Entendendo os módulos do Java 9

O que eles são e como os utilizar

Por Paul Deitel

Neste artigo, apresento o JPMS (Java 9 Platform Module System), a mais importante nova tecnologia de engenharia de software em Java desde a sua criação. A modularidade - o resultado do Project Jigsaw - ajuda os desenvolvedores em todos os níveis a serem mais produtivos à medida que criam, mantêm e desenvolvem sistemas de software, especialmente sistemas grandes.

O que é um módulo?

A modularidade adiciona um nível mais alto de agregação acima dos pacotes. O novo elemento-chave da linguagem é o *módulo* - um grupo reutilizável e único de pacotes relacionados, bem como recursos (como imagens e arquivos XML) e um *descritor de módulo que* especifica

- o *nome* do módulo
- as *dependências* do módulo (isto é, outros módulos em que este módulo depende)
- os pacotes explicitamente disponibilizados para outros módulos (todos os outros pacotes no módulo estão *implicitamente indisponíveis* para outros módulos)
- os *serviços que oferece*
- os *serviços que consome*
- para que outros módulos permite *reflexão*

História

A plataforma Java SE existe desde 1995. Atualmente, existem aproximadamente 10 milhões de desenvolvedores usando tudo isso, desde pequenos aplicativos para dispositivos com recursos limitados, como os da Internet das Coisas (IoT) e de outros dispositivos incorporados, até grandes sistemas críticos de negócios e de missão crítica. Existem enormes quantidades de código legado por aí, mas até agora, a plataforma Java tem sido basicamente uma solução monolítica de tamanho único para todos. Ao longo dos anos, tem havido vários esforços voltados para a modularização do Java, mas nenhum é amplamente utilizado - e nenhum deles poderia ser usado para modularizar a plataforma Java.

A modularização da plataforma Java SE tem sido um desafio para implementar, e o esforço levou muitos anos. JSR 277: O Java Module System foi originalmente proposto em 2005 para o Java 7. Este JSR foi posteriormente substituído pelo JSR 376: Java Platform Module System e direcionado para o Java 8. A plataforma Java SE agora é modularizada no Java 9, mas somente após o Java 9 foi adiada até setembro de 2017.

Objetivos

De acordo com a JSR 376, os principais objetivos da modularização da plataforma Java SE são:

- Configuração confiável - A modularidade fornece mecanismos para declarar explicitamente as dependências entre os módulos de uma maneira que seja reconhecida tanto no tempo de compilação quanto no tempo de execução. O sistema pode percorrer essas dependências para determinar o subconjunto de todos os módulos necessários para dar suporte ao seu aplicativo.
- Encapsulamento forte - Os pacotes em um módulo só podem ser acessados por outros módulos se o módulo exportá-los explicitamente. Mesmo assim, outro módulo não pode usar esses pacotes a menos que explicitamente afirme que requer os recursos do outro módulo. Isso melhora a segurança da plataforma porque menos classes são acessíveis a invasores em potencial. Você pode descobrir que, considerando a modularidade, ajuda a criar designs mais limpos e mais lógicos.
- Plataforma Java escalonável - Anteriormente, a plataforma Java era um monólito que consistia em um grande número de pacotes, tornando difícil desenvolver, manter e evoluir. Não poderia ser facilmente subdividido. A plataforma agora é modularizada em 95 módulos (esse número pode mudar conforme o Java evolui). Você pode criar execuções personalizadas que consistem apenas em módulos necessários para seus aplicativos ou para os dispositivos que você está segmentando. Por exemplo, se um dispositivo não suportar GUIs, você poderá criar um tempo de execução que não inclua os módulos da GUI, reduzindo significativamente o tamanho do tempo de execução.
- Maior integridade da plataforma - Antes do Java 9, era possível usar muitas classes na plataforma que não eram destinadas às classes de um aplicativo. Com um forte encapsulamento, essas APIs internas são realmente encapsuladas e ocultas dos aplicativos que usam a plataforma. Isso pode tornar problemático migrar o código legado para o Java 9 modularizado se o seu código depender de APIs internas.

- Desempenho aprimorado - A JVM usa várias técnicas de otimização para melhorar o desempenho do aplicativo. O JSR 376 indica que essas técnicas são mais eficazes quando se sabe de antemão que os tipos necessários estão localizados apenas em módulos específicos.

Listando os Módulos do JDK

JEP 200: O JDK MODULAR

JEP 201: CÓDIGO FONTE MODULAR

JEP 220: IMAGENS DE TEMPO MODULAR

JEP 260: ENCAPSULAR APIS MAIS INTERNO

JEP 261: SISTEMA DE MÓDULO

JEP 275: EMBALAGEM DE APLICAÇÃO DE JAVA MODULAR

JEP 282: JLINK: O LINKER DE JAVA

JSR 376: SISTEMA DE MÓDULO DE PLATAFORMA JAVA

JSR 379: JAVA SE 9

Tabela 1. JEPs e JSRs de Modularidade Java

Um aspecto crucial do Java 9 é dividir o JDK em módulos para suportar várias configurações. (Consulte “JEP 200: O JDK Modular”. Todos os JEPs e JSRs de modularidade Java são mostrados na **Tabela 1.**) Usando o comando java da pasta bin do JDK com a opção `--list-modules`, como em:

```
java --list-modules
```

lista o conjunto de módulos do JDK, que inclui os módulos padrão que implementam o Java Language SE Specification (nomes iniciados por `java`), módulos JavaFX (nomes iniciados por `javafx`), módulos específicos do JDK (nomes iniciados por `jdk`) e módulos específicos do Oracle com `oracle`). Cada nome de módulo é seguido por uma sequência de versão - `@9` indica que o módulo pertence ao Java 9.

Declarações do Módulo

Como mencionamos, um módulo deve fornecer um descritor de módulo - metadados que especificam as dependências do módulo, os pacotes que o módulo disponibiliza para outros módulos e muito mais. Um descritor de módulo é a versão compilada de uma declaração de módulo que é definida em um arquivo chamado `module-info.java`. Cada declaração de módulo começa com a palavra-chave `module`, seguida por um nome de módulo exclusivo e um corpo de módulo entre chaves, como em:

```
module nome_do_modulo {

}
```

O corpo da declaração do módulo pode estar vazia ou pode conter várias *directivas do módulo* , incluindo `requires`, `exports`, `provides…with`, `uses` e `opens` (cada um dos quais nós discutiremos). Como você verá mais adiante, a compilação da declaração do módulo cria o descritor do módulo, que é armazenado em um arquivo chamado `module-info.class` na pasta raiz do módulo. Aqui nós introduzimos brevemente cada diretiva de módulo. Depois disso, apresentaremos as declarações reais do módulo.

As palavras chave `exports`, `module`, `open`, `opens`, `provides`, `requires`, `uses`, `with`, bem como `to` e `transitive`, o que nós introduziremos mais tarde, são palavras-chave restritas. Eles são palavras-chave apenas em declarações de módulo e podem ser usados como identificadores em qualquer outro lugar em seu código.

requires. Uma diretiva de módulo `requires` especifica que esse módulo depende de outro módulo - esse relacionamento é chamado de *dependência de módulo* . Cada módulo deve declarar explicitamente suas dependências. Quando o módulo A `requires` módulo B, o módulo A é dito para *ler o* módulo B e o módulo B é *lido pelo* módulo A. Para especificar uma dependência em outro módulo, use `requires`, como em:

```
requires nome_do_modulo;
```

Há também uma diretiva `requires static` para indicar que um módulo é necessário em tempo de compilação, mas é opcional no tempo de execução. Isso é conhecido como uma *dependência opcional* e não será discutido nesta introdução.

requires transitive - implied readability . Para especificar uma dependência em outro módulo e para garantir que outros módulos que leiam seu módulo também leiam essa dependência - conhecida como *legibilidade implícita* - use `requires transitive`, como em:

```
requires transitive nome_do_modulo;
```

Considere a seguinte diretiva da declaração do módulo `java.desktop`:

```
requires transitive java.xml;
```

Neste caso, qualquer módulo que leia `java.desktop` também lê implicitamente `java.xml`. Por exemplo, se um método do módulo `java.desktop` retornar um tipo do módulo `java.xml`, o código nos módulos que ler `java.desktop` se tornará dependente `java.xml`. Sem a declaração do módulo da diretiva na `java.desktop` `requires transitive`, tais módulos dependentes não compilarão a menos que eles leiam *explicitamente* `java.xml` .

De acordo com a JSR 379 , os módulos padrão do Java SE devem conceder legibilidade implícita em todos os casos, como o descrito aqui. Além disso, embora um módulo padrão Java SE possa depender de módulos não padrão, ele *não deve* conceder legibilidade implícita a eles. Isso garante que o código que depende apenas dos módulos padrão do Java SE seja portátil nas implementações do Java SE.

exports e **exports ... to.** Uma diretiva de módulo `exports` especifica um dos pacotes do módulo cujos tipos `public` (e seus tipos aninhados `public` e `protected`) devem estar acessíveis ao código em todos os outros módulos. Uma diretiva `exports…to` permite que você especifique em uma lista separada por vírgula com precisão qual código de módulo ou módulo pode acessar o pacote exportado - isso é conhecido como exportação *qualificada*.

uses. Uma diretiva de módulo `uses` especifica um serviço usado por este módulo - tornando o módulo um consumidor de serviço. Um *serviço* é um objeto de uma classe que implementa a interface ou estende a classe `abstrata` especificada na diretiva `uses`.

provides… with. Uma diretiva de módulo `provides…with` especifica que um módulo fornece uma implementação de serviço - tornando o módulo um *provedor de serviços* . A parte `provides` da diretiva especifica uma interface ou classe `abstrata` listada na diretiva `uses` de um módulo e a parte `with` da diretiva especifica o nome da classe do provedor de serviços que `implementa` a interface ou `estende` uma classe `abstrata`.

open, open e **opens ... to.** Antes do Java 9, a reflexão poderia ser usada para aprender sobre todos os tipos em um pacote e todos os membros de um tipo - até mesmo seus membros `private` - independente se você queria ou não permitir esse recurso. Assim, nada foi verdadeiramente encapsulado.

A motivação chave do sistema de módulos é um forte encapsulamento. Por padrão, um tipo em um módulo não é acessível a outros módulos, a menos que seja um tipo público *e* você exporte seu pacote. Você expõe apenas os pacotes que deseja expor. Com o Java 9, isso também se aplica à *reflection*.

Permitindo acesso somente em tempo de execução a um pacote. Uma diretiva de módulo de abertura no formato:

```
opens pacote
```

indica que um determinado *pacote* de tipos `public` (e seus tipos aninhados `public` e `protected`) são acessíveis ao código em outros módulos em apenas tempo de execução. Além disso, todos os tipos no pacote especificado (e todos os membros dos tipos) são acessíveis via *reflection*.

Permitir acesso em tempo de execução apenas a um pacote por módulos específicos. Uma diretiva de módulo `opens…to` no formato:

```
opens pacote to lista_separada_por_virgula_de_modulos
```

indica que um determinado *pacote* de tipos `public` (e seus tipos aninhados `public` e `protected`) são acessíveis ao código nos módulos listados em apenas tempo de execução. Todos os tipos no pacote especificado (e todos os membros dos tipos) são acessíveis por meio de reflexão para codificar nos módulos especificados.

Permitir acesso somente em tempo de execução a todos os pacotes em um módulo. Se todos os pacotes em um determinado módulo devem estar acessíveis em tempo de execução e através de *reflection* para todos os outros módulos, você pode `abrir (open)` o módulo inteiro, como em:

```
open module nome_do_modulo {

   // diretivas do modulo

}
```

Reflection Padrão

Por padrão, um módulo com acesso por *reflection* em tempo de execução a um pacote pode ver os tipos `public` do pacote (e seus tipos aninhados `public` e `protected`). No entanto, o código em outros módulos pode acessar *todos os* tipos no pacote exposto e *todos os* membros dentro desses tipos, incluindo `private` membros por meio de `setAccessible`, como nas versões anteriores do Java.

Para mais informações `setAccessible` e *reflection*, consulte a documentação do Oracle .

No restante deste artigo , aprenda mais sobre os padrões de *reflection* e crie um aplicativo simples que demonstra os fundamentos dos módulos.

Paul Deitel, CEO e diretor técnico da Deitel & Associates, é formado pelo MIT com 35 anos de experiência em computação. Ele é um campeão em Java e vem programando em Java há mais de 22 anos. Ele e seu coautor, Dr. Harvey M. Deitel, são os autores de linguagem de programação mais vendidos do mundo. Paul ministrou cursos de programação Java, Android, iOS, C #, C ++, C e internet para clientes industriais, governamentais e acadêmicos internacionalmente.

NOTA: Este artigo foi extraído da *revista Java Magazine de* setembro / outubro de 2017. Continue com o artigo completo , que foi adaptado do livro recentemente publicado *Java 9 para programadores* , de Paul e Harvey Deitel.