SimpleJMX Package

Version 1.0
August 2012

Gray Watson

Table of Contents

SimpleJMX

Version 1.0 – August 2012

This package provides some simple classes to help with the publishing of objects using Java's Management Extensions (JMX). These published objects can be investigated with jconsole (Sun's graphical JMX user interface application) or another JMX client. Included is also a programmatic JMX client which you can use to connect to and interrogate remote JMX servers.

To learn how to use SimpleJMX, see Chapter 1 [Using], page 2. You can also take a look at the the examples section of the document which has various working code packages. See Chapter 2 [Examples], page 8. There is also a HTML version of this documentation.

Gray Watson `http://256.com/gray/`

1 Using SimpleJMX

1.1 Downloading SimpleJMX Jar

To get started with SimpleJMX, you will need to download the jar files. The SimpleJMX release page is the default repository but the jars are also available from the central maven repository and from Sourceforge.

The code works with Java 5 or later.

1.2 Starting a JMX Server

The `JmxServer` class is used to publish a JMX server that jconsole and other JMX clients can connect to. The easiest way to do this is to choose a port to use, define the server, and then start it:

```
// create a new JMX server listening on a port
JmxServer jmxServer = new JmxServer(8000);
// start our server
jmxServer.start();
```

Before your program exits, it is best to stop the server, so the following try/finally block is a good pattern to use:

```
// create a new JMX server listening on a port
JmxServer jmxServer = new JmxServer(8000);
try {
   // start our server
   jmxServer.start();
   ...
   // register objects with the server anddo other stuff here
   ...
} finally {
   // un-register objects
   // stop our server
   jmxServer.stop();
}
```

1.3 Registering Objects Using JMX

To published objects via the server via JMX you must register them with the `JmxServer`:

```
jmxServer.register(someObject);
```

There also is an `unregister(...)` method which will un-publish from the server:

```
jmxServer.unregister(someObject);
```

The objects that are registered must be named and the fields and methods that are to be exposed must be specified.

1.4 Naming Objects In JMX

When you publish an object you need to tell JMX what the object's *unique* name is and where it should be located in the various folders shown by jconsole. There are a couple of different ways to do this with SimpleJMX:

1.4.1 @JmxResource Annotation

The `@JmxResource` annotation is used to define that the resource is to be published via JMX and how it is to be named.

```
@JmxResource(domainName = "j256", description = "Runtime counter")
public class RuntimeCounter {
   ...
```

The above example shows that the `RunetimeCounter` object is to be published via JMX inside of the `j256` folder with a text description.

The fields in `@JmxResource` are:

`domainName`

Domain name of the object which turns into the top-level folder inside of jconsole. This must be specified unless the object is self-naming. See Section 1.4.2 [Self Naming], page 4.

`beanName`

Name of the JMX object in the jconsole folder. The default is to use the class name.

`folderNames`

Optional array of strings which translate into sub-folders below the domain-name that was specified above. Default is for the object to show up in the domain-name folder. The folder names can either be in `name=value` format in which case they should be in alphabetic order by name. They can also just be in `value` format.

Folders are used when you have a large number of JMX objects being published and you want to group the objects so that you can find them faster than scrolling through a large list. For example, all of your database objects could go in the folder "Database" while the database connections could go into the sub-folder "Database/Connections".

```
@JmxResource(domainName = "j256",
    folderNames = { "Database", "Connections" },
    description = "MySql datatbase connection")
public class MySqlDatabaseConnection {
```

`description`

Textual description of the class for jconsole or other JMX clients. Default is something like: "Information about class-name".

1.4.2 Self Naming Objects

Instead of using the `@JmxResource` annotation to define the name/folders for your JMX object, you can have the object implement the `JmxSelfNaming` interface. This allows the object to name itself and will override any settings from the `@JmxResource` annotation, if it is specified.

It is particularly necessary to make your object `JmxSelfNaming` if there are to be multiple of them published via JMX. For example, if you have multiple database connections that you want to publish then to ensure that they have a *unique* name, each of the objects should be self-naming and should provide a unique name that identifies itself:

```
// we only use this to set the domain name and the folders
@JmxResource(domainName = "j256", folderNames = { "Database", "Connections" })
public class DatabaseConnection extends BaseJmxSelfNaming
      implements JmxSelfNaming {
    @Override
    public String getJmxNameOfObject() {
        // return our toString as our name
        return toString();
    }
}
```

In the above example, we extend the `BaseJmxSelfNaming` abstract class which has default implementations for all of the `JmxSelfNaming` methods, so all we need to do is override what we want to change.

The methods in the `JmxSelfNaming` interface are:

`String getJmxDomainName();`
: Return the domain name of the object. Return null to use the one from the `@JmxResource` annotation instead.

`String getJmxNameOfObject();`
: Return the name of the object. Return null to use the one from the `@JmxResource` annotation instead.

`JmxFolderName[] getJmxFolderNames();`
: Return the appropriate array of folder names used to built the associated object name. Return null for no folders in which case the bean will be at the top of the hierarchy in jconsole without any sub-folders.

1.5 Exposing Fields and Methods Over JMX

Once we have named our object, we need to tell the JMX server which fields and methods should be exposed to the outside world. JMX can expose what it calls attributes, operations, and notifications. At this time, only attributes and operations are supported.

Attributes can be primitives or simple types such as `String` or `java.util.Date`. With SimpleJMX you can expose them by using reflection on the object's fields directly using the `@JmxAttributeField` annotation or instead via the get/set/is methods using the `@JmxAttributeMethod` annotation.

Operations are methods that do *not* start with get/set/is but which perform some function (ex: `resetTimer()`, `clearCache()`, etc.). They can be exposed with the `@JmxOperation` annotation.

1.5.1 @JmxAttributeField Annotation

SimpleJMX allows you to publish your primitive or simple types by annotating your fields with the `@JmxAttributeField` annotation.

```
@JmxAttributeField(description = "Start time in millis",
   isWritable = true)
private long startTimeMillis;
```

In the above example, the `startTimeMillis` long field will be visible via JMX. It will show its value which can be changed because `isWriable` is set to true – `isReadable` is set to true by default. The description is available in jconsole when you hover over the attribute.

The fields in the `@JmxAttributeField` annotation are:

`String description`
: Description of the attribute for jconsole. Default is something like: "someField attribute".

`boolean isReadible`
: Set to false if the field should not be read through JMX. Default is true.

`boolean isWritable`
: Set to true if the field can be written by JMX. Default is false.

1.5.2 @JmxAttributeMethod Annotation

Instead of publishing the fields directly, SimpleJMX also allows you to publish your attributes by decorating the get/set/is methods using the `@JmxAttributeMethod` annotation. This is *only* for methods that start with `getXxx()`, `setXxx(...)`, or `isXxx()`.

The `Xxx` name should match precisely to line up the get and set JMX features. For example, if you are getting and setting the `fooBar` field then it should be `getFooBar()` and `setFooBar(...)`. `isFooBar()` is also allowed if `foobar` is a `boolean` or `Boolean` field.

Notice that although the field-name is `fooBar` with a lowercase 'f', the method name camel-cases it and turns it into `getFooBar()` with a capital 'F'. In addition, the `getXxx()` method must not return void and must have no arguments. The `setXxx(...)` method must return `void` and must take a single argument. The `isXxx()` method is allowed if it returns boolean or Boolean and the method has no arguments.

Exposing a get method also allows you to do some data conversion when the value is published. Exposing a set method allows you to do data validation.

```
@JmxAttributeMethod(description = "Run time in seconds or milliseconds")
public long getRunTime() {
```

The only field in the `@JmxAttributeMethod` annotation is the description. The annotation on the `get...` method shows that it is readable and the annotation on the `set...` method shows that it is writable.

1.5.3 @JmxOperation Annotation

Operations are methods that do *not* start with get/set/is but which perform some function. They can be exposed with the `@JmxOperation` annotation. The method can either return `void` or an object. It is recommended that the method return a primitive or a simple object that will be for sure in jconsole's classpath. It also should not throw an unknown `Exception` class.

The fields in the `@JmxOperation` annotation are:

`String description`
: Description of the attribute for jconsole. Default is something like "someMethod operation".

`String[] parameterNames`
: Optional array of strings which gives the name of each of the method parameters. The array should be the same length as the `parameterDescriptions` array. Default is something like "p0".

```
@JmxOperation(parameterNames = { "minValue", "maxValue" },
   parameterDescriptions = { "low water mark",
      "high water mark" }
public void resetMaxMin(int minValue, int maxValue) {
   ...
```

`String[] parameterDescriptions`
: Optional array of strings which describes each of the method parameters. The array should be the same length as the `parameterNames` array.

`OperationAction operationAction`
: This optional field is used by the JMX system to describe what sort of work is being done in this operation.

1.6 Publishing Using Code Definitions

Sometimes, you want to expose a class using JMX but you don't control the source code or maybe you don't want to put the SimpleJMX annotations everywhere. If this is the case then you also have the option to expose just about any object programmatically.

The `JmxServer` has a register function which takes just an `Object`, its `ObjectName`, and an array of attribute-fields, attribute-methods, and operations.

The attribute-fields are specified as an array of `JmxAttributeFieldInfo` objects that are associated with fields that are exposed through reflection:

```
JmxAttributeFieldInfo[] attributeFieldInfos =
   new JmxAttributeFieldInfo[] {
     new JmxAttributeFieldInfo("startMillis", true /* readable */,
        false /* not writable */, "When our timer started"),
   };
```

The attribute-methods are specified as an array of `JmxAttributeMethodInfo` objects that are associated with fields that are exposed through get/set/is methods:

```
JmxAttributeMethodInfo[] attributeMethodInfos =
   new JmxAttributeMethodInfo[] {
      new JmxAttributeMethodInfo("getRunTime",
         "Run time in seconds or milliseconds"),
   };
```

The operations are specified as an array of `JmxOperationInfo` objects that are associated with operation methods:

```
JmxOperationInfo[] operationInfos =
   new JmxOperationInfo[] {
      new JmxOperationInfo("restartTimer", null /* no params */,
         null  /* no params */, OperationAction.UNKNOWN,
         "Restart the timer"),
   };
```

To register the object you would then do:

```
jmxServer.register(someObject,
   ObjectNameUtil.makeObjectName("j256", "SomeObject"),
   attributeFieldInfos, attributeMethodInfos, operationInfos);
```

Take a look at the random-object example test for a working example. See [random object example], page 8.

1.7 Using the JMX Client

SimpleJMX also includes a programmatic, simple JMX client which you can use to interrogate JMX servers. You connect to the server by specifying the host/port of the server.

```
JmxClient = client = new JmxClient("server1", 8000);
```

You can then get attributes:

```
boolean showSeconds =
    jmxClient.getAttribute("j256", "RuntimeCounter", "showSeconds");
```

If you need to construct the object name directly then you can use `ObjectName` static methods:

```
int availableProcessors =
   client.getAttribute(
      ObjectName.getInstance("java.lang:type=OperatingSystem"),
      "AvailableProcessors");
```

You can also call operations:

```
client.invokeOperation(
   ObjectName.getInstance("java.lang:type=Memory"),
   "gc");
```

You can also list the available attributes and operations and do a number of other things. See the `JmxClient` javadocs for more information

2 Example Code

Here is some example code to help you get going with SimpleJMX. I often find that code is the best documentation of how to get something working. Please feel free to suggest additional example packages for inclusion here. Source code submissions are welcome as long as you don't get piqued if we don't chose your's.

Simple, basic
: This is a simple application which publishes a single object. See the source code in SVN.

Random object example
: This is an example showing how to programmatically expose using JMX a random object without SimpleJMX annotations: See the source code in SVN.

3 Open Source License

Index of Concepts