# Proxy mode

# Jolokia agent as standalone application ·

Issue #280 · rhuss/jolokia https://github.com/rhuss/jolokia/issues/280

Would it be possible to build a runable JAR file so that I can run Jolokia as a service on all servers? We have some customers that do not accept that we deploy the Jolokia javaagent with their application, so I would like to deploy a jolokia service that can then connect to the JMX port of hte service (which is open). JMX is only listening on localhost so I will need to have many jolokias running.

I've currently built a NOOP class that I can run as a application and connect jolokia jvm agent to, but it would be nice if it was possible to deploy a standalone jolokia app.

Thomas

Actually that is the idea of using the jolokia proxy-mode : <https://jolokia.org/reference/html/proxy.html>

The idea is that you deploy a jolokia.war on a separate Tomcat and the agent proxies you to plain JSR-160 JMX via RMI.

Sorry for the late answer, issue slipped through somehow ...

Does this answer your question ?

Jolokia is typically used as an agent which attached externally without changing the existing application.

The alternative is to start Jolokia programmatically from within your application. For this there is a decent[Spring integration](https://jolokia.org/reference/html/jmx.html#jvm-spring) but you can also simply use the [JolokiaServer](https://github.com/rhuss/jolokia/blob/master/agent/jvm/src/main/java/org/jolokia/jvmagent/JolokiaServer.java) programmatically.

# vveloso/docker-jolokia-wildfly-proxy:

Docker image for a Jolokia proxy with support for Wildflys' remote JMX connections over HTTP. https://github.com/vveloso/docker-jolokia-wildfly-proxy

Docker image for a Jolokia proxy with support for connecting to Wildfly hosts via JMX over HTTP remoting.

Includes:

* [Wildfly](http://wildfly.org/) 10 client, thus supporting its additional remoting protocols for JMX.
* [Jolokia](https://jolokia.org/)
* [Tomcat](http://tomcat.apache.org/) server to host Jolokia as a proxy.

# rhuss/jolokia: JMX on Capsaicin

https://github.com/rhuss/jolokia

Jolokia is a fresh way to access JMX MBeans remotely. It is different from JSR-160 connectors in that it is an agent-based approach which uses JSON over HTTP for its communication in a REST-stylish way.

Multiple agents are provided for different environments:

* **WAR Agent** for deployment as web application in a JEE Server.
* **OSGi Agent** for deployment in an [OSGi](http://www.osgi.org/) container. This agent is packaged as a bundle and comes in two flavors (minimal, all-in-one).
* **Mule Agent** for usage within a [Mule](http://www.mulesoft.org/) ESB
* **JVM JDK6 Agent** which can be used with any Oracle/Sun JVM, Version 6 or later and which is able to attach to a running Java process dynamically.

打包到WAR，OSGi，Mule，JVM

Features

The agent approach as several advantages:

* **Firewall friendly**

Since all communication is over HTTP, proxying through firewalls becomes mostly a none-issue (in contrast to RMI communication, which is the default mode for JSR-160)

* **Polyglot**

No Java installation is required on the client side. E.g. [Jmx4Perl](http://www.jmx4perl.org/) provides a rich Perl client library and Perl based tools for accessing the agents.

* **Simple Setup**

The Setup is done by a simple agent deployment. In contrast, exporting JMX via JSR-160 can be remarkable complicated (see these blog posts for setting up [Weblogic](http://labs.consol.de/blog/jmx4perl/configuring-remote-jmx-access-for-weblogic) and [JBoss](http://labs.consol.de/blog/jmx4perl/jboss-remote-jmx) for native remote JMX exposure setup)

Additionally, the agents provide extra features not available with JSR-160 connectors:

* **Bulk requests**

In contrast to JSR-160 remoting, Jolokia can process many JMX requests with a single round trip. A single HTTP POST request puts those requests in its JSON payload which gets dispatched on the agent side. These bulk requests can increase performance drastically, especially for monitoring solutions. The Nagios plugin [check\_jmx4perl](http://search.cpan.org/~roland/jmx4perl/scripts/check_jmx4perl) uses bulk requests for its multi-check feature.

* **Fine grained security**

In addition to standard HTTP security (SSL, HTTP-Authentication) Jolokia supports a custom policy with fine grained restrictions based on multiple properties like the client's IP address or subnet, and the MBean names, attributes, and operations. The policy is defined in an XML format with support for allow/deny sections and wildcards.

* **Proxy mode**

Jolokia can operate in an agentless mode where the only requirement on the target platform is the standard JSR-160 export of its MBeanServer. A proxy listens on the front side for Jolokia requests via JSON/HTTP and propagates these to the target server through remote JSR-160 JMX calls. Bulk requests get dispatched into multiple JSR-160 requests on the proxy transparently.

Resources

* The [Jolokia Forum](https://jolokia.org/forum.html) can be used for questions about Jolokia (and Jmx4perl).
* For bug reports, please use the [Github Issue tracker](https://github.com/rhuss/jolokia/issues).
* Most of the time, I'm hanging around at [Freenode](http://webchat.freenode.net/?channels=jolokia) in #jolokia, too.

Even more information on Jolokia can be found at [www.jolokia.org](https://www.jolokia.org/), including a complete [reference manual](https://www.jolokia.org/reference/html/index.html).

Contributions

Contributions in form of pull requests are highly appreciated. All your work must be donated under the Apache Public License, too. Please sign-off your work before doing a pull request. The sign-off is a simple line at the end of the patch description, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are very simple: if you can certify the below (from [developercertificate.org](http://developercertificate.org/)):

Developer Certificate of Origin

Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.

660 York Street, Suite 102,

San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this

license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I

have the right to submit it under the open source license

indicated in the file; or

(b) The contribution is based upon previous work that, to the best

of my knowledge, is covered under an appropriate open source

license and I have the right under that license to submit that

work with modifications, whether created in whole or in part

by me, under the same open source license (unless I am

permitted to submit under a different license), as indicated

in the file; or

(c) The contribution was provided directly to me by some other

person who certified (a), (b) or (c) and I have not modified

it.

(d) I understand and agree that this project and the contribution

are public and that a record of the contribution (including all

personal information I submit with it, including my sign-off) is

maintained indefinitely and may be redistributed consistent with

this project or the open source license(s) involved.

Then you just add a line to every git commit message:

Signed-off-by: Max Morlock <max.morlock@fcn.de>

Using your real name (sorry, no pseudonyms or anonymous contributions.)

If you set your user.name and user.email git configs, you can sign your commit automatically with git commit -s.

If you fix some documentation (typos, formatting, ...) you are not required to sign-off. It is possible to sign you commits in retrospective, [too](http://stackoverflow.com/questions/13043357/git-sign-off-previous-commits) if you forgot it the first time.

# 6.6 Proxy requests

For proxy requests, POST must be used as HTTP method so that the given JSON request can contain an extra section for the target which should be finally reached via this proxy request. A typical proxy request looks like

{

"type" : "read",

"mbean" : "java.lang:type=Memory",

"attribute" : "HeapMemoryUsage",

"target" : {

"url" : "service:jmx:rmi:///jndi/rmi://targethost:9999/jmxrmi",

"user" : "jolokia",

"password" : "s!cr!t"

}

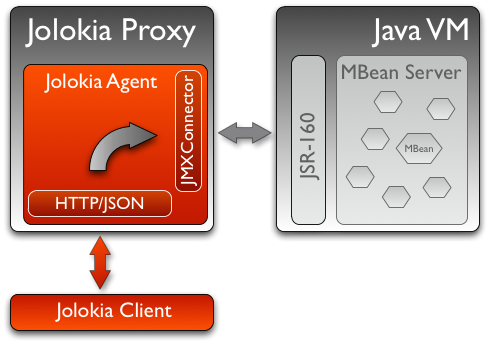
}

url within the target section is a JSR-160 service URL for the target server reachable from within the proxy agent. user andpassword are optional credentials used for the JSR-160 communication.

# Jolokia – JMX Proxy

https://jolokia.org/features/proxy.html

There are situations, where a deployment of an Jolokia agent on the target platform is not possible. This might be for political reasons or an already established JSR-160 export on the instrumented servers. In these environments, Jolokia can operate as a **JMX Proxy**. In this setup, the agent is deployed on a dedicated proxy JEE server (or other supported agent platform). The proxy bridges between Jolokia JSON request and responses to remote JSR-160 calls to the target server. The following diagrams gives an illustration of this setup.



A Jolokia proxy is universal and agnostic to the target server as it gets its information for the target via an incoming request (the same as for an HTTP proxy). Due to this required extended information, only Jolokia POST requests can be used for proxying since there is currently no way to encapsulate the target information within a GET Url. The base Jolokia URL for the request is that of the proxy server, whereas the target parameters are included in the request. In the next example, a proxied Jolokia request queries the number of active threads for a server jboss-as via a proxy tomcat-proxy, which has an agent deployed under the context jolokia. The agent URL then is something like

http://jolokia-proxy:8080/jolokia

and the POST payload of the request is

  {  
    "type":"READ"  
    "mbean":"java.lang:type=Threading",  
    "attribute":"ThreadCount",  
    "target": {   
                "url":"service:jmx:rmi:///jndi/rmi://jboss-as:8686/jmxrmi",  
                "password":"admin",  
                "user":"s!cr!t"  
              },  
  }

The target is part of the request and can contain authentication information as well (with params userand password)

Limitations

Operating Jolokia as a JMX proxy has some limitations compared to a native agent deployment:

* [Bulk requests](https://jolokia.org/features/bulk-requests.html) are possible but not as efficient as for direct operation. The reason is, that JSR-160 remoting doesn't know about bulk requests, so that a Jolokia bulk request arriving at the proxy gets dispatched into multiple JSR-160 requests for the target. The JSR-160 remote connection has to be established only once, though.
* The JMX target URL addresses the MBeanServer directly, so MBeanServer merging as it happens for direct operation is not available. Also, certain workarounds for bugs in the server's JMX implementation are not available. (e.g. see this [blog post](http://labs.consol.de/blog/jmx4perl/jboss-remote-jmx/) for a JBoss bug when accessing MXBeans in the PlatformMBeanServer)
* When no-standard Java types are returned by JMX operations or attribute read calls, these types must be available on the proxy, too. Using the Jolokia agent directly, complex data types are serialized deeply into a JSON representation automatically.
* For each Jolokia request, a new JMX connection (likely using RMI) is created which is an expensive operation. A future version of Jolokia will tackle this by providing some sort of optional JSR-160 connection pooling.

Next

* See why how Jolokia can [secure](https://jolokia.org/features/security.html) JMX access in a very fine granular way.
* Learn something about [bulk JMX requests](https://jolokia.org/features/bulk-requests.html).
* Go Back to the [Features Overview](https://jolokia.org/features-nb.html).

# Jolokia – Overview

https://jolokia.org/

Jolokia is a JMX-HTTP bridge giving an alternative to JSR-160 connectors. It is an agent based approach with support for many platforms. In addition to basic JMX operations it enhances JMX remoting with unique features like bulk requests and fine grained security policies.

Starting points

* Overview of [features](https://jolokia.org/features-nb.html) which make Jolokia unique for JMX remoting.
* The [documentation](https://jolokia.org/documentation.html) includes a [tutorial](https://jolokia.org/tutorial.html) and a [reference manual](https://jolokia.org/reference/html/index.html).
* [Agents](https://jolokia.org/agent.html) exist for many platforms (JEE, OSGi, Mule, JVM).
* [Support](https://jolokia.org/support.html) is available through various channels.
* [Contributions](https://jolokia.org/support.html#Source) are highly appreciated, too.

News

Polished with 1.3.5

2016-10-04

Here comes a minor update with some smaller goodies:

* Support of JSON streaming also for the AgentServlet which is included in the WAR and OSGi Agent (in addition to the JVM agent which got this support in the last release). This leads to much less temporary heap memory consumption when serializing the internal JSON objects to character data in the HTTP response. You still need to be careful when doing large operations like list since there is still a full in-memory representation of the data sent.
* Avoid an NPE in the Websphere detector and added detection of a Payara server
* Re-add hooks for creating custom restrictors as protected methods in AgentServlet which allows for simple programmatic customization.

Summer fun with Jolokia 1.3.4

2016-07-31

It has beed taken a bit, but just right now befire the summerbreak 1.3.4 is here with some nice new [features](https://jolokia.org/changes-report.html):

* SSL support for the J4pClient.
* JSON response streaming to reduce memory activity. This is enabled by default but can be switched off by setting the config option "streaming" to false.
* Allow a basic auth as alternative to client cert authentication when both a user and client certifcates are used.
* A "quiet" and a java.util.logging LogHandler which can be directly used.

In parallel 2.0 takes comes into shape. The current version 2.0.0-M3 is available and already used with success in some production setups. In addition to the new features like notification support or new extension hooks, it is fully backwards comptabile to 1.x, except that some default values will be changed. However, an upgrade will be trivial. If you are curious, I'm going to present the new 2.0 features at [JavaZone](https://2016.javazone.no/program/jolokia-2-0) in September.

That's it for now, enjoy your summer break ;-)

Jolokia 1.3.3

2016-02-16

Beside bug fixes as described in the [changelog](https://jolokia.org/changes-report.html), this minor release brings some small features:

* Custom restrictors for tuning access control can be added to the JVM and WAR agents (which already is supported by the OSGi agent for quite some time)
* Global configuration option allowErrorDetails can be used when starting the agent to avoid exposure of stack traces and exception messages globally.
* Configuration allowDnsReverseLookup can be set to false in order to avoid reverse DNS lookup for doing security host checks. That also implies that if switched off only plain IP adressess can be used in a jolokia-access.xml policy file.
* The password for opening a JVM agent's keystore can now be encrypted, too. You can use the java -jar jolokia-agent.jar encrypt CLI to encrypt a password which then can be used in the agent's configuration.

Welcome to 2016 - the year Jolokia 2.0 will see the light of day

2016-01-07

We are getting closer. I'm happy to announce that the first milestone release 2.0.0-M1 is out and available from Maven central. Of course, it is highly experimental. The main new features are JMX notification support (pull and SSE mode) and refactorings leading to an internal modularization (which you will see when looking into WAR agent).

I would be more than happy if you would try out the [JAR](http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-agent-jvm/2.0.0-M1/jolokia-agent-jvm-2.0.0-M1-agent.jar) and [WAR](http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-agent-war/2.0.0-M1/jolokia-agent-war-2.0.0-M1.war) agent which are supposed to be drop in replacements for Jolokia 1.3.2.

More information can be found on my [Blog](https://ro14nd.de/). Soon there will be also demo and screencast showing the new features.

Jolokia 1.3.2 is still the latest stable version and will receive minor updates in the future, too.

TLS updates for the JVM agent

2015-10-5

It was quite calm around Jolokia this summer and not much happened in Jolokia-land. Not many bugs arrived, too, which I take as a good sign :)

Now let's start a next round with some revamped TLS support for https connections. Version 1.3.2 introduces a handful of new options for advanced configuration of the JVM agent's TLS connector:

In addition to the keystore (option keystore) the CA and the server cert as well as the server cert's key can be provided as PEM files with the options caCert, serverCert and serverKey, respectively.

Client cert validation has also be enhanced. In addition to validating the CA signature of a client cert, one can now also check that the *extended key usage* block of the cert was created for client usage (option extendedClientCheck). Also, one or more principals can be configured with clientPrincipal which are also compared againt the subject within a client certificate.

For simple use cases where no server validation is required, Jolokia is now able to create self-signed server certificates on the fly. This happens if neither a keystore nor a server PEM cert is provided. So, the easiest way to enable https is simply to add protocol=https. Of course, the client needs to disable cert validation then and it is recommended to use basic-authentication to authenticate the connection.

The changes affect the [JVM agent](https://jolokia.org/agent/jvm.html) only and are explained in the [reference manual](https://jolokia.org/reference/html/agents.html#agents-jvm).

That's it for now mostly, but see the [changelog](https://jolokia.org/changes-report.html) for some other minor additions. Progress on Jolokia 2.0 continues slowly, won't tell much here until I have a M1 release. No promises either :)

Delegating Authentication with Jolokia 1.3.1

2015-05-28

This minor release introduces one single new feature: A delegating authentication provider for the JVM agent. This can be switched on with configuration options and allow to delegate the authentication decision to an external service so that an easy SSO e.g. via OAuth2 is possible.

For example, if you are an OpenShift user and want to participate in OpenShift's OAuth2 SSO, then you can specify the following startup parameters, assuming that you OpenShift API server is running as openshift:8443:

java -javaagent:jolokia.jar=\  
                authMode=delegate,\  
                authUrl=https://openshift:8443/osapi/v1beta3/users/~,\  
                authPrincipalSpec=json:metadata/name,\  
                authIgnoreCerts=true\  
                ...

More about this can be found in the [reference manual](https://jolokia.org/reference/html/agents.html#agents-jvm). Note, that the parameterauthenticationClass has been renamed to authClass for consistencies sake. Please raise an[issue](https://github.com/rhuss/jolokia/issues/new) if this doesn't work for you.

Jolokia 1.3.0

2015-05-07

After quite some winter sleep Jolokia is back with a fresh release. This is mostly a [bug fix release](https://jolokia.org/changes-report.html) with some new features:

* A simple MBeanPlugin hook for registering own MBeans with the agent
* Support for OSGi's ConfigAdmin Service
* New possibility to hook into the deserialization process for responses in the Java client
* Proxy can be specified for the Java client
* Constructor based deserialization of Strings
* Support for Mule 3.6.1

There is one important change in the default behaviour of the WAR agent: Up to 1.2.3 Jolokia truncates any collection in the response value at a threshold of 1000 elements by default. This limit can be overwritten permanently in the configuration or per request as query parameter (maxCollectionSize). However, it turned out that this limit was [not large enough](https://github.com/hawtio/hawtio/issues/1725). So the new default behaviour is to have **no limit** at all. As said, if you need it you always can set a hard limit in the agent's configuration.

But the biggest news is probably something complete different: I'm super happy to announce that I (roland) joined Red Hat since May, where I will able to continue to work on Jolokia with an even higher intensity. Before looking into the future, acknowledgements go to my former employer [ConSol](http://www.consol.com/). Without the support donated by ConSol Jolokia would probably never has been grown from the original personal pet project to a full featured, production ready JMX remote access solution as it is today. Thank you !

What are the next steps ? [Jolokia](http://www.slideshare.net/roland.huss/jolokia-devoxx2014/11) [2.0](https://github.com/rhuss/jolokia/wiki) (code name: "Duke Nukem Forever") is not so far away, all changes from 1.x has been already merged up to the 2.0 branch. A release candidate should be available soon, however I can't give any estimates yet. But what I can say: Jolokia is alive and kicking more than ever!

Autumn edition 1.2.3

2014-11-08

Meh, that was a busy summer. Apologies for the delay and breaking the usual one-release-per-month cycle.

Nevertheless there are some nice goodies in this release:

* SSL handling of the JVM agent has been fixed and improved. Authentication with client certificates works now and you have much more influence of the SSL setup. Kudos to[Neven Radovanović](https://github.com/nevenr) for providing a patch.
* The Mule agent has been updated to support Mule 3.5. Thanks to [Fei Wong Reed](https://github.com/FeiWongReed) for the pull request.
* The configuration option "policyLocation" has now system property and environment expansions.
* Quite a bunch of bugs has been fixed. Please refer to the [changes report](https://jolokia.org/changes-report.html) for all changes.

If you want to get a quick introduction into Jolokia and a peek preview to Jolokia 2.0 come to my "Tools in Action" session at [Devoxx](http://cfp.devoxx.be/2014/talk/LVC-7834/Spicing_up_JMX_with_Jolokia) 2014 in Antwerp.

Last announcement for now: I started a blog at [https://ro14nd.de](https://ro14nd.de/) about various technical topics like Jolokia, Docker or other stuff.

Knock, knock: Let's welcome 1.2.2

2014-06-14

Let's welcome Jolokia's next minor release which is not *so* minor as it might seems.

* Custom authenticator support for the Java client. The standard authenticator allows preemptive authentication now as well.
* Support for "\*" wildcard in paths. See below.
* Finally an update to json-simple-1.1.1 which is mavenized, but still has its issues and not much traction to fix it. No problem we have a good workaround and it is still rock solid.
* Bug fixes. Yep.

The biggest new feature with the most impact is path wildcard support. You probably know[pattern read requests](https://jolokia.org/reference/html/protocol.html#read) which allow for fetching multiple patterns by using patterns for MBean names and attributes (not to be confused with [bulk requests](https://jolokia.org/reference/html/protocol.html#post-request)). When using pattern read requests, the value in the returned JSON structure is not a single return value for an attribute but a more complex structure containing the full MBean names and attributes which are matched by the pattern. Of course, it is not easy to use a path to navigate on this structure, the path has to know the full MBean name (well, why using a pattern then ?). That's the main reason why path access was not supported for pattern read requests up to release 1.2.1

Starting with 1.2.2 it is possible to use "\*" wildcards in patterns, which match a complete 'level' in the JSON object. This makes it easy to fetch all same-named attributes on arbitrary MBeans and extract only parts of their values. In fact, it is not so easy explain wildcard pathes, but here is a try (another try can be found in the [reference manual](https://jolokia.org/reference/html/protocol.html#read)):

* If using a literal path, then everything works as expected: The value the path points to is returned. Mostly this is a scalar value because that is what paths was introduced for.
* If the path contains a single "\*" as a part, then when coming to *this level* everything is included. A path containing a wildcard cannot be a scalar anymore, but is a JSON object or array. The remaining path parts are included as described above to each element at this level.
* A path can contain multiple wildcards, but wildcards can be used only on its own. If a "\*" is used as part of a path part (like 'current\*'), it's taken literally (which most of the time doesn't make much sense). This might change in the future.
* The net effect is, that literal path parts are "squeezed" (i.e. removed) in the resulting answer, whereas wildcard parts stay as extra levels.

You see, wildcard path handling is somewhat complex. For pattern read request they make quite some sense, for all other requests, I couldn't find good use cases yet. Please open an issue if any suspicious behaviour during path-wildcard using occurs.

Finally, I would also like to mention a new GitHub project [jolokia-extra](https://github.com/rhuss/jolokia-extra) which holds additional goodies. One design goal of Jolokia is to keep it focused. That's not so easy as there are tons of ideas out there, all backed by a particular use case. And they all want to get into the game. Beside that someone has to implement that (hint: still looking for contributions ;-), I opened a new playground for all that stuff which might not be of general interest but are still pearls. That's what jolokia-extra is for.

The beginning makes a 1.5 year old [pull request](https://github.com/rhuss/jolokia/pull/50) from [Marcin Płonka](https://github.com/mplonka) (Thanks a lot and sorry for the long, long delay, BTW). It's all about simplifying access to JSR-77 enabled JEE-Servers. You should know that [JSR 77: J2EE Management](https://jcp.org/en/jsr/detail?id=77) was a cool attempt to standardize naming and JMX exposed metrics for JEE. Unfortunately it was abandoned, but still lives in quite a bunch of JEE servers. Not at its full beauty, but still valuable enough to be supported. Astonishingly, WebSphere, even the latest 8.5 versions, has the best support for it. Using JSR-77 conform MBeans with plain Jolokia returns unnecessarily complex JSON structures which are hard to parse and understand. jolokia-extra adds a set of simplifier for make the usage with JSR-77 simpler (but add an extra of 50k to the agent). I recommend to have a look at it, especially if you are working with WebSphere.

In the future, it might be the case, that some lesser used additions (Spring and Spring Roo integration, JBoss Forge support, ...) will go into jolokia-extra as well.

Enough blubber, enjoy this release. And just in case, if anybody is wondering about 2.0 (BTW, is there **anyone** out there carrying about this next generation JMX transcended super-hero ?), just drop a note with twitter ([@jolokia\_jmx](https://twitter.com/jolokia_jmx)) or mail ([2.0@jolokia.org](mailto:2.0@jolokia.org)).

1.2.1 is in the house

2014-04-29

This minor release fixes some bugs and brings some [smaller features](https://jolokia.org/changes-report.html):

* An ActiveMQ server detector has been added
* The Java client library has been updated to the latest Apache HTTP components 4.x. If you are forced to still use Apache HTTP Client 3.x, you still can use the Java Client Lib from Jolokia 1.2.0 which will work with a Jolokia agent 1.2.1 nicely.
* Bug fix for JBoss 4.2.3 (yeah, seems still to be used)
* Cleaned up logging for discovery requests
* Placeholders can be used when specifying the agent URL which will be used in discovery responses. That way you can configure the URL flexibly from you server configuration.

And finally there is an important addition to the configuration of Jolokia's access policy. You might know, that you can configure CORS so the agent allows access only from certain origins. CORS is used by browsers for cross origin sharing and is a pure client side check. I.e. the browser asks the server and if the server says "no" the browser forbids any Ajax request to this server from any script. However, this still allows non-Ajax requests from any origin. To restrict this, too, a new configuration directive <strict-checking> has been added to the <cors>section which, if given, will do also a server-side check of a Origin: header when provided by the browser. If a security policy is used, it is highly recommended to set this flag (which for compatibility reason is switched off by default). And yes, it is of course highly recommended to use a [jolokia-access.xml](https://jolokia.org/reference/html/security.html#security-policy) policy in production (and not only for servers exposed to the bad internet directly). This is especially important if you can access Jolokia agents directly via a browser which is also used for internet access (hint: [CSRF](http://en.wikipedia.org/wiki/Cross-site_request_forgery)).

No news about 2.0 ? Yes, indeed. The giant is still sleeping, "Jolokia forever", you know. But the pressure rises, for some conferences I have some CFPs out which hopefully will lead to some nice CDD sessions ("conference driven development", yeah).

Find your agents with 1.2.0

2014-02-24

New year, new release. Ok, it's not the BIG 2.0 which I already somewhat promised. Anyways, another big feature jumped on the 1.x train in the last minute. It is now possible to find agents in your network by sending an UDP packet to the multicast group 239.192.48.84, port 24884. Agents having this discovery mechanism enabled will respond with their meta data including the access URL. This is especially useful for clients who want to provide access to agents without much configuration. I.e. the excellent [hawt.io](http://hawt.io/) will probably use it one way or the other. In fact, it was hawt.io which put me [on track](https://github.com/rhuss/jolokia/pull/126) for this nice little feature ;-)

Discovery is enabled by default for the [JVM agent](https://jolokia.org/reference/html/agents.html#agents-jvm), but not for the [WAR](https://jolokia.org/reference/html/agents.html#agents-war) agent. It can be easily enabled for the WAR agent by using servlet init parameters, system properties or environment variables. All the nifty details can be found in the [reference manual](https://jolokia.org/reference/html/index.html).

The protocol for the discovery mechanism is also [specified](https://jolokia.org/reference/html/protocol.html#discovery) in the reference manual. One of the first clients supporting this discovery mode is [Jmx4Perl](http://search.cpan.org/~roland/jmx4perl/lib/JMX/Jmx4Perl.pm) in its newest version. The Jolokia Java client will follow in one of the next minor releases.

But you don't need client support for multicast requests if you know already the URL for one agent. Each agent registers a MBean jolokia:type=Discovery which perform the multicast discovery request for you if you trigger the operation lookupAgents. The returned value contains the agent information and is described [here](https://jolokia.org/reference/html/mbeans.html#mbean-discovery).

This feature has been tested in various environments, but since low level networking can be, well, "painful", I would ask you to open an [issue](https://github.com/rhuss/jolokia/issues) in case of any problems.

Although it has been quiet some time with respect to the shiny new Jolokia 2.0, I'm quite close to a first milestone. All planned features has been implemented in an initial version, what's missing is to finish the heavy refactoring and modularisation of the Jolokia core. More on this later, please stay tuned ...

Tiny 1.1.5

2013-11-08

This is by far the smallest release ever: A [single char](https://github.com/rhuss/jolokia/commit/98e62c0478b166d188e0c9f173b278a59eed1fc8#diff-cdf118c56620542b85bc48650eb8ddb6R162) has been added on top of 1.1.4 fixing a silly bug when using Glassfish with the AMX system. So, no need to update if you are not using Glassfish.

Next week is Devoxx time and as last year (and the years before) you have the change to meet [me](https://twitter.com/ro14nd) in Antwerp. Ping me or look for the guy with the [Jolokia hoodie](http://jolokia.spreadshirt.de/men-s-hooded-jacket-A17809556/) ;-)

Step by step ... 1.1.4

2013-09-27

Some bug fixes and two new features has been included for the [autumn release](https://jolokia.org/changes-report.html):

A new configuration parameter "authenticatorClass" can be used for the JVM agent to specify an alternate authentication handler in addition to the default one (which simply checks for user and password).

With the configuration parameter "logHandlerClass" an alternative log handler can be specified. This can be used for the WAR and JVM agent in order to tweak Jolokia's logging behaviour. For the OSGi agent you already could use a LogService for customizing logging.

That's it and I hope you enjoy this release. I know, I'm late with 2.0, but as things happens, I have too much to do in 'real life' (i.e. feeding my family ;-). But I still hope to get it out this year, and yes, the 2.0 branch is growing (slowly).

BTW, the slides to my talk for the small but very fine [JayDay 2013](http://www.jayday.de/) are [online](http://www.jolokia.org/jayday-2013), too. These are "implemented" in JavaScript including live demos, where the JavaScript can be directly inserted in the browser (tested with Chrome & Firefox). For the sample code, simply push the blue buttons at the bottom of a [demo slide](http://www.jolokia.org/jayday-2013/#/step-20).

Small fixes with 1.1.3

2013-07-30

No big news in Jolokia land, but some bug fixes come with [1.1.3](https://jolokia.org/changes-report.html). Especially some issues with the JavaScript client's basic authentication and cross origin requests has been fixed. Otherwise I'm busy with 2.0 (and tons of other stuff ...). You can have a sneak preview of Jolokia 2.0 on this [branch](https://github.com/rhuss/jolokia/tree/2.0) including basic notification support and quite some refactoring with respect to the service architecture.

So please stay tuned ....

Stopover on the road to 2.0: Jolokia 1.1.2 released

2013-05-28

In order to ease waiting for 2.0, Jolokia version 1.1.2 has been released. It contains some minor bug fixes as explained in the [changelog](https://jolokia.org/changes-report.html). Depending on the bug reports and pull request dropping in there might be even a 1.1.3 release before 2.0 will be finished.

In the meantime, you can also see Jolokia live at [JayDay](http://www.jayday.de/) where I will give a talk about Jolokia's JavaScript support. The forthcoming JMX notification support will presented, too. It is also a good chance to have a cold bavarian beer with me ;-)

Some small goodies served by 1.1.1

2013-03-27

This last feature release before work on 2.0.0 starts brings some small goodies.

* BigDecimal and BigInteger can now be used for operation arguments and return values.
* A new processing parameter [ifModifiedSince](https://jolokia.org/reference/html/protocol.html#processing-parameters) has been introduced. This parameter can be used with a timestamp for fetching the list of available MBeans only when there has been some changes in the MBean registration on any observed MBeanServer since that time. If there has been no changes an answer with status code "302" (Not modified) is returned. This feature is also supported for "search" requests. In a future version of Jolokia, there will be also custom support for own "read" and "exec" request so that expensive operations can be called conditionally.
* For the JVM agent, if a port of 0 is given, then an arbitrary free port will be selected and printed out on standard output as part of the Jolokia agent URL. If no host is given, the JVM agent will now bind to localhost and if host of "0.0.0.0" or "\*" is provided, the agent will bind on all interfaces.
* For the Java client an extra property errorValue has been added which holds the JSON serialized exception if the processiong parameter serializeException is active.
* The JavaScript client's [jolokia.register()](https://jolokia.org/reference/html/clients.html#js-poller) can now take an optional config element for specifying processing parameters for a certain scheduler job. Also, the new optiononlyIfModified can be used so that the callback for list and search request is only called, if the set of registered MBean has changed. This is especially useful for web based client which want to refresh the MBean tree only if there are changes.
* The Expires: header of a Jolokia response has now a valid date as value (instead of '-1') which points to one hour in the past. This change should help clients which do not ignore according to RFC-2616 invalid date syntax and treat them as 'already expired'.

Links to the corresponding GitHub issues and the bugs fixed in this release can be found in the [change report](https://jolokia.org/changes-report.html).

This is the last feature release in the 1.x series. Work has already started on exciting new features for Jolokia 2.0. E.g. JMX notification support is coming, an initial pull model has been already implemented (on branch [notification](https://github.com/rhuss/jolokia/tree/notification)). There are even more ideas and some refactorings will happening along with some modest changes in the module structure. So, please stay tuned ...

1.1.0 with Spring support and @JsonMBean

2013-02-26

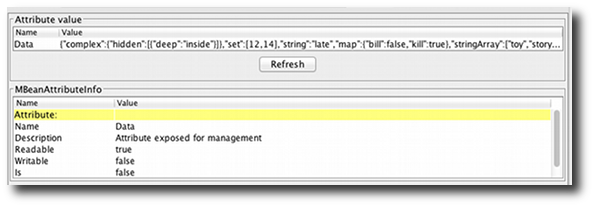
It took some time, but it was worth it. Along with the usual bug fix parade, several new features has been added to Jolokia.

A new module jolokia-spring has been added which makes integration of Jolokia in Spring applications even easier. Simply add the following line (along with the corresponding namespace) to you application context and agent will be fired up during startup:

<jolokia:agent>  
   <jolokia:config  
           autoStart="true"  
           host="0.0.0.0"  
           port="8778"  
   ....  
   />  
</jolokia:agent>

More details can be found here in the [reference manual](https://jolokia.org/reference/html/jmx.html#jvm-spring).

The new jolokia-jmx module provides an own MBeanServer which never gets exposed via JSR-160 remoting. By registering your MBeans at the Jolokia MBeanServer you can make them exclusively available for Jolokia without worrying about JSR-160 access e.g. via jconsole. However, if you annotate your MBeans with @JsonMBean and register it at the Jolokia MBeanServer your get automatic translation of complex data types to JSON even for JSR-160 connections:



The details can be found [here](https://jolokia.org/reference/html/jmx.html).

Several new processing options enter the scene. These can be given either as global configuration parameters or as query parameters:

* **canonicalNaming** influences the order of key properties in object names
* **serializeExceptions** adds a JSON representation of exceptions in an error case
* **includeStackTrace** can switch on/off the sending of an lengthy stack trace in an error case

That's it for now, all changes are summarized as always in the [change report](https://jolokia.org/changes-report.html).

Some other, more organizational stuff for now:

* Bugtracking and feature requests switch over completely to [Github](https://github.com/rhuss/jolokia/issues). Since I'm currently collecting features for 2.0, it's a good time for feature requests ;-). All ideas entered at[jolokia.idea.informer.com](http://jolokia.idea.informer.com/) has been transformed into Github issues.
* If you are close to Germany it might be of interest to you, that I'm giving a training on Jolokia and Jmx4Perl, with focus on Java Monitoring with Nagios. This will happen at 16./17.04.2013 in Munich, details can be found on our [web site](http://www.consol.de/open-source-monitoring/schulungen/java-monitoring-mit-nagios/) (in german).

And finally a very **hot** recommendation: Please have a look at [hawt.io](http://hawt.io/) a super cool HTML5 console which uses Jolokia for backend communication exclusively. Most of the new ideas included in this Jolokia release were inspired by discussions with James Strachan, one of the driving forces behind hawt. Thanks for that ;-)

1.0.6 cosmetics

2012-11-23

Although it has been quite calm in Jolokia land for some months, there is quite some momentum around Jolokia. This minor release brings some cosmetic changes, mostly for tuning the ordering within MBeans names and some JavaScript fixes. More on this in the[changelog](https://jolokia.org/changes-report.html).

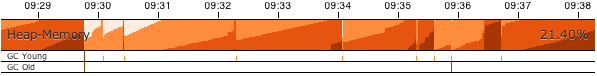
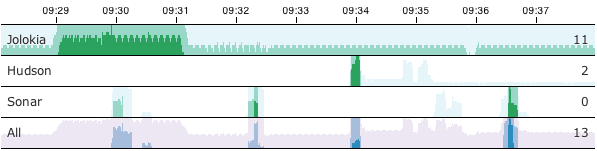
Some other tidbits:

* The new [Talks and Screencast](https://jolokia.org/talks.html) section collects some fancy multimedia introducing Jolokia
* I'm going to talk about Jolokia at [jayday 2012](http://www.jayday.de/?lang=en), a brand new, low cost conference in Munich on 3th December 2012. Hopefully there will be some brand new stuff to show, too.
* Some completely irrelevant stuff: Jolokia T-Shirts can be found in the [Jolokia Shop](http://jolokia.spreadshirt.de/) The shop was too easy to setup for not doing it ;-) And they look freaking hot ....

Cubism support in 1.0.5

2012-07-22

Jolokia 1.0.5 has been released. Beside minor [improvements and bug fixes](https://jolokia.org/changes-report.html), one great new feature has been introduced: As already mentioned Jolokia has now support for [Cubism](http://square.github.com/cubism/), a fine time series charting library based on [d3.js](http://d3js.org/). Cubism provides support for an innovative charting type, the [horizon charts](http://vis.berkeley.edu/papers/horizon/):

A very cool live demo where a Jolokia JavaScript client fetches live data from our servers and plot it with Cubism can be found on this [demo page](https://jolokia.org/client/javascript-cubism.html). The documentation can be found in the[reference manual](https://jolokia.org/reference/html/clients.html#js-cubism).

Jolokia uses also a [Travis](http://travis-ci.org/#!/rhuss/jolokia) build in addition to our own [CI Server](http://labs.consol.de/jenkins). (Did I mentioned already, that we have a quite I high [Sonar](http://labs.consol.de/sonar/dashboard/index/org.jolokia:jolokia) score ?). Travis is a quite nice supplement to Github, and brings CI testing to a higher level.

That's it for now. The next months of my open-source work will be spent now on [Ají](https://github.com/rhuss/aji), Jolokia's new fancy sister. Sorry for pushing thinks like notifications down the Jonlokia back-log, but it's not forgotten.

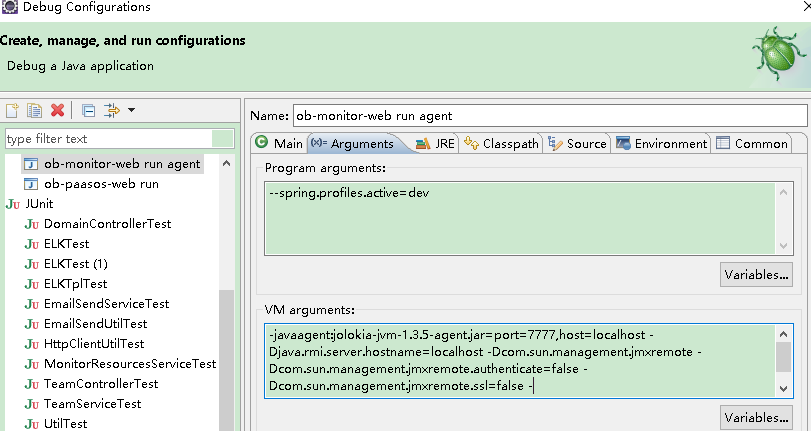
# Roland Huß

https://ro14nd.de/

# Eclipse配置

--spring.profiles.active=dev

-javaagent:jolokia-jvm-1.3.5-agent.jar=port=7777,host=localhost -Djava.rmi.server.hostname=localhost -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.port=10001



# github

logzio/jmx2graphite: JMX to Graphite every x seconds in one command line (Docker based) https://github.com/logzio/jmx2graphite

Jolokia - JMX on Capsaicin https://github.com/jolokia-org

https://github.com/rhuss/jolokia

jolokia-org/jolokia-client-javascript: Jolokia JavaScript Client https://github.com/jolokia-org/jolokia-client-javascript

square/cubism: Cubism.js: A JavaScript library for time series visualization. https://github.com/square/cubism

Cubism.js http://square.github.io/cubism/

## logzio/jmx2graphite:

JMX to Graphite every x seconds in one command line (Docker based)

https://github.com/logzio/jmx2graphite

jmx2graphite

jmx2graphite is a one liner tool for polling JMX and writes into Graphite (every 30 seconds by default). You install & run it on every machine you want to poll its JMX.

Currently it has two flavors:

1. Docker image which reads JMX from a jolokia agent running on a JVM, since exposing JMX is the simplest and easiest through Jolokia agent (1 liner - see below).
2. Run as a java agent, and get metrics directly from MBean Platform

The metrics reported have the following names template:

[service-name].[service-host].[metric-name]

* service-name is a parameter you supply when you run jmx2graphite. For example "LogReceiever", or "FooBackend"
* service-host is a parameter you supply when you run jmx2graphite. If not supplied it's the hostname of the jolokia URL. For example: "172\_1\_1\_2" or "log-receiver-1\_foo\_com"
* metric-name the name of the metric taken when polling Jolokia. For example: java\_lang.type\_Memory.HeapMemoryUsage.used

How to run?

Using Docker (preferred)

If you don't have docker, install it first - instructions [here](http://docs.docker.com/engine/installation/).

docker run -i -t -d --name jmx2graphite \

-e "JOLOKIA\_URL=http://172.1.1.2:11001/jolokia/" \

-e "SERVICE\_NAME=MyApp" \

-e "GRAPHITE\_HOST=graphite.foo.com" \

-e "GRAPHITE\_PROTOCOL=pickled" \

-v /var/log/jmx2graphite:/var/log/jmx2graphite \

--rm=true

logzio/jmx2graphite

Environment variables

* JOLOKIA\_URL: The full URL to jolokia on the JVM you want to sample. When jolokia (and the java app) is running inside a docker container there are two ways to specify the host in the jolokia URL so this URL will be reachable by jmx2graphite which also runs inside a docker instance:
  + The easy one: On the docker running your java app and Jolokia, makes sure to expose the jolokia port (using -v), and then use the host IP of the machine running the dockers.
  + Container linking: You can use a hostname you invent like "myapp.com", and then when running jmx2graphite using Docker, add the option: --link myservice-docker-name:myapp.com". So if your app is running in docker named "crazy\_service" then you would write jolokia URL as "<http://myapp.com:8778/jolokia>", and when running jmx2graphite using docker add the option "--link crazy\_service:myapp.com". What this does is add mapping between the host name myapp.com to the internal IP of the docker running your service to the /etc/hosts file.
* SERVICE\_NAME: The name of the service (it's role).
* GRAPHITE\_HOST: The hostname/IP of graphite
* GRAPHITE\_PROTOCOL: Protocol for graphite communication. Possible values: udp, tcp, pickled

Rest of command

* -v /var/log/jmx2graphite:/var/log/jmx2graphite: jmx2graphite by defaults writes its log (using Logback) to/var/log/jmx2graphite. This argument maps this directory to the host directory so you can easily view the logs from the place you run the docker command
* --rm=true: removes the docker image created upon using docker run command, so you can just call docker runcommand again.

Optional environment variables

* GRAPHITE\_PORT: Protocol port of graphite. Defaults to 2004.
* SERVICE\_HOST: By default the host is taken from Jolokia URL and serves as the service host, unless you use this variable.
* INTERVAL\_IN\_SEC: By default 30 seconds unless you use this variable.

Using bash

1. Clone the repository git clone https://github.com/logzio/jmx2graphite
2. cd jmx2graphite
3. Build it: ./gradlew build
4. Copy the tar/zip file created from build/distributions/\*.tar or \*.zip to the host computer you wish to run this on
5. Unzip it in any directory you'd like
6. Move the directory jmx2graphite from opt/jmxgraphite to a location which fits you. Normally you would move it to/opt
7. Edit the configuration file at jmx2graphite/conf/application.conf: The mandatory items are:
   1. service/jolokiaFullUrl - Fill in the full URL to the JVM running Jolokia (It exposes your JMX as a REST service, normally under port 8778).
   2. service/name - The role name of the service.
   3. graphite/hostname - Graphite host name the metrics will be sent to
8. cd jmx2graphite/bin
9. run ./jmx2graphite. This runs interactively, so pressing ctrl-c will make it stop.
10. If you wish to run this as a service you need to create a service wrapper for it. Any pull requests for making it are welcome! If it's possible running it as docker making it simpler.

As Java Agent

This lib can also get the metrics from MBean Platform instead of jolokia. In order to do so, we need to run inside the JVM.

* First, get the java agent jar from the releases page
* Modify your app JVM arguments and add the following: java -javaagent:/path/to/jmx2graphite-1.1.0-javaagent.jar=GRAPHITE\_HOSTNAME=graphite.host;SERVICE\_NAME=Myservice ...
* The parameters are key-value pairs, in the format of key=value;key=value;...
* The parameters names and functions are exactly as described in Environment Variables section. (Except no need to specify JOLOKIA\_URL of course)
* The javaagent.jar is an "Uber-Jar" that shades all of its dependencies inside, to prevent class collisions
* For example: java -javaagent:/opt/jmx2graphite-1.1.0-javaagent.jar=GRAPHITE\_HOSTNAME=graphite.example.com;SERVICE\_NAME=PROD.MyAwesomeCategory example.jar

How to expose JMX Metrics using Jolokia Agent

1. Download Jolokia JVM Agent JAR file [here](http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-jvm/1.3.2/jolokia-jvm-1.3.2-agent.jar).
2. Add the following command line option to your line running java:

-javaagent:path-to-jolokia-jar-file.jar

For example:

-javaagent:/opt/jolokia/jolokia-jvm-1.3.2-agent.jar

By default it exposes an HTTP REST interface on port 8778. See [here](https://jolokia.org/reference/html/agents.html#jvm-agent) if you want to change it and configure it more. We run all of ours apps using Docker, so to avoid clashes when we map the 8778 port to a unique external port belonging only to this application.

Installing and Configuring Graphite

If you never installed Graphite, this small guide below might be a good place to start. I'm using Docker since it's very easy to install this way.

Installing Graphite

We will install Graphite using a great docker image by [hopsoft](https://github.com/hopsoft/docker-graphite-statsd). I tried several and it was by far the easiest to work with.

1. Run the following to get basic Graphite up and running
2. docker run -d \
3. --name graphite \
4. -p 80:80 \
5. -p 2003:2003 \
6. -p 2004:2004 \
7. -p 8125:8125/udp \
8. -p 8126:8126
9. Now, let's copy out all of its existing configuration files so it will be easy to modify. I will assume you will place it at/home/ubuntu
10. cd /home/ubuntu
11. mkdir graphite
12. docker cp graphite:/opt/graphite/conf graphite
13. docker cp graphite:/opt/graphite/webapp/graphite graphite/webapp
14. Stop graphite by running docker stop graphite
15. Configuring Graphite: Now edit the following files:
16. /home/ubuntu/graphite/conf/carbon.conf:
    * MAX\_CREATES\_PER\_MINUTE: Make sure to place high values - for example 10000. The default of 50 means that the first time you run jmx2graphite, all of your metrics are reported at once. If you have more than 50, all other metrics will be dropped.
    * MAX\_UPDATES\_PER\_SECOND: I read a lot there should be a formula for calcualting the value for this field, but that is once you reach high I/O disk utilization. For now, simply place 10000 value there. Otherwise you will get a 1-2 minute lag from the moment jmx2graphite pushes the metric to Graphite until it is viewable in Graphite dashboard
17. /home/ubuntu/graphite/conf/storage-schemas.conf:
    * in the default section (default\_1min\_for\_1day) make sure retentions is set to the same interval as you are using in jmx2graphite (30seconds by default). Here is an example
18. [default\_1min\_for\_1day]
19. pattern = .\*
20. retentions = 30s:24h,1min:7d,10min:1800d

If you have 10s:24h then when doing derivative, you will get null values for each missing 2 points in the 30sec window and the graph will be empty

1. Create some directories which normally are crearted by the docker image but since we're mounting /var/log to an empty directory of ours in the host, they don't exists:
2. mkdir -p /home/ubuntu/log/nginx
3. mkdir -p /home/ubuntu/log/carbon

mkdir -p /home/ubuntu/log/graphite

1. Run Graphite. I use the following short bash script run-graphite.sh:
2. #!/bin/bash
3. docker run -d \
4. --name graphite \
5. --rm=true \
6. --restart=always \
7. -p 80:80 \
8. -p 2003:2003 \
9. -p 2004:2004 \
10. -p 8125:8125/udp \
11. -p 8126:8126 \
12. -v /home/ubuntu/graphite/storage:/opt/graphite/storage \
13. -v /home/ubuntu/log:/var/log \
14. -v /home/ubuntu/graphite/conf:/opt/graphite/conf \
15. -v /home/ubuntu/graphite/webapp/graphite:/opt/graphite/webapp/graphite \

hopsoft/graphite-statsd

Configuring Graphite

If you have an existing Graphite installation see the section above "configuring Graphite: Now edit the following files:".

Motivation

I was looking for a tool I can just drop in place, have a 1-liner run command which will then run every 10 seconds, poll my JVM JMX entirely and dump it to Graphite. Of course I started Googling and saw the following:

* [JMXTrans](https://github.com/jmxtrans/jmxtrans) I had several issues which got me stomped:
  + You can't instruct it to sample all JMX metrics. Instead you have to specify exactly which MBeans which you want and also their attributes - this can be quite a long list. In order to compose this list you have to fire up JMX Console, find the bean you are interested at, extract its name and add several lines of config to your config file. Then you have to copy the attribute names you want from this mbean. Rinse and repeat for every bean. For me, I just wanted all, since when you benchmark a JVM you don't know where the problem is so you want to start with everything at hand. From my handful experience with JMX, polling all beans doesn't impact the running JVM. Graphite can be boasted with hardware if it will become the bottleneck. Essentially I would like to add blacklist/whitelist to jmx2graphite, but it should be straightforward wildcard expession and not regular expression based.
  + I had trouble understanding how to configure it polling several JVMs. It invovles writing a YAML file and then running a CLI for generating the configuration file for JMXTrans. Too complicated in my opinion.
* [jmxproxy](https://github.com/mk23/jmxproxy) It's an HTTP REST server allowing you to fetch mbeans from a given JVM using REST to it. You are supposed to have one per your cluster. Great work there. The biggest drawback here was that you have to specify a predefined list of mbeans to retrieve - I wanted it all - it's too much work to compose the list of mbeans for: Camel, Kafka, Zookeeper, your own, etc.
* [Sensu plugin](https://github.com/sensu/sensu-community-plugins/blob/master/plugins/http/http-json-graphite.rb) - Aside from the prequisite of Sensu, again you must supply a predefined list of beans.
* [Collectd plugin](https://collectd.org/wiki/index.php/Plugin:GenericJMX) - Must have collectd and also, same as before, specify a list of mbeans and their attributes in a quite complicated config file. This also requires installing another collectd plugin.
* [Fluentd JMX plugin](https://github.com/niyonmaruz/fluent-plugin-jmx) - Must have fluentd installed. Must specify list of mbeans and their attributes. Works against Jolokia only (same as jmx2graphite)

So after spending roughly 1.5 days fighting with those tools and not getting what I wanted, I sat down to write my own.

Why Docker?

Docker enables jmx2graphite to install and run in one command line! Just about any other solution will requires more steps for installation, and not to mention the development efforts.

Why Jolokia?

* When running JVM application inside docker it is sometime quite complex getting JMX to work, especially around ports.
* Composing JMX URI seems very complicated and not intuitive. Jolokia REST endpoint is straight forward.
* Can catch reading several MBeans into one call (not using this feature yet though)

Features Roadmap

* Add Integration Tests using Vagrant
* Add support for reading using JMX RMI protocol for those not using Jolokia.
* Support whiltelisting/blacklisting for metrics

Contributing

We welcome any contribution! You can help in the following way:

* Open an issue (Bug, Feature request, etc)
* Pull requests for any addition you can think of

Building and Deploying

Build

./gradlew build

docker build -t logzio/jmx2graphite .

Build Java Agent

./gradlew build javaAgent

Deploy

docker login

docker push logzio/jmx2graphite

Changelog

* v1.1.1
  + Added support for 2 additional protocols when sending metrics to Graphite: tcp, udp. This is in addition to the existing Pickle protocol (Contributed by: jdavisonc)
* v1.1.0
  + Major refactoring - jmx2graphite now comes in two flavors: standalone using docker as it was in 1.0.x, and as a Java Agent running alongside you app. This is useful if your app is running inside Docker on Mesos and coupling it with another container just to read its metrics contradicts the Mesos paradigm.
  + Added java agent capabilities, through MBeans Platform
  + Changed logback to log4j
* v1.0.8
  + First migration step to Kotlin language
* v1.0.7
  + Issue #2: Log file appender will show TRACE debug level as well
* v1.0.6
  + Fixes #4: logback will save history for 7 days
* v1.0.5
  + logback.xml now scan it self every 10 seconds instead of 30 to get that fast feedback loop
  + Added an XML element sample to logback.xml to trace the metrics that are sent to Graphite
* v1.0.4
  + logback.xml now scan it self every 30 seconds. Improved error message printed to the log
* v1.0.3
  + Wouldn't recover from Graphite server restart (failed on broken pipe for a long time)
* v1.0.2
  + MBean name properties (the part that is after the ':') retrieved from jolokia were sorted lexically by property name. This removed any creation order of those properties which actually represent a tree path, thus the fix is to maintain the creation order.
* v1.0.1
  + MBean name got its dots converted into \_ which results in flattening your beans too much. Now the dot is kept.

#License

See the LICENSE file for license rights and limitations (MIT).

# Docker

https://hub.docker.com/u/jolokia/

## flanneld

/usr/bin/flanneld -etcd-endpoints=http://192.168.0.180:4012 -etcd-prefix=/flannel/network -iface=eno16777984

# Jenkins

ssh -i ${JENKINS\_HOME}/.id\_rsa root@192.168.0.180 "docker rm -f jvmviewer"

ssh -i ${JENKINS\_HOME}/.id\_rsa root@192.168.0.180 "docker run -d --name jvmviewer --restart=always -p 7778:8080 docker.dev.yihecloud.com/base/jvmviewer:1.0\_$BUILD\_TIMESTAMP"

ssh -i ${JENKINS\_HOME}/.id\_rsa root@192.168.0.180 "/root/update\_jvmviewer.sh 1.0\_$BUILD\_TIMESTAMP"

docker rm -f jvmviewer

docker run -d --name jvmviewer \

--restart=always \

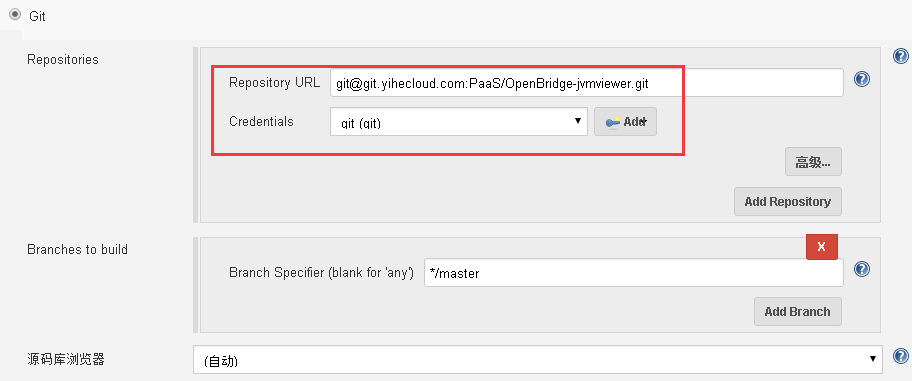
-p 7778:8080 \

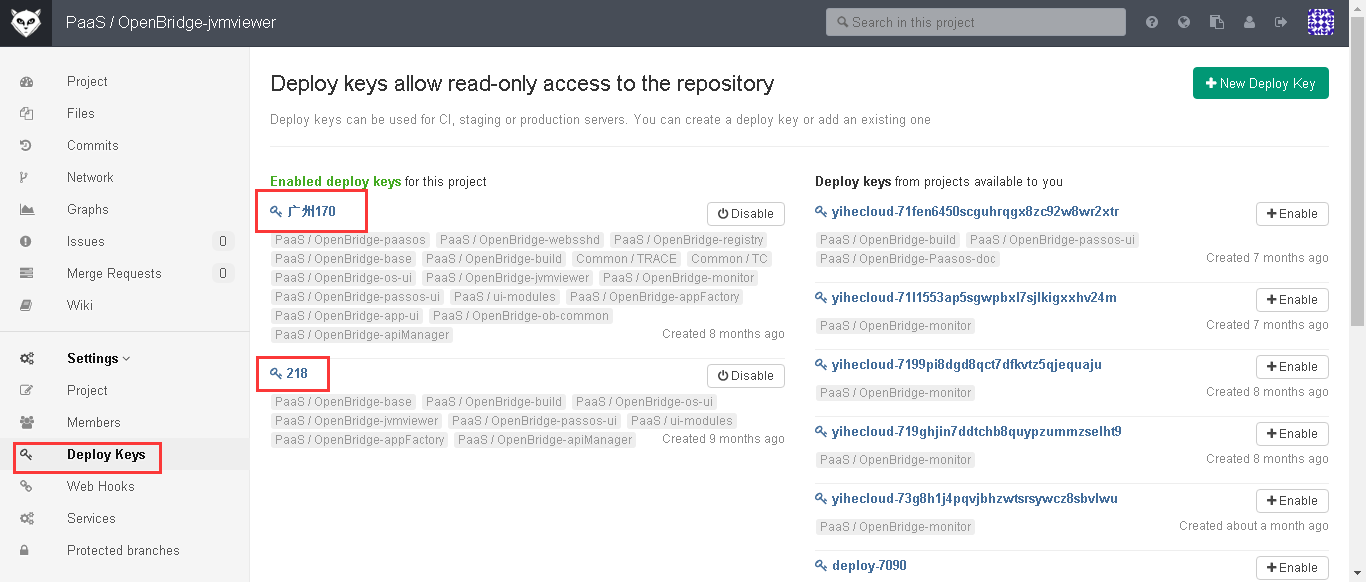
docker.dev.yihecloud.com/base/jvmviewer:$1

ssh -i /mnt/jenkins\_home/.id\_rsa -o "StrictHostKeyChecking=no" root@192.168.0.178

## git

git中设置key

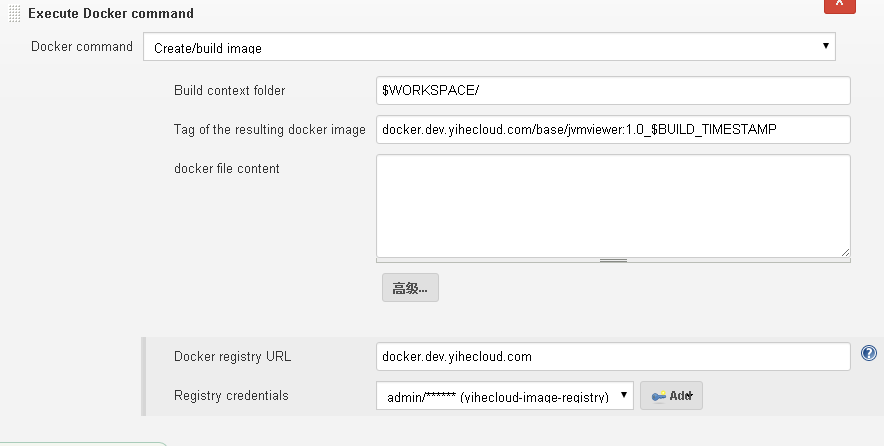


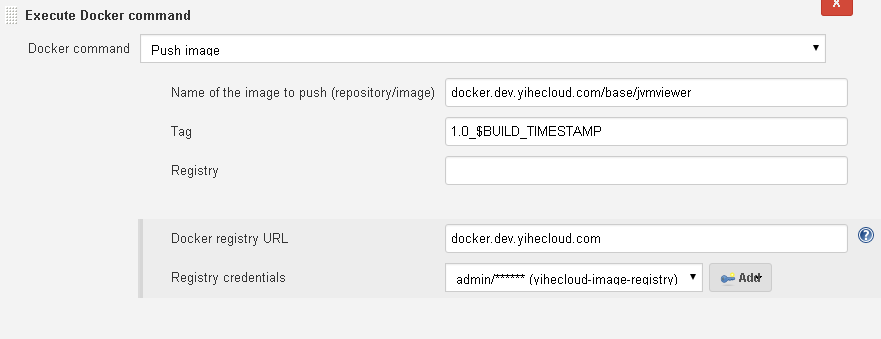


## docker

$WORKSPACE/

1.0\_$BUILD\_TIMESTAMP





# Proxy

Proxy模式仅在war包部署方式才能生效

WAR Agent

The WAR agent jolokia.war deploys as a regular web archive (WAR) in a JEE server. Deployment is simple (often only a copy in a certain directory) and the agent can be tuned like a normal web application. Setting it up the agent servlet for secure communication is a well known business (but specific to every application server) and the same as for any other web archive. The runtime behaviour like connection pooling or dedicated HTTP connector can be tuned very easily (for Tomcat, see this [example](http://labs.consol.de/blog/nagios/putting-jmx4perl-on-the-fast-lane-for-tomcat/) for setting up an extra HTTP connector for this agent).

Also, this is the agent for the [proxy mode](https://jolokia.org/features/proxy.html) where it is deployed in a simple, dedicated application server like Tomcat or Jetty.

The WAR agent has been tested to work on

* JBoss 4.2.3, 5.1.0, 6.1.0, 7.0.2, 7.1.1, 8.0.0
* Oracle WebLogic 9.2.3.0, 10.0.2.0, 10.3.6.0
* Glassfish 2.1.1, 3.0.1, 3.1.2, 4.0.0
* IBM Websphere 6.1.0.33, 7.0.0.11, 8.0.0.1, 8.5
* Apache Tomcat 5.5.35, 6.0.37, 7.0.52, 8.0.3
* Jetty 5.1.15, 6.1.26, 7.6.9, 8.1.9, 9.1.2
* Resin 3.1.9
* Jonas 4.10.7, 5.1.1, 5.2.1
* Apache Geronimo 2.1.6, 2.2.1, 3.0.0
* Spring dm Server 2.0.0.RELEASE
* Eclipse Virgo 2.1.0

This is the most widely used agent. Read more about the [WAR agent](https://jolokia.org/agent/war.html) and its installation.

# 测试例子

http://127.0.0.1:7777/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/used

#图形例子

http://localhost:8080/monitor/jolokia/plot.html

## 外网访问

#服务端

java -Djava.rmi.server.hostname=192.168.0.179 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.port=10001 -jar openbridge-monitor.jar

#jolokia客户端

java -jar jolokia-jvm-1.3.5-agent.jar start 49942 --host=192.168.0.179

## proxy模式

curl -XPOST 192.168.0.179:8888/jolokia -d '{

"type": "read",

"mbean": "java.lang:type=Memory",

"attribute": "HeapMemoryUsage",

"target": {

"url": "service:jmx:rmi:///jndi/rmi://192.168.0.179:10001/jmxrmi"

}

}'

curl -XPOST 127.0.0.1:7777/jolokia -d '{

"type": "read",

"mbean": "java.lang:type=Memory",

"attribute": "HeapMemoryUsage",

"target": {

"url":"service:jmx:rmi:///jndi/rmi://192.168.0.179:10001/jmxrmi"

}

}'

curl -XPOST 192.168.31.166:7777/jolokia/ -d '{

"type": "read",

"mbean": "java.lang:type=Memory",

"attribute": "HeapMemoryUsage",

"target": {"url": "service:jmx:rmi:///jndi/rmi://192.168.31.166:10001/jmxrmi"}

}'

## Java021-dev环境

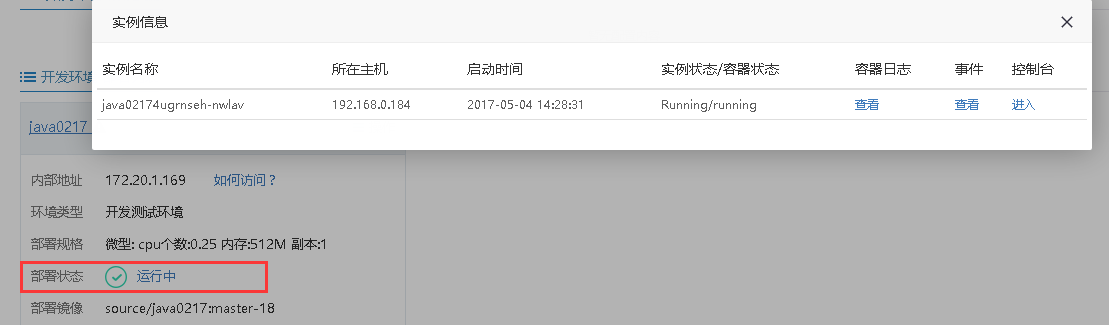
http://192.168.0.180:7778/jvmviewer/jolokia.html?ip=10.1.74.8&port=44446

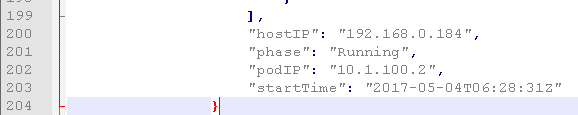
https://paas.dev.yihecloud.com/os/project/env/index?projectId=734h0gc5j48msmbfqpxrdth77bpa2yx&envId=734h38iehfe2bgkxdvheqyqk8eyckeb



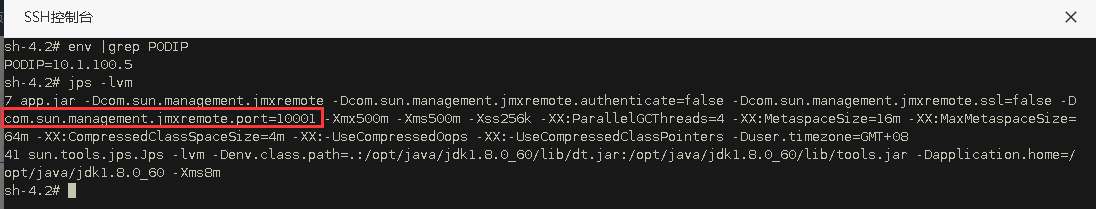
点击运行中的请求中，返回PODIP

https://paas.dev.yihecloud.com/os/project/deploy/getPods.do



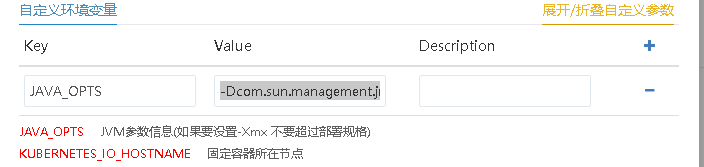


### 查看ip端口



## K8s中部署

新增环境变量



-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.port=10001 -Xmx500m -Xms500m

获取PODIP



yihecloud/startup.sh at master · testyihecloud/yihecloud

https://github.com/testyihecloud/yihecloud/blob/master/demo\_java/docker/startup.sh

将JAVA\_OPTS中的值传入到java -jar中去



根据PODIP和PORT获取

curl -XPOST 192.168.0.180:7777/jolokia/ -d '{

"type": "read",

"mbean": "java.lang:type=Memory",

"attribute": "HeapMemoryUsage",

"target": {

"url":"service:jmx:rmi:///jndi/rmi://10.1.100.2:10001/jmxrmi"

}

}'

file:///C:/Users/yuanZmy/Desktop/JVM%E7%9B%91%E6%8E%A7%EF%BC%88jolokia)/jolokia.html?ip=10.1.100.2&port=10001

docker run -d --name jolokia -p 7777:8080 docker.dev.yihecloud.com/bodsch/docker-jolokia:11

docker run --rm -it -v `pwd`:/data/webapps -p 7777:8080 docker.dev.yihecloud.com/base/tomcat:2.1

docker-jolokia/Dockerfile at master ·

bodsch/docker-jolokia https://github.com/bodsch/docker-jolokia/blob/master/Dockerfile

# jconsole

## 输入参数

127.0.0.1:7777/jolokia/read/java.lang:type=Runtime/InputArguments http://127.0.0.1:7777/jolokia/read/java.lang:type=Runtime/InputArguments

{

* **request**:

{

* + **mbean**: "java.lang:type=Runtime",
  + **attribute**: "InputArguments",
  + **type**: "read"

},

* **value**:

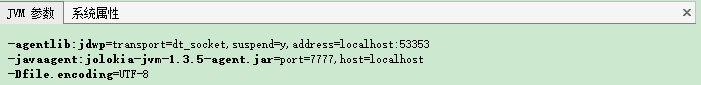
[

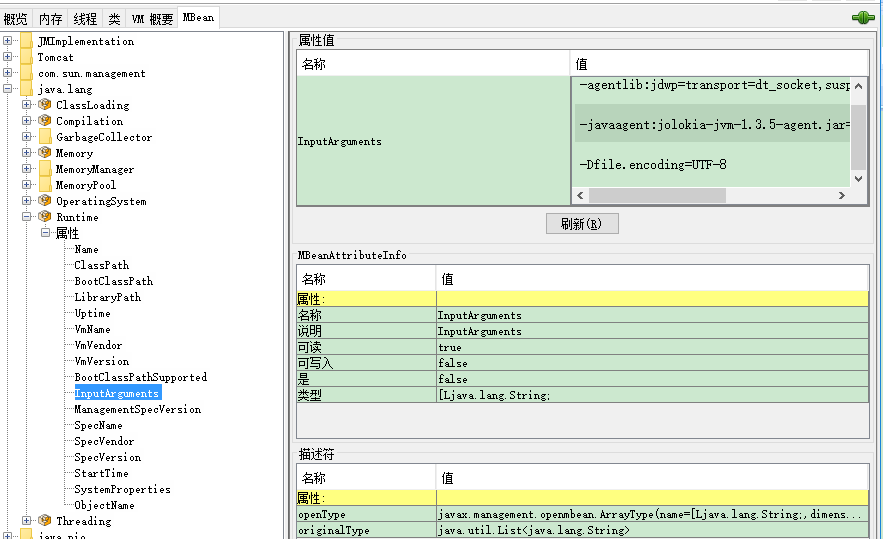
* + "-agentlib:jdwp=transport=dt\_socket,suspend=y,address=localhost:53353",
  + "-javaagent:jolokia-jvm-1.3.5-agent.jar=port=7777,host=localhost",
  + "-Dfile.encoding=UTF-8"

],

* **timestamp**: 1492394831,
* **status**: 200

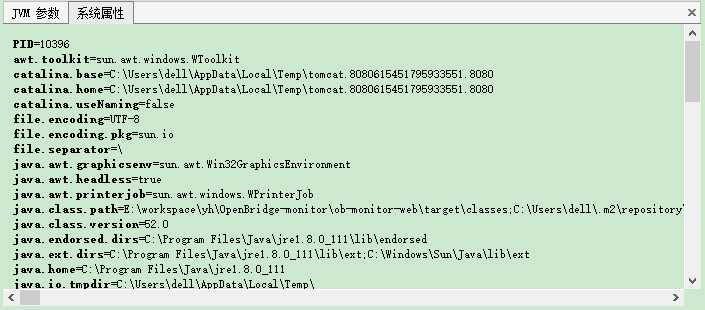
}





## 系统属性

http://127.0.0.1:7777/jolokia/read/java.lang:type=Runtime/SystemProperties



## 内存使用

http://127.0.0.1:7777/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/used

{

* **request**:

{

* + **mbean**: "java.lang:type=Memory",
  + **attribute**: "HeapMemoryUsage",
  + **type**: "read"

},

* **value**:

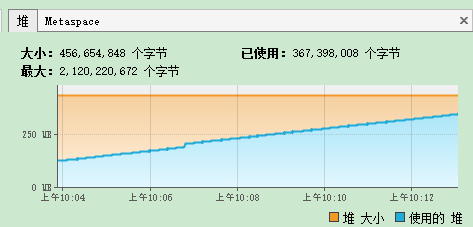
{

* + **init**: 134217728,
  + **committed**: 434110464,
  + **max**: 1884815360,
  + **used**: 383785024

},

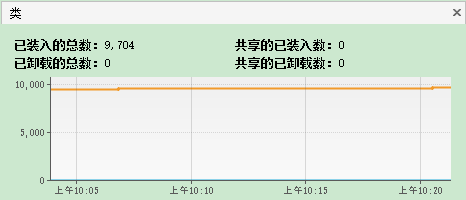
* **timestamp**: 1492395225,
* **status**: 200

}



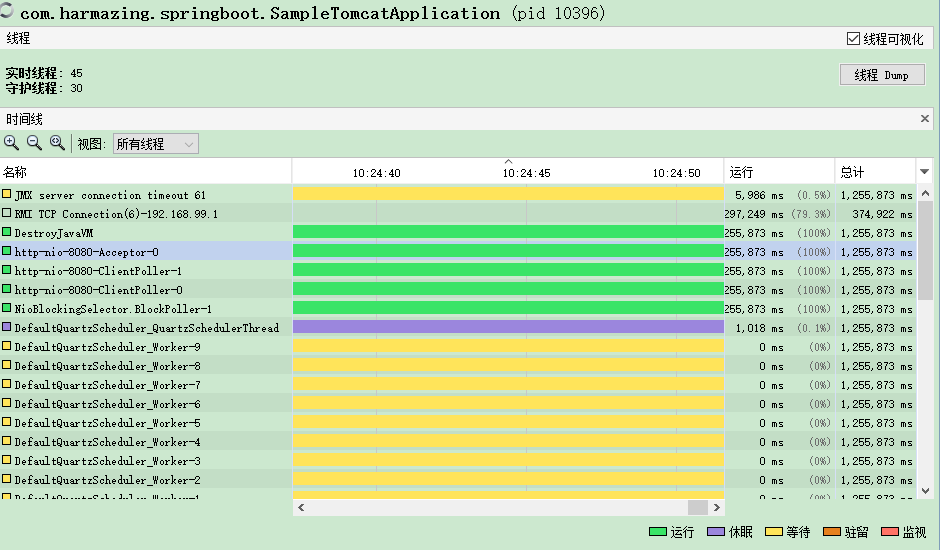
## 内加载

http://127.0.0.1:7777/jolokia/read/java.lang:type=ClassLoading



## 线程

http://127.0.0.1:7777/jolokia/read/java.lang:type=Threading



# java动态获取jvm参数 - liudezhicsdn的博客 - 博客频道 - CSDN.NET

http://blog.csdn.net/liudezhicsdn/article/details/51058504

JDK提供[**Java**](http://lib.csdn.net/base/javase).lang.management包， 其实就是基于JMX技术规范，提供一套完整的MBean，动态获取JVM的运行时数据，达到监控JVM性能的目的。   
java.lang.management包，是[**Java SE**](http://lib.csdn.net/base/javase) 5 中新引入的 JMX API。

package com.ldz.jvm;

import java.lang.management.CompilationMXBean;

import java.lang.management.GarbageCollectorMXBean;

import java.lang.management.ManagementFactory;

import java.lang.management.MemoryMXBean;

import java.lang.management.MemoryPoolMXBean;

import java.lang.management.MemoryUsage;

import java.lang.management.OperatingSystemMXBean;

import java.lang.management.RuntimeMXBean;

import java.lang.management.ThreadMXBean;

import java.util.List;

public class JVMMXBeanDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

//==========================Memory=========================

System.out.println("==========================Memory=========================");

MemoryMXBean memoryMBean = ManagementFactory.getMemoryMXBean();

MemoryUsage usage = memoryMBean.getHeapMemoryUsage();

System.out.println("初始化 Heap: " + (usage.getInit()/1024/1024) + "mb");

System.out.println("最大Heap: " + (usage.getMax()/1024/1024) + "mb");

System.out.println("已经使用Heap: " + (usage.getUsed()/1024/1024) + "mb");

System.out.println("Heap Memory Usage: " + memoryMBean.getHeapMemoryUsage());

System.out.println("Non-Heap Memory Usage: " + memoryMBean.getNonHeapMemoryUsage());

//==========================Runtime=========================

System.out.println("==========================Runtime=========================");

RuntimeMXBean runtimeMBean = ManagementFactory.getRuntimeMXBean();

System.out.println("JVM name : " + runtimeMBean.getVmName());

System.out.println("lib path : " + runtimeMBean.getLibraryPath());

System.out.println("class path : " + runtimeMBean.getClassPath());

System.out.println("getVmVersion() " + runtimeMBean.getVmVersion());

//java options

List<String> argList = runtimeMBean.getInputArguments();

for(String arg : argList){

System.out.println("arg : " + arg);

}

//==========================OperatingSystem=========================

System.out.println("==========================OperatingSystem=========================");

OperatingSystemMXBean osMBean = (OperatingSystemMXBean) ManagementFactory.getOperatingSystemMXBean();

//获取操作系统相关信息

System.out.println("getName() "+ osMBean.getName());

System.out.println("getVersion() " + osMBean.getVersion());

System.out.println("getArch() "+osMBean.getArch());

System.out.println("getAvailableProcessors() " + osMBean.getAvailableProcessors());

//==========================Thread=========================

System.out.println("==========================Thread=========================");

//获取各个线程的各种状态，CPU 占用情况，以及整个系统中的线程状况

ThreadMXBean threadMBean=(ThreadMXBean)ManagementFactory.getThreadMXBean();

System.out.println("getThreadCount() " + threadMBean.getThreadCount());

System.out.println("getPeakThreadCount() " + threadMBean.getPeakThreadCount());

System.out.println("getCurrentThreadCpuTime() " + threadMBean.getCurrentThreadCpuTime());

System.out.println("getDaemonThreadCount() " + threadMBean.getDaemonThreadCount());

System.out.println("getCurrentThreadUserTime() "+ threadMBean.getCurrentThreadUserTime());

//==========================Compilation=========================

System.out.println("==========================Compilation=========================");

CompilationMXBean compilMBean=(CompilationMXBean)ManagementFactory.getCompilationMXBean();

System.out.println("getName() " + compilMBean.getName());

System.out.println("getTotalCompilationTime() " + compilMBean.getTotalCompilationTime());

//==========================MemoryPool=========================

System.out.println("==========================MemoryPool=========================");

//获取多个内存池的使用情况

List<MemoryPoolMXBean> mpMBeanList= ManagementFactory.getMemoryPoolMXBeans();

for(MemoryPoolMXBean mpMBean : mpMBeanList){

System.out.println("getUsage() " + mpMBean.getUsage());

System.out.println("getMemoryManagerNames() "+ mpMBean.getMemoryManagerNames().toString());

}

//==========================GarbageCollector=========================

System.out.println("==========================GarbageCollector=========================");

//获取GC的次数以及花费时间之类的信息

List<GarbageCollectorMXBean> gcMBeanList=ManagementFactory.getGarbageCollectorMXBeans();

for(GarbageCollectorMXBean gcMBean : gcMBeanList){

System.out.println("getName() " + gcMBean.getName());

System.out.println("getMemoryPoolNames() "+ gcMBean.getMemoryPoolNames());

}

//==========================Other=========================

System.out.println("==========================Other=========================");

//Java 虚拟机中的内存总量,以字节为单位

int total = (int)Runtime.getRuntime().totalMemory()/1024/1024;

System.out.println("内存总量 ：" + total + "mb");

int free = (int)Runtime.getRuntime().freeMemory()/1024/1024;

System.out.println("空闲内存量 ： " + free + "mb");

int max = (int) (Runtime.getRuntime().maxMemory() /1024 / 1024);

System.out.println("最大内存量 ： " + max + "mb");

}

}

输出结果：   
==========================Memory=========================   
初始化 Heap: 16mb   
最大Heap: 247mb   
已经使用Heap: 0mb   
Heap Memory Usage: init = 16777216(16384K) used = 972640(949K) committed = 16252928(15872K) max = 259522560(253440K)   
Non-Heap Memory Usage: init = 163840(160K) used = 2474752(2416K) committed = 3145728(3072K) max = -1(-1K)   
==========================Runtime=========================   
JVM name : [**Java**](http://lib.csdn.net/base/java)HotSpot(TM) Client VM   
lib path : C:\Program Files\Java\jdk1.8.0\_60\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:/Program Files/Java/jre1.8.0\_60/bin/client;C:/Program Files/Java/jre1.8.0\_60/bin;C:/Program Files/Java/jre1.8.0\_60/lib/i386;C:\ProgramData\[**Oracle**](http://lib.csdn.net/base/oracle)\Java\javapath;C:\Program Files\Common Files\NetSarang;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;D:\apache-maven-3.3.3\bin;%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;D:\eclipse;;.   
class path : D:\workspace\test\target\test-classes;D:\workspace\test\target\classes;C:\Users\Administrator.m2\repository\junit\junit\4.11\junit-4.11.jar;C:\Users\Administrator.m2\repository\org\hamcrest\hamcrest-core\1.3\hamcrest-core-1.3.jar;C:\Users\Administrator.m2\repository\org\springframework\[**spring**](http://lib.csdn.net/base/javaee)-core\4.0.2.RELEASE\spring-core-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\commons-logging\commons-logging\1.1.3\commons-logging-1.1.3.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-web\4.0.2.RELEASE\spring-web-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-beans\4.0.2.RELEASE\spring-beans-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-context\4.0.2.RELEASE\spring-context-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-oxm\4.0.2.RELEASE\spring-oxm-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-tx\4.0.2.RELEASE\spring-tx-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-jdbc\4.0.2.RELEASE\spring-jdbc-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-webmvc\4.0.2.RELEASE\spring-webmvc-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-expression\4.0.2.RELEASE\spring-expression-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-aop\4.0.2.RELEASE\spring-aop-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\aopalliance\aopalliance\1.0\aopalliance-1.0.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-context-support\4.0.2.RELEASE\spring-context-support-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\springframework\spring-test\4.0.2.RELEASE\spring-test-4.0.2.RELEASE.jar;C:\Users\Administrator.m2\repository\org\mybatis\mybatis\3.2.6\mybatis-3.2.6.jar;C:\Users\Administrator.m2\repository\org\mybatis\mybatis-spring\1.2.2\mybatis-spring-1.2.2.jar;C:\Users\Administrator.m2\repository\javax\javaee-api\7.0\javaee-api-7.0.jar;C:\Users\Administrator.m2\repository\com\sun\mail\javax.mail\1.5.0\javax.mail-1.5.0.jar;C:\Users\Administrator.m2\repository\javax\activation\activation\1.1\activation-1.1.jar;C:\Users\Administrator.m2\repository\[**MySQL**](http://lib.csdn.net/base/mysql)\mysql-connector-java\5.1.30\mysql-connector-java-5.1.30.jar;C:\Users\Administrator.m2\repository\commons-dbcp\commons-dbcp\1.2.2\commons-dbcp-1.2.2.jar;C:\Users\Administrator.m2\repository\commons-pool\commons-pool\1.3\commons-pool-1.3.jar;C:\Users\Administrator.m2\repository\jstl\jstl\1.2\jstl-1.2.jar;C:\Users\Administrator.m2\repository\log4j\log4j\1.2.17\log4j-1.2.17.jar;C:\Users\Administrator.m2\repository\com\alibaba\fastjson\1.1.41\fastjson-1.1.41.jar;C:\Users\Administrator.m2\repository\org\slf4j\slf4j-api\1.7.7\slf4j-api-1.7.7.jar;C:\Users\Administrator.m2\repository\org\slf4j\slf4j-log4j12\1.7.7\slf4j-log4j12-1.7.7.jar;C:\Users\Administrator.m2\repository\org\codehaus\jackson\jackson-mapper-asl\1.9.13\jackson-mapper-asl-1.9.13.jar;C:\Users\Administrator.m2\repository\org\codehaus\jackson\jackson-core-asl\1.9.13\jackson-core-asl-1.9.13.jar;C:\Users\Administrator.m2\repository\commons-fileupload\commons-fileupload\1.3.1\commons-fileupload-1.3.1.jar;C:\Users\Administrator.m2\repository\commons-io\commons-io\2.4\commons-io-2.4.jar;C:\Users\Administrator.m2\repository\commons-codec\commons-codec\1.9\commons-codec-1.9.jar   
getVmVersion() 25.60-b23   
arg : -Dfile.encoding=UTF-8   
==========================OperatingSystem=========================   
getName() Windows 7   
getVersion() 6.1   
getArch() x86   
getAvailableProcessors() 3   
==========================Thread=========================   
getThreadCount() 5   
getPeakThreadCount() 5   
getCurrentThreadCpuTime() 140400900   
getDaemonThreadCount() 4   
getCurrentThreadUserTime() 78000500   
==========================Compilation=========================   
getName() HotSpot Client Compiler   
getTotalCompilationTime() 9   
==========================MemoryPool=========================   
getUsage() init = 163840(160K) used = 701888(685K) committed = 720896(704K) max = 33554432(32768K)   
getMemoryManagerNames() [Ljava.lang.String;@139a55   
getUsage() init = 0(0K) used = 1848776(1805K) committed = 2424832(2368K) max = -1(-1K)   
getMemoryManagerNames() [Ljava.lang.String;@1db9742   
getUsage() init = 4521984(4416K) used = 972640(949K) committed = 4521984(4416K) max = 71630848(69952K)   
getMemoryManagerNames() [Ljava.lang.String;@106d69c   
getUsage() init = 524288(512K) used = 0(0K) committed = 524288(512K) max = 8912896(8704K)   
getMemoryManagerNames() [Ljava.lang.String;@52e922   
getUsage() init = 11206656(10944K) used = 0(0K) committed = 11206656(10944K) max = 178978816(174784K)   
getMemoryManagerNames() [Ljava.lang.String;@25154f   
==========================GarbageCollector=========================   
getName() Copy   
getMemoryPoolNames() [Ljava.lang.String;@10dea4e   
getName() MarkSweepCompact   
getMemoryPoolNames() [Ljava.lang.String;@647e05   
==========================Other=========================   
内存总量 ：15mb   
空闲内存量 ： 14mb   
最大内存量 ： 247mb

参考博文：<http://www.what21.com/programming/java/java-summary/java-jvm-args.html>

# python

 python -m json.tool

# 2@Jolokia架构介绍 - 随风溜达的向日葵

https://my.oschina.net/chkui/blog/708639

**jolokia架构**

    虽然jolokia是为了满足JSR-160的要求，但是他和JSR-160连接器有巨大的差异。其中最引人注目的区别是jolokia传递数据是无类型的数据（说白了就是使用了Json数据传递，替代了RMI传递Java序列化数据的方式）。

    2003年提交的JSR-160规定客户端可以透明的调用MBean服务，无论被调用的MBean是驻留在本地还是在远程的MBean服务中。这样做的好处是提供了一个简洁通用的Java API接口。但是JSR-160的实现存在许多问题：

1. 它非常危险，因为它隐性暴露了JMX的远程接口。
2. 它还存在性能问题。无论是远程还是本地调用，调用者至少要知道调用过程是怎么样的、会收到什么结果。在实际使用时，需要有明确的远程消息传递模式，让调用者知道现在是在使用响应较慢的远程调用。
3. 使用RMI（JSR-160连接器的默认协议栈）时需要使用Java对象的序列化与反序列化机制来构建传递管道。这样做就阻碍了Java技术栈之外的环境来使用它。

    以上3个原因大概就是RMI（JSR-160连接器的默认协议栈）在远程传输协议上逐渐失去市场份额的原因。

    Jolokia是无类型的数据，使用了Json这种轻量化的序列化方案来替代RMI方案。使用这样的方法当然存在一些缺点（比如需要额外增加一层代理），但是带来了一些优势，至少这样的实现方案在JMX世界是独一无二的。

**Jolokia植入模式（Agent mode）**



    上如展示了Jolokia 植入模式的体系结构，说明了与之有关的运行环境。

    Jolokia植入模式是在本地基于http协议提供了一个使用Json作为数据格式的外部接口，此时Jolokia会桥接到本地的JMX MBeans接口。Jolokia使用http服务扩展了JSR-160，因此需要针对Jolokia的运行进行一些额外的处理。多种技术可以工作于http协议，最常规的方法是将jolokia放置到servlet容器中，比如Tomcat或Jetty，这样Jolokia完全可以看做是一个常规的Java web应用，让所有的开发人员都能够很好理解并快速的从中读取数据。

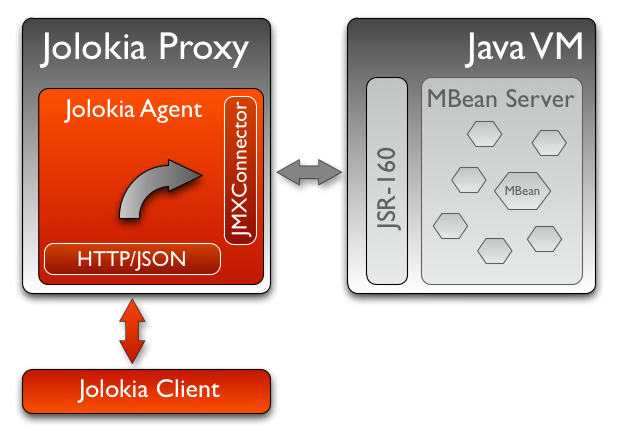
    当然还有更多的方式使用Jolokia植入，比如使用OSGi HttpService或嵌入到有Jetty-Server的应用中。Jvm代理者需要使用Java1.6以上版本，在他运行时，可以连接到任何本地运行的Java进程。

    附注——关于“植入模式”的称呼的说明：官方名为“Agent mode”，按照字面意思应该译为“代理者模式”。但是后面又一个模式叫代理模式（Proxy Mode），为了更便于理解和表达中文意思，这里命名其为“植入模式”。

**Jolokia代理模式**

    代理模式用于无法将Jolokia部署到目标平台上（说白了就是无法部署到同一台服务器）。在这个模式下，唯一可用的方式就是目标服务开启了JSR-160连接。这样做大部分是规范原因（原文是“political reasons”——政治原因-\_-）——有时候根本不允许在目标服务器部署一个额外的软件系统，或者是这样做需要等待一个漫长的审批流程。还有一个原因是目标服务器已经通过RMI开启了JSR-160连接，并且我们不想额外再去在本地部署Jolokia。

    可以将jolokia.war部署到servlet容器中（这个war包也可用于植入模式）。下图是一个典型的代理模式架构。



一个jolokia客户端发送常规的请求到jolokia代理服务，这个请求包含了额外的数据用于标记要查询的目标。所有的路由信息包含在请求信息中，使得代理服务无需特别的配置即可工作。

**结尾**

    如果没有什么特别的限制，优先使用植入模式。植入模式比代理模式有更多的优势，因为他没有附加层、减少了维度成本和技术复杂性、而且性能也优于代理模式。此外，一些jolokia特性也无法在代理模式中使用，例如“merging of MBeanServers”。

# jolokia使用心得 - ilxlf

https://my.oschina.net/u/145002/blog/31965

最近一直在找一个开源的工具，用来管理和配置集群环境里面的配置文件和配置项。后来发现jolokia可以做类似的东西。所以就拿过来用一用看看效果。

    做了两个例子来验证jolokia的效果和功能。

    例子1是一个standalone的应用程序，很简单的一个MBean。我把我做的流程记录下来供将来参考：

    HelloMBean.java  |   Hello.java    |    HelloAgent.java

    HelloMBean.java定义了MBean的接口:

public interface HelloMBean {

public String getName();

public void setName(String name);

public void printHello();

public void printHello(String whoName);

}

    Hello.java实现了这个接口:

public class Hello implements HelloMBean {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void printHello() {

System.out.println("Hello World, " + name);

}

public void printHello(String whoName) {

System.out.println("Hello , " + whoName);

}

}

    HelloAgent.java注册MBean，并启动这个应用程序:

import javax.management.MBeanServer;

import java.lang.management.ManagementFactory;

import javax.management.MBeanServerFactory;

import javax.management.ObjectName;

public class HelloAgent {

public static void main(String[] args) throws Exception {

MBeanServer server = ManagementFactory.getPlatformMBeanServer();

ObjectName helloName = new ObjectName("jolokia:name=HelloWorld");

server.registerMBean(new Hello(), helloName);

System.out.println("start.....");

Thread.sleep(1000000);

}

}

    下面的工作就是如果用jolokia来监视，配置这个应用程序：

     下载jolokia，然后解压缩到一个目录下面例如：/home/ilxlf/jolokia/jolokia-0.95

      $> cd ~/jolokia/jolokia-0.95/agents/

      有一个jar包叫做：jolokia-jvm6.jar 这里我们用到的是jolokia的一种MBean监控方式：jvm方式。实质上就是把这个jar包attach到你的应用程序，以此来达到监控的目的。

      $> java -jar jolokia-jvm6.jar

      运行上面的命令会显示当前该机器上正在运行的MBean server的所有程序的PID。

      我们上面的HelloAgent也在其中。这里假设HelloAgent的PID是27463

      $> java -jar jolokia-jvm6.jar --agentContext /HelloAgent start 27463

      Started Jolokia for PID 27463

      http://localhost:8778/HelloAgent/

      这样我们客户端就可以通过上面的这个link来访问、控制、修改我们的HelloMBean了。

      客户端代码如下：

import org.jolokia.client.\*;

import org.jolokia.client.request.\*;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

public class Config {

public static void main(String[] args) throws Exception {

J4pClient j4pClient = new J4pClient("http://localhost:8778/HelloAgent/");

J4pReadRequest req = new J4pReadRequest("jolokia:name=HelloWorld");

J4pReadResponse resp = j4pClient.execute(req);

Map<String,String> vals = resp.getValue();

Set<String> sset = vals.keySet();

Iterator<String> iter = sset.iterator();

while(iter.hasNext()){

String key = iter.next();

System.out.println(key);

System.out.println(vals.get(key));

}

}

}

运行这个程序就可以看到HelloMBean的name属性了。

例子2是一个war应用。如何用jolokia的第二个功能? 用jolokia war来监视我们自己的war程序。

这里我选择了tomcat来部署war包。

我用Eclipse jee创建了一个J2ee的小程序，非常小。包含一个MBean和一个servlet。

先列出MBean的接口和实现类：

public interface ConfigurationMBean {

public String getName();

public void setName(String name);

public String getValue();

public void setValue(String value);

public void printHello();

}

public class Configuration implements ConfigurationMBean {

private String name;

private String value;

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

public Configuration(String name){

this.name = name;

this.value = "1234";

}

public Configuration(){

this.name = "first class";

this.value = "123";

}

@Override

public String getName() {

// TODO Auto-generated method stub

return name;

}

@Override

public void setName(String name) {

// TODO Auto-generated method stub

this.name = name;

}

@Override

public void printHello() {

// TODO Auto-generated method stub

System.out.println(this.name);

}

}

servlet的代码如下：

import java.io.IOException;

import java.lang.management.ManagementFactory;

import javax.management.Attribute;

import javax.management.AttributeNotFoundException;

import javax.management.InstanceAlreadyExistsException;

import javax.management.InstanceNotFoundException;

import javax.management.IntrospectionException;

import javax.management.InvalidAttributeValueException;

import javax.management.MBeanAttributeInfo;

import javax.management.MBeanException;

import javax.management.MBeanInfo;

import javax.management.MBeanRegistrationException;

import javax.management.MBeanServer;

import javax.management.MBeanServerFactory;

import javax.management.MalformedObjectNameException;

import javax.management.NotCompliantMBeanException;

import javax.management.ObjectName;

import javax.management.ReflectionException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import com.ilxlf.jmx.common.Configuration;

public class JMXServlet extends HttpServlet {

/\*\*

\*

\*/

private static final long serialVersionUID = 1L;

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

// TODO Auto-generated method stub

System.out.println("enter--doGet");

super.doGet(req, resp);

System.out.println("exit--doGet");

}

@Override

public void init() throws ServletException {

// TODO Auto-generated method stub

System.out.println("enter--init");

super.init();

MBeanServer server = MBeanServerFactory.createMBeanServer("com.ilxlf.jmx.common");

ObjectName configuration = null;

try {

configuration = new ObjectName("com.ilxlf.jmx.common=NewValue");

server.registerMBean(new Configuration(), configuration);

} catch (MalformedObjectNameException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InstanceAlreadyExistsException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (MBeanRegistrationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (NotCompliantMBeanException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (MBeanException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("exit--init");

}

}

然后把这个j2ee工程打成war包(eclipse j22有export war功能)。然后拷贝到tomcat webapps目录下面。

下面在把jolokia war也拷贝到tomcat webapps目录下面。然后启动tomcat。

下面我们就可以用客户端来访问你自己的war里面的MBean了。

客户端代码如下：

import org.jolokia.client.\*;

import org.jolokia.client.request.\*;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

public class Config {

public static void main(String[] args) throws Exception {

J4pClient j4pClient = new J4pClient("http://localhost:8080/jolokia");

J4pReadRequest req = new J4pReadRequest("com.ilxlf.jmx.common:name=NewValue");

J4pReadResponse resp = j4pClient.execute(req);

Map<String,String> vals = resp.getValue();

Set<String> sset = vals.keySet();

Iterator<String> iter = sset.iterator();

while(iter.hasNext()){

String key = iter.next();

System.out.println(key);

System.out.println(vals.get(key));

}

}

}

运行这个client，会显示出ConfigurationMBean的两个属性: name and value

后面的工作就是分两个阶段：修改属性和进一步的深入了解jolokia的功能，包括在集群环境中如何使用jolokia来集中管理配置项。

# Jolokia - Reference Documentation

https://jolokia.org/reference/html/index.html



**Table of Contents**

[1. Introduction](https://jolokia.org/reference/html/introduction.html)

[2. Architecture](https://jolokia.org/reference/html/architecture.html)

[2.1. Agent mode](https://jolokia.org/reference/html/architecture.html#agent-mode)

[2.2. Proxy Mode](https://jolokia.org/reference/html/architecture.html#proxy-mode)

[3. Agents](https://jolokia.org/reference/html/agents.html)

[3.1. JEE Agent (WAR)](https://jolokia.org/reference/html/agents.html#agents-war)

[3.1.1. Installation and Configuration](https://jolokia.org/reference/html/agents.html#war-agent-installation)

[3.1.2. Security Setup](https://jolokia.org/reference/html/agents.html#agent-war-security)

[3.1.3. Programmatic usage of the Jolokia agent servlet](https://jolokia.org/reference/html/agents.html#agent-war-programmatic)

[3.2. OSGi Agents](https://jolokia.org/reference/html/agents.html#agents-osgi)

[3.2.1. jolokia-osgi.jar](https://jolokia.org/reference/html/agents.html#agents-osgi-pure)

[3.2.2. Running on Glassfish v3 upwards](https://jolokia.org/reference/html/agents.html#running-on-glassfish)

[3.2.3. jolokia-osgi-bundle.jar](https://jolokia.org/reference/html/agents.html#agents-osgi-bundle)

[3.2.4. Programmatic servlet registration](https://jolokia.org/reference/html/agents.html#agents-osgi-servlet)

[3.2.5. Restrictor service](https://jolokia.org/reference/html/agents.html#d0e1600)

[3.3. Mule Agent](https://jolokia.org/reference/html/agents.html#agents-mule)

[3.4. JVM Agent](https://jolokia.org/reference/html/agents.html#agents-jvm)

[3.4.1. Jolokia as JVM Agent](https://jolokia.org/reference/html/agents.html#jvm-agent)

[3.4.2. Attaching a Jolokia agent on the fly](https://jolokia.org/reference/html/agents.html#jvm-attach)

[4. Security](https://jolokia.org/reference/html/security.html)

[4.1. Policy based security](https://jolokia.org/reference/html/security.html#security-policy)

[4.1.1. IP based restrictions](https://jolokia.org/reference/html/security.html#d0e2784)

[4.1.2. Commands](https://jolokia.org/reference/html/security.html#d0e2800)

[4.1.3. Allow and deny access to certain MBeans](https://jolokia.org/reference/html/security.html#d0e2871)

[4.1.4. HTTP method restrictions](https://jolokia.org/reference/html/security.html#d0e2922)

[4.1.5. Cross-Origin Resource Sharing (CORS) restrictions](https://jolokia.org/reference/html/security.html#d0e2946)

[4.1.6. Example for a security policy](https://jolokia.org/reference/html/security.html#d0e2987)

[4.1.7. Policy Location](https://jolokia.org/reference/html/security.html#security-policy-location)

[4.2. Jolokia Restrictors](https://jolokia.org/reference/html/security.html#security-restrictor)

[5. Proxy Mode](https://jolokia.org/reference/html/proxy.html)

[5.1. Limitations of proxy mode](https://jolokia.org/reference/html/proxy.html#d0e3160)

[6. Jolokia Protocol](https://jolokia.org/reference/html/protocol.html)

[6.1. Requests and Responses](https://jolokia.org/reference/html/protocol.html#request-response)

[6.1.1. GET requests](https://jolokia.org/reference/html/protocol.html#get-request)

[6.1.2. POST requests](https://jolokia.org/reference/html/protocol.html#post-request)

[6.1.3. Responses](https://jolokia.org/reference/html/protocol.html#responses)

[6.1.4. Paths](https://jolokia.org/reference/html/protocol.html#paths)

[6.2. Jolokia operations](https://jolokia.org/reference/html/protocol.html#jolokia-operations)

[6.2.1. Reading attributes (read)](https://jolokia.org/reference/html/protocol.html#read)

[6.2.2. Writing attributes (write)](https://jolokia.org/reference/html/protocol.html#write)

[6.2.3. Executing JMX operations (exec)](https://jolokia.org/reference/html/protocol.html#exec)

[6.2.4. Searching MBeans (search)](https://jolokia.org/reference/html/protocol.html#search)

[6.2.5. Listing MBeans (list)](https://jolokia.org/reference/html/protocol.html#list)

[6.2.6. Getting the agent version (version)](https://jolokia.org/reference/html/protocol.html#version)

[6.3. Processing parameters](https://jolokia.org/reference/html/protocol.html#processing-parameters)

[6.4. Object serialization](https://jolokia.org/reference/html/protocol.html#serialization)

[6.4.1. Response value serialization](https://jolokia.org/reference/html/protocol.html#serialization-response)

[6.4.2. Request parameter serialization](https://jolokia.org/reference/html/protocol.html#serialization-request)

[6.4.3. Jolokia and MXBeans](https://jolokia.org/reference/html/protocol.html#serialization-mxbean)

[6.5. Tracking historical values](https://jolokia.org/reference/html/protocol.html#history)

[6.6. Proxy requests](https://jolokia.org/reference/html/protocol.html#protocol-proxy)

[6.7. Agent Discovery](https://jolokia.org/reference/html/protocol.html#discovery)

[6.8. Jolokia protocol versions](https://jolokia.org/reference/html/protocol.html#versions)

[7. Jolokia MBeans](https://jolokia.org/reference/html/mbeans.html)

[7.1. Configuration MBean](https://jolokia.org/reference/html/mbeans.html#mbean-config)

[7.1.1. Debugging](https://jolokia.org/reference/html/mbeans.html#mbean-debugging)

[7.1.2. History store](https://jolokia.org/reference/html/mbeans.html#mbean-history)

[7.2. Server Handler](https://jolokia.org/reference/html/mbeans.html#mbean-serverhandler)

[7.3. Discovery MBean](https://jolokia.org/reference/html/mbeans.html#mbean-discovery)

[8. Clients](https://jolokia.org/reference/html/clients.html)

[8.1. Javascript Client Library](https://jolokia.org/reference/html/clients.html#client-javascript)

[8.1.1. Installation](https://jolokia.org/reference/html/clients.html#js-installation)

[8.1.2. Usage](https://jolokia.org/reference/html/clients.html#js-request)

[8.1.3. Simple API](https://jolokia.org/reference/html/clients.html#js-simple)

[8.1.4. Request scheduler](https://jolokia.org/reference/html/clients.html#js-poller)

[8.1.5. Jolokia as a Cubism Source](https://jolokia.org/reference/html/clients.html#js-cubism)

[8.1.6. Maven integration](https://jolokia.org/reference/html/clients.html#js-maven)

[8.2. Java Client Library](https://jolokia.org/reference/html/clients.html#client-java)

[8.2.1. Tutorial](https://jolokia.org/reference/html/clients.html#client-java-tutorial)

[8.2.2. J4pClient](https://jolokia.org/reference/html/clients.html#client-client)

[8.2.3. Request types](https://jolokia.org/reference/html/clients.html#client-java-requests)

[8.2.4. Exceptions](https://jolokia.org/reference/html/clients.html#client-java-execptions)

[9. Jolokia JMX](https://jolokia.org/reference/html/jmx.html)

[9.1. Jolokia MBeanServer](https://jolokia.org/reference/html/jmx.html#jolokia-mbeanserver)

[9.1.1. MBeanServer merging](https://jolokia.org/reference/html/jmx.html#mbeanserver-merging)

[9.2. @JsonMBean](https://jolokia.org/reference/html/jmx.html#json-mbean)

[9.3. Spring Support](https://jolokia.org/reference/html/jmx.html#jvm-spring)

[9.3.1. JVM agent](https://jolokia.org/reference/html/jmx.html#jvm-spring-agent)

[9.3.2. Jolokia MBeanServer](https://jolokia.org/reference/html/jmx.html#jvm-spring-mbean-server)

[9.3.3. Jolokia Spring plugin](https://jolokia.org/reference/html/jmx.html#jvm-spring-plugin)

[10. Tools](https://jolokia.org/reference/html/tools.html)

[10.1. Jmx4Perl](https://jolokia.org/reference/html/tools.html#tools-jmx4perl)

[10.2. Jolokia Roo Addon](https://jolokia.org/reference/html/tools.html#tools-roo)

**List of Examples**

6.1. [JSON Request](https://jolokia.org/reference/html/protocol.html#request-example)

6.2. [JSON Response](https://jolokia.org/reference/html/protocol.html#response-example)

# Chapter 1. Introduction

JMX (Java Management Extensions) is the standard management solution in the Java world. Since JDK 1.5 it is available in every Java Virtual Machine and especially JEE application servers use JMX for their management business.

I love JMX. It is a well crafted specification, created in times where other concepts like EJBs failed spectacularly. Even more than ten years after its incubation it is still the one-and-only when it comes to management in the Java world. Especially the various levels of sophistications for implementing MBeans, starting with dead simple Standard MBeans and ending in very flexible Open MBeans and MXBeans, are impressive.

However, some of the advanced JMX concepts didn't really appeal to the public and are now effectively obsolete. Add-on standards like[JSR-77](http://jcp.org/en/jsr/detail?id=77) didn't received the adoption level they deserved. And then there is [JSR-160](http://jcp.org/en/jsr/detail?id=160), JMX remoting. This specificatiion is designed for ease of usage and has the ambition to transparently hide the technical details behind the remote communication so that is makes (nearly) no difference, whether MBeans are invoked locally or remotely. Unfortunately, the underlying transport protocol (RMI) and programing model is very Java centric and is not usable outside the Java world.

This is where Jolokia steps in. It is an agent based approach, living side by side with JSR-160, but uses the much more open HTTP for its transport business where the data payload is serialized in JSON. This opens a whole new world for different, non-Java clients. Beside this protocol switch, Jolokia provides new features for JMX remoting, which are not available in JSR-160 connectors: Bulk requests allow for multiple JMX operations with a single remote server roundtrip. A fine grained security mechanism can restrict the JMX access on specific JMX operations. Other features like the JSR-160 proxy mode or history tracking are specific to Jolokia, too.

This reference manual explains the details of Jolokia. After an overview of Jolokia's architecture in [Chapter 2, *Architecture*](https://jolokia.org/reference/html/architecture.html), installation and configuration of the various Jolokia agents are described in [Chapter 3, *Agents*](https://jolokia.org/reference/html/agents.html). Jolokia's security policy mechanism ([Chapter 4,*Security*](https://jolokia.org/reference/html/security.html)) and proxy mode ([Chapter 5, *Proxy Mode*](https://jolokia.org/reference/html/proxy.html)) are covered in the following chapters. For implementors of Jolokia client bindings the protocol definition is probably the most interesting part ([Chapter 6, *Jolokia Protocol*](https://jolokia.org/reference/html/protocol.html)). Jolokia itself comes with the preregistered MBeans listed in [Chapter 7, *Jolokia MBeans*](https://jolokia.org/reference/html/mbeans.html). The available client bindings are described in [Chapter 8, *Clients*](https://jolokia.org/reference/html/clients.html).

# Chapter 2. Architecture

https://jolokia.org/reference/html/architecture.html#proxy-mode

Chapter 2. Architecture

The architecture of Jolokia is quite different to that of JSR-160 connectors. One of the most striking difference is Jolokia's typeless approach.

JSR-160, released in 2003, has a different design goal than Jolokia. It is a specification with which a client can transparently invoke MBean calls, regardless whether the MBean resides within a local or remote MBeanServer. This provides a good deal of comfort for Java clients of this API, but it is also dangerous because it hides the remoteness of JMX calls. There are several subtle issues, performance being one of them. It does matter whether a call is invoked locally or remotely. A caller should at least be aware what happens and what the consequences are. On the other side, there are message passing models which include remoting explicitly, so that the caller knows from the programming model that she is calling a potentially expensive remote call. This is probably the main reason why RMI (the default protocol stack of JSR-160 connectors) lost market share to more explicit remote protocols.

One problem with JSR-160 is it implicit reliance on RMI and its requirement for a complete (Java) object serialization mechanism for passing management information over the wire. This closes the door for all environments which are not Java (or more precisely, JVM) aware. Jolokia uses a typeless approach, where some sort of lightweight serialization to JSON is used (in both directions, but a bitasymmetrically in its capabilities). Of course this approach has some drawbacks, too, but also quite some advantages. At least it is unique in the JMX world ;-).

2.1. Agent mode

[Figure 2.1, “Jolokia architecture”](https://jolokia.org/reference/html/architecture.html#fig-architecture-overview) illustrates the environment in which Jolokia operates. The agent exports on the frontside a JSON based protocol over HTTP that gets bridged to invocation of local JMX MBeans. It lives outside the JSR-160 space and hence requires a different setup. Various techniques are available for exporting its protocol via HTTP. The most prominent being to put the agent into a servlet container. This can be a light weight one like Tomcat or Jetty or a full-blown JEE Server. Since it acts like a usual web application the deployment of the agent is well understood and should pose no entry barrier for any developer who has ever dealt with Java web applications.

**Figure 2.1. Jolokia architecture**



But there are more options. Specialized agents are able to use an OSGi HttpService or come with an embedded Jetty-Server in case of the Mule agent. The JVM agent uses the HTTP-Server included with every Oracle JVM 6 and can be attached dynamically to any running Java process. Agents are described in detail in [Chapter 3, *Agents*](https://jolokia.org/reference/html/agents.html).

Jolokia can be also integrated into one's own applications very easily. The jolokia-core library (which comes bundled as a jar), includes a servlet which can be easily added to a custom application. [Section 3.1.3, “Programmatic usage of the Jolokia agent servlet”](https://jolokia.org/reference/html/agents.html#agent-war-programmatic)contains more information about this.

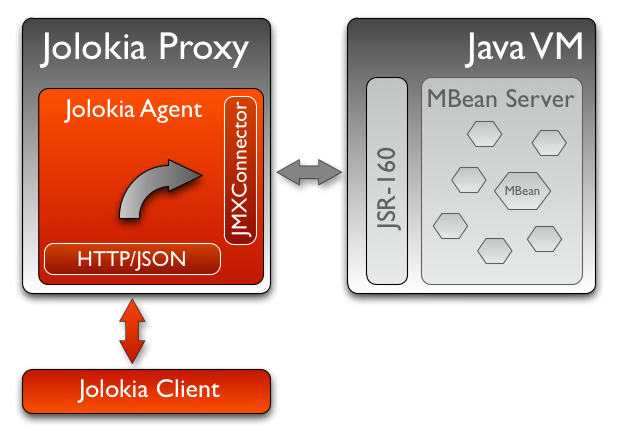
2.2. Proxy Mode

Proxy mode is a solution for when it is impossible to deploy the Jolokia agent on the target platform. For this mode, the only prerequisite for accessing the target server is a JSR-160 connection. Most of the time this happens for political reasons, where it is simply not allowed to deploy an extra piece of software or where doing so requires a lengthy approval process. Another reason could be that the target server already exports JMX via JSR-160 and you want to avoid the extra step of deploying the agent.

A dedicated proxy servlet server is needed for hosting jolokia.war, which by default supports both the agent mode and the proxy mode. A lightweight container like Tomcat or Jetty is a perfect choice for this kind of setup.

Figure [Figure 2.2, “Jolokia as JMX Proxy”](https://jolokia.org/reference/html/architecture.html#fig-architecture-proxy) describes a typical setup for the proxy mode. A client sends a usual Jolokia request containing an extra section for specifying the target which should be queried. All routing information is contained in the request itself so that the proxy can act universally without the need of a specific configuration.

**Figure 2.2. Jolokia as JMX Proxy**



Having said all that, the proxy mode has some limitations which are listed in [Chapter 5, *Proxy Mode*](https://jolokia.org/reference/html/proxy.html).

To summarize, the proxy mode should be used only when required. The agent servlet on its own is more powerful than the proxy mode since it eliminates an additional layer adding to the overall complexity and performance. Also, some features like merging of MBeanServers are not available in the proxy mode.

# 2@Chapter 3. Agents

https://jolokia.org/reference/html/agents.html

Chapter 3. Agents

Jolokia is an agent based approach to JMX, which requires that clients install an extra piece of software, the so-called agent. This software either needs to be deployed on the target server which should be accessed via remote JMX ([Section 2.1, “Agent mode”](https://jolokia.org/reference/html/architecture.html#agent-mode)), or it can be installed on a dedicated proxy server ([Section 2.2, “Proxy Mode”](https://jolokia.org/reference/html/architecture.html#proxy-mode)). For both operational modes, there are four different kind of agents[[1](https://jolokia.org/reference/html/agents.html" \l "ftn.d0e153)].

**Webarchive (War) agent**

This agent is packaged as a JEE Webarchive (War). It is the standard installation artifact for Java webapplications and probably one of the best known deployment formats. Jolokia ships with a war-agent which can be deployed like any other web application. This agent has been tested on many JEE servers, from well-known market leaders to rarer species.

**OSGi agent**

[OSGi](http://www.osgi.org/) is a middleware specification focusing on modularity and a well defined dynamic lifecycle [[2](https://jolokia.org/reference/html/agents.html" \l "ftn.d0e172)]. The Jolokia OSGi agent bundles comes in two flavors: a minimal one with a dependency on a running [OSGi HttpService](http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/HttpService.html), and a all-in-one bundle including an embedded HttpService implementation (which is exported, too). The former is the recommended, puristic solution, the later is provided for a quick startup for initial testing the OSGi agent (but should be replaced with the minimal bundle for production setups).

**Mule agent**

[Mule](http://www.mulesoft.org/) is one of the leading Open Source Enterprise Service Busses[[3](https://jolokia.org/reference/html/agents.html" \l "ftn.d0e193)] (ESB). It provides a management API into which a dedicated Jolokia agent plugs in nicely. This agent includes an embedded Jetty for providing JMX HTTP access.

**JVM agent**

Starting with Java 6 the JDK provided by Oracle contains a lightweight HTTP-Server which is used e.g. for the reference WebService stack implementation included in Java 6. Using the Java-agent API (normally used by profilers and other development tools requiring the instrumentation during the class loading phase), the JVM 6 Jolokia agent is the most generic one. It is able to instrument any Java application running on a Oracle JDK 6[[4](https://jolokia.org/reference/html/agents.html" \l "ftn.d0e209)]. This Jolokia agent variant is fully featured, however tends to be a bit slow since the provided HTTP-Server is not optimized for performance. However it is useful for servers like Hadoop or Teracotta, which do not provide convenient hooks for an HTTP-exporting agent on their own.

## 3.1. JEE Agent (WAR)

3.1.1. Installation and Configuration

The WAR agent is the most popular variant, and can be deployed in a servlet container just like any other JEE web application.

**Tomcat example**

A simple example for deploying the agent on Tomcat can be found in the Jolokia [quickstart](http://www.jolokia.org/tutorial.html).

Often, installation is simply a matter of copying the agent WAR to a deployment directory. On other platforms an administrative Web GUI or a command line tool need to be used for deployment. Providing detailed installation instructions for every servlet container is out of scope for this document.

The servlet itself can be configured in two ways:

**Servlet Init Parameters**

Jolokia can be configured with init-param declarations within the servlet definition in WEB-INF/web.xml. The known parameters are described in [Table 3.1, “Servlet init parameters”](https://jolokia.org/reference/html/agents.html#agent-war-init-params). The stock agent needs to be repackaged, though, in order to modify the internal web.xml.

**Servlet Context Parameters**

A more convenient possibility might be to use servlet context parameters, which can be configured outside the WAR archive. This is done differently for each servlet container but involves typically the editing of a configuration file. E.g. for [Tomcat](http://tomcat.apache.org/tomcat-7.0-doc/config/context.html), the context for the Jolokia agent can be adapted by putting a file jolokia.xml below $TC/conf/Catalina/localhost/ with a content like:

<Context>

<Parameter name="maxDepth" value="1"/>

</Context>

The configuration options discoveryEnabled and discoveryAgentUrl can be provied via environent variables or system properties, too. See the below for details.

**Table 3.1. Servlet init parameters**

| **Parameter** | **Description** | **Example** |
| --- | --- | --- |
| dispatcherClasses | Classnames (comma separated) ofRequestDispatcher used in addition to theLocalRequestDispatcher. Dispatchers are a technique used by the JSR-160 proxy to dispatch (or 'route') a request to a different destination. | org.jolokia.jsr160.Jsr160RequestDispatcher(this is the dispatcher for the JSR-160 proxy) |
| policyLocation | Location of the policy file to use. This is either a URL which can read from (like a file: or http:URL) or with the special protocol classpath:which is used for looking up the policy file in the web application's classpath. See [Section 4.1.7, “Policy Location”](https://jolokia.org/reference/html/security.html#security-policy-location) for details about this parameter. | file:///home/jolokia/jolokia-access.xml for a file based access to the policy file. Default isclasspath:/jolokia-access.xml |
| restrictorClass | Full classname of an implementation oforg.jolokia.restrictor.Restrictor which is used as a custom restrictor for securing access via Jolokia. | com.mycompany.jolokia.CustomRestrictor(which must be included in the war file and must implementorg.jolokia.restrictor.Restrictor) |
| allowDnsReverseLookup | Access can be restricted based on the remote host accessing Jolokia. This host can be specified as address or an hostname. However, using the hostname normally requires a reverse DNS lookup which might slow down operations. In order to avoid this reverse DNS lookup set this property to false. | Default: true |
| debug | Debugging state after startup. Can be changed via the config MBean during runtime. | Default: false |
| logHandlerClass | Loghandler to use for providing logging output. By default logging is written to standard out and error but you can provide here a Java class implementing org.jolokia.util.LogHandlerfor an alternative log output. Two alternative implementations are included in this agent:   * org.jolokia.util.QuietLogHandlerwhich switches off logging completely. * org.jolokia.util.JulLogHandler which uses a java.util.logging Logger with name org.jolokia | Example: org.jolokia.util.LogHandler.Quiet |
| historyMaxEntries | Entries to keep in the history. Can be changed at runtime via the config MBean. | Default: 10 |
| debugMaxEntries | Maximum number of entries to keep in the local debug history (if enabled). Can be changed via the config MBean at runtime. | Default: 100 |
| maxDepth | Maximum depth when traversing bean properties. If set to 0, depth checking is disabled | Default: 15 |
| maxCollectionSize | Maximum size of collections returned when serializing to JSON. When set to 0, collections are never truncated. | Default: 1000 |
| maxObjects | Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed. | Default: 0 |
| mbeanQualifier | Qualifier to add to the ObjectName of Jolokia's own MBeans. This can become necessary if more than one agent is active within a servlet container. This qualifier is added to theObjectName of this agent with a comma. For example a mbeanQualifier with the valuequalifier=own will result in Jolokia server handler MBean with the namejolokia:type=ServerHandler,qualifier=own |  |
| mimeType | MIME to use for the JSON responses | Default: text/plain |
| canonicalNaming | This option specifies in which order the key-value properties within ObjectNames as returned bylist or search are returned. By default this is the so called 'canonical order' in which the keys are sorted alphabetically. If this option is set tofalse, then the natural order is used, i.e. the object name as it was registered. This option can be overridden with a query parameter of the same name. | Default: true |
| includeStackTrace | Whether to include a stacktrace of an exception in case of an error. By default it it set to true in which case the stacktrace is always included. If set to false, no stacktrace is included. If the value is runtime a stacktrace is only included for RuntimeExceptions. This global option can be overridden with a query parameter. | Default: true |
| serializeException | When this parameter is set to true, then an exception thrown will be serialized as JSON and included in the response under the keyerror\_value. No stacktrace information will be included, though. This global option can be overridden by a query parameter of the same name. | Default: false |
| allowErrorDetails | If set to true then no error details like a stack trace (when includeStackTrace is set) or a serialized exception (when serializeExceptinis set) are included. This can be user as a startup option to avoid exposure of error details regardless of other options. | Default: true |
| detectorOptions | Extra options passed to an detector after successful detection of an application server. See below for an explanation. |  |
| discoveryEnabled | Is set to true then this servlet will listen for multicast request (multicastgroup 239.192.48.84, port 24884). By default this option is disabled in order to avoid conflicts with an JEE standards (though this should't harm anyways). This option can also be switched on with an environment variableJOLOKIA\_DISCOVERY or the system propertyjolokia.discoveryEnabled set to true. | Default: false |
| discoveryAgentUrl | Sets the URL to respond for multicast discovery requests. If given, discoveryEnabled is set implicetly to true. This URL can also be provied by an environment variableJOLOKIA\_DISCOVERY\_URL or the system property jolokia.discoveryUrl. Within the value you can use the placeholders ${host} and${ip} which gets replaced by the autodetected local host name/address. Also with${env:ENV\_VAR} and ${sys:property}environment and system properties can be referenced, respectively. | http://10.9.11.87:8080/jolokia |
| agentId | A unique ID for this agent. By default a unique id is calculated. If provided it should be ensured that this id is unique among all agent reachable via multicast requests used by the discovery mechanism. It is recommended not to set this value. Within the agentId specification you can use the same placeholders as indiscoveryAgentUrl. | my-unique-agent-id |
| agentDescription | An optional description which can be used for clients to present a human readable label for this agent. | Intranet Timebooking Server |

Jolokia has various detectors which can detect the brand and version of an application server it is running in. This version is revealed with the version command. With the configuration parameter detectorOptions extra options can be passed to the detectors. These options take the form of a JSON object, where the keys are productnames and the values other JSON objects containing the specific configuration. This configuration is feed to a successful detector which can do some extra initialization on agent startup. Currently the following extra options are supported:

**Table 3.2. Detector Options**

| **Product** | **Option** | **Description** |
| --- | --- | --- |
| glassfish | bootAmx | If false and the agent is running on Glassfish, this will cause the AMX subsystem not to be booted during startup. By default, AMX which contains all relevant MBeans for monitoring Glassfish is booted. |

3.1.2. Security Setup

In order use JEE security within the war, some extrat configuration steps are required within web.xml.

**Using jmx4perl's jolokia tool**

[jmx4perl](http://www.jmx4perl.org/) comes with a nice command line utility called [jolokia](http://search.cpan.org/~roland/jmx4perl/scripts/jolokia)which allows for an easy setup of security within a givenjolokia.war. See [Section 10.1, “Jmx4Perl”](https://jolokia.org/reference/html/tools.html#tools-jmx4perl) for more details.

There is a commented section which can serve as an example. All current client libraries are able to use BASIC HTTP authentication with user and password. The <login-config>should be set accordingly. The <security-constraint> specifies the URL pattern (which is in the default setup specify all resources provided by the Jolokia servlet) and a role name which is used to find the proper authentication credentials. This role must be referenced outside the agent WAR within the servlet container, e.g. for Tomcat the role definition can be found in $TOMCAT/config/tomcat-users.xml.

3.1.3. Programmatic usage of the Jolokia agent servlet

The Jolokia agent servlet can be integrated into one's own web-applications as well. Simply add a servlet with the servlet class org.jolokia.http.AgentServlet to your own web.xml. The following example maps the agent to the context /jolokia:

<servlet>

<servlet-name>jolokia-agent</servlet-name>

<servlet-class>org.jolokia.http.AgentServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>jolokia-agent</servlet-name>

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

</servlet-mapping>

Of course, any init parameter as described in [Table 3.1, “Servlet init parameters”](https://jolokia.org/reference/html/agents.html#agent-war-init-params) can be used here as well.

In order for this servlet definition to find the referenced Java class, the JAR jolokia-core.jar must be included. This jar can be found in[Maven central](http://repo1.maven.org/maven2/org/jolokia/jolokia-core). Maven users will can declare a dependency on this jar artifact:

<project>

<!-- .... -->

<dependencies>

<dependency>

<groupId>org.jolokia</groupId>

<artifactId>jolokia-core</artifactId>

<version>${jolokia.version}</version>

</dependency>

</dependencies>

<!-- .... -->

</project>

The org.jolokia.http.Agent can be subclassed, too in order to provide a custom restrictor or a custom log handler. See [Section 4.2, “Jolokia Restrictors”](https://jolokia.org/reference/html/security.html#security-restrictor) for details.[[5](https://jolokia.org/reference/html/agents.html" \l "ftn.d0e858)]

Also, multiple Jolokia agents can be deployed in the same JVM without problem. However, since the agent deploys some Jolokia-specific MBeans on the single PlatformMBeansServer, for multi-agent deployments it is important to use the mbeanQualifier init parameter to distinguish multiple Jolokia MBeans by adding an extra propery to those MBeans' names. This also needs to be done if multiple webapps containing Jolokia agents are deployed on the same JEE server.

## 3.2. OSGi Agents

There are several free implementations available of OSGi HttpService. This bundle has been tested with the [Pax Web](http://wiki.ops4j.org/display/paxweb/Pax+Web) and[Apache Felix](http://felix.apache.org/site/apache-felix-http-service.html) HttpService, both of which come with an embedded Jetty as servlet container by default.

Jolokia agents are also available as [OSGi](http://www.osgi.org/) bundles. There are two flavors of this agent: A nearly bare agent jolokia-osgi.jar declaring all its package dependencies as imports in its Manifest and an all-in-one bundle jolokia-osgi-bundle.jar with minimal dependecies. The pure bundle fits best with the OSGi philosophy and is hence the recommended bundle. The all-in-one monster is good for a quick start since normally no additional bundles are required.

3.2.1. jolokia-osgi.jar

This bundle depends mostly on a running [OSGi HttpService](http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/HttpService.html) which it uses for registering the agent servlet.

All package imports of this bundle are listed in [Table 3.3, “Package Imports of jolokia-osgi.jar (SB: exported by system bundle)”](https://jolokia.org/reference/html/agents.html#table-agents-osgi-deps). Note that the org.osgi.framework.\* and javax.\* packages are typically exported by the system bundle, so no extra installation effort is required here. Whether the org.osgi.service.\* interfaces are available depends on your OSGi container. If they are not provided, they can be easily fetched and installed from e.g. [maven central](http://repo1.maven.org/maven2/org/osgi/org.osgi.compendium/4.2.0/org.osgi.compendium-4.2.0.jar). Often the LogService interface is exported out of the box, but not the HttpService. You will notice any missing package dependency during the resolve phase while installing jolokia-osgi.jar.

**Table 3.3. Package Imports of jolokia-osgi.jar (SB: exported by system bundle)**

| **Package** | **SB** | **Package** | **SB** | **Package** | **SB** | **Package** | **SB** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| org.osgi.framework | X | javax.servlet |  | org.w3c.dom | X | javax.management | X |
| org.osgi.service.http |  | javax.servlet.http |  | org.xml.sax | X | javax.management.openmbean | X |
| org.osgi.service.log | ? | javax.naming | X | javax.xml.parsers | X | javax.management.remote | X |
| org.osgi.util.tracker | X |  |  |  |  |  |  |

This agent bundle consumes two services by default: As stated above, an org.osgi.service.http.HttpService which is used to register (deregister) the Jolokia agent as a servlet under the context /jolokia by default as soon as the HttpService becomes available (unavailable). Secondly, an org.osgi.service.log.LogService is used for logging, if available. If such a service is not registered, the Jolokia bundle uses the standard HttpServlet.log() method for its logging needs.

The Jolokia OSGi bundle can be configured via the OSGi Configuration Admin service using the PID org.jolokia.osgi (e.g. if using Apache Karaf, place properties in etc/org.jolokia.osgi.cfg), or alternatively via global properties which typically can be configured in a configuration file of the OSGi container. All properties start with the prefix org.jolokia and are listed in [Table 3.4, “Jolokia Bundle Properties”](https://jolokia.org/reference/html/agents.html#table-agents-osgi-properties). They are mostly the same as the init-param options for a Jolokia servlet when used in a JEE WAR artifact.

**Table 3.4. Jolokia Bundle Properties**

| **Property** | **Default** | **Description** |
| --- | --- | --- |
| org.jolokia.user |  | User used for authentication with HTTP Basic Authentication. If not given, no authentication is used. |
| org.jolokia.password |  | Password used for authentication with HTTP Basic Authentication. |
| org.jolokia.agentContext | /jolokia | Context path of the agent servlet |
| org.jolokia.agentId |  | A unique ID for this agent. By default a unique id is calculated. If provided it should be ensured that this id is unique among all agent reachable via multicast requests used by the discovery mechanism. It is recommended not to set this value. Within the agentIdspecification you can use the same placeholders as in discoveryAgentUrl. |
| org.jolokia.agentDescription |  | An optional description which can be used for clients to present a humand readable label for this agent. |
| org.jolokia.dispatcherClasses |  | Class names (comma separated) of request dispatchers used in addition to the LocalRequestDispatcher. E.g using a value oforg.jolokia.jsr160.Jsr160RequestDispatcher allows the agent to play the role of a JSR-160 proxy. |
| org.jolokia.debug | false | Debugging state after startup. This can be changed via the Config MBean (jolokia:type=Config) at runtime |
| org.jolokia.debugMaxEntries | 100 | Maximum number of entries to keep in the local debug history if switched on. This can be changed via the config MBean at runtime. |
| org.jolokia.maxDepth | 0 | Maximum depth when traversing bean properties. If set to 0, depth checking is disabled |
| org.jolokia.maxCollectionSize | 0 | Maximum size of collections returned when serializing to JSON. When set to 0, collections are not truncated. |
| org.jolokia.maxObjects | 0 | Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed. |
| org.jolokia.historyMaxEntries | 10 | Number of entries to keep in the history. This can be changed at runtime via the Jolokia config MBean. |
| org.jolokia.listenForHttpService | true | If true the bundle listens for an OSGi HttpService and if available registers an agent servlet to it. |
| org.jolokia.httpServiceFilter |  | Can be any valid OSGi filter for locating a whichorg.osgi.service.http.HttpService is used to expose the Jolokia servlet. The syntax is that used by the org.osgi.framework.Filter which is in turn a [RFC 1960 based filter](http://www.ietf.org/rfc/rfc1960.txt). The use of this property is described in [Section 3.2.2, “Running on Glassfish v3 upwards”](https://jolokia.org/reference/html/agents.html#running-on-glassfish) |
| org.jolokia.useRestrictorService | false | If true the Jolokia agent will use any org.jolokia.restrictor.Restrictor service for applying access restrictions. If this option is false the standard method of looking up a security policy file is used, as described in [Section 4.1, “Policy based security”](https://jolokia.org/reference/html/security.html#security-policy). |
| org.jolokia.canonicalNaming | true | This option specifies in which order the key-value properties within ObjectNames as returned by list or search are returned. By default this is the so called 'canonical order' in which the keys are sorted alphabetically. If this option is set to false, then the natural order is used, i.e. the object name as it was registered. This option can be overridden with a query parameter of the same name. |
| org.jolokia.includeStackTrace | true | Whether to include a stacktrace of an exception in case of an error. By default it it set to true in which case the stacktrace is always included. If set to false, no stacktrace is included. If the value is runtime a stacktrace is only included for RuntimeExceptions. This global option can be overridden with a query parameter. |
| org.jolokia.serializeException | false | When this parameter is set to true, then an exception thrown will be serialized as JSON and included in the response under the key error\_value. No stactrace infornmation will be included, though. This global option can be overridden by a query parameter of the same name. |
| org.jolokia.detectorOptions |  | An optional JSON representation for application specific options used by detectors for post-initialization steps. See the description of detectorOptions in [Table 3.1, “Servlet init parameters”](https://jolokia.org/reference/html/agents.html#agent-war-init-params) for details. |
| org.jolokia.discoveryEnabled | false | Is set to true then this servlet will listen for multicast request (multicastgroup 239.192.48.84, port 24884). By default this option is disabled in order to avoid conflicts with an JEE standards (though this should't harm anyways). This option can also be switched on with an environment variable JOLOKIA\_DISCOVERY or the system property jolokia.discoveryEnabled set to true. |
| org.jolokia.discoveryAgentUrl |  | Sets the URL to respond for multicast discovery requests. If given, discoveryEnabledis set implicetly to true. This URL can also be provied by an environment variableJOLOKIA\_DISCOVERY\_URL or the system property jolokia.discoveryUrl. Within the value you can use the placeholders ${host} and ${ip} which gets replaced by the autodetected local host name/address. Also with ${env:ENV\_VAR} and${sys:property} environment and system properties can be referenced, respectively. |
| org.jolokia.realm | jolokia | Sets the security realm to use. If the authMode is set to jaas this is also used as value for the security domain. E.g. for Karaf 3 and later, this realm should be karafsince all JMX MBeans are guarded by this security domain. |
| org.jolokia.authMode | basic | Can be either basic (the default) or jaas. If jaas is used, the user and password given in the Authorization: header are used for loging in via JAAS and, if successful, the return subject is used for all Jolokia operation. This has only an effect, if user is set. |

This bundle also exports the service org.jolokia.osgi.servlet.JolokiaContext which can be used to obtain context information of the registered agent like the context path under which this servlet can be reached. Additionally, it exportsorg.osgi.service.http.HttpContext, which is used for authentication. Note that this service is only available when the agent servlet is active (i.e. when an HttpService is registered).

3.2.2. Running on Glassfish v3 upwards

You have a couple of choices when running jolokia on Glassfish v3 and up, since Glassfish is a both a fully fledged JEE container and an OSGi container. If you choose to run the [Section 3.1, “JEE Agent (WAR)”](https://jolokia.org/reference/html/agents.html#agents-war) then it is completely straight forward just deploy the war in the normal way. If you choose to deploy the [Section 3.2, “OSGi Agents”](https://jolokia.org/reference/html/agents.html#agents-osgi) then you will need to configure theorg.jolokia.httpServiceFilter option with a filter to select either the Admin HttpService (4848 by default) or the DefaultHttpService which is where WAR files are deployed to.

In Glassfish 3.1.2 the OSGi bundle configuration is done in glassfish/conf/osgi.properties in version's prior to this the configuration is by default in glassfish/osgi/felix/conf/config.properties or if you are using Equinoxglassfish/osgi/equinox/configuration/config.ini

# Restrict the jolokia http service selection to the admin host

org.jolokia.httpServiceFilter=(VirtualServer=\_\_asadmin)

# Or alternatively to the normal http service use : (VirtualServer=server)

Deploying the bundle can be either be done by coping the jolokia-osgi.jar into the domainglassfish/domains/<domain>/autodeploy/bundles directory or it can be added to all instances by copying the jar toglassfish/modules/autostart

By default the agent will be available on http://localhost:<port>/osgi/jolokia rather than http://localhost:<port>/jolokiaas with WAR deployment.

3.2.3. jolokia-osgi-bundle.jar

The all-in-one bundle includes an implementation of org.osgi.service.http.HttpService, i.e. the [Felix implementation](http://felix.apache.org/site/apache-felix-http-service.html). The HttpService will be registered as OSGi service during startup, so it is available for other bundles as well. The only package import requirement for this bundle is org.osgi.service.LogService, since the Felix Webservice requires this during startup. As mentioned above, normally the LogService interface gets exported by default in the standard containers, but if not, you need to install it e.g. from the OSGi [compendium](http://repo1.maven.org/maven2/org/osgi/org.osgi.compendium/4.2.0/org.osgi.compendium-4.2.0.jar)definitions.

This bundle can be configured the same way as the pure bundle as described in [Section 3.2.1, “jolokia-osgi.jar”](https://jolokia.org/reference/html/agents.html#agents-osgi-pure). Additionally, the embedded Felix HttpService can be configured as described in its [documentation](http://felix.apache.org/site/apache-felix-http-service.html#ApacheFelixHTTPService-ConfigurationProperties). e.g. setting the port to 9090 instead of the default port 8080, a property org.osgi.service.http.port=9090 needs to be set. This might be useful, if this bundle is used within containers which already occupy the default port (Glassfish, Eclipse Virgo) but don't expose an OSGi HttpService.

3.2.4. Programmatic servlet registration

It is also possible to register the Jolokia agent servlet manually instead of relying of the OSGi bundle activator which comes with the agents. For this use case jolokia-osgi.jar should be used. This bundle exports the package org.jolokia.osgi.servlet which includes the servlet class JolokiaServlet. This class has three constructors: A default constructor without arguments, one with a single BundleContext argument and finally one with an additional Restrictor (see [Section 4.2, “Jolokia Restrictors”](https://jolokia.org/reference/html/security.html#security-restrictor) for details how access restrictions can be applied). The constructor with a BundleContext as its argument has the advantage that it will use an OSGiLogService if available and adds various OSGi server detectors which adds server information like product name and version to theversion command. Refer to [Section 6.2.6, “Getting the agent version (version)”](https://jolokia.org/reference/html/protocol.html#version) for details about the server infos provided.

Please note that for this use case the bundle org.jolokia.osgi should not be started but left in the state resolved. Otherwise, as soon as an OSGi HttpService registers, this bundle will try to add yet another agent servlet to this service, which is probably not what you want. Alternatively, the bundle property org.jolokia.listenForHttpService can be set to false in which case there will be never an automatic servlet registration to an HttpService.

3.2.5. Restrictor service

As described in [Section 4.2, “Jolokia Restrictors”](https://jolokia.org/reference/html/security.html#security-restrictor), the Jolokia agent can use custom restrictors implementing the interfaceorg.jolokia.restrictor.Restrictor. If the bundle property org.jolokia.useRestrictorService is set to true and no restrictor is configured by other means, the agent will use one or more OSGi service which register under the nameorg.jolokia.restrictor.Restrictor. If no such service is available, access to the agent is always denied. If one such restrictor service is available, the access decision is delegated to this service. When more than one restrictor service is available, access is ony granted if all of them individually grant access. A sample restrictor service as a maven project can be found in the Jolokia source atagent/osgi/restrictor-sample.

## 3.3. Mule Agent

Jolokia's [Mule](http://www.mulesoft.org/) agent uses Mule's own agent interface for plugging into the ESB running in standalone mode.

The agent needs to be included into the Mule configuration as shown in the following example, which is the way how to configure the agent for Mule 3:

<mule xmlns="http://www.mulesoft.org/schema/mule/core"

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

xmlns:management="http://www.mulesoft.org/schema/mule/management"

xmlns:spring="http://www.springframework.org/schema/beans"

xsi:schemaLocation="

http://www.mulesoft.org/schema/mule/core

http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.mulesoft.org/schema/mule/management

http://www.mulesoft.org/schema/mule/management/3.1/mule-management.xsd">

<!-- .... -->

<custom-agent name="jolokia-agent" class="org.jolokia.mule.JolokiaMuleAgent">

<spring:property name="port" value="8899"/>

</custom-agent>

<management:jmx-server/>

</mule>

For Mule 2, the configuration is slightly different since the <custom-agent> is contained in the management namespace for Mule 2 (<management:custom-agent>)

This agent knows about the following configuration parameters

**Table 3.5. Mule agent configuration options**

| **Parameter** | **Description** | **Example** |
| --- | --- | --- |
| host | Hostaddress to which the HTTP server should bind to. | InetAddress.getLocalHost() |
| port | Port the HTTP server should listen to. | 8888 |
| user | Use to authenticate against. This switches on security and requires a client to provide a user and password. |  |
| password | Password to check against when security is switched on. |  |
| debug | Debugging state after startup. Can be changed via the [Section 7.1, “Configuration MBean”](https://jolokia.org/reference/html/mbeans.html#mbean-config) during runtime. | false |
| historyMaxEntries | Entries to keep in the history. Can be changed at runtime via the[Section 7.1, “Configuration MBean”](https://jolokia.org/reference/html/mbeans.html#mbean-config). | 10 |
| debugMaxEntries | Maximum number of entries to keep in the local debug history (if enabled). Can be changed via the [Section 7.1, “Configuration MBean”](https://jolokia.org/reference/html/mbeans.html#mbean-config) at runtime. | 100 |
| maxDepth | Maximum depth when traversing bean properties. If set to 0, depth checking is disabled | 5 |
| maxCollectionSize | Maximum size of collections returned when serializing to JSON. When set to 0, collections are never truncated. | 0 |
| maxObjects | Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed. | 10000 |
| canonicalNaming | This option specifies in which order the key-value properties within ObjectNames as returned by list or search are returned. By default this is the so called 'canonical order' in which the keys are sorted alphabetically. If this option is set to false, then the natural order is used, i.e. the object name as it was registered. This option can be overridden with a query parameter of the same name. | true |
| includeStackTrace | Whether to include a stacktrace of an exception in case of an error. By default it it set to true in which case the stacktrace is always included. If set to false, no stacktrace is included. If the value is runtime a stacktrace is only included for RuntimeExceptions. This global option can be overridden with a query parameter. | true |
| serializeException | When this parameter is set to true, then an exception thrown will be serialized as JSON and included in the response under the keyerror\_value. No stactrace infornmation will be included, though. This global option can be overridden by a query parameter of the same name. | false |

The context under which the agent is reachable is fixed to /jolokia. As an alternative to this Mule agent, the [Section 3.4, “JVM Agent”](https://jolokia.org/reference/html/agents.html#agents-jvm)can be used for Mule, too. This agent also knows about SSL encryption and authentication.

## 3.4. JVM Agent

The JVM agent is right agent when it comes to instrument an arbitrary Java application which is not covered by the other agents. This agent can be started by any Java program by providing certain startup options to the JVM. Or it can be dynamically attached (and detached) to an already running Java process. This universal agent uses the [JVM agent API](http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/index.html) and is available for every Sun/Oracle JVM 1.6 and later.

3.4.1. Jolokia as JVM Agent

The JVM agent uses the [JVM Agent interface](http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/index.html) for linking into any JVM. Under the hood it uses an HTTP-Server, which is available on every Oracle/Sun JVM from version 1.6 upwards.

The JDK embedded HTTP-Server is not the fastest one (it is used e.g. for the JAXWS reference implementation), but for our monitoring needs the performance is sufficient. There are several configuration options for tuning the HTTP server's performance. See below for details.

3.4.1.1. Installation

This agent gets installed by providing a single startup option -javaagent when starting the Java process.

java -javaagent:agent.jar=port=7777,host=localhost

agent.jar is the filename of the Jolokia JVM agent. The agent can be downloaded like the others from the [download page](https://jolokia.org/download.html). When downloading from a Maven repository you need to check for the classifier agent (i.e. the jar to download looks likejolokia-jvm-1.3.1-agent.jar, not jolokia-jvm-1.1.5.jar). Options can be appended as a comma separated list. The available options are the same as described in [Table 3.1, “Servlet init parameters”](https://jolokia.org/reference/html/agents.html#agent-war-init-params) plus the one described in table [Table 3.6, “JVM agent configuration options”](https://jolokia.org/reference/html/agents.html#agent-jvm-config). If an options contains a comma, an equal sign or a backslash, it must be escaped with a backslash.

**Table 3.6. JVM agent configuration options**

| **Parameter** | **Description** | **Example** |
| --- | --- | --- |
| agentContext | Context under which the agent is deployed. The full URL will be protocol://host:port/agentContext. The default context is /jolokia. | /j4p |
| agentId | A unique ID for this agent. By default a unique id is calculated. If provided it should be ensured that this id is unique among all agent reachable via multicast requests used by the discovery mechanism. It is recommended not to set this value. Within the agentId specification you can use the same placeholders as indiscoveryAgentUrl. | my-unique-agent-id |
| agentDescription | An optional description which can be used for clients to present a human readable label for this agent. | Intranet Timebooking Server |
| host | Hostaddress to which the HTTP server should bind to. If "\*" or "0.0.0.0" is given, the servers binds to every network interface. | localhost |
| port | Port the HTTP server should listen to. If set to 0, then an arbitrary free port will be selected. | 8778 |
| user | User to be used for authentication (along with apassword) |  |
| password | Password used for authentication (user is then required, too) |  |
| realm | Sets the security realm to use. If the authMode is set tojaas this is also used as value for the security domain. E.g. for Karaf 3 and later, this realm should be karafsince all JMX MBeans are guarded by this security domain. | jolokia |
| authMode | Can be either basic (the default), jaas or delegate. Ifjaas is used, the user and password given in theAuthorization: header are used for login in via JAAS and, if successful, the return subject is used for all Jolokia operation. This has only an effect, if user is set. For authentication mode delegate, the authentication decision is delegated to a service specified by authUrl(see below for details). | basic |
| authClass | Fully qualified name of an authenticator class. Class must be on classpath and must extendcom.sun.net.httpserver.Authenticator. Class can declare a constructor that takes one argument of a typeorg.jolokia.config.Configuration in which case Jolokia runtime configuration will be passed (useful in cases where authenticator requires additional configuration). If no such constructor is found, default (no-arg) constructor will be use to create an instance. |  |
| authUrl | URL of a service used for checking the authentication. This configuration option is only effective if authMode is set todelegate. This URL can have a HTTP or HTTPS scheme. The initially provided Authorization: header is copied over to the request against this URL. |  |
| authPrincipalSpec | Expression used for extracting a principal name from the response of a delegate authentication service. This parameter is only in use when the authMode is set todelegate. The following expressions are supported:  **json:path**  a path into a JSON response which points to the principal. E.g. a principal specjason:metadata/name will select the "name" property within the JSON object specified by the "metadata" property. For navigate into arrays, numeric indexes can be used.  **empty:**  Always extracts an empty ("") principal.  If this option is not specified, not principal is extracted. |  |
| authIgnoreCerts | If given, the authMode is set to delegate and the delegate URL is as HTTPS-URL then the server certificate as well as the server's DNS name will not be verified. This useful in order to avoid (or introduce) complex keymanagement issues, but is of course less secure. By default certs a verified with the local keystore. |  |
| protocol | HTTP protocol to use. Should be either http or https. For the SSL stack there are various additional configuration options. | http |
| backlog | Size of request backlog before requests get discarded. | 10 |
| executor | Threading model of the HTTP server:  **fixed**  Thread pool with a fixed number of threads (see alsothreadNr)  **cached**  Cached thread pool which creates threads on demand  **single**  A single thread only | single |
| threadNr | Number of threads to be used when the fixed execution model is chosen. | 5 |
| keystore | Path to the SSL keystore to use (https only) |  |
| keystorePassword | Keystore password (https only). If the password is given embedded in brackets [[...]], then it is treated as an encrypted password which was encrypted with java -jar jvm-agent.jar encrypt. See below for details. |  |
| useSslClientAuthentication | Whether client certificates should be used for authentication. The presented certificate is validated that it is signed by a known CA which must be in the keystore (https only). (true or false). | false |
| secureSocketProtocol | Secure protocol that will be used for establishing HTTPS connection (https only) | TLS |
| keyStoreType | SSL keystore type to use (https only) | JKS |
| keyManagerAlgorithm | Key manager algorithm (https only) | SunX509 |
| trustManagerAlgorithm | Trust manager algorithm (https only) | SunX509 |
| caCert | If HTTPs is to be used and no keystore is given, thencaCert can be used to point to a PEM encoded CA certification file. This is use to verify client certificates when useSslClientAuthentication is switched on (https only) |  |
| serverCert | For SSL (and when no keyStore is used) then this path must point to server certificate which is presented to clients (https only) |  |
| serverKey | Path to the PEM encoded key file for signing the server cert during TLS handshake. This is only used when nokeyStore is used. For decrypting the key the password given with keystorePassword is used (https only). |  |
| serverKeyAlgorithm | Encryption algorithm to use for decrypting the key given with serverKey (https only) | RSA |
| clientPrincipal | The principal which must be given in a client certificate to allow access to the agent. This can be one or or more relative distinguished names (RDN), separated by commas. The subject of a given client certificate must match on all configured RDNs. For example, when the configuration is "O=jolokia.org,OU=Dev" then a client certificate's subject must contain "O=jolokia.org" and "OU=Dev" to allow the request. Multiple alternative principals can be configured by using additional options with consecutive index suffix like in clientPrincipal.1,clientPrincipal.2, ... Please remember that a ,separating RDNs must be escaped with a backslash (\,) when used on the commandline as agent arguments. (https and useSslAuthentication only) |  |
| extraClientCheck | If switched on the agent performs an extra check for client authentication that the presented client cert contains a client flag in the extended key usage section which must be present. (https and useSslAuthentication only) |  |
| bootAmx | If set to true and if the agent is attached to a Glassfish server, then during startup the AMX subsystem is booted so that Glassfish specific MBeans are available. Otherwise, if set to false the AMX system is not booted. | true |
| config | Path to a properties file from where the configuration options should be read. Such a property file can contain the configuration options as described here as key value pairs (except for the config property of course :) |  |
| discoveryEnabled | Is set to false then this agent will not listen for multicast request (multicastgroup 239.192.48.84, port 24884). By default this option is enabled. | Default: true |
| discoveryAgentUrl | Sets the URL to respond for multicast discovery requests. If given, discoveryEnabled is set implicitly to true. Within the value you can use the placeholders ${host}and ${ip} which gets replaced by the autodetected local host name/address. Also with ${env:ENV\_VAR} and${sys:property} environment and system properties can be referenced, respectively. | http://10.9.11.87:8778/jolokia |
| sslProtocol | The list of SSL / TLS protocols enabled. Valid options are available in the documentation on SunJSSEProvider for your JDK version. Using only TLSv1.1 and TLSv1.2 is recommended in Java 1.7 and Java 1.8. Using onlyTLSv1 is recommended in Java 1.6. Multiple protocols can be configured by using additional options with consecutive index suffixes like in sslProtocol.1,sslProtocol.2, ... | TLSv1.2 |
| sslCipherSuite | The list of SSL / TLS cipher suites to enable. The table of available cipher suites is available under the "Default Enabled Cipher Suites" at the SunJSSEProvider documentation [here](http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunJSSEProvider). Multiple cipher suites can be configured by using additional options with consecutive index suffixes like in sslCipherSuite.1,sslCipherSuite.2, ... |  |

Upon successful startup the agent will print out a success message with the full URL which can be used by clients for contacting the agent.

3.4.2. Attaching a Jolokia agent on the fly

A Jolokia agent can be attached to any running Java process as long as the user has sufficient access privileges for accessing the process. This agent uses the [Java attach API](http://download.oracle.com/javase/6/docs/jdk/api/attach/spec/com/sun/tools/attach/VirtualMachine.html) for dynamically attaching and detaching to and from the process. It works similar to JConsole connecting to a local process. The Jolokia advantage is, that after the start of the agent, it can be reached over the network.

The JAR containing the JVM agent also contains a client application which can be reached via the -jar option. Call it with --help to get a short usage information:

$ java -jar jolokia-jvm-1.3.4-agent.jar --help

Jolokia Agent Launcher

======================

Usage: java -jar jolokia-jvm-1.3.4-agent.jar [options] <command> <pid/regexp>

where <command> is one of

start -- Start a Jolokia agent for the process specified

stop -- Stop a Jolokia agent for the process specified

status -- Show status of an (potentially) attached agent

toggle -- Toggle between start/stop (default when no command is given)

list -- List all attachable Java processes (default when no argument is given at all)

encrypt -- Encrypt a password which is given as argument or read from standard input

[options] are used for providing runtime information for attaching the agent:

--host <host> Hostname or IP address to which to bind on

(default: InetAddress.getLocalHost())

--port <port> Port to listen on (default: 8778)

--agentContext <context> HTTP Context under which the agent is reachable (default: /jolokia)

--agentId <agent-id> VM unique identifier used by this agent (default: autogenerated)

--agentDescription <desc> Agent description

--authMode <mode> Authentication mode: 'basic' (default), 'jaas' or 'delegate'

--authClass <class> Classname of an custom Authenticator which must be loadable from

the classpath

--authUrl <url> URL used for a dispatcher authentication (authMode == delegate)

--authPrincipalSpec <spec> Extractor specification for getting the principal

(authMode == delegate)

--authIgnoreCerts Whether to ignore CERTS when doing a dispatching authentication

(authMode == delegate)

--user <user> User used for Basic-Authentication

--password <password> Password used for Basic-Authentication

--quiet No output. "status" will exit with code 0 if the agent is running,

1 otherwise

--verbose Verbose output

--executor <executor> Executor policy for HTTP Threads to use (default: single)

"fixed" -- Thread pool with a fixed number of threads (default: 5)

"cached" -- Cached Thread Pool, creates threads on demand

"single" -- Single Thread

--threadNr <nr threads> Number of fixed threads if "fixed" is used as executor

--backlog <backlog> How many request to keep in the backlog (default: 10)

--protocol <http|https> Protocol which must be either "http" or "https" (default: http)

--keystore <keystore> Path to keystore (https only)

--keystorePassword <pwd> Password to the keystore (https only)

--useSslClientAuthentication Use client certificate authentication (https only)

--secureSocketProtocol <name> Secure protocol (https only, default: TLS)

--keyStoreType <name> Keystore type (https only, default: JKS)

--keyManagerAlgorithm <name> Key manager algorithm (https only, default: SunX509)

--trustManagerAlgorithm <name> Trust manager algorithm (https only, default: SunX509)

--caCert <path> Path to a PEM encoded CA cert file (https & sslClientAuth only)

--serverCert <path> Path to a PEM encoded server cert file (https only)

--serverKey <path> Path to a PEM encoded server key file (https only)

--serverKeyAlgorithm <algo> Algorithm to use for decrypting the server key (https only, default: RSA)

--clientPrincipal <principal> Allow only this principal in the client cert (https & sslClientAuth only)

If supplied multiple times, any one of the clientPrincipals must match

--extendedClientCheck <t|f> Additional validation of client certs for the proper key usage

(https & sslClientAuth only)

--discoveryEnabled <t|f> Enable/Disable discovery multicast responses (default: true)

--discoveryAgentUrl <url> The URL to use for answering discovery requests. Will be autodetected

if not given.

--sslProtocol <protocol> SSL / TLS protocol to enable, can be provided multiple times

--sslCipherSuite <suite> SSL / TLS cipher suite to enable, can be provided multiple times

--debug Switch on agent debugging

--debugMaxEntries <nr> Number of debug entries to keep in memory which can be fetched from the

Jolokia MBean

--maxDepth <depth> Maximum number of levels for serialization of beans

--maxCollectionSize <size> Maximum number of element in collections to keep when serializing the

response

--maxObjects <nr> Maximum number of objects to consider for serialization

--restrictorClass <class> Classname of an custom restrictor which must be loadable from the classpath

--policyLocation <url> Location of a Jolokia policy file

--mbeanQualifier <qualifier> Qualifier to use when registering Jolokia internal MBeans

--canonicalNaming <t|f> whether to use canonicalName for ObjectNames in 'list' or 'search'

(default: true)

--includeStackTrace <t|f> whether to include StackTraces for error messages (default: true)

--serializeException <t|f> whether to add a serialized version of the exception in the Jolokia

response (default: false)

--config <configfile> Path to a property file from where to read the configuration

--help This help documentation

--version Version of this agent (it's 1.3.4 btw :)

<pid/regexp> can be either a numeric process id or a regular expression. A regular expression is matched

against the processes' names (ignoring case) and must be specific enough to select exactly one process.

If no <command> is given but only a <pid> the state of the Agent will be toggled

between "start" and "stop"

If neither <command> nor <pid> is given, a list of Java processes along with their IDs

is printed

There are several possible reasons, why attaching to a process can fail:

\* The UID of this launcher must be the very \*same\* as the process to attach too. It not sufficient

to be root.

\* The JVM must have HotSpot enabled and be a JVM 1.6 or larger.

\* It must be a Java process ;-)

For more documentation please visit www.jolokia.org

Every option described in [Table 3.6, “JVM agent configuration options”](https://jolokia.org/reference/html/agents.html#agent-jvm-config) is reflected by a command line option for the launcher. Additionally, the option --quiet can be used to keep the launcher silent and --verbose for adding some extra logging.

The launcher knows various operational modes, which needs to be provided as a non-option argument and possibly require an extra argument.

**start**

Use this to attach an agent to an already running, local Java process. The additional argument is either the process id of the Java process to attach to or a regular expression which is matched against the Java processes names. In the later case, exactly one process must match, otherwise an exception is raised. The command will return with an return code of 0 if an agent has been started. If the agent is already running, nothing happens and the launcher returns with 1. The URL of the Agent will be printed to standard out on an extra line except when the --quiet option is used.

**stop**

Command for stopping an running and dynamically attached agent. The required argument is the Java process id or an regular expression as described for the start command. If the agent could be stopped, the launcher exits with 0, it exits with 1 if there was no agent running.

**toggle**

Starts or stops an dynamically attached agent, depending on its current state. The Java process ID is required as an additional argument. If an agent is running, toggle will stop it (and vice versa). The launcher returns with an exit code of 0 except when the operation fails. When the agent is started, the full agent's URL is printed to standard out. toggle is the default command when only a numeric process id is given as argument or a regular expression which not the same as a known command.

**status**

Command for showing the current agent status for a given process. The process id or a regular expression is required. The launcher will return with 0 when the agent is running, otherwise with 1.

**list**

List all local Java processes in a table with the process id and the description as columns. This is the default command if no non-option argument is given at all. list returns with 0 upon normal operation and with 1 otherwise.

**encrypt**

Encrypt the keystore password. You can add the password to encrypt as an additional argument or, if not given, it is read from standard input. The output of this command is the encrypted password in the format [[....]], which should be used literally (excluding the final newline) for the keystore password when using the option keystorePassword in the agent configuration.

The launcher is especially suited for one-shot, local queries. For example, a simple shell script for printing out the memory usage of a local Java process, including (temporarily) attaching an Jolokia agent looks simply like in the following example. With a complete client library like [Jmx4Perl](http://www.jmxp4perl.org/) even more one shot scripts are possible[[6](https://jolokia.org/reference/html/agents.html" \l "ftn.rest-comment)].

#!/bin/sh

url=`java -jar agent.jar start $1 | tail -1`

memory\_url="${url}read/java.lang:type=Memory/HeapMemoryUsage"

used=`wget -q -O - "${memory\_url}/used" | sed 's/^.\*"value":\([0-9]\*\).\*$/\1/'`

max=`wget -q -O - "${memory\_url}/max" | sed 's/^.\*"value":\([0-9]\*\).\*$/\1/'`

usage=$((${used}\*100/${max}))

echo "Memory Usage: $usage %"

java -jar agent.jar --quiet stop $1

[[1](https://jolokia.org/reference/html/agents.html" \l "d0e153)]Although the proxy mode is available for all four agents, you are normally free to setup the proxy environment. The recommendation here is the war-agent for which very lightweight servlet container exists. Tomcat or Jetty are both a perfect choice for a Jolokia proxy server.

[[2](https://jolokia.org/reference/html/agents.html" \l "d0e172)]Of course, there is much more to OSGi, a platform and programing model which I really like. This is my personal pet agent, so to speak ;-).

[[3](https://jolokia.org/reference/html/agents.html" \l "d0e193)][What is the proper plural form of "bus"?](http://answers.yahoo.com/question/index?qid=20071216082739AAfkxnm)

[[4](https://jolokia.org/reference/html/agents.html" \l "d0e209)]You could even instrument a JEE application server this way, however this is not recommended.

[[5](https://jolokia.org/reference/html/agents.html" \l "d0e858)]Replace org.jolokia.osgi.http.AgentServlet with org.jolokia.http.AgentServlet to use the servlet in a non-OSGi environment.

[[6](https://jolokia.org/reference/html/agents.html" \l "rest-comment)]And in fact, some support for launching this dynamic agent is planned for a forthcoming release of jmx4perl.

# Chapter 6. Jolokia Protocol

<https://jolokia.org/reference/html/protocol.html>

Jolokia uses a JSON-over-HTTP protocol which is described in this chapter. The communication is based on a request-response paradigm, where each request results in a single response.

**GET URLs are chatty**

Keep in mind that many web servers log the requested path of every request, including parameters passed to a GET request, so sending messages over GET often bloats server logs.

Jolokia requests can be sent in two ways: Either as a HTTP GET request, in which case the request parameters are encoded completely in the URL. Or as a POST request where the request is put into a JSON payload in the HTTP request's body. GET based requests are mostly suitable for simple use cases and for testing the agent via a browser. The focus here is on simplicity. POST based requests uses a JSON representation of the request within the HTTP body. They are more appropriate for complex requests and provide some additional features (e.g. bulk requests are only possible with POST).

The response returned by the agent uses always JSON for its data representation. It has the same format regardless whether GET or POST requests are used.

The rest of this chapter is divided into two parts: First, the general structure of requests and responses are explained after which the representation of Jolokia supported operations defined.

Unfortunately the term *operation* is used in different contexts which should be distinguished from one another. *Jolokia operations* denote the various kind of Jolokia requests, whereas *JMX operations* are methods which can be invoked on an JMX MBean. Whenever the context requires it, this documents uses *Jolokia* or *JMX* as prefix.

## 6.1. Requests and Responses

Jolokia knows about two different styles of handling requests, which are distinguished by the HTTP method used: GET or POST. Regardless of what method is used, the agent doesn't keep any state on the server side (except of course that MBeans are obviously stateful most of the time). So in this aspect, the communication can be considered [REST](http://en.wikipedia.org/wiki/Representational_State_Transfer) like[[7](https://jolokia.org/reference/html/protocol.html#ftn.rest-comment)].

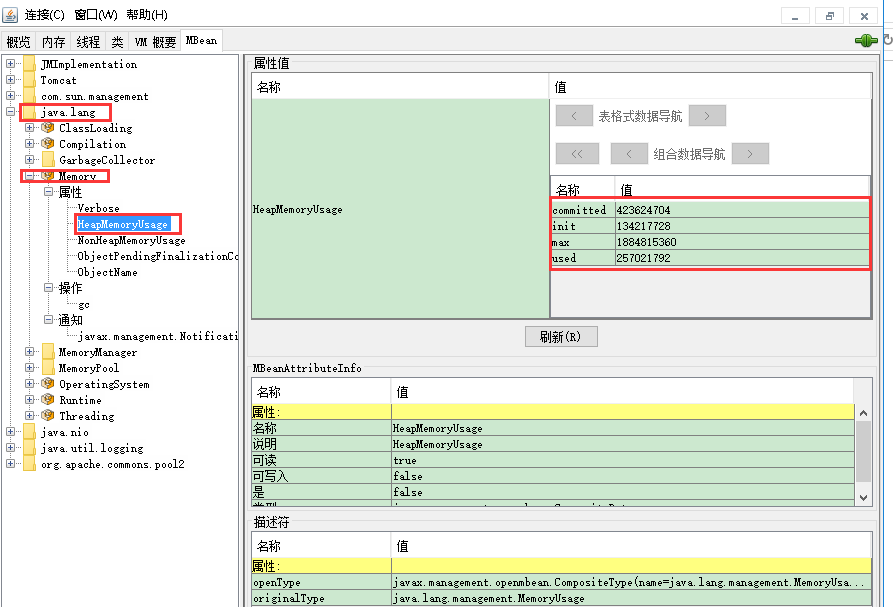
6.1.1. GET requests

The simplest way to access the Jolokia agent is by sending HTTP GET requests. These requests encode all their parameters within the access URL. Typically, Jolokia uses the path-info part of an URL to extract the parameters. Within the path-info, each part is separated by a slash (/). In general, the request URL looks like

<base-url>/<type>/<arg1>/<arg2>/..../

The <base-url> specifies the URL under which the agent is accessible. It normally looks like http://localhost:8080/jolokia, but depends on your deployment setup. The last part of this URL is the context root of the deployed agent, which by default is based on the agent's filename (e.g. jolokia.war). <type> specifies one of the supported Jolokia operations (described in the next section), followed by one or more operation-specific parameters separated by slashes.

Jconsole中提供MBean为java.lang:type=Memory



For example, the following URL executes a read Jolokia operation on the MBean java.lang:type=Memory for reading the attributeHeapMemoryUsage (see [Section 6.2.1, “Reading attributes (read)”](https://jolokia.org/reference/html/protocol.html#read)). It is assumed, that the agent is reachable under the base URLhttp://localhost:8080/jolokia:

http://localhost:7777/jolokia/read/java.lang:type=Memory/HeapMemoryUsage

http://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage

**Why escaping ?**

You might wonder why simple URI encoding isn't enough for escaping slashes. The reason is that JBoss/Tomcat has a strange behaviour when returning an HTTP responseHTTP/1.x 400 Invalid URI: noSlashfor any URL which contains an escaped slash in the path info (i.e.%2F). The reason behind this behaviour is security related, slashes get decoded on the agent side before the agent-servlet gets the request. Other appservers might exhibit a similar behaviour, so Jolokia uses an own escaping mechanism.

If one of the request parts contain a slash (/) (e.g. as part of you bean's name) it needs to be escaped. An exclamation mark (!) is used as escape character[[8](https://jolokia.org/reference/html/protocol.html" \l "ftn.no-backslash)]. An exclamation mark itself needs to be doubled for escaping. Any other characted preceded by an exclamation mark is taken literally. Table [Table 6.1, “Escaping rules”](https://jolokia.org/reference/html/protocol.html#escape-rules) illustrates the escape rules as used in GET requests. Also, if quotes are part of an GET request the need to be escaped with !".

**Table 6.1. Escaping rules**

| **Escaped** | **Unescaped** |
| --- | --- |
| !/ | / |
| !! | ! |
| !" | " |
| !(anything else) | (anything else) |

For example, to read the atrribute State on the MBean namedjboss.jmx:alias=jmx/rmi/RMIAdaptor, an access URL like this has to be constructed:

.../read/jboss.jmx:alias=jmx!/rmi!/RMIAdaptor/State

Client libraries like [JMX::Jmx4Perl](http://search.cpan.org/~roland/jmx4perl) do this sort of escaping transparently.

Escaping can be avoided alltogether if a slightly different variant for a request is used (which doesn't look that REST-stylish, though). Instead of providing the information as path-info, a query parameter p can be used instead. This should be URL encoded, though. For the example above, the alternative is

http://localhost:8080/jolokia?p=/read/jboss.jmx:alias=jmx%2Frmi%2FRMIAdaptor/State

This format must be used for GET requests containing backslashes (\) since backslashes can not be sent as part of an URL at all.

6.1.2. POST requests

POST requests are the most powerful way to communicate with the Jolokia agent. There are fewer escaping issues and it allows for features which are not available with GET requests. POST requests uses a fixed URL and put their payload within the HTTP request's body. This payload is represented in [JSON](http://www.json.org/), a data serialization format originating from the JavaScript world.

The JSON format for a single request is a JSON object, which is essentially a map with keys (or attributes) and values. All requests have a common mandatory attribute, type, which specifies the kind of JMX operation to perform. The other attributes are either operation specific as described in [Section 6.2, “Jolokia operations”](https://jolokia.org/reference/html/protocol.html#jolokia-operations) or are processing parameters which influence the overall behaviour and can be mixed in to any request. See [Section 6.3, “Processing parameters”](https://jolokia.org/reference/html/protocol.html#processing-parameters) for details. Operation specific attributes can be either mandatory or optional and depend on the operation type. In the following, if not mentioned otherwise, attributes are mandatory. Processing parameters are always optional, though.

A sample read request in JSON format looks like the following example. It has a type "read" (case doesn't matter) and the three attributes mbean, attribute and path which are specific to a read request.

**Example 6.1. JSON Request**

{

"type" : "read",

"mbean" : "java.lang:type=Memory",

"attribute" : "HeapMemoryUsage",

"path" : "used",

}

Each request JSON object results in a single JSON response object contained in the HTTP answer's body. A bulk request contains multiple Jolokia requests within a single HTTP request. This is done by putting individual Jolokia requests into a JSON array:

[

{

"type" : "read",

"attribute" : "HeapMemoryUsage",

"mbean" : "java.lang:type=Memory",

"path" : "used",

},

{

"type" : "search"

"mbean" : "\*:type=Memory,\*",

}

]

This request will result in a JSON array containing multiple JSON responses within the HTTP response. They are returned in same order as the requests in the initial bulk request.

6.1.3. Responses

Responses are always encoded in UTF-8 JSON, regardless whether the requst was a GET or POST request. In general, two kinds of responses can be classified: In the normal case, a HTTP Response with response code 200 is returned, containing the result of the operation as a JSON payload. In case of an error, a 4xx or 5xx code will be returned and the JSON payload contains details about the error occured (e.g. 404 means "not found"). (See [this page](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for more information about HTTP error codes..). If the processing optionifModifiedSince is given and the requested value as been not changed since then, a response code of 304 is returned. This option is currently only supported by the LIST request, for other request types the value is always fetched.

In the non-error case a JSON response looks mostly the same for each request type except for the value attribute which is request type specific.

The format of a single Jolokia response is

**Example 6.2. JSON Response**

{

"value": .... ,

"status" : 200,

"timestamp" : 1244839118,

"request": {

"type": ...,

....

},

"history":[

{"value": ... ,

"timestamp" : 1244839045

}, ....

]

}

For successful requests, the status is always 200 (the HTTP success code). The timestamp contains the epoch time[[9](https://jolokia.org/reference/html/protocol.html" \l "ftn.epoch-time)] when the request has been handled. The request leading to this response can be found under the attribute request. Finally and optionally, if history tracking is switched on (see [Section 6.5, “Tracking historical values”](https://jolokia.org/reference/html/protocol.html#history)), an entry with key history contains a list of historical values along with their timestamps. History tracking is only available for certain type of requests (read, write and exec). The valueis specific for the type of request, it can be a single scalar value or a monster JSON structure.

If an error occurs, the status will be a number different from 200. An error response looks like

{

"status":400,

"error\_type":"java.lang.IllegalArgumentException",

"error":"java.lang.IllegalArgumentException: Invalid request type 'java.lang:type=Memory'",

"stacktrace":"java.lang.IllegalArgumentException: Invalid request type 'java.lang:type=Memory'\n

\tat org.cpan.jmx4perl.JmxRequest.extractType(Unknown Source)\n

\tat org.cpan.jmx4perl.JmxRequest.<init>(Unknown Source) ...."

}

For status codes it is important to distinguish status codes as they appear in Jolokia JSON response objects and the HTTP status code of the (outer) HTTP response. There can be many Jolokia status codes, one for each Jolokia request contained in the single HTTP request. The HTTP status code merely reflect the status of agent itself (i.e. whether it could perform the operation at all), whereas the Jolokia response status reflects the result of the operation (e.g. whether the performed operation throws an exception). So it is not uncommon to have an HTTP status code of 200, but the contained JSON response(s) indicate some errors.

I.e. the status has a code in the range 400 .. 499 or 500 .. 599 [as it is specified for HTTP return codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes). The error member contains an error description. This is typically the message of an exception occured on the agent side[[10](https://jolokia.org/reference/html/protocol.html" \l "ftn.mbeanexception-wrapping)]. Finally, error\_type contains the Java class name of the exception occured. The stacktrace contains a Java stacktrace occured on the server side (if any stacktrace is available).

For each type of operation, the format of the value entry is explained in [Section 6.2, “Jolokia operations”](https://jolokia.org/reference/html/protocol.html#jolokia-operations)

6.1.4. Paths

An inner path points to a certain substructure (plain value, array, hash) within a a complex JSON value. Think of it as something like "XPath lite". This is best explained by an example:

The attribute HeapMemoryUsage of the MBean java.lang:type=Memory can be requested with the URLhttp://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage which returns a complex JSON structure like

{

"status" : 200,

"value" : {

"committed" : 18292736,

"used" : 15348352,

"max" : 532742144,

"init" : 0

},

"request" : { .... },

"timestamp" : ....

}

In order to get to the value for used heap memory you should specify an inner path used, so that the requesthttp://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/used results in a response of 15348352for the value:

{

"status" : 200,

"value" : 15348352,

"request" : { .... },

"timestamp" : ....

}

If the attribute contains arrays at some level, use a numeric index (0 based) as part of the inner path if you want to traverse into this array.

For both, GET and POST requests, paths must be escaped as described in [Table 6.1, “Escaping rules”](https://jolokia.org/reference/html/protocol.html#escape-rules) when they contain slashes (/) or exclamation marks (!).

Paths support wildcards \* in a simple form. If given as a path part exclusively, it matches any entry and path matching continues on the next level. This feature is especially useful when using pattern read request together with paths. See [Section 6.2.1, “Reading attributes (read)”](https://jolokia.org/reference/html/protocol.html#read) for details. A \* mixed with other characters in a path part has no special meaning and is used literally.

## 6.2. Jolokia operations

6.2.1. Reading attributes (read)

Reading MBean attributes is probably the most used JMX method, especially when it comes to monitoring. Concerning Jolokia, it is also the most powerful one with the richest semantics. Obviously the value of a single attribute can be fetched, but Jolokia supports also fetching of a list of given attributes on a single MBean or even on multiple MBeans matching a certain pattern.

Reading attributes are supported by both kinds of requests, GET and POST.

Don't confuse fetching multiple attributes on possibly multiple MBeans with bulk requests. A single read request will always result in a single read response, even when multiple attribute values are fetched. Only the single response's structure of the value will differ depending on what kind of read request was performed.

A read request for multiple attributes on the same MBean is initiated by giving a list of attributes to the request. For a POST request this is an JSON array, for a GET request it is a comma separated list of attribute names (where slashes and exclamation marks must be escaped as described in [Table 6.1, “Escaping rules”](https://jolokia.org/reference/html/protocol.html#escape-rules)). If no attribute is provided, then all attributes are fetched. The MBean name can be given as a pattern in which case the attributes are read on all matching MBeans. If a MBean pattern and multiple attributes are requested, then only the value of attributes which matches both are returned, the others are ignored.

Paths can be used with pattern and multiple attribute read as well. In order to skip the extra value levels introduced by a pattern read, the wildcard \* can be used. For example, a read request for the MBean Pattern java.lang:type=GarbageCollector,\* for the Attribute LastGcInfo returns a complex structure holding information about the last garbage collection. If one is interested only for the duration of the garbage collection, a path used could be used if this request wouldn't be a pattern request (i.e. refers a specific, single MBean). But in this case since a nested map with MBean and Attribute names is returned, the path \*/\*/used has to be used in order to skip the two extra levels for applying the path. The two levels are returned nevertheless, though. Note that in the following example the final value is not the full GC-Info but only the value of its used entry:

value: {

"java.lang:name=PS MarkSweep,type=GarbageCollector": {

LastGcInfo: null

},

"java.lang:name=PS Scavenge,type=GarbageCollector": {

LastGcInfo: 7

}

}

The following rule of thumb applies:

* If a wildcard is used, everything at that point in the path is matched. The next path parts are used to match from there on. All the values on this level are included.
* Every other path part is literally compared against the values on that level. If there is a match, this value is removed in the answer so that at the end you get back a structure with the values on the wildcard levels and the leaves of the matched parts.
* If used with wildcards, paths behave also like filters. E.g. you can use a path \*/\*/used on the MBean pattern java.lang:\* and get back only that portions which contains "used" as key, all others are ignored.

6.2.1.1. GET read request

The GET URL for a read request has the following format:

<base-url>/read/<mbean name>/<attribute name>/<inner path>

**Table 6.2. GET Read Request**

| **Part** | **Description** | **Example** |
| --- | --- | --- |
| <mbean name> | The [ObjectName](http://download.oracle.com/javase/6/docs/api/javax/management/ObjectName.html) of the MBean for which the attribute should be fetched. It contains two parts: A domain part and a list of properties which are separated by:. Properties themselves are combined in a comma separated list of key-value pairs. This name can be a pattern in which case multiple MBeans are queried for the attribute value. | java.lang:type=Memory |
| <attribute name> | Name of attribute to read. This can be a list of Attribute names separated by comma. Slashes and exclamations marks need to be escaped as described in[Table 6.1, “Escaping rules”](https://jolokia.org/reference/html/protocol.html#escape-rules). If no attribute is given, all attributes are read. | HeapMemoryUsage |
| <inner path> | This optional part describes an inner path as described in [Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths) | used |

With this URL the used heap memory can be obtained:

http://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/used

6.2.1.2. POST read request

A the keys available for read POST requests are shown in the following table.

**Table 6.3. POST Read Request**

| **Key** | **Description** | **Example** |
| --- | --- | --- |
| type | **read** |  |
| mbean | MBean's ObjectName which can be a pattern | java.lang:type=Memory |
| attribute | Attribute name to read or a JSON array containing a list of attributes to read. No attribute is given, then all attributes are read. | HeapMemoryUsage,[ "HeapMemoryUsage", "NonHeapMemoryUsage" ] |
| path | Inner path for accessing the value of a complex value ([Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths)) | used |

The following request fetches the number of active threads:

{

"type":"read",

"mbean":"java.lang:type=Threading",

"attribute":"ThreadCount"

}

6.2.1.3. Read response

The general format of the JSON response is described in [Section 6.1.3, “Responses”](https://jolokia.org/reference/html/protocol.html#responses) in detail. A typical response for an attribute read operation for an URL like

http://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/

looks like

{

"value":{

"init":134217728,

"max":532742144,

"committed":133365760,

"used":19046472

},

"status":200,

"timestamp":1244839118,

"request":{

"mbean":"java.lang:type=Memory",

"type":"read",

"attribute":"HeapMemoryUsage"

},

"history":[{"value":{

"init":134217728,

"max":532742144,

"committed":133365760,

"used":18958208

},

"timestamp":1244839045

}, ....

]

}

The value contains the response's value. For simple data types it is a scalar value, more complex types are serialized into a JSON object. See [Section 6.4, “Object serialization”](https://jolokia.org/reference/html/protocol.html#serialization) for detail on object serialization.

For a read request to a single MBean with multiple attributes, the returned value is a JSON object with the attribute names as keys and their values as values. For example a request to http://localhost:8080/jolokia/read/java.lang:type=Memory leads to

{

"timestamp": 1317151518,

"status": 200,

"request": {"mbean":"java.lang:type=Memory","type":"read"},

"value":{

"Verbose": false,

"ObjectPendingFinalizationCount": 0,

"NonHeapMemoryUsage": {"max":136314880,"committed":26771456,"init":24317952,"used":15211720},

"HeapMemoryUsage": {"max":129957888,"committed":129957888,"init":0,"used":2880008}

}

}

A request to a MBean pattern returns as value a JSON object, with the MBean names as keys and as value another JSON object with the attribute name as keys and the attribute values as values. For example a requesthttp://localhost:8080/jolokia/read/java.lang:type=\*/HeapMemoryUsage returns something like

{

"timestamp": 1317151980,

"status": 200,

"request": {"mbean":"java.lang:type=\*","attribute":"HeapMemoryUsage","type":"read"},

"value": {

"java.lang:type=Memory": {

"HeapMemoryUsage": {"max":129957888,"committed":129957888,"init":0,"used":3080912}

}

}

}

6.2.2. Writing attributes (write)

Writing an attribute is quite similar to reading one, except that the request takes an additional value element.

6.2.2.1. GET write request

Writing an attribute wit an GET request, an URL with the following format has to be used:

<base url>/write/<mbean name>/<attribute name>/<value>/<inner path>

**Table 6.4. GET Write Request**

| **Part** | **Description** | **Example** |
| --- | --- | --- |
| <mbean name> | MBean's ObjectName | java.lang:type=ClassLoading |
| <attribute name> | Name of attribute to set | Verbose |
| <value> | The attribute name to value. The value must be serializable as described in [Section 6.4.2, “Request parameter serialization”](https://jolokia.org/reference/html/protocol.html#serialization-request). | true |
| <path> | Inner path for accessing the parent object on which to set the value. (See also [Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths)). Note, that this is not the path to the attribute itself, but to the object carrying this attribute. With a given path it is possible to deeply set an value on a complex object. |  |

For example, you can set the garbage collector to verbose mode by using something like

http://localhost:8080/jolokia/write/java.lang:type=Memory/Verbose/true

6.2.2.2. POST write request

The keys which are evaluated for a POST write request are:

**Table 6.5. POST Write Request**

| **Key** | **Description** | **Example** |
| --- | --- | --- |
| type | **write** |  |
| mbean | MBean's ObjectName | java.lang:type=ClassLoading |
| attribute | Name of attribute to set | Verbose |
| value | The attribute name to value. The value must be serializable as described in[Section 6.4.2, “Request parameter serialization”](https://jolokia.org/reference/html/protocol.html#serialization-request). | true |
| path | An optional inner path for specifying an inner object on which to set the value. See[Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths) for more on inner paths. |  |

6.2.2.3. Write response

As response for a write operation the old attribute's value is returned. For a request

http://localhost:8080/jolokia/write/java.lang:type=ClassLoading/Verbose/true

you get the answer (supposed that verbose mode was switched off for class loading at the time this request was sent)

{

"value":"false",

"status":200,

"request": {

"mbean":"java.lang:type=ClassLoading",

"type":"write",

"attribute":"Verbose",

"value":true

}

}

The response is quite similar to the read operation except for the additional value element in the request (and of course, the differenttype).

6.2.3. Executing JMX operations (exec)

Beside attribute provides a way for the execution of exposed JMX operations with optional arguments. The same as for writing attributes, Jolokia must be able to serialize the arguments. See [Section 6.4, “Object serialization”](https://jolokia.org/reference/html/protocol.html#serialization) for details. Execution of overloaded methods is supported. The JMX specifications recommends to avoid overloaded methods when exposing them via JMX, though.

6.2.3.1. GET exec request

The format of an GET exec request is

<base url>/exec/<mbean name>/<operation name>/<arg1>/<arg2>/....

**Table 6.6. GET Exec Request**

| **Part** | **Description** | **Example** |
| --- | --- | --- |
| <mbean name> | MBean's ObjectName | java.lang:type=Threading |
| <operation name> | Name of the operation to execute. If this is an overloaded method, it is mandatory to provide a method signature as well. A signature consist the fully qualified argument class names or native types, separated by commas and enclosed with parentheses. For calling a non-argument overloaded method use () as signature. | loadUsers(java.lang.String,int) |
| <arg1>, <arg2>, ... | String representation for the arguments required to execute this operation. Only certain data types can be used here as described in[Section 6.4.2, “Request parameter serialization”](https://jolokia.org/reference/html/protocol.html#serialization-request). | "true","true" |

The following request will trigger a garbage collection:

http://localhost:8080/jolokia/exec/java.lang:type=Memory/gc

6.2.3.2. POST exec request

**Table 6.7. POST Exec Request**

| **Key** | **Description** | **Example** |
| --- | --- | --- |
| type | **exec** |  |
| mbean | MBean's ObjectName | java.lang:type=Threading |
| operation | The operation to execute, optionally with a signature as described above. | dumpAllThreads |
| arguments | An array of arguments for invoking this operation. The value must be serializable as described in [Section 6.4.2, “Request parameter serialization”](https://jolokia.org/reference/html/protocol.html#serialization-request). | [true,true] |

The following request dumps all threads (along with locked monitors and locked synchronizers, thats what the boolean arguments are for):

{

"type":"EXEC",

"mbean":"java.lang:type=Threading",

"operation":"dumpAllThreads",

"arguments":[true,true]

}

6.2.3.3. Exec response

For an exec operation, the response contains the return value of the operation. null is returned if either the operation returns a null value or the operation is declared as void. A typical response for an URL like

http://localhost:8080/jolokia/exec/java.util.logging:type=Logging/setLoggerLevel/global/INFO

looks like

{

"value":null,

"status":200,

"request": {

"type":"exec",

"mbean":"java.util.logging:type=Logging",

"operation":"setLoggerLevel",

"arguments":["global","INFO"]

}

}

The return value get serialized as described in [Section 6.4, “Object serialization”](https://jolokia.org/reference/html/protocol.html#serialization).

6.2.4. Searching MBeans (search)

With the Jolokia search operation the agent can be queried for MBeans with a given pattern. Searching will be performed on everyMBeanServer found by the agent.

6.2.4.1. GET search request

The format of the search GET URL is:

<base-url>/search/<pattern>

This mode is used to query for certain MBean. It takes a single argument pattern for specifying the search parameter like in

http://localhost:8080/jolokia/search/\*:j2eeType=J2EEServer,\*

You can use patterns as described [here](http://java.sun.com/j2se/1.5.0/docs/api/javax/management/ObjectName.html), i.e. it may contain wildcards like \* and ?. The Mbean names matching the query are returned as a list within the response.

6.2.4.2. POST search request

A search POST request knows the following keys:

**Table 6.8. POST Search Request**

| **Key** | **Description** | **Example** |
| --- | --- | --- |
| type | **search** |  |
| mbean | The MBean pattern to search for | java.lang:\* |

The following request searches for all MBeans registered in the domain java.lang

{

"type":"SEARCH",

"mbean":"java.lang:\*"

}

6.2.4.3. Search response

The answer is a list of MBean names which matches the pattern or an empty list if there was no match.

For example, the request

http://localhost:8888/jolokia/search/\*:j2eeType=J2EEServer,\*

results in

{

"value": [

"jboss.management.local:j2eeType=J2EEServer,name=Local"

],

"status":200,

"timestamp":1245305648,

"request": {

"mbean":"\*:j2eeType=J2EEServer,\*","type":"search"

}

}

The returned MBean names are properly [quoted](http://download.oracle.com/javase/1.5.0/docs/api/javax/management/ObjectName.html) so that they can be directly used as input for other requests.

6.2.5. Listing MBeans (list)

The list operation collects information about accessible MBeans. This information includes the MBean names, their attributes, operations and notifications along with type information and description (as far as they are provided by the MBean author which doesn't seem to be often the case).

6.2.5.1. GET list request

The GET request format for a Jolokia list request is

<base-url>/list/<inner path>

The <inner path>, as described in [Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths) specifies a subset of the complete response. You can use this to select a specific domain, MBean or attribute/operation. See the next section for the format of the complete response.

6.2.5.2. POST list request

A list POST request has the following keys:

**Table 6.9. POST list Request**

| **Key** | **Description** | **Example** |
| --- | --- | --- |
| type | **list** |  |
| path | Inner path for accessing the value of a subset of the complete list ([Section 6.1.4, “Paths”](https://jolokia.org/reference/html/protocol.html#paths)). | java.lang/type=Memory/attr |

The following request fetches the information about the MBean java.lang:type=Memory

{

"type":"LIST",

"path":"java.lang/type=Memory"

}

6.2.5.3. List response

The value has the following format:

{

<domain> :

{

<prop list> :

{

"attr" :

{

<attr name> :

{

"type" : <attribute type>,

"desc" : <textual description of attribute>,

"rw" : true/false

},

....

},

"op" :

{

<operation name> :

{

"args" : [

{

"type" : <argument type>

"name" : <argument name>

"desc" : <textual description of argument>

},

.....

],

"ret" : <return type>,

"desc" : <textual description of operation>

},

.....

},

"not" :

{

"name" : <name>,

"desc" : <desc>,

"types" : [ <type1>, <type2> ]

}

},

....

},

....

}

The domain name and the property list together uniquely identify a single MBean. The property list is in the so called canonical order, i.e. in the form "<key1>=<val1>,<key2>=<val2>,.." where the keys are ordered alphabetically. Each MBean has zero or more attributes and operations which can be reached in an MBeans JSON object with the keys attr and op respectively. Within these groups the contained information is explained above in the schema and consist of Java types for attributes, arguments and return values, descriptive information and whether an attribute is writable (rw == true) or read-only.

As for reading attributes you can fetch a subset of this information using an path. E.g a path of domain/prop-list would return the value for a single bean only. For example, a request

http://localhost:8080/jolokia/list/java.lang/type=Memory

results in an answer

{

"value":

{

"op":

{

"gc":

{

"args":[],

"ret":"void",

"desc":"gc"

}

},

"class":"sun.management.MemoryImpl",

"attr":

{

"NonHeapMemoryUsage":

{

"type":"javax.management.openmbean.CompositeData",

"rw":false,

"desc":"NonHeapMemoryUsage"

},

"Verbose":

{

"type":"boolean",

"rw":true,

"desc":"Verbose"

},

"HeapMemoryUsage":

{

"type":"javax.management.openmbean.CompositeData",

"rw":false,

"desc":"HeapMemoryUsage"

},

"ObjectPendingFinalizationCount":

{

"type":"int",

"rw":false,

"desc":"ObjectPendingFinalizationCount"

}

}

},

"status":200,

"request":

{

"type":"list",

"path":"java.lang\/type=Memory"

}

}

6.2.5.4. Restrict depth of the returned tree

The optional parameter maxDepth can be used to restrict the depth of the return tree. Two value are possible: A maxDepth of 1 restricts the return value to a map with the JMX domains as keys, a maxDepth of 2 truncates the map returned to the domain names (first level) and the MBean's properties (second level). The final values of the maps don't have any meaning and are dummy values.

6.2.6. Getting the agent version (version)

The Jolokia command version returns the version of the Jolokia agent along with the protocol version.

6.2.6.1. GET version request

The GET URL for a version request has the following format:

<base-url>/version

For GET request the version part can be omitted since this is the default command if no command is provided as path info.

6.2.6.2. POST version request

A version POST request has only a single key type which has to be set to **version**.

6.2.6.3. Version response

The response value for a version request looks like:

{

"timestamp":1287143106,

"status":200,

"request":{"type":"version"},

"value":{

"protocol":"7.1",

"agent":"1.2.0",

"config": {

"agentDescription": "Servicemix ESB",

"agentId": "EF87BE-jvm",

"agentType": "jvm",

"serializeException": "false"

},

"info": {

"product": "glassfish",

"vendor": "Oracle",

"version": "4.0",

"extraInfo": {

"amxBooted": false

},

}

}

protocol in the response value contains the protocol version used, agent is the version of the Jolokia agent. See [Section 6.8, “Jolokia protocol versions”](https://jolokia.org/reference/html/protocol.html#versions) for the various protocol versions and the interoperability. If the agent is able to detect the server, additional meta information about this server is returned (i.e. the product name, the vendor and optionally some extra information added by the server detector).

# Jolokia – Jolokia Cubism Demo

https://jolokia.org/client/javascript-cubism.html

[Cubism](http://square.github.com/cubism/) is a fine Javascript library for plotting timeseries data based on [d3.js](http://d3js.org/). It provides support for various backend sources like [Graphite](http://graphite.wikidot.com/) or [Cube](http://square.github.com/cube/) and also for Jolokia. It is easy to use and provides innovative chart types like a [horizon](http://vis.berkeley.edu/papers/horizon/) chart. The Jolokia integration polls the Jolokia agent periodically and remembers the values locally. It uses the [scheduling facility](https://jolokia.org/reference/html/clients.html#js-poller) of the Jolokia Javascript library by sending a single bulk request for fetching the data for all charts and is hence very efficient.

The following sections show some simple real time demos of this integration. After that, some concepts are explained. The full Javascript source can be downloaded [here](https://jolokia.org/js/javascript-cubism.js).

**HeapMemory**

The following demo directly queries Jolokia's [CI](http://labs.consol.de/jenkins) which is a plain Tomcat 7. The memory charts show the heap memory usage as a fraction of the maximum available heap. Note that different colors indicate different value ranges in this horizon chart. The activity of the two garbage collectors for the young and old generation are shown below. Feel free to trigger a garbage collection on your own by pressing the button and look how the chart is changing.

03:0203:0303:0403:0503:0603:0703:0803:0903:1003:11

Heap-Memory

GC Young

GC Old

Trigger Garbage Collection

**Requests (per 10 seconds)**

The second demo visualizes the number of requests served by this Tomcat instance. The requests are grouped by 10s, so the values are the number of requests received in the last 10 seconds. The green charts show the requests for the [Jolokia agent](http://jolokia.org/jolokia) and the [Jenkins CI server](http://labs.consol.de/jenkins). Since this demo queries the Jolokia agent every second, the first chart should show up at least 10 request per 10 seconds. Finally the number of requests served by all deployed servlets is drawn in blue.

03:0203:0303:0403:0503:0603:0703:0803:0903:1003:11

Jolokia

Jenkins

All

**Examples**

Plotting the result of a single Jolokia request is simple and follows the general pattern used by Cubism. You first create a Jolokia source from the Cubism context and create metrics from this source. When a metric is created, it registers one or more Jolokia request for the [Jolokia scheduler](https://jolokia.org/reference/html/clients.html#js-poller)

  // Create a top-level Cubism Context  
  var context = cubism.context();  
    
  // Create a source for Jolokia metrics pointing to the agent   
  // at 'http://jolokia.org/jolokia'  
  var jolokia = context.jolokia("http://jolokia.org/jolokia");  
  
  // Create a metric for the absolute Heap memory usage  
  var memoryAbs = jolokia.metric({  
                      type: 'read',   
                      mbean: 'java.lang:type=Memory',  
                      attribute: 'HeapMemoryUsage',  
                      path: 'used'  
                  },"HeapMemory Usage");  
   
   // Use d3 to attach the metrics with a specific graph type   
   // ('horizon' in this case) to the document  
   d3.select("#charts").call(function(div) {  
       div.append("div")  
           .data([memoryAbs])  
           .call(context.horizon())  
   });

The following example present an advanced concept if more flexibility is required. When the first argument to jolokia.metric() is a function, this function is feed periodically with Jolokia response objects resulting from the requests object given as second argument. The final argument can be an options object, which in this case indicates the label of the chart and the type to be a delta chart, measuring only the increase rate for ten seconds.

This sample also shows how to use wildcard patterns in a read request to fetch multiple values at once in a generic fashion. Wildcard reading is explained in detail in the [reference manual](https://jolokia.org/reference/html/protocol.html#read).

  var allRequestsMetric = jolokia.metric(  
    function (resp) {  
        var attrs = resp.value;  
        var sum = 0;  
        for (var key in attrs) {  
            sum += attrs[key].requestCount;  
        }  
        return sum;  
    },  
    {   
       type: "read",   
       mbean: "Catalina:j2eeType=Servlet,\*",  
       attribute: "requestCount"  
    },   
    {  
       name: "All Requests",   
       delta: 10 \* 1000  
    });

# Jolokia JVM Monitoring in OpenShift – RHD Blog

https://developers.redhat.com/blog/2016/03/30/jolokia-jvm-monitoring-in-openshift/

*Posted by*[*Andrew Block*](https://developers.redhat.com/blog/author/sabre1041/)*on March 30, 2016*

Cloud based technology offers the ability to build, deploy and scale applications with ease; however, deploying to the cloud is only half of the battle. How cloud applications are monitored becomes a paramount concern with operations teams.

When issues arise, teams and their monitoring systems must be able to detect, react, and rectify the situation. CPU, system memory, and disk space are three common indicators used to monitor applications, and are typically reported by the operating system.

However, for Java applications – which we’ll be focusing on in this article – most solutions tap into JMX (Java Monitoring eXtensions) to monitor the Java Virtual Machine (JVM). For applications leveraging Java [xPaaS middleware services](https://www.openshift.com/enterprise/middleware-services.html) on OpenShift, they have built-in functionality to provide the capabilities to monitor and manage their operation.

Communicating with applications

Communication with applications – or in the case of xPaaS, an application server – is typically performed using JMX technology, and is generally accomplished using RMI (Remote Method Invocation). RMI technology is not well suited for cloud based technologies, neither is it suited for microservices architectures that focus on lightweight communication over HTTP; therefore, OpenShift xPaaS applications expose JMX operations through an HTTP bridge provided by the [Jolokia](https://jolokia.org/) project, where simple REST based methods and JSON based payloads provide a simplified and lightweight approach to remote application monitoring and management.

The Jolokia package is embedded within each xPaaS Docker image as a JVM agent for instrumenting the running application server. Clients can communicate with the agent using port 8778, which is exposed by default from the docker image, and can be seen by inspecting the JBoss EAP xPaaS image using the following command:

**docker inspect registry.access.redhat.com/jboss-eap-6/eap64-openshift**

In the response, the list of exposed ports can be seen as shown below:

**...**

**"ExposedPorts": {**

**"8080/tcp": {},**

**"8443/tcp": {},**

**"8778/tcp": {}**

**},**

**...**

In order to provide load balanced communication between multiple containers on OpenShift, exposed ports are typically mapped as service entries and can be further exposed through a route to provide access outside of the cluster. Most [templates](https://docs.openshift.com/enterprise/latest/architecture/core_concepts/templates.html) used with JBoss EAP only expose ports 8080 and 8443 as services to serve HTTP/HTTPS traffic.

Since Jolokia provides a method to manage and monitor a single JVM, it would be impractical to load balance across multiple containers with the goal of targeting a single instance, and without a service and a corresponding route it would typically be impossible to access a resource that is only exposed to the internal pod network.

OpenShift, however, provides the ability to proxy through the API server in order to access a pod inside the cluster (via underlying Kubernetes). The API server is the lifeblood of OpenShift. Everything – from nodes bringing themselves into the desired state to conform to [etcd](https://docs.openshift.com/enterprise/latest/architecture/infrastructure_components/kubernetes_infrastructure.html), to clients communicating with either the web console, or the command line tool – leverages the API server via RESTful invocations.

Using this same method, access to (typically non-exposed) internal resources can be obtained through application routes.  The following diagram demonstrates the paths and components involved with proxying traffic to the pod network through the API server:

As the diagram depicts (on the left hand side), traffic signified as Jolokia Proxy Traffic path will be the focus of the upcoming discussion.

The right hand side of the diagram on the other hand, is the typical flow for application traffic – destined for externally exposed HTTP/HTTPS pod resources. This traffic is initially served by an integrated HAProxy router listening on ports 80 and 443. A service lookup is performed to determine an available endpoint on the pod network signifying a listening application container. The request is then finally routed to the address and port on the pod network. The majority of the application transport process is abstracted from the end user.

For traffic destined for Jolokia via the API proxy (left hand side), the flow begins by initiating a request to the OpenShift API on the master instance using port 8443. Based on input parameters, the request sent from the API to the pod network and ultimately to the destination pod and port. Let’s walk through each of these approaches.

Spinning up an example on OpenShift

To demonstrate accessing resources through the application traffic path and then finally using the API proxy path (as depicted in the diagram above), we’ll implement the following via a provided application example template:

1. A deployment of the [JBoss Ticket Monster application](http://www.jboss.org/ticket-monster/)
2. A JBoss EAP 6.4 image for the running container
3. A Jolokia configuration

The following two OpenShift CLI commands can be executed to spin up this environment:

**oc new-project jolokia**

**oc new-app --template=eap64-basic-s2i -p=APPLICATION\_NAME=ticketmonster,SOURCE\_REPOSITORY\_URL=https://github.com/jboss-developer/ticket-monster,SOURCE\_REPOSITORY\_REF=2.7.0.Final,CONTEXT\_DIR=demo**

The template automatically creates the necessary objects in OpenShift, triggers an application build, and deploys the resulting image. Also included is the creation of a route for application traffic from resources outside the OpenShift cluster. This can all be validated by logging into the OpenShift web console at https://master\_host:8443 and select the Jolokia project.

OpenShift automatically generates a “master host” name based on application name, project name and default subdomain – this can be seen at the top of the page. Select the URL to open the Ticket Monster application in the browser; a successful response validates the application traffic path.

With the application traffic path functional, focus can be shifted toward the true goal of obtaining information about the JVM resources from Jolokia through the API proxy.

The first step is to target the OpenShift REST API that is exposed on the master OpenShift instance athttp://master\_host:8443/api. The next step is to determine and obtain the necessary parameters required by the proxy resource including the following:

1. Project (namespace) name
2. Pod name
3. Port exposed on the pod
4. Whether communication will be facilitated over https

Fortunately, these can be determined easily given the deployed application:

1. The application is deployed in the project named “jolokia”.
2. To obtain the name of the running pod containing the application, run the `oc get pods` OpenShift CLI command (Example result: ticketmonster-1-op5j0)
3. As discussed previously, Jolokia exposes port 8778 for Java xPaaS images.
4. Finally, Jolokia does utilize https communication, but the implementation is abstracted from the end user as it is only applicable for communication between API proxy and Jolokia. The end user needs to be concerned only with communicating to the API.

For pod based API proxy requests, they take on the following URL format:

**https://<master\_host>:8443/api/v1/namespaces/<project\_name>/pods/<secure\_scheme>:<pod\_name>:<pod\_port>/proxy**

So for an application running in a pod called ticketmonster-1-op5j0 in the jolokia project, the URL to communicate with the Jolokia port would be formatted as:

**https://<master\_host>:8443/api/v1/namespaces/jolokia/pods/https:ticketmonster-1-op5j0:8778/proxy**

This URL provides access to the resources listening on the port in the pod. However, Jolokia exposes itself on the /jolokia context path, so it would need to be added to the end of the URL as shown below:

**https://<master\_host>:8443/api/v1/namespaces/<project\_name>/pods/https:<pod\_name>:8778/proxy/jolokia**

The final step that is needed prior to invoking the API is to add the OAuth token for authentication. It can be obtained from the OpenShift CLI by running the <em>oc whoami –t command.

Now combine each of the data points into the following request:

**curl -k -H "Authorization: Bearer <api\_token>" https://<master\_host>:8443/api/v1/namespaces/<projcet\_name>/pods/https:<pod\_name>:8778/proxy/jolokia/**

Note: The –k flag is passed into the curl command as this specific OpenShift environment is using self signed certificates.

The request results in the following response:

**{"request":{"type":"version"},"value":{"agent":"1.3.2","protocol":"7.2","config":{"maxDepth":"15","discoveryEnabled":"false","maxCollectionSize":"0","agentId":"10.1.0.8-154-5cad8086-jvm","debug":"false","agentType":"jvm","historyMaxEntries":"10","agentContext":"\/jolokia","maxObjects":"0","debugMaxEntries":"100"},"info":{"product":"jboss","vendor":"RedHat","version":"7.5.4.Final-redhat-4"}},"timestamp":1458702143,"status":200}**

To access useful JVM metrics, such as the amount of memory usage, the following URL can be used.

**curl -k -H "Authorization: Bearer <api\_token>" https://<master\_host>:8443/api/v1/namespaces/<project\_name>/pods/https:<pod\_name>:8778/proxy/jolokia/read/java.lang:type=Memory/HeapMemoryUsage**

Which results in the following response:

**{"request":{"mbean":"java.lang:type=Memory","attribute":"HeapMemoryUsage","type":"read"},"value":{"init":1367343104,"committed":1364721664,"max":1364721664,"used":167167296},"timestamp":1458703005,"status":200}**

The full set of requests and responses that can be sent to Jolokia can be found in the [Jolokia product documentation](https://jolokia.org/documentation.html).

JVM Monitoring in Action

Given the ability to instantly access JVM metrics from running containers within OpenShift, the floodgates of opportunity are open to all the ways this data can be used. Everything from monitoring and reporting to alerting is on the table.

While it has been demonstrated that resources from Jolokia can be queried, a visual example always provides substance for these types of concepts. An html/javascript application has been developed to provide a demonstration of querying and displaying metrics obtained from Jolokia exposed containers running on OpenShift.

The code for the application is found on GitHub and can be cloned to a local machine by running the following command:

**git clone https://github.com/sabre1041/ose-jolokia-demo**

Communication in the demo between the client and the OpenShift API is facilitated using a combination of jQuery and the Jolokia JavaScript client. Since the application is running locally and not hosted on the same instance as the API, Cross-Origin Resource Sharing (CORS) will come into play and deny requests by default. OpenShift can be configured to effectively disable the restrictions imposed by CORS. To disable CORS restrictions, edit the OpenShift master configuration file located at/etc/origin/master/master-config.yaml. You’ll need to add <strong>- .\* in new line under the**corsAllowedOrigins** section as follows:

**corsAllowedOrigins:**

**- 10.0.2.15:8443**

**- 127.0.0.1**

**- localhost**

**- .\***

Once you’ve updatead the file, you’ll need to restart the OpenShift master:

**systemctl restart atomic-openshift-master**

Now that OpenShift is properly configured, navigate to the location containing the cloned project resources. In this directory, locate and open the index.html file (in a web browser). This file contains the application, which will attempt to communicate with all pods in a given project that expose Jolokia resources. Based on the located resources, it will display graphs of the current memory consumption, thread count, and HTTP web requests.

On the webpage, three textboxes are presented:

* The location of the OpenShift API
* The token used to communicate with the API
* The namespace (project) to search for resources.

Using the material previously retrieved, enter the information into the input textboxes and press submit. Data will be returned from the pods on a 5 second interval to populate the graphs.

In a separate browser tab or window, navigate once again to the ticketmonster application. Hit the refresh or the F5 key to simulate several requests. Return to the demo application to visualize the increase in the number of requests received.

Conclusion

By being able to query JVM metrics from running Java applications in OpenShift, doors are opened to the potential ways that applications can be monitored and managed in a cloud environment.

As we’ve seen in this article, Jolokia is a JMX-HTTP bridge giving an alternative to JSR-160 connectors, and it is useful for exposing JVM monitoring APIs via non-traditional protocols.

# 2@ctheu.com | All the things we can do with JMX

https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#jolokia-jmx-to-http

If you’re working with Java or Scala, you probably already heard of JMX or already using it. Most of us probably already used jconsole or jvisualvm to access the “JMX data”, to get some insights about the internals of a Java process. If you did not, you’re going to wonder why you never did. If you did, you may be interested by all the integrations we’re going to show.

This article is a tentative to explain globally what is JMX. What is its purpose? What can we do with it? Is it simple to use? What are the existing integrations we can use? What about its ecosystem? We’ll use a bunch of tools that are using it to make it clear.

**Summary**

* [What is JMX?](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#what-is-jmx)
  + [MBeans & Co](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#mbeans-amp-co)
  + [How to declare a custom MBean](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#how-to-declare-a-custom-mbean)
* [How to use JMX?](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#how-to-use-jmx)
  + [UI: JMX Client Connectors: jconsole, jvisualvm, jmc](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#ui-jmx-client-connectors-jconsole-jvisualvm-jmc)
  + [Programmatically](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#programmatically)
  + [Connect to a distant JMX Agent](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#connect-to-a-distant-jmx-agent)
* [Jolokia: JMX to HTTP](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#jolokia-jmx-to-http)
  + [A Java Agent](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#a-java-agent)
  + [Queries](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#queries)
* [Camel: stay awhile and listen](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#camel-stay-awhile-and-listen)
  + [Listen to MBeans modifications](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#listen-to-mbeans-modifications)
  + [Monitor Camel internals with JMX](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#monitor-camel-internals-with-jmx)
* [Kamon and JMX](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#kamon-and-jmx)
  + [Kamon's features](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#kamons-features)
  + [Exposing metrics to JMX](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#exposing-metrics-to-jmx)
  + [kamon-akka: Monitoring Akka's actors](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#kamon-akka-monitoring-akkas-actors)
* [JMXTrans: Send JMX metrics anywhere](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#jmxtrans-send-jmx-metrics-anywhere)
  + [Standalone](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#standalone)
  + [Queries](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#queries)
  + [Example: Kafka as source](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#example-kafka-as-source)
  + [JMXTrans as an Agent](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#jmxtrans-as-an-agent)
  + [The raw Graphite protocol: using nc and ngrep](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#the-raw-graphite-protocol-using-nc-and-ngrep)
* [The Swiss Java Knife: jvm-tools / sjk](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#the-swiss-java-knife-jvm-tools-sjk)
  + [mxdump: The whole JMX tree into JSON](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#mxdump-the-whole-jmx-tree-into-json)
  + [mx: query the MBeans](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#mx-query-the-mbeans)
* [Conclusion](https://www.ctheu.com/2017/02/14/all-the-things-we-can-do-with-jmx/#conclusion)

What is JMX?

It’s a standard originally from the [JSR 3: Java™ Management Extensions (JMX™) Specification](https://jcp.org/en/jsr/detail?id=003) (came with J2SE 5.0), that defines a way and an API to manage and expose resources (custom and of the JVM itself, called MBeans) in an application. It was later consolidated by the [JSR 160: Java™ Management Extensions (JMX) Remote API](https://jcp.org/en/jsr/detail?id=160) to handle JMX remote management (with RMI).

With JMX, we can retrieve or change some application resources values (MBeans attributes), or call methods on them, on the fly, to alter the behavior of the application and to monitor its internals.

We can use JMX for anything, the possibilities are quite infinite. For instance:

* Know the memory and CPU the application is using.
* Trigger the GC.
* How many requests were processed by the application?
* What are the database latency percentiles?
* How many elements are contained in the caches?
* Change the load-balancing strategy in real-time.
* Force an internal circuit-breaker to be open.
* …

It all depends on what the application is “offering”.

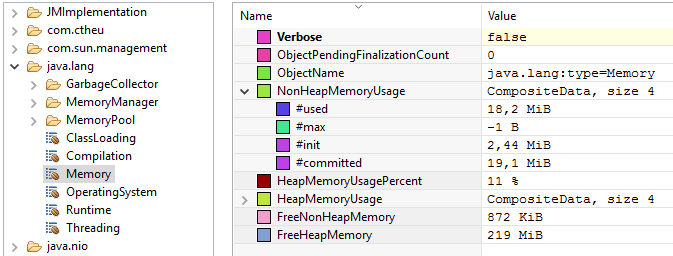
It’s almost like we had a reactive database inside the application and we were exposing HTTP REST services (GET, PUT) over it, without coding anything, without the hassle, and with standard request/response payloads anyone (exterior) can interact with.

MBeans & Co

All those values we talked about (that we can read or write) and methods we can call, must be contained inside MBeans.

* MBean stands for Managed Bean. It’s simply a Java Bean following some constraints (implements an interface xxxMBean and provides gets/sets).
* MBeans have the possibility to send notifications on changes but it’s not mandatory (they are often read by just polling them at a regular interval).
* A evolution are the [MXBeans](http://docs.oracle.com/javase/8/docs/api/javax/management/MXBean.html): they are MBeans that handle a pre-defined set of [Open Types](https://docs.oracle.com/javase/8/docs/api/javax/management/openmbean/OpenType.html) necessary for a better inter-operability.
* There are pre-existing platform MXBeans: the ones already packaged with the JRE that expose the JVM internals (memory, cpu, threads, system, classes).

Here is the platform MXBean java.lang:type=Memory attributes and values:



The values we see in the screenshot are the exact as we can get in the code with:

val mem = ManagementFactory.getMemoryMXBean

mem.setVerbose(true)

mem.getNonHeapMemoryUsage.getUsed *// 8688760*

*// we can call methods!*

mem.gc()

The platform MXBeans have static accessors in ManagementFactory (because they are Java standards) and they are strongly typed.

It’s also possible to grab them dynamically, as any other MBean. To do that, we need to build aObjectName, which is the path of the MBean:

*// MBeans belongs to a MBeans server as we say: the Java API can create one if we ask*

val server = ManagementFactory.getPlatformMBeanServer()

println(server.getMBeanCount())

*// 22*

val info = server.getMBeanInfo(ObjectName.getInstance("java.lang:type=Memory"))

info.getAttributes() *// MBeanAttributeInfo[]*

info.getOperations() *// MBeanOperationInfo[]*

How to declare a custom MBean

The MBeans must be declared by the application in a standard way.

* It has to implement an interface with the MBean suffix. (or MXBean if [Open Typed](https://docs.oracle.com/javase/8/docs/api/javax/management/openmbean/OpenType.html))
* It needs getters and/or setters (properties can be readonly). Because we are working in Scala, we can use @BeanProperty to generate them but they still need to be declared in the interface/trait.
* It can have methods with parameters.

trait MetricMBean {

def getValue(): Double

def setValue(d: Double): Unit

}

class Metric(@BeanProperty var value: Double) extends MetricMBean

Finally, to be accessible, we need to register an instance of it into a MBeans server.

* A MBeans server is the entity that manages the resources (MBeans), provides methods to register/unregister them, invoke methods on the MBeans and so on.
* A MBeans server is part of a JMX agent, which runs in the same JVM. The JMX Agent exposes a JMX server connector for JMX client connector to be able to connect to it (local or remote), list the MBeans, get the attributes values, invoke methods, and do whatever they want with them.

To register a MBean instance, we must provide an object name composed of a domain and key values pairs to form the path:

object JMX extends App {

val server = ManagementFactory.getPlatformMBeanServer()

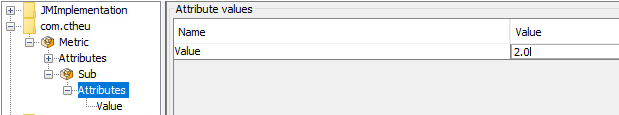
server.registerMBean(new Metric(1.0), ObjectName.getInstance("com.ctheu:type=Metric"))

server.registerMBean(new Metric(2.0), ObjectName.getInstance("com.ctheu:type=Metric,subtype=Sub"))

Thread.sleep(60000)

}

We can see our metrics in JConsole, and modify them:



The application itself can monitor this value or be notified to adapt its behavior. Other applications can do the same by connecting themselves to the application JMX Agent, through RMI.

How to use JMX?

UI: JMX Client Connectors: jconsole, jvisualvm, jmc

The Java JDK already embeds several JMX client connectors with more or less complex UIs, that provides more or less general features:

* [jconsole](http://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html): the simplest, the fastest.
* [Java VisualVM](http://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/index.html): the middle-ground, it has more options and handle plugins. It’s also on [GitHub](https://github.com/visualvm/visualvm.src).
* [Java Mission Control](https://docs.oracle.com/javacomponents/jmc-5-5/jmc-user-guide/toc.htm): part of the Oracle commercial features, the UI is more polished, it has a complete recorder feature that can really help to find problems source.

All three are packaged by default with the JDK installation and can connect to a local or remote JMX Agent (exposing a MBeans server).

Programmatically

Java exposes a client API in javax.management[.remote] to connect to any JMX agent through RMI and retrieve a MBeanServerConnection to request the MBeans attributes, their values, etc.

The connection scheme is quite ugly: service:jmx:rmi:///jndi/rmi://localhost:9010/jmxrmi but trust me, it works! The important part being localhost:9010. (here, 9010 is the RMI registry port I pick, we’ll see that just after)

Here is a program that output the whole MBeans hierarchy attributes and values, then calls some JMX methods:

*// The program was started with:*

*// -Dcom.sun.management.jmxremote.port=9010*

*// -Dcom.sun.management.jmxremote.authenticate=false*

*// -Dcom.sun.management.jmxremote.ssl=false*

object JMXTestConnection extends App {

*// we listen to our own JMX agent!*

val url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9010/jmxrmi")

val connector = JMXConnectorFactory.connect(url)

val server = connector.getMBeanServerConnection()

val all = server.queryMBeans(null, null).asScala

println(all.map(\_.getObjectName)

.map(name => s"$name\n" + attributes(name)))

*// we can also call the JMX methods: "gc", "change" (custom MBean):*

server.invoke(ObjectName.getInstance("java.lang:type=Memory"), "gc", null, null)

server.invoke(ObjectName.getInstance("com.ctheu:type=Metric"), "change", Array(new Integer(18)), null)

*// helpers*

private def attributes(name: ObjectName) = {

server.getMBeanInfo(name).getAttributes.toList.map(attribute(name, \_)).mkString("\n")

}

private def attribute(name: ObjectName, attr: MBeanAttributeInfo) = {

s"- ${attr.getName} (${attr.getType}) = ${attributeValue(name, attr)}"

}

private def attributeValue(name: ObjectName, attr: MBeanAttributeInfo) = {

*// it's possible getAttribute throws an exception, see the output below*

Try(server.getAttribute(ObjectName.getInstance(name), attr.getName))

}

}

The output looks like this:

Set(java.lang:type=MemoryPool,name=Code Cache

- Name (java.lang.String) = Success(Code Cache)

- Type (java.lang.String) = Success(NON\_HEAP)

- CollectionUsage (javax.management.openmbean.CompositeData) = Success(null)

- CollectionUsageThreshold (long) = Failure(javax.management.RuntimeMBeanException: java.lang.UnsupportedOperationException: CollectionUsage threshold is not supported)

...

Set(java.nio:type=BufferPool,name=mapped

- Name (java.lang.String)=Success(mapped)

- MemoryUsed (long)=Success(0)

...

Set(java.lang:type=GarbageCollector,name=PS Scavenge

- LastGcInfo (javax.management.openmbean.CompositeData)=Success(...)

...

We can see it’s possible for an application to monitor itself, connecting to its own MBean server. Some values could be easier to catch there than using some third-party APIs, or when it’s just impossible to grab elsewhere.

But it’s mostly useful to connect to another application, or pool of applications, to grab some specific attributes, and act upon their values (monitoring, alerting, routing, load balancing…).

Scala wrapper: jajmx

There is a library which implements the JMX API with some Scala wrappers: [jajmx](https://github.com/dacr/jajmx). This way, no need of this Java non-sense (ok, it’s not that complicated but still).

import jajmx.\_

val jmx = JMX()

import jmx.\_

mbeans.take(10).map(\_.name).foreach(println)

The API is a bit more Scala’ish.

java.lang:type=Memory

java.lang:type=MemoryPool,name=PS Eden Space

java.lang:type=MemoryPool,name=PS Survivor Space

...

It also provides some smart sh scripts to query any application with JMX and retrieve specific values, list threads, use filters… Take a [look](https://github.com/dacr/jajmx)!

Connect to a distant JMX Agent

By default, it’s not possible to connect to a distant JMX Agent. The distant application must add some Java options to allow the connection.

This is the purpose of the [JSR 160: Java™ Management Extensions (JMX) Remote API](https://jcp.org/en/jsr/detail?id=160): to handle JMX remote management with RMI.

The most common options to use on the distant application are (the others are mostly security related):

# -Dcom.sun.management.jmxremote

-Dcom.sun.management.jmxremote.local.only=false

-Dcom.sun.management.jmxremote.port=9010

-Dcom.sun.management.jmxremote.authenticate=false

-Dcom.sun.management.jmxremote.ssl=false

* -Dcom.sun.management.jmxremote: was necessary until J2SE6, is not needed anymore, but we can still stumbled upon it.
* -Dcom.sun.management.jmxremote.local.only: by default, it’s true to accept only local connections. If a portis specified, it is automatically switch to true.
* -Dcom.sun.management.jmxremote.port: publish a RMI connector on this port for a remote application to connect to. It’s just the registry port (default is 1099), there is another port which is the server port (random).
* -Dcom.sun.management.jmxremote.authenticate=false: by default, the authentication is enabled. The user must beforehands update the config files in JRE\_HOME/lib/management to specify users, passwords, permissions… It’s often disabled! You must be sure nobody from the exterior can access it.
* -Dcom.sun.management.jmxremote.ssl=false: by default, SSL is enabled when connecting remotely. The user must beforehands update create a certificate and import it into the keystore of the JVM. Often disabled! You must be sure nobody from the exterior can access it.

Finally, 2 more options are necessary when the private networks are different between the client and the server, or for other networking reasons:

-Djava.rmi.server.hostname=server

-Djava.rmi.server.useLocalHostname=true

* -Djava.rmi.server.hostname=server: it is the address the client (the RMI stub) will use to query the remote server. For instance, if the server is not on the same local network than ours, the default address will the be private IP of the server, that we can’t reach. But it’s possible we can reach it through its hostname or another IP, so we set this property.
* -Djava.rmi.server.useLocalHostname: if java.rmi.server.hostname is not specified, the client will use the hostname of the server instead of the local IP (it’s useful when the IP is a private one belonging to another network). It’s a shortcut/alternative to java.rmi.server.hostname.

More documentation is available in the [Agent technotes](http://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html).

If you want to connect through SSH tunnel, there is a nice [SO thread](http://stackoverflow.com/questions/15093376/jconsole-over-ssh-local-port-forwarding/32418821#32418821) to explain how to.

Now that we’ve seen the theory, let’s dive into the JMX ecosystem!

Jolokia: JMX to HTTP

A Java Agent

Jolokia is a Java Agent used to expose JMX through HTTP (as JSON), which is universal.

A Java agent is some piece of code started when the JVM starts, that can instrument classes before the real application starts OR it can be plugged on any JVM application on the fly.

Jolokia supports attributes listing, reading, writing, and methods execution. Jolokia simplifies how to use JMX because JSON through HTTP is way more accessible and can be used by any language. Jolokia[provides](https://jolokia.org/features/polyglot.html) some client libraries to simplify the flow (Java, Javascript (with jQuery, erk), Perl), but anything can query the HTTP endpoint, it’s plain JSON.

The installation of Jolokia is quite straight-forward:

* We [download](https://jolokia.org/download.html) a .jar because we work with pure Java applications: [jolokia-jvm-1.3.5-agent.jar](http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-jvm/1.3.5/jolokia-jvm-1.3.5-agent.jar).
* We add -javaagent to the command line when we start Java to take our .jar into account ([it’s also possible to start the agent on an already running JVM](https://jolokia.org/reference/html/agents.html#jvm-attach)). Configuring the command line can be done through the IDE project configuration or directly in build.sbt when we use sbt run:

fork in run := true

javaOptions += "-javaagent:jolokia-jvm-1.3.5-agent.jar=port=7777,host=localhost"

mainClass in (Compile, run) := Some("com.ctheu.JMXTest")

We’ll get a log stating it’s all good:

[info] I> No access restrictor found, access to any MBean is allowed

[info] Jolokia: Agent started with URL http://127.0.0.1:7777/jolokia/

Now, when we query http://localhost:7777/jolokia/, we get the agent version:

{

"request": {

"type": "version"

},

"value": {

"agent": "1.3.5",

"protocol": "7.2",

"config": {

"maxDepth": "15",

"discoveryEnabled": "true",

...

Queries

From there, we can list, read, or write any attributes and execute methods.

List

When we are looking around:

http://localhost:7777/jolokia/list

*# or a particular namespace*

http://localhost:7777/jolokia/list/java.lang

*# or a particular attribute*

http://localhost:7777/jolokia/list/java.lang/type=Memory/attr/HeapMemoryUsage

{

"request": { "type": "list" },

"value": {

"JMImplementation": {},

"java.util.logging": {},

"java.lang": {

"name=PS Scavenge,type=GarbageCollector": {},

"type=Threading": {},

"name=PS Old Gen,type=MemoryPool": {},

"type=Memory": {

"op": { "gc": { "args": [], "ret": "void", "desc": "gc" } },

"attr": {

"ObjectPendingFinalizationCount": {},

"Verbose": { "rw": true, "type": "boolean", "desc": "Verbose" },

"HeapMemoryUsage": {

"rw": false,

"type": "javax.management.openmbean.CompositeData",

"desc": "HeapMemoryUsage"

},

...

Note that this route does not return the values, but only the JMX metadata.

Read

It’s perfect if we know what we are looking for.

It’s the route to use when we want to monitor some specific metrics in a monitoring system and renders some nice charts because it exposes the values.

http://localhost:7777/jolokia/read/java.lang:type=Memory

*# or a particular attribute*

http://localhost:7777/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/used

{

"request": { "mbean": "java.lang:type=Memory", "type": "read" },

"value": {

"ObjectPendingFinalizationCount": 0,

"Verbose": false,

"HeapMemoryUsage": {

"init": 268435456,

"committed": 257425408,

"max": 3814195200,

"used": 59135648

},

"NonHeapMemoryUsage": {

"init": 2555904,

"committed": 17235968,

"max": -1,

"used": 16706800

},

"ObjectName": { "objectName": "java.lang:type=Memory" }

},

"timestamp": 1485728539,

"status": 200

}

Write

Let’s say Jolokia has some MBeans that return these values:

*// http://localhost:7777/jolokia/read/jolokia:type=Config*

{ "HistorySize": 82, "MaxDebugEntries": 100, "HistoryMaxEntries": 10, "Debug": false }

*// http://localhost:7777/jolokia/list/jolokia/type=Config/attr/Debug*

{ "rw": true, "type": "boolean", "desc": "Attribute exposed for management" }

We see jolokia:type=Config > Debug is writeable (rw: true) and we have its current value.

We can modify it with a classic GET (with the value at the end):

http://localhost:7777/jolokia/write/jolokia:type=Config/Debug/true

If we read it again:

{ "HistorySize": 82, "MaxDebugEntries": 100, "HistoryMaxEntries": 10, "Debug": true }

Method execution

There are already some existing MBeans in the JRE we can call:

http://localhost:7777/jolokia/exec/java.lang:type=Memory/gc

*# or with arguments*

http://localhost:7777/jolokia/exec/java.util.logging:type=Logging/setLoggerLevel/global/FINER

Those are truly useful when methods are doing complex operations. We can basically call any method remotely that will affect the process (or just return a result), thanks to JMX.

It’s possible to do all those queries with POST when a GET is not enough to pass arguments properly (such as maps, arrays, complex types). GET has only a basic support of arrays based on the “a,b,c” notation.

Note that the agent has a lot of options available, we can get them by getting the help from the agent.jar itself:

$ java -jar jolokia-jvm-1.3.5-agent.jar --help

...

--host <host> Hostname or IP address to which to bind on

(default: InetAddress.getLocalHost())

--port <port> Port to listen on (default: 8778)

--agentContext <context> HTTP Context under which the agent is reachable (default: /jolokia)

...

--user <user> User used for Basic-Authentication

--password <password> Password used for Basic-Authentication

--quiet No output. "status" will exit with code 0 if the agent is running, 1 otherwise

--verbose Verbose output

...

As we can see, the endpoint security is builtin in Jolokia. All the options are also listed on the[reference guide](https://jolokia.org/reference/html/agents.html#agents-jvm).

Jolokia is a very nice tool to consider when we want to quickly plug an application into an existing monitoring system which has probably already something to read metrics from HTTP. It’s useless to develop a custom HTTP service expose metrics. It’s better to expose them through JMX, then, by HTTP with Jolokia. That will provide 2 ways to read the metrics.

Camel: stay awhile and listen

Camel is a generic sources and sinks connector, that can be used to create complex pipelines of events publishing/consuming. It has a support for absolutely every possible source or sink (files, messages queues, sockets, aws, ftp, mail, irc, and [more more more](https://camel.apache.org/components.html)). Here, we’re just going to talk about the JMX part.

Camel handles it in two ways:

* It can listen to MBeans modifications, this is the [JMX Component](https://camel.apache.org/jmx.html).
* We can use JMX to monitor Camel internals, this is [Camel JMX](http://camel.apache.org/camel-jmx.html).

Listen to MBeans modifications

Camel can subscribe to MBeans that are listenable: MBeans that implementsjavax.management.NotificationBroadcaster (this just offers a simple publisher/subscriber interface).

It’s quite common to inherit directly from NotificationBroadcasterSupport that implements it and support Executors (to notify the listeners asynchronously).

A typical MBean implementation would be:

trait MyMetricMBean {

def getValue(): Double

def update(): Unit *// we'll use it after*

}

class MyMetric(var value: Double) extends NotificationBroadcasterSupport with MyMetricMBean {

*// a notification needs a unique sequence number*

private val sequence = new AtomicInteger()

override def getValue(): Double = value

def setValue(newValue: Double) = {

val oldValue = value

this.value = newValue

*// this is quite verbose but at least, it contains everything we could think of*

this.sendNotification(new AttributeChangeNotification(this,

sequence.incrementAndGet(), Instant.now().toEpochMilli,

"Value changed", "Value", "Double",

oldValue, newValue

))

}

override def update(): Unit = setValue(math.random)

}

Now, we can register an instance of this MBean into the local MBeans server, and ask Camel to subscribe to its notifications, and act upon them (here, we’ll just log the notification on stdout):

object WithCamel extends App with LazyLogging {

val server = ManagementFactory.getPlatformMBeanServer

val metric = new MyMetric(0)

server.registerMBean(metric, ObjectName.getInstance("com.ctheu:type=MyMetric"))

val context = new DefaultCamelContext()

context.addRouteDefinition(new RouteDefinition(

"timer:updateValue?period=800")

.bean(metric, "update"))

context.addRouteDefinition(new RouteDefinition(

"jmx:platform?objectDomain=com.ctheu&key.type=MyMetric")

.to("log:com.ctheu:INFO"))

*// it's possible to be notified only according to some thresholds (<0.05 or >0.95 here)*

*/\* context.addRouteDefinition(new RouteDefinition(*

*"jmx:platform?objectDomain=com.ctheu&key.type=MyMetric" +*

*"&observedAttribute=Value&monitorType=gauge&granularityPeriod=500" +*

*"&notifyHigh=true&notifyLow=true&thresholdHigh=0.95&thresholdLow=0.05")*

*.to("log:com.ctheu:INFO")) \*/*

context.start()

Thread.sleep(60000)

context.stop()

}

We can find the JMX values in this stdout extract:

Route: route1 started and consuming from: timer://updateValue?period=800

Route: route2 started and consuming from: jmx://platform?key.type=MyMetric&objectDomain=com.ctheu&observedAttribute=Value

Total 2 routes, of which 2 are started.

Apache Camel 2.18.1 (CamelContext: camel-1) started in 0.647 seconds

INFO com.ctheu:INFO - Exchange[ExchangePattern: InOnly, BodyType: String, Body:

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

<AttributeChangeNotification xmlns="urn:org.apache.camel.component:jmx">

<source>com.ctheu:type=MyMetric</source>

<message>Value changed</message>

<sequence>1</sequence>

<timestamp>1486933777838</timestamp>

<dateTime>2017-02-12T22:09:37.838+01:00</dateTime>

<type>jmx.attribute.change</type>

<attributeName>Value</attributeName>

<attributeType>Double</attributeType>

<newValue>0.23164144347463556</newValue>

<oldValue>0.0</oldValue>

</AttributeChangeNotification>]

INFO com.ctheu:INFO - Exchange[ExchangePattern: InOnly, BodyType: String, Body:

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

<AttributeChangeNotification xmlns="urn:org.apache.camel.component:jmx">

<source>com.ctheu:type=MyMetric</source>

<message>Value changed</message>

<sequence>2</sequence>

<timestamp>1486933778624</timestamp>

<dateTime>2017-02-12T22:09:38.624+01:00</dateTime>

<type>jmx.attribute.change</type>

<attributeName>Value</attributeName>

<attributeType>Double</attributeType>

<newValue>0.37393058222906805</newValue>

<oldValue>0.23164144347463556</oldValue>

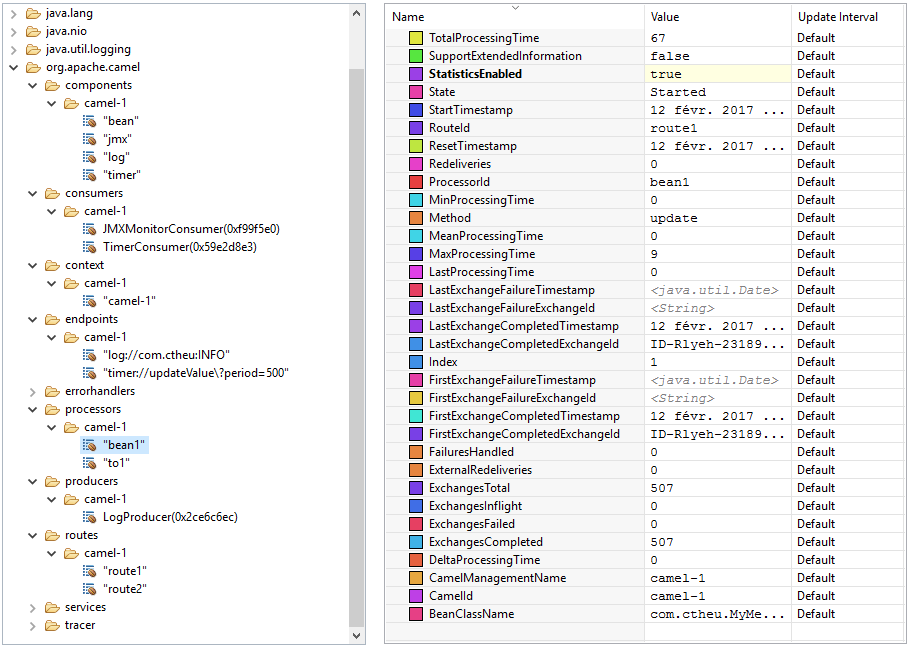
</AttributeChangeNotification>]

From there, we can trigger any pipeline and send those data to messages brokers, monitoring systems, files, anything.

Monitor Camel internals with JMX

Camel exposes a TONS of MBeans.

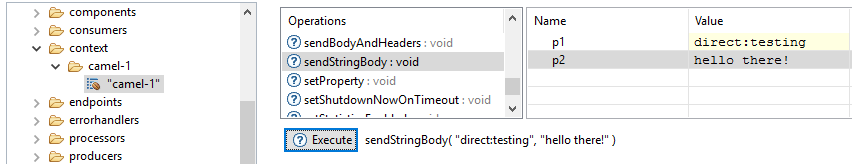
* The Camel context: generally only one per application. It exposes a bunch of global metrics, exchanges counts, processing time.
* All instantiated components: from which the endpoints were created.
* All endpoints: their configuration (mutable).
* All processors: the things doing the work. They exposes the same metrics as the context but just about them.
* All consumers: the things that listen to some incoming messages.
* All producers: the things that publish the messages.
* All routes: the pipelines of consumers and producers. Same metrics as processors but about the whole pipeline.



All those MBeans have operations to get part of their state (getters), dump the stats and routes as XML, pause/stop/start the context/consumers/producers/processors/routes…

Also, through JMX, we can send custom data into the Camel context directly. Let’s say we create a simple direct endpoint:

context.addRouteDefinition(new RouteDefinition("direct:testing").to("log:com.ctheu:INFO"))



[RMI TCP Connection(5)-10.0.75.1]

INFO com.ctheu:INFO - Exchange[ExchangePattern: InOnly, BodyType: String, Body: hello there!]

Because any program can invoke the JMX methods, and Camel is able to handle them and put them into some pipeline, the possibilities and combinaisons are endless.

A program running Camel can get its data from another application, do some transformations, merge multiple sources, send and broadcast new events anywhere else: it can just act as a big processor itself.

Kamon and JMX

Kamon’s features

[Kamon](http://kamon.io/introduction/get-started/) is a delightful metrics framework written in Scala.

[Dropwizard’s Metrics](http://metrics.dropwizard.io/3.1.0/getting-started/) is another good metrics framework written in Java.

Without writing any code, Kamon can already provide some classic metrics (JVM, System), but it’s mostly useful to create custom metrics to expose and measure the internals of our application (database latency, count of items, time to execute some code…). The documentation is clear, the API is good and not overwhelming.

Kamon has a lot of features:

* Provides different types of metrics (counters, histograms…).
* Measures the time to execute any code block.
* Measures the Futures execution time.
* Measures the JVM and System metrics.
* Provides metrics about Executor Services (threads pools).

It also provides some plugins for specific frameworks: Akka, Play Framework, JDBC, Elasticsearch…

And finally, Kamon is able to send the metrics to tons of backends: stdout, StatsD, FluentD, …JMX! The one we care about here.

<http://kamon.io/backends/jmx/>

Exposing metrics to JMX

Here is a complete example that simulate some gets and insertions into a database:

libraryDependencies ++= Seq("io.kamon" %% "kamon-core" % "0.6.5",

"io.kamon" %% "kamon-jmx" % "0.6.5")

object WithKamon extends App {

Kamon.start()

implicit val system = ActorSystem("with-kamon")

implicit val ec = system.dispatcher

val scheduler = system.scheduler

val latency = Kamon.metrics.histogram("database-get-ms", Time.Milliseconds)

val counter = Kamon.metrics.counter("inserted-records")

scheduler.schedule(0 second, 10 millis)

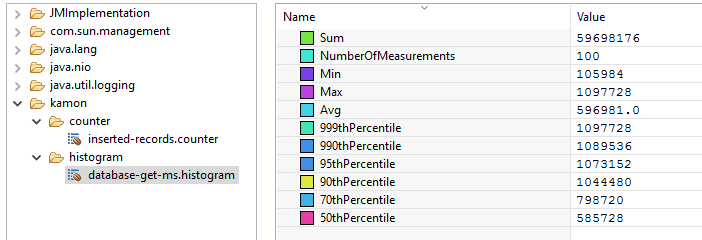
(latency.record((math.random \* 1000000).toInt + 100000))

scheduler.schedule(0 second, 15 millis)

(counter.increment((math.random \* 10).toInt))

}

Thanks to the JMX backend, we can check our metrics through JMX:



It can be very handy to, for instance, add some alerting (email, slack) if the database latency is greater than 1s or if the count of items is 0 while we don’t expect this case.

Small tips: by default, Kamon sends the metrics to the backends every 10s. To change this interval, we can add kamon.metric.tick-interval = 1 second into our application.conf.

kamon-akka: Monitoring Akka’s actors

A very nice Kamon plugin is [kamon-akka](http://kamon.io/integrations/akka/overview/).

Thanks to it, it’s very easy to monitor the internals of any actors in the application (which is something not trivial). It rely on a Java Agent that must be started with the JVM (to alter the bytecode).

Let’s say we have a program with a main actor PingActor that sends a Ping(i+1) to 10 PongActors that each reply with i+1 to the unique PingActor:

case object Start

case class Ping(i: Int) extends AnyVal

case class Pong(i: Int) extends AnyVal

class PingActor(target: ActorRef) extends Actor {

override def receive = {

*// The Ping(n) will be broadcast to all PongActors by the router*

case Start => target ! Ping(0)

case Pong(i) => sender ! Ping(i+1)

}

}

class PongActor extends Actor {

override def receive = {

case Ping(i) => sender ! Pong(i+1)

}

}

object WithKamon extends App {

Kamon.start()

implicit val system = ActorSystem("with-kamon")

val router = system.actorOf(BroadcastPool(10).props(Props[PongActor]), "routero")

val pingo = system.actorOf(Props(classOf[PingActor], router), "pingo")

pingo ! Start

}

We use a generic configuration to monitor all actors, dispatchers, and routers of the system:

kamon.metric.filters {

akka-actor {

includes = ["with-kamon/user/\*\*", "with-kamon/system/\*\*"]

excludes = []

}

akka-dispatcher {

includes = ["with-kamon/akka.actor.default-dispatcher"]

excludes = []

}

akka-router {

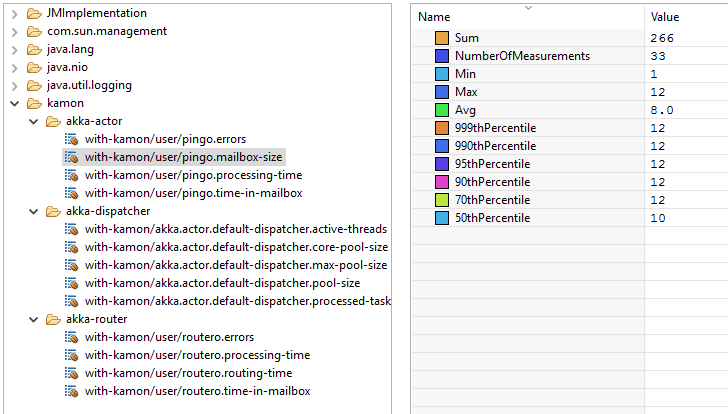
includes = [ "with-kamon/\*\*" ]

excludes = []

}

}

Then, we can see the graal in our JMX connector:



We can find back our routero router and pingo actor and monitor their Akka internal state:

* mailboxes size: it should stay small
* processing time: it should be tiny
* errors: it should be 0

We can create nice dashboards with those metrics to clearly have the picture of what’s going on within the actors, that’s truly useful.

JMXTrans: Send JMX metrics anywhere

Standalone

JMXTrans is mostly a scheduler (based on [quartz](https://github.com/quartz-scheduler/quartz)) that pulls data from any JMX source and send them to one or multiple sinks (to store them and draw dashboards).

We just have to set some config file and start the application anywhere, it will try to connect to the sources and the sinks at a regular pace (1min by default).

I’ve created a repository [chtefi/jmxtrans-docker](https://github.com/chtefi/jmxtrans-docker) for this part, feel free to use it.

We can download JMXTrans here <http://central.maven.org/maven2/org/jmxtrans/jmxtrans/263/>, specifically the built distribution archive: [jmxtrans-263-dist.tar.gz](http://central.maven.org/maven2/org/jmxtrans/jmxtrans/263/jmxtrans-263-dist.tar.gz).

Then we can execute it:

$ tar zxvf jmxtrans-263-dist.tar.gz

$ java -jar jmxtrans-263/lib/jmxtrans-all.jar --help

The important options are:

-f, --json-file

-q, --quartz-properties-file

The Quartz server properties.

-s, --run-period-in-seconds

The seconds between server job runs.

Default: 60

* -f: the main configuration to provide for JMXTrans to know which source and sink(s) to connect. It can contain multiple sources and multiple sinks. It’s also possible to put several config file in a folder (-j).
* -q: if we want to specify some quartz properties. JMXTrans has some defaults in the file quartz-server.properties. Quartz has [tons of options](http://www.quartz-scheduler.org/documentation/quartz-2.2.x/configuration/) such as its threadpool config, listeners, plugins, misc thresholds.
* -s: to change the default 60s poll interval (runPeriod).

Queries

The configuration is what JMXTrans calls [Queries](https://github.com/jmxtrans/jmxtrans/wiki/Queries).

This is where we define the JMX sources and the sinks where we want to send the JMX values.

For instance, it can listen to the JMX data on localhost:9010 and send the results to stdout:

{

"servers": [{

"port": "9010",

"host": "localhost",

"queries": [{

"outputWriters": [{

"@class": "com.googlecode.jmxtrans.model.output.StdOutWriter"

}],

"obj": "java.lang:type=OperatingSystem",

"attr": [ "SystemLoadAverage", "AvailableProcessors", "TotalPhysicalMemorySize",

"FreePhysicalMemorySize", "TotalSwapSpaceSize", "FreeSwapSpaceSize",

"OpenFileDescriptorCount", "MaxFileDescriptorCount" ]

}],

"numQueryThreads": 2

}]

}

JMXTrans will watch the given properties of the JMX “node” java.lang:type=OperatingSystem:

Result(attributeName=SystemLoadAverage,

className=sun.management.OperatingSystemImpl,

objDomain=java.lang,

typeName=type=OperatingSystem,

values={SystemLoadAverage=-1.0},

epoch=1485905825980,

keyAlias=null)

Result(attributeName=FreePhysicalMemorySize,

className=sun.management.OperatingSystemImpl,

objDomain=java.lang,

typeName=type=OperatingSystem,

values={FreePhysicalMemorySize=5871636480},

epoch=1485905825980,

keyAlias=null)

...

If we had a custom Java application with custom JMX MBeans, we could use:

trait RandomMetricsMBean {

def getValue(): Double

}

class RandomMetrics(@BeanProperty var value: Double) extends RandomMetricsMBean

val metrics = new RandomMetrics(0d)

server.registerMBean(metrics, ObjectName.getInstance("com.ctheu:type=RandomMetrics"))

"queries": [{

"outputWriters": [{

"@class": "com.googlecode.jmxtrans.model.output.StdOutWriter"

}],

"obj": "com.ctheu:type=RandomMetrics",

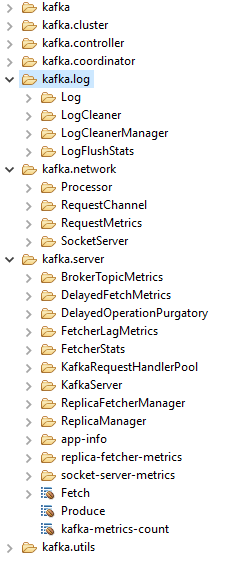
"attr": []

}],

By default, if no attributes are specified, JMXTrans will just take them all.

Example: Kafka as source

Kafka exposes tons of MBeans about its internals.



We can retrieve get the start/end offsets of each partitions, get metrics about elections, logs flushing, queues size, messages/bytes per seconds (globally, per topic), and so much more.

Let’s say we want to monitor the FetcherStats metrics, we setup our JMXTrans queries with this:

{

"outputWriters": [{

"@class": "com.googlecode.jmxtrans.model.output.GraphiteWriterFactory",

"port": "2003",

"host": "192.168.0.11",

"flushStrategy": "always",

"typeNames": ["name", "clientId", "brokerHost", "brokerPort" ]

}],

"obj": "kafka.server:type=FetcherStats,\*",

"resultAlias": "kafka",

"attr": []

}

The typeNames are used when we are using the wildcard \* to grab the full hierarchy undernealth. This will properly set the path names of the metrics in Graphite.

For instance, the complete ObjectName of one of theses nodes is:

kafka.server:type=FetcherStats,

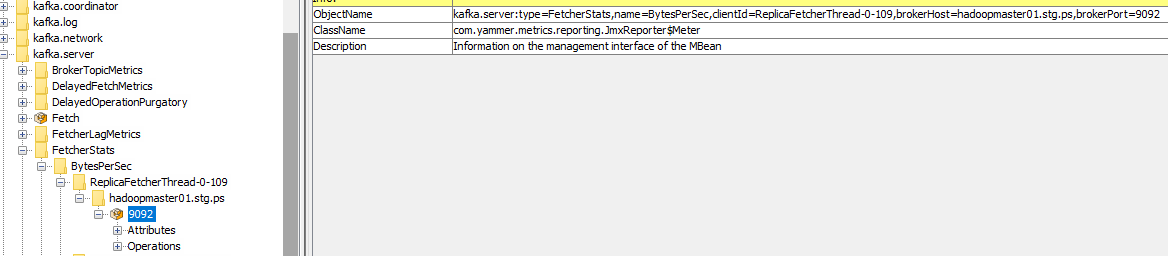
name=BytesPerSec,

clientId=ReplicaFetcherThread-0-109,

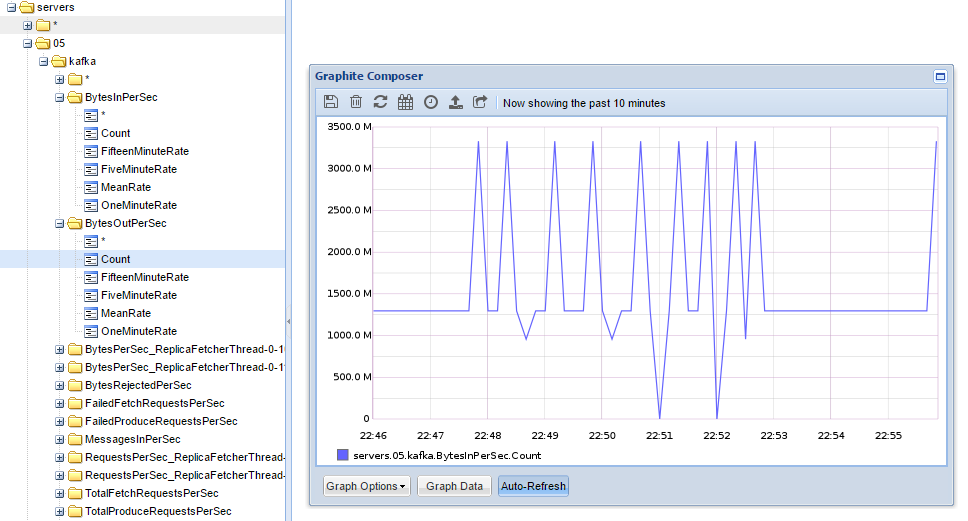
brokerHost=hadoopmaster01.stg.ps,

brokerPort=9092

As seen in jconsole:



Thanks to typeNames, we’ll have a clear and distinct path in Graphite for all the attributes:



JMXTrans as an Agent

It’s also possible to run JMXTrans not a standalone application, but as an Java agent, starting directly with the application it monitors.

This is the purpose of the project [jmxtrans-agent](https://github.com/jmxtrans/jmxtrans-agent) which embeds JMXTrans, and just rely on a config.xml file (that contain the JMX queries to do). It’s too bad it’s not the same format as the JMXTrans standalone application.

We already used an agent previously, Jolokia. This is the same story:

* We [download](https://github.com/jmxtrans/jmxtrans-agent/releases/latest) the agent: [jmxtrans-agent-1.2.4.jar](https://github.com/jmxtrans/jmxtrans-agent/releases/download/jmxtrans-agent-1.2.4/jmxtrans-agent-1.2.4.jar).
* We add -javaagent to the command line when we start Java to take our .jar into account. Configuring the command line can be done through the IDE project configuration or directly in build.sbt when we use sbt run:

fork in run := true

javaOptions += "-javaagent:jmxtrans-agent-1.2.4.jar=path/to/jmxtrans-agent.xml"

mainClass in (Compile, run) := Some("com.ctheu.JMXTest") *// a simple blocking app*

Let’s use the macro feature (explained below) of the agent (#...#) and create this config file, that will output the metrics to stdout every 10s:

<jmxtrans-agent>

<queries>

<query objectName="java.lang:type=OperatingSystem"

resultAlias="os.#attribute#"/>

<query objectName="java.lang:type=Memory"

attribute="HeapMemoryUsage" resultAlias="mem.#key#"/>

<query objectName="java.lang:type=Runtime"

attribute="InputArguments" resultAlias="args.#position#"/>

</queries>

<outputWriter class="org.jmxtrans.agent.ConsoleOutputWriter"/>

<collectIntervalInSeconds>10</collectIntervalInSeconds>

</jmxtrans-agent>

If we don’t put a resultAlias, it will still work but the metrics name will be empty!

If everything is right, when we sbt run, we should get:

[info] 2017-02-16 23:28:10.483 INFO [main] org.jmxtrans.agent.JmxTransAgent -

Starting 'JMX metrics exporter agent: 1.2.4' with configuration 'path/to/jmxtrans-agent.xml'...

[info] 2017-02-16 23:28:10.499 INFO [main] org.jmxtrans.agent.JmxTransAgent -

PropertiesLoader: Empty Properties Loader

[info] 2017-02-16 23:28:15.527 INFO [main] org.jmxtrans.agent.JmxTransAgent -

JmxTransAgent started with configuration 'path/to/jmxtrans-agent.xml'

[info] os.CommittedVirtualMemorySize 479408128 1487285558

[info] os.FreePhysicalMemorySize 4773953536 1487285558

[info] os.FreeSwapSpaceSize 2861543424 1487285558

[info] os.ProcessCpuLoad 0.008171516430434778 1487285558

...

[info] mem.committed 257425408 1487285558

[info] mem.init 268435456 1487285558

[info] mem.max 3814195200 1487285558

[info] mem.used 22726360 1487285558

[info] args.0 -javaagent:jmxtrans-agent-1.2.4.jar=src/main/resources/jmxtrans-agent.xml 1487285558

...

We clearly see our os, mem, and args metrics, with the macros replaced.

Macros automatically pick the name of the resources to name the metrics:

* #attribute#: for classic JMX MBeans properties (the fields).
* #key#: for CompositeData (maps) fields.
* #position#: for arrays.

And, last but not least, multiple backends exists, especially Graphite:

<outputWriter class="org.jmxtrans.agent.GraphitePlainTextTcpOutputWriter">

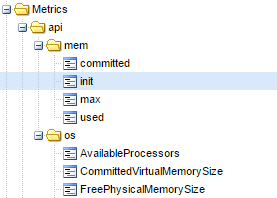
<host>localhost</host>

<port>2003</port>

<namePrefix>api.</namePrefix>

</outputWriter>

Results in Graphite:



Easy as pie right?

I invite you to check the repository [jmxtrans-agent](https://github.com/jmxtrans/jmxtrans-agent) to read a bit more about the other available backends. (files, statsd, influxdb)

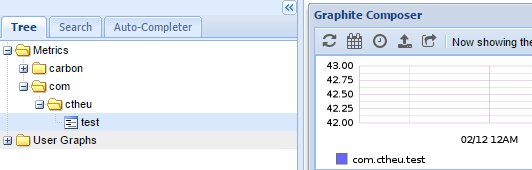
The raw Graphite protocol: using nc and ngrep

Not directly related to JMX but more to Graphite and specially Carbon (which is the piece that listen to incoming metrics), it’s possible and very easy to send any metrics to Carbon using simple tool such asnc.

The input format of Carbon is [metric path] [value] [timestamp]. The JMXTrans Graphite connector simply translates the JMX attributes path and values to this format when we use Graphite as sink.

For instance, to send a metric value:

echo "com.ctheu.test 42 $(date +%s)" | nc 192.168.0.11 2003



When running into troubles, it’s possible to check if the carbon port receives something using ngrep:

# ngrep -d any port 2003

interface: any

filter: (ip or ip6) and ( port 2003 )

####

T 172.17.0.1:54598 -> 172.17.0.2:2003 [AP]

com.ctheu.test 42 1486316847.

########

T 172.17.0.1:54600 -> 172.17.0.2:2003 [AP]

com.ctheu.test 43 1486316862.

####

The Swiss Java Knife: jvm-tools / sjk

[jvm-tools](https://github.com/aragozin/jvm-tools) provides several general Java command line tool (monitor threads, gc, memory…), and things about JMX, what we care about here.

Click [here](https://bintray.com/artifact/download/aragozin/generic/sjk-plus-0.4.2.jar) to download the fat-jar.

mxdump: The whole JMX tree into JSON

It’s useful when we just want to deal with JSON without thinking twice. We can put this into some NoSQL database and query it the way we want.

$ java -jar sjk-plus-0.4.2.jar mxdump -p 3032

{

"beans" : [ {

"name" : "java.lang:type=MemoryPool,name=Metaspace",

"modelerType" : "sun.management.MemoryPoolImpl",

"Name" : "Metaspace",

"Type" : "NON\_HEAP",

"Valid" : true,

"UsageThreshold" : 0,

"UsageThresholdSupported" : true,

"Usage" : {

"committed" : 227868672,

"init" : 0,

"max" : -1,

"used" : 219723800

},

mx: query the MBeans

[mx](https://github.com/aragozin/jvm-tools/blob/master/sjk-core/COMMANDS.md#mx-command) is more granular and allow us to pick which MBean we’d like to query attributes, and provide setter and invoke methods.

It’s useful to use it with some bash scripts: the script can monitor and affect the Java process internals.

$ java -jar sjk-plus-0.4.2.jar mx -p 3032 -b java.lang:type=Memory --info

java.lang:type=Memory

sun.management.MemoryImpl

- Information on the management interface of the MBean

(A) HeapMemoryUsage : CompositeData

(A) NonHeapMemoryUsage : CompositeData

(A) ObjectPendingFinalizationCount : int

(A) Verbose : boolean - WRITEABLE

(A) ObjectName : javax.management.ObjectName

(O) gc() : void

(A)ttributes, and (O)perations!

$ java -jar sjk-plus-0.4.2.jar mx -p 3032 -b java.lang:type=Memory --attribute HeapMemoryUsage --get

java.lang:type=Memory

committed: 2112618496

init: 2147483648

max: 2112618496

used: 1116738328

As we said, it’s possible to call operations, for instance let’s call [dumpAllThreads](https://docs.oracle.com/javase/7/docs/api/java/lang/management/ThreadMXBean.html#dumpAllThreads(boolean,%20boolean)) exposed by theThreadMXBean:

$ java -jar sjk-plus-0.4.2.jar mx -p 3032 -b java.lang:type=Threading \

--operation dumpAllThreads --call \

--arguments [false,false]

java.lang:type=Threading

blockedCount|blockedTime|inNative|lockInfo |lockName |lockOwnerId|lockOwnerName|lockedMonitors|lockedSynchronizers|stackTrace |suspended|threadId|threadName |threadState |waitedCount|waitedTime

------------+-----------+--------+----------------------------------------+----------------------------------------+-----------+-------------+--------------+-------------------+----------------------------------------+---------+--------+----------------------------------------+-------------+-----------+----------

2 |-1 |false |{className=[I,identityHashCode=167611...|[I@63e785f6 |-1 |null | | |{className=java.lang.Object,fileName=...|false |217 |JMX server connection timeout 217 |TIMED\_WAITING|3 |-1

2 |-1 |false |{className=[I,identityHashCode=315703...|[I@12d13f02 |-1 |null | | |{className=java.lang.Object,fileName=...|false |216 |JMX server connection timeout 216 |TIMED\_WAITING|3 |-1

...

Conclusion

JMX is a simple and powerful technology to expose any Java application internals and react upon their modifications.

* The Jolokia project is perfect to expose the MBeans values directly through HTTP, to be able to consume them with absolutely any application in any language.
* We can integrate JMX updates or notifications into Camel pipelines.
* With Kamon to expose custom metrics and Akka Actors internals, JMXTrans to poll values, and a backend such as Graphite to store them, it’s possible to create useful monitoring and alerting systems, and some pretty dashboards to display the evolution of any metrics.

Exposing metrics through JMX is useful when we don’t want our application to push its metrics itself somewhere (such as: Kamon ➔ FluentD ➔ Graphite): we can let any application pull them directly from our JMX Agent.

# Threading

C:\Java\jdk1.8.0\_111\jre\lib\rt.jar!\sun\management\ThreadImpl.class

[2,0]表示线程2，stackTrace无值

curl 'http://localhost:7777/jolokia/' -XPOST -d '{

"type":"EXEC",

"mbean":"java.lang:type=Threading",

"operation":"getThreadInfo(long,int)",

"arguments":[2,0]

}'

public ThreadInfo getThreadInfo(long var1) {  
 long[] var3 = new long[]{var1};  
 ThreadInfo[] var4 = this.getThreadInfo(var3, 0);  
 return var4[0];  
}  
  
public ThreadInfo getThreadInfo(long var1, int var3) {  
 long[] var4 = new long[]{var1};  
 ThreadInfo[] var5 = this.getThreadInfo(var4, var3);  
 return var5[0];  
}  
  
public ThreadInfo[] getThreadInfo(long[] var1) {  
 return this.getThreadInfo(var1, 0);  
}

# Jolokia - posts requests on jolokia agent returns with 'no context found erorr'

http://jolokia.963608.n3.nabble.com/posts-requests-on-jolokia-agent-returns-with-no-context-found-erorr-td3453847.html

Hi,   
  
I am getting errors when i try to post requests for a jolokia that is installed as a JVM agent.   
  
When running:   
curl '<http://ubuntu.eng.com:8778/jolokia'> -XPOST -d '{"type":"read","mbean":"java.lang:type=Memory","attribute":"HeapMemoryUsage"}'   
  
This error returns: 

404 Not Found

No context found for request   
  
But running the following works on jolokia installed as a JVM agent:   
curl <http://ubuntu.eng.com:8778/jolokia/read/java.lang:type=Memory/HeapMemoryUsage>  
  
And also running:   
curl '<http://127.0.0.1:8080/jolokia'> -XPOST -d '{"type":"read","mbean":"java.lang:type=Memory","attribute":"HeapMemoryUsage"}'   
  
works in case of installing jolokia as a war.   
  
Can you help me solve the mystery?   
Is the a problem using post in case of an agent installation?   
  
Thanks!

Could you please try the URL with a trailing slash ? E.g.

curl 'http://ubuntu.eng.com:8778/jolokia**/**' -XPOST -d '{"type":"read","mbean":"java.lang:type=Memory","attribute":"HeapMemoryUsage"}'

# JavaAgent 简单例子 - 小单的博客专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/catoop/article/details/51034739

JavaAgent 是JDK 1.5 以后引入的，也可以叫做[**Java**](http://lib.csdn.net/base/javase)代理。

JavaAgent 是运行在 main方法之前的拦截器，它内定的方法名叫 premain ，也就是说先执行 premain 方法然后再执行 main 方法。

那么如何实现一个 JavaAgent 呢？很简单，只需要增加 premain 方法即可。

看下面的代码和代码中的注释说明：

package com.shanhy.demo.agent;

import java.lang.instrument.Instrumentation;

/\*\*

\* 我的Java代理

\*

\* @author 单红宇(365384722)

\* @myblog http://blog.csdn.net/catoop/

\* @create 2016年3月30日

\*/

public class MyAgent {

/\*\*

\* 该方法在main方法之前运行，与main方法运行在同一个JVM中

\* 并被同一个System ClassLoader装载

\* 被统一的安全策略(security policy)和上下文(context)管理

\*

\* @param agentOps

\* @param inst

\* @author SHANHY

\* @create 2016年3月30日

\*/

public static void premain(String agentOps, Instrumentation inst) {

System.out.println("=========premain方法执行========");

System.out.println(agentOps);

}

/\*\*

\* 如果不存在 premain(String agentOps, Instrumentation inst)

\* 则会执行 premain(String agentOps)

\*

\* @param agentOps

\* @author SHANHY

\* @create 2016年3月30日

\*/

public static void premain(String agentOps) {

System.out.println("=========premain方法执行2========");

System.out.println(agentOps);

}

}

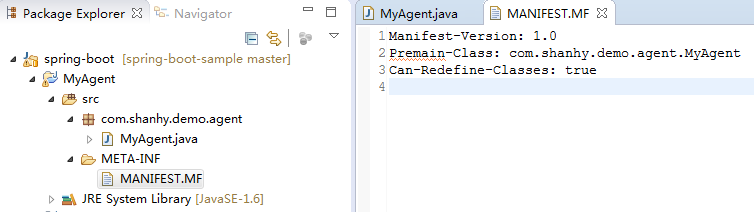
写完这个类后，我们还需要做一步配置工作。

在 src 目录下添加 META-INF/MANIFEST.MF 文件，内容按如下定义：

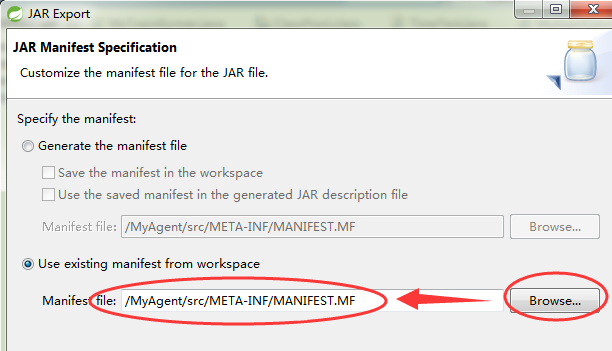
Manifest-Version: 1.0

Premain-Class: com.shanhy.demo.agent.MyAgent

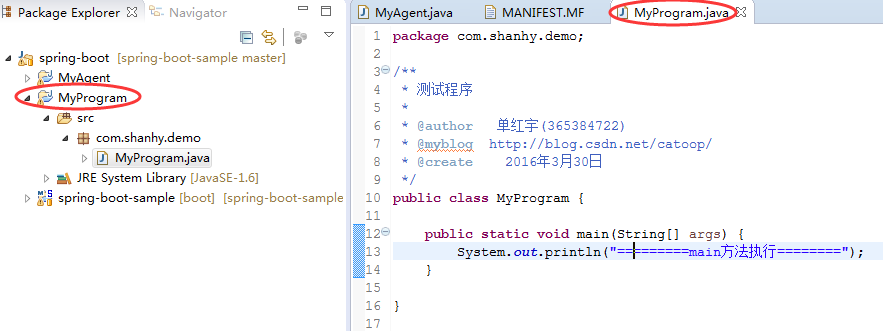
Can-Redefine-Classes: true

要特别注意，一共是四行，第四行是空行，还有就是冒号后面的一个空格，如下截图：   


然后我们打包代码为 myagent.jar

注意打包的时候选择我们自己定义的 MANIFEST.MF   


接着我们在创建一个带有main方法的主程序工程，截图如下： 



然后将该主程序打包为 myapp.jar

如何执行 myagent.jar ？我们通过 -javaagent 参数来指定我们的Java代理包，值得一说的是 -javaagent 这个参数的个数是不限的，如果指定了多个，则会按指定的先后执行，执行完各个 agent 后，才会执行主程序的 main 方法。

命令如下：

java -javaagent:G:\myagent.jar=Hello1 -javaagent:G:\myagent.jar=Hello2 -javaagent:G:\myagent.jar=Hello3 -jar myapp.jar

* 1
* 1

输出结果：

G:\>java -javaagent:G:\myagent.jar=Hello1 -javaagent:G:\myagent.jar=Hello2 -javaagent:G:\myagent.jar=Hello3 -jar myapp.jar

=========premain方法执行========

Hello1

=========premain方法执行========

Hello2

=========premain方法执行========

Hello3

=========main方法执行========

特别提醒：如果你把 -javaagent 放在 -jar 后面，则不会生效。也就是说，放在主程序后面的 agent 是无效的。

比如执行：

java -javaagent:G:\myagent.jar=Hello1 -javaagent:G:\myagent.jar=Hello2 -jar myapp.jar -javaagent:G:\myagent.jar=Hello3

只会有前个生效，第三个是无效的。   
输出结果：

G:\>java -javaagent:G:\myagent.jar=Hello1 -javaagent:G:\myagent.jar=Hello2 -jar myapp.jar -javaagent:G:\myagent.jar=Hello3

=========premain方法执行========

Hello1

=========premain方法执行========

Hello2

=========main方法执行========

命令中的Hello1为我们传递给 premain 方法的字符串参数。

至此，我们会使用 javaagent 了，但是单单看这样运行的效果，好像没有什么实际意义嘛。

我们可以用 javaagent 做什么呢？下篇文章我们来介绍如何在项目中应用 javaagent。

最后说一下，还有一种，在main方法执行后再执行代理的方法，因为不常用，而且主程序需要配置 Agent-Class，所以不常用，如果需要自行了解下 agentmain(String agentArgs, Instrumentation inst) 方法。

# JavaAgent - 小飞的日志 - 网易博客

http://blog.163.com/wangfei\_hello/blog/static/175637430201211985312384/

-**javaagent** 这个JVM参数是JDK 5引进的.

java -help的帮助里面写道：

-**javaagent**:<jarpath>[=<options>] load Java programming language agent, see java.lang.instrument

JDK 工具文档里面，并没有很详细的说明。

**1.**代理 (agent) 是在你的main方法前的一个拦截器 (interceptor)，也就是在main方法执行之前，执行agent的代码。

agent的代码与你的main方法在同一个JVM中运行，并被同一个system classloader装载，被同一的安全策略 (security policy) 和上下文 (context) 所管理。

叫代理（agent）这个名字有点误导的成分，它与我们一般理解的代理不大一样。java agent使用起来比较简单。

怎样写一个java agent? 只需要实现premain这个方法

public static void premain(String agentArgs, Instrumentation inst)JDK 6 中如果找不到上面的这种premain的定义，还会尝试调用下面的这种premain定义：

public static void premain(String agentArgs)

**2.**Agent 类必须打成jar包，然后里面的 **META-INF/MAINIFEST.MF** 必须包含 **Premain-Class**这个属性。

下面是一个MANIFEST.MF的例子：

Manifest-Version: 1.0 Premain-Class:MyAgent1 Created-By:1.6.0\_06

然后把MANIFEST.MF 加入到你的jar包中。

**3.**所有的这些Agent的jar包，都会自动加入到程序的classpath中。所以不需要手动把他们添加到classpath。  
除非你想指定classpath的顺序。

**4.**一个java程序中-**javaagent**这个参数的个数是没有限制的，所以可以添加任意多个java agent。

所有的java agent会按照你定义的顺序执行。  
例如：

java -**javaagent**:MyAgent1.jar -**javaagent**:MyAgent2.jar -jar MyProgram.jar

假设MyProgram.jar里面的main函数在MyProgram中。

MyAgent1.jar, MyAgent2.jar, 这2个jar包中实现了premain的类分别是MyAgent1, MyAgent2  
程序执行的顺序将会是

MyAgent1.premain -> MyAgent2.premain -> MyProgram.main

**5.**另外，**放在main函数之后的premain是不会被执行的**，

例如

java -**javaagent**:MyAgent1.jar -jar MyProgram.jar -**javaagent**:MyAgent2.jar

MyAgent2 和MyAgent3 都放在了MyProgram.jar后面，所以MyAgent2的premain都不会被执行，

所以执行的结果将是

MyAgent1.premain -> MyProgram.main

**6.**每一个java agent 都可以接收一个字符串类型的参数，也就是premain中的**agentArgs**，这个agentArgs是通过java option中定义的。  
如：

java -**javaagent**:MyAgent2.jar=thisIsAgentArgs -jar MyProgram.jar

MyAgent2中premain接收到的agentArgs的值将是”thisIsAgentArgs” （不包括双引号）

**7.** 参数中的Instrumentation：

通过参数中的Instrumentation inst，添加自己定义的ClassFileTransformer，来改变class文件。

**8.**通过java agent就可以不用修改原有的java程序代码，通过agent的形式来修改或者增强程序了，或者做热启动等等。

**9.** JDK 6 中还增加了agentmain，用在JVM启动之后调用，具体大家可以看JDK文档说明。

**10. an example**

package sizeof.agent;

import java.lang.instrument.Instrumentation;  
import java.lang.reflect.Array;  
import java.lang.reflect.Field;  
import java.lang.reflect.Modifier;  
import java.util.IdentityHashMap;  
import java.util.Map;  
import java.util.Stack;

/\*\* Instrumentation agent used \*/  
public class SizeOfAgent {

 static Instrumentation inst;  
   
 /\*\* initializes agent \*/  
 public static void premain(String agentArgs, Instrumentation instP) {  
  inst = instP;    
 }  
   
 /\*\*  
  \* Returns object size without member sub-objects.  
  \* @param o object to get size of  
  \* @return object size  
  \*/  
 public static long sizeOf(Object o) {  
  if(inst == null) {  
   throw new IllegalStateException("Can not access instrumentation environment.\n" +  
     "Please check if jar file containing SizeOfAgent class is \n" +  
     "specified in the java's \"-javaagent\" command line argument.");  
  }  
  return inst.getObjectSize(o);  
 }  
   
 /\*\*  
  \* Calculates full size of object iterating over  
  \* its hierarchy graph.  
  \* @param obj object to calculate size of  
  \* @return object size  
  \*/  
 public static long fullSizeOf(Object obj) {  
  Map<Object, Object> visited = new IdentityHashMap<Object, Object>();  
  Stack<Object> stack = new Stack<Object>();

     long result = internalSizeOf(obj, stack, visited);  
     while (!stack.isEmpty()) {  
       result += internalSizeOf(stack.pop(), stack, visited);  
     }  
     visited.clear();  
     return result;  
 }    
     
    private static boolean skipObject(Object obj, Map<Object, Object> visited) {  
     if (obj instanceof String) {  
       // skip interned string  
       if (obj == ((String) obj).intern()) {  
         return true;  
       }  
     }  
     return (obj == null) // skip visited object  
         || visited.containsKey(obj);  
  }

    private static long internalSizeOf(Object obj, Stack<Object> stack, Map<Object, Object> visited) {  
     if (skipObject(obj, visited)){  
      return 0;  
     }  
     visited.put(obj, null);  
       
     long result = 0;  
     // get size of object + primitive variables + member pointers   
     result += SizeOfAgent.sizeOf(obj);  
       
     // process all array elements  
     Class clazz = obj.getClass();  
     if (clazz.isArray()) {  
       if(clazz.getName().length() != 2) {// skip primitive type array  
        int length =  Array.getLength(obj);  
     for (int i = 0; i < length; i++) {  
      stack.add(Array.get(obj, i));  
        }   
       }         
       return result;  
     }  
       
     // process all fields of the object  
     while (clazz != null) {  
       Field[] fields = clazz.getDeclaredFields();  
       for (int i = 0; i < fields.length; i++) {  
         if (!Modifier.isStatic(fields[i].getModifiers())) {  
           if (fields[i].getType().isPrimitive()) {  
            continue; // skip primitive fields  
           } else {  
             fields[i].setAccessible(true);  
             try {  
               // objects to be estimated are put to stack  
               Object objectToAdd = fields[i].get(obj);  
               if (objectToAdd != null) {                
                 stack.add(objectToAdd);  
               }  
             } catch (IllegalAccessException ex) {   
              assert false;   
             }  
           }  
         }  
       }  
       clazz = clazz.getSuperclass();  
     }  
     return result;  
  }  
}

# 2@Spring boot监控初探 - 推酷

http://www.tuicool.com/articles/uaiqiaE

时间 2017-03-26 01:19:13  [大名Dean鼎](http://www.tuicool.com/sites/nMnmEnY)

原文  [http://www.deanwangpro.com/2017/03/22/spring-boot-monitor/](http://www.deanwangpro.com/2017/03/22/spring-boot-monitor/?utm_source=tuicool&utm_medium=referral)

主题 [Spring Boot](http://www.tuicool.com/topics/11070125)

最近对devOps这个话题有点兴趣，所以研究了一下monitor相关的开源项目，翻到medium上的[***一篇文章***](https://medium.com/@brunosimioni/near-real-time-monitoring-charts-with-spring-boot-actuator-jolokia-and-grafana-1ce267c50bcc#.il5xmlnv7) ,而且实际项目中也曾看到devOps组的同事搭过类似的监控，就想过把瘾，了解一下监控可视化。

被监控服务配置

本地正好有spring-boot的项目，并且也依赖了 **jolokia** （主要就是为了把JMX的mbean通过HTTP暴露出去）

项目配置也少不了

endpoints:

enabled: true

jmx:

enabled: true

jolokia:

enabled: true

management:

security:

enabled: false

访问一下URL看看是不是ok

http://localhost:8080/jolokia/**read**/org.springframework.boot:name=metricsEndpoint,type=Endpoint/Data

搭建监控系统

如果能看到数据，说明server端配置没问题了，下面我们怎么搭建Telegraf + InfluxDB + Grafana呢，这个三个组件是这么配合的，Telegraf实际就是收集信息的，比如每隔10s访问一次上面那个URL得到metrics，收集到的数据存到InfluxDB，然后Grafana做数据可视化。

但是如果纯手动安装实在太麻烦，求助万能的github，找到一个非常棒的项目( [***https://github.com/samuelebistoletti/docker-statsd-influxdb-grafana***](https://github.com/samuelebistoletti/docker-statsd-influxdb-grafana) ), 直接fork然后修改一些配置就可以为自己的项目服务了。如果你不了解相关配置可以先直接run起来，然后通过ssh进去一探究竟。

**ssh** root@localhost -p 22022

配置方面，主要是要修改Telegraf的，因为它是对接不同项目的，你需要收集什么样的信息，比如cpu，disk，net等等都要在Telegraf里配。简单起见，我只设置了三个输入。

# /etc/telegraf/telegraf.conf

[[inputs.jolokia]]

context = "/jolokia"

[[inputs.jolokia.servers]]

name = "springbootapp"

host = "{app ip address}"

port = "8080"

[[inputs.jolokia.metrics]]

name = "metrics"

mbean = "org.springframework.boot:name=metricsEndpoint,type=Endpoint"

attribute = "Data"

[[inputs.jolokia.metrics]]

name = "tomcat\_max\_threads"

mbean = "Tomcat:name=\"http-nio-8080\",type=ThreadPool"

attribute = "maxThreads"

[[inputs.jolokia.metrics]]

name = "tomcat\_current\_threads\_busy"

mbean = "Tomcat:name=\"http-nio-8080\",type=ThreadPool"

attribute = "currentThreadsBusy"

其实就是spring-boot标准的metrics以及tomcat的Threads。

完成之后重启服务 **/etc/init.d/telegraf restart**

查看监控数据

我们访问InfluxDB看看有数据了没有 **http://localhost:3004/** ，切换数据库到Telegraf。输入以下命令试试吧

**SHOW** MEASUREMENTS

**SELECT** \* **FROM** jolokia

**SELECT** \* **FROM** cpu

**SELECT** \* **FROM** mem

**SELECT** \* **FROM** diskio

比如输入 **SELECT \* FROM jolokia** 就能看到spring-boot暴露了哪些数据，从time列也可以看出Telegraf是每隔10s收集一次，太频繁了对server也是压力。



上面基本涵盖了cpu，内存和存储的一些metrics。

其实也可以配置网络相关的，感兴趣的可以看官方的telegraf.conf，里面有配置[[inputs.net]]的例子。

数据可视化

数据有了，下一步就是可视化。

按照Github上面说的进入 **http://localhost:3003/** ，

1. Using the wizard click on **Add data source**
2. Choose a **name** for the source and flag it as **Default**
3. Choose **InfluxDB** as **type**
4. Choose **direct** as **access**
5. Fill remaining fields as follows and click on **Add** without altering other fields

**Url**: **http**://localhost:8086

**Database**: telegraf

**User**: telegraf

**Password**: telegraf

添加好InfluxDB后，新建一个Dashboard，然后快速的ADD几个Graph来。

为了演示，我添加了三个，分别使用下面三组查询语句来渲染出三张图表

**SELECT** MEAN(usage\_system) + MEAN(usage\_user) **AS** cpu\_total **FROM** cpu **WHERE** $timeFilter **GROUP** **BY** **time**($interval)

**SELECT** mean("total") **as** "total" **FROM** "mem" **WHERE** $timeFilter **GROUP** **BY** **time**($interval) fill(null)

**SELECT** mean("used") **as** "used" **FROM** "mem" **WHERE** $timeFilter **GROUP** **BY** **time**($interval) fill(null)

**SELECT** mean("metrics\_heap.used") **as** "heap\_usage" **FROM** "jolokia" **WHERE** $timeFilter **GROUP** **BY** **time**($interval) fill(null)

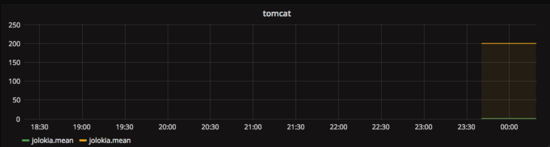
第一张是CPU占用率；第二张是内存占用情况，绿线是Total，黄线是Used；第三张是jolokia提供的jvm heap的使用，可以到看到GC的情况。



刚才还配置了Tomcat的收集，想看Tomcat的Thread情况也是妥妥的。

**SELECT** mean("tomcat\_max\_threads") **FROM** "jolokia" **WHERE** $timeFilter **GROUP** **BY** **time**($interval) fill(null)

**SELECT** mean("tomcat\_current\_threads\_busy") **FROM** "jolokia" **WHERE** $timeFilter **GROUP** **BY** **time**($interval) fill(null)



小结

可以看到搭建这样一套环境其实很快，原理也并不复杂，监控数据可视化的难点在于

* 哪些metrics需要监控
* 哪些metrics需要配合起来可以判断问题，比如diskio+net是不是可以判断系统整体IO的瓶颈。

这都是需要多年的经验总结才能获得的，我还是菜鸟一枚，再接再厉。

# 2@使用 Spring Boot Actuator、Jolokia 和 Grafana 实现准实时监控 - 后端 - 掘金

https://juejin.im/entry/58b7e545a22b9d005ed087cb

使用 Spring Boot Actuator、Jolokia 和 Grafana 实现准实时监控

阅读 408

收藏 24

2017-03-02

原文链接：[http://www.jianshu.com/p/954cc0caea6c](https://link.juejin.im/?target=http%3A%2F%2Fwww.jianshu.com%2Fp%2F954cc0caea6c)

由于最近在做监控方面的工作，因此也读了不少相关的经验分享。其中有这样一篇文章总结了一些基于 Spring Boot 的监控方案，因此翻译了一下，希望可以对大家有所帮助。 —— 由[aglice](https://juejin.im/user/58ae99592f301e0068f02130)分享

由于最近在做监控方面的工作，因此也读了不少相关的经验分享。其中有这样一篇文章总结了一些基于Spring Boot的监控方案，因此翻译了一下，希望可以对大家有所帮助。

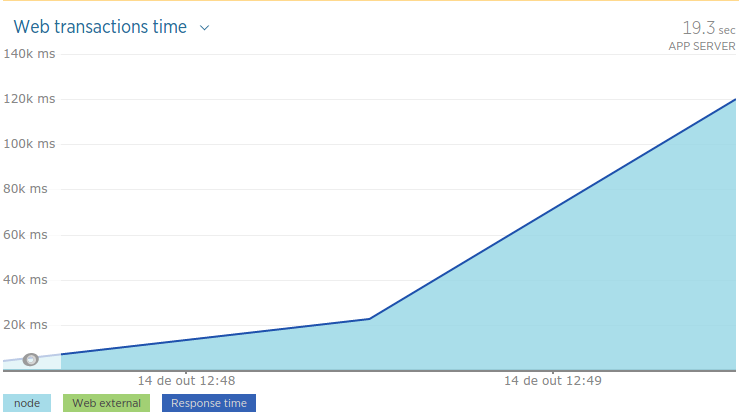
原文：[Near real-time monitoring charts with Spring Boot Actuator, Jolokia and Grafana](https://link.juejin.im/?target=https%3A%2F%2Fmedium.com%2F%40brunosimioni%2Fnear-real-time-monitoring-charts-with-spring-boot-actuator-jolokia-and-grafana-1ce267c50bcc%23.il5xmlnv7)

Spring Boot Actuator通过/metrics端点，以开箱即用的方式为应用程序的性能指标与响应统计提供了一个非常友好的监控方式。

由于在集群化的弹性环境中，应用程序的节点可以增长、扩展，并由非常大量的应用实例所组成。对于孤立节点的监控可能即费力又没有什么实际效果。所以，使用基于时间序列的数据聚合工具将获得更好的效果。

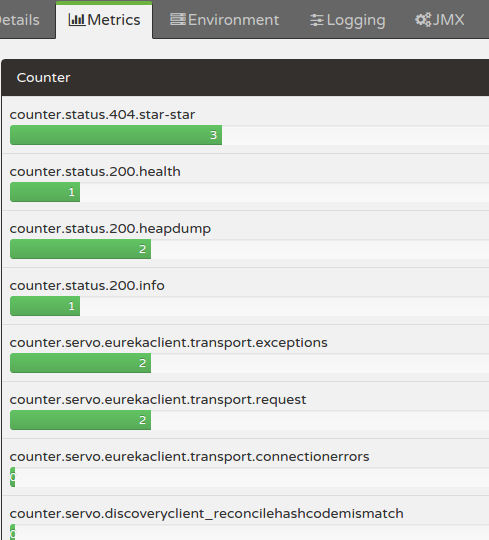
本文的目标在于找出一种仅需要通过工具和配置的方式就能实现的解决方案，来对Spring Boot Metrics实现基于时间序列的监控。

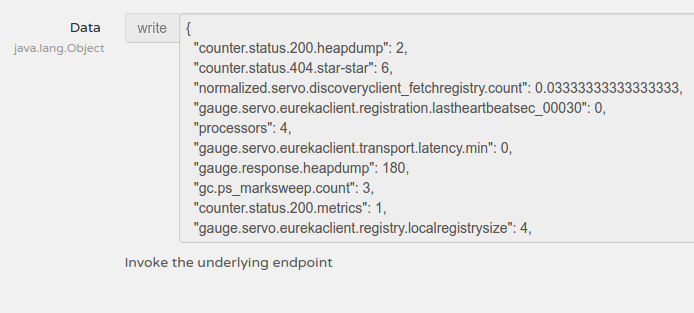
像NewRelic, AppDynamics或DataDog这些APM系统都能很好地完成这样的任务，它们通过使用JVM和字节码工具来生成自己的指标、分析工具和相关事务。也可以通过使用@Timed注释方法来实现。但是，这些方法将忽略所有Spring Boot Actuator库所提供的可用资源。另外，使用这些方法还有一个与保留数据相关的问题，它们对于短时间窗口内的监控是相对模糊的。



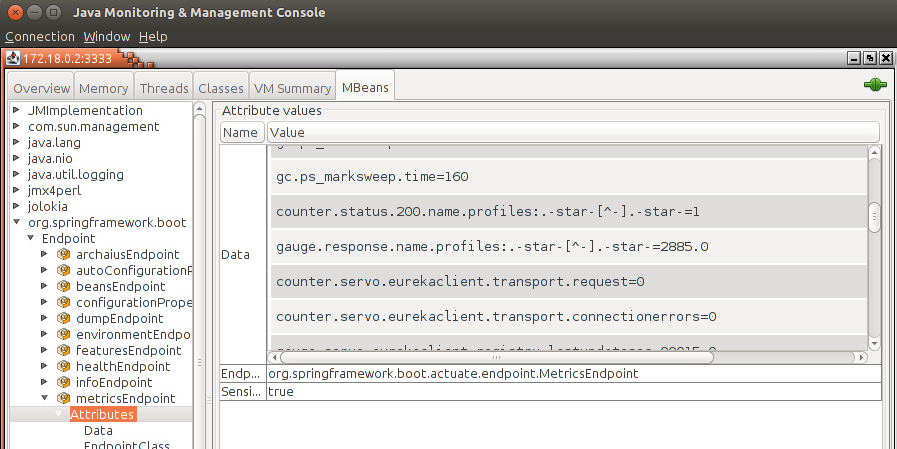
*NewRelic在1分钟时间窗口内被发现和检测的事务*

spring-boot-admin 可以作为另外一个备选方案，因为它可以连接到Spring Boot的实例、并且可以聚合节点等。但是， /metrics 端点并不是根据时间轴来进行监控的，同时在不同节点上的相同应用模块（水平扩展）也没有得到聚合。这意味着您将面对这两种情况：没有时间序列的监控数据、只有对孤立节点的监控数据快照。

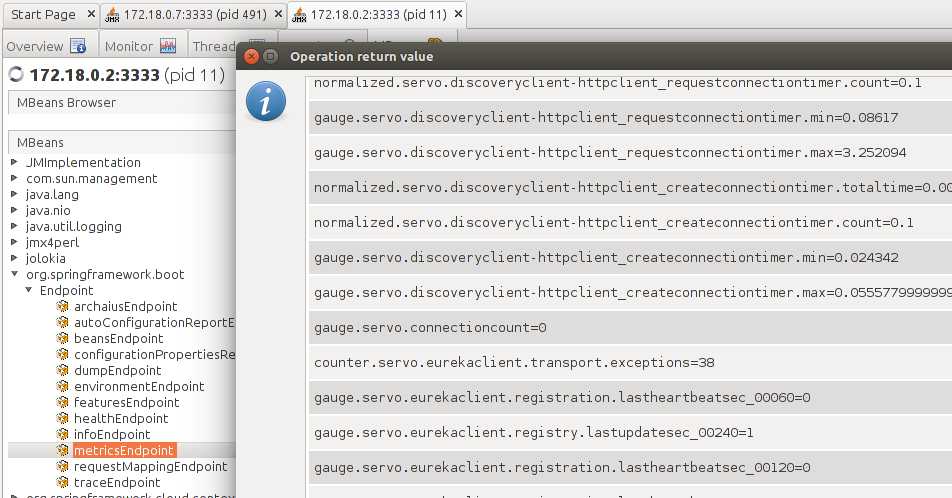
  
*Spring Boot Admin with metrics from Actuator: a snapshot of metrics data of a given application node*

  
*Spring Boot Admin with JMX and MBeans read data of a give application node*

jconsole和visualvm可能是另外一种选择，它们通过RMI直接连接到JMX节点。Actuator存储来自JMX的MBean内的Metrics数据。另外，通过使用 [Jolokia](https://link.juejin.im/?target=https%3A%2F%2Fjolokia.org%2F)，MBeans以RESTful HTTP端点的方式暴露，/jolokia。所以，相同的信息可以通过两个端点来获取：JMX MBean Metrics和Rest HTTP Jolokia端点。然而，这种方式存在同样的问题，它们直接连接到集群环境中的单个节点，另外还伴随着痛苦的老式RMI协议。



*JConsole old-school JMX Metrics of a given application node*



*VisualVM JMX Metrics of a give application node*

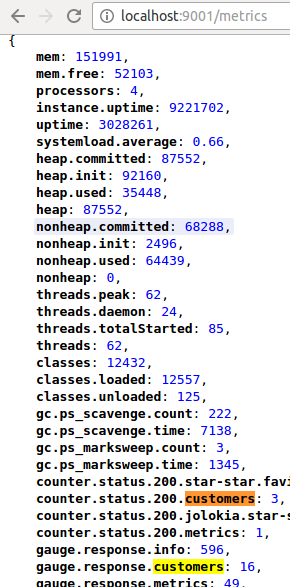
继续前进，我尝试了一些可能可以解决这些问题的现代化运维工具：

* [**Prometheus**](https://link.juejin.im/?target=https%3A%2F%2Fprometheus.io%2F): 由SoundCloud编写，它存储一系列的监控数据并赋予漂亮的图标展现。Prometheus Gauges和Actuator Metrics并不完全兼容，所以人们写了 [一个数据转换器](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fprometheus%2Fclient_java%2Fpull%2F114)。你也可以配置Prometheus来收集JMX数据。
* [**Sensu**](https://link.juejin.im/?target=https%3A%2F%2Fsensuapp.org%2F): 作为Nagios和Zabbix的现代化替代品，它有一个插件可以直接连接到Spring Boot，[但是这个仓库最近已经不太更新了](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fsensu-plugins%2Fsensu-plugins-springboot)，所以我决定放弃它。
* [**StatsD**](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fetsy%2Fstatsd): Spring Boot有一篇文章是[关于自定义导出数据给StatsD](https://link.juejin.im/?target=http%3A%2F%2Fdocs.spring.io%2Fspring-boot%2Fdocs%2Fcurrent-SNAPSHOT%2Freference%2Fhtmlsingle%2F%23production-ready-metric-writers-export-to-statsd)。然而，你除了要为Spring Boot应用安装StatsD实例之外，还不得不实现一些存根来让它工作起来。
* [**Graphite**](https://link.juejin.im/?target=http%3A%2F%2Fgraphiteapp.org%2F)**:** You got to be [a hero to install and get Graphite](https://link.juejin.im/?target=https%3A%2F%2Fgraphite.readthedocs.io%2Fen%2Flatest%2Finstall.html) running. If you get there,[you can configure it along StatsD to get metrics working in a chart](https://link.juejin.im/?target=https%3A%2F%2Fmatt.aimonetti.net%2Fposts%2F2013%2F06%2F26%2Fpractical-guide-to-graphite-monitoring%2F).
* [**OpenTSDB**](https://link.juejin.im/?target=http%3A%2F%2Fopentsdb.net%2F)**:** Spring Boot有一篇文章[关于连接数据到OpenTSBD](https://link.juejin.im/?target=http%3A%2F%2Fdocs.spring.io%2Fspring-boot%2Fdocs%2Fcurrent-SNAPSHOT%2Freference%2Fhtmlsingle%2F%23production-ready-metric-writers-export-to-open-tsdb). 然而，这种方式与StatsD类似，你必须实现和维护自定义的代码来让它工作起来。另外，OpenTSDB没有开箱即用的图形可视化工具。
* [**JMXTrans**](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fjmxtrans%2Fjmxtrans): 可以用来提取数据并发送到其他的监控工具，它也需要具体的实现。
* [**Ganglia**](https://link.juejin.im/?target=http%3A%2F%2Fganglia.info%2F): 也是基于JVM上的工具，记录所有Actuator资源。与之前所说的APM有相同问题。

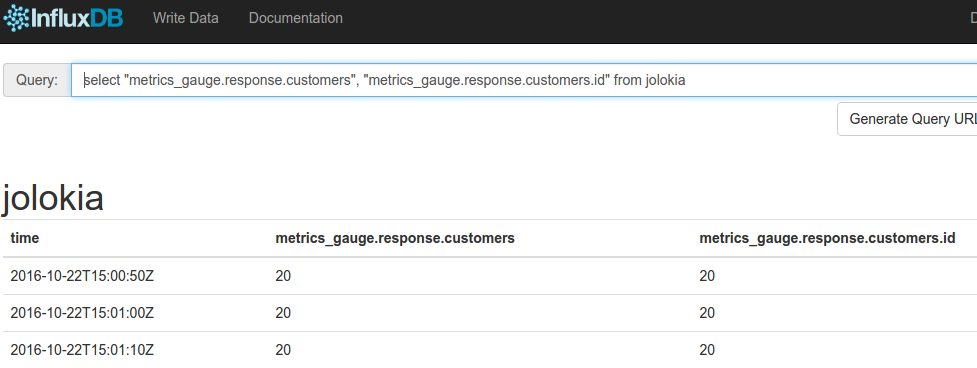
经过一番研究，我发现了一个更好的解决方案：通过InfluxDB 和Telegraf实现，零编码，只需要通过一些正确的配置。

* [**Jolokia**](https://link.juejin.im/?target=https%3A%2F%2Fjolokia.org%2F): Spring Boot [认可使用Jolokia来通过HTTP导出export JMX数据](https://link.juejin.im/?target=http%3A%2F%2Fdocs.spring.io%2Fspring-boot%2Fdocs%2Fcurrent-SNAPSHOT%2Freference%2Fhtmlsingle%2F%23production-ready-jolokia)。你只需要在工程类路径中增加一些依赖项，一切都是开箱即用的。不需要任何额外的实现。
* [**Telegraf**](https://link.juejin.im/?target=https%3A%2F%2Fwww.influxdata.com%2Ftime-series-platform%2Ftelegraf%2F)**:** Telegraf支持通过整合Jolokia来集成JMX数据的收集。[它有一个预制的输入插件](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Finfluxdata%2Ftelegraf%2Ftree%2Fmaster%2Fplugins%2Finputs%2Fjolokia)，它是开箱即用的。不需要任何额外的实现。只需要做一些配置即可。
* [**InfluxDB**](https://link.juejin.im/?target=https%3A%2F%2Fwww.influxdata.com%2Ftime-series-platform%2Finfluxdb%2F)**:** InfluxDB通过 [输出插件](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Finfluxdata%2Ftelegraf%2Ftree%2Fmaster%2Fplugins%2Foutputs%2Finfluxdb)从Telegraf接收指标数据，它是开箱即用的，不需要任何额外的实现。
* [**Grafana**](https://link.juejin.im/?target=http%3A%2F%2Fgrafana.org%2F): Grafana通过[连接InfluxDB作为数据源](https://link.juejin.im/?target=http%3A%2F%2Fdocs.grafana.org%2Fdatasources%2Finfluxdb%2F)来渲染图标。它是开箱即用的，不需要额外的实现。

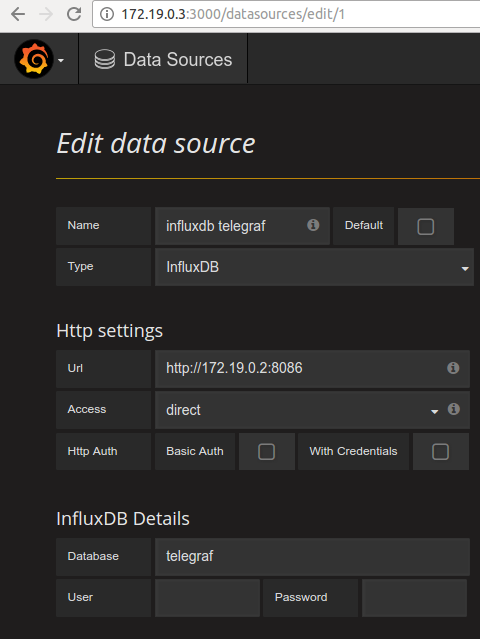
简而言之，配置所有这些东西都非常的简单。

  
*Spring Boot Actuator Raw*

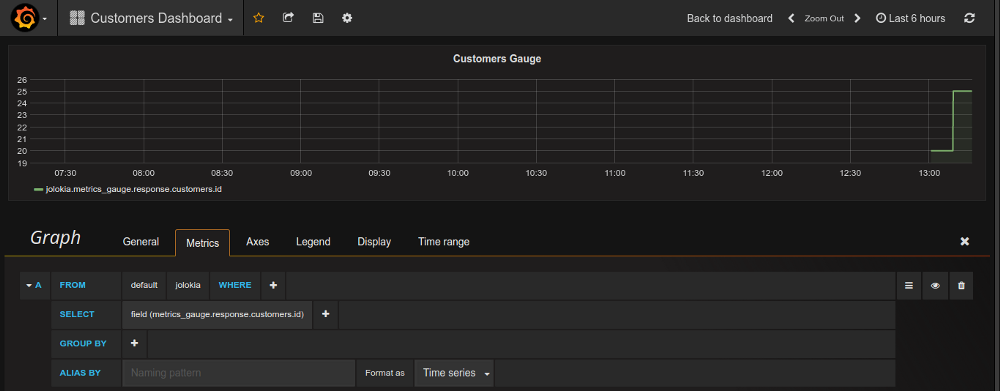
*Metrics*



*Metrics sent by Telegraf to InfluxDB, collected by Jolokia and JMX over HTTP*



*Grafana InfluxDB data source configuration*



*Grafana Metric chart query and configuration: gauges of an API*

# 2@利用 Telegraf 进行简单的系统监控 - 推酷

http://www.tuicool.com/articles/EJrEZfN

时间 2017-03-14 23:09:26  [伪架构师](http://www.tuicool.com/sites/EjEFJfN)

原文  [http://blog.fleeto.us/content/li-yong-telegraf-jin-xing-jian-dan-de-xi-tong-jian-kong](http://blog.fleeto.us/content/li-yong-telegraf-jin-xing-jian-dan-de-xi-tong-jian-kong?utm_source=tuicool&utm_medium=referral)

主题 [系统监控](http://www.tuicool.com/topics/11000134)[InfluxDB](http://www.tuicool.com/topics/11030128)

InfluxData 除了广为人知的 InfluxDB 之外，还有几个其他的产品，合称 TICK：

* Telegraf：数据采集
* InfluxDB：数据存储
* Chronograf：数据展现
* Kapacitor：数据分析、告警

在翻看 InfluxDB 的时候偶然发现了这个东西，虽然 Tick 四兄弟捆起来也不够看，不过 Telegraf 足够小巧，而且自动化的可能性更大，更符合目前的做事风格，所以就学习一下。

官宣： The plugin-driven server agent for collecting & reporting metrics.

所以 Telegraf 主要是一个框架，由数据输入、处理、输出三大类插件完成各种功能。Github 的 README.md 中列出了主要插件： **https://github.com/influxdata/telegraf** 。总的来说还是比较丰富的，下面的操作将利用简单的输入插件结合 InfluxDB 输出插件完成一个初步的指标收集过程。

安装

CentOS

生成如下的 repo 文件：

**[influxdb]**

name = InfluxDB Repository - RHEL $releasever

baseurl = https://repos.influxdata.com/rhel/$releasever/$basearch/stable

enabled = 1

gpgcheck = 0

**yum install -y telegraf** 即可完成安装。

Docker

docker pull telegraf

docker pull registry.alauda.cn/library/telegraf

配置

yum 安装后在 **/etc/telegraf** 下会生成一个 **telegraf.conf** 文件。

配置文件中可以使用 "$ENV\_ITEM" 的形式使用环境变量。

global\_tags

这里记录的内容将作为 Tags 保存到 InfluxDB 的每个 Item 中。

agent

这一节内容是数据搜集服务的行为定义。这里暂时无需进行改动

outputs.influxdb

这里用于定义写入的 InfluxDB。

urls = ["http://localhost:8086"]

database = "telegraf"

timeout = "5s"

username = "telegraf"

password = "abcde!@#$%"

urls 参数是一个数组，代表一个集群，如果其中包含多个服务，则每次只会选择其中一台进行写入。

而在 inputs 一节中，缺省启用了很多系统属性，例如磁盘，网络等，这里我们添加一点 http 监控内容：

**[[inputs.http\_response]]**

address = "http://163.com"

response\_timeout = "5s"

method = "GET"

**[[inputs.http\_response]]**

address = "http://sina.com.cn"

response\_timeout = "5s"

method = "GET"

小窍门：可以用 **telegraf -config telegraf.conf -input-filter http\_response -test** 命令，来检查配置的正确性。

配置文件编写完成之后，就可以利用 **systemctl start telegraf** ，启动 telegraf 服务了。

启动之后，Telegraf 会在一定的时间间隔里向 InfluxDB 汇报数据。我们可以在 InfluxDB UI 中利用

**select** \* **from** cpu

这样的语句来查询数据，或者接入 Grafana 等进行展现。

# [翻译]现代java开发指南 第二部分 - htoooth - 博客园

http://www.cnblogs.com/htoooth/p/5436503.html

Parallel Universe

目录

[**现代 Java 的打包和部署**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_0)  
[**日志**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_1)  
[**用jcmd和jstat进行监控和管理**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_2)  
[**使用JMX进行监控和管理**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_3)  
[**写一个自定义的MBeans**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_4)  
[**使用Metrics进行健康和性能监控**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_5)  
[**性能分析**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_6)  
[**高级话题：使用Byteman进行性能分析和调试**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_7)  
[**高级话题：使用JMH进行基准测试**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_8)  
[**目前为止我们已经学了什么？**](http://www.cnblogs.com/htoooth/p/5436503.html#_caption_9)

现代java开发指南 第二部分

第二部分：部署、监控 & 管理，性能分析和基准测试

[第一部分](http://www.cnblogs.com/htoooth/p/5418328.html)，[第二部分](http://www.cnblogs.com/htoooth/p/5436503.html)

===================

欢迎来到现代 Java 开发指南第二部分。在[第一部分](http://www.cnblogs.com/htoo/p/5418328.html)中，我们已经展示了有关 Java 新的语言特性，库和工具。这些新的工具使 Java 变成了相当轻量级的开发环境，这个开发环境拥有新的构建工具、更容易使用的文档、富有表现力的代码还有用户级线程的并发。而在这部分中，我们将比代码层次更高一层，讨论 Java 的运维———— Java 的部署、监控&管理，性能分析和基准测试。尽管这里的例子都会用 Java 来做示意，但是我们讨论的内容与所有的 JVM 语言都相关，而不仅仅是 Java 语言。

在开始之前，我想简短地回答一下第一部分读者的问题，并且澄清一下说的不清楚的地方。第一部分中最受争议的地方出现在构建工具这一节。在那一节中，我写到现代的 Java 开发者使用 Gradle。有些读者对此提出异议，并且举出了例子来证明 Maven 同样也是一个很好的工具。我个人喜欢 Gradle 漂亮 DSL 和能使用指令式代码来编写非通用的构建操作，同时我也能够理解喜欢完全声明式的 Maven 的偏好,即使这样做需要大量的插件。因此，我承认：现代的 Java 开发者可能更喜欢 Maven 而不是 Gradle 。我还想说，虽然使用 Gradle 不用了解 Groovy ，甚至人们希望在不是那么标准的事情中也不用了解 Groovy 。但是我不会这样，我从 Gradle 的在线例子中已经学习了很多有用的 Groovy 的语句。

有些读者指出我在第一部分的代码示例中使用 Junit 和 Guava ，意味着我有意推广它们。好吧，我确实有这样的想法。[Guava](https://code.google.com/p/guava-libraries/) 是一个非常有用的库，而 [JUnit](http://junit.org/) 是一个很好的单元测试框架。虽然 TestNG 也很好，但是 JUnit 非常常见，很少有人会选择别的就算有优势的测试框架。  
同样，就示例代码中测试使用 [Hamcrest](http://hamcrest.org/JavaHamcrest/) ，一个读者指出 [AssertJ](http://joel-costigliola.github.io/assertj/index.html)，可能是一个比 Hamcrest 更好的选择。

需要理解到本系列指南并不打算覆盖到 Java 的方方面面，能认识到这一点很重要。所以当然会有很多很好的库因为没有在文章中出现，我们没有去探索它们。我写这份指南的本意就是给大家示意一下现代 Java 开发可能是什么样的。

有些读者表达了他们更喜欢短的 Javadoc 注释，这种注释不必像 Javadoc 标准形式那样需要把所有的字段都写上。如下面的例子：

/\*\*

\* This method returns the result.

\* @return the result

\*/

int getResult();

更喜欢这样：

/\*\*

\* Returns the result

\*/

int getResult();

我完全同意。我在例子中简单示范了混合 Markdown 和标准的 Javadoc 标签的使用。这只是用来展示如何使用，并不是意图把这种使用方式当成指导方针。

最后，关于 Android 我有一些话要说。 Android 系统通过一系列变换之后，能够执行用 java (还有可能是别的 JVM 语言)写的代码，但是 Android 不是 JVM，并且事实上 Android 无论在正式场合和实际使用中也不完全是 Java (造成这个问题的原因是两个跨国公司，这里指谷歌和甲骨文，没有就 Java 的使用达成一个许可协议)。正因为 Android 不完全是 Java ，所以在第一部分中讨论的内容对 Android 可能有用或者也可能没有用，而且因为 Android 没有包括 JVM ，所以在这部分讨论的内容很少能应用到 Android 上面。

好了，现在让我们回到正文。

现代 Java 的打包和部署

对于不熟悉 Java 生态体系的人来说，Java（或者任何 JVM 语言）源文件，被编绎成 .class 文件（本质上是 Java 二进制文件），每一个类一个文件。打包这些 class 文件的基本机制就把这些文件打包在一起（这项工作通常由构建工具或者IDE来完成）放到JAR（Java存档）文件，JAR 文件叫 Java 二进制包。 JAR 文件仅仅是 Zip 压缩文件，它包括 class 文件，还有一个附加的清单文件用来描述内容，清单中还可以包括其它的关于分发的信息（如在被[签名的 JARs](http://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html)中，清单可以包括数字签名）。如果你打包一个应用（与此相反是打包一个库）到 JAR 中，清单文件应该指出应用的主类（也就是 main 函数所在类），在这种情况下，应用通过命令java -jar app.jar启动，我们称这个 JAR 文件为可执行的 JAR 。

Java 库被打包成 JAR 文件，然后部署到 Maven 仓库中（这个仓库能被所有的 JVM 构建工具使用，不仅仅是 Maven ）。 Maven 仓库管理这些库二进制文件的版本和依赖（当你发一个请求想从Maven仓库中加载一个库，此外你请求了该库所有的依赖）。开源 Java 库经常托管在这个[中央仓库](http://search.maven.org/)中，或者其它类似的公开仓库中。并且组织机构通过 [Artifactory](http://www.jfrog.com/home/v_artifactory_opensource_overview) 或者 [Nexus](http://www.sonatype.com/nexus-repository-sonatype) 等工具，管理他们私有 Maven 仓库。你甚至能在 [GitHub 上建立自己的 Maven 仓库](https://cemerick.com/2010/08/24/hosting-maven-repos-on-github/)。但是 Maven 仓库在构建过程中应该能正常使用，并且 Maven 仓库通常托管库形式 JAR 而不是可执行的 JAR 。

Java 网站应用传统上应该在应用服务器（或者 servlet 容器）中执行。这些容器能运行多个网站应用，能按需加载或卸载应用。 Java 网站应用以 WAR 的形式部署在 servlet 容器中。WAR 也是 JAR 文件，它的内容以某种标准形式排好，并且包括额外的配置信息。但是，正如我们将在第三部分看到一样，就现代 Java 开发而言，[Java 应用服务器已死](http://www.slideshare.net/ewolff/java-application-servers-are-dead)。

Java 桌面应用经常被打包成与平台相关的二进制文件，还包括一个平台相关的 JVM。 JDK 工具包中有一个[打包工具](http://docs.oracle.com/javafx/2/deployment/self-contained-packaging.htm)来做这个事情([这里](https://getpocket.com/a/read/613195504)是讲的是如何在 NetBeans 中使用它)。第三方工具 [Packer](https://github.com/libgdx/packr)也提供了类似的功能。对于游戏和桌面应用来说，这种打包机非常好。但是对于服务器软件来说，这种打包机制就不是我想要的。此外，因为要打包一个 JVM 的拷贝，这种机制不能以补丁形式安全和平滑地升级应用。

对服务器端代码，我们想要的是一种简单、轻量、能自动的打包和部署的工具。这个工具最好能利用可执行 JAR 的简单和平台无关性。但是可执行 JAR 有几个不足的地方。每一个库通常打包到各自的 JAR 文件中，然后和所有的依赖一起打包成单个 JAR 文件，这一过程可能造成冲突，特别是已打包的资源库（没有 class 文件的库）一起打包时。还有，一个原生库在打包时不能直接放到 JAR 中。打包中可能最重要的是， JVM 配置信息（如 heap 的大小）对用户来说是遗漏的，这个工作必须在命令行下才能做。像 [Maven’s Shade plugin](http://maven.apache.org/plugins/maven-shade-plugin/) 和 [Gradle’s Shadow plugin](https://github.com/johnrengelman/shadow) 等工具，解决了资源冲突的问题，而 [One-Jar](http://one-jar.sourceforge.net/) 支持原生的库，但是这些工具都可能对应用产生影响，而且也没有解决 JVM 参数配置的问题。 Gradle 能把应用打包成一个 ZIP 文件，并且产生一个与系统相关的启脚本去配置 JVM ，但是这种方法要求安装应用。我们可以做的比这样更轻量级。同样，我们有强大的、普遍存在的资源像 Maven 仓库任我们使用，如果不充分利用它们是件令人可耻的事。

这一系列博客打算讲讲用现代 Java 工作是多么简单和有趣（不需牺牲任何性能），但是当我去寻找一种有趣、简单和轻量级的方法去打包、分发和部署服务器端的 Java 应用时，我两手空空。所以 [Capsule](https://github.com/puniverse/capsule) 诞生了（如果你知道有其它更好的选择，请告诉我）。

Capsule 使用平台独立的可执行 JAR 包，但是没有依赖，并且（可选的）能整合强大和便捷的 Maven 仓库。一个 capsule 是一个 JAR 文件，它包括全部或者部分的 Capsule 项目 class，和一个包括部署配置的清单文件。当启动时(java -jar app.jar)， capsule 会依次执行以下的动作：解压缩 JAR 文件到一个缓存目录中，下载依赖，寻找一个合适的 JVM 进行安装，然后配置和运行应用在一个新的JVM进程中。

现在让我们把 Capsule 拿出来溜一溜。我们把[第一部](http://blog.paralleluniverse.co/2014/05/01/modern-java/)的 JModern 项目做为开始的项目。这是我们的 build.gradle 文件：

apply plugin: 'java'

apply plugin: 'application'

sourceCompatibility = '1.8'

mainClassName = 'jmodern.Main'

repositories {

mavenCentral()

}

configurations {

quasar

}

dependencies {

compile "co.paralleluniverse:quasar-core:0.5.0:jdk8"

compile "co.paralleluniverse:quasar-actors:0.5.0"

quasar "co.paralleluniverse:quasar-core:0.5.0:jdk8"

testCompile 'junit:junit:4.11'

}

run {

jvmArgs "-javaagent:${configurations.quasar.iterator().next()}"

}

这里是我们的 jmodern.Main 类：

package jmodern;

import co.paralleluniverse.fibers.Fiber;

import co.paralleluniverse.strands.Strand;

import co.paralleluniverse.strands.channels.Channel;

import co.paralleluniverse.strands.channels.Channels;

public class Main {

public static void main(String[] args) throws Exception {

final Channel<Integer> ch = Channels.newChannel(0);

new Fiber<Void>(() -> {

for (int i = 0; i < 10; i++) {

Strand.sleep(100);

ch.send(i);

}

ch.close();

}).start();

new Fiber<Void>(() -> {

Integer x;

while((x = ch.receive()) != null)

System.out.println("--> " + x);

}).start().join(); // join waits for this fiber to finish

}

}

为了测试一下我们的程序工作是正常的，我们运行一下gradle run。

现在，我们来把这个应用打包成一个 capsule 。在构建文件中，我们将增加 capsule 配置。然后，我们增加依赖包：

capsule "co.paralleluniverse:capsule:0.3.1"

当前 Capsule 有两种方法来创建 capsule （虽然你也可以混合使用）。第一种方法是创建应用时把所有的依赖都加入到 capsule 中；第二种方法是第一次启动 capsule 时让它去下载依赖。我来试一下第一种—— "full" 模式。我们添加下面的任务到构建文件中：

task capsule(type: Jar, dependsOn: jar) {

archiveName = "jmodern-capsule.jar"

from jar // embed our application jar

from { configurations.runtime } // embed dependencies

from(configurations.capsule.collect { zipTree(it) }) { include 'Capsule.class' } // we just need the single Capsule class

manifest {

attributes(

'Main-Class' : 'Capsule',

'Application-Class' : mainClassName,

'Min-Java-Version' : '1.8.0',

'JVM-Args' : run.jvmArgs.join(' '), // copy JVM args from the run task

'System-Properties' : run.systemProperties.collect { k,v -> "$k=$v" }.join(' '), // copy system properties

'Java-Agents' : configurations.quasar.iterator().next().getName()

)

}

}

好了，现在我们输入gradle capsule构建 capsule ，然后运行：

java -jar build/libs/jmodern-capsule.jar

如果你想准确的知道 Capsule 现在在做什么，可以把-jar换成-Dcapsule.log=verbose，但是因为它是一个包括依赖的 capsule ,第一次运行时， Capsule 会解压 JAR 文件到一个缓存目录下  
(这个目录是在当前用户的根文件夹中下.capsule/apps/jmodern.Main)，然后启动一个新通过 capsule 清单文件配置好的 JVM 。如果你已经安装好了 Java7 ，你可以使用 Java7 启动 capsule （通过设置 JAVA\_HOME 环境变量）。虽然 capsule 能在 java7 下启动，但是因为 capsule 指定了最小的 Java 版本是 Java8 (或者是 1.8，同样的意思)， capsule 会寻找 Java8 并且用它来跑我们的应用。

现在讲讲第二方法。我们将创建一个有外部依赖的 capsule 。为了使创建工作简单点，我们先在构建文件中增加一个函数(你不需要理解他；做成 Gradle 的插件会更好，欢迎贡献。但是现在我们手动创建这个 capsule )：

// converts Gradle dependencies to Capsule dependencies

def getDependencies(config) {

return config.getAllDependencies().collect {

def res = it.group + ':' + it.name + ':' + it.version +

(!it.artifacts.isEmpty() ? ':' + it.artifacts.iterator().next().classifier : '')

if(!it.excludeRules.isEmpty()) {

res += "(" + it.excludeRules.collect { it.group + ':' + it.module }.join(',') + ")"

}

return res

}

}

然后，我们改变构建文件中capsule任务，让它能读：

task capsule(type: Jar, dependsOn: classes) {

archiveName = "jmodern-capsule.jar"

from sourceSets.main.output // this way we don't need to extract

from { configurations.capsule.collect { zipTree(it) } }

manifest {

attributes(

'Main-Class' : 'Capsule',

'Application-Class' : mainClassName,

'Extract-Capsule' : 'false', // no need to extract the capsule

'Min-Java-Version' : '1.8.0',

'JVM-Args' : run.jvmArgs.join(' '),

'System-Properties' : run.systemProperties.collect { k,v -> "$k=$v" }.join(' '),

'Java-Agents' : getDependencies(configurations.quasar).iterator().next(),

'Dependencies': getDependencies(configurations.runtime).join(' ')

)

}

}

运行gradle capsule，再次运行：

java -jar build/libs/jmodern-capsule.jar

首次运行， capsule 将会下载我们项目的所有依赖到一个缓存目录下。其他的 capsule 共享这个目录。 相反你不需要把依赖列在 JAR 清单文件中，取而代之，你可以把项目依赖列在 pom 文件中（如果你使用 Maven 做为构建工具，这将特别有用），然后放在 capsule 的根目录。详细信息可以查看 [Capsule 文档](https://github.com/puniverse/capsule/blob/master/README.md)。

最后，因为这篇文章的内容对于任何 JVM 语言都是有用的，所以[这里](https://github.com/puniverse/capsule-avatar-demo)有一个小例子用来示意把一个 Node.js 的应用打包成一个 capsule 。这个小应用使用了 [Avatar](https://avatar.java.net/) ，该项目能够在 JVM 上运行 javascript 应用  
，就像 Nodejs 一样。代码如下：

var http = require('http');

var server = http.createServer(function (request, response) {

response.writeHead(200, {"Content-Type": "text/plain"});

response.end("Hello World\n");

});

server.listen(8000);

console.log("Server running at http://127.0.0.1:8000/");

应用还有两个 Gradle 构建文件。[一个](https://github.com/puniverse/capsule-avatar-demo/blob/master/build1.gradle)用来创建full模式的 capsule ，另外[一个](https://github.com/puniverse/capsule-avatar-demo/blob/master/build2.gradle)用来创建external模式的 capsule 。这个例子示范了打包原生库依赖。创建该 capsule ，运行：

gradle -b build1.gradle capsule

就得到一个包括所有依赖的 capsule 。或者运行下面的命令：

gradle -b build2.gradle capsule

就得到一个不包括依赖的 capsule （里面包括 [Gradle wrapper](https://docs.gradle.org/current/userguide/gradle_wrapper.html)，所以你不需要安装 Gradle ，简单的输入./gradlew就能构建应用）。

运行它，输入下面的命令：

java -jar build/libs/hello-nodejs.jar

[Jigsaw](http://openjdk.java.net/projects/jigsaw/)，原计划在包括在 Java9 中。该项目的意图是解决 Java 部署和一些其它的问题，例如：一个被精减的JVM发行版，减少启动时间([这里](https://www.parleys.com/tutorial/52549d02e4b0a43ac12124be/about)有一个有趣演讲关于 Jigsaw )。同时，对于现代 Java 开发打包和布署，Capsule 是一个非常合适的工具。Capsule 是无状态和不用安装的。

日志

在我们进入 Java 先进的监控特性之前，让我们把日志搞定。据我所知，Java 有大量的日志库，它们都是建立在 JDK 标准库之上。如果你需要日志，用不着想太多，直接使用 [slf4j](http://www.slf4j.org/manual.html) 做为日志 API 。它变成了事实上日志 API 的标准，而且已绑定几乎所有的日志引擎。一但你使用 SLF4J，你可以推迟选择日志引擎时机(你甚至能在部署的时候决定使用哪个日志引擎)。 SLF4J 在运行时选择日志引擎，这个日志引擎可以是任何一个只要做为依赖添加的库。大部分库现在都使用SLF4J，如果开发中有一个库没有使用SLF4J，它会让你把这个库的日志导回SLF4J，然后你就可以再选择你的日志引擎。谈谈选择日志引擎事，如果你想选择一个简单的，那就 JDK 的[java.util.logging](http://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html)。如果你想选择一个重型的、高性能的日志引擎，就选择 [Log4j2](http://logging.apache.org/log4j/2.x/) （除了你感觉真的有必要尝试一下其它的日志引擎）。

现在我们来添加日志到我们的应用中。在依赖部分，我们增加：

compile "org.slf4j:slf4j-api:1.7.7" // the SLF4J API

runtime "org.slf4j:slf4j-jdk14:1.7.7" // SLF4J binding for java.util.logging

如果运行gradle dependencies命令，我们可以看到当前的应用有哪些依赖。就当前来说，我们依赖了 Log4j ，这不是我们想要的。因此好得在build.gradle的配置部分增加一行代码：

all\*.exclude group: "org.apache.logging.log4j", module: "\*"

好了，我们来给我们的应用添加一些日志：

package jmodern;

import co.paralleluniverse.fibers.Fiber;

import co.paralleluniverse.strands.Strand;

import co.paralleluniverse.strands.channels.Channel;

import co.paralleluniverse.strands.channels.Channels;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Main {

static final Logger log = LoggerFactory.getLogger(Main.class);

public static void main(String[] args) throws Exception {

final Channel<Integer> ch = Channels.newChannel(0);

new Fiber<Void>(() -> {

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

Strand.sleep(100);

log.info("Sending {}", i); // log something

ch.send(i);

if (i % 10 == 0)

log.warn("Sent {} messages", i + 1); // log something

}

ch.close();

}).start();

new Fiber<Void>(() -> {

Integer x;

while ((x = ch.receive()) != null)

System.out.println("--> " + x);

}).start().join(); // join waits for this fiber to finish

}

}

然后运行应用（gradle run），你会看见日志打印到标准输出（这个默认设置；我们不打算深入配置日志引擎，你想做的话，可以参考想关文档）。info和warn级的日志都默认输出。日志的输出等级可以在配置文件中设置（现在我们不打算改了），或者一会可以看到，我们在运行时进行修改设置，

用jcmd和jstat进行监控和管理

JDK 中已经包括了几个用于监控和管理的[工具](http://docs.oracle.com/javase/8/docs/technotes/tools/index.html#monitor)，而这里我们只会简短介绍其中的一对工具：[jcmd](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jcmd.html)和 [jstat](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html) 。

为了演示它们，我们要使我们的应用程序别那么快的终止。所以我们把for循环次数从10改成1000000,然后在终端下运行应用gradle run。在另外一个终端中，我们运行jcmd。如果你的JDK安装正确并且jcmd在你的目录中，你会看到下面的信息：

22177 jmodern.Main

21029 org.gradle.launcher.daemon.bootstrap.GradleDaemon 1.11 /Users/pron/.gradle/daemon 10800000 86d63e7b-9a18-43e8-840c-649e25c329fc -XX:MaxPermSize=256m -XX:+HeapDumpOnOutOfMemoryError -Xmx1024m -Dfile.encoding=UTF-8

22182 sun.tools.jcmd.JCmd

上面信息列出了所有正在JVM上运行的程序。再远行下面的命令：

jcmd jmodern.Main help

你会看到打印出了特定 JVM 程序的 jcmd 支持的命令列表。我们来试一下：

jcmd jmodern.Main Thread.print

打印出了 JVM 中所有线程的当前堆栈信息。试一下这个：

jcmd jmodern.Main PerfCounter.print

这将打印出一长串各种 JVM 性能计数器（你问问谷歌这些参数的意思）。你可以试一下其他的命令（如GC.class\_histogram）。

jstat对于 JVM 来说就像 Linux 中的 top ，只有它能查看关于 GC 和 JIT 的活动信息。假设我们应用的 pid 是95098（可以用 jcmd 和 jps 找到这个值）。现在我们运行：

jstat -gc 95098 1000

它将会每 1000 毫秒打印 GC 的信息。看起来像这样：

S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT

80384.0 10752.0 0.0 10494.9 139776.0 16974.0 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

这些数字表示各种 GC 区域当前的容量。想知道每一个的意思，查看 [jsata 文档](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html)。

使用JMX进行监控和管理

JVM 最大的一个优点就是它能在运行时监控和管理时，暴露每一个操作的详细信息。JMX（Java Management Extensions），是 JVM 运行时管理和监控的标准。 JMX 详细说明了 MBeans ，该对象用来暴露有关 JVM 、 JDK 库和 JVM 应用的监控和管理操作方法。 JMX 还定义了连接 JVM 实例的标准方法，包括本地连接和远程连接的方式。还有定义了如何与 MBeans 交互。实际上， jcmd 就是使用 JMX 获得相关的信息的。在本文后面，我们也写一个自己的 MBeans ，但是还是首先来看看内置的 MBeans 如何使用。

当我们的应用运行在一个终端，运行 jvisualvm 命令（该工具是 JDK 的一部分）在另一个终端。这会启动 [VisualVM](https://visualvm.java.net/) 。在我们开始使用之前，还需要装一些插件。打开 Tools->Plugins 菜单，选择可以可以使用的插件。当前的演示，我们只需要VisualVM-MBeans，但是你可能除了 VisualVM-Glassfish 和 BTrace Workbench ，其他的插件都装上。现在在左边面板选择 jmodern.Main ，然后选择监控页。你会看到如下信息：

该监控页把 JMX-MBeans 暴露的使用信息用图表的型式表达出来。我们也可以通过 Mbeans 选项卡选择一些 MBeans （有些需要安装完成插件后才能使用），我们能查看和交互已注册的 MBeans 。例如有个常用的堆图，就在 java.lang/Memory 中（双击属性值展开它）：

现在我们选择 java.util.logging/Logging MBean 。在右边面板中，属性 LoggerNames 会列出所有已注册的 logger ，包括我们添加到 jmodern.Main （双击属性值展开它）：

MBeans 使我们不仅能够探测到监测值，还可以改变这些值，然后调用各种管理操作。选择 Operations 选项卡（在右面板中，位于属性选项卡的右边）。我们现在在运行时通过 JMX-MBean 改变日志等级。在 setLoggerLevel 属性中，第一个地方填上 jmodern.Main ，第二个地方填上 WARNING ，载图如下：

现在，点击 setLoggerLevel 按钮， info 级的日志信息不再会打印出来。如果调整成 SEVERE，就没有信息打印。 VisualVM 对 MBean 都会生成简单的 GUI，不用费力的去写界面。

我们也可以在远程使用 VisualVM 访问我们的应用，只用增加一些系统的设置。在构建文件中的run部分中增加如下代码：

systemProperty "com.sun.management.jmxremote", ""

systemProperty "com.sun.management.jmxremote.port", "9999"

systemProperty "com.sun.management.jmxremote.authenticate", "false"

systemProperty "com.sun.management.jmxremote.ssl", "false"

（在生产环境中，你应该打开安全选项）

正如我们所看到的，除了 MBean 探测， VisualVM 也可以使用 JMX 提供的数据创建自定义监控视图：监控线程状态和当前所有线程的堆栈情况，查看 GC 和通用内存使用情况，执行堆转储和核心转储操作，分析转储堆和核心堆，还有更多的其它功能。因此，在现代 Java 开发中， VisualVM 是最重要的工具之一。这是 VisualVM 跟踪插件提供的监控信息截图:

现代 Java 开发人员有时可能会喜欢一个 CLI 而不是漂亮的 GUI 。 [jmxterm](http://wiki.cyclopsgroup.org/jmxterm/) 提供了一个 CLI 形式的 JMX-MBeans 。不幸的是,它还不支持 Java7 和 Java8 ，但开发人员表示将很快来到(如果没有,我们将发布一个补丁,我们已经有一个分支在做这部分工作了)。

不过，有一件事是肯定的。现代 Java 开发人员喜欢 REST-API (如果没有其他的原因,因为它们无处不在,并且很容易构建 web-GUI )。虽然 JMX 标准支持一些不同的本地和远程连接器，但是标准中没有包括 HTTP 连接器(应该会在 Java9 中)。现在，有一个很好的项目 [Jolokia](https://jolokia.org/)，填补这个空白。它能让我们使用 RESTful 的方式访问 MBeans 。让我们来试一试。将以下代码合并到build.gradle文件中:

configurations {

jolokia

}

dependencies {

runtime "org.jolokia:jolokia-core:1.2.1"

jolokia "org.jolokia:jolokia-jvm:1.2.1:agent"

}

run {

jvmArgs "-javaagent:${configurations.jolokia.iterator().next()}=port=7777,host=localhost"

}

（我发现 Gradle 总是要求对于每一个依赖重新设置 Java agent，这个问题一直困扰我。）

改变构建文件 capsule 任务的 Java-Agents属性，可以让 Jolokia 在 capsule 中可用。代码如下：

'Java-Agents' : getDependencies(configurations.quasar).iterator().next() +

+ " ${getDependencies(configurations.jolokia).iterator().next()}=port=7777,host=localhost",

通过 gradle run 或者 gradle capsule; java -jar build/libs/jmodern-capsule.jar 运行应用，然后打开浏览器输入 http://localhost:7777/jolokia/version 。如果 Jolokia 正常工作，会返回一个JSON。现在我们要查看一下应用的堆使用情况，可以这样做：

curl http://localhost:7777/jolokia/read/java.lang:type\=Memory/HeapMemoryUsage

设置日志等级，你可以这样做：

curl http://localhost:7777/jolokia/exec/java.util.logging:type\=Logging/setLoggerLevel\(java.lang.String,java.lang.String\)/jmodern.Main/WARNING

Jolokia 提供了 Http API ，这就就使用 GET 和 POST 方法进行操作。同时还提供安全访问的方法。需要更多的信息，请查看[文档](http://jolokia.org/reference/html/index.html)。

有了 JolokiaHttpAPI 就能通过Web进行管理。这里有一个[例子](https://jolokia.org/client/javascript-cubism.html)，它使用[Cubism](http://square.github.io/cubism/)为 GUI 进行 JMX MBeans进行管理。还有如 [hawtio](http://hawt.io/) ， JBoss 创建的项目，它使用 JolokiaHttpAPI 构建了一个全功能的网页版的管理应用。与 VisualVM 静态分析功能不同的是， hawatio 意图是为生产环境提供一个持续监控和管理的工具。

写一个自定义的MBeans

写一个 Mbeans 并注册很容易：

package jmodern;

import co.paralleluniverse.fibers.Fiber;

import co.paralleluniverse.strands.Strand;

import co.paralleluniverse.strands.channels.\*;

import java.lang.management.ManagementFactory;

import java.util.concurrent.atomic.AtomicInteger;

import javax.management.MXBean;

import javax.management.ObjectName;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Main {

static final Logger log = LoggerFactory.getLogger(Main.class);

public static void main(String[] args) throws Exception {

final AtomicInteger counter = new AtomicInteger();

final Channel<Object> ch = Channels.newChannel(0);

// create and register MBean

ManagementFactory.getPlatformMBeanServer().registerMBean(new JModernInfo() {

@Override

public void send(String message) {

try {

ch.send(message);

} catch (Exception e) {

throw new RuntimeException(e);

}

}

@Override

public int getNumMessagesReceived() {

return counter.get();

}

}, new ObjectName("jmodern:type=Info"));

new Fiber<Void>(() -> {

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

Strand.sleep(100);

log.info("Sending {}", i); // log something

ch.send(i);

if (i % 10 == 0)

log.warn("Sent {} messages", i + 1); // log something

}

ch.close();

}).start();

new Fiber<Void>(() -> {

Object x;

while ((x = ch.receive()) != null) {

counter.incrementAndGet();

System.out.println("--> " + x);

}

}).start().join(); // join waits for this fiber to finish

}

@MXBean

public interface JModernInfo {

void send(String message);

int getNumMessagesReceived();

}

}

我们添加了一个 JMX-MBean ，让我们监视第二个 fiber 收到消息的数量，也暴露了一个发送操作，能将一条消息进入 channel 。当我们运行应用程序时，我们可以在 VisualVM 中看到监控的属性：

双击，绘图：

在 Operations 选项卡中，使用我们定义在MBean的操作，来发个消息:

使用Metrics进行健康和性能监控

[Metrics](http://metrics.codahale.com/) 一个简洁的监控 JVM 应用性能和健康的现代库，由 Coda Hale 在 Yammer 时创建的。 Metrics 库中包含一些通用的指标集和发布类，如直方图，计时器，统计议表盘等。现在我们来看看如何使用。

首先，我们不需要使用 Jolokia ，把它从构建文件中移除掉，然后添加下面的代码：

compile "com.codahale.metrics:metrics-core:3.0.2"

Metrics 通过 JMX-MBeans 发布指标，你可以将这些指标值写入 CSV 文件，或者做成 RESTful 接口，还可以发布到 Graphite 和 Ganglia  
中。在这里只是简单发布到 JMX （第三部分中讨论到 Dropwizard 时，会使用 HTTP ）。这是我们修改后的 Main.class ：

package jmodern;

import co.paralleluniverse.fibers.Fiber;

import co.paralleluniverse.strands.Strand;

import co.paralleluniverse.strands.channels.\*;

import com.codahale.metrics.\*;

import static com.codahale.metrics.MetricRegistry.name;

import java.util.concurrent.ThreadLocalRandom;

import static java.util.concurrent.TimeUnit.\*;

public class Main {

public static void main(String[] args) throws Exception {

final MetricRegistry metrics = new MetricRegistry();

JmxReporter.forRegistry(metrics).build().start(); // starts reporting via JMX

final Channel<Object> ch = Channels.newChannel(0);

new Fiber<Void>(() -> {

Meter meter = metrics.meter(name(Main.class, "messages" , "send", "rate"));

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

Strand.sleep(ThreadLocalRandom.current().nextInt(50, 500)); // random sleep

meter.mark(); // measures event rate

ch.send(i);

}

ch.close();

}).start();

new Fiber<Void>(() -> {

Counter counter = metrics.counter(name(Main.class, "messages", "received"));

Timer timer = metrics.timer(name(Main.class, "messages", "duration"));

Object x;

long lastReceived = System.nanoTime();

while ((x = ch.receive()) != null) {

final long now = System.nanoTime();

timer.update(now - lastReceived, NANOSECONDS); // creates duration histogram

lastReceived = now;

counter.inc(); // counts

System.out.println("--> " + x);

}

}).start().join(); // join waits for this fiber to finish

}

}

在例子中，使用了 Metrics 记数器。现在运行应用，启动 VisualVM ：

性能分析

性能分析是一个应用是否满足我们对性能要求的关键方法。只有经过性能分析我们才能知道哪一部分代码影响了整体执行速度，然后集中精力只改进这一部分代码。一直以来，Java 都有很好的性能分析工具，它们有的在 IDE 中，有的是一个单独的工具。而最近 Java 的性能分析工具变得更精确和轻量级，这要得益于 HotSpot 把 JRcokit  
JVM 中的代码合并自己的代码中。在这部分讨论的工具不是开源的，在这里讨论它们是因为这些工具已经包括在标准的 OracleJDK 中，你可以在开发环境中自由使用（但是在生产环境中你需要一个商业许可）。

开始一个测试程序，修改后的代码：

package jmodern;

import co.paralleluniverse.fibers.Fiber;

import co.paralleluniverse.strands.Strand;

import co.paralleluniverse.strands.channels.\*;

import com.codahale.metrics.\*;

import static com.codahale.metrics.MetricRegistry.name;

import java.util.concurrent.ThreadLocalRandom;

import static java.util.concurrent.TimeUnit.\*;

public class Main {

public static void main(String[] args) throws Exception {

final MetricRegistry metrics = new MetricRegistry();

JmxReporter.forRegistry(metrics).build().start(); // starts reporting via JMX

final Channel<Object> ch = Channels.newChannel(0);

new Fiber<Void>(() -> {

Meter meter = metrics.meter(name(Main.class, "messages", "send", "rate"));

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

Strand.sleep(ThreadLocalRandom.current().nextInt(50, 500)); // random sleep

meter.mark();

ch.send(i);

}

ch.close();

}).start();

new Fiber<Void>(() -> {

Counter counter = metrics.counter(name(Main.class, "messages", "received"));

Timer timer = metrics.timer(name(Main.class, "messages", "duration"));

Object x;

long lastReceived = System.nanoTime();

while ((x = ch.receive()) != null) {

final long now = System.nanoTime();

timer.update(now - lastReceived, NANOSECONDS);

lastReceived = now;

counter.inc();

double y = foo(x);

System.out.println("--> " + x + " " + y);

}

}).start().join();

}

static double foo(Object x) { // do crazy work

if (!(x instanceof Integer))

return 0.0;

double y = (Integer)x % 2723;

for(int i=0; i<10000; i++) {

String rstr = randomString('A', 'Z', 1000);

y \*= rstr.matches("ABA") ? 0.5 : 2.0;

y = Math.sqrt(y);

}

return y;

}

public static String randomString(char from, char to, int length) {

return ThreadLocalRandom.current().ints(from, to + 1).limit(length)

.mapToObj(x -> Character.toString((char)x)).collect(Collectors.joining());

}

}

foo 方法进行了一些没有意义的计算，不用管它。当运行应用（gradle run）时，你会注意到 Quasar 发出了警告，警告说有一个 fiber 占用了过多的 CPU 时间。为了弄清楚发生了什么，我们开始进行性能分析：

我们使用的分析器能够统计非常精确的信息,同时具有非常低的开销。该工具包括两个组件：第一个是 [Java Flight Recorder](http://docs.oracle.com/javase/8/docs/technotes/guides/jfr/toc.html) 已经嵌入到 HotSpotVM 中。它能记录 JVM 中发生的事件，可以和 jcmd [配合](http://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/run.htm#CHDIDCHG)使用，在这部分我们通过第二个工具来控制它。第二个工具是 JMC ([Java Mission Control](http://docs.oracle.com/javase/8/docs/technotes/guides/jmc/))，也在 JDK 中。它的作用等同于 VisualVM ，只是它比较难用。在这里我们用 JMC 来控制 Java Flight Recorder ，分析记录的信息（我希望 Oracle 能把这部分功能移到 VisualVM 中）。

Flight Recorder 在默认已经加入到应用中，只是不会记录任何信息也不会影响性能。先停止应用，然后把这行代码加到 build.gradle 中的 run ：

jvmArgs "-XX:+UnlockCommercialFeatures", "-XX:+FlightRecorder"

UnlockCommercialFeatures 标志是必须的，因为 Flight Recorder 是商业版的功能，不过可以在开发中自由使用。现在，我们重新启动应用。

在另一个终端中，我们使用 jmc 打开 Mission Control 。在左边的面板中，右击 jmodern.Main，选择 Start Flight Recording… 。在引导窗口中选择 Event settings 下拉框，点击 Profiling - on server ，然后 Next > ，注意不是 Finish 。

接下来，选择 Heap Statistics 和 Allocation Profiling ，点击 Finish ：

JMC 会等 Flight Recorder 记录结束后，打开记录文件进行分析，在那时你可以关掉你的应用。

在 Code 部分的 Hot Methods 选项卡中，可以看出 randomString 是罪魁祸首，它占用了程序执行时间的 90%：

在 Memory 部分的 Garbage Collection 选项卡中，展示了在记录期间堆的使用情况：

在 GC 时间选项卡中，显示了GC的回收情况：

也可以查看内存分配的情况：

应用堆的内容：

Java Flight Recorder 还有[一个不被支持的API](http://hirt.se/blog/?p=277)，能记录应用事件。

高级话题：使用Byteman进行性能分析和调试

像第一部分一样，我们用高级话题来结束本期话题。首先讨论的是用 Byteman 进行性能分析和调试。我在第一部分提到， JVM 最强大的特性之一就是在运行时动态加载代码（这个特性远超本地原生应用加载动态链接库）。不只这个，JVM 还给了我们来回变换运行时代码的能力。

JBoss 开发的 [Byteman](https://www.jboss.org/byteman.html) 工具能充分利用 JVM 的这个特性。 Byteman 能让我们在运行应用时注入跟踪、调试和性能测试相关代码。这个话题之所以是一个高级话题，是因为当前 Byteman 只支持 Java7 ，对 Java8 的支持还不可靠，需要打补丁才能工作。这个项目当前开发活跃，但是正在落后。因此在这里使用一些 Byteman 非常基础的代码。

这是主类：

package jmodern;

import java.util.concurrent.ThreadLocalRandom;

public class Main {

public static void main(String[] args) throws Exception {

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

System.out.println("Calling foo");

foo(i);

}

}

private static String foo(int x) throws InterruptedException {

long pause = ThreadLocalRandom.current().nextInt(50, 500);

Thread.sleep(pause);

return "aaa" + pause;

}

}

foo 模拟调用服务器操作，这些操作要花费一定时间进行。

接下来，把下面的代码合并到构建文件中：

configurations {

byteman

}

dependencies {

byteman "org.jboss.byteman:byteman:2.1.4.1"

}

run {

jvmArgs "-javaagent:${configurations.byteman.iterator().next()}=listener:true,port:9977"

// remove the quasar agent

}

想在 capsule 中试一试 Byteman 使用，在构建文件中改一下 Java-Agents 属性：

'Java-Agents' : "${getDependencies(configurations.byteman).iterator().next()}=listener:true,port:9977",

现在，从[这里](http://byteman.jboss.org/downloads/)下载 Byteman ，因为需要使用 Byteman 中的命令行工具，解压文件，设置环境变量 BYTEMAN\_HOME 指向 Byteman 的目录。

启动应用gradle run。打印结果如下：

Calling foo

Calling foo

Calling foo

Calling foo

Calling foo

我们想知道每次调用 foo 需要多长有时间，但是我们没有测量并记录这个信息。现在使用 Byteman 在运行时插入相关日志记录信息。

打开编辑器，在项目目录中创建文件 jmodern.btm ：

RULE trace foo entry

CLASS jmodern.Main

METHOD foo

AT ENTRY

IF true

DO createTimer("timer")

ENDRULE

RULE trace foo exit

CLASS jmodern.Main

METHOD foo

AT EXIT

IF true

DO traceln("::::::: foo(" + $1 + ") -> " + $! + " : " + resetTimer("timer") + "ms")

ENDRULE

上面列的是 Byteman rules ，就是当前我们想应用在程序上的 rules。我们在另一个终端中运行命令：

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977 jmodern.btm

之后，运行中的应用打印信息：

Calling foo

::::::: foo(152) -> aaa217 : 217ms

Calling foo

::::::: foo(153) -> aaa281 : 281ms

Calling foo

::::::: foo(154) -> aaa282 : 283ms

Calling foo

::::::: foo(155) -> aaa166 : 166ms

Calling foo

::::::: foo(156) -> aaa160 : 161ms

查看哪个 rules 正在使用：

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977

卸载 Byteman 脚本：

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977 -u

运行该命令之后，注入的日志代码就被移出。

Byteman 是在 JVM 灵活代码变换的基础上创建的一个相当强大的工具。你可以使用这个工具来检查变量和日志事件，插入延迟代码等操作，甚至还可以轻松设置一些自定义的 Byteman 行为。更多的信息，参考[Byteman documentation](http://byteman.jboss.org/documentation/)。

高级话题：使用JMH进行基准测试

当代硬件构架和编译技术的进步使考察代码性能的唯一方法就是基准测试。一方面，由于现代 CPU 和编译器非常聪明（可以看[这里](http://www.infoq.com/presentations/click-crash-course-modern-hardware)），它能为代码（可以是 c，甚至是汇编）自动地创建一个理论上非常高效的运行环境，就像 90 年代末一些游戏程序员做的那些非常不可思议的事一样。另一方面，正是因为聪明的 CPU 和编译器，让微基准测试非常困难，因为这样的话，代码的执行速度非常依赖具体的执行环境（如：代码速度受 CPU 缓存状态的影响，而 CPU 缓存状态又受其它线程操作的影响）。而对一个 Java 进行微基准测试又会更加的困难，因为 JVM 有 JIT ，而 JIT 是一个以性能优化为导向的编绎器，它能在运行时影响代码执行的上下文环境。因此在 JVM 中，同一段代码在微基准测试和实际程序中执行时间可能不一样，有时可能快，有时也可能慢。

[JMH](http://openjdk.java.net/projects/code-tools/jmh/) 是由 Oracle 创建的 Java 基准测试工具。你可以相信由 JMH 测试出来的数据（可以看看这个由 JMH 主要作者Aleksey Shipilev的[演讲](http://medianetwork.oracle.com/video/player/2630310904001)，[幻灯片](http://shipilev.net/pub/talks/jvmls-July2013-benchmarking.pdf)）。 Google 也做了一个基准测试的工具叫 Caliper，但是这个工具很不成熟，有时还会有错误的结果。不要使用它。

我们马上来使用一下 JMH ，但是在这之前首先有一个忠告：过早优化是万恶之源。在基测试中，两种算法或者数据结构中，一种比另一种快 100 倍，而这个算法只占你应用运行时间的 1％ ，这样测试是没有意义的。因为就算你把这个算法改进的非常快行但也只能加快你的应用 2% 时间。基准测试只能是已经对应用进行了性能测试后，用来发现哪一个小部分改变能得到最大的加速成果。

增加依赖：

testCompile 'org.openjdk.jmh:jmh-core:0.8'

testCompile 'org.openjdk.jmh:jmh-generator-annprocess:0.8'

然后增加bench任务：

task bench(type: JavaExec, dependsOn: [classes, testClasses]) {

classpath = sourceSets.test.runtimeClasspath // we'll put jmodern.Benchamrk in the test directory

main = "jmodern.Benchmark";

}

最后，把测试代码放到 src/test/java/jmodern/Benchmark.java 文件中。我之前提到过 90 年代的游戏程序员，是为了说明古老的技术现在仍然有用，这里我们测试一个开平方根的计算，使用[fast inverse square root algorithm](https://en.wikipedia.org/wiki/Fast_inverse_square_root)（平方根倒数速算法，这是 90 年代的程序）：

package jmodern;

import java.util.concurrent.TimeUnit;

import org.openjdk.jmh.annotations.\*;

import org.openjdk.jmh.profile.\*;

import org.openjdk.jmh.runner.Runner;

import org.openjdk.jmh.runner.options.OptionsBuilder;

import org.openjdk.jmh.runner.parameters.TimeValue;

@State(Scope.Thread)

@BenchmarkMode(Mode.AverageTime)

@OutputTimeUnit(TimeUnit.NANOSECONDS)

public class Benchmark {

public static void main(String[] args) throws Exception {

new Runner(new OptionsBuilder()

.include(Benchmark.class.getName() + ".\*")

.forks(1)

.warmupTime(TimeValue.seconds(5))

.warmupIterations(3)

.measurementTime(TimeValue.seconds(5))

.measurementIterations(5)

.build()).run();

}

private double x = 2.0; // prevent constant folding

@GenerateMicroBenchmark

public double standardInvSqrt() {

return 1.0/Math.sqrt(x);

}

@GenerateMicroBenchmark

public double fastInvSqrt() {

return invSqrt(x);

}

static double invSqrt(double x) {

double xhalf = 0.5d \* x;

long i = Double.doubleToLongBits(x);

i = 0x5fe6ec85e7de30daL - (i >> 1);

x = Double.longBitsToDouble(i);

x = x \* (1.5d - xhalf \* x \* x);

return x;

}

}

随便说一下，像第一部分中讨论的 Checker 一样， JMH 使用使用注解处理器。但是不同 Checker ， JMH 做的不错，你能在所有的 IDE 中使用它。在下面的图中，我们可以看到， NetBeans 中，一但忘加 @State 注解， IDE 就会报错：

写入命令 gradle bench ，运行基准测试。会得到以下结果：

Benchmark Mode Samples Mean Mean error Units

j.Benchmark.fastInvSqrt avgt 10 2.708 0.019 ns/op

j.Benchmark.standardInvSqrt avgt 10 12.824 0.065 ns/op

很漂亮吧，但是你得知道 fast-inv-sqrt 结果是一个粗略近似值， 只在需要大量开平方的地方适用（如图形计算中）。

在下面的例子中， JMH 用来报到 GC 使用的时间和方法栈的调用时间：

package jmodern;

import java.util.\*;

import java.util.concurrent.\*;

import org.openjdk.jmh.annotations.\*;

import org.openjdk.jmh.profile.\*;

import org.openjdk.jmh.runner.Runner;

import org.openjdk.jmh.runner.options.OptionsBuilder;

import org.openjdk.jmh.runner.parameters.TimeValue;

@State(Scope.Thread)

@BenchmarkMode(Mode.AverageTime)

@OutputTimeUnit(TimeUnit.NANOSECONDS)

public class Benchmark {

public static void main(String[] args) throws Exception {

new Runner(new OptionsBuilder()

.include(Benchmark.class.getName() + ".\*")

.forks(2)

.warmupTime(TimeValue.seconds(5))

.warmupIterations(3)

.measurementTime(TimeValue.seconds(5))

.measurementIterations(5)

.addProfiler(GCProfiler.class) // report GC time

.addProfiler(StackProfiler.class) // report method stack execution profile

.build()).run();

}

@GenerateMicroBenchmark

public Object arrayList() {

return add(new ArrayList<>());

}

@GenerateMicroBenchmark

public Object linkedList() {

return add(new LinkedList<>());

}

static Object add(List<Integer> list) {

for (int i = 0; i < 4000; i++)

list.add(i);

return list;

}

}

这是 JMH 的打印出来的信息：

Iteration 3: 33783.296 ns/op

GC | wall time = 5.000 secs, GC time = 0.048 secs, GC% = 0.96%, GC count = +97

|

Stack | 96.9% RUNNABLE jmodern.generated.Benchmark\_arrayList.arrayList\_AverageTime\_measurementLoop

| 1.8% RUNNABLE java.lang.Integer.valueOf

| 1.3% RUNNABLE java.util.Arrays.copyOf

| 0.0% (other)

|

JMH 是一个功能非常丰富的框架。不幸的是，在文档方面有些薄弱，不过有一个相当好代码示例[教程](http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/)，用来展示 Java 中微基测试的陷阱。你也可以读读[这篇](http://psy-lob-saw.blogspot.co.il/2013/04/writing-java-micro-benchmarks-with-jmh.html)介绍 JMH 的入门文章。

目前为止我们已经学了什么？

在这篇文章中，我们讨论了在 JVM 管理、监控和性能测试方面最好的几个工具。 JVM 除了很好的性能外，它还非常深思熟虑地提供了能深度洞察它运行状态的能力，这就是我不会用其它的技术来取代 JVM 做为重要的、长时间运行的服务器端应用平台的主要原因。  
此外，我们还见识到了当使用 Byteman 等工具修改运行时代码时， JVM 是多么强大。

我们还介绍了 Capsule ，一个轻量级的、单文件、无状态、不用安装的部署工具。另外，通过一个公开或者组织内部的 Maven 仓库，它还支持整个Java应用自动升级，或者还是仅仅升级一个依赖库。

在[第三部分](http://blog.paralleluniverse.co/2014/05/15/modern-java-pt3/)中，我们将讨论如何使用 [Dropwizard](http://www.dropwizard.io/) ， [Comsat](http://docs.paralleluniverse.co/comsat/) , [Web Actors](http://blog.paralleluniverse.co/2014/01/28/web-actors-1/) ,和 DI 来写一个轻量级、可扩展的http服务。

原文地址：[An Opinionated Guide to Modern Java, Part 2: Deployment, Monitoring & Management, Profiling and Benchmarking](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/)

水平有限，如果看不懂请直接看英文版。

# An Opinionated Guide to Modern Java, Part 2: Deployment, Monitoring & Management, Profiling and Benchmarking

http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/

An Opinionated Guide to Modern Java, Part 2: Deployment, Monitoring & Management, Profiling and Benchmarking

**By Ron**

This is part 2 in a three-part series: [part 1](http://blog.paralleluniverse.co/2014/05/01/modern-java/), [part 3](http://blog.paralleluniverse.co/2014/05/15/modern-java-pt3/)

Welcome to part 2 of the OGMJ. In [part 1](http://blog.paralleluniverse.co/2014/05/01/modern-java/) we presented new Java language features, libraries and tools, that make Java into a much more lightweight development environment: new build tools, easier docs, expressive code and lightweight concurrency. In this post, we’ll go beyond the code to discuss Java operations, namely deployment, monitoring and management, profiling and benchmarking. Even though the examples will be in Java, most of what we discuss in this post is relevant to all JVM languages as much as it is for Java, the language.

But before we begin, I’d like to shortly go over some of the responses to the previous post raised by readers, and clarify a couple of things. It turns out that the most contentious recommendation I made in part 1 was the build tool. I wrote, “the modern Java developer uses [Gradle](http://www.gradle.org/)”. Some readers took issue with that, and made a good case for [Maven](http://maven.apache.org/). While I personally prefer Gradle’s nice DSL and the ability to use imperative code for non-common build operations, I can understand the preference for the fully declarative Maven, even if it requires lots of plugins. The modern Java developer, then, might prefer Maven to Gradle. I would like to say, though, that in order to use Gradle one does not need to know Groovy, even if one wishes to do some non-standard stuff; I don’t. I just learned a few useful Groovy expressions that I found in Gradle examples online.

Also, some readers took my use of JUnit and Guava in the example to mean I endorse them. Well, I do. [Guava](https://code.google.com/p/guava-libraries/) is a very useful library, and [JUnit](http://junit.org/) is a fine unit-test framework. TestNG is a fine unit-testing framework as well, but JUnit is so ubiquitous that there is little reason to choose something else, even if another library has some advantages. Also, the unit test example used [Hamcrest](http://hamcrest.org/JavaHamcrest/) matchers. One reader pointed me at [AssertJ](http://joel-costigliola.github.io/assertj/index.html), which looks like a very nice alternative to Hamcrest.

It is important to understand that this guide is not intended to be comprehensive. There are so many good Java libraries out there that we can’t possibly explore them all. My intention is to give a taste of what’s possible with modern Java.

Some readers expressed their preference to short Javadoc comments that don’t necessarily fill out all of the fields in the Javadoc “standard form”. For example, this:

*/\*\**

*\* Returns the result*

*\*/*

**int** getResult**();**

is preferable to:

*/\*\**

*\* This method returns the result.*

*\* @return the result*

*\*/*

**int** getResult**();**

With that I wholeheartedly agree. My example simply demonstrated mixing Markdown with standard Javadoc taglets, and was not intended as a guideline.

Finally, a few words regarding Android. While Android can execute, via a series of transformations, code written in Java (and perhaps some other JVM languages), Android is not a JVM, and in fact Android, both officially and in practice, is not Java (that’s the result of two multinational corporations not being able to reach a licensing agreement). Because Android is not Java, what we covered in part 1 may or may not apply to it, and because Android does not have a JVM, little if anything in this post applies to it.

Now back to our business.

Modern Java Packaging and Deployment

For those of you unfamiliar with the Java ecosystem, Java (or any JVM language) source files, are compiled into .class files (essentially Java binaries), one for each class. The basic mechanism of packaging class files is bundling them (this is normally done by the build tool or IDE) into JAR (Java Archive) files, which are Java binary packages. JARs are just ZIP files containing class files, and an additional manifest file describing the contents, and possibly containing other information about the distribution (the manifest file also contains the electronic signatures in [signed JARs](http://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html)). If you package an application (as opposed to a library) in a JAR, the manifest can point to the app’s main class, in which case the application can be launched with the command java -jar app.jar; that’s called an executable JAR.

Java libraries are packaged into JARs, and then deployed into Maven repositories (those are used by practically all JVM build tools – not just Maven). Maven repositories manage binaries’ versioning and dependencies (when you request a library from a Maven repository, you can ask for all its transitive dependencies as well)[1](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fn:1). Open source JVM libraries are often hosted at the [Central Repository](http://search.maven.org/), or other similar public repositories, and organizations manage their own private Maven repositories with tools like [Artifactory](http://www.jfrog.com/home/v_artifactory_opensource_overview) or [Nexus](http://www.sonatype.com/nexus). You can even host your own [Maven repository on GitHub](http://cemerick.com/2010/08/24/hosting-maven-repos-on-github/). But Maven repos are normally accessed (by your build tool) at build time, and normally host libraries rather than executables.

Java web applications have traditionally been run in application servers, or servlet containers. Those containers can run multiple web applications, loading and unloading apps on demand. Java web applications are deployed to servlet containers in WAR (Web Archive) files, which are really JARs whose contents are arranged in some standard way, and contain additional configurations. But, as we’ll see in part 3 of this post series, as far as modern Java is concerned, [Java application servers are dead](http://www.slideshare.net/ewolff/java-application-servers-are-dead).

Java desktop applications are often packaged and deployed as platform specific binaries, bundled along with their own JVM. The JDK contains a [tool to do just that](http://docs.oracle.com/javafx/2/deployment/self-contained-packaging.htm) ([here’s](https://netbeans.org/kb/docs/java/native_pkg.html) how to use it in NetBeans), and a third-party tool called [Packr](https://github.com/libgdx/packr) provides similar functionality. This mechanism is great for desktop apps and games, but not what we want for server software: the packages tend to be very big and require installation by the user. In addition, because they bundle a copy of the JVM, they cannot be patched with security and performance upgrades.

What we want is a simple, lightweight, fire-and-forget packaging and deployment tool for server-side code. Preferably we would like to take advantage of executable JARs’ simplicity and platform independence. But executable JARs have several deficiencies. Every library is usually packaged into its own JAR, and merging all dependencies into a single JAR, might cause collisions, especially with packaged resources (non-class files). Also, a native library can’t just be dropped into the JAR, and, perhaps most importantly, configuring the JVM (setting heap sizes etc.) falls to the user, and must be done at the command line. Tools like [Maven’s Shade plugin](http://maven.apache.org/plugins/maven-shade-plugin/) or [Gradle’s Shadow plugin](https://github.com/johnrengelman/shadow) solve the collision issue, and [One-Jar](http://one-jar.sourceforge.net/) also supports native libraries, but both might interfere with the application in subtle ways, and neither solve the JVM configuration problem. Gradle can bundle the application in a ZIP and generate os-specific launch scripts to configure the JVM, but that approach requires installation, and we can go much more lightweight than that. Also, with a powerful, ubiquitous resource like Maven repositories at our disposal, it would be a shame not to take advantage of them.

This blog post series is meant to be about how easy and fun it is to work with modern Java (without sacrificing any of it power), but when I looked for a fun, easy and lightweight way to package, distribute and deploy server-side Java apps, I came up empty handed. And so [Capsule](https://github.com/puniverse/capsule)[2](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fn:2) was born (if you know of any alternatives, please let me know).

Capsule uses the nice platform independence of executable JARs – but without their deficiencies – and (optionally) combines it with the power and convenience of Maven repositories. A capsule is a JAR that contains all or some of the Capsule project’s classes, and a manifest with deployment configuration options. When launched (with a simple java -jar app.jar), the capsule will do all or some of the following: extract the JAR into a cache directory, download and cache Maven dependencies, find an appropriate JVM installation, and configure and run the application in a new JVM process.

Now let’s take Capsule for a spin. We’ll begin with our JModern project that we created in [part 1](http://blog.paralleluniverse.co/2014/05/01/modern-java/). This is our build.gradle file:

apply plugin: 'java'

apply plugin: 'application'

sourceCompatibility **=** '1.8'

mainClassName **=** 'jmodern.Main'

repositories **{**

mavenCentral**()**

**}**

configurations **{**

quasar

**}**

dependencies **{**

compile "co.paralleluniverse:quasar-core:0.5.0:jdk8"

compile "co.paralleluniverse:quasar-actors:0.5.0"

quasar "co.paralleluniverse:quasar-core:0.5.0:jdk8"

testCompile 'junit:junit:4.11'

**}**

run **{**

jvmArgs "-javaagent:${configurations.quasar.iterator().next()}"

**}**

and here’s our jmodern.Main class:

**package** jmodern**;**

**import** co.paralleluniverse.fibers.Fiber**;**

**import** co.paralleluniverse.strands.Strand**;**

**import** co.paralleluniverse.strands.channels.Channel**;**

**import** co.paralleluniverse.strands.channels.Channels**;**

**public** **class** **Main** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**final** Channel**<**Integer**>** ch **=** Channels**.**newChannel**(**0**);**

**new** Fiber**<**Void**>(()** **->** **{**

**for** **(int** i **=** 0**;** i **<** 10**;** i**++)** **{**

Strand**.**sleep**(**100**);**

ch**.**send**(**i**);**

**}**

ch**.**close**();**

**}).**start**();**

**new** Fiber**<**Void**>(()** **->** **{**

Integer x**;**

**while((**x **=** ch**.**receive**())** **!=** **null)**

System**.**out**.**println**(**"--> " **+** x**);**

**}).**start**().**join**();** *// join waits for this fiber to finish*

**}**

**}**

To test if our program is working correctly, we’ll try a gradle run.

Now, let’s package it into a capsule. In the build file, we’ll add a capsule configuration. Then, we’ll add the following line to our dependencies:

capsule "co.paralleluniverse:capsule:0.3.1"

There are two basic ways to create a capsule (although you can mix them both). The first is to embed all of the dependencies in the capsule, and the second is to let the capsule download them when first launched. We’ll try the first approach – the “full” capsule – first. We’ll add the following to the bottom of our build file:

task **capsule(**type: Jar**,** dependsOn: jar**)** **{**

archiveName **=** "jmodern-capsule.jar"

from jar *// embed our application jar*

from **{** configurations**.**runtime **}** *// embed dependencies*

from**(**configurations**.**capsule**.**collect **{** zipTree**(**it**)** **})** **{** include 'Capsule.class' **}** *// we just need the single Capsule class*

manifest **{**

attributes**(**

'Main-Class' **:** 'Capsule'**,**

'Application-Class' **:** mainClassName**,**

'Min-Java-Version' **:** '1.8.0'**,**

'JVM-Args' **:** run**.**jvmArgs**.**join**(**' '**),** *// copy JVM args from the run task*

'System-Properties' **:** run**.**systemProperties**.**collect **{** k**,**v **->** "$k=$v" **}.**join**(**' '**),** *// copy system properties*

'Java-Agents' **:** configurations**.**quasar**.**iterator**().**next**().**getName**()**

**)**

**}**

**}**

Now let’s build the capsule with gradle capsule, and run it:

java -jar build/libs/jmodern-capsule.jar

If you want to see exactly what Capsule is doing, preface -jar with -Dcapsule.log=verbose, but, because it’s a capsule with embedded dependencies, Capsule will extract the JAR into a cache directory (.capsule/apps/jmodern.Main in the user’s home directory) – the first time it’s run – and then launch a new JVM, configured according to the capsule’s manifest. If you have a Java 7 installation, you can try launching the capsule under Java 7 (by setting the JAVA\_HOME environment variable to Java 7’s home directory). Even though it’s launched under Java 7, because the capsule specifies a minimum Java version of 8 (or 1.8, which is the same thing), the capsule will find the Java 8 installation and use it to run our app.

Now for the second approach. We’ll create a capsule with external dependencies. To make the capsule creation easier, we’ll first add a function to our build file (you don’t need to understand it; a Gradle plugin will make this a lot easier – contributions are welcome, BTW – but for now we’ll create the capsule “manually”):

*// converts Gradle dependencies to Capsule dependencies*

**def** **getDependencies(**config**)** **{**

**return** config**.**getAllDependencies**().**collect **{**

**def** res **=** it**.**group **+** ':' **+** it**.**name **+** ':' **+** it**.**version **+**

**(!**it**.**artifacts**.**isEmpty**()** **?** ':' **+** it**.**artifacts**.**iterator**().**next**().**classifier **:** ''**)**

**if(!**it**.**excludeRules**.**isEmpty**())** **{**

res **+=** "(" **+** it**.**excludeRules**.**collect **{** it**.**group **+** ':' **+** it**.**module **}.**join**(**','**)** **+** ")"

**}**

**return** res

**}**

**}**

Then we’ll change the capsule task in the build file to read:

task **capsule(**type: Jar**,** dependsOn: classes**)** **{**

archiveName **=** "jmodern-capsule.jar"

from sourceSets**.**main**.**output *// this way we don't need to extract*

from **{** configurations**.**capsule**.**collect **{** zipTree**(**it**)** **}** **}**

manifest **{**

attributes**(**

'Main-Class' **:** 'Capsule'**,**

'Application-Class' **:** mainClassName**,**

'Extract-Capsule' **:** 'false'**,** *// no need to extract the capsule*

'Min-Java-Version' **:** '1.8.0'**,**

'JVM-Args' **:** run**.**jvmArgs**.**join**(**' '**),**

'System-Properties' **:** run**.**systemProperties**.**collect **{** k**,**v **->** "$k=$v" **}.**join**(**' '**),**

'Java-Agents' **:** getDependencies**(**configurations**.**quasar**).**iterator**().**next**(),**

'Dependencies'**:** getDependencies**(**configurations**.**runtime**).**join**(**' '**)**

**)**

**}**

**}**

Now let’s build the new capsule with gradle capsule, and run it again:

java -jar build/libs/jmodern-capsule.jar

The first time it’s run, the capsule will download all of our project’s dependencies into a cache directory, where they will be shared by other capsules using them. Instead of listing the dependencies in the JAR manifest, you can place your project’s pom file (especially useful if you’re using Maven as a build tool), into the capsule’s root. See the [Capsule docs](https://github.com/puniverse/capsule/blob/master/README.md) for details.

Finally, because this post is applicable to any JVM language, [here’s a tiny project](https://github.com/puniverse/capsule-avatar-demo) packaging a Node.js app in a capsule. The app uses [Project Avatar](https://avatar.java.net/), which allows running Node.js-like JavaScript applications on the JVM[3](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fn:3). It consists of this source code:

**var** http **=** require('http');

**var** server **=** http.createServer(**function** (request, response) {

response.writeHead(200, {"Content-Type": "text/plain"});

response.end("Hello World\n");

});

server.listen(8000);

console.log("Server running at http://127.0.0.1:8000/");

And two Gradle build files. [One](https://github.com/puniverse/capsule-avatar-demo/blob/master/build1.gradle) creating a “full” capsule (with embedded dependencies), and [the other](https://github.com/puniverse/capsule-avatar-demo/blob/master/build2.gradle) packaging a capsule with external dependencies. This example demonstrates a capsule with native library dependencies. To build the capsule, run

gradle -b build1.gradle capsule

for a full capsule, or:

gradle -b build2.gradle capsule

for a capsule with external dependencies (the project includes a [Gradle wrapper](http://www.gradle.org/docs/current/userguide/gradle_wrapper.html), so you don’t even need Gradle installed to build it; simply type./gradlew instead of gradle to build).

To run:

java -jar build/libs/hello-nodejs.jar

[Project Jigsaw](http://openjdk.java.net/projects/jigsaw/), scheduled for inclusion in Java 9, is intended to fix Java deployment and a host of other issues, like stripped JVM distributions, reduced startup time ([this](http://parleys.com/play/52549d02e4b0a43ac12124be/about) is an interesting talk about Jigsaw). In the meantime, Capsule is a lean, and quite satisfactory solution for modern Java packaging and deployment. Capsule is stateless and installation-free.

Logging

Before we get into Java’s more advanced monitoring features, let’s get logging out of the way. Java is known to have a bazillion – give or take – logging libraries, on top of the one built into the JDK. Don’t think about that too much. If you need logging, use [SLF4J](http://www.slf4j.org/manual.html) as the logging API, period. It’s become the de-facto logging standard, and it has bindings for virtually all logging engines. Once you use SLF4J, you can leave the choice of a logging engine for later (you can even pick an engine at deployment time). SLF4J chooses a logging engine at runtime, based on whatever relevant JARs are included as dependencies. Most libraries now use SLF4J, and if one of your dependencies doesn’t, SLF4J lets you pipe calls to any logging libraries back to SLF4J, and from there to your engine of choice. Speaking of choosing a logging engine, if your needs are simple, pick the JDK’s[java.util.logging](http://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html). For heavy-duty, high-performance logging, pick the [Log4j 2](http://logging.apache.org/log4j/2.x/) (unless you feel really tied to some other logging engine).

Let’s add logging to our app. To our dependencies, we’ll add:

compile "org.slf4j:slf4j-api:1.7.7" *// the SLF4J API*

runtime "org.slf4j:slf4j-jdk14:1.7.7" *// SLF4J binding for java.util.logging*

If we run gradle dependencies we see that our app’s dependencies, in turn, depend on Log4j, which we don’t want for the purpose of this demonstration, add the following line to the build.gradle’s configuration section:

all**\*.**exclude group: "org.apache.logging.log4j"**,** module: "\*"

Finally, we’ll add some logging to our code:

**package** jmodern**;**

**import** co.paralleluniverse.fibers.Fiber**;**

**import** co.paralleluniverse.strands.Strand**;**

**import** co.paralleluniverse.strands.channels.Channel**;**

**import** co.paralleluniverse.strands.channels.Channels**;**

**import** org.slf4j.Logger**;**

**import** org.slf4j.LoggerFactory**;**

**public** **class** **Main** **{**

**static** **final** Logger log **=** LoggerFactory**.**getLogger**(**Main**.**class**);**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**final** Channel**<**Integer**>** ch **=** Channels**.**newChannel**(**0**);**

**new** Fiber**<**Void**>(()** **->** **{**

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

Strand**.**sleep**(**100**);**

log**.**info**(**"Sending {}"**,** i**);** *// log something*

ch**.**send**(**i**);**

**if** **(**i **%** 10 **==** 0**)**

log**.**warn**(**"Sent {} messages"**,** i **+** 1**);** *// log something*

**}**

ch**.**close**();**

**}).**start**();**

**new** Fiber**<**Void**>(()** **->** **{**

Integer x**;**

**while** **((**x **=** ch**.**receive**())** **!=** **null)**

System**.**out**.**println**(**"--> " **+** x**);**

**}).**start**().**join**();** *// join waits for this fiber to finish*

**}**

**}**

If you now run the app (gradle run), you’ll see the log statements printed to the standard output (that’s the default; we’re not going to get into configuring log files – refer to your logging engine docs for that). Both “info” and “warn” logs are printed by default. The logging level can be set in the log configuration (which, again, we’re not going to do now), or at runtime, as we’ll soon see.

Monitoring and Management with jcmd and jstat

The JDK includes several [command line monitoring and management tools](http://docs.oracle.com/javase/8/docs/technotes/tools/index.html#monitor), but here we’ll only shortly cover a couple: [jcmd](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jcmd.html) and [jstat](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html).

In order to play with them, we’re going to have to make our app not terminate so quickly, so change the for loop limit in the first fiber from 10 to, say, 1000000, and run it in a terminal with gradle run. In another terminal window, run jcmd. If your JDK is installed correctly and jcmd is on your path, you’ll see something like this:

22177 jmodern.Main

21029 org.gradle.launcher.daemon.bootstrap.GradleDaemon 1.11 /Users/pron/.gradle/daemon 10800000 86d63e7b-9a18-43e8-840c-649e25c329fc -XX:MaxPermSize=256m -XX:+HeapDumpOnOutOfMemoryError -Xmx1024m -Dfile.encoding=UTF-8

22182 sun.tools.jcmd.JCmd

It’s a list of all currently running JVM processes. Now, run:

jcmd jmodern.Main help

You will see a list of jcmd commands that the particular JVM process supports. Let’s try:

jcmd jmodern.Main Thread.print

This will print out the current stack trace for all threads running in the JVM. Now try:

jcmd jmodern.Main PerfCounter.print

This will print out a long list of various JVM performance counters (you’ll have to Google for their meaning). You can now try some of the other commands (like GC.class\_histogram).

jstat is like top for the JVM, only it displays information about GC and JIT activity. Suppose our app’s pid is 95098 (you can find it by runningjcmd or jps in the shell). Running

jstat -gc 95098 1000

Will print GC information every 1000 milliseconds. It will look like this:

S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT

80384.0 10752.0 0.0 10494.9 139776.0 16974.0 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

80384.0 10752.0 0.0 10494.9 139776.0 16985.1 148480.0 125105.4 ? ? 65 1.227 8 3.238 4.465

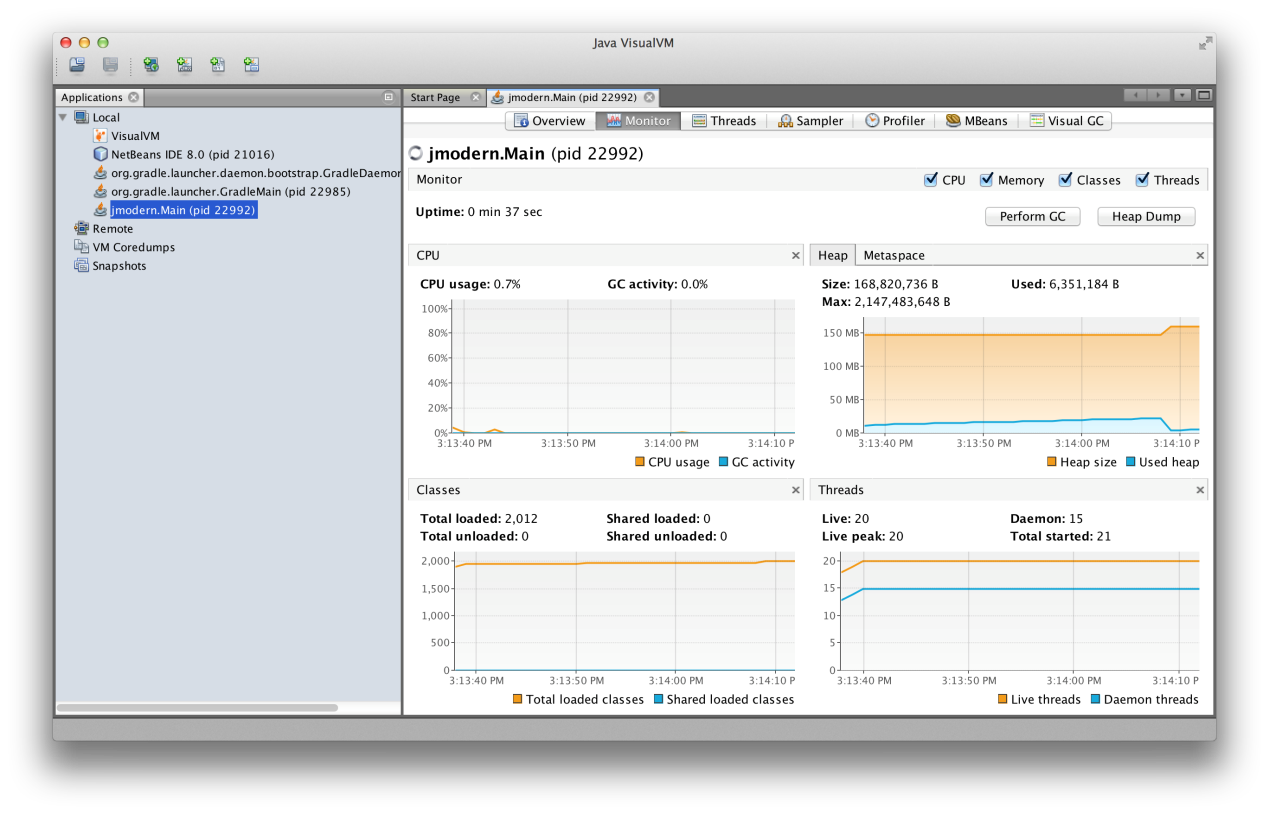
The numbers are the current capacity of various GC regions. To learn more about what each means, see the [jstat documentation](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html).

Both jcmd and jstat can connect to JVMs running on remote machines; see their documentation for details ([jcmd](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jcmd.html), [jstat](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html)).

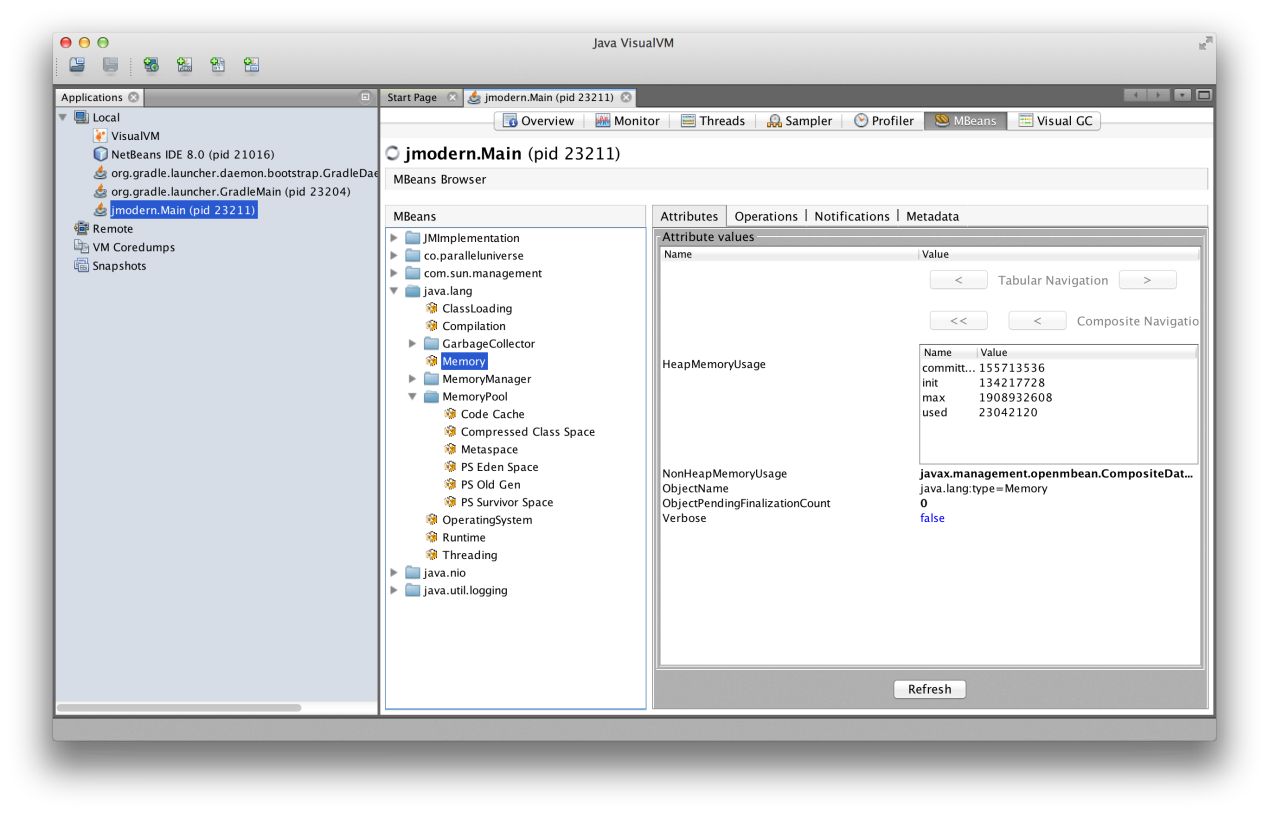
Monitoring and Management with JMX

One of the JVM’s greatest strengths is how it exposes every single detail of its operation – and that of the standard libraries – for runtime monitoring and management. JMX (Java Management Extensions), is a runtime management and monitoring standard. JMX specifies simple Java objects, called MBeans, that expose monitoring and management operations of the JVM itself, the JDK libraries, and any JVM application. JMX also specifies standard ways of connecting to JVM instances – either locally or remotely – to interact with the MBeans. In fact, jcmd gets much of its information with JMX. We will see how to write our own MBeans in the next section, but let’s first see how we can examine the baked-in ones.

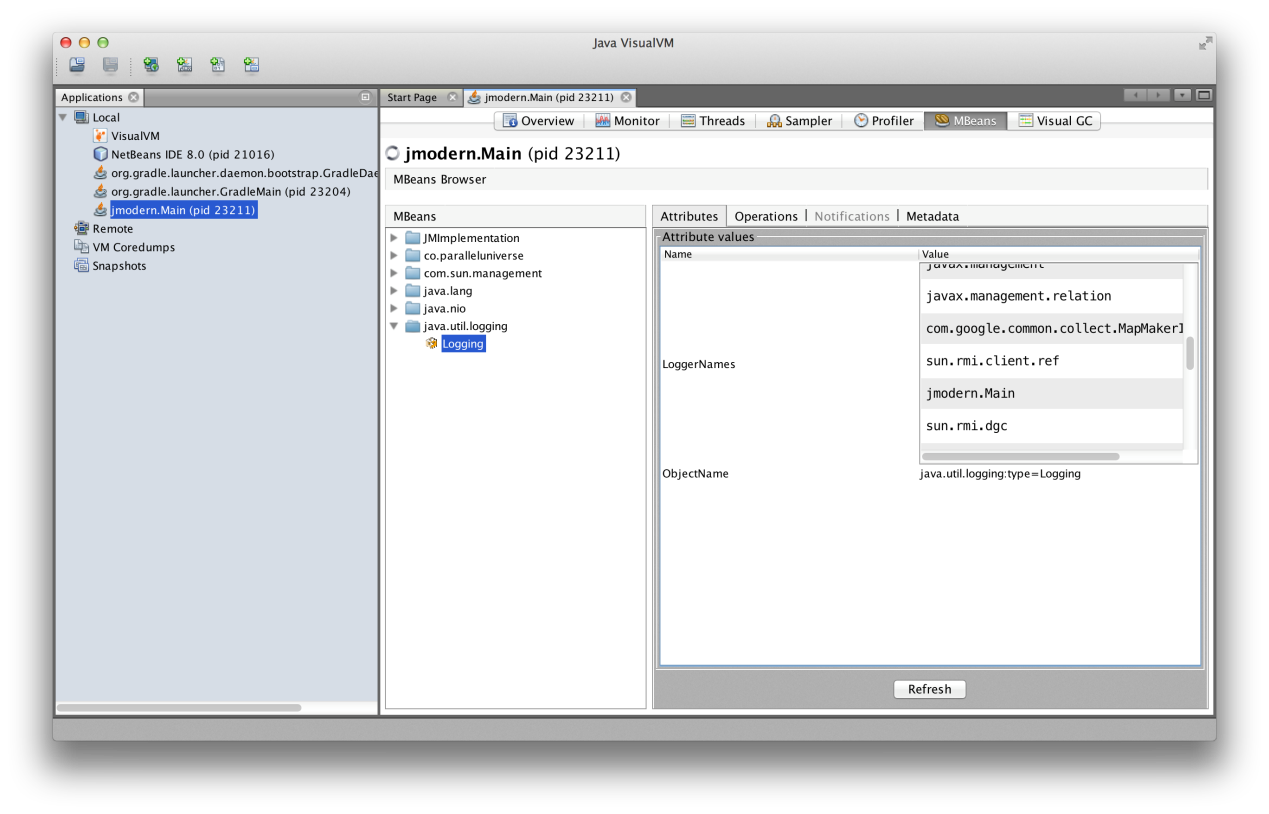
With our app running in one terminal, run jvisualvm (included as part of the JDK) in another. This will launch [VisualVM](https://visualvm.java.net/). Before we start playing with it, we need to install some plugins. Go to Tools->Plugins and pick Available Plugins. For the purpose of our demonstration we only need VisualVM-MBeans, but you might as well install all of them except maybe VisualVM-Glassfish and BTrace Workbench). Now pick jmodern.Main in the left pane, and choose the Monitor tab. You’ll see something like this:



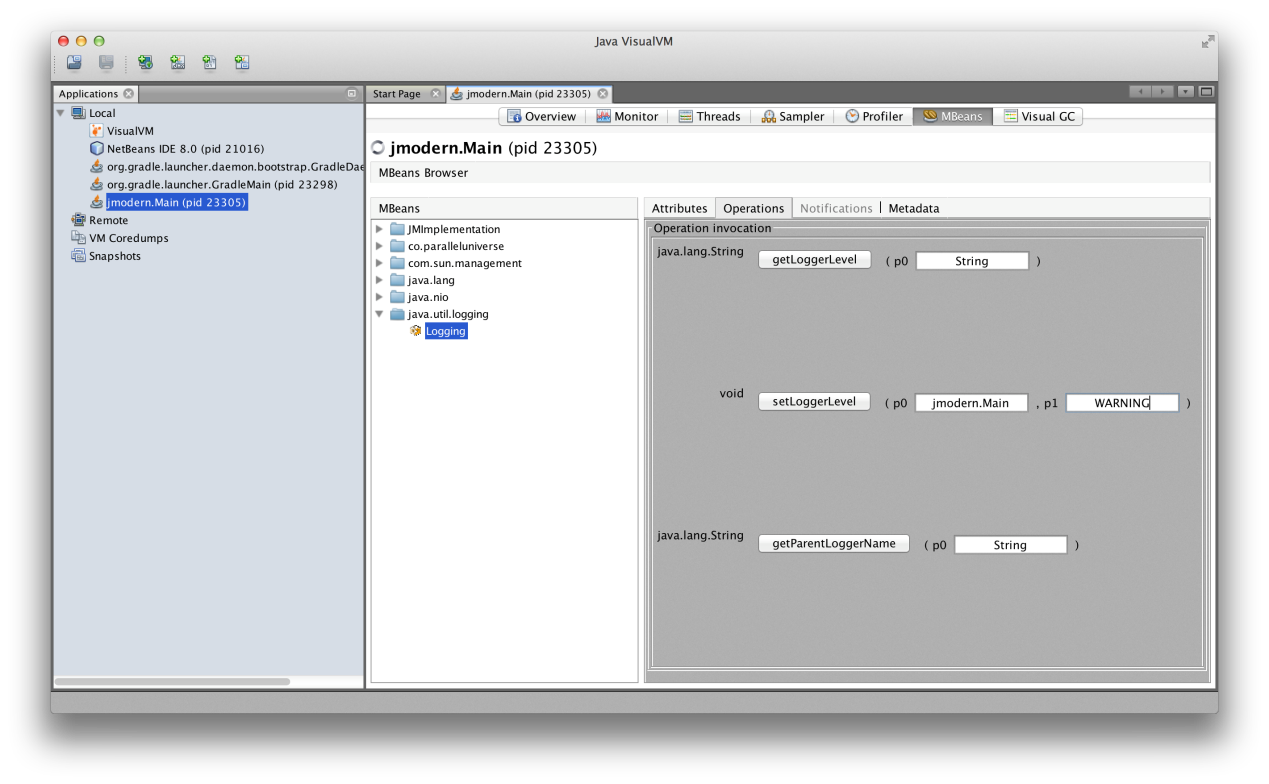
The monitor tab uses information exposed as JMX MBeans about the running JVM and displays them graphically, but we can also manually examine those MBeans (and many more) by choosing the MBeans tab (which will be available only after installing the VisualVM-MBeans plugin), we can examine and interact with all MBeans registered on our JVM instance. The one used in the heap plot, for example, is found under java.lang/Memory(double-click the attribute value in order to expand it):



Now let’s pick the java.util.logging/Logging MBean. The LoggerNames attribute in the right pane, will list all registered logger, including the one we’ve added to our code, jmodern.Main (double-click the attribute value in order to expand it).



MBeans let us not only inspect monitoring values, but also to set them, and invoke various management operations. Pick the Operations tab (in the right pane, next to the Attributes tab). We can now change the logging level at runtime via the JMX MBean. In the setLoggerLevel, fill two values:jmodern.Main in the first, and WARNING in the second (the new logging level), as in the screenshot below:



Now, when you click the setLoggerLevel button, the “info” log messages will no longer be displayed. If you set the level to SEVERE, both log messages will stop appearing. VisualVM automatically generates this simple GUI without any effort on the developer creating the MBean.

We can allow VisualVM (and other JMX consoles) to access our app remotely, by adding some system properties. To do that, we add the following lines to our build file’s run section:

systemProperty "com.sun.management.jmxremote"**,** ""

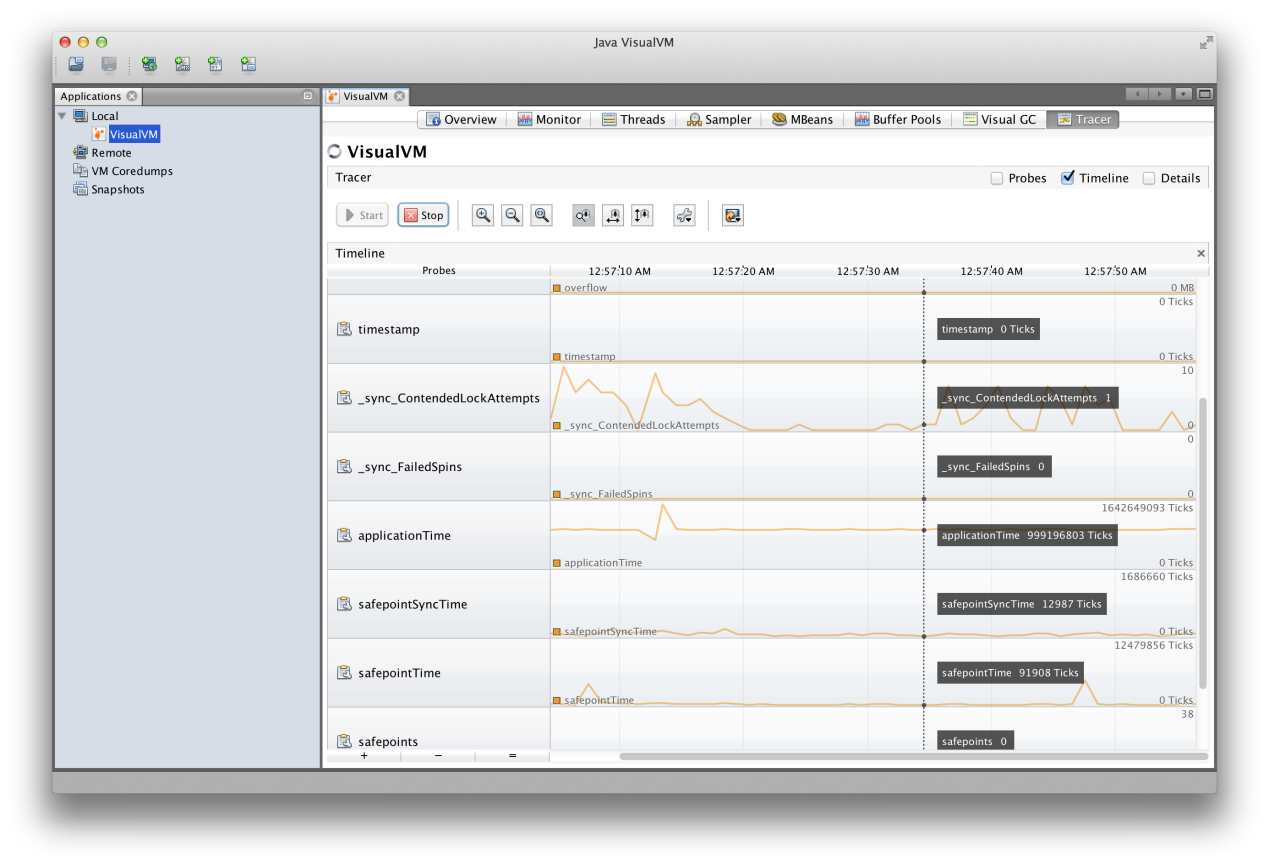
systemProperty "com.sun.management.jmxremote.port"**,** "9999"

systemProperty "com.sun.management.jmxremote.authenticate"**,** "false"

systemProperty "com.sun.management.jmxremote.ssl"**,** "false"

(in production, you’d naturally want to enable security).

As we’ve seen, in addition to MBean inspection, VisualVM also has custom monitoring views, some relying on JMX for data and some on other means: it monitors thread state and current stack trace for all threads, it provides insights into the GC and general memory usage, performs and analyzes heap dumps and core dumps, and much, much more. VisualVM is one of the most important tools in the modern Java developer’s toolbox. Here’s a screenshot of some advanced monitoring information provided by VisualVM’s traces plugins:



A modern Java developer might sometimes prefer a CLI to a nice GUI. A nice project called [jmxterm](http://wiki.cyclopsgroup.org/jmxterm), provides a CLI for JMX MBeans. Unfortunately, it does not yet support Java 7 and 8, but the developer says it will soon (if not, we will release a fix; we already have a working fork).

One thing is certain, though. The modern Java developer likes REST APIs (if for no other reason that they’re ubiquitous and are easy to build web GUIs for). While the JMX standard supports a few different local and remote connectors, it does not yet include an HTTP connector (it’s supposed to in Java 9). However, a beautiful project called [Jolokia](http://www.jolokia.org/) fills that void, and gives us RESTful access to our MBeans. Let’s give it a try. Merge the following into the your build.gradle:

configurations **{**

jolokia

**}**

dependencies **{**

runtime "org.jolokia:jolokia-core:1.2.1"

jolokia "org.jolokia:jolokia-jvm:1.2.1:agent"

**}**

run **{**

jvmArgs "-javaagent:${configurations.jolokia.iterator().next()}=port=7777,host=localhost"

**}**

(the fact that Gradle requires a new configuration for each dependency used as a Java agent annoys me to no end. Dear Gradle team: please, please make it easier!).

We also want to have Jolokia in our capsule, so we’ll change the Java-Agents attribute in capsule task to read:

'Java-Agents' **:** getDependencies**(**configurations**.**quasar**).**iterator**().**next**()** **+**

**+** " ${getDependencies(configurations.jolokia).iterator().next()}=port=7777,host=localhost"**,**

Run the application with gradle run, or with the capsule (gradle capsule; java -jar build/libs/jmodern-capsule.jar), and point your browser athttp://localhost:7777/jolokia/version. If Jolokia is working properly, you’ll get a JSON response. Now, to examine our app’s heap usage, do:

curl http://localhost:7777/jolokia/read/java.lang:type\=Memory/HeapMemoryUsage

To change our logger’s log level, you can do:

curl http://localhost:7777/jolokia/exec/java.util.logging:type\=Logging/setLoggerLevel\(java.lang.String,java.lang.String\)/jmodern.Main/WARNING

Jolokia has a very nice HTTP API, that can use both GET and POST operations, and it also allows for secure access. For more information, consult the (excelleny) [Jolokia documentation](http://www.jolokia.org/reference/html/index.html).

An HTTP API opens the door to web management consoles. The Jolokia site has a [demo](http://www.jolokia.org/client/javascript-cubism.html) of a [Cubism](http://square.github.io/cubism/) GUI for JMX MBeans. Another project by JBoss, called [hawtio](http://hawt.io/) uses the Jolokia HTTP API to construct a full featured, browser-based monitoring and manangement console for JVM applications. Aside from being a browser app, hawatio differs from VisualVM in that it is intended as a continuous manitoring/management tool for production code, while VisualVM is more of a troubleshooting tool.

Writing Your Own MBeans

Writing and registering your own MBeans is easy:

**package** jmodern**;**

**import** co.paralleluniverse.fibers.Fiber**;**

**import** co.paralleluniverse.strands.Strand**;**

**import** co.paralleluniverse.strands.channels.\***;**

**import** java.lang.management.ManagementFactory**;**

**import** java.util.concurrent.atomic.AtomicInteger**;**

**import** javax.management.MXBean**;**

**import** javax.management.ObjectName**;**

**import** org.slf4j.Logger**;**

**import** org.slf4j.LoggerFactory**;**

**public** **class** **Main** **{**

**static** **final** Logger log **=** LoggerFactory**.**getLogger**(**Main**.**class**);**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**final** AtomicInteger counter **=** **new** AtomicInteger**();**

**final** Channel**<**Object**>** ch **=** Channels**.**newChannel**(**0**);**

*// create and register MBean*

ManagementFactory**.**getPlatformMBeanServer**().**registerMBean**(new** JModernInfo**()** **{**

@Override

**public** **void** send**(**String message**)** **{**

**try** **{**

ch**.**send**(**message**);**

**}** **catch** **(**Exception e**)** **{**

**throw** **new** RuntimeException**(**e**);**

**}**

**}**

@Override

**public** **int** getNumMessagesReceived**()** **{**

**return** counter**.**get**();**

**}**

**},** **new** ObjectName**(**"jmodern:type=Info"**));**

**new** Fiber**<**Void**>(()** **->** **{**

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

Strand**.**sleep**(**100**);**

log**.**info**(**"Sending {}"**,** i**);** *// log something*

ch**.**send**(**i**);**

**if** **(**i **%** 10 **==** 0**)**

log**.**warn**(**"Sent {} messages"**,** i **+** 1**);** *// log something*

**}**

ch**.**close**();**

**}).**start**();**

**new** Fiber**<**Void**>(()** **->** **{**

Object x**;**

**while** **((**x **=** ch**.**receive**())** **!=** **null)** **{**

counter**.**incrementAndGet**();**

System**.**out**.**println**(**"--> " **+** x**);**

**}**

**}).**start**().**join**();** *// join waits for this fiber to finish*

**}**

@MXBean

**public** **interface** **JModernInfo** **{**

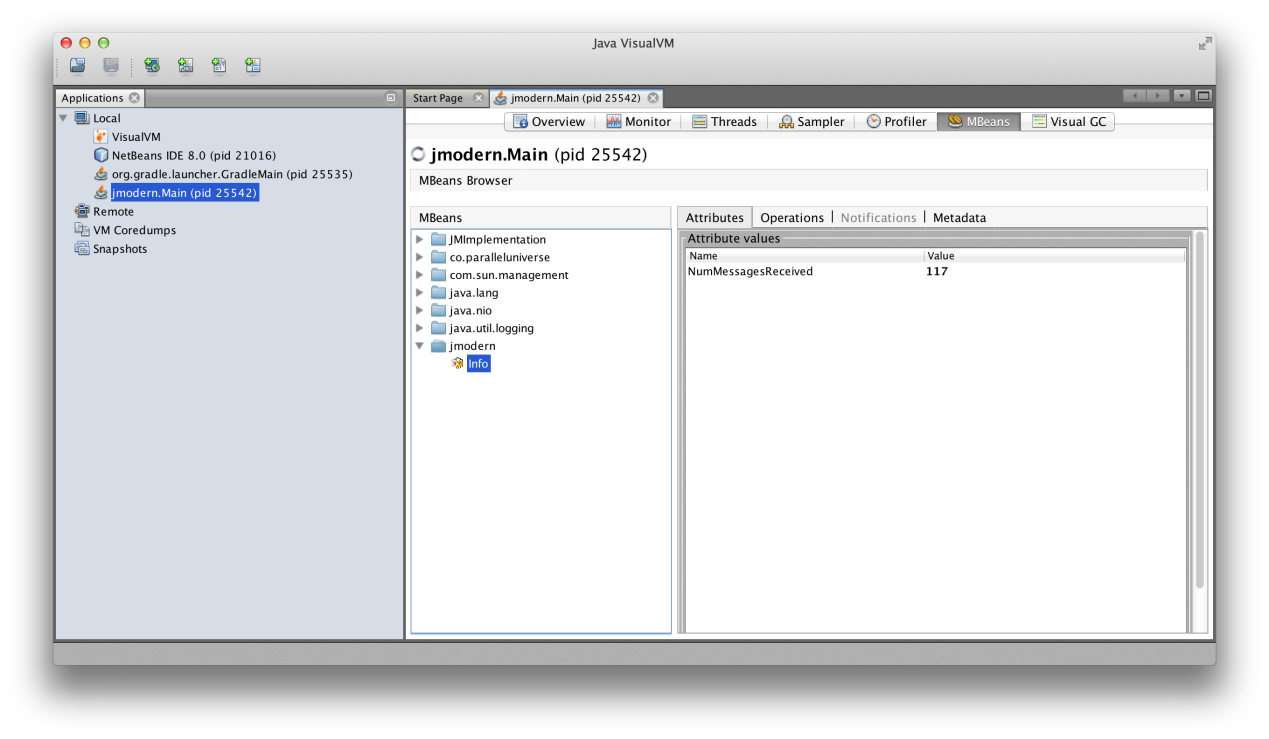
**void** send**(**String message**);**

**int** getNumMessagesReceived**();**

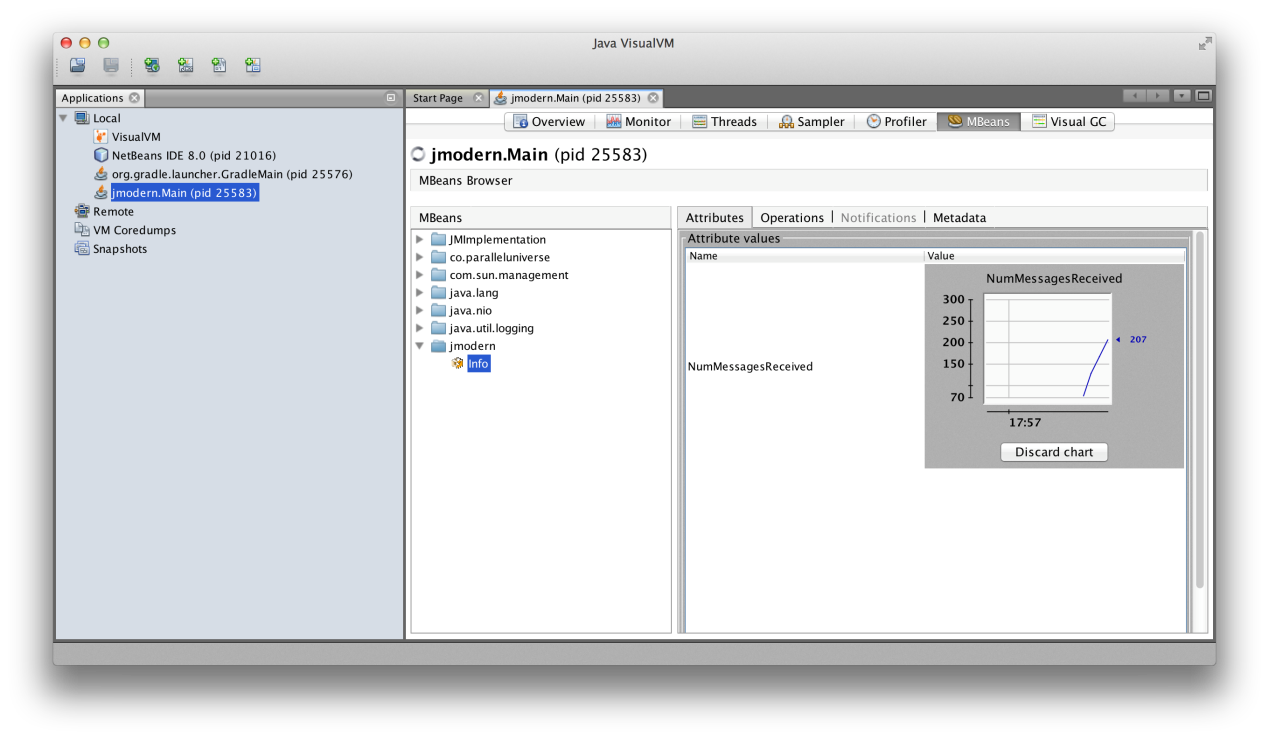
**}**

**}**

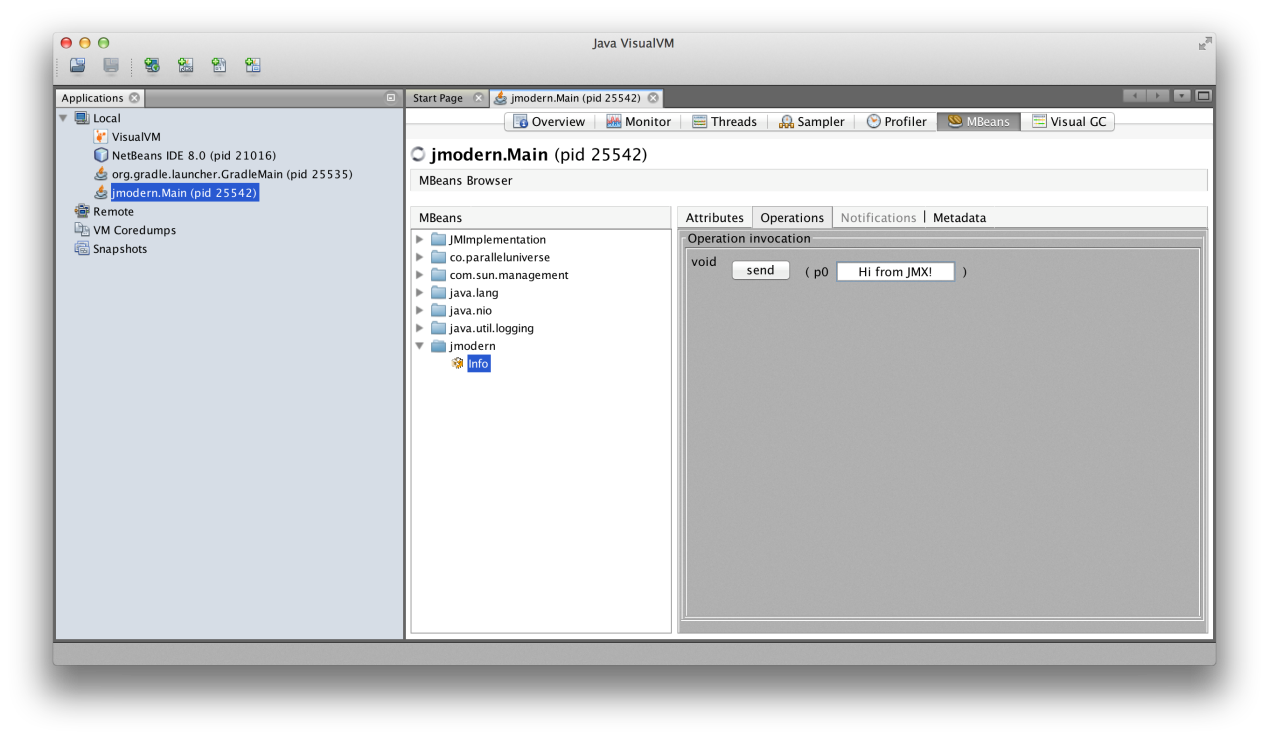
We’ve added an MBean that lets us monitor the number of messages received by the second fiber, and also exposes a send operation, that will slip a message into the channel. When we run the app, we can now see our monitored property in VisualVM:



plot it (by double clicking the value):



and use our MBean operation in the Operations tab to sneak a message into the channel:



Easy Health and Performance Monitoring with Metrics

[Metrics](http://metrics.codahale.com/) is a cool, modern, library for easy performance and health monitoring, built by Coda Hale back when he was at Yammer. It contains common metrics collection and reporting classes like histograms, timers, counters gauges, etc. Let’s take it out for a spin.

We’re not going to be using Jolokia now, so you can take all of the Jolokia stuff out from the build file. Instead, you’ll need to add the following dependency:

compile "com.codahale.metrics:metrics-core:3.0.2"

Metrics can report its metrics via JMX MBeans, write them to CSV files, expose them via a RESTful interface, or publish them to Graphite or Ganglia. We will just be reporting to JMX (although we’ll see Metrics’s HTTP reporting in part 3, when we discuss Dropwizard). This will be our new Main class:

**package** jmodern**;**

**import** co.paralleluniverse.fibers.Fiber**;**

**import** co.paralleluniverse.strands.Strand**;**

**import** co.paralleluniverse.strands.channels.\***;**

**import** com.codahale.metrics.\***;**

**import** static com**.**codahale**.**metrics**.**MetricRegistry**.**name**;**

**import** java.util.concurrent.ThreadLocalRandom**;**

**import** static java**.**util**.**concurrent**.**TimeUnit**.\*;**

**public** **class** **Main** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**final** MetricRegistry metrics **=** **new** MetricRegistry**();**

JmxReporter**.**forRegistry**(**metrics**).**build**().**start**();** *// starts reporting via JMX*

**final** Channel**<**Object**>** ch **=** Channels**.**newChannel**(**0**);**

**new** Fiber**<**Void**>(()** **->** **{**

Meter meter **=** metrics**.**meter**(**name**(**Main**.**class**,** "messages" **,** "send"**,** "rate"**));**

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

Strand**.**sleep**(**ThreadLocalRandom**.**current**().**nextInt**(**50**,** 500**));** *// random sleep*

meter**.**mark**();** *// measures event rate*

ch**.**send**(**i**);**

**}**

ch**.**close**();**

**}).**start**();**

**new** Fiber**<**Void**>(()** **->** **{**

Counter counter **=** metrics**.**counter**(**name**(**Main**.**class**,** "messages"**,** "received"**));**

Timer timer **=** metrics**.**timer**(**name**(**Main**.**class**,** "messages"**,** "duration"**));**

Object x**;**

**long** lastReceived **=** System**.**nanoTime**();**

**while** **((**x **=** ch**.**receive**())** **!=** **null)** **{**

**final** **long** now **=** System**.**nanoTime**();**

timer**.**update**(**now **-** lastReceived**,** NANOSECONDS**);** *// creates duration histogram*

lastReceived **=** now**;**

counter**.**inc**();** *// counts*

System**.**out**.**println**(**"--> " **+** x**);**

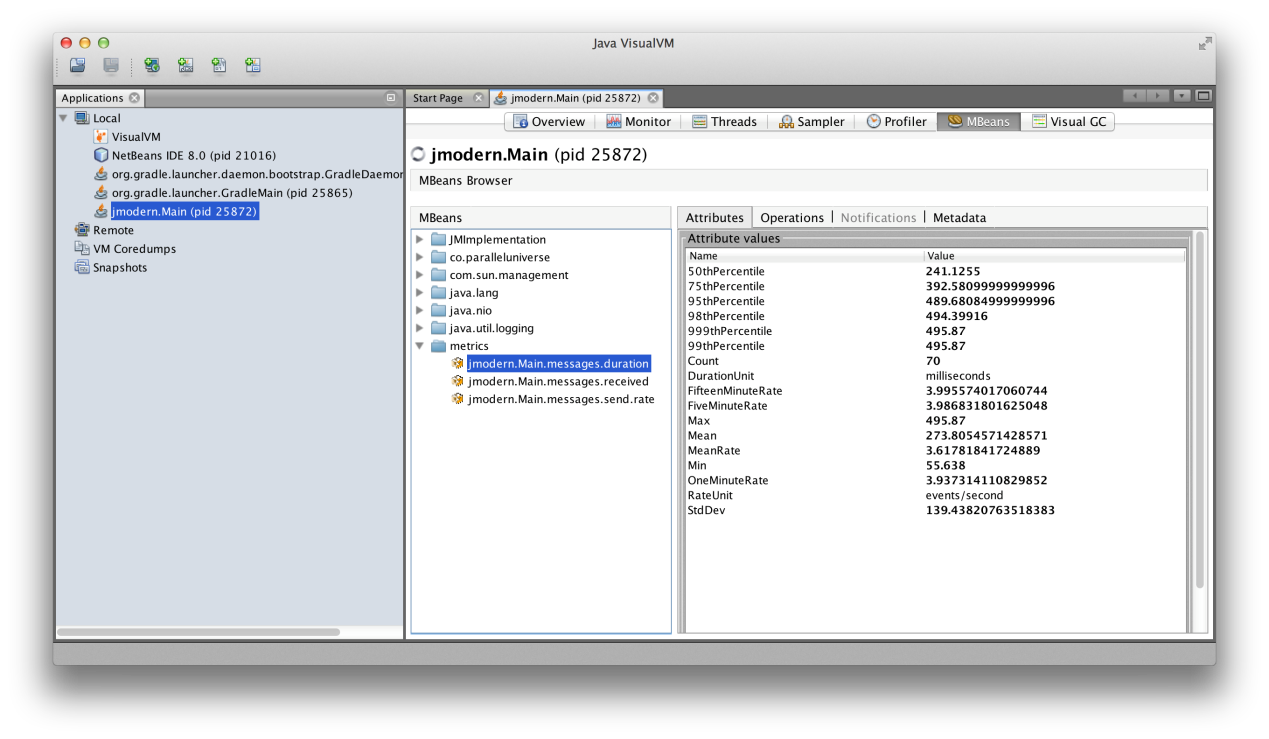
**}**

**}).**start**().**join**();** *// join waits for this fiber to finish*

**}**

**}**

In this example we’ve used a Metrics counter (which counts things), a meter (measures rate of events) and a timer (produces a histogram of time durations). Now let’s run the app and fire up VisualVM. Here are our metrics (the screenshot shows the timer metric):



Profiling

Profiling an application is the key to meeting our performance requirements. Only by profiling can we know which parts of the code affect overall execution speed, and concentrate our efforts to optimize them, and only them. Java has always had good profilers, both as part of its IDEs or as separate products. But as a result of merging parts of the JRockit JVM code into HotSpot, Java recently gained its most precise, and lightweight profiler yet. While the tool (or, rather, a combination of two tools), we will explore in this section are not open-source, we discuss them because they are included in the standard Oracle JDK, and are free to use in you development environment (but require a commercial license for use on code deployed to production).

We will begin with a test program. It’s our original demo with some artificial work added:

**package** jmodern**;**

**import** co.paralleluniverse.fibers.Fiber**;**

**import** co.paralleluniverse.strands.Strand**;**

**import** co.paralleluniverse.strands.channels.\***;**

**import** com.codahale.metrics.\***;**

**import** static com**.**codahale**.**metrics**.**MetricRegistry**.**name**;**

**import** java.util.concurrent.ThreadLocalRandom**;**

**import** static java**.**util**.**concurrent**.**TimeUnit**.\*;**

**public** **class** **Main** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**final** MetricRegistry metrics **=** **new** MetricRegistry**();**

JmxReporter**.**forRegistry**(**metrics**).**build**().**start**();** *// starts reporting via JMX*

**final** Channel**<**Object**>** ch **=** Channels**.**newChannel**(**0**);**

**new** Fiber**<**Void**>(()** **->** **{**

Meter meter **=** metrics**.**meter**(**name**(**Main**.**class**,** "messages"**,** "send"**,** "rate"**));**

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

Strand**.**sleep**(**ThreadLocalRandom**.**current**().**nextInt**(**50**,** 500**));** *// random sleep*

meter**.**mark**();**

ch**.**send**(**i**);**

**}**

ch**.**close**();**

**}).**start**();**

**new** Fiber**<**Void**>(()** **->** **{**

Counter counter **=** metrics**.**counter**(**name**(**Main**.**class**,** "messages"**,** "received"**));**

Timer timer **=** metrics**.**timer**(**name**(**Main**.**class**,** "messages"**,** "duration"**));**

Object x**;**

**long** lastReceived **=** System**.**nanoTime**();**

**while** **((**x **=** ch**.**receive**())** **!=** **null)** **{**

**final** **long** now **=** System**.**nanoTime**();**

timer**.**update**(**now **-** lastReceived**,** NANOSECONDS**);**

lastReceived **=** now**;**

counter**.**inc**();**

**double** y **=** foo**(**x**);**

System**.**out**.**println**(**"--> " **+** x **+** " " **+** y**);**

**}**

**}).**start**().**join**();**

**}**

**static** **double** foo**(**Object x**)** **{** *// do crazy work*

**if** **(!(**x **instanceof** Integer**))**

**return** 0.0**;**

**double** y **=** **(**Integer**)**x **%** 2723**;**

**for(int** i**=**0**;** i**<**10000**;** i**++)** **{**

String rstr **=** randomString**(**'A'**,** 'Z'**,** 1000**);**

y **\*=** rstr**.**matches**(**"ABA"**)** **?** 0.5 **:** 2.0**;**

y **=** Math**.**sqrt**(**y**);**

**}**

**return** y**;**

**}**

**public** **static** String randomString**(char** from**,** **char** to**,** **int** length**)** **{**

**return** ThreadLocalRandom**.**current**().**ints**(**from**,** to **+** 1**).**limit**(**length**)**

**.**mapToObj**(**x **->** Character**.**toString**((char)**x**)).**collect**(**Collectors**.**joining**());**

**}**

**}**

The function foo does some nonsensical computations, so don’t try to make sense of them. When you run the application (gradle run), you may notice warning from Quasar, that one of the fibers is consuming an inordinate amount of CPU time. To figure out what’s going on, we need to profile.

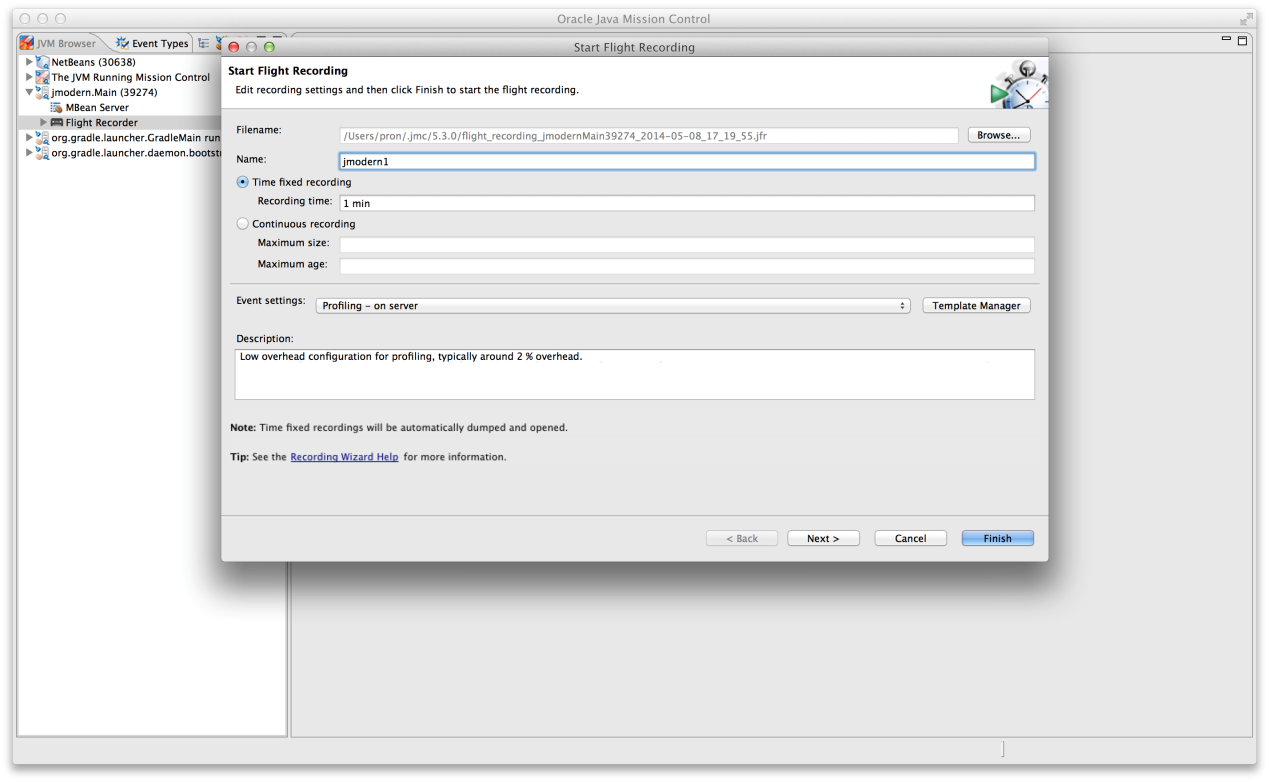
The profiler we’ll use is very precise and has a very low overhead. It’s made of two components, The first, [Java Flight Recorder](http://docs.oracle.com/javase/8/docs/technotes/guides/jfr/toc.html) is baked into the HotSpot VM. It can record many different JVM events into very efficient buffers. It can be triggered to [start and stop recording with jcmd](http://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/run.htm#CHDIDCHG), but we will control it with the second tool. The second tool is [Java Mission Control](http://docs.oracle.com/javase/8/docs/technotes/guides/jmc/) (or JMC), which is included in the JDK installation. JMC was the analog to VisualVM in the JRockit VM, and it performs many of the same functions (like MBean inspection), only it is far uglier. But we will be using JMC for it’s ability to control the Java Flight Recorder, and analyze its recordings (I hope Oracle eventually move this functionality into the much prettier VisualVM).

Flight Recorder has to be enabled when the program is launched (it won’t record anything yet and won’t affect performance), so we’ll stop the execution, and add this line to build.gradle’s run section:

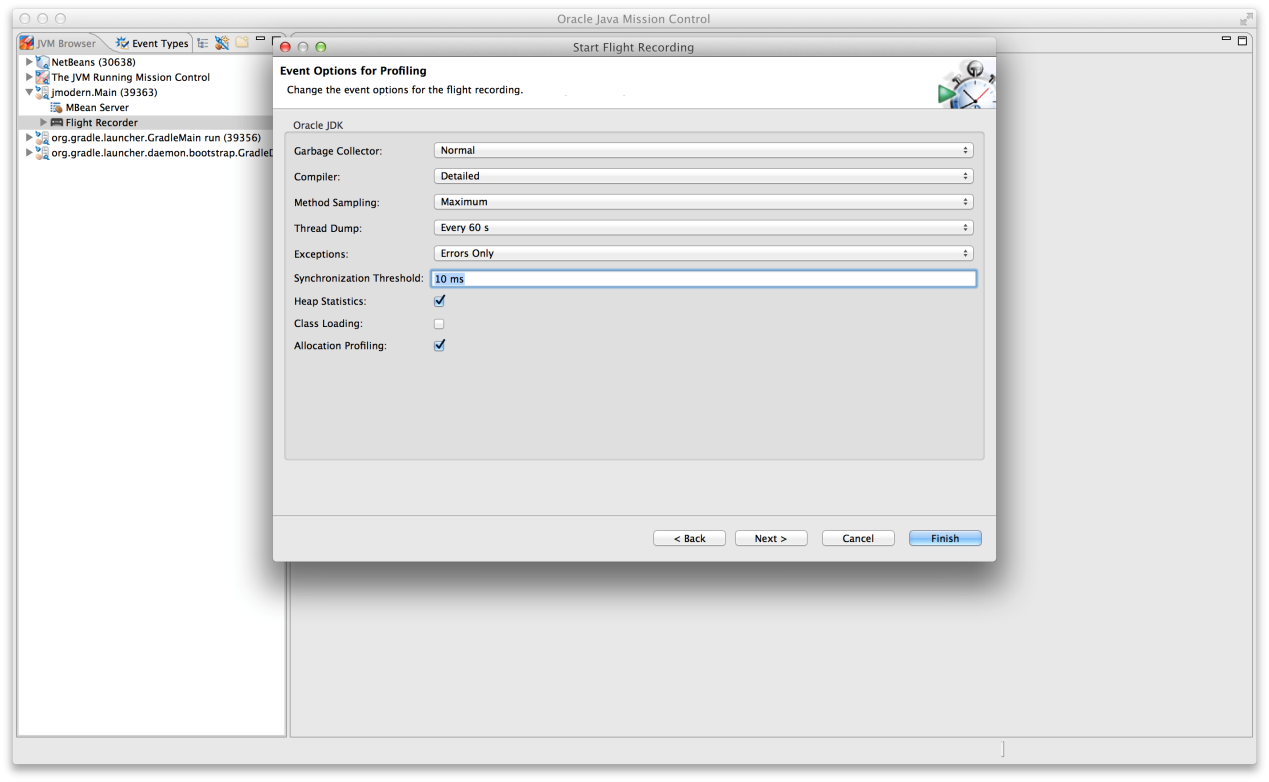
jvmArgs "-XX:+UnlockCommercialFeatures"**,** "-XX:+FlightRecorder"

The UnlockCommercialFeatures flag is necessary because Flight Recorder is a commercial feature, but it’s free for use in development. Now let’s launch the program again.

In another terminal, let’s fire up Mission Control with the jmc command. In the left pane, right-click jmodern.Main, and choose “Start Flight Recording…”. In the wizard window’s “Event settings” dropbox pick “Profiling - on server”, and click “Next >” (not “Finish”!).

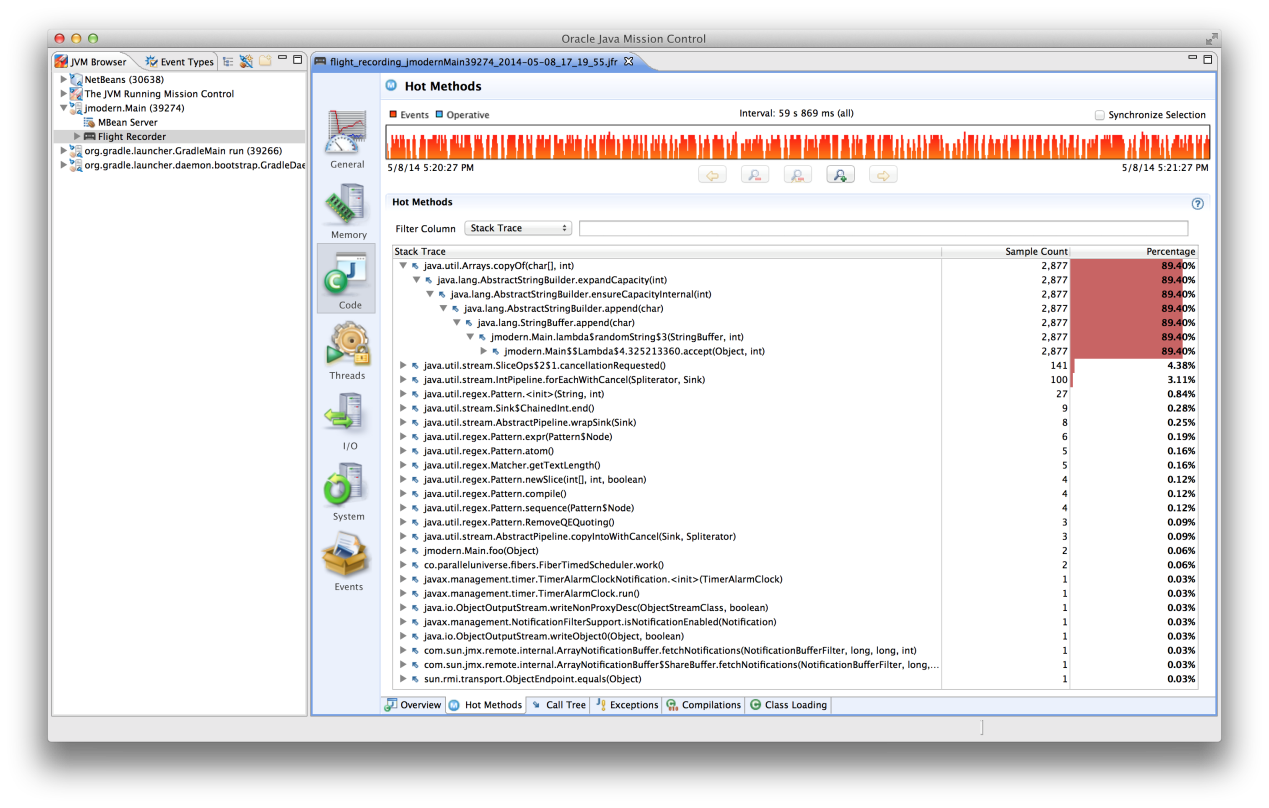


In the next screen, check the “Heap Statistics” and “Allocation Profiling” boxes, and click “Finish”.

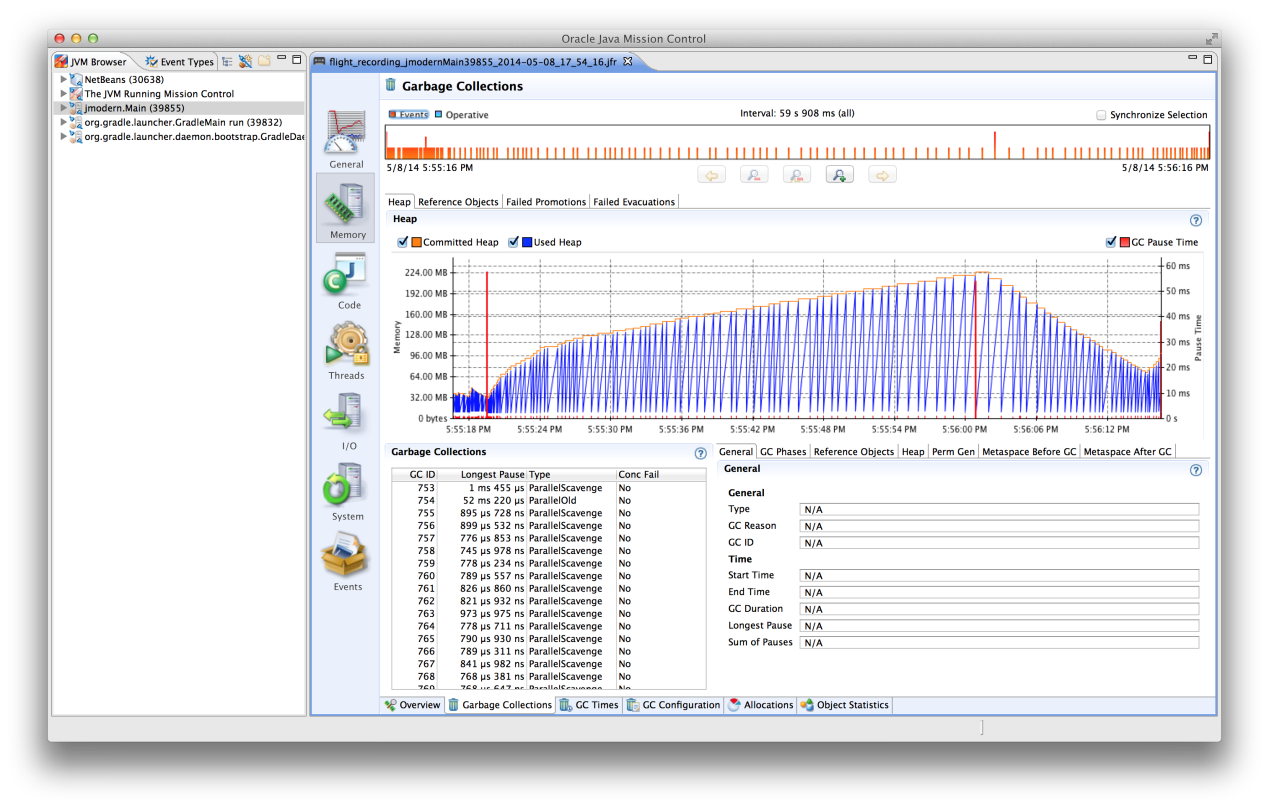


JMC will now wait for one minute until Flight Recorder concludes the recording. It will then open the recording file for analysis, at which point you can safely terminate our application.

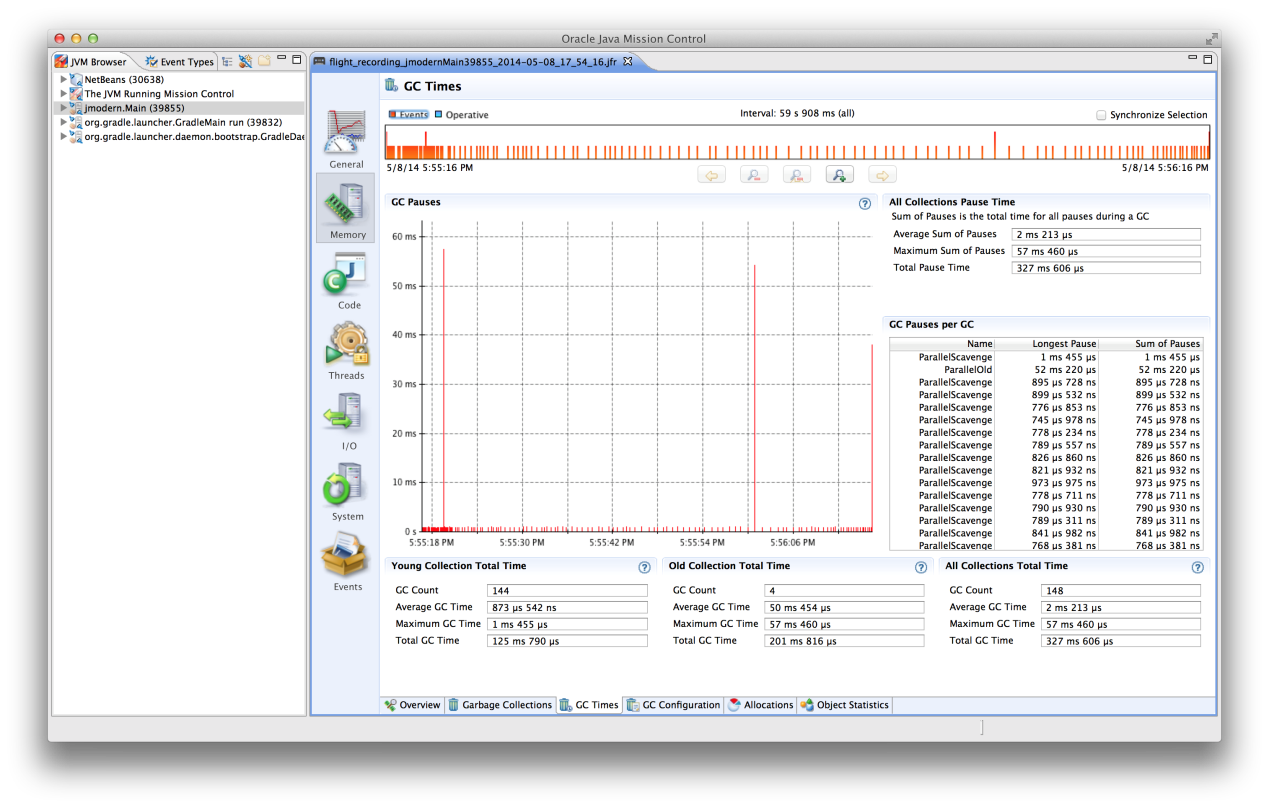
The “Code” section’s Hot Methods tab immediately uncovers the randomString method as the culprit, responsible for almost 90% of the program’s execution time.



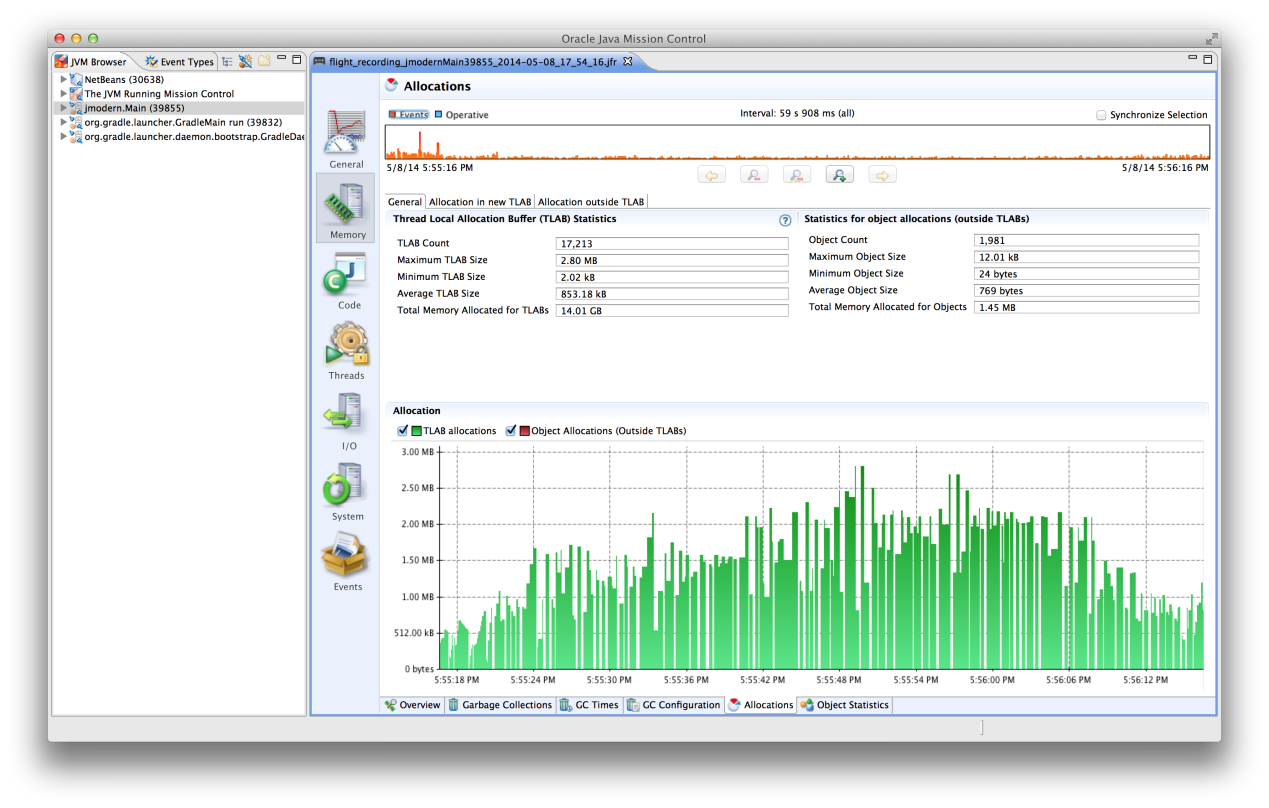
The Memory section’s Garbage Collection tab shows the heap usage, over the duration of the recording:



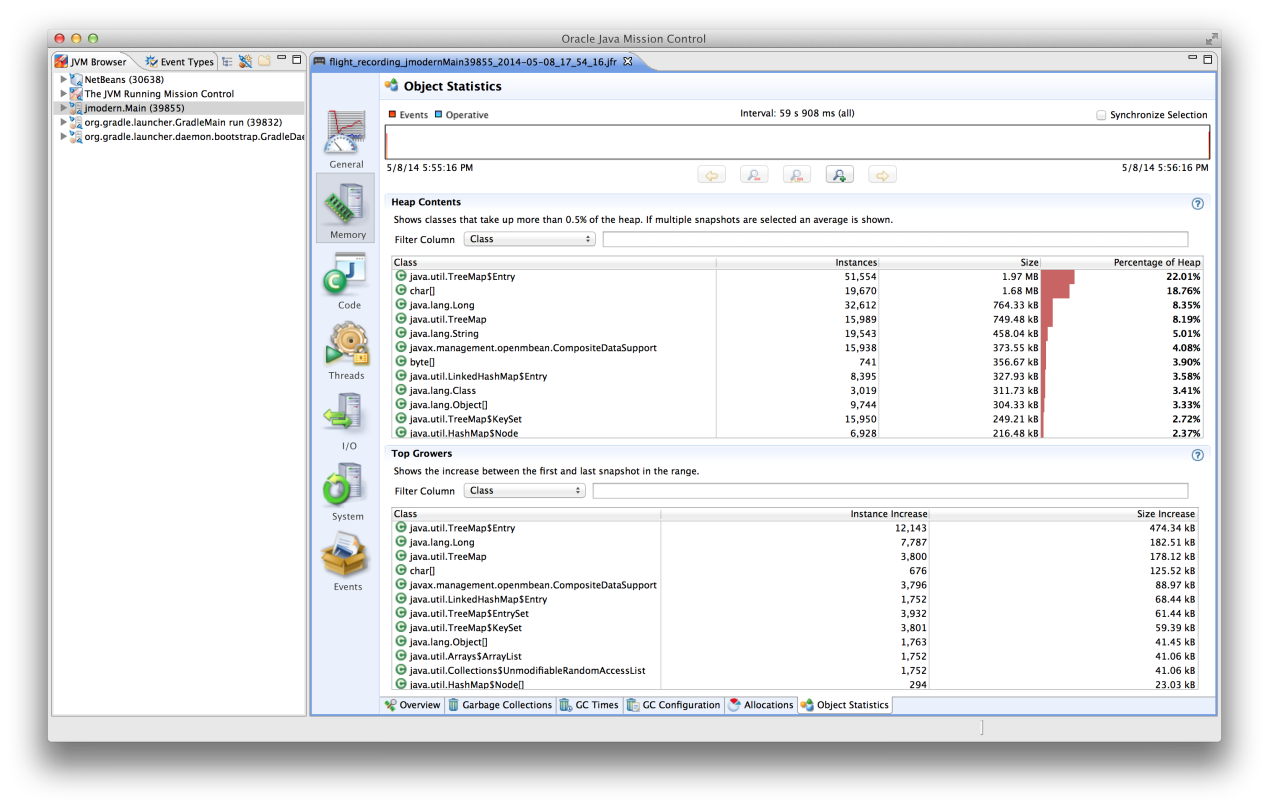
And the GC Times tab shows the durtation of the garbage collector’s collections:



We can also examine allocation behavior:



and the heap’s contents:



Java Flight Recorder also has an [(unsupported) API](http://hirt.se/blog/?p=277) that lets us record our own application events in JFR’s recordings.

Advanced Topic: Profiling and Debugging with Byteman

Like last time, we will conclude this installment with advanced topics. First up is profiling and debugging with Byteman. As we mentioned in part 1, one of the JVM’s most powerful features is the ability to dynamically load code at runtime (which goes far beyond loading dynamic libraries in native applications). Not only that: the JVM gives us the ability to transform, and re-transform, already running code.

A useful tool which takes advantage of this ability is [Byteman](https://www.jboss.org/byteman.html) by JBoss. Byteman allows us to inject tracing, debugging and profiling code into a running application. It is included here as an advanced topic because its support for Java 7, let alone 8, is a bit shaky, and must require fixes (to Byteman, that is). The project is actively developed, but lagging behind. We will therefore use byteman on very basic code.

This will be our main class:

**package** jmodern**;**

**import** java.util.concurrent.ThreadLocalRandom**;**

**public** **class** **Main** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**for** **(int** i **=** 0**;;** i**++)** **{**

System**.**out**.**println**(**"Calling foo"**);**

foo**(**i**);**

**}**

**}**

**private** **static** String foo**(int** x**)** **throws** InterruptedException **{**

**long** pause **=** ThreadLocalRandom**.**current**().**nextInt**(**50**,** 500**);**

Thread**.**sleep**(**pause**);**

**return** "aaa" **+** pause**;**

**}**

**}**

foo simulates calling a service which may take some unknown time to complete.

Next, we’ll merge the following into our build file:

configurations **{**

byteman

**}**

dependencies **{**

byteman "org.jboss.byteman:byteman:2.1.4.1"

**}**

run **{**

jvmArgs "-javaagent:${configurations.byteman.iterator().next()}=listener:true,port:9977"

*// remove the quasar agent*

**}**

If you want to test Byteman with a capsule, you can change the Java-Agents attribute in the build file to read:

'Java-Agents' **:** "${getDependencies(configurations.byteman).iterator().next()}=listener:true,port:9977"**,**

Now, we’ll download Byteman [here](http://byteman.jboss.org/downloads) (we can’t rely only on the dependencies because we’re going to use some of Byteman’s command line tools), unzip the archive, and set the environment variable BYTEMAN\_HOME to point to Byteman’s directory.

Now, launch the app in one terminal:

gradle run

It will print something like this:

Calling foo

Calling foo

Calling foo

Calling foo

Calling foo

We want to know how long each call to foo takes, but we’ve forgotten to measure and log that. We’ll use Byteman to insert that log while the program is running.

Start an editor, and create the file jmodern.btm in the project’s directory:

RULE trace foo entry

CLASS jmodern.Main

METHOD foo

AT ENTRY

IF true

DO createTimer("timer")

ENDRULE

RULE trace foo exit

CLASS jmodern.Main

METHOD foo

AT EXIT

IF true

DO traceln("::::::: foo(" + $1 + ") -> " + $! + " : " + resetTimer("timer") + "ms")

ENDRULE

These are Byteman rules, which we will apply to the program. While the program is still running in one terminal, open another, and type:

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977 jmodern.btm

Our app will now start printing something like this in the first terminal:

Calling foo

::::::: foo(152) -> aaa217 : 217ms

Calling foo

::::::: foo(153) -> aaa281 : 281ms

Calling foo

::::::: foo(154) -> aaa282 : 283ms

Calling foo

::::::: foo(155) -> aaa166 : 166ms

Calling foo

::::::: foo(156) -> aaa160 : 161ms

To see which rules are applied:

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977

Finally, to unload our Byteman script:

$BYTEMAN\_HOME/bin/bmsubmit.sh -p 9977 -u

And now our injected log messages are gone!

Byteman is an extremely powerful tool, made possible by the JVMs flexible code transformations. You can use it to examine variables, log events (either to the standard output or to a file), insert delays and more. You can even easily add your own Byteman actions (say, if you want to log events using your logging engine). For more information, please refer to the [Byteman documentation](http://byteman.jboss.org/documentation).

Advanced Topic: Benchmarking with JMH

Advances in both hardware architecture as well as compiler technology have made benchmarking the only viable way to reason about code performance. Modern CPUs (and modern compilers) are so clever (see this [excelent talk](http://www.infoq.com/presentations/click-crash-course-modern-hardware) by Cliff Click on modern hardware) that creating a mental performance profile of our code – like game programmers did until the end of the 90s – is damn-near impossible; even for a program written in C; heck, even for a program written in Assembly. On the other hand, the cleverness of compilers and CPUs are exactly what makes micro-benchmarking (benchmarking small snippets of code) so hard, as execution speed is very dependent on context (for example. it is affected by the state of the CPU cahce, which, in turn, is affected by what other threads are doing). Microbenchmarking JVM programs is doubly tricky because, as I mentioned in part 1, the JVM’s JIT is a profile-guided optimizing compiler[4](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fn:jvm), which is very much affected by the context in which code is run. Therefore, code in a microbenchmark can be much faster, or much slower, than the same code in the context of a larger program.

[JMH](http://openjdk.java.net/projects/code-tools/jmh/) is an open-source Java benchmarking harness by Oracle, that runs code snippets in just the right way so you can truly reason about their performance (see [this great talk](http://medianetwork.oracle.com/video/player/2630310904001) ([slides](http://shipilev.net/pub/talks/jvmls-July2013-benchmarking.pdf)) by Aleksey Shipilev, JMH’s main author, about Java performance benchmarking). There is another, older tool, called Caliper, which was made by Google to serve the same purpose as JMH, but it is far cruder, and might even give wrong results. Don’t use it.

We will take JMH for a spin right away, but first the usual warning regarding microbenchmarks: premature optimization is the root of all evil. There’s is no point in benchmarking two algorithms or data structures to find out that one is 100 times faster than the other, if the algorithm accounts for a grand total of 1% of your app’s execution time. Even making that algorithm run infinitely fast would only save your 2%. Benchmark only after you’ve profiled your application and determined which bits would generate the most gain if accelerated.

As always, we’ll begin with the build file. Add these to your dependencies:

testCompile 'org.openjdk.jmh:jmh-core:0.8'

testCompile 'org.openjdk.jmh:jmh-generator-annprocess:0.8'

and put this bench task at the bottom of the build file:

task **bench(**type: JavaExec**,** dependsOn: **[**classes**,** testClasses**])** **{**

classpath **=** sourceSets**.**test**.**runtimeClasspath *// we'll put jmodern.Benchamrk in the test directory*

main **=** "jmodern.Benchmark"**;**

**}**

Finally, we’ll put our benchmark code in src/test/java/jmodern/Benchmark.java. I mentioned 90s game programmers before, and in order to show that some of their techniques still work, we’ll benchmark a “standard” inverse square-root computation, with the [fast inverse square root algorithm](http://en.wikipedia.org/wiki/Fast_inverse_square_root), misattributed to John Carmack:

**package** jmodern**;**

**import** java.util.concurrent.TimeUnit**;**

**import** org.openjdk.jmh.annotations.\***;**

**import** org.openjdk.jmh.profile.\***;**

**import** org.openjdk.jmh.runner.Runner**;**

**import** org.openjdk.jmh.runner.options.OptionsBuilder**;**

**import** org.openjdk.jmh.runner.parameters.TimeValue**;**

@State**(**Scope**.**Thread**)**

@BenchmarkMode**(**Mode**.**AverageTime**)**

@OutputTimeUnit**(**TimeUnit**.**NANOSECONDS**)**

**public** **class** **Benchmark** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**new** Runner**(new** OptionsBuilder**()**

**.**include**(**Benchmark**.**class**.**getName**()** **+** ".\*"**)**

**.**forks**(**1**)**

**.**warmupTime**(**TimeValue**.**seconds**(**5**))**

**.**warmupIterations**(**3**)**

**.**measurementTime**(**TimeValue**.**seconds**(**5**))**

**.**measurementIterations**(**5**)**

**.**build**()).**run**();**

**}**

**private** **double** x **=** 2.0**;** *// prevent constant folding*

@GenerateMicroBenchmark

**public** **double** standardInvSqrt**()** **{**

**return** 1.0**/**Math**.**sqrt**(**x**);**

**}**

@GenerateMicroBenchmark

**public** **double** fastInvSqrt**()** **{**

**return** invSqrt**(**x**);**

**}**

**static** **double** invSqrt**(double** x**)** **{**

**double** xhalf **=** 0.5d **\*** x**;**

**long** i **=** Double**.**doubleToLongBits**(**x**);**

i **=** 0x5fe6ec85e7de30daL **-** **(**i **>>** 1**);**

x **=** Double**.**longBitsToDouble**(**i**);**

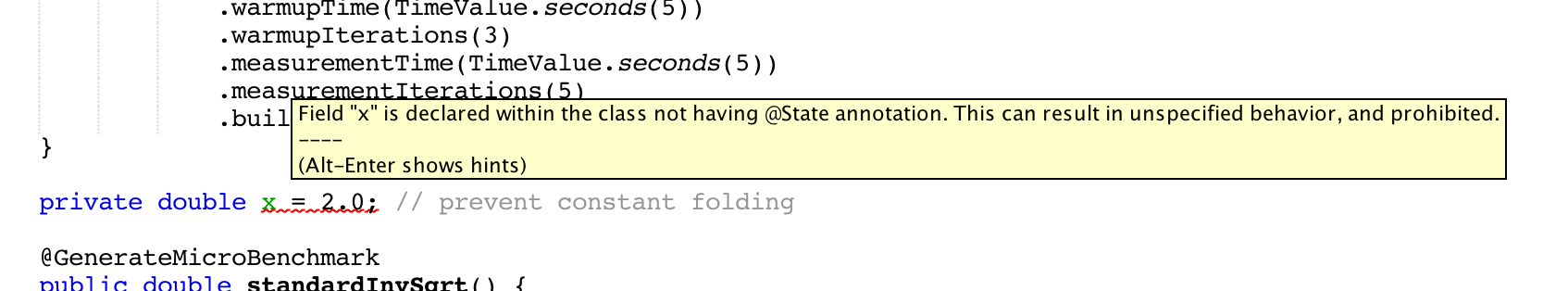
x **=** x **\*** **(**1.5d **-** xhalf **\*** x **\*** x**);**

**return** x**;**

**}**

**}**

By the way, like the Checker framework we discussed in part 1, JMH uses an annotation processor. Unlike Checker, JMH does it right, so you get automatic IDE integration with all IDEs. Here for example, is what happens in NetBeans if you forget to include the @State annotation:



To run the benchmarks, type gradle bench at the shell. I got these results:

Benchmark Mode Samples Mean Mean error Units

j.Benchmark.fastInvSqrt avgt 10 2.708 0.019 ns/op

j.Benchmark.standardInvSqrt avgt 10 12.824 0.065 ns/op

Nice, but keep in mind that fast-inv-sqrt is a rough approximation, good only to a few decimal places.

Here’s another example, this one also reports the time spent garbage-collecting, and gives a crude method stack profile:

**package** jmodern**;**

**import** java.util.\***;**

**import** java.util.concurrent.\***;**

**import** org.openjdk.jmh.annotations.\***;**

**import** org.openjdk.jmh.profile.\***;**

**import** org.openjdk.jmh.runner.Runner**;**

**import** org.openjdk.jmh.runner.options.OptionsBuilder**;**

**import** org.openjdk.jmh.runner.parameters.TimeValue**;**

@State**(**Scope**.**Thread**)**

@BenchmarkMode**(**Mode**.**AverageTime**)**

@OutputTimeUnit**(**TimeUnit**.**NANOSECONDS**)**

**public** **class** **Benchmark** **{**

**public** **static** **void** main**(**String**[]** args**)** **throws** Exception **{**

**new** Runner**(new** OptionsBuilder**()**

**.**include**(**Benchmark**.**class**.**getName**()** **+** ".\*"**)**

**.**forks**(**2**)**

**.**warmupTime**(**TimeValue**.**seconds**(**5**))**

**.**warmupIterations**(**3**)**

**.**measurementTime**(**TimeValue**.**seconds**(**5**))**

**.**measurementIterations**(**5**)**

**.**addProfiler**(**GCProfiler**.**class**)** *// report GC time*

**.**addProfiler**(**StackProfiler**.**class**)** *// report method stack execution profile*

**.**build**()).**run**();**

**}**

@GenerateMicroBenchmark

**public** Object arrayList**()** **{**

**return** add**(new** ArrayList**<>());**

**}**

@GenerateMicroBenchmark

**public** Object linkedList**()** **{**

**return** add**(new** LinkedList**<>());**

**}**

**static** Object add**(**List**<**Integer**>** list**)** **{**

**for** **(int** i **=** 0**;** i **<** 4000**;** i**++)**

list**.**add**(**i**);**

**return** list**;**

**}**

**}**

Here is an example of the profiling info printed by JMH:

Iteration 3: 33783.296 ns/op

GC | wall time = 5.000 secs, GC time = 0.048 secs, GC% = 0.96%, GC count = +97

|

Stack | 96.9% RUNNABLE jmodern.generated.Benchmark\_arrayList.arrayList\_AverageTime\_measurementLoop

| 1.8% RUNNABLE java.lang.Integer.valueOf

| 1.3% RUNNABLE java.util.Arrays.copyOf

| 0.0% (other)

|

JMH is a very rich framework. Unfortunately, it is a little thin on documentation, but it does have a rather nice tutorial (which also demonstrates the pitfalls of naive Java microbenchmarking) written as [a series of code samples](http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/). You can also read this [good introductory post](http://psy-lob-saw.blogspot.co.il/2013/04/writing-java-micro-benchmarks-with-jmh.html) by Nitsan Wakart.

So, What Have We Learned So Far?

In this post we’ve covered some of the best tools for JVM management, monitoring and profiling. The JVM is very serious about providing deep insight into its operation, which is, in addition to its great performance, the main reason I wouldn’t replace the JVM as the platform for a heavy duty, long-running server-side app, with any other technology. We’ve also seen how powerful the JVM is when it comes to modifying running code with tools like Byteman.

We’ve also introduced Capsule, a lightweight, single-file, stateless, installation-free deployment package for JVM apps. It optionally supports automatic upgrades – for the entire app, or just a library dependency – via a public, or organizational Maven repository.

In [part 3](http://blog.paralleluniverse.co/2014/05/15/modern-java-pt3/), we will discuss writing lightweight, scalable HTTP services with [Dropwizard](http://dropwizard.io/) and [Comsat](http://docs.paralleluniverse.co/comsat/), [Web Actors](http://blog.paralleluniverse.co/2014/01/28/web-actors-1/), and dependency injection with JSR-330.

[Discuss on Hacker News](https://news.ycombinator.com/item?id=7715901)

1. Maven repositories also usually store not just the binaries, but additional artifacts like sources and Javadocs; those are downloaded by IDEs to allow stepping into third-party code and to provide inline documentation for library calls. [↩](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fnref:1)
2. While Capsule is – for the time being, at least – managed by Parallel Universe, I hope it becomes a community-led project. [↩](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fnref:2)
3. I know some of you are thinking that running Node.js code on the JVM is like taking a chocolate-smeared four-year-old from school in a brand new Ferrari, but I’m not passing judgement. [↩](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fnref:3)
4. For more information about how the JVM works, you can watch these talks: [A JVM does that?](https://www.youtube.com/watch?v=uL2D3qzHtqY) by Cliff Click, and [JVM Mechanics – A Peek Under the Hood](http://www.infoq.com/presentations/JVM-Mechanics) by Gil Tene) [↩](http://blog.paralleluniverse.co/2014/05/08/modern-java-pt2/#fnref:jvm)

# Client-side server monitoring with Jolokia and JMX - DZone DevOps

https://dzone.com/articles/client-side-server-monitoring

*The DevOps Zone is brought to you in partnership with Sonatype Nexus. The*[*Nexus Suite*](https://www.sonatype.com/nexus-lifecycle?utm_source=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016&utm_medium=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016&utm_campaign=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016)*helps scale your DevOps delivery with continuous component intelligence integrated into development tools, including Eclipse, IntelliJ, Jenkins, Bamboo, SonarQube and more.*[*Schedule a demo today*](https://www.sonatype.com/nexus-lifecycle?utm_source=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016&utm_medium=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016&utm_campaign=DZONE%20-%20Nexus%20Lifecycle%20-%20September%202016)*.*

The choice of Java monitoring tools is tremendous (random selection and order powered by Google):

Besides, there are various dedicated tools e.g. for [ActiveMQ](http://activemq.apache.org/web-console.html), [JBoss](https://community.jboss.org/wiki/WebConsole), [Quartz scheduler](http://code.google.com/p/jwatch),[Tomcat/tcServer](http://blog.springsource.org/2010/03/10/springsource-tc-server-2-0)... So which one should you use as an ultimate monitoring dashboard? Well, none of them provide out-of-the-box features you might require. In some applications you have to constantly monitor the contents and size of a given JMS queue. Others are known to have memory or CPU problems. I have also seen software where system administrators had to constantly run some SQL query and check the results or even parse the logs to make sure some essential background process is running. Possibilities are endless, because it really depends on the software and its use-cases. To make matters worse, your customer doesn't care about GC activity, number of open JDBC connections and whether this nasty batch process is not hanging. It should just work.   
  
In this post we will try to develop easy, cheap, but yet powerful management console. It will be built around the idea of a single binary result – it works or not. If this single health indicator is green, no need to go deeper. But! If it turned red, we can easily drill-down. It is possible because instead of showing hundreds of unrelated metrics we will group them in a tree-like structure. The health status of each node in a tree is as bad as the worst child. This way if anything bad happens with our application, it will bubble-up.   
  
  
We are not forcing system administrator to constantly monitor several metrics. We decide what is important and if even tiniest piece of our software is malfunctioning, it will pop-up. Compare this to a continuous integration server that does not have green/red builds and e-mail notifications. Instead you have to go to the server every other build and manually check whether the code is compiling and all tests were green. The logs and results are there, but why parse them and aggregate manually? This is what we are trying to avoid in our home-grown monitoring solution.   
  
As a foundation I have chosen (not for the [first time](http://nurkiewicz.blogspot.com/search/label/jolokia)) [Jolokia](http://draft.blogger.com/blogger.g?blogID=6753769565491687768) JMX to HTTP bridge. JVM already provides the monitoring infrastructure so why reinvent it? Also thanks to Jolokia the whole dashboard can be implemented in JavaScript on the client side. This has several advantages: server footprint is minimal, also it allows us to rapidly tune metrics by adding them or changing alert thresholds.   
  
We'll start by downloading various JMX metrics onto the client (browser). I have developed some small application for demonstration purposes employing as many technologies as possible – Tomcat, Spring, Hibernate, ActiveMQ, Quartz, etc. I am not using the built-in [JavaScript client library](http://www.jolokia.org/client/javascript.html) for Jolokia as I found it a bit cumbersome. But as you can see it is just a matter of a single AJAX call to fetch great deal of metrics.

function request() {

var mbeans = [

"java.lang:type=Memory",

"java.lang:type=MemoryPool,name=Code Cache",

"java.lang:type=MemoryPool,name=PS Eden Space",

"java.lang:type=MemoryPool,name=PS Old Gen",

"java.lang:type=MemoryPool,name=PS Perm Gen",

"java.lang:type=MemoryPool,name=PS Survivor Space",

"java.lang:type=OperatingSystem",

"java.lang:type=Runtime",

"java.lang:type=Threading",

'Catalina:name="http-bio-8080",type=ThreadPool',

'Catalina:type=GlobalRequestProcessor,name="http-bio-8080"',

'Catalina:type=Manager,context=/jmx-dashboard,host=localhost',

'org.hibernate:type=Statistics,application=jmx-dashboard',

"net.sf.ehcache:type=CacheStatistics,CacheManager=jmx-dashboard,name=org.hibernate.cache.StandardQueryCache",

"net.sf.ehcache:type=CacheStatistics,CacheManager=jmx-dashboard,name=org.hibernate.cache.UpdateTimestampsCache",

"quartz:type=QuartzScheduler,name=schedulerFactory,instance=NON\_CLUSTERED",

'org.apache.activemq:BrokerName=localhost,Type=Queue,Destination=requests',

"com.blogspot.nurkiewicz.spring:name=dataSource,type=ManagedBasicDataSource"

];

return \_.map(mbeans, function(mbean) {

return {

type:'read',

mbean: mbean

}

});

}

$.ajax({

url: 'jmx?ignoreErrors=true',

type: "POST",

dataType: "json",

data: JSON.stringify(request()),

contentType: "application/json",

success: function(response) {

displayRawData(response);

}

});

Just to give you an overview what kind of information is accessible on the client side, we will first dump everything and display it on [jQuery UI](http://jqueryui.com/) [accordion:](http://jqueryui.com/demos/accordion/)

function displayRawData(fullResponse) {

\_(fullResponse).each(function (response) {

var content = $('<pre/>').append(JSON.stringify(response.value, null, '\t'));

var header = $('<h3/>').append($("<a/>", {href:'#'}).append(response.request.mbean));

$('#rawDataPanel').

append(header).

append($('<div/>').append(content));

});

$('#rawDataPanel').accordion({autoHeight: false, collapsible: true});

}

Remember that this is just for reference and debug purposes, we are **not** aiming to display endless list of JMX attributes.

As you can see it is actually possible to implement complete jconsole port inside a browser with Jolokia and JavaScript... maybe next time (anyone care to help?). Back to our project, let's pick few essential metrics and display them in a list:

The list itself looks very promising. Instead of displaying charts or values I have assigned an icon to each metric (more on that later). But I don't want to go through the whole list all the time. Why can't I just have a single indicator that aggregates several metrics? Since we are already using [jsTree](http://www.jstree.com/), the transition is relatively simple: 

On the first screenshot you see a healthy system. There is really no need to drill down since Overallmetric is green. However the situation is worse on the second screenshot. System load is alarmingly high, also the Swap space needs attention, but is less important. As you can see the former metrics bubbles up all the way to the overall, top metric. This way we can easily discover what is working incorrectly in our system. You might be wondering how did we achieved this pretty tree while at the beginning we only had raw JMX data? No magic here, see how am I constructing the tree:

function buildTreeModel(jmx) {

return new CompositeNode('Overall', [

new CompositeNode('Servlet container', [

new Node(

'Active HTTP sessions',

jmx['Catalina:context=/jmx-dashboard,host=localhost,type=Manager'].activeSessions,

Node.threshold(200, 300, 500)

),

new Node(

'HTTP sessions create rate',

jmx['Catalina:context=/jmx-dashboard,host=localhost,type=Manager'].sessionCreateRate,

Node.threshold(5, 10, 50)

),

new Node(

'Rejected HTTP sessions',

jmx['Catalina:context=/jmx-dashboard,host=localhost,type=Manager'].rejectedSessions,

Node.threshold(1, 5, 10)

),

new Node(

'Busy worker threads count',

jmx['Catalina:name="http-bio-8080",type=ThreadPool'].currentThreadsBusy,

Node.relativeThreshold(0.85, 0.9, 0.95, jmx['Catalina:name="http-bio-8080",type=ThreadPool'].maxThreads)

)

]),

//...

new CompositeNode('External systems', [

new CompositeNode('Persistence', [

new Node(

'Active database connections',

jmx['com.blogspot.nurkiewicz.spring:name=dataSource,type=ManagedBasicDataSource'].NumActive,

Node.relativeThreshold(0.75, 0.85, 0.95, jmx['com.blogspot.nurkiewicz.spring:name=dataSource,type=ManagedBasicDataSource'].MaxActive)

)

]),

new CompositeNode('JMS messaging broker', [

new Node(

'Waiting in "requests" queue',

jmx['org.apache.activemq:BrokerName=localhost,Destination=requests,Type=Queue'].QueueSize,

Node.threshold(2, 5, 10)

),

new Node(

'Number of consumers',

jmx['org.apache.activemq:BrokerName=localhost,Destination=requests,Type=Queue'].ConsumerCount,

Node.threshold(0.2, 0.1, 0)

)

])

])

]);

}

The tree model is quite simple. Root node can have a list of child nodes. Every child node can be either a leaf representing a single evaluated JMX metric or a composite node representing set of grandchildren. Each grandchild can in turns be a leaf or yet another composite node. Yes, it is a simple example of [Composite](http://en.wikipedia.org/wiki/Composite_pattern) design pattern! However it is not obvious where [Strategy](http://en.wikipedia.org/wiki/Strategy_pattern) pattern was used. Look closer, each leaf node object has three properties: label (what you see on the screen), value (single JMX metric) and an odd function Node.threshold(200, 300, 500)... What is it? It is actually a higher order function (function returning a function) used later to interpret JMX metric. Remember, the raw value is meaningless, it has to be interpreted and translated into good-looking icon indicator. Here is how this implementation works:

Node.threshold = function(attention, warning, fatal) {

if(attention > warning && warning > fatal) {

return function(value) {

if(value > attention) { return 1.0; }

if(value > warning) { return 0.5; }

if(value > fatal) { return 0.0; } else { return -1.0; }

}

}

if(attention < warning && warning < fatal) {

return function(value) {

if(value < attention) { return 1.0; }

if(value < warning) { return 0.5; }

if(value < fatal) { return 0.0; } else { return -1.0; }

}

}

throw new Error("All thresholds should either be increasing or decreasing: " + attention + ", " + warning + ", " + fatal);

}

Now it becomes clear. The function receives level thresholds and returns a function that translates them to number in -1:1 range. I could have returned icons directly but I wanted to abstract tree model from GUI representation. If you now go back to Node.threshold(200, 300, 500) example of Active HTTP sessions metric it is finally obvious: if the number of active HTTP sessions exceed 200, show attention icon instead of OK. If it exceeds 300, warning appears. Above 500 fatal icon will appear. This function is a strategy that understands the input and handles it somehow.  
  
Of course these values/functions are only examples, but this is where real hard work manifests – for each JMX metric you have to define a set of sane thresholds. Is 500 HTTP sessions a disaster or only a high load we can deal with? Is 90% CPU load problematic or maybe if it is really low we should start worrying? Once you fine-tune these levels it should no longer be required to monitor everything at the same time. Just look at the top level **single metric**. If it is green, have a break. If it is not, drill-down in few seconds to find what the real problem is. Simple and effective. And did I mention it does not require any changes on the server side (except adding Jolokia and mapping it to some URL)?  
  
Obviously this is just a small proof-of-concept, not a complete monitoring solution. However if you are interested in trying it out and improving, the whole source code is [available](https://github.com/nurkiewicz/jmx-dashboard) - as always on my GitHub [account](https://github.com/nurkiewicz).

From[*http://nurkiewicz.blogspot.com/2012/02/client-side-server-monitoring-with.html*](http://nurkiewicz.blogspot.com/2012/02/client-side-server-monitoring-with.html)

*The DevOps Zone is brought to you in partnership with Sonatype Nexus. Use the*[*Nexus Suite*](https://www.sonatype.com/get-nexus-sonatype?utm_source=DZONE%20-%20Get%20Nexus%20-%20September%202016&utm_medium=DZONE%20-%20Get%20Nexus%20-%20September%202016&utm_campaign=DZONE%20-%20Get%20Nexus%20-%20September%202016)*to automate your software supply chain and ensure you're using the highest quality open source components at every step of the development lifecycle.*[*Get Nexus today*](https://www.sonatype.com/get-nexus-sonatype?utm_source=DZONE%20-%20Get%20Nexus%20-%20September%202016&utm_medium=DZONE%20-%20Get%20Nexus%20-%20September%202016&utm_campaign=DZONE%20-%20Get%20Nexus%20-%20September%202016)*.*

# jquery.flot.js简介 - Ada zheng - 博客园

http://www.cnblogs.com/ada-zheng/p/3760913.html

[jquery.flot.js简介](http://www.cnblogs.com/ada-zheng/p/3760913.html)

[JQuery图表插件之Flot](http://www.cnblogs.com/wintersun/archive/2012/10/17/2727431.html)

[Flot](https://github.com/flot/flot)是一个Jquery下图表插件，具有简单使用，交互效果，具有吸引力外观特点。目前支持 Internet Explorer 6+, Chrome, Firefox 2+, Safari 3+ and Opera 9.5+ 等浏览器，是一个基于Javascript和Jquery纯客端户的脚本库，下面看一个简单的示例，先插入js:

<script language="javascript" type="text/javascript" src="../jquery.js"></script>

<script language="javascript" type="text/javascript" src="../jquery.flot.js"></script>

如果要支持IE9以下的浏览器,您需要使用[Excanvas](http://code.google.com/p/explorercanvas/), 一个canvas 模拟器，所以还需要加入这段标签:

<!--[if lte IE 8]><script language="javascript" type="text/javascript" src="excanvas.min.js"></script><![endif]-->

然后放置一个DIV：

<div id="placeholder" style="width:600px;height:300px;"></div>

接着Data:

<script type="text/javascript">

$(function () {

var d1 = [];

for (var i = 0; i < 14; i += 0.5)

d1.push([i, Math.sin(i)]);

var d2 = [[0, 3], [4, 8], [8, 5], [9, 13]];

// a null signifies separate line segments

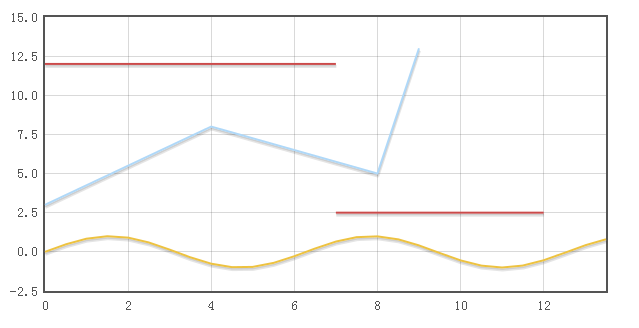
var d3 = [[0, 12], [7, 12], null, [7, 2.5], [12, 2.5]];

$.plot($("#placeholder"), [ d1, d2, d3 ]);

});

</script>

打开页面你就能看到这样的效果了：

[](http://images.cnblogs.com/cnblogs_com/wintersun/201210/201210171140232568.png)

这是一个简单的示例，它的特色之一是支持Ajax动态显示，请查看官方的[示例](http://people.iola.dk/olau/flot/examples/ajax.html)，支持JSON的数据格式。同样，它也是开源的，您可以在这儿找到它的[源代码](https://github.com/flot/flot)   
希望对您Web开发有帮助。

# JS绘图Flot应用-简单曲线图 - JS,插件,绘图,flot,曲线 - web - ITeye论坛

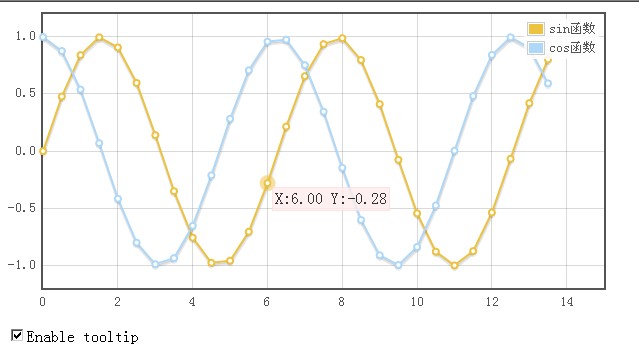
http://www.iteye.com/topic/1122003

首先对Flot做简单介绍：

[flot 是一个基于jquery的开源javascript库,是一个纯粹的 jQuery JavaScript 绘图库](http://www.iteye.com/)，可以在客户端即时生成图形，使用非常简单，支持放大缩小以及鼠标追踪等交互功能。该插件支持 IE6/7/8/9, Firefox 2.x+, Safari 3.0+, Opera 9.5+ 以及 Konqueror 4.x+。使用的是 Safari 最先引入的 Canvas 对象(html5中新增的对象)，目前所有主流浏览器都支持该对象， IE8以下等不支持的浏览器, 使用 JavaScript 进行模拟。

由于浏览器的支持问题，我们在做页面时一共需要三个页面，JQuery库、Flot库、excanvas.js这三个文件。

[做出后效果如下，这个例子是对官方例子的简单修改而成的，增加了一些注释](http://www.iteye.com/)。



我们来看一下代码：

**Html代码  [收藏代码](javascript:void())**

1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"**>**
2. **<html** xmlns="http://www.w3.org/1999/xhtml"**>**
3. **<head>**
4. **<meta** http-equiv="Content-Type" content="text/html; charset=gb2312" **/>**
5. **<title>**Flot曲线图**</title>**
6. <!--[if lte IE 8]><script language="javascript" type="text/javascript" src="excanvas.min.js"></script><![endif]-->
7. **<script** language="javascript" type="text/javascript" src="jquery.js"**></script>**
8. **<script** language="javascript" type="text/javascript" src="jquery.flot.js"**></script>**
9. **<script** type="text/javascript"**>**
10. $(function () {
11. var sin = [], cos = [];
12. // 初始化数据
13. for (var i = 0; i **<** **14**; i += 0.5) {
14. sin.push([i, Math.sin(i)]);
15. cos.push([i, Math.cos(i)]);
16. }
17. var plot = $.plot(
18. $("#placeholder"),
19. [ { data: sin, label: "sin函数"}, { data: cos, label: "cos函数" } ], // 数据和右上角含义的提示
20. {
21. series: {
22. lines: { show: true }, // 点之间是否连线
23. points: { show: true } // 是否显示点
24. },
25. grid: { hoverable: true, clickable: true }, // 是否可以悬浮，是否可以点击
26. yaxis: { min: -1.2, max: 1.2 }, // Y 轴 的最大值和最小值
27. xaxis: { min: 0, max: 15 }      // X 轴 的最大值和最小值
28. });
29. var previousPoint = null;
30. // 邦定事件
31. $("#placeholder").bind("plothover", function (event, pos, item) {
32. if ($("#enableTooltip:checked").length **>** 0) { // 如果允许提示
33. if (item) {
34. if (previousPoint != item.dataIndex) {
35. previousPoint = item.dataIndex;
36. $("#tooltip").remove();
37. var x = item.datapoint[0].toFixed(2),
38. y = item.datapoint[1].toFixed(2);
39. showTooltip(item.pageX, item.pageY,
40. "X:" + x + " Y:" + y);
41. //item.series.label + " of " + x + " = " + y); // 悬浮点时提示的内容
42. }
43. }else {
44. $("#tooltip").remove();
45. previousPoint = null;
46. }
47. }
48. });
49. // 悬浮点时进行提示
50. function showTooltip(x, y, contents) {
51. $('**<div** id="tooltip"**>**' + contents + '**</div>**').css( {
52. position: 'absolute',
53. display: 'none',
54. top: y + 5,
55. left: x + 5,
56. border: '1px solid #fdd',
57. padding: '2px',
58. 'background-color': '#fee',
59. opacity: 0.80
60. }).appendTo("body").fadeIn(200);
61. }
62. });
63. **</script>**
64. **</head>**
65. **<body>**
66. **<div** id="placeholder" style="width:600px;height:300px"**></div>**
67. **<p><input** id="enableTooltip" type="checkbox"**>**Enable tooltip**</p>**
68. **</body>**
69. **</html>**

以上我们初始化一些数据， 然后进行设置，其中可以设置是否能够提示！

示例简单，希望能够帮助一些人吧，最后示例需要的文件和示例页面如下。

欢迎大家支持我的博客：<http://cuisuqiang.iteye.com/> ！

* [flot.rar](http://dl.iteye.com/topics/download/ceb000ec-d819-32bb-ad66-f44d06a9a775) (85.7 KB)
* 下载次数: 321