Shiro学习笔记--core

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  | |

目录

[1 Authentication（登录认证） 4](#_Toc443467547)

[1.1 基本概念 4](#_Toc443467548)

[1.1.1 Principal 4](#_Toc443467549)

[1.1.2 Primary Principal 4](#_Toc443467550)

[1.1.3 Credentials 4](#_Toc443467551)

[1.2 认证流程 4](#_Toc443467552)

[1.2.1 收集认证信息 5](#_Toc443467553)

[1.2.2 执行认证 5](#_Toc443467554)

[1.2.3 判断认证是否成功 5](#_Toc443467555)

[1.2.4 认证流程 6](#_Toc443467556)

[1.3 Authenticator 7](#_Toc443467557)

[1.3.1 定制Authenticator 7](#_Toc443467558)

[1.4 AuthenticationStrategy 7](#_Toc443467559)

[1.4.1 调用周期 7](#_Toc443467560)

[1.4.2 常用策略 8](#_Toc443467561)

[1.4.3 定制AuthenticationStrategy 8](#_Toc443467562)

[1.5 realm调用顺序 8](#_Toc443467563)

[2 Authorization（授权） 9](#_Toc443467564)

[2.1 基本概念 9](#_Toc443467565)

[2.1.1 Permissions 9](#_Toc443467566)

[2.1.2 Roles 9](#_Toc443467567)

[2.1.3 Users 10](#_Toc443467568)

[2.2 权限控制 10](#_Toc443467569)

[2.2.1 编程方式 10](#_Toc443467570)

[2.2.1.1 基于角色的访问控制 10](#_Toc443467571)

[2.2.1.2 基于权限的访问控制(推荐) 11](#_Toc443467572)

[2.2.2 注解方式 12](#_Toc443467573)

[2.2.3 标签方式 12](#_Toc443467574)

[2.3 权限控制流程 13](#_Toc443467575)

[2.4 ModularRealmAuthorizor 13](#_Toc443467576)

[2.5 realm调用顺序 14](#_Toc443467577)

[2.6 配置PermissionResolver 14](#_Toc443467578)

[2.6.1 配置全局的PermissionResolver 14](#_Toc443467579)

[2.6.2 配置具体realm的PermissionResolver 14](#_Toc443467580)

[2.7 配置RolePermissionResolver 15](#_Toc443467581)

[2.7.1 配置全局的RolePermissionResolver 15](#_Toc443467582)

[2.7.2 配置具体realm的RolePermissionResolver 15](#_Toc443467583)

[2.8 定制Authorizer 15](#_Toc443467584)

[3 权限的理解 15](#_Toc443467585)

[3.1 通配符 15](#_Toc443467586)

[3.2 权限检查 16](#_Toc443467587)

[3.3 性能考量 16](#_Toc443467588)

[4 Realm 16](#_Toc443467589)

[4.1 Realm的理解 16](#_Toc443467590)

[4.2 Realm配置 17](#_Toc443467591)

[4.3 Realm认证 17](#_Toc443467592)

[4.3.1 Realm认证流程 17](#_Toc443467593)

[4.3.2 定制realm 17](#_Toc443467594)

[4.3.3 CredentialsMatcher 17](#_Toc443467595)

[4.3.3.1 SimpleCredentialsMatcher 18](#_Toc443467596)

[4.3.3.2 HashedCredentialsMatcher 18](#_Toc443467597)

[4.3.4 禁用realm的authentication功能 20](#_Toc443467598)

# Authentication（登录认证）

|  |
| --- |
|  |

## 基本概念

### Principal

Subject的身份属性，用来标识用户身份；比如：用户名，邮箱等；

### Primary Principal

能够作为整个系统中唯一的标识；比如：QQ帐号；Shiro建议一个系统中应该有一个明确的Primary Principal；

### Credentials

证书，可以理解为认证身份的材料，可以是密码、生物识别数据（比如指纹、视网膜扫描等）、数字证书（X.509）等；

## 认证流程

登录认证过程大概分为3步：

##### 收集认证信息；

##### 执行认证操作；

##### 处理认证结果；

### 收集认证信息

shiro实现了认证信息来源与认证过程的解耦。Shiro通过AuthenticationToken的实现类进行认证，至于token中的信息从何而来完全由用户决定，比如：登录表单、cli等；如下是token的类图：

|  |
| --- |
| *//获取用户名和密码,创建token* UsernamePasswordToken token = **new** UsernamePasswordToken(**"zhangsan"**, **"passwd"**); token.setRememberMe(**true**); |

AuthenticationToken是所有token的基类，根据不同的应用场景有不同的实现。UsernamePasswordToken类是使用用户名/密码进行登录认证的token；

|  |
| --- |
|  |

### 执行认证

认证过程只需执行Subject的login方法，传入封装认证信息的token对象即可。

|  |
| --- |
| *//通过token执行登录操作* currUser.login(token); |

### 判断认证是否成功

如果判断是否登录，直接使用Subject对象的isAuthenticated()方法即可；如果实在登录过程中判断登录认证是否成功，则可以通过捕获AuthenticationException及其子类，来处理认证失败的情况。

|  |
| --- |
| *//5.检查登录操作,如果未登录,则执行登录操作* **if** (!currUser.isAuthenticated()) {  **try** {  *//5.1 获取用户名和密码,创建token* UsernamePasswordToken token = **new** UsernamePasswordToken(**"zhangsan"**, **"passwd"**);  token.setRememberMe(**true**);   *//5.2 通过token执行登录操作* currUser.login(token);   *//通过异常划分认证失败的场景  //注:出于安全考虑,不应该给出认证失败的细节原因,只给出笼统提示即可* } **catch** (UnknownAccountException | IncorrectCredentialsException | LockedAccountException e) {  ***log***.info(**"用户帐号无效或者密码错误!"**);  *//认证类异常基类* } **catch** (AuthenticationException e) {  ***log***.info(**"因未知原因,登录失败!"**);  } } |

### 认证流程

|  |
| --- |
| http://shiro.apache.org/assets/images/ShiroAuthenticationSequence.png |

##### 调用Subject.login方法，传递AuthenticationToken作为principals和credentials；

##### 将Subject实例（通常是DelegatingSubject或其子类）委托给系统的SecurityManager对象，认证开始；

##### SecurityManager对象接收token，通过调用authenticator.authenticate(token)委托给内部封装的Authenticator对象（通常为ModularRealmAuthenticator实例，能够将多个realm作为可插拔模块）；

##### 如果应用配置了多个realm，则ModularRealmAuthenticator会启用AuthenticationStrategy对象，在调用realm之前、期间、之后的整个生命周期中，调用AuthenticationStrategy对象的相应方法，根据认证策略计算认证结果；对于只配置了一个realm的应用，则ModularRealmAuthenticator直接调用realm的方法进行认证，AuthenticationStrategy不生效；

##### 认证过程中，ModularRealmAuthenticator会询问每个realm是否支持提交的AuthenticationToken对象。如果支持，则调用Realm的getAuthenticationInfo(token)方法进行认证；

|  |
| --- |
| TODO 补充流程图 |

## Authenticator

### 定制Authenticator

|  |
| --- |
|  |

## AuthenticationStrategy

### 调用周期

AuthenticationStrategy是一个无状态的组件，负责在配置多个realm的情况下，根据配置策略合并每个realm的调用结果，封装到单个AuthenticationInfo对象中。在认证过程中，AuthenticationStrategy会被调用4次：

* 在调用所有realm的getAuthenticationInfo方法之前；
* 在调用单个realm的getAuthenticationInfo方法之前；
* 在调用单个realm的getAuthenticationInfo方法之后；
* 在调用所有realm的getAuthenticationInfo方法之后；

### 常用策略

|  |
| --- |
|  |

AuthenticationStrategy有3个具体的实现类：

|  |  |
| --- | --- |
| AtLeastOneSuccessfulStrategy | 或，只要有一个realm认证成功，就算成功； |
| FirstSuccessfulStrategy | 短路或，只要有一个realm认证成功，就算成功； |
| AllSuccessfulStrategy | 与，所有realm认证成功，才算成功； |

### 定制AuthenticationStrategy

|  |
| --- |
|  |

注：

如果需要定制AuthenticationStrategy，则建议以AbstractAuthenticationStrategy为起点，该类实现了将每个realm的结果合并到一个AbstractAuthenticationStrategy实例的方法；

## realm调用顺序

realm的调用顺序，遵循SecurityManager中存放realm集合的迭代顺序。比如：在INI文件中配置realm时，将按照realm在文件中的配置顺序调用；

|  |
| --- |
|  |

# Authorization（授权）

|  |
| --- |
|  |

## 基本概念

### Permissions

代表安全策略的基本元素，定义了系统中可以访问的资源和操作；权限划分的粒度根据系统需求的不同而不同。

### Roles

可以理解为一系列权限的集合；根据对访问控制方式的不同，分为两类：

* 基于角色的访问控制；（缺陷：角色更改后，需要修改关联角色的多个代码或者配置，违反开闭原则）
* 基于资源(权限)的访问控制；

### Users

定义了系统的使用者；shiro提供了灵活的api，系统可以将用户与权限直接关联，也可以通过角色关联；

注：

shiro通过realm实现了权限控制数据和具体业务数据的解耦，提供了一系列权限控制的api供我们调用；但具体业务仍然需要我们自己设计和实现；

## 权限控制

shiro提供了3中方式给资源添加权限控制：

* **编程方式：**直接在代码中通过if-else进行判断；
* **注解方式：**通过在java方法上添加注解；
* **标签方式：**在jsp/gsp中使用标签；

### 编程方式

编程方式是最简洁明了的方式，但也是耦合度最高的方式。即可以使用if语句进行判断，然后执行相应处理；也可是使用断言的方式，当断言不满足时，抛出AuthorizationException。根据权限判定方式的不同，划分为基于角色的访问控制和基于权限的访问控制。

#### 基于角色的访问控制

示例：

|  |
| --- |
| *//判断用户是否具有角色* **if** (currUser.hasRole(**"PM"**)) {  ***log***.info(**"You are PM!"**); } **else** {  ***log***.info(**"You are not a PM!"**); }  *//通过断言判断用户是否具有角色* **try** {  currUser.checkRole(**"QA"**); } **catch** (AuthorizationException ae) {  ***log***.info(**"用户不具备QA角色"**); } |

参考API：

|  |
| --- |
|  |

#### 基于权限的访问控制(推荐)

示例：

|  |
| --- |
| *//通过权限对象判断用户是否具有权限* Permission codingPermission = **new** WildcardPermission(**"coding"**); **if**(!currUser.isPermitted(codingPermission)) {  ***log***.info(**"用户不具备coding权限!"**); }  *//判断用户是否具有权限* **if** (currUser.isPermitted(**"coding"**)) {  ***log***.info(**"you can enjoy youer coding works!"**); } **else** {  ***log***.info(**"Buddy,can you coding!"**); }  *//判断是否具有嵌套权限* **if** (currUser.isPermitted(**"server:online:operation:view"**)) {  ***log***.info(**"you can play all the servers online!"**); } **else** {  ***log***.info(**"Sorry,don't touch my precious!"**); } |

API参考：

|  |
| --- |
|  |

### 注解方式

|  |  |  |  |
| --- | --- | --- | --- |
| 注解 | 等效代码 | | |
| @RequiresAuthentication |  | | |
| @RequiresGuest |  | | |
| @RequiresUser | |  | |
| @RequiresPermissions("account:create") | | |  |
| @RequiresRoles("administrator") | | |  |

### 标签方式

shiro提供了JSP/GSP控制标签库，能够对web页面进行权限控制；

## 权限控制流程

|  |
| --- |
| http://shiro.apache.org/assets/images/ShiroAuthorizationSequence.png |

##### 应用程序调用Subject对象的hasRole\*，checkRole\*,isPermitted\*，或者checkPermission\*方法；

##### Subject(通常为DelegatingSubject对象)，将调用SecurityManager相应的hasRole\*,checkRole\*,isPermitted\*,checkPermission\*方法；（SecurityManager实现了Authroizer接口，猜测可能使用代理模式）

##### SecurityManager调用其封装的Authorizer对象（ModularRealmAuthorizer）相应的hasRole\*,checkRole\*,isPermitted\*,checkPermission\*方法；

##### 检查每个realm是否实现相同的Authorizer接口，如果是，则调用每个realm的hasRole\*,checkRole\*,isPermitted\*,checkPermission\*方法；

## ModularRealmAuthorizor

能够协调处理多个realm，处理策略如下：

* 如果Realm实现Authroizor接口，则Realm相应的hasRole\*,checkRole\*,isPermitted\*,checkPermission\*方法将会被调用；反之，则忽略Realm；
* 如果发生异常，则异常将会被传播给Subject，权限控制流程结束；
* 如果hasRole\*，isPermitted\*方法返回true，则返回true，权限控制流程结束；之所以使用短路策略，是为了提高性能；

## realm调用顺序

和Authenticator调用顺序类似；realm的调用顺序，遵循SecurityManager中存放realm集合的迭代顺序。比如：在INI文件中配置realm时，将按照realm在文件中的配置顺序调用；

|  |
| --- |
|  |

## 配置PermissionResolver

### 配置全局的PermissionResolver

通过权限字符串验证权限时，需要使用PermissionResolver对象对权限字符串进行解析。Shiro默认使用WildCardPermissionResolver进行权限字符串解析；如果需要自定义格式的权限字符串，则可以定制实现相应的PermissionResolver；配置如下：

|  |
| --- |
|  |

注：当配置全局的PermissionResolver时，所有的realm都必须实现PermisionResolverAware接口，以保证resolver对象能够被注入到每一个realm中；

### 配置具体realm的PermissionResolver

当仅需要改变部分realm的PermissionResolver时，可以通过配置，指定每个realm的PermissionResolver实现；

|  |
| --- |
|  |

注：当配置全局的RolePermissionResolver时，所有的realm都必须实现RolePermisionResolverAware接口，以保证resolver对象能够被注入到每一个realm中；

## 配置RolePermissionResolver

### 配置全局的RolePermissionResolver

与PermissionResolver类似，RolePermissionResolver也是解析权限字符串，进行权限检查，但是后者的字符串只能是角色名，用来将角色名转换成一系列的Permission对象；RolePermissionResolver主要用在realm不支持permission这一概念的场景；配置如下：

|  |
| --- |
|  |

### 配置具体realm的RolePermissionResolver

当仅需要改变部分realm的RolePermissionResolver时，可以通过配置，指定每个realm的RolePermissionResolver实现；

|  |
| --- |
|  |

## 定制Authorizer

|  |
| --- |
|  |

# Permission（权限）

权限是安全策略中的最小单位，决定了系统中的What；

## 通配符

使用权限字符串进行权限检查时，shiro默认使用WildCardPermission实例；该实例可以支持一系列通配符；如下两种权限检查方式是等价的：

|  |
| --- |
| **if** (currUser.isPermitted(**"coding"**)) { |

|  |
| --- |
| **if** (currUser.isPermitted(**new** WildcardPermission(**"coding"**))) { |

嵌套权限

|  |
| --- |
| printer:print |

多个权限：printer:print，printer:query

|  |
| --- |
| printer:print,query |

匹配所有权限

|  |
| --- |
| printer:\*  \*:view |

\*和,可以被用在权限字符串的任意地方；

## 权限检查

通常，权限检查中使用的权限字符串表达越明确越好；

## 性能考量

1. 短路
2. 缓存
3. 定制Realm/Permission

# Realm

|  |
| --- |
|  |

## Realm的理解

1. Realm用来接收应用具体的数据，比如用户信息、角色、权限等信息，并将其与Shiro的模型进行映射，以便于shiro能够与应用相结合；
2. Realm与数据源通常是1对1映射；数据源可以是关系数据库、LDAP目录、文件系统等；
3. Realm子类实现了Realm接口，能够通过不同的方式从数据源获取权限相关（用户、权限等）数据；比如：JDBC，文件IO，Hibernate/JPA等；
4. Realm可以理解为一个安全相关的DAO；
5. Realm既包含认证操作，也包含权限检查操作；

## Realm配置

|  |
| --- |
|  |

## Realm认证

|  |
| --- |
| realm接口定义 |

Realm在调用getAuthenticationInfo(token)进行认证之前，会调用realm的supperts(token)方法，确认当前realm是否支持传入token，如果不支持，则返回FALSE；只有返回True时，getAuthenticationInfo(token)才会被调用；当且仅当token符合认证，realm才返回一个非空的AuthenticationInfo实例；当认证失败时，会抛出AuthenticationException；

### Realm认证流程

1. 检查token中的principal；
2. 根据principal，从数据源中查询相关的账户数据；
3. 确认token提供的证书与数据源存放的数据相符合；
4. 如果证书符合，则返回一个存放认证信息的AuthenticationInfo实例；
5. 如果证书不符合，则抛出AuthenticationException；

### 定制realm

注：一般而言，直接实现Realm接口是比较耗时且容易出错的；继承AuthorizingRealm抽象类相对简单，该类实现了通用的认证和权限检查流程，能够节约时间；

### CredentialsMatcher

|  |
| --- |
|  |

证书匹配过程有realm执行，因为证书的格式和存储方式与realm的类型强相关；而Authenticator只负责整个认证流程控制；

|  |
| --- |
|  |

几乎所有应用中的证书匹配流程都是大致一样的，仅是数据匹配方式的不同，为了保证组件的可插拔性和可定制性，shiro提供了AuthenticatingRealm及其子类，能够支持CredentialsMatcher这一概念，将数据匹配的实现放到具体的CredentialsMatcher实例中，当realm获取数据源的认证数据后，将使用CredentialsMatcher实例判定token数据与数据源存放数据是否匹配；shrio提供了多种Matcher，如： [SimpleCredentialsMatcher](http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authc/credential/SimpleCredentialsMatcher.html)，[HashedCredentialsMatcher](http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authc/credential/HashedCredentialsMatcher.html)；不同的数据匹配方式可以使用不同的CredentialsMatcher实例进行；如果已有的Matcher不满足，也可以定制实现CredentialsMatcher，示例如下：

编程式：

|  |
| --- |
|  |

使用ini配置文件：

|  |
| --- |
|  |

#### SimpleCredentialsMatcher

直接对证书和token中的认证信息进行相等性比较；比如：如果使用UsernamePasswordToken对象，则直接比较token中的password和数据源中存放的password是否一致；

#### HashedCredentialsMatcher

相比较直接存放原始认证信息而言，使用hash算法处理认证信息后再存放更加安全；如下是使用hash方式处理认证信息的示例：

编程方式：

|  |
| --- |
|  |

Ini配置方式

|  |
| --- |
|  |

使用HashedCredentialsMatcher，将返回SaltedAuthenticationInfo实例，该类型实例存放了用来进行hash的salt信息；

|  |
| --- |
|  |

### 禁用realm的authentication功能

有些情况需要禁用某个realm的authentication功能，则只需要使realm的supports(token)方法永远返回false即可；

# Session Management（会话管理）

Shiro支持简单独立的企业级会话管理，避免了因为使用会话而不得不引入Servler、EJB等框架；

## 支持特性

## 使用Session

|  |
| --- |
| Subject currUser = SecurityUtils.*getSubject*();  *//3.获取session* Session session = currUser.getSession(); |

Suject.getSession(boolean create) 类似于HttpServletRequest.getsession(Boolean create)，但是可以在任何应用中调用；

1. 如果Subject已经有session，则返回session实例，参数忽略；
2. 如果Subject没有session，且参数为true，则创建session实例并返回；
3. 如果Subject没有session，且参数为false，则返回null；

## SessionManager

1. SessionManager负责管理系统中所有Subject的session，包括：创建、删除、生效、失效等；SessionManager作为顶层组件，直接被SecurityManager管理；
2. SessionManager的默认实现是DefaultSessionManager，它提供了所有企业级的会话管理特性；（Web应用默认使用的是Servlet容器默认的SessionManager）
3. 像SecurityManager的其它顶层组件一样，SessionManager也可以通过setter方法进行设置，或者通过INI配置文件注入；

|  |
| --- |
|  |

### 超时设置

session默认的超时时间是30分钟，即从用户最后一次访问session到当前时间大于30分钟，就视作超时。超时后的session将不可以使用；超时时间可以通过INI配置文件进行设置：

|  |
| --- |
|  |

globalSessionTimeout默认对所有新创建的session生效；如果仅想改变某个session的超时时间，可以单独调用setTimeout(long maxIdleTimeInMillis)方法设置超时时间;

### SessionListener（session监听）

Shiro的session支持监听器注册，可以通过实现SessionListener接口或者继承SessionListenerAdapter来创建监听器；监听器可以在INI配置文件中配置：

|  |
| --- |
|  |

注：session监听器，针对系统中的所有session进行监听，而非针对特定的session进行监听；

### SessionDao（session数据持久化）

Shiro针对Session数据的持久化，提供了SessionDAO接口，提供了对session数据的CRUD操作API；通过不同的DAO实现，能够将session数据持久化到各种数据源中，如：文件系统、关系数据库、内存等；

|  |
| --- |
|  |

SessionDAO可以通过INI配置，样例如下：

|  |
| --- |
|  |

注：对于Web应用而言，shiro使用的是Servlet容器默认的SessionManager，该SessionManager属于Servlet的API，并不支持SessionDao；如果需要支持定制化的SessionDao，则需要设置SecurityManager的SessionManager为DefaultWebSessionManager；

|  |
| --- |
|  |

### Ehcache支持

1. 缓存支持：Shiro支持使用EHCache，能够将session数据持久化至缓存中，并在内存溢出时，将溢出数据持久化到硬盘上；
2. 集群支持：当结合EHCache和TerraCotta时，能够实现相对独立与具体容器的集群，不需要担心具体的集群策略；

配置步骤如下：

1. 使用ehcache需要将shiro-ehcache-<version>.jar包含在类路径下；
2. 在INI配置文件中配置；

|  |
| --- |
|  |

设置SecurityManager的cacheManager为EhCacheManager，则默认会使用EnterpriseCacheSessionDAO；

1. 配置ehcache.xml；shiro提供了默认的ehcache.xml配置，如下：

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

如果需要更改，则至少需要保证如下两个属性配置不变：

* overflowToDisk="true" 确保当内存溢出时，能够将溢出数据保存到硬盘上；
* eternal="true" 确保缓存节点不过期，因为shiro本身提供了session失效的机制，无需底层缓存去决定数据是否失效；

设置Session的缓存名称

|  |
| --- |
|  |

### SessionIdGenerator（定制sessionId）

Shiro使用内部的SessionIdGenerator为每个新建的session生成Id，默认为JavaUuidSessionIdGenerator；可以通过实现SessionIdGenerator接口来定制实现；

|  |
| --- |
|  |

### SessionValidationScheduler（session有效性检查）

1. 和Servlet容器的session不同，shiro的session在过期后，并不会销毁对象。要查看session是否有效，只能通过subject.getSession()方法来判断；这可能导致出现大量的过期session对象撑满内存的情况出现；为防止这种情况，shiro提供了SessionValidationScheduler接口及其实现类，能够周期性的检查session对象的有效性，并将失效对象删除；
2. shiro默认使用ExecutorServiceSessionValidationScheduler作为SessionValidationScheduler实现类，该类使用JDK原生的ScheduledExecutorService控制定时任务；
3. 默认情况下，定时任务1小时执行依次，可以通过配置文件修改任务执行时间；

|  |
| --- |
|  |

1. 定制SessionValidationScheduler：

|  |
| --- |
|  |

1. 禁用session有效性检查；

|  |
| --- |
|  |

1. 删除无效session；

当检查到无效session时，shiro会通过SessionDAO.delete(session) 方法删除无效session数据；可以通过配置，禁止删除无效session的数据；

|  |
| --- |
|  |

## session集群支持

shiro能够支持session的集群，且SessionManager不需要与具体集群信息强耦合；实现思路是通过实现集群策略的SessionDao与集群进行交互，而SessionManager只需要调用SessionDao的接口即可，无须关心与集群是如何交互的；

### 分布式缓存

分布式缓存例如ehcache+terracotta，GigaSpaces，Oracle Coherence，Memcached等已经解决了分布式数据持久化的问题，所以只需要配置shiro支持某种分布式缓存即可；这使得可以为系统灵活选择合适的缓存；

注：使用分布式缓存需要保证如下两个条件：

1. 分布式缓存有足够内存存放所有session；
2. 如果内存不足，分布式缓存能够将溢出数据存放到磁盘；

### EnterpriseCacheSessionDAO

Shiro提供了支持分布式缓存的SessionDAO，配置如下：

|  |
| --- |
|  |

如上配置，直接指定session的cache对象；但是如果需要让shiro的所有组件都能够使用cache，则需要为SecurityManager指定CacheManager；配置如下：

|  |
| --- |
|  |

问：配置中没有说明SessionDao要使用Cache或者CacheManager，那么SessionDao如何使用分布式缓存呢？

1. 通过指定SecurityManager的CacheManager时，shiro会检查所有实现CacheManagerAware接口的实例，并将CacheManager注入给相应的实例；
2. 而EnterpriseCacheSessionDAO实现了CacheManagerAware，所以可以获取CacheManager对象；
3. 当需要访问缓存时，SessionDAO将根据缓存名称从CacheManager中获取相应的缓存对象，并执行CRUD操作；

### Ehcache+Terracotta配置

配置样例：

1. 配置ehcache.xml文件

|  |
| --- |
|  |

通过<terracottaConfig url="localhost:9510"/>可以配置terracotta服务器；需要配置overflowToDisk="false"，diskPersistent="false"；因为集群不支持；

1. 配置INI文件

|  |
| --- |
|  |

注：由于INI配置文件的顺序性所致，需要将securityManager.cacheManager = $cacheManager放在最后，以确保所有实现了CacheManagerAware

接口的对象都能够获取到CacheManager对象；

### Session状态

Shiro使用session存放Subject的Principal信息，认证状态(调用subject.isAuthenticated()方法)等；典型场景是在登录后或者通过rememberMe功能加载Subject时；如此实现带来的好处有如下两点：

1. 能够方便的使用Subject.Builder来给未绑定session的请求创建Subject对象；比如：在第三方系统中需要通过sessionId获取相应的Subject对象，但该对象实际上存放在另一个系统中；

|  |
| --- |
|  |

1. rememberMe功能仅需要在第一次请求过来时持久化，而不需要在每次请求过来时都对rememberMe进行判断；从而提高程序的性能；

### 无session应用（无状态）

有些应用本身不需要维护请求状态，即不需要使用session保存请求的状态；故在用户认证时，需要在每一次请求中执行认证操作；

#### 设置无状态session

|  |
| --- |
|  |

注：该配置仅是禁用session的存储策略，对每次请求的状态不进行保存。仍然可以创建session和获取session;

#### 有状态和无状态session混合

有些场景需要保证有状态session和无状态session同时存在，比如：通过浏览器访问的请求需要维护状态，而通过soap等协议访问的请求不需要维护状态；则需要通过实现SessionStorageEvaluator接口，来控制哪些Subject的session是有状态的，哪些是无状态的；

|  |
| --- |
|  |

配置

|  |
| --- |
|  |

### web中url过滤有状态和无状态请求

对与web应用中，同时包含有状态请求和无状态请求的应用，可以通过urls中配置的过滤规则进行过滤区分；该过滤器配置需要放在urls部分的最前面；

|  |
| --- |
|  |

对于未创建session的请求，如果调用如下方法，将会抛出DisabledSessionException异常；对于已有session的请求，则不会；

* httpServletRequest.getSession()
* httpServletRequest.getSession(true)
* subject.getSession()
* subject.getSession(true)