本章内容：

- Acegi 安全系统介绍
- 使用 Servlet 过滤器保护 Web 应用系统
- 基于数据库或 LDAP 进行身份认证
- 透明地对方法调用进行保护

你是否曾注意到在电视连续剧中大多数人是不锁门的？这是司空见惯的。在情景喜剧 Seinfeld 中，Krammer 常常到 Jerry 的房间里自己从冰箱里拿东西吃。在 Friends 中，各种各样的剧中人经常不敲门就不假思索地进入别人的房间。甚至有一次在伦敦，Ross 突然进入 Chandler 的旅馆房间，差点儿撞见 Chandler 和 Ross 的妹妹的私情。

在 20 世纪 50 年代 Leave It to Beaver 热播的时代，并不值得为人们不锁门这一现象大惊小怪。但在如今这个隐私和安全极受重视的时代，看到电视剧中的角色允许他人大摇大摆地进入自己的家中或房间中实在让人觉得难以置信。

类似地，在软件系统中，允许任何人可以访问敏感或者私密的信息是不明智的。应用系统必须通过质询来验证用户身份，据此决定是允许还是拒绝他访问受限制的信息。无论你是通过用户名/密码来保护一个电子邮件账号，还是基于交易个人身份号码来保护一个佣金账户，安全性都是大多数应用系统的一个重要切面。

我们有意选择“切面”这个词来描述应用系统的安全性。安全性是超越应用系统功能特性的一个关注点。应用系统的绝大部分不应该亲自参与到与安全相关的处理中。尽管你能够把与安全相关的处理直接编码到应用系统中（这种情况并不少见），但更好的做法还是将安全有关的关注点与应用系统的关注点分开。

如果你想到这听上去好像安全性是通过 “面向切面”技术实现的，那你猜对了。在本章中，我们将向你介绍 Acegi 安全系统，并探讨使用 Spring AOP 和 Servlet 过滤器[1]来保护应用系统的各种手段。

11.1 Acegi 安全系统介绍

Acegi 是一个能够为基于 Spring 的应用系统提供描述性安全保护的安全框架。它提供了一组可以在 Spring 应用上下文中配置的 Bean，充分利用了 Spring 对依赖注入和面向切面编程的支持。

当保护 Web 应用系统时，Acegi 使用 Servlet 过滤器来拦截 Servlet 请求，以实施身份认证并强制安全性。并且，在第 11.4.1 节你将会看到，Acegi 采取了一种独特的机制来声明 Servlet 过滤器，使你可以使用 Spring IoC 注入它所依赖的其他对象。

Acegi 也能够通过保护方法调用在更底层的级别上强制安全性。使用 Spring AOP，Acegi 代理对象，将“切面”应用于对象，以确保用户只有在拥有恰当授权时才能调用受保护的方法。

无论你正在保护一个 Web 应用程序还是需要方法调用级别的安全性，Acegi 都是使用如图 11.1

所示的 4 个主要组件来实施安全性。

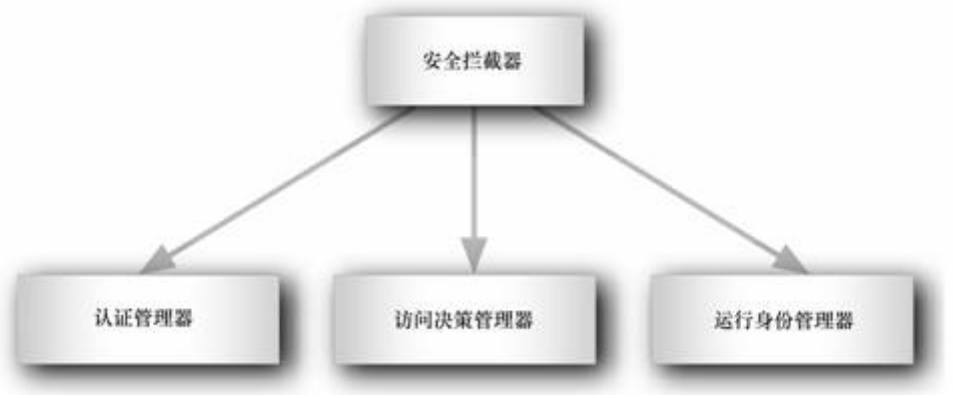

图 11.1　Acegi 安全的基本组件

通过本章，我们会揭示每一个组件的细节。但在开始考察 Acegi 安全机制的本质之前，首先让我们居高临下地考察一下每个组件扮演的角色。

11.1.1　安全拦截器

为了释放锁舌并打开门，你必须先把钥匙插到锁孔中，并恰当地拨动锁的制栓。如果钥匙和锁不匹配，就无法拨动制栓并释放锁舌。但如果你有正确的钥匙，所有的制栓就会接受这把钥匙，锁舌就会释放，从而允许你把门打开。

在 Acegi 中，可以认为安全拦截器像一把锁的锁舌，能够阻止对应用系统中受保护资源的访问。为了释放“锁舌”并通过安全拦截器，你必须向系统提供“钥匙”（通常是一对用户名和密码）。“钥匙”会尝试拨开安全拦截器的“制栓”，从而允许你访问受保护的资源。

11.1.2　认证管理器

第一道必须打开的安全拦截器的制栓是认证管理器。认证管理器负责决定你是谁。它是通过考虑你的主体（通常是一个用户名）和你的凭证（通常是一个密码）做到这点的。

你的主体定义了你是谁，你的凭证是确认你身份的证据。如果你的凭证足以使认证管理器相信你的主体可以标识你的身份，Acegi 就能知道它在和谁打交道。

11.1.3　访问决策管理器

一旦 Acegi 决定了你是谁，它就必须决定你是否拥有访问受保护的资源的恰当授权。访问决策管理器是 Acegi 锁中第二道必须被打开的制栓。访问决策管理器进行授权，通过考虑你的身份认证信息和与受保护资源关联的安全属性决定是否让你进入。

例如，安全规则也许规定只有主管才允许访问某个受保护资源。如果你被授予主管权限，则第二道也是最后一道制栓——访问决策管理器——会被打开，并且安全拦截器会给你让路，让你取得受保护资源的访问权。

11.1.4　运行身份管理器

当你通过认证管理器和访问决策管理器，安全拦截器会被开启，门已经可以被打开。但在你转动门把手进入之前，安全拦截器也许还有一件事要做。

即使你已经通过身份认证并且已经获得了访问被保护资源的授权，门后也许还有更多的安全

限制在等着你。比如，你也许已被授权访问查看某个 Web 页面，但用于创建该页面的对象也许和页面本身有不同的安全需求。一个运行身份管理器可以用另一个身份替换你的身份，从而允许你访问应用系统内部更深处的受保护对象。

运行身份管理器的用处在大多数应用系统中是有限的。幸运的是，当你使用 Acegi 保护应用系统时可以不必使用甚至不必完全理解运行身份管理器。因此，我们认为运行身份管理器是一个高级课题，在下文中不再深入地探讨它。

现在，你已经看到了 Acegi 安全性的全貌，让我们回过头来看一下如何配置 Acegi 安全系统的每一个部分，首先由认证管理器开始。

11.2 管理身份验证

决定是否允许用户访问受保护资源的第一步是判断用户的身份。在大多数应用系统中，这意味着用户在一个登录屏上提供用户名和密码。用户名（或者主体）告诉应用系统用户声明自己是谁。为了确证用户的身份，用户需要同时提供一个密码（或凭证）。如果应用系统的安全机制确认密码是正确的，则系统假设用户的实际身份与他声明的身份相同。

在 Acegi 中，是由认证管理器负责确定用户身份的。一个认证管理器由接口 net.sf.acegisecurity.AuthenticationManager 定义：

```
public interface AuthenticationManager {
  public Authentication authenticate(Authentication authentication)
      throws AuthenticationException;
}
```

认证管理器的 authenticate()方法需要一个 net.sf.acegisecurity.Authentication 对象(其中可能只包括用户名和密码）作为参数，它会尝试验证用户身份。如果认证成功，authenticate()方法返回一个完整的 Authentication 对象,其中包括用户已被授予的权限(将由授权管理器使用)。如果认证失败，则它会抛出一个 AuthenticationException。

正如你所见到的，AuthenticationManager 接口非常简单，而且你可以相当容易地实现自己的 AuthenticationManager。但是 Acegi 提供了 ProviderManager，作为 AuthenticationManager 的一个适用于大多数情形的实现。所以，我们不讨论开发自己的认证管理器，而是看一下如何使用 ProviderManager。

11.2.1 配置 ProviderManager

ProviderManager 是认证管理器的一个实现，它将验证身份的责任委托给一个或多个认证提供者，如图 11.2 所示。

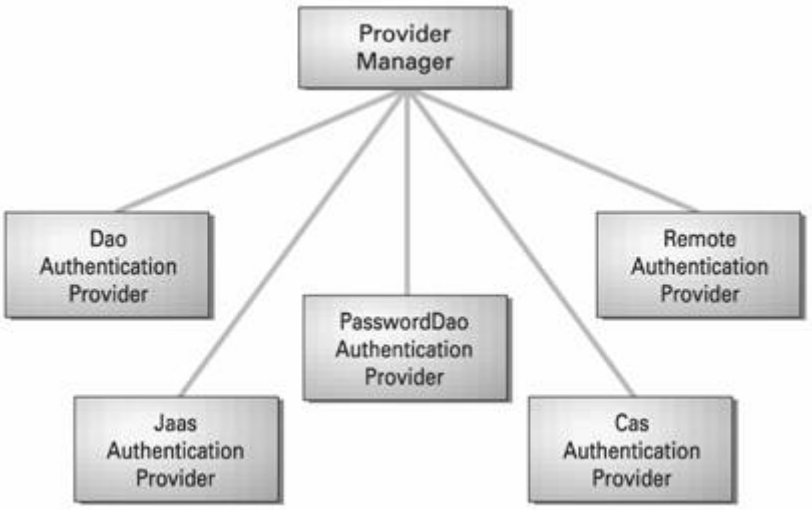

图 11.2　ProviderManager 将身份验证的职责委托给一个或多个认证提供者

ProviderManager 的思路是使你能够根据多个身份管理源来认证用户。它不是依靠自己实现身份验证，而是逐一遍历一个认证提供者的集合，直到某一个认证提供者能够成功地验证该用户的身份（或者已经尝试完了该集合中所有的认证提供者）。

你可以在 Spring 配置文件中按如下方式配置一个 ProviderManager：

```
<bean id="authenticationManager"
      class="net.sf.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref bean="daoAuthenticationProvider"/>
      <ref bean="passwordDaoProvider"/>
    </list>
  </property>
</bean>
```

通过 providers 属性可以为 ProviderManager 提供一个认证提供者的列表。通常你只需要一个认证提供者，但在某些情况下，提供由若干个认证提供者组成的列表是有用的。在这种情况下，如果一个认证提供者验证身份失败，可以尝试另一个认证提供者。一个认证提供者是由 net.sf.acegisecurity.provider.AuthenticationProvider 接口定义的。Spring 提供了若干个 AuthenticationProvider 的有用实现，如下表所列：

表 11.1　　Acegi 选择的认证提供者

认证提供者	目　　的
net.sf.acegisecurity.adapters. AuthByAdapterProvider	使用容器的适配器验证身份。

net.sf.acegisecurity.providers.cas.CasAuthenticationProvider	根据 Yale 中心认证服务验证身份。
net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider	从数据库中获取用户信息，包括用户名和密码。
net.sf.acegisecurity.providers.jaas.JaasAuthenticationProvider	从 JAAS 登录配置中获取用户信息。
net.sf.acegisecurity.providers.dao.PasswordDaoAuthenticationProvider	从数据库中获取用户信息，但让底层的数据源完成实际的身份验证。
net.sf.acegisecurity.providers.rcp.RemoteAuthenticationProvider	根据远程服务验证用户身份。
net.sf.acegisecurity.runas.RunAsImplAuthenticationProvider	针对身份已经被运行身份管理器替换的用户进行认证。
net.sf.acegisecurity.providers.TestingAuthenticationProvider	用于单元测试。自动认为一个 TestingAuthenticationToken 是有效的。不应用于生产环境。

你可以认为一个 AuthenticationProvider 是一个下属的 AuthenticationManager。事实上，AuthenticationProvider 接口也有一个 authenticate() 方法，该方法的签名与 AuthenticationManager 的 authenticate()方法完全一样。

在本节中，我们关注表 11.1 中列出的三个最常用的认证提供者。首先从使用 DaoAuthenticationProvider 进行简单的基于数据库验证身份开始。

11.2.2 根据数据库验证身份

大多数应用系统将包括用户名和密码在内的用户信息保存在数据库中。如果这和你的情况相符，则你会发现 Acegi 提供的以下两个认证提供者是有用的：

- DaoAuthenticationProvider；
- PasswordDaoAuthenticationProvider。

这两个认证提供者都能使你通过将用户的主体和密码与数据库记录进行比较来验证用户身份。两者的不同之处在于真正的身份验证是在哪里进行的。DaoAuthenticationProvider 使用 Dao 来获取用户名和密码，并使用它们来验证用户身份。而 PasswordDaoAuthenticationProvider 将身份验证的责任推给 Dao 自己完成。这是一个重要的区别，等到我们在 11.2.3 节中讨论 PasswordDaoAuthenticationProvider 时，这个区别会变

得更清楚。

在本节中，我们看一下如何使用 DaoAuthenticationProvider 根据保存在某个数据源（通常是关系数据库）中的用户信息进行简单的身份验证。在下一节中你将看到如何使用 PasswordDaoAuthenticationProvider 根据一个 LDAP（轻型目录访问协议）用户库进行身份验证。

声明一个 DAO 认证提供者

一个 DaoAuthenticationProvider 是一个简单的认证提供者，它使用 DAO 来从数据库中获取用户信息（包括用户的密码）。

取得了用户名和密码之后，DaoAuthenticationProvider 通过比较从数据库中获取的用户名和密码以及来自认证管理器的通过 Authentication 对象中传入的主体和凭证完成身份验证（见图 11.3）。如果用户名和密码与主体和凭证匹配，则用户通过身份验证，同时返回给认证管理器一个已完全填充的 Authentication 对象。否则会抛出一个 AuthenticationException，表明身份验证失败。

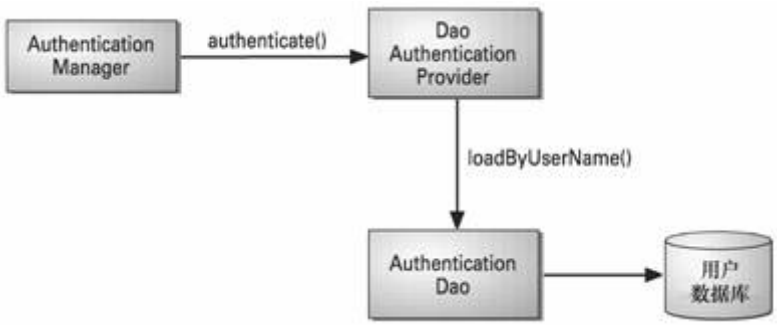

图 11.3　DaoAuthenticationProvider 通过从数据库中获取用户信息帮助认证管理器进行身份验证

配置一个 DaoAuthenticationProvider 再简单不过了。下一段 XML 摘要显示了如何声明一个 DaoAuthenticationProvider Bean，并且装配上它所依赖的 DAO。

```
<bean id="authenticationProvider" class="net.sf.acegisecurity.
          providers.dao.DaoAuthenticationProvider">
  <property name="authenticationDao">
    <ref bean="authenticationDao"/>
  </property>
</bean>
```

属性 authenticationDao 指定了一个用于从数据库中获取用户信息的 Bean。这个属性期望赋予一个 net.sf.acegisecurity.providers.dao.AuthenticationDao 的实例。接下来的问题就是该如何配置 authenticationDao Bean 了。

Acegi 提供了两个可供选择的 AuthenticationDao 的实例：InMemoryDaoImpl 和 JdbcDaoImpl。

我们首先配置一个 InMemoryDaoImpl 作为 authenticationDao Bean 的实例，然后再使用更实用的 JdbcDaoImpl 替换它。

使用内存 DAO

尽管假定 AuthenticationDao 对象总是通过查询关系数据库获取用户信息是一种自然的想法，事实情形却不必总是如此。如果你的应用系统的身份验证需求是微不足道的，或者是为了开发期间方便起见，也许更简单的做法是在 Spring 配置文件中直接配置你的用户信息。

为此，Acegi 提供了 InMemoryDaoImpl，一个从 Spring 配置文件中获取用户信息的 AuthenticationDao。你能够在 Spring 配置文件中通过以下方式配置一个 InMemoryDaoImpl：

```
<bean id="authenticationDao" class="net.sf.acegisecurity.
          providers.dao.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      palmerd=4moreyears,ROLE_PRESIDENT
      bauerj=ineedsleep,ROLE_FIELD_OPS,ROLE_DIRECTOR
      myersn=traitor,disabled,ROLE_CENTRAL_OPS
    </value>
  </property>
</bean>
```

属性 userMap 使用一个 net.sf.acegisecurity.providers.dao.memory.UserMap 对象来定义一组用户名、密码和权限。幸运的是，当装配一个 InMemoryDaoImpl 时，你不必为配置一个 UserMap 实例而操心，因为 Acegi 提供了一个属性编辑器，它能够帮你把一个字符串转化为一个 UserMap 对象。

userMap 字符串的每一行都是一个名字—值对，其中名字是用户名，值是一个由逗号分隔的列表，它以用户密码开头，后面跟着一个或多个赋予该用户的权限的名字（可以将权限看作角色）。

Myersn=traitor,
disabled,ROLE_CENTRAL_OPS

以上的 authenticationDao 声明中定义了三个用户：palmerd、bauerj 和 myersn。这三个用

户的密码分别是 4moreyears、ineedsleep、和 traitor。用户 palmerd 被定义为拥有权限 ROLE_PRESIDENT，bauerj 被赋予权限 ROLE_FIELD_OPS 和 ROLE_DIRECTOR，并且用户 myersn 被给予 ROLE_CENTRAL_OPS 授权。

注意用户 myersn 的密码后面有 disabled 这个单词。这是一个特殊的标志，表明该用户已被禁用。

InMemoryDaoImpl 有明显的局限性。最主要的一点是，对安全性进行管理时要求你重新编辑 Spring 的配置文件并且重新部署应用。虽然在开发环境下这是可以接受的（而且可能还是有帮助的），但对于生产用途而言这种做法就太笨拙了。因此，我们强烈反对在生产环境下使用 InMemoryDaoImpl，而是应该考虑使用 JdbcDaoImpl。

声明一个 JDBC DAO

JdbcDaoImpl 是一个简单而灵活的认证 DAO。以它最简单的形式，只需要一个 javax.sql.DataSource 对象的引用，可以通过以下方式在 Spring 配置文件中进行声明：

```
<bean id="authenticationDao"
      class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

JdbcDaoImpl 假设你在数据库中已经建立了某些用于存放用户信息的表。特别地，它假设有一张“Users”表和一张“授权”表，如图 11.4 所示。

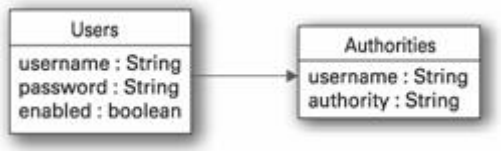

图 11.4　JdbcDaoImpl 假设的数据库表

当 JdbcDaoImpl 查找用户信息时，它会使用“SELECT username，password，enabled FROM users WHERE username = ?”作为查询语句。类似地，当查找授权时，它会使用“SELECT username，authority FROM authorities WHERE username = ?”。

尽管 JdbcDaoImpl 假定的表结构非常直接，它们很可能与你已经为应用系统建立的表结构不一致。比如，在 Spring 培训应用中，Student 表保存用户名（在 login 列中）和密码。是否这意味着你无法在 Spring 培训应用中使用 JdbcDaoImpl 来验证学生的身份？

当然不是。但你必须通过设置 usersByUserNameQuery 属性告诉 JdbcDaoImpl 如何找到用户信息。下面是对 authenticationDao Bean 的调整使它更适合 Spring 培训应用：

```
<bean id="authenticationDao"
     class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="usersByUserNameQuery">
    <value>SELECT login, password
           FROM student WHERE login=?</value>
  </property>
</bean>
```

现在 JdbcDaoImpl 知道如何在 Student 表中查找用户的认证信息了。但是，还有一件事遗漏了。Student 表中没有标志表明用户是使能的还是禁用的。事实上，我们一直假设所有的学生都是使能的。但我们如何告诉 JdbcDaoImpl 做同样的假设？

JdbcDaoImpl 还有一个 usersByUserNameMapping 属性，它引用一个 MappingSqlQuery 实例。正如你回忆起第 4 章中所介绍的，MappingSqlQuery 的 mapRow()方法将一个 ResultSet 中的字段映射为一个领域对象。对于 JdbcDaoImpl，提供给 usersByUserNameMapping 属性的 MappingSqlQuery 对象要求能够将一个 ResultSet（通过执行用户查询获得）转换为一个 net.sf.acegisecurity.UserDetails 对象。

UsersByUserNameMapping（程序清单 11.1）显示了一个 MappingSqlQuery 的实现，它适合将学生用户表的一个查询结果转换为一个 UserDetails 对象。它从 ResultSet 中抽取出 username 和 password，但总是设置 enabled 属性为 true。

程序清单 11.1　将学生查询的结果映射为一个 UserDetails 对象

```
public class UsersByUsernameMapping extends MappingSqlQuery {
  protected UsersByUsernameMapping(DataSource dataSource) {
    super(dataSource, usersByUsernameQuery);
    declareParameter(new SqlParameter(Types.VARCHAR));
    compile();
  }

  protected Object mapRow(ResultSet rs, int rownum)
      throws SQLException {
    String username = rs.getString(1);
    String password = rs.getString(2);
```

```
      UserDetails user = new User(username, password, true,
           new GrantedAuthority[]
                {new GrantedAuthorityImpl("HOLDER")});

      return user;
    }
  }
```

n

剩下惟一需要做的事就是声明一个 UsersByUsernameMapping Bean，并将它装配到 usersByUserNameMapping 属性中。以下的 authenticationDao Bean 的声明将一个内部 Bean 装配至 usersByUserNameMapping 属性中，从而可以应用新的用户映射：

```
  <bean id="authenticationDao"
       class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
    <property name="usersByUserNameQuery">
      <value>SELECT login, password
              FROM student WHERE login=?</value>
    </property>
    <property name="usersByUserNameMapping">
      <bean class=
         "com.springinaction.training.security.UsersByUsernameMapping"/>
    </property>
  </bean>
```

你也能改变 JdbcDaoImpl 查询用户权限的方式。与属性 usersByUserNameQuery 和 usersByUserNameMapping 定义 JdbcDaoImpl 如何查询用户认证信息相同，属性 authoritiesByUserNameQuery 和 authoritiesByUserNameMapping 告诉 JdbcDaoImpl 如何查询用户的权限：例如，你可以使用以下代码从 user_privileges 表中查询已授予一个用户的权限。

```
<bean id="authenticationDao"
     class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="usersByUserNameQuery">
    <value>SELECT login, password
           FROM student WHERE login=?</value>
  </property>
  <property name="usersByUserNameMapping">
    <bean class="com.springinaction.training.
         security.UsersByUsernameMapping"/>
  </property>
  <property name="authoritiesByUserNameQuery">
    <value>SELECT login, privilege
           FROM user_privileges where login=?</value>
  </property>
</bean>
```

你也可以将属性 authoritiesByUserNameMapping 设置成一个定制的 MappingSqlQuery 对象，从而可以定制权限查询的结果如何映射为一个 net.sf.acegisecurity.GrantedAuthority 对象。但是，由于默认的 MappingSqlQuery 对上面给出的查询来说已经足矣，我们就不再画蛇添足了。

使用加密的密码

默认地，DaoAuthenticationProvider 假设用户的密码是以明文方式（未加密的方式）存储的。但在与从数据库中取出的密码进行比较之前，可以使用一个密码编码器加密用户输入的明文密码。Acegi 提供了三个密码编码器：

- PlaintextPasswordEncoder（默认）——不对密码进行编码，直接返回未经改变的密码；
- Md5PasswordEncoder ——对密码进行消息摘要（MD5）编码；
- ShaPasswordEncoder ——对密码进行安全哈希算法（SHA）编码。

你可以通过设置 DaoAuthenticationProvider 的 passwordEncoder 属性改变它的密码编码器。例如，要使用 MD5 编码可以用以下代码：

```
<property name="passwordEncoder">
  <bean class=
    "net.sf.acegisecurity.providers.encoding.Md5PasswordEncoder"/>
```

```
</property>
```

你也需要设置编码器的种子源（salt source）。一个种子源为编码提供种子（salt），或者称编码的密钥。Acegi 提供两个种子源：

n　ReflectionSaltSource——使用用户的 User 对象中某个指定的属性来获取种子；

n　SystemWideSaltSource——对系统中所有用户使用相同的种子。

SystemWideSaltSource 适用于大多数情形。以下一段 XML 将一个 SystemWideSaltSource 装配到 DaoAuthenticationProvider 的 saltSource 属性中：

```
<property name="saltSource">
  <bean class=
    "net.sf.acegisecurity.providers.dao.SystemWideSaltSource">
    <property name="systemWideSalt">
      <value>123XYZ</value>
    </property>
  </bean>
</property>
```

ReflectionSaltSource 使用用户对象的某个特定属性作为用户密码的编码种子。由于这意味着每个用户的密码都会以不同的方式编码，因此更安全。若要装配一个 ReflectionSaltSource，可以通过如下方式将它装配到 saltSource 属性中：

```
<property name="saltSource">
  <bean class="net.sf.acegisecurity.
        providers.dao.ReflectionSaltSource">
    <property name="userPropertyToUse">
      <value>userName</value>
    </property>
  </bean>
</property>
```

在这里，用户的 userName 属性被用作种子来加密用户的密码。要特别重视的是必须保证种子是静态的，永远不会改变；否则，一旦种子改变，就再也不可能对用户身份进行验证了。

缓存用户信息

每次当请求一个受保护的资源时，认证管理器就被调用以获取用户的安全信息。但如果获取用户信息涉及到查询数据库，每次都查询相同的数据可能在性能上表现得很糟糕。注意到用

户信息不会频繁改变，也许更好的做法是在第一次查询时缓存用户信息，并在后续的查询中直接从缓存中获取用户信息。
DaoAuthenticationProvider 通过 net.sf.acegisecurity.providers.dao.UserCache 接口的实现类支持对用户信息进行缓存。

```
public interface UserCache {
  public UserDetails getUserFromCache(String username);
  public void putUserInCache(UserDetails user);
  public void removeUserFromCache(String username);
}
```

顾名思义，接口 UserCache 中方法提供了向缓存中放入、取得和删除用户明细信息的功能。写一个你自己的 UserCache 实现类是相当简单的。然而，在你考虑开发自己的 UserCache 实现类之前，应该首先考虑 Acegi 提供的两个方便的 UserCache 实现类：

- net.sf.acegisecurity.providers.dao.cache.NullUserCache
- net.sf.acegisecurity.providers.dao.cache.EhCacheBasedUserCache

NullUserCache 事实上不进行任何缓存。任何时候调用它的 getUserFromCache 方法，得到的返回值都是 null。这是 DaoAuthenticationProvider 使用的默认 UserCache 实现。
EhCacheBasedUserCache 是一个更实用的缓存实现。类如其名，它是基于开源项目 ehcache 实现的。ehcache 是一个简单快速的针对 Java 的缓存解决方案，同时也是 Hibernate 默认的和推荐的缓存方案。（关于 ehcache 的更多信息，请访问 ehcache 的网站 http://ehcache.sourceforge.net）
在 DaoAuthenticationProvider 中使用 ehcache 是很简单的，只需要简单地声明一个 EhCacheBasedUserCase Bean 即可：

```
<bean id="userCache"

class="net.sf.acegisecurity.providers.dao.cache.EhCacheBasedUserCache">
    <property name="minutesToIdle">15</property>
</bean>
```

属性 minutesToIdle 告诉缓存器一条用户信息在没有访问的情况下应该在缓存中保存多久。这里，我们设定在 15 分钟的非活动期后删除该条用户信息。
声明了 userCache Bean 之后，下面惟一要做的事就是把它装配到 DaoAuthenticationProvider 的 userCache 属性中：

```
<bean id="authenticationProvider"
          class="net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userCache">
    <ref bean="userCache"/>
  </property>
</bean>
```

11.2.3 根据 LDAP 仓库进行身份验证

DaoAuthenticationProvider 的工作方式是从数据库中获取用户的主体和凭据，并且与用户在登录时输入的主体和凭据进行比较。如果你希望由认证提供者最终负责认证决策，这是一种很好的方式。但也许你更希望将认证的职责委托给一个第三方的系统。

例如，根据 LDAP 服务器进行认证很常见。在这种情况下，是由 LDAP 服务器帮助应用系统完成身份验证的工作。应用系统自己甚至看不到用户存储的凭据。

PasswordDaoAuthenticationProvider 与 DaoAuthenticationProvider 有相似的目标，惟一的不同之处在于它只负责获取用户明细信息，实际的认证决策是委托给 DAO 完成的。而且，正如你将要看到的，在使用 LDAP 的情况下，DAO 进一步把身份验证委托给了 LDAP 服务器。

要使用 PasswordDaoAuthenticationProvider，你需要在 Spring 配置文件中以如下方式声明它：

```
<bean id="authenticationProvider" class="net.sf.acegisecurity.
          providers.dao.PasswordDaoAuthenticationProvider">
  <property name="passwordAuthenticationDao">
    <ref bean="passwordAuthenticationDao"/>
  </property>
</bean>
```

属性 passwordAuthenticationDao 装配了一个同名的 Bean 的引用。装配到这个属性的 Bean 是负责获取用户信息并进行身份验证的 DAO。这个 Bean 实现类必须实现 net.sf.acegisecurity.providers.dao.PasswordAuthenticationDao 接口：

```
public interface PasswordAuthenticationDao {
  public UserDetails loadUserByUsernameAndPassword(String username,
       String password) throws DataAccessException,
       BadCredentialsException;
}
```

这个接口与 AuthenticationDao 接口类似，惟一的不同之处在于，由于我们期望这个 DAO 不但要获取用户信息，还需要完成实际的身份验证，它的 loadUserByUsernameAndPassword()

方法需要一个 String 类型的 password 参数，并且在身份验证失败时会抛出一个 BadCredentialsException 异常。

与你在本章中将要看到的其他大多数 Acegi 接口不同的是，Acegi 的最新版本(0.6.1 版)没有提供 PasswordAuthenticationDao 接口的任何实用的实现。但是，你不需要付出很多努力去找一个实现类。在我们写作本章的时候，Acegi 的 CVS 沙箱[2]中包含了 LdapPasswordAuthenticationDao，一个提供 LDAP 身份验证的 PasswordAuthenticationDao 的实现类。它目前还不是 Acegi 的官方组成部分，但如果你想将它从沙箱中取出来并让它开始工作，只需要按照如下方式声明一个 passwordAuthenticationDao Bean 即可：

```
<bean id="passwordAuthenticationDao" class="net.sf.acegisecurity.
        providers.dao.ldap.LdapPasswordAuthenticationDao">
  <property name="host">
    <value>security.springinaction.com</value>
  </property>
  <property name="port">
    <value>389</value>
  </property>
  <property name="rootContext">
    <value>DC=springtraining,DC=com</value>
  </property>
  <property name="userContext">
    <value>CN=user</value>
  </property>
  <property name="rolesAttributes">
    <list>
      <value>memberOf</value>
      <value>roles</value>
    </list>
  </property>
</bean>
```

LdapPasswordAuthenticationDao 有一些用于指导它根据 LDAP 服务器进行身份验证的属性。其中，惟一必须指定的属性是 host，它指定了 LDAP 服务器的主机名。但你很可能需要调整其他一个或多个属性。

属性 port 指定了 LDAP 服务器监听的端口。该属性的默认值是 389（LDAP 的著名端口），但为了更好地说明 port 属性的含义，我们在这里显式地设置了端口的值。

属性 rootContext 代表 LDAP 的根上下文。该属性的默认值为空，所以你很可能需要重载它。下图显示了如何使用 rootContext 属性（以及 host 和 port 属性）来构造一个 LDAP 服务器的提供者 URL。

```
Ldap://security.springinaction.com:389/DC=springtraining,DC=com
```

属性 userContext 规定了保存用户信息的 LDAP 上下文。该属性的默认值是 CN=Users，但我们在这里用 CN=User 重载了这个值。rootContext、userContext 和 userName 共同构成用户的主体。

```
CN=bauerj, CN=user, DC=springtraining, DC=com
```

最后，属性 rolesAttributes 允许你列出一个或多个与 LDAP 条目关联的用于保存用户角色的属性。默认地，该属性只包括一项 memberOf，但我们在列表中增加了 roles 属性。

关于 roles 属性要注意的一个重要之处是当 LdapPasswordAuthenticationDao 从 LDAP 中获取属性时，它会自动为属性值加上 ROLE_前缀。当我们在第 11.3.2 节中讨论角色投票者时，你将会看到这个前缀的用途。

11.2.4 基于 Acegi 和 Yale CAS 实现单次登录

你有多少个密码？如果你和大多数人一样，你很可能使用十余个乃至更多的密码来登录日常访问的各种系统。保管所有这些密码是一个挑战，而被迫登录多个系统则令人厌烦。如果能够只登录一次就自动登录了所有需要使用的系统，那该有多好。

单次登录（Single Sign On，SSO）是一个热门的安全话题。这个名字就已经表达了一切：一次登录，访问一切。耶鲁大学技术和规划组已经创建了一个名为中心认证服务（Central Authentication Service，CAS）的优秀 SSO 解决方案，它可以和 Acegi 共同工作。

关于设置和使用 CAS 的细节远远超出了本书的范围。但我们将讨论 CAS 采用的基本身份验证方式，并探讨如何与 CAS 一起使用 Acegi。如果想知道关于 CAS 的更多信息，我们强烈推荐你访问 CAS 的主页 http://tp.its.yale.edu/tiki/tiki-index.php?page=CentralAuthenticationService。

为了理解 Acegi 在一个 CAS 身份验证应用中的作用，重要的是先理解一个典型的 CAS 身份验证场景是如何工作的。考虑一个请求受保护服务的流程，如图 11.5 所示。

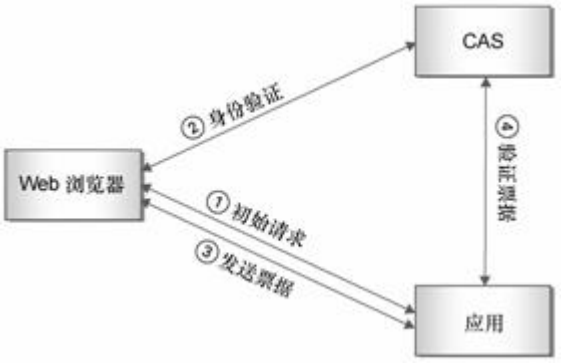

图 11.5　使用 Yale CAS 保护一个应用

当 Web 浏览器请求一个服务时①，服务通过在请求中寻找一个 CAS 票据来判断用户身份是否已通过验证。如果未找到票据，则意味着该用户尚未通过身份验证。作为结果，用户被重定向到 CAS 的登录页面②。

在 CAS 的登录页面，用户输入他/她的用户名和密码。如果 CAS 成功认证了该用户，则创建一个与请求的服务相关联的票据。接着，CAS 服务器将用户重定向到用户原先请求的服务（此时请求中已经有票据了）③。

服务再次在请求中寻找票据。这一次它找到了票据，并与 CAS 服务器联系以确认票据是有效的④。如果 CAS 的响应表明票据对当前请求的服务而言是有效的，服务就会允许用户访问应用。

以后，当用户请求访问另一个支持 CAS 的应用系统时，那个应用仍会与 CAS 联系。由于用户之前已经认证，CAS 会返回一个对该新应用有效的服务票据，而不会提示用户再次登录。

你应该理解的关于 CAS 的关键概念之一是：被保护的应用从不处理用户的凭证。当用户被提示登录应用系统时，他们实际上是登录到了 CAS 服务器。应用系统自己从来没有看到过用户的凭证。应用系统使用的惟一安全形式是通过询问 CAS 服务器来验证用户的票据。这意味着只有一个应用（即 CAS 服务器）负责处理用户认证，因此是非常好的解决方案。

当 Acegi 与 CAS 共同使用时，它承担着帮助应用系统向 CAS 服务器验证 CAS 票据的任务。这就使应用系统本身从 CAS 认证过程中解脱了出来。

Acegi 通过使用 CasAuthenticationProvider 来完成这一任务。这是一个不关心用户名和密码的认证提供者，它只接受 CAS 票据作为凭证。你可以在 Spring 配置文件中按如下方式配置一个 CasAuthenticationProvider Bean：

```
<bean id="casAuthenticationProvider"
        class="net.sf.acegisecurity.providers.cas.CasAuthenticationProvider">
  <property name="ticketValidator">
    <ref bean="ticketValidator"/>
```

```
    </property>
    <property name="casProxyDecider">
      <ref bean="casProxyDecider"/>
    </property>
    <property name="statelessTicketCache">
      <ref bean="statelessTicketCache"/>
    </property>
    <property name="casAuthoritiesPopulator">
      <ref bean="casAuthoritiesPopulator"/>
    </property>
    <property name="key">
      <value>some_unique_key</value>
    </property>
  </bean>
```

正如你所看到的，CasAuthenticationProvider 是通过和其他若干个 Bean 相互合作来完成它的工作的。这些 Bean 中的第一个是 ticketValidator Bean，它被装配到 ticketValidator 属性中。在 Spring 配置文件中它是这样声明的：

```
  <bean id="ticketValidator" class="net.sf.acegisecurity.
          providers.cas. ticketvalidator.CasProxyTicketValidator">
    <property name="casValidate">
      <value>https://localhost:8443/cas/proxyValidate</value>
    </property>
    <property name="serviceProperties">
      <ref bean="serviceProperties"/>
    </property>
  </bean>
```

CasProxyTicketValidator 通过联系 CAS 服务器来验证 CAS 服务票据。属性 casValidate 指定了 CAS 服务器处理验证请求的 URL。
配置中引用的 serviceProperities Bean 中包含了与 CAS 相关的 Bean 的重要配置信息：

```
  <bean id="serviceProperties"
       class="net.sf.acegisecurity.ui.cas.ServiceProperties">
    <property name="service">
```

```
    <value>https://localhost:8443/training/
              j_acegi_cas_security_check</value>
  </property>
</bean>
```

属性 service 指定了一个 URL，CAS 在用户成功登录之后应该将用户重定向至该 URL。以后，在第 11.4.3 节中，你会看到该 URL 是如何被服务的。

回到 casAuthenticationProvider Bean，属性 casProxyDecider 装配了一个指向 casProxyDecider Bean 的引用，即一个到类型为 net.sf.acegisecurity.providers.cas.CasProxyDecider 的 Bean 的引用。为了理解 casProxyDecider Bean 的作用，你必须理解 CAS 如何支持代理服务。

CAS 支持代理服务的概念，代理服务帮助另一个应用程序实现用户的身份验证。一个典型的代理服务的例子是门户，门户帮助由它所代表的 portlet 应用程序完成用户身份验证。当用户登录到一个门户时，门户通过代理票据也确保用户隐含地登录到它包含的应用系统中。

CAS 如何处理代理票据是一个高级话题。我们建议你查阅 CAS 的文档（http://tp.its.yale.edu/tiki/tiki-index.php?page=CasTwoOverview）获取关于代理票据的更详细的信息。在这里，我们只需要指出 CasProxyDecider 负责决定是否接受代理票据。Acegi 提供了 CasProxyDecider 的三个实现类：

- AcceptAnyCasProxy——接受来自任何服务的代理请求；
- NamedCasProxyDecider——接受来自一个已命名服务的列表的代理请求；
- RejectProxyTickets——拒绝任何代理请求。

为简单起见，让我们假设你的应用系统不涉及代理服务。在这种情况下，RejectProxyTicket 就成为对 casProxyDecider Bean 来说最合适的 CasProxyDecider：

```
<bean id="casProxyDecider"class="net.sf.acegisecurity.
          providers.cas.proxy.RejectProxyTickets"/>
```

属性 statelessTicketCache 用于支持无状态的客户端（比如远程服务的客户端），它们无法在 HttpSession 中存储 CAS 票据。不幸的是，即使没有无状态客户端要访问你的应用系统，statelessTicketCache 属性也是必不可少的。Acegi 仅仅提供了一个实现，所以声明一个 statelessTicketCache 相当简单：

```
<bean id="statelessTicketCache"class="net.sf.acegisecurity.
          providers.cas.cache.EhCacheBasedTicketCache">
  <property name="minutesToIdle"><value>20</value></property>
</bean>
```

最后一个与CasAuthenticationProvider协同工作的Bean是casAuthoritiesPopulator Bean。作为一个SSO实现，CAS只负责身份验证——它不关心权限是如何分配给用户的。为了弥补这一差距，你需要一个net.sf.acegisecurity.providers.cas.CasAuthoritiesPopulator Bean。Acegi只提供了一个CasAuthoritiesPopulator的实现。DaoCasAuthoritiesPopulator使用一个认证DAO（如11.2.2节中讨论的）从数据库中加载用户明细信息。可以这样声明一个casAuthoritiesPopulator Bean：

```
<bean id="casAuthoritiesPopulator" class="net.sf.acegisecurity.
        providers.cas.populator.DaoCasAuthoritiesPopulator">
  <property name="authenticationDao">
    <ref bean="inMemoryDaoImpl"/>
  </property>
</bean>
```

最后，CasAuthenticationManager的key属性指定了一个 String 值，认证管理器使用该字符串来识别之前已经认证的标志。你可以把这个属性设为任意值。

关于使用CAS和Acegi实现SSO还有比CasAuthenticationManager更多的内容。我们仅仅讨论了CasAuthenticationProvider是如何进行身份验证的。在第11.4.3节中，你将看到当CasAuthenticationManager验证用户身份失败时，是如何将用户重定向到CAS登录页面的。但就现在而言，让我们先看一下Acegi是如何判断一个已通过身份验证的用户是否拥有访问受保护资源所需的恰当权限的。

11.3 控制访问

身份验证只是Acegi安全保护机制的第一步。一旦Acegi知道用户的身份，它必须决定是否允许用户访问由它保护的资源。这就引出了访问决策管理器。

正如认证管理器负责确定用户的身份，访问决策管理器负责决定用户是否有恰当的权限访问受保护的资源。一个访问决策管理器是由net.sf.acegisecurity.AccessDecisionManager接口定义的：

```
public interface AccessDecisionManager {
  public void decide(Authentication authentication, Object object,
       ConfigAttributeDefinition config)
       throws AccessDeniedException;
  public boolean supports(ConfigAttribute attribute);
  public boolean supports(Class clazz);
}
```

supports()方法根据受保护资源的类以及它的配置属性（受保护资源的访问需求）判断该访问管理器是否能够做出针对该资源的访问决策。decide()方法是完成最终决策的地方。如果它没有抛出 AccessDeniedException 而返回，则允许访问受保护的资源。否则，访问被拒绝。

11.3.1 访问决策投票

编写一个你自己的 AccessDecisionManager 看上去是非常简单的。但是，为什么做那些你本来用不着亲自做的事？Acegi 提供了适用于大多数情形的 AccessDecisionManager 的三个实现类：

- net.sf.acegisecurity.vote.AffirmativeBased
- net.sf.acegisecurity.vote.ConsensusBased
- net.sf.acegisecurity.vote.UnanimousBased

这三个访问决策管理器的名字都相当奇怪，但当你考察过 Acegi 的授权策略之后就会明白它们的意思。

Acegi 的访问决策管理器负责最终决定一个通过身份验证的用户是否拥有访问权限。然而，它们不是完全靠自己而完成这一决策的，而是通过征询一个或多个对某用户是否有权访问受保护资源进行投票的对象。一旦获得所有的投票结果，决策管理器统计得票情况，并完成最终决策。

区分不同的访问决策管理器的是它们如何计算出最终的决策。表 11.2 描述了每一个认证决策管理器是如何决定是否允许访问的。

表 11.2 Acegi 的访问决策管理器如何计票

访问决策管理器	如 何 决 策
AffirmativeBased	当至少有一个投票者投允许访问票时允许访问
ConsensusBased	当所有投票者都投允许访问票时允许访问
UnanimousBased	当没有投票者投拒绝访问票时允许访问

在 Spring 配置文件中，所有的访问决策管理器都是以相同的方式进行配置的。例如，以下一段 XML 摘要配置了一个 UnanimousBased 访问决策管理器：

```
<bean id="accessDecisionManager"
    class="net.sf.acegisecurity.vote.UnanimousBased">
  <property name="decisionVoters">
    <list>
      <ref bean="roleVoter"/>
    </list>
  </property>
</bean>
```

你可以通过 decisionVoters 属性为访问决策管理器提供一组投票者。在上述情况中，只有一个投票者，它引用了一个名为 roleVoter 的 Bean。让我们看一下 roleVoter 是如何配置的。

11.3.2 决定如何投票

尽管访问决策投票者对是否授权访问某个受保护资源没有最终发言权，它们在访问决策过程中扮演了重要的角色。一个访问决策投票者的工作是同时考虑用户已拥有的授权和受保护资源的配置属性中要求的授权。基于这一信息，访问决策投票者通过投票为访问决策管理器做出决策提供支持。

一个访问决策投票者是任何实现了 net.sf.acegisecurity.vote.AccessDecisionVoter 接口的对象：

```
public interface AccessDecisionVoter {
  public static final int ACCESS_GRANTED = 1;
  public static final int ACCESS_ABSTAIN = 0;
  public static final int ACCESS_DENIED = -1;

  public boolean supports(ConfigAttribute attribute);
  public boolean supports(Class clazz);
  public int vote(Authentication authentication, Object object,
       ConfigAttributeDefinition config);
}
```

可以看到，AccessDecisionVoter 接口和 AccessDecisionManager 接口非常相似。最大的区别在于，AccessDecisionVoter 没有 AccessDecisionManager 接口的返回类型为 void 的 decide()方法，而是有一个返回 int 的 vote()方法。这是因为访问决策投票者并不决定是否允许访问，它仅仅就是否允许访问投出它自己的一票。

在获得投票机会时，访问决策投票者能够以下面三种方式进行投票：

- ACCESS_GRANTED——投票者希望允许访问受保护的资源
- ACCESS_DENIED——投票者希望拒绝对受保护资源的访问
- ACCESS_ABSTAIN——投票者不关心

与大多数 Acegi 的组件相同，你可以自由地编写自己的 AccessDecisionVoter 的实现类。然而，Acegi 提供了一个很实用的投票者实现类 RoleVoter，它当受保护资源的配置属性代表一个角色时进行投票。说得更具体一些，RoleVoter 当受保护资源有一个名字由 ROLE_开始的配置属性时参与投票。

RoleVoter 决定投票结果的方式是简单地将受保护资源的所有配置属性（以 ROLE_作为前缀）与认证用户的所有授权进行比较。如果 RoleVoter 发现其中有一个是匹配的，则它投

ACCESS_GRANTED 票。否则，它将投 ACCESS_DENIED 票。
RoleVoter 只在访问所需的授权不是以 ROLE_为前缀时放弃投票。例如，如果受保护的资源仅仅需要一个非角色的授权（诸如 CREATE_USER），则 RoleVoter 将放弃投票。
你可以在 Spring 配置文件通过以下方式配置一个 RoleVoter：

```
<bean id="roleVoter"
      class="net.sf.acegisecurity.vote.RoleVoter"/>
```

如前所述，RoleVoter 只在受保护资源有以 ROLE_为前缀的配置属性才进行投票。然而，ROLE_前缀只是默认值。你可以选择通过设置 rolePrefix 属性来重载这个默认前缀：

```
<bean id="roleVoter"
      class="net.sf.acegisecurity.vote.RoleVoter">
  <property name="rolePrefix">
    <value>GROUP_</value>
  </property>
</bean>
```

在这里，默认的前缀被重载为 GROUP_。因此，这个 RoleVoter 现在将只针对以 GROUP_为前缀的权限进行授权投票。

11.3.3 处理投票弃权

在知道了任何一个投票者都可能投允许、拒绝或弃权票之后，现在你可能会问：如果所有投票者都投弃权票，会发生什么？用户究竟会被授权还是拒绝访问？
默认地，当全部投票者都投弃权票时，所有的访问决策管理者都将拒绝访问资源。然而，你可以通过设置访问决策管理者的 allowIfAllAbstain 属性为 true 来重载这个默认行为：

```
<bean id="accessDecisionManager"
      class="net.sf.acegisecurity.vote.UnanimousBased">
  <property name="decisionVoters">
    <list>
      <ref bean="roleVoter"/>
    </list>
  </property>
  <property name="allowIfAllAbstain">
    <value>true</value>
  </property>
```

</bean>

通过设置 allowIfAllAbstain 为 true，你建立了一个“沉默即同意”的策略。换句话说，如果所有的投票者都放弃投票，则如同它们都投赞成票一样，访问被授权。

现在，你已经看到 Acegi 的身份验证和访问控制管理器是如何工作的了，让我们把它们投入使用。在下一节，你将看到如何使用 Acegi 的一组 Servlet 过滤器来保护一个 Web 应用程序。再接下来，在第 11.5 节，我们将深入到一个应用系统内部，考察如何使用 Spring AOP 在方法调用的级别上实施保护。

11.4 保护 Web 应用程序

Acegi 对 Web 安全性的支持大量地依赖 Servlet 过滤器。这些过滤器拦截对应用系统的请求，并且在应用系统真正处理请求之前进行某些安全处理。Acegi 提供了若干个过滤器，能够拦截 Servlet 请求，并将这些请求转给认证和访问决策管理器处理，从而强制安全性。根据你的需要，你可以使用多达 6 个的过滤器来保护你的应用系统。表 11.3 描述了 Acegi 的每个过滤器。

表 11.3 Acegi 的 Servlet 过滤器

过 滤 器	目 的
通道处理过滤器	确保请求是在安全通道（比如 HTTPS）之上传输的
认证处理过滤器	接受认证请求，并将它们转交给认证管理器进行身份验证
CAS 处理过滤器	接受 CAS 服务票据，验证 Yale CAS 是否已经对用户进行了认证
HTTP 基本授权过滤器	处理使用 HTTP 基本认证的身份验证请求
集成过滤器	处理认证信息在请求间的存储（比如在 HTTP 会话中）
安全强制过滤器	确保用户已经认证，并且满足访问一个受保护 Web 资源的权限需求

当一个请求被提交到一个由 Acegi 保护的 Web 应用时，它是按照如下顺序（参见图 11.6）逐一通过 Acegi 过滤器的：

1. 如果配置了一个通道处理过滤器，则由它首先处理请求。通道处理过滤器将检查请求的传输通道（通常为 HTTP 或 HTTPS）并判断通道是否满足安全需求。如果不满足，则请求被重定向到同样的 URL，但修改通道以使之与安全需求相符。

2. 接下来，某个认证处理过滤器（其中包括 CAS 处理过滤器和 HTTP 基本授权过滤器）将决定该请求是否是一个认证请求。如果是，则它会从请求中取得恰当的用户信息（通常为用户名/密码），并转交给认证管理器来决定用户的身份。如果这不是一个认证请求，则请求会继续沿着过滤器链向下移动。

图 11.6　流经每一个 Acegi 过滤器的请求

3．集成过滤器试图从某个由它在请求间保持的位置（通常为 HTTP 会话）中获取用户的认证信息。如果能够找到用户的认证信息，则该信息被放置到一个 Context-Holder 对象中（基本上就是一个 ThreadLocal），以方便所有 Acegi 组件获取。

4．最后，安全强制过滤器最终决定是否授权用户访问被保护的资源。首先，安全强制过滤器询问认证管理器。如果用户还没有认证成功，则安全强制过滤器会将用户重定向到认证入口点（例如一个登录页面）。下一步，安全强制过滤器将询问访问决策管理器，以决定用户是否拥有合适的可以访问被保护资源的权限。如果没有，则返回一个 HTTP 403（禁止访问）消息至用户浏览器。

5．如果用户成功通过了安全强制过滤器的检查，则他/她将可以访问被保护的 Web 资源。

我们将逐一仔细考察这些过滤器。但在你能够开始使用它们之前，你需要首先学习 Acegi 是如何对 Servlet 过滤器使用一个与 Spring 类似的技巧。

11.4.1　代理 Acegi 的过滤器

如果你曾经使用过 Servlet 过滤器，你知道要让它们生效，必须在 Web 应用的 web.xml 文件中使用<filter>和<filter-mapping>元素配置它们。虽然这样做未尝不可，但它与 Spring 的通过依赖注入的方式配置组件的方式是不一致的。

例如，假设你在 web.xml 文件中声明了以下过滤器：

```
<filter>
  <filter-name>Foo</filter-name>
  <filter-class>FooFilter</filter-class>
</filter>
```

现在，假设 FooFilter 需要引用一个 Bar Bean 来完成它的工作。你该如何向 FooFilter 中注入 Bar 的一个实例呢？简单的回答是：你无法做到这一点。web.xml 文件没有依赖注入的概念，

也没有直接的方式可以从 Spring 应用上下文中获取 Bean 并装配到 Servlet 过滤器中。惟一可以选择的办法是使用 Spring 的 WebApplicationContextUtils 从 Spring 上下文中获取“bar" Bean：

```
ApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
Bar bar = (Bar) ctx.getBean("bar");
```

但这种方法的问题是你必须在 Servlet 过滤器中编写 Spring 特定的代码。此外，你最终在程序中硬编码了 bar Bean 的名字。

好在 Acegi 通过 FilterToBeanProxy 提供了一种更好的方式。FilterToBeanProxy 是一个特殊的 Servlet 过滤器，它本身做的事并不多，而是将自己的工作委托给 Spring 应用上下文中的另一个 Bean 来完成。被委托的 Bean 和其他的 Servlet 过滤器一样，实现了 javax.servlet.Filter 接口，但它是通过 Spring 配置文件而不是 web.xml 配置的。

通过使用 FilterToBeanProxy，你能够在 Spring 中配置真正的过滤器，充分利用 Spring 依赖注入的优势。如图 11.7 所示，web.xml 文件只包含 FilterToBeanProxy 的<filter>声明。真正的 FooFilter 是在 Spring 配置文件中配置的，并且它使用了 setter 注入将一个 Bar Bean 的引用设置到了 bar 属性中。

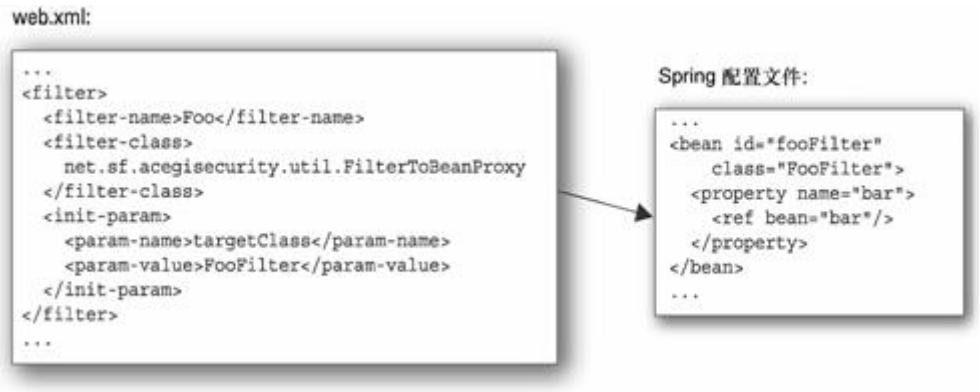

图 11.7　FilterToBeanProxy 代理过滤器处理对 Spring 应用上下文中的一个委托过滤器 Bean

要使用 FilterToBeanProxy，必须在 web.xml 文件中建立一个<filter>条目。例如，如果要使用 FilterToBeanProxy 配置一个 FooFilter，你可能使用以下代码：

```
<filter>
  <filter-name>Foo</filter-name>
  <filter-class>net.sf.acegisecurity.util.
        FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
```

```
      <param-value>
        FooFilter
      </param-value>
    </init-param>
  </filter>
```

在这里，targetClass 初始化参数被设置为被委托的过滤器 Bean 的完整类名。当这个 FilterToBeanProxy 过滤器被初始化时，它会在 Spring 上下文中找一个类型为 FooFilter 的 Bean，并且把自己的过滤工作委托给在 Spring 上下文中找到的 FooFilter Bean 来完成：

```
  <bean id="fooFilter"
      class="FooFilter">
    <property name="bar">
      <ref bean="bar"/>
    </property>
  </bean>
```

如果找不到 FooFilter Bean，则会抛出一个异常。如果找到了多于一个的相匹配的 Bean，则会使用第一个 Bean。
可选地，你也可以通过设置 targetBean 初始化参数而不是 targetClass 从 Spring 上下文中选取一个特定的 Bean。例如，通过以下方式设置 targetBean 属性，你可以根据名字选取 fooFilter Bean：

```
  <filter>
    <filter-name>Foo</filter-name>
    <filter-class>net.sf.acegisecurity.
                        util.FilterToBeanProxy</filter-class>
    <init-param>
      <param-name>targetBean</param-name>
      <param-value>fooFilter</param-value>
    </init-param>
  </filter>
```

targetBean 初始化参数使你能够更具体地指定将过滤工作委托给哪个 Bean 来完成，但需要你精确匹配 web.xml 和 Spring 配置文件中受委托的 Bean 的名字。当你决定重命名 Bean 时，这会带来额外的工作量。出于这个原因，使用 targetClass 而不是 targetBean 很可能是更好的

选择。
无论你选择 targetClass 还是 targetBean，FilterToBeanProxyx 必须能够访问 Spring 应用上下文。这就意味着必须使用 Spring 的 ContetLoaderListener 或 ContextLoaderServlet（参见第 8 章）加载 Spring 上下文。
最后，你可能需要将过滤器与一个 URL 模式相关联。下面的<filter-mapping>将 FilterToBeanProxy 的 Acegi-Authentication 实例与 URL 模式/*相关联，从而所有的请求都将被该过滤器处理：

```
<filter-mapping>
  <filter-name>Acegi-Authentication</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

/*是所有 Acegi 过滤器推荐的 URL 模式。这样做的意图是 Acegi 应该拦截所有请求，并由其依赖的安全管理器来决定是否和如何保护该请求。

注意：值得注意的是，FilterToBeanProxy 中并没有任何与 Acegi 或与保护 Web 应用程序相关的特定部分。你会发现当配置你自己的 Servlet 过滤器时，FilterToBeanProxy 也是非常有用的。事实上，由于它是如此有用，在 Spring 开发者邮件列表的一些讨论中，有人提议可以把 FilterToBeanProxy 从 Acegi 中移出，并放到核心 Spring 项目的某个后续版本中。

现在你已经知道如何使用 FilterToBeanProxy 了，你已经准备就绪，可以开始用它来建立与 Acegi 安全相关的 Web 组件了。让我们从 Acegi 安全的主要过滤器——安全强制过滤器——开始。

11.4.2 强制 Web 安全性

每当用户请求你的 Web 应用中的一个页面时，这个页面可能是受保护的，也可能不是。在 Acegi 中，一个安全强制过滤器负责拦截请求，判断请求是否安全，并且给予认证和访问决策管理器一个机会来验证用户的身份和权限。在 Spring 配置文件中，安全强制过滤器是通过以下方式在 Spring 配置文件中声明的：

```
<bean id="securityEnforcementFilter" class="net.sf.acegisecurity.
          intercept.web.SecurityEnforcementFilter">
  <property name="securityInterceptor">
    <ref bean="securityInterceptor"/>
  </property>
  <property name="authenticationEntryPoint">
```

```
    <ref bean="authenticationEntryPoint"/>
  </property>
</bean>
```

在这里，SecurityEnforcementFilter 装配了另外两个 Bean 的引用：authenticationEntryPoint 和 securityInterceptor。我们将在稍候更详细地讨论 authenticationEntryPoint 属性。现在，让我们首先关注 securityInterceptor 属性。

使用一个过滤器安全拦截器

属性 securityInterceptor 装配了一个指向同名的 Bean 的引用。如果你回想起本章开始时的门锁比喻，安全拦截器就如同开门前必须被释放的锁舌。正是它使认证管理器、访问决策管理器和运行身份管理器能够协同工作。

为了保证 Web 安全，Acegi 的 FilterSecurityInterceptor 类负责执行安全拦截器的工作。它是通过以下方式在 Spring 配置文件中声明的：

```
<bean id="securityInterceptor" class="net.sf.acegisecurity.
        intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref bean="accessDecisionManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      \A/admin/.*\Z=ROLE_ADMIN
      \A/student/.*\Z=ROLE_STUDENT,ROLE_ALUMNI
      \A/instruct/.*\Z=ROLE_INSTRUCTOR
    </value>
  </property>
</bean>
```

在这里装配的前两个属性是指向本章之前定义的认证管理器和访问决策管理器 Bean 的引用。安全拦截器使用认证管理器来决定用户是否已登录，并获取用户已拥有的授权。它使用访问决策管理器来判断用户是否拥用合适的授权来访问受保护的资源。

属性 objectDefinitionSource 告诉安全拦截器被拦截的各种请求所需要的授权是什么。该属

性有一个对应的属性编辑器，因此可以方便地作为一个字符串值进行配置。该字符串包括若干行，每一行可以是一个指令，或者一个从URL到授权的映射。

从上面的XML摘要中可以看到，属性objectDefinitionSource的值的第一行是一个指令，表明在比较请求的URL在和紧跟其后定义的模式之前必须首先正规化为小写字母。该属性的其余几行将URL模式映射为允许用户访问这些URL时必须授予用户的权限。可以看到，URL模式是以正则表达式的形式描述的。因此，securityInterceptor Bean的objectDefinitionSource属性定义了：

n /admin/reports.htm 要求用户拥有 ROLE_ADMIN 权限；

n /student/manageSchedule.htm 要求用户拥有 ROLE_STUDENT 或 ROLE_ALUMNI 权限；

n /instruct/postCourseNotes.htm 要求用户拥有 ROLE_INSTUCTOR 权限。

如果愿意，你也可以使用与Ant相似的URL模式而不是正则表达式，只需在objectDefinitionSource属性中增加一条PATTERN_TYPE_APACHE_ANT指令即可。例如，以下的objectDefinitionSource属性定义与上面给出的那个定义是等价的：

```
<property name="objectDefinitionSource">
  <value>
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
    PATTERN_TYPE_APACHE_ANT
    /admin/**=ROLE_ADMIN
    /student/**=ROLE_STUDENT,ROLE_ALUMNI
    /instruct/**=ROLE_INSTRUCTOR
  </value>
</property>
```

与所有Acegi的过滤器一样，安全强制过滤器是一个由FilterToBeanProxy作为前置的过滤委托Bean。这意味着配置一个安全强制过滤器的第一步是在应用的web.xml文件中为FilterToBeanProxy增加<filter>和<filter-mapping>元素：

```
<filter>
  <filter-name>Acegi-Security</filter-name>
  <filter-class>net.sf.acegisecurity.util.
        FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetBean</param-name>
    <param-value>securityEnforcementFilter</param-value>
  </init-param>
```

```
</filter>
…
<filter-mapping>
  <filter-name>Acegi-Security</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

值得注意的是，<filter-mapping>的<url-pattern>将所有请求映射到安全强制过滤器之上。这是 Acegi 过滤器配置的典型做法。这样做的意图是对所有的请求进行过滤，由安全强制过滤器通过对象定义源属性来决定是否需要进行实际的处理。

注意：本节由始至终，你会在 web.xml 中增加若干个<filter>和<filter-mapping>元素，它们都使用 FilterToBeanProxy。尽管过滤器类都是一样的，但不要认为它们之间可以相互替代。尽管这些过滤器都使用同一个 FilterToBeanProxy 类，它们服务于不同的目的，委托 Spring 应用上下文中不同的 Bean 来完成各自的工作。除非特别说明，它们都是 Acegi 的安全保护机制中不可或缺的部分。

假设用户提交了一个到指定受到保护的页面的请求。如果用户已经通过身份验证并且拥有恰当的权限，则安全强制过滤器会允许用户访问该页面。但如果用户尚未通过身份验证呢？

11.4.3 处理登录

回想一下，上一节的安全强制过滤器装配了一个指向 authenticationEntryPoint 的引用。当安全强制过滤器判断出一个用户尚未通过身份验证时，会将控制转交到一个认证入口点。

认证入口点的主要目的是提示用户登录。Acegi 提供了三个现成的认证入口点：

- BasicProcessingFilterEntryPoint——通过向浏览器发送一个 HTTP 401（未授权）消息，由浏览器弹出登录对话框，提示用户登录；
- AuthenticationProcessingFilterEntryPoint——将用户重定向到一个基于 HTML 表单的登录页面；
- CasProcessingFilterEntryPoint——将用户重定向至一个 Yale CAS 登录页面。

无论使用哪种认证入口点，都会提示用户需要提供用户名/密码来标识他/她自己的身份。当用户提交用户名和密码之后，Acegi 需要通过某种方式给予认证管理器一个验证用户身份的机会。

处理认证请求的工作由认证处理过滤器负责完成。Acegi 提供了三个认证处理过滤器：

- BasicProcessingFilter——处理 HTTP 基本身份验证请求；
- AuthenticationProcessingFilter——处理基于表单的身份验证请求；
- CasProcessingFilter——基于 CAS 服务票据的存在性和有效性验证用户身份。

可以看出，三种认证处理过滤器和三种认证入口点是一一对应的。事实上，每一种认证入口点都需要和对应的认证处理过滤器配合才能完成身份验证的完整过程。图 11.8 展示了这一协

作过程。

图 11.8　认证入口点和认证处理过滤器协同工作以验证 Web 用户的身份

认证入口点通过提示用户登录启动身份验证过程。当用户提交认证所需的信息，认证处理过滤器试图验证该用户的身份（在认证管理器的帮助下）。

让我们更深入地考察一下 Acegi 中提供的三种身份验证方式是如何工作的。首先由基本身份验证开始。

基本身份验证

基于 Web 的各种身份验证方式中最简单的形式称为基本身份验证。基本身份验证的工作机制是服务器发送一个 HTTP 401（未授权）响应至 Web 浏览器。当浏览器看到这个响应，它就可以知道服务器需要用户进行登录。于是，浏览器弹出一个对话框提示用户输入用户名和密码。当用户提交登录请求时，浏览器将请求送回至服务器进行身份验证。如果验证成功，用户会被送往他期望访问的目标 URL。否则，服务器会再回送一个 HTTP 401 响应，并且浏览器会再次提示用户登录。

在 Acegi 中使用基本身份验证，需要先配置一个 BasicProcessingFilterEntryPoint Bean：

```
<bean id="authenticationEntryPoint" class="net.sf.acegisecurity.
          ui.basicauth.BasicProcessingFilterEntryPoint">
  <property name="realmName">
    <value>Spring Training</value>
  </property>
</bean>
```

BasicProcessingFilterEntryPoint 只有一个属性。属性 realmName 指定了一个任意字符串，该字符串会显示在登录对话框上，用于提示用户为什么需要登录。

当用户点击登录对话框上的“确定”按钮，用户名和密码会通过 HTTP 头提交回服务器端。这时 BasicProcessFilter 就可以取得用户名和密码并进行处理。

```
<bean id="basicProcessingFilter" class="net.sf.acegisecurity.
         ui.basicauth.BasicProcessingFilter">
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="authenticationEntryPoint">
```

```
    <ref bean="authenticationEntryPoint"/>
  </property>
</bean>
```

BasicProcessingFilter 从 HTTP 头中取得用户名和密码，将它们转交给通过 authenticationManager 属性装配的认证管理器。如果认证成功，会将一个 Authentication 对象放到会话中供以后引用。否则，如果认证失败，会将控制转交给认证入口点（通过 authenticationEntryPoint 属性装配），给予用户一个再次登录的机会。

与 Acegi 的所有过滤器一样，BasicProcessingFilter 需要一个在 web.xml 中配置的 FilterToBeanProxy：

```
<filter>
  <filter-name>Acegi-Authentication</filter-name>
  <filter-class>net.sf.acegisecurity.
                    util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetBean</param-name>
    <param-value>
     net.sf.acegisecurity.ui.basicauth.BasicProcessingFilter
    </param-value>
  </init-param>
</filter>
...
<filter-mapping>
  <filter-name>Acegi-Authentication</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

基于表单的身份验证

尽管基本身份验证对简单的应用而言很不错，但它有一些局限性。主要原因在于浏览器弹出的对话框对用户而言既不友好，又不美观。基于表单的身份验证克服了这个局限，对大多数应用而言更加合适。用户不用面对弹出式的登录对话框，而是可以通过一个基于 Web 的表单进行登录。

Acegi 的 AuthenticationProcessingFilterEntryPoint 是一个提供给用户基于 HTML 的登录表单的认证入口点。你可以在 Spring 配置文件中用以下方式来配置它：

```
<bean id="authenticationEntryPoint" class="net.sf.acegisecurity.
```

```
        ui.webapp. AuthenticationProcessingFilterEntryPoint">
    <property name="loginFormUrl">
      <value>/jsp/login.jsp</value>
    </property>
    <property name="forceHttps"><value>true</value></property>
  </bean>
```

属性 loginFromUrl 配置了一个登录表单的 URL。当需要用户登录时，AuthenticationProcessingFilterEntryPoint 会将用户重定向到该 URL。根据上述配置中，它会重定向到一个 JSP 文件，JSP 中可能包含下面的 HTML 表单：

```
  <form method="POST" action="j_acegi_security_check">
    <input type="text" name="j_username"><br>
    <input type="password" name="j_password"><br>
    <input type="submit">
  </form>
```

这个登录表单中必须有两个名为 j_username 和 j_password 的域，供用户输入用户名和密码。表单的 action 属性被设置为 j_acegi_security_check，这会被 AuthenticationProcessingFilter 拦截。
AuthenticationProcessingFilter 是处理基于表单身份验证的过滤器。在 Spring 配置文件中它是以如下方式配置的：

```
  <bean id="authenticationProcessingFilter"
        class="net.sf.acegisecurity.
     ui.webapp.AuthenticationProcessingFilter">
    <property name="filterProcessesUrl">
      <value>/j_acegi_security_check</value>
    </property>
    <property name="authenticationFailureUrl">
      <value>/jsp/login.jsp?failed=true</value>
    </property>
    <property name="defaultTargetUrl">
      <value>/</value>
    </property>
    <property name="authenticationManager">
```

```
    <ref bean="authenticationManager"/>
  </property>
</bean>
```

属性 filterProcessesUrl 告诉 AuthenticationProcessingFilter 应该拦截哪个 URL。这个 URL 与登录表单中 action 属性的值一样。它的默认值为/j_acegi_security_check，但我们在这里显式定义了该值，用于说明你可以根据需要改变这个值。

属性 authenticationFailureUrl 指定当身份验证失败时用户应该被送往哪里。对于上述配置，我们会将用户送回到登录页面，同时传递一个参数用于表明认证失败（因此可以在登录页面上显示一个错误消息）。

正常情况下，当身份验证成功后，AuthenticationProcessingFilter 会在 HTTP 会话中放置一个 Authentication 对象，并且将用户重定向到他们期望访问的页面。因为 SecurityEnforcementFilter 在把控制转交给认证入口点之前，已经把原先的目标 URL 放到 HTTP 会话中了，所以 AuthenticationProcessingFilter 知道用户期望访问的页面是什么。当 AuthenticationProcessingFilter 成功验证用户身份之后，它会从 HTTP 会话中取得目标 URL 并将用户重定向到该 URL。

属性 defaultTargetUrl 定义了当出现目标 URL 不在 HTTP 会话中的异常情况时将发生什么。这可能发生在用户通过浏览器书签或其他方式而不是通过 SecurityEnforcementFilter 到达登录页面的情况下。

在 Spring 中定义了 AuthenticationProcessingFilter 之后，最后一件需要做的事就是配置一个 FilterToBeanProxy，它能够将过滤处理委托给 authenticationProcessingFilter Bean：

```
<filter>
  <filter-name>Acegi-Authentication</filter-name>
  <filter-class>net.sf.acegisecurity.
                    util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter
    </param-value>
  </init-param>
</filter>
…
<filter-mapping>
  <filter-name>Acegi-Authentication</filter-name>
```

```
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

CAS 身份验证

在第 11.2.4 节中，你已经看到如何配置一个 CasAuthenticationManager，从而可以根据 CAS 服务器来认证 CAS 的服务票据。但那一节还遗留着一个有待回答的大问题：如何首先将用户送到 CAS 登录页面。

Acegi 的 CasProcessingFilterEntryPoint 是一个能够将用户送到 CAS 服务器进行登录的认证入口点。你可以在 Spring 配置文件中按以下方式声明它：

```
  <bean id="authenticationEntryPoint" class="net.sf.acegisecurity.
          ui.cas.CasProcessingFilterEntryPoint">
    <property name="loginUrl">
      <value>https://localhost:8443/cas/login</value>
    </property>
    <property name="serviceProperties">
      <ref bean="serviceProperties"/>
    </property>
  </bean>
```

CasProcessingFitlerEntryPoint 中两个属性的意思是不言自明的。属性 loginUrl 指定了 CAS 登录页面的 URL,而属性 serviceProperties 是指向在 11.2.4 节中声明的 serviceProperties Bean 的引用。

无论用户是否成功登录 CAS，你必须首先明确一点，CasAuthenticationManager 总是会在允许用户访问受保护资源之前尝试验证 CAS 票据的真实性。CasProcessingFilter 是一个认证处理过滤器，它会拦截来自 CAS 服务器的包含待验证票据的请求。

```
  <bean id="authenticationProcessingFilter"
       class="net.sf.acegisecurity.ui.cas.CasProcessingFilter">
    <property name="filterProcessesUrl">
      <value>/j_acegi_cas_security_check</value>
    </property>
    <property name="authenticationManager">
      <ref bean="authenticationManager"/>
    </property>
    <property name="authenticationFailureUrl">
      <value>/authenticationfailed.jsp</value>
```

```
    </property>
    <property name="defaultTargetUrl">
      <value>/</value>
    </property>
  </bean>
```

CasProcessingFilter 和 AuthenticationProcessingFilter 有相同的属性集。但需要特别注意 filterProcessesUrl 属性。在这里，它被设置为/j_acegi_cas_security_check。在第 11.2.4 节中，我们设置 serviceProperties Bean 的 service 属性为一个以同样模式结尾的 URL。当用户在 CAS 服务器上登录成功之后，CAS 会将用户重定向到一个服务 URL。在非 Acegi 应用系统中，它可以是受保护应用的任意 URL。但如果使用 Acegi 保护一个应用系统，你必须确保首先调用 CasAuthenticationManager 来处理 Acegi 这边的身份验证并检查用户的授权。

在 CAS 服务器那端，serviceProperties Bean 的 service 属性告诉 CAS 当登录成功之后转向哪里。在客户端，filterProcessesUrl 属性确保有一个 CasProcessingFilter 回答该请求，并将 CAS 票据转交给 CasAuthenticationManager 进行验证。

11.4.4　设置一个安全上下文

在处理请求的过程中，用户的认证信息是通过一个 ContextHolder（实质上是一个 ThreadLocal）进行传递的。Acegi 过滤器链中的每一个过滤器都通过 ContextHolder 获取用户的认证信息。

但是，ThreadLocal 不能跨越多个请求存在。因此，Acegi 必须找到一个方便的位置存放用户的认证信息，从而当下一个请求到来时该信息仍然可用。这就是 Acegi 的集成过滤器要做的事。

一个集成过滤器的生命周期开始于从一个知名位置查找用户的 Authentication 对象——通常是 HTTP 会话。然后，它构造一个新的 ContextHolder 对象，并且将 Authentication 对象放到其中。

当请求处理完成之后，集成过滤器从 ContextHolder 中取得 Authentication 对象，并将它放回到知名位置中，供下一个请求使用。

Acegi 提供了若干个集成过滤器，其中 HttpSessionIntegrationFilter 适用于大多数情形。它将 Authentication 对象保存在 HTTP 会话中，使之能够跨越多个请求。你可以在 Spring 配置文件中按以下方式配置它：

```
  <bean id="integrationFilter" class="net.sf.acegisecurity.
            ui.webapp.HttpSessionIntegrationFilter"/>
```

最后，你需要在 web.xml 中配置一个 FilterToBeanProxy 过滤器，它负责将过滤处理委托给 integrationFilter Bean 执行：

```
<filter>
  <filter-name>Acegi-Integration</filter-name>
  <filter-class>net.sf.acegisecurity.util.FilterToBeanProxy
          </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>net.sf.acegisecurity.ui.AutoIntegrationFilter
          </param-value>
  </init-param>
</filter>
…
<filter-mapping>
  <filter-name>Acegi-Integration</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

要把集成过滤器的<filter-mapping>配置项放到所有其他 Acegi 过滤器的<filter-mapping>的后面，这点很重要。

11.4.5 确保通道安全性

受保护的 Web 应用系统中的一些页面可能带有敏感信息。如果这些信息是通过一个非安全的通道传输的（比如 HTTP），则存在被目无法纪的黑客截取数据并用于破坏性目的的风险。

这方面的常见例子包括登录页面或者输入或显示用户的信用卡信息的页面。万一这些信息被泄露，则某人的私人数据将被用于购买物品，或者用于假冒该用户的身份。要让用户感觉到他们的信息机密性是有保障的，这一点是非常重要的，否则他们会不再使用你的站点。或者更糟糕地，他们会诉诸法律来确保你赔偿了他们的损失。

HTTPS 对服务器和浏览器之间传输的数据进行加密，有助于阻止高科技罪犯从互联网上截取敏感数据。一般说来，使用 HTTPS 就像在 URL 中使用“https://”而不是“http://”一样简单。然而，这需要你记住每次链接到一个显示敏感数据的页面时都加上“s”。虽然看上去非常简单，但在我们自己的经验中，忘记加上“s”的次数也是数不胜数的。

Acegi 通过 ChannelProcessingFilter 提供了一个解决方案。ChannelProcessingFilter 确保 Web 应用的页面是通过合适的通道（HTTP 或 HTTPS）传输的——无论你是否记得在链接的 URL 前加了“https://”。

要使用 ChannelProcessingFilter，你必须在 Web 应用的 web.xml 文件中再增加一个 FilterToBeanProxy 配置：

```
<filter>
  <filter-name>Acegi-Channel</filter-name>
  <filter-class>net.sf.acegisecurity.util.FilterToBeanProxy
          </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      net.sf.acegisecurity.securechannel.ChannelProcessingFilter
    </param-value>
  </init-param>
</filter>
…
<filter-mapping>
  <filter-name>Acegi-Channel</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

ChannelProcessingFilter 的<filter-mapping>定义必须出现在任何其他的<filter-mapping>之前，这点非常重要。这是因为 ChannelProcessingFilter 需要在允许其他过滤器处理之前确保请求是经过合适的通道传输的。

当你在 web.xml 中配置了 FilterToBeanProxy 之后，就需要在 Spring 配置文件中声明受委托的过滤器 Bean：

```
<bean id="channelProcessingFilter" class="net.sf.acegisecurity.
          securechannel.ChannelProcessingFilter">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      \A/secure/.*\Z=REQUIRES_SECURE_CHANNEL
      \A/login.jsp.*\Z=REQUIRES_SECURE_CHANNEL
      \A/j_acegi_security_check.*\Z=REQUIRES_SECURE_CHANNEL
      \A.*\Z=REQUIRES_INSECURE_CHANNEL
    </value>
  </property>
  <property name="channelDecisionManager">
    <ref bean="channelDecisionManager"/>
```

```
    </property>
  </bean>
```

属性 filterInvocationDefinitionSource 用于定义哪个页面是安全的或者是非安全的。和 FilterSecurityInterceptor（第 11.4.2 节）一样，这个属性的值是由一个属性编辑器负责解释的。第一行表明请求的 URL 在与随后各行中的模式进行比较之前必须首先转换成小写。第一行之后的每一行都将一个通道规则关联到一个 URL 模式上。在上述配置中，我们使用正则表达式来定义 URL 模式，但正如安全拦截器一样，你也可以使用与 Ant 相似的模式，只需要增加一个 PATTERN_TYPE_APACHE_ANT 指令即可。

在 URL 模式之上可以应用两种通道规则：

n　REQUIRES_SECURE_CHANNEL——规定与该模式匹配的 URL 必须经过一个安全通道传输（例如，HTTPS）；

n　REQUIRES_INSECURE_CHANNEL——规定与该模式匹配的 URL 必须经过一个非安全的通道传输（例如，HTTP）。

对于上面的例子，我们定义了登录页面、认证过滤器（/j_acegi_security_check），以及所有位于“/secure”下的页面必须经过一个安全通道进行传输。所有其他的 URL 必须通过一个非安全的通道进行传输。

ChannelProcessingFilter 并非独自完成强制通道安全性的工作的。它和由 ChannelDecisionManager 属性所引用一个 ChannelDecisionManager 合作，而 ChannelDecisionManager 又将责任进一步委托给一个或多个 ChannelProcessor。这让人联想起在 AccessDecisionManager 和 AccessDecisionVoters 之间的关系。图 11.9 展示了这种关系。

图 11.9　通道处理过滤器依赖通道决策管理器来决定是否要切换到安全通道或非安全通道。如果需要切换，则由通道处理器执行实际的切换

通道决策管理器负责判断请求 URL 的通道是否满足通道安全性规则（由 ChannelProcessingFilter 的 filterInvocationDefinitionSource 属性定义）。然而，ChannelDecisionManagerImpl 作为 Acegi 惟一内置的 ChannelDecisionManager 实现的，是把这个决策交给通道处理器来完成。

ChannelDecisionManagerImpl 轮询它的所有通道处理器，给予他们重载请求通道的机会。每个通道处理器根据通道安全规则检查请求。一旦某个通道处理器觉得请求的通道有问题，它就会进行重定向以确保请求是足够安全的。

既然你已经看到 ChannelProcessingFilter 是如何工作的了，现在可以把所有的片断合为一

体。正如你先前看到的，channelProcessingFilter Bean 的 channelDecisionManager 属性装配了一个指向 channelDecisionManager Bean 的引用。这个 channelDecisionManager Bean 是按照以下方式声明的：

```
<bean id="channelDecisionManager" class= "net.sf.acegisecurity.
          securechannel.ChannelDecisionManagerImpl">
  <property name="channelProcessors">
    <list>
      <ref bean="secureChannelProcessor"/>
      <ref bean="insecureChannelProcessor"/>
    </list>
  </property>
</bean>
```

ChannelDecisionManagerImpl 的通道处理器是通过 channelProcessors 属性提供的。在上面的例子中，我们为它提供了两个通道处理器，这些通道处理器通过以下 XML 声明的：

```
<bean id="secureChannelProcessor" class="net.sf.acegisecurity.
          securechannel.SecureChannelProcessor"/>
<bean id="insecureChannelProcessor" class=
     "net.sf.acegisecurity.securechannel.InsecureChannelProcessor"/>
```

SecureChannelProcessor 按照以下方式考察与请求的 URL 相关联的通道安全性规则。如果规则是 REQUIRES_SECURE_CHANNEL，而请求是非安全的，则 SecureChannelProcessor 将请求重定向到一种安全的形式。例如，基于上面给出的 channelProcessingFilter Bean 的 filterInvocationDefinitionSource 属性：

n http://www.springinaction.com/training/secure/editCourse.htm 会被重定向到 https://www.springinaction.com/training/secure/editCourse.htm，因为它与一个关联着 REQUIRES_SECURE_CHANNEL 规则的 URL 模式相匹配。

n http://www.springinaction.com/training/j_acegi_security_check 会被重定向到 https://www.springinaction.com/training/j_acegi_security_check，因为它与一个关联着 REQUIRES_SECURE_CHANNEL 规则的 URL 模式相匹配。

n http://www.springinaction.com/training/displayCourse.htm 不会被重定向，因为它匹配的 URL 模式没有 REQUIRES_SECURE_CHANNEL 规则。

n https://www.springinaction.com/training/j_acegi_security_check 不会被重定向，因为它已经是安全的了。

InsecureChannelProcessor 的功能与 SecureChannelProcessor 正好相反。它不是确保一个请

求是通过安全通道传输的，而是确保一个请求是通过非安全的通道传输的。例如：

n https://www.springinaction.com/training/displayCourse.htm 会被重定向为http://www.springinaction.com/training/displayCourse.htm，因为它与一个关联REQUIRES_INSECURE_CHANNEL规则的URL模式匹配。

n https://www.springinaction.com/training/j_acegi_security_check 不会被重定向，因为它匹配的URL模式没有关联的REQUIRES_INSECURE_CHANNEL规则。

n http://www.springinaction.com/training.displayCourse.htm 不会被重定向，因为它匹配的URL模式关联一条REQUIRES_INSECURE_CHANNEL规则，而它已经是非安全的了。

在我们告别Acegi对Web安全性的支持之前，让我们看一下如何使用Acegi的标签库在Web应用的页面内部强制安全规则。

11.4.6 使用Acegi的标签库

称之为标签库可能有点言过其辞了。实际上,Acegi只提供了一个JSP标签:<authz:authorize>标签。

虽然Acegi的安全强制过滤器能够阻止用户浏览他们没有权限看到的页面，但最好的做法是从一开始就不提供指向受限制页面的链接。<authz:authorize>标签能够根据当前用户是否拥有恰当权限来决定显示或隐藏Web页面的内容。

<authz:authorize>是一个流程控制标签，能够在满足特定安全需求的条件下显示它的内容体。它有三个互斥的参数：

n ifAllGranted——是一个由逗号分隔的权限列表，用户必须拥有所有列出的权限才能渲染标签体；

n ifAnyGranted——是一个由逗号分隔的权限列表，用户必须至少拥有其中的一个才能渲染标签体；

n ifNotGranted——是一个由逗号分隔的权限列表，用户必须不拥有其中的任何一个才能渲染标签体。

你可以轻松地想像在JSP中如何使用<authz:authorize>标签根据用户的权限来限制他们的行为。例如，Spring培训应用有一个向用户显示课程有关信息的课程明细页面。对管理员来说,如果能够从课程明细页面直接跳转到课程编辑页面从而可以更新课程信息是很方便的。但你不希望这个链接对除了管理员之外的其他用户可见。

使用<authz:authorize>标签，在用户没有管理员权限的情况下，你可以避免渲染到课程编辑页面的链接：

```
<authz:authorize ifAllGranted="ROLE_ADMINISTRATOR">
  <a href="admin/editCourse.htm?courseId=${course.id}">
    Edit Course
  </a>
</authz:authorize>
```

这里，我们使用了 ifAllGranted 参数，由于这里只需要检查一个授权，所以 ifAllGranted 标签也是可以使用的。Web 应用的安全性只是 Acegi 功能的一个方面。现在让我们考察它的另一面——保护方法调用。

11.5 保护方法调用

虽然 Acegi 保护 Web 请求的手段是使用 Servlet 过滤器，它却是利用 Spring 对 AOP 的支持来提供方法级别的声明式保护的。这意味着不是设置一个 SecurityEnforcementFilter 来强制安全性，而是设置一个 Spring AOP 代理来拦截方法调用，并将控制转交给安全拦截器。

11.5.1 创建一个安全切面

也许设置一个 AOP 代理的最简单的方式是使用 Spring 的 BeanNameAutoProxyCreator，并且简单地列举出你需要保护的 Bean[3]。例如，假设你希望保护 courseService 和 billingService Bean：

```
<bean id="autoProxyCreator" class="org.springframework.aop.framework.
        autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list>
      <value>securityInterceptor</value>
    </list>
  </property>
  <property name="beanNames">
    <list>
      <value>courseService</value>
      <value>billingService</value>
    </list>
  </property>
</bean>
```

在这个例子中，我们要求自动代理创建器 autoProxyCreator 通过一个名为 securityInterceptor 拦截器代理它的 Bean。该 securityInterceptor Bean 按照以下方式配置：

```
<bean id="securityInterceptor" class="net.sf.acegisecurity.
      intercept.method.MethodSecurityInterceptor">
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
```

```
    </property>
    <property name="accessDecisionManager">
      <ref bean="accessDecisionManager"/>
    </property>
    <property name="objectDefinitionSource">
      <value>
        com.springinaction.springtraining.service.
                CourseService.createCourse=ROLE_ADMIN
        com.springinaction.springtraining.service.
                CourseService.enroll*=ROLE_ADMIN,ROLE_REGISTRAR
      </value>
    </property>
  </bean>
```

MethodSecurityInterceptor 针对方法调用所做的一切与 FilterSecurityFilter 针对 Servlet 请求所做的是相同的。具体地说，它拦截调用，并协调认证管理器和访问决策管理器的工作，以确保方法的安全性需求得到满足。

你可以注意到 authenticationManager 和 accessDecisionManager 属性与 FilterSecurityInterceptor 是一样的。事实上，可以把应用于 FilterSecurityInterceptor 的同一个 Bean 装配到这些属性中。

MethodSecurityInterceptor 也和 FilterSecurityInterceptor 一样有一个 objectDefinitionSource 属性。尽管它和 FilterSecurityInterceptor 的同名属性的目标是一致的，在配置方式上两者还是有些许不同。它不是将 URL 模式与权限相关联，而是将方法的模式与调用该方法所需要的权限进行关联。

一个方法模式包含全称类名和需要保护的方法的名字。值得注意的是你可以在方法模式的开头和结尾使用通配符以匹配多个方法。

```
Com.springinaction…CoruseService.enroll *
     =ROLE_ADMIN, ROLE_REGISTRAR
```

当一个受保护的方法被调用时，MethodSecurityInterceptor 将判断用户是否已通过身份验证并且拥有合适的授权来调用该方法。如果是，则会继续调用目标方法。如果不是，则会抛出

一个 AcegiSecurityException。更具体地说，如果用户身份无法得到验证，会抛出一个 AuthenticationException，或者，如果用户没有足够权限调用本方法，则会抛出一个 AccessDeniedException。

为了与 Spring 的异常理念相一致，AcegiSecurityException 是一个未检查的异常。进行调用的代码可以选择捕捉或者忽略该异常。

在 Spring 配置文件中规定方法安全属性仅仅是声明方法级别安全性的一种手段。你已经看到如何使用 Jakarta Commons Attributes 来定义事务策略（第 4 章）和 URL 映射（第 8 章)了。现在让我们看一下如何使用 Jakarta Commons Attributes 来声明安全属性。

11.5.2　使用元数据保护方法

与事务和 URL 处理器映射机制相同，你首先需要做的是声明一个元数据实现以告诉 Spring 如何装载元数据。如果现在还没有在应用上下文中增加一个 CommonsAttributes Bean，你需要增加一个：

```
<bean id="attributes"
    class="org.springframework.metadata.commons.CommonsAttributes"/>
```

接下来，你需要声明一个对象定义源。在第 11.5.1 节中，你通过将 objectDefinitionSource 属性设置为一个将安全属性映射到方法的字符串定义了一个对象定义源。而现在你将在被保护对象的源代码中直接声明安全属性。Acegi 的 MethodDefinitionAttributes 是一个对象定义源，它能够从受保护对象的元数据中获取它的安全属性：

```
<bean id="objectDefinitionSource"class="net.sf.acegisecurity.
        intercept. method.MethodDefinitionAttributes">
  <property name="attributes"><ref bean="attributes"/></property>
</bean>
```

MethodDefinitionAttributes 的 attributes 属性装配了一个指向 attributes Bean 的引用，因此它能够知道如何使用 Jakarta Commons Attributes 来获取安全属性。[4]

现在，属性 objectDefinitionSource 已经配置好了，我们把它装配到 MethodSecurityInterceptor 的 objectDefinitionSource 属性中（代替在 11.5.1 节中的字符串属性）：

```
<bean id="securityInterceptor" class="net.sf.acegisecurity.
        intercept.method.MethodSecurityInterceptor">
…
  <property name="objectDefinitionSource">
```

```
    <ref bean="objectDefinitionSource"/>
  </property>
</bean>
```

现在可以开始为你的代码加上安全属性标记了。你惟一需要知道的安全属性是SecurityConfig，它在权限和方法之间建立关联。例如，以下的代码段展示了如何标记CourseService 的 enrollStudentInCourse() 方法，声明它需要 ROLE_ADMIN 或ROLE_REGISTRAR 属性：

```
/**
 * @@net.sf.acegisecurity.SecurityConfig("ROLE_ADMIN")
 * @@net.sf.acegisecurity.SecurityConfig("ROLE_REGISTRAR")
 */
public void enrollStudentInCourse(Course course,
     Student student) throws CourseException;
```

在 enrollStudentInCourse 方法上声明这些安全属性和第 11.5.1 节中定义的objectDefinitionSource 声明等价的。

11.6 小结

安全性对于许多应用系统而言都是很重要的方面。Acegi 安全系统提供了一种基于 Spring 的低耦合、依赖注入和面向切面编程的保护应用系统的机制。

你可能已经注意到了本章中只提供了很少的 Java 代码。希望你不会因此而感到失望。缺少 Java 代码显示了 Acegi 的关键强项——应用系统与安全性之间的低耦合。安全性是一个超越应用系统核心关注点的切面。使用 Acegi，你可以不需要应用代码中直接编写任何与安全有关的代码而达到保护应用系统的目的。

另一件你可能已经注意到的事，是大多数使用 Acegi 保护应用系统所需的配置并不需要知道它所保护的应用系统本身。惟一真正需要知道被保护应用系统细节的 Acegi 组件是用于关联被保护资源和访问该资源所需权限的对象定义源。无论对于 Acegi，还是受 Acegi 保护的应用系统，两者间的松耦合性都得到了充分的体现。