Introduction to Java multitenancy

Learn about a new feature for cloud systems in the IBM SDK Java Technology Edition, Version 7 Release 1

Graeme Johnson
Michael Dawson

August 06, 2015
(First published September 17, 2013)

The multitenant JVM recently became available as part of the IBM SDK Java™ Technology Edition, Version 7 Release 1 as a tech preview. By running multiple applications within a single multitenant JVM, a cloud system can speed applications' start times and reduce their memory footprint. This article introduces the technology behind multitenant cloud JVM and discusses the main costs and benefits.
The technical preview is no longer available. For other options on multitenant deployment, read about IBM containers that support MT deployment.

Cloud providers must weigh the cost of the infrastructure that's required to run systems and deliver services against the benefits that the providers derive. These cost-benefit considerations are pushing providers to consider various architectures. Their choices range along a spectrum from *no-sharing* to *shared multitenant* architectures. With no sharing, the provider offers hardware, software, and applications that are fully dedicated to individual customers. In shared multitenancy, multiple customers' applications are supported by using a single application, and all of the underlying hardware and software is shared.

The main tradeoff as you move along this architectural spectrum is *isolation* versus *density*. Density is the number of systems and services that can be delivered for a specific set of hardware and software. The more resources that are shared, the higher the density. Higher density lowers the provider's costs. At the same time, increased sharing reduces the level of isolation between *tenants* — the individual systems or services that are being delivered. Isolation is the degree to which one tenant can affect the activity and data of other tenants.

For Java-based tenants, positions along the architectural spectrum include either sharing or not sharing the JVM. In any architecture in which the top-level application is shared, the JVM must be shared. Sharing the JVM saves both memory and processor time. But with traditional JVM technology, sharing the JVM normally removes any remaining isolation from the infrastructure layer, requiring the top-level application itself to provide that isolation. This article introduces

the multitenant feature that's available for trial use in IBM's 7 R1 release as a tech preview (see Related topics). This feature enables deployments to gain the advantages of sharing the JVM while it maintains better isolation than can be achieved when a traditional JVM is shared.

Benefits and costs of the multitenant JVM

The main benefit of using the multitenant JVM is that deployments avoid the memory consumption that's typically associated with using multiple standard JVMs. This overhead has several causes:

- The Java heap consumes hundreds of megabytes of memory. Heap objects cannot be shared between JVMs, even when the objects are identical. Furthermore, JVMs tend to use all of the heap that's allocated to them even if they need the peak amount for only a short time.
- The Just-in-time (JIT) compiler consumes tens of megabytes of memory, because generated code is private and consumes memory. Generated code also takes significant processor cycles to produce, which steals time from applications.
- Internal artifacts for classes (many of which, such as `String` and `Hashtable`, exist for all applications) consume memory. One instance of each of these artifacts exists for each JVM.
- Each JVM has a garbage-collector helper thread per core by default and also has multiple compilation threads. Compilation or garbage-collection activity can occur simultaneously in one or more of the JVMs, which can be suboptimal as the JVMs will compete for limited processor time.

In addition to lowering memory and processing costs, the multitenant JVM also provides better isolation than running multiple applications in a single traditional JVM.

Another benefit is that after the shared JVM's first tenant starts, subsequent applications require less time to start because the JVM is already running. Reduced start times are particularly useful for short-running applications that are often used for scripting.

The main cost of using the multitenant JVM is that tenants are less isolated than multiple applications that run in separate JVMs. For example, a native crash in the multitenant JVM affects all tenants.

Also, a small performance tax results from the work the JVM must do to implement the multitenancy extensions. However, the impact of this performance hit decreases as the number of tenants increases — because you avoid the processor and memory cost from running multiple JVMs in the same system.

Using the multitenant JVM

To opt into sharing a run time with other tenants, the application user adds a single argument, `-Xmt`, to the command line when you launch the application. For example:

```
java -Xmt -jar one.jar
```

The result is that the application behaves (subject to the limitations we describe later in this article) as if it were running on a dedicated JVM. But in reality it runs side-by-side with other applications.

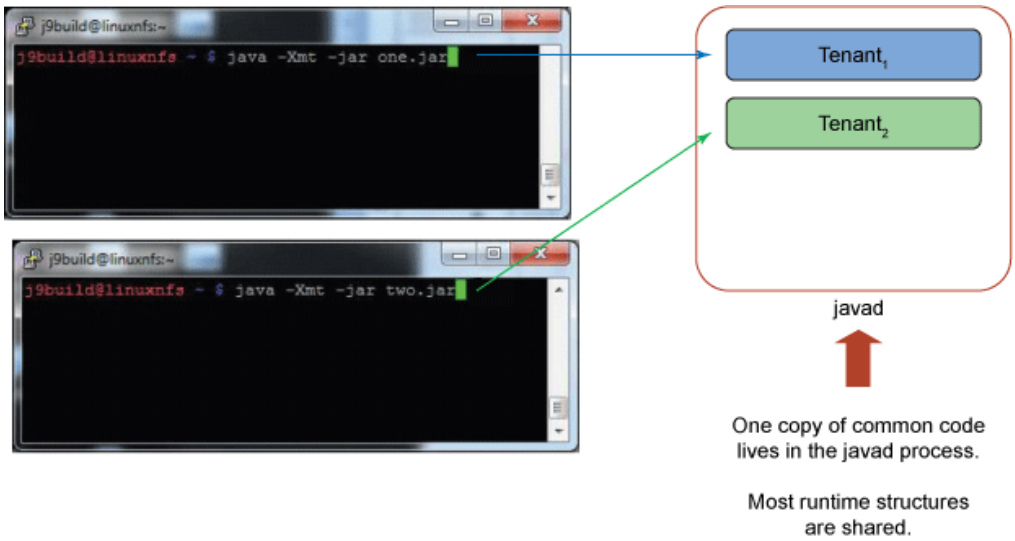
The extensions in the multitenant JVM make launching in this manner possible and provide isolation between the tenants that are sharing the JVM.

When a tenant is launched, the JVM launcher either locates the existing shared JVM daemon (`javad`) or starts it if necessary, as Figure 1 illustrates:

Figure 1. JVM launcher locates (and if necessary, starts) the shared JVM daemon automatically

When a second tenant is started, that tenant finds the existing shared JVM daemon and runs within that JVM, as Figure 2 illustrates:

Figure 2. JVM launcher locates and connects to existing JVM daemon



The result is that one copy of the bootstrap code that's common to both tenants lives in the `javad` process. This arrangement enables the tenants to share most runtime structures.

It's easy to run existing applications using the multitenant JVM, because only limited changes to the command line are required.

Achieving isolation

Two or more applications that run in the same (conventional) JVM would not normally be isolated from one another. Each application's activity would affect what the other one can get done. In

addition, data that can be shared through static fields would be accessible to all applications. The multitenant JVM addresses these issues in two ways: *static field isolation* and *resource constraints*.

Static field isolation

In the multitenant JVM, the invariant parts of classes are shared among tenants. These parts include the compiled code for methods, data structures that the JVM uses, and other similar artifacts. This sharing results in a memory savings because the separate copies that would exist if multiple JVMs were used are unnecessary. However, the multitenant JVM gives each tenant its own copy of static fields. Because of static field isolation — along with the fact that each tenant can generally only get access to instances of objects that it created — each tenant can only access data that's associated with itself. The result is data isolation between tenants.

Resource constraints: Dealing with bad behaviour

In a perfect world, tenants would co-operate and use shared resources in an appropriate manner. However, bugs and malicious behaviour both can occur in this imperfect world. The multitenant JVM provides controls that can be configured to limit a tenant's ability to misbehave and use resources in a way that affects other tenants. Values that can be controlled include:

- Processor time
- Heap size
- Thread count
- File I/O: read bandwidth, write bandwidth
- Socket I/O: read bandwidth, write bandwidth

These controls can be specified in the `-Xmt` command line. For example:

- **`-Xlimit:cpu=10-30`** (10 percent minimum CPU, 30 percent maximum)
- **`-Xlimit:cpu=30`** (30 percent maximum CPU)
- **`-Xlimit:netIO=20M`** (maximum bandwidth of 20 Mbps)
- **`-Xms8m-Xmx64m`** (initial 8 MB heap, 64 MB maximum)

The Java 7 R1 documentation includes information on all of the available options (see Related topics).

Performance and footprint

As a test to compare application performance and memory footprint on nonshared and multitenant JVMs, we add applications to each JVM configuration until the system swaps. (When it swaps we consider the system to be "full.") In the nonshared case we run the application in a separate JVM and start a new JVM for each additional application. In the multitenant case, we run the application as another tenant in the single multitenant JVM.

Table 1 and Table 2 show the results that we achieved using a machine with 1 GB of memory and a 64-bit JVM (the compressed references JVM with the balanced garbage-collection policy in

all cases). The "Hand-tuned" column in both tables shows the results from the regular JVM after we hand-tuned the command-line options to try to achieve the best possible density (Table 1) or startup times (Table 2). The Default column shows the results for the regular JVM with the default options.

The multitenant JVM achieved 1.2x to 4.9x times the density of the nonshared JVM — depending on the application — as shown in Table 1:

Table 1. Maximum number of concurrent applications

Application	Description	Multitenant	Hand-tuned	Default	Improvement with multitenant JVM
Hello World	Print "HelloWorld" and then sleep	309	73	63	4.2X to 4.9X
Jetty	Start Jetty and wait for requests	34	-	18	1.9X
Tomcat	Start Tomcat and wait for requests	28	-	13	2.1X
JRuby	Start JRuby and wait for requests	32	26	15	1.2X to 2.1X

The increased density results from the sharing of key artifacts, including:

- Classes and related artifacts that are loaded by the bootstrap and extensions class loaders, the heap `Class` object for each of the classes that the loaders load, and heap objects that can safely be shared across tenants (for example, interned `String`s).
- JIT-compiled code and metadata for JIT-compiled classes.
- Heap: One tenant can use space that's available in the heap when it is not required by other tenants.

Table 2 shows that we achieved 1.2x to 6x faster average startup times with the multitenant JVM:

Table 2. Startup times (first/average)

Application	Description	Multitenant	Hand-tuned	Default	Improvement with multitenant JVM
Hello World	Print "HelloWorld" and then sleep	5709/138ms	514/400ms	3361/460ms	3.3X
Jetty	Start Jetty and wait for requests	7478/2116ms	-	6296/12624ms	6X
Tomcat	Start Tomcat and wait for requests	9333/6005ms	-	7802/7432ms	1.2X
JRuby	Start JRuby and wait for requests	12391/3277ms	14847/4101ms	7849/6058ms	1.25X to 1.8X

In Table 2, you can see that the startup time for the first application instance is in general slower on the multitenant JVM than on the standard JVM. This result is expected because the first instance incurs some additional startup delay due to extra path length caused by multitenancy extensions. The startup time of subsequent instances is consistently better for the multitenant JVM.

These early results were produced with development JVMs, and more improvements are possible. Further, these examples don't factor in the sharing that can take place when applications need resources at different times. In a typical JVM, the memory footprint for each JVM tends to grow toward the ceiling that it requires over its lifetime. In the standard JVM much of this footprint is not shared. With the multitenant JVM, if the resource requirements do not overlap, memory for heap and native artifacts can be more easily shared.

Limitations

A goal of the multitenant JVM is to be able to run all Java applications without modification. This is not currently possible because of limitations that the Java specifications impose and limitations in our current implementation. The key known limitations include:

- **Java Native Interface (JNI) natives**: The multitenant JVM doesn't provide isolation for JNI natives. Applications with user-supplied JNI natives might not be safe to run with the multitenant JVM. Such applications can affect the operation of the overall JVM and access data from other tenants. In cases that entail sufficient "trust" in the natives (for example, well-known middleware), the risk might be acceptable. In addition, the OS allows the shared JVM process to load only one copy of a shared library, which is where natives are located. The result is that multiple tenants can't load the same natives if they are in the same shared library.
- **Java Virtual Machine Tool Interface (JVMTI)**: Because debugging and profiling activities affect all tenants that share the JVM server, these features are currently not supported in the multitenant JDK. This is an area in which we plan more work.
- **GUI programs**: Libraries such as SWT appear to maintain global state in the native layer and so are not supported in the multitenant JDK.

Conclusion

This article introduced the multitenant JVM, how it can be used, and the costs and benefits of using it. We hope that we have piqued your interest and that you try out the beta and give us feedback. We believe that the multitenant JVM can provide significant benefits for the right environments.

Related topics

- Java 7 R1 documentation (Linux): Multitenancy feature : Learn how multiple apps can share a single JVM instance and reduce resource requirements for memory and processing power.
- JVM Support for Multitenant Applications: View Graeme Johnson's JavaOne 2012 presentation on multitenant JVMs.
- Java 7 R1 downloads (Linux) : Download the IBM Developer Kit for Linux, Java Edition (and User Guides).
- Java 7 R1 downloads (AIX) : Download Developer Kits, User Guides and Service information for Java Standard Edition on AIX and IBM Websphere Real Time on AIX.
- Java 7 R1 downloads (z/OS) : Get Java products for z/OS that are full function and have passed the Java compatible test suites.