Project Jigsaw: Modular services

Jigsaw team

12 June 2012

Terms

- **Service interface**: An interface or class
- **Service interface module**: A module which exports [a package containing] a service interface
- **Service provider class**: A non-abstract class which implements/extends a service interface
- **Service provider module**: A module which binds a service interface to a service provider class in the module via "`provides service ... with ...`"
- **Service consumer module**: A module which denotes it's use of a service interface via "`requires [optional] service ...`"
- **Service (interface) instance**: An object whose class is a service provider class. The vast majority of references to such an object are made through it's implemented service interface

Modular Service Declarations

- Services are declared and used in the Java language, in `module-info.java`
  - No META-INF/services files
- A service interface module

```
module api@1.0 {
  exports acme;
}
```

- A service consumer module

```
module consumer@1.0 {
  requires api;
  requires service acme.Foo;
}
```

- A service provider module

```
module provider@1.0 {
  requires api;
  provides service acme.Foo with ajax.FooImpl;
}
```

Modular Service Declarations

- A service consumer module C may be the service interface module for the service interface consumed by C
- A service provider module P may be the service interface module for the service interface implemented by P's service provider class
- A module may be a service consumer module and a service provider module, and provide services to itself

Optional Services

- “`requires optional service acme;`”
- Indicates that a service consumer module requires zero or more service provider classes to implement the service interface (rather than one or more)
- Note that the service interface must always be visible to the service consumer module; there is no optionality there.
- Optional service dependencies are verified when root module configurations are generated (which may be at module install time)

Service instance creation

- Service instances are created lazy using `java.util.ServiceLoader` and the `load` methods (see example later on)
- Service instance creation implies service interface visibility
  - If you cannot see the interface you cannot create the instances
- Service instance creation is scoped from the service consumer module
  - Not scoped from the configuration
- The `permits` declaration affects service instance creation
  - Not all service provider modules may be visible to a service consumer module

Service instance creation

- The module **C**lass **L**oader (**CL**) of the service consumer module is utilized to obtain the set of service provider class for a given service interface
  - Specifically the context associated with the CL will reference the service provider class names and the service provider modules those classes are associated with
  - From that information a service provider class can be loaded, using the CL of the corresponding service provider module, and then instantiated
- The service interface module may not be the same as the service consumer module
  - Not a commonly observed pattern (in the modularized JDK at least)
  - The CL of the service interface module cannot be used to load the set of service instance
- Accessibility of the service interface of a service instance is the same as the accessibility of an exported type

Service instance creation

- Once a service instance is created, the accessibility of it's service interface is not restricted to the scope of the service consumer module
- A service instance S may be used through its service interface I by code in a module M as long as M can see the type I.
- Even if M is a service consumer module and the service provider module, providing S, is not visible to M
  - i.e. S is not a member of the set of service instance created by M

Service creation and query example

```
Class<Foo> serviceInterface = ...;
ClassLoader serviceConsumer = ...;

// Lazy
// No service instances are instantiated
Iterable<Foo> services = ServiceLoader.load(
    serviceInterface,
    serviceConsumer);

// Instantiation occurs on each call to Iterator.next()
for (Foo service : services) {
  if (service.isCapableOf(...)) {
    return service;
  }
}

return new DefaultFoo();
```

Modular services goals

- Parity with `j.u.ServiceLoader`
- Update JDK
  - Work in classpath and modular mode for services
  - Transform certain functionality into modular services while maintaining backwards compatibility in classpath mode
- Improve services

ServiceLoader implementation

- `j.u.ServiceLoader` modified to switch between classpath and module mode
  - Based on the type of CL
- Static `load` methods modified, with a clever hack, to select a CL that **might often** correspond to the CL of the service consumer
  - Tactical and **temporary** solution to get something working without modifications to callers in the JDK

A note on **T**hread **C**ontext **C**lass **L**oader (**TCCL**) in module mode

- TCCL will by default be set to the CL of the entry module
  - TCCL == System CL == CL of entry module
- The CL of the entry module will invariably not be the correct CL for the creation of service instances
- A TCCL will not be the correct CL for the creation of service instances for multiple service consumer modules
- Avoid where possible the following pattern:

```
final ClassLoader _tccl =
    Thread.currentThread().getContextClassLoader();
try {
  ClassLoader tccl = ...
  Thread.currentThread().setContextClassLoader(tccl);
  ...
} finally {
  Thread.currentThread().setContextClassLoader(_tccl);
}
```

ServiceLoader methods: classpath mode vs module mode

`ServiceLoader` method	Classpath mode	Current module mode
`load(Class )`	TCCL	The caller CL
`load(Class ClassLoader )`	If the CL parameter == null then the system CL, otherwise the CL parameter	If the CL parameter == null or == System CL then the caller CL, otherwise the CL parameter
`LoadInstalled( Class )`	Extension CL if present, otherwise System CL if not null, otherwise Bootstrap CL	The caller CL

Existing usage in JDK (1)

- `com.sun.net.httpserver.spi.HttpServerProvider`

```
104    private static boolean loadProviderAsService() {
105        Iterator<HttpServerProvider> i =
106            ServiceLoader.load(HttpServerProvider.class,
107                               ClassLoader.getSystemClassLoader())
108                .iterator();
```

- System CL replaced with caller CL
- `HttpServerProvider` is the caller and is exported from the `jdk.httpserver` module, that requires the service
  - The appropriate CL of the service consumer module is currently selected

Existing usage in JDK (2)

- `java.sql.DriverManager`

```
502             ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);
503              Iterator driversIterator = loadedDrivers.iterator();
```

- `DriverManager` is the caller and is in module `jdk.jdbc`, that requires the service
  - The appropriate CL of the service consumer module is currently selected

Existing usage in JDK (3)

- `sun.awt.im.InputMethodManager`

```
435                 for (InputMethodDescriptor descriptor :
436                     ServiceLoader.loadInstalled(InputMethodDescriptor.class)) {
```

- `InputMethodManager` is the caller and is in module `jdk.desktop`, that requires the service
  - The appropriate CL of the service consumer module is currently selected

Observations

- Using the caller CL is fragile
  - Too contextual to the “identity” of the caller, which could change
  - In addition it is known to be slow compared to explicit declaration of CL
- ServiceLoader.load* methods have to be retrofitted to select the “best” CL in module mode
  - No ideal fit if the caller CL is used or not

Observations

- The "`permits`" clause is complicating matters
  - Service instance accessibility is not scoped to the service consumer module
  - Service instances may be accessed by **any** module permitted to access the service interface
- How can a service provider module possibly know what service consumer modules should be permitted or not?
  - Contrary to the notion of providing a service where the provider is **decoupled** from the consumer

An alternative solution

- The "`permits`" clause does not apply to service provider classes for the purpose of service creation
- Service instance creation is scoped from the configuration
  - There is one configuration per application
- It does not matter what CL is used as long as it is a module CL
  - The CL of the root module can be used to create the same set of service instance as the CL of a service interface/consumer/provider module
  - The CL provides a level of indirection to the configuration

An alternative solution

- No tweaks required to `ServiceLoader.load*` methods
  - Different `Iterator<S>` implementations if CL is module CL or non-module CL
  - Minimal changes to JDK service loading code
  - Works correctly with non-module CL
- From the perspective of the developer the solution is consistent and simple