JUDCon

JBoss Users & Developers Conference

Boston:2011

Modular Class Loading

With JBoss Modules

David M. Lloyd
Senior Software Engineer, Red Hat, Inc.

“The Class Path is Dead”

- Mark Reinhold, 2009

What does this mean?

- The limitations inherent in -classpath are well-known
- The days of a large, flat include-the-world class loader are coming to an end... for better or worse
- Modularity is on the Java 8 road map
- This means Java EE will likely follow suit
- Other frameworks which are based on modular concepts - like OSGi - are on the rise

Class-Path Problems

- Apps start simple - one or two JARs
- JAR count grows as requirements increase
- Version conflicts may come into play
- All JARs are always loaded, whether used or not
- No way to restrict visibility between JARs
  - META-INF/services
  - Properties and other resources, especially in the root package

Today's Solutions

- Java EE and OSGi
  - Both require a container environment
  - Both solve much, much more than the problem at hand
  - OSGi modularity semantics are complex
  - Java EE is currently quite limited in this regard
- Class Worlds
  - Used by Maven - last release in 2004
- Simple-DM ("Simple Dynamic Modules")
  - Tied to Maven repository

What is JBoss Modules?

- A standalone implementation of *modular* (non-hierarchical) class loading and program execution
  - Implements a thread-safe, fast, and highly concurrent delegating class loader model
- Not a container like OSGi, more like a thin wrapper to bootstrap the modular environment
- Executable JAR; run via “java -jar” with no class path needed

What *else* is JBoss Modules?

- Not much!
- Small, very fast, easy to use
- Solves exactly one problem
- Used as the basis for:
  - JBoss OSGi
  - JBoss AS 7 Java EE implementation
- All JBoss AS 7 internal JARs are modules
  - Super-fast bootstrap

What is a Module?

- A *module* is a named set of classes (typically just a single JAR) coupled with information about what other modules it uses, as well as what classes and resources it exports, usually in the form of an XML descriptor file (depending on the *module loader*)

```
<module name="com.bananodyne.mymain">
   <main-class name="com.bananodyne.Main"/>
   <resources>
      <resource-root path="mymain.jar"/>
   </resources>
   <dependencies>
      <module name="org.slf4j"/>
      <module name="org.jboss.vfs"/>
      <module name="org.junit"/>
   </dependencies>
</module>
```

Module Names

- Module names are dot-separated, a bit like package names or Maven group IDs. Examples:

```
org.jboss.shrinkwrap.api
org.apache.xalan
org.dom4j
```

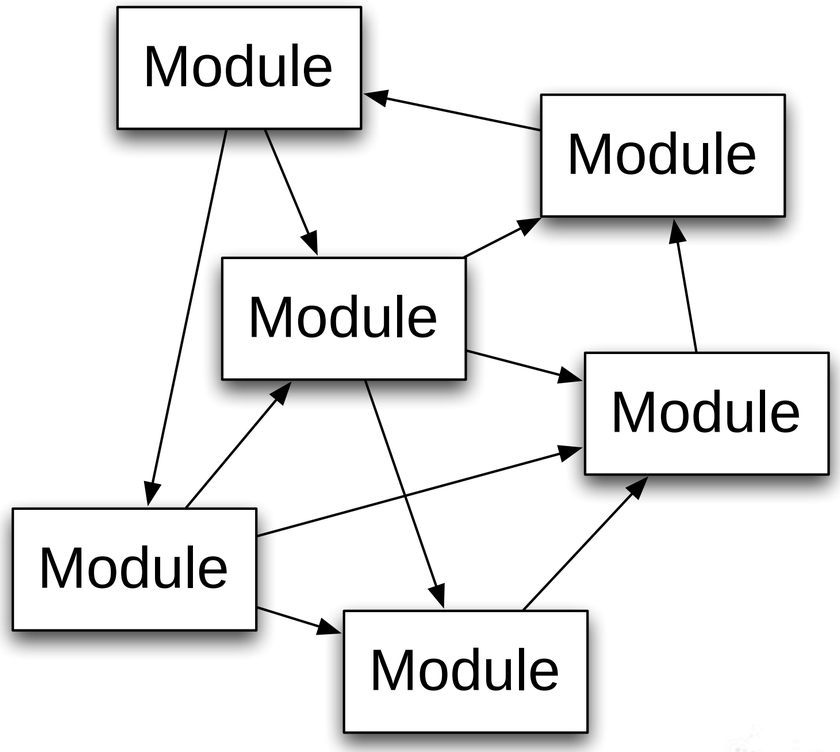
- However there is no actual relationship between a module's name and the names of the packages which comprise it
- Nor is there a relationship to Maven artifact identifiers

Versions

- JBoss Modules does not support version resolution, period
- Too complex, non-deterministic, “magical”
- However there is a version “slot”
- Allows two modules with the “same” name to coexist
  - E.g. two different versions of a module which are completely incompatible
- The version slot defaults to “main”
- Future versions **may** introduce simple runtime checking
  - I.e. check for log4j >= 1.2.13, fail otherwise
- KISS!

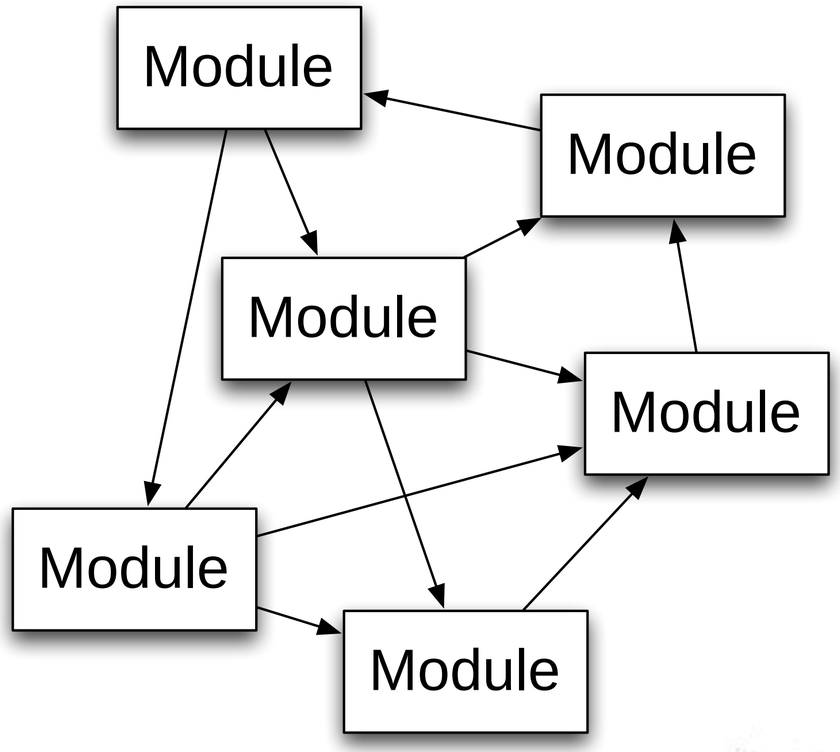
Module Delegation

- Modules delegate to one another as peers (no parents)
- A graph, not a tree
- Much like how multi-module projects are configured in an IDE
- JDK classes are a Module too
- All modules “see” `java.*`



Module Delegation

- No more “big ball of mud”
- Every module is isolated from every other module
- A module imports only the modules that it directly uses (and does not “see” classes or resources that it does not use)
- In particular, modules do not normally “see” their **transitive** dependencies



What is a Module Loader?

- A *module loader* is a class that knows how to locate, load and define modules
- Similar to how a class loader loads and defines classes
- JBoss Modules ships with a basic filesystem-based module loader implementation
- This is the standard module loader created when jboss-modules.jar is executed from the command line

LocalModuleLoader

- Uses a simple directory structure to locate modules on demand
- Directory name is decomposed module name plus version slot
- Module definition contained in “module.xml” file
- JARs are relative to the module root directory
- Use existing JARs as-is

```
modules/
├── com/
│   └── sun/
│       └── httpserver/
│           └── main/
│               ├── httpserver-20100812-JBoss.jar
│               └── module.xml
├── jline/
│   └── main/
│       ├── jline-0.9.94.jar
│       └── module.xml
└── org/
    ├── apache/
    │   ├── commons/
    │   │   ├── codec/
    │   │   │   └── main/
    │   │   │       ├── commons-codec-1.4.jar
    │   │   │       └── module.xml
    │   │   ├── collections/
    │   │   │   └── main/
    │   │   │       ├── commons-collections-3.1.jar
    │   │   │       └── module.xml
    :   :   :
```

Module Descriptor

- Defines resource roots, dependencies, filtering, main class
- Simple XML format; well-documented XSD

```
<module xmlns="urn:jboss:module:1.0" name="org.jboss.xnio">
   <resources>
      <resource-root path="xnio-api-3.0.0.Beta1"/>
   </resources>

   <dependencies>
      <module name="org.jboss.xnio.nio" services="import"/>
      <module name="org.jboss.logging"/>
   </dependencies>
</module>
```

Execution

- Execute a module from the repository which has a main class specified in its module.xml descriptor

```
java -jar jboss-modules.jar -mp <path> module.name
```

- Alternately, execute an arbitrary JAR file which may reference (or in the future, contain) modules

```
java -jar jboss-modules.jar -mp <path> -jar myapp.jar
```

- Uses a simple MANIFEST.MF property for dependencies

```
Dependencies: org.jboss.logging, org.xnio.api
```

- Same syntax used for modular dependencies in JBoss AS 7 deployments

JDK Obstacles

- Custom LogManagers (java.util.logging) normally need to be on the application class path
  - Use the “-logmodule” option to load a LogManager from a module
- Changing the default JAXP provider normally requires endorsed libs
  - Use the “-jaxpmodule” option to specify the new default provider

Third-Party Problems

- Just about anything which uses the TCCL!
  - Many frameworks assume that the TCCL has **everything** on it, from the JDK to application classes to implementation classes to other APIs
  - Usually more correct to load classes from their own class loader
    - Modules for APIs can be specified to import their implementations
  - Or to accept a class loader as an argument to which the caller can provide their own

Dealing with Problems

- Most common problem is missing dependencies
- Java does not gracefully handle linkage errors
  - Obscure stack traces and exception messages
- JBoss Modules enhances exceptions whenever possible with the name of the module which triggered the problem and a description of what happened
- Linkage errors are logged
- Still more research to be done into diagnostics
  - Probably need some fairly esoteric hacks to get additional information

Comparing to Alternatives

- More basic and low-level than OSGi
  - This is good and bad
  - Simpler to use, smaller learning curve, faster... but..
  - Lacking advanced resolution features, no service layer
  - No resolver = fast **but** certain integrity checks are not possible
- Much more powerful than the simple extension mechanisms provided by Java SE and EE
- For static applications, makes much more sense than 100 JARs on your class path

Modules and Maven

- Tricky problem - Maven artifacts are quite similar to modules
- Build dependency management is a different problem from run-time dependency management
  - Builds strive for repeatability
    - Always use the exact same plugin and dependency versions so you get the same result every time
- Run time environment evolves over time
  - One version of module A might be compatible with multiple versions of module B... even future versions
  - Fixed version deps are not well-suited to modules

Modules and Maven

- Maven is presently built around the flat classpath philosophy
- Thus without major changes (Maven 4? 5?), Maven may not survive the “modular revolution” of Java 8
- Future of build tools for modular environments is still unclear
  - Maybe Maven will adapt
  - Maybe Gradle will mature
  - Or a new player

Future Directions

- Build integration
  - Easy way to build a complete modular environment from Maven
- Complete module environments as distributions
  - Think /usr/lib for Java applications; install with package management tools similar to yum
  - Stop worrying about dependencies, just write code
  - A bit similar to Ruby Gems, Perl CPAN, Python Eggs, etc.
  - Friendly to OS distributions
  - Possible future direction of JBoss EAP

Future Directions

- Tooling
  - Make it easier to figure out needed and missing dependencies (perhaps a modified version of Tattletale)
- IDE integration
  - Make it easier to run (and debug) modular code from within an IDE
- Visualization tools

Take it out for a spin!

- Download from Maven: org.jboss.modules:jboss-modules
- Get the sources at http://github.com/jbossas/jboss-modules
- File issues at https://issues.jboss.org/browse/MODULES
- Discuss your ideas on irc.freenode.net: #jboss-msc or #jboss-as7
- Try out the JBoss AS 7 betas

Q & A

JUDCon

JBoss Users & Developers Conference

Boston:2011