24.外部化配置

Spring Boot允许你去进行外部化配置，以便于你能在不同环境中相同的程序代码能够运行。你可以使用属性文件，YAML文件，环境变量和命令行参数进行外部配置。属性值可以使用`@Value`注释直接注入到bean中，可以通过Spring的`Environment`抽象来访问，也可以通过`@ConfigurationProperties`绑定到结构化对象。

Spring Boot使用一个特殊的`PropertySource`命令，这个命令可以允许你去重写值。属性按照下面的顺序被考虑：

1. 主目录中的Devtools全局设置属性（`~/.spring-boot-devtools.properties`当devtools被激活时）。
2. 你的测试文件中的`@TestPropertySource`注解。
3. 测试文件中的`@SpringBootTest#properties`注解属性
4. 命令行参数。
5. 从`SPRING_APPLICATION_JSON`中获取的属性（环境变量或系统属性中嵌入的JSON属性）。
6. `ServletConfig`初始化属性。
7. `ServletContext`初始化属性。
8. 从`java:comp/env`中获取的JNDI属性。
9. Java系统属性（`System.getProperties`）。
10. 操作系统环境变量。
11. 仅仅有`random.*`属性的`RandomValuePropertySource`。
12. 被打包的jar文件外部的特定Profile应用属性（`application-{profile}.properties`和YAML文件）
13. 在你的jar包中的特定Profile应用属性（`application-{profile}.properties`和YAML文件）
14. 你的jar包外部的应用属性（`applicaiton.properties`和YAML文件）
15. 你的jar包内的应用属性文件（`applicaiton.properties`和YAML文件）
16. `@Configuration`注解标注的类中的`@PropertySource`注解
17. 默认属性（由`SpringApplication.setDefaultProperties`）。

现在我们提供一个具体的例子，假如你开发了一个`@Component`并且使用了一个`name`属性：

```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;

@Component
public class MyBean
{
    @Value("${name}")
    private String name;
    // ...
}
```

在你的应用类路径中（在jar）你可以使用`application.properties`提供默认的`name`属性值。当运行在新的环境中时，在你的jar包之外的`application.properties`可以完成覆盖`name`属性的功能。。对于一个一次性的测试，你也可以运行如下指定命令行程序（`java -jar app.jar --name="Spring"`）。

可以使用环境变量在命令行上提供`SPRING_APPLICATION_JSON`属性。例如，在UNIX shell中： $ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar 在这个例子中，Spring `Environment` 使用`foo.bar=spam`结束。你也可以通过系统变量来配置JSON作为`spring.application.json`，例如： $ java

-Dspring.application.json='{"foo":"bar"}' -jar myapp.jar 或者命令行参数： $ java -jar myapp.jar --spring.application.json='{"foo":"bar"}' 或者使用一个JNDI变量：java:comp/env/spring.application.json。

24.1 配置随机数值

RandomValuePropertySource在注入随机值时是非常有用的。它可以生成int值、long值、uuid或者字符串值。

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

random.int*语法是OPEN value (,max) CLOSE。在这里OPEN,CLOSE是任何字符，value,max值是整数值。如果max值被提供了，那么value是最小的值，max是最大的值（不包括该值）。

24.2 访问命令行属性

默认情况下，SpringApplication会将任何命令行选项参数（以' - '开头，例如--server.port = 9000）转换为property并将其添加到SpringEnvironment。正如上面所说的，命令行属性始终优先于其他属性源。

如果你不想让命令行参数添加到Environment中，你可以使用SpringApplication.setAddCommandLineProperties(false)禁用他们。

24.3 应用属性文件

SpringApplication将会从下面位置中的application.properties中加载配置，并将其添加到Spring Environment中：

1. 当前目录的子目录
2. 当前目录
3. 类路径/config包
4. 类路径根目录

这个列表按照优先级进行排序（在更高的位置定义的会覆盖低级位置定义的属性）。

你也可以使用YAML文件去指定".properties"。

如果你不喜欢applicaiton.properties作为配置文件名字，你可以通过spring.config.name环境变量去指定另一个名字。你也可以使用spring.config.location去指定特定的位置去配置环境变量。

```
$ java -jar myproject.jar --spring.config.name=myproject
```

或者：

```
$ java -jar myproject.jar --
spring.config.location=classpath:/default.properties,classpath:/override.propertie
s
```

spring.config.name和spring.config.location被用来决定哪个文件要被加载，因此它们必须作为一个环境属性被定义。

如果spring.config.location包含目录（而不是文件）它们应该用/结尾（并且会在加载之前附加从spring.config.name生成的名称，包括配置文件特定的文件名）。 在spring.config.location中指定的文件按原样使用，不支持配置文件特定的变体，并且将被任何配置文件特定的属性覆盖。

配置位置按相反顺序搜索。默认情况下，配置的位置如下classpath:/,classpath:/config/,file:./file:./config/。搜索顺序是：

1. file:./config/
2. file:./
3. classpath:/config/
4. classpath:/

当自定义配置位置被配置后，这些位置将会添加到默认位置。自定义位置将会在默认位置之前被搜索。例如，如果自定义位置classpath:/custom-config/,file:./custom-config/被配置，那么搜索顺序会变为：

1. file:./custom-config
2. classpath:custom-config/
3. file:./config/
4. file:./
5. classpath:/config/
6. classpath:/

搜索顺序允许你在一个配置文件中指定默认值，并且在另一个中选择性的覆盖他们。你可以在application.properties中提供默认值（或者是任何你在spring.config.name中指定名字的文件）。这些默认值能够在一个自定义目录中运行时覆盖这些参数。

如果你是用的是环境变量而不是系统属性，大多数操作系统不允许去使用.分隔键名称，但是你可以使用下划线命名法（使用SPRING_CONFIG_NAME而不是spring.config.name）。

如果你的程序运行在一个容器中，那么JNDI属性（在java:comp/env中）或者servlet上下文初始化参数会被使用，除此之外还有环境变量和系统属性。

24.4 Propile-specific 属性

除了application.properties文件，profile-specific属性文件应该使用application-{profile}.properties的格式进行命名。Environment拥有一系列默认的profiles（默认情况下是[default]），如果没有profiles被激活，那么将会使用这个（如果没有profiles被明确激活，那么applicaiton-default.properties将会被加载）。

特定于配置文件的属性从标准application.properties的相同位置加载，配置文件特定的文件始终覆盖非特定的文件，而不管特定于配置文件的文件是在打包的jar内部还是外部。

如果指定了多个配置文件，则应用最后一个赢取策略。 例如，由spring.profiles.active属性指定的配置文件将添加到通过SpringApplication API配置的配置文件之后，因此优先。

24.5 属性文件中的Placeholders

在你的`application.properties`中，你可以使用在`Environment`中存在的属性，例如：

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

24.6 使用YAML代替属性文件

YAML是JSON的超集，并且它是一个方便的格式化特定结构式配置信息的文件。只要你拥有一个SnakeYAML包在你的类路径中，`SpringApplication`会自动支持YAML。

如果你使用的是Spring Boot的启动器，那么`spring-boot-starter`会提供YAML支持。

24.6.1 加载YAML

Spring框架提供了两种边界的类用来加载YAML文档。`YamlPropertiesFactoryBean`将会加载YAML成为`Properties`，而`YamlMapFactoryBean`将会加载YAML成为一个`Map`。

例如，下面的YAML文件： environments: dev: url:http://dev.bar.com name:Developer Setup prod: url:http://foo.bar.com name:My Cool App 它将会被转化为如下的Properties: environments.dev.url=http://dev.bar.com environments.dev.name=Developer Setup environments.prod.url=http://foo.bar.com environments.prod.name=My Cool App YAML列表被表示为带有`[index]`引用的属性键，例如这个YAML：

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

如果想像使用Spring `DataBinder`工具那样（它是由`@ConfigurationProperties`实现的）你需要在你的目标bean中存在一个类型为`java.util.List`（或者为`Set`）的属性，或者提供一个setter，或者使用一个变值初始化它，等等，这样才能保证属性被绑定。

```
@ConfigurationProperties(prefix="my")
public class Config
{
    private List<String> servers = new ArrayList<String>();

    public List<String> getServers()
    {
        return this.servers;
    }
}
```

使用这种方式配置列表时要特别小心，因为覆盖不会像您希望的那样工作。在上面的例子中，当my.servers被几个地方重定义时，单个元素将会被覆盖，而不是列表。为了确保拥有高优先级的PropertySource能够覆盖列表，你可以按照一个单值属性一样定义它。

```
my:
    servers:dev.bar.com,foo.bar.com
```

24.6.2 在Spring Environment中作为属性暴露YAML

YamlPropertySourceLoader类能在Spring Environment中将YAML 作为PropertySource进行暴露。这样你就可以使用熟悉的@Value注解和占位符语法来访问YAML属性。

24.6.3 Multi-profile YAML文件

你可以在一个文件中指定多个用于配置的profile-specific YAML文档，靠spring.profiles指定哪个文档被使用，例如：

```
server:
    address: 192.168.1.100
---
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production
server:
    address: 192.168.1.120
```

在上面的例子中，如果development选项被激活，那么server.address属性应该是127.0.0.1。如果development和production选项都未被激活，那么属性值应该是192.168.1.100。

如果你没有明确指定哪个profile会被激活，那么应用启动时默认的profile会被激活。所以在这个YAML文件中我们设定一个仅仅在默认情况下能够被访问的security.user.password值。

```
server:
    port:8000
---
spring:
    profiles:default
security:
    user:
        password:weak
```

而在本例中，由于密码未附加到任何配置文件，因此始终设置密码，必须根据需要在所有其他配置文件中明确重置密码。

```
server:
    port:8000
security:
    user:
        password:weak
```

Spring profiles designated using the "spring.profiles" element may optionally be negated using the ! character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match and no negated profiles may match.

24.6.4 YAML 缺点

YAML不能通过`@PropertySource`注解进行加载。所以在这种情况下你需要去加载属性值，你应该使用一个属性文件。

24.6.5 合并YAML列表

就像上面我们看到的，任何的YAML内容最终都会被转化为属性。通过配置文件覆盖list属性时，这个过程可能不够直观。

例如，假如创建一个`MyPojo`对象其`name`和`description`属性值默认都为`null`。让我们通过`FooProperties`来注入一个`MyPojo`列表。

```
@ConfigurationProperties("foo")
public class FooProperties
{
    private final List<MyPojo> list = new ArrayList<>();

    public List<MyPojo> getList()
    {
        return this.list;
    }
}
```

考虑下面的配置：

```
foo:
    list:
      - name: my name
        description: my description
---
spring:
    profiles:dev
```

```
foo:
    list:
      - name:my another name
```

如果dev选项未被激活，FooProperties.list将会包含如上述定义的要给MyPojo对象。如果dev选项被激活，那么list将会只包含一个对象（其名字为“my another name”并且其描述为null）。这种配置将不能添加第二个MyPojo实例到该列表中，并且它也不能合并实例。

当在多个profiles中一个集合被指定时，那么最高优先级的将会被使用（并且仅仅只有这一个）：

```
foo:
    list:
      - name: my name
        description:my description
      - name: another name
        description: another description
---
spring:
    profiles: dev
foo:
    list:
      - name: my another name
```

在上面的例子中，如果dev选项被激活了，那么FooProperties.list将只会包含一个MyPojo实例（名字为“my another name”，描述为null）。

24.7 类型安全配置属性

使用@Value("${property}")注解注入配置属性有时是很笨重的，尤其是如果你使用多值属性或者你的信息是结构化的。Spring Boot提供了一种使用属性的替代方法，该属性允许强类型的bean管理和验证应用程序的配置。

```
package com.example;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("foo")
public class FooProperties {

    private boolean enabled;

    private InetAddress remoteAddress;
```

```
    private final Security security = new Security();

    public boolean isEnabled() { ... }

    public void setEnabled(boolean enabled) { ... }

    public InetAddress getRemoteAddress() { ... }

    public void setRemoteAddress(InetAddress remoteAddress) { ... }

    public Security getSecurity() { ... }

    public static class Security {

        private String username;

        private String password;

        private List<String> roles = new ArrayList<>
(Collections.singleton("USER"));

        public String getUsername() { ... }

        public void setUsername(String username) { ... }

        public String getPassword() { ... }

        public void setPassword(String password) { ... }

        public List<String> getRoles() { ... }

        public void setRoles(List<String> roles) { ... }

    }
}
```

上面定义的POJO遵循下面的属性：

- `foo.enabled`，默认情况下是`false`。
- `foo.remote-address`， 是一个能从`String`强制转型的类型。
- `foo.security.username`，带有嵌套的“安全性”，其名称由属性的名称确定。 特别是返回类型根本没有使用，可能是SecurityProperties
- `foo.security.password`
- `foo.security.roles`,一个`String`集合。

getter和setter方法是被强制的，因为绑定是通过标准Java Beans属性描述完成的，就像Spring MVC中的。下面的情况可能会忽略setter：

- Maps，只要他们被初始化，他们只需要一个getter，setter是不需要的，因为他们能通过binder进行更改。
- 集合和数组都能通过索引进行访问，（通常使用YAML）使用一个包含逗号分隔符的值作为属性。在后面的例子中，一个setter是必须的。我们建议对于任何的类型你最好都提供一个setter。如果你初始化类一个集合，请确保它不总是一成不变的（就像上面的例子一样）。
- 如果嵌入式的POJO属性要被初始化（就像上面的`Security`），那么一个setter不是必须的。如果你想去绑定一个使用默认构造器新创建的及时实例，你将需要一个setter。

一些人使用Lombok项目去自动添加getters和setters。确保Lombok不会为这种类型生成任何特定的构造函数，因为它将被容器自动使用来实例化对象。

你需要在`@EnableConfigurationProperties`注解中列出要注册的属性类：

```
@Configuration
@EnableConfigurationProperties(FooProperties.class)
public class MyConfiguration {
}
```

当使用`@ConfigurationProperties`bean通过这种方式进行注册，那么这个bean将会有一个固定格式的名字：`<prefix>-<fqn>`，其中`prefix`是在`@ConfigurationProperties`中指定的前缀，并且`fqn`是这个bean的全限定名。如果这个注解没有提供任何前缀，那么bean的名字仅仅只有全限定名。这里给出一个例子`foo-com.example.FooProperties`。