Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en Java EE, Linux y optimización de aplicaciones empresariales.
Ingeniero en Informática y Certified ScrumMaster.
Seguir @alejandropgarci
Si te gusta lo que ves, puedes contratarle para darte ayuda con soporte experto, impartir cursos presenciales en tu empresa o para que realicemos tus proyectos como factoría (Madrid).
Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.

Primeros pasos con los módulos de Java 9 y Maven – proyecto Jigsaw, JSR 376

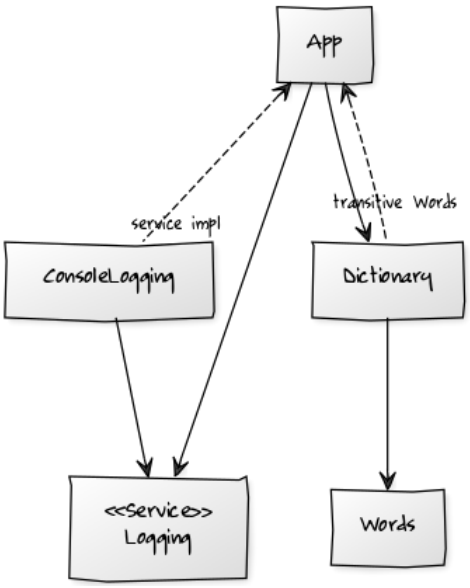
mayo 4, 2017 Alejandro Pérez García Sin comentarios Tutoriales 802 visitas

En este tutorial vamos a dar nuestros primeros pasos con los módulos de Java 9 y veremos cómo podemos combinarlos con Maven para conseguir lo mejor de los dos mundos: con los módulos de Java 9 gestionaremos la visibilidad entre las clases hasta un grano muy fino, mejorando la así la encapsulación; y con Maven especificaremos cómo queremos componer los artefactos de nuestra aplicación.

Índice de contenidos

Los más visitados de la semana

Monitorizando equipos y servicios con Nagios + NagiosQl + PN...
VirtualBox. Configuración de la conexión de red.
Instalación y uso del plugin de comentarios de Facebook en n...
Parte 1. Aprendiendo HTML para crear una página web
SoapUI: jugando con web services



1. Introducción

El proyecto Jigsaw (JSR 376 – Java Platform Module System), nace con los siguientes objetivos en mente:

- Hacer que la plataforma Java SE, y la JDK, se puedan descomponer en trozos más pequeños para ser usadas en pequeños dispositivos.
- Mejorar la seguridad y mantenimiento en general de las implementaciones de la plataforma Java SE, y en particular de la JDK.
- Posibilitar la mejora de rendimiento de las aplicaciones.
- Facilitar a los desarrolladores la construcción y mantenimiento de grandes librerías y aplicaciones, tanto para la plataforma Java SE y EE.

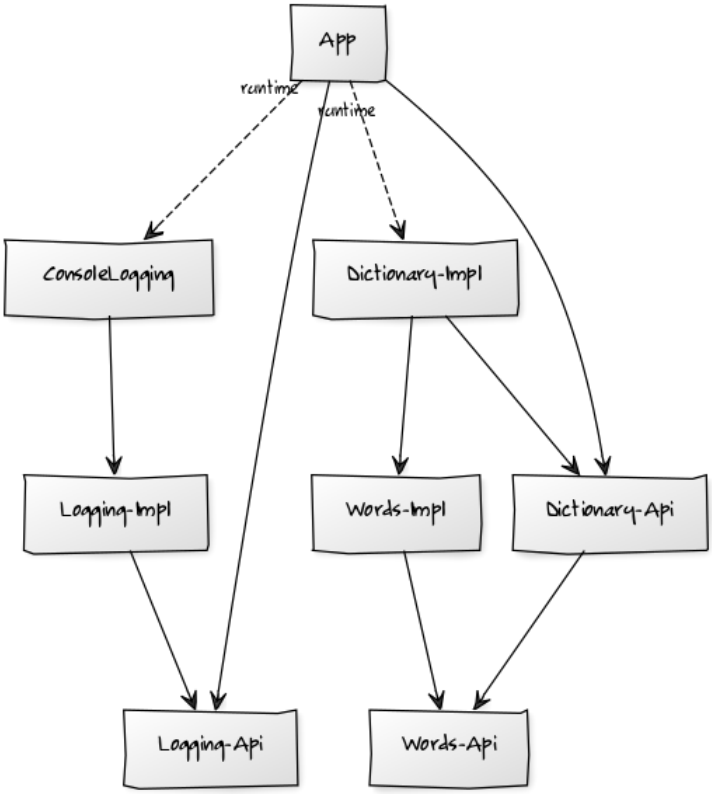
Todo esto quiere decir básicamente que vamos a poder romper nuestra aplicación en trozos, que llamaremos *módulos*, donde cada uno de estos módulos tiene perfectamente definidas sus dependencias con otros módulos. Y estas dependencias van a tener un impacto directo en la visibilidad de nuestras clases.

Alguien podría pensar que esto ya lo teníamos resuelto con los archivos JAR y la gestión de dependencias de Maven (o Gradle), pero nada más lejos. Efectivamente un JAR es una agrupación de clases y con Maven podemos definir las dependencias tanto a nivel de compilación como de ejecución que estos JAR tienen entre ellos, pero tiene un gran problema y es que cualquier clase `public` es visible en cualquier parte del sistema, y esto va en contra de los principios básicos del diseño de aplicaciones rompiendo con el bajo acoplamiento y la alta cohesión.

Esto sucede porque, por muchos JAR que tengamos, básicamente lo que hace Java es cargar todas las clases de todos los JARs de nuestra aplicación en el mismo *class loader*, con lo que todas las clases `public` son visibles por el resto de clases independientemente del JAR o del paquete en el que se encuentren definidas.

A nivel de Maven no mejora demasiado ya que podemos expresar dependencias entre JARs, por ejemplo A → B, pero una vez se establece esta dependencia, desde A podremos acceder a todas las clases `public` que se encuentren en B. Por esto, para mejorar el encapsulamiento es habitual encontrarse nombres de paquetes del estilo `xxxx.internal.yyyy` o `xxxx.impl.yyyy` donde ese `internal` y ese `impl` nos están indicando: *¡cuidado! ¡más allá hay dragones!* Pero es simplemente eso, una advertencia, porque nada nos impide acceder a las clases que se encuentran tras esas "barreras".

Incluso es bastante común encontrar librerías partidas en varios JARs (API e implementación) para conseguir que, por lo menos a nivel de compilación, no accedamos a las clases que no debemos (protección que luego no existirá en tiempo de ejecución). Por ejemplo para conseguir con Maven un esquema de dependencias similar



Es decir, prácticamente cada JAR lo tenemos que partir en dos y especificar la dependencia en compilación con el API y en ejecución con la implementación. Además recordemos otra vez que esto sólo nos va a afectar mientras compilamos, pero que en runtime podremos acceder a cualquier clase, por ejemplo usando reflection, por lo que todo este trasiego de JARs tampoco nos garantiza realmente la encapsulación de nuestras clases.

Aquí tenéis un repositorio de GitHub con un ejemplo completo de todo lo que se habla en este tutorial.

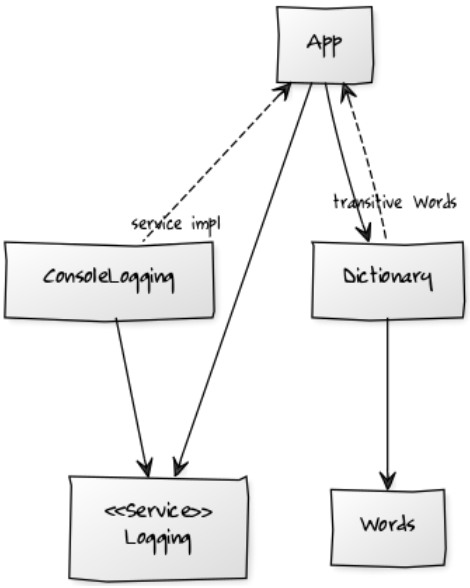
2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15” (2.5 GHz Intel i7, 16GB 1600 Mhz DDR3, 500GB Flash Storage).
- AMD Radeon R9 M370X
- Sistema Operativo: macOS Sierra 10.12.4
- Oracle Java 9-ea167
- Maven 3.5.0

3. Definición de un módulo

Volvemos a poner aquí el gráfico que pusimos al principio del tutorial y que representa lo que queremos conseguir en este ejemplo.



Vemos que tenemos un módulo `App` que depende de un servicio de `Logging` y de un módulo de `Dictionary`. Las dependencias a estos dos módulos son las únicas directas que tiene `App`, sin embargo de forma *transitive* también será capaz de acceder al módulo `Words`. Y en runtime accederá al módulo `ConsoleLogging` que es la implementación del servicio. Lo bueno de esto es que realmente la aplicación no sabe quien es el proveedor del servicio, es decir estamos consiguiendo inversión de control (que no inyección de dependencias).

Para definir un módulo debemos añadir un fichero `module-inform.java` en el directorio raíz de nuestro código fuente (en el directorio donde veríamos nuestro paquete raíz `com.`). Por ejemplo, en un proyecto Maven sería en el directorio `src/main/java`.

- Por convención el directorio raíz donde se encuentra el código del módulo debería tener el nombre del propio módulo, es decir `src/main/com.autentia.logging/…<estructura de paquetes>….` Por ahora nos vamos a saltar esta convención para mantener el `src/main/java/…<estructura de paquetes>…` y así simplificar la configuración de Maven.

Para nuestro caso más sencillo que es el módulo de `Logging`, este fichero tendría el siguiente aspecto:

```
1 module com.autentia.logging {
2
3     exports com.autentia.logging;
4
5 }
```

Vemos cómo después de la palabra reservada `module` viene el *ID* del módulo, en nuestro caso es `como.autentia.logging`. Este ID tiene forma de nombre de paquete, pero es sólo una convención, ya que simplemente se trata de una cadena, por lo que podríamos poner lo que quisiéramos. Cuando desde otro módulo queramos hacer referencia a este, siempre lo haremos usando este ID.

Después entre llaves vemos la definición del módulos. En este caso sólo vemos la palabra reservada `exports` y a continuación el nombre del paquete que queremos exponer hacia afuera. Es decir, de este módulo sólo serán visibles desde otros módulos las clases públicas que se encuentren en el paquete `como.autentia.logging`. El resto de clases públicas que estén en el módulo pero en otros paquetes, quedarán ocultas tanto en tiempo de compilación como en tiempo de ejecución.

4. Dando más visibilidad a los modulos amigos

Hay ocasiones donde podemos tener varios paquetes que están relacionados. Entre estos paquetes podemos querer exportar más clases que las que exponemos al resto del mundo. Así en nuestro ejemplo vamos a suponer que el módulo `Words` quiere compartir con el módulo `Dictionary` ciertas clases que no serán visibles para el resto del mundo:

```
1 module com.autentia.words {
```

```
7 }
```

Vemos cómo tenemos un segundo `exports` con una cláusula `to`. Esto indica que el paquete que hay a la izquierda del `to` sólo será visible por el módulo cuyo ID hemos puesto a la derecha del `to` (importante recalcar que a la izquierda hemos puesto un nombre de paquete mientras que a la derecha hemos puesto un ID de un módulo). A la derecha del `to` podemos poner una lista de IDs de módulos, separados por comas ",".

5. Dependencia transitiva de un módulo

Por defecto no hay dependencias transitivas, es decir, si nuestra `App` quiere usar el módulo `Dictionary` y este devuelve en su API pública clases del módulo `Words`, el módulo `App` estaría obligado a declarar también la dependencia con el módulo `Words`. Esto no parece ni práctico ni cómodo, ya que esta doble dependencia (el módulo `Dictionary` y el módulo `Words`) la tendríamos que declarar en todos los otros módulos que quieran usar el módulo `Dictionary`.

Para evitar esta redundancia de tener que declarar siempre la dependencia al módulo `Words` se puede evitar usando la palabra reservada `transitive`. Por ejemplo:

```
Java
2
3     exports com.autentia.dictionary;
4
5     requires transitive com.autentia.words;
6
7 }
```

Aquí, en la línea 5 estamos indicando que todo aquel que requiera como dependencia el módulo `Dictionary`, automáticamente de forma transitiva también tendrá disponibles las clases del módulo `Words`.

6. Definición de un servicio

En nuestro ejemplo ya hemos visto cómo tenemos un módulo de `Logging` donde vamos a definir un servicio, que es simplemente una `Interface` de Java. Este servicio (interfaz) será utilizado en otras partes del sistema, pero la gracia está en que no se sepa quién implementa dicho servicio (interfaz). De esta manera conseguimos que nuestro sistema sea más sencillo de mantener ya que podremos cambiar la implementación de dicho servicio sin necesidad de tocar ni una sola línea de código del resto del sistema.

Para ello en el módulo `Logging` ya vimos que simplemente se hacía export de un paquete. En este paquete está la interfaz que define el servicio (esta interfaz no tiene nada en especial, es la implementación de una interfaz de Java de toda la vida).

Ahora vamos a proporcionar una implementación donde los mensajes simplemente se escriben en la consola. Esto lo haremos en el módulo `ConsoleLogging`:

```
1 module com.autentia.logging.console {
2
3     requires com.autentia.logging;
4
5     provides com.autentia.logging.Logger with com.autentia.logging.console.ConsoleLogger;
6
7 }
```

Aquí en la línea 5 podemos ver cómo con la palabra reservada `provides` indicamos el nombre de la interfaz (que está definia en el módulo `Logging`), y con la palabra reservada `with` indicamos la clase, dentro del módulo `ConsoleLogging`, que va a implementar dicha interfaz.

7. Localización de un servicio

En el punto 3 hemos visto cómo definimos el servicio, y en el punto 6 hemos visto cómo indicar qué clase implementa dicho servicio. Ahora sólo nos falta localizar dicho servicio para poder usarlo. Así en el módulo `App` que representa nuestra aplicación declararemos:

```
7
8 }
```

Aquí destacamos cómo en la línea 5 usamos la palabra reservada `uses` para indicar que en este módulo vamos a usar el servicio. Nótese especialmente que en este módulo `App` no tenemos ningún `requires` al módulo `ConsoleLogging` sino que simplemente lo tenemos al módulo `Logging` que es donde se define la interfaz. De esta forma el código de nuestra aplicación queda totalmente desacoplado de la implementación del servicio, ya que no hay ninguna dependencia, ni en compilación ni en ejecución, entre ambos módulos.

El código de nuestra aplicación quedará así:

```
1  public class App {
2
3      private final Logger log = ServiceLoader.load(Logger.class).findFirst().get();
4      private final Dictionary dictionary = new Dictionary();
5
6      public static void main(String... args) {
7          new App().execute();
8      }
9
10     private void execute() {
11         final Word word1 = dictionary.getWord();
12         final Word word2 = dictionary.getWord();
13         final Word word3 = dictionary.getWord();
14
15         log.error(word1.toString());
16         log.info(word2.toString());
17         log.debug(word3.toString());
18     }
19
20 }
```

Donde vemos cómo en la línea 3 estamos localizando la implementación del servicio sin saber cuál es.

En el ejemplo se ve cómo localizamos la primera implementación del servicio con `findFirst`, pero aquí podríamos tener toda la lógica que quisiéramos, incluso no sería complicado pasar de un patrón de *Service Locator* a un contenedor de *Dependency injection*.

8. ¿Y qué pasa con el código antiguo? ¿cómo lo migramos al sistema de módulos?

Cuando intentemos cargar una clase que no se encuentra en ningún módulo, esta se va a buscar en el class path normal de Java (el class path de toda la vida). Si se encuentra en el class path, automáticamente esta clase se añade a un módulo especial llamado “módulo sin nombre” (*unnamed module*). De esta forma este unnamed module contendrá todas las clases que no estén definidas en ningún módulo explícitamente.

El módulo sin nombre puede acceder a todas las clases públicas que estén exportadas en el resto de módulos con nombre (módulos explícitos). Por el contrario los módulos con nombre no pueden acceder a ninguna clase que se encuentre en el módulo sin nombre, de hecho ni siquiera se puede especificar una dependencia hacia el unnamed module. Esta restricción está puesta a propósito para forzar a los desarrolladores a realizar una correcta configuración de los módulos.

Todo esto nos permite hacer una migración gradual de nuestras aplicaciones ya que pueden convivir perfectamente los nuevos módulos con JAR que todavía no tengan módulos definidos.

Teniendo en cuenta las restricciones antes mencionadas, esta migración la tendremos que hacer de *abajo arriba*. Es decir en un grafo de dependencias empezaríamos a migrar las hojas (JARs que no dependen de otros, en nuestro ejemplo serían `Words`, `ConsoleLogging`, …), e iríamos subiendo en el grafo de dependencias (en nuestro ejemplo terminaríamos por el módulo `App`).

El problema que tiene esta migración de abajo arriba es que nosotros podemos tocar nuestro código, pero siempre vamos a depender de librarías de terceros que, si no están migradas al nuevo sistema de módulos, no podrán ser accedidas desde nuestros módulos. En estos casos lo que podemos hacer es tratar estos JAR como un “módulo automático” (*automatic module*), simplemente colocando el JAR en el *module path* en lugar del *class path*. De esta forma se genera un módulo de forma implícita, donde su nombre vendrá determinado por el nombre del archivo. Y así podremos usarlo como si se tratara de cualquier otro módulo con nombre.

- Imágenes generadas con yUML
- GitHub – Java 9 Modules Example with Maven and JUnit
- Consol Labs – Getting Started with Java 9 Modules
- Open JDK – Project Jigsaw
- Open JDK – Project Jigsaw: Module System Quick-Start Guide
- Open JDK – The State of the Module System
- SlideShare – Java SE 9 modules (JPMS) – an introduction

10. Conclusiones

Los módulos de Java 9 se presentan como una potente herramienta para la encapsulación. Hemos visto cómo podemos declarar las dependencias, hacer estas transitivas, definir implementar y usar servicios, …

Sin embargo ¿qué pasa con Maven/Gradle? En el ejemplo completo de código podéis ver como ambos pueden convivir a día de hoy, existiendo una pequeña duplicidad de conceptos, ya que nos vemos obligados a definir las dependencias en dos puntos: los módulos de Java 9 y los módulos de Maven para su compilación. Veremos si en versiones posteriores de Maven se lee esta información directamente de los módulos de Java 9 ¿o acaso es hora de un nuevo sistema de empaquetamiento?

Y si podemos definir e inyectar servicios ¿qué pasa con Spring? ¿Acaso es su fin? ¿Habrá un enganche entre ambos? Desde luego habrá movimiento porque con las nuevas restricciones que impone el *module path* Spring no va a tener tan fácil hacer sus "automagias".

En general hay muchas incertidumbres (por ejemplo, como hemos comentado antes, las migraciones de abajo arriba no van a ser nada sencillas porque implica que todo el mundo tiene que hacerla), y de hecho hay unos cuantos detractores, por ejemplo aquí tenéis un buen artículo Concerns Regarding Jigsaw(JSR-376, Java Platform Module System). Ojo que es largo, pero merece la pena para ver la de esquinas que todavía quedan por pulir.

Con todo esto lo que nos queda es estudiar, practicar y observar hacia dónde se mueven las cosas, a ver si finalmente se imponen los módulos de Java 9 o por el contrario se sigue optando por soluciones de terceros como OSGi, que por otro lado son mucho más potentes.

11. Sobre el autor

Alejandro Pérez García (@alejandropgarci)
Ingeniero en Informática (especialidad de Ingeniería del Software) y Certified ScrumMaster

Socio fundador de Autentia Real Business Solutions S.L. – "Soporte a Desarrollo"

Socio fundador de ThE Audience Megaphone System, S.L. – TEAMS – "Todo el potencial de tus grupos de influencia a tu alcance"

Name (required)

E-Mail (required)

Website

Enviar comentario

Lo último

- Serializa tus datos con Protocol Buffers (1/2)
- Contratos inteligentes en Ethereum
- Servidor mock con hercule y drakov
- Maquetación con CSS Grid
- Carga resiliente de datos en elasticsearch usando alias

Datos de contacto

Edificio BestPoint Avd. de Castilla, 1, Planta 2, Oficina 21B (San Fernando de Henares)
Phone: 916 75 33 06
E-Mail: adictos@adictosaltrabajo.com
Web: https://www.autentia.com

Powered by

autentia
Soporte a desarrollo informático