Perfmon4j

Perfmon4j User Guide

Performance Monitoring API for Java Enterprise Applications

Purpose

Perfmon4j has been successfully deployed in hundreds of production java systems over the last 5 years. It has proven to be a highly successful tool to measure performance while production servers are under load.

We have found Perfmon4j typically enables us to diagnose performance related issues in hours instead of weeks. This is accomplished by the ability to deploy Perfmon4j in a very low overhead dormant state. If/when performance issues occur it is possible to activate monitoring on targeted portions of the application code to isolate any bottlenecks. The overhead of this targeted monitoring is minimal allowing measurements to be obtained as the problems are occurring. Since monitoring can be configured at runtime there is no need for inconvenience users with an application restart.

Java Versions

1.5 and 1.6

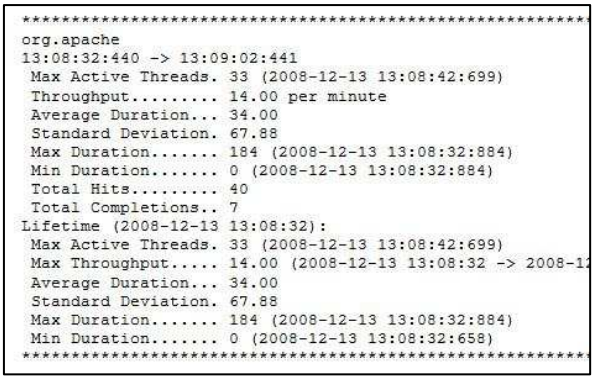
License

Features

Interval Timers

Perfmon4j groups timing into configurable intervals. This allows you to not only evaluate the duration of operations, but how those durations respond over time.

Perfmon4j takes snapshots of performance characteristics at specific configurable intervals. For example you could configure a monitor that would record web request/response time every minute throughout the day.



For each interval the following attributes can be recorded:

- Operation duration (Max, Min, Average, Median, Standard Deviation)
- Throughput (Operations started and completed during the duration)
- Configurable thresholds (Percentage of operations exceeding a specified duration)
- Max concurrent threads
- What percentage of my method execution time is in SQL?

Capturing this data at specified intervals allows you to analyze how your system responds over time and under load.

For example:

- What is my peak load?
- What time of day is my system under peak load?
- When and how often throughout the day does my system not meet my response time criteria?
- Where are the bottlenecks in my system? (Based on max concurrent threads)

Timers can be specified through the several methods, many of which do not require source code modification. Timers can be added through the following methods:

- Class/Method Timers can be inserted dynamically, via Java Agent instrumentation, as classes are loaded. This method allows you to monitor java applications or classes even without source code access.
- Web Request Timers can be added through a Servlet filter or an Apache Tomcat valve without source code modification.
- Annotation based Timers can be used to mark specific methods for timing. The Java Agent will then build timers at class load time.
- Inline Code Based Timers can be added via the Perfmon4j API to arbitrary code segments.

```
@DeclarePerfMonTimer("CircMainBean.processLostCopy")
private void processLostCopy( SessionStoreProxy store, CircMainPac
    RemoteException {

    String statusString = CopySpecs.getCopyStatusAsString(circPack

    AuditFacade af = AuditFacadeSpecs.getInterface();
    boolean patronEmpowered = circPackage.circulationMode == CircM
```

```
PerfMonTimer timer = PerfMonTimer.start("Browse.inner");
try {
    if (!tempHeadings.isEmpty()) {
        Object[] values = tempHeadings.toArray();
    ...
        Arrays.sort(values);
    }
    results.setHeadings(realHeadings);
} finally {
    PerfMonTimer.stop(timer);
}
```

Interval timer hirearchy

Perfmon4j bundles timers in a package like hierarchy. This provides the ability to choose to monitor performance/load from an overview level or zoom in to detail.

For example, consider an application that has the following class structure:

```
org.myapp.circ.CheckoutBean, org.myapp.circ.CheckInBean; org.myapp.cat.AddTitleBean
```

If timers where inserted in this code through Class/Method timers you could monitor performance at the package, class, or method level. For instance a monitor name of "`org.myapp.circ`" would measure the total time spent in any methods of the `CheckoutBean` and `CheckInBean` classes. A monitor on "`org.myapp`" would also include any methods on the `AddTitleBean`.

When configuring monitors a pattern can be applied to the root monitor name to indicate which timers should be active within a package. For example you could specify "`org.myapp.circ`" and include the pattern "`/**`" to specify monitors on all descendents. This would result in output for both, `CheckoutBean` and `CheckInBean` as well as each individual method.

Thread Trace Sampling

Thread trace sampling provides a quick way to sample a process thread on a production system. This technique is particularly successful at tracking down bottlenecks in high volume processes.

For example consider a keyword search engine that handles several thousand requests per minute. In this environment you might notice during periods of peak load throughout the day where average response time slows to an unacceptable level. It would be very helpful to have a detailed picture of where each of these process threads spends the majority of their time. At the same time any overhead of extensive monitoring during these peak loads could result in further performance degradation.

```
2009-05-06 18:19:30    INFO  PerfMon
*******************************************************************************
+-18:19:29:636 (953) BibImportBean
|    +-18:19:29:636 (953) BibImportBean.importBib.inner
|    |    +-18:19:29:636 (187) BibImportBean.addOrUpdateBib.findDuplicateTitle
|    |    |    +-18:19:29:636 (187) TitleAEDBean.findDuplicateTitleHashMap
|    |    |    |    +-18:19:29:636 (109) TitleAEDBean.findDuplicateTitleID.strict
|    |    |    |    +-18:19:29:745 TitleAEDBean.findDuplicateTitleID.strict
|    |    |    +-18:19:29:823 TitleAEDBean.findDuplicateTitleHashMap
|    |    +-18:19:29:823 BibImportBean.addOrUpdateBib.findDuplicateTitle
|    |    +-18:19:29:870 (156) BibImportBean.addOrUpdateBib.attachSite
|    |    +-18:19:30:026 BibImportBean.addOrUpdateBib.attachSite
|    |    +-18:19:30:042 (469) BibImportBean.importBib.copies
|    |    +-18:19:30:511 BibImportBean.importBib.copies
|    +-18:19:30:589 BibImportBean.importBib.inner
+-18:19:30:589 BibImportBean
*******************************************************************************
```

With a thread trace sampler you can specify a random sampling factor. This factor will limit the monitoring overhead to only a small subset of the total load. The thread trace will provide a detailed stack trace hierarchy for the sampled process. This stack trace will provide details on the duration of each sub-process. This trace can quickly identify bottlenecks and areas for optimization.

Snap Shot Monitors

Snap shot monitors can be configured to take performance measurements at arbitrary intervals. Snapshot classes can be configured to measure virtually any system counter/event. For example you could measure the total number of bytes sent/received by a web container.

SQL/JDBC Monitoring (New in Perfmon4j 1.1.0)

Perfmon4j can instrument and insert monitors on the classes and methods associated with any JDBC driver. This option must be selected at system startup via the "`-eSQL`" javaagent parameter (See JavaAgent Configuration for Details). This is available for use in two powerful ways.

1. You can monitor the average duration and frequency of specific JDBC methods. For example to monitor the frequency of calls to "`Statement.executeQuery`" you can create a monitor on "`SQL.executeQuery`.
2. When this feature is available all other monitors will contain information on the duration, in milliseconds, the particular monitor spent in JDBC/SQL processing. This is particularly valuable on a request level as it will allow you to determine the amount of time the request spent in Java vs. SQL. In the example below the output indicates for each web request to an average of 524 milliseconds, of that time 79 milliseconds was spent in JDBC/SQL requests.

```
************************************************************
2010-09-28 15:50:22     INFO    TextAppender    ()
************************************************************
WebRequest.cataloging
15:49:22:735 -> 15:50:22:774
 Max Active Threads. 47 (2010-09-28 15:49:55:947)
 Throughput..... 648.00 per minute
 Average Duration... 524.00
 Median Duration.... 520.0
 > 2 seconds..... 0.43%
 > 5 seconds..... 0.00%
 > 10 seconds..... 0.00%
 Standard Deviation. 0.00
 Max Duration..... 699 (2010-09-28 15:49:56:471)
 Min Duration..... 245 (2010-09-28 15:49:54:949)
 Total Hits..... 648
 Total Completions.. 648
 (SQL)Avg. Duration. 79.00
 (SQL)Std. Dev..... 0.00
 (SQL)Max Duration.. 79 (2010-09-28 15:49:56:471)
 (SQL)Min Duration.. 12 (2010-09-28 15:49:54:494)
************************************************************
```

Note: this example shows the data output to a text appender, this same data can also be written to a SQL table for reporting/processing.

Appender (Output) architecture

Perfmon4j can output data to an application server log or SQL Database. It is also possible to write and deploy custom appenders to extend this functionality.

Configuration

Configuration of perfmon4j is divided into a launch time and runtime sections.

Launch time Configuration

At launch time the perfmon4j java agent is installed and configured. With this configuration you must decide which classes, methods and annotations will be instrumented and

> Important
>
> Perfmon4j uses the javassist project to insert instrumentation via byte code modification. The javassist.jar class distributed with perfmon4j must be installed as an "endorsed" jar file.

available for monitoring. The instrumentation process is done through byte time engineering. Based on the load time configuration timer code will be inserted but will not be active. While inactive the timer overhead is extremely minimal.

For a web application additional launch time configuration can be used to configure web request timings. This configuration can be done via a Servlet filter or an Apache Tomcat Valve

JavaAgent Configuration

Perfmon4j is installed through the java agent command on the java command line. The format to add the java agent is:

```
-javaagent:<path to perfmon4j.jar>/perfmon4j.jar=<configuration options>
```

The configuration options are specified with a comma separated list of options. The available options are:

Option	Description	Examples
`-f`	This option specifies the location of the perfmonconfig.xml configuration file. This file will be checked for modification every 60 seconds. If the file does not exist perfmon4j will start in an inactive mode.	`-f../config/perfmonconfig.xml`
`-e`	(repeatable)This option is used to indicate a package or class name for instrumentation. Every method of any class within the specified package hierarchy will have a timer inserted. By default the extreme instrumentation will not create a monitor for methods that match the bean setter/getter pattern. You can enable instrumentation on getter and `setters` by prefixing the class/package name with `(+getter,+setter)`	`-eorg.jboss,-ecom.follett,` `-e(+getter,+setter)com.follet.bean`

`-eSQL (1.1.0+)`	(Repeatable)Perfmon4j can instrument the classes associated with a JDBC driver. This will create a monitor on each method associated with the drive implementation in the format `SQL.<method name>`. By default all classes loaded will be examined to find the JDBC classes. You can limit which classes are instrumented by specifying the following options:
	`JTDS` - Instrument JTDS driver for Microsoft SQL Server.
	`MYSQL` - Instrument the MYSQL driver
	`DERBY` - Instrument the DERBY driver
	`POSTGRESQL` - Instrument the postgresql driver.
	`ORACLE` - Instrument the oracle driver
	`<package>` - Only instrument classes in the specified package.
		`-eSQL`
		`-eSQL(JTDS)`
		`-eSQL(MYSQL)`
		`-eSQL(POSTGRESQL)`
		`-eSQL(ORACLE)`
		`-eSQL(org.my.jdbc.impl)`

`-a`	(repeatable) This option is used to indicate a package or class name for instrumentation. Any methods, containing a `DeclarePerfMonTimer` annotation will, have a timer inserted	`-aorg.jboss,-acom.follett`
`-i`	(repeatable) This option is used to override the –e and –a options. Any class or packages within the specified package hierarchy will be ignored.	`-eorg.jboss,`**`-iorg.jboss.system`** This will instrument all classes within `org.jboss`, except any classes in `org.jboss.system`
`-d`	Enables debug level logging in perfmon4j. Defaults to false.	`-dtrue`
`-v`	Enables verbose instrumentation output. This will output each method that is instrumented and the name of the monitor created.	`-vtrue`
`-b`	This is an experimental feature. When this option is true perfmon4j will instrument ALL classes loaded by the bootstrap loader. When this option is false (default) classes that are loaded before the javaagent is loaded are NOT instrumented.	`-btrue`
`-r`	Used with the `-f` option. Will override the duration in seconds that the perfmonconfig.xml file will be checked for modification.	`-fc:/perfmonconfig.xml,`**`-r360`**
`-g`	(Added in 1.0.2.GA)	`-gtrue`

	This is a bit of an "out of place" feature for a performance monitoring software, however we did find it fit a need for our application and we wanted to make it available to others. The use case is a desire to disable all calls to `System.gc()`, similar to the Sun jvm parameter `-XX:-DisableExplicitGC`. However, unlike that parameter we still want the ability to force a complete GC, including a memory compaction, when the system is NOT under extreme load. This parameter can be used to disable `System.gc()` while still allowing calls to `Runtime.getRuntime().gc()`.	

Examples:

The following will instrument any annotation within `com.follett.fsc` package along with every method on every class within the `catalina` package. The runtime configuration will be loaded from `c:\perfmon\config.xml`.

```
-javaagent:../perfmon4j.jar=-fc:/perfmon/config.xml,-acom.follett.fsc,-eorg.apache.catalina
```

The following will instrument all classes within `com.follett` EXCEPT the class `com.follett.StringHelper`

```
-javaagent:../perfmon4j.jar=-fc:/perfmon/config.xml,-ecom.follett,-icom.follett.StringHelper
```

Servlet Filter Configuration

To add Servlet request monitoring to a web application you can install the Perfmon4j `org.perfmon4j.servlet.PerfMonFilter` through your web applications web.xml file. This filter will intercept all web requests, based on the filter-mapping, with an interval timer. See http://java.sun.com/products/servlet/Filters.html for details on how to install a Servlet filter. The filter can be configured with the following init parameters:

```
<filter>
  <filter-name>PerfMon4JFilter</filter-name>
  <filter-class>org.perfmon4j.servlet.PerfMonFilter</filter-class>
 </filter>

<filter-mapping>
  <filter-name>PerfMon4JFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Parameter	Description	Default Value
BASE_FILTER_CATEGORY	This is the root Perfmon4j category that all web request will be grouped under. The Servlet url will be parsed to build the category hierarchy. For example the web request `/circulation/handlecheckout?barcode=T1` will be mapped to the `WebRequest.circulation.handlecheckout` category.	`WebRequest`
ABORT_TIMER_ON_REDIRECT	This option allows you to ignore redirects from the total completion counts (total completions, max, min, average duration, etc).	`false`
ABORT_TIMER_ON_IMAGE_RESPONSE	This option allows you to ignore request that return an image mime type from the total completion counts (total completions, max, min, average duration, etc).	`false`
ABORT_TIMER_ON_URL_PATTERN	This option allows you to provide a Java Regular Expression pattern. Any request matching this pattern will NOT be included in the total completion counts (total completions, max, min, average duration, etc).	`null`

Valve Configuration

Perfmon4j can be configured to monitor all web requests via a Web Valve under Tomcat and JBossWeb installation.

Note: The Perfmon4j-tomcat-ConfigGuide documentation contains an example configuration of a web valve.

Runtime Configuration

The ability to enable/disable monitoring at run time is an important feature to enable diagnoses of performance issues on production systems. Many types of bottlenecks can be difficult to reproduce in a test environment. Also when a slow-down occurs in production you probably don't have the luxury of shutting down the system to configure additional monitors. In order to work in a production environment Perfmon4j allows monitoring to be configured "on the fly".

The java load time configuration, documented above, inserts timers in a low overhead, passive state. Typically the only measurable performance penalty is an increase in java class load time. Those timers remain in this passive state until you activate them, at runtime, with the `perfmonconfig.xml` file. (The name/location of this file is specified via the `javaagent` command line).

• Root XML Element

The root xml element for the config file is the PerfMonConfig element. This element has one optional tag attribute named enabled which can be set to true or false. The enable attribute provides a convenient method to toggle Perfmon4j between an active and inactive state.

All other elements must exist in the body of the root tag.

Tag Attributes

Name	Values	Description
enabled	true OR false	Default value is true. Provides a quick way to enable/disable perfmon4j. When disabled perfmon4j will be set to an inactive state.

• Defining Appenders

Appenders are java classes that perform the task of polling monitors to retrieve and output event data at regular defined intervals. The `org.perfmon4j.TextAppender` is the default class used to output logging events to the default log output (Log4j, Java Logging or `stdout`). Custom appenders can be written to output the data to SQL tables or any other sources by extending the `org.perfmon4j.Appender` class.

Appenders are defined by the `appender` tag inside the body of the `PerfMonConfig` tag.

Tag Attributes

Name	Values	Description
name	Any String	An arbitrary symbolic name that is use to map monitors to appenders.
className	A java class that extends the `org.perfmon4j.Appender` class	
interval	A duration that indicates how often monitors attached to this appender should be polled.	Valid duration strings include a quantity and a unit of measure. Acceptable units of measure include: seconds, minutes and hours.

The body of the appender tag can contain 1 or more attribute tags. Two optional attributes are the `MedianCalculator` and the `ThresholdCalculator`.

Adding a `MedianCalculator` to an appender will result in the median duration of all timings to be calculated. The body of the median calculator can optionally contain the following values `factor` and `maxElements`.

`factor` is used to specify a divisor to group durations for median calculation. The default `factor` is 100 which indicates durations will be grouped into 1/10 of a second blocks.

`maxElements` is the maximum number of unique instances occurrences that will be used to calculate the median. Since a true median calculation requires storing each occurrences of a value in a range, the median calculator uses an algorithm to limit the total storage size. In cases of overflow the median calculator will return a median value that indicates the lowest or highest possible value for the median. The default `maxElements` is 500.

A `ThresholdCalculator` can be used to measure a percentage of durations that exceed one or more threshold durations. The body of the threshold attribute must contain a string with a comma separate list of durations.

Derived appenders can define additional attributes. When the appender is constructed the appropriate setter will be invoked to pass the values to the appender instance.


```
<Perfmon4JConfig enabled='true'>
      <appender name='HighRes' className='org.perfmon4j.TextAppender' interval='30 seconds'/>
      <appender name='MedRes' className='org.perfmon4j.TextAppender' interval='5 minutes'/>
      <appender name='LowRes' className='org.perfmon4j.TextAppender' interval='1 hour'>
            <attribute name='medianCalculator'>factor=10</attribute>
            <attribute name='thresholdCalculator'>1 second, 2 seconds, 5 seconds</attribute>
      </appender>
      • • •
```

(Version 1.1.0+) When perfmon4j is enabled to include SQL/JDBC implementation a text appender can be optionally configured to omit the SQL detail information associated with a monitor. The attribute to use this feature is: `includeSQL`. By default SQL information will be included.

Example:

```
<Perfmon4JConfig enabled='true'>
      <appender name='high-res' className='org.perfmon4j.TextAppender' interval='1 minutes'>
            <!-- Do NOT include optional SQL durations -->
            <attribute name='includeSQL'>false</attribute>
      </appender>
      ...
```

• Defining Monitors

The monitor tag is used to attach interval timers to appenders. The monitor tag is used to specify a root interval timer category. One or more appenders can be attached to an interval timer by adding an appender tag to the body of the monitor

Tag Attributes

Name	Values	Description
name	Any String	A string that represents an interval timer or a root level package of an interval timer.

The body of the tag should contain one or more appender body tags. These tags are used to associate this monitor with an appender. The appender name attribute must match a defined appender (See: Defining Appenders).

Appender Mapping Tag Attributes

Name	Values	Description
name	Any String	Must match an appender defined with an appender tag in the root element
pattern	One of the hierarchy patterns (See	The pattern is used to indicate a set of timers to attach to the appender.

Hierarchy Pattern

'.'	Parent only
'./*'	Parent and all children
'./**'	Parent and all descendents
'/*'	All children
'/**'	All descendents

Using hierarchy patterns, groups of timers can be attached to a single appender.


```
<Perfmon4JConfig enabled='true'>
      <appender name='MedRes' className='org.perfmon4j.TextAppender' interval='5 minutes'/>
      <appender name='LowRes' className='org.perfmon4j.TextAppender' interval='1 hour'/>

      <monitor name='WebRequest'>
            <appender name='LowRes' pattern='/*'/>
            <appender name='MedRes' pattern='.'/>
      </monitor>
      • • •
```

• Defining Thread Trace Monitors

The `threadTrace` tag is used to configure associate a thread trace monitor with any interval monitor. These monitors are typically defined in a 'sampling' mode where a detailed stack snap shot is captured based on a randomly selected execution.

The `threadTrace` tag contains the following optional attributes:

- `maxDepth` – Indicates the maximum nesting depth of the monitor stack to capture. This is useful to reduce the overall volume of data captured.
- `minDurationToCapture` – Indicates the minimum duration required to include a nested timer in the output.
- `radomSamplingFactor` – For detail monitoring of high throughput operations a sampling factor is used to reduce the overhead associated with capturing detailed thread stack diagnostic information. The value indicates the percentage of operations that will be monitored. For example a value of 100 indicates the thread trace detail will be captured for 1 out of every 100 executions.

```
<appender name='high-res' className='org.perfmon4j.TextAppender' interval='1 minutes'/>

<threadTrace monitorName='WebRequest'
      maxDepth='10'
      minDurationToCapture='10 ms'
      randomSamplingFactor='500'>
      <appender name='high-res' pattern="."/>
</threadTrace>
```

The thread trace configuration can also take an optional Triggers tag (1.0.2.GA+). This tag can define one or more triggers that will be used to initiate a stack trace (if any one of the triggers match the stack trace will be displayed). The following example has all of the available Triggers defined:

```
<Perfmon4JConfig enabled='true'>
      <appender name='myappender' className='org.perfmon4j.TextAppender'
            interval='1 minutes'/>

      <threadTrace monitorName="WebRequest.examples.servlets" maxDepth="10">
            <appender name='myappender'/>
            <Triggers>
                  <HTTPRequestTrigger name="firstname" value="Dave"/>
                  <HTTPSessionTrigger attributeName='UserID' attributeValue='200'/>
                  <ThreadNameTrigger threadName='Processor-http:localhost:8080'/>
                  <ThreadPropertyTrigger name='jobID' value='300'/>
                  <HTTPCookieTrigger name="MyCookie" value="1"/>
            </Triggers>
      </threadTrace>
</Perfmon4JConfig>
```

• Defining Snap Shot Monitors

Documentation Coming Soon

For more information

See the Perfmon4j_ConfigSamples.pdf.

Rev 4 - Updated 10/5/2010

