# Spring boot特性

·创建独立的spring应用

·内嵌tomcat，jetty，Undertow（你不再需要部署war包）

·提供自己入门级的pom以简化maven配置

·尽可能自动配置Spring

·绝对不会生成代码，也不需要XML配置

·为所有Spring开发提供一个更快，更广泛的入门体验。

·开箱即用，但随着需求开始偏离默认设置，快速退出

·提供大型项目（如嵌入式服务器，安全性，指标，运行状况检查，外部配置）通用的一系列非功能性特性。

快速入门

从<https://start.spring.io/>下载简单的demo，运行

# servlet容器

以下内嵌servlet容器是开箱即用的

| **Name** | **Servlet Version** | **Java Version** |
| --- | --- | --- |
| Tomcat 8 | 3.1 | Java 7+ |
| Tomcat 7 | 3.0 | Java 6+ |
| Jetty 9.3 | 3.1 | Java 8+ |
| Jetty 9.2 | 3.1 | Java 7+ |
| Jetty 8 | 3.0 | Java 6+ |
| Undertow 1.3 | 3.1 | Java 7+ |

# 安装Spring Boot

Spring Boot可以与“经典”Java开发工具一起使用，也可以作为命令行工具安装。 无论如何，您将需要Java SDK v1.6或更高版本。 在开始之前，您应该检查您当前的Java安装：java –version。尽管Spring Boot与Java 1.6兼容，但如果可能的话，应该考虑使用最新版本的Java。

## 使用Maven安装spring boot

Spring Boot与Apache Maven 3.2或更高版本兼容。 如果您还没有安装Maven，您可以按照 [maven.apache.org](https://maven.apache.org/)上的说明进行操作。

在许多操作系统上，Maven可以通过包管理器来安装。 如果您是OSX Homebrew用户，请尝试brew install maven安装maven。Ubuntu用户可以运行sudo apt-get install maven。

Spring Boot依赖使用org.springframework.boot作为groupId。 通常，您的Maven POM文件将从spring-boot-starter-parent项目继承，并向一个或多个“Starter”声明依赖关系。 Spring Boot还提供了一个可选的Maven插件来创建可执行的jar文件。

典型的spring boot pom.xml的配置文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>myproject</artifactId>

<version>0.0.1-SNAPSHOT</version>

*<!-- Inherit defaults from Spring Boot -->*

*<!-- 继承Spring Boot的默认值 -->*

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.10.RELEASE</version>

</parent>

*<!-- Add typical dependencies for a web application -->*

*<!-- 为Web应用程序添加典型的依赖关系-->*

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

*<!-- Package as an executable jar -->*

*<!-- 打包成可执行的jar文件 -->*

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Spring-Boot-starter-parent是使用Spring Boot的好方法，但它可能并不适合所有的情况。 有时您可能需要从不同的父POM继承，或者您可能不喜欢我们的默认设置。 有关使用导入范围的替代解决方案，请参见第13.2.2节“使用没有父POM的Spring Boot”。

### //TODO Gradle installation待定

# 安装Spring Boot CLI

Spring Boot CLI是一个命令行工具，如果你想用Spring来快速创建原型，可以使用它。 它允许你运行Groovy脚本，这意味着你有一个熟悉的类Java语法，不需要太多的样板代码。您可能不需要使用CLI来使用Spring Boot，但它绝对是搭建Spring应用程序的最快捷方式。

手动安装

您可以从Spring软件存储库下载Spring CLI发行版

下载地址：

* [spring-boot-cli-1.5.10.RELEASE-bin.zip](https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.10.RELEASE/spring-boot-cli-1.5.10.RELEASE-bin.zip)
* [spring-boot-cli-1.5.10.RELEASE-bin.tar.gz](https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.10.RELEASE/spring-boot-cli-1.5.10.RELEASE-bin.tar.gz)

最新的快照版本也是可用的。

下载之后，请按照解压缩归档中的INSTALL.txt说明进行操作。 总结：在.zip文件的bin /目录下有一个spring脚本（适用于Windows的spring.bat），或者可以使用带有.jar文件的java -jar（该脚本可以帮助您确保类路径设置正确）。

INSTALL.txt文件内容如下：

SPRING BOOT CLI - INSTALLATION

==============================

Thank you for downloading the Spring Boot CLI tool. Please follow these instructions

in order to complete your installation.

Prerequisites

-------------

Spring Boot CLI requires Java JDK v1.6 or above in order to run. Groovy v2.4.13

is packaged as part of this distribution, and therefore does not need to be installed (any

existing Groovy installation is ignored).

The CLI will use whatever JDK it finds on your path, to check that you have an appropriate

version you should run:

java -version

Alternatively, you can set the JAVA\_HOME environment variable to point an suitable JDK.

Environment Variables

---------------------

No specific environment variables are required to run the CLI, however, you may want to

set SPRING\_HOME to point to a specific installation. You should also add SPRING\_HOME/bin

to your PATH environment variable.

Shell Completion

----------------

Shell auto-completion scripts are provided for BASH and ZSH. Add symlinks to the appropriate

location for your environment. For example, something like:

ln -s ./shell-completion/bash/spring /etc/bash\_completion.d/spring

ln -s ./shell-completion/zsh/\_spring /usr/local/share/zsh/site-functions/\_spring

Checking Your Installation

--------------------------

To test if you have successfully install the CLI you can run the following command:

spring --version

./spring --version

Spring CLI v1.5.10.RELEASE

# Part III. Using Spring Boot

## 13. Build systems

强烈建议您选择支持依赖管理的构建系统，并且可以使用发布到“Maven Central”存储库的工件。 官方建议您选择Maven或Gradle。虽然 Spring Boot可以与其他构建系统（例如Ant）一起工作，但是它们不会得到特别好的支持。

## 13.1 Dependency management

Spring Boot的每个发行版都提供了一个支持的依赖列表。 在实践中，您不需要为构建配置中的任何这些依赖项提供一个版本，因为Spring Boot正在为您进行管理。 当您升级Spring Boot本身时，这些依赖关系也将以一致的方式升级。

如果您觉得有必要，您仍然可以指定一个版本并覆盖Spring Boot的建议版本。

支持的依赖列表包含您可以使用Spring Boot的所有模块以及第三方库的精炼列表。 该列表可作为标准的spring-boot-dependencies清单提供，并且还提供Maven和Gradle的其他专用支持。

Spring Boot的每个版本都与Spring Framework的基础版本相关联，因此官方强烈建议您不要自行指定其版本。

## 13.2 Maven

Maven用户可以从spring-boot-starter-parent项目继承，以获得合理的默认值。 父项目提供以下功能：

·Java 1.6作为默认的编译器级别。

·UTF-8源码编码。

·依赖管理部分，继承自Spring-Boot-dependencies POM的通用依赖项，允许您省略的<version>标记。

·合理的的资源过滤。

·合理的插件配置（exec plugin, surefire, Git commit ID, shad）。

·对application.properties和application.yml进行合理的资源过滤，包括特定于配置文件的文件（例如application-foo.properties和application-foo.yml）

### 13.2.1 Inheriting the starter parent

要将项目配置为从spring-boot-starter-parent继承，只需设置父项：

*<!-- Inherit defaults from Spring Boot -->*

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.10.RELEASE</version>

</parent>

通过该设置，您还可以通过在自己的项目中重写属性来覆盖各个依赖项。 例如，要升级到另一个Spring Data发行版，您需要将以下内容添加到您的pom.xml中。

<properties>

<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>

</properties>

检查spring-boot-dependencies pom以获取支持的属性列表。

<https://github.com/spring-projects/spring-boot/blob/v1.5.10.RELEASE/spring-boot-dependencies/pom.xml>

### 13.2.2 Using Spring Boot without the parent POM

不是每个人都喜欢从spring-boot-starter-parent POM继承。 你可能有你自己的企业标准的parent可以使用，或者你可能只是喜欢显式声明所有的Maven配置。

如果你不想使用spring-boot-starter-parent，你仍然可以通过使用scope = import dependency来保持依赖管理的好处（但不是插件管理）：

<dependencyManagement>

<dependencies>

<dependency>

*<!-- Import dependency management from Spring Boot -->*

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.10.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

如上所述，该设置不允许您使用属性覆盖单个依赖项。 为了达到同样的结果，你需要在spring-boot-dependencies条目之前在项目的dependencyManagement中添加一个条目。 例如，要升级到另一个Spring Data发行版，您需要将以下内容添加到您的pom.xml中。

<dependencyManagement>

<dependencies>

*<!-- Override Spring Data release train provided by Spring Boot -->*

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-releasetrain</artifactId>

<version>Fowler-SR2</version>

<scope>import</scope>

<type>pom</type>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.10.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

### 13.2.3 Changing the Java version

spring-boot-starter-parent选择相当保守的Java兼容性。 如果您想遵循官方的建议并使用较新的Java版本，则可以添加一个java.version属性：

<properties>

<java.version>1.8</java.version>

</properties>

### 13.2.4 Using the Spring Boot Maven plugin

Spring Boot包含一个Maven插件，可以将项目打包为可执行的jar文件。 如果你想使用它，请将插件添加到<plugins>部分：

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

如果你使用Spring Boot starter parent pom，你只需要添加插件，除非你想改变在父代中定义的设置，否则不需要进行配置。

## 13.3 Gradle

Gradle用户可以直接在他们的依赖部分导入'starters'。 与Maven不同的是，没有““super parent”可以导入来共享某些配置。

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web:1.5.10.RELEASE")

}

spring-boot-gradle-plugin也是提供了创建可执行的jar和从源代码运行项目的任务的功能。 它还提供依赖性管理，还允许您省略由Spring Boot管理的任何依赖项的版本号：

plugins {

id 'org.springframework.boot' version '1.5.10.RELEASE'

id 'java'

}

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

## 13.5 Starters

Starters是一套方便的依赖描述符，可以包含在应用程序中。 您可以获得所需的所有Spring及相关技术的一站式服务，无需搜索示例代码，也不需要复制粘贴大量的依赖关系描述符。 例如，如果你想开始使用Spring和JPA进行数据库访问，只需在你的项目中加入spring-boot-starter-data-jpa依赖项，你就可以开始了。

Starters包含很多您需要快速启动并运行一个项目的依赖项，并使用一组支持的传递依赖项。

所有官方starters都遵循类似的命名模式; spring-boot-starter- \*，其中\*是特定类型的应用程序。 这种命名结构旨在帮助您找到启动器。 许多IDE中的Maven集成允许您按名称搜索依赖项。 例如，安装适当的Eclipse或STS插件后，只需在POM编辑器中点击ctrl-space并键入“spring-boot-starter”即可获得完整列表。

第三方starters不应该以spring-boot命名作为开始，因为它是为官方的Spring Boot预留的。 acme的第三方starter通常被命名为acme-spring-boot-starter。

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter | Core starter, including auto-configuration support, logging and YAML |
| spring-boot-starter-activemq | Starter for JMS messaging using Apache ActiveMQ |
| spring-boot-starter-amqp | Starter for using Spring AMQP and Rabbit MQ |
| spring-boot-starter-aop | Starter for aspect-oriented programming with Spring AOP and AspectJ |
| spring-boot-starter-artemis | Starter for JMS messaging using Apache Artemis |
| spring-boot-starter-batch | Starter for using Spring Batch |
| spring-boot-starter-cache | Starter for using Spring Framework’s caching support |
| spring-boot-starter-cloud-connectors | Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku |
| spring-boot-starter-data-cassandra | Starter for using Cassandra distributed database and Spring Data Cassandra |
| spring-boot-starter-data-couchbase | Starter for using Couchbase document-oriented database and Spring Data Couchbase |
| spring-boot-starter-data-elasticsearch | Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch |
| spring-boot-starter-data-gemfire | Starter for using GemFire distributed data store and Spring Data GemFire |
| spring-boot-starter-data-jpa | Starter for using Spring Data JPA with Hibernate |
| spring-boot-starter-data-ldap | Starter for using Spring Data LDAP |
| spring-boot-starter-data-mongodb | Starter for using MongoDB document-oriented database and Spring Data MongoDB |
| spring-boot-starter-data-neo4j | Starter for using Neo4j graph database and Spring Data Neo4j |
| spring-boot-starter-data-redis | Starter for using Redis key-value data store with Spring Data Redis and the Jedis client |
| spring-boot-starter-data-rest | Starter for exposing Spring Data repositories over REST using Spring Data REST |
| spring-boot-starter-data-solr | Starter for using the Apache Solr search platform with Spring Data Solr |
| spring-boot-starter-freemarker | Starter for building MVC web applications using FreeMarker views |
| spring-boot-starter-groovy-templates | Starter for building MVC web applications using Groovy Templates views |
| spring-boot-starter-hateoas | Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS |
| spring-boot-starter-integration | Starter for using Spring Integration |
| spring-boot-starter-jdbc | Starter for using JDBC with the Tomcat JDBC connection pool |
| spring-boot-starter-jersey | Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to [spring-boot-starter-web](https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/#spring-boot-starter-web) |
| spring-boot-starter-jooq | Starter for using jOOQ to access SQL databases. An alternative to [spring-boot-starter-data-jpa](https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/#spring-boot-starter-data-jpa) or [spring-boot-starter-jdbc](https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/#spring-boot-starter-jdbc) |
| spring-boot-starter-jta-atomikos | Starter for JTA transactions using Atomikos |
| spring-boot-starter-jta-bitronix | Starter for JTA transactions using Bitronix |
| spring-boot-starter-jta-narayana | Spring Boot Narayana JTA Starter |
| spring-boot-starter-mail | Starter for using Java Mail and Spring Framework’s email sending support |
| spring-boot-starter-mobile | Starter for building web applications using Spring Mobile |
| spring-boot-starter-mustache | Starter for building MVC web applications using Mustache views |
| spring-boot-starter-security | Starter for using Spring Security |
| spring-boot-starter-social-facebook | Starter for using Spring Social Facebook |
| spring-boot-starter-social-linkedin | Stater for using Spring Social LinkedIn |
| spring-boot-starter-social-twitter | Starter for using Spring Social Twitter |
| spring-boot-starter-test | Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito |
| spring-boot-starter-thymeleaf | Starter for building MVC web applications using Thymeleaf views |
| spring-boot-starter-validation | Starter for using Java Bean Validation with Hibernate Validator |
| spring-boot-starter-web | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container |
| spring-boot-starter-web-services | Starter for using Spring Web Services |
| spring-boot-starter-websocket | Starter for building WebSocket applications using Spring Framework’s WebSocket support |

除了应用程序starter之外，还可以将以下starter添加到准生产功能：

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter-actuator | Spring Boot的 Actuator Starter帮助您监控和管理您的应用程序 |
| spring-boot-starter-remote-shell | 使用CRaSH Starter远程shell通过SSH监视和管理您的应用程序。 自1.5以来已弃用 |

最后，Spring Boot还包括一些可以用来排除或交换特定技术方面的starter:

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter-jetty | 用Jetty作为内嵌的servlet容器的starter。spring-boot-starter-tomcat的替代方案 |
| spring-boot-starter-log4j2 | 用Log4j2记录日志的Starter。spring-boot-starter-logging的替代方案。 |
| spring-boot-starter-logging | 使用Logback进行日志记录的Starter。 默认日志启动器 |
| spring-boot-starter-tomcat | 使用Tomcat作为嵌入式servlet容器的starter。 spring-boot-starter-web使用的默认servlet容器启动器 |
| spring-boot-starter-undertow | 使用Undertow作为内嵌的servlet容器。spring-boot-starter-tomcat的替代方案。 |

## 14. Structuring your code

Spring Boot不需要任何特定的代码布局，但是，有一些最佳实践可以提供帮助

## 14.1 Using the “default” package

当一个类不包含包声明时，它被认为是在“默认包”中。 通常不鼓励使用“默认软件”，应该避免使用“默认软件”。 对于使用@ComponentScan，@EntityScan或@SpringBootApplication注释的Spring Boot应用程序来说，这可能会导致问题，因为每个jar的每个类都将被读取。

官方建议您遵循Java推荐的软件包命名约定，并使用反向域名（例如，com.example.project）。

## 14.2 Locating the main application class

官方通常建议您将主应用程序类放在其他类的根包中。 @EnableAutoConfiguration注解通常放在主类上，它隐式地为某些项目定义了一个基本的“搜索包”。 例如，如果您正在编写JPA应用程序，则被@EnableAutoConfiguration注解的类的所在的包将用于搜索@Entity。

使用根包也使得@ComponentScan注释不需要指定basePackage属性。 如果您的主类位于根包中，也可以使用@SpringBootApplication注释。

这是一个典型的布局：

com

+- example

+- myproject

+- Application.java

|

+- domain

| +- Customer.java

| +- CustomerRepository.java

|

+- service

| +- CustomerService.java

|

+- web

+- CustomerController.java

## 15. Configuration classes

Application.java文件声明main方法以及基本的@Configuration。

**package** com.example.myproject;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

*@Configuration*

*@EnableAutoConfiguration*

*@ComponentScan*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

Spring Boot支持基于Java的配置。 虽然可以使用XML来调用SpringApplication.run（），但通常建议您的主要来源是@Configuration类。 通常，定义main方法的类也是可以使用@Configuration注解。

## 15.1 Importing additional configuration classes

你不需要把所有的@Configuration放到一个类中。 @Import注释可用于导入其他配置类。 或者，您可以使用@ComponentScan自动获取所有Spring组件，包括@Configuration类。

## 15.2 Importing XML configuration

如果您必须要使用基于XML的配置，建议您仍以@Configuration类开头。 然后您可以使用额外的@ImportResource注解来加载XML配置文件。

## 16. Auto-configuration

Spring Boot自动配置会尝试根据您添加的jar依赖项自动配置您的Spring应用程序。 例如，如果HSQLDB在您的类路径中，并且您没有手动配置任何数据库连接Bean，那么Spring Boot将自动配置一个内存数据库。

您需要通过将@EnableAutoConfiguration或@SpringBootApplication注释添加到其中一个@Configuration类来选择自动配置。

您可以只添加一个@EnableAutoConfiguration注释。 我们通常建议您将其添加到您的被@Configuration注解的主类上。

## 16.1 Gradually replacing auto-configuration

自动配置是非侵入式的，您可以随时开始定义自己的配置来替换自动配置的特定部分。 例如，如果添加自己的DataSource Bean，则默认的嵌入式数据库支持将退出。

如果您需要了解当前正在应用的自动配置，以及为什么。使用--debug开关启动您的应用程序。 这将启用选择核心记录器的调试日志，并将自动配置报告记录到控制台。

## 16.2 Disabling specific auto-configuration

如果您发现正在应用您不需要的特定自动配置类，则可以使用@EnableAutoConfiguration的exclude属性来禁用它们。

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.boot.autoconfigure.jdbc.\*;

**import** org.springframework.context.annotation.\*;

*@Configuration*

*@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})*

**public** **class** MyConfiguration {

}

如果类不在类路径中，则可以使用注释的excludeName属性，并指定完全限定的名称。 最后，您还可以通过spring.autoconfigure.exclude属性来控制自动配置类的列表。您可以在注释级别和使用属性中定义排除。

## 17. Spring Beans and dependency injection

您可以自由使用任何标准的Spring框架技术来定义您的bean及其注入的依赖关系。 为了简单起见，我们经常发现使用@ComponentScan来查找bean，并结合使用@Autowired构造函数注入效果很好。

如果按上面的建议构建代码（在根包中查找应用程序类），则可以添加@ComponentScan而不带任何参数。 所有的应用程序组件（@Component，@Service，@Repository，@Controller等）都将被自动注册为Spring Bean。

这里是一个示例@Service Bean，它使用构造函数注入来获得必需的RiskAssessor bean。

**package** com.example.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

*@Service*

**public** **class** DatabaseAccountService **implements** AccountService {

*@Autowired*

**private** **final** RiskAssessor riskAssessor;

*// ...*

}

## 18. Using the @SpringBootApplication annotation

许多Spring Boot开发人员总是使用@Configuration，@EnableAutoConfiguration和@ComponentScan注解其主类。 由于这些注释经常一起使用（特别是如果您遵循以上最佳实践），Spring Boot提供了一种方便的@SpringBootApplication替代方法。

@SpringBootApplication注释等价于使用@Configuration，@EnableAutoConfiguration和@ComponentScan三个注解的默认属性：

**package** com.example.myproject;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

*@SpringBootApplication* *// same as @Configuration @EnableAutoConfiguration @ComponentScan*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

@SpringBootApplication还提供别名来自定义@EnableAutoConfiguration和@ComponentScan的属性。

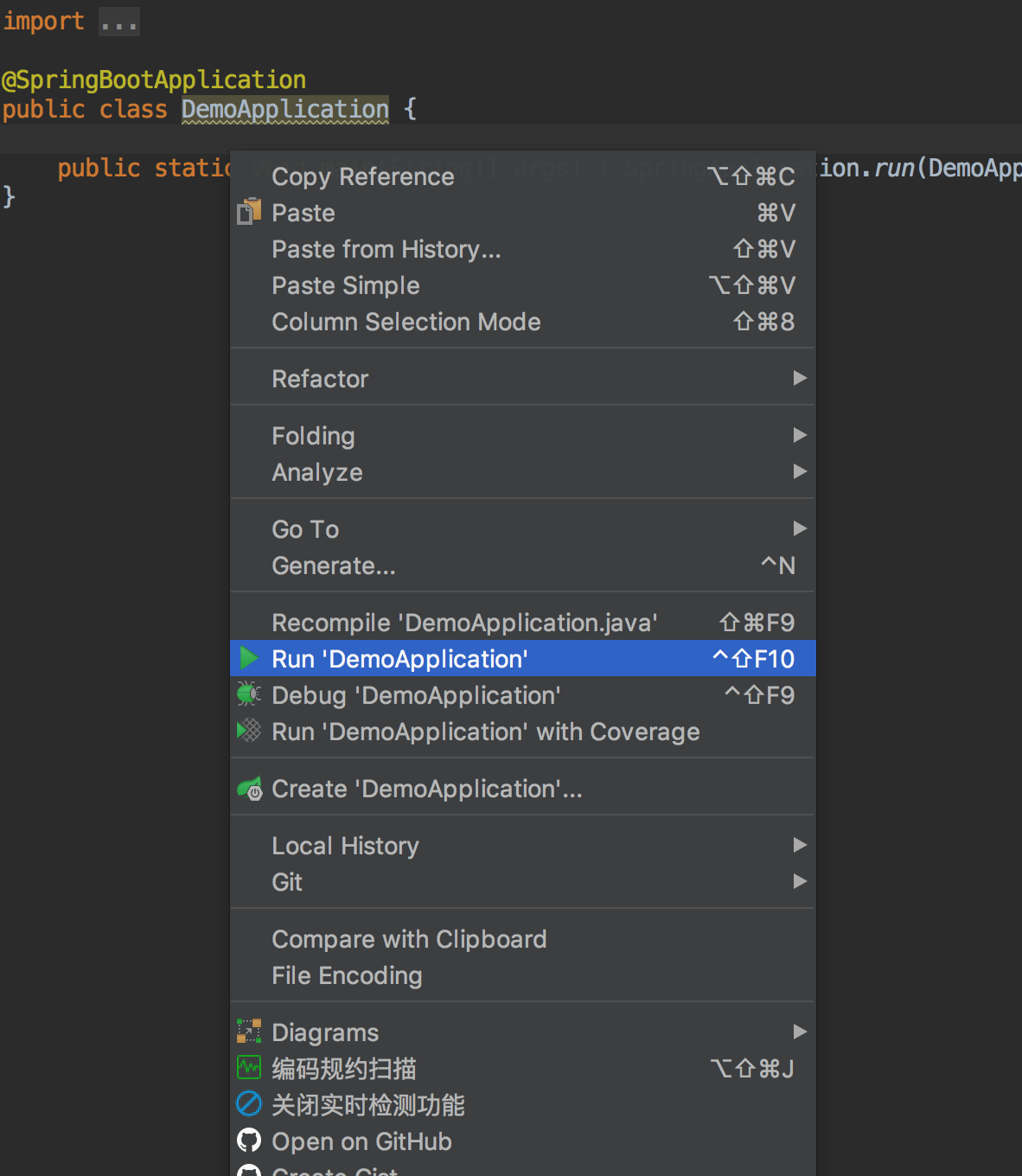
/\*  
 \* Copyright 2012-2017 the original author or authors.  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/  
  
package org.springframework.boot.autoconfigure;  
  
import java.lang.annotation.Documented;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Inherited;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
import org.springframework.boot.SpringBootConfiguration;  
import org.springframework.boot.context.TypeExcludeFilter;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.ComponentScan.Filter;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.FilterType;  
import org.springframework.core.annotation.AliasFor;  
  
*/\*\*  
 \* Indicates a {****@link*** *Configuration configuration} class that declares one or more  
 \* {****@link*** *Bean @Bean} methods and also triggers {****@link*** *EnableAutoConfiguration  
 \* auto-configuration} and {****@link*** *ComponentScan component scanning}. This is a convenience  
 \* annotation that is equivalent to declaring {****@code*** *@Configuration},  
 \* {****@code*** *@EnableAutoConfiguration} and {****@code*** *@ComponentScan}.  
 \*  
 \** ***@author*** *Phillip Webb  
 \** ***@author*** *Stephane Nicoll  
 \** ***@since*** *1.2.0  
 \*/*@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(excludeFilters = {  
 @Filter(type = FilterType.*CUSTOM*, classes = TypeExcludeFilter.class),  
 @Filter(type = FilterType.*CUSTOM*, classes = AutoConfigurationExcludeFilter.class) })  
public @interface SpringBootApplication {  
  
 */\*\*  
 \* Exclude specific auto-configuration classes such that they will never be applied.  
 \** ***@return*** *the classes to exclude  
 \*/* @AliasFor(annotation = EnableAutoConfiguration.class, attribute = "exclude")  
 Class<?>[] exclude() default {};  
  
 */\*\*  
 \* Exclude specific auto-configuration class names such that they will never be  
 \* applied.  
 \** ***@return*** *the class names to exclude  
 \** ***@since*** *1.3.0  
 \*/* @AliasFor(annotation = EnableAutoConfiguration.class, attribute = "excludeName")  
 String[] excludeName() default {};  
  
 */\*\*  
 \* Base packages to scan for annotated components. Use {****@link*** *#scanBasePackageClasses}  
 \* for a type-safe alternative to String-based package names.  
 \** ***@return*** *base packages to scan  
 \** ***@since*** *1.3.0  
 \*/* @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")  
 String[] scanBasePackages() default {};  
  
 */\*\*  
 \* Type-safe alternative to {****@link*** *#scanBasePackages} for specifying the packages to  
 \* scan for annotated components. The package of each class specified will be scanned.  
 \* <p>  
 \* Consider creating a special no-op marker class or interface in each package that  
 \* serves no purpose other than being referenced by this attribute.  
 \** ***@return*** *base packages to scan  
 \** ***@since*** *1.3.0  
 \*/* @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")  
 Class<?>[] scanBasePackageClasses() default {};  
  
}

## 19.1 Running from an IDE

将应用程序打包为jar并使用内嵌的HTTP服务器的最大优点之一就是您可以像运行其他应用程序一样运行应用程序。 调试Spring Boot应用程序也很容易; 你不需要任何特殊的IDE插件或扩展。

## 19.1 Running from an IDE

您可以从IDE运行Spring Boot应用程序作为一个简单的Java应用程序，但是，首先您需要导入您的项目。 导入步骤取决于您的IDE和构建系统。 大多数IDE可以直接导入Maven项目。



如果您不小心运行了两次Web应用程序，则会看到“端口已被使用”错误。 STS用户可以使用“重新启动”按钮而不是“运行”来确保关闭任何现有的实例。

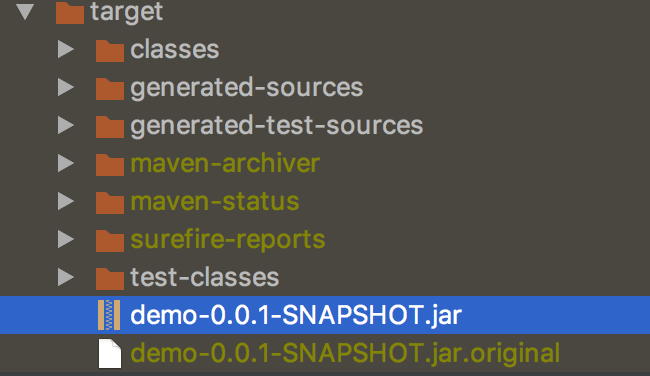
Description:

The Tomcat connector configured to listen on port 8080 failed to start. The port may already be in use or the connector may be misconfigured.

## 19.2 Running as a packaged application

如果您使用Spring Boot Maven或Gradle插件创建可执行jar，则可以使用java -jar运行应用程序。 例如：

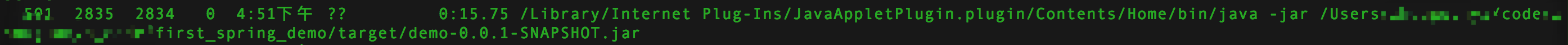
假设现在打了一个jar包：



执行以下命令行：java -jar target/demo-0.0.1-SNAPSHOT.jar

ps -ef | grep spring

可以看到jar包已运行



## 19.3 Using the Maven plugin

Spring Boot Maven插件包含一个可用于快速编译和运行应用程序的运行目标。 应用程序以分解形式运行，就像在IDE中一样。

mvn spring-boot:run

如果还想使用一些的操作系统环境变量可以如下操作：

export MAVEN\_OPTS=-Xmx1024m -XX:MaxPermSize=128M

## 19.4 Using the Gradle plugin

Spring Boot Gradle插件也包含一个bootRun任务，可用于以分解形式运行您的应用程序。 只要导入spring-boot-gradle-plugin，都会添加bootRun任务：

gradle bootRun

如果还想使用一些的操作系统环境变量可以如下操作：

export JAVA\_OPTS=-Xmx1024m -XX:MaxPermSize=128M

## 19.5 Hot swapping

由于Spring Boot应用程序只是普通的Java应用程序，所以JVM可以提供开箱即用热插拔技术。 JVM热交换在某种程度上受限于它可以替换的字节码，为了获得更完整的解决方案，可以使用JRebel或Spring Loaded项目。 spring-boot-devtools模块包括了对快速应用程序重新启动的支持。

## 20. Developer tools

Spring Boot包含一组额外的工具，可以使应用程序开发体验更愉快。 spring-boot-devtools模块可以包含在任何项目中以节省开发时间。 要包含devtools支持，只需将模块依赖关系添加到您的版本：

**Maven.**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

**Gradle.**

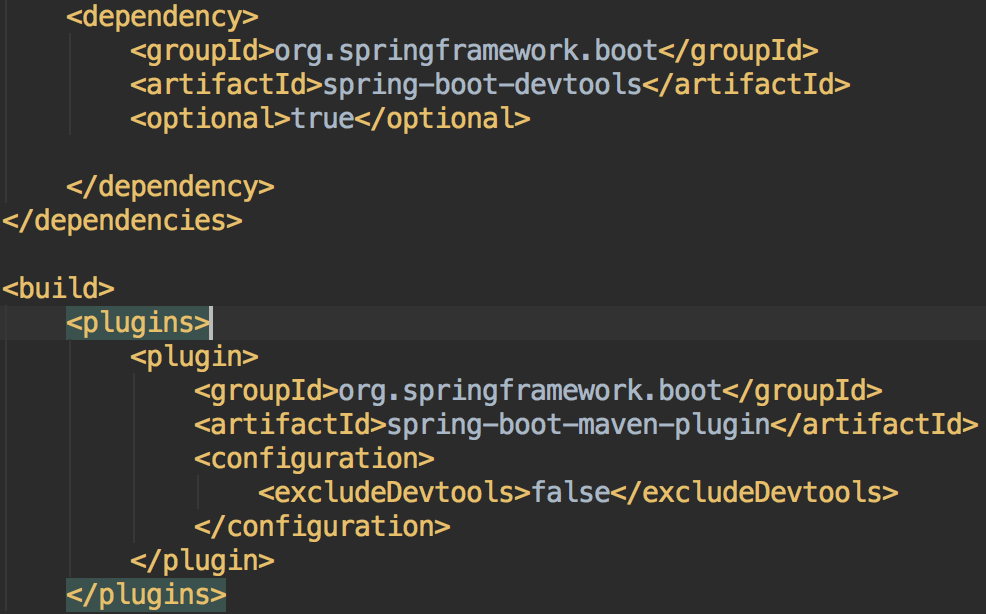
dependencies {

compile("org.springframework.boot:spring-boot-devtools")

}

运行完整打包的应用程序时，开发者工具会自动禁用。如果您的应用程序是使用java -jar启动的，或者如果它是使用特殊的类加载器启动的，那么它就被认为是“生产应用程序”。 将开发者工具依赖关系标记为可选项是一种最佳实践，可以防止devtools通过项目传递到其他模块。 Gradle不支持可选的依赖关系，因此您可能需要查看一下propdeps插件。

如果您想使用某些远程devtools功能，则需要禁用excludeDevtools构建属性。 该属性Maven和Gradle插件都支持。



## 20.2 Automatic restart

使用spring-boot-devtools的应用程序将在类路径上的文件发生更改时自动重启。 在IDE中工作时，这是一个非常有用的功能，因为它为代码更改提供了一个非常快速的反馈。 默认情况下，将监视指向文件夹的类路径中的任何更改。 请注意，某些资源（如静态资产和视图模板）不需要重新启动应用程序。

由于DevTools监视类路径资源，触发重启的唯一方法是更新类路径。 导致类路径更新的方式取决于您使用的IDE。 在Eclipse中，保存修改后的文件将导致类路径更新并触发重启。 在IntelliJ IDEA中，构建项目（Build - > Make Project）将具有相同的效果。

DevTools在重新启动期间依靠应用程序上下文的shutdown钩子发挥作用。 如果您禁用了shutdown钩子（SpringApplication.setRegisterShutdownHook（false）），它将无法正常工作。

Spring Boot提供的重启技术通过使用两个类加载器来工作。 没有变更的类（例如来自第三方jar的类）被加载到基类加载器中。 您正在开发的类将加载到重启类加载器中。 当应用程序重新启动时，重启类加载器将被丢弃，并创建一个新的。 这种方法意味着应用程序重新启动通常比“冷启动”快得多，因为基类加载器已经可用并且已经加载了一些类。

如果您发现重新启动对于您的应用程序来说不够快，或者遇到类加载问题，则可以考虑从ZeroTurnaround中寻找重新加载技术，例如JRebel。 这些技术通过重写被加载的类，使他们更容易重新加载。 Spring Loaded也是一种选择方式，但是它支持的框架不多，并且没有商业支持。

### 20.2.1 Excluding resources

某些资源不一定需要在更改时触发重新启动。 例如，Thymeleaf模板可以就地编辑。 默认情况下，更改/ META-INF / maven，/ META-INF / resources，/ resources，/ static，/ public或/ templates中的资源不会触发重新启动，但会触发实时重新加载。 如果你想自定义排除一些资源，你可以使用spring.devtools.restart.exclude属性。 例如，要仅排除/ static和/ public，您可以设置以下内容：

spring.devtools.restart.exclude=static/\*\*,public/\*\*

如果要保留这些默认值并添加其他排除项，请改为使用spring.devtools.restart.additional-exclude属性。

### 20.2.2 Watching additional paths

您可能希望在更改不在类路径中的文件时重新启动或重新加载应用程序。 为此，请使用spring.devtools.restart.additional-paths属性来配置其他路径以监视更改。 您可以使用上述的spring.devtools.restart.exclude属性来控制额外路径下的更改是否会触发完全重新启动或仅实时重新加载。

### 20.2.3 Disabling restart

如果您不想使用重新启动功能，则可以使用spring.devtools.restart.enabled属性将其禁用。 在大多数情况下，你可以在你的application.properties中设置它（这将仍然初始化restart classloader，但它不会监视文件的变化）。

如果您需要完全禁用重新启动支持，例如，比如它不适用于特定的库，则需要在调用SpringApplication.run（...）之前设置System属性。 例如：

**public** **static** **void** main(String[] args) {

System.setProperty("spring.devtools.restart.enabled", "false");

SpringApplication.run(MyApp.**class**, args);

}

如果您使用编译器的时候频繁的编译更改文件的文件，你可能只希望在特定时间触发重新启动。 要做到这一点，你可以使用“触发文件”，这是一个特殊的文件，当你想要实际触发重新启动检查时必须修改这个文件。更改文件会触发检查，只有Devtools检测到必须执行某些操作时才会重新启动。 触发文件可以手动更新，也可以通过IDE插件进行更新。

要使用触发器文件，请使用spring.devtools.restart.trigger-file属性。

### 20.2.5 Customizing the restart classloader

重新启动功能是通过使用两个类加载器来实现的。 对于大多数应用程序来说，这种方法运行良好，但有时候会导致类加载问题。

默认情况下，IDE中的任何打开的项目都将使用重启类加载器加载，任何常规的\*.jar文件都将使用基本类加载器加载。 如果您使用多模块项目，而不是将每个模块导入IDE，则可能需要自定义项目。 要做到这一点，你可以创建一个META-INF / spring-devtools.properties文件。

spring-devtools.properties文件可以包含restart.exclude.和restart.include.前缀属性。 include元素是应该被拉入到“重启”类加载器中的项目，exclude元素是应该被下推到“基本”类加载器中的项目。 该属性的值是一个作用于类路径的正则表达式。

restart.exclude.companycommonlibs=/mycorp-common-[\\w-]+\.jar

restart.include.projectcommon=/mycorp-myproj-[\\w-]+\.jar

### 20.2.6 Known limitations

对于使用标准ObjectInputStream进行反序列化的对象，重新启动功无法正常工作。如果您需要反序列化数据，则可将Spring的ConfigurableObjectInputStream与Thread.currentThread（）.getContextClassLoader（）一起使用。

spring-boot-devtools模块包含一个嵌入式LiveReload服务器，当资源发生变化时，可用于触发浏览器刷新。 LiveReload浏览器扩展可从livereload.com的Chrome，Firefox和Safari免费获得。

如果您不想在应用程序运行时启动LiveReload服务器，则可以将spring.devtools.livereload.enabled属性设置为false。

## 27. Developing web applications

Spring Boot非常适合Web应用程序的开发。 您可以使用嵌入式Tomcat，Jetty或Undertow轻松创建自包含的HTTP服务器。 大多数Web应用程序将使用spring-boot-starter-web模块来快速启动和运行。

## 27.1 The ‘Spring Web MVC framework’

Spring Web MVC框架（通常简称为“Spring MVC”）是一个丰富的“模型视图控制器”Web框架。 Spring MVC允许您创建特殊的@Controller或@RestController bean来处理传入的HTTP请求。 您的控制器中的方法使用@RequestMapping注释映射到HTTP。

这里是一个典型的例子@RestController来提供JSON数据：

*@RestController*

*@RequestMapping(value="/users")*

**public** **class** MyRestController {

*@RequestMapping(value="/{user}", method=RequestMethod.GET)*

**public** User getUser(*@PathVariable* Long user) {

*// ...*

}

*@RequestMapping(value="/{user}/customers", method=RequestMethod.GET)*

List<Customer> getUserCustomers(*@PathVariable* Long user) {

*// ...*

}

*@RequestMapping(value="/{user}", method=RequestMethod.DELETE)*

**public** User deleteUser(*@PathVariable* Long user) {

*// ...*

}

}

### 27.1.1 Spring MVC auto-configuration

Spring Boot为Spring MVC提供了自动配置，可以与大多数应用程序配合使用。自动配置在Spring的默认设置之上添加以下功能：

·包含ContentNegotiatingViewResolver和BeanNameViewResolver bean。

·支持静态资源，包括对WebJars的支持（见下文）。

·Converter，GenericConverter，Formatter beans的自动注册。

·支持HttpMessageConverters（见下文）。

·自动注册MessageCodesResolver（见下文）。

·静态index.html支持。

·自定义的Favicon支持（见下文）。

·自动使用ConfigurableWebBindingInitializer（见下文）。

如果你想保持Spring Boot的MVC特性，并且你还想添加额外的MVC配置（拦截器，格式化器，视图控制器等），你可以添加你自己的@Configuration类型的WebMvcConfigurerAdapter。 如果您希望提供RequestMappingHandlerMapping，RequestMappingHandlerAdapter或ExceptionHandlerExceptionResolver的自定义实例，则可以声明提供此类组件的WebMvcRegistrationsAdapter实例。

如果你想完全控制Spring MVC，你可以添加你自己的用@EnableWebMvc注解的@Configuration。

### 27.1.2 HttpMessageConverters

Spring MVC使用HttpMessageConverter接口来转换HTTP请求和响应。 默认包括拆箱即用的功能，例如对象可以自动转换为JSON（使用Jackson库）或XML（如果Jackson XML可以使用则使用Jackson XML扩展，否则使用JAXB）。 字符串默认使用UTF-8编码。如果您需要添加或自定义转换器，可以使用Spring Boot的HttpMessageConverters类

**import** org.springframework.boot.autoconfigure.web.HttpMessageConverters;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.http.converter.\*;

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

**public** HttpMessageConverters customConverters() {

HttpMessageConverter<?> additional = ...

HttpMessageConverter<?> another = ...

**return** **new** HttpMessageConverters(additional, another);

}

}

### 27.1.3 Custom JSON Serializers and Deserializers自定义JSON序列化器和反序列化器

如果您使用Jackson来序列化和反序列化JSON数据，则可能需要编写自己的JsonSerializer和JsonDeserializer类。 自定义序列化器通常通过模块向Jackson注册，但Spring Boot提供了一个替代的@JsonComponent注释，这使得更容易直接注册Spring Bean。

**import** java.io.\*;

**import** com.fasterxml.jackson.core.\*;

**import** com.fasterxml.jackson.databind.\*;

**import** org.springframework.boot.jackson.\*;

*@JsonComponent*

**public** **class** Example {

**public** **static** **class** Serializer **extends** JsonSerializer<SomeObject> {

*// ...*

}

**public** **static** **class** Deserializer **extends** JsonDeserializer<SomeObject> {

*// ...*

}

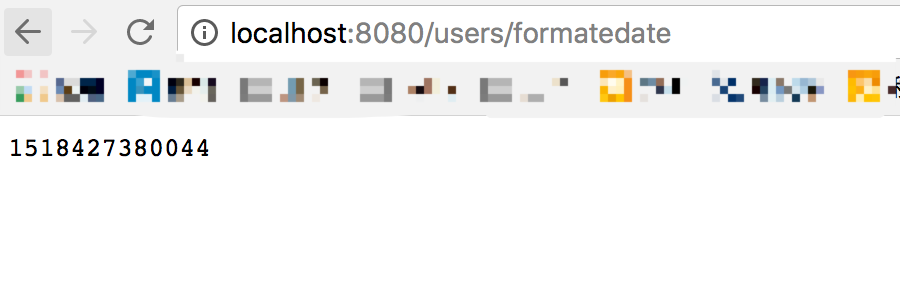
}

ApplicationContext中的所有@JsonComponent bean将被自动注册到Jackson，并且由于@JsonComponent使用@Component进行元注释，所以通常的组件扫描规则适用。

Spring Boot还提供了JsonObjectSerializer和JsonObjectDeserializer基类，在序列化对象时为标准Jackson版本提供了有用的替代方法。

假设我们有个controller可以获取当前日期

@RequestMapping("formatedate")  
@JsonSerialize  
public TestDate getFormatDate() {  
 TestDate testDate = new TestDate();  
 testDate.setCreatedate(new Date());  
 return testDate;  
}

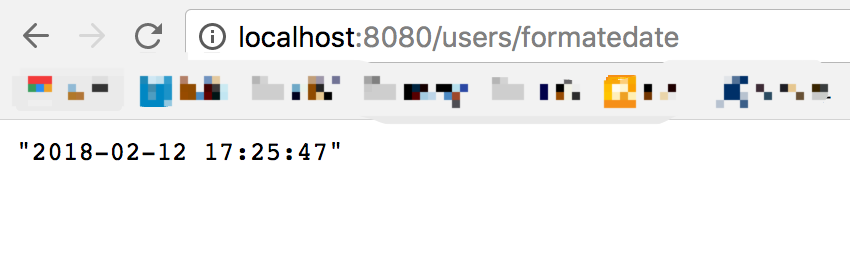


如果不加任何日期序列化工具，将会返回当前日期对应的毫秒数。

如果我们自定义一个日期序列化工具，实现日期的格式化输出：

@JsonComponent  
public class DateSerializer {  
  
 public static class Serializer extends JsonSerializer<Date> {  
 @Override  
 public void serialize(Date date, JsonGenerator jsonGenerator, SerializerProvider serializers) throws IOException, JsonProcessingException {  
 SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 jsonGenerator.writeString(simpleDateFormat.format(date));  
 }  
 }  
  
 public static class Deserializer extends JsonDeserializer<Date> {  
 @Override  
 public Date deserialize(JsonParser jsonParser, DeserializationContext ctxt) throws IOException, JsonProcessingException {  
 String date = jsonParser.getText();  
 SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 try {  
 return simpleDateFormat.parse(date);  
 } catch (ParseException e) {  
 e.printStackTrace();  
 }  
 return new Date();  
 }  
 }  
  
}

观察输出，可以发现返回我们需要的日期格式



同样反序列也也是一样的道理：

@RequestMapping(value = "str2date")  
public void getFormatDate(@RequestBody DateVO dateVO) {  
 System.*out*.println(dateVO);  
}

@Data  
public class DateVO {  
 private Date date;  
}

入参我们用一个字符串时间：{"date":"2018-01-01 00:00:00"}

我们自定的反序列化工具可以实现将字符串转换成Date对象

DateVO(date=Mon Jan 01 00:00:00 CST 2018)

**27.1.4 MessageCodesResolver**

Spring MVC有一个策略，用于从绑定的errors产生用来渲染错误信息的错误码：MessageCodesResolver。如果设置spring.mvc.message-codes-resolver.format属性为PREFIX\_ERROR\_CODE或POSTFIX\_ERROR\_CODE（具体查看DefaultMessageCodesResolver.Format枚举值），Spring Boot会为你创建一个MessageCodesResolver。

### 27.1.5 Static Content（http://www.webjars.org/）

默认情况下，Spring Boot将从类路径中的/ static（或/ public或/ resources或/ META-INF / resources）目录或从ServletContext的根目录中提供静态内容。 它使用Spring MVC中的ResourceHttpRequestHandler，因此可以通过添加自己的WebMvcConfigurerAdapter并重写addResourceHandlers方法来修改该行为。

默认情况下，资源被映射到/ \*\*，但是你可以通过spring.mvc.static-path-pattern来调整。 例如，将所有资源重定位到/ resources / \*\*可以实现如下：

spring.mvc.static-path-pattern=/resources/\*\*

您还可以使用spring.resources.static-locations（使用目录位置列表替换默认值）自定义静态资源位置。 如果您这样做，默认的欢迎页面检测将切换到您的自定义位置。 因此，如果您的任何位置在启动时都有index.html，它将成为应用程序的主页。

除了上面的“标准”静态资源位置之外，Webjars是一个特殊情况。 如果资源以Webjars格式打包，并且在/ webjars / \*\*目录下，将会被当成jar包使用。

如果您的应用程序将被打包为jar，请不要使用src / main / webapp目录。 虽然这个目录是一个通用的标准，但它只能用于war包，如果你要生成一个jar包，该目录会被大多数构建工具默默地忽略。

### 27.1.6 Custom Favicon

Spring Boot会在配置的静态内容位置和类路径的根目录（按此顺序）中查找favicon.ico。 如果这样的文件存在，它会自动用作应用程序的图标。

### 27.1.7 ConfigurableWebBindingInitializer

Spring MVC使用WebBindingInitializer为特定请求初始化WebDataBinder。 如果您创建自己的ConfigurableWebBindingInitializer @Bean，Spring Boot将自动配置Spring MVC以使用它。

### 27.1.8 Template engines

除了REST Web服务，您还可以使用Spring MVC为动态HTML内容提供服务。 Spring MVC支持各种模板技术，包括Thymeleaf，FreeMarker和JSP。 许多其他模板引擎也提供了自己的Spring MVC集成。

Spring Boot包含以下模板引擎的自动配置支持：

* [FreeMarker](http://freemarker.org/docs/)
* [Groovy](http://docs.groovy-lang.org/docs/next/html/documentation/template-engines.html#_the_markuptemplateengine)
* [Thymeleaf](http://www.thymeleaf.org/)
* [Mustache](https://mustache.github.io/)

如果可能的话，JSP应该被避免，在嵌入的servlet容器中使用它们有几个已知的限制。

当您使用默认配置的模板引擎之一时，您的模板将自动从src / main / resources / templates中选取。

IntelliJ IDEA根据你不同的运行应用程序的方式排列类路径。 通过主要方法在IDE中运行应用程序将导致与使用Maven或Gradle或从其打包的jar运行应用程序时不同的顺序。 这可能会导致Spring Boot无法在类路径中找到模板。 如果您受到这个问题的影响，您可以在IDE中重新排序类路径，以便首先放置模块的类和资源。 或者，您可以配置模板前缀以搜索类路径中的每个模板目录：classpath \*：/ templates /。

举个栗子：

我现在想返回freemaker模本，需要做的准备工作如下：

·修改application.properties

## Freemarker 配置  
## 文件配置路径  
spring.freemarker.template-loader-path=classpath:/templates/  
spring.freemarker.cache=false  
spring.freemarker.charset=UTF-8  
spring.freemarker.check-template-location=true  
spring.freemarker.content-type=text/html  
spring.freemarker.expose-request-attributes=true  
spring.freemarker.expose-session-attributes=true  
spring.freemarker.request-context-attribute=request  
spring.freemarker.suffix=.ftl

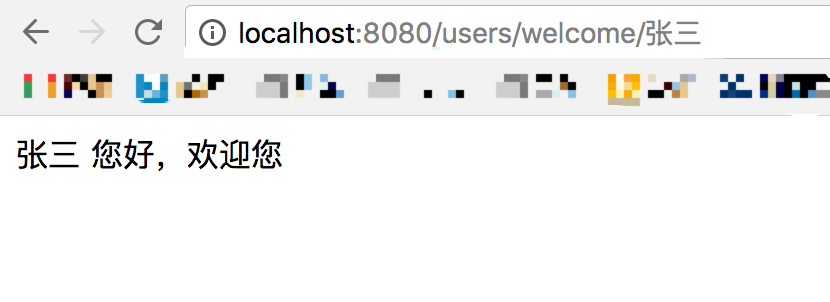
·新建controller返回视图名称

@RequestMapping("/welcome/{name}")  
public ModelAndView demo(@PathVariable String name) {  
 Map<String, Object> objectMap = new HashMap<>(1);  
 objectMap.put("name", name);  
 return new ModelAndView("welcome");  
}

·新增/templates/welcome.ftl这个freemaker模版

<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>Insert title here</title>  
</head>  
<body>  
 **${**name**}** 您好，欢迎您  
<br/>  
</body>  
</html>

执行以下<http://localhost:8080/users/welcome/>张三，发现返回如下页面：



更多关于freemaker的配置可以参考如下：

spring.freemarker.allow-request-override=false # Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.

spring.freemarker.allow-session-override=false # Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.

spring.freemarker.cache=false # Enable template caching.

spring.freemarker.charset=UTF-8 # Template encoding.

spring.freemarker.check-template-location=true # Check that the templates location exists.

spring.freemarker.content-type=text/html # Content-Type value.

spring.freemarker.enabled=true # Enable MVC view resolution for this technology.

spring.freemarker.expose-request-attributes=false # Set whether all request attributes should be added to the model prior to merging with the template.

spring.freemarker.expose-session-attributes=false # Set whether all HttpSession attributes should be added to the model prior to merging with the template.

spring.freemarker.expose-spring-macro-helpers=true # Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".

spring.freemarker.prefer-file-system-access=true # Prefer file system access for template loading. File system access enables hot detection of template changes.

spring.freemarker.prefix= # Prefix that gets prepended to view names when building a URL.

spring.freemarker.request-context-attribute= # Name of the RequestContext attribute for all views.

spring.freemarker.settings.\*= # Well-known FreeMarker keys which will be passed to FreeMarker's Configuration.

spring.freemarker.suffix= # Suffix that gets appended to view names when building a URL.

spring.freemarker.template-loader-path=classpath:/templates/ # Comma-separated list of template paths.

spring.freemarker.view-names= # White list of view names that can be resolved.

### 27.1.9 Error Handling

Spring Boot默认提供了一个/error映射，以合理的方式处理所有的错误，并在servlet容器中被注册为一个“全局”错误页面。 对于机器客户端，它将产生一个JSON响应，包含错误的详细信息，HTTP状态和异常消息。 对于浏览器客户端，有一个'whitelabel'错误视图，它以HTML格式呈现相同的数据（自定义它只是添加一个解决'错误'的视图）。 要完全替换默认行为，可以实现ErrorController并注册该类型的bean定义，或者简单地添加一个ErrorAttributes类型的bean来使用现有的机制，但是替换内容

BasicErrorController可以用作自定义ErrorController的基类。 如果您想为新的内容类型添加处理程序（默认情况下专门处理text / html并提供其他所有内容的回退），这一点尤其有用。 要做到这一点，只需扩展BasicErrorController并添加一个带有@RequestMapping属性的公共方法，并创建一个新类型的bean。

您还可以定义@ControllerAdvice来定制JSON文档以返回特定的控制器和/或异常类型.

*@ControllerAdvice(basePackageClasses = FooController.class)*

**public** **class** FooControllerAdvice **extends** ResponseEntityExceptionHandler {

*@ExceptionHandler(YourException.class)*

*@ResponseBody*

ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {

HttpStatus status = getStatus(request);

**return** **new** ResponseEntity<>(**new** CustomErrorType(status.value(), ex.getMessage()), status);

}

**private** HttpStatus getStatus(HttpServletRequest request) {

Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status\_code");

**if** (statusCode == null) {

**return** HttpStatus.INTERNAL\_SERVER\_ERROR;

}

**return** HttpStatus.valueOf(statusCode);

}

}

在上面的例子中，如果由与FooController相同的包中定义的控制器抛出YourException，则将使用CustomErrorType POJO的json表示而不是ErrorAttributes表示形式。

举个栗子：

@ControllerAdvice(basePackageClasses = MyRestController.class)  
public class CommonControllerAdvice extends ResponseEntityExceptionHandler {  
  
 @ExceptionHandler(Exception.class)  
 @ResponseBody  
 ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {  
 HttpStatus status = getStatus(request);  
 return new ResponseEntity<>(new CustomErrorType(status.value(), ex.getMessage()), status);  
 }  
  
 private HttpStatus getStatus(HttpServletRequest request) {  
 Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status\_code");  
 if (statusCode == null) {  
 return HttpStatus.*INTERNAL\_SERVER\_ERROR*;  
 }  
 return HttpStatus.*valueOf*(statusCode);  
 }  
  
}

@RequestMapping("exception")  
public void exceptionDemo(){  
 throw new RuntimeException("An uncaught Exception happened");  
}

可以看到控制台输出：

java.lang.RuntimeException: An uncaught Exception happened

at com.example.springboot.controller.MyRestController.exceptionDemo(MyRestController.java:55) ~[classes/:na]

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0\_161]

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0\_161]

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0\_161]

at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0\_161]

at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:205) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:133) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:97) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:827) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:738) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:85) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:967) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:901) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:970) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:861) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at javax.servlet.http.HttpServlet.service(HttpServlet.java:635) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:846) ~[spring-webmvc-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at javax.servlet.http.HttpServlet.service(HttpServlet.java:742) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52) ~[tomcat-embed-websocket-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:99) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.springframework.web.filter.HttpPutFormContentFilter.doFilterInternal(HttpPutFormContentFilter.java:108) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal(HiddenHttpMethodFilter.java:81) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:197) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107) ~[spring-web-4.3.14.RELEASE.jar:4.3.14.RELEASE]

at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199) ~[tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:504) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:140) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:790) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1459) [tomcat-embed-core-8.5.27.jar:8.5.27]

at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49) [tomcat-embed-core-8.5.27.jar:8.5.27]

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149) [na:1.8.0\_161]

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) [na:1.8.0\_161]

at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61) [tomcat-embed-core-8.5.27.jar:8.5.27]

at java.lang.Thread.run(Thread.java:748) [na:1.8.0\_161]

#### Custom error pages

如果要为给定的状态代码显示自定义的HTML错误页面，请将文件添加到/ error文件夹。 错误页面可以是静态HTML（即添加在任何静态资源文件夹下），也可以使用模板构建。 该文件的名称应该是确切的状态码或一系列掩码。

例如，要将404映射到静态HTML文件，您的文件夹结构如下所示：

src/

+- main/

+- java/

| + <source code>

+- resources/

+- public/

+- error/

| +- 404.html

+- <other public assets>

要使用FreeMarker模板来映射所有的5xx错误，你会有这样的结构：

src/

+- main/

+- java/

| + <source code>

+- resources/

+- templates/

+- error/

| +- 5xx.ftl

+- <other templates>

对于更复杂的映射，您还可以添加实现ErrorViewResolver接口的bean。

**public** **class** MyErrorViewResolver **implements** ErrorViewResolver {

*@Override*

**public** ModelAndView resolveErrorView(HttpServletRequest request,

HttpStatus status, Map<String, Object> model) {

*// Use the request or status to optionally return a ModelAndView*

**return** ...

}

}

您还可以使用常规的Spring MVC功能，如@ExceptionHandler方法和@ControllerAdvice。 ErrorController将会接收任何未处理的异常。

#### Mapping error pages outside of Spring MVC

对于不使用Spring MVC的应用程序，可以使用ErrorPageRegistrar接口直接注册ErrorPages。 这个抽象直接与底层的嵌入式servlet容器一起工作，即使你没有Spring MVC的DispatcherServlet也可以工作。

*@Bean*

**public** ErrorPageRegistrar errorPageRegistrar(){

**return** **new** MyErrorPageRegistrar();

}

*// ...*

**private** **static** **class** MyErrorPageRegistrar **implements** ErrorPageRegistrar {

*@Override*

**public** **void** registerErrorPages(ErrorPageRegistry registry) {

registry.addErrorPages(**new** ErrorPage(HttpStatus.BAD\_REQUEST, "/400"));

}

}

注： 如果你注册了一个错误页面，并且这个错误页面将被一个过滤器处理的路径（例如，像一些非Spring web框架，比如Jersey和Wicket），那么这个过滤器必须被明确地注册为一个错误调度器。

*@Bean*

**public** FilterRegistrationBean myFilter() {

FilterRegistrationBean registration = **new** FilterRegistrationBean();

registration.setFilter(**new** MyFilter());

...

registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.**class**));

**return** registration;

}

（默认FilterRegistrationBean不包括ERROR调度程序类型）。

#### Error Handling on WebSphere Application Server

当部署到servlet容器时，Spring Boot使用其错误页面过滤器将具有错误状态的请求转发到适当的错误页面。 如果响应尚未提交，则只能将请求转发到正确的错误页面。 默认情况下，WebSphere Application Server 8.0和更高版本在成功完成servlet的服务方法后提交响应。 您应该通过将com.ibm.ws.webcontainer.invokeFlushAfterService设置为false来禁用此行为

### 27.1.10 Spring HATEOAS

如果您正在开发使用超媒体的RESTful API，Spring Boot为Spring HATEOAS提供了自动配置，可以与大多数应用程序配合使用。 自动配置取代了使用@EnableHypermediaSupport并注册大量bean的需求，以便构建基于超媒体的应用程序，包括LinkDiscoverers（用于客户端支持）以及配置为将响应正确编组为所需表示的ObjectMapper。 ObjectMapper将根据spring.jackson.\*属性或Jackson2ObjectMapperBuilder bean（如果存在）进行自定义。

您可以使用@EnableHypermediaSupport来控制Spring HATEOAS的配置。 请注意，这将禁用上述的ObjectMapper自定义。

### 27.1.11 CORS support

跨源资源共享（CORS）是大多数浏览器都支持的W3C规范，允许您以灵活的方式指定授权哪种跨域请求，而不是使用像IFRAME或JSONP这样一些不太安全和功能较弱的方法。

从4.2版开始，Spring MVC支持CORS开箱即用。 使用控制器方法在Spring Boot应用程序中使用@CrossOrigin标注的CORS配置不需要任何特定的配置。 可以通过向自定义addCorsMappings（CorsRegistry）方法注册WebMvcConfigurer bean来定义全局CORS配置：

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

**public** WebMvcConfigurer corsConfigurer() {

**return** **new** WebMvcConfigurerAdapter() {

*@Override*

**public** **void** addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*");

}

};

}

}

## 27.3 Embedded servlet container support

Spring Boot包含对嵌入式Tomcat，Jetty和Undertow服务器的支持。 大多数开发人员将简单地使用适当的“Starter”来获取完全配置的实例。 默认情况下，嵌入式服务器将侦听端口8080上的HTTP请求。

如果您选择在CentOS上使用Tomcat，请注意，默认情况下会使用临时目录来存储编译的JSP，文件上载等。当您的应用程序运行时，该目录可能会被tmpwatch删除，从而导致失败。 为了避免这种情况，您可能需要自定义您的tmpwatch配置，以便不删除tomcat.\*目录，或配置server.tomcat.basedir以便嵌入式Tomcat使用不同的位置。

### 27.3.1 Servlets, Filters, and listeners

当使用嵌入式servlet容器时，您可以通过使用Spring bean或通过扫描Servlet组件来注册Servlet规范中的Servlet，Filters和所有监听器（例如HttpSessionListener）。

#### Registering Servlets, Filters, and listeners as Spring beans

任何作为Spring bean的Servlet，Filter或Servlet \* Listener实例都将在嵌入容器中注册。 如果要在配置期间从application.properties中引用值，这可能特别方便。

默认情况下，如果上下文只包含一个Servlet，它将映射到/。 在多个Servlet bean的情况下，bean名称将被用作路径前缀。 过滤器将映射到/ \*。

如果基于约定的映射不够灵活，则可以使用ServletRegistrationBean，FilterRegistrationBean和ServletListenerRegistrationBean类进行完全控制。

### 27.3.2 Servlet Context Initialization

嵌入式servlet容器不会直接执行Servlet 3.0+ javax.servlet.ServletContainerInitializer接口或Spring的org.springframework.web.WebApplicationInitializer接口。 这是故意这么设计的，旨在降低第三方库在war包中运行而破坏Spring Boot应用程序的风险。

如果您需要在Spring Boot应用程序中执行servlet上下文初始化，则应该注册一个实现org.springframework.boot.context.embedded.ServletContextInitializer接口的bean。 单个onStartup方法提供对ServletContext的访问，并且可以根据需要轻松用作现有WebApplicationInitializer的适配器。

#### Scanning for Servlets, Filters, and listeners

在使用嵌入式容器时，可以使用@ServletComponentScan启用@WebServlet，@ WebFilter和@WebListener注释类的自动注册。

@ServletComponentScan在独立容器中将不起作用，而独立容器将使用容器的内置发现机制。

### 27.3.3 The EmbeddedWebApplicationContext

Spring Boot使用一种新型的ApplicationContext来实现嵌入式servlet容器支持。 EmbeddedWebApplicationContext是一种特殊类型的WebApplicationContext，通过搜索单个EmbeddedServletContainerFactory bean来引导自身。 通常，TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory将自动配置。

您通常不需要知道这些实现类。 大多数应用程序将自动配置，并且将代表您创建相应的ApplicationContext和EmbeddedServletContainerFactory。

### 27.3.4 Customizing embedded servlet containers

通用的servlet容器设置可以使用Spring Environment属性进行配置。 通常你会在你的application.properties文件中定义属性。

·网络设置：监听HTTP请求的端口（server.port），绑定到server.address的接口地址等。

·会话设置：会话是否持久（server.session.persistence），会话超时（server.session.timeout），会话数据的位置（server.session.store-dir）和会话cookie配置（server.session.cookie.\*）。

·错误管理：错误页面的位置（server.error.path）等

·SSL

·HTTP压缩

Spring Boot尽可能地尝试暴露常见设置，但这并非总是可行。 对于这些情况，专用名称空间提供了特定于服务器的定制（请参阅server.tomcat和server.undertow）。 例如，可以使用嵌入式servlet容器的特定功能来配置访问日志。

#### Programmatic customization

如果您需要以编程方式配置嵌入式Servlet容器，则可以注册一个实现了EmbeddedServletContainerCustomizer接口的Spring bean。 EmbeddedServletContainerCustomizer提供对ConfigurableEmbeddedServletContainer的访问，其中包含许多定制的setter方法。

**import** org.springframework.boot.context.embedded.\*;

**import** org.springframework.stereotype.Component;

*@Component*

**public** **class** CustomizationBean **implements** EmbeddedServletContainerCustomizer {

*@Override*

**public** **void** customize(ConfigurableEmbeddedServletContainer container) {

container.setPort(9000);

}

}

#### Customizing ConfigurableEmbeddedServletContainer directly

如果上述定制技术过于有限，则可以自己注册TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory bean。

*@Bean*

**public** EmbeddedServletContainerFactory servletContainer() {

TomcatEmbeddedServletContainerFactory factory = **new** TomcatEmbeddedServletContainerFactory();

factory.setPort(9000);

factory.setSessionTimeout(10, TimeUnit.MINUTES);

factory.addErrorPages(**new** ErrorPage(HttpStatus.NOT\_FOUND, "/notfound.html"));

**return** factory;

}

为许多配置选项提供安装程序。 如果你需要做一些更奇特的事情，还提供了一些受保护的方法“钩子”。 有关详细信息，请参阅源代码文档。

### 27.3.5 JSP limitations

·使用Tomcat时，如果使用war包，即可执行war包，那么jsp是可以运行的，并且也可以部署到标准容器（不限于Tomcat），也可以部署到Tomcat。 由于Tomcat中的硬编码文件模式，在可执行jar包里jsp无法工作。

·使用Jetty时，如果您使用war包，那么jsp是可以运行的，并且也可以部署到任何标准容器，它应该可以工作。

·Undertow不支持JSP。

·创建自定义error.jsp页面不会覆盖错误处理的默认视图，而是会使用自定义错误页面。

## 28. Security(//TODO)

## 29. Working with SQL databases(homework)

Spring框架提供了对关系型数据库的支持。从直接使用JdbcTempalte到完整的对象关系映射如Hibernate，Spring都提供了支持。Spring Data提供了额外的实用功能，从接口创建Repository的实现，并且用约定俗成的方式从我们的方法名称生成查询语句

29.1 Configure a DataSource

Java中的 javax.sql.DataSource接口，提供了连接数据库的标准方法，通常，DataSource使用一个URI和一些认证信息来和数据库做连接。

Embedded Database Support

通常使用内存中内嵌的数据库来开发应用是很方便的。显然，内存数据库不提供持久化的存储。需要在应用启动的时候保存数据，在应用停止的时候丢弃数据。

Spring Boot可以自动配置内嵌的H2，HSQL，Derby数据库。您不需要提供任何的URL连接，简单的包含一个您想使用的内嵌数据库的构建依赖就可以了。

如果您在单元测试中使用这一特性，您需要注意，同一个数据库将在您所有的测试用例中使用，不管你用了多少个应用上下文。如果您想让每一个上下文使用独立的内嵌数据库，您需要设置spring.datasource.generate-unique-name=true

典型的pom依赖如下：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<scope>runtime</scope>

</dependency>

需要注意的是，您需要依赖一个spring-jdbc以使得内嵌的数据库可以完成自动配置。在上面的例子中，是通过spring-boot-starter-data-jpa实现的。

不管是什么原因，你为内置的数据库配置了连接URL，必须要确保数据库的自动关闭被禁止了。如果您使用的是H2，您需要设置DB\_CLOSE\_ON\_EXIT=FALSE。如果您使用的是HSQLDB，您需要确保shutdown=true没有被使用。阻止数据库自动关闭可以让数据库关闭的时候由Spring Boot来控制。

Connection to a production database

生产数据库连接也可以通过使用DataSource连接池实现自动配置，下面是选择特定实现的算法：

·鉴于tomcat数据库连接池的性能和并发性，官方更加推荐tomcat。

·否则，如果HikariCP可用，将使用HikariCP

·如果tomcat数据库连接池和HikariCP都不可用，如果Commons DBCP可以使用，将使用Commons DBCP，但是官方并不推荐使用Commons DBCP，并且已经不在对齐提供支持。

·最后，如果以上都不满足，Commons DBCP2可用的话，就使用Commons DBCP2。

如果您使用spring-boot-starter-jdbc starter或者spring-boot-starter-data-jpa starter，您将自动得到tomcat-jdbc的依赖。

DataSource配置受外部配置项spring.datasource.\*的控制。举个栗子：

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

应该使用spring.datasource.url声明属性，否则Spring Boot将会尝试使用内置的数据库来自动配置连接。

通常不需要指明driver-class-name属性，因为Spring Boot可以从大部分的数据库连接的URL上推断出来。

查看[DataSourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.10.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceProperties.java)查看更多支持的选项。这些选项是标准选项。如果使用Tomcat connection pool，你可以自定义许多额外的选项。

# Number of ms to wait before throwing an exception if no connection is available.

spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the same time.

spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.

spring.datasource.tomcat.test-on-borrow=true

Connection to a JNDI DataSource

如果你要部署你的Spring Boot应用到应用服务器，你可能想使用应用服务器的特性配置和管理你的数据源，并且通过JNDI来访问数据源。那么spring.datasource.jndi-name、spring.datasource.username、spring.datasource.password属性来访问JNDI数据源。举个栗子，下面的application.properties配置项显示了如何访问一个JBoss数据源：

spring.datasource.jndi-name=java:jboss/datasources/customers

29.2 Using JdbcTemplate

Spring的JdbcTemplate和NamedParameterJdbcTemplate类是自动配置的，你可以在你自己的Bean中使用@Autowire来自动配置。

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Component;

@Component

public class MyBean {

private final JdbcTemplate jdbcTemplate;

@Autowired

public MyBean(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

// ...

}

29.3 JPA and ‘Spring Data’

Java持久化API是一种将对象映射到关系数据库的一种标准化技术。spring-boot-starter-data-jpa提供了以下的依赖：

·Hibernate——最流行JPA实现

·Spring Data JPA

·Spring ORMs——Spring框架核心ORM支持。

默认的，Spring Boot支持Hibernate 5.0.x。

Entity Classes

通常，JPA ‘Entity’类是声明在persistence.xml文件中的。但是如果您使用Spring Boot，那么此文件就不是必须的了。相反，您使用‘Entity Scanning’可以取而代之。默认的，在您的主要配置类（@EnableAutoConfiguration/ @SpringBootApplication）下的所有包都会被扫描到。

任何被@Entity, @Embeddable or @MappedSuperclass注解的类都将被作为实体类。一个典型的实体类可能是这样的：

package com.example.myapp.domain;

import java.io.Serializable;

import javax.persistence.\*;

@Entity

public class City implements Serializable {

@Id

@GeneratedValue

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false)

private String state;

// ... additional members, often include @OneToMany mappings

protected City() {

// no-args constructor required by JPA spec

// this one is protected since it shouldn't be used directly

}

public City(String name, String state) {

this.name = name;

this.country = country;

}

public String getName() {

return this.name;

}

public String getState() {

return this.state;

}

// ... etc

}

您可以使用@EntityScan注解来个性化定制实体类的扫描路径。

Spring Data JPA Repositories

Spring Data JPA repositories是你可以定义用来访问数据的接口。JPA从您的方法名自动发起查询。比如：CityRepository接口可能声明了一个findAllByState(String state)方法来查询某个洲的所有城市。

如果您想要一个更加复杂的查询，您可以用Spring Data的查询注解来注解您的方法。

Spring Data repositories通常继承自Repository or CrudRepository接口。如果您使用自动配置，Spring Data repositories将从您的主要配置类（@EnableAutoConfiguration或@SpringBootApplication类）所在的包向下扫描。

一个典型的Spring Data repository：

package com.example.myapp.domain;

import org.springframework.data.domain.\*;

import org.springframework.data.repository.\*;

public interface CityRepository extends Repository<City, Long> {

Page<City> findAll(Pageable pageable);

City findByNameAndCountryAllIgnoringCase(String name, String country);

}

Creating and dropping JPA databases

默认情况下，JPA数据库只有在您使用内嵌的数据库（H2, HSQL or Derby）才会自动创建。您可以明确的使用spring.jpa.\*属性来配置JPA。例如：创建或删除表，您可以在您的application.properties文件中加入如下配置：

spring.jpa.hibernate.ddl-auto=create-drop

默认情况下，DDL将在ApplicationContext运行的时候启动。

Open EntityManager in View

如果您在运行一个web应用，Spring Boot将默认注册OpenEntityManagerInViewInterceptor来使用Open EntityManager in View模式来达到web视图的懒加载。如果您不想使用这种配置，您可以在application.properties中配置spring.jpa.openin-view = false

29.4 Using H2’s web console

## 29.5 Using jOOQ

Java面向对象查询（jOOQ）是Data Geekery的一款流行产品，它可以从您的数据库中生成Java代码，并允许您通过流畅的API构建类型安全的SQL查询。 商业和开源版本都可以与Spring Boot一起使用。

为了使用jOOQ类型安全的查询，您需要从数据库模式生成Java类。您可以按照jOOQ用户手册中的说明进行操作。 如果您使用的是jooq-codegen-maven插件（并且您还使用spring-boot-starter-parent“parent POM”），则可以安全地省略插件的<version>标记。 您还可以使用Spring Boot定义的版本变量（例如h2.version）来声明插件的数据库依赖性。 这是一个例子：

<plugin>

<groupId>org.jooq</groupId>

<artifactId>jooq-codegen-maven</artifactId>

<executions>

...

</executions>

<dependencies>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>${h2.version}</version>

</dependency>

</dependencies>

<configuration>

<jdbc>

<driver>org.h2.Driver</driver>

<url>jdbc:h2:~/yourdatabase</url>

</jdbc>

<generator>

...

</generator>

</configuration>

</plugin>

Using DSLContext

jOOQ提供的流畅的API是通过org.jooq.DSLContext接口来初始化的。 Spring Boot会自动配置一个DSLContext作为Spring Bean并将其连接到您的应用程序DataSource。 要使用DSLContext，您只需@Autowire它。

然后，您可以使用DSLContext来构建您的查询：

public List<GregorianCalendar> authorsBornAfter1980() {

return this.create.selectFrom(AUTHOR)

.where(AUTHOR.DATE\_OF\_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))

.fetch(AUTHOR.DATE\_OF\_BIRTH);

}

定制jOOQ

您可以通过在application.properties中设置spring.jooq.sql-dialect来自定义jOOQ使用的SQL方言。 例如，要指定Postgres，您可以添加：

spring.jooq.sql-dialect=Postgres

更高级的自定义可以通过定义自己的@Bean定义来实现，这将在创建jOOQ配置时使用。 您可以为以下jOOQ类型定义bean：

• ConnectionProvider

• TransactionProvider

• RecordMapperProvider

• RecordListenerProvider

• ExecuteListenerProvider

• VisitListenerProvider

如果你想完全控制jOOQ配置，你也可以创建你自己的org.jooq.Configuration @Bean。

更多jOOQ的使用请参考：

·<https://www.jooq.org/>

·<https://www.jianshu.com/p/46164f9ba53c>

# 30. Working with NoSQL technologies

Spring Data提供的其他项目可帮助您访问各种NoSQL技术，包括MongoDB，Neo4J，Elasticsearch，Solr，Redis，Gemfire，Cassandra，Couchbase和LDAP。 Spring Boot为Redis，MongoDB，Neo4j，Elasticsearch，Solr Cassandra，Couchbase和LDAP提供自动配置; 你可以使用其他项目，但你需要自己配置它们。 请参阅projects.spring.io/spring-data上的相应参考文档。

## 30.1 Redis

Redis是一个缓存，消息代理和功能丰富的键值存储。 Spring Boot为Jedis客户端库和在它基础上的Spring Data Redis提供的抽象类提供了基本的自动配置。 有一个spring-boot-starter-data-redis'Starter'以方便的方式收集依赖关系。

Connecting to Redis

您可以像其他Spring Bean一样注入自动配置的RedisConnectionFactory，StringRedisTemplate或RedisTemplate实例。 默认情况下，实例将尝试使用localhost:6379连接到Redis服务器。

Demo地址：<https://github.com/PerseveranceForever/springboot_redis_demo.git>

以上demo是SpringBoot整合mybatis+redis。通过从mysql中读取数据，存入redis。

需要注意的是：

RedisTemplate 操作redis时发现存储在redis中的key值，前面还多出了许多类似\xac\xed\x00\x05t\x00\x04，这是因为spring-data-redis的RedisTemplate<K, V>模板类在操作redis时默认使用JdkSerializationRedisSerializer来进行序列化，如下：

if (defaultSerializer == null) {  
  
 defaultSerializer = new JdkSerializationRedisSerializer(  
 classLoader != null ? classLoader : this.getClass().getClassLoader());  
}

解决方案如下：

@Configuration  
@EnableCaching  
public class RedisCacheConfig {  
 @Bean  
 public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory){  
 RedisTemplate<String, String> redisTemplate = new RedisTemplate<String,String>();  
 redisTemplate.setConnectionFactory(factory);  
 // key序列化方式;（不然会出现乱码;）,但是如果方法上有Long等非String类型的话，会报类型转换错误；  
 // 所以在没有自己定义key生成策略的时候，以下这个代码建议不要这么写，可以不配置或者自己实现ObjectRedisSerializer  
 // 或者JdkSerializationRedisSerializer序列化方式;  
 RedisSerializer<String> redisSerializer = new StringRedisSerializer();  
 redisTemplate.setKeySerializer(redisSerializer);  
 redisTemplate.setHashKeySerializer(redisSerializer);  
 return redisTemplate;  
 }  
  
}

# 30.2 MongoDB

MongoDB是一个开源的NoSQL文档数据库，它使用类似JSON的模式而不是传统的基于表格的关系数据。 Spring Boot为MongoDB提供了一些便利，包括spring-boot-starter-data-mongodb。

**Connecting to a MongoDB database**

您可以将自动配置的org.springframework.data.mongodb.MongoDbFactory注入

访问Mongo数据库。 默认情况下，实例将尝试使用URL mongodb：// localhost / test连接到mongoDB服务器。

您可以设置spring.data.mongodb.uri属性来更改URL并配置其他设置，例如副本集：

spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo2.example.com:23456/test

另外，如果你使用Mongo 2.x，指定一个主机/端口就可以了。 例如，您可以在application.properties中声明以下内容：

spring.data.mongodb.host=mongoserver

spring.data.mongodb.port=27017

如果你使用的是Mongo 3.0 Java驱动程序，spring.data.mongodb.host和spring.data.mongodb.port是不支持的。这种情况下，必须使用spring.data.mongodb.uri来配置mongodb的连接。

如果未指定spring.data.mongodb.port，则使用默认值27017。 您可以简单地从上面的示例中删除这一行。

MongoTemplate

Spring Data Mongo提供了一个MongoTemplate类，它的设计与Spring的JdbcTemplate非常相似。 与JdbcTemplate一样，Spring Boot会自动配置一个bean，以便您简单地注入：

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.mongodb.core.MongoTemplate;

import org.springframework.stereotype.Component;

@Component

public class MyBean {

@Autowired

private MongoTemplate mongoTemplate;

// ...

}

### 30.2.3 Spring Data MongoDB repositories

Spring Data包含对MongoDB的存储库支持。 与前面讨论的JPA存储库一样，基本原则是查询是基于方法名称自动为您构建的。

实际上，Spring Data JPA和Spring Data MongoDB共享相同的通用基础结构; 所以你可以从之前的JPA例子中，假设City现在是一个Mongo数据类而不是JPA @Entity，它将以相同的方式工作。

您可以使用@EntityScan注解自定义文档扫描位置。

更多更详细的Spring Data MongoDB特性，包括它的对象映射技术，请参考 [reference documentation](https://projects.spring.io/spring-data-mongodb/)

Demo地址：

<https://github.com/PerseveranceForever/springboot_mongodb_demo.git>

## 30.3 Neo4j

## 30.4 Gemfire

## 30.5 Solr

Apache Solr是一个搜索引擎。 Spring Boot为Solr 5客户端库和Spring Data Solr提供的抽象类提供了基本的自动配置。 有一个spring-boot-starter-data-solr'Starter'用于以一种方便的方式收集依赖关系。

### 30.5.1 Connecting to Solr

您可以像其他任何Spring bean一样注入一个自动配置的SolrClient实例。 默认情况下，实例将尝试使用localhost:8983/solr连接到服务器：

*@Component*

**public** **class** MyBean {

*@Autowired*

**private** SolrClient solr;

*// ...*

}

如果你添加@Bean注解到自定义的类型为SolrClient的bean上，它将取代默认配置。

### 30.5.2 Spring Data Solr repositories

Spring Data包含Apache Solr的存储库支持。 与前面讨论的JPA存储库一样，基本原则是查询是基于方法名称自动为您构建的。

实际上，Spring Data JPA和Spring Data Solr共享相同的通用基础结构; 所以你可以从之前的JPA例子中，假设City现在是@SolrDocument类而不是JPA @Entity，它将以相同的方式工作。

更多关于Spring Data Solr的特性请参考：[reference documentation](https://projects.spring.io/spring-data-solr/).