟踪程序类型：APM 系统在生成报告时将假定所有轮询事件之间的值不变。图 4 演示了此模式：

图 4. 轮询收集模式



* **监听**：这种通用数据模式是 Observer 模式的一种形式。收集器将其自身注册为目标 PDS 的事件监听程序，它将在相关的事件发生时接受回调。作为回调结果发出的跟踪值取决于回调有效负荷本身的内容，但收集器至少可以跟踪每个回调的事件。图 5 演示了此模式：

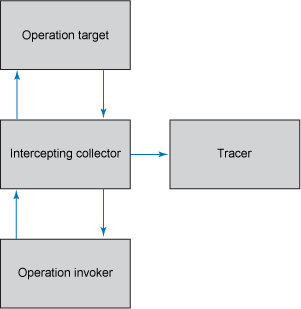
图 5：监听收集模式



* **截取**：在此模式中，收集器将自己作为截取程序插入到目标和它的调用程序之间。对于通过该截取程序的各个活动实例，截取程序将生成一个度量并跟踪它。当截取模式是 request/response 时，收集器可以测定请求数量、响应时间、请求或响应的有效负荷。例如，HTTP 代码服务器可以充当收集器，它可以：
  + 计算请求数，可以选择根据 HTTP 类型（GET 和 POST 等）或统一资源标识符（URI）来分类。
  + 请求的响应时间。
  + 测定请求和响应的大小。

由于您可以假定截取收集器能 “看到” 每一个事件，因此实现的跟踪程序通常为平均时间间隔类型。因此，如果时间间隔到期且没有活动发生，则该时间间隔的聚合值将为零，而与之前时间间隔中的活动无关。图 6 演示了此模式：

图 6. 截取收集模式



现在，我已经介绍了性能数据跟踪 API、它的底层数据类型和数据收集的模式。接下来，我将通过一些用例和示例来演示 API 的应用。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

监控 JVM

从 JVM 开始实现性能监控是个明智的选择。首先，我将介绍所有 JVM 共同的性能指标，然后再介绍企业给应用程序中经常使用的一些 JVM 驻留组件。通常，Java 应用程序实例是受底层操作系统支持的进程，因此，JVM 监控的某些方面最好是从主机 OS 的视角来理解，这些内容将在第 3 部分中介绍。

在 Java Platform, Standard Edition 5 (Java SE) 发行之前，能够在运行时有效和可靠收集的内部及标准化 JVM 诊断信息非常有限。现在，java.lang.management 接口提供了一些有用的监控点，该接口是所有兼容 Java SE 5（和更新版本）的 JVM 版本的标准。这些 JVM 的某些实现提供了额外的属性指标，但是它们的访问模式却基本相同。我将重点介绍可以通过 JVM 的 MXBeans 访问的标准模式 — 部署在 VM 内部的 JMX MBeans 公开了一个管理和监控接口（请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm1/#resources)）：

* ClassLoadingMXBean：监控类加载系统。
* CompilationMXBean：监控编译系统。
* GarbageCollectionMXBean：监控 JVM 的垃圾收集器。
* MemoryMXBean：监控 JVM 的堆和非堆内存空间。
* MemoryPoolMXBean：监控 JVM 分配的内存池。
* RuntimeMXBean：监控运行时系统。该 MXBean 提供的有用监控指标很少，但它确实提供了 JVM 的输入参数和启动时间及运行时间，这两者在其他派生指标中都是很有用的。
* ThreadMXBean：监控线程系统。

JMX 收集器的前提是它将获取一个 MBeanServerConnection 对象，该对象可以读取部署在 JVM 中的 MBeans 的属性，读取目标属性的值，并使用 ITracer API 跟踪它们。对于这种类型的收集，决定部署收集器的位置非常关键。可行的选择包括本地部署 和远程部署。

权限问题

本地和远程部署都可能会受到各种配置权限的限制，这些权限会阻止收集器访问 JVM 数据。这些问题大多都有相应的解决方法，但是它们的多变性使本文无法详细介绍这方面的内容。

在本地部署中，收集器和它的调用调度程序部署在目标 JVM 中。随后，JMX 收集器组件将使用 PlatformMBeanServer（可以通过 JVM 内部的 MBeanServerConnection 来连接它）访问 MXBeans。在远程部署中，收集器运行在一个单独的进程中，并使用某种形式的 JMX Remoting 来连接目标 JVM。这可能没有本地部署那么高效，但它不需要在目标系统中部署任何额外的组件。JMX Remoting 不在本文的讨论范围之内，但它的实现方法非常简单：部署一个 RMIConnectorServer 或在 JVM 中启用外部连接（请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm1/#resources)）。

示例 JMX 收集器

本文的示例 JMX 收集器（请阅读 [下载](https://www.ibm.com/developerworks/cn/java/j-rtm1/#download) 一节，获取本文的完整源代码）包含三个单独的方法，可用于获取 MBeanServerConnection。该收集器可以：

* 通过调用静态 java.lang.management.ManagementFactory.getPlatformMBeanServer() 方法，为本地 JVM 的平台 MBeanServer 获取一个MBeanServerConnection。
* 通过调用静态 javax.management.MBeanServerFactory.findMBeanServer(String agentId) 方法，为部署在本地 JVM 平台中的备用MBeanServer 获取一个 MBeanServerConnection。注意，一个 JVM 中可以存在多个 MBeanServer，并且，Java Platform, Enterprise Edition (Java EE) 服务器等较为复杂的系统几乎始终拥有特定于应用服务器的 MBeanServer，它是独立的且有别于平台 MBeanServer（请参阅 [交叉注册 MBeans](https://www.ibm.com/developerworks/cn/java/j-rtm1/##crossregister) 侧边栏）。
* 使用 javax.management.remote.JMXServiceURL 通过标准 RMI Remoting 获取一个远程 MBeanServerConnection。

清单 2 是摘录自 JMXCollector collect() 方法的代码段，它显示了 ThreadMXBean 中的收集和线程跟踪活动。点击 [此处](http://www.ibm.com/developerworks/java/library/j-rtm1/sidefile.html) 查看完整清单：

清单 2. 示例 JMX 收集器的 collect() 方法的部分代码，它使用 ThreadMXBean

.

.

objectNameCache.put(THREAD\_MXBEAN\_NAME, new ObjectName(THREAD\_MXBEAN\_NAME));

.

.

public void collect() {

CompositeData compositeData = null;

String type = null;

try {

log("Starting JMX Collection");

long start = System.currentTimeMillis();

ObjectName on = null;

.

.

// Thread Monitoring

on = objectNameCache.get(THREAD\_MXBEAN\_NAME);

tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on,"TotalStartedThreadCount"),

hostName, "JMX", on.getKeyProperty("type"), "StartedThreadRate");

tracer.traceSticky((Integer)jmxServer.getAttribute(on, "ThreadCount"), hostName,

"JMX", on.getKeyProperty("type"), "CurrentThreadCount");

.

.

// Done

long elapsed = System.currentTimeMillis()-start;

tracer.trace(elapsed, hostName, "JMX", "JMX Collector",

"Collection", "Last Elapsed Time");

tracer.trace(new Date(), hostName, "JMX", "JMX Collector",

"Collection", "Last Collection");

log("Completed JMX Collection in ", elapsed, " ms.");

} catch (Exception e) {

log("Failed:" + e);

tracer.traceIncident(hostName, "JMX", "JMX Collector",

"Collection", "Collection Errors");

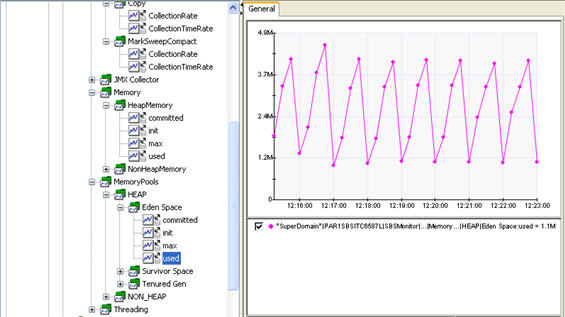
}

}

清单 2 中的代码将跟踪 TotalThreadsStarted 和 CurrentThreadCount 的值。由于它是轮询收集器，因此两个跟踪都使用粘附选项。但是，由于 TotalThreadsStarted 是一个不断增加的数值，因此最吸引人的地方不是绝对值，而是已创建线程的速率。这样，该跟踪程序将使用DeltaSticky 选项。

图 7 显示了此收集器创建的 APM 指标树：

图 7. JMX 收集器 APM 指标树



JMX 收集器的一些方面并未显示在清单 2 中（但是可以在 [完整源代码](https://www.ibm.com/developerworks/cn/java/j-rtm1/#download) 中看到），比如说调度注册，它将每隔 10 分钟为 collect() 方法创建一个定期回调。

在清单 2 中，不同跟踪程序类型和数据类型的实现方法将由数据源决定。例如：

* TotalLoadedClasses 和 UnloadedClassCount 将作为粘附增量被跟踪，因为它们的值始终递增，而且增量在测定类加载活动方面比绝对值更加有用。
* ThreadCount 变量可增加或减少，因此它将作为粘附类型被跟踪。
* 收集错误 将作为内部事件被跟踪，它将在收集遇到异常时递增。

为了追求效率，由于目标 MXBeans 的 JMX ObjectName 在目标 JVM 的生存期不会更改，因此收集器使用 ManagementFactory 常量名来缓存名称。

对于 MXBeans 的两种类型 — GarbageCollector 和 MemoryPool— 准确的 ObjectName 无法预先知晓，但是您可以提供一个通用的模式。在这些情况下，在初次执行收集时，您将对 MBeanServerConnection 发起一个查询，并请求与提供模式相匹配的所有 MBeans 的列表。为避免未来在目标 JVM 的生存期执行查询，返回的匹配 MBean ObjectName 将缓存在内存中。

在某些情况下，收集的目标 MBean 属性可能不是纯数值类型。MemoryMXBean 和 MemoryPoolMXBean 就是这种情况。对于这些情况，属性类型是可查询键和值的 CompositeData 对象。对于 java.lang.management JVM 管理接口，MXBean 标准采用了 JMX Open Types 模型，在该模型中，所有属性都是语言无关的类型，如 java.lang.Boolean 和 java.lang.Integer。或者，对于javax.management.openmbean.CompositeType 等复杂类型，这些类型可以被分解为相同简单类型的键/值对。简单类型的完整列表枚举在静态 javax.management.openmbean.OpenType.ALLOWED\_CLASSNAMES 字段中。该模型支持一个类型独立层，使 JMX 客户机不用依赖于非标准的类，并且还可以支持非 Java 客户机，因为底层类型相对比较简单。有关 JMX Open Types 的更多信息，请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-rtm1/#resources)。

对于目标 MBean 属性是非标准复杂类型的情况，您需要确保定义该类型的类在收集器的类路径中。并且，您必须实现一些自定义代码来呈现检索到的复杂对象中的有用数据。

如果获取了单个连接并为所有收集保留了该连接，则需要通过错误检测和修复来创建一个新连接，以防止该连接出现故障。某些收集 API 提供断开监控程序，可以提示收集器关闭、消除和创建新连接。如果收集器尝试连接到由于维护而停机或由于其他原因而无法访问的 PDS，则收集器应该以合适的频率轮询并重新连接。跟踪连接的运行时间还可用于在检测到关机时减少收集的频率。这可以减少已超负荷运行了一段时间的目标 JVM 的开销。

这些示例中未实现的两个额外技巧可以改进 JMX 收集器的效率，并减少它在目标 JVM 中运行所需的开销。第一个技巧适用于从一个 MBean 中查询多个属性的情况。借助 getAttributes(ObjectName name, String[] attributes)，您可以在一个调用中请求多个属性，而不必使用getAttribute(ObjectName name, String attribute) 一次请求一个属性。这种差异在本地收集中可以忽略，但是在远程收集中却可以显著减少资源的使用，因为它可以减少网络调用的数量。第二个技巧是使用监控收集模式代替轮询模式，从而进一步减少 JMX 公开内存池的轮询开销。MemoryPoolMXBean 支持建立一个使用率阀值，超过该阀值时将触发向监控程序发送一个通知，而监控程序将跟踪该值。当内存使用率增加时，使用率阀值可以相应地增加。这种方法是缺陷是，如果使用率阀值没有微小的增量，则一些粒度级的数据可能会丢失，并且阀值下方的内存使用率模式将变为不可见。

最后一个未实现的技巧是测定运行时间和垃圾收集总运行时间的范围，并实现一些简单的算法来计算垃圾收集器处于活动状态的时间在已运行时间中的百分比。这是一个有用的指标，因为一些垃圾收集器（当前）是大多数应用程序必须要面对的问题。由于某些收集（分别执行了一段时间）是期望执行的，因此运行垃圾收集占用的时间可以更加清楚地反映 JVM 的内存健康状况。根据经验（因应用程序而大不相同），占用任何 15 分钟时间段内的 10% 以上则表示存在潜在问题。

收集器的外部配置

为便于演示收集流程，本文介绍的 JMX 收集器经过了适当简化，但它仅限于硬编码的收集方式。理想情况下，收集器将实现数据访问方式，而外部提供的配置将提供内容。这种设计使收集器更具实用性，且易于重用。对于最高级别的重用，外部配置的收集器应该支持这些配置点：

* PDS 连接工厂指令，为收集器提供用于连接到 PDS 的接口，以及在连接时使用的配置。
* 执行收集的频率。
* 尝试重新连接的频率。
* 收集的目标 MBean，或通配符形式的对象名称。
* 对于各目标，跟踪复合名称或者应该跟踪的度量片段，以及应该跟踪的数据类型。

清单 3 演示了 JMX 收集器的外部配置：

清单 3. JMX 收集器的外部配置示例

<?xml version="1.0" encoding="UTF-8"?>

<JMXCollector>

<attribute name="ConnectionFactoryClassName">

collectors.jmx.RemoteRMIMBeanServerConnectionFactory

</attribute>

<attribute name="ConnectionFactoryProperties">

jmx.rmi.url=service:jmx:rmi://127.0.0.1/jndi/rmi://127.0.0.1:1090/jmxconnector

</attribute>

<attribute name="NamePrefix">AppServer3.myco.org,JMX</attribute>

<attribute name="PollFrequency">10000</attribute>

<attribute name="TargetAttributes">

<TargetAttributes>

<TargetAttribute objectName="java.lang:type=Threading"

attributeName="ThreadCount" Category="Threading"

metricName="ThreadCount" type="SINT"/>

<TargetAttribute objectName="java.lang:type=Compilation"

attributeName="TotalCompilationTime" Category="Compilation"

metricName="TotalCompilationTime" type="SDINT"/>

</TargetAttributes>

</attribute>

</JMXCollector>

注意，TargetAttribute 元素包含一个名为 type 的属性，它表示智能类型跟踪程序的参数化变量。SINT 类型表示粘附 int，SDINT 类型表示增量粘附 int。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

通过 JMX 监控应用程序资源

目前为止，我已经讨论了通过 JMX 监控惟一标准的 JVM 资源。但是，许多应用程序架构，如 Java EE，可以通过 JMX 公开重要的特定于应用程序的指标（这取决于供应商）。一个典型的例子是 DataSource 利用率。DataSource 是一个用于将连接池化到外部资源（通常为数据库）的服务，这限制了并发连接的数量，以保护资源不受恶意应用程序的占用。监控数据源是整个监控计划中的关键环节。由于 JMX 抽象层，该流程与之前介绍的类似。

下面是来自 JBoss 4.2 应用服务器实例的典型数据源指标：

* **可用连接数**：当前池中可用连接的数量。
* **连接数**：连接池与数据库建立的实际物理连接的数量。
* **最大使用连接数**：池中正在使用的连接的上限标记。
* **正在使用的连接数**：当前正在使用的连接数量。
* **已创建的连接数**：为该池创建的连接总数。
* **已部署的连接数**：为该池部署的连接总数。

现在，收集器将使用批属性检索，并在一个调用中获取所有属性。惟一需要注意的是，您需要查询返回的数据，以接通不同的数据和跟踪程序类型。DataSource 指标在没有活动时也是不会变化的，因此，要使数值变化，您需要生成一些负载。清单 4 显示 DataSource 收集器的collect() 方法：

清单 4. DataSource 收集器

public void collect() {

try {

log("Starting DataSource Collection");

long start = System.currentTimeMillis();

ObjectName on = objectNameCache.get("DS\_OBJ\_NAME");

AttributeList attributes = jmxServer.getAttributes(on, new String[]{

"AvailableConnectionCount",

"MaxConnectionsInUseCount",

"InUseConnectionCount",

"ConnectionCount",

"ConnectionCreatedCount",

"ConnectionDestroyedCount"

});

for(Attribute attribute: (List<Attribute>)attributes) {

if(attribute.getName().equals("ConnectionCreatedCount")

|| attribute.getName().equals("ConnectionDestroyedCount")) {

tracer.traceDeltaSticky((Integer)attribute.getValue(), hostName,

"DataSource", on.getKeyProperty("name"), attribute.getName());

} else {

if(attribute.getValue() instanceof Long) {

tracer.traceSticky((Long)attribute.getValue(), hostName, "DataSource",

on.getKeyProperty("name"), attribute.getName());

} else {

tracer.traceSticky((Integer)attribute.getValue(), hostName,

"DataSource",on.getKeyProperty("name"), attribute.getName());

}

}

}

// Done

long elapsed = System.currentTimeMillis()-start;

tracer.trace(elapsed, hostName, "DataSource", "DataSource Collector",

"Collection", "Last Elapsed Time");

tracer.trace(new Date(), hostName, "DataSource", "DataSource Collector",

"Collection", "Last Collection");

log("Completed DataSource Collection in ", elapsed, " ms.");

} catch (Exception e) {

log("Failed:" + e);

tracer.traceIncident(hostName, "DataSource", "DataSource Collector",

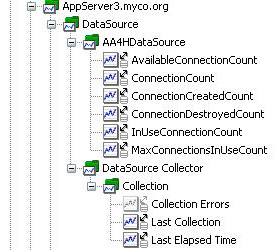
"Collection", "Collection Errors");

}

}

图 8 显示了 DataSource 收集器的相应指标树：

图 8. DataSource 收集器指标树



[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

监控 JVM 中的组件

交叉注册 MBeans

在许多情况下，可以将目标 MBeans 注册到相同 JVM 的不同 MBeanServer 中。例如，java.lang MXBean 注册在平台代理（也称作 JVM MBeanServer）中，而 JBoss 服务器中的 DataSource MBeans 位于 jboss MBeanServer 中。在远程监控实现中，这种情况有时会增加额外的开销和配置复杂度，因为每个 MBeanServer都需要两个远程连接。此外，为远程连接公开平台MBeanServer 也会带来额外的开销。在这些情况下，通常采用交叉注册 MBeans 的方式，这样所有的目标 MBeans 都可以通过相同的 MBeanServer 接口来监控。请查阅本文的 [源代码](https://www.ibm.com/developerworks/cn/java/j-rtm1/#download)，获取名称为 map-platform-mxbeans.bsh 的示例 bean-shell 脚本。如果您将此脚本部署到 JBoss 服务器中，它将在 JBoss MBeanServer中交叉注册平台 MBeanServer 的 MXBeans。

本节介绍的技巧可用于监控应用程序组件、服务、类和方法。相关的主要区域如下：

* **调用速率**：调用服务或方法的速率。
* **调用响应速率**：服务或方法响应的速率。
* **调用错误率**：服务或方法生成错误的比率。
* **调用运行时间**：调用在每个间隔时间内的平均、最短和最长运行时间。
* **调用并发性**：并发调用服务或方法时执行的线程数。

使用 Java SE 5（和更新版本）ThreadMXBean 的一些实现提供的指标，还可以收集以下指标：

* **系统和用户 CPU 时间**：调用某方法占用的 CPU 时间。
* **等待数量和总等待时间**：调用某方法或服务时，等待线程的实例数量和总占用时间。当线程进入 WAITING 或 TIMED\_WAITING 等待状态并暂停另一个线程的活动时将发生等待事件。
* **阻塞数量和总阻塞时间**：在调用某个方法或服务时，处于 BLOCKED 状态的线程的实例数量和总占用时间。当线程等待监控锁进入或重新进入同步阻塞时会发生阻塞事件。

还可以使用备选工具集和本机接口来确定这些指标和其他指标，但这通常涉及某种级别的开销，从而造成不必要的生产运行时监控。已经说过，指标本身，甚至在收集时，是低级的。它们的作用也许仅限于分析趋势，并且很难与无法通过其他手段确定的因果效应相关联。

所有上述指标都可以通过插装类和方法的流程来收集，以便于收集和跟踪目标 APM 系统的性能数据。可以采用各种技巧来直接插装 Java 类，或者通过它们来间接计算性能指标：

* **源代码插装：**最基本的技巧是在源代码阶段添加插装，这样编译和部署后的类就已经在运行时包含了插装。在某些情况下，这种方法具有意义，并且一些特定的实践使它成为可行的流程和投资。
* **截取**：通过截取程序（执行测定和跟踪）转移调用，可以实现有效和准确的跟踪，而无需接触目标类、它们的源代码和运行时字节码。这种实践简单可取，因为存在许多 Java EE 框架和其他流行的 Java 框架：
  + 通过配置支持抽象。
  + 支持类注入和通过接口引用。
  + 有某些情况下直接支持截取栈概念。执行流程经过定义了配置的对象栈，其作用是接收调用并执行一些处理，然后继续传递。
* **字节码插装**：该流程将字节码注入到应用程序类中。注入的字节码将添加性能数据收集插装，该插装被作为新类的一部分调用。这个流程有时极为有效，因为插装是完全经过编译的字节码，并且代码的执行路径以最细化的方式扩展，同时仍然能够收集数据。它的另一个优点是无需修改初始源代码，并且其对环境的配置更改也可能最少。此外，通用模式和字节码注入技巧允许对源代码不可用的类和库进行插装，许多第三方类属于这种情况。
* **类包装**：该流程使用另一个类来包装或替换目标类，前者实现了相同功能，同时也包含了插装。

在本文的第 1 部分中，我只讨论基于源代码的插装；您将在第 2 部分中了解更多关于截取、字节码插装和类包装的信息。（从拓扑学的角度来说，截取、字节码插装和类包装的本质完全相同，但它们实现结果的操作有稍微不同的含义）。

异步插装

异步插装是类插装中的基本问题。上一节讨论了轮询性能数据的概念。如果轮询完成得足够好，则它应该不会对核心应用程序性能或开销造成影响。相反，插装应用程序代码本身会直接修改和影响核心代码的执行。任何插装的目标都必须是无论如何，不产生危害。开销损失必须尽可能接近忽略不计。事实上，消除测量本身中的极细微的损失是不可能的，但是，在获取性能数据之后，保持其余跟踪进程异步是非常重要的。可以采用若干种模式来实现异步跟踪。图 9 演示了异步跟踪的实现方法概览：

图 9. 异步跟踪

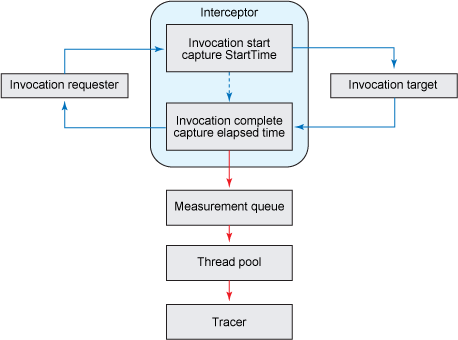


图 9 演示了一个简单的插装截取程序，它通过捕获调用的起始时间来测量它的运行时间，然后将测量数据（运行时间和指标复合名称）分发给处理队列。然后，线程池读取该队列，获取测量数据并完成跟踪流程。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

源代码中的 Java 类插装

本节将讨论如何实现源代码级插装，并将提供一些最佳实践和示例代码。文章还介绍了一些新的跟踪结构，我将在源代码插装的上下文中阐明它们的操作和它们的插装模式。

虽然其他选择已经流行，但源代码插装在某些实例中是无法避免的；在某些情况下，它是惟一的解决方案。借助一些明智的预防措施，它可以实现良好的效果。需要考虑的事项包括：

* 如果插装代码的方案可用，并且无法实现配置更改来垂直影响插装，则使用可配置和灵活的跟踪 API 非常重要。
* 抽象的跟踪 API 类似于事件记录 API（如 log4j），它们的共用属性包括：
  + **运行时冗长控制**：log4j 记录程序和附加程序的冗长等级可以在系统启动时配置，并随后在运行时修改。同样，跟踪 API 应该能够根据分级名称模式来控制哪些指标名称受跟踪支持。
  + **输出端点配置**：log4j 通过记录程序发起记录声明，并将它分发给附加程序。经过配置，附加程序可以将记录流发送给各种输出，如文件、套接字和电子邮件。跟踪 API 不需要多样的输出方式，但抽象专有或特定于 APM 系统的库的能力将保护源代码不受外部配置的更改。
* 在某些情况下，通过其他方法来跟踪具体的项目不太可行。通常，我将这种情况称作上下文跟踪。我使用该术语描述的性能数据并不是很重要，但它为主要数据添加上下文。

上下文跟踪

上下文跟踪受具体的应用程序影响极大，但是可以考虑一个经过简化的例子：含有 processPayroll(long clientId) 方法的 payroll-processing 类。当被调用时，该方法计算并存储各客户员工的薪水。您可以通过各种方法插装该方法，但是，执行中的底层模式清楚表明，调用时间的增加与员工的数量不成比例。因此，研究 processPayroll 的运行时间趋势没有上下文可供参考，除非您知道程序每次处理的员工数量。简单来讲，对于特定的时间段，processPayroll 平均耗时 x 毫秒。无法确定这个值反映的性能是好还是坏，因为您不知道它处理的员工数量是 1 还是 150，而两种情况反映的性能差别巨大。清单 5 在代码中显示了这个简化的概念：

清单 5. 上下文跟踪的例子

public void processPayroll(long clientId) {

Collection<Employee> employees = null;

// Acquire the collection of employees

//...

//...

// Process each employee

for(Employee emp: employees) {

processEmployee(emp.getEmployeeId(), clientId);

}

}

此处的主要挑战是，根据大多数插装技巧，processPayroll() 方法中的任何东西都是不可触及的。因此，虽然能够插装 processPayroll 甚至 processEmployee，但是却无法跟踪员工的数量，从而不能为方法的性能数据提供上下文。清单 6 显示了一个拙劣的硬编码示例（且有点效率不高），它将捕获上面提到的上下文数据：

清单 6. 上下文跟踪示例

public void processPayrollContextual(long clientId) {

Collection<Employee> employees = null;

// Acquire the collection of employees

employees = popEmployees();

// Process each employee

int empCount = 0;

String rangeName = null;

long start = System.currentTimeMillis();

for(Employee emp: employees) {

processEmployee(emp.getEmployeeId(), clientId);

empCount++;

}

rangeName = tracer.lookupRange("Payroll Processing", empCount);

long elapsed = System.currentTimeMillis()-start;

tracer.trace(elapsed, "Payroll Processing", rangeName, "Elapsed Time (ms)");

tracer.traceIncident("Payroll Processing", rangeName, "Payrolls Processed");

log("Processed Client with " + empCount + " employees.");

}

清单 6 中的关键部分是 tracer.lookupRange 调用。Ranges 是指定的收集，它由数值范围限制键控，并且拥有一个表示数值范围名称的String 值。不再跟踪薪水处理的简单无格式运行时间，清单 6 将员工计数划分为范围，有效分隔运行时间并根据基本类似的员工计数将它们分组。图 10 显示了 APM 系统生成的指标树：

图 10：根据范围分组的薪水处理时间

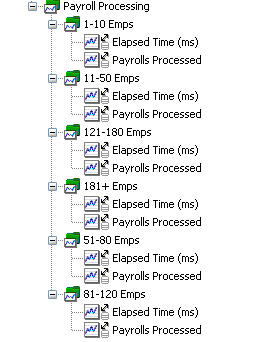
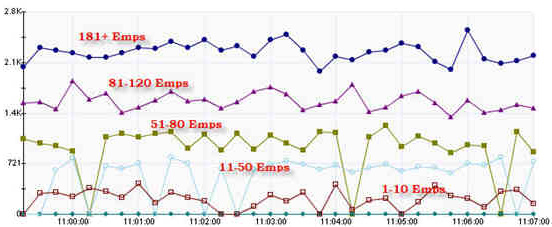


图 11 演示了根据员工计数划分的薪水处理运行的时间，它揭示了员工数量和运行时间之间的相互关系：

图 11. 各范围的薪水处理运行时间



跟踪程序配置属性允许在属性文件中包括 URL，并能在其中定义范围和阀值（我将简单介绍一下阀值）。属性将在跟踪程序的构造时间被读取，并为 tracer.lookupRange 实现提供后台数据。清单 7 显示了 Payroll Processing 范围的示例配置。我选择使用java.util.Properties 的 XML 表示，因为它更能兼容奇怪的字符。

清单 7. 范围配置示例

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

<comment>Payroll Process Range</comment>

<entry key="L:Payroll Processing">181+ Emps,10:1-10 Emps,50:11-50 Emps,

80:51-80 Emps,120:81-120 Emps,180:121-180 Emps</entry>

</properties>

注入外部定义的范围可以使您的应用程序不必频繁更新源代码，这受益于预期的调整和服务水平协议（SLA）在业务方面的变更。当范围和阀值更改生效之后，您只需更新外部文件，而不是应用程序本身。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

跟踪阀值和 SLA

外部可配置上下文跟踪的灵活性支持以更加准确和粒度化的方式来定义和测量性能阀值。范围 定义一系列数值区间，可以在其中对测量数据进行分类，而阀值 是对范围的进一步分类，它根据测量数据的确定范围对获取的测量数据进行分类。在分析收集的性能数据时，一个常见的需求是确定和报告执行是 “成功” 还是 “失败”（因为它们未在指定时间发生）。这些数据的总和可以作为关于系统运行健康状况和性能的通用成绩单，或者作为某种形式的 SLA 遵从性评价。

使用薪水处理系统示例，考虑一个内部服务级目标，它将薪水的执行时间（在定义的员工数范围之内）定义为 Ok、Warn 和 Critical 3 个区间。生成阀值计数的流程从概念上来说非常简单。您只需为跟踪程序提供您认为是各类别各区间的上限运行时间的值，并引导跟踪程序为分类的运行时间发起一个 tracer.traceIncident，然后 — 为简化报告 — 提供一个总数。表 2 显示了一些经过设计的 SLA 运行时间：

表 2. 薪水处理阀值

| **员工数** | **Ok (ms)** | **Warn (ms)** | **Critical (ms)** |
| --- | --- | --- | --- |
| **1-10** | 280 | 400 | >400 |
| **11-50** | 850 | 1200 | >1200 |
| **51-80** | 900 | 1100 | >1100 |
| **81-120** | 1100 | 1500 | >1500 |
| **121-180** | 1400 | 2000 | >2000 |
| **181+** | 2000 | 3000 | >3000 |

ITracer API 使用与范围中相同的 XML（属性）文件中定义的值实现了阀值报告。范围和阀值定义在两个方面稍有不同。首先，阀值定义的关键值是一个正则表达式。当 ITracer 在跟踪一个数值 时，它会检查阀值正则表达式是否匹配被跟踪指标的复合名称。如果匹配，则阀值会将测量数据分类为 Ok、Warn 或 Critical，并为跟踪附加一个额外的 tracer.traceIncident。其次，由于阀值只定义了两个值（根据定义，Critical 值大于 warn 值），因此配置只由两个数值组成。清单 8 显示了之前介绍的薪水处理 SLA 的阀值配置：

清单 8. 薪水处理的阀值配置

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

<!-- Payroll Processing Thresholds -->

<entry key="Payroll Processing.\*81-120 Emps.\*Elapsed Time \(ms\)">1100,1500</entry>

<entry key="Payroll Processing.\*1-10 Emps.\*Elapsed Time \(ms\)">280,400</entry>

<entry key="Payroll Processing.\*11-50 Emps.\*Elapsed Time \(ms\)">850,1200</entry>

<entry key="Payroll Processing.\*51-80 Emps.\*Elapsed Time \(ms\)">900,1100</entry>

<entry key="Payroll Processing.\*121-180 Emps.\*Elapsed Time \(ms\)">1400,2000</entry>

<entry key="Payroll Processing.\*181\+ Emps.\*Elapsed Time \(ms\)">2000,3000</entry>

</properties>

图 12 显示添加了阀值指标的薪水处理的指标树：

图 12. 添加了阀值的薪水处理指标

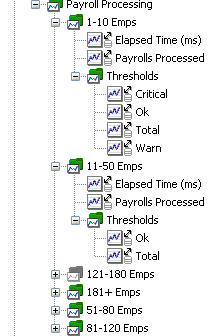
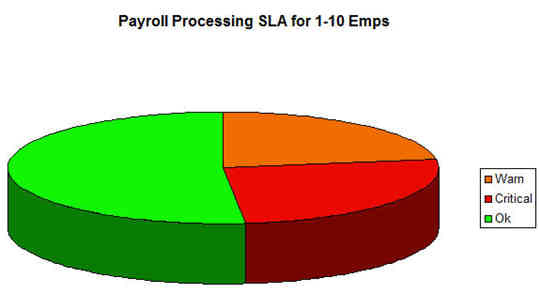


图 13 演示了哪些收集的数据可以表示在饼形图中：

图 13. 薪水处理的 SLA 汇总（1 到 10 名员工）



确保查找上下文和阀值分类的效率和速度非常重要，因为它们在完成实际工作的线程中执行。在 ITracer 实现中，所有指标名称在第一次被跟踪程序发现时，将存储在（线程安全）为具备和不具备阀值的指标指定的映射中。当特定指标的跟踪事件发生后，阀值确定过程占用的时间是一个 Map 查找时间，它的速度通常足够快。如果阀值条目或指标名称的数量非常大，则一种合理的解决方案是推迟阀值确定，并在异步跟踪线程池中处理它们。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-rtm1/#ibm-pcon)

第 1 部分结束语

本系列文章的第 1 部分介绍了一些监控反面模式和一些 APM 系统需要的属性。我总结了一些通用性能数据收集模式，并介绍了 ITracer 接口，我将在本系列文章的其余部分继续使用它。我已经演示了监控 JVM 健康状况的技巧，以及如何通过 JMX 获取通用性能数据。最后，我总结了各种实现高效和防代码更改的源代码级插装方法（用于监控原始性能统计数据和上下文派生统计数据），以及如何使用这些统计数据生成关于应用程序 SLA 的报告。第 2 部分将探究插装 Java 系统而无需修改应用程序源代码的技巧，具体方法是使用截取、类包装和动态字节码插装。