Apache Shiro权限框架实战

主讲：安燚

目录

Contents

注意：代码和资料在视频课程最后章节
请在电脑浏览器上进行下载！

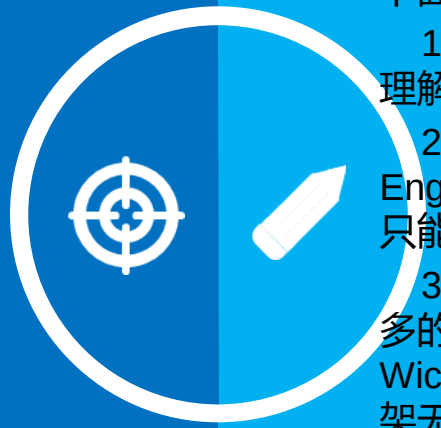
1 shiro简介

Apache Shiro是什么

Apache Shiro 是功能强大并且容易集成的开源权限框架，它能够完成认证、授权、加密、会话管理、与Web集成、缓存等。认证和授权为权限控制的核心，简单来说，

“认证”就是证明你是谁？ Web 应用程序一般做法通过表单提交用户名及密码达到认证目的。

“授权”即是否允许已认证用户访问受保护资源。



为何对 Shiro 情有独钟

Spring Security和Shiro ？

下面对两者略微比较：

1、简单性，Shiro 在使用上较 Spring Security 更简单，更容易理解。适合于入门。

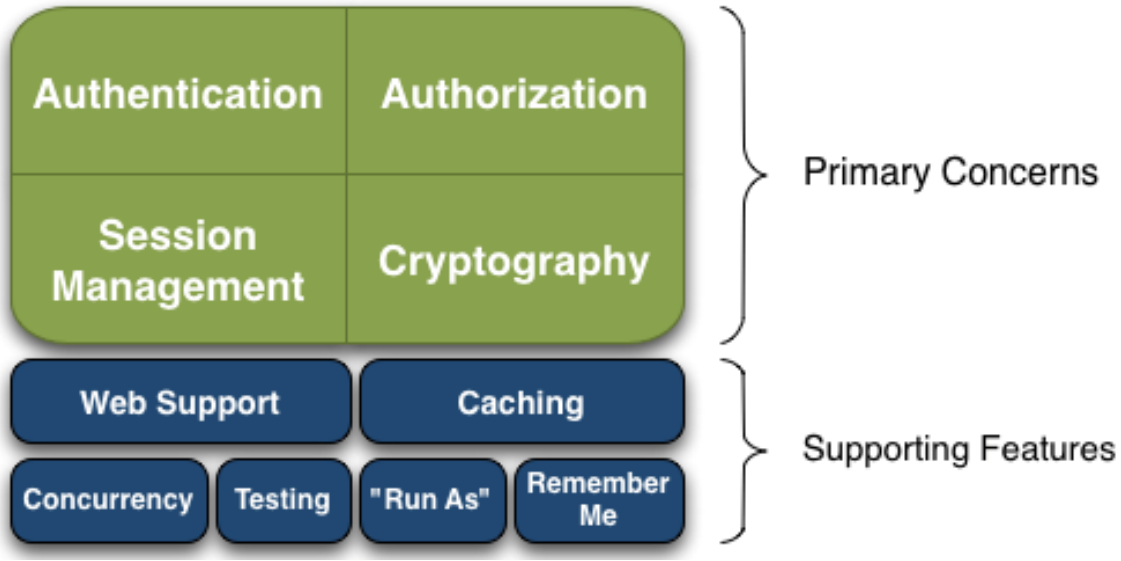
2、灵活性，Shiro 可运行在 Web、EJB、IoC、Google App Engine 等任何应用环境，却不依赖这些环境。而 Spring Security 只能与 Spring 一起集成使用。

3、可插拔，Shiro 干净的 API 和设计模式使它可以方便地与许多的其它框架和应用进行集成。Shiro 可以与诸如 Spring、Grails、Wicket、Tapestry、Mule、Apache Camel、Vaadin 这类第三方框架无缝集成。

Spring Security 在这方面就显得有些捉衿见肘。

1 shiro简介

基本功能



Authentication：身份认证/登录，验证用户是不是拥有相应的身份；

Authorization： 授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；

Session Manager：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通JavaSE环境的，也可以是如Web环境的；

Cryptography： 加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；

Web Support：Web支持，可以非常容易的集成到Web环境；

Caching：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率；

Concurrency： shiro支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；

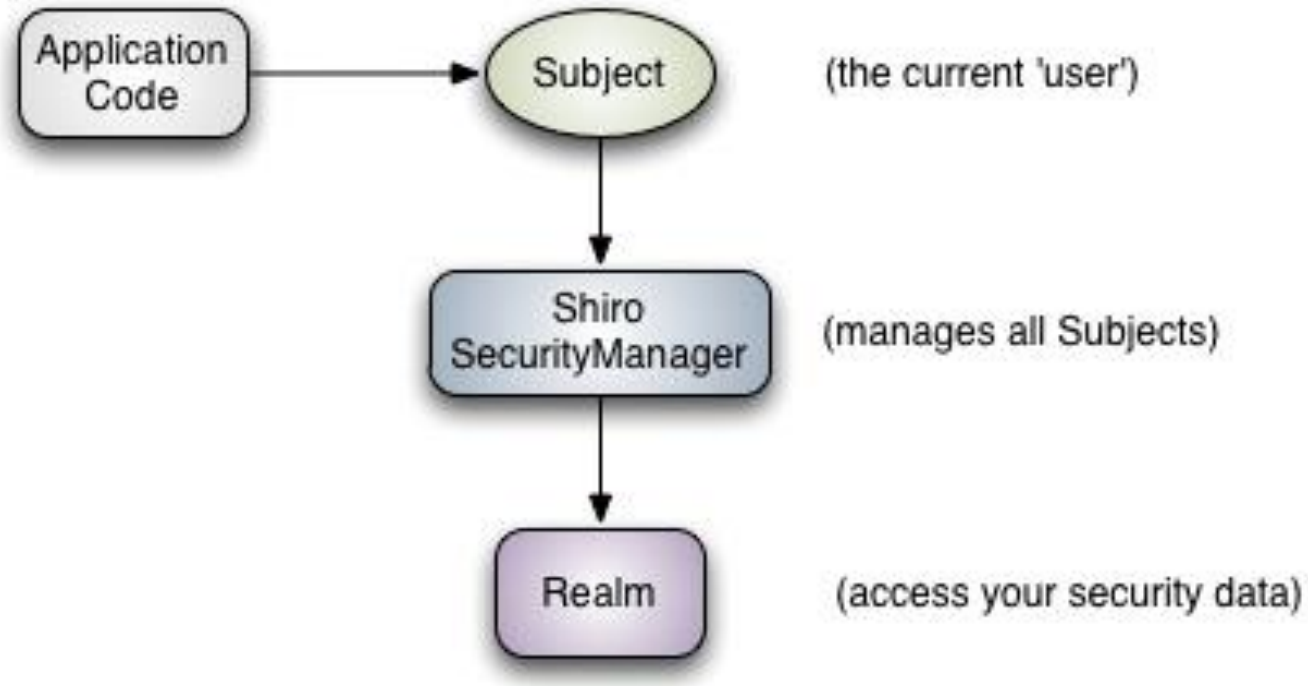
Testing：提供测试支持；

Run As：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；

Remember Me：记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了。

注意：Shiro不会去维护用户、维护权限；这些需要我们自己去设计/提供；然后通过相应的接口注入给Shiro即可。

1 shiro简介



从应用程序角度的来观察如何使用Shiro完成工作

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
2、我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。
从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。

1 shiro简介

从应用程序角度的来观察如何使用Shiro完成工作

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
2、我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。
从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。

1 shiro简介

从应用程序角度的来观察如何使用Shiro完成工作

可以看到：应用代码直接交互的对象是Subject，也就是说Shiro的对外API核心就是Subject；其每个API的含义：

Subject：主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等；即一个抽象概念；所有Subject 都绑定到SecurityManager，与Subject的所有交互都会委托给SecurityManager；可以把Subject认为是一个门面；SecurityManager才是实际的执行者；

SecurityManager：安全管理器；即所有与安全有关的操作都会与SecurityManager 交互；且它管理着所有Subject；可以看出它是Shiro 的核心，它负责与后边介绍的其他组件进行交互，如果学习过SpringMVC，你可以把它看成DispatcherServlet前端控制器；

Realm：域，Shiro从从Realm获取安全数据（如用户、角色、权限），就是说SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法；也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行操作；可以把Realm看成DataSource，即安全数据源。

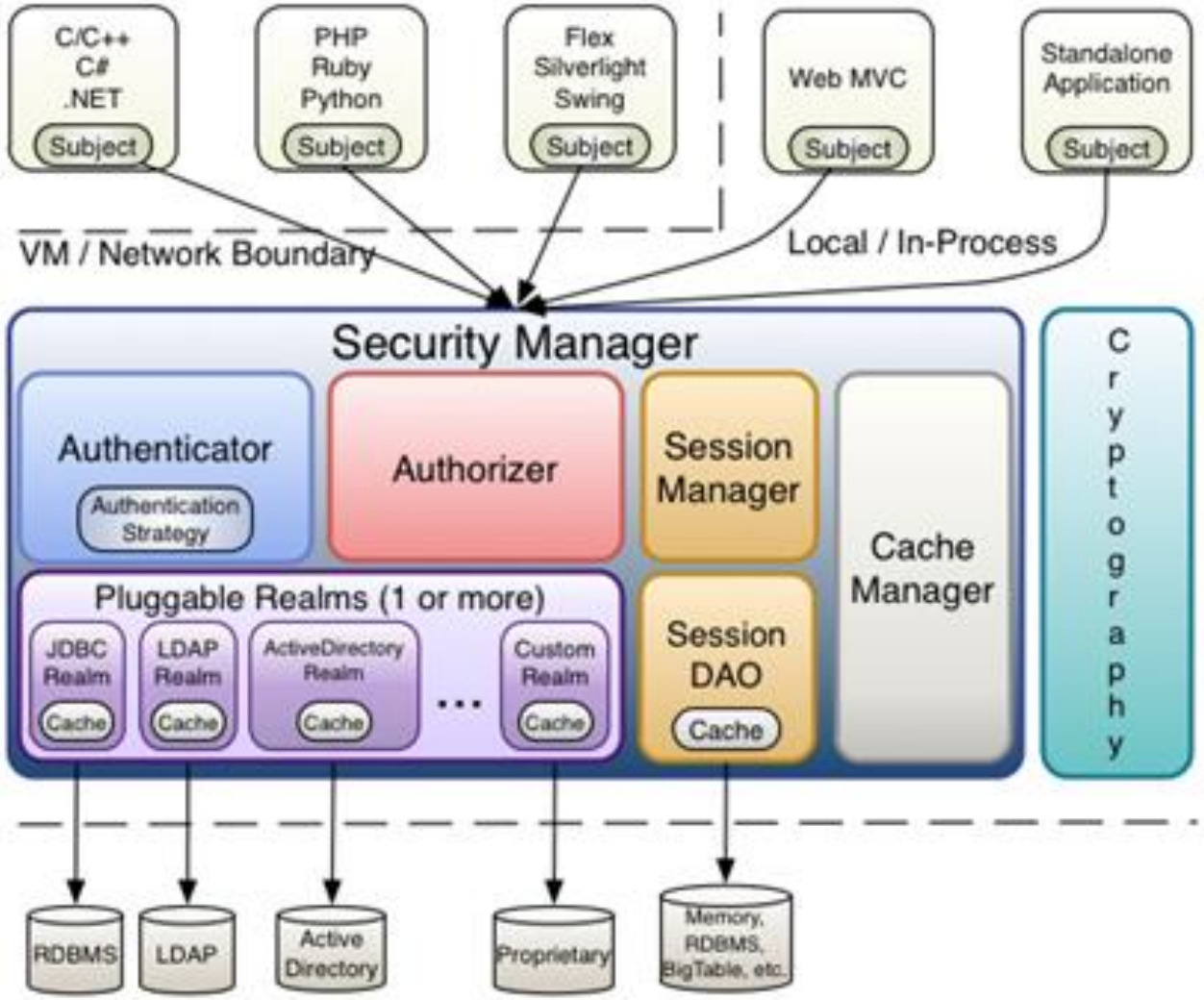
也就是说对于我们而言，最简单的一个Shiro 应用：

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；

2、我们需要给Shiro 的SecurityManager 注入Realm，从而让SecurityManager 能得到合法的用户及其权限进行判断。

1 shiro简介

Shiro的内部架构



1 shiro简介

Shiro的内部架构

Subject：主体，可以看到主体可以是任何可以与应用交互的“用户”；

SecurityManager ：相当于SpringMVC 中的DispatcherServlet 或者Struts2 中的FilterDispatcher；是Shiro的心脏；所有具体的交互都通过SecurityManager进行控制；它管理着所有Subject、且负责进行认证和授权、及会话、缓存的管理。

Authenticator： 认证器，负责主体认证的，这是一个扩展点，如果用户觉得Shiro 默认的不好，可以自定义实现；其需要认证策略（Authentication Strategy），即什么情况下算用户认证通过了；

Authrizer：授权器，或者访问控制器，用来决定主体是否有权限进行相应的操作；即控制着用户能访问应用中的哪些功能；

Realm：可以有1个或多个Realm，可以认为是安全实体数据源，即用于获取安全实体的；可以是JDBC 实现，也可以是LDAP 实现，或者内存实现等等；由用户提供；注意：Shiro不知道你的用户/权限存储在哪及以何种格式存储；所以我们一般在应用中都需要实现自己的Realm；

SessionManager：如果写过Servlet就应该知道Session的概念，Session呢需要有人去管理它的生命周期，这个组件就是SessionManager；而Shiro 并不仅仅可以用在Web 环境，也可以用在如普通的JavaSE 环境、EJB 等环境；所有呢，Shiro 就抽象了一个自己的Session来管理主体与应用之间交互的数据；这样的话，比如我们在Web 环境用，刚开始是一台Web 服务器；接着又上了台EJB 服务器；这时想把两台服务器的会话数据放到一个地方，这个时候就可以实现自己的分布式会话（如把数据放到Memcached服务器）；

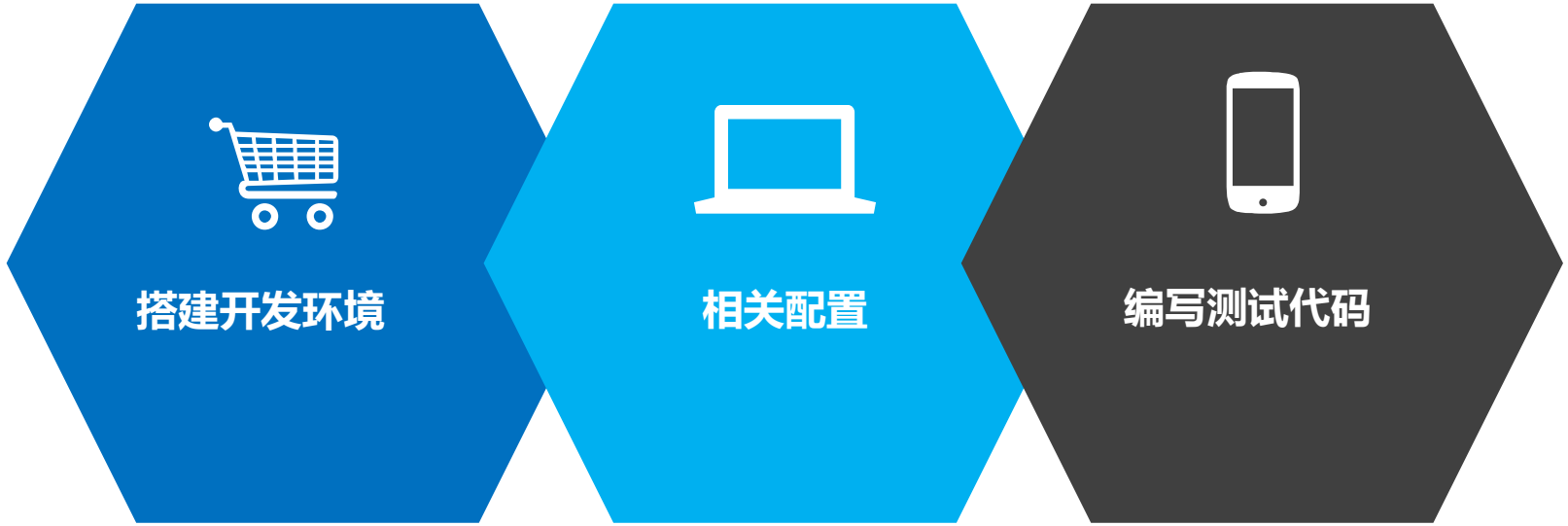
SessionDAO：DAO 大家都用过，数据访问对象，用于会话的CRUD，比如我们想把Session保存到数据库，那么可以实现自己的SessionDAO，通过如JDBC 写到数据库；比如想把Session 放到Memcached 中，可以实现自己的Memcached SessionDAO；另外SessionDAO中可以使用Cache进行缓存，以提高性能；

CacheManager：缓存控制器，来管理如用户、角色、权限等的缓存的；因为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能

Cryptography：密码模块，Shiro 提高了一些常见的加密组件用于如密码加密/解密的。

2 QuickStart

下载地址：
http://shiro.apache.org/download.html



创建java project
加入如下jar包
commons-beanutils-1.8.3.jar
log4j-1.2.17.jar
shiro-core-1.3.2.jar (shiro-all)
slf4j-api-1.6.4.jar
slf4j-log4j12-1.7.2.jar

log4j.properties
shiro.ini

取得相关配置，并模拟
进行登录验证

3 spring和shiro的整合

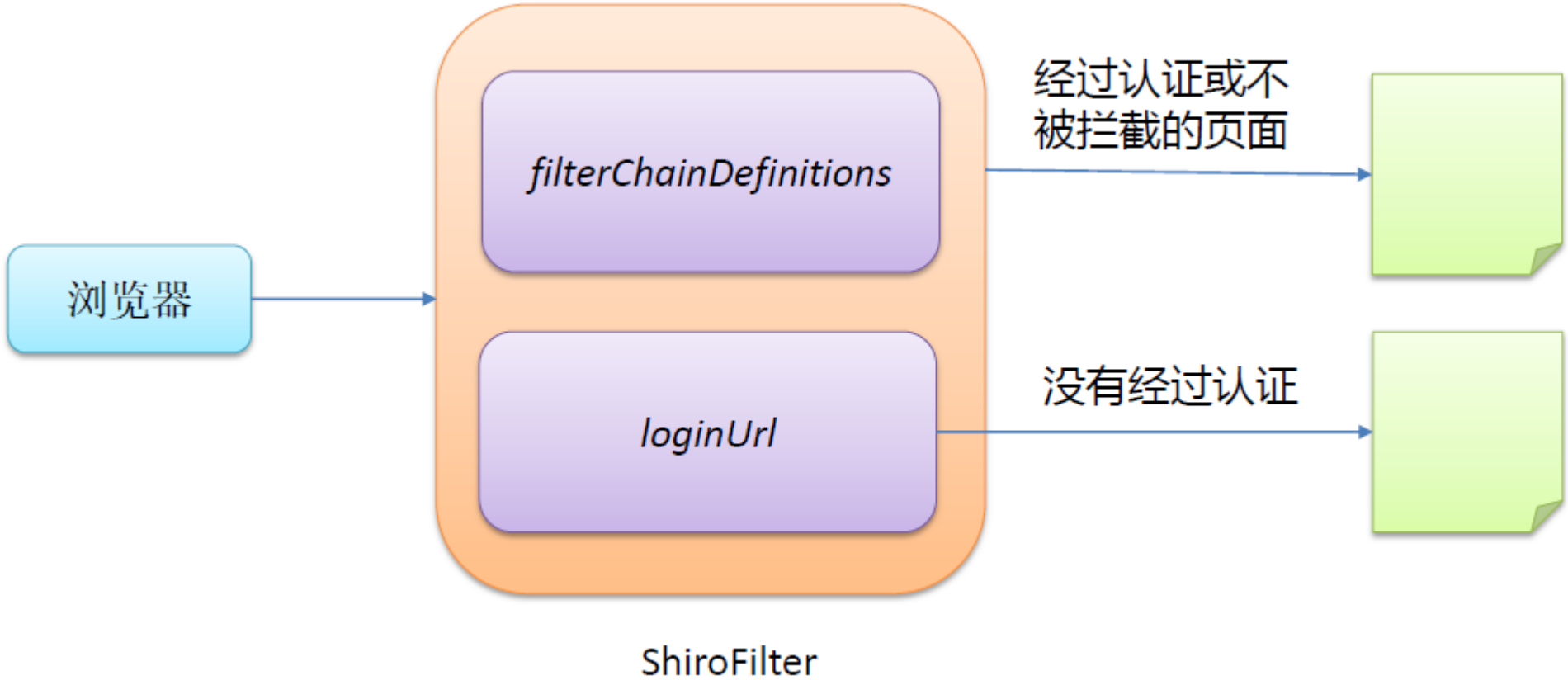
spring配置
web.xml
applicationContext.xml

springmvc配置
web.xml
spring-servlet.xml

shiro配置
web.xml
applicationContext.xml

4 shiro工作流程和注意事项

ShiroFilter工作原理



4 shiro工作流程和注意事项

注意事项

```
<filter>
<filter-name>shiroFilter</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
<init-param>
<param-name>targetFilterLifecycle</param-name>
<param-value>true</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>shiroFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

DelegatingFilterProxy作用是自动到spring容器查找名字为shiroFilter（filter-name）的bean
并把所有Filter的操作委托给它。然后将ShiroFilter 配置到spring容器即可

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
<property name="securityManager" ref="securityManager"/>
</bean>
```

5 url匹配规则

匹配规则

urls格式是 url=拦截器[参数]

/login.jsp = anon

通配符

url模式使用Ant风格模式

Ant路径通配符支持?、*、**，注意通配符匹配不包括目录分隔符“/”：

?：匹配一个字符，如”/admin?”将匹配/admin1;/admin2；但不匹配/admin

或*：匹配零个或多个字符串，如/admin*将匹配/admin、/admin123，但不匹配/admin/1；

：匹配路径中的零个或多个路径，如/admin/将匹配/admin/a或/admin/a/b。

5 url匹配规则

url模式匹配顺序

url 模式匹配顺序是按照在配置中的声明顺序匹配，即从头开始使用第一个匹配的url 模式对应的拦截器链。如：

/bb/**=filter1
/bb/aa=filter2
/**=filter3
如果请求的url是“/bb/aa”，因为按照声明顺序进行匹配，那么将使用filter1进行拦截。

5 url匹配规则

Shiro的内部架构

Shiro 内置了很多默认的拦截器，比如身份验证、授权等相关的。默认拦截器
org.apache.shiro.web.filter.mgt.DefaultFilter中的枚举拦截器：

默认拦截器名	拦截器类	说明（括号里的表示默认值）
身份验证相关的		
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter	基于表单的拦截器；如“/**=authc”，如果没有登录会跳到相应的登录页面登录；主要属性：usernameParam：表单提交的用户名参数名（ username）；passwordParam：表单提交的密码参数名（password）； rememberMeParam：表单提交的密码参数名（rememberMe）；loginUrl：登录页面地址（/login.jsp）；successUrl：登录成功后的默认重定向地址； failureKeyAttribute：登录失败后错误信息存储 key（shiroLoginFailure）；
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter	Basic HTTP 身份验证拦截器，主要属性：applicationName：弹出登录框显示的信息（application）；
logout	org.apache.shiro.web.filter.authc.LogoutFilter	退出拦截器，主要属性：redirectUrl：退出成功后重定向的地址（/）;示例“/logout=logout”
user	org.apache.shiro.web.filter.authc.UserFilter	用户拦截器，用户已经身份验证/记住我登录的都可；示例“/**=user”
anon	org.apache.shiro.web.filter.authc.AnonymousFilter	匿名拦截器，即不需要登录即可访问；一般用于静态资源过滤；示例“/static/**=anon”

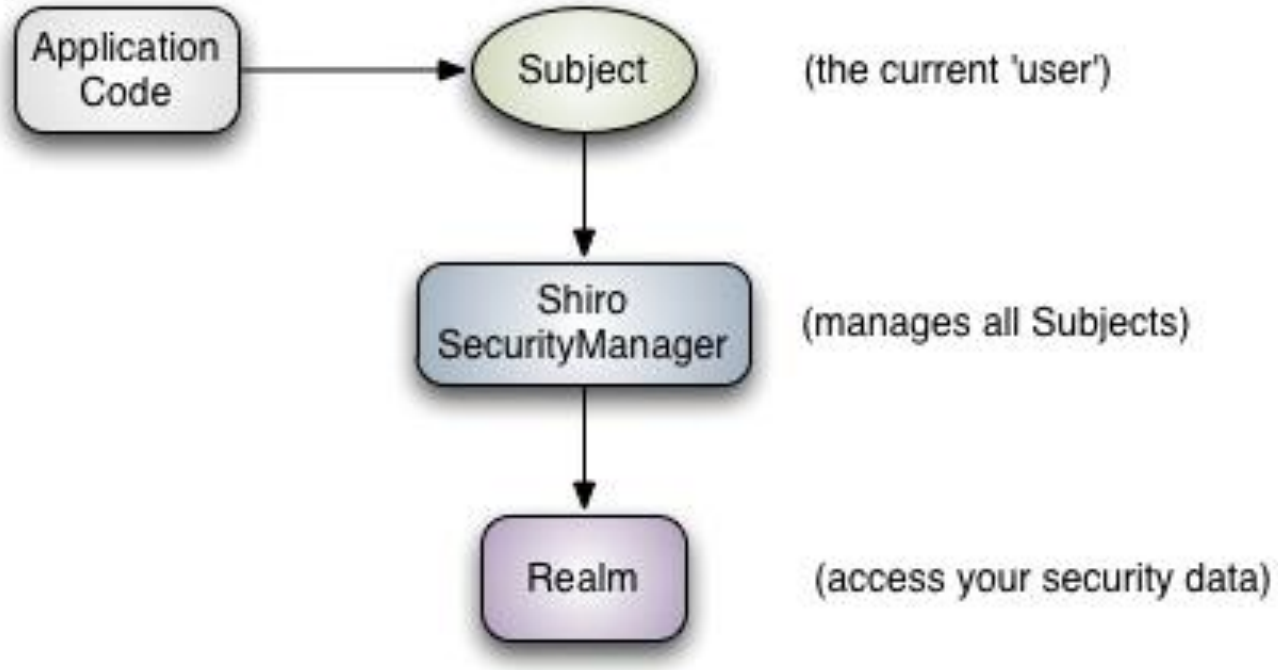
5 url匹配规则

Shiro的内部架构

Shiro 内置了很多默认的拦截器，比如身份验证、授权等相关的。默认拦截器可以参
org.apache.shiro.web.filter.mgt.DefaultFilter中的枚举拦截器：

授权相关的		
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter	角色授权拦截器，验证用户是否拥有角色；主要属性： loginUrl：登录页址（/login.jsp）；unauthorizedUrl：权后重定向的地址；示“/admin/**=roles[admin]”
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter	权限授权拦截器，验证用户是否拥有所有权限；属性和 roles 一样；示例“/user/**=perms["user:create"]”
port	org.apache.shiro.web.filter.authz.PortFilter	端口拦截器，主要属性：port（80）：可以通过的端口；示例“/test= port[80]”，如果用户访问该页面是非 80，将自动将请求端口改为 80 并重定向到该 80 端口，其他路径/参数等都一样
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter	rest 风格拦截器，自动根据请求方法构建权限字符串（GET=read, POST=create,PUT=update,DELETE=delete,HEAD=read,TRACE=read,OPTIONS=read, MKCOL=create）构建权限字符串；示例“/users=rest[user]”，会自动拼出“user:read,user:create,user:update,user:delete”权限字符串进行权限匹配（所有都得匹配，isPermittedAll）；
ssl	org.apache.shiro.web.filter.authz.SslFilter	SSL 拦截器，只有请求协议是 https 才能通过；否则自动跳转会 https 端口（443）；其他和 port 拦截器一样；
其他		
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter	不创建会话拦截器，调用 subject.getSession(false)不会有什么问题，但是如果 subject.getSession(true)将抛出 DisabledSessionException 异常；

6 认证流程原理

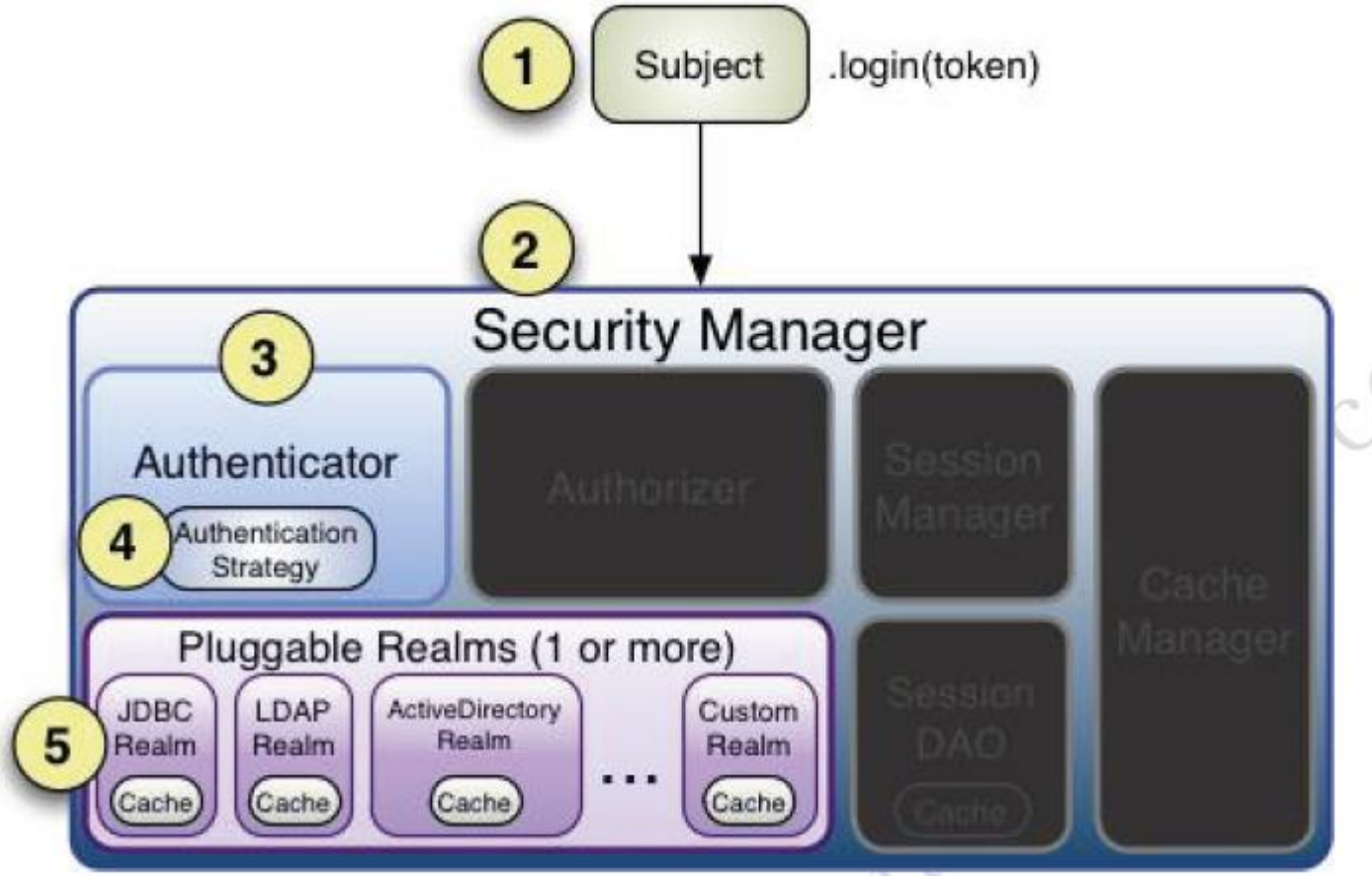


从应用程序角度的来观察如何使用Shiro完成工作

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
2、我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。
从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。

6 认证流程原理

身份认证流程



6 认证流程原理

流程如下：

1、首先调用Subject.login(token)进行登录，其会自动委托给Security Manager，调用之前必须通过SecurityUtils. setSecurityManager()设置；
2、SecurityManager负责真正的身份验证逻辑；它会委托给Authenticator进行身份验证；
3、Authenticator才是真正的身份验证者，Shiro API中核心的身份认证入口点，此处可以自定义插入自己的实现；
4、Authenticator可能会委托给相应的AuthenticationStrategy进行多Realm身份验证，默认ModularRealmAuthenticator会调用AuthenticationStrategy进行多Realm身份验证；
5、Authenticator 会把相应的token 传入Realm，从Realm 获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个Realm，将按照相应的顺序及策略进行访问。

6 认证流程原理

身份验证的基本流程

1、收集用户身份/凭证，即如用户名/密码
2、调用subject.login进行登录，如果失败将返回相应的authenticationException异常，
根据异常提示用户错误信息；否则登录成功。
3、创建自定义的RealM类，继承org.apache.shiro.realm.Authorizingrealm类，
实现doGetAuthenticationInfo()方法

7 认证流程实现

01 创建一个表单页面，录入用户名、密码

02 把请求提交到 SpringMVC 的 controller

applicationContext.xml中添加匿名
访问: /shiro/login=anon

03 获取用户名和密码进行认证

在realm中实现认证

7 认证流程实现

身份认证代码

```
// 检测当前用户是否没有被认证.
if (!currentUser.isAuthenticated()) {
    // 创建封装了用户名和密码的 UsernamePasswordToken 对象.
    UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");
    token.setRememberMe(true);
    try {
        currentUser.login(token);
    }
    // 若登陆失败, 则可以通过捕获异常的方式来处理其他各种情况
    catch (UnknownAccountException uae) {
        log.info("There is no user with username of " + token.getPrincipal());
    } catch (IncorrectCredentialsException ice) {
        log.info("Password for account " + token.getPrincipal() + " was incorrect!");
    } catch (LockedAccountException lae) {
        log.info("The account for username " + token.getPrincipal() + " is locked.  " +
                "Please contact your administrator to unlock it.");
    }
    // ... catch more exceptions here (maybe custom ones specific to your application?
    catch (AuthenticationException ae) {
        //unexpected condition?  error?
    }
}
```

7 认证流程实现

认证异常

如果身份认证失败的场合，尽量捕获authenticationException类或其子类
但是提示信息需要使用规范，不可提示【用户名错误】、【密码错误】
应该提示【用户名/密码错误】

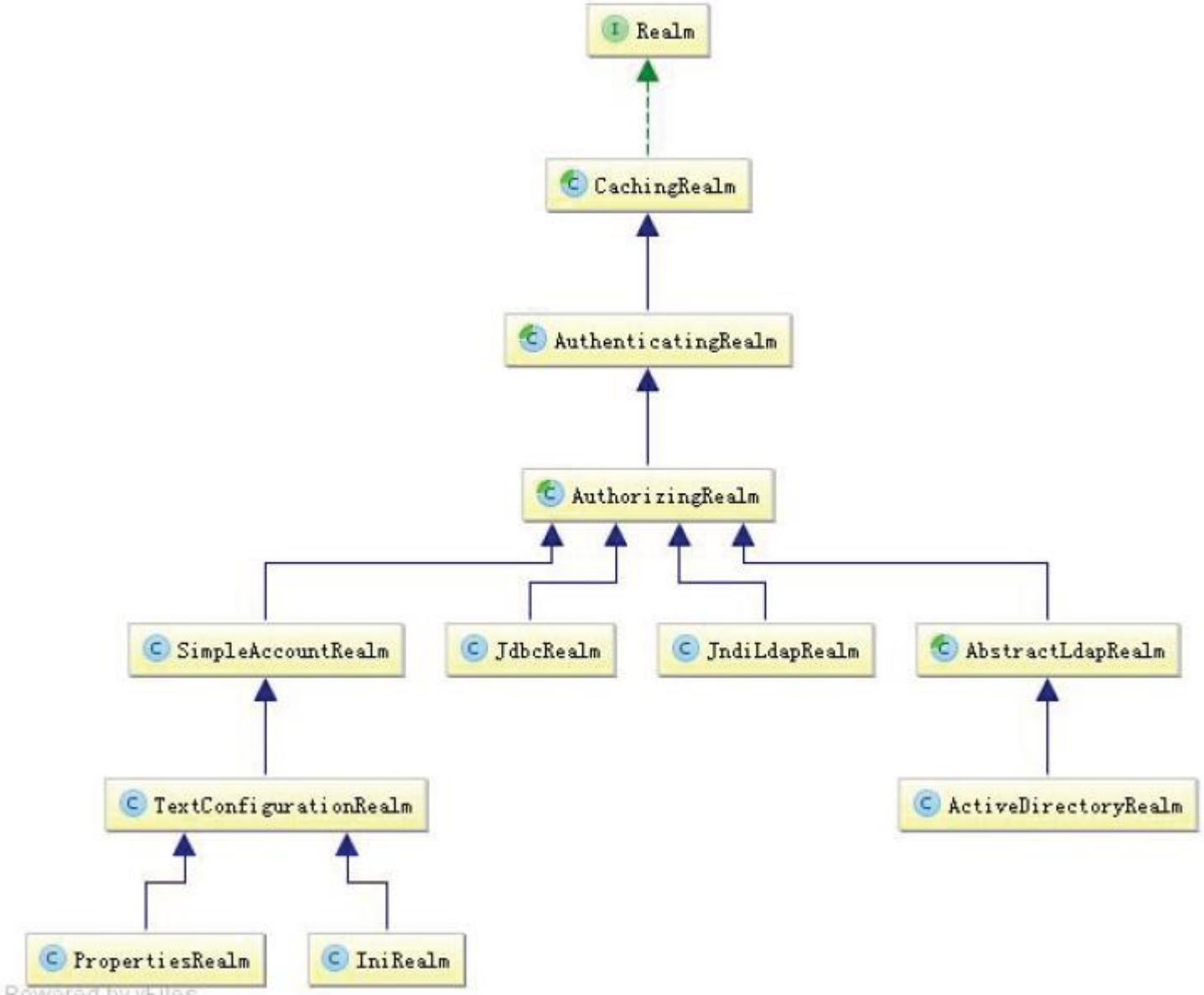
7 认证流程实现

Realm
Realm：域，Shiro 从从Realm获取安全数据（如用户、角色、权限），就是说SecurityManager
要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法；
也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行操作；可以把Realm看
成DataSource ， 即安全数据源。如我们之前的ini 配置方式将使用
org.apache.shiro.realm.text.IniRealm。
org.apache.shiro.realm.Realm接口如下：

String getName(); //返回一个唯一的Realm名字
boolean supports(AuthenticationToken token); //判断此Realm是否支持此Token
AuthenticationInfo getAuthenticationInfo(AuthenticationToken token)
throws AuthenticationException; //根据Token获取认证信息

7 认证流程实现

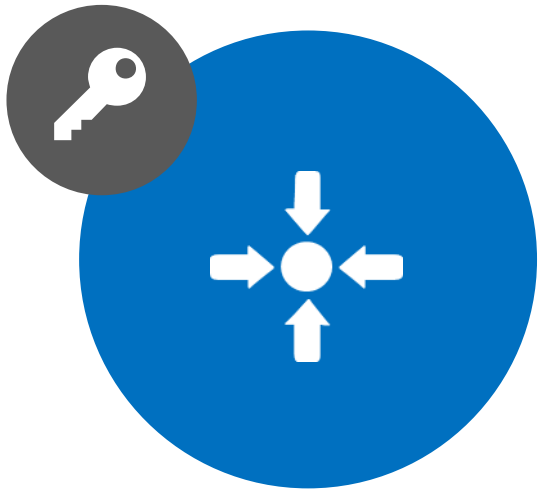
接口图


Realm
CachingRealm
AuthenticatingRealm
AuthorizingRealm
SimpleAccountRealm
JdbcRealm
JndiLdapRealm
AbstractLdapRealm
TextConfigurationRealm
ActiveDirectoryRealm
PropertiesRealm
IniRealm

7 认证流程实现

一般继承AuthorizingRealm（授权）即可；其继承了AuthenticatingRealm（即身份验证），而且也间接继承了CachingRealm（带有缓存实现）。其中主要默认实现如下：
org.apache.shiro.realm.text.IniRealm：[users]部分指定用户名/密码及其角色；[roles]部分指定角色即权限信息；
org.apache.shiro.realm.text.PropertiesRealm：user.username=password,role1,role2指定用户名/密码及其角色；role.role1=permission1,permission2指定角色及权限信息；
org.apache.shiro.realm.jdbc.JdbcRealm：通过sql查询相应的信息，如“select password from users where username = ?”获取用户密码，“select password, password_salt from users where username = ?”获取用户密码及盐；“select role_name from user_roles where username = ?”获取用户角色；“select permission from roles_permissions where role_name = ?”获取角色对应的权限信息；也可以调用相应的api进行自定义sql；

8 密码认证和加密



01 shiro是如何验证密码

02 如何进行密码的加密

03 盐值加密

解决普通加密的问题

9 多realm认证和策略

Authenticator

Authenticator的职责是验证用户帐号，是Shiro API中身份验证核心的入口点：

public AuthenticationInfo authenticate(AuthenticationToken authenticationToken)

throws AuthenticationException;

如果验证成功，将返回AuthenticationInfo 验证信息；此信息中包含了身份及凭证；如果验证失败将抛出相应的AuthenticationException实现。

SecurityManager接口继承了Authenticator，另外还有一个ModularRealmAuthenticator实现，

其委托给多个Realm 进行验证，验证规则通过AuthenticationStrategy 接口指定，默认提供的实现：

FirstSuccessfulStrategy：只要有一个Realm验证成功即可，只返回第一个Realm身份验证成功的认证信息，其他的忽略；

AtLeastOneSuccessfulStrategy：只要有一个Realm验证成功即可，和FirstSuccessfulStrategy不同，返回所有Realm身份验证成功的认证信息；

AllSuccessfulStrategy：所有Realm验证成功才算成功，且返回所有Realm身份验证成功的认证信息，如果有一个失败就失败了。

ModularRealmAuthenticator默认使用AtLeastOneSuccessfulStrategy策略。

10 授权流程原理

授权

授权，也叫访问控制，即在应用中控制谁能访问哪些资源（如访问页面/编辑数据/页面操作等）。在授权中需了解的几个关键对象：主体（Subject）、资源（Resource）、权限（Permission）、角色（Role）。

主体

主体，即访问应用的用户，在Shiro中使用Subject代表该用户。用户只有授权后才允许访问相应的资源。

资源

在应用中用户可以访问的任何东西，比如访问JSP 页面、查看/编辑某些数据、访问某个业务方法、打印文本等等都是资源。用户只要授权后才能访问。

权限

安全策略中的原子授权单位，通过权限我们可以表示在应用中用户有没有操作某个资源的权力。即权限表示在应用中用户能不能访问某个资源，如：

访问用户列表页面

查看/新增/修改/删除用户数据（即很多时候都是CRUD（增查改删）式权限控制）

打印文档等等。。。

角色

角色代表了操作集合，可以理解为权限的集合，一般情况下我们会赋予用户角色而不是权限，即这样用户可以拥有一组权限，赋予权限时比较方便。典型的如：项目经理、技术总监、CTO、开发工程师等都是角色，不同的角色拥有一组不同的权限。

10 授权流程原理

授权方式

Shiro 支持三种方式的授权：
编程式：通过写if/else 授权代码块完成：

```
Subject subject = SecurityUtils.getSubject();
if(subject.hasRole(“admin”)) {
//有权限
} else {
//无权限
}
```

注解式：通过在执行的Java方法上放置相应的注解完成：

```
@RequiresRoles("admin")
public void hello() {
//有权限
}
```

没有权限将抛出相应的异常；

JSP/GSP 标签：在JSP/GSP 页面通过相应的标签完成：

```
<shiro:hasRole name="admin">
<!— 有权限—>
</shiro:hasRole>
```

10 授权流程原理

Shiro的内部架构

Shiro 内置了很多默认的拦截器，比如身份验证、授权等相关的。默认拦截器可以参
org.apache.shiro.web.filter.mgt.DefaultFilter中的枚举拦截器：

授权相关的		
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter	角色授权拦截器，验证用户是否拥有所角色；主要属性： loginUrl：登录页面址（/login.jsp）；unauthorizedUrl：未权后重定向的地址；示“/admin/**=roles[admin]”
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter	权限授权拦截器，验证用户是否拥有所有权限；属性和 roles 一样；示例“/user/**=perms["user:create"]”
port	org.apache.shiro.web.filter.authz.PortFilter	端口拦截器，主要属性：port（80）：可以通过的端口；示例“/test= port[80]”，如果用户访问该页面是非 80，将自动将请求端口改为 80 并重定向到该 80 端口，其他路径/参数等都一样
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter	rest 风格拦截器，自动根据请求方法构建权限字符串（GET=read,POST=create,PUT=update,DELETE=delete,HEAD=read,TRACE=read,OPTIONS=read, MKCOL=create）构建权限字符串；示例“/users=rest[user]”，会自动拼出“user:read,user:create,user:update,user:delete”权限字符串进行权限匹配（所有都得匹配，isPermittedAll）；
ssl	org.apache.shiro.web.filter.authz.SslFilter	SSL 拦截器，只有请求协议是 https 才能通过；否则自动跳转会 https 端口（443）；其他和 port 拦截器一样；
其他		
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter	不创建会话拦截器，调用 subject.getSession(false)不会有什么问题，但是如果 subject.getSession(true)将抛出 DisabledSessionException 异常；

10 授权流程原理

授权流程分析

认证的realm我们需要实现extends AuthenticatingRealm

授权的话我们需要实现extends AuthorizingRealm，AuthorizingRealm是AuthenticatingRealm的子类。但是没有实现AuthenticatingRealm 中的doGetAuthenticationInfo

所以认证和授权只需要继承 AuthorizingRealm 就可以了. 同时实现他的两个抽象方法.

protected AuthorizationInfo doGetAuthorizationInfo用于授权的方法

protected AuthenticationInfo doGetAuthenticationInfo用于认证的方法

11 授权流程实现

实战流程如下：

1、shirorealm的继承类修改一下AuthorizingRealm

2、添加了doGetAuthorizationInfo方法来回调

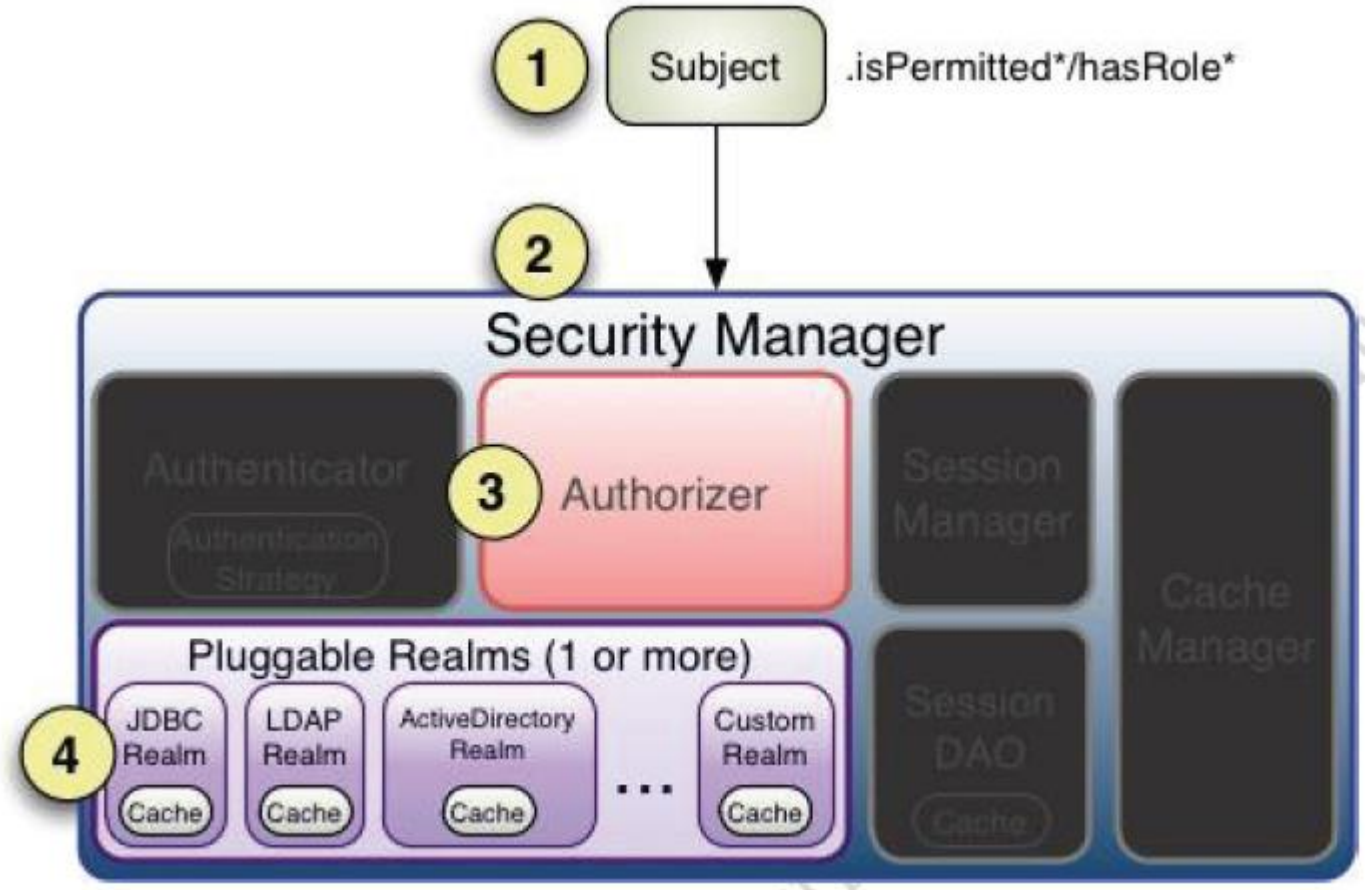
3、从PrincipalCollection中来获取登录的信息

4、利用登录的用户信息来获取当前用户的角色或权限

5、根据AuthorizationInfo接口创建实现类SimpleAuthorizationInfo，并设置roles属性

11 授权流程实现

授权流程



11 授权流程实现

流程如下：参考quickstart

1、首先调用Subject.isPermitted*/hasRole*接口，其会委托给SecurityManager，而SecurityManager接着会委托给Authorizer；

2、Authorizer是真正的授权者，如果我们调用如isPermitted(“user:view”)，其首先会通过PermissionResolver把字符串转换成相应的Permission实例；

3、在进行授权之前，其会调用相应的Realm获取Subject相应的角色/权限用于匹配传入的角色/权限；

4、Authorizer会判断Realm的角色/权限是否和传入的匹配，如果有多个Realm，会委托给ModularRealmAuthorizer 进行循环判断，如果匹配如isPermitted*/hasRole*会返回true，否则返回false表示授权失败。

11 授权流程实现

ModularRealmAuthorizer进行多Realm匹配流程：
1、首先检查相应的Realm是否实现了实现了Authorizer；
2、如果实现了Authorizer，那么接着调用其相应的isPermitted*/hasRole*接口进行匹配；
3、如果有一个Realm匹配那么将返回true，否则返回false。

如果Realm进行授权的话，应该继承AuthorizingRealm，其流程是：
1.1、如果调用hasRole*，则直接获取AuthorizationInfo.getRoles()与传入的角色比较即可；
1.2、首先如果调用如isPermitted(“user:view”)，首先通过PermissionResolver 将权限字符串转换成相应的Permission 实例，默认使用WildcardPermissionResolver，即转换为通配符的WildcardPermission；
2 、通过AuthorizationInfo.getObjectPermissions() 得到Permission 实例集合；通过AuthorizationInfo. getStringPermissions()得到字符串集合并通过PermissionResolver 解析为Permission 实例；然后获取用户的角色，并通过RolePermissionResolver 解析角色对应的权限集合（默认没有实现，可以自己提供）；
3、接着调用Permission. implies(Permission p)逐个与传入的权限比较，如果有匹配的则返回true，否则false。

11 授权流程实现

Permission
字符串通配符权限
规则： “资源标识符：操作：对象实例ID” 即对哪个资源的哪个实例可以进行什么操作。其默认支持通配符权限字符串，
“:”表示资源/操作/实例的分割； “,”表示操作的分割；
“*” 表示任意资源/操作/实例。

1、单个资源单个权限
subject().checkPermissions("system:user:update");
用户拥有资源 “system:user”的 “update”权限。

2、单个资源多个权限
role41=system:user:update,system:user:delete
然后通过如下代码判断
subject().checkPermissions("system:user:update", "system:user:delete");
用户拥有资源 “system:user”的 “update”和 “delete”权限。

如上可以简写成：（表示角色4 拥有system:user资源的update和delete 权限）
role42="system:user:update,delete"
接着可以通过如下代码判断
subject().checkPermissions("system:user:update,delete");
通过 “system:user:update,delete”验证"system:user:update, system:user:delete"是没问题的，

11 授权流程实现

3、单个资源全部权限
role51="system:user:create,update,delete,view"
然后通过如下代码判断
用户拥有资源“system:user”的“create”、“update”、“delete”和“view”所有权限。
如上可以简写成：（表示角色5拥有system:user的所有权限）
role52=system:user:*
也可以简写为（推荐上边的写法）：
role53=system:user
然后通过如下代码判断
subject().checkPermissions("system:user:*");
subject().checkPermissions("system:user");
通过“system:user:*”验证“system:user:create,delete,update:view”可以，但是反过来是不成立的。

4、所有资源全部权限
role61=*:view
然后通过如下代码判断
subject().checkPermissions("user:view");
用户拥有所有资源的“view”所有权限。假设判断的权限是“"system:user:view”，那么需要“role5=*:*:view”这样写才行。
role51="system:user:create,update,delete,view"
subject().checkPermissions("system:user:create,delete,update:view");

11 授权流程实现

5、实例级别的权限
5.1、单个实例单个权限
role71=user:view:1
对资源user的1 实例拥有view权限。
然后通过如下代码判断
subject().checkPermissions("user:view:1");

5.2、单个实例多个权限
role72="user:update,delete:1"
对资源user的1 实例拥有update、delete权限。然后通过如下代码判断
subject().checkPermissions("user:delete,update:1");
subject().checkPermissions("user:update:1", "user:delete:1");

5.3、单个实例所有权限
role73=user:*:1
对资源user的1 实例拥有所有权限。然后通过如下代码判断
subject().checkPermissions("user:update:1", "user:delete:1", "user:view:1");

5.4、所有实例单个权限
role74=user:auth:*
对资源user的1 实例拥有所有权限。然后通过如下代码判断
subject().checkPermissions("user:auth:1", "user:auth:2");

11 授权流程实现

5.5、所有实例所有权限
role75=user:*:*
对资源user的1 实例拥有所有权限。
然后通过如下代码判断
subject().checkPermissions("user:view:1", "user:auth:2");

6、Shiro对权限字符串缺失部分的处理
如“user:view”等价于“user:view:*”；而“organization”等价于“organization:*”或者“organization:*:*”。可以这么理解，这种方式实现了前缀匹配。
另外如“user:*”可以匹配如“user:delete”、“user:delete”可以匹配如“user:delete:1”、“user:*:1”可以匹配如“user:view:1”、“user”可以匹配“user:view”或“user:view:1”等。即*可以匹配所有，不加*可以进行前缀匹配；但是如“*:view”不能匹配“system:user:view”，需要使用“*:*:view”，即后缀匹配必须指定前缀（多个冒号就需要多个*来匹配）。

7、WildcardPermission
如下两种方式是等价的：
subject().checkPermission("menu:view:1");
subject().checkPermission(new WildcardPermission("menu:view:1"));
因此没什么必要的话使用字符串更方便。

8、性能问题
通配符匹配方式比字符串相等匹配来说是更复杂的，因此需要花费更长时间，但是一般系统的权限不会太多，且可以配合缓存来提供其性能，如果这样性能还达不到要求我们可以实现位操作算法实现性能更好的权限匹配。另外实例级别的权限验证如果数据量太大也不建议使用，可能造成查询权限及匹配变慢。可以考虑比如在sql 查询时加上权限字符串之类的方式在查询时就完成了权限匹配。

12 标签

JSP 标签
Shiro 提供了JSTL标签用于在JSP/GSP 页面进行权限控制，如根据登录用户显示相应的页面按钮。
导入标签库
<%@taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
标签库定义在shiro-web.jar包下的META-INF/shiro.tld 中定义。

guest标签
<shiro:guest>
欢迎游客访问，登录
</shiro:guest>
用户没有身份验证时显示相应信息，即游客访问信息。

user标签
<shiro:user>
欢迎[<shiro:principal/>]登录，退出
</shiro:user>
用户已经身份验证/记住我登录后显示相应的信息。

12 标签

authenticated标签
<shiro:authenticated>
用户[<shiro:principal/>]已身份验证通过
</shiro:authenticated>
用户已经身份验证通过，即Subject.login登录成功，不是记住我登录的。

notAuthenticated标签
<shiro:notAuthenticated>
未身份验证（包括记住我）
</shiro:notAuthenticated>
用户已经身份验证通过，即没有调用Subject.login进行登录，包括记住我自动登录的也属于未进行身份验证。

principal标签
<shiro: principal/>
显示用户身份信息，默认调用Subject.getPrincipal()获取，即Primary Principal。
<shiro:principal property="username"/>
相当于((User)Subject.getPrincipals()).getUsername()。

lacksPermission标签
<shiro:lacksPermission name="org:create">
用户[<shiro:principal/>]没有权限org:create
</shiro:lacksPermission>
如果当前Subject没有权限将显示body体内容。

12 标签

hasRole标签
<shiro:hasRole name="admin">
用户[<shiro:principal/>]拥有角色admin
</shiro:hasRole>
如果当前Subject有角色将显示body体内容。

hasAnyRoles标签
<shiro:hasAnyRoles name="admin,user">
用户[<shiro:principal/>]拥有角色admin 或user
</shiro:hasAnyRoles>
如果当前Subject有任意一个角色（或的关系）将显示body体内容。

lacksRole标签
<shiro:lacksRole name="abc">
用户[<shiro:principal/>]没有角色abc
</shiro:lacksRole>
如果当前Subject没有角色将显示body体内容。

hasPermission标签
<shiro:hasPermission name="user:create">
用户[<shiro:principal/>]拥有权限user:create
</shiro:hasPermission>
如果当前Subject有权限将显示body体内容。

13 权限注解

Shiro 权限注解
Shiro 提供了相应的注解用于权限控制，如果使用这些注解就需要使用AOP 的功能来进行判断，如Spring AOP；Shiro 提供了Spring AOP 集成用于权限注解的解析和验证。

@RequiresAuthentication
表示当前Subject已经通过login 进行了身份验证；即Subject. isAuthenticated()返回true。

@RequiresUser
表示当前Subject已经身份验证或者通过记住我登录的。

@RequiresGuest
表示当前Subject没有身份验证或通过记住我登录过，即是游客身份。

@RequiresRoles(value={“admin”, “user”}, logical= Logical.AND)
表示当前Subject需要角色admin 和user。

@RequiresPermissions (value={“user:a”, “user:b”}, logical= Logical.OR)
表示当前Subject需要权限user:a或user:b。

14 数据库中初始化资源及权限

如果配置项目较多，则可以将其放入数据库中，通过spring的实例工厂模式进行实现。

```
<property name="filterChainDefinitions">
                <value>
                        /login.jsp = anon
                        /shiro/login = anon
                        /shiro/logout = logout
                        /user.jsp = roles[user]
                        /admin.jsp = roles[admin]
                        /** = authc
                </value>
</property>
```

15 会话管理

会话管理

Shiro 提供了完整的企业级会话管理功能，不依赖于底层容器（如web容器tomcat），不管JavaSE 还是JavaEE环境都可以使用，提供了会话管理、会话事件监听、会话存储/持久化、容器无关的集群、失效/过期支持、对Web 的透明支持、SSO 单点登录的支持等特性。即直接使用Shiro 的会话管理可以直接替换如Web 容器的会话管理。

会话
所谓会话，即用户访问应用时保持的连接关系，在多次交互中应用能够识别出当前访问的用户是谁，且可以在多次交互中保存一些数据。如访问一些网站时登录成功后，网站可以记住用户，且在退出之前都可以识别当前用户是谁。

Shiro 提供的会话可以用于JavaSE/JavaEE 环境，不依赖于任何底层容器，可以独立使用，是完整的会话模块。

15 会话管理

api

登录成功后使用Subject.getSession()即可获取会话；其等价于Subject.getSession(true)，即如果当前没有创建Session 对象会创建一个；另外Subject.getSession(false)，如果当前没有创建Session 则返回null（不过默认情况下如果启用会话存储功能的话在创建Subject 时会主动创建一个Session）。

session.getId();
获取当前会话的唯一标识。

session.getHost();
获取当前Subject的主机地址，该地址是通过HostAuthenticationToken.getHost()提供的。

session.getTimeout();
session.setTimeout(毫秒);
获取/设置当前Session的过期时间；如果不设置默认是会话管理器的全局过期时间。

session.getStartTimestamp();
session.getLastAccessTime();
获取会话的启动时间及最后访问时间；如果是JavaSE应用需要自己定期调用session.touch()去更新最后访问时间；如果是Web应用，每次进入ShiroFilter都会自动调用session.touch()来更新最后访问时间。

15 会话管理

session.touch();
session.stop();
更新会话最后访问时间及销毁会话；当Subject.logout()时会自动调用stop 方法来销毁会话。
如果在web中，调用javax.servlet.http.HttpSession. invalidate()也会自动调用Shiro Session.stop
方法进行销毁Shiro 的会话。

session.setAttribute("key", "123");
Assert.assertEquals("123", session.getAttribute("key"));
session.removeAttribute("key");
设置/获取/删除会话属性；在整个会话范围内都可以对这些属性进行操作。

15 会话管理

会话管理器

会话管理器管理着应用中所有Subject的会话的创建、维护、删除、失效、验证等工作。是Shiro 的核心组件，顶层组件SecurityManager 直接继承了SessionManager，且提供了SessionsSecurityManager 实现直接把会话管理委托给相应的SessionManager ，DefaultSecurityManager 及DefaultWebSecurityManager 默认SecurityManager 都继承了SessionsSecurityManager。

SecurityManager提供了如下接口：

Session start(SessionContext context); //启动会话

Session getSession(SessionKey key) throws SessionException; //根据会话Key获取会话

15 会话管理

会话监听器

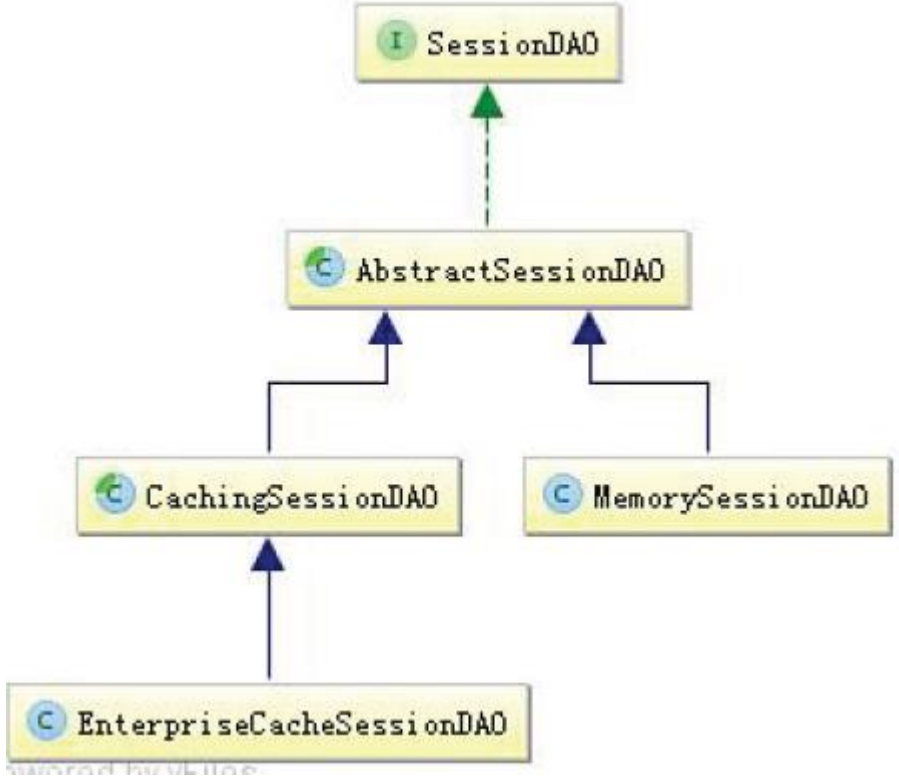
会话监听器用于监听会话创建、过期及停止事件

```
public class MySessionListener1 implements SessionListener {
@Override
public void onStart(Session session) {//会话创建时触发
System.out.println("会话创建：" + session.getId());
}
@Override
public void onExpiration(Session session) {//会话过期时触发
System.out.println("会话过期：" + session.getId());
}
@Override
public void onStop(Session session) {//退出/会话过期时触发
System.out.println("会话停止：" + session.getId());
}
}
```

15 会话管理

sessiondao

Shiro 内嵌了如下SessionDAO 实现：
AbstractSessionDAO提供了SessionDAO的基础实现，如生成会话ID等；CachingSessionDAO 提供了对开发者透明的会话缓存的功能，只需要设置相应的CacheManager 即可；
MemorySessionDAO 直接在内存中进行会话维护；而EnterpriseCacheSessionDAO 提供了缓存功能的会话维护，默认情况下使用MapCache 实现，内部使用ConcurrentHashMap 保存缓存的会话。



15 会话管理

会话存储/持久化

Shiro 提供SessionDAO 用于会话的CRUD，即DAO（Data Access Object）模式实现：

```
//如DefaultSessionManager 在创建完session 后会调用该方法；如保存到关系数据库/文件系统/NoSQL 数据库；即可以实现会话的持久化；返回会话ID；主要此处返回的ID.equals(session.getId())；
Serializable create(Session session);
//根据会话ID获取会话
Session readSession(Serializable sessionId) throws UnknownSessionException;
//更新会话；如更新会话最后访问时间/停止会话/设置超时时间/设置移除属性等会调用
void update(Session session) throws UnknownSessionException;
//删除会话；当会话过期/会话停止（如用户退出时）会调用
void delete(Session session);
//获取当前所有活跃用户，如果用户量多此方法影响性能
```

15 会话管理

会话验证

Shiro 提供了会话验证调度器，用于定期的验证会话是否已过期，如果过期将停止会话；出于性能考虑，一般情况下都是获取会话时来验证会话是否过期并停止会话的；但是如在web环境中，如果用户不主动退出是不知道会话是否过期的，因此需要定期的检测会话是否过期，Shiro 提供了会话验证调度器SessionValidationScheduler来做这件事情。

16 缓存

缓存机制

Shiro 提供了类似于Spring的Cache抽象，即Shiro 本身不实现Cache，但是对Cache 进行了又抽象，方便更换不同的底层Cache实现。

Shiro提供的Cache接口：
Shiro提供的CacheManager接口：
Shiro还提供了CacheManagerAware用于注入CacheManager：

```
public interface Cache<K, V> {
//根据Key获取缓存中的值
public V get(K key) throws CacheException;
//往缓存中放入key-value，返回缓存中之前的值
public V put(K key, V value) throws CacheException;
//移除缓存中key对应的值，返回该值
public V remove(K key) throws CacheException;
//清空整个缓存
public void clear() throws CacheException;
public int size(); //返回缓存大小
public Set<K> keys(); //获取缓存中所有的key
//获取缓存中所有的value
public Collection<V> values();
}
```

16 缓存

接口

Shiro提供的CacheManager接口：

```
public interface CacheManager {
//根据缓存名字获取一个Cache
public <K, V> Cache<K, V> getCache(String name) throws CacheException;
}
```

Shiro还提供了CacheManagerAware用于注入CacheManager：

```
public interface CacheManagerAware {
//注入CacheManager
void setCacheManager(CacheManager cacheManager);
}
```

Shiro 内部相应的组件（DefaultSecurityManager）会自动检测相应的对象（如Realm）是否实现了CacheManagerAware并自动注入相应的CacheManager。

16 缓存

Realm 缓存
Shiro 提供了CachingRealm，其实现了CacheManagerAware接口，提供了缓存的一些基础实现；另外AuthenticatingRealm及AuthorizingRealm分别提供了对AuthenticationInfo 和AuthorizationInfo信息的缓存。

Session 缓存
如 securityManager实现了SessionsSecurityManager，其会自动判断SessionManager是否实现了CacheManagerAware接口，如果实现了会把CacheManager设置给它。然后sessionManager会判断相应的sessionDAO（如继承自CachingSessionDAO）是否实现了CacheManagerAware，如果实现了会把CacheManager设置给它；
缓存的SessionDAO；其会先查缓存，如果找不到才查数据库。

17 记住我

RememberMe

Shiro 提供了记住我（RememberMe）的功能，比如访问如淘宝等一些网站时，关闭了浏览器下次再打开时还是能记住你是谁，下次访问时无需再登录即可访问，基本流程如下：

1、首先在登录页面选中RememberMe 然后登录成功；如果是浏览器登录，一般会把RememberMe的Cookie 写到客户端并保存下来；
2、关闭浏览器再重新打开；会发现浏览器还是记住你的；
3、访问一般的网页服务器端还是知道你是谁，且能正常访问；
4、但是比如我们访问淘宝时，如果要查看我的订单或进行支付时，此时还是需要再进行身份认证的，以确保当前用户还是你。

17 记住我

判断方法

subject.isAuthenticated()表示用户进行了身份验证登录的，即使有Subject.login进行了登录；
subject.isRemembered()：表示用户是通过记住我登录的，此时可能并不是真正的你（如你的朋友使用你的电脑，或者你的cookie 被窃取）在访问的；且两者二选一，即subject.isAuthenticated()==true，则subject.isRemembered()==false；反之一样。

建议，一般这样使用：
访问一般网页，如个人在主页之类的，我们使用user 拦截器即可，user 拦截器只要用户登录(isRemembered()==true or isAuthenticated()==true)过即可访问成功；
访问特殊网页，如我的订单，提交订单页面，我们使用authc拦截器即可，authc 拦截器会判断用户是否是通过Subject.login（isAuthenticated()==true）登录的，如果是才放行，否则会跳转到登录页面叫你重新登录。
因此RememberMe使用过程中，需要配合相应的拦截器来实现相应的功能，用错了拦截器可能就不能满足你的需求了。

感谢您的参与！

主讲：安燚