FELIX V6 ANDROID SUPPORT

Implementation Notes

Abstract

This document provides notes and insight into changes made to the Felix Framework v6 code base in order to provide Android Support.

Tom Rutchik

tom@phinneyridge.com

Contents

Apache Project: Android Support for Felix version 6

Problem statement:

Felix Framework version 6 stopped its support for Android. This was done due to fact that prior support for android was based on using the class "dalvik.system.DexFile". This class was deprecated in Android API version 26 and above. Although it still is available, Android documentation states:

This class was deprecated in API level 26.

This class should not be used directly by applications. It will hurt performance in most cases and will lead to incorrect execution of bytecode in the worst case. Applications should use one of the standard classloaders such as **`PathClassLoader`** instead. This API will be removed in a future Android release.

Android support was dropped purely for pragmatic time constraints. The purpose of this project is provide a new implementation to Felix version 6 that provides Android support that corrects this issue.

Implementation Notes:

Approach

The decision was made to use PathClassLoader for loading classes. Android compiles and stores all classes in a classes.dex file and PathClassLoader is the recommended ClassLoader to use to load classes contain in it. Its constructor takes 3 arguments: a path to dex files, a path to native code directories, and a parent ClassLoader. The strategy is to create a PathClassLoader instant for each BundleWiringImpl. The PathClassLoader is constructed with its path set to the location of where its corresponding bundle revision is stored in the Felix framework. The trick of course is to set the PathClassLoader's parent ClassLoader to be the BundleClassLoader associated with the given BundleWiringImpl instance. This is necessary so that the normal Felix class resolution process is maintained. For example, if a class loaded from the given bundle has a dependency on another class that is imported from a different bundle, we would want the PathClassLoader to be able to find it. Since it's not in the bundles dex file, the PathClassLoader can still find it because its parent (BundleClassLoader) can resolve it. Obviously, if the required class has not been properly configured to be located according to the osgi bundle configuration rules, then the class load needs to fail.

There are a couple of constraints on the implementation:

1. The PathClassLoader (dalvik.system.PathClassLoader) is not available in standard Java VM's. In order to use this class, it needs to be created and instantiated through the use of java reflection.
2. Changes made should not affect the current implementation of Felix running on a Java VM, and that includes no measurable performance degradation.
3. The same code base is used for both Java and Dalvik VM's. Note, the resulting Felix jar file(s) must be dexified in order to run on a Dalvik VM, but that doesn't involve any code change. Dexifying is a process where Java byte code contain for each class is convert into Dalvik byte code classes and the resulting set of Dalvik byte code classes are packed into a "classes.dex" file.

The native code directory path is used so that the PathClassLoader knows where to look for native code requests. All PathClassLoader instances append /system/lib and /vendor/lib automagically to the path whether a path is supplied or not. All additional paths added to the PathClassLoader are based on the standard OSGI native library specification. A path will be added for all native code clauses that meet the current environment and all applied OSGI filters.

Native code is stored in a bundle at the location specified in a native code clause relative the bundles root. The PathClassLoader is unfortunately not able process paths contain in a bundle (jar file). It requires a directory[1]. So in order to make the native code available, it needs to be copied out of the bundle and stored in a directory. The path to those directories are the paths given to the PathClassLoader.

The Java VM or Dalvik VM Test

The test for whether the framework is running on a Java VM or a Dalvik VM is done by using the following code:

```
boolean requireDex = false;
try {
    requireDex = (Class.forName("dalvik.system.PathClassLoader") != null);
} catch (ClassNotFoundException e){}
```

This test is now done in "org.apache.felix.framework.initializeFrameworkProperties()" method and the result is cached in Map<String, Object>m_config. This map is available to every bundle. It is stored as a String with the value "true" or "false" for the key FelixConstant.***FELIX_REQUIRE_DEX_PROPERTY*** ("felix.require.dex"). This is only called once, so it streamlines the test on whether the Felix framework is running on a Java or Dalvik VM to a simple string comparison. Although this test could have been performed in a static initializer for the BundleWiringImpl class, IMHO it is akin to a property like operating system, processor architecture; so placing the property at that level in the Felix architecture seem most appropriate.

BundleWiringClassImpl Changes

Support for Android is all contained in the BundleWiringImpl class. This class contains an inner class called BundleClassLoader. Bundle class loader extends SecureClassLoader which in turn extends the abstract class ClassLoader. BundleClassLoader overrides the protect loadClass(String, boolean) and findClass(String) methods. The fact that these methods are overridden, breaks the normal class loader delegation model. In the normal class loader delegation model, the ClassLoader's parent is given the first attempt to resolve a class and if the parent is not able to resolve the class, then it uses the ClassLoader's implementation to resolve the class. In the case of the BundleClassLoader, the overridden loadClass method calls the method "findClassOrResourceByDeleation". This method eventual tries to resolve a class using the parent class, but only for class whose packages are defined as belonging to the Felix framework. The parent class will not be used to try and resolve classes contained in the bundle. In the case of the Java VM, when the loadClass method does not resolve the class, it calls the findClass method which tries to define the class by getting the java byte code contained in the bundle, and a set of woven classes it needs to construct it. This unfortunately does not work on a Dalvik VM, because we don't have any reliable why to get the Dalvik byte code from the classes.dex file. The 5.6.10 Felix release used the DexFile class and called it loadClass method as an alternative way to get a class from a "classes.dex file. Unfortunately, the DexFile class is now deprecated as of Android version 26. We unfortunately, can't just replace the DexFile implementation with a PathClassLoader implementation. The reason is the PathClassLoader is a ClassLoader and DexFile is not. When a ClassLoader is called to load or find a class, it must contain everything is needs to

[1] See https://stackoverflow.com/questions/39224397/loading-native-libraries-from-a-dynamically-loaded-apk-using-dexclassloader

resolve a class. That is, it must not only have the class, it must also have its referent classes also available. The implication is that if we called a PathClassLoader whose path contained the bundle, it would only work if class request and all it referent classes where on its path. It would fail to find the class if the class contain a referent class contained in another bundle. However, if we set the PathClassLoader parent to be the BundleClassLoader we now have access the classes contained in the Bundle and any referent classes resolvable by the BundleClassLoader. In the 5.6.10 release if we replaced the DexFile call to a call a PathClassloader whose parent is a BundleClassLoader, it'll still fail due to a cycle check test in the findClassOrResourceByDelegation method. Additional code could be added to bypass the cycle check when the PathClassLoader is called, but that solution makes the code far more complex to understand and support, and adds an inefficiency to resolving classes when dex is required (e.g. we make two passes through the BundleClassLoader before classes contained in a bundle are resolved).

We also can't just replace the instance of the BundleClassLoader with an instance of a PathClassLoader (whose parent is the BundleClassLoader) because the BundleWiringImpl exposes public access to it associated BundleClassLoader instance. And, it doesn't do any good creating a BundleClassLoader with its parent set to a PathClassLoader, because as mentioned above, the normal class delegation model is now changed and the parent is only use to resolve classes exposed by the Felix framework (the boot ClassLoader)

What we do to get around these issues is to add a new inner class called "DexBundleClassLoader" to the BundleWiringImpl class. DexBundleClassLoader extends the BundleClassLoader class and overrides the loadClass(String, boolean) and findClass(String method). DexBundleClassLoader serves as a wrapper for a PathClassLoader whose parent is the BundleClassLoader. The overridden method loadClass(String,boolean) simply delegates the work to the wrapped PathClassLoader. We really don't need to use the findClass(String) method, but we go ahead and redefine it to simply throw a ClassNotFoundException. This new PathClassLoader class does adhere to the default class delegation model. That is, it first tries its parent class (i.e. BundleClassLoader) to resolve a class, and if that fails it tries the PathClassLoader's own method, which has its path set to the path of the bundle associated with the given instance of the BundleWiringImpl. If the class exists in the bundle, the class is resolved. If the class doesn't exist it calls t(hen calls its findClass(String) method. Since we aren't overriding this method in the PathClassLoader class, its default implementation is to throw a ClassNotFoundException which is what we want to happen.

There is one change made in the implementation of the BundleClassLoader. If we are running on a Dalvik VM, we short-circuit the attempt to define a class using byte code extracted from the bundle and its associated woven classes; the attempt is going to fail anyhow. No need wasting time trying something we know is going to fail.

The BundleWiringImpl public method getBundleClassLoader calls the private method getBundleClassLoaderInternal. That method makes the decision on whether to return a BundleClassLoader or a DexBundleClassLoader. Since DexBundleClassLoader is a subclass or BundleClassLoader, the external code that uses them are not affected whether they got a BundleClassLoader or its subclass the DexBundleClassLoader; they both support the entire BundleClassLoader interface. The getBundleClassLoaderInternal gets the BundleClassLoader or DexBundleClassLoader instance by calling the protected method "_getBundleClassLoaderInternal" or "_getDexBundleClassLoaderInternal" respectively. When the DexBundleClassLoader creates its wrapped PathClassLoader, it uses the method "_getBundleClassLoaderInternal" to obtain the parent class used in creating the PathClassLoader instance.

Native Code support

As mentioned earlier, the implementation provides OSGI native library support.

A bundle declaratively specifies its library and associated platform requirements in their bundle manifest using the Bundle-NativeCode header. This header value used for making native code libraries available through the System.loadLibrary() function when it is called from within the bundle.
The Bundle-NativeCode header specifies a number of comma-delimited clauses. For example:

```
Bundle-NativeCode: lib/x86/http.os; lib/x86_64/zlib.os;
    processor=x86; process=x86_64; osname=linux,
  lib/macosx/libhttp.dylib; lib/macosx/libzlib.jnilib;
    osname=macosx; processor=x86; processor=x86_64,
  *
```

A clause specifies one or more libraries. A clause is matched to the current platform using a set of parameters (e.g., processor, osname, osversion, language, or a selection-filter). Parameters of different types are AND'd together; parameters of the same type are OR'd together. The DexBundleClassLoader uses the BundleWiringImpl library list, which it gets from the BundleRevisionImp.getDeclaredNativeLibraries method. The BundleRevisionImpl is responsible for providing the list of the resultant NativeLibrary classes.

When the DexBundleClassLoader is first constructed, each resultant NativeLibrary is copied out of the bundle and stored in a corresponding directory relative the cached bundles root directory. The path to these directories are added the native library path of the PathClassLoader that the DexBundleClassLoader wraps. If two or more clauses contained libraries mapped to the same directory, the resultant native library path given to the constructor of the PathClassLoader will include the repeated directory only once in the path string.

Class Not Found Error

The ClassNotFoundException and UnsupportedOperationException (can't load this type of class files) are two common errors that first time users might see. Here are the most common cause for these type of errors:

- The android application running the Felix Framework must be configured for multidex enabled. Bundles that you add to the framework should not be authored as multidex enabled. The only reason one would configure an individual bundle as multidex enabled would be for bundles that contain more than 65K methods. If a bundle you're authoring runs into this situation, you probably don't understand the purpose of OSGI components. And for that reason, this implementation makes no attempt to handle multidex packaged bundles. Bundles authored with multidex enabled could produce non multidex jar and they would work, but it a possible time bomb, if the next time the bundle is re authored and it then becomes a multidexed jar; making it a tough problem to track down.
- The class name or path is incorrectly specified. Path and class names are case sensitive.
- The class you're attempting to load requires a class that can't be found. This can be a little harder to track down since the error you get is that the class you're attempting to load can't be found, when in reality that class can be found, but one of its internal dependent classes can't. The result however is the same since the ClassLoader doesn't have the resources it needs to return the class object. In this situation, you need to internally inspect the class to see what

classes it needs and see that they exist for the ClassLoader is trying to load the class. Assuming you got the name correctly specified, this is the most likely cause to your problem. Check your bundle import configuration and make sure the osgi framework has a bundle installed that will resolve the needed class and that they properly specified in the bundles MANIFEST.MF file.

- There have been claims that some of these issues can solved by disabling “Instance Run” or by authoring the bundle with optimization disabled. In my experience, these are generally not solutions to this class of error. There may have been a problem in some specific beta version of the program authoring environment (i.e. eclipse, android studio), so there may be some validity to these claims, but in most current program authoring environments this is generally not an applicable solution.