RFC 99: Automatic Class Unloading

Confidential, Draft

11 Pages

Abstract

This RFC proposes changing the OSGi specification to allow the framework to unload classes to save resources. The RFC defines the conditions under which a framework may unload a class and examines the visible side-effects of unloading a class.

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

Design notes are in this font.

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Sept. 24, 2004	First Draft. Kevin Riff, Espial, kriff@espial.com
2nd Draft	Dec. 3, 2004	Kevin Riff, Espial, kriff@espial.com • Added terminology section • Added info re interactions with selected APIs from the R3 spec

1 Introduction

The new declarative services feature [3] will allow services to be registered "on demand". To support that behaviour, R4 will include a change in the specification to allow a bundle's classloader to be created lazily. This will make framework implementations more efficient because classloaders, which are pretty hefty objects, will not be created until just before they are required. This saves on time and memory.

A complementary feature to lazy classloader creation is eager classloader removal. This change will further improve the framework's efficiency by enabling it to better manage the limited resources that are available to it on an embedded device. Section 3 makes the case that any classloader that remains in memory beyond its useful lifespan constitutes a memory leak. Section 5 shows how to plug the leak by allowing frameworks to unload classes (and classloaders) that they determine are no longer in use.

All Page Within This Box

2 Application Domain

2.1 The ClassLoader Life-Cycle

2.1.1 ClassLoaders in R3

In an R3-compliant OSGi framework, a bundle's classloader is created when the bundle is resolved. The framework is free to resolve the bundle at any time between when the bundle is installed and when the bundle is first activated. From that point on, the classloader must remain in memory at least until the bundle is updated or uninstalled. Even then, the classloader may remain if the bundle has exported any packages. The only sure way to discard the bundle's classloader is to refresh its packages.

2.1.2 ClassLoaders in R4

The R4 specification will no longer require that a bundle in the `RESOLVED` state must have a classloader. Instead, the creation of a classloader may be delayed until the absolute last possible moment – just before a class is actually loaded from the bundle. This implies that even bundles in the `ACTIVE` state may not have a classloader. This change makes it possible to have the framework eagerly resolve bundles without incurring the overhead of unnecessarily creating a classloader.

It looks like R4 will also require that bundles have exactly one classloader (The R3 spec allowed multiple classloaders per bundle) but that change doesn't have any impact on this RFC.

2.2 Running Applications on a PC

A typical desktop PC will have many applications installed, but only a few of them will be running at a time. No PC – even those with gigabytes of RAM – can afford to keep all of its applications resident in memory. Therefore it's important that only running applications consume resources (except for harddrive space, of course). If a running application is shutdown, the resources it was using are freed for other applications to use. In this way, a PC may have many more applications installed than it can actually run simultaneously.

2.3 Terminology

Application	For the purposes of this RFC, an *application* is simply one or more bundles that together perform some task for the user. This should not be confused with "MEG applications" as defined by RFC 91 [4], which are a special case of applications in general.
Strong Reference	A *strong reference* is an ordinary reference in Java that prevents an object from being reclaimed by the garbage collector. If one object holds a strong reference to another object then the second object MUST NOT be garbage collected unless the first object is also eligible for garbage collection.
Weak Reference	A *weak reference* is a reference that does not prevent its referent from being reclaimed by the garbage collector. Weak references are embodied by instances of the `java.lang.ref.Reference` class. When an object is only reachable via weak references then those references MAY be cleared and the referent reclaimed by the garbage collector. See the JavaDocs [5] for more details.

All Page Within This Box

3 Problem Description

Every class loaded into the VM consumes memory. Even very small classes can consume a surprising amount of RAM (*TODO: Need a source to support this*). If these classes are not unloaded when no longer needed then they are needlessly consuming memory that could be used for other things.

Bundles are automatically resolved by the framework, but the only transitions to an unresolved state all require explicit activation. (The allowed transitions are uninstall, update and refresh.) This asymmetry ensures that once a bundle is resolved, it tends to stay resolved. These resolved bundles can loiter in memory long after their useful lifespan, which by some definitions would constitute a memory leak.

Compare applications on the PC (section 2.2) with applications in an OSGi framework. Each application is distributed as one or more bundles. These correspond with the executable files and shared libraries that make up a normal application on a PC. As applications are loaded, their bundles are resolved and started and when the user is done with each application, its bundle(s) are stopped. However, the application's bundles are not unresolved which means that applications can linger in memory even when not in use. It's even possible that a new application could fail to run because the device's resources are tied up by applications that are no longer active.

In theory, an intelligent management agent could be designed that used refreshPackages to unresolve any bundles that are no longer required. Such an agent would need to have detailed knowledge of the dependencies between bundles. This hypothetical agent should not be too eager to unload applications – the user may intend to reload them very soon. Making an agent that can intelligently unload bundles only when the framework requires more resources is very difficult. For these reasons, using the refreshPackages method is considered impractical.

The key thing to realize is that the framework can use weak references to hold onto each bundle's classloader. Then the problem of determining which classes should be unloaded naturally falls on the garbage collector – a task that it is well suited to perform. An outside agent, on the other hand, would probably need VM-specific hooks to receive notifications when memory starts to run low so that it can respond by unresolving bundles.

4 Requirements

[req-1] Frameworks MUST be allowed (though not required) to unload classes if they can determine that those classes are no longer necessary.

[req-2] Backwards compatibility MUST be maintained. Bundles that worked on R3-compatible frameworks MUST continue to work.

[req-3] There SHOULD NOT be any undesirable side-effects of unloading the classes.

All Page Within This Box

Unfortunately there may be undesirable side-effects in some circumstances. See section 5.1.1 for details.

5 Technical Solution

The proposed solution is to allow frameworks to unload classes that they deem to be unnecessary. The following sections discuss how unloading is accomplished and under what circumstances unloading is allowed.

5.1 Unloading Classes

A class may only be unloaded from the VM if its classloader is unreachable. Since every class contains a reference to its classloader, the classloader itself will be unreachable only if all the classes it has loaded, and all the instances of those classes, are themselves unreachable. So in order to unload a class, the framework must arrange for its classloader to become unreachable.

As a historical note, JDK 1.1 allowed any class to be unloaded so long as no instances of it existed. This lead to problems for applets that had classes that were never instantiated but nevertheless held important static data. A number of workarounds were proposed until Sun modified the rules for class unloading in Java 2. (See section 5.1.1 for a discussion of the visible side-effects of unloading a class.)

JDK 1.0 did not support unloading classes so it didn't suffer from this issue.

The framework MAY unload classes that are no longer strongly referenced. If it does, the framework MUST unload all the classes loaded from the same bundle at the same time. The framework MUST hold a strong reference to each bundle's activator (if any), and to each service object in the service registry.

The rule that all classes must be unloaded at the same time is a natural consequence of Java 2's rules for class unloading. The requirement to hold a strong reference to all activators and services establishes a root set of references that determine which classloaders are "live".

Frameworks SHOULD be conservative about unloading classes. They SHOULD favor unloading classes that have not been used in a while over unloading classes that have been recently used.

The above behavior can be accomplished if the framework uses a `SoftReference` (instead of a `WeakReference`).

5.1.1 Visible Side-Effects of Unloading a Class

When a class is unloaded any static data it contains is discarded. Depending on what that data is, this could be a bad thing.

For example, consider a simple library bundle that includes a class for generating unique IDs. The class keeps a counter as static data and simply increments it each time it issues a new ID. This is a very simple way to ensure that each ID is unique. However, if the UniqueID class is reloaded then the internal counter variable will be reset. A naïve implementation might accidentally give out duplicate IDs in this case.

All Page Within This Box

The solution is for the UniqueID class to store its counter in a more durable location, like the bundle's private data area. When the class is (re)loaded, a static initializer reads the most recent value from permanent storage. Each time a new ID is issued, the class updates the value in permanent storage. (Actually, for performance reasons, it would probably "check out" multiple IDs in advance so that each request for a UID does not have to wait for I/O.) In this way, the current state of the variable is durable across reloads of this class.

It should be noted that all of the above issues also apply if the bundle's classloader is discarded as a result of a call to the refreshPackages method. Therefore, bundles that are sensitive to these issues should already be designed to handle them.

5.1.2 Preventing a Class from Being Unloaded

Bundles that include an activator (provided the activator's class is not imported from another bundle) will not be unloaded while the bundle is active. The framework is required to hold a strong reference to the bundle's activator, which indirectly holds a reference to the bundle's classloader, preventing it from being reclaimed. However, once the bundle is stopped, the framework must release its reference to the activator which would make the bundle's classes eligible for being unloaded (unless other references from other bundles still exist).

Bundles that do not include an activator, but nevertheless register one or more services are immune from being unloaded while a service is registered with the framework. The framework is required to hold a strong reference to any service object in the service registry. This case could only happen with declarative services [3].

When a bundle first loads a class or resource from another bundle, the framework must create a strong reference from the first bundle's classloader to the second bundle's classloader. This ensures that once a bundle has been exposed to a class loaded by a given classloader, it will never be exposed to the same class loaded by a different class loader. If a bundle does nothing but export a package, and that package contains only static utility classes for which there are no instances, then it will still be considered "live" so long as it has active clients using its exported package. But note that simply importing the package does not create a strong reference. Bundles must actually use a class from that package before the link will be established.

5.2 Interactions with Other OSGi APIs

5.2.1 PackageAdmin Service

The existence or non-existence of a classloader does not affect the packages that a bundle either imports or exports. If a bundle's classloader is reclaimed because it is no longer referenced, the PackageAdmin service continues to report the same imported and exported packages for that bundle. Packages exported by a bundle who's classloader was reclaimed are not pending removal, unless they were pending removal before the classloader was reclaimed.

When the framework moves a bundle from the RESOLVED state to the INSTALLED state, it MUST release any direct or indirect references it may be holding to the bundle's classloader, including weak references.

5.2.2 PermissionAdmin Service

Prior to performing a security check, the PermissionAdmin service must construct actual instances for each of the permissions registered with it. This creates an indirect reference to the bundle's classloader which may prevent it from being garbage collected. The PermissionAdmin service MUST use strong references to these Permission instances.

If the PA service used weak references instead, the Permission instances could be reclaimed without necessarily making the classloader eligible for garbage collection. These instances would need to be recreated over and over

for no good reason. Even if the classloader could be reclaimed, it would only need to be recreated for the next security check. This sort of classloader thrashing would likely be a performance problem.

5.2.3 Log Service

A log entry may contain a reference to an exception. If that exception's class was loaded from a bundle then the reference from the log service to an instance of one of the bundle's classes may prevent it from becoming unreachable and reclaimed in a timely manner.

In R3, the log service is allowed to use a proxy in place of the real exception to prevent this kind of undesirable dependency. For R4, the language of the specification should be strengthened to say that the log service MUST use a proxy if the exception's class comes from a bundle (other than the system bundle). The log service MUST NOT hold any strong references to exceptions loaded from another bundle.

The log service may also hold references to LogListeners registered with the service via the addLogListener method. A typical usage pattern is to create the listener and register it with the log service without keeping any references to it. Therefore, the log service MUST keep a strong reference to any listeners registered with it, otherwise they would be eligible for garbage collection.

5.2.4 ServiceTracker

A ServiceTracker may hold references to the services it is tracking. These MUST be strong references, otherwise the "service" associates with each service reference it is tracking could become eligible for garbage collection. This could be an issue if the tracker has been customized to associate some kind of proxy with each tracked reference, rather than the actual service.

Note that since the framework is required to hold a strong reference to the service object, the extra reference from the service tracker does not cause the bundle's classloader to loiter any longer than it should.

6 Considered Alternatives

6.1 Automatically Unresolve Unused Bundles

A variation on the mark-and-sweep garbage collection algorithm is used to determine which bundles are "in use". The framework starts with a graph containing all bundles that are in the `STARTING`, `ACTIVE` or `STOPPING` state. These form the "root set" for the garbage collection process.

Then it adds to the graph any bundles which are exporting packages to bundles in the graph. The framework repeats this step until there are no bundles outside the graph that export one or more packages to bundles in the graph.

The bundles outside of the dependency graph are considered "unused". If any of these bundles are in the `RESOLVED` state then the framework may choose to unresolve them at any time. The framework does this by setting the bundle's state back to `INSTALLED`.

This approach is designed with the R3 semantics in mind. It has the advantage that all of the visible effects of unloading classes are identical to someone invoking `refreshPackages` on the unused bundles. Therefore bundles should already be prepared to handle any issues that might arise. It also provides an easy way to "pin" the classloader and prevent it from being unloaded simply by starting the bundle.

Espial's DeviceTop product implements this approach and it has proven very effective.

However, this approach doesn't seem appropriate for the new R4 semantics. In particular, if a bundle may enter the `RESOLVED` state without first creating a classloader then it doesn't make sense that it must leave the `RESOLVED` state before the framework is allowed to discard that same classloader. There are interactions with the declarative services model [3] proposed for R4 that make this approach undesirable.

6.1.1 A Failed Use Case

This is a use case that would not work in R4:

Suppose a bundle (A) declares a service (x) that depends on some other service (y) located in a different bundle (B). A is started and the declarative services sub-system determines that x's dependencies are satisfied (because B is active and has registered the service y) so it instantiates x and registers it in the service registry. At some point later, y is removed. Since x's dependencies are no longer met, the declarative services sub-system unregisters it.

Now it may happen that another y is not registered for a very long time, if ever. It would be nice if A's classes could be unloaded to free up memory for other purposes. But A must remain in the `ACTIVE` state so that if a service y is ever registered again then the declarative services sub-system will be able to instantiate and register another service x. Therefore A cannot be unresolved which is why the above approach won't work in R4.

6.2 Garbage Collection Manifest Header

The first version of this RFC had a flaw in that bundles which only export packages would not necessarily have any strong references to keep their classloaders alive. This could lead to important static data being unloaded at an inopportune moment, or the classloader being constantly discarded and recreated.

To solve the problem, it was suggested that bundles might carry a special manifest header that declares the bundle's (un)willingness to be garbage collected. There are a number of problems with this idea:

- Class unloading is an optional feature that frameworks MAY implement, but many won't. But for correct operation, **all** bundles would need to carry the special manifest header.
- There are occasions when the framework MUST discard the bundle's classloader, such as during a refreshPackages operation. A bundle cannot declare that it is unwilling to be refreshed since that would make it difficult (or impossible) to update or uninstall the bundle.
- The side effects of letting the classloader be garbage collected are nearly identical to a refreshPackages operation. Since a refresh can happen at any time, bundles must be designed to deal with any issues that might arise regardless of their willingness or unwillingness to be garbage collected at other times.

The problem with static data being unloaded has been solved by adding a strong reference from each bundle's classloader to the classloaders of the bundles from which it is using classes or resources.

7 Security Considerations

This RFC does not introduce any new security issues.

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. RFC 80: Declarative Services. http://membercvs.osgi.org/rfcs/rfc0080/

[4]. RFC 91: MEG Application Model. http://membercvs.osgi.org/rfcs/rfc0091/

[5]. The `java.lang.ref` Package. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ref/package-summary.html

8.2 Author's Address

Name	Kevin Riff
Company	Espial Group, Inc.
Address	Suite 901, 200 Elgin St., Ottawa, ON, Canada, K2P 1L8
Voice	(613) 230-4770 x1136
e-mail	kriff@espial.com

8.3 Acronyms and Abbreviations

All Page Within This Box

8.4 End of Document