Retroweaver – A Developer's Guide

Gotta Get Back In Time

Toby Reyelts

10/2004

Chapter 1: Getting Started

Downloading Retroweaver

If you haven't done so already, you should download the latest version of Retroweaver and its associated documentation from http://retroweaver.sf.net.

The Fine Print

Before you begin using Retroweaver, you might like to have your well-paid lawyers review the BSD-style license for use. (NB: There's nothing really fine about fine print.)

Installation

Once you've downloaded retroweaver-<version>.zip, you can unzip the contents to your favorite folder. When you've finished that process, you'll be presented with the following set of folders:

- docs – All of the documentation for Retroweaver (including the Developer's Guide you're reading right now).
- lib – Third party libraries that Retroweaver needs in order to perform the bytecode weaving. These libraries are only required during weaving, not deployment, of your application.
- release – Contains the Retroweaver runtime library, retroweaver-rt-<version>.jar, which you must deploy with your application. Contains the full Retroweaver application, retroweaver-<version>.jar, which you use to weave your application or start the Retroweaver graphical interface. Also contains retroweaver-all-<version>.jar which includes all Retroweaver classes along with the required libraries.
- src – All of the Java source code for both the bytecode weaver and the runtime library. Feel free to browse through here and send some advice my way.
- src/com/rc/retroweaver/tests – The source for a suite of test cases that get executed against Retroweaver. If you're not sure that Retroweaver can handle a particular construct or condition, write a test case. If you're so inclined, send it my way. I'd love to include it in Retroweaver's test suite.
- release/retroweaver-tests-<version>.jar – The compiled and retroweaved classes from the test suite. You can run these to give yourself warm fuzzies about Retroweaver's behavior.

Chapter 2: An Overview

Why Retroweaver?

Sun has just released their latest and greatest version of the Java developer's kit – JDK 1.5. There is something that really differentiates this version from earlier releases, though – new language features.

JDK 1.5 has new support for generics, autoboxing, static imports, enums, extended for loops, annotations, and varargs. The Java language has just undergone its largest change ever, and if you're anything like me, you want to take advantage of all of these new wonderful enhancements. Unfortunately, if you're anything like me, you also have some problems that are likely to get in your way:

- JDK 1.5 is a new product, and you or your manager may not be very fond of relying on untested virtual machines.
- Your product may rely on other 3rd party products that aren't supported on 1.5 (persistence engines, ui frameworks, and application servers are common examples).
- Your clients may not be able to install the latest version of the JVM – a typical scenario for companies that deploy applets or applications via WebStart.

Retroweaver is a solution to all of these problems. Retroweaver enables you to develop your product using the new 1.5 Java language, while retaining binary compatibility with previous VMs – all the way back to JDK 1.2.

Using Retroweaver

If you've bothered to read this far, you're probably interested in learning how to use Retroweaver with your own product. The process is surprisingly easy.

1. Download and install JDK 1.5. (Seriously – it's very tough to develop Java programs without a Java compiler).
2. Write your source code using all of those new powerful language features that you love in 1.5.
3. Compile your code and run Retroweaver against the class files. You can run Retroweaver in a few different ways.
    - Use the graphical interface. Just double-click on the executable jar file, retroweaver-<version>.jar, or execute java -jar release/retroweaver-<version>.jar. This is a good way to start getting used to Retroweaver.
    - Run Retroweaver from the command line. That looks like:

      ```
      java -cp release\retroweaver-<version>.jar;lib\asm-
      2.2.jar;lib\asm-commons-2.2.jar com.rc.retroweaver.Weaver
      -source classes
      ```

      or when weaving a jar file:

      ```
      java -cp release\retroweaver-<version>.jar;lib\asm-
      2.2.jar;lib\asm-commons-2.2.jar com.rc.retroweaver.Weaver -jar
      input.jar output.jar
      ```

    - Add Retroweaver to your build script. This is the best way to integrate Retroweaver into your day to day development cycle. Retroweaver comes with a built-in Ant task – com.rc.retroweaver.ant.RetroWeaverTask. See the Appendix for details.
    - Weave classes on the fly at runtime with WeaveRunner, a custom ClassLoader weaving classes at runtime:

      ```
      java -cp release\retroweaver-all-<version>.jar
      com.rc.retroweaver.WeaveRunner -cp <application class path>
      <Main Class> <arguments>
      ```

4. Deploy your application, including the Retroweaver runtime library, retroweaver-rt.jar.

Chapter 3: Reference Verification

Excuse Me?

While cross-compiling your code to another virtual machine, you need to make sure you don't reference runtime code that doesn't exist in your target VM. Prior to JDK 1.5, this was most easily achieved by using the switch -Xbootclasspath/p, and pointing it to the rt.jar of your target VM. The java compiler would complain if you referenced classes, methods, or fields that didn't exist in the target VM. For lack of a better name, we'll call this process, *reference verification*.

Now What?

The bad news is that, due to the nature of Retroweaver, it is not possible to use the bootclasspath switch to perform reference verification in conjunction with Retroweaver. The good news is that Retroweaver comes with its own reference verifier. Retroweaver's reference verifier will warn you if any of your code references a class, method, or field that it can't find in your target VM. To turn on Retroweaver's reference verifier, specify the -verifyrefs switch. You'll have to pass the classpath that contains your target JDK's rt.jar, Retroweaver's runtime library, and all of the classes that your application uses. For example,

```
java -cp release\retroweaver-<version>.jar;lib\asm-
2.2.jar;lib\asm-commons-2.2.jar com.rc.retroweaver.Weaver -source
classes -verifyrefs c:\java\jdk1.4\lib\rt.jar;release\retroweaver-
rt-<version>.jar;compiled-classes;lib\lib1.jar;lib\lib2.jar
```

When Retroweaver encounters a reference to a class/method/field that can't be located on the classpath specified with -verifyrefs, it issues a warning, so you can locate and correct the issue.

Chapter 4: The Dirty Details

Why This Chapter?

Since Retroweaver functions by bytecode enhancement, it operates more-or-less, like a black box. This section strives to assuage any concerns that people may have about this technique, by explaining the underlying pinnings of Retroweaver, while avoiding the tediousness of reviewing the entire source code base.

Why Bytecode Enhancement?

There were a set of design approaches available to achieving Retroweaver's end goal – 1.5 source code running on an earlier virtual machine. The top approaches, aside from bytecode enhancement, were:

- Generate 1.4 source code from 1.5 source code.

  While this approach sounds nice, it's simply infeasible. It's just not possible to generate 1.4 source code, for all possible language constructs in 1.5. Sun's compiler team has confirmed this, itself.

- Develop a 1.5 compiler that can target 1.4.

  This is definitely a valid approach. In fact, given all the time in the world, this is probably the best approach. The primary problem, here, is that developing a 1.5 compiler is a significantly complex task – much moreso than just the development of Retroweaver. For example, to date, there are still significant bugs in Sun's own 1.5 compiler. Eventually, I believe we will see compilers for 1.5, that also support -target 1.4. Until then, there is Retroweaver.

Step By Step

The changes that Retroweaver makes to class files, can be roughly divided into two categories – format changes and runtime changes.

Format changes are changes which are required due to specification updates in the JVM. They include:

1. Replacement of "+" characters with "$" characters in identifiers.

   The new JDK 1.5 specification has relaxed the rules concerning Java language identifiers, to allow "+" characters. Retroweaver replaces the "+" characters with "$" characters, which are legal in earlier virtual machines. Retroweaver also renames class files which have "+" characters in their names.

2. Replacement of LDC and LDC_W instructions which have a CONSTANT_Class target.

   In JDK 1.5, these two instructions have been updated to work on class literals, for example, String.class. Prior to JDK 1.5, a programmer's use of a class literal resulted in the compiler generation of a method to execute a Class.forName. Retroweaver preserves the older behavior.

3. Replacement of Synthetic access specifiers with Synthetic attributes.

   In JDK 1.5, Synthetic access specifiers were introduced to replace Synthetic attributes and reduce the size of class files. Retroweaver reverses the operation by replacing Synthetic access specifiers with Synthetic attributes.

4. Removal of JDK 1.5 bit assignments from access specifiers.

   The JVM specification (second edition) states that unassigned bits of the access specifiers for classes, methods, and fields should be set to 0 by compilers and bytecode generators. It also states that JVMs must ignore those bits, but to be perfectly compliant, Retroweaver resets them to 0.

Runtime changes are changes which are required due to the addition of new classes to the JDK runtime library. They include:

1. Replacement of calls to StringBuilder with StringBuffer.

   StringBuilder is a new class introduced into 1.5 as an unsynchronized version of StringBuffer that maintains the same interface as StringBuffer. The JDK 1.5 compiler generates calls to StringBuilder, when you use the "+" operator on Strings. Retroweaver replaces those calls to StringBuilder, with calls to

StringBuffer.

2. Replacement of calls to <PrimitiveWrapper>.valueOf(<primitive>) methods.

   In JDK 1.5, the new autoboxing specification prompted the introduction of new valueOf() methods to the primitive wrapper classes, for example, Long.valueOf(long). These methods facilitate autoboxing, not only by boxing a primitive value, but by supporting other features of the autoboxing specification (i.e. mandated/optional caching behavior). Retroweaver replaces calls to these methods with calls to its own runtime library, which implements autoboxing according to the specification.

3. Replacement of references to java.lang.Enum.

   The new support for first class enumerations in JDK 1.5 requires a base enum class, java.lang.Enum. Retroweaver replaces references to java.lang.Enum with com.rc.retroweaver.runtime.Enum_, which is primarily a clone of java.lang.Enum. This means that enum values are made subclasses of Enum_, and methods which would operate on Enum, operate on Enum_, instead. Enum_ implements the enum behavior specified in the enumeration specification, including guaranteed singleton behavior, even in the face of serialization.

4. Replacement of references to java.lang.Iterable.

   The new extended for loop syntax in JDK 1.5 required a new interface, java.lang.Iterable. Retroweaver replaces references to java.lang.Iterable with its own runtime class, com.rc.retroweaver.runtime.Iterable_. This allows developers to continue to use the extended for loop and to even create implementations of Iterable, as they would with JDK 1.5.

Other Features

“Wait,”, you say, “I thought Retroweaver also supports static imports, varargs, and generics, but I saw no mention of them, here.”

It is true that Retroweaver does support these features, it's just that these features require no special support from Retroweaver.

The new static import language feature is just a compiler directive. It has no effect whatsoever on generated class files.

The new varargs language feature introduces a new access specifier, but that access specifier is used only by JDK 1.5 compilers and ignored by earlier compilers. Vararg methods will appear as vararg under 1.5, while appearing with array arguments under previous compilers, as you would expect. For example,

```
public void foo( String... ) {
}
```

would appear as

```
public void foo( String[] ) {
}
```

The addition of generics requires a new Signature attribute, which, again, is only used by JDK 1.5 compilers and ignored by earlier compilers. Generic methods appear generic under 1.5, but appear as their type-erased equivalents under previous compilers. For example,

```
public class Foo<T extends Comparable> {
  public void foo( T t ) {
  }
}
```

would appear as

```
public class Foo {
  public void foo( Comparable t ) {
  }
}
```

Chapter 5: Pitfall Harry

Frequently Asked Questions

- Why am I getting a NoClassDefFoundError/ClassNotFoundException?

  It depends.

  Is it for a class that is in com.rc.retroweaver.runtime? You should make sure you're including retroweaver-rt-<version>.jar in your application class path.

  Is it for one of the JDK classes? If you're using a class that is new to JDK 1.5, stop. Retroweaver has support for a few special classes (java.lang.Enum, java.lang.Iterable, and java.lang.StringBuilder), but that's it. You can't use any other classes that are new to JDK 1.5. You can turn on the -verifyrefs option on Retroweaver to receive warnings when you reference classes, fields, or methods that don't exist in the VM that you are targetting.

- Why am I getting a NoSuchMethodException/Error or a NoSuchFieldException/Error?

  Most likely, you're using a class that did exist prior to JDK 1.5, but a new method/field for that class that didn't exist prior to JDK 1.5. You can't use new JDK 1.5 classes/methods/fields in an earlier JVM. You can turn on the -verifyref option on Retroweaver to receive warnings when you reference classes, fields, or methods that don't exist in the VM that you are targetting.

- How does serialization for enums work?

  The new enum specification requires changes to the serialization specification, which are outside of the reach of Retroweaver. This means that serialization of enums behaves correctly, but differently, between 1.5 and earlier VMs. In other words, you can't just naively exchange enums between 1.5 and earlier VMs via serialization.

- Why do I get a java.lang.IncompatibleClassChangeError?

  Does it happen when you are using an Iterable? Currently, Retroweaver is unable to handle an assignment of a JDK 1.5 class that implements java.lang.Iterable to a java.lang.Iterable reference. See the test case, ItTest.java for more details. You can workaround this problem by wrapping the reference to the JDK 1.5 class in your own Iterable adapter.

Chapter 6: Appendix

Ant Task Documentation

Description

com.rc.retroweaver.ant.RetroWeaverTask

Runs Retroweaver on a directory or set of directories to convert classes produced by a JDK 1.5 compliant compiler to a class file format supported by older JVM's.

Parameters

Attribute	Description	Required
srcdir	The directory containing classes to process.	One of either *srcdir, inputjar* or a nested fileset element.
destdir	The destination directory for the processed classes.	No. If not specified, the processed classes overwrite the source classes.
inputjar	The jar file to proces.	One of either *srcdir, inputjar* or a nested fileset element.
outputjar	The jar file for the processed classes.	Yes if inputjar is specified.
classpath	The classpath for reference verification. For example, retroweaver-rt-<version>.jar;c:\java\jdk1.4\lib\rt.jar;my-classes	No. If not specified, Retroweaver will not verify references.
verify	Indicates whether the verifier should be called. Note that the verifier is skipped if c*lasspath* is not defined.	No. Defaults to true.
target	The target version as either "1.2", "1.3", or "1.4".	No. Defaults to "1.4".
stripSignatures	Indicates whether the generic signatures should be stripped.	No. Defaults to false.

lazy	Indicates if classes that already have the target version should be skipped. If the destination directory is different from the source directory, such classes are copied to the destination with preserved timestamp.	No. Defaults to true.
failonerror	Indicates if the build should fail if an error occurs while processing classes. If false, a warning is logged but the build continues.	No. Defaults to true.
verbose	Indicates if the names of processed files should be logged.	No. Defaults to false.

Parameters specified as nested elements

fileset

One ore more filesets can be specified instead of (or in addition to) the *srcdir* attribute. Make sure that the fileset only includes class files.

Examples

Declare the Retroweaver class. The example assumes that the property "retroweaver.home" points to the RetroWeaver installation.

```
  <taskdef name="retroweaver"
classname="com.rc.retroweaver.ant.RetroWeaverTask">
    <classpath>
      <fileset dir="${retroweaver.home}/lib" includes="**/*"/>
      <pathelement location="${retroweaver.home}/release/retroweaver-
<version>.jar"/>
    </classpath>
  </taskdef>
```

Convert a set of classes in a single directory to JDK 1.4 compatible format.

```
  <target name="weave" depends="compile">
    <retroweaver srcdir="classes"/>
  </target>
```

Convert a set of classes, using a fileset, to JDK 1.3 compatible format in another directory.

```
  <target name="weave" depends="compile">
    <mkdir dir="classes-13"/>
    <retroweaver destdir="classes-13" version="1.3">
      <fileset dir="classes">
        <include name="**/*.class"/>
      </fileset>
```

```
  </retroweaver>
</target>
```

Chapter 6: Acknowledgments

Thanks Go To

- My wife, who continually puts up with my pursuit of all things arcane.
- Neal Gafter, who was responsible for providing most of the information that made Retroweaver possible.
- Sean Shubin, who has donated some usability improvements to Retroweaver.
- Gunnar Grim, who has donated the Ant task.