第一部分 Spring基础

第一章 Spring起步

1.1 什么是Spring

Spring的核心是提供了一个容器（container），通常称为Spring应用上下文（Spring application context），它会创建和管理应用组件。这些组件也可以称为bean，会在Spring应用上下文中装配在一起，从而形成一个完整的应用程序。将bean装配在一起的行为是通过一种基于依赖注入（dependency injection，DI）的模式实现的。DI通常是通过构造起参数和属性访问方法来实现的。

Spring自动配置能够自动发现应用类路径下的组件并将它们创建成Spring应用上下文的bean。自动装配起源于所谓的自动配置（autowiring）和组件扫描（component scanning）。只有不能进行自动配置的时候，基于XML的方式和基于Java的配置才是必要的。

1.2 初始化Spring应用

1.2.1 使用Spring Tool Suite初始化Spring项目

1.2.2 检查Spring项目结构

应用源代码在`src/main/java`中，测试代码在`src/test/java`中，非Java的资源放到了`src/mian/resources`。项目结构中：

- mvnw和mvnw.cmd：这是Maven包装器（wrapper）脚本。借助这些脚本，即使你的机器上没有安装Maven，也可以构建项目。
- pom.xml：这是Maven构建规范。

- XxxApplication.java：这个是Spring主类，它会启动该项目。
- application.properties：这个文件起初是空的，但是它为我们提供了指定配置属性的地方。
- static：在这个文件夹下，可以存放任意为浏览器提供服务的静态内容（图片、样式表、JavaScript等），该文件夹初始为空。
- templates：这个文件夹中用来存放渲染内容到浏览器的模板文件。
- XxxApplicationTests.java：这是一个简单的测试类，它能保证Spring上下文可以成功加载。

探索构建规范

pom.xml中

`<packaging>`表示打包方式，默认为JAR。尽管WAR非常适合部署到传统的Java应用服务器上，但对于绝大多数云平台来说它们并不是最理想的选择。有些云平台也能够部署和运行WAR文件，但是所有的Java云平台都能够运行可执行的JAR文件。

`<parent>`表明此项目的父POM，父POM为Spring项目为Spring项目的一些库提供了依赖管理。

`<dependencies>`元素声明了项目的依赖，artifact ID上带有starter的依赖的特别之处在于它们本身并不包含库代码，而是传递性的拉去其他的库。

starter依赖有三个好处：

- 构建文件会显著减小并且易于管理，因为这样不必为每个依赖库都声明依赖。
- 能够根据它们所提供的功能来思考依赖，而不是根据库名称。如开发Web应用只需添加web starter就可以了。
- 不必再担心版本兼容问题。可以直接相信给定版本的Spring Boot，传递性引入的库都是兼容的，只需要担心使用哪个版本的Spring Boot就可以了。

构建规范还包含了一个Spring Boot插件。这个插件提供了一些重要功能：

- 它提供了一个Maven goal，允许我们使用Maven来运行应用。
- 他会确保依赖的所有库都会包含在可执行JAR文件中，并能够保证它们在运行时类路径下是可用的。
- 它会在JAR中生成一个manifest文件，将引导类声明为可执行JAR的主类。

引导应用

在XxxApplication中只有很少的代码，其中@SpringBootApplication注解明确表明这是一个Spring Boot应用，它是一个组合注解：

- @SpringBootConfiguration：将该类声明为配置类，这个注解实际上是@Configuration注解的特殊形式。
- @EnableAutoConfiguration：启用SpringBoot的自动配置。这个注解会告诉Spring自动配置它认为我们会用到的组件。
- @ComponentScan：启用组件扫描。这样我们就能通过像@Component、@Controller、@Service这样的注解声明其他类，Spring会自动发现它们并将它们注册为Spring应用的上下文中的组件。

XxxApplication中的main()方法会调用SpringApplication中静态的run()方法，后者会真正执行引导过程，也就是创建Spring的应用上下文。对于简单的应用程序来说，在引导类中配置一两个组件是非常方便的，但是对于绝大多数应用来说，最好还是要为没有实现自动配置的功能创建一个简单的配置类。

测试应用

XxxApplicationTests类中的内容不多，只有一个空的测试方法。即便如此，这个测试类还是会执行部必要的检查，确保Spring应用上下文能够成功加载。这个类带有@RunWith(SpringRunner.class)注解，@RunWith是JUnit的注解，它会提供一个测试运行器来指导JUnit如何运行测试。

测试运行器的其他名称

SpringRunner是SpringJUnit4ClassRunner的别名，是在Spring4.3中引入的，以便于移除对特定JUnit版本的关联。@SpringTest会告诉JUnit在启动测试的时候要添加上Spring Boot的功能。

1.3 编写Spring应用

1.3.1 处理Web请求

Spring自带了一个强大的Web框架，名为Spring MVC。Spring MVC的核心是控制器（即@Controller）的理念。@Controller的主要目的是将其修饰的类识别为一个组件。实际上有一些其他注解与@Controller有着类似的目的（包括@Component、@Service和@Repository），其作用是完全相同的。

为何使用Thymeleaf

Spring Boot建议使用Thymeleaf。

模板路径是“src/main/resources/templates/home.html”

1.3.2 定义视图

展现页面Logo的`<img>`标签，使用了Thymeleaf的th:src属性和@{...}表达式，以便于引用相对于上下文路径的图片。

1.3.3 测试控制器

@WebMvcTest注解是Spring Boot提供的一个特殊注解，它会让这个测试在Spring MVC应用上下文中执行。它会仿造Spring MVC的运行机制，测试类仿佛被注入了一个MockMvc能够让测试实现mockup。

1.3.4 构建应用并运行

日志最后会提示Tomcat已经在port(s):8080(http)启动的日志。

1.3.5 了解Spring Boot DevTools

DevTools为Spring开发人员提供了一些便利的开发期工具，其中包括：

- 代码变更后应用会自动重启；
- 当面向浏览器的资源（如模板、javaSpcript、样式表）等发生变化时，会自动刷新浏览器；
- 自动禁用模板缓存；
- 如果使用H2数据库的话，内置了H2控制台。

DevTools不是IDE在各种编辑器中都能很好的运行，它的目的仅仅是用于开发，所以能很智能地在生产环境中把自己禁用掉。

应用自动重启

当DevTools运行的时候，应用程序会被加载到JVM的两个独立的类容器中。其中一个类加载器会加载Java代码、属性文件以及项目中“src/main”下的几乎所有内容。另外一个加载器会加载依赖的库，这些库不太可能发生变化。当探测到变化时DevTools只会重新加载包含项目代码的类加载器，并重启Spring的应用上下文，这个策略非常精细，可以减少启动时间。但是如果我们在构建规范中添加、变更或移除依赖的时候，为了让变更生效，就需要重启应用。

浏览器自动刷新和禁用模板缓存

DevTools通过禁用模板缓存来实现模板变更时的自动刷新，DevTools在运行的时候，它会和你的应用程序一起，同时启动一个LiveReload服务器。LiveReload服务器在与LivaReload六拉起插件结合起来的时候，就能够在模板、图片、样式表。JavaScript等（几乎涵盖为浏览器提供服务的所有内容）发生变化时，自动刷新浏览器。

内置的H2控制台

如果你使用H2数据库进行开发，DevTools将会自动启用H2。

1.3.6 回顾一下

Spring应用的构建过程：

- 使用Spring Initializr创建初始项目结构；
- 编写控制器类处理针对主页的请求；
- 定义了一个视图模板来渲染主页；
- 编写了一个简单的测试类来验证工作符合预期。

构建规范，在pom.xml文件中，我们声明了对Web和Thymeleaf starter的依赖。这两项依赖会传递引入大量其他依赖，包括：

- Spring的MVC框架；
- 嵌入式的Tomcat；
- Thymeleaf和Thymeleaf布局方言；

还引入了Spring Boot的自动配置库。当启动应用时，Spring Boot的自动配置将会探测到这些库，并自动完成如下功能：

- 在Spring应用上下文中配置bean以启用Spring MVC；
- 在Spring应用上下文中配置嵌入式的Tomcat服务器；
- 配置Thymeleaf视图解析器，以便于Thymeleaf模板渲染Spring MVC视图。

1.4 俯瞰Spring风景线

1.4.1 Spring核心框架

Spring MVC还可以用来创建REST API，以生成非HTML的输出。在第二章中会详细介绍Spring MVC，在第六章会重新学习如何使用它来创建REST API。

第三章会学习Spring核心框架的数据持久化基础支持，尤其是基于模板的JDBC支持。

第三部分中将会学习Spring反应式编程模型，并在第十一章学习Spring WebFlux。

1.4.2 Spring Boot

除了starter依赖和自动配置，Spring Boot还提供了大量其他有用的特性：

- Actuator能够洞察应用运行时的颞部工作状况，包括指标、线程dump信息、应用的健康状况以及应用可用的环境属性；
- 灵活的环境属性规范；
- 在核心框架的测试辅助功能之上提供了对测试的额外支持。

除此之外Spring Boot还提供了一个基于Groovy脚本的编程模型，称为Spring Boot命令行接口（Command-Line Interface，CLI）。使用CLI我们可以将整个应用程序编写为Groovy脚本的集合，并通过命令行运行它们。

1.4.3 Spring Data

Spring Data提供数据持久化支持：将应用程序的数据repository定义为简单的Java接口，在定义驱动存储和检索数据的方法时使用一种命名约定即可。

此外，Spring Data能够处理多种不同类型的数据库，包括关系型数据库（JPA）、文档数据库（Mongo）、图数据库（Neo4j）等。

1.4.4 Spring Security

Spring Security解决了应用程序通用的安全性需求，包括身份验证、授权和API安全性。

1.4.5 Spring Integration和Spring Batch

Spring Integration和Spring Batch为基于Spring的应用程序提供了一些应用集成模式的实现。Spring Integration解决了实时集成问题，数据在可用时马上就会得到处理。Spring Batch解决的是批处理的集成问题，数据可以收集一段时间，直到某个触发器发出信号才会对数据进行处理。

1.4.6 Spring Cloud

解决微服务相关的一些挑战，Spring Cloud是使用Spring开发云原生应用程序的一组项目。

1.5 小结

- Spring旨在简化开发人员所面临的挑战，比如创建Web应用、处理数据库、保护应用程序以及实现微服务。
- Spring Boot构建在Spring之上，通过简化依赖管理、自动配置和运行时间洞察，使Spring更加易用。
- Spring应用可以使用Spring Initializr进行初始化。Spring Initializr使基于Web的应用，并且为大多数Java开发环境提供了原生支持。
- 在Spring应用上下文中，组件（通常称为bean）既可以使用Java或XML显示声明。也可以通过组件扫描发现，还可以使用Spring Boot自动配置功能实现自动化配置。

第二章 开发Web应用

2.1展现信息

在Spring Web应用中，获取和处理数据是控制器的任务，而将数据渲染到HTML中并在浏览器中展现则是视图的任务。为了支撑taco的创建页面，我们需要创建如下组件：

- 用来定义taco配料属性的领域类；
- 用来获取配料信息并将其传递至视图的Spring MVC控制器类。
- 用来在用户的浏览器中渲染配料列表的视图模板。

2.1.1 构建领域类

应用的领域指的是它所要解决的主要范围：也就是会影响到对应用理解的理念和概念。

2.1.2 创建控制器类

在Spring MVC框架中，控制器是重要的参与者。它们的主要职责是处理HTTP请求，要么将请求传递给视图以便于渲染HTML，要么直接将数据写入响应体（RESTful）。

处理GET请求

@GetMapping和@RequestMapping有完全相同的属性，可以在类级别上使用@RequestMapping，以便于指定基本路径。在每个处理器方法上使用更具体的@GetMapping、@PostMapping等注解。

2.1.3 设计视图

Spring提供了多种定义视图的方式，包括JavaServer Pages（JSP）、Thymeleaf、FreeMarker、Mustache和基于Croovy的模板。像Thymeleaf这样的视图库在设计时是与特定的Web框架解耦的。这样的话，它们无法感知Spring的模型抽象，因此无法与控制器放到Model中的数据协同工作。但是它们可以与Serverlet的request属性协作。所以在Spring将请求转移到视图之前，它会把模型数据复制到request属性中，Thymeleaf和其他的视图模板方案就能访问到它们了。

2.2 处理表单提交

表单提交的时候，浏览器会收集表单中的所有数据，并以HTTP请求的形式将其发送至服务器。

`return "redirect:/order/current";`是一个重定向视图，它表明在方法执行的最后，用户的浏览器将会重定向到相对路径“/order/current”。

2.3 校验表单输入

Spring支持Java的Bean校验API（Bean Validation API，也被称为JSR-303）。要在Spring MVC中应用校验，我们需要。

- 在要被校验的类上声明校验规则；
- 在控制器方法中声明要进行校验；
- 修改表单视图以展现校验错误。

2.3.1 声明校验规则

@NotNull：要求属性不为null。

@Size：校验字符串长度。

@NotBlank：确保不为空。

……

2.3.2 在表单绑定的时候执行校验

在Controller的方法参数前添加一个Java Bean Validation API的@Valid注解，校验时机是在它绑定完表单数据之后。调用方法之前。

2.3.3 展现校验错误

Thymeleaf提供了便捷访问Errors对象的方法，这就是借助fields及其th:errors属性。

2.4 使用视图控制器

如果一个控制器非常简单，不需要填充模型或处理输入，那么还有一种方式可以定义控制器。即视图控制器，也就是只将请求转发到视图而不做其他事情的控制器。

2.5 选择视图模板库

模板	Spring Boot starter依赖
FreeMarker	spring-boot-starter-freemarker
Groovy Templates	spring-boot-starter-groovy-templates
Java Server Pages	无(由Tomcat或Jetty提供)
Mustache	spring-boot-starter-mustache
Thymeleaf	spring-boot-starter-thymeleaf

通常来讲，只需要选择想要的视图模板库，将其作为依赖项添加到构建文件中，然后就可以在“/template”目录下（在基于Maven或Gradle构建的项目中，它会在“src/main/resource”目录下）编写模板了。Spring Boot会探测到你所选择的模板库，并自动配置为Spring MVC控制器生成视图所需的各种组件。

缓存模板

默认情况下，模板只有在第一次使用的时候解析一次，解析的结果会被后续的请求所使用。

模板	启用缓存的属性
FreeMarker	spring.freemarker.cache
Groovy Templates	spring.groovy.template.cache

模板	启用缓存的属性
Mustache	spring.mustache.cache
Thymeleaf	spring.thymeleaf.cache

我们可以在配置文件中将缓存属性设置为false来禁止所选模板的缓存。

2.6 小结

- Spring提供了一个强大的Web框架，名为Spring MVC，能够用来为Spring应用开发Web前端。
- Spring MVC是基于注解的，通过像@RequestMapping、@GetMapping和@PostMapping这样的注解来启用请求处理方式的声明。
- 大多数的请求处理方法最终会返回一个视图逻辑名称，比如Thymeleaf模板，请求会转发到这样的视图上（同事会带有任意的模型数据）。
- Spring MVC支持校验，这就是通过Java Bean Validation API和Validation API的实现（如Hibernate Validation）完成的。
- 对于没有模型数据和逻辑处理的HTTP GET请求，可以使用视图控制器。
- 除了Thymeleaf之外，Spring支持各种视图方案，包括FreeMarker、Groovy Templates和Mustache。

第三章 使用数据

3.1 使用JDBC读取和写入数据

处理关系型数据库时，最常见的是JDBC和JPA。Spring对JDBC的支持要归功于JdbcTemplate类。JdbcTemplate提供了一种特殊的方式，通过这种方式，开发人员再对关系型数据库执行SQL操作的时候能够避免使用JDBC时常见的繁文缛节和样板式代码。

3.1.1 调整领域对象以适应持久化

在将对象持久化到数据库的时候，通常最好有一个字段作为对象的唯一标识，除此之外，还会为每个对象添加一个字段来捕获它所创建的日期和时间。

3.1.2 使用JdbcTemplate

只需将Spring Boot的JDBC starter依赖添加到构建文件中就可以了。

定义JDBC repository

使用@Repository注解，添加该注解后，Spring的组件扫描就会自动发现它，并且会将其初始化为Spring应用上下文中的bean。创建bean的时候，Spring会通过@Autowire标注的构造器将JdbcTemplate注入进来。这个构造器将JdbcTemplate复制给一个实例变量，这个变量会被其他方法用来执行数据库查询和插入操作。

插入一行数据

JdbcTemplate的update()方法可以用来执行向数据库中写入或更新数据的查询语句。

3.1.3 定义模式和预加载数据

Spring应用启动时会执行根类路径（src/main/resources）下的schema.sql和data.sql文件。来进行数据库的结构定义和数据预加载。

3.1.4 插入数据

借助JdbcTemplate，我们有以下两种保存数据的方法：

- 直接使用update()方法；
- 使用SimpleJdbcInsert包装器类。

使用JdbcTemplate保存数据

保存数据时所使用的update()方法无法帮助我们得到所生成的ID，所以在这里我们需要一个不同的update()方法。这里的update()方法需要接收一个PrepareStatementCreator和一个KeyHolder。KeyHolder会为我们提供生成的ID。但是为了使用该方法，我们还需要创建一个PrepareStatementCreator。

使用SimpleJdbcInsert插入数据

SimpleJdbcInsert有两个非常有用的方法来执行数据插入操作：excute()和executeAndReturnKey()。它们都接受Map<String, Object>作为参数，其中Map的key对应表中要插入数据的列名，而Map中的value对应要插入到列中的实际值。

3.2使用Spring Data JPA持久化数据

Spring Data是一个非常大的伞形项目，由多个子项目组成，其中大多数子项目都关注对不同的数据库类型进行数据持久化。比较流行的几个Spring Data项目包括：

- Spring Data JPA：基于关系型数据库进行JPA持久化。
- Spring Data MongoDB：持久化到Mongo文档数据库。
- Spring Data Neo4j：持久化到Neo4j图数据库。
- Spring Data Redis：持久化到Redis key-value存储。
- Spring Data Cassandra：持久化到Cassandra数据库。

3.2.1 添加Spring Data JPA到项目中

Spring Boot应用可以通过JPA starter来添加Spring Data JPA。这个starter依赖不仅会引入Spring Data JPA，还会传递性地将Hibernate作为JPA实现引入进来。

如果想要使用不同的JPA实现，那么至少需要将Hibernate依赖排除出去并将你所选择的JPA库包含进来。

3.2.2 将领域对象标注为实体

在创建repository方面，Spring Data为我们做了很多非常棒的事情。但是，在使用JPA映射注解标注领域对象方面，它却没有提供太多的助益。

使用@Entity注解将类声明为实体，它的id属性需要使用@Id注解，以便于将其指定为数据库中唯一标识该实体的属性。除了JPA特定的注解，JPA需要实体有一个无参的构造器，Lombok的@NoArgsConstructor注解能够帮助我们实现这一点。

@Table("table_name")注解表明实体应该持久化到数据库名为table_name的表中，如果没有它JPA将会默认将实体持久化到以类名为名的表中。

3.2.3 声明JPA repository

在JDBC版本的repository中，我们显式声明想要repository提供的方法。但是借助Spring Data，我们可以拓展CrudRepository接口。

CrudRepository定义了很多用于CRUD（创建、读取、更新、删除）操作的方法。它是参数化的，第一个参数是repository要持久化的实体类型，第二个参数是实体ID属性的类型。Spring Data会在运行期间自动生成实现类，我们只需要像使用基于JDBC的实现那样将它们注入控制器中就可以了。

3.2.4 自定义JPA repository

假设除了CrudRepository所提供的基本CRUD操作之外，我们还需要获取投递到指定邮编（Zip）的订单。实际上我们只需要添加如下方法声明到repository中就可以了。

```
List<Order> findByDeliveryZip(String deliveryZip);
```

当创建repository实现的时候，Spring Data会检查repository接口的所有方法，解析方法的名称，并基于被持久化的对象来试图推测方法的目的。本质上，Spring Data定义了一组小型的领域特定语言（Domain-Specific Language，DSL），在这里持久化的细节都是通过repository方法的签名来描述的。

尽管方法名称约定对于相对简单的查询非常有用，但是对于更为复杂的查询，方法名可能会面临是空的风险。在这种情况下，可以将方法定义为任何你想要的名称，并为其添加@Query注解，从而明确指明方法调用时要执行的查询，如下面的样例所示：

```
@Query("from Order o where o.deliveryCity='Seattle'")
List<Order> readOrdersDeliveredInSeattle();
```

3.3 小结

- Spring的JdbcTemplate能够极大地简化JDBC的使用。
- 在我们需要知道数据库所生成的ID值时，可以组合使用PreparedStatementCreator和KeyHolder。
- 为了简化数据的插入，可以使用SimpleJdbcInsert。
- Spring Data JPA能够极大的简化JPA持久化，我们只需要编写repository接口即可。

第四章 保护Spring

4.1 启用Spring Security

保护Spring应用的第一步就是将Spring Boot security starter添加到构建文件中。当应用启动时，自动配置功能会探测到Spring Security出现在了类路径之中，因此它会初始化一些基本的安全配置。启动应用访问主页，应用将会弹出一个HTTP basic认证对话框并提示认证。要想通过认证，需要一个用户名和密码，用户名是user密码是随机生成的，它会被写入应用的日志文件中。

通过将security starter添加到项目的构建文件中，我们得到了如下安全特性：

- 所有的HTTP请求路径都需要认证；
- 不需要特定的角色和权限；
- 没有登陆页面；
- 认证过程是通过HTTP basic认证对话框实现的；
- 系统只有一个用户，用户名为user。

如果想要确保应用的安全性，我们还有很多工作要做。我们至少要配置Spring Security实现如需哎功能：

- 通过登录页面来提示用户进行认证，而不是使用HTTP basic对话框；
- 提供多个用户，并提供一个注册页面，这样新用户能够注册进来；
- 对不同请求路径，执行不同的安全规则。举例来说，主页和注册页面根本不需要进行认证。

4.2 配置Spring Security

事实上，Spring Security为配置用户存储提供了多个可选方案，包括：

- 基于内存的用户存储；
- 基于JDBC的用户存储；
- 以LDAP作为后端的用户存储；
- 自定义用户详情服务。

不管使用哪种用户存储，你都可以通过覆盖WebSecurityConfigurerAdapter基础配置类中定义的config()方法来进行配置。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  ...
}
```

4.2.1 基于内存的用户存储

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth.inMemoryAuthentication()
```

```
    .withUser("buzz")
    .password("{noop}infinity")
    .authorities("ROLE_USER")
    .and()
    .withUser("woody")
    .password("{noop}bullseye")
    .authorities("ROLE_USER");
}
```

4.2.2 基于JDBC的用户存储

用户信息通常会在关系型数据库中进行维护，基于JDBC的用户存储方案回更加合理一些。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .jdbcAuthentication()
    .dataSource(dataSource);
}
```

这里的config()实现中，调用了AuthenticationManagerBuilder的jdbcAuthentication()方法，需要设置一个DataSource，这里的DataSource是通过自动装配的技巧获取的。

重写默认的用户查询功能

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .jdbcAuthentication()
    .dataSource(dataSource)
    .usersByUsernameQuery(
        "select username, password, enable from Users " +
        "where username=?")
    .authoritiesByUsernameQuery(
                "select username, authority from UserAuthorities " +
                "where username=?");
}
```

在本例中，我们只重写了认证和基本权限的查询语句，但是通过调用groupAuthoritiesByUsername()方法，我们也能够将群组权限重写为自定义的查询语句。将莫尔尼的SQL查询替换为自定义的设计时，很重要的一点就是要遵循查询的基本协议。

使用转码后的密码

我们需要借助passwordEncoder()方法指定一个密码转码器（encoder）：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .jdbcAuthentication()
    .dataSource(dataSource)
    .usersByUsernameQuery(
        "select username, password, enable from Users " +
        "where username=?")
    .authoritiesByUsernameQuery(
                "select username, authority from UserAuthorities " +
                "where username=?")
    .passwordEncoder(new StandardPasswordEncoder("53cr3t"););
}
```

passwordEncoder()方法可以接受Spring Security中PasswordEncoder接口的任意实现。Spring Security的加密模块包括了多个这样的实现。

- BCryptPasswordEncoder：使用bcrypt强哈希加密。
- NoOpPasswordEncoder：不进行任何转码。
- Pbkdf2PasswordEncoder：使用PBKDF2。
- SCryptPasswordEncoder：使用scrypt哈希加密。
- StandardPasswordEncoder：使用SHA-256哈希加密。

密码不会解密，密码的对比是在PasswordEncoder的match()方法中进行的。

4.2.3 以LDAP作为后端的用户存储

为了配置Spring Security使用基于LDAP认证，我们可以使用ldapAuthentication()方法。这个方法在功能上类似jdbcAuthencation()，只不过是LDAP版本

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .ldapAuthentication()
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}");
}
```

方法userSearchFilter()和groupSearchFilter()对于用户来为基础LDAP查询提供过滤条件，分别用户搜索用户和组。userSearchBase()方法为查找用户提供了基础查询。groupSearchBase()为查找组指定了基础查询。

配置密码比对

基于LDAP认证的默认策略是进行绑定操作，直接通过LDAP服务器认证用户。另一种可选方式是进行比对操作。这涉及将输入的密码发送到LDAP目录上，并要求服务器将则合格密码和用户的密码进行比对。因为比对是在LDAP服务器内完成的，实际的密码能保持私密。如果你希望通过密码比对进行认证，可以通过passwordCompare()方法来实现,，可通过passwordAttribute()方法来声明密码属性的名称：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .ldapAuthentication()
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .passwordCompare()
    .passwordEncoder(new BCryptPasswordEncoder())
    .passwordAttribute("passcode");
}
```

引用远程的LDAP服务器

默认情况下，Spring Security的LDAP认证假设LDAP服务器监听本机的33389端口。但是如果你的LDAP服务器在另一台机器上，那么可以使用contextSource()方法来配置这个地址：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .ldapAuthentication()
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .passwordCompare()
    .passwordEncoder(new BCryptPasswordEncoder())
    .passwordAttribute("passcode")
    .and()
    .contextSource()
```

```
    .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");
}
```

contextSource()方法会返回一个ContextSourceBuilder对象，这个对象除了其他功能以外，还提供了url()方法来指定LDAP服务器地址。

配置嵌入式的LDAP服务器

如果没有现成的LDAP服务器供认证使用，Spring Security还为我们提供了嵌入式的LDAP服务器。不需要设置远程LDAP服务器的URL，只需通过root()方法指定嵌入式服务器的根前缀就可以了：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .ldapAuthentication()
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .passwordCompare()
    .passwordEncoder(new BCryptPasswordEncoder())
    .passwordAttribute("passcode")
    .and()
    .contextSource()
    .root("dc=tacocloud,dc=com")
    .ldif("classpath:user.ldif");
}
```

可以通过调用ldif()方法来明确指定加载哪个LDIF文件

4.2.4 自定义用户认证

上一章中采用Spring Data JPA作为数据的持久化方案所以采用相同的方式来持久化用户数据是非常有意义的。如果这样做，数据最终应该位于关系型数据库中。因此我们可以使用基于JDBC的认证，但更好的办法是使用Spring Data repository来存储用户。

自定义用户领域对象和持久化

实体定义完之后定义repository接口Spring Data JPA会在运行时自动生成这个接口的实现。

创建用户详情服务

我们要将自定义的用户详情服务与Spring Security配置在一起：

```
@Autowired
private UserDetailService userDetailService;

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
  auth
    .userDetailService(userDetailService);
}
```

在这里我们只是简单地调用userDetailService()方法，并将自动装配到SecurityConfig中的UserDetailService实例传递了进去。

像基于JDBC的认证一样，我们可以（也应该）配置一个密码转换器，这样在数据库中的密码是转码过的。我们首先需要声明一个PassordEncoder类型的bean，然后通过调用passwordEncoder()方法将它注入到用户详情服务中：

```
@Bean
public PasswordEncoder encoder() {
  return new StandardPasswordEncoder("53cr3t");
}

@Override
protected void config(AuthenticationManagerBuilder auth) throws Exception {
  auth
    .userDetailsService(userDetailsService)
    .passwordEncoder(encoder());
}
```

现在我们已经有了自定义的用户详情服务，它会通过JPA repository读取用户信息，接下来我们需要一个注册界面。

注册用户

尽管安全性方面，Spring Security会为我们处理很多事情，但是它没有直接涉及用户注册的流程，所以我们需要借助Spring MVC的一些技能来完成这个任务。RegistrationController类负责展现和处理注册表单。RegistrationController使用@Controller注解表明它是一个控制器，并且允许组件扫描功能发现它。它还使用了@RequestMapping注解，这样就能处理路径为”/register“的请求了。具体来讲，对”/register“的GET请求会由registerFrom()方法来处理，它只是简单的返回一个逻辑视图名registration。

4.3 保护Web请求

为了配置安全性规则，需要介绍一下WebSecurityConfigurerAdapter的其他config：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        ...
}
```

config()方法接受一个HttpSecurity对象，能够用来配置在Web级别该如何处理安全性。我们可以使用HttpSecurity配置的功能包括：

- 在为某个请求提供服务之前，需要预先满足特定的条件；
- 配置自定义的登录页；
- 支持用户退出应用；
- 预防跨站请求伪造；

4.3.1 保护请求

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequest()
    .antMatchers("/design", "orders")
    .hasRole("ROLE_USER")
    .antMatchers("/", "/**").permitAll();
}
```

对authorizeRequest()的调用会返回一个对象（ExpressionInterceptUrlRegister），基于它我们可以指定URL路径和这些路径的安全需求。在本例中，我们指定了两条安全规则：

- 具备ROLE_USER权限的用户才能访问"/design"和"/orders"；
- 其他请求允许所有用户访问。

声明在前面的安全规则比后面声明的规则有更高的优先级。

方法	能够做什么
access(String)	如果给定的SpEL表达式计算结果为true，就允许访问
anonymous()	允许匿名用户访问

方法	能够做什么
authenticated()	允许认证过的用户访问
denyAll()	无条件拒绝所有访问
fullyAuthenticated()	如果用户是完整认证的（不是通过Remember-me功能认证的），就允许访问
hasAnyAuthority(String...)	如果用户具备给定权限中的某一个，就允许访问
hasAnyRole(String...)	如果用户具备给定角色中的某一个，就允许访问
hasAuthority()	如果用户具备给定权限，就允许访问
hasIpAddress(String)	如果请求来自给定IP地址就允许访问
hasRole(String)	如果用户具备给定角色，就允许访问
not()	对其他访问方法的结果求反
permitAll()	无条件允许访问
rememberMe()	如果用户是通过Remember-me功能认证的，就允许访问

我们还可以使用access()方法，通过为其提供SpEL表达式来声明更丰富的安全规则。Spring Security扩展了SpEL，包含了多个安全相关的值和函数：

安全表达式	计算结果
authentication	用户的认证对象
denyAll	结果始终为false
hasAnyRole(list of roles)	如果用户被授予了列表中任意的指定角色，结果为true
hasRole(role)	如果用户被授予了指定角色，结果为true
hasIpAddress(IP Address)	如果请求来自给定IP地址，结果为true
isAnonymous()	如果当前用户为匿名用户结果为true
isAuthenticated()	如果当前用户进行了认证，结果为true
isFullyAuthenticated()	如果用户是完整认证的（不是通过Remember-me功能认证的），结果为true
isRememberMe()	如果用户是通过Remember-me功能认证的，结果为true
permitAll()	结果始终为true

安全表达式	计算结果
principal	用户的principal对象

使用SpEL可以实现各种各样的安全性限制。

4.3.2 创建自定义的登录页

为了替换内置的登录页，我们首先需要告诉Spring Security自定义登录页路径是什么。这可以通过调用传入到configure()中的HttpSecurity对象的fromLogin()方法来实现

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequest()
    .antMatchers("/design", "/orders")
    .access("hasRole('ROLE_USER')")
    .antMatchers("/", "/**").access("permitAll")

    .and()
    .formLogin()
    .loginPage("/login")
    .defaultSuccessUrl("/design", true)
    .loginProcessingUrl("/authenticate")
    .usernameParameter("user")
    .passwordParameter("pwd");
}
```

当Spring Security断定用户没有认证并且需要登录的时候，它就会将用户重定向到“/login”。如下方法声明了登录页面的视图控制器：

```
@Override
public void addViewController(ViewControllerRegistry registry) {
  registry.addViewController("/").setViewName("home");
  registry.addViewController("/login");
}
```

Spring Security会监听对“/autnenticate”的请求来处理登录信息的提交。同时用户名的密码字段名应该是user和pwd。默认情况下，登录成功之后，用户名将会被导航到Spring Security决定让用户登录之前的页面。如果用户直接访问登陆页面，那么用户登录成功之后用户将会被导航至根路径。但是我们可以通过指定默认的成功页来更改这种行为，即上面代码中配置的“/design”页面。我们还可以通过指定defaultSuccessUrl方法的第二个参数为true来强制用户在登录成功之后统一访问“/design”页面。

4.3.3 退出

为了启用退出功能，我们只需在HttpSecurity对象上调用logout方法：

```
.and
  .logout()
  .logoutSuccessUrl("/")
```

在用户登出后，他们的session将会被清除，默认情况下用户会被重定向到登录页面，但是如果你想要将用户导航至不同的页面，那么可以调用logoutSuccessUrl()指定退出后的不同页面，在本例中用户退出后将会回到主页。

4.3.4 防止跨站请求伪造

跨站请求伪造（Cross-Site Request Frogery，CSRF）是一种常见的安全攻击。它会让用户在一个恶意的Web页面上填写信息，然后自动（通常是秘密的）将表单以攻击受害者的身份提交到另外一个应用上。为了防止这种类型的攻击，应用可以在展现表单的时候生成一个CSRF token，并放到隐藏域中，然后将其临时存储起来，以便后续在服务器上使用。在提交表单的时候，token将和其他的表单数据一起发送至服务器端。请求会被服务器拦截，并与最初生成的token进行对比。如果token匹配，那么请求将会允许处理；否则，表单肯定是由恶意网站渲染的，以为你它不知道服务器所生成的token。

Spring Security提供了内置的CSRF保护，默认是开启的，我们需要确保应用中每个表单都要由一个名为“_csrf”的字段，它会持有CSRF token。

可以通过：

```
.and()
  .csrf()
  .disable()
```

来禁用CSRF防护。

4.4 了解用户是谁

有多种方法可以确定用户是谁，常用方式如下：

- 注入Principal对象到控制器中；
- 注入Authentication到对象控制器中；

- 使用SecurityContextHolder来获取安全上下文；
- 使用@AuthenticationPrincipal注解来标注方法。

注入principal会在与安全无关的功能中掺杂安全性代码。使用Authentication对象之后我们就可以通过getPrincipal()方法来获取principal对象。最整洁的方案可能是在方法中直接接受一个User对象，不过我们需要为其添加@AuthenticationPrincipal注解，这样它才会变成认证的principal。@AuthenticationPrincipal非常好的一点在于它不需要类型转换，同时能将安全相关的代码仅仅局限于注解本身。

另外一种方式能够识别当前认证用户是谁，但是这种方法有点麻烦，它还会包含大量安全性相关的代码。我们可以从安全上下文中获取一个Authentication对象，然后像下面这样获取它的principal：

```
Authentication authentication =
  SecurityContextHolder.getContext().getAuthentication();
User user = (User) authentication.getPrincipal();
```

尽管这个代码片段充满了安全性相关的代码，但是它与前文所述的其他方法相比有一个优势：它可以在应用程序的任何地方使用，而不是仅仅在控制器的处理方法中。这使得它非常适合在较低级别的代码中使用。

4.5 小结

- Spring Security的自动配置是实现基本安全性功能的好办法，但是大多数的应用都需要显式的安全配置，这样才能满足特定的安全需求。
- 用户详情可以通过用户存储进行管理，它的后端可以是关系型数据库、LDAP或完全自定义实现。
- Spring Security会自动防范CSRF攻击。
- 已认证用户的信息可以通过SecurityContext对象（该对象可以由SecurityContextHolder.getContext()返回）来获取，也可以借助@AuthenticationPrincipal注解将其注入到控制器中。