Dynamic code downloading using Java RMI (Using the java.rmi.server.codebase Property)

This tutorial is organized as follows:

1.0 Starting out

One of the most significant capabilities of the Java platform is the ability to dynamically download Java software from any Uniform Resource Locator (URL) to a virtual machine (VM) running in a separate process, usually on a different physical system. The result is that a remote system can run a program, for example an applet, which has never been installed on its disk. For the first few sections of this document, codebase with regard to applets will be discussed in order to help describe codebase with regard to Java Remote Method Invocation (Java RMI).

For example, a VM running from within a web browser can download the bytecodes for subclasses of `java.applet.Applet` and any other classes needed by that applet. The system on which the browser is running has most likely never run this applet before, nor installed it on its disk. Once all the necessary classes have been downloaded from the server, the browser can start the execution of the applet program using the local resources of the system on which the client browser is running.

Java RMI takes advantage of this capability to download and execute classes and on systems where those classes have never been installed on disk. Using the Java RMI API any VM, not only those in browsers, can download any Java class file including specialized Java RMI stub classes, which enable the execution of method calls on a remote server using the server system's resources.

The notion of a codebase originates from the use of `ClassLoader`s in the Java programming language. When a Java program uses a `ClassLoader`, that class loader needs to know the location(s) from which it should be allowed to load classes. Usually, a class loader is used in conjunction with an HTTP server that is serving up compiled classes for the Java platform. Most likely, the first `ClassLoader`/codebase pairing that you came into contact with was the `AppletClassLoader`, and the "codebase" part of the `<applet>` HTML tag, so this tutorial will assume that you have some experience with Java RMI programming, as well as writing HTML files that contain applet tags. For example, the HTML source will contain something like:

```
<applet height=100 width=100 codebase="myclasses/" code="My.class">
        <param name="ticker">
</applet>
```

2.0 What is a codebase?

A codebase can be defined as a source, or a place, from which to load classes into a virtual machine. For example, if you invited a new friend over for dinner, you would need to give that friend directions to the place where you lived, so that he or she could locate your house. Similarly, you can think of a codebase as the directions that you give to a VM, so it can find your [potentially remote] classes.

You can think of your `CLASSPATH` as a "local codebase", because it is the list of places on disk from which you load local classes. When loading classes from a local disk-based source, your `CLASSPATH` variable is consulted. Your `CLASSPATH` can be set to take either relative or absolute path names to directories and/or archives of class files. So just as `CLASSPATH` is a kind of "local codebase", the codebase used by applets and remote objects can be thought of as a "remote codebase".

3.0 How does it work?

3.1 How codebase is used in applets

To interact with an applet, that applet and any classes that it needs to run must be accessible by remote clients. While applets can be accessed from "`ftp://`" or local "`file:///`" URLs, they are usually accessed from a remote HTTP server.

1. The client browser requests an applet class that is not found in the client's `CLASSPATH`
2. The class definition of the applet (and any other class(es) that it needs) is downloaded from the server to the client using HTTP
3. The applet executes on the client

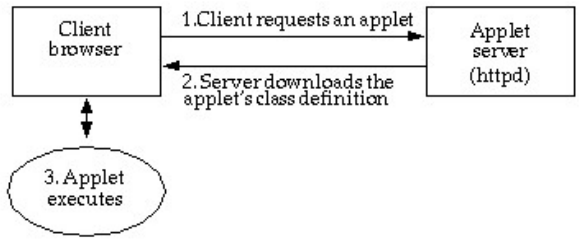

Figure 1: Downloading applets

The applet's codebase is always relative to the URL of the HTML page in which the `<applet>` tag is contained.

3.2 How codebase is used in Java RMI

Using Java RMI, applications can create remote objects that accept method calls from clients in other VMs. In order for a client to call methods on a remote object, the client must have a way to communicate with the remote object. Rather than having to program the client to speak the remote object's protocol, Java RMI uses special classes called stubs that can be downloaded to the client that are used to communicate with (make method calls on) the remote object. The `java.rmi.server.codebase` property value represents one or more URL locations from which these stubs (and any classes needed by the stubs) can be downloaded.

Like applets, the classes needed to execute remote method calls can be downloaded from "`file:///`" URLs, but like applets, a "`file:///`" URL generally requires that the client and the server reside on the same physical host, unless the file system referred to by the URL is made available using some other protocol, such as NFS.

Generally, the classes needed to execute remote method calls should be made accessible from a network resource, such as an HTTP or FTP server.

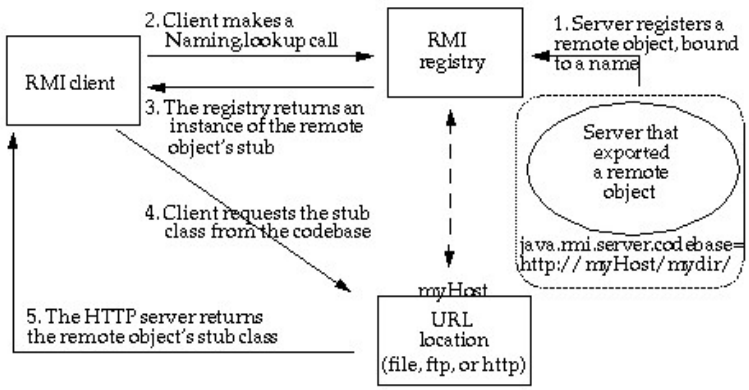

Figure 2: Downloading Java RMI stubs

1. The remote object's codebase is specified by the remote object's server by setting the `java.rmi.server.codebase` property. The Java RMI server registers a remote object, bound to a name, with the Java RMI registry. The codebase set on the server VM is annotated to the remote object reference in the Java RMI registry.
2. The Java RMI client requests a reference to a named remote object. The reference (the remote object's stub instance) is what the client will use to make remote method calls to the remote object.
3. The Java RMI registry returns a reference (the stub instance) to the requested class. If the class definition for the stub instance can be found locally in the client's `CLASSPATH` , which is always searched before the codebase, the client will load the class locally. However, if the

definition for the stub is not found in the client's `CLASSPATH`, the client will attempt to retrieve the class definition from the remote object's codebase.

4. The client requests the class definition from the codebase. The codebase the client uses is the URL that was annotated to the stub instance when the stub class was loaded by the registry. Back in step 1, the annotated stub for the exported object was then registered with the Java RMI registry bound to a name.
5. The class definition for the stub (and any other class(es) that it needs) is downloaded to the client.

   Note: *Steps 4 and 5 are the sames steps that the registry took to load the remote object class, when the remote object was bound to a name in (registered with) the Java RMI registry. When the registry attempted to load the remote object's stub class, it requested the class definition from the codebase associated with that remote object.*

6. Now the client has all the information that it needs to invoke remote methods on the remote object. The stub instance acts as a proxy to the remote object that exists on the server; so unlike the applet which uses a codebase to execute code in its local VM, the Java RMI client uses the remote object's codebase to execute code in another, potentially remote VM, as illustrated in Figure 3:

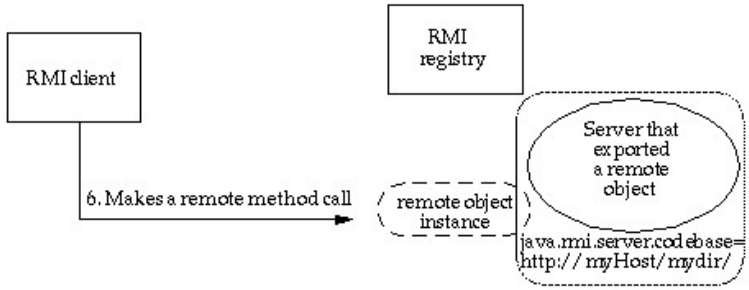

Figure 3: Java RMI client making a remote method call

4.0 Using codebase in Java RMI for more than just stub downloading

In addition to downloading stubs and their associated classes to clients, the `java.rmi.server.codebase` property can be used to specify a location from which any class, not only stubs, can be downloaded.

When a client makes a method call to a remote object, the method that it calls could be written to accept no arguments or a number of arguments. There are three distinct cases that may occur, based on the data type(s) of the method argument(s).

In the first case, all of the method parameters (and return value) are primitive data types, so the remote object knows how to interpret them as method parameters, and there is no need to check its `CLASSPATH` or any codebase.

In the second case, at least one remote method parameter or the return value is an object, for which the remote object can find the class definition locally in its `CLASSPATH`.

In the third case (shown as Step 6, in Figure 4), the remote method receives an object instance, for which the remote object cannot find the class definition locally in its `CLASSPATH`. This type of remote method call is illustrated in Figure 4. The class of the object sent by the client will be a subtype of the declared parameter type. A subtype is either:

- An implementation of the interface that is declared as the method parameter (or return) type
- A subclass of the class that is declared as the method parameter (or return) type

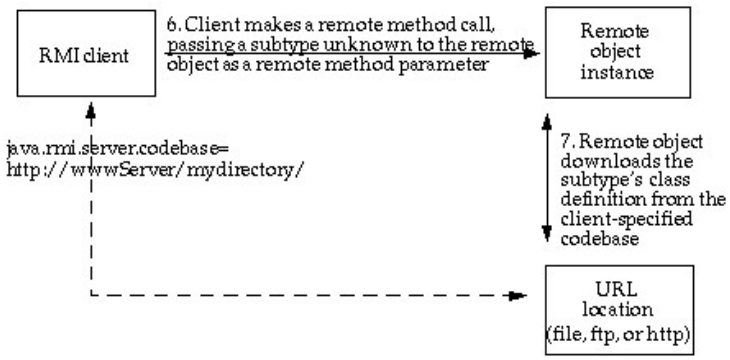

Figure 4: Java RMI client making a remote method call, passing an unknown subtype as a method parameter

7. Like the applet's codebase, the client-specified codebase is used to download `Remote` classes, non-remote classes, and interfaces to other VMs. If the `codebase` property is set on the client application, then that codebase is annotated to the subtype instance when the subtype class is loaded by the client. If the codebase is not set on the client, the remote object will mistakenly use its own codebase.

5.0 Command-line examples

In the case of an applet, the applet codebase value is embedded in an HTML page, as we saw in the HTML example in the first section of this tutorial.

In the case of Java RMI codebase, rather than having a reference to the class embedded in an HTML page, the client first contacts the Java RMI registry for a reference to the remote object. Because the remote object's codebase can refer to any URL, not just one that is relative to a known URL, the value of the Java RMI codebase must be an absolute URL to the location of the stub class and any other classes needed by the stub class. This value of the `codebase` property can refer to:

- The URL of a directory in which the classes are organized in package-named sub-directories
- The URL of a JAR file in which the classes are organized in package-named directories
- A space-delimited string containing multiple instances of JAR files and/or directories that meet the criteria above

Note: When the `codebase` property value is set to the URL of a *directory*, the value must be terminated by a "/".

Examples

If the location of your downloadable classes is on an HTTP server named "webvector", in the directory "export" (under the web root), your `codebase` property setting might look like this:

```
-Djava.rmi.server.codebase=http://webvector/export/
```

If the location of your downloadable classes is on an HTTP server named "webline", in a JAR file named "mystuff.jar", in the directory "public" (under the web root), your `codebase` property setting might look like this:

```
-Djava.rmi.server.codebase=http://webline/public/mystuff.jar
```

Now let's suppose that the location of your downloadable classes has been split between two JAR files, "myStuff.jar" and "myOtherStuff.jar". If these JAR files are located on different servers (named "webfront" and "webwave"), your `codebase` property setting might look like this:

```
-Djava.rmi.server.codebase="http://webfront/myStuff.jar http://webwave/myOtherStuff.jar"
```

6.0 Troubleshooting tips

Any serializable class, including Java RMI stubs, can be downloaded if your Java RMI programs are configured properly. Here are the conditions under which dynamic stub downloading will work:

A. The stub class and any of the classes that the stub relies on are served up from a URL reachable from the client.
B. The `java.rmi.server.codebase` property has been set on the server program (or in the case of activation, the "setup" program) that makes the call to `bind` or `rebind`, such that:
   - The value of the `codebase` property is the URL in step A

     and

   - If the URL specified as the value of the `codebase` property is a directory, it must end in a trailing "/"

C. The `rmiregistry` cannot find the stub class or any of the classes that the stub relies on in its `CLASSPATH`. This is so the codebase gets annotated to the stub when the registry does its class load of the stub, as a result of calls to `bind` or `rebind` in the server or setup code.
D. The client has installed a `SecurityManager` that allows the stub to be downloaded. This means that the client must also have a properly configured security policy file.

There are two common problems associated with the `java.rmi.server.codebase` property, which are discussed next.

6.1 If you encounter a problem running your Java RMI server

The first problem you might encounter is the receipt of a `ClassNotFoundException` when attempting to `bind` or `rebind` a remote object to a name in the registry. This exception is usually due to a malformed `codebase` property, resulting in the registry not being able to locate the remote object's stubs or other classes needed by the stub.

It is important to note that the remote object's stub implements all the same interfaces as the remote object itself, so those interfaces, as well as any other custom classes declared as method parameters or return values, must also be available for download from the specified codebase.

Most frequently, this exception is thrown as a result of omitting the trailing slash from the URL value of the property. Other reasons would include: the value of the property is not a URL; the path to the classes specified in the URL is incorrect or misspelled; the stub class or any other necessary classes are not all available from the specified URL.

The exception that you may encounter in such a case would look like this:

```
java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: examples.callback.MessageReceiverImpl_Stub
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: examples.callback.MessageReceiverImpl_Stub
java.lang.ClassNotFoundException: examples.callback.MessageReceiverImpl_Stub
        at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Compiled Code)
        at sun.rmi.transport.StreamRemoteCall.executeCall(Compiled Code)
        at sun.rmi.server.UnicastRef.invoke(Compiled Code)
        at sun.rmi.registry.RegistryImpl_Stub.rebind(Compiled Code)
        at java.rmi.Naming.rebind(Compiled Code)
        at examples.callback.MessageReceiverImpl.main(Compiled Code)
RemoteException occurred in server thread; nested exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: examples.callback.MessageReceiverImpl_Stub
```

6.2 If you encounter a problem running your Java RMI client

The second problem you could encounter is the receipt of a `ClassNotFoundException` when attempting to `lookup` a remote object in the registry. If you receive this exception in a stacktrace resulting from an attempt to run your Java RMI client code, then your problem is the `CLASSPATH` with which your Java RMI registry was started. See requirement C in section 6.0. Here is what the exception will look like:

```
java.rmi.UnmarshalException: Return value class not found; nested exception is:
        java.lang.ClassNotFoundException: MyImpl_Stub
        at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:109
        at java.rmi.Naming.lookup(Naming.java:60)
        at RmiClient.main(MyClient.java:28)
```


Contact Us