第1部分 Spring的核心

第1章 Spring之旅

1.1 简化Java开发

Spring采取以下4种策略降低Java开发的复杂性

- 基于POJO的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

1.1.1 激发POJO的潜能

Spring不会强迫实现Spring规范的接口或继承Spring规范的类，在基于Spring构建的应用中，它的类通常没有任何痕迹表明使用了Spring。Spring的非侵入编程模型意味着class在Spring应用和非Spring应用中都可以发挥同样的作用。

1.1.2 依赖注入

按照传统做法，每个对象负责管理与自己相互协作的对象（即它所依赖的对象）的引用。

通过DI，对象的依赖关系将由系统中负责协调个对象的第三方组件在创建对象的时候进行设定。

Spring通过应用上下文装载bean的定义并把它们组装起来。

1.1.3 应用切面

DI能够让相互协作的软件保持松耦合，而面向切面编程运行把遍布应用各处的功能分离出来形成可重用的组件。

比如日志、事务管理和安全这样的系统服务。

```
public class BraveKnight {

    private Quest quest;

    public BraveKnight(Quest quest) {
        this.quest = quest;
    }

    public void embarkOnQuest() {
        quest.embark();
    }
}
```

```
public class Minstrel {
    private PrintStream stream;
```

```
    public Minstrel(PrintStream stream) {
        this.stream = stream;
    }

    public void singBeforeQuest() {
        stream.println("Fa la la, the knight is so brave");
    }

    public void singAfterQuest() {
        stream.println("Tee hee hee, the brave knight did embark on a quest");
    }
}
```

```
public interface Quest {
    void embark();
}
```

```
public class SlayDragonQuest implements Quest {

    private PrintStream stream;

    public SlayDragonQuest(PrintStream stream) {
        this.stream = stream;
    }

    @Override
    public void embark() {
        stream.println("Embarking on quest to slay th dragon");
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-
aop.xsd">


    <bean id="knight" class="com.flw.semantic.service.BraveKnight">
        <constructor-arg ref="quest" />
    </bean>

    <bean id="quest" class="com.flw.semantic.service.SlayDragonQuest">
        <constructor-arg value="#{T(System).out}" />
    </bean>


    <bean id="minstrel" class="com.flw.semantic.service.Minstrel">          <!-- 声明Minstrel
```

```
bean -->
        <constructor-arg value="#{T(System).out}" />
    </bean>

    <aop:config>
        <aop:aspect ref="minstrel">
            <aop:pointcut id="embark" expression="execution(* *.embarkOnQuest(..))"/>
<!-- 定义切点 -->

            <aop:before pointcut-ref="embark" method="singBeforeQuest"/>     <!-- 声明前置通知 -->
            <aop:after pointcut-ref="embark" method="singAfterQuest"/>       <!-- 声明后置通知 -->
        </aop:aspect>
    </aop:config>

</beans>
```

1.1.4 使用模版消除板式代码

样板式代码的一个常见范例是使用JDBC访问数据查询数据。

可以使用JdbcTemplate来消除。

1.2 容纳你的Bean

在基于Spring的应用中，对象生存于Spring容器中。Spring自带了多个容器实现，可以归为两种不同的类型：

- BeanFactory：最简单的容器，由`org.springframework.beans.factory.BeanFactory`定义，提供基本的DI支持。
- 应用上下文：由`org.springframework.context.ApplicationContext`接口定义，基于BeanFactory构建，并提供应用框架级别的服务。

BeanFactory对大多数应用来说太低级了。

1.2.1 使用应用上下文

Spring自带了多种类型的应用上下文：

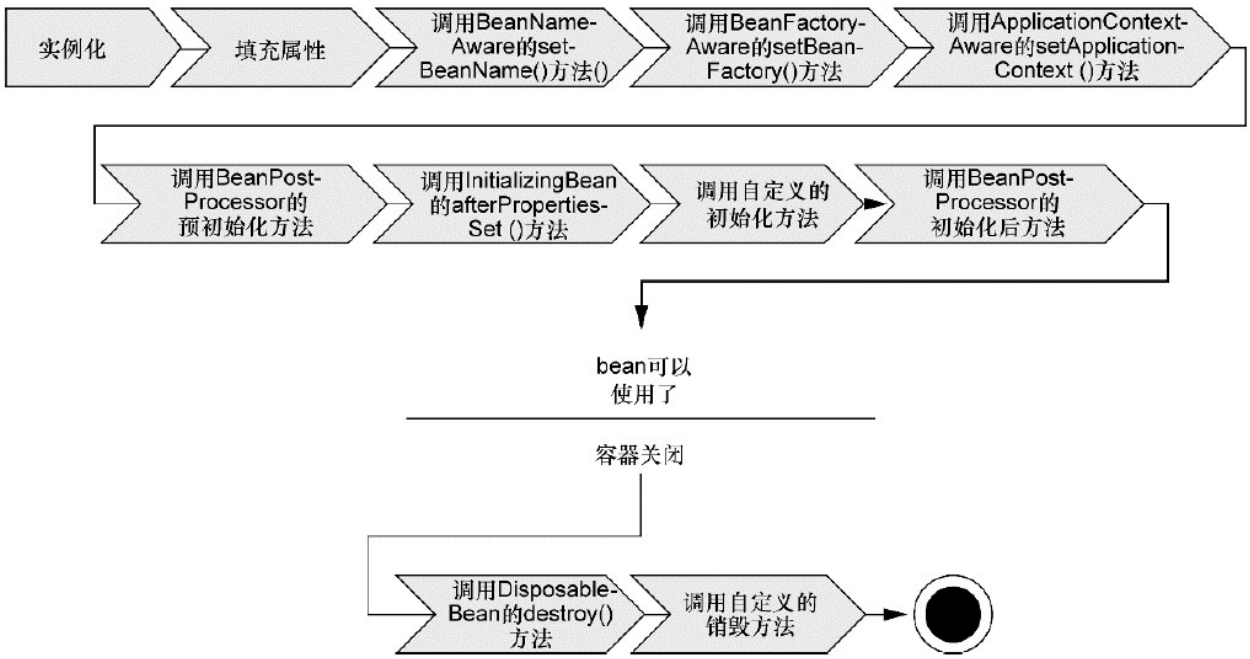
- AnnotationConfigApplicationContext：从一个或多个基于Java的配置类中加载Spring应用上下文。
- AnnotationConfigWebApplicationContext：从一个或多个基于Java的配置类中加载Spring Web应用上下文。
- ClassPathXmlApplicationContext：从类路径下的一个或多个XML配置文件中加载上下文定义，把应用上下文的定义文件作为类资源。
- FileSystemXmlApplicationcontext：从文件系统下的一个或多个XML配置文件中加载上下文定义。
- XmlWebApplicationContext：从Web应用下的一个或多个XML配置文件中加载上下文定义。

FileSystemXmlApplicationcontext和ClassPathXmlApplicationContext

```
ApplicationContext context = new FileSystemXmlApplicationcontext("c:/knight.xml");
// or
ApplicationContext context = new ClassPathXmlApplicationContext("knight.xml");
```

ClassPathXmlApplicationContext是在所有的类路径下查找knight.xml文件。

1.2.2 bean的生命周期



1. Spring对bean进行实例化；
2. Spring将值和bean的引用注入到bean对应的属性中；
3. 如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBean-Name()方法；
4. 如果bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方法，将BeanFactory容器实例传入；
5. 如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext()方法，将bean所在的应用上下文的引用传入进来；
6. 如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-ProcessBeforeInitialization()方法；
7. 如果bean实现了InitializingBean接口，Spring将调用它们的after-PropertiesSet()方法。类似地，如果bean使用init-method声明了初始化方法，该方法也会被调用；
8. 如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-ProcessAfterInitialization()方法；
9. 此时，bean已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；
10. 如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方法。同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。

第2章 装配Bean

2.1 Spring配置的可选方案

三种主要的装配机制：

- 在XML中进行显式配置
- 在Java中进行显式配置
- 隐式的bean发现机制和自动装配

2.2 自动化装配bean

Spring从两个角度来实现自动化装配：

- 组件扫描：Spring会自动发现应用上下文中所创建的bean
- 自动装配：Spring自动满足bean之间的依赖

2.2.1 创建可被发现的bean

组件扫描默认是不启用的，需要显式配置一下：

```
@Configuration
@ComponentScan
public class CDPlayerConfig {}
```

`@ComponentScan`默认会***扫描与配置类相同的包及这个包下所有的子包。***

创建JUnit测试：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(CDPlayerConfig.class)
public class CDPlayerTest{ /*..*/}
```

使用Spring的`@RunWith(SpringJUnit4ClassRunner.class)`，***以便在测试开始的时候自动创建Spring的应用上下文。*** `@ContextConfiguration(CDPlayerConfig.class)`告诉Spring在CDPlayerConfig中加载配置

2.2.2 为组件扫描的bean命名

在前面的例子中，Spring会根据类型给其指定一个ID，***就是将类名的第一个字母变为小写。***

可以显式地指定ID：

```
@Component("lonelyHeartsClub")
public class SgtPeppers implements CompactDisc {}
```

2.2.3 设置组件扫描的基本包

显式设置扫描的基本包

```
@Configuration
@ComponentScan(basePackages = {"soundsystem", "video"})
public class CDPlayerConfig {}
```

这种方式设置的基础包是以String类型表示的，但这种方法是类型不安全的，如果重构代码，所指定的基础包可能会出现错误。

还可以将其指定为***包中所包含的类或接口***：

```
@Configuration
@ComponentScan(basePackageClasses = {CDPlayer.class, DVDPlayer.class})
public class CDPlayerConfig {}
```

CDPlayer和DVDPlayer这两个类所在的包将会作为组件扫描的基础包。

2.2.4 通过为bean添加注解实现自动装配

`@Autowired`可以使用在成员变量和方法上。如果没有匹配的bean，Spring会抛出一个异常，可以通过`@Autowired(required=false)`来避免（如果没有匹配的bean，Spring会让这个bean处于未装配的状态，即为null）；如果有多个bean都能满足依赖关旭，Spring将会抛出一个异常， 表明没有明确指定要选择哪个bean进行自动装配。

2.3 通过Java代码装配bean

2.3.2 声明简单的bean

```
@Bean
public CompactDisc sgtPeppers() { return new SgtPeppers(); }
```

默认情况下，bean的ID与带有@Bean注解的***方法名***是一样的。如果想为其设置一个不同的名字，可以通过name属性指定一个不同的名字：`@Bean(name="lonelyHeartsClubBand")`。

2.3.3 借助JavaConfig实现注入

```
@Bean
public CDPlayer cdPlayer() { return new CDPlayer(sgtPeppers()); }
```

看起来CompactDisc是通过调用`sgtPeppers()`得到的，但因为`sgtPeppers()`方法上添加了`@Bean`注解，Spring将会拦截所有对它的调用，***并确保直接返回该方法所创建的bean（默认是单例）***，而不是每次都对其进行实际的调用。

还有另外一种更好的方式：

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) { return new CDPlayer(compactDisc); }
```

这种方式不需要将CompactDisc声明到同一个配置类中。可以将配置分散到多个配置类、xml文件以及自动扫描和装配bean之中，不管CompactDisc采用什么方式创建出来，Spring都会将其传入到配置文件中，并用来创建CDPlayer的bean。

```
@Configuration
public class CDConfig {
    @Bean
    public CompactDisc compactDisc() { return new SgtPeppers(); }
}
```

```
@Configuration
public class CDPlayerConfig {
    @Bean
    public CDPlayer cdPlayer(CompactDisc compactDisc) { return new CDPlayer(compactDisc); }
}
```

```
@Configuration
@Import({CDPlayerConfig.class, CDConfig.class})
public class SoundSystemConfig {}
```

2.5 导入和混合配置

2.5.1 在JavaConfig中引用XML配置

假设compactDisc的bean定义在根路径下的cd-config.xml中：

```
@Configuration
public class CDPlayerConfig {
    @Bean
    public CDPlayer cdPlayer(CompactDisc compactDisc) { return new CDPlayer(compactDisc); }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-
aop.xsd">

   <bean id="compactDisc" class="soundsystem.BlankDisc">
       <!-- .. -->
   </bean>
</beans>
```

```
@Configuration
@Import(CDPlayerConfig.class)
@ImportResource("classpath:cd-config.xml")
@ComponentScan     //可以在这个根配置中配置组件扫描
public class SoundSystemConfig {}
```

2.5.2 在XML配置中引用JavaConfig

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-
aop.xsd">

    <context:component-scan base-package=".."/>         <!-- 可以在这个根配置中配置组件扫描 -->

    <bean class="soundsystem.CDPlayerConfig" />
    <import resource="cd-config.xml" />

</beans>
```

2.6 小结

自动化配置 > 基于Java的配置 > 基于XML的配置

第3章 高级装配

3.1 环境与profile

在不同的环境中某个bean会有所不同，一种解决方式是在单独的配置类中配置每个bean，然后在构建阶段（可能会使用Maven的profiles）确定要将哪一个配置编译到可部署的应用中。***这种方式的问题在于要为每种环境重新构建应用。***

3.1.1 配置profile bean

Spring提供的方案不是在构建的时候，而是等到***运行时再来根据环境决定创建哪个bean。***这样的结果就是同一个部署单元（可能是WAR文件）能够适用于所有的环境，没有必要进行重新构建。

要使用profile，首先要将所有不同的bean定义整理到一个或多个profile之中，在将应用部署到每个环境时，要确保对应的profile处理激活状态。

在Java配置中，可以使用`@Profile`注解指定某个bean属于哪一个profile。

```
@Configuration
@Profile("dev")
public class DevelopmentProfileConfig {

    @Bean(destroyMethod = "shutdown")
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .addScript("classpath:schema.sql")
                .addScript("classpath:test-data.sql")
                .build();
    }
}
```

`@Profile`注解应用在***类级别***上，会告诉Spring这个配置类中的bean只有在dev profile被激活时才会创建。如果dev profile没有被激活，那么这个类中带有@Bean注解的方法都会被忽略掉。

`@Profile`注解也可以应用在方法上，与`@Bean`注解一同使用。

在XML中配置profile

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd"
  profile="dev">

  <jdbc:embedded-database id="dataSource">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:test-data.sql" />
  </jdbc:embedded-database>

</beans>
```

重复使用元素来指定多个profile

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <beans profile="dev">                    <-------------- dev profile 的 bean
    <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:schema.sql" />
      <jdbc:script location="classpath:test-data.sql" />
    </jdbc:embedded-database>
  </beans>

  <beans profile="qa">                     <-------------- qa profile 的 bean
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:url="jdbc:h2:tcp://dbserver/~/test"
          p:driverClassName="org.h2.Driver"
          p:username="sa"
          p:password="password"
          p:initialSize="20"
          p:maxActive="30" />
  </beans>

  <beans profile="prod">                   <-------------- prod profile 的 bean
     <jee:jndi-lookup id="dataSource"
                      jndi-name="jdbc/myDatabase"
                      resource-ref="true"
                      proxy-interface="javax.sql.DataSource" />
  </beans>
</beans>
```

3.1.2 激活profile

Spring在确定哪个profile处于激活状态时，需要依赖两个独立的属性：

- spring.profiles.active
- spring.profiles.default

> 如果设置了spring.profiles.active属性的话，那么它的值就会用来确定哪个profile是激活的。但如果没有设置，那Spring将会查找spring.profiles.default的值。如果spring.profiles.active和spring.profiles.default均没有设置的话，那就没有激活的profile，因此只会创建那些没有定义在profile中的bean。

有多种方式来设置这两个属性：

- 作为DispatcherServlet的初始化参数
- 作为Web应用的上下文参数
- 作为环境变量
- 作为JVM的系统属性
- 在集成测试类上，使用@ActiveProfiles注解设置

3.2 条件化的bean

假设你希望一个或多个bean只有在应用的类路径下包含特定的库时才创建。或者我们希望某个bean只有当另外某个特定的bean也声明了之后才会创建。我们还可能要求只有某个特定的环境变量设置之后，才会创建某个bean。

例如，假设有一个名为MagicBean的类，我们希望只有设置了magic环境属性的时候，Spring才会实例化这个类。如果环境中没有这个属性，那么MagicBean将会被忽略。

```
@Bean
@Conditional(MagicExistsCondition.class)
public MagicBean magicBean() { return new MagicBean(); }
```

`@!Conditional`将会根据Condition接口进行条件对比：

```
public interface Condition {
    boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);
}
```

设置给`@Conditional`的类可以是任意实现了Condition接口的类型。如果matches()返回true，就会创建带有`@Conditional`注解的bean。

```
public class MagicExistsCondition implements Condition {
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        Environment env = context.getEnvironment();
        return env.containsProperty("magic");    //检查magic属性
    }
}
```

借助ConditionContext和AnnotatedTypeMetadata，可以得到很多信息

3.3 处理自动装配的歧义性

3.3.1 标示首选的bean

```
@Component
@Primary
public class IceCream implements Dessert {}

@Component
public class Cake implements Dessert {}
```

当首选bean的数量超过一个时，我们没有其他的方法进一步缩小可选的范围

3.3.2 限定自动装配的bean

`@Qualifier`是使用限定符的注意方式，可以与`@Autowired`协同使用，为`@Qualifier`注解所设置的参数就是想要注入的bean的ID。

```
@Autowired
@Qualifier("iceCream")
public void setDessert(Dessert dessert) { this.dessert = dessert; }
```

更准确地讲，`@Qualifier("iceCream")`所引用的bean要具有String类型的"iceCream"作为限定符，如果没有指定其他的限定符，***所有的bean都会给定一个默认的限定符—bean ID***。

我们可以为bean设置自己的限定符，而不是依赖于bean ID作为限定符。

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert {..}
```

3.4 bean的作用域

默认情况下，Spring应用上下文中所有bean都是***单例***的。

Spring定义了多种作用域：

- 单例（Singleton）
- 原型（Prototype）：每次注入或者通过Spring应用上下文获取的时候，都会创建一个新的bean实例
- 会话（Session）：在web应用中，为每个会话创建一个bean实例
- 请求（Request）：在web应用中，为每个请求创建一个bean实例

如果选择其他的作用域，要使用`@Scope`注解

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Notepad {}
```

3.5 运行时值注入

3.5.1 注入外部的值

在Spring中，处理外部值最简单的方式就是声明属性源并通过Spring的Environment来检索属性。

```
@Configuration
@PropertySource("classpath:/com/soundsystem/app.properties")
public class ExpressiveConfig {
    @Autowired
    private Environment env;

    @Bean
    public BlankDisc disc() {
        return new BlankDisc(
            env.getProperty("disc.title"),
            env.getProperty("disc.artist")
        );
    }
}
```

解析属性占位符

```
<bean id="sgtPeppers" class="soundsystem.BlankDisc">
    <property name="title" value="${disc.title}" />
    <property name="artist" value="${disc.artist}" />
</bean>
```

这些值是从配置文件以外的一个源中解析得到的。

如果依赖组件扫描和自动装配来创建和初始化bean的时候，可以使用`@Value`注解

```
@Component
public BlankDisc {

    private String title;
    private String artist;

    public BlankDisc (
             @Value("${disc.title}") String title,
             @Value("${disc.artist}") String artist
        ) {
        this.title = title;
        this.artist = artist;
    }
}


//或者
@Component
public BlankDisc {

    @Value("${disc.title}")
    private String title;

    @Value("${disc.artist}")
    private String artist;
}
```

注：`@Value`必须和`@Component`或者`@Configuration`等注解一起使用。

为了使用占位符，必须要配置一个`PropertySourcesPlaceholderConfigurer`：

```
@Bean
public static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
}
```

3.5.2 使用Spring表达式语言进行装配

SpEL表达式要放到"#{...}"之中。

- `#{T(System).currentTimeMillis()}`：计算表达式的那一刻当前时间的毫秒数。`T()`表达式会将java.lang.System视为Java中对应的类型。因此可以调用其static修饰的currentTimeMillis()方法。
- `#{sgtPeppers.artist}`：引用了一个ID为sgtPeppers的bean的artist的属性
- `#{systemProperties['disc.title']}`：引用系统属性

第4章 面向切面的Spring

横切关注点：散布于应用中多处的功能被称为横切关注点。

把横切关注点与业务逻辑相分离正是面向切面编程要解决的问题

4.1 什么是面向切面编程

4.1.1 定义AOP术语

通知

通知定义了切面是什么以及何时使用。

Spring切面可以应用5种类型的通知：

- 前置通知
- 后置通知
- 返回通知
- 异常通知
- 环绕通知

Spring AOP是切面在***应用运行***的某个时刻被织入。一般情况下，在织入切面时，***AOP容器会为目标对象动态地创建一个代理对象***。

4.1.2 Spring对AOP的支持

Spring提供了4种类型的AOP支持：

- 基于代理的经典Spring AOP；
- 纯POJO切面；
- @AspectJ注解驱动的切面
- 注入式AspectJ切面

因为Spring AOP构建在动态代理基础之上，因此Spring对AOP的支持局限于***方法拦截***。

Spring在运行时通知对象

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。代理类封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候，Spring才会创建被代理的对象。

Spring只支持方法级别的连接点

4.3 使用注解创建切面

4.3.1 定义切面

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class Audience {
    @Before("execution(** concert.Performance.perform(..))")
    public void silenceCellPhones() {
        System.out.println("silence cell phone");           //表演之前
    }

    @Before("execution(** concert.Performance.perform(..))")
    public void takeSeats() {
        System.out.println("take seats");           //表演之前
    }

    @AfterReturning("execution(** concert.Performance.perform(..))")
    public void applause() {
        System.out.println("clap clap clap");           //表演之后鼓掌
    }

    @AfterThrowing("execution(** concert.Performance.perform(..))")
    public void demandRefund() {
        System.out.println("demand a refund");          //表演失败之后要求退款
    }
}
```

可以通过定义可重用的切点来减少注解里的重复

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

    @Pointcut("execution(** concert.Performance.perform(..))")
    public void performance(){}         //方法本身只是一个标识

    @Before("performance()")

    public void silenceCellPhones() {
```

```
        System.out.println("silence cell phone");           //表演之前
    }

    @Before("performance()")
    public void takeSeats() {
        System.out.println("take seats");           //表演之前
    }

    @AfterReturning("performance()")
    public void applause() {
        System.out.println("clap clap clap");           //表演之后鼓掌
    }

    @AfterThrowing("performance()")
    public void demandRefund() {
        System.out.println("demand a refund");           //表演失败之后要求退款
    }

    @Around("performance()")                        //环绕通知
    public void watchPerformance(ProceedingJoinPoint jp) {
        try {
            System.out.println("silence cell phone");
            System.out.println("take seats");

            jp.proceed();

            System.out.println("clap clap clap");
        }catch (Throwable e) {
            System.out.println("demand a refund");
        }
    }
}
```

此时Audience只是Spring容器中的一个bean，不会被视为切面，需要配置`@EnableAspectJAutoProxy`注解启动自动代理功能：

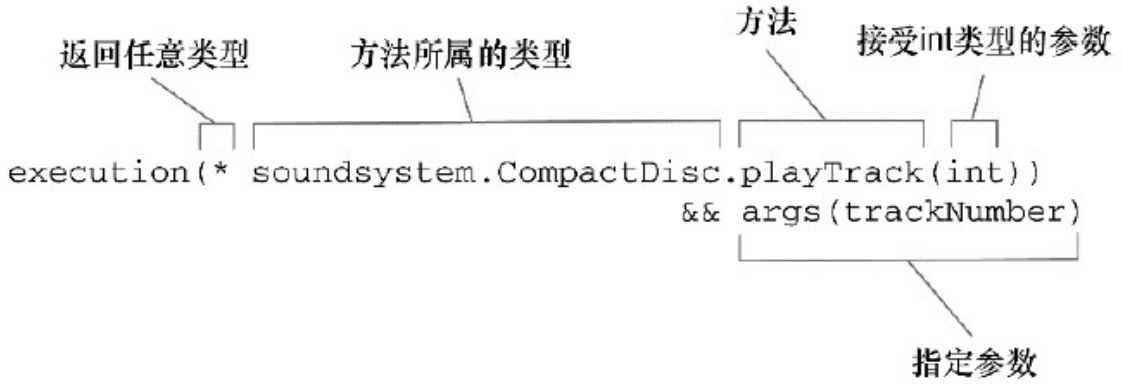
```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan
public class ConcertConfig {
    @Bean
    public Audience audience() {
        return new Audience();
    }
}
```

4.3.2 创建环绕通知

实际上就像在一个通知方法中同时编写前置通知和后置通知。

如果不调用`jp.proceed();`方法，那么通知实际上会阻塞被通知方法的调用。

4.3.3 处理通知中的参数



切点表达式中的`args(trackNumber)`限定符表明传递给`playTrack()`方法的int类型参数也会传递到通知中去。参数的名称trackNumber也与切点方法签名中的参数相匹配。

这个参数会传递到通知方法中：

```
@Aspect
public class TrackCounter {
    @Pointcut("execution(* soundsystem.CompactDisc.playTrack(int)) && args(trackNumber)")
    public void trackPlayed(int trackNumber){}

    @Before("trackPlayed(trackNumber)")
    public void countTrack(int trackNumber) {
        //...
    }
}
```

第2部分 Web中的Spring

第5章 构建Spring Web应用程序

5.1 Spring MVC起步

5.1.1 跟踪Spring MVC的请求

请求 ==> 单实例的DispatcherServlet（前端控制器Servlet） ==> Controller ==> 处理后将请求连同模型和视图名发送回DispatcherServlet ==> DispatcherServlet使用视图解析器将逻辑视图名匹配为一个特定的视图实现 ==> 视图使用模型数据渲染输出 ==> 输出通过响应对象传递给客户端

5.1.2 搭建Spring MVC

配置DispatcherServlet

传统上，像DispatcherServlet这样的Servlet会配置在web.xml中。

但是在Servlet 3规范和Spring 3.1的功能增强中，可以使用Java将DIspatcherServlet配置在Servlet容器中，而不会使用web.xml文件。

```
package spittr.config;

public class SpittrWebAppInitializer
                extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected String[] getServletMappings() {       //将DispatcherServlet映射到"/"
        return new String[] { "/" };
    }
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {    //指定配置类
        return new Class<?>[] { WebConfig.class };
    }
}
```

扩展`AbstractAnnotationConfigDispatcherServletInitializer`的任意类都会自动地配置DispatcherServlet和Spring应用上下文，*Spring的应用上下文会位于应用程序的Servlet上下文之中*。

在Servlet 3.0环境中，容器会在类路径下查找实现了`javax.servlet.ServletContainerInitializer`接口的类，如果发现，就会用它配置Servlet容器。Spring提供了一个名为`SpringServletContainerInitializer`的这个接口的实现类，这个类反过来又会查找实现`WebApplicationInitializer`的类并将配置的任务交给它们，Spring 3.2引入了一个实现类，也就是`AbstractAnnotationConfigDispatcherServletInitializer`。

本例中映射的是 "/"，表示会是应用的默认Servlet，会处理进入应用的所有请求。

DispatcherServlet和ContextLoaderListener（Servlet监听器）

当DispatcherServlet启动时，会创建Spring应用上下文并加载声明的bean。

在Spring Web应用中，还有另外一个应用上下文（由ContextLoaderListener创建的）。

我们希望：

- DispatcherServlet创建的应用上下文：*加载包含Web组件的bean，如控制器，视图解析器，处理器映射器*
- ContextLoaderListener创建的应用上下文：*加载其他bean，如驱动应用后端的中间层和数据层组件*

`AbstractAnnotationConfigDispatcherServletInitializer`会同时创建`DispatcherServlet`和`ContextLoaderListener`。

1. `getServletMappings()`：将一个或多个路径映射到DispatcherServlet上。
2. `getServletConfigClasses()`：返回的带有`@Configuration`注解的类用来定义DispatcherServlet创建的应用上下文中的bean。
3. `getRootConfigClasses()`：返回的带有`@Configuration`注解的类用来配置ContextLoaderListener创建的应用上下文中的bean。

启用Spring MVC

```
package spittr.config;

//最小的Spring MVC配置
@Configuration
@EnableWebMvc           //启用Spring MVC
public clas WebConfig {
}
```

上述代码可以运行起来，但是有问题：

- 没有视图解析器

  Spring会默认使用BeanNameView-Resolver视图解析器

- 没有启用组件扫描

  Spring只能找到显式声明在WebConfig中的控制器

- DispatcherServlet会映射为应用的默认Servlet

  会处理所有请求，包括对静态资源的请求。

```
package spittr.config;

/**
 * 加载包含Web组件的bean，如控制器，视图解析器，处理器映射器
 */
//最小但可用的Spring MVC配置
@Configuration
@EnableWebMvc           //启用Spring MVC
@ComponentScan("spittr.web")                //启动组件扫描
public clas WebConfig extends WebMvcConfigurerAdapter {

   @Bean
    public ViewResolver viewResolver() {    //配置JSP视图解析器
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeContextBeansAsAttributes(true);
        return resolver;
    }

    @Override               //配置静态资源的处理
    public void configureDefaultServletHandling(
                            DefaultServletHandlerConfigurer configurer) {
        configurer.enable();    //要求DispatcherServlet将对静态资源的请求转发到Servlet容器默认的
Servlet上，而不是使用DispatcherServlet本身来处理此类请求。
    }
}
```

`InternalResourceViewResolver`会查找JSP文件，在查找的时候，会在视图名称上加一个特定的前缀和后缀。

```
package spitter.config;

/**
 * 加载其他bean，如驱动应用后端的中间层和数据层组件
 */
@Configuration
@ComponentScan(basePackages = {"spitter"},
        excludeFilters = {
            @ComponentScan.Filter(type = FilterType.ANNOTATION, value = EnableWebMvc.class)
        })      //扫描除了WebConfig类之外的其他带有spring注解的类
public class RootConfig {
}
```

5.2 编写基本的控制器

控制器只是方法上添加了`@RequestMapping`注解的类。

```
package spitter.web;

@Controller
public class HomeController {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home() { return "home"; }
}
```

5.2.1 测试控制器

```
public class HomeControllerTest {
    @Test
    public void home() throws Exception {
        HomeController controller = new HomeController();
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(controller).build();

        mockMvc.perform(MockMvcRequestBuilders.get("/"))
                .andExpect(MockMvcResultMatchers.view().name("home"));
    }
}
```

5.2.2 定义类级别的请求处理

```
@Controller
@RequestMapping(value = {"/", "/homepage"})
public class HomeController {
    ...
}
```

5.3 接受请求的输入

Spring MVC允许多种方式将客户端的数据传送到控制器的处理器方法中：

- 查询参数：Query Parameter
- 表单参数：Form Parameter
- 路径变量：Path Variable

5.3.1 处理查询参数

```
/spitters?max=238900&count=50
```

```
@RequestMapping("/spitters", method = RequestMethod.GET)
public List<Spittle> spittles(
                @RequestParam("max") long max,
                @RequestParam("count") int count){
    ..
}
```

5.3.2 通过路径参数接受输入

```
/spittes/12345
```

```
@RequestMapping("/spitters/{spittleId}", method = RequestMethod.GET)
public Spittle spittle(
                @PathVariable("spittleId") long spittleId){
    ..
}
```

如果`@PathVariable`没有value属性，会假设占位符的名称与方法的参数名相同。

5.4 处理表单

```
@RequestMapping("/spitters/register", method = RequestMethod.POST)
public String processRegistration(Spitter spitter){
    ..
}
```

5.4.2 校验表单

使用Spring对Java校验API（又称SR-303）。在Spring MVC中要使用Java校验API，只要在类路径下包含这个API的实现即可，比如Hibernate Validator

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifact>hibernate-validator</artifact>
    <version>6.0.9.Final</version>
</dependency>

<!-- 或者 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

```
public class Spitter {
    private Long id;

    @NotNull
    @Size(min=5, max=16)
    private String username;

    @NotNull
    @Size(min=5, max=25)
    private String password;
}
```

```
@RequestMapping("/spitters/register", method = RequestMethod.POST)
public String processRegistration(@Valid Spitter spitter, Errors errors){
    if (errors.hasErrors()) {
        return "registerForm";          //如果校验出现错误，则重新返回表单
    }
}
```

Errors参数要紧跟在带有@Valid注解的参数后面。

第6章 渲染Web视图

6.1 理解视图解析

Spring MVC定义了名为`ViewResolver`的接口，在第5章中使用的名为`InternalResourceViewResolver`的视图解析器是这个接口的一个实现类。

```
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

给`resolveViewName()`传入一个视图名和Locale对象，会返回一个view实例。

```
public interface View {
    String getContentType();
    void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
                        throws Exception;
}
```

View接口的作用是接受模型以及Servelt的request和response对象，并将结果渲染到response中。

Spring提供了13中视图解析器（`ViewResolver`接口的实现类）。`InternalResourceViewResolver`一般会用于JSP，`FreeMarkerViewResolver`和`VelocityViewResolver`分别用于Freemarker和Velocity模版视图。

6.2 创建JSP视图

使用`InternalResourceViewResolver`会将逻辑视图名解析为`InternalResourceView`。

6.4 使用Thymeleaf

6.4.1 配置Thymeleaf视图解析器

为了要在Spring中使用Thymeleaf，我们需要配置三个启用Thymeleaf与Spring集成的bean：

- `ThymeleafViewResolver`：将逻辑视图名称解析为Thymeleaf模板视图；
- `SpringTemplateEngine`：处理模板并渲染结果；
- `TemplateResolver`：加载Thymeleaf模板。

```
package spittr.config;

//最小但可用的Spring MVC配置
@Configuration
@EnableWebMvc           //启用Spring MVC
@ComponentScan("spittr.web")                //启动组件扫描
public clas WebConfig extends WebMvcConfigurerAdapter {
    //Spring 5以后WebMvcConfigurerAdapter过期了，可以implements WebMvcConfigurer代替

    @Bean
    public ViewResolver viewResolver(SpringTemplateEngine templateEngine) {//Thymeleaf视图解析器
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setTemplate(templateEngine);
        return resolver;
    }

    @Bean
    public TemplateEngine templateEngine(TemplateResolver templateResolver) {//模版引擎
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);
        return templateEngine;
    }

    @Bean
    public TemplateResolver templateResolver() {//模版解析器
        TemplateResolver templateResolver = new ServletContextTemplateResolver();

        templateResolver.setPrefix("/WEB-INF/templates/");
```

```
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        return templateResolver;
    }
}
```

第7章 Spring MVC的高级技术

7.1 Spring MVC配置的替代方案

7.1.3 在web.xml中声明DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <context-param>     <!-- 使用Java配置，不使用xml配置 -->
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <context-param>     <!-- 指定根配置类 -->
        <param-name>contextConfigLocation</param-name>
        <param-value>
            com.habuma.spitter.config.RootConfig
        </param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
               org.springframework.web.context.support.AnnotationConfigWebApplicationContext
            </param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>

            <param-value>
```

```
                com.habuma.spitter.config.WebConfigConfig
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <multipart-config>          <!-- 配置文件上传 -->
            <location>/tmp/spitter/uploads</location>
            <max-file-size>2097152</max-file-size>
            <max-request-size>4194304</max-request-size>
        </multipart-config>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

7.2 处理multipart形式的数据

7.2.2 处理multipart请求

```
<form method="POST" enctype="multipart/form-data">
    <input type="file" accept="image/jpeg, image/png" />
</form>
```

```
@PostMapping(vale = "/register")
public String processRegistration (
    @RequestPart("profilePicture") byte[] profilePicture,
) {
    ...
}
```

7.4 为控制器添加通知

作用：在多个控制器中处理异常。

控制器通知是任意带有`@ControllerAdvice`注解的类，这个类包含一个或多个如下类型的方法：

- `@ExceptionHandler`注解标注的方法
- `@InitBinder`注解标注的方法
- `@ModelAttribute`注解标注的方法

```
//统一异常处理
@ControllerAdvice
public class AppWideExceptionHandler {
    @ExceptionHandler(Exception.class)
    public String exceptionHandler() {
        return "error/error";
    }
}
```

7.5 跨重定向请求传递数据

在处理完POST请求后，一个最佳实践就是执行一下重定向。可以防止用户点击浏览器的刷新按钮或后退箭头时，客户端重新执行危险的POST请求。

> 一般来讲，当一个处理器方法完成之后，该方法所指定的模型数据将会复制到请求中，并作为请求中的属性，请求会转发到视图上进行渲染。因为控制器方法和视图所处理的是**同一个请求**，所以在转发的过程中，请求属性能够得以保存。

对于重定向来说，传递数据的方案：

- 使用URL模版以路径变量和`/`或查询参数的形式传递数据
- 通过flash属性发送数据

7.5.1 通过URL模版进行重定向

```
@PostMapping(value = "/register")
public String processRegistration (
    Spitter spitter, Model model
) {
    spitterRepository.save(spitter);
    model.addAttribute("username", "vermouth");
    model.addAttribute("spitterId", 121112);
    return "redirect:/spitter/{username}";      //"redirect:/spitter/vermouth?spitterId=121112"
}
```

这种方式的话username中所有的不安全字符都会进行转义，会更加安全。

这种方式的缺点：只能发送简单的值，没有办法发送更为复杂的值。

7.5.2 使用flash属性

如果要发送spitter对象，就不能像路径变量或查询参数一样发送。

有个解决办法是将spitter放到会话中，**会话能够长期存在，并且能跨越多个请求。**

Spring提供了通过RedirectAttributes设置flash属性的方法，RedirectAttributes提供了`addFlashAttribute()`方法来添加flash属性。

> 在重定向执行之前，所有的flash属性都会复制到会话中，在重定向后，存在会话中的flash属性会被取出，并从会话转移到模型之中。

```

@PostMapping(value = "/spitter/register")
```

```
public String processRegistration (
    Spitter spitter, RedirectAttributes model
) {
    spitterRepository.save(spitter);
    model.addAttribute("username", "vermouth");
    model.addFlashAttribute("spitter", spitter);
    return "redirect:/spitter/{username}";
}

@GetMapping(value = "/spitter/{username}")
public String showSpitterProfile (
    @PathVariable String username, Model model
) {
    if(!model.containsAttribute("spitter")) {
        model.addAttribute(spitterRepository.findByUsername(username));
    }
    return "profile";
}
```

第8章 使用Spring Web Flow

Spring Web Flow是Spring MVC的扩展，它支持开发基于流程的应用程序。

第9章 保护Web应用

安全性是绝大部分应用程序中的一个重要***切面***。

9.1 Spring Security简介

它能够在Web请求级别和方法调用级别处理身份认证和授权。Spring Security还能够使用Spring AOP保护方法调用——借助于**对象代理和使用通知**，能够确保只有具备适当权限的用户才能访问安全保护的方法。

9.1.2 过滤Web请求

Spring Security借助一系列**Servlet Filter**来提供各种安全性功能。

`DelegatingFilterProxy`是一个特殊的Servlet Filter， 任务就是将工作委托给一个`javax.servlet.Filter`实现类，这个实现类作为一个`<bean>`注册在Spring应用的上下文中。

在web.xml中：

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

这里最重要的是`<filter-name>`设置成了`springSecurityFilterChain`。这是因为我们马上就会将Spring Security配置在Web安全性之中，这里会有一个名为springSecurityFilterChain的Filter bean，DelegatingFilterProxy会将过滤逻辑委托给它。

```
package spitter.config;

import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebInitializer
        extends AbstractSecurityWebApplicationInitializer {
}
```

`AbstractSecurityWebApplicationInitializer`实现了`WebApplicationInitializer`，因此Spring会发现它，并用它在Web容器中注册`DelegatingFilterProxy`。

不管我们通过web.xml还是通过AbstractSecurityWebApplicationInitializer的子类来配置DelegatingFilterProxy，它都会拦截发往应用中的请求，并将请求委托给ID为`springSecurityFilterChain`的 bean。

9.1.3 编写简单的安全性配置

Spring 3.2引入了新的Java配置方案，完全不再需要通过XML来配置安全性功能。

```
package spitter.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

@Configuration
@EnableWebSecurity
public class SecurityWebInitializer
        extends AbstractSecurityWebApplicationInitializer {
}
```

`@EnableWebSecurity`注解将会启用Web安全供能，但它本身没有什么用处，Spring Security必须配置在：

- 一个实现了`WebSecurityConfigurer`的bean中
- 继承`WebSecurityConfigurerAdapter`（最简单的方式）。

```
@Configuration
@EnableWebMvcSecurity       //已过期，Spring4.0中用@EnableWebSecurity代替
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

如果希望指定Web安全的细节，需要通过重载`WebSecurityConfigurerAdapter`中的一个或多个方法来实现。

方法·	描述
configure(WebSecurity)	通过重载，配置Spring Security的Filter链
configure(HttpSecurity)	通过重载，配置如何通过拦截器保护请求
configure(AuthenticationManagerBuilder)	通过重载，配置user-detail服务

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .httpBasic();
}
```

这个配置指定了该如何保护HTTP请求，以及客户端认证用户的方案。通过调用`authorizeRequests()`和`anyRequest().authenticated()`就会要求所有进入应用的HTTP请求都要进行认证。它也配置Spring Security支持基于表单的登录以及HTTP Basic方式的认证。

同时，因为我们没有重载configure(AuthenticationManagerBuilder)方法，所以没有用户存储支撑认证过程。没有用户存储，实际上就等于没有用户。所以，在这里所有的请求都需要认证，但是没有人能够登录成功。

为了让Spring Security满足我们应用的需求，还需要再添加一点配置：

- 配置用户存储；
- 指定哪些请求需要认证，哪些请求不需要认证，以及所需要的权限；
- 提供一个自定义的登录页面，替代原来简单的默认登录页。

9.2 选择查询用户详细信息的服务

Spring Security能够基于各种数据存储来认证用户。它内置了多种常见的用户存储常见、如内存、关系型数据库以及LDAP，以及自定义的用户存储实现。

9.2.1 使用基于内存的用户存储

因为继承了`WebSecurityConfigurerAdapter`，因此配置用户存储的最简单方式就是重载`configure()`方法，并以`AuthenticationManagerBuilder`作为传入参数。通过`inMemoryAuthentication()`方法，我们可以启用、配置并任意填充基于内存的用户存储。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
            .withUser("user").password("password").roles("USER")
        .and()
            .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

上面和下面等价：

```
@Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("sherry").password("password").authorities("ROLE_USER")
            .and()
            .withUser("admin").password("password").authorities("ROLE_USER", "ROLE_ADMIN");
    }
```

9.2.2 基于数据库表进行认证

用户数据通常会存储在关系型数据库中，并通过JDBC进行访问。为了配置Spring Security使用以JDBC为支撑的用户存储，可以使用`jdbcAuthentication()`方法。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .jdbcAuthentication()
        .dataSource(datasource)
        .usersByUsernameQuery(
        "select username, password, true from spitter where username=?"
    )
        .authoritiesByUsernameQuery(
        "select username, 'ROLE_USER' from spitter where username=?"
    )
        .passwordEncoder(new StandardPasswordEncoder("53cr3t"));
}
```

必须配置的是一个DataSource。

第3部分 后端中的Spring

第10章 通过Spring和JDBC征服数据库