JEP 158: Unified JVM Logging

Authors	Staffan Larsen, Fredrik Arvidsson, Marcus Larsson
Owner	Marcus Larsson
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	9
Component	hotspot / svc
Discussion	serviceability dash dev at openjdk dot java dot net
Effort	M
Duration	M
Relates to	JEP 271: Unified GC Logging
Reviewed by	Mikael Vidstedt
Endorsed by	Mikael Vidstedt
Created	2012/02/27 20:00
Updated	2024/09/10 21:30
Issue	8046148

Summary

Introduce a common logging system for all components of the JVM.

Goals

- Common command-line options for all logging
- Log messages are categorized using tags (e.g. compiler, gc, classload, metaspace, svc, jfr, ...). One message can have multiple tags (*tag-set*)
- Logging is performed at different levels: `error, warning, info, debug, trace, develop`.
- Possible to select what messages that are logged based on level.
- Possible to redirect logging to console or file.
- The default configuration is that all messages using warning and error level are output to stderr.
- File rotation of log files by size and number of files to keep (similar to what is available for GC logs today)
- Print line-at-a-time (no interleaving within same line)
- Logging messages are in human-readable plain text
- Messages can be "decorated". The default decorations are: **`uptime, level, tags`**.
- Ability to configure **which** decorations that will be printed.
- Existing '`tty->print...`' logging should use unified logging as output
- Logging can be configured dynamically at runtime via jcmd or MBeans
- Tested and supported -- should not crash if/when enabled by user/customer

Stretch goals:

- Multi-line logging: several lines can be logged in a way that keeps them together (non interleaved) when output
- Enable/disable individual log messages (for example by using `__FILE__` / `__LINE__`)
- Implement syslog and Windows Event Viewer output
- Ability to configure in **which order** decorations should be printed

Non-Goals

It is outside the scope of this JEP to add the actual logging calls from all JVM components. This JEP will only provide the infrastructure to do the logging.

It is also outside the scope of the JEP to enforce a logging format, apart from the format of the decorations and the use of human-readable plain text.

This JEP will not add logging to Java code in the JDK.

Motivation

The JVM is complex system-level component where root-cause analysis is often a difficult and time-consuming task. Without extensive serviceability features it is often close to impossible to find the root cause of intermittent crashes or performance quirks in a production environment. Fine-grained, easy-to-configure JVM logging available for use by support and sustaining engineering is one such feature.

JRockit has a similar feature and it has been instrumental in providing support to customers.

Description

Tags

The logging framework defines a set of *tags* in the JVM. Each tag is identified by its name (for example: `gc`, `compiler`, `threads`, etc). The set of tags can be changed in the source code as required. When a log message is added it should be associated with a *tag-set* classifying the information logged. A *tag-set* consists of one or more tags.

Levels

Each log message has a logging *level* associated with it. The available levels are `error`, `warning`, `info`, debug, `trace` and `develop` in increasing order of verbosity. The `develop` level is only available in non-product builds.

For each output, a logging level can be configured to control the amount of information written to that output. The alternative `off` disables logging completely.

Decorations

Logging messages are *decorated* with information about the message. Here is a list of the possible decorations:

- `time` -- Current time and date in ISO-8601 format
- `uptime` -- Time since the start of the JVM in seconds and milliseconds (e.g., `6.567s`)
- `timemillis` -- The same value as generated by `System.currentTimeMillis()`
- `uptimemillis` -- Milliseconds since the JVM started
- `timenanos` -- The same value as generated by System.nanoTime()
- `uptimenanos` -- Nanoseconds since the JVM started
- `pid` -- The process identifier
- `tid` -- The thread identifier
- `level` -- The level associated with the log message
- `tags` -- The tag-set associated with the log message

Each output can be configured to use a custom set of decorators. The order of them is always the one above though. The decorations to be used can be configured by the user in runtime. Decorations will be prepended to the log message

Example: `[6.567s][info][gc,old] Old collection complete`

Output

There are currently three types of output supported:

- stdout -- Outputs to stdout.
- stderr -- Outputs to stderr.
- text file -- Outputs to text file(s).

  Can be configured to handle file rotation based on written size and a number of files to rotate. Example: rotate log file each 10MB, keep 5 files in rotation. The files names will be appended with their number in the rotation. Example: `hotspot.log.1, hotspot.log.2, ...,`

`hotspot.log.5` Currently open file will not have any number appended. Example: `hotspot.log`. The size of the files is not guaranteed to be exactly the size configured. The size can overflow at most the size of the last log message written.

Some output types may require additional configuration. Additional output types could be easily implemented using a simple and well defined interface.

Command-line options

A new command-line option will be added, to control logging from all components of the JVM.

- `-Xlog`

Multiple arguments will be applied in the order they appear on command line. Multiple ´-Xlog´ arguments for the same output will override each other in their given order. Last configuration rules.

The following syntax will be used to configure the logging:

```
-Xlog[:option]
    option         :=  [<what>][:[<output>][:[<decorators>][:<output-options>]]]
                       'help'
                       'disable'
    what           :=  <selector>[,...]
    selector       :=  <tag-set>[*][=<level>]
    tag-set        :=  <tag>[+...]
                       'all'
    tag            :=  name of tag
    level          :=  trace
                       debug
                       info
                       warning
                       error
    output         :=  'stderr'
                       'stdout'
                       [file=]<filename>
    decorators     :=  <decorator>[,...]
                       'none'
    decorator      :=  time
                       uptime
                       timemillis
                       uptimemillis
                       timenanos
                       uptimenanos
                       pid
                       tid
                       level
                       tags
    output-options :=  <output_option>[,...]
    output-option  :=  filecount=<file count>
                       filesize=<file size>
                       parameter=value
```

The '`all`' tag is a meta tag consisting of all tag-sets available. '*' in ´tag-set´ definition denotes 'wildcard' tag match. Not using '*' means 'all messages matching exactly the specified tags'.

Omitting 'what' alltogether will default to tag-set `all` and level `info` .

Omitting 'level' will default to `info`

Omitting 'output' will default to `stdout`

Omitting 'decorators' will default to `uptime, level, tags`

The '`none`' decorator is special and used to turn off all decorations.

Levels provided are aggregated. Example, if an output is configured to use ´level´ `info`. All messages matching tags in ´what´ with log level `info`, `warning` and `error` will be output.

```
-Xlog:disable
    this turns off all logging and clears all configuration of the
    logging framework. Even warnings and errors.

-Xlog:help
    prints -Xlog usage syntax and available tags, levels, decorators
    along with some example command lines.
```

Default configuration:

```
-Xlog:all=warning:stderr:uptime,level,tags
    - default configuration if nothing is configured on command line
    - 'all' is a special tag name aliasing all existing tags
    - this configuration will log all messages with a level that
    matches ´warning´ or ´error´ regardless of what tags the
    message is associated with
```

Simple Examples:

```
-Xlog
```

is the same as

```
-Xlog:all
    - log messages using 'info' level to stdout
    - level 'info' and output 'stdout' are default if nothing else
    is provided
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc
    - log messages tagged with 'gc' tag using 'info' level to
    'stdout'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc=debug:file=gc.txt:none
    - log messages tagged with 'gc' tag using 'debug' level to
    a file called 'gc.txt' with no decorations
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc=trace:file=gctrace.txt:uptimemillis,pid:filecount=5,filesize=1M
    - log messages tagged with 'gc' tag using 'trace' level to
    a rotating fileset with 5 files with size 1MB with base name
    'gctrace.txt' and use decorations 'uptimemillis' and 'pid'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc::uptime,tid
    - log messages tagged with 'gc' tag using default 'info' level to
    default output 'stdout' and use decorations 'uptime' and 'tid'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc*=info,rt*=off
    - log messages tagged with at least 'gc' using 'info' level but turn
    off logging of messages tagged with 'rt'
    - messages tagged with both 'gc' and 'rt' will not be logged
    - default output of all messages at level 'warning' to 'stderr'
```

```
    will still be in effect

-Xlog:disable -Xlog:rt=trace:rttrace.txt
    - turn off 'all' logging, even warnings and errors, except
    messages tagged with 'rt' using 'trace' level
    - output to a file called 'rttrace.txt'
```

Complex examples:

```
-Xlog:gc+rt*=debug
    - log messages tagged with at least 'gc' and 'rt' tag using 'debug'
    level to 'stdout'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc+meta*=trace,rt*=off:file=gcmetatrace.txt
    - log messages tagged with at least 'gc' and 'meta' tag using 'trace'
    level to file 'metatrace.txt' but turn off all messages tagged
    with 'rt'
    - again, messages tagged with 'gc', 'meta' and 'rt' will not be logged
    since 'rt' is set to off
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc+meta=trace
    - log messages tagged with exactly 'gc' and 'meta' tag using 'trace'
    level to 'stdout'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect

-Xlog:gc+rt+compiler*=debug,meta*=warning,svc*=off
    - log messages tagged with at least 'gc' and 'rt' and 'compiler' tag
    using 'trace' level to 'stdout' but only log messages tagged
    with 'meta' with level 'warning' or 'error' and turn off all
    messages tagged with 'svc'
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect
```

Controlling at runtime

Logging can be controlled at runtime through Diagnostic Commands (the `jcmd` utility). Everything that can be specified on the command line can also be specified dynamically with Diagnostic Commands. Since diagnostic commands are automatically exposed as MBeans it will be possible to use JMX to change logging configuration in runtime.

Aditional support to enumerate over logging configuration and parameters will be added to the list of runtime control commands.

JVM interface

In the JVM a set of macros will be created with an API **similar** to:

```
log_<level>(Tag1[,...])(fmtstr, ...)
    syntax for the log macro
```

Example:

```
log_info(gc, rt, classloading)("Loaded %d objects.", object_count)
    the macro is checking the log level to avoid uneccessary
    calls and allocations.

log_debug(svc, debugger)("Debugger interface listening at port %d.", port_number)
```

Level information:

```
LogHandle(gc, meta, classunloading) log;
if (log.is_trace()) {
    ...
}

if (log.is_debug()) {
    ...
}
```

To avoid executing code that produces data only used for logging it is possible to ask a Log class about what log level it currently is configured as.

Performance

The different log levels should have guidelines that define the expected performance overhead for the level. For example: "warning level shouldn't affect performance; info level should be acceptable for production; debug, trace and error levels do not have performance requirements." Running with logging disabled should have as little performance impact as possible. It will always cost to log though.

Future possible extensions

In the future, it may make sense to add a Java API for writing log messages to this infrastructure, for use from classes in the JDK.

Initially, only three backends will be developed: stdout, stderr and file. Future projects could add other backends. For example: syslog, Windows Event Viewer, socket, etc.

Open issues

- Should we provide an alternative in the API to have the level provided as a parameter to the macro?
- Should decorations be surrounded by something else than `[]` to make it easier to parse the output?
- What is the exact format of the datestamp decorations? ISO 8601 is proposed.

Testing

It is extremely important that logging in itself does not cause any instabilities, thus extensive testing is required.

Functional testing will have to be done by enabling certain log messages and checking for their presence on stderr or files.

Because it will be possible to dynamically enable logging, we need to stress test this by continuously enabling and disabling logging while running applications.

The logging framework will be tested using unit tests.

Risks and Assumptions

The design outlined above may not satisfy all uses of logging in the JVM today. If that is the case, the design will have to be revisited.

Impact

- Compatibility: Log message formats will change and possibly the meaning of some JVM options.
- Security: File permissions need to be verified.
- Performance/scalability: Performance will be impacted if lots of logging is enabled.
- User experience: Command-line options will change. Logging output will change.
- I18n/L10n: Log messages will not be localized or internationalized.
- Documentation: The new options and their usage will have to be documented.