Java私塾《深入浅出学Shiro》

——系列精品教程

整体课程概览

- 系统学习Shiro 的核心开发知识，循序渐进，系统理解和掌握
- 第一章：权限管理基础
  包括：理解常见权限系统的两个大部分（分配权限和验证权限）、理解安全实体和权限的含义、理解权限的继承性、理解权限的最近匹配原则等
- 第二章：Shiro入门
  包括：是什么、能干什么、有什么、高层概览架构、完整架构、HelloWorld
- 第三章：Shiro的配置
  包括：程序配置、ini配置的方式（包括各个部分的配置）、权限字符串方式等
- 第四章：Shiro的身份认证
  包括：认证示例、理解Remembered和Authenticated、理解认证的流程、多个Realm等
- 第五章：Shiro的授权
  包括：授权的要素和粒度、编程授权、注解授权、理解授权的流程、理解ModularRealmAuthorizer等

整体课程概览

- 第六章：Shiro的Realms
  包括：理解Realms的认证实现、Shiro默认的Realms的认证实现、使用默认的JdbcRealm、自定义Realm、在spring的配置文件中使用等
- 第七章：Shiro的Session管理
  包括：概述和配置使用、理解SessionDAO、Web应用中的Session、自定义SessionDAO
- 第八章：Shiro和Spring的集成
  包括：基本应用的配置、Web应用的配置、和Struts2+Spring3的集成、和SpringMVC+Spring3的集成
- 第九章：Shiro的Web
  包括：URLS配置、Shiro的Tag Libary的使用等
- 第十章：Shiro的Cache
  包括：Shiro的Caching API、Caching配置使用、包装使用其他的Cache框架、缓存数据同步更新的解决方案

第一章：权限管理设计基础

权限管理设计基础-1

- 为了让大家更好的理解后面讲述的知识，先介绍一点权限系统的基础知识：

  几乎所有的权限系统都分成两个部分，一个是分配权限部分，一个是验证权限部分，为了理解它们，首先解释两个基本的名词：安全实体和权限。

  安全实体：就是被权限系统保护的对象，比如工资数据。
  权限：就是需要被校验的权限对象，比如查看、修改等。

- 安全实体和权限通常要一起描述才有意义：

  比如有这么个描述：“现在要检测登录人员对工资数据是否有查看的权限”，“工资数据”这个安全实体和“查看”这个权限一定要一起描述。如果只出现安全实体描述，那就变成这样：“现在要检测登录人员对工资数据”，对工资数据干什么呀，没有后半截，一看就知道不完整；当然只有权限描述也不行，那就变成：“现在要检测登录人员是否有查看的权限”，对谁的查看权限啊，也不完整。所以安全实体和权限通常要一起描述。

权限管理设计基础-2

- 了解了上面两个名词，来看看什么是分配权限和验证权限：

所谓分配权限是指：把对某些安全实体的某些权限分配给某些人员的过程。

所谓验证权限是指：判断某个人员或程序对某个安全实体是否拥有某个或某些权限的过程。

也就是说，**分配权限过程即是权限的分配过程，而验证权限过程则是权限的匹配过程**。在目前应用系统的开发中，多数是利用数据库来存放授权过程产生的数据，也就是说：分配权限是向数据库里面添加数据、或是维护数据的过程，而验证权限过程就变成了从数据库中获取相应数据进行匹配的过程了。

权限管理设计基础-3

- 再来介绍一下验证权限涉及到的两个名词：

- **权限的继承性**指的是：如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会继承包含它的安全实体的相应权限。

  比如：某个大楼和楼内的房间都是安全实体，很明显大楼这个安全实体会包含楼内的房间这些安全实体，可以认为大楼是楼内房间的父级实体。现在来考虑一个具体的权限——进入某个房间的权限。如果这个房间没有门，也就是谁都可以进入，相当于这个房间对应的安全实体，没有进入房间的权限限制，那么是不是说所有的人都可以进入这个房间呢？当然不是，某人能进入这个房间的前提是：这个人要有权限进入这个大楼，也就是说，这个时候房间这个安全实体，它本身没有进入权限的限制，但是它会继承父级安全实体的进入权限。

权限管理设计基础-4

- **权限的最近匹配原则**指的是：如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会向上寻找并匹配相应权限限制，直到找到一个离这个安全实体最近的拥有相应权限限制的安全实体为止。如果把整个层次结构都寻找完了都没有匹配到相应权限限制的话，那就说明所有人对这个安全实体都拥有这个相应的权限限制。

继续上面权限继承性的例子，如果现在这个大楼是坐落在某个机关大院内，这就演变成了，要进入某个房间，首先要有进入大楼的权限，要进入大楼又需要有能进入机关大院的权限。

所谓最近匹配原则就是，如果某个房间没有门，也就意味着这个房间没有进入的权限限制，那么它就会向上继续寻找并匹配，看看大楼有没有进入的权限限制，如果有就使用这个权限限制，终止寻找；如果没有，继续向上寻找，直到找到一个匹配的为止。如果最后大院也没有进入的权限限制，那就变成所有人都可以进入到这个房间里面来了。

第二章：Shiro入门

Shiro是什么和能干什么

- Shiro是什么

  Apache Shiro是一个强大易用的Java安全框架，提供了认证、授权、加密和会话管理等功能

- Shiro能做什么

  认证：验证用户来核实他们的身份

  授权：对用户执行访问控制，如：

  判断用户是否被分配了一个确定的安全角色

  判断用户是否被允许做某事

  会话管理：在任何环境下使用Session API，即使没有Web 或EJB 容器。

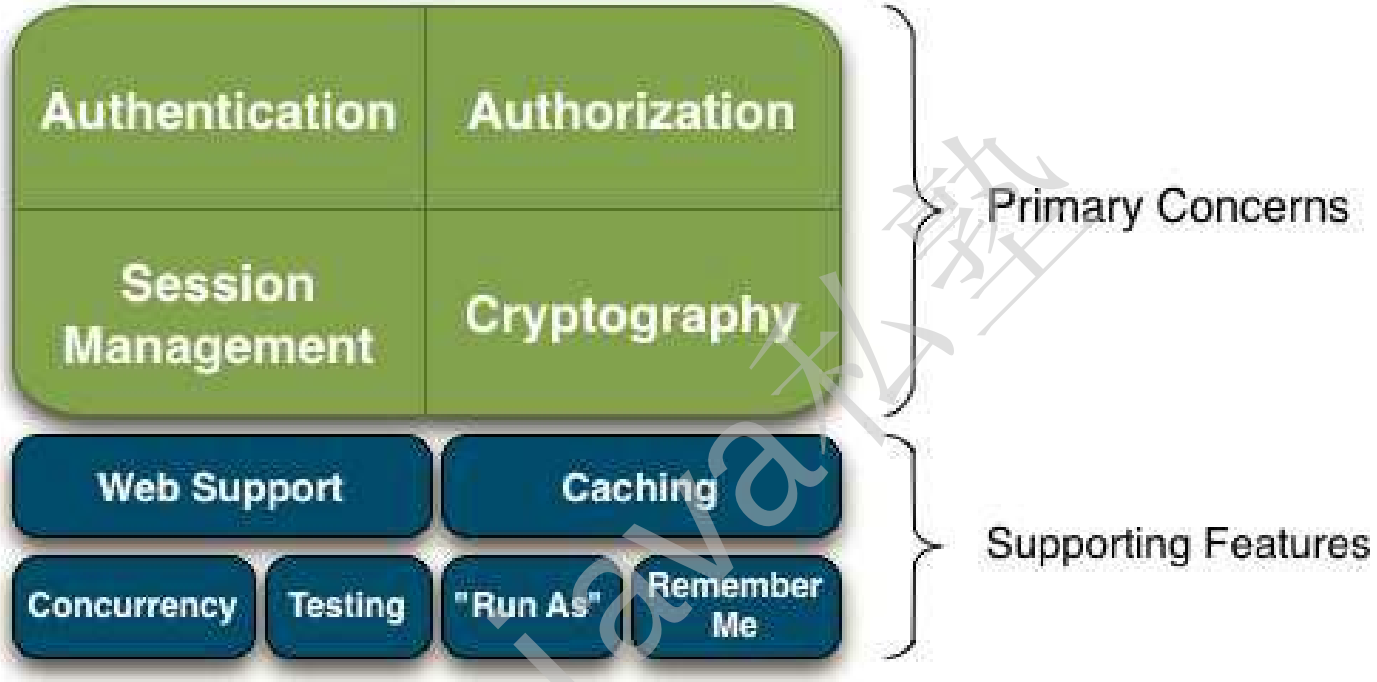
  加密：以更简洁易用的方式使用加密的功能，保护或隐藏数据防止被偷窥

  Realms：聚集一个或多个用户安全数据的数据源，并作为一个单一的复合用户“视图”。

  启用**单点登录**（SSO）功能。

  为没有关联到登录的用户启用"Remember Me"服务

Shiro有什么-1



- Shiro的四大部分——身份验证，授权，会话管理和加密

Authentication：有时也简称为“登录”，这是证明用户是他们所说的他们是谁的行为。
Authorization：访问控制的过程，也就是绝对“谁”去访问“什么”。
Session Management：管理用户特定的会话，即使在非Web 或EJB 应用程序。
Cryptography：通过使用加密算法保持数据安全同时易于使用

Shiro有什么-2

- 除了以上功能，shiro还提供很多扩展

  Web Support：主要针对web应用提供一些常用功能。

  Caching：缓存可以使应用程序运行更有效率。

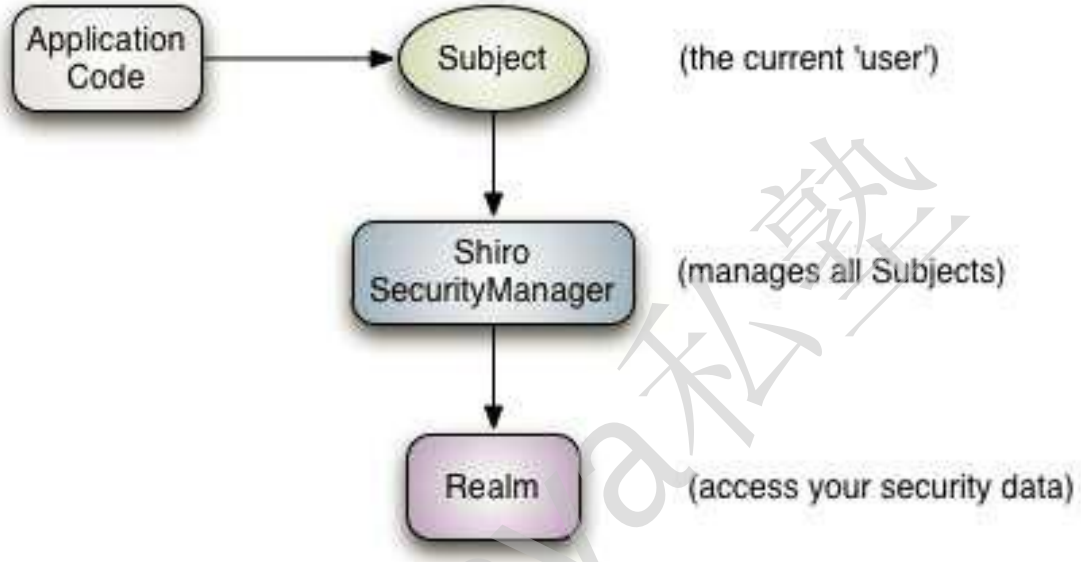
  Concurrency：多线程相关功能。

  Testing：帮助我们进行测试相关功能

  "Run As"：一个允许用户假设为另一个用户身份（如果允许）的功能，有时候在管理脚本很有用。

  “Remember Me”：记住用户身份，提供类似购物车功能。

Shiro的高层概览架构-1



- Subject

Subject 实质上是一个当前执行用户的特定的安全“视图”。Subject 可以是一个人，也可以代表第三方服务，或其他类似的任何东西——基本上是当前正与软件进行交互的任何东西。

所有Subject 实例都被绑定到（且这是必须的）一个SecurityManager 上。当你与一个Subject 交互时，那些交互作用转化为与SecurityManager 交互的特定subject 的交互作用。

Shiro的高层概览架构-2

- SecurityManager

  SecurityManager 是Shiro 架构的心脏，用来协调内部的安全组件共同构成一个对象图，管理内部组件实例，并通过它来提供安全管理的各种服务。

  实际开发中，程序人员主要与Subject交互，但是要认识到，当你正与一个Subject进行交互时，实质上是幕后的SecurityManager 处理所有繁重的Subject 安全操作。

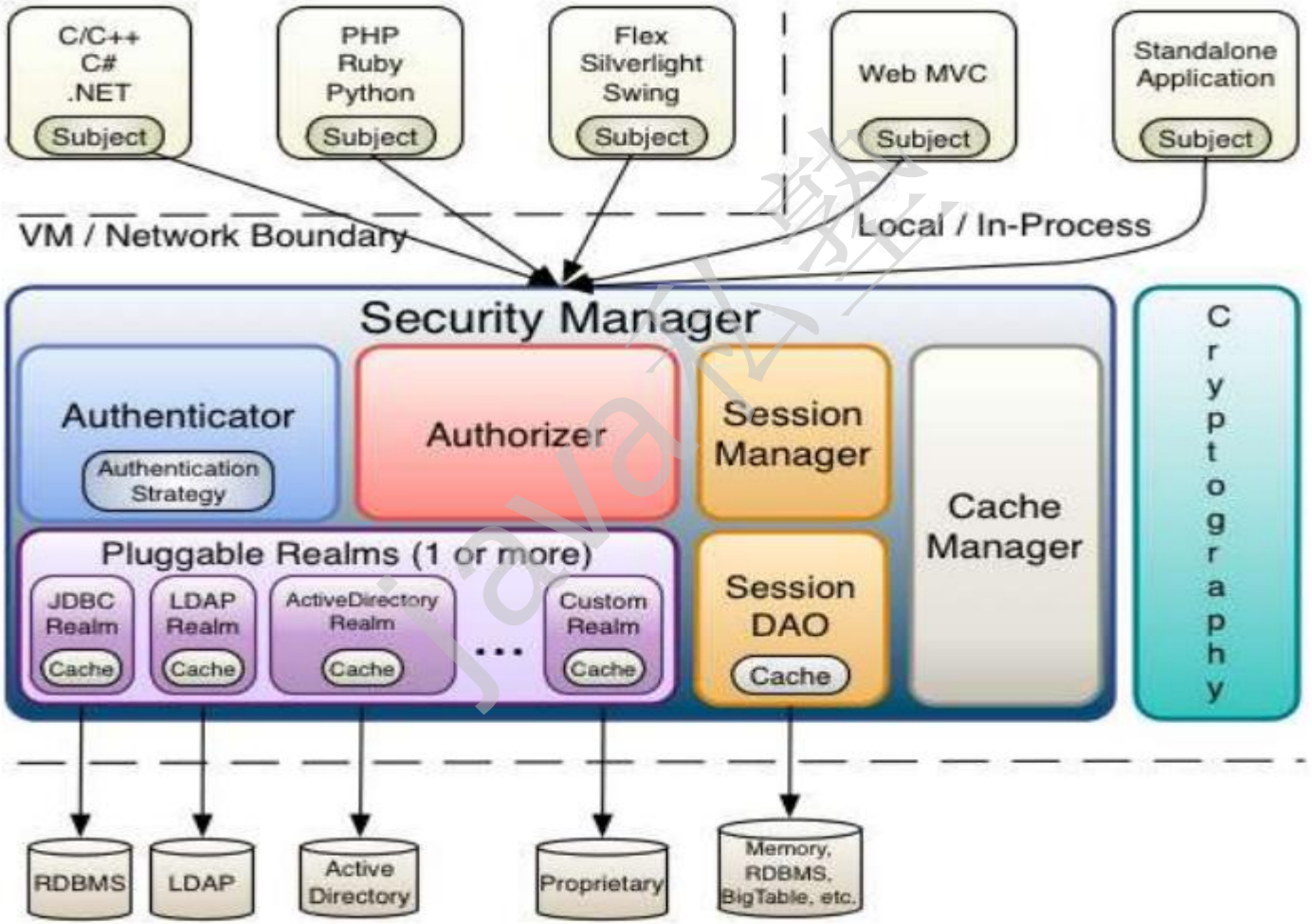
- Realms

  Realms 担当Shiro 和你的应用程序的安全数据之间的“桥梁”或“连接器”。当它实际上与安全相关的数据如用来执行身份验证（登录）及授权（访问控制）的用户帐户交互时，Shiro 从一个或多个为应用程序配置的Realm 中寻找许多这样的东西。

  Realm 本质上是一个特定安全的DAO：它封装了数据源的连接详细信息，使Shiro 所需的相关的数据可用。当配置Shiro 时，你必须指定至少一个Realm 用来进行身份验证和/或授权。SecurityManager可能配置多个Realms，但至少有一个是必须的。

  Shiro 提供了立即可用的Realms 来连接一些安全数据源（即目录），如LDAP，关系数据库（JDBC），文本配置源，像INI 及属性文件，以及更多。你可以插入你自己的Realm实现来代表自定义的数据源，如果默认地Realm 不符合你的需求。

Shiro的完整架构-1



Shiro的完整架构-2

- Authenticator ：是一个执行对用户的身份验证（登录）的组件。Authenticator 知道如何与一个或多个Realm 协调来存储相关的用户/帐户信息。从这些 Realm 中获得的数据被用来验证用户的身份来保证用户确实是他们所说的他们是谁。
- 如果存在多个realm，则接口AuthenticationStrategy会确定什么样算是登录成功（例如，如果一个Realm成功，而其他的均失败，是否登录成功？）。
- Authorizer ：授权实质上就是访问控制 - 控制用户能够访问应用中的哪些内容，比如资源、Web页面等等。
- SessionManager ：提供可在任何应用或架构层一致地使用Session API。
- SessionDAO：代表SessionManager 执行Session 持久化（CRUD）操作。
- CacheManager ：对Shiro的其他组件提供缓存支持。
- Cryptography：Shiro的api大幅度简化java api中繁琐的密码加密
- Realms：Realms 在Shiro 和你的应用程序的安全数据之间担当“桥梁”或“连接器”。简单的说，shiro通过Realms来获取相应的安全数据

Shiro的HelloWorld-1

- 构建开发和运行环境：

1：在eclipse里面创建一个基本的java工程

2：加入所需要的jar包

- 在src下创建TestShiro.ini文件，内容如下：

```
[users]
javass = cc,role1

[roles]
role1 = p1,p2
```

Shiro的HelloWorld-2

n 创建cn.javass.hello包，并创建HelloWorld类

```
public class HelloWorld {
    public static void main(String[] args) {
        Factory<org.apache.shiro.mgt.SecurityManager> f = new
    IniSecurityManagerFactory("classpath:TestShiro.ini");
        org.apache.shiro.mgt.SecurityManager s = f.getInstance();
        SecurityUtils.setSecurityManager(s);

        UsernamePasswordToken token = new UsernamePasswordToken("javass","cc");
        token.setRememberMe(true);

        Subject currentUser = SecurityUtils.getSubject();
        currentUser.login(token);

        boolean flag = currentUser.isPermitted("p1");
        System.out.println("flag==" + flag);
    }
}
```

第三章：Shiro的配置

Shiro配置基础知识

- Shiro 被设计成能够在任何环境下工作，从最简单的命令行应用程序到最大的的企业群集应用。由于环境的多样性，使得许多配置机制适用于它的配置。

- 兼容JavaBean格式的

  Shiro的SecurityManager 实现及所支持的组件都是兼容JavaBean 的。这使得Shiro几乎能使用任何配置格式，如regular Java，XML(Spring, JBoss, Guice,等等)，YAML，JSON，Groovy Builder markup，以及更多的配置。

程序配置的方式

- 创建SecurityManger示例

```
Realm r = new IniRealm();
DefaultSecurityManager s = new DefaultSecurityManager(r);
SecurityUtils.setSecurityManager(s);
```

SecurityUtils.setSecurityManager 方法调用在一个VM 静态单例中实例化 SecurityManager 实例

- SecurityManger的对象图

SecurityManager 实现实质上是一个特定安全的嵌套组件中的模块化对象图。因为它们也是兼容JavaBean 的，你可以调用任何嵌套组件的getter 和 setter 方法来配置SecurityManager以及它的内部对象图。

例如，如果你想配置SecurityManager 实例来使用自定义的SessionDAO 来定制 Session Management，你可以通过嵌套的SessionManager 的setSessionDAO 方法直接设置SessionDAO：

```
SessionDAO sessionDAO = new MySessionDAO();
((DefaultSessionManager)s.getSessionManager()).setSessionDAO(sessionDAO);
```

ini配置的方式-1

- Ini配置

  INI基本上是一个文本配置，包含了由唯一命名的项来 组织的键/值对。键只是每个项 唯一，而不是在整个配置中（与JDK 属性不同）

- Ini配置示例：

```
# =======================
# Shiro INI configuration
# =======================

[main]
# Objects and their properties are defined here,
# Such as the securityManager, Realms and anything
# else needed to build the SecurityManager

[users]
# The 'users' section is for simple deployments
# when you only need a small number of statically-defined
# set of User accounts.

[roles]
# The 'roles' section is for simple deployments
# when you only need a small number of statically-defined
# roles.

[urls]
# The 'urls' section is used for url-based security
# in web applications.  We'll discuss this section in the
# Web documentation
```

ini配置的方式-2

- **[main]**

  配置应用程序的SecurityManager 实例及任何它的依赖组件（如Realms）的地方，示例如：

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher

myRealm = com.company.security.shiro.DatabaseRealm
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
myRealm.password = secret
myRealm.credentialsMatcher = $sha256Matcher

securityManager.sessionManager.globalSessionTimeout = 1800000
```

- 上述示例包括了：

1：定义对象

2：设置对象属性，如果是原始类型的值，就直接设置；如果是引用类型的值，就是用$+名称的方式来设置

3：可以使用遍历对象图的方式来设置数据

ini配置的方式-3

- 对于Byte Array的值

  因为原始的字节数组本身不能使用文本格式，所以我们必须使用文本编码的字节数组。能够指定的值是一个Base64编码的字符串（默认），后一个16 进制编码的字符串。默认是Base64 是因为Base64 编码只需较少的文本来表示值——它拥有一个较大的编码表，意味着你的token 都是较短的。如：

```
# The 'cipherKey' attribute is a byte array.    By default, text values
# for all byte array properties are expected to be Base64 encoded:

securityManager.rememberMeManager.cipherKey = kPH+bIxk5D2deZiIxcaaaA==
...
```

如果你喜欢使用16 进制编码，你必须在字符串token 前加上0x（"zero" "x"）前缀

```
securityManager.rememberMeManager.cipherKey = 0x3707344A4093822299F31D008
```

ini配置的方式-4

- 对于Collection的值

  对于Set 和list 而言，只需指定一组由逗号分隔的值或对象的引用。如：

```
sessionListener1 = com.company.my.SessionListenerImplementation
...
sessionListener2 = com.company.my.other.SessionListenerImplementation
...
securityManager.sessionManager.sessionListeners = $sessionListener1, $sessionListener2
```

对于Map，你指定一系列由逗号分隔的键-值对，每个键-值对通过冒号“:”被限定：

```
object1 = com.company.some.Class
object2 = com.company.another.Class
...
anObject = some.class.with.a.Map.property

anObject.mapProperty = key1:$object1, key2:$object2
```

ini配置的方式-5

- 注意

1：顺序问题：

INI 格式和约定都非常便捷且易于理解，但它没有其他基于text/XML 的配置机制强大。在使用上面的机制时最重要的问题是理解顺序问题，请记住：
每个对象的实例化以及每个值的分配都是按照它们在[main] section中出现的顺序来执行的。这些配置行最终转化成一个JavaBean 的getter/setter 方法调用，因此，这些方法以同样的顺序被调用！

2：实例覆盖的问题

任何对象能够被配置中后来新定义的实例覆盖。如下

```
...
myRealm = com.company.security.MyRealm
...
myRealm = com.company.security.DatabaseRealm
...
```

这将导致myRealm 成为一个com.company.security.DatebaseRealm 实例，且之前的实例将永不会被使用（同时被垃圾回收）

ini配置的方式-6

- 注意

3：缺省的SecurityManage：

你可能已经注意到在上面的示例中，SecurityManager 实例的类并没有定义，我们仅在右边设定一个嵌套属性：

```
myRealm = ...
securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

这是因为securityManager 实例是一个特殊的实例——它已经为你实例化并准备好使用，所以你不需要知道用来实例化的具体SecurityManager 实例类。

当然，如果你确实想指定你自己的实例，你可以只定义你自己的实现，如下所示：

```
...
securityManager = com.company.security.shiro.MyCustomSecurityManager
...
```

ini配置的方式-7

- **[users]**

  [users] section 允许你定义一组静态的用户帐户。这在大部分拥有少数用户帐户或用户帐户不需要在运行时被动态地创建的环境下是很有用的，比如：

```
[users]
admin = secret
lonestarr = vespa, goodguy, schwartz
darkhelmet = ludicrousspeed, badguy, schwartz
```

- 每行的格式

  username = password, roleName1, roleName2, …, roleNameN

- 自动初始化realm

  仅定义非空的[users]或[roles] section 将会自动地触发org.apache.shiro.realm.text.IniRealm 实例的创建，并使它在[main] section 中可用且名为iniRealm。

ini配置的方式-8

- 密码加密

如果你不想[users] section 中密码是纯文本的，你可以使用你喜爱的散列算法（MD5，Sha1，Sha256，等等）来进行加密，并使用生产的字符串作为密码值。默认情况下，密码字符串是16 进制编码，但可以使用Base64 编码代替16进制编码来配置。

一旦你指定了文本密码散列值，你得告诉Shiro 这些都是加密的。你可以通过配置在[main] section 中隐式地创建iniRealm 来使用合适的CredentialsMatcher 实现来对应到你所指定的哈希算法。比如在ini文件中：

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
iniRealm.credentialsMatcher = $sha256Matcher
[users]
javass
    =355b1bbfc96725cdce8f4a2708fda310a80e6d13315aec4e5eed2a75fe8032ce,
    role1
```

ini配置的方式-9

- 获取密码的 hex 加密字符串

```
String ss = new Sha256Hash("cc").toHex();
```

- 你也可以像任何其他对象一样在CredentialsMatcher 上配置任何属性，以反映你哈希策略，反正是JavaBean风格的
- 比如：指定Base64编码

```
[main]
sha256Matcher =
   org.apache.shiro.authc.credential.Sha256CredentialsMatcher
sha256Matcher.storedCredentialsHexEncoded=false
iniRealm.credentialsMatcher = $sha256Matcher

[users]
javass =NVsbv8lnJc30j0onCP2jEKgObRMxWuxOXu0qdf6AMs4=,role1
```

- 当然获取密码的 Base64 加密字符串得用下面的语句：

```
String ss = new Sha256Hash("cc").toBase64();
```

ini配置的方式-10

- **[roles]**

  [roles] section 允许你把定义在[users] section 中的角色与权限关联起来。另外，这在大部分拥有少数用户帐户或用户帐户不需要在运行时被动态地创建的环境下是很有用的，比如：

```
[roles]
# 'admin' role has all permissions, indicated by the wildcard '*'
admin = *
# The 'schwartz' role can do anything (*) with any lightsaber:
schwartz = lightsaber:*
# The 'goodguy' role is allowed to 'drive' (action) the winnebago (type) with
# license plate 'eagle5' (instance specific id)
goodguy = winnebago:drive:eagle5
```

- 每行的格式

rolename = permissionDefinition1, permissionDefinition2, …

permissionDefinition 是一个任意的字符串，但大多数人将会使用符合 org.apache.shiro.authz.permission.WildcardPermission 格式的字符串。

ini配置的方式-11

- **注意事项**

1：如果一个独立的permissionDefinition 需要被内部逗号分隔（例如，printer:5thFloor:print,info），你需要用户双引号环绕该定义，以避免错误解析。

2：如果角色不想关联权限，那你不需要在[roles] section 中间把他们列出来。只需定义在[user]section 中定义角色名就足以创建尚不存在的角色。

- **[urls]**

这个放到学web的时候再讲

Shiro的Permissions-1

- 概述

  Shiro 将权限定义为一个规定了明确行为或活动的声明。权限是在安全策略中最低级别的构造，它们明确地定义了应用程序只能做“什么”，但它们从不描述“谁”能够执行这些动作。

  一些权限的例子：

  打开文件

  浏览‘/user/list’页面

  打印文档

  删除‘jsmith’用户

- 引入Wildcard Permissions

  将这些权限语句解释为自然语言字符串，并判断用户是否被允许执行该行为在计算上是非常困难的。

  为了使用易于处理且仍然可读的权限语句，Shiro 提供了强大而直观的语法，称之为WildcardPermission

Shiro的Permissions-2

- 基础语法之 简单的字符串

  就是用简单的字符串来表示一个权限，如：queryPrinter

- 基础语法之 多层次管理

1：比如：printer:print

printer:manage

在这个例子中的冒号是一个特殊字符，它用来分隔权限字符串的下一部件。

其中第一部分是权限被操作的领域（打印机），第二部分是被执行的操作。

2：多个值

每个部件能够保护多个值。因此，除了授予用户“printer:print”和“printer:query”权限外，你可以简单地授予他们一个

printer:print, query

3：还可以用*号代替所有的值，如：printer:* ， 当然你也可以写：*:view，表示某个用户在所有的领域都有view的权限

Shiro的Permissions-3

- 基础语法之 实例级访问控制

1：这种情况通常会使用三个部件——第一个是域，第二个是操作，第三个是被付诸实施的实例。如：printer:query:lp7200

也可以使用通配符来定义，如：

```
printer:print:*
printer:*:*
printer:*:lp7200
printer:query, print:lp7200
```

2：部分省略：缺少的部件意味着用户可以访问所有与之匹配的值，比如：

printer:print 等价于 printer:print:*

printer 等价于 printer:*:*

但是请记住：只能从字符串的结尾处省略部件，也就是说

printer:lp7200 并不等价于 printer:*:lp7200

第四章：Shiro的身份认证（Authentication）

Authentication概述

- 概述

Authentication 是指身份验证的过程——即证明一个用户实际上是不是他们所说的他们是谁。也就是说通过提交用户的身份和凭证给Shiro，以判断它们是否和应用程序预期的相匹配。

- 基本概念

1：Principals(身份)：是Subject 的‘identifying attributes(标识属性)’。比如我们登录提交的用户名。

2：Credentials(凭证)：通常是只被Subject 知道的秘密值，它用来作为一种起支持作用的证据，此证据事实上包含着所谓的身份证明。比如我们登录提供的密码

- 认证的基本步骤

1. 收集Subjects 提交的Principals(身份)和Credentials(凭证)；
2. 提交Principals(身份)和Credentials(凭证)进行身份验证；
3. 如果提交成功，则允许访问，否则重新进行身份验证或者阻止访问。

认证样例-1

- 使用用户名/密码的样例

```
UsernamePasswordToken token = new UsernamePasswordToken(username,
    password);
token.setRememberMe(true);
```

样例使用UsernamePasswordToken 来支持最常见的用户名/密码的身份验证方法。这是Shiro的org.apache.shiro.authc.AuthenticationToken 的接口，是Shiro 代表提交的Principals(身份)和Credentials(凭证)的身份验证系统所使用的基本接口的一个实现。

- 提交用户名/密码进行认证

```
Subject currentUser = SecurityUtils.getSubject();
currentUser.login(token);
```

- 处理认证成功和失败

如果认证成功，会没有返回，也没有例外，通过。

如果认证失败，会抛出例外，你可以在程序中捕获并处理，如下示例：

认证样例-2

```
try {
    currentUser.login(token);
} catch ( UnknownAccountException uae ) { …
} catch ( IncorrectCredentialsException ice ) { …
} catch (LockedAccountException lae ) { …
} catch (ExcessiveAttemptsException eae ) { …
} … catch your own …
```

- logout(注销)

  currentUser.logout();

  当你调用logout，任何现有的Session 都将会失效，而且任何身份都将会失去关联（例如，在Web 应用程序中，RememberMe cookie 也将被删除）。在Subject 注销后，该Subject的实例被再次认为是匿名的，当然，除了Web 应用程序。

  注意：由于在Web 应用程序记住身份往往是依靠Cookies，然而Cookies 只能在Response 被committed 之前被删除，所以强烈建议在调用subject.logout()后立即将终端用户重定向到一个新的视图或页面。

  这样能够保证任何与安全相关的Cookies 都能像预期的一样被删除。这是HTTP cookies 的功能限制，而不是Shiro的。

Remembered和Authenticated

- Remembered(记住我)

  一个记住我的Subject 不是匿名的，是有一个已知的身份ID（也就是subject.getPrincipals()是非空的）。但是这个被记住的身份ID 是在之前的session 中被认证的。如果subject.isRemembered()返回true，则Subject 被认为是被记住的。

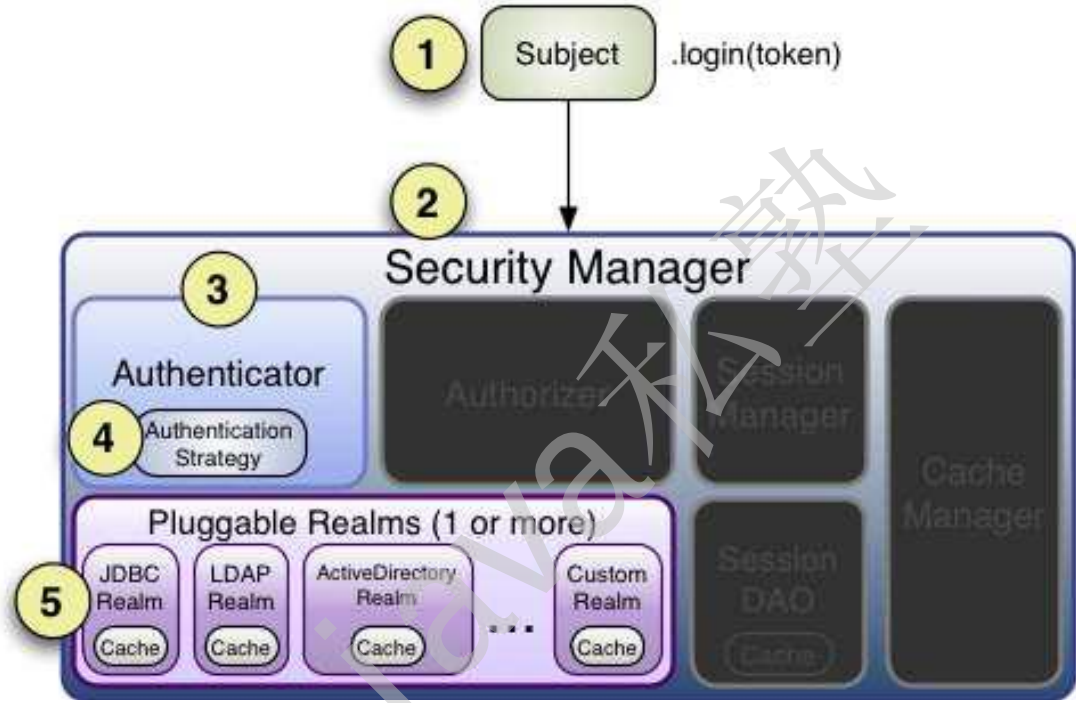
- Authenticated(已认证)

  一个已认证的Subject 是指在当前Session 中被成功地验证过了（也就是说，login方法被调用并且没有抛出异常）。如果subject.isAuthenticated()返回true 则认为Subject 已通过验证。

- 注意他们是互斥的

  Remembered 和Authenticated 是互斥的——若其中一个为真则另一个为假，反之亦然

认证顺序-1



- Step 1：应用程序代码调用Subject.login 方法，传递创建好的包含终端用户的Principals(身份)和Credentials(凭证)的AuthenticationToken 实例。
- Step 2：Subject实例，通常是DelegatingSubject（或子类）委托应用程序的SecurityManager通过调用securityManager.login(token)开始真正的验证。

认证顺序-2

- Step3：SubjectManager 接收token 以及简单地委托给内部的Authenticator 实例通过调用authenticator.authenticate(token)。这通常是一个ModularRealmAuthenticator 实例，支持在身份验证中协调一个或多个Realm 实例。

- Step 4：如果应用程序中配置了一个以上的Realm，ModularRealmAuthenticator实例将利用配置好的AuthenticationStrategy 来启动Multi-Realm 认证尝试。在Realms 被身份验证调用之前，期间和以后，AuthenticationStrategy 被调用使其能够对每个Realm 的结果作出反应。

- Step 5：每个配置的Realm 用来帮助看它是否支持提交的AuthenticationToken。如果支持，那么支持Realm 的getAuthenticationInfo 方法将会伴随着提交的token 被调用。getAuthenticationInfo 方法有效地代表一个特定Realm 的单一的身份验证尝试。

初识自定义Realm

n 这里先来个例子，认识一下：

```
public class MyRealm extends AuthorizingRealm{
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
          String userName = (String) getAvailablePrincipal(principals);
          //通过用户名去获得用户的所有资源，并把资源存入info中
          SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
          Set<String> s = new HashSet<String>();
          s.add("p1");  s.add("p2"); info.setStringPermissions(s);
          Set<String> r = new HashSet<String>();
          r.add("r1"); r.add("r2"); info.setRoles(r);
          return info;}
    protected AuthenticationInfo doGetAuthenticationInfo(
                    AuthenticationToken token) throws AuthenticationException {
          //token中储存着输入的用户名和密码
          UsernamePasswordToken upToken = (UsernamePasswordToken)token;
          String username = upToken.getUsername();
          String password = String.valueOf(upToken.getPassword());
          //通常是与数据库中用户名和密码进行比对，这里就省略了
          //比对成功则返回info，比对失败则抛出对应信息的异常AuthenticationException
          SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(username,
    password .toCharArray(),getName());
          return info;         }}
```

配置多个Realm-1

- 上面的例子可以作为第一个Realm
- 再复制一份，定义为MyRealm2，在返回user前添加抛出一个例外，表示认真没有通过，如下：

```
if(username.equals("javass")){
    throw new AuthenticationException("MyRealm2 认证失败");
}
SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(username,
    password .toCharArray(),getName());
```

- 在配置文件里面添加Realm的定义

```
myRealm1=cn.javass.hello.MyRealm
myRealm2=cn.javass.hello.MyRealm2
```

由于有多个realm，一般就需要配置AuthenticationStrategy了，而AuthenticationStrategy是跟Authenticator（认证器）相关的。

配置多个Realm-2

- 配置Authenticator和AuthenticationStrategy

```
authenticator = org.apache.shiro.authc.pam.ModularRealmAuthenticator
authcStrategy = org.apache.shiro.authc.pam.AllSuccessfulStrategy
authenticator.authenticationStrategy = $authcStrategy
authenticator.realms=$myRealm2,$myRealm1
```

- 当然，你可以扩展并实现自己的Authenticator，一般没有必要
- 最后把Authenticator设置给securityManager

```
securityManager.authenticator = $authenticator
```

- 关于AuthenticationStrategy的配置，有三种：

AtLeastOneSuccessfulStrategy ：如果一个（或更多）Realm 验证成功，则整体的尝试被认为是成功的。如果没有一个验证成功，则整体尝试失败。

FirstSuccessfulStrategy 只有第一个成功地验证的Realm 返回的信息将被使用。所有进一步的Realm 将被忽略。如果没有一个验证成功，则整体尝试失败

AllSucessfulStrategy 为了整体的尝试成功，所有配置的Realm 必须验证成功。如果没有一个验证成功，则整体尝试失败。

ModularRealmAuthenticator 默认的是AtLeastOneSuccessfulStrategy

配置多个Realm-3

- 自定义自己的AuthenticationStrategy，通常是扩展自AbstractAuthenticationStrategy，示例如下：

```
public class MyAuthenticationStrategy extends AbstractAuthenticationStrategy{
    public AuthenticationInfo afterAttempt(Realm realm, AuthenticationToken token,
    AuthenticationInfo singleRealmInfo, AuthenticationInfo aggregateInfo, Throwable
    t) throws AuthenticationException {
          if(realm.getName().equals("myRealm2")){
                if(singleRealmInfo==null ||
    singleRealmInfo.getPrincipals()==null){
          throw new AuthenticationException("主战认证未通过");
                }
          }
return super.afterAttempt(realm, token, singleRealmInfo, aggregateInfo, t);
    }
}
```

至于具体覆盖扩展什么方法，需要根据你具体的策略来定。

多个Realm的验证顺序

- 概述

  非常重要的一点是：ModularRealmAuthenticator 将与Realm 实例以迭代的顺序进行交互。

  在SecurityManager 中已经配置好了ModularRealmAuthenticator 对Realm 实例的访问。当执行一个认证尝试时，它将会遍历该集合，并对每一个支持提交AuthenticationToken 的Realm 调用Realm 的getAuthenticationInfo 方法

- 隐式排列

  当你配置多个realm的时候，处理的顺序默认就是你配置的顺序。

  这种情况通常就是只定义了realm，而没有配置securityManager的realms

- 显示排列

  也就是显示的配置securityManager.realms，那么执行的顺序就是你配置该值的realm的顺序。

  通常更推荐显示排列。

第五章：Shiro的授权（Authorization）

Authorization概述-1

- 概述

  授权，又称作为访问控制，是对资源的访问管理的过程。换句话说，控制谁有权限在应用程序中做什么。

  授权检查的例子是：该用户是否被允许访问这个网页，编辑此数据，查看此按钮，或打印到这台打印机？这些都是决定哪些是用户能够访问的。

- 授权的三要素

  授权有着三个核心元素：权限、角色和用户 。

  我们需要在应用程序中对用户和权限建立关联，通常的做法就是将权限分配给某个角色，然后将这个角色关联一个或多个用户。

- 权限

  是Shiro安全机制最核心的元素。它在应用程序中明确声明了被允许的行为和表现。一个格式良好的权限声明可以清晰表达出用户对该资源拥有的权限。

- 权限声明和粒度

  在shiro中主要通过前面学过的通配符表达式来完成。

Authorization概述-2

- 角色

角色是一个命名的实体，通常代表一组行为或职责。这些行为演化为你在一个软件应用中能或者不能做的事情。角色通常是分配给用户帐户的，因此，通过分配，用户能够“做”的事情可以归属于各种角色。

- Shiro支持的角色类型

1：隐式角色：一个角色代表着一系列的操作，当需要对某一操作进行授权验证时，只需判断是否是该角色即可。这种角色权限相对简单、模糊，不利于扩展。

2：显式角色：一个角色拥有一个权限的集合。授权验证时，需要判断当前角色是否拥有该权限。这种角色权限可以对该角色进行详细的权限描述，适合更复杂的权限设计。 Shiro官方推荐使用这种方式。

- Shiro的三种授权方式

1：编写代码——在Java 代码中用像if 和else 块的结构执行授权检查。

2：JDK 的注解——你可以添加授权注解给你的Java 方法。

3：JSP/GSP 标签库——你可以控制基于角色和权限的JSP 或者GSP 页面输出。

编程授权-1

- 通过使用subject的方法来实现角色的判断，常见的api：

hasRole(String roleName) ：返回true 如果Subject 被分配了指定的角色

hasRoles(List<String> roleNames) ：返回一个与方法参数中目录一致的hasRole结果的数组。

hasAllRoles(Collection<String> roleNames)：返回true 如果Subject 被分配了所有的角色

- 断言支持

Shiro还支持以断言的方式进行授权验证。断言成功，不返回任何值，程序继续执行；断言失败时，将抛出异常信息。方法大致如下：

checkRole(String roleName) 、checkRoles(Collection<String>roleNames)、checkRoles(String… roleNames)

- 基于权限对象的实现

```
Permission printPermission = new PrinterPermission("laser400n", "print");
```

相关方法：isPermitted(Permission p)、isPermitted(List<Permission> perms)、isPermittedAll(Collection<Permission> perms)

编程授权-2

- 基于字符串的实现

```
if (currentUser.isPermitted("printer:print:laserjet4400n"))
```

相关方法：isPermitted(String perm)、isPermitted(String... perms)、isPermittedAll(String... perms)

- 当然上述权限的实现也都可以采用断言的方式

相关方法：

```
checkPermission(Permission p)
checkPermission(String perm)
checkPermissions(Collection<Permission> perms)
checkPermissions(String... perms)
```

基于注解的授权-1

n 需要有AOP框架的支持，这里选择spring，先看看怎么集成配置，看例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:context="http://www.springframework.org/schema/context"
            xmlns:aop="http://www.springframework.org/schema/aop"
            xmlns:tx="http://www.springframework.org/schema/tx"
            xsi:schemaLocation="
            http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
    beans-3.0.xsd
            http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
    3.0.xsd
            http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-
    3.0.xsd">
        <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
        <context:component-scan base-package="cn.javass"></context:component-scan>
        <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
                        depends-on="lifecycleBeanPostProcessor" >
                        <property name="proxyTargetClass" value="true"/>
            </bean>
            <bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>
```

基于注解的授权-2

```
        <bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
                <property name="securityManager" ref="securityManager" />
        </bean>
        <bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
                <property name="staticMethod"
    value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
                <property name="arguments" ref="securityManager"/>
        </bean>

    <bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">
            <property name="cacheManager" ref="cacheManager"/>
         <property name="realm" ref="myRealm"/>
         <property name="sessionManager" ref="sessionManager"/>
        </bean>
        <bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager">
        </bean>
        <bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager" />
        <bean id="myRealm" class="org.apache.shiro.realm.text.IniRealm">
                <property name="resourcePath" value="D:/wp/src/TestShiro.ini"></property>
        </bean>
</beans>
```

基于注解的授权-3

- 测试用的HelloAnno

```
@Service
public class HelloAnno {
    @Autowired
    private org.apache.shiro.mgt.SecurityManager sm = null;

    @RequiresAuthentication
    @RequiresPermissions({"p1"})
    public void t(){
            System.out.println("ok=========");
    }
    public void login(){
            UsernamePasswordToken token = new UsernamePasswordToken("javass","cc");
            token.setRememberMe(true);
            SecurityUtils.setSecurityManager(sm);

            Subject currentUser = SecurityUtils.getSubject();
            currentUser.login(token);
    }
    public static void main(String[] args) {
             ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
             HelloAnno t = (HelloAnno)ctx.getBean("helloAnno");
             t.login();
             t.t();
    }
}
```

基于注解的授权-4

- Shiro提供的注解

1：@RequiresAuthentication ：要求当前Subject 已经在当前的session 中被验证通过才能被注解的类/实例/方法访问或调用。

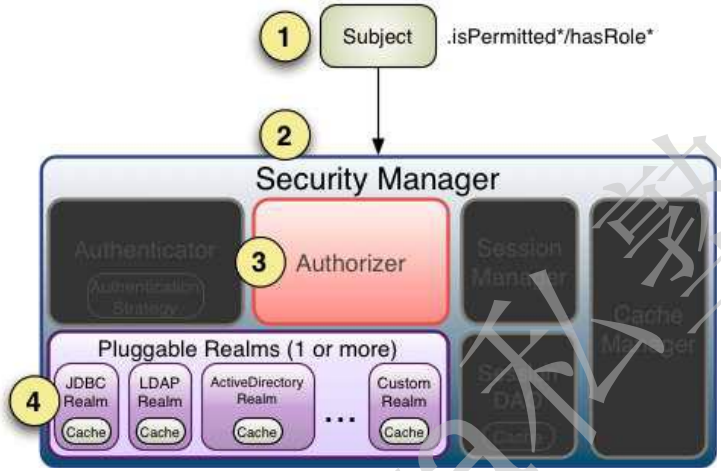
2：@RequiresGuest ：要求当前的Subject 是一个“guest”，也就是他们必须是在之前的session中没有被验证或记住才能被注解的类/实例/方法访问或调用。

3：@RequiresPermissions：要求当前的Subject 被允许一个或多个权限，以便执行注解的方法，比如：@RequiresPermissions("account:create")

4：@RequiresRoles：要求当前的Subject 拥有所有指定的角色。如果他们没有，则该方法将不会被执行，而且AuthorizationException 异常将会被抛出。比如：@RequiresRoles("administrator")

5：@RequiresUser：需要当前的Subject 是一个应用程序用户才能被注解的类/实例/方法访问或调用。要么是通过验证被确认，或者在之前session 中的'RememberMe'服务被记住。

授权的顺序



- Step 1：应用程序或框架代码调用任何Subject的hasRole*, checkRole*, isPermitted*,或者checkPermission*方法的变体，传递任何所需的权限或角色
- Step 2：Subject的实例，通常是DelegatingSubject（或子类）代表应用程序的SecurityManager 通过调用securityManager的几乎各自相同的方法。
- Step 3：SecurityManager，实现org.apache.shiro.authz.Authorizer 接口，他定义了所有Subject具体的授权方法 。默认情况下，authorizer 实例是一个ModularRealmAuthorizer 实例，它支持协调任何授权操作过程中的一个或多个Realm 实例。
- Step 4：每个配置好的Realm 被检查是否实现了相同的Authorizer接口。如果是，Realm 各自的hasRole*, checkRole*,isPermitted*，或checkPermission*方法将被调用。

理解ModularRealmAuthorizer

- ModularRealmAuthorizer 将遍历其内部的Realm 集合，并按迭代顺序与每一个进行交互。每个Realm 的交互功能如下：

1：如果Realm 自己实现了Authorizer 接口，它的各个Authorizer方法将被调用。

（1）如果Realm 的方法导致异常，该异常将会以AuthorizationException 的形式传递给调用者。这将短路授权过程，任何剩余的Realm 将不会被访问

（2）如果该Realm 的方法是一个返回布尔值的hasRole*或者isPermitted*的变体，并且该返回值为true，真值将会立即被返回，同时任何剩余的Realm 都将被短路，这种行为能提高性能。

2：如果Realm 不实现Authorizer 接口，它会被忽略

了解全局的PermissionResolver

- 当执行基于字符串的权限检查是，大多数Shiro 的默认Realm 实现首先将该字符串转换成一个实际的Permission 实例，用的是内部默认实现的WildcardPermissionResolver

- 如果你想要支持自己的权限字符串语法，而且你想要所有配置的Realm 实例支持该语法，你可以将你的PermissionResolver 设置为全局的

- 如果你想配置一个全局的PermissionResolver，每个用来接收配置的PermissionResolver 的Realm 必须实现PermissionResolverAware 接口。这样保证了配置的实例能够被每个支持该配置的Realm 转发。

- 类似的，还有全局的RolePermissionResolver，但请注意：由于这种转换角色名到权限的概念非常特定于应用程序，Shiro 默认Realm 的实现并不使用它们

第六章：Shiro的Realms

Realms概述

- 概述

  Realm 是一个能够访问应用程序特定的安全数据（如用户、角色及权限）的组件。

  Realm 通常和数据源是一对一的对应关系，如关系数据库，LDAP 目录，文件系统，或其他类似资源。Realm 实质上就是一个特定安全的DAO。

  因为这些数据源大多通常存储身份验证数据（如密码的凭证）以及授权数据（如角色或权限），每个Realm能够执行身份验证和授权操作。

- 关于Realm的配置

  这个在前面讲过了，这个就不去赘述了

理解Realms的认证实现

- 前面学到过，Shiro的认证过程最终会交由Realm执行，这时会调用Realm的getAuthenticationInfo(token)方法。

  该方法通常会在org.apache.shiro.realm.AuthenticatingRealm中实现，当然，这个方法中会调用到具体realm实现的方法。

- 该方法主要执行以下操作：

1、检查提交的进行认证的令牌信息
2、根据令牌信息从数据源(通常为数据库)中获取用户信息
3、对用户信息进行匹配验证。
4、验证通过将返回一个封装了用户信息的AuthenticationInfo实例。
5、验证失败则抛出AuthenticationException异常信息。

这是对所有Realm getAuthenticationInfo 实现的最高级别的工作流。

验证通过后，就返回一个非空的AuthenticationInfo 实例来代表来自于该数据源的Subject 帐户信息。

Shiro默认的Realms的认证实现-1

- 所有Shiro 立即可用的Realm 的实现默认使用SimpleCredentialsMatcher。SimpleCredentialsMatcher 执行一个普通的直接相等性的检查，也就是在存储的帐户credentials 与在AuthenticationToken 所提交的之间的检查。
- 使用Hashing Credentials

  如果要使用Hashing Credentials，那么需要在配置中告诉验证器，使用相应的匹配器，这个在前面示例过。

  但是前面直接使用的Sha256Matcher，已经不推荐使用了，现在推荐使用统一的HashedCredentialsMatcher，然后配置具体的算法名称，这些名称按照Java Security Framework里面的标准名称来配置。常见的名称有：

  MD5、AES 、DES 、SHA-1、SHA-256、SHA-384、SHA-512……很多

  具体可以参见：

  http://docs.oracle.com/javase/6/docs/technotes/guides/security/StandardNames.html

Shiro默认的Realms的认证实现-2

- 新的实现配置示例：

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.HashedCredentialsMatcher
sha256Matcher.hashAlgorithmName = SHA-256
sha256Matcher.storedCredentialsHexEncoded=false
iniRealm.credentialsMatcher = $sha256Matcher
myRealm1=cn.javass.hello.MyRealm
myRealm1.credentialsMatcher = $sha256Matcher
authenticator = org.apache.shiro.authc.pam.ModularRealmAuthenticator
authcStrategy = org.apache.shiro.authc.pam.AllSuccessfulStrategy
authenticator.authenticationStrategy = $authcStrategy
authenticator.realms=$myRealm1,iniRealm
securityManager.authenticator = $authenticator
[users]
javass =NVsbv8lnJc3OjOonCP2jEKgObRMxWuxOXuOqdf6AMs4=,role1
[roles]
role1 = p1,p2
```

当然，别忘了在MyRealm中，创建SimpleAuthenticationInfo时传的密码就应该是加密后的字符串了

Shiro默认的Realms的认证实现-3

- 你还可以在密码加密的时候，加点salt，使密码更安全。这种方式目前默认的iniRealm没有支持，只能是在自己扩展的Realm里面使用
- 新的配置文件示例：

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.HashedCredentialsMatcher
sha256Matcher.hashAlgorithmName = SHA-256
sha256Matcher.storedCredentialsHexEncoded=false
sha256Matcher.hashIterations = 10
myRealm1=cn.javass.hello.MyRealm
myRealm1.credentialsMatcher = $sha256Matcher
authenticator = org.apache.shiro.authc.pam.ModularRealmAuthenticator
authenticator.realms=$myRealm1
securityManager.authenticator = $authenticator
[users]
javass =kExd2f52W1M/wXidIRjOfMDj76DVo6e2md+7Rn4ubmY=,role1
[roles]
role1 = p1,p2
```

Shiro默认的Realms的认证实现-4

- 获得加盐后的密码字符串

```
String ss = new Sha256Hash("cc","javass",10).toBase64();
```

- 在自定义的realm中，返回的SimpleAuthenticationInfo需要修改一下，要加入salt的信息，当然，密码也需要是加密后的字符串，修改为：

```
SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(
    username,
    "kExd2f52W1M/wXidIRjOfMDj76DVo6e2md+7Rn4ubmY=" .toCharArray(),
    ByteSource.Util.bytes("javass".getBytes()),
    getName()
);
```

要保证你的Realm 实现必须返回一个SaltedAuthenticationInfo 实例而不是一个普通的AuthenticationInfo 实例。

使用默认的JdbcRealm

- 这个需要在数据库中建立相应的表
- 然后配置相应的数据库连接，然后才能使用，这里以spring中的bean定义来说明一下，示例如下：

```
<bean id="myRealm" class="org.apache.shiro.realm.jdbc.JdbcRealm">
    <property name="dataSource" ref="dataSource" />
    <property name="authenticationQuery"
          value="select u.pwd from tbl_user u where u.uuid = ?" />
    <property name="userRolesQuery"
          value="select r.uuid from tbl_user_role ur left join tbl_role r on
    ur.roleUuid = r.uuid where ur.userUuid = ? " />
    <property name="permissionsQuery"
          value="select p.uuid from tbl_role r left join tbl_role_permission rp on
    r.uuid = rp.roleUuid left join tbl_permission p on rp.permissionUuid = p.uuid
    where r.uuid = ? " />
    <property name="permissionsLookupEnabled" value="true" />
    <property name="saltStyle" value="NO_SALT" />
</bean>
```

自定义Realm-1

- 自定义Reald非常简单，通常是继承AuthorizingRealm 抽象类，这个类实现了常用的authentication 及authorization 工作流来节省你的时间和精力。
- 然后覆盖doGetAuthenticationInfo，在这个方法里面实现获取用户信息
- 基本的示例如下：

```
protected AuthenticationInfo doGetAuthenticationInfo(
    AuthenticationToken token) throws AuthenticationException {
        //token中储存着输入的用户名和密码
        UsernamePasswordToken upToken = (UsernamePasswordToken)token;
        String username = upToken.getUsername();
        //通常是根据用户名去数据库中查询相应信息，这里就省略了
        //当然这里也可以对用户名和密码进行校验
        SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(username,
    password .toCharArray(),getName());
        return info;
}
```

- 然后覆盖doGetAuthorizationInfo，在这个方法里面实现获取用户权限的信息

自定义Realm-2

n 基本的示例如下：

```
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
    principals) {
    String userName = (String) getAvailablePrincipal(principals);
    //通过用户名去获得用户的所有资源，并把资源存入info中
    //当然这里通常会操作数据库去获取
    SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
    Set<String> s = new HashSet<String>();
    s.add("p1");
    s.add("p2");
    info.setStringPermissions(s);
    Set<String> r = new HashSet<String>();
    r.add("r1");
    r.add("r2");
    info.setRoles(r);
    return info;
}
```

在spring的配置文件中使用

- 在Spring的配置文件中，配置的内容和ini文件是一样，只不过转换成xml的风格，用bean的方式来配置
- 基本的示例如下：

```
<bean id="sha256Matcher"
    class="org.apache.shiro.authc.credential.HashedCredentialsMatcher">
    <property name="hashAlgorithmName" value="SHA-256"></property>
    <property name="storedCredentialsHexEncoded" value="false"></property>
    <property name="hashIterations" value="10"></property>
</bean>
<bean id="myRealm2" class="cn.javass.hello.MyRealm">
    <property name="credentialsMatcher" ref="sha256Matcher"></property>
</bean>
```

而且，由于把我们的Realm配置成bean了，自然就可以使用依赖注入等功能了。

第七章：Shiro的Session管理

概述和配置使用-1

- 概述

  Shiro提供安全框架界独一无二的东西：一个完整的企业级Session 解决方案，可以为任意的应用提供session支持，包括web和非web应用，并且无需部署你的应用程序到Web 容器或使用EJB容器。

- 基本使用

可以通过与当前执行的Subject 交互来获取Session：

```
Subject currentUser = SecurityUtils.getSubject();
Session session = currentUser.getSession();
session.setAttribute("someKey", someValue);
```

- 关于SessionManager

  SessionManager是用来管理Session的组件，包括：创建，删除，inactivity(失效)及验证，等等。SessionManager 也是一个由SecurityManager维护的顶级组件。

  shiro提供了默认的SessionManager实现，一般没有必要自定义这个。

配置使用-2

- 设置Sessioin的过期时间

  Shiro 的SessionManager 实现默认是30 分钟会话超时。

  你可以设置SessionManager 默认实现的globalSessionTimeout 属性来为所有的会话定义默认的超时时间。例如，

```
[main]
# 3,600,000 milliseconds = 1 hour
securityManager.sessionManager.globalSessionTimeout = 3600000
```

- Sessioin的事件监听

  你可以实现SessionListener 接口（或扩展易用的SessionListenerAdapter）并与相应的会话操作作出反应。 配置示例：

```
[main]
aSessionListener = com.foo.my.SessionListener
anotherSessionListener = com.foo.my.OtherSessionListener
securityManager.sessionManager.sessionListeners = $aSessionListener,
    $anotherSessionListener
```

SessionDAO-1

- 概述

每当一个会话被创建或更新时，它的数据需要持久化到一个存储位置以便它能够被稍后的应用程序访问，实现这个功能的组件就是SessionDAO。

你能够实现该接口来与你想要的任何数据存储进行通信。这意味着你的会话数据可以驻留在内存中，文件系统，关系数据库或NoSQL 的数据存储，或其他任何你需要的位置。

- 基本配置

SessionDAO是作为一个属性配置在默认的SessionManager 实例上

```
[main]
sessionDAO = com.foo.my.SessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
```

这种SessionDAO主要在本地应用中起作用。

SessionDAO-2

- 基于EHCache的SessionDAO，基本配置如下：

```
[main]
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
securityManager.cacheManager = $cacheManager
```

- Shiro提供了默认的EHCache的配置xml，如果你要配置自己的EHCache.xml，需要注意以下几点：

1：overflowToDisk=“true” - 这确保当你溢出进程内存时，会话不丢失且能够被序列化到磁盘上。

2： eternal=“true” - 确保缓存项（Session 实例）永不过期或被缓存自动清除。这是很有必要的，因为Shiro 基于计划过程完成自己的验证。如果我们关掉这项，缓存将会在Shiro 不知道的情况下清扫这些Sessions，这可能引起麻烦。

3：如果你想使用一个不同的名字而不是默认的，你可以在EnterpriseCacheSessionDAO 上配置名字，例如：sessionDAO.activeSessionsCacheName = myname
只要确保在ehcahe.xml 中有一项与这个名字匹配

Web应用中的Session

- 在web应用上，默认使用的是容器的会话，如果你想基于Web 应用程序启用 SessionDAO 来自定义会话存储或会话群集，你将不得不首先配置一个本地的Web 会话管理器。例如：

```
[main]
sessionManager=org.apache.shiro.web.session.mgt.DefaultWebSessionManager
securityManager.sessionManager = $sessionManager
# Configure a SessionDAO and then set it:
securityManager.sessionManager.sessionDAO = $sessionDAO
```

- 在web应用上，如果想要在每一个请求的基础上启用或禁用会话的创建，可以在配置中的[urls] 里面，为相应的url设置一个noSessionCreation过滤器，如下：

```
[urls]
/rest/** = noSessionCreation, authcBasic
```

自定义SessionDAO-1

- 在某些场景中，我们需要管理用户的Session信息，比如把Session信息放到数据库中，这样就可以记录一个操作日志，或是统计在线人员等等。
- 自定义SessionDAO也非常简单，通常是继承AbstractSessionDAO，实现对Session数据的CRUD即可，简单示例如下：

```
public class MySessionDAO extends AbstractSessionDAO{
private Map<Serializable,Session> map = new HashMap<Serializable,Session>();
    public void update(Session session) throws UnknownSessionException {
          System.out.println("now update session");
          map.put(session.getId(),session);
    }
    public void delete(Session session) {
          System.out.println("now delete session");
          map.remove(session.getId());
    }
    public Collection<Session> getActiveSessions() {
          System.out.println("now getActiveSessions session");
          return map.values();
    }
```

自定义SessionDAO-2

```
    protected Serializable doCreate(Session session) {
          System.out.println("now doCreate session");
          Serializable sessionId = generateSessionId(session);
                assignSessionId(session, sessionId);
                map.put(sessionId, session);

          return sessionId;
    }
    protected Session doReadSession(Serializable sessionId) {
          System.out.println("now doReadSession session");
          return map.get(sessionId);
    }
}
```

- 基本的配置示例：

```
sessionDAO = cn.javass.hello.MySessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
```

第八章：Shiro和Spring的集成

Standalone Applications

- Shiro 应用程序需要一个具有单例SecurityManager 实例的应用程序。请注意，这不会是一个静态的单例，但应该只有一个应用程序能够使用的实例，无论它是否是静态单例的。
- 在Spring 应用程序中启用应用程序单例SecurityManager的最简单配置：

```
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod"
    value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
    <property name="arguments" ref="securityManager"/>
</bean>
<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">
      <property name="realm" ref="myRealm"/>
</bean>
<bean id="myRealm" class="org.apache.shiro.realm.text.IniRealm">
    <property name="resourcePath" value="D:/wp/src/TestShiro.ini"></property>
</bean>
```

Wb应用-1

- 在web.xml中

```
<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>
```

- 在spring的applicationContext.xml 文件中，定义web 支持的SecurityManager和'shiroFilter' bean 将会被web.xml 引用

Wb应用-2

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <!—以下为可选配置，请按照实际项目需要进行配置-->
    <property name="loginUrl" value="/login.jsp"/>
    <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
    <!- filters属性也是可选的，用来声明一些可以在filterChainDefinitions 里面使用的filter。如果你声明的filter
     名称是shiro默认有的，那么将会覆盖默认的filter -->
    <property name="filters">
        <util:map>
     <!- 比如：这个filter是shiro里面有的，那么自定义的filter将会覆盖默认的-->
            <entry key="authc" value-ref="formAuthenticationFilter"/>
            <!- 比如：这个filter是项目新加的filter-->
            <entry key="jCaptchaValidate" value-ref="jCaptchaValidateFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
     <!— 这里配置urls，对不同的url配置需要经过的filter-->
            /jcaptcha* = anon
            /logout = logout
            /login = jCaptchaValidate,authc
        </value>
    </property>
</bean>
```

Wb应用-3

```
<!--替换默认的form 验证过滤器-->
<bean id="formAuthenticationFilter"
   class="cn.javass.CustomFormAuthenticationFilter">
    ......
</bean>
<!—然后就是SecurityManger、Realms等-->
<bean id="securityManager"
   class="org.apache.shiro.mgt.DefaultSecurityManager">
      <property name="realm" ref="myRealm"/>
</bean>
<bean id="myRealm" class="org.apache.shiro.realm.text.IniRealm">
   <property name="resourcePath"
   value="D:/wp/src/TestShiro.ini"></property>
</bean>
```

Wb应用-4

- 开启注解

```
<bean
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoP
    roxyCreator"
    depends-on="lifecycleBeanPostProcessor" >
    <property name="proxyTargetClass" value="true"/>
</bean>
<bean
    class="org.apache.shiro.spring.security.interceptor.AuthorizationAttri
    buteSourceAdvisor">
    <property name="securityManager" ref="securityManager" />
</bean>
```

和Struts2+Spring3的集成-1

- 构建一个动态web工程，然后把包加好，web.xml的配置如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    metadata-complete="true" version="3.0">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:applicationContext*.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <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>
```

和Struts2+Spring3的集成-2

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

    <filter>
            <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
    </filter>
    <filter-mapping>
            <filter-name>struts2</filter-name>
            <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

- Struts2的配置比较简单：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true"/>
    <constant name="struts.locale" value="zh_CN"/>
    <constant name="struts.i18n.encoding" value="utf-8"/>
</struts>
```

和Struts2+Spring3的集成-3

- Spring的配置文件基本就是前面讲到的，合在一起了，示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
    3.0.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
    3.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-
    3.0.xsd ">
    <aop:aspectj-autoproxy proxy-target-class="true"/>
    <tx:annotation-driven transaction-manager="txManager" />
    <context:component-scan base-package="cn.javass"></context:component-scan>
    <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
                    depends-on="lifecycleBeanPostProcessor" >
          <property name="proxyTargetClass" value="true"/>          </bean>
    <bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
          <property name="securityManager" ref="securityManager" />   </bean>
    <bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod" value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
          <property name="arguments" ref="securityManager"/>
    </bean>
```

和Struts2+Spring3的集成-4

```
    <bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
           <property name="securityManager" ref="securityManager" />
           <property name="filterChainDefinitions">
                  <value>
                         /userAction!toList.action = roles[role1]
                  </value>
           </property>
    </bean>
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor" />
    <bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
        <property name="cacheManager" ref="cacheManager"/>
        <property name="realm" ref="myRealm"/>
        <property name="sessionManager" ref="sessionManager"/>
    </bean>
    <bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager"></bean>
    <bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager" />
    <bean id="myRealm" class="org.apache.shiro.realm.text.IniRealm">
           <property name="resourcePath" value="D:/shiroweb/src/TestShiro.ini"></property>
    </bean>
</beans>
```

和Struts2+Spring3的集成-5

- 对应的TestShiro.int做了足够的简化，示例如下：

```
[users]
javass=cc,role1
[roles]
role1 = p1,p2
```

- 写一个LoginAction，示例如下：

```
@Action(
            value="/login",
            results={
                                    @Result(name = "toList", location = "/list.jsp"),
                                    @Result(name = "success", location = "/ok.jsp")})
public class LoginAction extends ActionSupport{
    private String userId = "";
    private String pwd = "";
    //getter setter
    public String execute(){
            UsernamePasswordToken token = new UsernamePasswordToken(userId,pwd);
            token.setRememberMe(true);
            Subject currentUser = SecurityUtils.getSubject();
            currentUser.login(token);
    System.out.println(currentUser.getPrincipal()+"====has r1 === "+currentUser.hasRole("role1"));
            return this.SUCCESS;
    }
}
```

和Struts2+Spring3的集成-6

- 写一个UserAction进行权限测试，示例如下：

```
@Action(
            value="/user",
            results={
                                @Result(name = "toList", location = "/list.jsp"),
                                @Result(name = "success", location = "/ok.jsp")
            }
)
public class UserAction extends ActionSupport{
    @RequiresPermissions({"p1"})
    public String execute(){
            Subject currentUser = SecurityUtils.getSubject();
            System.out.println("now
    user===="+currentUser.getPrincipal()+" ,role=="+currentUser.hasRole("role1")+" ,
    p=="+currentUser.isPermitted("p1"));
            return "toList";
    }
}
```

- 和Struts2集成的时候，如果使用注解的方式进行权限判断，而struts2又使用spring的依赖注入逻辑层的对象的时候，是存在bug的，可能会导致取不到注入的对象。这个bug是struts2的，主要是都是使用的aop，有一些冲突。需要修正，可以加入我们修正的代码包。同时要注意：需要使用getter/setter来注入，不要使用属性的方式。

和SpringMVC+Spring3的集成-1

- 构建一个动态web工程，然后把包加好，web.xml的配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
     id="WebApp_ID" version="3.0"    metadata-complete="true">
     <context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>classpath*:applicationContext*.xml</param-value>
     </context-param>
     <listener>
        <listener-class>
                        org.springframework.web.context.ContextLoaderListener
          </listener-class>
    </listener>
     <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>
```

和SpringMVC+Spring3的集成-2

```
    <filter-mapping>
        <filter-name>shiroFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-servlet-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

- 对应的TestShiro.ini跟前面struts2集成是一样的
- Spring的配置文件基本跟前面那个差不多，只是有部分配置移到springmvc的配置文件中了，示例如下：

和SpringMVC+Spring3的集成-3

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:context="http://www.springframework.org/schema/context"
            xmlns:aop="http://www.springframework.org/schema/aop"
            xmlns:tx="http://www.springframework.org/schema/tx"
            xsi:schemaLocation="
            http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
    beans-3.0.xsd
            http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
    3.0.xsd
            http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-
    3.0.xsd
            "
            >
    <context:component-scan base-package="cn.javass">
            <context:exclude-filter type="annotation"
                        expression="org.springframework.stereotype.Controller" />
    </context:component-scan>
    <aop:aspectj-autoproxy proxy-target-class="true"/>
```

和SpringMVC+Spring3的集成-4

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

                        </value>
            </property>
    </bean>
    <bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor" />
    <bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
        <property name="cacheManager" ref="cacheManager"/>
       <property name="realm" ref="myRealm"/>
       <property name="sessionManager" ref="sessionManager"/>
    </bean>
    <bean id="sessionManager" class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager"></bean>
    <bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager" />
    <bean id="myRealm" class="org.apache.shiro.realm.text.IniRealm">
            <property name="resourcePath" value="D:/newlesson/shiro/webwp/mvc/src/TestShiro.ini"></property>

    </bean>
</beans>
```

和SpringMVC+Spring3的集成-5

- Springmvc的配置文件，示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd"
    default-lazy-init="true">
    <!-- 开启controller注解支持 -->
    <context:component-scan base-package="cn.javass"
            use-default-filters="false">
            <context:include-filter type="annotation"
                    expression="org.springframework.stereotype.Controller" />
    </context:component-scan>
    <!-- 会自动注册了validator ConversionService -->
    <mvc:annotation-driven ></mvc:annotation-driven>
    <mvc:default-servlet-handler />
    <mvc:resources mapping="/static/**" location="/WEB-INF/static/" />
```

和SpringMVC+Spring3的集成-6

```
    <!-- 默认的视图解析器 在上边的解析错误时使用 (默认使用html)- -->
    <bean id="defaultViewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
          <property name="viewClass"          value="org.springframework.web.servlet.view.JstlView" />
          <property name="contentType" value="text/html" />
          <property name="prefix" value="/WEB-INF/jsp/" />
          <property name="suffix" value=".jsp" />
    </bean>
    <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
                  depends-on="lifecycleBeanPostProcessor" >
                  <property name="proxyTargetClass" value="true"/>
          </bean>
          <bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
                  <property name="securityManager" ref="securityManager" />
          </bean>
          <bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
                  <property name="staticMethod"
    value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
                  <property name="arguments" ref="securityManager"/>
          </bean>
</beans>
```

和SpringMVC+Spring3的集成-7

- 简单的HelloWorldController，示例如下：

```
@Controller
public class HelloWorldController {
    @Autowired
    private MyService ms = null;
    @RequestMapping("/login")
public String login(@RequestParam("username") String username,@RequestParam("password") String password) {
            System.out.println("user==" +username+" , pwd=="+password);
            UsernamePasswordToken token = new UsernamePasswordToken(username,password);
            token.setRememberMe(true);
            Subject currentUser = SecurityUtils.getSubject();
            currentUser.login(token);
            System.out.println(currentUser.getPrincipal()+"====has r1 === "+currentUser.hasRole("r1")+" ,
    isau=="+currentUser.isAuthenticated());
            return "login";
    }
    @RequestMapping("/h3")
    @RequiresAuthentication
    @RequiresPermissions({"p1"})
    public String h3(@ModelAttribute("nn")String nn ) {
            ms.doService();
            System.out.println("nn=="+SecurityUtils.getSubject().isAuthenticated());
            return "ok";
    }
}
```

和SpringMVC+Spring3的集成-8

- 对应的MyService非常简单，示例如下：

```
@Service
public class MyService {
    public void doService(){
            System.out.println("now do Service");
    }
}
```

- 再来一个Login.jsp，就可以测试了，示例如下：

```
<form action="/shirolessonmvc/login" method="post">
<table>
    <tr>
            <td>用户名：</td>
            <td><input type="text" name="username"></input></td>
    </tr>
    <tr>
            <td>密码：</td>
            <td><input type="password" name="password"></input></td>
    </tr>
    <tr>
            <td>记住我</td>
            <td><input type="checkbox" name="rememberMe" checked/></td>
            <td><input type="submit"></td>
    </tr>
</table>    </form>
```

第九章：Shiro的Web

URLS配置-1

- Shiro可以和普通web集成的，但考虑到现在的应用基本都会使用spring，所以就不去讲最基本的web集成了。跟Spring集成的方式前面已经讲过了。
- 在Web应用中，可以配置[urls]：

  [urls]项允许你做一些在我们已经见过的任何Web 框架都不存在的东西：在你的应用程序中定义自适应过滤器链来匹配URL 路径!
- [urls]的格式

  URL_Ant_Path_Expression = Path_Specific_Filter_Chain

例如：

```
...
[urls]
/index.html = anon
/user/create = anon
/user/** = authc
/admin/** = authc, roles[administrator]
/rest/** = authc, rest
/remoting/rpc/** = authc, perms["remot:invoke"]
```

URLS配置-2

- URL Path Expressions

  等号左边是一个与Web 应用程序上下文根目录相关的Ant 风格的路径表达式。

  它是第一次匹配优先的方式，比如：

  ```
  /account/** = ssl, authc
  /account/signup = anon
  ```

  如果传入的请求旨在访问/account/signup/index.html（所有'anon'ymous 用户都能访问），那么它将永不会被处理！原因是因为/account/**的模式第一个匹配了传入的请求，“短路”了其余的定义。

- Filter Chain Definitions

  等号右边是逗号隔开的过滤器列表，用来执行匹配该路径的请求。它必须符合以下格式：filter1[optional_config1], filter2[optional_config2], ...

（1）filterN 是一个定义在[main]项中的filter bean 的名字

（2）[optional_configN]是一个可选的括号内的对特定的路径，特定的过滤器有特定含义的字符串（每个过滤器，每个路径的具体配置！）。若果该过滤器对该URL 路径并不需要特定的配置，你可以忽略括号，于是filteNr[]就变成了filterN.

URLS配置-3

- 默认的Filter 实例

Filter Name	Class
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter
perms	org.apache.shiro.web.filter.authz.PermissionAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authz.UserFilter

URLS配置-4

- anon:例子/admins/**=anon 没有参数，表示可以匿名使用
- authc:例如/admins/user/**=authc表示需要认证(登录)才能使用，没有参数
- authcBasic：例如/admins/user/**=authcBasic没有参数表示httpBasic认证

- logout：注销登录的时候，完成一定的功能，任何现有的Session 都将会失效，而且任何身份都将会失去关联（例如，在Web 应用程序中，RememberMe cookie 也将被删除）

- noSessionCreation：阻止在请求期间创建新的会话来保证无状态的体验

- perms：例子/admins/user/**=perms[user:add:*],参数可以写多个，多个时必须加上引号，并且参数之间用逗号分割，例如 /admins/user/**=perms[“user:add:*,user:modify:*”]，当有多个参数时必须每个参数都通过才通过，想当于isPermitedAll()方法。

URLS配置-5

- port：例子/admins/user/**=port[8081],指定请求访问的端口
- rest：例子/admins/user/**=rest[user],根据请求的方法，相当于/admins/user/**=perms[user:method] ,其中method为post，get，delete等。
- roles：例子/admins/user/**=roles[admin],参数可以写多个，多个时必须加上引号，并且参数之间用逗号分割，当有多个参数时，例如admins/user/**=roles[“admin,guest”],每个参数通过才算通过，相当于hasAllRoles()方法。
- ssl:例子/admins/user/**=ssl没有参数，表示安全的url请求，协议为https
- user:例如/admins/user/**=user没有参数表示必须存在用户，当登入操作时不做检查

Tag Libary-1

- 引入Tag

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
```

- The guest tag

guest 标签将显示它包含的内容，仅当当前的Subject 被认为是‘guest’时。‘guest’是指没有身份ID 的任何Subject。也就是说，我们并不知道用户是谁，因为他们没有登录并且他们没有在上一次的访问中被记住（RememberMe 服务）， guest 标签与user 标签逻辑相反。例子：

```
<shiro:guest>
   Hi there! Please <a href="login.jsp">Login</a> or <a
        href="signup.jsp">Signup</a>today!
</shiro:guest>
```

Tag Libary-2

- The user tag

  user 标签将显示它包含的内容，仅当当前的Subject 被认为是‘user’时。‘user’在上下文中被定义为一个已知身份ID的Subject，或是成功通过身份验证及通过‘RememberMe’服务的。请注意这个标签在语义上与authenticated 标签是不同的，authenticated 标签更为严格。usre 标签与guest 标签逻辑相反。

- The authenticated tag

  仅仅只当当前用户在当前会话中成功地通过了身份验证authenticated 标签才会显示包含的内容。它比‘user’标签更为严格。它在逻辑上与‘notAuthenticated’标签相反。

- The notAuthenticated tag

  notAuthenticated 标签将会显示它所包含的内容，如果当前Subject 还没有在其当前会话中成功地通过验证。

- The principal tag

  principal 标签将会输出Subject 的主体（标识属性）或主要的属性。

Tag Libary-3

- The hasRole tag

  hasRole 标签将会显示它所包含的内容，仅当当前Subject 被分配了具体的角色。 hasRole 标签与lacksRole 标签逻辑相反。 例如：

```
<shiro:hasRole name="administrator">
<a href="admin.jsp">Administer the system</a>
</shiro:hasRole>
```

- The lacksRole tag

lacksRole 标签将会显示它所包含的内容，仅当当前Subject 未被分配具体的角色

- The hasAnyRoles tag

  hasAnyRole 标签将会显示它所包含的内容，如果当前的Subject 被分配了任意一个来自于逗号分隔的角色名列表中的具体角色。例如：

```
<shiro:hasAnyRoles name="developer, project manager, administrator">
You are either a developer, project manager, or administrater.
</shiro:hasAnyRoles>
```

Tag Libary-4

- The hasPermission tag

  hasPermission 标签将会显示它所包含的内容，仅当当前Subject“拥有”（蕴含）特定的权限。也就是说，用户具有特定的能力。hasPermission 标签与 lacksPermission 标签逻辑相反。例如：

```
<shiro:hasPermission name="user:create">
<a href="createUser.jsp">Create a new User</a>
</shiro:hasPermission>
```

- The lacksPermission tag

  lacksPermission 标签将会显示它所包含的内容，仅当当前Subject 没有拥有（蕴含）特定的权限。也就是说，用户没有特定的能力。

第十章：Shiro的Cache

概述

- Shiro开发团队明白在许多应用程序中性能是至关重要的。Caching 是Shiro 中的一个重要功能，以确保安全操作保持尽可能的快。

- 但是，Shiro并不实现缓存的功能，Shiro 的缓存支持基本上是一个抽象的（包装）API，它将“坐”在一个基本的缓存机制产品（例如，Ehcache，OSCache，Terracotta，Coherence，GigaSpaces，JBossCache 等）之上。这允许Shiro终端用户配置他们喜欢的任何缓存机制。

Caching API

- Shiro 有三个重要的缓存接口：

1：CacheManager - 负责所有缓存的主要管理组件，它返回Cache 实例

2：Cache - 维护key/value 对

3：CacheManagerAware - 通过想要接收和使用CacheManager 实例的组件来实现

- CacheManager 返回Cache 实例，各种不同的Shiro 组件使用这些Cache 实例来缓存必要的数据。任何实现了CacheManagerAware 的Shiro 组件将会自动地接收一个配置好的CacheManager，该CacheManager 能够用来获取Cache 实例。

- Shiro 的SecurityManager 实现及所有AuthorizingRealm实现都实现了CacheManagerAware

- Shiro 提供了一个个立即可用的EhCacheManager 实现

Caching配置

- 通过在SecurityManager上设置了CacheManger，它反过来也会将它设置到实现了CacheManagerAware 的各种不同的Realm 上，示例如下：

```
cacheManager = org.apache.shiro.cache.ehcache.EhcacheManager
securityManager.cacheManager = $cacheManager
```

- 默认的EHCache使用一个Shiro特定的ehcache.xml文件来配置，大致内容如下：

```
<cache name="shiro-activeSessionCache"
           maxElementsInMemory="10000"
           overflowToDisk="true"
           eternal="true"
           timeToLiveSeconds="0"
           timeToIdleSeconds="0"
           diskPersistent="true"
           diskExpiryThreadIntervalSeconds="600"/>
```

包装使用其他的Cache框架

- 可以通过写一个类来实现Shiro的CacheManager，在这个类里面包装使用任何你想要使用的Cache框架，这里以使用Srping的缓存框架为例，参考如下：

```
public class MyCacheManager implements CacheManager {
    public <K, V> Cache<K, V> getCache(String name) throws CacheException {
        org.springframework.cache.Cache springCache = cacheManager.getCache(name);
        return new SpringCacheWrapper(springCache);
    }
    class SpringCacheWrapper implements Cache {
        private org.springframework.cache.Cache springCache;
        SpringCacheWrapper(org.springframework.cache.Cache springCache) {
            this.springCache = springCache;
        }
        public Object get(Object key) throws CacheException {
            Object value = springCache.get(key);
            if (value instanceof SimpleValueWrapper) {
                return ((SimpleValueWrapper) value).get();
            }
            return value;
        }
//等等，还有几个需要实现的方法，都可以使用你要使用的缓存框架去实现
    }
}
```

缓存数据同步更新的解决方案-1

- 使用Shiro的时候，缓存数据最大的问题就在于数据同步更新。
- 因为Shiro只负责验证部分，如果应用程序修改了人员的权限，那么就需要同步更新到Shiro里面去，也就是要同步Shiro的缓存数据。

- 一个解决方案就是完全废弃Shiro的缓存机制，自己在应用中控制数据的缓存

- 这里给出另一种简易可行的方案：

1：如果你使用的Spring，而且是自定义的Realm，那么可以在你的Realm里面添加一个方法来删除该用户的缓存数据，这样下次shiro在验证这个用户的时候，就会重新去获取数据，从而实现数据的同步

2：由于是自定义的Realm，可以把该对象作为Spring的bean，注入到你的业务对象中，在需要的时候就可以调用该方法来删除shiro的缓存数据了

缓存数据同步更新的解决方案-2

- 示例，比如在前面自定义的MyRealm中，添加如下方法，示例如下：

```
public void removeUserCache(String userId){
    SimplePrincipalCollection pc = new SimplePrincipalCollection();
    pc.add(userId, super.getName());
    super.clearCachedAuthorizationInfo(pc);
}
```

- 然后在HelloAnno中进行测试，示例如下：

1：要注入MyRealm，但注意需要使用getter/setter来注入

2：在main方法中，示例如下：

```
public static void main(String[] args) {
            ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
            HelloAnno t = (HelloAnno)ctx.getBean("helloAnno");
            t.login();
            t.t();

            t.t();

            t.getMr().removeUserCache("javass");
            t.t();
    }
```