Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 264 (4) (2011) 51–57
www.elsevier.com/locate/entcs

JRebel Tool Demo

Jevgeni Kabanov[1]

Dept. of Computer Science
University of Tartu
Tartu, Estonia

Abstract

JRebel started as an academical project that became a successful commercial product used by thousands of developers worldwide. It extends the Java Virtual Machine with a mechanism that allows seamless class reloading. It uses bytecode manipulation extensively, both for the just-in-time class translator and numerous integrations with the Java SE and EE APIs. In this live demo we will show how it can be used in real-life projects to cut development time by 8 to 18 per cent.

Keywords: bytecode, JRebel, ClassLoader, API, retroactive

1 Introduction

Java EE development day-to-day activities involves deploying the application to the Java EE server. This step is necessary after the application has been compiled and packaged into an archive as per Java EE specification. Every time developers want to make changes to the running application they need to deploy it, which can take from a few seconds in the best case to several minutes in the worst.

An alternative way to update an application is using the HotSwap protocol [1], available from the Java EE debugger. This allows to update the application classes without redeploying it. Unfortunately only a very restricted set of changes is allowed; namely HotSwap allows changes to the method bodies, but does not allow changing the class signature or inheritance hierarchy. Thus no new methods, fields or constructors are allowed.

At the end of 2006 we came up with an idea for extending the Java virtual machine with a mechanism that would allow to change the class bytecode beyond the limits of the HotSwap protocol [2]. During 2007 we developed and released a prototype initially code-named “Badger” and for the public release renamed to

[1] Email: ekabanov@gmail.com

doi:10.1016/j.entcs.2011.02.005

"JavaRebel". In 2009 we released the version 2.0, which supported a layer of indirection on top of the ClassLoader API that allowed the users to edit classes and resources in their workspace instead of packaging them into .WAR or .EAR archives as per Java EE specification. We also introduced an extension API that allowed to make use of JavaRebel features in third-party applications as well as build plugins for JavaRebel to support changes in framework configuration. We also renamed the tool once more to "JRebel" due to trademark issues.

When we started working on the tool most of the research was focused on using ClassLoaders to dynamically update code [3] or on modifying JVMs to do so [4]. Recently there has been more investigation into similar systems [5,6], but no industry tools are available to compete with JRebel.

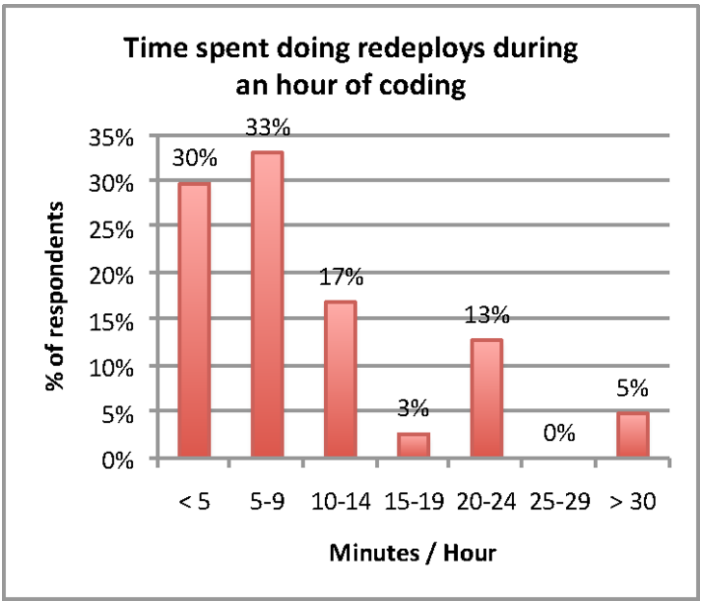
At the moment we estimate over ten thousand of JRebel users around the world. This number includes commercial users as well as various otherwise licensed users, e.g. Open Source developers and Scala developers who can request a free license.

The rest of the paper is organized as follows. In Section 2 we review the problem and its scope, Section 3 gives a brief overview of the tool, Section 4 describes the technical background and Section 5 covers the supporting artifacts and third-party extensions.

This work was partially supported by the OÜ Tarkvara Tehnoloogia Arenduskeskus, Enterprise Estonia and Estonian Science Foundation grant No. 8421.

2 Problem Scope

In 2009 we conducted a survey reaching over 1000 Java developers to estimate the amount of time spent in the redeploy phase of development. The survey asked how long a server redeploy takes and how many times an hour they are performed and is summarized by the following chart:



The average is about *10.5 minutes per hour*, accounting for *17.5%* of total development time. The standard deviation is *8*, which means that the actual per cent varies quite a lot. The estimated number of Java developers worldwide is *nine million*. Estimating conservatively that only half of them develop for Java EE we have an estimate for the annual worldwide cost to the economy of *$56,700,000,000* a year (assuming 48 working weeks a year, 5 hours of coding per day and a $30 per hour salary).

3 Tool Overview

JRebel installs a `-javaagent` JVM plugin that monitors the classes and resources in the workspace and propagates their changes to the running application. The following types of changes are supported:

- Changes to Java classes beyond what is supported by HotSwap.
- Changes to framework configuration (e.g. Spring XML files and annotations, Struts mappings, etc).
- Any changes to static resources (e.g. JSPs, HTMLs, CSSs, XMLs, .properties, etc)

JRebel works by rewriting the bytecode of Java classes to enable versioning. To do that we use just-in-time bytecode translation in a manner akin to dynamic languages compilation and runtime support. This enables us to support changes to Java class schema, though not to the type hierarchy.

Type of change	HotSwap	JRebel
Changes to method bodies	Yes	Yes
Adding/removing methods	No	Yes
Adding/removing constructors	No	Yes
Adding/removing fields	No	Yes
Replacing superclass	No	No
Adding/removing implemented interfaces	No	No

Since version 2.0 we extend the ClassLoader API to allow injecting classes and resources from locations outside default classpath. We use this functionality to allow our users to specify the layout of their projects on the filesystem using a `rebel.xml` configuration file and make application servers read the classes and resources directly from those projects, instead of the .WAR or .EAR archives as prescribed by the Java EE specification. As most of the build phase time is spent packaging those classes and resources into the archive, it allows us to save most of the time spent in that phase.

To make the tool more convenient to use we provide IDE plugins for Eclipse, IntelliJ IDEA and NetBeans. These plugins improve debugging with JRebel by

hiding the synthetic fields and methods introduced by the translation process. We also provide a plugin for the Maven build system, that automatically generates the `rebel.xml` configuration file necessary to make use of the project direct mapping functionality.

4 Technical Background

To explain how JRebel works we need to start with the reasons why support for full schema change was not implemented in Java HotSwap. This section draws mainly on [7,8] and some private conversations with Thomas Wuerthinger.

When loaded into the JVM, an object is represented by a structure in memory, occupying a continuous region of memory with a specific size (its fields plus metadata). In order to add a field, we would need to resize that structure, but since nearby regions may already be occupied, we would need to relocate the whole structure to a different region where there is enough free space to fit it in. Now, since we're actually updating a class (and not just a single object) we would have to do this to every object of that class.

Fortunately object relocation is something that Java does all the time. Java garbage collectors relocate objects every time they compact the heap. However the problem is that the abstraction of one heap is just that, an abstraction. The actual layout of memory depends on the garbage collector that is currently active and, to be compatible with all of them, the relocation would have to be implemented in the active garbage collector. This, however, presents quite a challenge, as the Sun JVM features at least four garbage collectors (some of them multi-threaded), two JIT compilers and a multitude of hardware platforms and operating systems it supports. Implementing this functionality in each of the garbage collectors and ensuring compatibility with the other components of the environment is a challenging enough task that since the 2001 when the initial HotSwap implementation the full schema update has yet to make it into the Sun JVM.

It would seem that adding methods to classes would be easier, but due to optimizations in the class layout (specifically inlined v-tables), it assumes resizing and relocating the class structure, returning to the same issue.

How does JRebel solve this problem?—JRebel works on a different level of abstraction than HotSwap. Whereas HotSwap works at the virtual machine level and is dependent on the inner workings of the JVM, JRebel makes use of two remarkable features of the JVM abstract bytecode and classloaders. Classloaders allow JRebel to recognize the moment when a class is loaded, then translate the bytecode on-the-fly to create another layer of abstraction between the virtual machine and the executed code.

The problem in reloading classes is that once a class has been loaded it cannot be unloaded or changed; but we are free to load new classes as needed. To understand how we could theoretically reload classes, lets take a look at the implementation of JRuby [9].

Ruby is required to support any runtime changes to the object, including adding

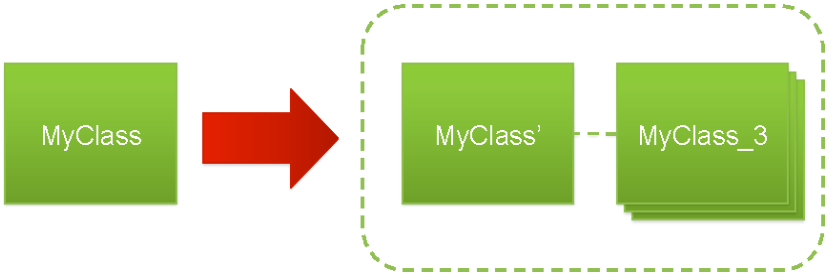
fields and methods (albeit named differently). JRuby implements those features on the JVM, by treating objects as not much more than a runtime map from method names to their implementations and from field names to their values. The implementations for those methods are contained in anonymously named classes that are generated when the method is encountered. When a method is added, JRuby generates a new anonymous class that includes the body of that method. As each anonymous class has a unique name there are no issues loading them and as a result the application is updated on-the-fly.

We could then use the same transformation as JRuby and split all Java classes into a holder class and method body classes. Unfortunately, such an approach would be subject to (at least) the following problems:

Perfomance Such a setup would mean that each method invocation would be subject to indirection. We could optimize, but the application would be at least an order of magnitude slower. Memory use would also skyrocket, as so many classes are created.

Compatibility Although Java is a static language it includes some dynamic features like reflection and dynamic proxies. If we apply the "JRuby" transformation none of those features will work unless we replace the Reflection API with our own classes, aware of the transformation.

Therefore, JRebel does not take such an approach. Instead we transform the class into a frontend class with a signature compatible to the original and an anonymous backend class, a new version of which can be loaded when the original class is updated. We rewrite all invocations among transformed classes introducing a level of indirection where necessary. However we use advanced Just-In-Time compilation techniques to inline as many indirections as we can, so as to keep performance overhead to a minimum.



To demonstrate the extent of our optimization we chose the Chameneos [10] benchmark that is highly concurrent and is implemented in multiple classes thus needing a lot of indirection in a naive implementation. Running this benchmark with JRebel agent enabled we can see that there is no discernible difference from running it in vanilla configuration. Even if we update all of the classes in the benchmark the overhead is still under 60%.

Time	Vanilla	JRebel	JRebel Updated
real	0m38.673s	0m37.457s	0m58.130s
user	1m10.747s	1m7.946s	1m38.661s
sys	0m0.821s	0m0.832s	0m1.317s

Sixty per cent may sound like a lot, but this implies that every single class in the application is updated, which is an extraordinary case. We optimize heavily to reduce overhead for the unchanged classes, as even a 60% overhead on updated classes will translate to a negligible total overhead as only a small fraction of an application is usually updated.

5 Artifacts and Extensions

The JRebel distribution includes an installer, extensive reference manual and configuration wizard. Dozens of articles available from our website and third-party publications describe various applications of JRebel in the real world. A support forum with over 2000 posts is also available to our users.

Although JRebel is a commercial product, a significant portion of its code is available as Open Source in the Subversion repository at `http://repos.zeroturnaround.com/svn/`. The parts unavailable as Open Source include the just-in-time translation engine and high-level `rebel.xml` handling.

The following Open Source artifacts are available from the Subversion repository:

Test suite To ensure the stability and compatibility of the product we have compiled a test suite that contains test cases for both JVM compatibility and application server compatibility.

Tool plugins Plugins for Eclipse, IntelliJ IDEA, NetBeans and Maven.

JRebel SDK and utils The SDK and utility classes that support writing JRebel plugins or using it in a third-party environment.

JRebel plugins Plugins for various servers (Tomcat, JBoss, Weblogic, ...) and frameworks (Spring, Struts, ...).

The JRebel SDK enables third-party contributers to submit additional plugins for JRebel. To date the plugins for Struts 2, Stripes, Wicket and Log4J have been contributed.

References

[1] Java HotSwap, `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html#hotswap`.

[2] Kabanov, Jevgeni, *METHOD AND ARRANGEMENT FOR RE-LOADING A CLASS*, 2008 (US Patent Application 20080282266).

[3] Liang, S. and Bracha, G., *Dynamic class loading in the Java virtual machine*, ACM SIGPLAN Notices **33/10**, ACM, 1998.

[4] Malabarba, Scott and Pandey, Raju and Gragg, Jeff and Barr, Earl and Barnes, J. Fritz, *Runtime Support for Type-Safe Dynamic Java Classes*, ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming [337–361], Springer-Verlag, London, 2000.

[5] Gregersen, A.R. and Jørgensen, B.N., *Extending eclipse RCP with dynamic update of active plug-ins*, Journal of Object Technology **6/6** [67–89], 2007.

[6] Pukall, M. and Kästner, C. and Saake, G., *Towards unanticipated runtime adaptation of Java applications*, Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC) [85-92], 2009.

[7] Dmitriev, Mikhail, *Towards flexible and safe technology for runtime evolution of java language applications*, OOPSLA Workshop on Engineering Complex Object-Oriented Systems for Evolution, 2001.

[8] Thomas Wuerthinger, *Dynamic Code Evolution for the Java HotSpot(TM) Virtual Machine*, 2009.

[9] Nutter, C.O. and Enebo, T.E., *JRuby – Java powered Ruby implementation*, 2003.

[10] Kaiser, C. and Pradat-Peyre, JF, *Chameneos, a Concurrency Game for Java, Ada and Others.*, ACS/IEEE Int. Conf. AICCSA03.