



Jlink en Java 9

por Shubham Agarwal MVB · 09 de enero, 18 · Zona de Java

Obtenga el Edge con un IDE Java profesional. Prueba gratuita de 30 días .

Jlink es la nueva herramienta de línea de comandos de Java a través de la cual podemos crear nuestro propio **JRE personalizado** .

Por lo general, ejecutamos nuestro programa utilizando el JRE predeterminado, pero en caso de que desee crear su propio JRE, puede ir con el concepto de jlink.

¿Por qué construir tu propio JRE?

Veamos un ejemplo.

Supongamos que tenemos un programa simple de "hello world" como:

```
1  clase Test  {  
    public static void main ( String [] args )  
2      Sistema . a cabo . println ( "Hola mundo"  
3  
4      }  
5  }
```

Si quiero ejecutar este pequeño programa en nuestro sistema, necesito instalar un JRE predeterminado. Después de instalar el JRE predeterminado, puedo

Si ejecuto mi aplicación "hello world" con el JRE predeterminado, se ejecutarán todos los archivos .class predefinidos. Pero si solo necesito 3-4 archivos .class para ejecutar mi aplicación "hello world", entonces ¿por qué necesito mantener los archivos .class externos?

Entonces, el problema con el JRE predeterminado es que ejecuta todos los archivos .class predefinidos, ya sea que quieras o no.

Y si también miras el tamaño predeterminado de JRE, entonces encontrarás que es 203 MB. Para ejecutar mi código simple de 1 KB, tengo que mantener 203 MB de JRE en mi máquina. Es un desperdicio completo de memoria.

Entonces, usar el JRE predeterminado significa:

- Pérdida de memoria y un golpe de rendimiento
- No podrá desarrollar microservicios que contengan muy poca memoria.
- No es adecuado para dispositivos IoT

Por lo tanto, Java no era la mejor opción para microservicios y dispositivos IoT, pero eso solo era un problema a través de Java 1.8. Mientras tanto, Java 1.9 viene con jlink. Con jlink, podemos crear nuestro propio pequeño JRE que contenga las únicas clases relevantes que queremos tener. No habrá pérdida de memoria y el rendimiento aumentará.

aplicación basada en módulos personalizados o una predeterminado, puede usar el comando:

```
1 java -module-path out -m demoModule / knoldus.Tes
```

Pero como comentamos, nuestro programa "hello world" requería solo unos pocos archivos .class: String.class, System.class y Object.class. Estos archivos .class son parte del paquete java.lang, y el paquete java.lang es parte del módulo java.base. Entonces, si quiero ejecutar mi programa "hello world", solo se requieren dos módulos: DemoModule y el módulo java.base. Con estos dos módulos, podemos crear nuestro propio JRE personalizado para ejecutar esta aplicación.

Puede encontrar el módulo java.base en la ruta:


```
1 java \ jdk-9 \ jmods
```

Así que simplemente copie el módulo java.base y péguelo en la carpeta que tiene el archivo Test.class. Ahora podemos crear nuestro propio JRE usando el comando:

```
1 jlink -module-path out -add-modules demoModule, j
```

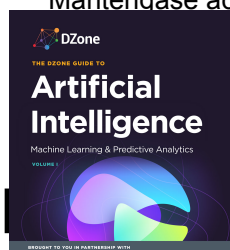
Obtenga el IDE de Java que entiende el código y hace que el desarrollo sea agradable. Sube de nivel tu código con IntelliJ IDEA. Descargue la versión de prueba gratuita .

Temas: JAVA, JLINK, JAVA 9, JRE, TUTORIAL

Publicado en DZone con el permiso de Shubham Agarwal , DZone MVB . [Vea el artículo original aquí.](#) 
Las opiniones expresadas por los contribuidores de DZone son suyas.

Obtenga lo mejor de Java en su bandeja de entrada.

Manténgase actualizado con DZone's Bi-weekly Java



PAR UN EJEMPLO



R

s para socios de

La Guía de Inteligencia Artificial 2017: Aprendizaje automático y análisis predictivo

Vea cómo la 2.ª edición de esta guía crítica de

- Descubra patrones en Analytics usando el poder del aprendizaje automático
- Aprenda sobre las redes neuronales usando las bibliotecas de Java
- Vea cómo el proyecto de código abierto de Google puede enriquecer las aplicaciones empresariales

Descargar My Free PDF

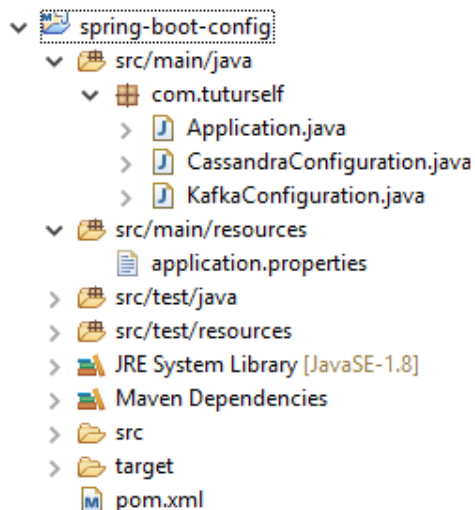
Externalizing Configuration Sources in Spring Boot Apps

by Arpan Das · Jan 05, 18 · Java Zone

Learn how to troubleshoot and diagnose some of the most common performance issues in Java today. Brought to you in partnership with AppDynamics.

As per Spring's documentation, Spring Boot allows us to externalize configurations, so you can work with the same application code in different environments. You can use property files, YAML files, environment variables, and command-line arguments to externalize configurations. But in this article we will mostly check how to read configurations from property or yml files. True externalization requires reading property or YAML files from external cloud sources like Consul, where Consul properties like consul host, port, and keys are provided to the application via environment variables. Then our same application code can run in different environments. We will cover that in some other article.

Here, we will discuss how configuration keys are bound to actual objects in Spring Boot applications. The most basic way to bind your configurations is from property or YAML files to POJO classes, which we can use later in



Let us have a look at our `pom.xml` first. Here we are using Spring Boot parent version 1.5.9.RELEASE.

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.tuturself</groupId>
4   <artifactId>spring-boot-config</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <name>spring-boot-config</name>
7   <parent>
8     <groupId>org.springframework.boot</groupI
9     <artifactId>spring-boot-starter-parent</a
10    <version>1.5.9.RELEASE</version>
11  </parent>
```

```

26         <plugins>
27             <plugin>
28                 <groupId>org.springframework.boot
29                 <artifactId>spring-boot-maven-plu
30             </plugin>
31         </plugins>
32     </build>
33 </project>

```

And the following are the properties that we will bind to our POJO classes from the `application.properties` file:

```

1  # Cassandra Configuration
2  #####
3  # Collection based binding
4  cassandra.server=127.0.0.1:9042,127.0.0.2:9042
5
6  # Variable based binding
7  cassandra.user=dbUser1
8  cassandra.password=dbUsEr)!
9
10 # Nested class based binding
11 cassandra.keyspace.name=test_keyspace
12 cassandra.keyspace.readConsistency=ONE
13 cassandra.keyspace.writeConsistency=ONE
14

```

This property file has connection information and other configurations for Cassandra and Kafka. First, let's check the POJO classes, which we have used to bind these properties. Cassandra information will be bound to the `CassandraConfiguration.java` class:

```
1 package com.tuturself;
2
3 import java.util.List;
4
5 import org.springframework.boot.context.properties
6
7 import lombok.AccessLevel;
8 import lombok.Data;
9 import lombok.Getter;
10
11 @Data
12 @ConfigurationProperties(prefix = "cassandra")
13 public class CassandraConfiguration {
14
15     private List server;
16     private String user;
17     private String password;
18
19     @Getter(AccessLevel.NONE)
20     private Keyspace keyspace;
21
22     public Keyspace getKeyspace() {
23         if (this.keyspace == null) {
```


let's look at the other binding strategies.

Variable-Based Binding

The following properties are directly bound to the matching attributes in the class:

```
1  cassandra.user=dbUser1 --> private String user;  
2  
3  cassandra.password=dbUser)! --> private String p
```

Collection-Based Binding

The comma separated Cassandra hosts are bound to a `List<String>` in the Class. This is an example of collection-based binding:

```
1  cassandra.server=127.0.0.1:9042,127.0.0.2:9042 to
```

Nested Property-Based Binding

The keyspace attributes are bound to an inner class named `keyspace`. This is an example of nested property binding. For nested property binding, we need to provide a getter to create the Object, or we can create the

In that case, we do not need the following part in our getter method:

```
1  if (this.keyspace == null) {  
2      this.keyspace = new Keyspace();  
3  }
```

The `@Data` annotation and `@Getter(AccessLevel.NONE)` are not related to Spring Boot property mapping. It is from Project Lombok, which will create the getters and setters in the domain class automatically. Read about project Lombok and its usages [here](#).

Now let's check another domain class that contains a Kafka configuration named `KafkaConfiguration.class`:

```
1  package com.tuturself;  
2  
3  import java.util.List;  
4  import java.util.Map;  
5  
6  import org.springframework.boot.context.properties.  
7  
8  import lombok.Data;  
9  
10 @Data  
11 @ConfigurationProperties(prefix = "kafka")  
12 public class KafkaConfiguration {  
13
```

```
1 # Map based binding
2 kafka.topicMap.one=topic-1
3 kafka.topicMap[two]=topic-2
4
5 To
6
7 private Map<String,String> topicMap;
```

Now let's define the Main class for our Spring Boot application to test it:

```

1 package com.tuturself;
2
3 import javax.annotation.PostConstruct;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6
7 import org.springframework.boot.SpringApplication;
8
9 import org.springframework.boot.autoconfigure.SpringBootApplication;
10
11 import org.springframework.boot.context.properties.EnableConfigurationProperties;
12
13
14 @SpringBootApplication
15 @EnableConfigurationProperties({
16     CassandraConfiguration.class,
17     KafkaConfiguration.class
18 })
19 public class Application {


```

```
28  
29     }  
30  
31     public static void main(String[] args) throws  
32     {  
33         SpringApplication.run(Application.class,  
34             System.out.println(cassandraConfiguration  
35             System.out.println(KafkaConfiguration.toS  
36     }  
37 }
```

`@EnableConfigurationProperties` automatically maps POJOs to a set of properties in the Spring Boot configuration file (by default: `application.properties`). We get the following output when we run the Spring Boot application:

```
1  CassandraConfiguration(  
2      server=[127.0.0.1:9042, 127.0.0.2:9042],  
3      user=dbUser1, password=dbUser!,  
4      keyspace=CassandraConfiguration.Keyspace(  
5          name=test_keyspace,  
6          readConsistency=ONE,  
7          writeConsistency=ONE  
8      )  
9  )  
10
```

Topics: JAVA, SPRING BOOT, PROPERTY BINDING, TUTORIAL

Published at DZone with permission of Arpan Das. [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

