9.技术概论

9.1 运行环境

Spring Security 3.0 要求Java 5.0 运行环境或更高。由于Spring Security目的是创建一个包含的方式，所以没必要在你的Java环境配置中额外使用多余的配置文件。另外，也没必要去配置一个验证和授权策略文件或者将Spring Security放到classpath中。

同样，如果您使用的是EJB容器或Servlet容器，则不需要在任何地方放置任何特殊配置文件，也不需要将Spring Security包含在服务器类加载器中。所有需要的配置文件都应该包含在你的应用程序中。

这种设计提供了最大的部署时间灵活性，因为您可以简单地将目标工件（不管是JAR，WAR还是EAR）从一个系统复制到另一个系统，并立即生效。

9.2 核心组件

在Spring Security 3.0中，`spring-security-core`jar包的内容被剥离到极小。它不再包含任何关于web应用程序安全、LDAP或者命名空间配置。我们在这章将会介绍你可以在core模块中发现的Java类型。它们代表了框架的构建块，所以如果您需要超越简单的名称空间配置，那么了解它们是什么是很重要的，即使您实际上并不需要直接与它们进行交互。

9.2.1 SecurityContextHolder，SecurityContext和Authentication 对象们

最基础的对象是`SecurityContextHolder`。这是我们存储应用程序当前安全上下文的详细信息的地方，其中包括当前正在使用该应用程序的主体的详细信息。默认情况下`SecurityContextHolder`使用一个`ThreadLocal`去存储这些细节，这证明安全上下文中的内容对于相同线程中执行的方法总是可见的，尽管这个安全上下文没有明显的作为参数传输到该方法中。以这种方式使用ThreadLocal是相当安全的，因为如果在处理当前委托人的请求之后谨慎清除线程，那么它是非常安全的。Spring Security为你自动处理这些，因此你没必要担心它。

因为线程的特殊工作方式，一些应用不是完全适用于使用`ThreadLocal`的。例如，一个Swing客户端可能想要在Java虚拟机中所有线程都使用相同的安全上下文。我们可以为`SecurityContextHolder`配置一个策略，即在启动时指定如何上下文应该怎么存储。对于一个单机的应用程序，你可能会使用`SecurityContextHolder.MODE_GLOBAL`策略。其他应用程序可能希望安全线程产生的线程也具有相同的安全身份。我们靠使用`SecurityContextHolder.MODE_INHERTABLETHREADLOCAL`来实现这个。用两种方法在去改变这个模式从默认的`SecurityContextHolder.MODE_THREADLOCAL`。第一个方法是设置系统参数，第二种方法是调用`SecurityContextHolder` 上的一个静态方法。大多数应用程序不需要去改变默认设置，但是如果你要改变设置的话，你可以了解`SecurityContextHolder`的JavaDOC文档去学习更多。

从当前用户中获得信息

在SecurityContextHolder中，我们存储当前与应用程序交互的委托人的详细信息。Spring Security用一个`Authentication`对象去存储这些信息。你不需要自己去创建一个`Authentication`对象，但是对于用户来说去查询`Authentication`对象是很平常的。你能在应用程序中的任何地方使用下面的代码块，去获得被验证用户的名字，例如：

```
Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
if(principal instanceof UserDetails)
```

```
{
    String username = ((UserDetails)principal).getUsername();
}else
{
    String username = principal.toString();
}
```

调用getContext()方法返回的是一个SecurityContext接口实例。这个对象保存在线程本地存储中。正如我们将在下面看到的，Spring Security中的大多数身份验证机制都返回一个UserDetails实例作为主体。

9.2.2 UserDetailsService

从上面的代码段中我们还意识到另一+件事就是你可以从Authentication对象中获得一个主题。这个主体仅仅是一个Object。大部分情况下它能被转型为一个UserDetails对象。UserDetails对象是Spring Security中的一个核心接口。它表现了一个实体，但以一种可扩展的和特定于应用程序的方式。考虑UserDetails作为一个在你的数据库和在SecurityContextHolder里Spring Security需要的东西的适配器。作为您自己的用户数据库中某些东西的表示形式，通常您会将UserDetails转换为您的应用程序提供的原始对象，以便您可以调用业务特定的方法（如getEmail（），getEmployeeNumber（）等）。

现在你可能想知道，什么时候我提供的一个UserDetails对象呢？我是怎么做到这个的？我想你说的这个东西是声明式的并且我不需要去写任何Java代码-谁提供的这个机制？简短的答案是这里有一个接口叫做UserDetailsService。这个接口中的方法接受一个基于String的用户名参数并返回一个UserDetails：

```
UserDetails loadUserByUsername(String username)throws UsernameNotFoundException;
```

这是在Spring Security中为用户加载信息的最常见方法，只要需要用户信息，您就会看到它在整个框架中使用。

在成功的验证中，UserDetails被用来创建存储在SecurityContextHolder的Authentication对象。好消息是我们提供了很多UserDetailsService实现，包括一个使用内存表（InMemoryDaoImpl）和另一个使用JDBC（JdbcDapImpl）。大多数用户计划去写他们自己的实现，但是它们的实现通常是基于已经存在的表示它们的雇员、消费者或者应用中其他用户的DAO层之上。记住这点，无论怎样你的UserDetailsService总能被调用上述代码从SecurityContextHolder获得。

对于UserDetailsService经常会有一些困惑。它纯粹是用于用户数据的DAO，除了将该数据提供给框架内的其他组件外，不执行其他功能。另外，它不认证用户，认证用户的工作由AuthenticationManager执行。在很多情况下，如果你要自定义验证过程，它对你实现AuthenticationProvider起到了很大作用。

9.2.3 GrantedAuthority

除了principal之外，Authentication提供的另一个重要方法是getAuthorities()。这个方法提供了一个GrantedAuthority对象的数组。一个GrantedAuthority是，一个被赋予给主体的权限。这些权限通常是“角色”，例如ROLE_ADMINISTRATOR或ROLE_HR_SUPERVISOR。稍后将为Web授权，方法授权和域对象授权配置这些角色。Spring Security的其他部分能够解释这些授权，并且将其实现。GrantedAuthority对象通常靠UserDetailsService被加载。

通常，GrantedAuthority对象是应用程序范围的权限。它们没必要被给定一个域对象。因此，您不可能有一个GrantedAuthority来代表54号员工对象的权限，因为如果有成千上万个这样的权限，您会很快耗尽内存（或者至少导致应用程序花费很长时间 时间来验证用户）。当然，Spring Security的设计明确是为了处理这个常见的需求，但是你应该使用项目的域对象安全功能来达到这个目的。

9.2.4 总结

回想一下，到目前为止我们看到的Spring Security的主要构建块是：

- SecurityContextHolder - 提供对SecurityContext的访问。
- SecurityContext - 来保存身份验证和可能的特定于请求的安全信息。
- Authentication - 以Spring-Security特定的方式保存实体
- GrantedAuthority - 去应一个在应用范围内对一个实体的授权
- UserDetails，从你应用程序的DAO层或者其他安全数据源提供必要的信息去创建一个Authentication。
- UserDetailsService - 当传入一个基于Sting的用户名创建一个UserDetails。

现在你已经了解了这些被重复使用的组件，让我们开始看验证过程。

9.3 验证

Spring Security可以参加到不同的验证环境。尽管我们建议人们使用Spring Security进行身份验证，并且不会与现有的容器管理身份验证集成，但仍然支持 - 与您自己的专有身份验证系统集成。

9.3.1 在Spring Security中的验证

让我们看一下每个人都熟悉的标准的验证情景：

1. 一个用户被提示使用用户密码进行登录。
2. 系统成功地验证了密码对于用户名是正确的。
3. 该用户的上下文信息被获取（角色列表等等）。
4. 对于该用户的安全上下文被建立。
5. 用户可能继续执行一些操作，该操作可能受访问控制机制保护，访问控制机制针对当前安全上下文信息检查操作所需的权限。

前三项构成了验证过程，所以我们现在将会看一下在Spring Security中这些是如何发生的：

1. 用户名和密码被获取和组合成一个UsernamePasswordAuthenticationToken实例（一个Authentication接口的实例，我们能轻易的找到它）。
2. 这个token将被发送到一个AuthenticationManager实例中去确认。
3. 当成功验证后AuthenticationManager将会返回一个完全填充的Authentication。
4. 安全上下文靠调用SecurityContextHolder.getContext().setAuthentication(...),传入返回的认证对象。

经过上面的过程，用户被考虑已经被验证过了。让我们看下下面的代码，作为一个例子：

```
import org.springframework.security.authentication.*;
import org.springframework.security.core.*;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
```

```
public class AuthenticationExample
{
    private static AuthenticationManager am = new SampleAuthenticationManager();

    public static void main(String [] args)throws Exception
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        while(true)
        {
            System.out.println("Please enter your username:");
            String name = in.readline();
            System.out.println("Please enter your password:");
            String password = in.readLine();
            try
            {
                Authentication request = new
UsernamePasswordAuthenticationToken(name,password);
                Authentication result = am.authenticate(result);
                SecurityContextHolder.getContext().setAuthentication(result);
                break;
            }catch(AuthenticationException e)
            {
                System.out.println("Authentication failed:"+e.getMessage());
            }
        }
        System.out.println("Successfully authenticated.Security context
contains:"+SecurityContextHolder.getContext().getAuthentication());
    }
}

class SampleAuthenticationManager implements AuthenticationManager
{
    static final List<GrantedAuthority> AUTHORITIES = new
ArrayList<GRANTEDAUTHORITY>();

    static
    {
        AUTHORITIES.add(new SimpleGrantedAuthority("ROLE_USER"));
    }

    public Authentication authenticate(Authentication auth)throws
AuthenticationException
    {
        if(auth.getName().equals(auth.getCredentials()))
        {
            return new
UsernamePasswordAuthenticationToken(auth.getName(),auth.getCredentials(),AUTHORITI
ES);
        }
```

```
            throw new BadCredentialsException("Bad Credentials");
        }
    }
```

这里我们写了个小程序，要求读者输入用户名和密码，并且展示了上述的验证流程。我们在这里实现的`AuthenticationManager`将会使所有用户名和密码都相同的用户验证通过。它为每个用户分配一个单一角色。上面代码的测试结果如下：

```
Please enter your username:
bob
Please enter your password:
password
Authentication failed: Bad Credentials
Please enter your username:
bob
Please enter your password:
bob
Successfully authenticated. Security context contains: \
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@44
1d0230: \
Principal: bob; Password: [PROTECTED]; \
Authenticated: true; Details: null; \
Granted Authorities: ROLE_USER
```

注意，你没必要去写这些代码。这些过程仅仅在内部发生，例如在一个web验证过滤器之中。我们在这里只是介绍Spring Security中集成的验证是怎样的过程。当`SecurityContextHolder`包含一个完全填充的`Authentication`对象时，这个用户就被通过验证了。

9.3.2 直接设置SecurityContextHolder内容

实际上，Spring Security并不介意如何将Authentication对象放在SecurityContextHolder中。唯一关键的要求是SecurityContextHolder包含一个Authentication，它代表AbstractSecurityInterceptor之前（我们稍后会看到更多）需要授权用户操作的一个实体。

您可以（以及许多用户）编写自己的过滤器或MVC控制器，以提供与基于Spring Security的身份验证系统的互操作性。例如，你可以使用一个容器管理的验证，当前用户的信息可以从一个ThreadLocal或者JNDI中得到。或者你工作的公司可能有一个遗留的验证系统，这个系统有一个专有的标准，你对它有很少的控制。在这种情况下，使用Spring Security是很简单的，并且依旧能提供验证功能。你需要做的仅仅就是写一个从一个地方读取第三方用户信息，建立Spring Security特定的`Authentication`对象，把它放到`SecurityContextHolder`的过滤器。在这种情况下，您还需要考虑通常由内置身份验证基础结构自动处理的事情。例如，在将响应写入客户端之前，您可能需要先创建一个HTTP会话来缓存请求之间的上下文（一旦响应呗提交，创建一个session就是不可能的了）。

如果你想知道`AuthenticationManager`在真实世界中是如何被实现的，我们将在核心服务章节中介绍。

9.4 在Web应用程序中的验证

现在让我们考虑你正在在你的web应用中使用Spring Security的情况（没有启动web.xml安全）。一个用户是如何被验证并且安全上下文是如何被创建的。

考虑一个世纪的web应用验证过程：

1. 你访问主页，然后点击链接。
2. 一个请求发送到服务器，服务器决定你正在访问一个被保护的资源。
3. 当你没有进行验证时，服务器发送回一个响应，要求你必须进行验证。这个响应可能是一个HTTP响应码，或者是重定向到某个特定页面。
4. 依赖与验证机制，你的浏览器将会被重定向到特定的网页，以便于你能够填充表单，或者浏览器浏览器将以某种方式检索您的身份（通过一个BASIC验证目录，一个cookie，一个X.509证书等）。
5. 浏览器将会发送一个响应到服务器。这个响应可能是一个HTTP POST请求，其内容是你所填写的表单，或者是器表头包含着你的验证细节。
6. 接下来，服务器将决定提交的凭证是否有效。如果他们是有效的，则进行下一步。否则，通常你的浏览器将会要求再次验证（所以你会跳到第二步）。
7. 原始的使你进行验证程序的请求将会再次发送。现在你已经被验证并对被保护的资源获得充分的权限。如果你对该资源有访问权限，那么这个请求将会成功。否则，你将会接收到一个HTTP错误码403，它的意思是'forbidden'。

Spring Security具有不同的类，负责上述大多数步骤。主要的参与者（按使用顺序）是`ExceptionTranslationFilter`，一个`AuthenticationEntryPoint`和一个“验证机制”，验证机制负责像我们上面看到的调用`AuthenticationManager`。

9.4.1 ExceptionTranslationFilter

ExceptionTranslationFilter是一个Spring Security过滤器，负责检测抛出的任何Spring安全异常。`AbstractSecurityInterceptor`通常会引发这些异常，它是授权服务的主要提供者。我们将在下一节讨论`AbstractSecurityInterceptor`，我们现在只需要知道它用来处理Java异常，关于HTTP毫无所知，对于如何去验证一个实体也一样。相反，`ExceptionTranslationFilter`提供此服务，具体负责返回错误代码403（如果主体已经过身份验证，因此根本没有足够的访问权限 - 按照上述步骤7），或者启动`AuthenticationEntryPoint`（如果主体尚未经过身份验证并且 因此我们需要开始第三步）。

9.4.2 AuthenticationEntryPoint

`AuthenticationEntryPoint`的职责是上面列出的第三步。正像你想象的那样，每个web应用程序都应该拥有一个默认的验证策略（这可以像Spring Security中几乎所有其他的一样配置，但现在让我们保持简单）。每个主要的验证系统都有自己的`AuthenticationEntryPoint`实现，它通常执行第三步中描述的事情。

9.4.3 验证机制

一旦你的浏览器提交了你的验证凭证（一个HTTP表单POST或者是HTTP报文头），服务器上的某些东西会手机这些验证细节。现在，我们在上述的第6步。在Spring Security中，我们为从用户代理（通常是Web浏览器）收集验证信息的方法提供了一个特殊的名称，称之为“验证机制”。例如表单验证和Basic验证。一旦验证细节从客户端被收集，一个`Authentication`请求对象将会被创建然后提交给`AuthenticationManager`。

9.4.4 在请求间存储SecurityContext

根据应用程序的类型，可能需要制定一个策略来存储用户操作之间的安全上下文。在一个实际的web应用中，一个用户一旦登陆，会被靠他们的session id进行认证。服务器缓存实体的session持续时间等主要信息。在

Spring Security中，在请求之间存储SecurityContext的责任归属于SecurityContextPersistenceFilter，SecurityContextPersistenceFilter默认将上下文存储为HTTP请求之间的HttpSession属性。它将每个请求的上下文恢复到SecurityContextHolder，并且在请求完成时清除SecurityContextHolder。出于安全目的，您不应该直接与HttpSession进行交互。没有理由这么做 - 总是使用SecurityContextHolder来代替。

对于其他类型的应用程序（例如，一个无状态的web服务）不应该使用HTTP session，他们在每次请求时重新认证。然而，`SecurityContextPersistenceFilter`被包含在过滤器链中也是很重要的，这确保了`SecurityContextHolder`在每个请求后被清除。

在一个单一会话中接受并发请求的应用程序中，相同的`SecurityContext`实例将会被在线程中共享。即使正在使用`ThreadLocal`，它也是从每个线程的`HttpSession`中检索的实例。如果您想临时更改线程正在运行的上下文，这会产生影响。你如果使用`SecurityContextHolder.getContext()`然后调用`setAuthentication(anAuthentication)`在返回的上下文对象上，那么所有并发线程上的`Authentication`对象都会改变，它会分享到相同的`SecurityContext`实例。如果你想自定义`SecurityContextPersistenceFilter`的做法使其对每个请求都创建一个新的`SecurityContext`，一次防止一个线程影响到另一个线程。可以选择在临时更改上下文的位置创建新实例。这个`SecurityContextHolder.createEmptyContext()`方法总是返回一个新的上下文实例。

9.5 Spring Security中的访问控制（授权）

在Spring Security中以访问控制决定为职责的是`AccessDecisionManager`接口。它有一个`decide`方法，它需要一个代表主体请求访问的`Authentication`对象，一个“安全对象”（见下文）和一个适用于该对象的安全元数据属性列表（例如被授予的角色列表）。

9.5.1 安全和AOP服务

如果你熟悉AOP，你知道这里有很多类型的服务被提供：before，after，throws，around。around服务非常有用，因为一个人可以选择是否进行方法调用，是否修改响应，是否抛出一个异常。Spring Security为方法调用和web请求提供了一个around服务。我们使用Spring标准AOP支持实现了一个方法调用的around服务，使用一个标准的过滤器实现对于web请求的around服务。

对于不了解AOP的人，你要理解的重点是Spring Security能帮助你保护你的方法调用和web请求。大多数人对在他们的服务层保护方法调用是非常感兴趣的。这是因为服务层是大多数业务逻辑驻留在当代Java EE应用程序中的地方。如果你想要去保护服务层的方法调用，Spring 的标准AOP就可以胜任。如果你想要直接保护域对象，你可能要去考虑使用AspectJ。

您可以选择使用AspectJ或Spring AOP执行方法授权，也可以选择使用过滤器执行Web请求授权。你可以选择使用0,1,2或者三个方法在一起使用。主流使用模式是执行一些Web请求授权，再加上服务层上的一些Spring AOP方法调用授权。

9.5.2 安全对象和AbstractSecurityInterceptor

到底什么是安全对象。Spring Security使用这个术语来指代可以有安全性的任何对象（比如授权决策）。最常见的例子是方法调用和Web请求。每个被支持的安全对象类型都有它自己的拦截器类，它是`AbstractSecurityInterceptor`类的子类。更重要的是，当`AbstractSecurityInterceptor`被调用时，如果实体被验证，`SecurityContextHolder`将会包含一个有效的`Authentication`。

`AbstractSecuritySecurityInterceptor`提供了一个统一的处理安全对象请求的工作流，如下：

1. 查阅和当前请求有关的“配置属性”。

2. 提交安全对象，当前的Authentication和配置交给AccessDecisionManager进行授权决策。
3. 可以选择更改调用发生的身份验证。
4. 允许安全对象调用继续（假设访问被授予）。
5. 如果配置了AfterInvocationManager,一旦调用被返回，调用AfterInvocationManager，如果调用抛出了一个异常，那么AfterInvocationManager不会调用。

什么是配置属性？

一个“配置属性”可以看成对于被AbstractSecurityInterceptor使用的具有特殊意义的字符串。它们被框架中的接口ConfigAttribute表示。他们可能是一个单一的角色名称或者具有更复杂的意义，这一切都取决于AccessDecisionManager实现的复杂程度。AbstractSecurityInterceptor被使用一个SecurityMetadataSource配置，SecurityMetadataSource被用来去查询安全对象属性。通常这个配置是对用户隐藏的。配置属性可以是被保护方法上的注解或者是被保护的URL的访问属性。例如，当我们在命名空间介绍中看到的

```
<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/>
```

上面的配置说明配置参数ROLE_A和ROLE_B适用于给定模式的web请求。事实上，对于默认的AccessDecisionManager配置，这个的意思是有一个GrantedAuthority匹配上述两个参数的人都允许访问。虽然严格意义上讲，它们只是一些属性，并且这些属性的解释最终依赖于AccessDecisionManager实现。使用前缀ROLE_作为标记是为了指示这些属性是角色属性，并且他们应该被Spring Security的RoleVoter使用。仅仅当基于voter的AccessDecisionManager使用时才是有关系的。我们将在授权章节看到AccessDecisionManager是如何实现的。

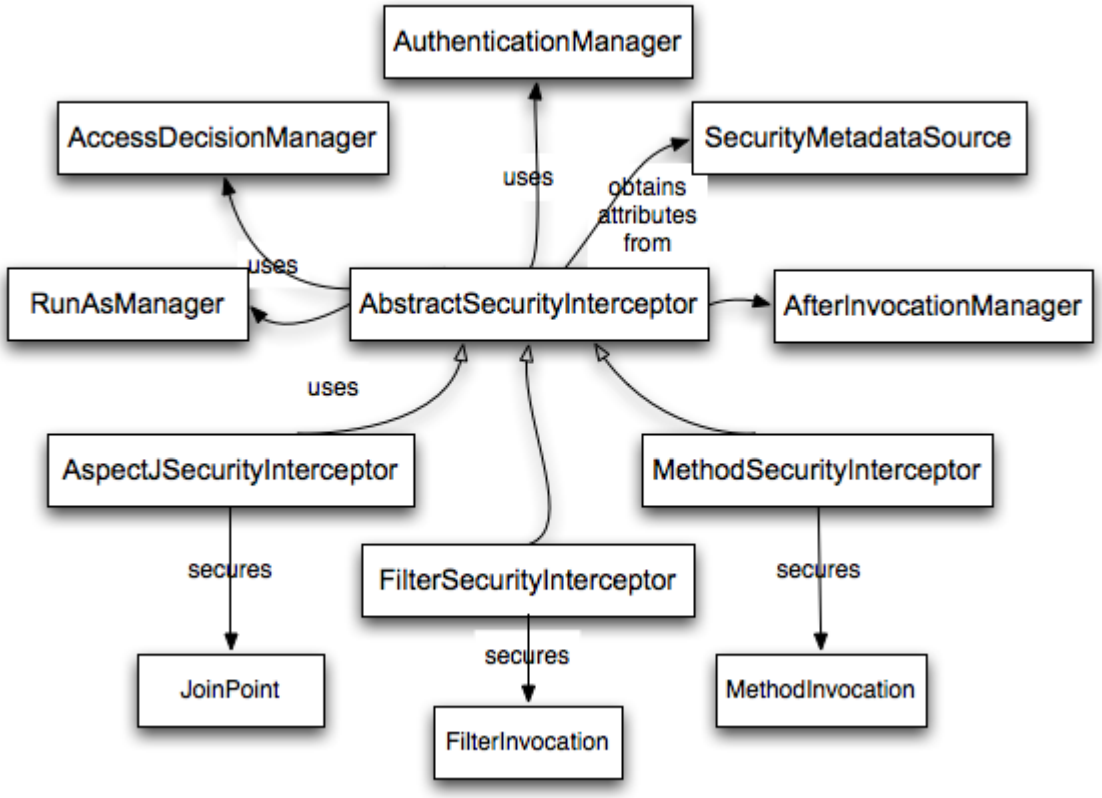
RunAsManager

假设AccessDecisionManager决定允许请求，那么AbstractSecurityInterceptor通常只会继续处理请求。尽管如此，在极少数情况下，用户可能希望使用不同的Authentication替换SecurityContext中的Authentication，由AccessDecisionManager调用RunAsManager来处理。这对于一些有原因的罕见环境可能是有用的，例如日过一个服务层方法需要去调用一个远程系统或存在一个不同的身份。由于Spring Security自动将安全身份从一台服务器传播到另一台服务器（假设您使用正确配置的RMI或HttpInvoker远程协议客户端），这可能是很有用的。

AfterInvocationManager

在安全对象调用过程之后，然后返回 - 这可能意味着方法调用完成或过滤器链处理 - AbstractSecurityInterceptor最终的机会处理调用。在这个阶段，AbstractSecurityInterceptor主要对修改返回对象更感兴趣。我们可能希望发生这种情况，因为授权决策无法在安全对象调用的“途中”中进行。高度的可拔插式的，AbstractSecurityInterceptor将会将这些控制传递给AfterInvocationManager去更改对象，如果需要的话。这个类甚至能完全替换对象，抛出一个异常，或者不更改对象。调用后检查只会在调用成功后才进行.如果一个异常出现了，这个额外的检查将会被跳过。

AbstractSecurityInterceptor和与他有关的对象都被展示在下图。



继承安全对象模型

只有开发人员想要采用全新的拦截和授权请求的方式，才需要直接使用安全对象。例如，可能想要建立一个新的安全对象去保护一个信息系统的消息。任何需要安全性并且还提供拦截调用的方式（如围绕通知语义的AOP）都可以变成安全的对象。话虽如此，大多数Spring应用程序将完全透明地使用目前支持的三种安全对象类型（AOP Alliance MethodInvocation，AspectJ JoinPoint和Web请求FilterInvocation）。

9.6 本地化

Spring Security 支持对用户可看到的异常信息的本地化。如果你的应用程序被设计给讲英文的用户，Spring Security的默认消息机制采用的就是英文。如果你想要支持其他的地方，这节包含了所有你需要知道的知识。

所有的异常信息都可以本地化，包括和验证实验有关的信息，和访问被拒绝有关的信息。被开发者或者系统部署者关心的异常和日志信息（包括不正确的属性，接口合同违规，使用不正确的构造函数，启动时间验证，调试级别日志记录）都不会被本地化，代替的是被使用英文硬编码在Spring Security的代码中。

在spring-security-core-xx.jar中你将会发现org.springframework.security包，该包中包含messages.properties文件以及一些常用语言的本地化版本。这应该由ApplicationContext引用，因为Spring Security类实现了Spring的MessageSourceAware接口，并期望在应用程序上下文启动时依赖注入消息解析器。通常，您只需在应用程序上下文中注册一个bean来引用这些消息。一个例子如下：

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
<property name="basename"
value="classpath:org/springframework/security/messages"/>
</bean>
```

`messages.properties`按照标准资源包进行命名，并表示Spring Security消息支持的默认语言。 这个默认文件是英文的。

如果你想要自定义`messages.properties`文件，或者支持其他语言，你应该复制这个文件，然后重新命名它，注册到你上面的bean定义中。在这个文件中没有很多消息key在李弥岸，所以本地化不被考虑为一个主要的举措。如果您确实执行了此文件的本地化，请考虑通过记录JIRA任务并附上您正确命名的messages.properties本地化版本来与社区分享您的工作。

Spring Security依靠Spring的本地化支持来实际查找适当的消息。为了达到这个目的，你必须确保传入请求的语言环境存储在Spring的org.springframework.context.i18n.LocaleContextHolder中。

Spring MVC的DispatcherServlet会自动为您的应用程序执行此操作，但由于在此之前调用了Spring Security的过滤器，因此需要在调用过滤器之前将LocaleContextHolder设置为包含正确的Locale。 你可以自己做一个过滤器（它必须在web.xml中的Spring Security过滤器之前），或者你可以使用Spring的RequestContextFilter。 有关在Spring中使用本地化的更多详细信息，请参阅Spring Framework文档。

“contacts”示例应用程序设置为使用本地化的消息。