跟我学Shiro

张开涛 著

示例：https://github.com/zhangkaitao/shiro-example

博客：http://jinnianshilongnian.iteye.com/

交流群：231889722/134755960

写给自己的话

简单，踏实。

2014 年 2 月 21 日 20 点 18 分 谨记

http://jinnianshilongnian.iteye.com/

目　录

http://jinnianshilongnian.iteye.com/

http://jinnianshilongnian.iteye.com/

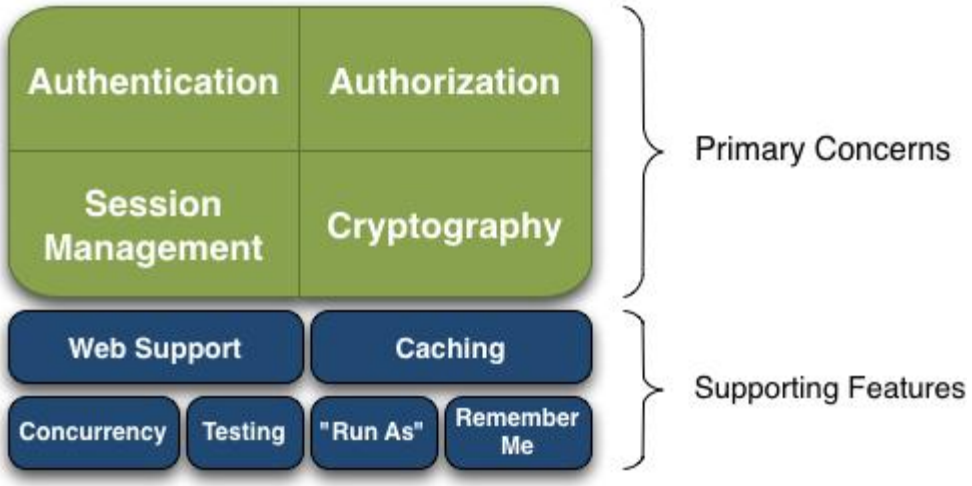
http://jinnianshilongnian.iteye.com/

第一章　Shiro 简介

简介

Apache Shiro 是 Java 的一个安全框架。目前，使用 Apache Shiro 的人越来越多，因为它相当简单，对比 Spring Security，可能没有 Spring Security 做的功能强大，但是在实际工作时可能并不需要那么复杂的东西，所以使用小而简单的 Shiro 就足够了。对于它俩到底哪个好，这个不必纠结，能更简单的解决项目问题就好了。
本教程只介绍基本的 Shiro 使用，不会过多分析源码等，重在使用。

Shiro 可以非常容易的开发出足够好的应用，其不仅可以用在 JavaSE 环境，也可以用在 JavaEE 环境。Shiro 可以帮助我们完成：认证、授权、加密、会话管理、与 Web 集成、缓存等。这不就是我们想要的嘛，而且 Shiro 的 API 也是非常简单；其基本功能点如下图所示：



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；

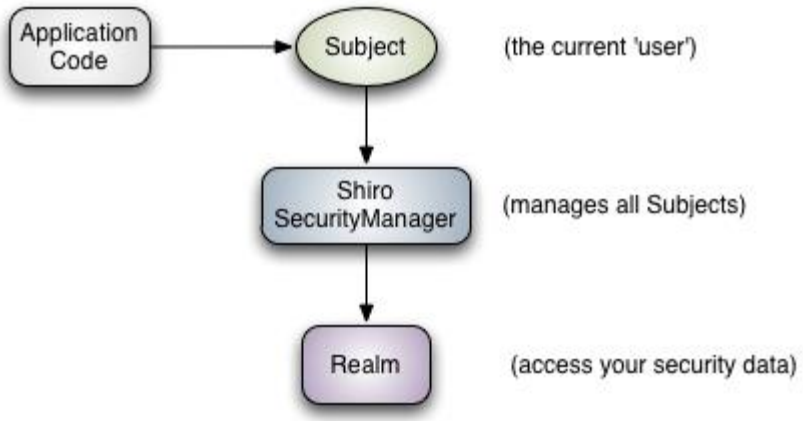
Run As：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；

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。

记住一点，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。

接下来我们分别从外部和内部来看看 Shiro 的架构，对于一个好的框架，从外部来看应该具有非常简单易于使用的 API，且 API 契约明确；从内部来看的话，其应该有一个可扩展的架构，即非常容易插入用户自定义实现，因为任何框架都不能满足所有需求。

首先，我们从外部来看 Shiro 吧，即从应用程序角度的来观察如何使用 Shiro 完成工作。如下图：



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；

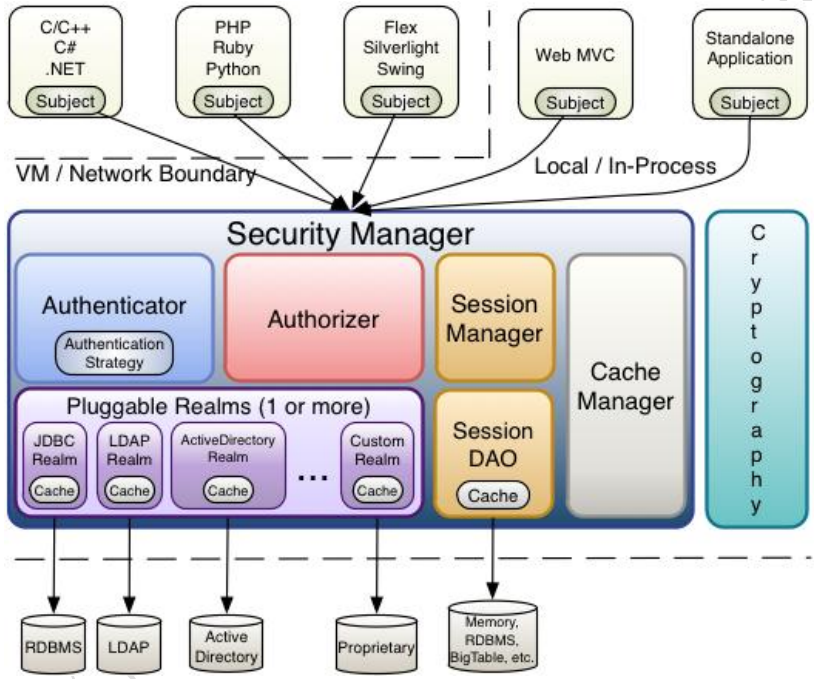
也需要从 Realm 得到用户相应的角色/权限进行验证用户是否能进行操作；可以把 Realm 看成 DataSource，即安全数据源。

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。

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。

接下来我们来从 Shiro 内部来看下 Shiro 的架构，如下图所示：



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。

到此 Shiro 架构及其组件就认识完了，接下来挨着学习 Shiro 的组件吧。

http://jinnianshilongnian.iteye.com/

第二章　身份验证

身份验证，即在应用中谁能证明他就是他本人。一般提供如他们的身份 ID 一些标识信息来表明他就是他本人，如提供身份证，用户名/密码来证明。

在 shiro 中，用户需要提供 principals （身份）和 credentials（证明）给 shiro，从而应用能验证用户身份：

principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。一个主体可以有多个 principals，但只有一个 Primary principals，一般是用户名/密码/手机号。

credentials：证明/凭证，即只有主体知道的安全值，如密码/数字证书等。

最常见的 principals 和 credentials 组合就是用户名/密码了。接下来先进行一个基本的身份认证。

另外两个相关的概念是之前提到的 **Subject** 及 **Realm**，分别是主体及验证主体的数据源。

环境准备

本文使用 Maven 构建，因此需要一点 Maven 知识。首先准备环境依赖：

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.9</version>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-core</artifactId>
        <version>1.2.2</version>
    </dependency>
</dependencies>
```

添加 junit、common-logging 及 shiro-core 依赖即可。

登录/退出

1、首先准备一些用户身份/凭据（shiro.ini）

```
[users]
zhang=123
wang=123
```

此处使用 ini 配置文件，通过[users]指定了两个主体：zhang/123、wang/123。

2、测试用例（com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest）

```
@Test
public void testHelloworld() {
    //1、获取 SecurityManager 工厂，此处使用 Ini 配置文件初始化 SecurityManager
    Factory<org.apache.shiro.mgt.SecurityManager> factory =
            new IniSecurityManagerFactory("classpath:shiro.ini");
    //2、得到 SecurityManager 实例 并绑定给 SecurityUtils
    org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
    SecurityUtils.setSecurityManager(securityManager);
    //3、得到 Subject 及创建用户名/密码身份验证 Token（即用户身份/凭证）
    Subject subject = SecurityUtils.getSubject();
    UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");

    try {
        //4、登录，即身份验证
        subject.login(token);
    } catch (AuthenticationException e) {
        //5、身份验证失败
    }

    Assert.assertEquals(true, subject.isAuthenticated()); //断言用户已经登录

    //6、退出
    subject.logout();
}
```

2.1、首先通过 new IniSecurityManagerFactory 并指定一个 ini 配置文件来创建一个 SecurityManager 工厂；

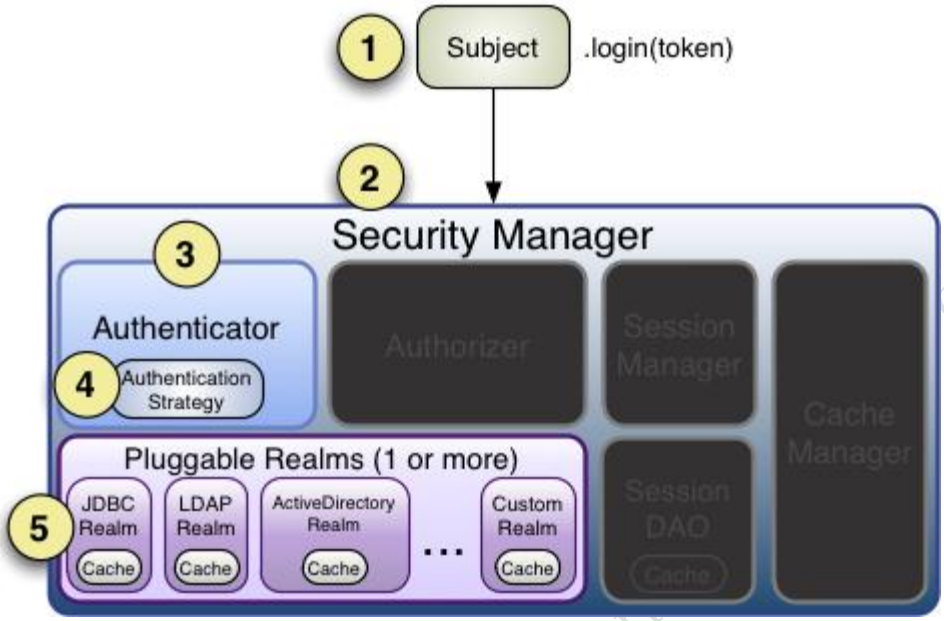
2.2、接着获取 SecurityManager 并绑定到 SecurityUtils，这是一个全局设置，设置一次即可；
2.3、通过 SecurityUtils 得到 Subject，其会自动绑定到当前线程；如果在 web 环境在请求结束时需要解除绑定；然后获取身份验证的 Token，如用户名/密码；
2.4、调用 subject.login 方法进行登录，其会自动委托给 SecurityManager.login 方法进行登录；
2.5、如果身份验证失败请捕获 AuthenticationException 或其子类，常见的如：DisabledAccountException（禁用的帐号）、LockedAccountException（锁定的帐号）、UnknownAccountException（错误的帐号）、ExcessiveAttemptsException（登录失败次数过多）、IncorrectCredentialsException （错误的凭证）、ExpiredCredentialsException（过期的凭证）等，具体请查看其继承关系；对于页面的错误消息展示，最好使用如“用户名/密码错误”而不是“用户名错误”/“密码错误”，防止一些恶意用户非法扫描帐号库；
2.6、最后可以调用 subject.logout 退出，其会自动委托给 SecurityManager.logout 方法退出。

从如上代码可总结出身份验证的步骤：
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，如果失败将得到相应的 AuthenticationException 异常，根据异常提示用户错误信息；否则登录成功；
3、最后调用 Subject.logout 进行退出操作。

如上测试的几个问题：
1、用户名/密码硬编码在 ini 配置文件，以后需要改成如数据库存储，且密码需要加密存储；
2、用户身份 Token 可能不仅仅是用户名/密码，也可能还有其他的，如登录时允许用户名/邮箱/手机号同时登录。

http://jinnianshilongnian.iteye.com/

身份认证流程



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。

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
```

单 Realm 配置

1、自定义 Realm 实现（com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1）：

```
public class MyRealm1 implements Realm {
    @Override
    public String getName() {
        return "myrealm1";
    }
    @Override
    public boolean supports(AuthenticationToken token) {
        //仅支持 UsernamePasswordToken 类型的 Token
        return token instanceof UsernamePasswordToken;
    }
    @Override
    public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        String username = (String)token.getPrincipal();  //得到用户名
        String password = new String((char[])token.getCredentials()); //得到密码
        if(!"zhang".equals(username)) {
            throw new UnknownAccountException(); //如果用户名错误
        }
        if(!"123".equals(password)) {
            throw new IncorrectCredentialsException(); //如果密码错误
        }
        //如果身份认证验证成功，返回一个 AuthenticationInfo 实现；
        return new SimpleAuthenticationInfo(username, password, getName());
    }
}
```

2、ini 配置文件指定自定义 Realm 实现(shiro-realm.ini)

```
#声明一个 realm
myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
#指定 securityManager 的 realms 实现
securityManager.realms=$myRealm1
```

通过$name 来引入之前的 realm 定义

3、测试用例请参考 com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest 的 testCustomRealm 测试方法，只需要把之前的 shiro.ini 配置文件改成 shiro-realm.ini 即可。

多 Realm 配置

1、ini 配置文件（shiro-multi-realm.ini）

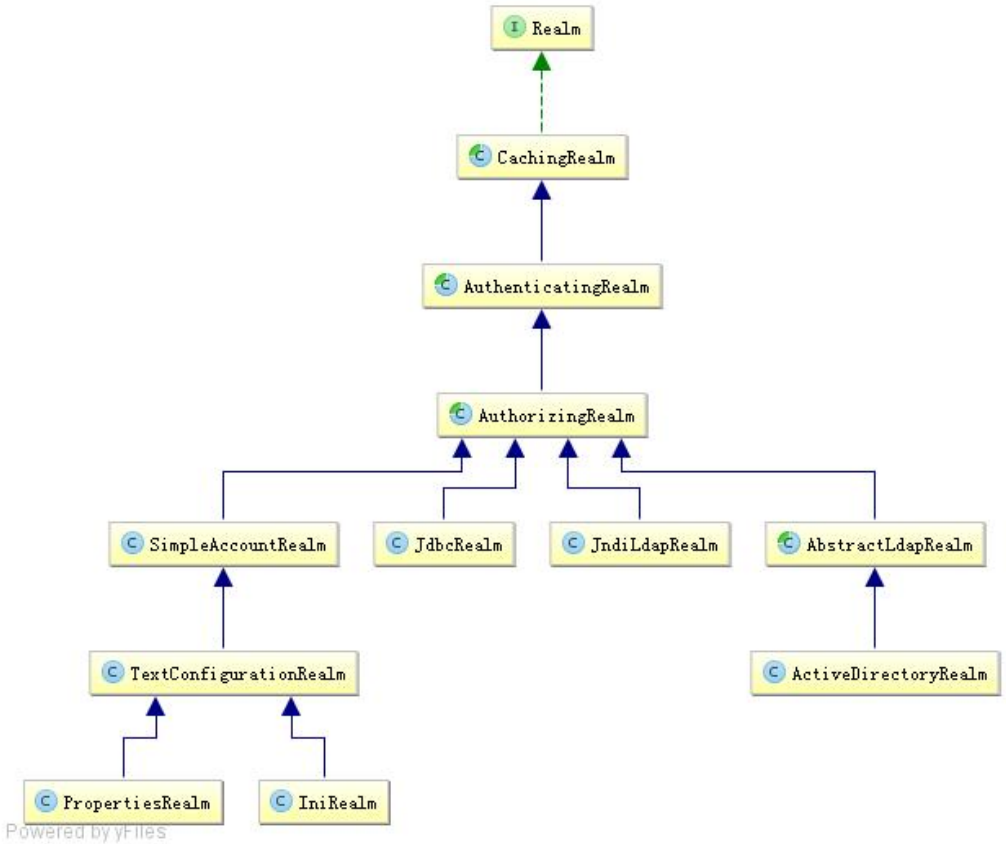
```
#声明一个 realm
myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
myRealm2=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm2
#指定 securityManager 的 realms 实现
securityManager.realms=$myRealm1,$myRealm2
```

securityManager 会按照 realms 指定的顺序进行身份认证。此处我们使用显示指定顺序的方式指定了 Realm 的顺序，如果删除“securityManager.realms=$myRealm1,$myRealm2”，那么 securityManager 会按照 realm 声明的顺序进行使用（即无需设置 realms 属性，其会自动发现），当我们显示指定 realm 后，其他没有指定 realm 将被忽略，如“securityManager.realms=$myRealm1”，那么 myRealm2 不会被自动设置进去。

2、测试用例请参考 com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest 的 testCustomMultiRealm 测试方法。

Shiro 默认提供的 Realm

http://jinnianshilongnian.iteye.com/



以后一般继承 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；

JDBC Realm 使用

1、数据库及依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.25</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>0.2.23</version>
</dependency>
```

本文将使用 mysql 数据库及 druid 连接池；

2、到数据库 shiro 下建三张表：users（用户名/密码）、user_roles（用户/角色）、roles_permissions（角色/权限），具体请参照 shiro-example-chapter2/sql/shiro.sql；并添加一个用户记录，用户名/密码为 zhang/123；

3、ini 配置（shiro-jdbc-realm.ini）

```
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
dataSource=com.alibaba.druid.pool.DruidDataSource
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/shiro
dataSource.username=root
#dataSource.password=
jdbcRealm.dataSource=$dataSource
securityManager.realms=$jdbcRealm
```

1、变量名=全限定类名 会自动创建一个类实例
2、变量名.属性=值 自动调用相应的 setter 方法进行赋值
3、$变量名 引用之前的一个对象实例

4、测试代码请参照 com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest 的 testJDBCRealm 方法，和之前的没什么区别。

Authenticator 及 AuthenticationStrategy

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。

假设我们有三个 realm：

myRealm1： 用户名/密码为 zhang/123 时成功，且返回身份/凭据为 zhang/123；

myRealm2： 用户名/密码为 wang/123 时成功，且返回身份/凭据为 wang/123；

myRealm3： 用户名/密码为 zhang/123 时成功，且返回身份/凭据为 zhang@163.com/123，和 myRealm1 不同的是返回时的身份变了；

1、ini 配置文件(shiro-authenticator-all-success.ini)

```
#指定 securityManager 的 authenticator 实现
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
securityManager.authenticator=$authenticator

#指定 securityManager.authenticator 的 authenticationStrategy
allSuccessfulStrategy=org.apache.shiro.authc.pam.AllSuccessfulStrategy
securityManager.authenticator.authenticationStrategy=$allSuccessfulStrategy
```

```
myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
myRealm2=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm2
myRealm3=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm3
securityManager.realms=$myRealm1,$myRealm3
```

2、测试代码（com.github.zhangkaitao.shiro.chapter2.AuthenticatorTest）

2.1、首先通用化登录逻辑

```
    private void login(String configFile) {
        //1、获取 SecurityManager 工厂，此处使用 Ini 配置文件初始化 SecurityManager
        Factory<org.apache.shiro.mgt.SecurityManager> factory =
                new IniSecurityManagerFactory(configFile);

        //2、得到 SecurityManager 实例 并绑定给 SecurityUtils
        org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
        SecurityUtils.setSecurityManager(securityManager);

        //3、得到 Subject 及创建用户名/密码身份验证 Token（即用户身份/凭证）
        Subject subject = SecurityUtils.getSubject();
        UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");

        subject.login(token);
    }
```

2.2、测试 AllSuccessfulStrategy 成功：

```
    @Test
    public void testAllSuccessfulStrategyWithSuccess() {
        login("classpath:shiro-authenticator-all-success.ini");
        Subject subject = SecurityUtils.getSubject();

        //得到一个身份集合，其包含了 Realm 验证成功的身份信息
        PrincipalCollection principalCollection = subject.getPrincipals();
        Assert.assertEquals(2, principalCollection.asList().size());
    }
```

即 PrincipalCollection 包含了 zhang 和 zhang@163.com 身份信息。

2.3、测试 AllSuccessfulStrategy 失败：

```
    @Test(expected = UnknownAccountException.class)
    public void testAllSuccessfulStrategyWithFail() {
        login("classpath:shiro-authenticator-all-fail.ini");
        Subject subject = SecurityUtils.getSubject();
}
```

shiro-authenticator-all-fail.ini 与 shiro-authenticator-all-success.ini 不同的配置是使用了 securityManager.realms=$myRealm1,$myRealm2；即 myRealm 验证失败。

对于 AtLeastOneSuccessfulStrategy 和 FirstSuccessfulStrategy 的区别，请参照 testAtLeastOneSuccessfulStrategyWithSuccess 和 testFirstOneSuccessfulStrategyWithSuccess 测试方法。唯一不同点一个是返回所有验证成功的 Realm 的认证信息；另一个是只返回第一个验证成功的 Realm 的认证信息。

自定义 AuthenticationStrategy 实现，首先看其 API：

```
//在所有 Realm 验证之前调用
AuthenticationInfo beforeAllAttempts(
        Collection<? extends Realm> realms, AuthenticationToken token)
        throws AuthenticationException;
//在每个 Realm 之前调用
AuthenticationInfo beforeAttempt(
        Realm realm, AuthenticationToken token, AuthenticationInfo aggregate)
        throws AuthenticationException;
//在每个 Realm 之后调用
AuthenticationInfo afterAttempt(
        Realm realm, AuthenticationToken token,
        AuthenticationInfo singleRealmInfo, AuthenticationInfo aggregateInfo, Throwable t)
        throws AuthenticationException;
//在所有 Realm 之后调用
AuthenticationInfo afterAllAttempts(
        AuthenticationToken token, AuthenticationInfo aggregate)
        throws AuthenticationException;
```

因为每个 AuthenticationStrategy 实例都是无状态的，所有每次都通过接口将相应的认证信息传入下一次流程；通过如上接口可以进行如合并/返回第一个验证成功的认证信息。

自定义实现时一般继承 org.apache.shiro.authc.pam.AbstractAuthenticationStrategy 即可，具体可以参考代码 com.github.zhangkaitao.shiro.chapter2.authenticator.strategy 包下 OnlyOneAuthenticatorStrategy 和 AtLeastTwoAuthenticatorStrategy。

到此基本的身份验证就搞定了，对于 AuthenticationToken 、AuthenticationInfo 和 Realm 的详细使用后续章节再陆续介绍。

第三章　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。。。

如上可以看出，权限代表了用户有没有操作某个资源的权利，即反映在某个资源上的操作允不允许，不反映谁去执行这个操作。所以后续还需要把权限赋予给用户，即定义哪个用户允许在某个资源上做什么操作（权限），Shiro 不会去做这件事情，而是由实现人员提供。

Shiro 支持粗粒度权限（如用户模块的所有权限）和细粒度权限（操作某个用户的权限，即实例级别的），后续部分介绍。

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。

隐式角色：即直接通过角色来验证用户有没有操作权限，如在应用中 CTO、技术总监、开发工程师可以使用打印机，假设某天不允许开发工程师使用打印机，此时需要从应用中删除相应代码；再如在应用中 CTO、技术总监可以查看用户、查看权限；突然有一天不允许技术总监查看用户、查看权限了，需要在相关代码中把技术总监角色从判断逻辑中删除掉；即粒度是以角色为单位进行访问控制的，粒度较粗；如果进行修改可能造成多处代码修改。

显示角色：在程序中通过权限控制谁能访问某个资源，角色聚合一组权限集合；这样假设哪个角色不能访问某个资源，只需要从角色代表的权限集合中移除即可；无须修改多处代码；即粒度是以资源/实例为单位的；粒度较细。

http://jinnianshilongnian.iteye.com/

请 google 搜索“RBAC”和“RBAC 新解”分别了解“基于角色的访问控制”“基于资源的访问控制(Resource-Based Access Control)”。

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>
```

后续部分将详细介绍如何使用。

授权

基于角色的访问控制（隐式角色）

1、在 ini 配置文件配置用户拥有的角色（shiro-role.ini）

```
[users]
zhang=123,role1,role2
wang=123,role1
```

规则即：“用户名=密码,角色 1，角色 2”，如果需要在应用中判断用户是否有相应角色，就需要在相应的 Realm 中返回角色信息，也就是说 Shiro 不负责维护用户-角色信息，需要应用提供，Shiro 只是提供相应的接口方便验证，后续会介绍如何动态的获取用户角色。

2、测试用例（com.github.zhangkaitao.shiro.chapter3.RoleTest）

```
@Test
public void testHasRole() {
    login("classpath:shiro-role.ini", "zhang", "123");
    //判断拥有角色：role1
    Assert.assertTrue(subject().hasRole("role1"));
    //判断拥有角色：role1 and role2
    Assert.assertTrue(subject().hasAllRoles(Arrays.asList("role1", "role2")));
    //判断拥有角色：role1 and role2 and !role3
    boolean[] result = subject().hasRoles(Arrays.asList("role1", "role2", "role3"));
    Assert.assertEquals(true, result[0]);
    Assert.assertEquals(true, result[1]);
    Assert.assertEquals(false, result[2]);
}
```

Shiro 提供了 hasRole/hasRole 用于判断用户是否拥有某个角色/某些权限；但是没有提供如 hashAnyRole 用于判断是否有某些权限中的某一个。

```
@Test(expected = UnauthorizedException.class)
public void testCheckRole() {
    login("classpath:shiro-role.ini", "zhang", "123");
    //断言拥有角色：role1
    subject().checkRole("role1");
    //断言拥有角色：role1 and role3 失败抛出异常
    subject().checkRoles("role1", "role3");
}
```

Shiro 提供的 checkRole/checkRoles 和 hasRole/hasAllRoles 不同的地方是它在判断为假的情况下会抛出 UnauthorizedException 异常。

到此基于角色的访问控制（即隐式角色）就完成了，这种方式的缺点就是如果很多地方进行了角色判断，但是有一天不需要了那么就需要修改相应代码把所有相关的地方进行删除；这就是粗粒度造成的问题。

基于资源的访问控制（显示角色）

1、在 ini 配置文件配置用户拥有的角色及角色-权限关系（shiro-permission.ini）

```
[users]
zhang=123,role1,role2
wang=123,role1
[roles]
role1=user:create,user:update
role2=user:create,user:delete
```

规则：“用户名=密码，角色 1，角色 2” “角色=权限 1，权限 2”，即首先根据用户名找到角色，然后根据角色再找到权限；即角色是权限集合；Shiro 同样不进行权限的维护，需要我们通过 Realm 返回相应的权限信息。只需要维护“用户——角色”之间的关系即可。

2、测试用例（com.github.zhangkaitao.shiro.chapter3.PermissionTest）

```
@Test
public void testIsPermitted() {
    login("classpath:shiro-permission.ini", "zhang", "123");
    //判断拥有权限：user:create
    Assert.assertTrue(subject().isPermitted("user:create"));
    //判断拥有权限：user:update and user:delete
    Assert.assertTrue(subject().isPermittedAll("user:update", "user:delete"));
    //判断没有权限：user:view
    Assert.assertFalse(subject().isPermitted("user:view"));
}
```

Shiro 提供了 isPermitted 和 isPermittedAll 用于判断用户是否拥有某个权限或所有权限，也没有提供如 isPermittedAny 用于判断拥有某一个权限的接口。

```
@Test(expected = UnauthorizedException.class)
public void testCheckPermission () {
    login("classpath:shiro-permission.ini", "zhang", "123");
    //断言拥有权限：user:create
    subject().checkPermission("user:create");
    //断言拥有权限：user:delete and user:update
    subject().checkPermissions("user:delete", "user:update");
    //断言拥有权限：user:view 失败抛出异常
    subject().checkPermissions("user:view");
}
```

限，但是失败的情况下会抛出 UnauthorizedException 异常。

到此基于资源的访问控制（显示角色）就完成了，也可以叫基于权限的访问控制，这种方

式的一般规则是“资源标识符：操作”，即是资源级别的粒度；这种方式的好处就是如果要修改基本都是一个资源级别的修改，不会对其他模块代码产生影响，粒度小。但是实现起来可能稍微复杂点，需要维护“用户——角色，角色——权限（资源：操作）”之间的关系。

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

ini 配置文件

```
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：
ini 配置（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，但是反过来是规则不成立。

3、单个资源全部权限

ini 配置

```
role51="system:user:create,update,delete,view"
```

然后通过如下代码判断

```
subject().checkPermissions("system:user:create,delete,update:view");
```

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：

ini 配置文件（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

ini 配置

```
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。

5、实例级别的权限

5.1、单个实例单个权限

ini 配置

```
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

ini 配置

```
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

ini 配置

```
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

ini 配置

```
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");
```

5.5、所有实例所有权限

ini 配置

```
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"));
```

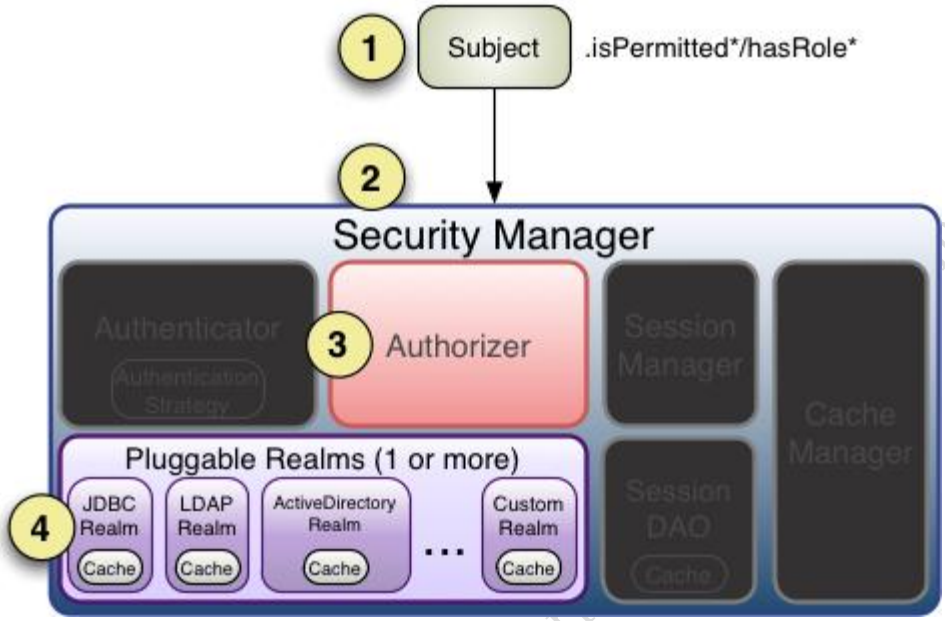
因此没什么必要的话使用字符串更方便。

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。

http://jinnianshilongnian.iteye.com/

授权流程



流程如下：

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。

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。

Authorizer、PermissionResolver 及 RolePermissionResolver

Authorizer 的职责是进行授权（访问控制），是 Shiro API 中授权核心的入口点，其提供了相应的角色/权限判断接口，具体请参考其 Javadoc。SecurityManager 继承了 Authorizer 接口，且提供了 ModularRealmAuthorizer 用于多 Realm 时的授权匹配。PermissionResolver 用于解析权限字符串到 Permission 实例，而 RolePermissionResolver 用于根据角色解析相应的权限集合。

我们可以通过如下 ini 配置更改 Authorizer 实现：

```
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
securityManager.authorizer=$authorizer
```

对于 ModularRealmAuthorizer，相应的 AuthorizingSecurityManager 会在初始化完成后自动将相应的 realm 设置进去，我们也可以通过调用其 setRealms()方法进行设置。对于实现自己的 authorizer 可以参考 ModularRealmAuthorizer 实现即可，在此就不提供示例了。

设置 ModularRealmAuthorizer 的 permissionResolver，其会自动设置到相应的 Realm 上（其实现了 PermissionResolverAware 接口），如：

```
permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
authorizer.permissionResolver=$permissionResolver
```

设置 ModularRealmAuthorizer 的 rolePermissionResolver，其会自动设置到相应的 Realm 上（其实现了 RolePermissionResolverAware 接口），如：

```
rolePermissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.MyRolePermission
Resolver
authorizer.rolePermissionResolver=$rolePermissionResolver
```

示例

1、ini 配置（shiro-authorizer.ini）

```
[main]
#自定义 authorizer
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
#自定义 permissionResolver
#permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
permissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.BitAndWildPermission
Resolver
authorizer.permissionResolver=$permissionResolver
#自定义 rolePermissionResolver
rolePermissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.MyRolePermission
Resolver
authorizer.rolePermissionResolver=$rolePermissionResolver

securityManager.authorizer=$authorizer
```

```
#自定义 realm 一定要放在 securityManager.authorizer 赋值之后（因为调用 setRealms 会将
realms 设置给 authorizer，并给各个 Realm 设置 permissionResolver 和 rolePermissionResolver）
realm=com.github.zhangkaitao.shiro.chapter3.realm.MyRealm
securityManager.realms=$realm
```

设置 securityManager 的 realms 一定要放到最后，因为在调用 SecurityManager.setRealms 时会将 realms 设置给 authorizer，并为各个 Realm 设置 permissionResolver 和 rolePermissionResolver。另外，不能使用 IniSecurityManagerFactory 创建的 IniRealm，因为其初始化顺序的问题可能造成后续的初始化 Permission 造成影响。

2、定义 BitAndWildPermissionResolver 及 BitPermission

BitPermission 用于实现位移方式的权限，如规则是：

权限字符串格式：+资源字符串+权限位+实例 ID；以+开头 中间通过+分割；权限：0 表示所有权限；1 新增（二进制：0001）、2 修改（二进制：0010）、4 删除（二进制：0100）、8 查看（二进制：1000）；如 +user+10 表示对资源 user 拥有修改/查看权限。

```
public class BitPermission implements Permission {
    private String resourceIdentify;
    private int permissionBit;
    private String instanceId;
```

```
    public BitPermission(String permissionString) {
        String[] array = permissionString.split("\\+");
        if(array.length > 1) {
            resourceIdentify = array[1];
        }
        if(StringUtils.isEmpty(resourceIdentify)) {
            resourceIdentify = "*";
        }
        if(array.length > 2) {
            permissionBit = Integer.valueOf(array[2]);
        }
        if(array.length > 3) {
            instanceId = array[3];
        }
        if(StringUtils.isEmpty(instanceId)) {
            instanceId = "*";
        }
    }

    @Override
    public boolean implies(Permission p) {
        if(!(p instanceof BitPermission)) {
            return false;
        }
        BitPermission other = (BitPermission) p;
        if(!("*".equals(this.resourceIdentify)                                    ||
this.resourceIdentify.equals(other.resourceIdentify))) {
            return false;
        }
        if(!(this.permissionBit ==0 || (this.permissionBit & other.permissionBit) != 0)) {
            return false;
        }
        if(!("*".equals(this.instanceId) || this.instanceId.equals(other.instanceId))) {
            return false;
        }
        return true;
    }
}
```

Permission 接口提供了 boolean implies(Permission p)方法用于判断权限匹配的；

```
public class BitAndWildPermissionResolver implements PermissionResolver {
    @Override
    public Permission resolvePermission(String permissionString) {
        if(permissionString.startsWith("+")) {
            return new BitPermission(permissionString);
        }
        return new WildcardPermission(permissionString);
    }
}
```

BitAndWildPermissionResolver 实现了 PermissionResolver 接口，并根据权限字符串是否以“+”开头来解析权限字符串为 BitPermission 或 WildcardPermission。

3、定义 MyRolePermissionResolver

RolePermissionResolver 用于根据角色字符串来解析得到权限集合。

```
public class MyRolePermissionResolver implements RolePermissionResolver {
    @Override
    public Collection<Permission> resolvePermissionsInRole(String roleString) {
        if("role1".equals(roleString)) {
            return Arrays.asList((Permission)new WildcardPermission("menu:*"));
        }
        return null;
    }
}
```

此处的实现很简单，如果用户拥有 role1，那么就返回一个“menu:*”的权限。

4、自定义 Realm

```
public class MyRealm extends AuthorizingRealm {
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
        authorizationInfo.addRole("role1");
        authorizationInfo.addRole("role2");
        authorizationInfo.addObjectPermission(new BitPermission("+user1+10"));
        authorizationInfo.addObjectPermission(new WildcardPermission("user1:*"));
```

```
        authorizationInfo.addStringPermission("+user2+10");
        authorizationInfo.addStringPermission("user2:*");
        return authorizationInfo;
    }
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        //和 com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1. getAuthenticationInfo
代码一样，省略
    }
}
```

此时我们继承 AuthorizingRealm 而不是实现 Realm 接口；推荐使用 AuthorizingRealm，因为：
AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)：表示获取身份验证信息；
AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)：表示根据用户身份获取授权信息。
这种方式的好处是当只需要身份验证时只需要获取身份验证信息而不需要获取授权信息。
对于 AuthenticationInfo 和 AuthorizationInfo 请参考其 Javadoc 获取相关接口信息。

另外我们可以使用 JdbcRealm，需要做的操作如下：
1、执行 sql/ shiro-init-data.sql 插入相关的权限数据；
2、使用 shiro-jdbc-authorizer.ini 配置文件，需要设置 jdbcRealm.permissionsLookupEnabled 为 true 来开启权限查询。

此次还要注意就是不能把我们自定义的如“+user1+10”配置到 INI 配置文件，即使有 IniRealm 完成，因为 IniRealm 在 new 完成后就会解析这些权限字符串，默认使用了 WildcardPermissionResolver 完成，即此处是一个设计权限，如果采用生命周期（如使用初始化方法）的方式进行加载就可以解决我们自定义 permissionResolver 的问题。

5、测试用例

```
public class AuthorizerTest extends BaseTest {

    @Test
    public void testIsPermitted() {
        login("classpath:shiro-authorizer.ini", "zhang", "123");
        //判断拥有权限：user:create
```

```
        Assert.assertTrue(subject().isPermitted("user1:update"));
        Assert.assertTrue(subject().isPermitted("user2:update"));
        //通过二进制位的方式表示权限
        Assert.assertTrue(subject().isPermitted("+user1+2"));//新增权限
        Assert.assertTrue(subject().isPermitted("+user1+8"));//查看权限
        Assert.assertTrue(subject().isPermitted("+user2+10"));//新增及查看

        Assert.assertFalse(subject().isPermitted("+user1+4"));//没有删除权限

        Assert.assertTrue(subject().isPermitted("menu:view"));//  通  过
MyRolePermissionResolver 解析得到的权限
    }
}
```

通过如上步骤可以实现自定义权限验证了。另外因为不支持 hasAnyRole/isPermittedAny 这种方式的授权，可以参考我的一篇《简单 shiro 扩展实现 NOT、AND、OR 权限验证》进行简单的扩展完成这个需求，在这篇文章中通过重写 AuthorizingRealm 里的验证逻辑实现的。

http://jinnianshilongnian.iteye.com/

第四章　INI 配置

之前章节我们已经接触过一些 INI 配置规则了，如果大家使用过如 Spring 之类的 IoC/DI 容器的话，Shiro 提供的 INI 配置也是非常类似的，即可以理解为是一个 IoC/DI 容器，但是区别在于它从一个根对象 securityManager 开始。

根对象 SecurityManager

从之前的 Shiro 架构图可以看出，Shiro 是从根对象 SecurityManager 进行身份验证和授权的；也就是所有操作都是自它开始的，这个对象是线程安全且真个应用只需要一个即可，因此 Shiro 提供了 SecurityUtils 让我们绑定它为全局的，方便后续操作。

因为 Shiro 的类都是 POJO 的，因此都很容易放到任何 IoC 容器管理。但是和一般的 IoC 容器的区别在于，Shiro 从根对象 securityManager 开始导航；Shiro 支持的依赖注入：public 空参构造器对象的创建、setter 依赖注入。

1、纯 Java 代码写法（com.github.zhangkaitao.shiro.chapter4.NonConfigurationCreateTest）：

```
DefaultSecurityManager securityManager = new DefaultSecurityManager();
//设置 authenticator
ModularRealmAuthenticator authenticator = new ModularRealmAuthenticator();
authenticator.setAuthenticationStrategy(new AtLeastOneSuccessfulStrategy());
securityManager.setAuthenticator(authenticator);

//设置 authorizer
ModularRealmAuthorizer authorizer = new ModularRealmAuthorizer();
authorizer.setPermissionResolver(new WildcardPermissionResolver());
securityManager.setAuthorizer(authorizer);

//设置 Realm
DruidDataSource ds = new DruidDataSource();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3306/shiro");
ds.setUsername("root");
ds.setPassword("");
```

jinnianshilongnian.iteye.com/

```
JdbcRealm jdbcRealm = new JdbcRealm();
jdbcRealm.setDataSource(ds);
jdbcRealm.setPermissionsLookupEnabled(true);
securityManager.setRealms(Arrays.asList((Realm) jdbcRealm));

//将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);

Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");
subject.login(token);
Assert.assertTrue(subject.isAuthenticated());
```

2.1、等价的 INI 配置（shiro-config.ini）

```
[main]
#authenticator
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy
authenticator.authenticationStrategy=$authenticationStrategy
securityManager.authenticator=$authenticator

#authorizer
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
authorizer.permissionResolver=$permissionResolver
securityManager.authorizer=$authorizer

#realm
dataSource=com.alibaba.druid.pool.DruidDataSource
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/shiro
dataSource.username=root
#dataSource.password=
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
jdbcRealm.dataSource=$dataSource
jdbcRealm.permissionsLookupEnabled=true
securityManager.realms=$jdbcRealm
```

即使没接触过 IoC 容器的知识，如上配置也是很容易理解的：
1、对象名=全限定类名　相对于调用 public 无参构造器创建对象
2、对象名.属性名=值　　相当于调用 setter 方法设置常量值
3、对象名.属性名=$对象引用　　相当于调用 setter 方法设置对象引用

2.2、Java 代码（com.github.zhangkaitao.shiro.chapter4.ConfigurationCreateTest）

```
Factory<org.apache.shiro.mgt.SecurityManager> factory =
        new IniSecurityManagerFactory("classpath:shiro-config.ini");

org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();

//将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);
Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");
subject.login(token);

Assert.assertTrue(subject.isAuthenticated());
```

如上代码是从 Shiro INI 配置中获取相应的 securityManager 实例：
1、默认情况先创建一个名字为 securityManager，类型为 org.apache.shiro.mgt.DefaultSecurityManager 的默认的 SecurityManager，如果想自定义，只需要在 ini 配置文件中指定“securityManager=SecurityManager 实现类”即可，名字必须为 securityManager，它是起始的根；
2、IniSecurityManagerFactory 是创建 securityManager 的工厂，其需要一个 ini 配置文件路径，其支持“classpath:”（类路径）、“file:”（文件系统）、“url:”（网络）三种路径格式，默认是文件系统；
3、接着获取 SecuriyManager 实例，后续步骤和之前的一样。

从如上可以看出 Shiro INI 配置方式本身提供了一个简单的 IoC/DI 机制方便在配置文件配置，但是是从 securityManager 这个根对象开始导航。

INI 配置

ini 配置文件类似于 Java 中的 properties（key=value），不过提供了将 key/value 分类的特性，key 是每个部分不重复即可，而不是整个配置文件。如下是 INI 配置分类：

http://jinnianshilongnian.iteye.com/

```
[main]
#提供了对根对象 securityManager 及其依赖的配置
securityManager=org.apache.shiro.mgt.DefaultSecurityManager
……
securityManager.realms=$jdbcRealm

[users]
#提供了对用户/密码及其角色的配置，用户名=密码，角色 1，角色 2
username=password,role1,role2

[roles]
#提供了角色及权限之间关系的配置，角色=权限 1，权限 2
role1=permission1,permission2

[urls]
#用于 web，提供了对 web url 拦截相关的配置，url=拦截器[参数]，拦截器
/index.html = anon
/admin/** = authc, roles[admin], perms["permission1"]
```

[main]部分

提供了对根对象 securityManager 及其依赖对象的配置。

创建对象

```
securityManager=org.apache.shiro.mgt.DefaultSecurityManager
```

其构造器必须是 public 空参构造器，通过反射创建相应的实例。

常量值 setter 注入

```
dataSource.driverClassName=com.mysql.jdbc.Driver
jdbcRealm.permissionsLookupEnabled=true
```

会自动调用 jdbcRealm.setPermissionsLookupEnabled(true)，对于这种常量值会自动类型转换。

对象引用 setter 注入

```
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy
authenticator.authenticationStrategy=$authenticationStrategy
securityManager.authenticator=$authenticator
```

会自动通过 securityManager.setAuthenticator(authenticator)注入引用依赖。

嵌套属性 setter 注入

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

也支持这种嵌套方式的 setter 注入。

byte 数组 setter 注入

```
#base64 byte[]
authenticator.bytes=aGVsbG8=
#hex byte[]
authenticator.bytes=0x68656c6c6f
```

默认需要使用 Base64 进行编码，也可以使用 0x 十六进制。

Array/Set/List setter 注入

```
authenticator.array=1,2,3
authenticator.set=$jdbcRealm,$jdbcRealm
```

多个之间通过“，”分割。

Map setter 注入

```
authenticator.map=$jdbcRealm:$jdbcRealm,1:1,key:abc
```

即格式是：map=key：value，key：value，可以注入常量及引用值，常量的话都看作字符串（即使有泛型也不会自动造型）。

实例化/注入顺序

```
realm=Realm1
realm=Realm12
```

```
authenticator.bytes=aGVsbG8=
authenticator.bytes=0x68656c6c6f
```

后边的覆盖前边的注入。

测试用例请参考配置文件 shiro-config-main.ini。

[users]部分

配置用户名/密码及其角色，格式：“用户名=密码，角色 1，角色 2”，角色部分可省略。如：

```
[users]
zhang=123,role1,role2
wang=123
```

密码一般生成其摘要/加密存储，后续章节介绍。

[roles]部分

配置角色及权限之间的关系，格式：“角色=权限 1，权限 2”；如：

```
[roles]
role1=user:create,user:update
role2=*
```

如果只有角色没有对应的权限，可以不配 roles，具体规则请参考授权章节。

[urls]部分

配置 url 及相应的拦截器之间的关系，格式：“url=拦截器[参数]，拦截器[参数]，如：

```
[urls]
/admin/** = authc, roles[admin], perms["permission1"]
```

具体规则参见 web 相关章节。

http://jinnianshilongnian.iteye.com/

第五章　编码/加密

在涉及到密码存储问题上，应该加密/生成密码摘要存储，而不是存储明文密码。比如之前的 600w csdn 账号泄露对用户可能造成很大损失，因此应加密/生成不可逆的摘要方式存储。

编码/解码

Shiro 提供了 base64 和 16 进制字符串编码/解码的 API 支持，方便一些编码解码操作。Shiro 内部的一些数据的存储/表示都使用了 base64 和 16 进制字符串。

```
String str = "hello";
String base64Encoded = Base64.encodeToString(str.getBytes());
String str2 = Base64.decodeToString(base64Encoded);
Assert.assertEquals(str, str2);
```

通过如上方式可以进行 base64 编码/解码操作，更多 API 请参考其 Javadoc。

```
String str = "hello";
String base64Encoded = Hex.encodeToString(str.getBytes());
String str2 = new String(Hex.decode(base64Encoded.getBytes()));
Assert.assertEquals(str, str2);
```

通过如上方式可以进行 16 进制字符串编码/解码操作，更多 API 请参考其 Javadoc。

还有一个可能经常用到的类 CodecSupport，提供了 toBytes(str, "utf-8") / toString(bytes, "utf-8")用于在 byte 数组/String 之间转换。

散列算法

散列算法一般用于生成数据的摘要信息，是一种不可逆的算法，一般适合存储密码之类的数据，常见的散列算法如 MD5、SHA 等。一般进行散列时最好提供一个 salt（盐），比如加密密码“admin”，产生的散列值是“21232f297a57a5a743894a0e4a801fc3”，可以到一些 md5 解密网站很容易的通过散列值得到密码“admin”，即如果直接对密码进行散列相对来说破解更容易，此时我们可以加一些只有系统知道的干扰数据，如用户名和 ID（即盐）；这样散列的对象是“密码+用户名+ID”，这样生成的散列值相对来说更难破解。

```
String str = "hello";
String salt = "123";
String md5 = new Md5Hash(str, salt).toString();//还可以转换为 toBase64()/toHex()
```

如上代码通过盐“123”MD5 散列“hello”。另外散列时还可以指定散列次数，如 2 次表示：md5(md5(str))：“new Md5Hash(str, salt, 2).toString()”。

```
String str = "hello";
String salt = "123";
String sha1 = new Sha256Hash(str, salt).toString();
```

使用 SHA256 算法生成相应的散列数据，另外还有如 SHA1、SHA512 算法。

Shiro 还提供了通用的散列支持：

```
String str = "hello";
String salt = "123";
//内部使用 MessageDigest
String simpleHash = new SimpleHash("SHA-1", str, salt).toString();
```

通过调用 SimpleHash 时指定散列算法，其内部使用了 Java 的 MessageDigest 实现。

为了方便使用，Shiro 提供了 HashService，默认提供了 DefaultHashService 实现。

```
DefaultHashService hashService = new DefaultHashService(); //默认算法 SHA-512
hashService.setHashAlgorithmName("SHA-512");
hashService.setPrivateSalt(new SimpleByteSource("123")); //私盐，默认无
hashService.setGeneratePublicSalt(true);//是否生成公盐，默认 false
hashService.setRandomNumberGenerator(new SecureRandomNumberGenerator());//用于生成公盐。默认就这个
hashService.setHashIterations(1); //生成 Hash 值的迭代次数

HashRequest request = new HashRequest.Builder()
                .setAlgorithmName("MD5").setSource(ByteSource.Util.bytes("hello"))
                .setSalt(ByteSource.Util.bytes("123")).setIterations(2).build();
String hex = hashService.computeHash(request).toHex();
```

1、首先创建一个 DefaultHashService，默认使用 SHA-512 算法；
2、可以通过 hashAlgorithmName 属性修改算法；
3、可以通过 privateSalt 设置一个私盐，其在散列时自动与用户传入的公盐混合产生一个新盐；

4、可以通过 generatePublicSalt 属性在用户没有传入公盐的情况下是否生成公盐；
5、可以设置 randomNumberGenerator 用于生成公盐；
6、可以设置 hashIterations 属性来修改默认加密迭代次数；
7、需要构建一个 HashRequest，传入算法、数据、公盐、迭代次数。

SecureRandomNumberGenerator 用于生成一个随机数：

```
SecureRandomNumberGenerator randomNumberGenerator =
        new SecureRandomNumberGenerator();
randomNumberGenerator.setSeed("123".getBytes());
String hex = randomNumberGenerator.nextBytes().toHex();
```

加密/解密

Shiro 还提供对称式加密/解密算法的支持，如 AES、Blowfish 等；当前还没有提供对非对称加密/解密算法支持，未来版本可能提供。

AES 算法实现：

```
AesCipherService aesCipherService = new AesCipherService();
aesCipherService.setKeySize(128); //设置 key 长度
//生成 key
Key key = aesCipherService.generateNewKey();
String text = "hello";
//加密
String encrptText =
  aesCipherService.encrypt(text.getBytes(), key.getEncoded()).toHex();
//解密
String text2 =
  new String(aesCipherService.decrypt(Hex.decode(encrptText), key.getEncoded()).getBytes());

Assert.assertEquals(text, text2);
```

更多算法请参考示例 com.github.zhangkaitao.shiro.chapter5.hash.CodecAndCryptoTest。

PasswordService/CredentialsMatcher

Shiro 提供了 PasswordService 及 CredentialsMatcher 用于提供加密密码及验证密码服务。

```
public interface PasswordService {
    //输入明文密码得到密文密码
    String encryptPassword(Object plaintextPassword) throws IllegalArgumentException;
}
```

```
public interface CredentialsMatcher {
    //匹配用户输入的 token 的凭证（未加密）与系统提供的凭证（已加密）
    boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info);
}
```

Shiro 默认提供了 PasswordService 实现 DefaultPasswordService；CredentialsMatcher 实现 PasswordMatcher 及 HashedCredentialsMatcher（更强大）。

DefaultPasswordService 配合 PasswordMatcher 实现简单的密码加密与验证服务

1、定义 Realm（com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm）

```
public class MyRealm extends AuthorizingRealm {
    private PasswordService passwordService;
    public void setPasswordService(PasswordService passwordService) {
        this.passwordService = passwordService;
    }
     //省略 doGetAuthorizationInfo，具体看代码
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        return new SimpleAuthenticationInfo(
                "wu",
                passwordService.encryptPassword("123"),
                getName());
    }
}
```

为了方便，直接注入一个 passwordService 来加密密码，实际使用时需要在 Service 层使用 passwordService 加密密码并存到数据库。

2、ini 配置（shiro-passwordservice.ini）

```
[main]
passwordService=org.apache.shiro.authc.credential.DefaultPasswordService
hashService=org.apache.shiro.crypto.hash.DefaultHashService
passwordService.hashService=$hashService
hashFormat=org.apache.shiro.crypto.hash.format.Shiro1CryptFormat
passwordService.hashFormat=$hashFormat
hashFormatFactory=org.apache.shiro.crypto.hash.format.DefaultHashFormatFactory
passwordService.hashFormatFactory=$hashFormatFactory

passwordMatcher=org.apache.shiro.authc.credential.PasswordMatcher
passwordMatcher.passwordService=$passwordService

myRealm=com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm
myRealm.passwordService=$passwordService
myRealm.credentialsMatcher=$passwordMatcher
securityManager.realms=$myRealm
```

2.1、passwordService 使用 DefaultPasswordService，如果有必要也可以自定义；
2.2、hashService 定义散列密码使用的 HashService，默认使用 DefaultHashService（默认 SHA-256 算法）；
2.3、hashFormat 用于对散列出的值进行格式化，默认使用 Shiro1CryptFormat，另外提供了 Base64Format 和 HexFormat，对于有 salt 的密码请自定义实现 ParsableHashFormat 然后把 salt 格式化到散列值中；
2.4、hashFormatFactory 用于根据散列值得到散列的密码和 salt；因为如果使用如 SHA 算法，那么会生成一个 salt，此 salt 需要保存到散列后的值中以便之后与传入的密码比较时使用；默认使用 DefaultHashFormatFactory；
2.5、passwordMatcher 使用 PasswordMatcher，其是一个 CredentialsMatcher 实现；
2.6、将 credentialsMatcher 赋值给 myRealm，myRealm 间接继承了 AuthenticatingRealm，其在调用 getAuthenticationInfo 方法获取到 AuthenticationInfo 信息后，会使用 credentialsMatcher 来验证凭据是否匹配，如果不匹配将抛出 IncorrectCredentialsException 异常。

3、测试用例请参考 com.github.zhangkaitao.shiro.chapter5.hash.PasswordTest。

另外可以参考配置 shiro-jdbc-passwordservice.ini，提供了 JdbcRealm 的测试用例，测试前请先调用 sql/shiro-init-data.sql 初始化用户数据。

如上方式的缺点是：salt 保存在散列值中；没有实现如密码重试次数限制。

HashedCredentialsMatcher 实现密码验证服务

Shiro 提供了 CredentialsMatcher 的散列实现 HashedCredentialsMatcher，和之前的 PasswordMatcher 不同的是，它只用于密码验证，且可以提供自己的盐，而不是随机生成盐，且生成密码散列值的算法需要自己写，因为能提供自己的盐。

1、生成密码散列值

此处我们使用 MD5 算法，“密码+盐（用户名+随机数）”的方式生成散列值：

```
String algorithmName = "md5";
String username = "liu";
String password = "123";
String salt1 = username;
String salt2 = new SecureRandomNumberGenerator().nextBytes().toHex();
int hashIterations = 2;

SimpleHash hash = new SimpleHash(algorithmName, password, salt1 + salt2, hashIterations);
```

如果要写用户模块，需要在新增用户/重置密码时使用如上算法保存密码，将生成的密码及 salt2 存入数据库（因为我们的散列算法是：md5(md5(密码+username+salt2))）。

2、生成 Realm（com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm2）

```
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
    String username = "liu"; //用户名及 salt1
    String password = "202cb962ac59075b964b07152d234b70"; //加密后的密码
    String salt2 = "202cb962ac59075b964b07152d234b70";
    SimpleAuthenticationInfo ai =
        new SimpleAuthenticationInfo(username, password, getName());
    ai.setCredentialsSalt(ByteSource.Util.bytes(username+salt2)); //盐是用户名+随机数
        return ai;
}
```

此处就是把步骤 1 中生成的相应数据组装为 SimpleAuthenticationInfo，通过 SimpleAuthenticationInfo 的 credentialsSalt 设置盐，HashedCredentialsMatcher 会自动识别这个盐。

如果使用 JdbcRealm，需要修改获取用户信息（包括盐）的 sql：“select password, password_salt from users where username = ?”，而我们的盐是由 username+password_salt 组成，所以需要通过如下 ini 配置（shiro-jdbc-hashedCredentialsMatcher.ini）修改：

```
jdbcRealm.saltStyle=COLUMN
jdbcRealm.authenticationQuery=select password, concat(username,password_salt) from users where username = ?
jdbcRealm.credentialsMatcher=$credentialsMatcher
```

1、saltStyle 表示使用密码+盐的机制，authenticationQuery 第一列是密码，第二列是盐；
2、通过 authenticationQuery 指定密码及盐查询 SQL；

此处还要注意 Shiro 默认使用了 apache commons BeanUtils，默认是不进行 Enum 类型转型的，此时需要自己注册一个 Enum 转换器“BeanUtilsBean.getInstance().getConvertUtils().register(new EnumConverter(), JdbcRealm.SaltStyle.class);”具体请参考示例“com.github.zhangkaitao.shiro.chapter5.hash.PasswordTest”中的代码。

另外可以参考配置 shiro-jdbc-passwordservice.ini，提供了 JdbcRealm 的测试用例，测试前请先调用 sql/shiro-init-data.sql 初始化用户数据。

3、ini 配置（shiro-hashedCredentialsMatcher.ini）

```
[main]
credentialsMatcher=org.apache.shiro.authc.credential.HashedCredentialsMatcher
credentialsMatcher.hashAlgorithmName=md5
credentialsMatcher.hashIterations=2
credentialsMatcher.storedCredentialsHexEncoded=true
myRealm=com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm2
myRealm.credentialsMatcher=$credentialsMatcher
securityManager.realms=$myRealm
```

1、通过 credentialsMatcher.hashAlgorithmName=md5 指定散列算法为 md5，需要和生成密码时的一样；
2、credentialsMatcher.hashIterations=2，散列迭代次数，需要和生成密码时的意义；
3、credentialsMatcher.storedCredentialsHexEncoded=true 表示是否存储散列后的密码为 16 进制，需要和生成密码时的一样，默认是 base64；

此处最需要注意的就是 HashedCredentialsMatcher 的算法需要和生成密码时的算法一样。另外 HashedCredentialsMatcher 会自动根据 AuthenticationInfo 的类型是否是 SaltedAuthenticationInfo 来获取 credentialsSalt 盐。

4、测试用例请参考 com.github.zhangkaitao.shiro.chapter5.hash.PasswordTest。

密码重试次数限制

如在 1 个小时内密码最多重试 5 次，如果尝试次数超过 5 次就锁定 1 小时，1 小时后可再次重试，如果还是重试失败，可以锁定如 1 天，以此类推，防止密码被暴力破解。我们通过继承 HashedCredentialsMatcher，且使用 Ehcache 记录重试次数和超时时间。

com.github.zhangkaitao.shiro.chapter5.hash.credentials.RetryLimitHashedCredentialsMatcher：

```
public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
        String username = (String)token.getPrincipal();
        //retry count + 1
        Element element = passwordRetryCache.get(username);
        if(element == null) {
            element = new Element(username , new AtomicInteger(0));
            passwordRetryCache.put(element);
        }
        AtomicInteger retryCount = (AtomicInteger)element.getObjectValue();
        if(retryCount.incrementAndGet() > 5) {
            //if retry count > 5 throw
            throw new ExcessiveAttemptsException();
        }

        boolean matches = super.doCredentialsMatch(token, info);
        if(matches) {
            //clear retry count
            passwordRetryCache.remove(username);
        }
        return matches;
}
```

如上代码逻辑比较简单，即如果密码输入正确清除 cache 中的记录；否则 cache 中的重试次数+1，如果超出 5 次那么抛出异常表示超出重试次数了。

第六章　Realm 及相关对象

Realm

【2.5 Realm】及【3.5 Authorizer】部分都已经详细介绍过 Realm 了，接下来再来看一下一般真实环境下的 Realm 如何实现。

1、定义实体及关系

即用户-角色之间是多对多关系，角色-权限之间是多对多关系；且用户和权限之间通过角色建立关系；在系统中验证时通过权限验证，角色只是权限集合，即所谓的显示角色；其实权限应该对应到资源（如菜单、URL、页面按钮、Java 方法等）中，即应该将权限字符串存储到资源实体中，但是目前为了简单化，直接提取一个权限表，【综合示例】部分会使用完整的表结构。

用户实体包括：编号(id)、用户名(username)、密码(password)、盐(salt)、是否锁定(locked)；是否锁定用于封禁用户使用，其实最好使用 Enum 字段存储，可以实现更复杂的用户状态实现。
角色实体包括：、编号(id)、角色标识符（role）、描述（description）、是否可用（available）；其中角色标识符用于在程序中进行隐式角色判断的，描述用于以后再前台界面显示的、是否可用表示角色当前是否激活。
权限实体包括：编号（id）、权限标识符（permission）、描述（description）、是否可用（available）；含义和角色实体类似不再阐述。

另外还有两个关系实体：用户-角色实体（用户编号、角色编号，且组合为复合主键）；角色-权限实体（角色编号、权限编号，且组合为复合主键）。

sql 及实体请参考源代码中的 sql\shiro.sql 和 com.github.zhangkaitao.shiro.chapter6.entity 对应的实体。

2、环境准备

为了方便数据库操作，使用了“org.springframework: spring-jdbc: 4.0.0.RELEASE”依赖，虽然是 spring4 版本的，但使用上和 spring3 无区别。其他依赖请参考源码的 pom.xml。

http://jinnianshilongnian.iteye.com/

3、定义 Service 及 Dao

为了实现的简单性，只实现必须的功能，其他的可以自己实现即可。

PermissionService

```
public interface PermissionService {
    public Permission createPermission(Permission permission);
    public void deletePermission(Long permissionId);
}
```

实现基本的创建/删除权限。

RoleService

```
public interface RoleService {
    public Role createRole(Role role);
    public void deleteRole(Long roleId);
    //添加角色-权限之间关系
    public void correlationPermissions(Long roleId, Long... permissionIds);
    //移除角色-权限之间关系
    public void uncorrelationPermissions(Long roleId, Long... permissionIds);//
}
```

相对于 PermissionService 多了关联/移除关联角色-权限功能。

UserService

```
public interface UserService {
    public User createUser(User user); //创建账户
    public void changePassword(Long userId, String newPassword);//修改密码
    public void correlationRoles(Long userId, Long... roleIds); //添加用户-角色关系
    public void uncorrelationRoles(Long userId, Long... roleIds);// 移除用户-角色关系
    public User findByUsername(String username);// 根据用户名查找用户
    public Set<String> findRoles(String username);// 根据用户名查找其角色
    public Set<String> findPermissions(String username); //根据用户名查找其权限
}
```

此处使用 findByUsername、findRoles 及 findPermissions 来查找用户名对应的帐号、角色及权限信息。之后的 Realm 就使用这些方法来查找相关信息。

UserServiceImpl

```
public User createUser(User user) {
    //加密密码
    passwordHelper.encryptPassword(user);
    return userDao.createUser(user);
}
public void changePassword(Long userId, String newPassword) {
    User user =userDao.findOne(userId);
    user.setPassword(newPassword);
    passwordHelper.encryptPassword(user);
    userDao.updateUser(user);
}
```

在创建账户及修改密码时直接把生成密码操作委托给 PasswordHelper。

PasswordHelper

```
public class PasswordHelper {
    private RandomNumberGenerator randomNumberGenerator =
            new SecureRandomNumberGenerator();
    private String algorithmName = "md5";
    private final int hashIterations = 2;
    public void encryptPassword(User user) {
        user.setSalt(randomNumberGenerator.nextBytes().toHex());
        String newPassword = new SimpleHash(
                algorithmName,
                user.getPassword(),
                ByteSource.Util.bytes(user.getCredentialsSalt()),
                hashIterations).toHex();
        user.setPassword(newPassword);
    }
}
```

之后的 CredentialsMatcher 需要和此处加密的算法一样。user.getCredentialsSalt()辅助方法返回 username+salt。

为了节省篇幅，对于 DAO/Service 的接口及实现，具体请参考源码 com.github.zhangkaitao.shiro.chapter6 。另外请参考 Service 层的测试用例 com.github.zhangkaitao.shiro.chapter6.service.ServiceTest。

4、定义 Realm

RetryLimitHashedCredentialsMatcher

和第五章的一样，在此就不罗列代码了，请参考源码 com.github.zhangkaitao.shiro.chapter6.credentials.RetryLimitHashedCredentialsMatcher。

UserRealm

```
public class UserRealm extends AuthorizingRealm {
    private UserService userService = new UserServiceImpl();
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        String username = (String)principals.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
        authorizationInfo.setRoles(userService.findRoles(username));
        authorizationInfo.setStringPermissions(userService.findPermissions(username));
        return authorizationInfo;
    }
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        String username = (String)token.getPrincipal();
        User user = userService.findByUsername(username);
        if(user == null) {
            throw new UnknownAccountException();//没找到帐号
        }
        if(Boolean.TRUE.equals(user.getLocked())) {
            throw new LockedAccountException(); //帐号锁定
        }
        //交给 AuthenticatingRealm 使用 CredentialsMatcher 进行密码匹配，如果觉得人家
的不好可以在此判断或自定义实现
        SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(
                user.getUsername(), //用户名
                user.getPassword(), //密码
                ByteSource.Util.bytes(user.getCredentialsSalt()),//salt=username+salt
                getName()  //realm name
        );
        return authenticationInfo;
    }
}
```

1、UserRealm 父类 AuthorizingRealm 将获取 Subject 相关信息分成两步：获取身份验证

信息（doGetAuthenticationInfo）及授权信息（doGetAuthorizationInfo）；

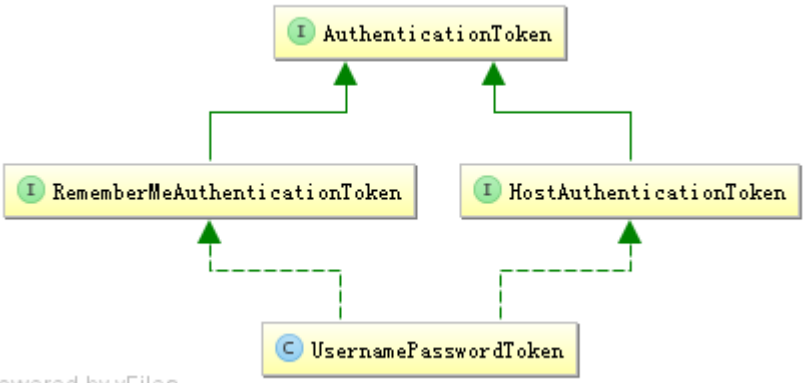
2、doGetAuthenticationInfo 获取身份验证相关信息：首先根据传入的用户名获取 User 信息；然后如果 user 为空，那么抛出没找到帐号异常 UnknownAccountException；如果 user 找到但锁定了抛出锁定异常 LockedAccountException；最后生成 AuthenticationInfo 信息，交给间接父类 AuthenticatingRealm 使用 CredentialsMatcher 进行判断密码是否匹配，如果不匹配将抛出密码错误异常 IncorrectCredentialsException；另外如果密码重试此处太多将抛出超出重试次数异常 ExcessiveAttemptsException；在组装 SimpleAuthenticationInfo 信息时，需要传入：身份信息（用户名）、凭据（密文密码）、盐（username+salt），CredentialsMatcher 使用盐加密传入的明文密码和此处的密文密码进行匹配。

3、doGetAuthorizationInfo 获取授权信息：PrincipalCollection 是一个身份集合，因为我们现在就一个 Realm，所以直接调用 getPrimaryPrincipal 得到之前传入的用户名即可；然后根据用户名调用 UserService 接口获取角色及权限信息。

5、测试用例

为了节省篇幅，请参考测试用例 com.github.zhangkaitao.shiro.chapter6.realm.UserRealmTest。包含了：登录成功、用户名错误、密码错误、密码超出重试次数、有/没有角色、有/没有权限的测试。

AuthenticationToken



AuthenticationToken 用于收集用户提交的身份（如用户名）及凭据（如密码）：

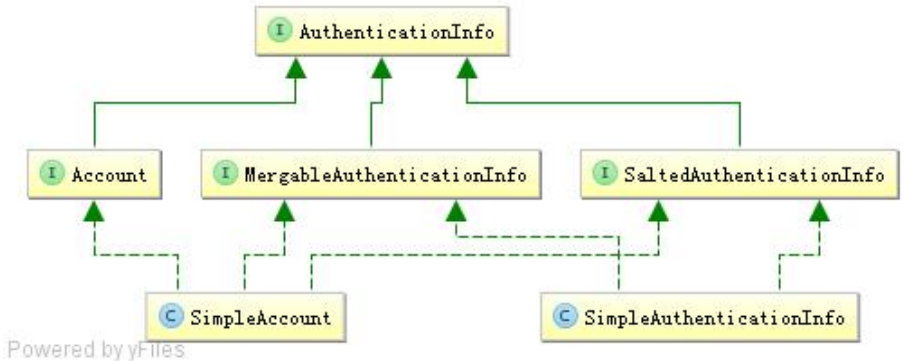
```
public interface AuthenticationToken extends Serializable {
    Object getPrincipal(); //身份
    Object getCredentials(); //凭据
}
```

扩展接口 RememberMeAuthenticationToken：提供了“boolean isRememberMe()”现“记住

我”的功能；
扩展接口是 HostAuthenticationToken：提供了“String getHost()”方法用于获取用户“主机”的功能。

Shiro 提供了一个直接拿来用的 UsernamePasswordToken，用于实现用户名/密码 Token 组，另外其实现了 RememberMeAuthenticationToken 和 HostAuthenticationToken，可以实现记住我及主机验证的支持。

AuthenticationInfo



AuthenticationInfo 有两个作用：
1、如果 Realm 是 AuthenticatingRealm 子类，则提供给 AuthenticatingRealm 内部使用的 CredentialsMatcher 进行凭据验证；（如果没有继承它需要在自己的 Realm 中自己实现验证）；
2、提供给 SecurityManager 来创建 Subject（提供身份信息）；

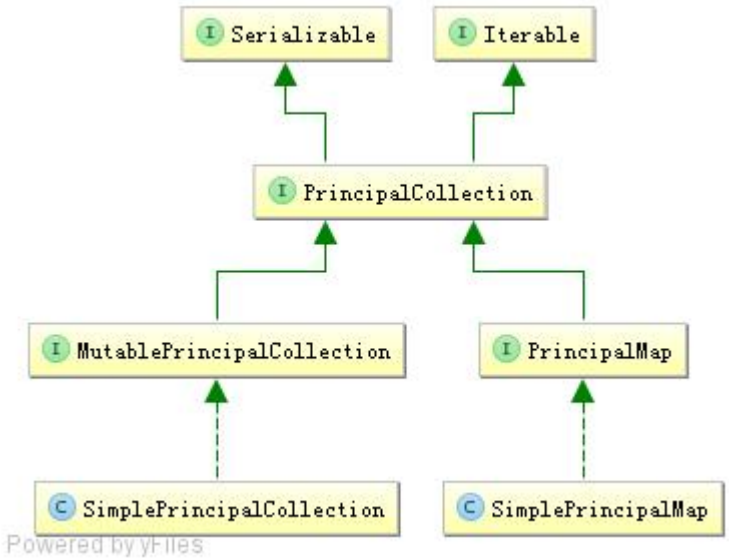
MergableAuthenticationInfo 用于提供在多 Realm 时合并 AuthenticationInfo 的功能，主要合并 Principal、如果是其他的如 credentialsSalt，会用后边的信息覆盖前边的。

比如 HashedCredentialsMatcher，在验证时会判断 AuthenticationInfo 是否是 SaltedAuthenticationInfo 子类，来获取盐信息。

Account 相当于我们之前的 User，SimpleAccount 是其一个实现；在 IniRealm、PropertiesRealm 这种静态创建帐号信息的场景中使用，这些 Realm 直接继承了 SimpleAccountRealm，而 SimpleAccountRealm 提供了相关的 API 来动态维护 SimpleAccount；即可以通过这些 API 来动态增删改查 SimpleAccount；动态增删改查角色/权限信息。及如果您的帐号不是特别多，可以使用这种方式，具体请参考 SimpleAccountRealm Javadoc。

其他情况一般返回 SimpleAuthenticationInfo 即可。

PrincipalCollection



因为我们可以在 Shiro 中同时配置多个 Realm，所以呢身份信息可能就有多个；因此其提供了 PrincipalCollection 用于聚合这些身份信息：

```
public interface PrincipalCollection extends Iterable, Serializable {
    Object getPrimaryPrincipal(); //得到主要的身份
    <T> T oneByType(Class<T> type); //根据身份类型获取第一个
    <T> Collection<T> byType(Class<T> type); //根据身份类型获取一组
    List asList(); //转换为 List
    Set asSet(); //转换为 Set
    Collection fromRealm(String realmName); //根据 Realm 名字获取
    Set<String> getRealmNames(); //获取所有身份验证通过的 Realm 名字
    boolean isEmpty(); //判断是否为空
}
```

因为 PrincipalCollection 聚合了多个，此处最需要注意的是 getPrimaryPrincipal，如果只有一个 Principal 那么直接返回即可，如果有多个 Principal，则返回第一个（因为内部使用 Map 存储，所以可以认为是返回任意一个）；oneByType / byType 根据凭据的类型返回相应的 Principal；fromRealm 根据 Realm 名字（每个 Principal 都与一个 Realm 关联）获取相应的 Principal。

MutablePrincipalCollection 是一个可变的 PrincipalCollection 接口，即提供了如下可变方法：

```
public interface MutablePrincipalCollection extends PrincipalCollection {
    void add(Object principal, String realmName); //添加 Realm-Principal 的关联
    void addAll(Collection principals, String realmName); //添加一组 Realm-Principal 的关联
    void addAll(PrincipalCollection principals);//添加 PrincipalCollection
    void clear();//清空
}
```

目前 Shiro 只提供了一个实现 SimplePrincipalCollection，还记得之前的 AuthenticationStrategy 实现嘛，用于在多 Realm 时判断是否满足条件的，在大多数实现中（继承了 AbstractAuthenticationStrategy） afterAttempt 方法会进行 AuthenticationInfo（实现了 MergableAuthenticationInfo）的 merge，比如 SimpleAuthenticationInfo 会合并多个 Principal 为一个 PrincipalCollection。

对于 PrincipalMap 是 Shiro 1.2 中的一个实验品，暂时无用，具体可以参考其 Javadoc。接下来通过示例来看看 PrincipalCollection。

1、准备三个 Realm

MyRealm1

```
public class MyRealm1 implements Realm {
    @Override
    public String getName() {
        return "a"; //realm name 为 “a”
    }
    //省略 supports 方法，具体请见源码
    @Override
    public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        return new SimpleAuthenticationInfo(
                "zhang", //身份 字符串类型
                "123",    //凭据
                getName() //Realm Name
        );
    }
}
```

MyRealm2

和 MyRealm1 完全一样，只是 Realm 名字为 b。

MyRealm3

```
public class MyRealm3 implements Realm {
    @Override
    public String getName() {
        return "c"; //realm name 为 “c”
    }
    //省略 supports 方法，具体请见源码
    @Override
    public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
        User user = new User("zhang", "123");
        return new SimpleAuthenticationInfo(
                user, //身份 User 类型
                "123",    //凭据
                getName() //Realm Name
        );
    }
}
```

和 MyRealm1 同名，但返回的 Principal 是 User 类型。

2、ini 配置（shiro-multirealm.ini）

```
[main]
realm1=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm1
realm2=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm2
realm3=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm3
securityManager.realms=$realm1,$realm2,$realm3
```

3、测试用例（com.github.zhangkaitao.shiro.chapter6.realm.PrincialCollectionTest）

因为我们的 Realm 中没有进行身份及凭据验证，所以相当于身份验证都是成功的，都将返回：

```
Object primaryPrincipal1 = subject.getPrincipal();
PrincipalCollection princialCollection = subject.getPrincipals();
Object primaryPrincipal2 = princialCollection.getPrimaryPrincipal();
```

我们可以直接调用 subject.getPrincipal 获取 PrimaryPrincipal（即所谓的第一个）；或者通过 getPrincipals 获取 PrincipalCollection；然后通过其 getPrimaryPrincipal 获取 PrimaryPrincipal。

```
Set<String> realmNames = princialCollection.getRealmNames();
```

获取所有身份验证成功的 Realm 名字。

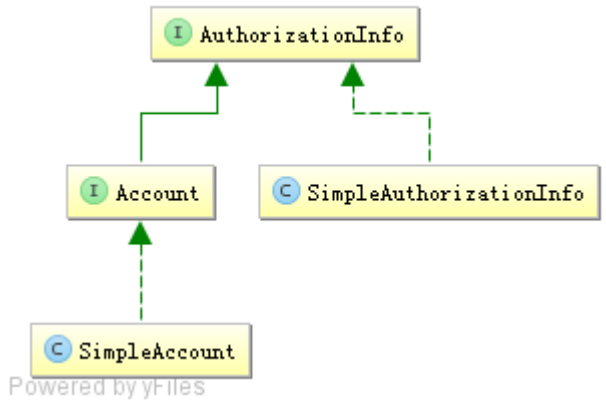
```
Set<Object> principals = princialCollection.asSet(); //asList 和 asSet 的结果一样
```

将身份信息转换为 Set/List，即使转换为 List，也是先转换为 Set 再完成的。

```
Collection<User> users = princialCollection.fromRealm("c");
```

根据 Realm 名字获取身份，因为 Realm 名字可以重复，所以可能多个身份，建议 Realm 名字尽量不要重复。

AuthorizationInfo



AuthorizationInfo 用于聚合授权信息的：

```
public interface AuthorizationInfo extends Serializable {
    Collection<String> getRoles(); //获取角色字符串信息
    Collection<String> getStringPermissions(); //获取权限字符串信息
    Collection<Permission> getObjectPermissions(); //获取 Permission 对象信息
}
```

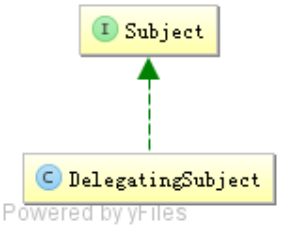
当我们使用 AuthorizingRealm 时，如果身份验证成功，在进行授权时就通过 doGetAuthorizationInfo 方法获取角色/权限信息用于授权验证。

Shiro 提供了一个实现 SimpleAuthorizationInfo，大多数时候使用这个即可。

对于 Account 及 SimpleAccount，之前的【6.3 AuthenticationInfo】已经介绍过了，用于

SimpleAccountRealm 子类，实现动态角色/权限维护的。

Subject



Subject 是 Shiro 的核心对象，基本所有身份验证、授权都是通过 Subject 完成。

1、身份信息获取

```
Object getPrincipal(); //Primary Principal
PrincipalCollection getPrincipals(); // PrincipalCollection
```

2、身份验证

```
void login(AuthenticationToken token) throws AuthenticationException;
boolean isAuthenticated();
boolean isRemembered();
```

通过 login 登录，如果登录失败将抛出相应的 AuthenticationException，如果登录成功调用 isAuthenticated 就会返回 true，即已经通过身份验证；如果 isRemembered 返回 true，表示是通过记住我功能登录的而不是调用 login 方法登录的。isAuthenticated/isRemembered 是互斥的，即如果其中一个返回 true，另一个返回 false。

3、角色授权验证

```
boolean hasRole(String roleIdentifier);
boolean[] hasRoles(List<String> roleIdentifiers);
boolean hasAllRoles(Collection<String> roleIdentifiers);
void checkRole(String roleIdentifier) throws AuthorizationException;
void checkRoles(Collection<String> roleIdentifiers) throws AuthorizationException;
void checkRoles(String... roleIdentifiers) throws AuthorizationException;
```

hasRole*进行角色验证，验证后返回 true/false；而 checkRole*验证失败时抛出 AuthorizationException 异常。

4、权限授权验证

```
boolean isPermitted(String permission);
boolean isPermitted(Permission permission);
boolean[] isPermitted(String... permissions);
boolean[] isPermitted(List<Permission> permissions);
boolean isPermittedAll(String... permissions);
boolean isPermittedAll(Collection<Permission> permissions);
void checkPermission(String permission) throws AuthorizationException;
void checkPermission(Permission permission) throws AuthorizationException;
void checkPermissions(String... permissions) throws AuthorizationException;
void checkPermissions(Collection<Permission> permissions) throws AuthorizationException;
```

isPermitted*进行权限验证，验证后返回 true/false；而 checkPermission*验证失败时抛出 AuthorizationException。

5、会话

```
Session getSession(); //相当于 getSession(true)
Session getSession(boolean create);
```

类似于 Web 中的会话。如果登录成功就相当于建立了会话，接着可以使用 getSession 获取；如果 create=true 如果没有会话将返回 null，而 create=true 如果没有会话会强制创建一个。

6、退出

```
void logout();
```

7、RunAs

```
void runAs(PrincipalCollection principals) throws NullPointerException, IllegalStateException;
boolean isRunAs();
PrincipalCollection getPreviousPrincipals();
PrincipalCollection releaseRunAs();
```

RunAs 即实现“允许 A 假设为 B 身份进行访问”；通过调用 subject.runAs(b)进行访问；接着调用 subject.getPrincipals 将获取到 B 的身份；此时调用 isRunAs 将返回 true；而 a 的身份需要通过 subject. getPreviousPrincipals 获取；如果不需要 RunAs 了调用 subject. releaseRunAs 即可。

8、多线程

```
<V> V execute(Callable<V> callable) throws ExecutionException;
void execute(Runnable runnable);
<V> Callable<V> associateWith(Callable<V> callable);
Runnable associateWith(Runnable runnable);
```

实现线程之间的 Subject 传播，因为 Subject 是线程绑定的；因此在多线程执行中需要传播到相应的线程才能获取到相应的 Subject。最简单的办法就是通过 execute(runnable/callable 实例)直接调用；或者通过 associateWith(runnable/callable 实例)得到一个包装后的实例；它们都是通过：1、把当前线程的 Subject 绑定过去；2、在线程执行结束后自动释放。

Subject 自己不会实现相应的身份验证/授权逻辑，而是通过 DelegatingSubject 委托给 SecurityManager 实现；及可以理解为 Subject 是一个面门。

对于 Subject 的构建一般没必要我们去创建；一般通过 SecurityUtils.getSubject()获取：

```
public static Subject getSubject() {
    Subject subject = ThreadContext.getSubject();
    if (subject == null) {
        subject = (new Subject.Builder()).buildSubject();
        ThreadContext.bind(subject);
    }
    return subject;
}
```

即首先查看当前线程是否绑定了 Subject，如果没有通过 Subject.Builder 构建一个然后绑定到现场返回。

如果想自定义创建，可以通过：

```
new Subject.Builder().principals(身份).authenticated(true/false).buildSubject()
```

这种可以创建相应的 Subject 实例了，然后自己绑定到线程即可。在 new Builder()时如果没有传入 SecurityManager，自动调用 SecurityUtils.getSecurityManager 获取；也可以自己传入一个实例。

对于 Subject 我们一般这么使用：

1、身份验证（login）

2、授权（hasRole*/isPermitted*或 checkRole*/checkPermission*）

3、将相应的数据存储到会话（Session）

4、切换身份（RunAs）/多线程身份传播

5、退出

而我们必须的功能就是 1、2、5。到目前为止我们就可以使用 Shiro 进行应用程序的安全控制了，但是还是缺少如对 Web 验证、Java 方法验证等的一些简化实现。

http://jinnianshilongnian.iteye.com/

第七章　与 Web 集成

Shiro 提供了与 Web 集成的支持，其通过一个 ShiroFilter 入口来拦截需要安全控制的 URL，然后进行相应的控制，ShiroFilter 类似于如 Strut2/SpringMVC 这种 web 框架的前端控制器，其是安全控制的入口点，其负责读取配置（如 ini 配置文件），然后判断 URL 是否需要登录/权限等工作。

准备环境

1、创建 webapp 应用

此处我们使用了 jetty-maven-plugin 和 tomcat7-maven-plugin 插件；这样可以直接使用“mvn jetty:run ”或“mvn tomcat7:run ”直接运行 webapp 了。然后通过 URLhttp://localhost:8080/chapter7/访问即可。

2、依赖

Servlet3

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
```

Servlet3 的知识可以参考 https://github.com/zhangkaitao/servlet3-showcase 及 Servlet3 规范 http://www.iteye.com/blogs/subjects/Servlet-3-1。

shiro-web

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-web</artifactId>
    <version>1.2.2</version>
</dependency>
```

其他依赖请参考源码的 pom.xml。

ShiroFilter 入口

1、Shiro 1.1 及以前版本配置方式

```
<filter>
    <filter-name>iniShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
    <init-param>
        <param-name>configPath</param-name>
        <param-value>classpath:shiro.ini</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>iniShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

1、使用 IniShiroFilter 作为 Shiro 安全控制的入口点，通过 url-pattern 指定需要安全的 URL；
2、通过 configPath 指定 ini 配置文件位置，默认是先从/WEB-INF/shiro.ini 加载，如果没有就默认加载 classpath:shiro.ini，即默认相对于 web 应用上下文根路径；
3、也可以通过如下方式直接内嵌 ini 配置文件内容到 web.xml

```
<init-param>
    <param-name>config</param-name>
    <param-value>
        ini 配置文件贴在这
    </param-value>
</init-param>
```

2、Shiro 1.2 及以后版本的配置方式

从 Shiro 1.2 开始引入了 Environment/WebEnvironment 的概念，即由它们的实现提供相应的 SecurityManager 及其相应的依赖。ShiroFilter 会自动找到 Environment 然后获取相应的依赖。

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>
</listener>
```

通过 EnvironmentLoaderListener 来创建相应的 WebEnvironment，并自动绑定到 ServletContext，默认使用 IniWebEnvironment 实现。

可以通过如下配置修改默认实现及其加载的配置文件位置：

```
<context-param>
    <param-name>shiroEnvironmentClass</param-name>
    <param-value>org.apache.shiro.web.env.IniWebEnvironment</param-value>
</context-param>
     <context-param>
          <param-name>shiroConfigLocations</param-name>
          <param-value>classpath:shiro.ini</param-value>
     </context-param>
```

shiroConfigLocations 默认是“/WEB-INF/shiro.ini”，IniWebEnvironment 默认是先从/WEB-INF/shiro.ini 加载，如果没有就默认加载 classpath:shiro.ini。

3、与 Spring 集成

```
<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"/>
     <!—忽略其他，详见与 Spring 集成部分 -->
</bean>
```

最后不要忘了使用 org.springframework.web.context.ContextLoaderListener 加载这个 spring 配置文件即可。

因为我们现在的 shiro 版本是 1.2 的，因此之后的测试都是使用 1.2 的配置。

Web INI 配置

ini 配置部分和之前的相比将多出对 url 部分的配置。

```
[main]
#默认是/login.jsp
authc.loginUrl=/login
roles.unauthorizedUrl=/unauthorized
perms.unauthorizedUrl=/unauthorized
[users]
zhang=123,admin
wang=123
[roles]
admin=user:*,menu:*
[urls]
/login=anon
/unauthorized=anon
/static/**=anon
/authenticated=authc
/role=authc,roles[admin]
/permission=authc,perms["user:create"]
```

其中最重要的就是[urls]部分的配置，其格式是： “url=拦截器[参数]，拦截器[参数]”；即如果当前请求的 url 匹配[urls]部分的某个 url 模式，将会执行其配置的拦截器。比如 anon 拦截器表示匿名访问（即不需要登录即可访问）；authc 拦截器表示需要身份认证通过后才能访问；roles[admin]拦截器表示需要有 admin 角色授权才能访问；而 perms["user:create"]拦截器表示需要有“user:create”权限才能访问。

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，但不匹配/admin 或/admin2；

***：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。

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。

拦截器将在下一节详细介绍。接着我们来看看身份验证、授权及退出在 web 中如何实现。

1、身份验证（登录）

1.1、首先配置需要身份验证的 url

```
/authenticated=authc
/role=authc,roles[admin]
/permission=authc,perms["user:create"]
```

即访问这些地址时会首先判断用户有没有登录，如果没有登录默会跳转到登录页面，默认是/login.jsp，可以通过在[main]部分通过如下配置修改：

```
authc.loginUrl=/login
```

1.2、登录 Servlet（com.github.zhangkaitao.shiro.chapter7.web.servlet.LoginServlet）

```
@WebServlet(name = "loginServlet", urlPatterns = "/login")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
          req.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
            String error = null;
            String username = req.getParameter("username");
            String password = req.getParameter("password");
            Subject subject = SecurityUtils.getSubject();
            UsernamePasswordToken token =
                    new UsernamePasswordToken(username, password);
```

```
            try {
                subject.login(token);
            } catch (UnknownAccountException e) {
                error = "用户名/密码错误";
            } catch (IncorrectCredentialsException e) {
                error = "用户名/密码错误";
            } catch (AuthenticationException e) {
                //其他错误，比如锁定，如果想单独处理请单独 catch 处理
                error = "其他错误：" + e.getMessage();
            }
            if(error != null) {//出错了，返回登录页面
                req.setAttribute("error", error);
                req.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(req, resp);
            } else {//登录成功
                req.getRequestDispatcher("/WEB-INF/jsp/loginSuccess.jsp").forward(req, resp);
            }
        }
    }
```

1、doGet 请求时展示登录页面；
2、doPost 时进行登录，登录时收集 username/password 参数，然后提交给 Subject 进行登录。如果有错误再返回到登录页面；否则跳转到登录成功页面（此处应该返回到访问登录页面之前的那个页面，或者没有上一个页面时访问主页）。
3、JSP 页面请参考源码。

1.3、测试

首先输入 http://localhost:8080/chapter7/login 进行登录，登录成功后接着可以访问 http://localhost:8080/chapter7/authenticated 来显示当前登录的用户：

```
${subject.principal}身份验证已通过。
```

当前实现的一个缺点就是，永远返回到同一个成功页面（比如首页），在实际项目中比如支付时如果没有登录将跳转到登录页面，登录成功后再跳回到支付页面；对于这种功能大家可以在登录时把当前请求保存下来，然后登录成功后再重定向到该请求即可。

Shiro 内置了登录（身份验证）的实现：基于表单的和基于 Basic 的验证，其通过拦截器实现。

2、基于 Basic 的拦截器身份验证

2.1、shiro-basicfilterlogin.ini 配置

```
[main]
authcBasic.applicationName=please login
……省略 users
[urls]
/role=authcBasic,roles[admin]
```

1、authcBasic 是 org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter 类型的实例，其用于实现基于 Basic 的身份验证；applicationName 用于弹出的登录框显示信息使用，如图：

2、[urls]部分配置了/role 地址需要走 authcBasic 拦截器，即如果访问/role 时还没有通过身份验证那么将弹出如上图的对话框进行登录，登录成功即可访问。

2.2、web.xml

把 shiroConfigLocations 改为 shiro-basicfilterlogin.ini 即可。

2.3、测试

输入 http://localhost:8080/chapter7/role，会弹出之前的 Basic 验证对话框输入“zhang/123”即可登录成功进行访问。

3、基于表单的拦截器身份验证

基于表单的拦截器身份验证和【1】类似，但是更简单，因为其已经实现了大部分登录逻辑；我们只需要指定：登录地址/登录失败后错误信息存哪/成功的地址即可。

3.1、shiro-formfilterlogin.ini

```
[main]
authc.loginUrl=/formfilterlogin
authc.usernameParam=username
authc.passwordParam=password
authc.successUrl=/
authc.failureKeyAttribute=shiroLoginFailure

[urls]
/role=authc,roles[admin]
```

1、authc 是 org.apache.shiro.web.filter.authc.FormAuthenticationFilter 类型的实例，其用于实现基于表单的身份验证；通过 loginUrl 指定当身份验证时的登录表单；usernameParam 指定登录表单提交的用户名参数名；passwordParam 指定登录表单提交的密码参数名；successUrl 指定登录成功后重定向的默认地址（默认是“/”）（如果有上一个地址会自动重定向带该地址）；failureKeyAttribute 指定登录失败时的 request 属性 key（默认 shiroLoginFailure）；这样可以在登录表单得到该错误 key 显示相应的错误消息；

3.2、web.xml

把 shiroConfigLocations 改为 shiro- formfilterlogin.ini 即可。

3.3、登录 Servlet

```
@WebServlet(name = "formFilterLoginServlet", urlPatterns = "/formfilterlogin")
public class FormFilterLoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
        doPost(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
        String errorClassName = (String)req.getAttribute("shiroLoginFailure");
        if(UnknownAccountException.class.getName().equals(errorClassName)) {
            req.setAttribute("error", "用户名/密码错误");
        } else if(IncorrectCredentialsException.class.getName().equals(errorClassName)) {
            req.setAttribute("error", "用户名/密码错误");
        } else if(errorClassName != null) {
            req.setAttribute("error", "未知错误：" + errorClassName);
```

```
        }
        req.getRequestDispatcher("/WEB-INF/jsp/formfilterlogin.jsp").forward(req, resp);
    }
}
```

在登录 Servlet 中通过 shiroLoginFailure 得到 authc 登录失败时的异常类型名，然后根据此异常名来决定显示什么错误消息。

4、测试

输入 http://localhost:8080/chapter7/role，会跳转到“/formfilterlogin”登录表单，提交表单如果 authc 拦截器登录成功后，会直接重定向会之前的地址“/role”；假设我们直接访问“/formfilterlogin”的话登录成功将直接到默认的 successUrl。

4、授权（角色/权限验证）

4.1、shiro.ini

```
[main]
roles.unauthorizedUrl=/unauthorized
perms.unauthorizedUrl=/unauthorized
 [urls]
/role=authc,roles[admin]
/permission=authc,perms["user:create"]
```

通过 unauthorizedUrl 属性指定如果授权失败时重定向到的地址。roles 是 org.apache.shiro.web.filter.authz.RolesAuthorizationFilter 类型的实例，通过参数指定访问时需要的角色，如“[admin]”，如果有多个使用“，”分割，且验证时是 hasAllRole 验证，即且的关系。Perms 是 org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter 类型的实例，和 roles 类似，只是验证权限字符串。

4.2、web.xml

把 shiroConfigLocations 改为 shiro.ini 即可。

4.3、RoleServlet/PermissionServlet

```
@WebServlet(name = "permissionServlet", urlPatterns = "/permission")
public class PermissionServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Subject subject = SecurityUtils.getSubject();
```

```
            subject.checkPermission("user:create");
            req.getRequestDispatcher("/WEB-INF/jsp/hasPermission.jsp").forward(req, resp);
        }
}
```

```
@WebServlet(name = "roleServlet", urlPatterns = "/role")
public class RoleServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            Subject subject = SecurityUtils.getSubject();
            subject.checkRole("admin");
            req.getRequestDispatcher("/WEB-INF/jsp/hasRole.jsp").forward(req, resp);
    }
}
```

4.4、测试

首先访问 http://localhost:8080/chapter7/login，使用帐号“zhang/123”进行登录，再访问/role 或/permission 时会跳转到成功页面（因为其授权成功了）；如果使用帐号“wang/123”登录成功后访问这两个地址会跳转到“/unauthorized”即没有授权页面。

5、退出

5.1、shiro.ini

```
[urls]
/logout=anon
```

指定/logout 使用 anon 拦截器即可，即不需要登录即可访问。

5.2、LogoutServlet

```
@WebServlet(name = "logoutServlet", urlPatterns = "/logout")
public class LogoutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            SecurityUtils.getSubject().logout();
            req.getRequestDispatcher("/WEB-INF/jsp/logoutSuccess.jsp").forward(req, resp);
    }
}
```

直接调用 Subject.logout 即可，退出成功后转发/重定向到相应页面即可。

5.3、测试

首先访问 http://localhost:8080/chapter7/login，使用帐号“zhang/123”进行登录，登录成功后访问/logout 即可退出。

Shiro 也提供了 logout 拦截器用于退出，其是 org.apache.shiro.web.filter.authc.LogoutFilter 类型的实例，我们可以在 shiro.ini 配置文件中通过如下配置完成退出：

```
[main]
logout.redirectUrl=/login

[urls]
/logout2=logout
```

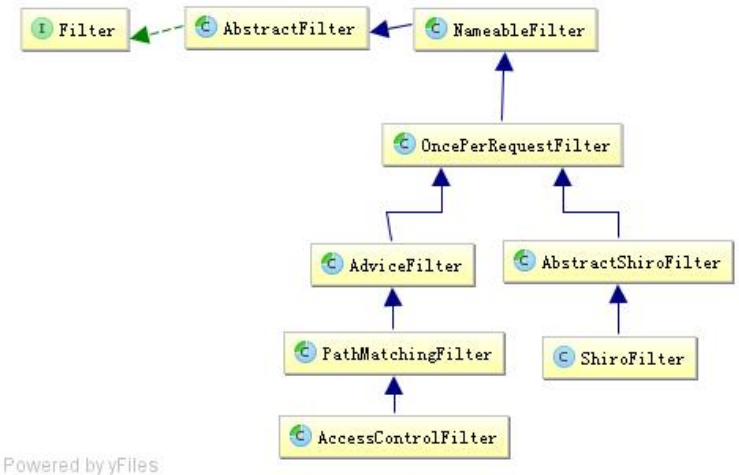
通过 logout.redirectUrl 指定退出后重定向的地址；通过/logout2=logout 指定退出 url 是/logout2。这样当我们登录成功后然后访问/logout2 即可退出。

http://jinnianshilongnian.iteye.com/

第八章 拦截器机制

拦截器介绍

Shiro 使用了与 Servlet 一样的 Filter 接口进行扩展；所以如果对 Filter 不熟悉可以参考《Servlet3.1 规范》http://www.iteye.com/blogs/subjects/Servlet-3-1 了解 Filter 的工作原理。
首先下图是 Shiro 拦截器的基础类图：



Powered by yFiles

1、NameableFilter

NameableFilter 给 Filter 起个名字，如果没有设置默认就是 FilterName；还记得之前的如 authc 吗？当我们组装拦截器链时会根据这个名字找到相应的拦截器实例；

2、OncePerRequestFilter

OncePerRequestFilter 用于防止多次执行 Filter 的；也就是说一次请求只会走一次拦截器链；另外提供 enabled 属性，表示是否开启该拦截器实例，默认 enabled=true 表示开启，如果不想让某个拦截器工作，可以设置为 false 即可。

3、ShiroFilter

ShiroFilter 是整个 Shiro 的入口点，用于拦截需要安全控制的请求进行处理，这个之前已经用过了。

4、AdviceFilter

AdviceFilter 提供了 AOP 风格的支持，类似于 SpringMVC 中的 Interceptor：

```
boolean preHandle(ServletRequest request, ServletResponse response) throws Exception
void postHandle(ServletRequest request, ServletResponse response) throws Exception
void afterCompletion(ServletRequest request, ServletResponse response, Exception exception)
throws Exception;
```

preHandler：类似于 AOP 中的前置增强；在拦截器链执行之前执行；如果返回 true 则继续拦截器链；否则中断后续的拦截器链的执行直接返回；进行预处理（如基于表单的身份验证、授权）
postHandle：类似于 AOP 中的后置返回增强；在拦截器链执行完成后执行；进行后处理（如记录执行时间之类的）；
afterCompletion：类似于 AOP 中的后置最终增强；即不管有没有异常都会执行；可以进行清理资源（如接触 Subject 与线程的绑定之类的）；

5、PathMatchingFilter

PathMatchingFilter 提供了基于 Ant 风格的请求路径匹配功能及拦截器参数解析的功能，如“roles[admin,user]”自动根据“，”分割解析到一个路径参数配置并绑定到相应的路径：

```
boolean pathsMatch(String path, ServletRequest request)
boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue)
throws Exception
```

pathsMatch：该方法用于 path 与请求路径进行匹配的方法；如果匹配返回 true；
onPreHandle：在 preHandle 中，当 pathsMatch 匹配一个路径后，会调用 opPreHandler 方法并将路径绑定参数配置传给 mappedValue；然后可以在这个方法中进行一些验证（如角色授权），如果验证失败可以返回 false 中断流程；默认返回 true；也就是说子类可以只实现 onPreHandle 即可，无须实现 preHandle。如果没有 path 与请求路径匹配，默认是通过的（即 preHandle 返回 true）。

6、AccessControlFilter

AccessControlFilter 提供了访问控制的基础功能；比如是否允许访问/当访问拒绝时如何处理等：

```
abstract boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object
mappedValue) throws Exception;
boolean onAccessDenied(ServletRequest request, ServletResponse response, Object
mappedValue) throws Exception;
abstract boolean onAccessDenied(ServletRequest request, ServletResponse response) throws
Exception;
```

isAccessAllowed：表示是否允许访问；mappedValue 就是[urls]配置中拦截器参数部分，如果允许访问返回 true，否则 false；
onAccessDenied：表示当访问拒绝时是否已经处理了；如果返回 true 表示需要继续处理；如果返回 false 表示该拦截器实例已经处理了，将直接返回即可。

onPreHandle 会自动调用这两个方法决定是否继续处理：

```
boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
    return isAccessAllowed(request, response, mappedValue) || onAccessDenied(request, response, mappedValue);
}
```

另外 AccessControlFilter 还提供了如下方法用于处理如登录成功后/重定向到上一个请求：

```
void setLoginUrl(String loginUrl) //身份验证时使用，默认/login.jsp
String getLoginUrl()
Subject getSubject(ServletRequest request, ServletResponse response) //获取 Subject 实例
boolean isLoginRequest(ServletRequest request, ServletResponse response)//当前请求是否是登录请求
void saveRequestAndRedirectToLogin(ServletRequest request, ServletResponse response) throws IOException //将当前请求保存起来并重定向到登录页面
void saveRequest(ServletRequest request) //将请求保存起来，如登录成功后再重定向回该请求
void redirectToLogin(ServletRequest request, ServletResponse response) //重定向到登录页面
```

比如基于表单的身份验证就需要使用这些功能。

到此基本的拦截器就完事了，如果我们想进行访问访问的控制就可以继承 AccessControlFilter；如果我们要添加一些通用数据我们可以直接继承 PathMatchingFilter。

拦截器链

Shiro 对 Servlet 容器的 FilterChain 进行了代理，即 ShiroFilter 在继续 Servlet 容器的 Filter 链的执行之前，通过 ProxiedFilterChain 对 Servlet 容器的 FilterChain 进行了代理；即先走 Shiro 自己的 Filter 体系，然后才会委托给 Servlet 容器的 FilterChain 进行 Servlet 容器级别的 Filter 链执行；Shiro 的 ProxiedFilterChain 执行流程：1、先执行 Shiro 自己的 Filter 链；2、再执行 Servlet 容器的 Filter 链（即原始的 Filter）。
而 ProxiedFilterChain 是通过 FilterChainResolver 根据配置文件中[urls]部分是否与请求的 URL 是否匹配解析得到的。

```
FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain);
```

即传入原始的 chain 得到一个代理的 chain。

Shiro 内部提供了一个路径匹配的 FilterChainResolver 实现：PathMatchingFilterChainResolver，其根据[urls]中配置的 url 模式（默认 Ant 风格）=拦截器链和请求的 url 是否匹配来解析得到配置的拦截器链的；而 PathMatchingFilterChainResolver 内部通过 FilterChainManager 维护着拦截器链，比如 DefaultFilterChainManager 实现维护着 url 模式与拦截器链的关系。因此我们可以通过 FilterChainManager 进行动态动态增加 url 模式与拦截器链的关系。

DefaultFilterChainManager 会默认添加 org.apache.shiro.web.filter.mgt.DefaultFilter 中声明的拦截器：

```
public enum DefaultFilter {
    anon(AnonymousFilter.class),
    authc(FormAuthenticationFilter.class),
    authcBasic(BasicHttpAuthenticationFilter.class),
    logout(LogoutFilter.class),
    noSessionCreation(NoSessionCreationFilter.class),
    perms(PermissionsAuthorizationFilter.class),
    port(PortFilter.class),
    rest(HttpMethodPermissionFilter.class),
    roles(RolesAuthorizationFilter.class),
    ssl(SslFilter.class),
    user(UserFilter.class);
}
```

下一节会介绍这些拦截器的作用。

如果要注册自定义拦截器，IniSecurityManagerFactory/WebIniSecurityManagerFactory 在启动时会自动扫描 ini 配置文件中的[filters]/[main]部分并注册这些拦截器到 DefaultFilterChainManager；且创建相应的 url 模式与其拦截器关系链。如果使用 Spring 后续章节会介绍如果注册自定义拦截器。

如果想自定义 FilterChainResolver，可以通过实现 WebEnvironment 接口完成：

```
public class MyIniWebEnvironment extends IniWebEnvironment {
    @Override
    protected FilterChainResolver createFilterChainResolver() {
        //在此处扩展自己的 FilterChainResolver
        return super.createFilterChainResolver();
    }
}
```

如果覆盖了 IniWebEnvironment 默认的 FilterChainResolver，需要自己来解析请求与 FilterChain 之间的关系。如果想动态实现 url-拦截器的注册，就可以通过实现此处的 FilterChainResolver 来完成，比如：

```
//1、创建 FilterChainResolver
PathMatchingFilterChainResolver filterChainResolver =
        new PathMatchingFilterChainResolver();
//2、创建 FilterChainManager
DefaultFilterChainManager filterChainManager = new DefaultFilterChainManager();
//3、注册 Filter
for(DefaultFilter filter : DefaultFilter.values()) {
    filterChainManager.addFilter(
        filter.name(), (Filter) ClassUtils.newInstance(filter.getFilterClass()));
}
//4、注册 URL-Filter 的映射关系
filterChainManager.addToChain("/login.jsp", "authc");
filterChainManager.addToChain("/unauthorized.jsp", "anon");
filterChainManager.addToChain("/**", "authc");
filterChainManager.addToChain("/**", "roles", "admin");

//5、设置 Filter 的属性
FormAuthenticationFilter authcFilter =
        (FormAuthenticationFilter)filterChainManager.getFilter("authc");
authcFilter.setLoginUrl("/login.jsp");
RolesAuthorizationFilter rolesFilter =
        (RolesAuthorizationFilter)filterChainManager.getFilter("roles");
rolesFilter.setUnauthorizedUrl("/unauthorized.jsp");

filterChainResolver.setFilterChainManager(filterChainManager);
return filterChainResolver;
```

此处自己去实现注册 filter，及 url 模式与 filter 之间的映射关系。可以通过定制 FilterChainResolver 或 FilterChainManager 来完成诸如动态 URL 匹配的实现。

然后再 web.xml 中进行如下配置 Environment：

```
<context-param>
  <param-name>shiroEnvironmentClass</param-name>
  <param-value>com.github.zhangkaitao.shiro.chapter8.web.env.MyIniWebEnvironment</para
  m-value>
</context-param>
```

自定义拦截器

通过自定义自己的拦截器可以扩展一些功能，诸如动态 url-角色/权限访问控制的实现、根据 Subject 身份信息获取用户信息绑定到 Request（即设置通用数据）、验证码验证、在线用户信息的保存等等，因为其本质就是一个 Filter；所以 Filter 能做的它就能做。

对于 Filter 的介绍请参考《Servlet 规范》中的 Filter 部分：
http://www.iteye.com/blogs/subjects/Servlet-3-1。

1、扩展 OncePerRequestFilter

OncePerRequestFilter 保证一次请求只调用一次 doFilterInternal，即如内部的 forward 不会再多执行一次 doFilterInternal：

```
public class MyOncePerRequestFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain) throws ServletException, IOException {
        System.out.println("==========once per request filter");
        chain.doFilter(request, response);
    }
}
```

然后再 shiro.ini 配置文件中：

```
[main]
myFilter1=com.github.zhangkaitao.shiro.chapter8.web.filter.MyOncePerRequestFilter
#[filters]
#myFilter1=com.github.zhangkaitao.shiro.chapter8.web.filter.MyOncePerRequestFilter
[urls]
/**=myFilter1
```

Filter 可以在[main]或[filters]部分注册，然后在[urls]部分配置 url 与 filter 的映射关系即可。

2、扩展 AdviceFilter

AdviceFilter 提供了 AOP 的功能，其实现和 SpringMVC 中的 Interceptor 思想一样：具体可参考我的 SpringMVC 教程中的处理器拦截器部分：
http://www.iteye.com/blogs/subjects/kaitao-springmvc

```
public class MyAdviceFilter extends AdviceFilter {
    @Override
    protected boolean preHandle(ServletRequest request, ServletResponse response) throws Exception {
        System.out.println("====预处理/前置处理");
        return true;//返回 false 将中断后续拦截器链的执行
    }
    @Override
    protected void postHandle(ServletRequest request, ServletResponse response) throws Exception {
        System.out.println("====后处理/后置返回处理");
    }
    @Override
    public void afterCompletion(ServletRequest request, ServletResponse response, Exception exception) throws Exception {
        System.out.println("====完成处理/后置最终处理");
    }
}
```

preHandle：进行请求的预处理，然后根据返回值决定是否继续处理（true：继续过滤器链）；可以通过它实现权限控制；
postHandle：执行完拦截器链之后正常返回后执行；
afterCompletion：不管最后有没有异常，afterCompletion 都会执行，完成如清理资源功能。

然后在 shiro.ini 中进行如下配置：

```
[filters]
myFilter1=com.github.zhangkaitao.shiro.chapter8.web.filter.MyOncePerRequestFilter
myFilter2=com.github.zhangkaitao.shiro.chapter8.web.filter.MyAdviceFilter
[urls]
/**=myFilter1,myFilter2
```

该过滤器的具体使用可参考我的 SpringMVC 教程中的处理器拦截器部分。

3、PathMatchingFilter

PathMatchingFilter 继承了 AdviceFilter，提供了 url 模式过滤的功能，如果需要对指定的请求进行处理，可以扩展 PathMatchingFilter：

```
public class MyPathMatchingFilter extends PathMatchingFilter {
    @Override
    protected boolean onPreHandle(ServletRequest request, ServletResponse response, Object
mappedValue) throws Exception {
        System.out.println("url matches,config is " + Arrays.toString((String[])mappedValue));
        return true;
    }
}
```

preHandle：会进行 url 模式与请求 url 进行匹配，如果匹配会调用 onPreHandle；如果没有配置 url 模式/没有 url 模式匹配，默认直接返回 true；
onPreHandle：如果 url 模式与请求 url 匹配，那么会执行 onPreHandle，并把该拦截器配置的参数传入。默认什么不处理直接返回 true。

然后在 shiro.ini 中进行如下配置：

```
[filters]
myFilter3=com.github.zhangkaitao.shiro.chapter8.web.filter.MyPathMatchingFilter
[urls]
/**= myFilter3[config]
```

/**就是注册给 PathMatchingFilter 的 url 模式，config 就是拦截器的配置参数，多个之间逗号分隔，onPreHandle 使用 mappedValue 接收参数值。

4、扩展 AccessControlFilter

AccessControlFilter 继承了 PathMatchingFilter，并扩展了了两个方法：

```
public boolean onPreHandle(ServletRequest request, ServletResponse response, Object
mappedValue) throws Exception {
    return isAccessAllowed(request, response, mappedValue)
            || onAccessDenied(request, response, mappedValue);
}
```

isAccessAllowed：即是否允许访问，返回 true 表示允许；
onAccessDenied：表示访问拒绝时是否自己处理，如果返回 true 表示自己不处理且继续拦截器链执行，返回 false 表示自己已经处理了（比如重定向到另一个页面）。

```
public class MyAccessControlFilter extends AccessControlFilter {
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
        System.out.println("access allowed");
        return true;
    }
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
        System.out.println("访问拒绝也不自己处理，继续拦截器链的执行");
        return true;
    }
}
```

然后在 shiro.ini 中进行如下配置：

```
[filters]
myFilter4=com.github.zhangkaitao.shiro.chapter8.web.filter.MyAccessControlFilter
[urls]
/**=myFilter4
```

5、基于表单登录拦截器

之前我们已经使用过 Shiro 内置的基于表单登录的拦截器了，此处自己做一个类似的基于表单登录的拦截器。

```
public class FormLoginFilter extends PathMatchingFilter {
    private String loginUrl = "/login.jsp";
    private String successUrl = "/";
    @Override
    protected boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
        if(SecurityUtils.getSubject().isAuthenticated()) {
            return true;//已经登录过
        }
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        if(isLoginRequest(req)) {
            if("post".equalsIgnoreCase(req.getMethod())) {//form 表单提交
                boolean loginSuccess = login(req); //登录
```

```
                if(loginSuccess) {
                    return redirectToSuccessUrl(req, resp);
                }
            }
            return true;//继续过滤器链
        } else {//保存当前地址并重定向到登录界面
            saveRequestAndRedirectToLogin(req, resp);
            return false;
        }
    }
    private boolean redirectToSuccessUrl(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        WebUtils.redirectToSavedRequest(req, resp, successUrl);
        return false;
    }
    private void saveRequestAndRedirectToLogin(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        WebUtils.saveRequest(req);
        WebUtils.issueRedirect(req, resp, loginUrl);
    }

    private boolean login(HttpServletRequest req) {
        String username = req.getParameter("username");
        String password = req.getParameter("password");
        try {
            SecurityUtils.getSubject().login(new UsernamePasswordToken(username, password));
        } catch (Exception e) {
            req.setAttribute("shiroLoginFailure", e.getClass());
            return false;
        }
        return true;
    }
    private boolean isLoginRequest(HttpServletRequest req) {
        return pathsMatch(loginUrl, WebUtils.getPathWithinApplication(req));
    }
}
```

onPreHandle 主要流程：

1、首先判断是否已经登录过了，如果已经登录过了继续拦截器链即可；
2、如果没有登录，看看是否是登录请求，如果是 get 方法的登录页面请求，则继续拦截器链（到请求页面），否则如果是 get 方法的其他页面请求则保存当前请求并重定向到登录页面；
3、如果是 post 方法的登录页面表单提交请求，则收集用户名/密码登录即可，如果失败了保存错误消息到“shiroLoginFailure”并返回到登录页面；
4、如果登录成功了，且之前有保存的请求，则重定向到之前的这个请求，否则到默认的成功页面。

shiro.ini 配置

```
[filters]
formLogin=com.github.zhangkaitao.shiro.chapter8.web.filter.FormLoginFilter
[urls]
/test.jsp=formLogin
/login.jsp=formLogin
```

启动服务器输入 http://localhost:8080/chapter8/test.jsp 测试时，会自动跳转到登录页面，登录成功后又会跳回到 test.jsp 页面。

此处可以通过继承 AuthenticatingFilter 实现，其提供了很多登录相关的基础代码。另外可以参考 Shiro 内嵌的 FormAuthenticationFilter 的源码，思路是一样的。

6、任意角色授权拦截器

Shiro 提供 roles 拦截器，其验证用户拥有所有角色，没有提供验证用户拥有任意角色的拦截器。

```
public class AnyRolesFilter extends AccessControlFilter {
    private String unauthorizedUrl = "/unauthorized.jsp";
    private String loginUrl = "/login.jsp";
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response,
Object mappedValue) throws Exception {
        String[] roles = (String[])mappedValue;
        if(roles == null) {
            return true;//如果没有设置角色参数，默认成功
        }
        for(String role : roles) {
            if(getSubject(request, response).hasRole(role)) {
```

```
                return true;
            }
        }
        return false;//跳到 onAccessDenied 处理
    }

    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response)
throws Exception {
        Subject subject = getSubject(request, response);
        if (subject.getPrincipal() == null) {//表示没有登录，重定向到登录页面
            saveRequest(request);
            WebUtils.issueRedirect(request, response, loginUrl);
        } else {
            if (StringUtils.hasText(unauthorizedUrl)) {//如果有未授权页面跳转过去
                WebUtils.issueRedirect(request, response, unauthorizedUrl);
            } else {//否则返回 401 未授权状态码

WebUtils.toHttp(response).sendError(HttpServletResponse.SC_UNAUTHORIZED);
            }
        }
        return false;
    }
}
```

流程：

1、首先判断用户有没有任意角色，如果没有返回 false，将到 onAccessDenied 进行处理；

2、如果用户没有角色，接着判断用户有没有登录，如果没有登录先重定向到登录；

3、如果用户没有角色且设置了未授权页面（unauthorizedUrl），那么重定向到未授权页面；否则直接返回 401 未授权错误码。

shiro.ini 配置

```
[filters]
anyRoles=com.github.zhangkaitao.shiro.chapter8.web.filter.AnyRolesFilter
[urls]
/test.jsp=formLogin,anyRoles[admin,user]
/login.jsp=formLogin
```

此处可以继承 AuthorizationFilter 实现，其提供了授权相关的基础代码。另外可以参考 Shiro 内嵌的 RolesAuthorizationFilter 的源码，只是实现 hasAllRoles 逻辑。

默认拦截器

Shiro 内置了很多默认的拦截器，比如身份验证、授权等相关的。默认拦截器可以参考 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”
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有角色；主要属性： loginUrl：登录页面地址（/login.jsp）；unauthorizedUrl：未授权后重定向的地址；示例“/admin/**=roles[admin]”

http://jinnianshilongnian.iteye.com/

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；

另外还提供了一个 org.apache.shiro.web.filter.authz.HostFilter，即主机拦截器，比如其提供了属性：authorizedIps：已授权的 ip 地址，deniedIps：表示拒绝的 ip 地址；不过目前还没有完全实现，不可用。

这些默认的拦截器会自动注册，可以直接在 ini 配置文件中通过“拦截器名.属性”设置其属性：

```
perms.unauthorizedUrl=/unauthorized
```

另外如果某个拦截器不想使用了可以直接通过如下配置直接禁用：

```
perms.enabled=false
```

http://jinnianshilongnian.iteye.com/

第九章 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，<a href="${pageContext.request.contextPath}/login.jsp">登录</a>
</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，<a href="${pageContext.request.contextPath}/logout">退出</a>
</shiro:user>
```

用户已经身份验证/记住我登录后显示相应的信息。

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 type="java.lang.String"/>
```

相当于 Subject.getPrincipals().oneByType(String.class)。

```
<shiro:principal type="java.lang.String"/>
```

相当于 Subject.getPrincipals().oneByType(String.class)。

```
<shiro:principal property="username"/>
```

相当于((User)Subject.getPrincipals()).getUsername()。

hasRole 标签

```
<shiro:hasRole name="admin">
    用户[<shiro:principal/>]拥有角色 admin<br/>
</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<br/>
</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<br/>
</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<br/>
</shiro:hasPermission>
```

如果当前 Subject 有权限将显示 body 体内容。

lacksPermission 标签

```
<shiro:lacksPermission name="org:create">
    用户[<shiro:principal/>]没有权限 org:create<br/>
</shiro:lacksPermission>
```

如果当前 Subject 没有权限将显示 body 体内容。

另外又提供了几个权限控制相关的标签：

导入自定义标签库

```
<%@taglib prefix="zhang" tagdir="/WEB-INF/tags" %>
```

示例

```
<zhang:hasAllRoles name="admin,user">
    用户[<shiro:principal/>]拥有角色 admin 和 user<br/>
</zhang:hasAllRoles>
<zhang:hasAllPermissions name="user:create,user:update">
    用户[<shiro:principal/>]拥有权限 user:create 和 user:update<br/>
</zhang:hasAllPermissions>
<zhang:hasAnyPermissions name="user:create,abc:update">
    用户[<shiro:principal/>]拥有权限 user:create 或 abc:update<br/>
</zhang:hasAnyPermissions>
```

hasAllRoles 表示拥有所有相关的角色；hasAllPermissions 表示拥有所有相关的权限；hasAnyPermissions 表示拥有任意一个相关的权限。

另外可以参考我的《简单 shiro 扩展实现 NOT、AND、OR 权限验证》实现 NOT、AND、OR 权限验证：http://jinnianshilongnian.iteye.com/blog/1864800。

第十章　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 的会话支持不仅可以在普通的 JavaSE 应用中使用，也可以在 JavaEE 应用中使用，如 web 应用。且使用方式是一致的。

```
login("classpath:shiro.ini", "zhang", "123");
Subject subject = SecurityUtils.getSubject();
Session session = subject.getSession();
```

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。

```
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。

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。

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：

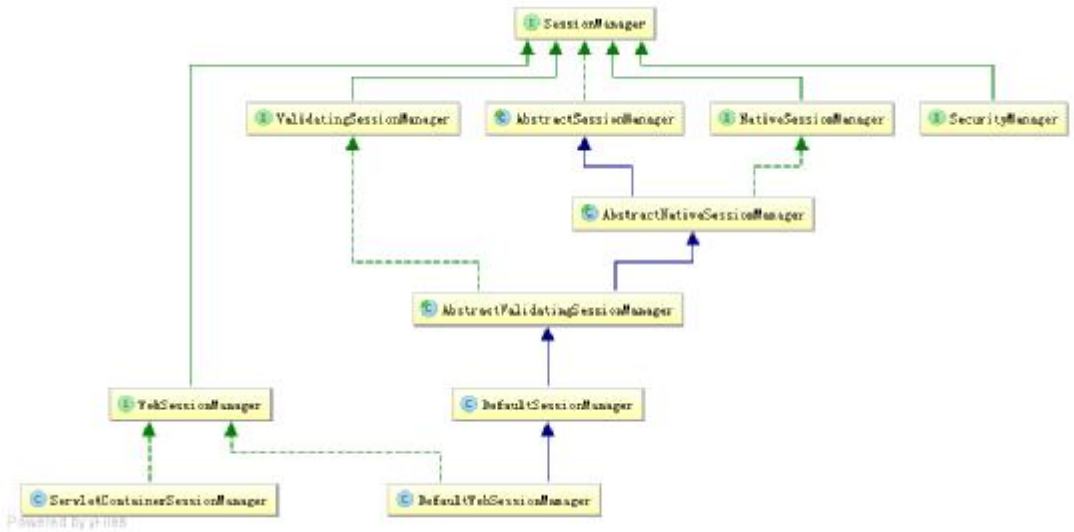
```
Session start(SessionContext context); //启动会话
Session getSession(SessionKey key) throws SessionException; //根据会话 Key 获取会话
```

另外用于 Web 环境的 WebSessionManager 又提供了如下接口：

```
boolean isServletContainerSessions();//是否使用 Servlet 容器的会话
```

Shiro 还提供了 ValidatingSessionManager 用于验资并过期会话：

```
void validateSessions();//验证所有会话是否过期
```



Shiro 提供了三个默认实现：

DefaultSessionManager：DefaultSecurityManager 使用的默认实现，用于 JavaSE 环境；

ServletContainerSessionManager：DefaultWebSecurityManager 使用的默认实现，用于 Web 环境，其直接使用 Servlet 容器的会话；

DefaultWebSessionManager：用于 Web 环境的实现，可以替代 ServletContainerSessionManager，自己维护着会话，直接废弃了 Servlet 容器的会话管理。

替换 SecurityManager 默认的 SessionManager 可以在 ini 中配置（shiro.ini）：

```
[main]
sessionManager=org.apache.shiro.session.mgt.DefaultSessionManager
securityManager.sessionManager=$sessionManager
```

Web 环境下的 ini 配置(shiro-web.ini)：

```
[main]
sessionManager=org.apache.shiro.web.session.mgt.ServletContainerSessionManager
securityManager.sessionManager=$sessionManager
```

另外可以设置会话的全局过期时间（毫秒为单位），默认 30 分钟：

```
sessionManager. globalSessionTimeout=1800000
```

默认情况下 globalSessionTimeout 将应用给所有 Session。可以单独设置每个 Session 的 timeout 属性来为每个 Session 设置其超时时间。

另外如果使用 ServletContainerSessionManager 进行会话管理，Session 的超时依赖于底层 Servlet 容器的超时时间，可以在 web.xml 中配置其会话的超时时间（分钟为单位）：

```
<session-config>
   <session-timeout>30</session-timeout>
</session-config>
```

在 Servlet 容器中，默认使用 JSESSIONID Cookie 维护会话，且会话默认是跟容器绑定的；在某些情况下可能需要使用自己的会话机制，此时我们可以使用 DefaultWebSessionManager 来维护会话：

```
sessionIdCookie=org.apache.shiro.web.servlet.SimpleCookie
sessionManager=org.apache.shiro.web.session.mgt.DefaultWebSessionManager
sessionIdCookie.name=sid
#sessionIdCookie.domain=sishuok.com
#sessionIdCookie.path=
sessionIdCookie.maxAge=1800
sessionIdCookie.httpOnly=true
sessionManager.sessionIdCookie=$sessionIdCookie
sessionManager.sessionIdCookieEnabled=true
securityManager.sessionManager=$sessionManager
```

sessionIdCookie 是 sessionManager 创建会话 Cookie 的模板：
sessionIdCookie.name：设置 Cookie 名字，默认为 JSESSIONID；
sessionIdCookie.domain：设置 Cookie 的域名，默认空，即当前访问的域名；
sessionIdCookie.path：设置 Cookie 的路径，默认空，即存储在域名根下；
sessionIdCookie.maxAge：设置 Cookie 的过期时间，秒为单位，默认-1 表示关闭浏览器时过期 Cookie；
sessionIdCookie.httpOnly：如果设置为 true，则客户端不会暴露给客户端脚本代码，使用 HttpOnly cookie 有助于减少某些类型的跨站点脚本攻击；此特性需要实现了 Servlet 2.5 MR6 及以上版本的规范的 Servlet 容器支持；
sessionManager.sessionIdCookieEnabled：是否启用/禁用 Session Id Cookie，默认是启用的；如果禁用后将不会设置 Session Id Cookie，即默认使用了 Servlet 容器的 JSESSIONID，且通过 URL 重写（URL 中的“;JSESSIONID=id”部分）保存 Session Id。

另外我们可以如“sessionManager. sessionIdCookie.name=sid”这种方式操作 Cookie 模板。

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());
    }
}
```

如果只想监听某一个事件，可以继承 SessionListenerAdapter 实现：

```
public class MySessionListener2 extends SessionListenerAdapter {
    @Override
    public void onStart(Session session) {
        System.out.println("会话创建：" + session.getId());
    }
}
```

在 shiro-web.ini 配置文件中可以进行如下配置设置会话监听器：

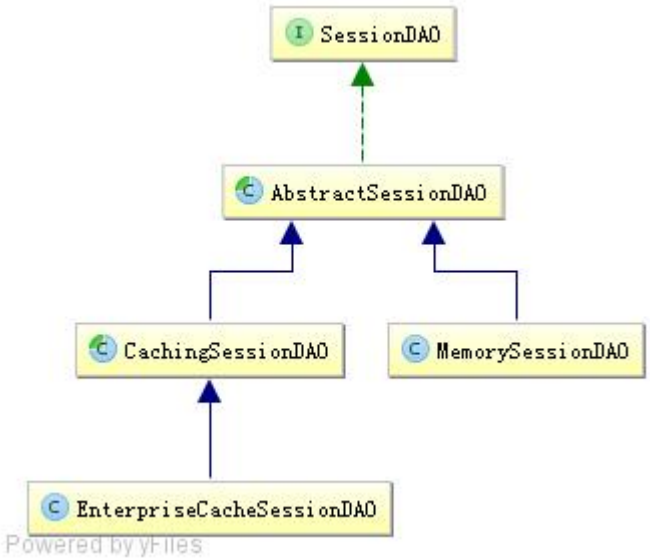
```
sessionListener1=com.github.zhangkaitao.shiro.chapter10.web.listener.MySessionListener1
sessionListener2=com.github.zhangkaitao.shiro.chapter10.web.listener.MySessionListener2
sessionManager.sessionListeners=$sessionListener1,$sessionListener2
```

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
```

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。

可以通过如下配置设置 SessionDAO：

```
sessionDAO=org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionManager.sessionDAO=$sessionDAO
```

Shiro 提供了使用 Ehcache 进行会话存储，Ehcache 可以配合 TerraCotta 实现容器无关的分布式集群。

首先在 pom.xml 里添加如下依赖：

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-ehcache</artifactId>
    <version>1.2.2</version>
</dependency>
```

接着配置 shiro-web.ini 文件：

```
sessionDAO=org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO. activeSessionsCacheName=shiro-activeSessionCache
sessionManager.sessionDAO=$sessionDAO
cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
cacheManager.cacheManagerConfigFile=classpath:ehcache.xml
securityManager.cacheManager = $cacheManager
```

sessionDAO. activeSessionsCacheName：设置 Session 缓存名字，默认就是 shiro-activeSessionCache；
cacheManager：缓存管理器，用于管理缓存的，此处使用 Ehcache 实现；
cacheManager.cacheManagerConfigFile：设置 ehcache 缓存的配置文件；
securityManager.cacheManager：设置 SecurityManager 的 cacheManager，会自动设置实现了 CacheManagerAware 接口的相应对象，如 SessionDAO 的 cacheManager；

然后配置 ehcache.xml：

```
<cache name="shiro-activeSessionCache"
       maxEntriesLocalHeap="10000"
       overflowToDisk="false"
       eternal="false"
       diskPersistent="false"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
       statistics="true"/>
```

Cache 的名字为 shiro-activeSessionCache，即设置的 sessionDAO 的 activeSessionsCacheName 属性值。

另外可以通过如下 ini 配置设置会话 ID 生成器：

```
sessionIdGenerator=org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator
sessionDAO.sessionIdGenerator=$sessionIdGenerator
```

用于生成会话 ID，默认就是 JavaUuidSessionIdGenerator，使用 java.util.UUID 生成。

如果自定义实现 SessionDAO，继承 CachingSessionDAO 即可：

```
public class MySessionDAO extends CachingSessionDAO {
    private JdbcTemplate jdbcTemplate = JdbcTemplateUtils.jdbcTemplate();
     protected Serializable doCreate(Session session) {
        Serializable sessionId = generateSessionId(session);
        assignSessionId(session, sessionId);
        String sql = "insert into sessions(id, session) values(?,?)";
        jdbcTemplate.update(sql, sessionId, SerializableUtils.serialize(session));
        return session.getId();
    }
    protected void doUpdate(Session session) {
        if(session instanceof ValidatingSession && !((ValidatingSession)session).isValid()) {
            return; //如果会话过期/停止 没必要再更新了
        }
        String sql = "update sessions set session=? where id=?";
        jdbcTemplate.update(sql, SerializableUtils.serialize(session), session.getId());
    }
    protected void doDelete(Session session) {
        String sql = "delete from sessions where id=?";
        jdbcTemplate.update(sql, session.getId());
    }
    protected Session doReadSession(Serializable sessionId) {
        String sql = "select session from sessions where id=?";
        List<String> sessionStrList = jdbcTemplate.queryForList(sql, String.class, sessionId);
        if(sessionStrList.size() == 0) return null;
```

doCreate/doUpdate/doDelete/doReadSession 分别代表创建/修改/删除/读取会话；此处通过把会话序列化后存储到数据库实现；接着在 shiro-web.ini 中配置：

```
sessionDAO=com.github.zhangkaitao.shiro.chapter10.session.dao.MySessionDAO
```

其他设置和之前一样，因为继承了 CachingSessionDAO；所有在读取时会先查缓存中是否存在，如果找不到才到数据库中查找。

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。

可以通过如下 ini 配置开启会话验证：

```
sessionValidationScheduler=org.apache.shiro.session.mgt.ExecutorServiceSessionValidationScheduler
sessionValidationScheduler.interval = 3600000
sessionValidationScheduler.sessionManager=$sessionManager
sessionManager.globalSessionTimeout=1800000
sessionManager.sessionValidationSchedulerEnabled=true
sessionManager.sessionValidationScheduler=$sessionValidationScheduler
```

sessionValidationScheduler：会话验证调度器，sessionManager 默认就是使用 ExecutorServiceSessionValidationScheduler，其使用 JDK 的 ScheduledExecutorService 进行定期调度并验证会话是否过期；
sessionValidationScheduler.interval：设置调度时间间隔，单位毫秒，默认就是 1 小时；
sessionValidationScheduler.sessionManager：设置会话验证调度器进行会话验证时的会话管理器；
sessionManager.globalSessionTimeout：设置全局会话超时时间，默认 30 分钟，即如果 30 分钟内没有访问会话将过期；
sessionManager.sessionValidationSchedulerEnabled：是否开启会话验证器，默认是开启的；
sessionManager.sessionValidationScheduler：设置会话验证调度器，默认就是使用 ExecutorServiceSessionValidationScheduler。

Shiro 也提供了使用 Quartz 会话验证调度器：

```
sessionValidationScheduler=org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler
sessionValidationScheduler.sessionValidationInterval = 3600000
sessionValidationScheduler.sessionManager=$sessionManager
```

使用时需要导入 shiro-quartz 依赖：

```
<dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-quartz</artifactId>
        <version>1.2.2</version>
</dependency>
```

如上会话验证调度器实现都是直接调用 AbstractValidatingSessionManager 的 validateSessions 方法进行验证，其直接调用 SessionDAO 的 getActiveSessions 方法获取所有会话进行验证，如果会话比较多，会影响性能；可以考虑如分页获取会话并进行验证，如 com.github.zhangkaitao.shiro.chapter10.session.scheduler.MySessionValidationScheduler：

```
//分页获取会话并验证
String sql = "select session from sessions limit ?,?";
int start = 0; //起始记录
int size = 20; //每页大小
List<String> sessionList = jdbcTemplate.queryForList(sql, String.class, start, size);
while(sessionList.size() > 0) {
   for(String sessionStr : sessionList) {
      try {
         Session session = SerializableUtils.deserialize(sessionStr);
         Method validateMethod =
            ReflectionUtils.findMethod(AbstractValidatingSessionManager.class,
                  "validate", Session.class, SessionKey.class);
         validateMethod.setAccessible(true);
         ReflectionUtils.invokeMethod(validateMethod,
            sessionManager, session, new DefaultSessionKey(session.getId()));
      } catch (Exception e) {
            //ignore
      }
   }
   start = start + size;
   sessionList = jdbcTemplate.queryForList(sql, String.class, start, size);
}
```

其直接改造自 ExecutorServiceSessionValidationScheduler，如上代码是验证的核心代码，可以根据自己的需求改造此验证调度器器；ini 的配置和之前的类似。

如果在会话过期时不想删除过期的会话，可以通过如下 ini 配置进行设置：

```
sessionManager.deleteInvalidSessions=false
```

默认是开启的，在会话过期后会调用 SessionDAO 的 delete 方法删除会话：如会话时持久化存储的，可以调用此方法进行删除。

如果是在获取会话时验证了会话已过期，将抛出 InvalidSessionException；因此需要捕获这个异常并跳转到相应的页面告诉用户会话已过期，让其重新登录，如可以在 web.xml 配置相应的错误页面：

```
<error-page>
    <exception-type>org.apache.shiro.session.InvalidSessionException</exception-type>
    <location>/invalidSession.jsp</location>
</error-page>
```

sessionFactory

sessionFactory 是创建会话的工厂，根据相应的 Subject 上下文信息来创建会话；默认提供了 SimpleSessionFactory 用来创建 SimpleSession 会话。

首先自定义一个 Session：

```
public class OnlineSession extends SimpleSession {
    public static enum OnlineStatus {
        on_line("在线"), hidden("隐身"), force_logout("强制退出");
        private final String info;
        private OnlineStatus(String info) {
            this.info = info;
        }
        public String getInfo() {
            return info;
        }
    }
    private String userAgent; //用户浏览器类型
    private OnlineStatus status = OnlineStatus.on_line; //在线状态
    private String systemHost; //用户登录时系统 IP
    //省略其他
}
```

OnlineSession 用于保存当前登录用户的在线状态，支持如离线等状态的控制。

接着自定义 SessionFactory：

```
public class OnlineSessionFactory implements SessionFactory {

    @Override
    public Session createSession(SessionContext initData) {
        OnlineSession session = new OnlineSession();
        if (initData != null && initData instanceof WebSessionContext) {
            WebSessionContext sessionContext = (WebSessionContext) initData;
            HttpServletRequest request = (HttpServletRequest) sessionContext.getServletRequest();
            if (request != null) {
                session.setHost(IpUtils.getIpAddr(request));
                session.setUserAgent(request.getHeader("User-Agent"));
                session.setSystemHost(request.getLocalAddr() + ":" + request.getLocalPort());
            }
        }
        return session;
    }
}
```

根据会话上下文创建相应的 OnlineSession。

最后在 shiro-web.ini 配置文件中配置：

```
sessionFactory=org.apache.shiro.session.mgt.OnlineSessionFactory
sessionManager.sessionFactory=$sessionFactory
```

更多请参考 https://github.com/zhangkaitao/es/tree/master/web/src/main/java/org/apache/shiro 中的相关代码。

http://j...

第十一章　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。对于 Cache 的一些概念可以参考我的《Spring Cache 抽象详解》：http://jinnianshilongnian.iteye.com/blog/2001040。

Shiro 提供的 Cache 接口：

```
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;
    //返回缓存大小
    public int size();
    //获取缓存中所有的 key
    public Set<K> keys();
    //获取缓存中所有的 value
    public Collection<V> values();
}
```

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。

本章用例使用了与第六章的代码。

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。

ini 配置

```
userRealm=com.github.zhangkaitao.shiro.chapter11.realm.UserRealm
userRealm.credentialsMatcher=$credentialsMatcher
userRealm.cachingEnabled=true
userRealm.authenticationCachingEnabled=true
userRealm.authenticationCacheName=authenticationCache
userRealm.authorizationCachingEnabled=true
userRealm.authorizationCacheName=authorizationCache
securityManager.realms=$userRealm

cacheManager=org.apache.shiro.cache.ehcache.EhCacheManager
cacheManager.cacheManagerConfigFile=classpath:shiro-ehcache.xml
securityManager.cacheManager=$cacheManager
```

userRealm.cachingEnabled：启用缓存，默认 false；
userRealm.authenticationCachingEnabled：启用身份验证缓存，即缓存 AuthenticationInfo 信息，默认 false；
userRealm.authenticationCacheName：缓存 AuthenticationInfo 信息的缓存名称；
userRealm. authorizationCachingEnabled：启用授权缓存，即缓存 AuthorizationInfo 信息，默认 false；
userRealm. authorizationCacheName：缓存 AuthorizationInfo 信息的缓存名称；
cacheManager：缓存管理器，此处使用 EhCacheManager，即 Ehcache 实现，需要导入相应的 Ehcache 依赖，请参考 pom.xml；

因为测试用例的关系，需要将 Ehcache 的 CacheManager 改为使用 VM 单例模式：
this.manager = new net.sf.ehcache.CacheManager(getCacheManagerConfigFileInputStream());
改为

this.manager = net.sf.ehcache.CacheManager.create(getCacheManagerConfigFileInputStream());

测试用例

```
@Test
public void testClearCachedAuthenticationInfo() {
    login(u1.getUsername(), password);
    userService.changePassword(u1.getId(), password + "1");

    RealmSecurityManager securityManager =
        (RealmSecurityManager) SecurityUtils.getSecurityManager();
    UserRealm userRealm = (UserRealm) securityManager.getRealms().iterator().next();
    userRealm.clearCachedAuthenticationInfo(subject().getPrincipals());

    login(u1.getUsername(), password + "1");
}
```

首先登录成功（此时会缓存相应的 AuthenticationInfo），然后修改密码；此时密码就变了；接着需要调用 Realm 的 clearCachedAuthenticationInfo 方法清空之前缓存的 AuthenticationInfo；否则下次登录时还会获取到修改密码之前的那个 AuthenticationInfo；

```
@Test
public void testClearCachedAuthorizationInfo() {
    login(u1.getUsername(), password);
    subject().checkRole(r1.getRole());
    userService.correlationRoles(u1.getId(), r2.getId());

    RealmSecurityManager securityManager =
       (RealmSecurityManager) SecurityUtils.getSecurityManager();
    UserRealm userRealm = (UserRealm) securityManager.getRealms().iterator().next();
    userRealm.clearCachedAuthorizationInfo(subject().getPrincipals());

    subject().checkRole(r2.getRole());
}
```

和之前的用例差不多；此处调用 Realm 的 clearCachedAuthorizationInfo 清空之前缓存的 AuthorizationInfo；

另外还有 clearCache，其同时调用 clearCachedAuthenticationInfo 和 clearCachedAuthorizationInfo，清空 AuthenticationInfo 和 AuthorizationInfo。

UserRealm 还提供了 clearAllCachedAuthorizationInfo、clearAllCachedAuthenticationInfo、clearAllCache，用于清空整个缓存。

在某些清空下这种方式可能不是最好的选择，可以考虑直接废弃 Shiro 的缓存，然后自己通过如 AOP 机制实现自己的缓存；可以参考：

https://github.com/zhangkaitao/es/tree/master/web/src/main/java/com/sishuok/es/extra/aop

另外如果和 Spring 集成时可以考虑直接使用 Spring 的 Cache 抽象，可以考虑使用 SpringCacheManagerWrapper，其对 Spring Cache 进行了包装，转换为 Shiro 的 CacheManager 实现：

https://github.com/zhangkaitao/es/blob/master/web/src/main/java/org/apache/shiro/cache/spring/SpringCacheManagerWrapper.java

Session 缓存

当我们设置了 SecurityManager 的 CacheManager 时，如：

```
securityManager.cacheManager=$cacheManager
```

当我们设置 SessionManager 时：

```
sessionManager=org.apache.shiro.session.mgt.DefaultSessionManager
securityManager.sessionManager=$sessionManager
```

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；如第九章的 MySessionDAO 就是带缓存的 SessionDAO；其会先查缓存，如果找不到才查数据库。

对于 CachingSessionDAO，可以通过如下配置设置缓存的名称：

```
sessionDAO=com.github.zhangkaitao.shiro.chapter11.session.dao.MySessionDAO
sessionDAO.activeSessionsCacheName=shiro-activeSessionCache
```

activeSessionsCacheName 默认就是 shiro-activeSessionCache。

第十二章　与 Spring 集成

Shiro 的组件都是 JavaBean/POJO 式的组件，所以非常容易使用 Spring 进行组件管理，可以非常方便的从 ini 配置迁移到 Spring 进行管理，且支持 JavaSE 应用及 Web 应用的集成。

在示例之前，需要导入 shiro-spring 及 spring-context 依赖，具体请参考 pom.xml。
spring-beans.xml 配置文件提供了基础组件如 DataSource、DAO、Service 组件的配置。

JavaSE 应用

spring-shiro.xml 提供了普通 JavaSE 独立应用的 Spring 配置：

```
<!-- 缓存管理器 使用 Ehcache 实现 -->
<bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">
    <property name="cacheManagerConfigFile" value="classpath:ehcache.xml"/>
</bean>

<!-- 凭证匹配器 -->
<bean id="credentialsMatcher" class="
  com.github.zhangkaitao.shiro.chapter12.credentials.RetryLimitHashedCredentialsMatcher">
    <constructor-arg ref="cacheManager"/>
    <property name="hashAlgorithmName" value="md5"/>
    <property name="hashIterations" value="2"/>
    <property name="storedCredentialsHexEncoded" value="true"/>
</bean>

<!-- Realm 实现 -->
<bean id="userRealm" class="com.github.zhangkaitao.shiro.chapter12.realm.UserRealm">
    <property name="userService" ref="userService"/>
    <property name="credentialsMatcher" ref="credentialsMatcher"/>
    <property name="cachingEnabled" value="true"/>
    <property name="authenticationCachingEnabled" value="true"/>
    <property name="authenticationCacheName" value="authenticationCache"/>
    <property name="authorizationCachingEnabled" value="true"/>
    <property name="authorizationCacheName" value="authorizationCache"/>
</bean>
```

```
<!-- 会话 ID 生成器 -->
<bean id="sessionIdGenerator"
   class="org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator"/>
<!-- 会话 DAO -->
<bean id="sessionDAO"
   class="org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO">
      <property name="activeSessionsCacheName" value="shiro-activeSessionCache"/>
      <property name="sessionIdGenerator" ref="sessionIdGenerator"/>
</bean>
<!-- 会话验证调度器 -->
<bean id="sessionValidationScheduler"
   class="org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler">
      <property name="sessionValidationInterval" value="1800000"/>
      <property name="sessionManager" ref="sessionManager"/>
</bean>
<!-- 会话管理器 -->
<bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager">
      <property name="globalSessionTimeout" value="1800000"/>
      <property name="deleteInvalidSessions" value="true"/>
      <property name="sessionValidationSchedulerEnabled" value="true"/>
      <property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>
      <property name="sessionDAO" ref="sessionDAO"/>
</bean>
<!-- 安全管理器 -->
<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">
      <property name="realms">
          <list><ref bean="userRealm"/></list>
      </property>
      <property name="sessionManager" ref="sessionManager"/>
      <property name="cacheManager" ref="cacheManager"/>
</bean>
<!-- 相当于调用 SecurityUtils.setSecurityManager(securityManager) -->
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
      <property name="staticMethod"
        value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
      <property name="arguments" ref="securityManager"/>
</bean>
```

```
<!-- Shiro 生命周期处理器-->
<bean id="lifecycleBeanPostProcessor"
   class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>
```

可以看出，只要把之前的 ini 配置翻译为此处的 spring xml 配置方式即可，无须多解释。LifecycleBeanPostProcessor 用于在实现了 Initializable 接口的 Shiro bean 初始化时调用 Initializable 接口回调，在实现了 Destroyable 接口的 Shiro bean 销毁时调用 Destroyable 接口回调。如 UserRealm 就实现了 Initializable，而 DefaultSecurityManager 实现了 Destroyable。具体可以查看它们的继承关系。

测试用例请参考 com.github.zhangkaitao.shiro.chapter12.ShiroTest。

Web 应用

Web 应用和普通 JavaSE 应用的某些配置是类似的，此处只提供一些不一样的配置，详细配置可以参考 spring-shiro-web.xml。

```
<!-- 会话 Cookie 模板 -->
<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
      <constructor-arg value="sid"/>
      <property name="httpOnly" value="true"/>
      <property name="maxAge" value="180000"/>
</bean>
<!-- 会话管理器 -->
<bean id="sessionManager"
   class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">
      <property name="globalSessionTimeout" value="1800000"/>
      <property name="deleteInvalidSessions" value="true"/>
      <property name="sessionValidationSchedulerEnabled" value="true"/>
      <property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>
      <property name="sessionDAO" ref="sessionDAO"/>
      <property name="sessionIdCookieEnabled" value="true"/>
      <property name="sessionIdCookie" ref="sessionIdCookie"/>
</bean>
<!-- 安全管理器 -->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
      <property name="realm" ref="userRealm"/>
      <property name="sessionManager" ref="sessionManager"/>
```

```
    <property name="cacheManager" ref="cacheManager"/>
</bean>
```

1、sessionIdCookie 是用于生产 Session ID Cookie 的模板；
2、会话管理器使用用于 web 环境的 DefaultWebSessionManager；
3、安全管理器使用用于 web 环境的 DefaultWebSecurityManager。

```
<!-- 基于 Form 表单的身份验证过滤器 -->
<bean id="formAuthenticationFilter"
   class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">
    <property name="usernameParam" value="username"/>
    <property name="passwordParam" value="password"/>
    <property name="loginUrl" value="/login.jsp"/>
</bean>
<!-- Shiro 的 Web 过滤器 -->
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="/login.jsp"/>
    <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
    <property name="filters">
        <util:map>
            <entry key="authc" value-ref="formAuthenticationFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            /index.jsp = anon
            /unauthorized.jsp = anon
            /login.jsp = authc
            /logout = logout
            /** = user
        </value>
    </property>
</bean>
```

1、formAuthenticationFilter 为基于 Form 表单的身份验证过滤器；此处可以再添加自己的 Filter bean 定义；

2、shiroFilter：此处使用 ShiroFilterFactoryBean 来创建 ShiroFilter 过滤器；filters 属性用于定义自己的过滤器，即 ini 配置中的[filters]部分；filterChainDefinitions 用于声明 url 和 filter 的关系，即 ini 配置中的[urls]部分。

接着需要在 web.xml 中进行如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:spring-beans.xml,
        classpath:spring-shiro-web.xml
    </param-value>
</context-param>
<listener>
   <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

通过 ContextLoaderListener 加载 contextConfigLocation 指定的 Spring 配置文件。

```
<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 会自动到 Spring 容器中查找名字为 shiroFilter 的 bean 并把 filter 请求交给它处理。

其他配置请参考源代码。

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。
为了测试，此处使用了 Spring MVC 来测试 Shiro 注解，当然 Shiro 注解不仅仅可以在 web 环境使用，在独立的 JavaSE 中也是可以用的，此处只是以 web 为例了。

在 spring-mvc.xml 配置文件添加 Shiro Spring AOP 权限注解的支持：

```
<aop:config proxy-target-class="true"></aop:config>
<bean class="
   org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
      <property name="securityManager" ref="securityManager"/>
</bean>
```

如上配置用于开启 Shiro Spring AOP 权限注解的支持；<aop:config proxy-target-class="true">表示代理类。

接着就可以在相应的控制器（AnnotationController）中使用如下方式进行注解：

```
@RequiresRoles("admin")
@RequestMapping("/hello2")
public String hello2() {
      return "success";
}
```

访问 hello2 方法的前提是当前用户有 admin 角色。

当验证失败，其会抛出 UnauthorizedException 异常，此时可以使用 Spring 的 ExceptionHandler（DefaultExceptionHandler）来进行拦截处理：

```
@ExceptionHandler({UnauthorizedException.class})
@ResponseStatus(HttpStatus.UNAUTHORIZED)
public ModelAndView processUnauthenticatedException(NativeWebRequest request,
UnauthorizedException e) {
      ModelAndView mv = new ModelAndView();
      mv.addObject("exception", e);
      mv.setViewName("unauthorized");
      return mv;
}
```

如果集成 Struts2，需要注意《Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired 失效》问题：
http://jinnianshilongnian.iteye.com/blog/1850425

权限注解

@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。

http://jinnianshilongnian.iteye.com/

第十三章　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。

RememberMe 配置

spring-shiro-web.xml 配置：

```
<!-- 会话 Cookie 模板 -->
<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
    <constructor-arg value="sid"/>
    <property name="httpOnly" value="true"/>
    <property name="maxAge" value="-1"/>
</bean>
<bean id="rememberMeCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
    <constructor-arg value="rememberMe"/>
    <property name="httpOnly" value="true"/>
    <property name="maxAge" value="2592000"/><!-- 30 天 -->
</bean>
```

sessionIdCookie：maxAge=-1 表示浏览器关闭时失效此 Cookie；
rememberMeCookie：即记住我的 Cookie，保存时长 30 天；

```
<!-- rememberMe 管理器 -->
<bean id="rememberMeManager"
   class="org.apache.shiro.web.mgt.CookieRememberMeManager">
    <property name="cipherKey" value="
          #{T(org.apache.shiro.codec.Base64).decode('4AvVhmFLUs0KTA3Kprsdag==')}"/>
     <property name="cookie" ref="rememberMeCookie"/>
</bean>
```

rememberMe 管理器，cipherKey 是加密 rememberMe Cookie 的密钥；默认 AES 算法；

```
<!-- 安全管理器 -->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    ……
    <property name="rememberMeManager" ref="rememberMeManager"/>
</bean>
```

设置 securityManager 安全管理器的 rememberMeManager；

```
<bean id="formAuthenticationFilter"
  class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">
    ……
    <property name="rememberMeParam" value="rememberMe"/>
</bean>
```

rememberMeParam，即 rememberMe 请求参数名，请求参数是 boolean 类型，true 表示 rememberMe。

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    ……
    <property name="filterChainDefinitions">
        <value>
            /login.jsp = authc
            /logout = logout
            /authenticated.jsp = authc
            /** = user
        </value>
    </property>
</bean>
```

“/authenticated.jsp = authc”表示访问该地址用户必须身份验证通过（Subject. isAuthenticated()==true）；而“/** = user”表示访问该地址的用户是身份验证通过或 RememberMe 登录的都可以。

测试：

1、访问 http://localhost:8080/chapter13/，会跳转到登录页面，登录成功后会设置会话及 rememberMe Cookie；

2、关闭浏览器，此时会话 cookie 将失效；

3、然后重新打开浏览器访问 http://localhost:8080/chapter13/，还是可以访问的；

4、如果此时访问 http://localhost:8080/chapter13/authenticated.jsp，会跳转到登录页面重新进行身份验证。

如果要自己做 RememeberMe，需要在登录之前这样创建 Token：UsernamePasswordToken(用户名，密码，是否记住我)，如：

```
Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken(username, password);
token.setRememberMe(true);
subject.login(token);
```

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。

另外对于过滤器，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。

http://jinnianshilongnian.iteye.com

第十四章　SSL

对于 SSL 的支持，Shiro 只是判断当前 url 是否需要 SSL 登录，如果需要自动重定向到 https 进行访问。

首先生成数字证书，生成证书到 D:\localhost.keystore

使用 JDK 的 keytool 命令，生成证书（包含证书/公钥/私钥）到 D:\localhost.keystore：

```
keytool -genkey -keystore "D:\localhost.keystore" -alias localhost -keyalg RSA
输入密钥库口令:
再次输入新口令:
您的名字与姓氏是什么?
  [Unknown]:  localhost
您的组织单位名称是什么?
  [Unknown]:  sishuok.com
您的组织名称是什么?
  [Unknown]:  sishuok.com
您所在的城市或区域名称是什么?
  [Unknown]:  beijing
您所在的省/市/自治区名称是什么?
  [Unknown]:  beijing
该单位的双字母国家/地区代码是什么?
  [Unknown]:  cn
CN=localhost, OU=sishuok.com, O=sishuok.com, L=beijing, ST=beijing, C=cn 是否正确
?
  [否]:  y

输入 <localhost> 的密钥口令
        (如果和密钥库口令相同, 按回车):
再次输入新口令:
```

通过如上步骤，生成证书到 D:\ localhost.keystore；

然后设置 tomcat 下的 server.xml

此处使用了 apache-tomcat-7.0.40 版本，打开 conf/server.xml，找到：

http://jinnianshilongnian.iteye.com/

```
<!--
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS" />
-->
```

替换为

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS"
           keystoreFile="D:\localhost.keystore" keystorePass="123456"/>
```

keystorePass 就是生成 keystore 时设置的密码。

添加 SSL 到配置文件（spring-shiro-web.xml）

此处使用了和十三章一样的代码：

```
<bean id="sslFilter" class="org.apache.shiro.web.filter.authz.SslFilter">
    <property name="port" value="8443"/>
</bean>
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    ……
    <property name="filters">
        <util:map>
            <entry key="authc" value-ref="formAuthenticationFilter"/>
            <entry key="ssl" value-ref="sslFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            /login.jsp = ssl,authc
            /logout = logout
            /authenticated.jsp = authc
            /** = user
        </value>
    </property>
</bean>
```

SslFilter 默认端口是 443，此处使用了 8443；“/login.jsp = ssl,authc”表示访问登录页面时需要走 SSL。

测试

最后把 shiro-example-chapter14 打成 war 包（mvn:package），放到 tomcat 下的 webapps 中，启动服务器测试，如访问 localhost:9080/chapter14/，会自动跳转到 https://localhost:8443/chapter14/login.jsp。

如果使用 Maven Jetty 插件，可以直接如下插件配置：

```
<plugin>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>8.1.8.v20121106</version>
    <configuration>
      <webAppConfig>
        <contextPath>/${project.build.finalName}</contextPath>
      </webAppConfig>
      <connectors>
      <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
        <port>8080</port>
      </connector>
      <connector implementation="org.eclipse.jetty.server.ssl.SslSocketConnector">
        <port>8443</port>
        <keystore>${project.basedir}/localhost.keystore</keystore>
        <password>123456</password>
        <keyPassword>123456</keyPassword>
      </connector>
      </connectors>
    </configuration>
</plugin>
```

http://

第十五章　单点登录

Shiro 1.2 开始提供了 Jasig CAS 单点登录的支持，单点登录主要用于多系统集成，即在多个系统中，用户只需要到一个中央服务器登录一次即可访问这些系统中的任何一个，无须多次登录。此处我们使用 Jasig CAS v4.0.0-RC3 版本：
https://github.com/Jasig/cas/tree/v4.0.0-RC3

Jasig CAS 单点登录系统分为服务器端和客户端，服务器端提供单点登录，多个客户端（子系统）将跳转到该服务器进行登录验证，大体流程如下：
1、访问客户端需要登录的页面 http://localhost:9080/ client/，此时会跳到单点登录服务器 https://localhost:8443/ server/login?service=https://localhost:9443/ client/cas；
2、如果此时单点登录服务器也没有登录的话，会显示登录表单页面，输入用户名/密码进行登录；
3、登录成功后服务器端会回调客户端传入的地址：https://localhost:9443/client/cas?ticket=ST-1-eh2cIo92F9syvoMs5DOg-cas01.example.org，且带着一个 ticket；
4、客户端会把 ticket 提交给服务器来验证 ticket 是否有效；如果有效服务器端将返回用户身份；
5、客户端可以再根据这个用户身份获取如当前系统用户/角色/权限信息。

本章使用了和《第十四章 SSL》一样的数字证书。

服务器端

我们使用了 Jasig CAS 服务器 v4.0.0-RC3 版本，可以到其官方的 github 下载：https://github.com/Jasig/cas/tree/v4.0.0-RC3 下载，然后将其 cas-server-webapp 模块封装到 shiro-example-chapter15-server 模块中，具体请参考源码。

1、数字证书使用和《第十四章 SSL》一样的数字证书，即将 localhost.keystore 拷贝到 shiro-example-chapter15-server 模块根目录下；
2、在 pom.xml 中添加 Jetty Maven 插件，并添加 SSL 支持：

http://jinnianshilongnian.iteye.com/

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.8.v20121106</version>
  <configuration>
    <webAppConfig>
      <contextPath>/${project.build.finalName}</contextPath>
    </webAppConfig>
    <connectors>
      <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
        <port>8080</port>
      </connector>
      <connector implementation="org.eclipse.jetty.server.ssl.SslSocketConnector">
        <port>8443</port>
        <keystore>${project.basedir}/localhost.keystore</keystore>
        <password>123456</password>
        <keyPassword>123456</keyPassword>
      </connector>
    </connectors>
  </configuration>
</plugin>
```

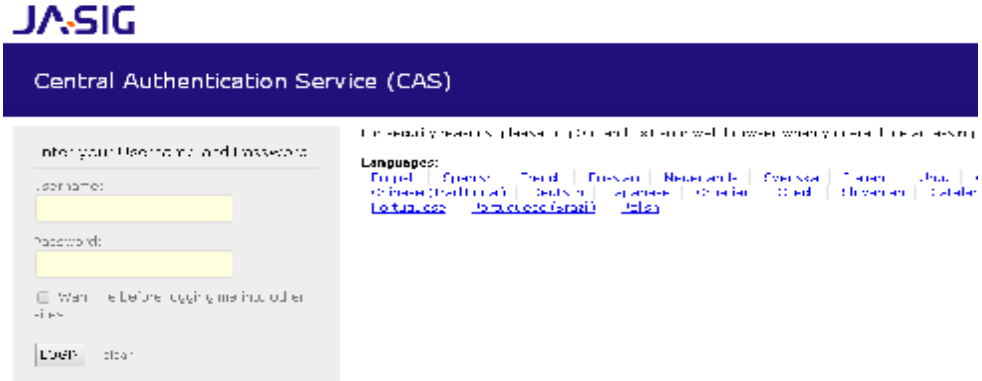
3、修改 src/main/webapp/WEB-INF/deployerConfigContext.xml，找到 primaryAuthenticationHandler，然后添加一个账户：

```
<entry key="zhang" value="123"/>
```

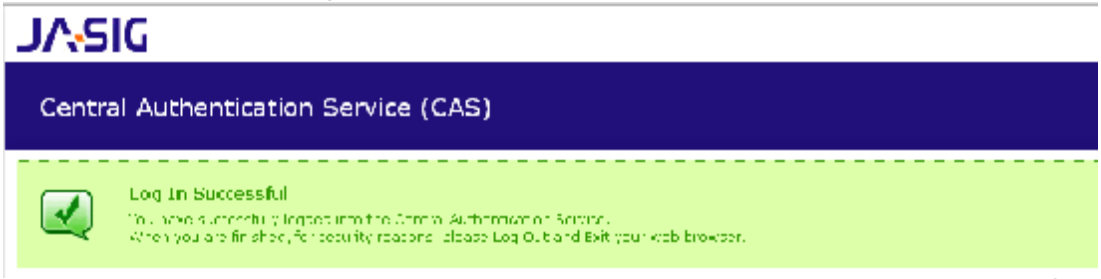
其也支持如 JDBC 查询，可以自己定制；具体请参考文档。

4、mvn jetty:run 启动服务器测试即可；

访问 https://localhost:8443/chapter15-server/login 将弹出如下登录页面：



输入用户名/密码，如 zhang/123，将显示登录成功页面：



到此服务器端的简单配置就完成了。

客户端

1、首先使用 localhost.keystore 导出数字证书（公钥）到 D:\localhost.cer

```
keytool -export -alias localhost -file D:\localhost.cer -keystore D:\localhost.keystore
```

2、因为 CAS client 需要使用该证书进行验证，需要将证书导入到 JDK 中：

```
cd D:\jdk1.7.0_21\jre\lib\security
keytool -import -alias localhost -file D:\localhost.cer -noprompt -trustcacerts -storetype jks -keystore cacerts -storepass 123456
```

如果导入失败，可以先把 security 目录下的 cacerts 删掉；

3、按照服务器端的 Jetty Maven 插件的配置方式配置 Jetty 插件；

4、在 shiro-example-chapter15-client 模块中导入 shiro-cas 依赖，具体请参考其 pom.xml；

5、自定义 CasRealm：

```
public class MyCasRealm extends CasRealm {
    private UserService userService;
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        String username = (String)principals.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
```

```
            authorizationInfo.setRoles(userService.findRoles(username));
            authorizationInfo.setStringPermissions(userService.findPermissions(username));
            return authorizationInfo;
        }
}
```

CasRealm 根据 CAS 服务器端返回的用户身份获取相应的角色/权限信息。

6、spring-shiro-web.xml 配置：

```
<bean id="casRealm" class="com.github.zhangkaitao.shiro.chapter13.realm.MyCasRealm">
    <property name="userService" ref="userService"/>
    ……
    <property name="casServerUrlPrefix" value="https://localhost:8443/chapter14-server"/>
    <property name="casService" value="https://localhost:9443/chapter14-client/cas"/>
</bean>
```

casServerUrlPrefix：是 CAS Server 服务器端地址；
casService：是当前应用 CAS 服务 URL，即用于接收并处理登录成功后的 Ticket 的；

如果角色/权限信息是由服务器端提供的话，我们可以直接使用 CasRealm：

```
<bean id="casRealm" class="org.apache.shiro.cas.CasRealm">
    ……
    <property name="defaultRoles" value="admin,user"/>
    <property name="defaultPermissions" value="user:create,user:update"/>
    <property name="roleAttributeNames" value="roles"/>
    <property name="permissionAttributeNames" value="permissions"/>
    <property name="casServerUrlPrefix" value="https://localhost:8443/chapter14-server"/>
    <property name="casService" value="https://localhost:9443/chapter14-client/cas"/>
</bean>
```

defaultRoles/ defaultPermissions：默认添加给所有 CAS 登录成功用户的角色和权限信息；
roleAttributeNames/ permissionAttributeNames：角色属性/权限属性名称，如果用户的角色/权限信息是从服务器端返回的（即返回的 CAS Principal 中除了 Principal 之外还有如一些 Attributes），此时可以使用 roleAttributeNames/ permissionAttributeNames 得到 Attributes 中的角色/权限数据；请自行查询 CAS 获取用户更多信息。

```
<bean id="casFilter" class="org.apache.shiro.cas.CasFilter">
    <property name="failureUrl" value="/casFailure.jsp"/>
</bean>
```

CasFilter 类似于 FormAuthenticationFilter，只不过其验证服务器端返回的 CAS Service Ticket。

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="https://localhost:8443/chapter14-server/login?service=https://localhost:9443/chapter14-client/cas"/>
    <property name="successUrl" value="/"/>
    <property name="filters">
        <util:map>
            <entry key="cas" value-ref="casFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            /casFailure.jsp = anon
            /cas = cas
            /logout = logout
            /** = user
        </value>
    </property>
</bean>
```

loginUrl：https://localhost:8443/chapter15-server/login 表示服务端端登录地址，登录成功后跳转到?service 参数对于的地址进行客户端验证及登录；
“/cas=cas”：即/cas 地址是服务器端回调地址，使用 CasFilter 获取 Ticket 进行登录。

7、测试，输入 http://localhost:9080/chapter15-client 地址进行测试即可，可以使用如 Chrome 开这 debug 观察网络请求的变化。

如果遇到以下异常，一般是证书导入错误造成的，请尝试重新导入，如果还是不行，有可能是运行应用的 JDK 和安装数字证书的 JDK 不是同一个造成的：

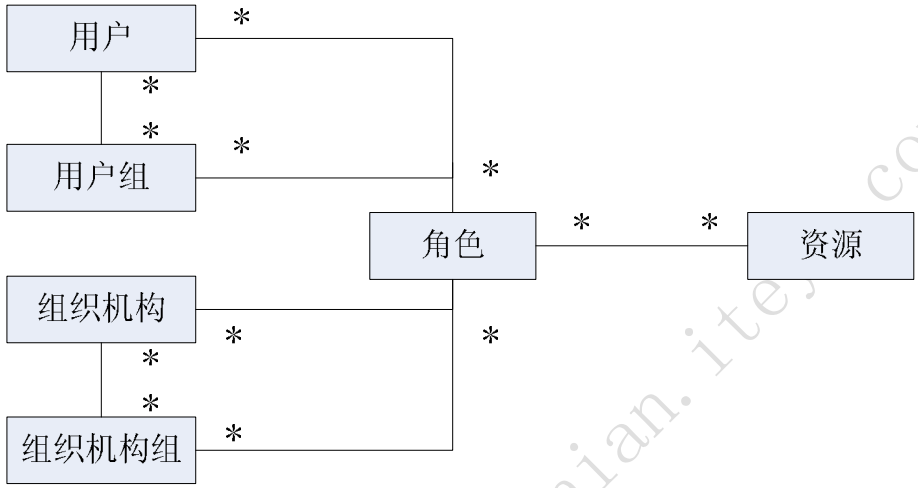
```
Caused by: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
    at sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:385)
    at sun.security.validator.PKIXValidator.engineValidate(PKIXValidator.java:292)
    at sun.security.validator.Validator.validate(Validator.java:260)
    at sun.security.ssl.X509TrustManagerImpl.validate(X509TrustManagerImpl.java:326)
```

```
        at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:231)
        at
sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:126)
        at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1323)
        ... 67 more
Caused by: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
        at
sun.security.provider.certpath.SunCertPathBuilder.engineBuild(SunCertPathBuilder.java:196)
        at java.security.cert.CertPathBuilder.build(CertPathBuilder.java:268)
        at sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:380)
        ... 73 more
```

http://jinnianshilongnian.iteye.com/

第十六章　综合实例

简单的实体关系图



简单数据字典

用户(sys_user)

名称	类型	长度	描述
id	bigint		编号 主键
username	varchar	100	用户名
password	varchar	100	密码
salt	varchar	50	盐
role_ids	varchar	100	角色列表
locked	bool		账户是否锁定

组织机构(sys_organization)

名称	类型	长度	描述
id	bigint		编号 主键
name	varchar	100	组织机构名
priority	int		显示顺序
parent_id	bigint		父编号
parent_ids	varchar	100	父编号列表
available	bool		是否可用

资源(sys_resource)

名称	类型	长度	描述
id	bigint		编号 主键

http://jinnianshilongnian.iteye.com/

name	varchar	100	资源名称
type	varchar	50	资源类型，
priority	int		显示顺序
parent_id	bigint		父编号
parent_ids	varchar	100	父编号列表
permission	varchar	100	权限字符串
available	bool		是否可用

角色(sys_role)

名称	类型	长度	描述
id	bigint		编号 主键
role	varchar	100	角色名称
description	varchar	100	角色描述
resource_ids	varchar	100	授权的资源
available	bool		是否可用

资源：表示菜单元素、页面按钮元素等；菜单元素用来显示界面菜单的，页面按钮是每个页面可进行的操作，如新增、修改、删除按钮；使用 type 来区分元素类型（如 menu 表示菜单，button 代表按钮），priority 是元素的排序，如菜单显示顺序；permission 表示权限；如用户菜单使用 user:*；也就是把菜单授权给用户后，用户就拥有了 user:*权限；如用户新增按钮使用 user:create，也就是把用户新增按钮授权给用户后，用户就拥有了 user:create 权限了；available 表示资源是否可用，如菜单显示/不显示。

角色：role 表示角色标识符，如 admin，用于后台判断使用；description 表示角色描述，如超级管理员，用于前端显示给用户使用；resource_ids 表示该角色拥有的资源列表，即该角色拥有的权限列表（显示角色），即角色是权限字符串集合；available 表示角色是否可用。

组织机构：name 表示组织机构名称，priority 是组织机构的排序，即显示顺序；available 表示组织机构是否可用。

用户：username 表示用户名；password 表示密码；salt 表示加密密码的盐；role_ids 表示用户拥有的角色列表，可以通过角色再获取其权限字符串列表；locked 表示用户是否锁定。

此处如资源、组织机构都是树型结构：

id	name	parent_id	parent_ids
1	总公司	0	0/
2	山东分公司	1	0/1/
3	河北分公司	1	0/1/
4	济南分公司	2	0/1/2/

parent_id 表示父编号，parent_ids 表示所有祖先编号；如 0/1/2/表示其祖先是 2、1、0；其中根节点父编号为 0。

为了简单性，如用户-角色，角色-资源关系直接在实体（用户表中的 role_ids，角色表中的 resource_ids）里完成的，没有建立多余的关系表，如要查询拥有 admin 角色的用户时，建议建立关联表，否则就没必要建立了。在存储关系时如 role_ids=1,2,3,；多个之间使用逗号分隔。

用户组、组织机构组本实例没有实现，即可以把一组权限授权给这些组，组中的用户/组织机构就自动拥有这些角色/权限了；另外对于用户组可以实现一个默认用户组，如论坛，不管匿名/登录用户都有查看帖子的权限。

更复杂的权限请参考我的《JavaEE 项目开发脚手架》：http://github.com/zhangkaitao/es。

表/数据 SQL

具体请参考
sql/ shiro-schema.sql （表结构）
sql/ shiro-data.sql （初始数据）

默认用户名/密码是 admin/123456。

实体

具体请参考 com.github.zhangkaitao.shiro.chapter16.entity 包下的实体，此处就不列举了。

DAO

具体请参考 com.github.zhangkaitao.shiro.chapter16.dao 包下的 DAO 接口及实现。

Service

具体请参考 com.github.zhangkaitao.shiro.chapter16.service 包下的 Service 接口及实现。以下是出了基本 CRUD 之外的关键接口：

```
public interface ResourceService {
    Set<String> findPermissions(Set<Long> resourceIds); //得到资源对应的权限字符串
    List<Resource> findMenus(Set<String> permissions); //根据用户权限得到菜单
}
```

```
public interface RoleService {
    Set<String> findRoles(Long... roleIds); //根据角色编号得到角色标识符列表
    Set<String> findPermissions(Long[] roleIds); //根据角色编号得到权限字符串列表
}
```

jinnianshilongnian.iteye.com/

```
public interface UserService {
    public void changePassword(Long userId, String newPassword); //修改密码
    public User findByUsername(String username); //根据用户名查找用户
    public Set<String> findRoles(String username);// 根据用户名查找其角色
    public Set<String> findPermissions(String username);// 根据用户名查找其权限
}
```

Service 实现请参考源代码，此处就不列举了。

UserRealm 实现

```
public class UserRealm extends AuthorizingRealm {
    @Autowired private UserService userService;
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        String username = (String)principals.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
        authorizationInfo.setRoles(userService.findRoles(username));
        authorizationInfo.setStringPermissions(userService.findPermissions(username));
        System.out.println(userService.findPermissions(username));
        return authorizationInfo;
    }
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        String username = (String)token.getPrincipal();
        User user = userService.findByUsername(username);
        if(user == null) {
            throw new UnknownAccountException();//没找到帐号
        }
        if(Boolean.TRUE.equals(user.getLocked())) {
            throw new LockedAccountException(); //帐号锁定
        }
        return new SimpleAuthenticationInfo(
                user.getUsername(), //用户名
                user.getPassword(), //密码
                ByteSource.Util.bytes(user.getCredentialsSalt()),//salt=username+salt
                getName()  //realm name
        );
    }
}
```

此处的 UserRealm 和《第六章 Realm 及相关对象》中的 UserRealm 类似，通过 UserService 获取帐号及角色/权限信息。

Web 层控制器

```
@Controller
public class IndexController {
    @Autowired
    private ResourceService resourceService;
    @Autowired
    private UserService userService;
    @RequestMapping("/")
    public String index(@CurrentUser User loginUser, Model model) {
        Set<String> permissions = userService.findPermissions(loginUser.getUsername());
        List<Resource> menus = resourceService.findMenus(permissions);
        model.addAttribute("menus", menus);
        return "index";
    }
}
```

IndexController 中查询菜单在前台界面显示，请参考相应的 jsp 页面；

```
@Controller
public class LoginController {
    @RequestMapping(value = "/login")
    public String showLoginForm(HttpServletRequest req, Model model) {
        String exceptionClassName = (String)req.getAttribute("shiroLoginFailure");
        String error = null;
        if(UnknownAccountException.class.getName().equals(exceptionClassName)) {
            error = "用户名/密码错误";
        } else if(IncorrectCredentialsException.class.getName().equals(exceptionClassName))
{
            error = "用户名/密码错误";
        } else if(exceptionClassName != null) {
            error = "其他错误：" + exceptionClassName;
        }
        model.addAttribute("error", error);
        return "login";
    }
}
```

LoginController 用于显示登录表单页面，其中 shiro authc 拦截器进行登录，登录失败的话会把错误存到 shiroLoginFailure 属性中，在该控制器中获取后来显示相应的错误信息。

```
@RequiresPermissions("resource:view")
@RequestMapping(method = RequestMethod.GET)
public String list(Model model) {
    model.addAttribute("resourceList", resourceService.findAll());
    return "resource/list";
}
```

在控制器方法上使用@RequiresPermissions 指定需要的权限信息，其他的都是类似的，请参考源码。

Web 层标签库

com.github.zhangkaitao.shiro.chapter16.web.taglib.Functions 提供了函数标签实现，有根据编号显示资源/角色/组织机构名称，其定义放在 src/main/webapp/tld/zhang-functions.tld。

Web 层异常处理器

```
@ControllerAdvice
public class DefaultExceptionHandler {
    @ExceptionHandler({UnauthorizedException.class})
    @ResponseStatus(HttpStatus.UNAUTHORIZED)
    public ModelAndView processUnauthenticatedException(NativeWebRequest request,
UnauthorizedException e) {
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception", e);
        mv.setViewName("unauthorized");
        return mv;
    }
}
```

如果抛出 UnauthorizedException，将被该异常处理器截获来显示没有权限信息。

Spring 配置——spring-config.xml

定义了 context:component-scan 来扫描除 web 层的组件、dataSource（数据源）、事务管理器及事务切面等；具体请参考配置源码。

Spring 配置——spring-config-cache.xml

定义了 spring 通用 cache，使用 ehcache 实现；具体请参考配置源码。

Spring 配置——spring-config-shiro.xml

定义了 shiro 相关组件。

```
<bean id="userRealm" class="com.github.zhangkaitao.shiro.chapter16.realm.UserRealm">
    <property name="credentialsMatcher" ref="credentialsMatcher"/>
    <property name="cachingEnabled" value="false"/>
</bean>
```

userRealm 组件禁用掉了 cache，可以参考 https://github.com/zhangkaitao/es/tree/master/web/src/main/java/com/sishuok/es/extra/aop 实现自己的 cache 切面；否则需要在修改如资源/角色等信息时清理掉缓存。

```
<bean id="sysUserFilter"
  class="com.github.zhangkaitao.shiro.chapter16.web.shiro.filter.SysUserFilter"/>
```

sysUserFilter 用于根据当前登录用户身份获取 User 信息放入 request；然后就可以通过 request 获取 User。

```
<property name="filterChainDefinitions">
  <value>
    /login = authc
    /logout = logout
    /authenticated = authc
    /** = user,sysUser
  </value>
</property>
```

如上是 shiroFilter 的 filterChainDefinitions 定义。

Spring MVC 配置——spring-mvc.xml

定义了 spring mvc 相关组件。

```
<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <bean class="com.github.zhangkaitao.shiro.chapter16
        .web.bind.method.CurrentUserMethodArgumentResolver"/>
  </mvc:argument-resolvers>
</mvc:annotation-driven>
```

此处注册了一个@CurrentUser 参数解析器。如之前的 IndexController，从 request 获取 shiro sysUser 拦截器放入的当前登录 User 对象。

Spring MVC 配置——spring-mvc-shiro.xml

定义了 spring mvc 相关组件。

```
<aop:config proxy-target-class="true"></aop:config>
<bean class="org.apache.shiro.spring.security
      .interceptor.AuthorizationAttributeSourceAdvisor">
   <property name="securityManager" ref="securityManager"/>
</bean>
```

定义 aop 切面，用于代理如@RequiresPermissions 注解的控制器，进行权限控制。

web.xml 配置文件

定义 Spring ROOT 上下文加载器、ShiroFilter、及 SpringMVC 拦截器。具体请参考源码。

JSP 页面

```
<shiro:hasPermission name="user:create">
      <a href="${pageContext.request.contextPath}/user/create">用户新增</a><br/>
</shiro:hasPermission>
```

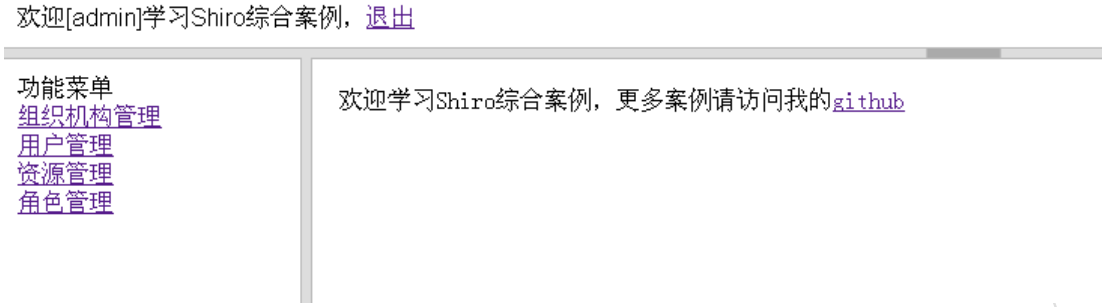
使用 shiro 标签进行权限控制。具体请参考源码。

系统截图

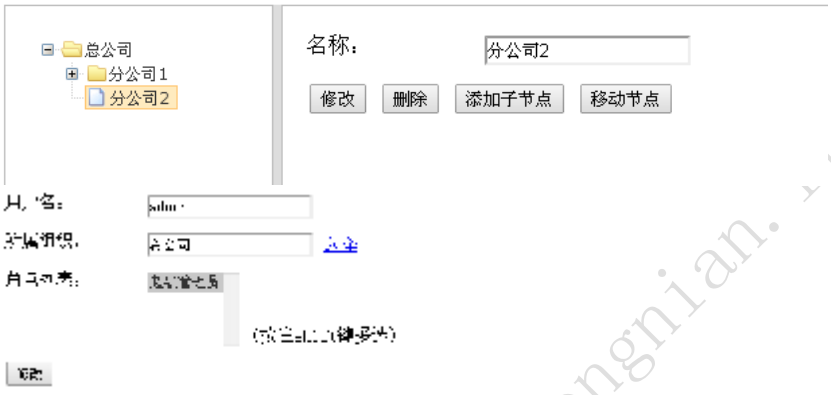
访问 http://localhost:8080/chapter16/；

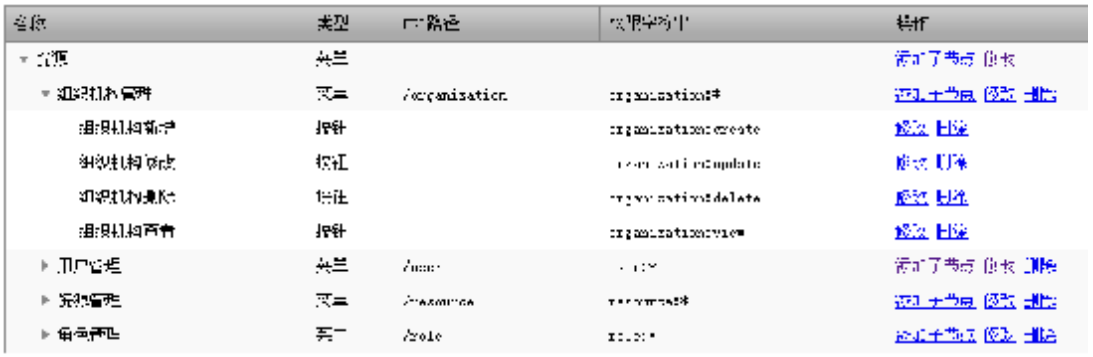
首先进入登录页面，输入用户名/密码（默认 admin/123456）登录：

登录成功后到达整个页面主页，并根据当前用户权限显示相应的菜单，此处菜单比较简单，没有树型结构显示



然后就可以进行一些操作，如组织机构维护、用户修改、资源维护、角色授权





相关资料

《跟我学 spring3》

http://www.iteye.com/blogs/subjects/spring3

《跟开涛学 SpringMVC》

http://www.iteye.com/blogs/subjects/kaitao-springmvc

《简单 shiro 扩展实现 NOT、AND、OR 权限验证》

http://jinnianshilongnian.iteye.com/blog/1864800

《Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired 失效》

http://jinnianshilongnian.iteye.com/blog/1850425

更复杂的权限请参考我的《JavaEE 项目开发脚手架》：http://github.com/zhangkaitao/es，提供了更加复杂的实现。

http://jinnianshilongnian.iteye.com/

第十七章　OAuth2 集成

目前很多开放平台如新浪微博开放平台都在使用提供开放 API 接口供开发者使用，随之带来了第三方应用要到开放平台进行授权的问题，OAuth 就是干这个的，OAuth2 是 OAuth 协议的下一个版本，相比 OAuth1，OAuth2 整个授权流程更简单安全了，但不兼容 OAuth1，具体可以到 OAuth2 官网 http://oauth.net/2/ 查看，OAuth2 协议规范可以参考 http://tools.ietf.org/html/rfc6749。目前有好多参考实现供选择，可以到其官网查看下载。

本文使用 Apache Oltu，其之前的名字叫 Apache Amber，是 Java 版的参考实现。使用文档可参考 https://cwiki.apache.org/confluence/display/OLTU/Documentation。

OAuth 角色

资源拥有者（resource owner）：能授权访问受保护资源的一个实体，可以是一个人，那我们称之为最终用户；如新浪微博用户 zhangsan；

资源服务器（resource server）：存储受保护资源，客户端通过 access token 请求资源，资源服务器响应受保护资源给客户端；存储着用户 zhangsan 的微博等信息。

授权服务器（authorization server）：成功验证资源拥有者并获取授权之后，授权服务器颁发授权令牌（Access Token）给客户端。

客户端（client）：如新浪微博客户端 weico、微格等第三方应用，也可以是它自己的官方应用；其本身不存储资源，而是资源拥有者授权通过后，使用它的授权（授权令牌）访问受保护资源，然后客户端把相应的数据展示出来/提交到服务器。“客户端”术语不代表任何特定实现（如应用运行在一台服务器、桌面、手机或其他设备）。

OAuth2 协议流程

1、客户端从资源拥有者那请求授权。授权请求可以直接发给资源拥有者，或间接的通过授权服务器这种中介，后者更可取。
2、客户端收到一个授权许可，代表资源服务器提供的授权。
3、客户端使用它自己的私有证书及授权许可到授权服务器验证。
4、如果验证成功，则下发一个访问令牌。
5、客户端使用访问令牌向资源服务器请求受保护资源。
6、资源服务器会验证访问令牌的有效性，如果成功则下发受保护资源。

更多流程的解释请参考 OAuth2 的协议规范 http://tools.ietf.org/html/rfc6749。

服务器端

本文把授权服务器和资源服务器整合在一起实现。

POM 依赖

此处我们使用 apache oltu oauth2 服务端实现，需要引入 authzserver（授权服务器依赖）和 resourceserver（资源服务器依赖）。

```
<dependency>
    <groupId>org.apache.oltu.oauth2</groupId>
    <artifactId>org.apache.oltu.oauth2.authzserver</artifactId>
    <version>0.31</version>
</dependency>
<dependency>
    <groupId>org.apache.oltu.oauth2</groupId>
    <artifactId>org.apache.oltu.oauth2.resourceserver</artifactId>
    <version>0.31</version>
</dependency>
```

其他的请参考 pom.xml。

数据字典

用户(oauth2_user)

名称	类型	长度	描述
id	bigint	10	编号 主键
username	varchar	100	用户名
password	varchar	100	密码
salt	varchar	50	盐

客户端(oauth2_client)

名称	类型	长度	描述
id	bigint	10	编号 主键
client_name	varchar	100	客户端名称
client_id	varchar	100	客户端 id
client_secret	varchar	100	客户端安全 key

用户表存储着认证/资源服务器的用户信息，即资源拥有者；比如用户名/密码；客户端表存储客户端的的客户端 id 及客户端安全 key；在进行授权时使用。

表及数据 SQL

具体请参考
sql/ shiro-schema.sql （表结构）
sql/ shiro-data.sql （初始数据）

默认用户名/密码是 admin/123456。

实体

具体请参考 com.github.zhangkaitao.shiro.chapter17.entity 包下的实体，此处就不列举了。

DAO

具体请参考 com.github.zhangkaitao.shiro.chapter17.dao 包下的 DAO 接口及实现。

Service

具体请参考 com.github.zhangkaitao.shiro.chapter17.service 包下的 Service 接口及实现。以下是出了基本 CRUD 之外的关键接口：

```
public interface UserService {
    public User createUser(User user);// 创建用户
    public User updateUser(User user);// 更新用户
    public void deleteUser(Long userId);// 删除用户
    public void changePassword(Long userId, String newPassword); //修改密码
    User findOne(Long userId);// 根据 id 查找用户
    List<User> findAll();// 得到所有用户
    public User findByUsername(String username);// 根据用户名查找用户
}
```

jinnianshilongnian.iteye.com/

```
public interface ClientService {
    public Client createClient(Client client);// 创建客户端
    public Client updateClient(Client client);// 更新客户端
    public void deleteClient(Long clientId);// 删除客户端
    Client findOne(Long clientId);// 根据 id 查找客户端
    List<Client> findAll();// 查找所有
    Client findByClientId(String clientId);// 根据客户端 id 查找客户端
    Client findByClientSecret(String clientSecret);//根据客户端安全 KEY 查找客户端
```

```
public interface OAuthService {
    public void addAuthCode(String authCode, String username);// 添加 auth code
    public void addAccessToken(String accessToken, String username); // 添加 access token
    boolean checkAuthCode(String authCode); // 验证 auth code 是否有效
    boolean checkAccessToken(String accessToken); // 验证 access token 是否有效
    String getUsernameByAuthCode(String authCode);// 根据 auth code 获取用户名
    String getUsernameByAccessToken(String accessToken);// 根据 access token 获取用户名
    long getExpireIn();//auth code / access token 过期时间
    public boolean checkClientId(String clientId);// 检查客户端 id 是否存在
     public boolean checkClientSecret(String clientSecret);// 坚持客户端安全 KEY 是否存在
}
```

此处通过 OAuthService 实现进行 auth code 和 access token 的维护。

后端数据维护控制器

具体请参考 com.github.zhangkaitao.shiro.chapter17.web.controller 包下的 IndexController、LoginController、UserController 和 ClientController，其用于维护后端的数据，如用户及客户端数据；即相当于后台管理。

授权控制器 AuthorizeController

```
@Controller
public class AuthorizeController {
   @Autowired
   private OAuthService oAuthService;
   @Autowired
   private ClientService clientService;
```

```
@RequestMapping("/authorize")
public Object authorize(Model model,    HttpServletRequest request)
        throws URISyntaxException, OAuthSystemException {
   try {
      //构建 OAuth  授权请求
      OAuthAuthzRequest oauthRequest = new OAuthAuthzRequest(request);
      //检查传入的客户端 id 是否正确
      if (!oAuthService.checkClientId(oauthRequest.getClientId())) {
         OAuthResponse response = OAuthASResponse
               .errorResponse(HttpServletResponse.SC_BAD_REQUEST)
               .setError(OAuthError.TokenResponse.INVALID_CLIENT)
               .setErrorDescription(Constants.INVALID_CLIENT_DESCRIPTION)
               .buildJSONMessage();
         return new ResponseEntity(
             response.getBody(), HttpStatus.valueOf(response.getResponseStatus()));
      }

      Subject subject = SecurityUtils.getSubject();
      //如果用户没有登录，跳转到登陆页面
      if(!subject.isAuthenticated()) {
         if(!login(subject, request)) {//登录失败时跳转到登陆页面
            model.addAttribute("client",
                  clientService.findByClientId(oauthRequest.getClientId()));
            return "oauth2login";
         }
      }

      String username = (String)subject.getPrincipal();
      //生成授权码
      String authorizationCode = null;
      //responseType 目前仅支持 CODE，另外还有 TOKEN
      String responseType = oauthRequest.getParam(OAuth.OAUTH_RESPONSE_TYPE);
      if (responseType.equals(ResponseType.CODE.toString())) {
         OAuthIssuerImpl oauthIssuerImpl = new OAuthIssuerImpl(new MD5Generator());
         authorizationCode = oauthIssuerImpl.authorizationCode();
         oAuthService.addAuthCode(authorizationCode, username);
      }
```

```
            //进行 OAuth 响应构建
            OAuthASResponse.OAuthAuthorizationResponseBuilder builder =
                OAuthASResponse.authorizationResponse(request,
                                                      HttpServletResponse.SC_FOUND);
            //设置授权码
            builder.setCode(authorizationCode);
            //得到到客户端重定向地址
            String redirectURI = oauthRequest.getParam(OAuth.OAUTH_REDIRECT_URI);

            //构建响应
            final OAuthResponse response = builder.location(redirectURI).buildQueryMessage();
            //根据 OAuthResponse 返回 ResponseEntity 响应
            HttpHeaders headers = new HttpHeaders();
            headers.setLocation(new URI(response.getLocationUri()));
            return new ResponseEntity(headers, HttpStatus.valueOf(response.getResponseStatus()));
        } catch (OAuthProblemException e) {
            //出错处理
            String redirectUri = e.getRedirectUri();
            if (OAuthUtils.isEmpty(redirectUri)) {
                //告诉客户端没有传入 redirectUri 直接报错
                return new ResponseEntity(
                    "OAuth callback url needs to be provided by client!!!", HttpStatus.NOT_FOUND);
            }
            //返回错误消息（如?error=）
            final OAuthResponse response =
                        OAuthASResponse.errorResponse(HttpServletResponse.SC_FOUND)
                                .error(e).location(redirectUri).buildQueryMessage();
            HttpHeaders headers = new HttpHeaders();
            headers.setLocation(new URI(response.getLocationUri()));
            return new ResponseEntity(headers, HttpStatus.valueOf(response.getResponseStatus()));
        }
    }

    private boolean login(Subject subject, HttpServletRequest request) {
        if("get".equalsIgnoreCase(request.getMethod())) {
            return false;
        }
```

```
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if(StringUtils.isEmpty(username) || StringUtils.isEmpty(password)) {
            return false;
        }

        UsernamePasswordToken token = new UsernamePasswordToken(username, password);
        try {
            subject.login(token);
            return true;
        } catch (Exception e) {
            request.setAttribute("error", "登录失败:" + e.getClass().getName());
            return false;
        }
    }
}
```

如上代码的作用：

1、首先通过如 http://localhost:8080/chapter17-server/authorize?client_id=c1ebe466-1cdc-4bd3-ab69-77c3561b9dee&response_type=code&redirect_uri=http://localhost:9080/chapter17-client/oauth2-login 访问授权页面；

2、该控制器首先检查 clientId 是否正确；如果错误将返回相应的错误信息；

3、然后判断用户是否登录了，如果没有登录首先到登录页面登录；

4、登录成功后生成相应的 auth code 即授权码，然后重定向到客户端地址，如 http://localhost:9080/chapter17-client/oauth2-login?code=52b1832f5dff68122f4f00ae995da0ed；在重定向到的地址中会带上 code 参数（授权码），接着客户端可以根据授权码去换取 access token。

访问令牌控制器 AccessTokenController

```
@RestController
public class AccessTokenController {
    @Autowired
    private OAuthService oAuthService;
    @Autowired
    private UserService userService;
```

```
@RequestMapping("/accessToken")
public HttpEntity token(HttpServletRequest request)
        throws URISyntaxException, OAuthSystemException {
    try {
        //构建 OAuth 请求
        OAuthTokenRequest oauthRequest = new OAuthTokenRequest(request);

        //检查提交的客户端 id 是否正确
        if (!oAuthService.checkClientId(oauthRequest.getClientId())) {
            OAuthResponse response = OAuthASResponse
                    .errorResponse(HttpServletResponse.SC_BAD_REQUEST)
                    .setError(OAuthError.TokenResponse.INVALID_CLIENT)
                    .setErrorDescription(Constants.INVALID_CLIENT_DESCRIPTION)
                    .buildJSONMessage();
            return new ResponseEntity(
                response.getBody(), HttpStatus.valueOf(response.getResponseStatus()));
        }

        // 检查客户端安全 KEY 是否正确
        if (!oAuthService.checkClientSecret(oauthRequest.getClientSecret())) {
            OAuthResponse response = OAuthASResponse
                    .errorResponse(HttpServletResponse.SC_UNAUTHORIZED)
                    .setError(OAuthError.TokenResponse.UNAUTHORIZED_CLIENT)
                    .setErrorDescription(Constants.INVALID_CLIENT_DESCRIPTION)
                    .buildJSONMessage();
            return new ResponseEntity(
                response.getBody(), HttpStatus.valueOf(response.getResponseStatus()));
        }

        String authCode = oauthRequest.getParam(OAuth.OAUTH_CODE);
        // 检查验证类型，此处只检查 AUTHORIZATION_CODE 类型，其他的还有
PASSWORD 或 REFRESH_TOKEN
        if (oauthRequest.getParam(OAuth.OAUTH_GRANT_TYPE).equals(
                GrantType.AUTHORIZATION_CODE.toString())) {
            if (!oAuthService.checkAuthCode(authCode)) {
                OAuthResponse response = OAuthASResponse
                        .errorResponse(HttpServletResponse.SC_BAD_REQUEST)
```

```
                    .setError(OAuthError.TokenResponse.INVALID_GRANT)
                    .setErrorDescription("错误的授权码")
                    .buildJSONMessage();
            return new ResponseEntity(
                response.getBody(), HttpStatus.valueOf(response.getResponseStatus()));
          }
        }

        //生成 Access Token
        OAuthIssuer oauthIssuerImpl = new OAuthIssuerImpl(new MD5Generator());
        final String accessToken = oauthIssuerImpl.accessToken();
        oAuthService.addAccessToken(accessToken,
          oAuthService.getUsernameByAuthCode(authCode));

        //生成 OAuth 响应
        OAuthResponse response = OAuthASResponse
                .tokenResponse(HttpServletResponse.SC_OK)
                .setAccessToken(accessToken)
                .setExpiresIn(String.valueOf(oAuthService.getExpireIn()))
                .buildJSONMessage();

        //根据 OAuthResponse 生成 ResponseEntity
        return new ResponseEntity(
            response.getBody(), HttpStatus.valueOf(response.getResponseStatus()));
      } catch (OAuthProblemException e) {
        //构建错误响应
        OAuthResponse res = OAuthASResponse
                .errorResponse(HttpServletResponse.SC_BAD_REQUEST).error(e)
                .buildJSONMessage();
        return new ResponseEntity(res.getBody(), HttpStatus.valueOf(res.getResponseStatus()));
      }
    }
}
```

如上代码的作用：

1、首先通过如 http://localhost:8080/chapter17-server/accessToken，POST 提交如下数据：client_id= c1ebe466-1cdc-4bd3-ab69-77c3561b9dee& client_secret= d8346ea2-6017-43ed-ad68-19c0f971738b&grant_type=authorization_code&code=828beda907066d058584f37bcfd597b6&redirect_uri=http://localhost:9080/chapter17-client/oauth2-login 访

问；

2、该控制器会验证 client_id、client_secret、auth code 的正确性，如果错误会返回相应的错误；

3、如果验证通过会生成并返回相应的访问令牌 access token。

资源控制器 UserInfoController

```
@RestController
public class UserInfoController {
    @Autowired
    private OAuthService oAuthService;

    @RequestMapping("/userInfo")
    public HttpEntity userInfo(HttpServletRequest request) throws OAuthSystemException {
        try {
            //构建 OAuth 资源请求
            OAuthAccessResourceRequest oauthRequest =
                    new OAuthAccessResourceRequest(request, ParameterStyle.QUERY);
            //获取 Access Token
            String accessToken = oauthRequest.getAccessToken();

            //验证 Access Token
            if (!oAuthService.checkAccessToken(accessToken)) {
                // 如果不存在/过期了，返回未验证错误，需重新验证
                OAuthResponse oauthResponse = OAuthRSResponse
                        .errorResponse(HttpServletResponse.SC_UNAUTHORIZED)
                        .setRealm(Constants.RESOURCE_SERVER_NAME)
                        .setError(OAuthError.ResourceResponse.INVALID_TOKEN)
                        .buildHeaderMessage();

                HttpHeaders headers = new HttpHeaders();
                headers.add(OAuth.HeaderType.WWW_AUTHENTICATE,
                    oauthResponse.getHeader(OAuth.HeaderType.WWW_AUTHENTICATE));
                return new ResponseEntity(headers, HttpStatus.UNAUTHORIZED);
            }
            //返回用户名
            String username = oAuthService.getUsernameByAccessToken(accessToken);
            return new ResponseEntity(username, HttpStatus.OK);
        } catch (OAuthProblemException e) {
            //检查是否设置了错误码
```

```
        String errorCode = e.getError();
        if (OAuthUtils.isEmpty(errorCode)) {
            OAuthResponse oauthResponse = OAuthRSResponse
                    .errorResponse(HttpServletResponse.SC_UNAUTHORIZED)
                    .setRealm(Constants.RESOURCE_SERVER_NAME)
                    .buildHeaderMessage();

            HttpHeaders headers = new HttpHeaders();
            headers.add(OAuth.HeaderType.WWW_AUTHENTICATE,
                oauthResponse.getHeader(OAuth.HeaderType.WWW_AUTHENTICATE));
            return new ResponseEntity(headers, HttpStatus.UNAUTHORIZED);
        }

        OAuthResponse oauthResponse = OAuthRSResponse
                    .errorResponse(HttpServletResponse.SC_UNAUTHORIZED)
                    .setRealm(Constants.RESOURCE_SERVER_NAME)
                    .setError(e.getError())
                    .setErrorDescription(e.getDescription())
                    .setErrorUri(e.getUri())
                    .buildHeaderMessage();

        HttpHeaders headers = new HttpHeaders();
        headers.add(OAuth.HeaderType.WWW_AUTHENTICATE, 、
            oauthResponse.getHeader(OAuth.HeaderType.WWW_AUTHENTICATE));
        return new ResponseEntity(HttpStatus.BAD_REQUEST);
      }
    }
}
```

如上代码的作用：

1、首先通过如 http://localhost:8080/chapter17-server/userInfo?access_token=828beda907066d058584f37bcfd597b6 进行访问；

2、该控制器会验证 access token 的有效性；如果无效了将返回相应的错误，客户端再重新进行授权；

3、如果有效，则返回当前登录用户的用户名。

Spring 配置文件

具体请参考 resources/spring*.xml，此处只列举 spring-config-shiro.xml 中的 shiroFilter 的 filterChainDefinitions 属性：

```
<property name="filterChainDefinitions">
    <value>
        / = anon
        /login = authc
        /logout = logout

        /authorize=anon
        /accessToken=anon
        /userInfo=anon

        /** = user
    </value>
</property>
```

对于 oauth2 的几个地址/authorize、/accessToken、/userInfo 都是匿名可访问的。

其他源码请直接下载文档查看。

服务器维护

访问 localhost:8080/chapter17-server/，登录后进行客户端管理和用户管理。
客户端管理就是进行客户端的注册，如新浪微博的第三方应用就需要到新浪微博开发平台进行注册；用户管理就是进行如新浪微博用户的管理。

对于授权服务和资源服务的实现可以参考新浪微博开发平台的实现：
http://open.weibo.com/wiki/授权机制说明
http://open.weibo.com/wiki/微博 API

客户端

客户端流程：如果需要登录首先跳到 oauth2 服务端进行登录授权，成功后服务端返回 auth code，然后客户端使用 auth code 去服务器端换取 access token，最好根据 access token 获取用户信息进行客户端的登录绑定。这个可以参照如很多网站的新浪微博登录功能，或其他的第三方帐号登录功能。

POM 依赖

此处我们使用 apache oltu oauth2 客户端实现。

```
<dependency>
   <groupId>org.apache.oltu.oauth2</groupId>
   <artifactId>org.apache.oltu.oauth2.client</artifactId>
   <version>0.31</version>
</dependency>
```

其他的请参考 pom.xml。

OAuth2Token

类似于 UsernamePasswordToken 和 CasToken；用于存储 oauth2 服务端返回的 auth code。

```
public class OAuth2Token implements AuthenticationToken {
    private String authCode;
    private String principal;
    public OAuth2Token(String authCode) {
        this.authCode = authCode;
    }
    //省略 getter/setter
}
```

OAuth2AuthenticationFilter

该 filter 的作用类似于 FormAuthenticationFilter 用于 oauth2 客户端的身份验证控制；如果当前用户还没有身份验证，首先会判断 url 中是否有 code（服务端返回的 auth code），如果没有则重定向到服务端进行登录并授权，然后返回 auth code；接着 OAuth2AuthenticationFilter 会用 auth code 创建 OAuth2Token，然后提交给 Subject.login 进行登录；接着 OAuth2Realm 会根据 OAuth2Token 进行相应的登录逻辑。

```
public class OAuth2AuthenticationFilter extends AuthenticatingFilter {
    //oauth2 authc code 参数名
    private String authcCodeParam = "code";
    //客户端 id
    private String clientId;
    //服务器端登录成功/失败后重定向到的客户端地址
    private String redirectUrl;
    //oauth2 服务器响应类型
    private String responseType = "code";
    private String failureUrl;
    //省略 setter
```

```
    protected AuthenticationToken createToken(ServletRequest request, ServletResponse response) throws Exception {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String code = httpRequest.getParameter(authcCodeParam);
        return new OAuth2Token(code);
    }
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {
        return false;
    }
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
        String error = request.getParameter("error");
        String errorDescription = request.getParameter("error_description");
        if(!StringUtils.isEmpty(error)) {//如果服务端返回了错误
            WebUtils.issueRedirect(request, response, failureUrl + "?error=" + error + "error_description=" + errorDescription);
            return false;
        }
        Subject subject = getSubject(request, response);
        if(!subject.isAuthenticated()) {
            if(StringUtils.isEmpty(request.getParameter(authcCodeParam))) {
                //如果用户没有身份验证，且没有 auth code，则重定向到服务端授权
                saveRequestAndRedirectToLogin(request, response);
                return false;
            }
        }
        //执行父类里的登录逻辑，调用 Subject.login 登录
        return executeLogin(request, response);
    }

    //登录成功后的回调方法 重定向到成功页面
    protected boolean onLoginSuccess(AuthenticationToken token, Subject subject, ServletRequest request,  ServletResponse response) throws Exception {
        issueSuccessRedirect(request, response);
        return false;
    }
```

```
    //登录失败后的回调
    protected boolean onLoginFailure(AuthenticationToken token, AuthenticationException
ae, ServletRequest request,
                                     ServletResponse response) {
        Subject subject = getSubject(request, response);
        if (subject.isAuthenticated() || subject.isRemembered()) {
            try { //如果身份验证成功了 则也重定向到成功页面
                issueSuccessRedirect(request, response);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            try { //登录失败时重定向到失败页面
                WebUtils.issueRedirect(request, response, failureUrl);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return false;
    }
}
```

该拦截器的作用：

1、首先判断有没有服务端返回的 error 参数，如果有则直接重定向到失败页面；

2、接着如果用户还没有身份验证，判断是否有 auth code 参数（即是不是服务端授权之后返回的），如果没有则重定向到服务端进行授权；

3、否则调用 executeLogin 进行登录，通过 auth code 创建 OAuth2Token 提交给 Subject 进行登录；

4、登录成功将回调 onLoginSuccess 方法重定向到成功页面；

5、登录失败则回调 onLoginFailure 重定向到失败页面。

OAuth2Realm

```
public class OAuth2Realm extends AuthorizingRealm {
    private String clientId;
    private String clientSecret;
    private String accessTokenUrl;
    private String userInfoUrl;
```

```
    private String redirectUrl;
    //省略 setter
    public boolean supports(AuthenticationToken token) {
        return token instanceof OAuth2Token; //表示此 Realm 只支持 OAuth2Token 类型
    }
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
        return authorizationInfo;
    }
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        OAuth2Token oAuth2Token = (OAuth2Token) token;
        String code = oAuth2Token.getAuthCode(); //获取  auth code
        String username = extractUsername(code); //  提取用户名
        SimpleAuthenticationInfo authenticationInfo =
                new SimpleAuthenticationInfo(username, code, getName());
        return authenticationInfo;
    }
    private String extractUsername(String code) {
        try {
            OAuthClient oAuthClient = new OAuthClient(new URLConnectionClient());
            OAuthClientRequest accessTokenRequest = OAuthClientRequest
                    .tokenLocation(accessTokenUrl)
                    .setGrantType(GrantType.AUTHORIZATION_CODE)
                    .setClientId(clientId).setClientSecret(clientSecret)
                    .setCode(code).setRedirectURI(redirectUrl)
                    .buildQueryMessage();
            //获取 access token
            OAuthAccessTokenResponse oAuthResponse =
                oAuthClient.accessToken(accessTokenRequest, OAuth.HttpMethod.POST);
            String accessToken = oAuthResponse.getAccessToken();
            Long expiresIn = oAuthResponse.getExpiresIn();
            //获取 user info
            OAuthClientRequest userInfoRequest =
                new OAuthBearerClientRequest(userInfoUrl)
                    .setAccessToken(accessToken).buildQueryMessage();
            OAuthResourceResponse resourceResponse = oAuthClient.resource(
                userInfoRequest, OAuth.HttpMethod.GET, OAuthResourceResponse.class);
```

```
                String username = resourceResponse.getBody();
                return username;
            } catch (Exception e) {
                throw new OAuth2AuthenticationException(e);
            }
        }
}
```

此 Realm 首先只支持 OAuth2Token 类型的 Token；然后通过传入的 auth code 去换取 access token；再根据 access token 去获取用户信息（用户名），然后根据此信息创建 AuthenticationInfo；如果需要 AuthorizationInfo 信息，可以根据此处获取的用户名再根据自己的业务规则去获取。

Spring shiro 配置（spring-config-shiro.xml）

```
<bean id="oAuth2Realm"
      class="com.github.zhangkaitao.shiro.chapter18.oauth2.OAuth2Realm">
   <property name="cachingEnabled" value="true"/>
   <property name="authenticationCachingEnabled" value="true"/>
   <property name="authenticationCacheName" value="authenticationCache"/>
   <property name="authorizationCachingEnabled" value="true"/>
   <property name="authorizationCacheName" value="authorizationCache"/>
   <property name="clientId" value="c1ebe466-1cdc-4bd3-ab69-77c3561b9dee"/>
   <property name="clientSecret" value="d8346ea2-6017-43ed-ad68-19c0f971738b"/>
   <property name="accessTokenUrl"
        value="http://localhost:8080/chapter17-server/accessToken"/>
   <property name="userInfoUrl" value="http://localhost:8080/chapter17-server/userInfo"/>
   <property name="redirectUrl" value="http://localhost:9080/chapter17-client/oauth2-login"/>
</bean>
```

此 OAuth2Realm 需要配置在服务端申请的 clientId 和 clientSecret；及用于根据 auth code 换取 access token 的 accessTokenUrl 地址；及用于根据 access token 换取用户信息（受保护资源）的 userInfoUrl 地址。

```
<bean id="oAuth2AuthenticationFilter"
      class="com.github.zhangkaitao.shiro.chapter18.oauth2.OAuth2AuthenticationFilter">
   <property name="authcCodeParam" value="code"/>
   <property name="failureUrl" value="/oauth2Failure.jsp"/>
</bean>
```

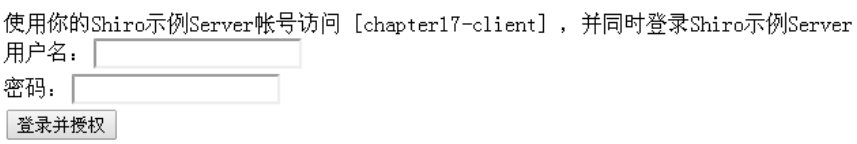
此 OAuth2AuthenticationFilter 用于拦截服务端重定向回来的 auth code。

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property                                                    name="loginUrl"
value="http://localhost:8080/chapter17-server/authorize?client_id=c1ebe466-1cdc-4bd3-ab69-7
7c3561b9dee&amp;response_type=code&amp;redirect_uri=http://localhost:9080/chapter17-clie
nt/oauth2-login"/>
    <property name="successUrl" value="/"/>
    <property name="filters">
        <util:map>
            <entry key="oauth2Authc" value-ref="oAuth2AuthenticationFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            / = anon
            /oauth2Failure.jsp = anon
            /oauth2-login = oauth2Authc
            /logout = logout
            /** = user
        </value>
    </property>
</bean>
```

此处设置 loginUrl 为 http://localhost:8080/chapter17-server/authorize ?client_id=c1ebe466-1cdc-4bd3-ab69-77c3561b9dee&response_type=code&redirect_uri=http://localhost:9080/chapter17-client/oauth2-login"；其会自动设置到所有的 AccessControlFilter，如 oAuth2AuthenticationFilter；另外/oauth2-login = oauth2Authc 表示 /oauth2-login 地址使用 oauth2Authc 拦截器拦截并进行 oauth2 客户端授权。

测试

1、首先访问 http://localhost:9080/chapter17-client/，然后点击登录按钮进行登录，会跳到如下页面：



2、输入用户名进行登录并授权；
3、如果登录成功，服务端会重定向到客户端，即之前客户端提供的地址 http://localhost:9080/chapter17-client/oauth2-login?code=473d56015bcf576f2ca03eac1a5bcc11，并带着 auth code 过去；
4、客户端的 OAuth2AuthenticationFilter 会收集此 auth code，并创建 OAuth2Token 提交给 Subject 进行客户端登录；
5、客户端的 Subject 会委托给 OAuth2Realm 进行身份验证；此时 OAuth2Realm 会根据 auth code 换取 access token，再根据 access token 获取受保护的用户信息；然后进行客户端登录。

到此 OAuth2 的集成就完成了，此处的服务端和客户端相对比较简单，没有进行一些异常检测，请参考如新浪微博进行相应 API 及异常错误码的设计。

http://jinnianshilongnian.iteye.com/

第十八章 并发登录人数控制

在某些项目中可能会遇到如每个账户同时只能有一个人登录或几个人同时登录，如果同时有多人登录：要么不让后者登录；要么踢出前者登录（强制退出）。比如 spring security 就直接提供了相应的功能；Shiro 的话没有提供默认实现，不过可以很容易的在 Shiro 中加入这个功能。

示例代码基于《第十六章 综合实例》完成，通过 Shiro Filter 机制扩展 KickoutSessionControlFilter 完成。

首先来看看如何配置使用（spring-config-shiro.xml）

kickoutSessionControlFilter 用于控制并发登录人数的

```
<bean id="kickoutSessionControlFilter"
  class="com.github.zhangkaitao.shiro.chapter18.web.shiro.filter.KickoutSessionControlFilter">
      <property name="cacheManager" ref="cacheManager"/>
      <property name="sessionManager" ref="sessionManager"/>

      <property name="kickoutAfter" value="false"/>
      <property name="maxSession" value="2"/>
      <property name="kickoutUrl" value="/login?kickout=1"/>
</bean>
```

cacheManager：使用 cacheManager 获取相应的 cache 来缓存用户登录的会话；用于保存用户—会话之间的关系的；
sessionManager：用于根据会话 ID，获取会话进行踢出操作的；
kickoutAfter：是否踢出后来登录的，默认是 false；即后者登录的用户踢出前者登录的用户；
maxSession：同一个用户最大的会话数，默认 1；比如 2 的意思是同一个用户允许最多同时两个人登录；
kickoutUrl：被踢出后重定向到的地址；

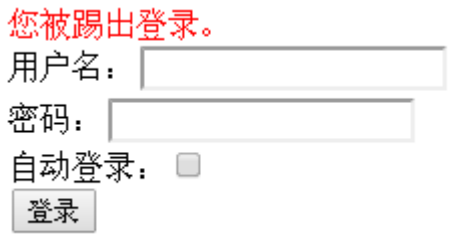
shiroFilter 配置

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
        <property name="securityManager" ref="securityManager"/>
        <property name="loginUrl" value="/login"/>
        <property name="filters">
            <util:map>
                <entry key="authc" value-ref="formAuthenticationFilter"/>
                <entry key="sysUser" value-ref="sysUserFilter"/>
                <entry key="kickout" value-ref="kickoutSessionControlFilter"/>
            </util:map>
        </property>
        <property name="filterChainDefinitions">
            <value>
                /login = authc
                /logout = logout
                /authenticated = authc
                /** = kickout,user,sysUser
            </value>
        </property>
    </bean>
```

此处配置除了登录等之外的地址都走 kickout 拦截器进行并发登录控制。

测试

此处因为 maxSession=2，所以需要打开 3 个浏览器（需要不同的浏览器，如 IE、Chrome、Firefox），分别访问 http://localhost:8080/chapter18/进行登录；然后刷新第一次打开的浏览器，将会被强制退出，如显示下图：



KickoutSessionControlFilter 核心代码：

```
protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
    Subject subject = getSubject(request, response);
    if(!subject.isAuthenticated() && !subject.isRemembered()) {
        //如果没有登录，直接进行之后的流程
        return true;
    }

    Session session = subject.getSession();
    String username = (String) subject.getPrincipal();
    Serializable sessionId = session.getId();

    //TODO 同步控制
    Deque<Serializable> deque = cache.get(username);
    if(deque == null) {
        deque = new LinkedList<Serializable>();
        cache.put(username, deque);
    }

    //如果队列里没有此 sessionId，且用户没有被踢出；放入队列
    if(!deque.contains(sessionId) && session.getAttribute("kickout") == null) {
        deque.push(sessionId);
    }

    //如果队列里的 sessionId 数超出最大会话数，开始踢人
    while(deque.size() > maxSession) {
        Serializable kickoutSessionId = null;
        if(kickoutAfter) { //如果踢出后者
            kickoutSessionId = deque.removeFirst();
        } else { //否则踢出前者
            kickoutSessionId = deque.removeLast();
        }
        try {
            Session kickoutSession =
                sessionManager.getSession(new DefaultSessionKey(kickoutSessionId));
            if(kickoutSession != null) {
                //设置会话的 kickout 属性表示踢出了
                kickoutSession.setAttribute("kickout", true);
```

```
                }
            } catch (Exception e) {//ignore exception
            }
        }

        //如果被踢出了，直接退出，重定向到踢出后的地址
        if (session.getAttribute("kickout") != null) {
            //会话被踢出了
            try {
                subject.logout();
            } catch (Exception e) { //ignore
            }
            saveRequest(request);
            WebUtils.issueRedirect(request, response, kickoutUrl);
            return false;
        }
        return true;
    }
```

此处使用了 Cache 缓存用户名—会话 id 之间的关系；如果量比较大可以考虑如持久化到数据库/其他带持久化的 Cache 中；另外此处没有并发控制的同步实现，可以考虑根据用户名获取锁来控制，减少锁的粒度。

另外可参考 JavaEE 项目开发脚手架，其提供了后台踢出用户的功能：
https://github.com/zhangkaitao/es/blob/master/web/src/main/java/com/sishuok/es/sys/user/web/controller/UserOnlineController.java

http://jinnianshilongnian

第十九章 动态 URL 权限控制

用过 Spring Security 的朋友应该比较熟悉对 URL 进行全局的权限控制，即访问 URL 时进行权限匹配；如果没有权限直接跳到相应的错误页面。Shiro 也支持类似的机制，不过需要稍微改造下来满足实际需求。不过在 Shiro 中，更多的是通过 AOP 进行分散的权限控制，即方法级别的；而通过 URL 进行权限控制是一种集中的权限控制。本章将介绍如何在 Shiro 中完成动态 URL 权限控制。

本章代码基于《第十六章 综合实例》，请先了解相关数据模型及基本流程后再学习本章。

表及数据 SQL

请运行 shiro-example-chapter19/sql/ shiro-schema.sql 表结构
请运行 shiro-example-chapter19/sql/ shiro-schema.sql 数据

实体

具体请参考 com.github.zhangkaitao.shiro.chapter19 包下的实体。

```
public class UrlFilter implements Serializable {
    private Long id;
    private String name; //url 名称/描述
    private String url; //地址
    private String roles; //所需要的角色，可省略
    private String permissions; //所需要的权限，可省略
}
```

表示拦截的 URL 和角色/权限之间的关系，多个角色/权限之间通过逗号分隔，此处还可以扩展其他的关系，另外可以加如 available 属性表示是否开启该拦截。

DAO

具体请参考 com.github.zhangkaitao.shiro.chapter19.dao 包下的 DAO 接口及实现。

Service

具体请参考 com.github.zhangkaitao.shiro.chapter19.service 包下的 Service 接口及实现。

```
public interface UrlFilterService {
    public UrlFilter createUrlFilter(UrlFilter urlFilter);
    public UrlFilter updateUrlFilter(UrlFilter urlFilter);
    public void deleteUrlFilter(Long urlFilterId);
    public UrlFilter findOne(Long urlFilterId);
    public List<UrlFilter> findAll();
}
```

基本的 URL 拦截的增删改查实现。

```
@Service
public class UrlFilterServiceImpl implements UrlFilterService {
    @Autowired
    private ShiroFilerChainManager shiroFilerChainManager;

    @Override
    public UrlFilter createUrlFilter(UrlFilter urlFilter) {
        urlFilterDao.createUrlFilter(urlFilter);
        initFilterChain();
        return urlFilter;
    }
    //其他方法请参考源码
    @PostConstruct
    public void initFilterChain() {
        shiroFilerChainManager.initFilterChains(findAll());
    }
}
```

UrlFilterServiceImpl 在进行新增、修改、删除时会调用 initFilterChain 来重新初始化 Shiro 的 URL 拦截器链，即同步数据库中的 URL 拦截器定义到 Shiro 中。此处也要注意如果直接修改数据库是不会起作用的，因为只要调用这几个 Service 方法时才同步。另外当容器启动时会自动回调 initFilterChain 来完成容器启动后的 URL 拦截器的注册。

ShiroFilerChainManager

```
@Service
public class ShiroFilerChainManager {
    @Autowired private DefaultFilterChainManager filterChainManager;
    private Map<String, NamedFilterList> defaultFilterChains;
```

```
    @PostConstruct
    public void init() {
        defaultFilterChains =
           new HashMap<String, NamedFilterList>(filterChainManager.getFilterChains());
    }
    public void initFilterChains(List<UrlFilter> urlFilters) {
        //1、首先删除以前老的 filter chain 并注册默认的
        filterChainManager.getFilterChains().clear();
        if(defaultFilterChains != null) {
            filterChainManager.getFilterChains().putAll(defaultFilterChains);
        }
        //2、循环 URL Filter 注册 filter chain
        for (UrlFilter urlFilter : urlFilters) {
            String url = urlFilter.getUrl();
            //注册 roles filter
            if (!StringUtils.isEmpty(urlFilter.getRoles())) {
                filterChainManager.addToChain(url, "roles", urlFilter.getRoles());
            }
            //注册 perms filter
            if (!StringUtils.isEmpty(urlFilter.getPermissions())) {
                filterChainManager.addToChain(url, "perms", urlFilter.getPermissions());
            }
        }
    }
}
```

1、init：Spring 容器启动时会调用 init 方法把在 spring 配置文件中配置的默认拦截器保存下来，之后会自动与数据库中的配置进行合并。

2、initFilterChains：UrlFilterServiceImpl 会在 Spring 容器启动或进行增删改 UrlFilter 时进行注册 URL 拦截器到 Shiro。

拦截器及拦截器链知识请参考《第八章 拦截器机制》，此处再介绍下 Shiro 拦截器的流程：

AbstractShiroFilter //如 ShiroFilter/ SpringShiroFilter 都继承该 Filter

doFilter //Filter 的 doFilter

doFilterInternal //转调 doFilterInternal

executeChain(request, response, chain) //执行拦截器链

FilterChain chain = getExecutionChain(request, response, origChain) //使用原始拦截器链获取新的拦截器链

chain.doFilter(request, response) //执行新组装的拦截器链

getExecutionChain(request, response, origChain) //获取拦截器链流程
FilterChainResolver resolver = getFilterChainResolver(); //获取相应的 FilterChainResolver
FilterChain resolved = resolver.getChain(request, response, origChain); //通过 FilterChainResolver 根据当前请求解析到新的 FilterChain 拦截器链

默认情况下如使用 ShiroFilterFactoryBean 创建 shiroFilter 时，默认使用 PathMatchingFilterChainResolver 进行解析，而它默认是根据当前请求的 URL 获取相应的拦截器链，使用 Ant 模式进行 URL 匹配；默认使用 DefaultFilterChainManager 进行拦截器链的管理。

PathMatchingFilterChainResolver 默认流程：

```
public FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain) {
    //1、首先获取拦截器链管理器
    FilterChainManager filterChainManager = getFilterChainManager();
    if (!filterChainManager.hasChains()) {
        return null;
    }
    //2、接着获取当前请求的 URL（不带上下文）
    String requestURI = getPathWithinApplication(request);
    //3、循环拦截器管理器中的拦截器定义（拦截器链的名字就是 URL 模式）
    for (String pathPattern : filterChainManager.getChainNames()) {
        //4、如当前 URL 匹配拦截器名字（URL 模式）
        if (pathMatches(pathPattern, requestURI)) {
            //5、返回该 URL 模式定义的拦截器链
            return filterChainManager.proxy(originalChain, pathPattern);
        }
    }
    return null;
}
```

默认实现有点小问题：
如果多个拦截器链都匹配了当前请求 URL，那么只返回第一个找到的拦截器链；后续我们可以修改此处的代码，将多个匹配的拦截器链合并返回。

DefaultFilterChainManager 内部使用 Map 来管理 URL 模式-拦截器链的关系；也就是说相同的 URL 模式只能定义一个拦截器链，不能重复定义；而且如果多个拦截器链都匹配时是无序的（因为使用 map.keySet()获取拦截器链的名字，即 URL 模式）。

FilterChainManager 接口：

```
public interface FilterChainManager {
    Map<String, Filter> getFilters(); //得到注册的拦截器
    void addFilter(String name, Filter filter); //注册拦截器
    void addFilter(String name, Filter filter, boolean init); //注册拦截器
    void createChain(String chainName, String chainDefinition); //根据拦截器链定义创建拦截器链
    void addToChain(String chainName, String filterName); //添加拦截器到指定的拦截器链
    void addToChain(String chainName, String filterName, String chainSpecificFilterConfig) throws ConfigurationException; //添加拦截器（带有配置的）到指定的拦截器链
    NamedFilterList getChain(String chainName); //获取拦截器链
    boolean hasChains(); //是否有拦截器链
    Set<String> getChainNames(); //得到所有拦截器链的名字
    FilterChain proxy(FilterChain original, String chainName); //使用指定的拦截器链代理原始拦截器链
}
```

此接口主要三个功能：注册拦截器，注册拦截器链，对原始拦截器链生成代理之后的拦截器链，比如

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
……
    <property name="filters">
        <util:map>
            <entry key="authc" value-ref="formAuthenticationFilter"/>
            <entry key="sysUser" value-ref="sysUserFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            /login = authc
            /logout = logout
            /authenticated = authc
            /** = user,sysUser
        </value>
    </property>
</bean>
```

filters 属性定义了拦截器；filterChainDefinitions 定义了拦截器链；如/**就是拦截器链的名字；而 user,sysUser 就是拦截器名字列表。

之前说过默认的 PathMatchingFilterChainResolver 和 DefaultFilterChainManager 不能满足我们的需求，我们稍微扩展了一下：

CustomPathMatchingFilterChainResolver

```
public class CustomPathMatchingFilterChainResolver
                    extends PathMatchingFilterChainResolver {
   private CustomDefaultFilterChainManager customDefaultFilterChainManager;
   public void setCustomDefaultFilterChainManager(
            CustomDefaultFilterChainManager customDefaultFilterChainManager) {
         this.customDefaultFilterChainManager = customDefaultFilterChainManager;
         setFilterChainManager(customDefaultFilterChainManager);
   }

   public FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain
originalChain) {
         FilterChainManager filterChainManager = getFilterChainManager();
         if (!filterChainManager.hasChains()) {
               return null;
         }
         String requestURI = getPathWithinApplication(request);
         List<String> chainNames = new ArrayList<String>();
         for (String pathPattern : filterChainManager.getChainNames()) {
            if (pathMatches(pathPattern, requestURI)) {
               chainNames.add(pathPattern);
            }
         }
         if(chainNames.size() == 0) {
            return null;
         }
         return customDefaultFilterChainManager.proxy(originalChain, chainNames);
   }
}
```

和默认的 PathMatchingFilterChainResolver 区别是，此处得到所有匹配的拦截器链，然后通过调用 CustomDefaultFilterChainManager.proxy(originalChain, chainNames)进行合并后代理。

CustomDefaultFilterChainManager

```
public class CustomDefaultFilterChainManager extends DefaultFilterChainManager {
    private Map<String, String> filterChainDefinitionMap = null;
    private String loginUrl;
    private String successUrl;
    private String unauthorizedUrl;
    public CustomDefaultFilterChainManager() {
        setFilters(new LinkedHashMap<String, Filter>());
        setFilterChains(new LinkedHashMap<String, NamedFilterList>());
        addDefaultFilters(true);
    }
    public Map<String, String> getFilterChainDefinitionMap() {
        return filterChainDefinitionMap;
    }
    public void setFilterChainDefinitionMap(Map<String, String> filterChainDefinitionMap) {
        this.filterChainDefinitionMap = filterChainDefinitionMap;
    }
    public void setCustomFilters(Map<String, Filter> customFilters) {
        for(Map.Entry<String, Filter> entry : customFilters.entrySet()) {
            addFilter(entry.getKey(), entry.getValue(), false);
        }
    }
    public void setDefaultFilterChainDefinitions(String definitions) {
        Ini ini = new Ini();
        ini.load(definitions);
        Ini.Section section = ini.getSection(IniFilterChainResolverFactory.URLS);
        if (CollectionUtils.isEmpty(section)) {
            section = ini.getSection(Ini.DEFAULT_SECTION_NAME);
        }
        setFilterChainDefinitionMap(section);
    }
    public String getLoginUrl() {
        return loginUrl;
    }
    public void setLoginUrl(String loginUrl) {
        this.loginUrl = loginUrl;
    }
```

```
    public String getSuccessUrl() {
        return successUrl;
    }
    public void setSuccessUrl(String successUrl) {
        this.successUrl = successUrl;
    }
    public String getUnauthorizedUrl() {
        return unauthorizedUrl;
    }
    public void setUnauthorizedUrl(String unauthorizedUrl) {
        this.unauthorizedUrl = unauthorizedUrl;
    }
    @PostConstruct
    public void init() {
        Map<String, Filter> filters = getFilters();
        if (!CollectionUtils.isEmpty(filters)) {
            for (Map.Entry<String, Filter> entry : filters.entrySet()) {
                String name = entry.getKey();
                Filter filter = entry.getValue();
                applyGlobalPropertiesIfNecessary(filter);
                if (filter instanceof Nameable) {
                    ((Nameable) filter).setName(name);
                }
                addFilter(name, filter, false);
            }
        }
        Map<String, String> chains = getFilterChainDefinitionMap();
        if (!CollectionUtils.isEmpty(chains)) {
            for (Map.Entry<String, String> entry : chains.entrySet()) {
                String url = entry.getKey();
                String chainDefinition = entry.getValue();
                createChain(url, chainDefinition);
            }
        }
    }
    protected void initFilter(Filter filter) {
        //ignore
    }
```

```
    public FilterChain proxy(FilterChain original, List<String> chainNames) {
        NamedFilterList configured = new SimpleNamedFilterList(chainNames.toString());
        for(String chainName : chainNames) {
            configured.addAll(getChain(chainName));
        }
        return configured.proxy(original);
    }
    private void applyGlobalPropertiesIfNecessary(Filter filter) {
        applyLoginUrlIfNecessary(filter);
        applySuccessUrlIfNecessary(filter);
        applyUnauthorizedUrlIfNecessary(filter);
    }
    private void applyLoginUrlIfNecessary(Filter filter) {
        //请参考源码
    }
    private void applySuccessUrlIfNecessary(Filter filter) {
        //请参考源码
    }
    private void applyUnauthorizedUrlIfNecessary(Filter filter) {
        //请参考源码
    }
}
```

1、CustomDefaultFilterChainManager：调用其构造器时，会自动注册默认的拦截器；
2、loginUrl、successUrl、unauthorizedUrl：分别对应登录地址、登录成功后默认跳转地址、未授权跳转地址，用于给相应拦截器的；
3、filterChainDefinitionMap：用于存储如 ShiroFilterFactoryBean 在配置文件中配置的拦截器链定义，即可以认为是默认的静态拦截器链；会自动与数据库中加载的合并；
4、setDefaultFilterChainDefinitions：解析配置文件中传入的字符串拦截器链配置，解析为相应的拦截器链；
5、setCustomFilters：注册我们自定义的拦截器；如 ShiroFilterFactoryBean 的 filters 属性；
6、init：初始化方法，Spring 容器启动时会调用，首先其会自动给相应的拦截器设置如 loginUrl、successUrl、unauthorizedUrl；其次根据 filterChainDefinitionMap 构建默认的拦截器链；
7、initFilter：此处我们忽略实现 initFilter，因为交给 spring 管理了，所以 Filter 的相关配置会在 Spring 配置中完成；
8、proxy：组合多个拦截器链为一个生成一个新的 FilterChain 代理。

Web 层控制器

请参考 com.github.zhangkaitao.shiro.chapter19.web.controller 包，相对于第十六章添加了 UrlFilterController 用于 UrlFilter 的维护。另外，移除了控制器方法上的权限注解，而是使用动态 URL 拦截进行控制。

Spring 配置——spring-config-shiro.xml

```
<bean id="filterChainManager"
      class="com.github.zhangkaitao.shiro.spring.CustomDefaultFilterChainManager">
      <property name="loginUrl" value="/login"/>
      <property name="successUrl" value="/"/>
      <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
      <property name="customFilters">
            <util:map>
                  <entry key="authc" value-ref="formAuthenticationFilter"/>
                  <entry key="sysUser" value-ref="sysUserFilter"/>
            </util:map>
      </property>
      <property name="defaultFilterChainDefinitions">
            <value>
                  /login = authc
                  /logout = logout
                  /unauthorized.jsp = authc
                  /** = user,sysUser
            </value>
      </property>
</bean>
```

filterChainManager 是我们自定义的 CustomDefaultFilterChainManager，注册相应的拦截器及默认的拦截器链。

```
<bean id="filterChainResolver"
      class="com.github.zhangkaitao.shiro.spring.CustomPathMatchingFilterChainResolver">
      <property name="customDefaultFilterChainManager" ref="filterChainManager"/>
</bean>
```

filterChainResolver 是自定义的 CustomPathMatchingFilterChainResolver，使用上边的 filterChainManager 进行拦截器链的管理。

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

shiroFilter 不再定义 filters 及 filterChainDefinitions，而是交给了 filterChainManager 进行完成。

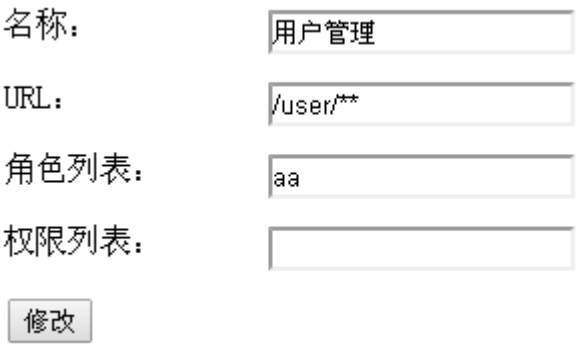
```
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="shiroFilter"/>
    <property name="targetMethod" value="setFilterChainResolver"/>
    <property name="arguments" ref="filterChainResolver"/>
</bean>
```

最后把 filterChainResolver 注册给 shiroFilter，其使用它进行动态 URL 权限控制。

其他配置和第十六章一样，请参考第十六章。

测试

1、首先执行 shiro-data.sql 初始化数据。
2、然后再 URL 管理中新增如下数据：



3、访问 http://localhost:8080/chapter19/user 时要求用户拥有 aa 角色，此时是没有的所以会跳转到未授权页面；
4、添加 aa 角色然后授权给用户，此时就有权限访问 http://localhost:8080/chapter19/user。

实际项目可以在此基础上进行扩展。

第二十章 无状态 Web 应用集成

在一些环境中，可能需要把 Web 应用做成无状态的，即服务器端无状态，就是说服务器端不会存储像会话这种东西，而是每次请求时带上相应的用户名进行登录。如一些 REST 风格的 API，如果不使用 OAuth2 协议，就可以使用如 REST+HMAC 认证进行访问。HMAC（Hash-based Message Authentication Code）：基于散列的消息认证码，使用一个密钥和一个消息作为输入，生成它们的消息摘要。注意该密钥只有客户端和服务端知道，其他第三方是不知道的。访问时使用该消息摘要进行传播，服务端然后对该消息摘要进行验证。如果只传递用户名+密码的消息摘要，一旦被别人捕获可能会重复使用该摘要进行认证。解决办法如：
1、每次客户端申请一个 Token，然后使用该 Token 进行加密，而该 Token 是一次性的，即只能用一次；有点类似于 OAuth2 的 Token 机制，但是简单些；
2、客户端每次生成一个唯一的 Token，然后使用该 Token 加密，这样服务器端记录下这些 Token，如果之前用过就认为是非法请求。

为了简单，本文直接对请求的数据（即全部请求的参数）生成消息摘要，即无法篡改数据，但是可能被别人窃取而能多次调用。解决办法如上所示。

服务器端

对于服务器端，不生成会话，而是每次请求时带上用户身份进行认证。

服务控制器

```
@RestController
public class ServiceController {
    @RequestMapping("/hello")
    public String hello1(String[] param1, String param2) {
        return "hello" + param1[0] + param1[1] + param2;
    }
}
```

当访问/hello 服务时，需要传入 param1、param2 两个请求参数。

加密工具类

com.github.zhangkaitao.shiro.chapter20.codec.HmacSHA256Utils：

```
//使用指定的密码对内容生成消息摘要（散列值）
public static String digest(String key, String content);
//使用指定的密码对整个 Map 的内容生成消息摘要（散列值）
public static String digest(String key, Map<String, ?> map)
```

对 Map 生成消息摘要主要用于对客户端/服务器端来回传递的参数生成消息摘要。

Subject 工厂

```
public class StatelessDefaultSubjectFactory extends DefaultWebSubjectFactory {
    public Subject createSubject(SubjectContext context) {
        //不创建 session
        context.setSessionCreationEnabled(false);
        return super.createSubject(context);
    }
}
```

通过调用 context.setSessionCreationEnabled(false)表示不创建会话；如果之后调用 Subject.getSession()将抛出 DisabledSessionException 异常。

StatelessAuthcFilter

类似于 FormAuthenticationFilter，但是根据当前请求上下文信息每次请求时都要登录的认证过滤器。

```
public class StatelessAuthcFilter extends AccessControlFilter {
  protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
      return false;
  }
  protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
    //1、客户端生成的消息摘要
    String clientDigest = request.getParameter(Constants.PARAM_DIGEST);
    //2、客户端传入的用户身份
    String username = request.getParameter(Constants.PARAM_USERNAME);
    //3、客户端请求的参数列表
```

```
        Map<String, String[]> params =
            new HashMap<String, String[]>(request.getParameterMap());
        params.remove(Constants.PARAM_DIGEST);
        //4、生成无状态 Token
        StatelessToken token = new StatelessToken(username, params, clientDigest);
        try {
            //5、委托给 Realm 进行登录
            getSubject(request, response).login(token);
        } catch (Exception e) {
            e.printStackTrace();
            onLoginFail(response); //6、登录失败
            return false;
        }
        return true;
    }
    //登录失败时默认返回 401 状态码
    private void onLoginFail(ServletResponse response) throws IOException {
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        httpResponse.getWriter().write("login error");
    }
}
```

获取客户端传入的用户名、请求参数、消息摘要，生成 StatelessToken；然后交给相应的 Realm 进行认证。

StatelessToken

```
public class StatelessToken implements AuthenticationToken {
    private String username;
    private Map<String, ?> params;
    private String clientDigest;
    //省略部分代码
    public Object getPrincipal() {    return username;}
    public Object getCredentials() {    return clientDigest;}
}
```

用户身份即用户名；凭证即客户端传入的消息摘要。

StatelessRealm

用于认证的 Realm。

```
public class StatelessRealm extends AuthorizingRealm {
    public boolean supports(AuthenticationToken token) {
        //仅支持 StatelessToken 类型的 Token
        return token instanceof StatelessToken;
    }
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        //根据用户名查找角色，请根据需求实现
        String username = (String) principals.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo =  new SimpleAuthorizationInfo();
        authorizationInfo.addRole("admin");
        return authorizationInfo;
    }
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        StatelessToken statelessToken = (StatelessToken) token;
        String username = statelessToken.getUsername();
        String key = getKey(username);//根据用户名获取密钥（和客户端的一样）
        //在服务器端生成客户端参数消息摘要
        String serverDigest = HmacSHA256Utils.digest(key, statelessToken.getParams());
        //然后进行客户端消息摘要和服务器端消息摘要的匹配
        return new SimpleAuthenticationInfo(
                username,
                serverDigest,
                getName());
    }

    private String getKey(String username) {//得到密钥，此处硬编码一个
        if("admin".equals(username)) {
            return "dadadswdewq2ewdwqdwadsadasd";
        }
        return null;
    }
}
```

此处首先根据客户端传入的用户名获取相应的密钥，然后使用密钥对请求参数生成服务器端的消息摘要；然后与客户端的消息摘要进行匹配；如果匹配说明是合法客户端传入的；否则是非法的。这种方式是有漏洞的，一旦别人获取到该请求，可以重复请求；可以考虑之前介绍的解决方案。

Spring 配置——spring-config-shiro.xml

```
<!-- Realm 实现 -->
<bean id="statelessRealm"
   class="com.github.zhangkaitao.shiro.chapter20.realm.StatelessRealm">
      <property name="cachingEnabled" value="false"/>
</bean>
<!-- Subject 工厂 -->
<bean id="subjectFactory"
   class="com.github.zhangkaitao.shiro.chapter20.mgt.StatelessDefaultSubjectFactory"/>
<!-- 会话管理器 -->
<bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager">
      <property name="sessionValidationSchedulerEnabled" value="false"/>
</bean>
<!-- 安全管理器 -->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
      <property name="realm" ref="statelessRealm"/>
      <property name="subjectDAO.sessionStorageEvaluator.sessionStorageEnabled"
         value="false"/>
      <property name="subjectFactory" ref="subjectFactory"/>
      <property name="sessionManager" ref="sessionManager"/>
</bean>
<!-- 相当于调用 SecurityUtils.setSecurityManager(securityManager) -->
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
      <property name="staticMethod"
         value="org.apache.shiro.SecurityUtils.setSecurityManager"/>
      <property name="arguments" ref="securityManager"/>
</bean>
```

sessionManager 通过 sessionValidationSchedulerEnabled 禁用掉会话调度器，因为我们禁用掉了会话，所以没必要再定期过期会话了。

```
<bean id="statelessAuthcFilter"
      class="com.github.zhangkaitao.shiro.chapter20.filter.StatelessAuthcFilter"/>
```

每次请求进行认证的拦截器。

```
<!-- Shiro 的 Web 过滤器 -->
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="filters">
        <util:map>
            <entry key="statelessAuthc" value-ref="statelessAuthcFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            /**=statelessAuthc
        </value>
    </property>
</bean>
```

所有请求都将走 statelessAuthc 拦截器进行认证。

其他配置请参考源代码。

SpringMVC 学习请参考：
5 分钟构建 spring web mvc REST 风格 HelloWorld
http://jinnianshilongnian.iteye.com/blog/1996071
跟我学 SpringMVC
http://www.iteye.com/blogs/subjects/kaitao-springmvc

客户端

此处使用 SpringMVC 提供的 RestTemplate 进行测试。请参考如下文章进行学习：
Spring MVC 测试框架详解——客户端测试
http://jinnianshilongnian.iteye.com/blog/2007180
Spring MVC 测试框架详解——服务端测试
http://jinnianshilongnian.iteye.com/blog/2004660

此处为了方便，使用内嵌 jetty 服务器启动服务端：

http://jinnianshilongnian.

```
public class ClientTest {
    private static Server server;
    private RestTemplate restTemplate = new RestTemplate();
    @BeforeClass
    public static void beforeClass() throws Exception {
        //创建一个 server
        server = new Server(8080);
        WebAppContext context = new WebAppContext();
        String webapp = "shiro-example-chapter20/src/main/webapp";
        context.setDescriptor(webapp + "/WEB-INF/web.xml");  //指定 web.xml 配置文件
        context.setResourceBase(webapp);  //指定 webapp 目录
        context.setContextPath("/");
        context.setParentLoaderPriority(true);
        server.setHandler(context);
        server.start();
    }
    @AfterClass
    public static void afterClass() throws Exception {
        server.stop(); //当测试结束时停止服务器
    }
}
```

在整个测试开始之前开启服务器，整个测试结束时关闭服务器。

测试成功情况

```
@Test
public void testServiceHelloSuccess() {
    String username = "admin";
    String param11 = "param11";
    String param12 = "param12";
    String param2 = "param2";
    String key = "dadadswdewq2ewdwqdwadsadasd";
    MultiValueMap<String, String> params = new LinkedMultiValueMap<String, String>();
    params.add(Constants.PARAM_USERNAME, username);
    params.add("param1", param11);
    params.add("param1", param12);
    params.add("param2", param2);
    params.add(Constants.PARAM_DIGEST, HmacSHA256Utils.digest(key, params));
```

```
        String url = UriComponentsBuilder
                .fromHttpUrl("http://localhost:8080/hello")
                .queryParams(params).build().toUriString();
        ResponseEntity responseEntity = restTemplate.getForEntity(url, String.class);
        Assert.assertEquals("hello" + param11 + param12 + param2, responseEntity.getBody());
}
```

对请求参数生成消息摘要后带到参数中传递给服务器端，服务器端验证通过后访问相应服务，然后返回数据。

测试失败情况

```
@Test
public void testServiceHelloFail() {
        String username = "admin";
        String param11 = "param11";
        String param12 = "param12";
        String param2 = "param2";
        String key = "dadadswdewq2ewdwqdwadsadasd";
        MultiValueMap<String, String> params = new LinkedMultiValueMap<String, String>();
        params.add(Constants.PARAM_USERNAME, username);
        params.add("param1", param11);
        params.add("param1", param12);
        params.add("param2", param2);
        params.add(Constants.PARAM_DIGEST, HmacSHA256Utils.digest(key, params));
        params.set("param2", param2 + "1");

        String url = UriComponentsBuilder
                .fromHttpUrl("http://localhost:8080/hello")
                .queryParams(params).build().toUriString();
        try {
                ResponseEntity responseEntity = restTemplate.getForEntity(url, String.class);
        } catch (HttpClientErrorException e) {
                Assert.assertEquals(HttpStatus.UNAUTHORIZED, e.getStatusCode());
                Assert.assertEquals("login error", e.getResponseBodyAsString());
        }
}
```

在生成请求参数消息摘要后，篡改了参数内容，服务器端接收后进行重新生成消息摘要发现不一样，报 401 错误状态码。

到此，整个测试完成了，需要注意的是，为了安全性，请考虑本文开始介绍的相应解决方案。

SpringMVC 相关知识请参考

5 分钟构建 spring web mvc REST 风格 HelloWorld
http://jinnianshilongnian.iteye.com/blog/1996071
跟我学 SpringMVC
http://www.iteye.com/blogs/subjects/kaitao-springmvc
Spring MVC 测试框架详解——客户端测试
http://jinnianshilongnian.iteye.com/blog/2007180
Spring MVC 测试框架详解——服务端测试
http://jinnianshilongnian.iteye.com/blog/2004660

http://jinnianshilongnian.iteye.com/

第二十一章 授予身份及切换身份

在一些场景中，比如某个领导因为一些原因不能进行登录网站进行一些操作，他想把他网站上的工作委托给他的秘书，但是他不想把帐号/密码告诉他秘书，只是想把工作委托给他；此时和我们可以使用 Shiro 的 RunAs 功能，即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。

本章代码基于《第十六章 综合实例》，请先了解相关数据模型及基本流程后再学习本章。

表及数据 SQL

请运行 shiro-example-chapter21/sql/ shiro-schema.sql 表结构
请运行 shiro-example-chapter21/sql/ shiro-schema.sql 数据

实体

具体请参考 com.github.zhangkaitao.shiro.chapter21 包下的实体。

```
public class UserRunAs implements Serializable {
    private Long fromUserId;//授予身份帐号
    private Long toUserId;//被授予身份帐号
}
```

该实体定义了授予身份帐号（A）与被授予身份帐号（B）的关系，意思是 B 帐号将可以假装为 A 帐号的身份进行访问。

DAO

具体请参考 com.github.zhangkaitao.shiro.chapter21.dao 包下的 DAO 接口及实现。

Service

具体请参考 com.github.zhangkaitao.shiro.chapter21.service 包下的 Service 接口及实现。

```
public interface UserRunAsService {
    public void grantRunAs(Long fromUserId, Long toUserId);
    public void revokeRunAs(Long fromUserId, Long toUserId);
    public boolean exists(Long fromUserId, Long toUserId);
```

http://jinnianshilongnian.iteye.com/

```
    public List<Long> findFromUserIds(Long toUserId);
    public List<Long> findToUserIds(Long fromUserId);
}
```

提供授予身份、回收身份、关系存在判断及查找 API。

Web 控制器 RunAsController

该控制器完成：授予身份/回收身份/切换身份功能。

展示当前用户能切换到身份列表，及授予给其他人的身份列表：

```
@RequestMapping
public String runasList(@CurrentUser User loginUser, Model model) {
    model.addAttribute("fromUserIds",
        userRunAsService.findFromUserIds(loginUser.getId()));
    model.addAttribute("toUserIds", userRunAsService.findToUserIds(loginUser.getId()));
    List<User> allUsers = userService.findAll();
    allUsers.remove(loginUser);
    model.addAttribute("allUsers", allUsers);

    Subject subject = SecurityUtils.getSubject();
    model.addAttribute("isRunas", subject.isRunAs());
    if(subject.isRunAs()) {
        String previousUsername =
                (String)subject.getPreviousPrincipals().getPrimaryPrincipal();
        model.addAttribute("previousUsername", previousUsername);
    }
    return "runas";
}
```

1、Subject.isRunAs()：表示当前用户是否是 RunAs 用户，即已经切换身份了；
2、Subject.getPreviousPrincipals()：得到切换身份之前的身份，一个用户可以切换很多次身份，之前的身份使用栈数据结构来存储；

授予身份

把当前用户身份授予给另一个用户，这样另一个用户可以切换身份到该用户。

```
@RequestMapping("/grant/{toUserId}")
public String grant(
        @CurrentUser User loginUser,
        @PathVariable("toUserId") Long toUserId,
        RedirectAttributes redirectAttributes) {
    if(loginUser.getId().equals(toUserId)) {
        redirectAttributes.addFlashAttribute("msg", "自己不能切换到自己的身份");
        return "redirect:/runas";
    }
    userRunAsService.grantRunAs(loginUser.getId(), toUserId);
    redirectAttributes.addFlashAttribute("msg", "操作成功");
    return "redirect:/runas";
}
```

1、自己不能授予身份给自己；

2、调用 UserRunAsService. grantRunAs 把当前登录用户的身份授予给相应的用户；

回收身份

把授予给某个用户的身份回收回来。

```
@RequestMapping("/revoke/{toUserId}")
public String revoke(
        @CurrentUser User loginUser,
        @PathVariable("toUserId") Long toUserId,
        RedirectAttributes redirectAttributes) {
    userRunAsService.revokeRunAs(loginUser.getId(), toUserId);
    redirectAttributes.addFlashAttribute("msg", "操作成功");
    return "redirect:/runas";
}
```

切换身份

```
@RequestMapping("/switchTo/{switchToUserId}")
public String switchTo(
        @CurrentUser User loginUser,
        @PathVariable("switchToUserId") Long switchToUserId,
        RedirectAttributes redirectAttributes) {
    Subject subject = SecurityUtils.getSubject();
    User switchToUser = userService.findOne(switchToUserId);
```

```
        if(loginUser.equals(switchToUser)) {
            redirectAttributes.addFlashAttribute("msg", "自己不能切换到自己的身份");
            return "redirect:/runas";
        }
        if(switchToUser == null || !userRunAsService.exists(switchToUserId, loginUser.getId())) {
            redirectAttributes.addFlashAttribute("msg", "对方没有授予您身份，不能切换");
            return "redirect:/runas";
        }
        subject.runAs(new SimplePrincipalCollection(switchToUser.getUsername(), ""));
        redirectAttributes.addFlashAttribute("msg", "操作成功");
        redirectAttributes.addFlashAttribute("needRefresh", "true");
        return "redirect:/runas";
    }
```

1、首先根据 switchToUserId 查找到要切换到的身份；
2、然后通过 UserRunAsService. exists()判断当前登录用户是否可以切换到该身份；
3、通过 Subject.runAs()切换到该身份；

切换到上一个身份

```
@RequestMapping("/switchBack")
public String switchBack(RedirectAttributes redirectAttributes) {
    Subject subject = SecurityUtils.getSubject();
    if(subject.isRunAs()) {
        subject.releaseRunAs();
    }
    redirectAttributes.addFlashAttribute("msg", "操作成功");
    redirectAttributes.addFlashAttribute("needRefresh", "true");
    return "redirect:/runas";
}
```

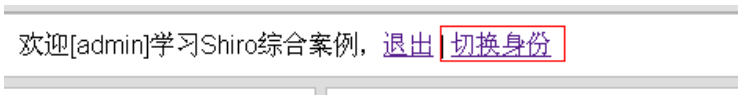
1、通过 Subject.releaseRunAs()切换会上一个身份；

此处注意的是我们可以切换多次身份，如 A 切换到 B，然后再切换到 C；那么需要调用两次 Subject. releaseRunAs()才能切换会 A；即内部使用栈数据结构存储着切换过的用户；Subject. getPreviousPrincipals()得到上一次切换到的身份，比如当前是 C；那么调用该 API 将得到 B 的身份。

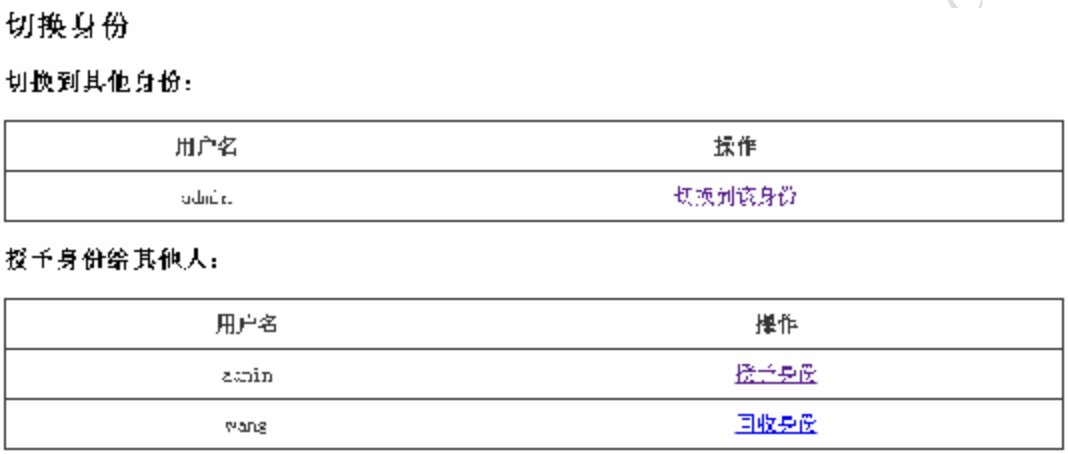
其他代码和配置和《第十六章 综合实例》一样，请参考该章。

测试

1、首先访问 http://localhost:8080/chapter21/，输入 admin/123456 进行登录；会看到如下界面：

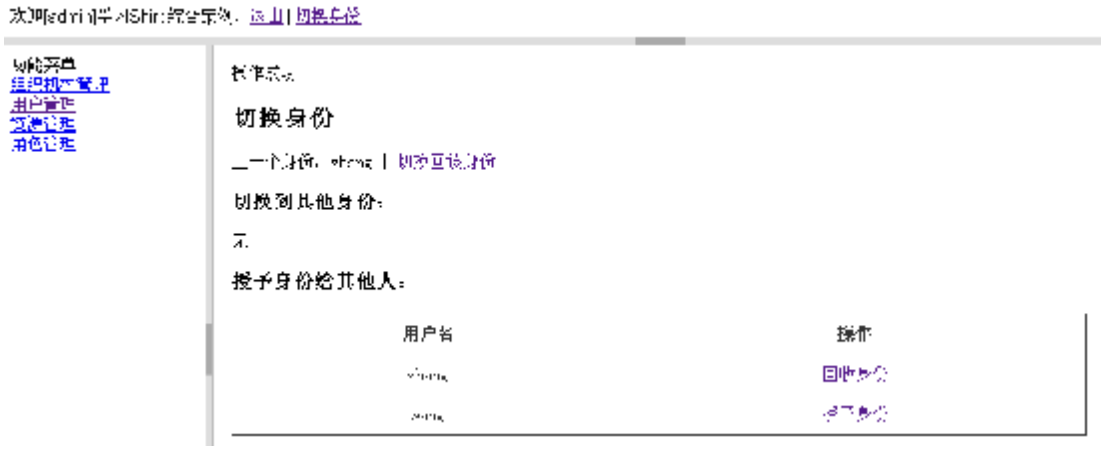


2、点击切换身份按钮，跳到如下界面：



在该界面可以授权身份给其他人（点击授权身份可以把自己的身份授权给其他人/点击回收身份可以把之前授予的身份撤回）、或切换到其他身份（即假装为其他身份运行）；

3、点击切换到该身份按钮，切换到相应的身份运行，如：



此时 zhang 用户切换到 admin 身份；如果点击切换回该身份，会把当前身份切换会 zhang。

第二十二章 集成验证码

在做用户登录功能时，很多时候都需要验证码支持，验证码常用目的是为了防止机器人模拟真实用户登录而恶意访问，如暴力破解用户密码/恶意评论等。目前也有一些验证码比较简单，通过一些 OCR 工具就可以解析出来；另外还有一些验证码比较复杂（一般通过如扭曲、加线条/噪点等干扰）防止 OCR 工具识别；但是在中国就是人多，机器干不了的可以交给人来完成，所以在中国就有很多打码平台，人工识别验证码；因此即使比较复杂的如填字、算数等类型的验证码还是能识别的。所以验证码也不是绝对可靠的，目前比较可靠还是手机验证码，但是对于用户来说相对于验证码还是比较麻烦的。

对于验证码图片的生成，可以自己通过如 Java 提供的图像 API 自己去生成，也可以借助如 JCaptcha 这种开源 Java 类库生成验证码图片；JCaptcha 提供了常见的如扭曲、加噪点等干扰支持。本章代码基于《第十六章 综合实例》。

一、添加 JCaptcha 依赖

```
<dependency>
    <groupId>com.octo.captcha</groupId>
    <artifactId>jcaptcha</artifactId>
    <version>2.0-alpha-1</version>
</dependency>
<dependency>
    <groupId>com.octo.captcha</groupId>
    <artifactId>jcaptcha-integration-simple-servlet</artifactId>
    <version>2.0-alpha-1</version>
    <exclusions>
        <exclusion>
            <artifactId>servlet-api</artifactId>
            <groupId>javax.servlet</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

com.octo.captcha . jcaptcha 提供了 jcaptcha 核心；而 jcaptcha-integration-simple-servlet 提供了与 Servlet 集成。

二、GMailEngine

来自 https://code.google.com/p/musicvalley/source/browse/trunk/musicvalley/doc/springSecurity/springSecurityIII/src/main/java/com/spring/security/jcaptcha/GMailEngine.java?spec=svn447&r=447（目前无法访问了），仿照 JCaptcha2.0 编写类似 GMail 验证码的样式；具体请参考 com.github.zhangkaitao.shiro.chapter22.jcaptcha.GMailEngine。

三、MyManageableImageCaptchaService

提供了判断仓库中是否有相应的验证码存在。

```
public class MyManageableImageCaptchaService extends
    DefaultManageableImageCaptchaService {
        public MyManageableImageCaptchaService(
            com.octo.captcha.service.captchastore.CaptchaStore captchaStore,
            com.octo.captcha.engine.CaptchaEngine captchaEngine,
            int minGuarantedStorageDelayInSeconds,
            int maxCaptchaStoreSize,
            int captchaStoreLoadBeforeGarbageCollection) {
                super(captchaStore, captchaEngine, minGuarantedStorageDelayInSeconds,
                        maxCaptchaStoreSize, captchaStoreLoadBeforeGarbageCollection);
        }
        public boolean hasCapcha(String id, String userCaptchaResponse) {
                return store.getCaptcha(id).validateResponse(userCaptchaResponse);
        }
}
```

四、JCaptcha 工具类

提供相应的 API 来验证当前请求输入的验证码是否正确。

```
public class JCaptcha {
        public static final MyManageableImageCaptchaService captchaService
                = new MyManageableImageCaptchaService(new FastHashMapCaptchaStore(),
                                new GMailEngine(), 180, 100000, 75000);
        public static boolean validateResponse(
                HttpServletRequest request, String userCaptchaResponse) {
                if (request.getSession(false) == null) return false;
                boolean validated = false;
                try {
```

```
            String id = request.getSession().getId();
            validated =
                captchaService.validateResponseForID(id, userCaptchaResponse)
                        .booleanValue();
        } catch (CaptchaServiceException e) {
            e.printStackTrace();
        }
        return validated;
    }
    public static boolean hasCaptcha(
        HttpServletRequest request, String userCaptchaResponse) {
        if (request.getSession(false) == null) return false;
        boolean validated = false;
        try {
            String id = request.getSession().getId();
            validated = captchaService.hasCapcha(id, userCaptchaResponse);
        } catch (CaptchaServiceException e) {
            e.printStackTrace();
        }
        return validated;
    }
}
```

validateResponse()：验证当前请求输入的验证码否正确；并从 CaptchaService 中删除已经生成的验证码；
hasCaptcha()：验证当前请求输入的验证码是否正确；但不从 CaptchaService 中删除已经生成的验证码（比如 Ajax 验证时可以使用，防止多次生成验证码）；

五、JCaptchaFilter

用于生成验证码图片的过滤器。

```
public class JCaptchaFilter extends OncePerRequestFilter {
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {

        response.setDateHeader("Expires", 0L);
        response.setHeader("Cache-Control", "no-store, no-cache, must-revalidate");
        response.addHeader("Cache-Control", "post-check=0, pre-check=0");
```

```
            response.setHeader("Pragma", "no-cache");
            response.setContentType("image/jpeg");
            String id = request.getRequestedSessionId();
            BufferedImage bi = JCaptcha.captchaService.getImageChallengeForID(id);
            ServletOutputStream out = response.getOutputStream();
            ImageIO.write(bi, "jpg", out);
            try {
                out.flush();
            } finally {
                out.close();
            }
        }
}
```

CaptchaService 使用当前会话 ID 当作 key 获取相应的验证码图片；另外需要设置响应内容不进行浏览器端缓存。

```
<!-- 验证码过滤器需要放到 Shiro 之后 因为 Shiro 将包装 HttpSession 如果不，可能造成两次的 sesison id 不一样 -->
<filter>
  <filter-name>JCaptchaFilter</filter-name>
  <filter-class>
    com.github.zhangkaitao.shiro.chapter22.jcaptcha.JCaptchaFilter
  </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>JCaptchaFilter</filter-name>
    <url-pattern>/jcaptcha.jpg</url-pattern>
</filter-mapping>
```

这样就可以在页面使用/jcaptcha.jpg 地址显示验证码图片。

六、JCaptchaValidateFilter

用于验证码验证的 Shiro 过滤器。

```
public class JCaptchaValidateFilter extends AccessControlFilter {
    private boolean jcaptchaEbabled = true;//是否开启验证码支持
    private String jcaptchaParam = "jcaptchaCode";//前台提交的验证码参数名
    private String failureKeyAttribute = "shiroLoginFailure"; //验证失败后存储到的属性名
```

```
    public void setJcaptchaEbabled(boolean jcaptchaEbabled) {
        this.jcaptchaEbabled = jcaptchaEbabled;
    }
    public void setJcaptchaParam(String jcaptchaParam) {
        this.jcaptchaParam = jcaptchaParam;
    }
    public void setFailureKeyAttribute(String failureKeyAttribute) {
        this.failureKeyAttribute = failureKeyAttribute;
    }
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response,
Object mappedValue) throws Exception {
        //1、设置验证码是否开启属性，页面可以根据该属性来决定是否显示验证码
        request.setAttribute("jcaptchaEbabled", jcaptchaEbabled);

        HttpServletRequest httpServletRequest = WebUtils.toHttp(request);
        //2、判断验证码是否禁用 或不是表单提交（允许访问）
        if (jcaptchaEbabled == false
|| !"post".equalsIgnoreCase(httpServletRequest.getMethod())) {
            return true;
        }
        //3、此时是表单提交，验证验证码是否正确
        return JCaptcha.validateResponse(httpServletRequest,
httpServletRequest.getParameter(jcaptchaParam));
    }
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response)
throws Exception {
        //如果验证码失败了，存储失败 key 属性
        request.setAttribute(failureKeyAttribute, "jCaptcha.error");
        return true;
    }
}
```

七、MyFormAuthenticationFilter

用于验证码验证的 Shiro 拦截器在用于身份认证的拦截器之前运行；但是如果验证码验证拦截器失败了，就不需要进行身份认证拦截器流程了；所以需要修改下如 FormAuthenticationFilter 身份认证拦截器，当验证码验证失败时不再走身份认证拦截器。

```
public class MyFormAuthenticationFilter extends FormAuthenticationFilter {
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
        if(request.getAttribute(getFailureKeyAttribute()) != null) {
            return true;
        }
        return super.onAccessDenied(request, response, mappedValue);
    }
}
```

即如果之前已经错了，那直接跳过即可。

八、spring-config-shiro.xml

```
<!-- 基于 Form 表单的身份验证过滤器 -->
<bean id="authcFilter"
  class="com.github.zhangkaitao.shiro.chapter22.jcaptcha.MyFormAuthenticationFilter">
    <property name="usernameParam" value="username"/>
    <property name="passwordParam" value="password"/>
    <property name="rememberMeParam" value="rememberMe"/>
    <property name="failureKeyAttribute" value="shiroLoginFailure"/>
</bean>
<bean id="jCaptchaValidateFilter"
  class="com.github.zhangkaitao.shiro.chapter22.jcaptcha.JCaptchaValidateFilter">
    <property name="jcaptchaEbabled" value="true"/>
    <property name="jcaptchaParam" value="jcaptchaCode"/>
    <property name="failureKeyAttribute" value="shiroLoginFailure"/>
</bean>
<!-- Shiro 的 Web 过滤器 -->
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="/login"/>
    <property name="filters">
        <util:map>
            <entry key="authc" value-ref="authcFilter"/>
            <entry key="sysUser" value-ref="sysUserFilter"/>
            <entry key="jCaptchaValidate" value-ref="jCaptchaValidateFilter"/>
        </util:map>
```

```
    </property>
    <property name="filterChainDefinitions">
        <value>
            /static/** = anon
            /jcaptcha* = anon
            /login = jCaptchaValidate,authc
            /logout = logout
            /authenticated = authc
            /** = user,sysUser
        </value>
    </property>
</bean>
```

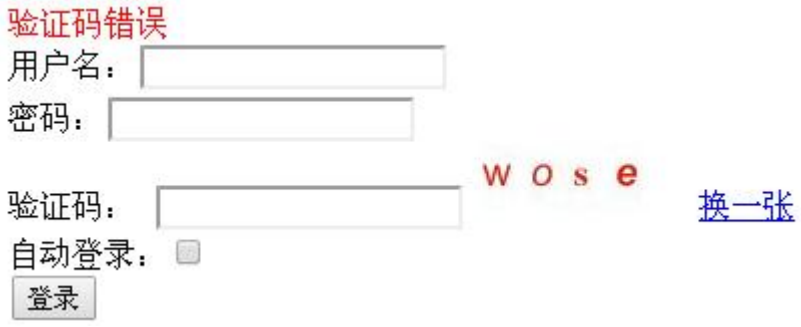
九、login.jsp 登录页面

```
<c:if test="${jcaptchaEbabled}">
    验证码：
    <input type="text" name="jcaptchaCode">
    <img class="jcaptcha-btn jcaptcha-img"
      src="${pageContext.request.contextPath}/jcaptcha.jpg" title="点击更换验证码">
    <a class="jcaptcha-btn" href="javascript:;">换一张</a>
    <br/>
</c:if>
```

根据 jcaptchaEbabled 来显示验证码图片。

十、测试

输入 http://localhost:8080/chapter22 将重定向到登录页面；输入正确的用户名/密码/验证码即可成功登录，如果输入错误的验证码，将显示验证码错误页面：



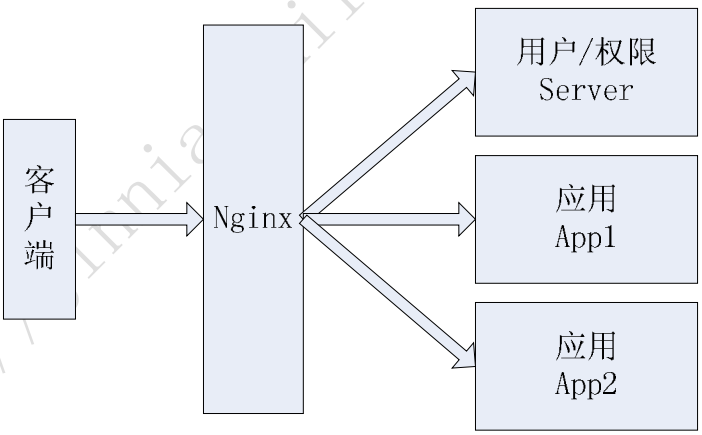
第二十三章 多项目集中权限管理及分布式会话

在做一些企业内部项目时或一些互联网后台时；可能会涉及到集中权限管理，统一进行多项目的权限管理；另外也需要统一的会话管理，即实现单点身份认证和授权控制。

学习本章之前，请务必先学习《第十章 会话管理》和《第十六章 综合实例》，本章代码都是基于这两章的代码基础上完成的。

本章示例是同域名的场景下完成的，如果跨域请参考《第十五章 单点登录》和《第十七章 OAuth2 集成》了解使用 CAS 或 OAuth2 实现跨域的身份验证和授权。另外比如客户端/服务器端的安全校验可参考《第二十章 无状态 Web 应用集成》。

部署架构



1、有三个应用：用于用户/权限控制的 Server（端口：8080）；两个应用 App1（端口 9080）和 App2（端口 10080）；

2、使用 Nginx 反向代理这三个应用，nginx.conf 的 server 配置部分如下：

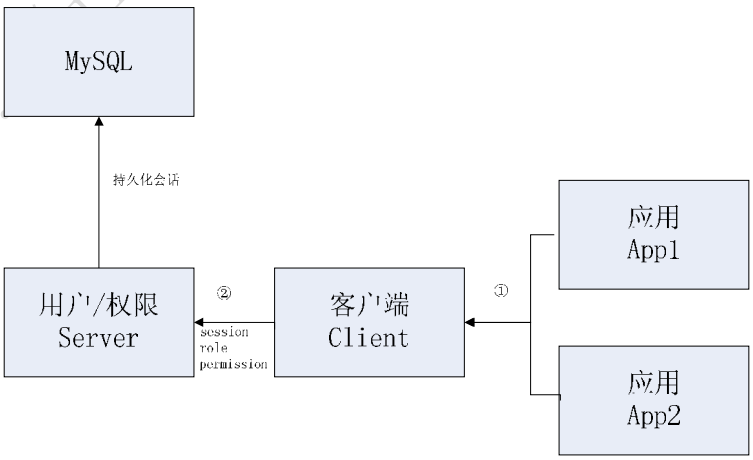
```
server {
    listen 80;
    server_name    localhost;
    charset utf-8;
```

```
        location ~ ^/(chapter23-server)/ {
            proxy_pass http://127.0.0.1:8080;
            index /;
                proxy_set_header Host $host;
        }
        location ~ ^/(chapter23-app1)/ {
            proxy_pass http://127.0.0.1:9080;
            index /;
                proxy_set_header Host $host;
        }
        location ~ ^/(chapter23-app2)/ {
            proxy_pass http://127.0.0.1:10080;
            index /;
                proxy_set_header Host $host;
        }
}
```

如访问 http://localhost/chapter23-server 会自动转发到 http://localhost:8080/chapter23-server；访问 http://localhost/chapter23-app1 会自动转发到 http://localhost:9080/chapter23-app1；访问 http://localhost/chapter23-app3 会自动转发到 http://localhost:10080/chapter23-app3；

Nginx 的安装及使用请自行搜索学习，本文不再阐述。

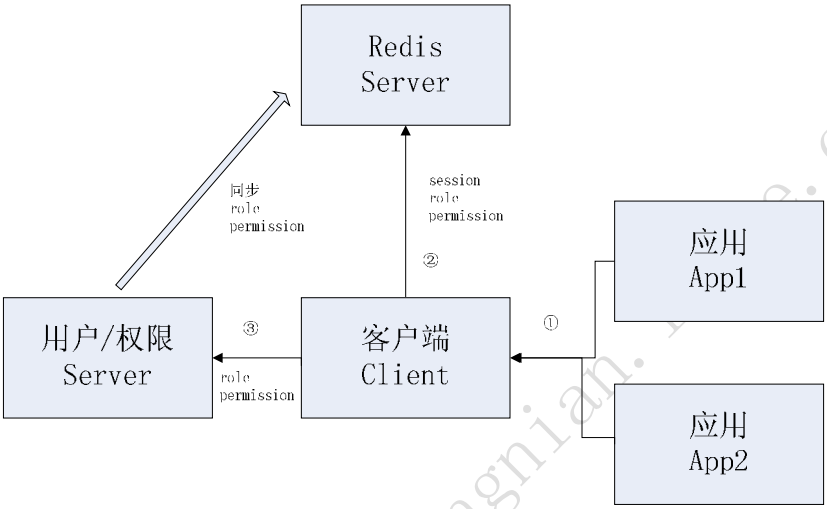
项目架构



http://jinnianshilongnian

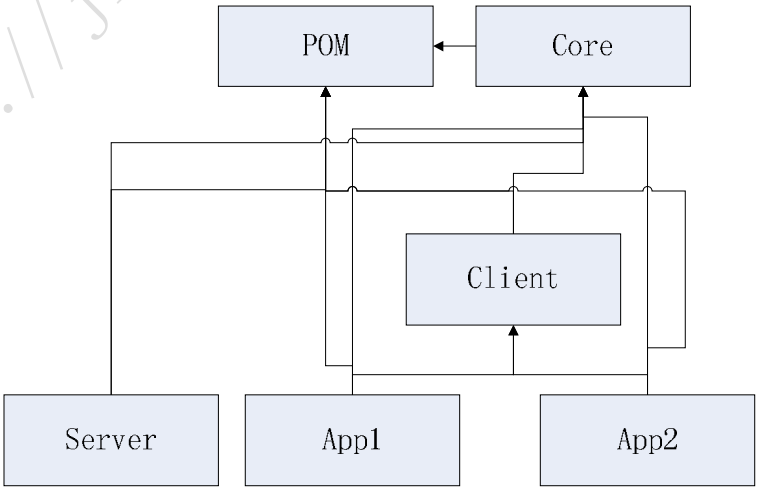
1、首先通过用户/权限 Server 维护用户、应用、权限信息；数据都持久化到 MySQL 数据库中；
2、应用 App1/应用 App2 使用客户端 Client 远程调用用户/权限 Server 获取会话及权限信息。

此处使用 Mysql 存储会话，而不是使用如 Memcached/Redis 之类的，主要目的是降低学习成本；如果换成如 Redis 也不会很难；如：

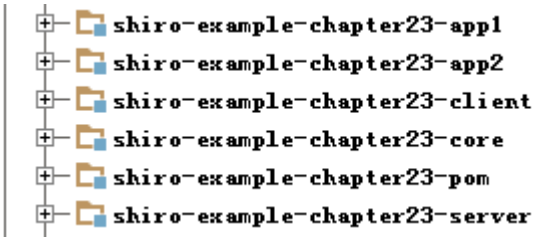


使用如 Redis 还一个好处就是无需在用户/权限 Server 中开会话过期调度器，可以借助 Redis 自身的过期策略来完成。

模块关系依赖



http://jinnianshilongnian.iteye.com/



1、shiro-example-chapter23-pom 模块：提供了其他所有模块的依赖；这样其他模块直接继承它即可，简化依赖配置，如 shiro-example-chapter23-server：

```
<parent>
    <artifactId>shiro-example-chapter23-pom</artifactId>
    <groupId>com.github.zhangkaitao</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

2、shiro-example-chapter23-core 模块：提供给 shiro-example-chapter23-server、shiro-example-chapter23-client、shiro-example-chapter23-app*模块的核心依赖，比如远程调用接口等；

3、shiro-example-chapter23-server 模块：提供了用户、应用、权限管理功能；

4、shiro-example-chapter23-client 模块：提供给应用模块获取会话及应用对应的权限信息；

5、shiro-example-chapter23-app*模块：各个子应用，如一些内部管理系统应用；其登录都跳到 shiro-example-chapter23-server 登录；另外权限都从 shiro-example-chapter23-server 获取（如通过远程调用）。

shiro-example-chapter23-pom 模块

其 pom.xml 的 packaging 类型为 pom，并且在该 pom 中加入其他模块需要的依赖，然后其他模块只需要把该模块设置为 parent 即可自动继承这些依赖，如 shiro-example-chapter23-server 模块：

```
<parent>
    <artifactId>shiro-example-chapter23-pom</artifactId>
    <groupId>com.github.zhangkaitao</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

简化其他模块的依赖配置等。

shiro-example-chapter23-core 模块

提供了其他模块共有的依赖，如远程调用接口：

```
public interface RemoteServiceInterface {
    public Session getSession(String appKey, Serializable sessionId);
    Serializable createSession(Session session);
    public void updateSession(String appKey, Session session);
    public void deleteSession(String appKey, Session session);
    public PermissionContext getPermissions(String appKey, String username);
}
```

提供了会话的 CRUD，及根据应用 key 和用户名获取权限上下文（包括角色和权限字符串）；shiro-example-chapter23-server 模块服务端实现；shiro-example-chapter23-client 模块客户端调用。

另外提供了 com.github.zhangkaitao.shiro.chapter23.core.ClientSavedRequest，其扩展了 org.apache.shiro.web.util.SavedRequest；用于 shiro-example-chapter23-app*模块当访问一些需要登录的请求时，自动把请求保存下来，然后重定向到 shiro-example-chapter23-server 模块登录；登录成功后再重定向回来；因为 SavedRequest 不保存 URL 中的 schema://domain:port 部分；所以才需要扩展 SavedRequest；使得 ClientSavedRequest 能保存 schema://domain:port；这样才能从一个应用重定向另一个（要不然只能在一个应用内重定向）：

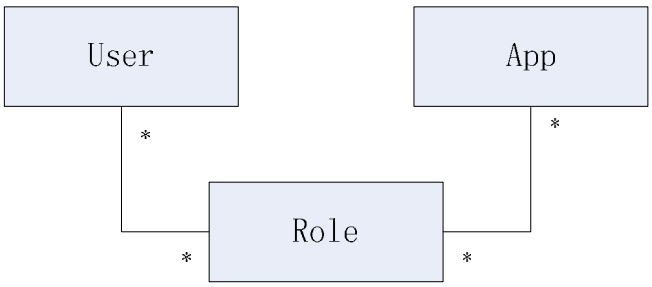
```
    public String getRequestUrl() {
        String requestURI = getRequestURI();
        if(backUrl != null) {//1
            if(backUrl.toLowerCase().startsWith("http://") ||
backUrl.toLowerCase().startsWith("https://")) {
                return backUrl;
            } else if(!backUrl.startsWith(contextPath)) {//2
                requestURI = contextPath + backUrl;
            } else {//3
                requestURI = backUrl;
            }
        }
        StringBuilder requestUrl = new StringBuilder(scheme);//4
        requestUrl.append("://");
        requestUrl.append(domain);//5
```

```
        //6
        if("http".equalsIgnoreCase(scheme) && port != 80) {
            requestUrl.append(":").append(String.valueOf(port));
        } else if("https".equalsIgnoreCase(scheme) && port != 443) {
            requestUrl.append(":").append(String.valueOf(port));
        }
        //7
        requestUrl.append(requestURI);
        //8
        if (backUrl == null && getQueryString() != null) {
            requestUrl.append("?").append(getQueryString());
        }
        return requestUrl.toString();
    }
```

1、如果从外部传入了 successUrl（登录成功之后重定向的地址），且以 http://或 https://开头那么直接返回（相应的拦截器直接重定向到它即可）；
2、如果 successUrl 有值但没有上下文，拼上上下文；
3、否则，如果 successUrl 有值，直接赋值给 requestUrl 即可；否则，如果 successUrl 没值，那么 requestUrl 就是当前请求的地址；
5、拼上 url 前边的 schema，如 http 或 https；
6、拼上域名；
7、拼上重定向到的地址（带上下文）；
8、如果 successUrl 没值，且有查询参数，拼上；
9 返回该地址，相应的拦截器直接重定向到它即可。

shiro-example-chapter23-server 模块

简单的实体关系图



简单数据字典

用户(sys_user)

名称	类型	长度	描述
id	bigint		编号 主键
username	varchar	100	用户名
password	varchar	100	密码
salt	varchar	50	盐
locked	bool		账户是否锁定

应用(sys_app)

名称	类型	长度	描述
id	bigint		编号 主键
name	varchar	100	应用名称
app_key	varchar	100	应用 key（唯一）
app_secret	varchar	100	应用安全码
available	bool		是否锁定

授权(sys_authorization)

名称	类型	长度	描述
id	bigint		编号 主键
user_id	bigint		所属用户
app_id	bigint		所属应用
role_ids	varchar	100	角色列表

用户：比《第十六章 综合实例》少了 role_ids，因为本章是多项目集中权限管理；所以授权时需要指定相应的应用；而不是直接给用户授权；所以不能在用户中出现 role_ids 了；
应用：所有集中权限的应用；在此处需要指定应用 key(app_key)和应用安全码（app_secret），app 在访问 server 时需要指定自己的 app_key 和用户名来获取该 app 对应用户权限信息；另外 app_secret 可以认为 app 的密码，比如需要安全访问时可以考虑使用它，可参考《第二十章 无状态 Web 应用集成》。另外 available 属性表示该应用当前是否开启；如果 false 表示该应用当前不可用，即不能获取到相应的权限信息。
授权：给指定的用户在指定的 app 下授权，即角色是与用户和 app 存在关联关系。

因为本章使用了《第十六章 综合实例》代码，所以还有其他相应的表结构（本章未使用到）。

表/数据 SQL

具体请参考
sql/ shiro-schema.sql （表结构）
sql/ shiro-data.sql （初始数据）

实体

具体请参考 com.github.zhangkaitao.shiro.chapter23.entity 包下的实体，此处就不列举了。

DAO

具体请参考 com.github.zhangkaitao.shiro.chapter23.dao 包下的 DAO 接口及实现。

Service

具体请参考 com.github.zhangkaitao.shiro.chapter23.service 包下的 Service 接口及实现。以下是出了基本 CRUD 之外的关键接口：

```
public interface AppService {
    public Long findAppIdByAppKey(String appKey);// 根据 appKey 查找 AppId
}
```

```
public interface AuthorizationService {
    //根据 AppKey 和用户名查找其角色
    public Set<String> findRoles(String appKey, String username);
    //根据 AppKey 和用户名查找权限字符串
    public Set<String> findPermissions(String appKey, String username);
}
```

根据 AppKey 和用户名查找用户在指定应用中对于的角色和权限字符串。

UserRealm

```
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
    String username = (String)principals.getPrimaryPrincipal();
    SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
    authorizationInfo.setRoles(
        authorizationService.findRoles(Constants.SERVER_APP_KEY, username));
    authorizationInfo.setStringPermissions(
        authorizationService.findPermissions(Constants.SERVER_APP_KEY, username));
    return authorizationInfo;
}
```

此处需要调用 AuthorizationService 的 findRoles/findPermissions 方法传入 AppKey 和用户名来获取用户的角色和权限字符串集合。其他的和《第十六章 综合实例》代码一样。

ServerFormAuthenticationFilter

```
public class ServerFormAuthenticationFilter extends FormAuthenticationFilter {
    protected void issueSuccessRedirect(ServletRequest request, ServletResponse response)
throws Exception {
```

```
        String fallbackUrl = (String) getSubject(request, response)
                .getSession().getAttribute("authc.fallbackUrl");
        if(StringUtils.isEmpty(fallbackUrl)) {
            fallbackUrl = getSuccessUrl();
        }
        WebUtils.redirectToSavedRequest(request, response, fallbackUrl);
    }
}
```

因为是多项目登录，比如如果是从其他应用中重定向过来的，首先检查 Session 中是否有“authc.fallbackUrl”属性，如果有就认为它是默认的重定向地址；否则使用 Server 自己的 successUrl 作为登录成功后重定向到的地址。

MySqlSessionDAO

将会话持久化到 Mysql 数据库；此处大家可以将其实现为如存储到 Redis/Memcached 等，实现策略请参考《第十章 会话管理》中的会话存储/持久化章节的 MySessionDAO，完全一样。

MySqlSessionValidationScheduler

和《第十章 会话管理》中的会话验证章节部分中的 MySessionValidationScheduler 完全一样。如果使用如 Redis 之类的有自动过期策略的 DB，完全可以不用实现 SessionValidationScheduler，直接借助于这些 DB 的过期策略即可。

RemoteService

```
public class RemoteService implements RemoteServiceInterface {
    @Autowired    private AuthorizationService authorizationService;
    @Autowired    private SessionDAO sessionDAO;

    public Session getSession(String appKey, Serializable sessionId) {
        return sessionDAO.readSession(sessionId);
    }
    public Serializable createSession(Session session) {
        return sessionDAO.create(session);
    }
    public void updateSession(String appKey, Session session) {
        sessionDAO.update(session);
    }
```

```
    public void deleteSession(String appKey, Session session) {
        sessionDAO.delete(session);
    }
    public PermissionContext getPermissions(String appKey, String username) {
        PermissionContext permissionContext = new PermissionContext();
        permissionContext.setRoles(authorizationService.findRoles(appKey, username));
        permissionContext.setPermissions(authorizationService.findPermissions(appKey,
username));
        return permissionContext;
    }
}
```

将会使用 HTTP 调用器暴露为远程服务，这样其他应用就可以使用相应的客户端调用这些接口进行 Session 的集中维护及根据 AppKey 和用户名获取角色/权限字符串集合。此处没有实现安全校验功能，如果是局域网内使用可以通过限定 IP 完成；否则需要使用如《第二十章 无状态 Web 应用集成》中的技术完成安全校验。

然后在 spring-mvc-remote-service.xml 配置文件把服务暴露出去：

```
<bean id="remoteService"
  class="com.github.zhangkaitao.shiro.chapter23.remote.RemoteService"/>
<bean name="/remoteService"
  class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="remoteService"/>
    <property name="serviceInterface"
      value="com.github.zhangkaitao.shiro.chapter23.remote.RemoteServiceInterface"/>
</bean>
```

Shiro 配置文件 spring-config-shiro.xml

和《第十六章 综合实例》配置类似，但是需要在 shiroFilter 中的 filterChainDefinitions 中添加如下配置，即远程调用不需要身份认证：

```
/remoteService = anon
```

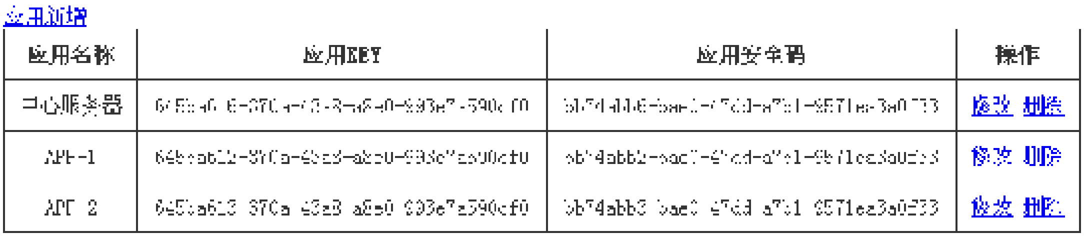
对于 userRealm 的缓存配置直接禁用；因为如果开启，修改了用户权限不会自动同步到缓存；另外请参考《第十一章 缓存机制》进行缓存的正确配置。

服务器端数据维护

1、首先开启 ngnix 反向代理；然后就可以直接访问 http://localhost/chapter23-server/；

2、输入默认的用户名密码：admin/123456 登录

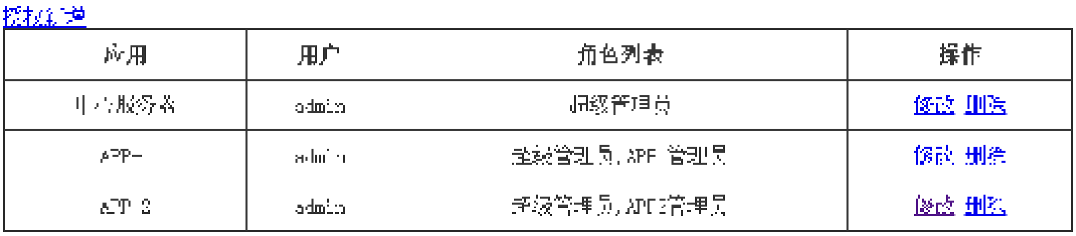
3、应用管理，进行应用的 CRUD，主要维护应用 KEY（必须唯一）及应用安全码；客户端就可以使用应用 KEY 获取用户对应应用的权限了。



应用新增

应用名称	应用KEY	应用安全码	操作
中心服务器	[illegible]	[illegible]	修改 删除
APP-1	[illegible]	[illegible]	修改 删除
APP-2	[illegible]	[illegible]	修改 删除

4、授权管理，维护在哪个应用中用户的角色列表。这样客户端就可以根据应用 KEY 及用户名获取到对应的角色/权限字符串列表了。



授权新增

应用	用户	角色列表	操作
中心服务器	admin	超级管理员	修改 删除
APP-1	admin	超级管理员,APP1管理员	修改 删除
APP-2	admin	超级管理员,APP2管理员	修改 删除

shiro-example-chapter23-client 模块

Client 模块提供给其他应用模块依赖，这样其他应用模块只需要依赖 Client 模块，然后再在相应的配置文件中配置如登录地址、远程接口地址、拦截器链等等即可，简化其他应用模块的配置。

配置远程服务 spring-client-remote-service.xml

```
<bean id="remoteService"
    class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <property name="serviceUrl" value="${client.remote.service.url}"/>
```

```
    <property name="serviceInterface"
      value="com.github.zhangkaitao.shiro.chapter23.remote.RemoteServiceInterface"/>
</bean>
```

client.remote.service.url 是远程服务暴露的地址；通过相应的 properties 配置文件配置，后续介绍。然后就可以通过 remoteService 获取会话及角色/权限字符串集合了。

ClientRealm

```
public class ClientRealm extends AuthorizingRealm {
    private RemoteServiceInterface remoteService;
    private String appKey;
    public void setRemoteService(RemoteServiceInterface remoteService) {
        this.remoteService = remoteService;
    }
    public void setAppKey(String appKey) {
        this.appKey = appKey;
    }
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        String username = (String) principals.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
        PermissionContext context = remoteService.getPermissions(appKey, username);
        authorizationInfo.setRoles(context.getRoles());
        authorizationInfo.setStringPermissions(context.getPermissions());
        return authorizationInfo;
    }
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        //永远不会被调用
        throw new UnsupportedOperationException("永远不会被调用");
    }
}
```

ClientRealm 提供身份认证信息和授权信息，此处因为是其他应用依赖客户端，而这些应用不会实现身份认证，所以 doGetAuthenticationInfo 获取身份认证信息直接无须实现。另外获取授权信息，是通过远程暴露的服务 RemoteServiceInterface 获取，提供 appKey 和用户名获取即可。

ClientSessionDAO

```
public class ClientSessionDAO extends CachingSessionDAO {
    private RemoteServiceInterface remoteService;
    private String appKey;
    public void setRemoteService(RemoteServiceInterface remoteService) {
        this.remoteService = remoteService;
    }
    public void setAppKey(String appKey) {
        this.appKey = appKey;
    }
    protected void doDelete(Session session) {
        remoteService.deleteSession(appKey, session);
    }
    protected void doUpdate(Session session) {
        remoteService.updateSession(appKey, session);
    }
    protected Serializable doCreate(Session session) {
        Serializable sessionId = remoteService.createSession(session);
        assignSessionId(session, sessionId);
        return sessionId;
    }
    protected Session doReadSession(Serializable sessionId) {
        return remoteService.getSession(appKey, sessionId);
    }
}
```

Session 的维护通过远程暴露接口实现，即本地不维护会话。

ClientAuthenticationFilter

```
public class ClientAuthenticationFilter extends AuthenticationFilter {
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {
        Subject subject = getSubject(request, response);
        return subject.isAuthenticated();
    }
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
        String backUrl = request.getParameter("backUrl");
        saveRequest(request, backUrl, getDefaultBackUrl(WebUtils.toHttp(request)));
```

```
        return false;
    }
    protected void saveRequest(ServletRequest request, String backUrl, String fallbackUrl) {
        Subject subject = SecurityUtils.getSubject();
        Session session = subject.getSession();
        HttpServletRequest httpRequest = WebUtils.toHttp(request);
        session.setAttribute("authc.fallbackUrl", fallbackUrl);
        SavedRequest savedRequest = new ClientSavedRequest(httpRequest, backUrl);
        session.setAttribute(WebUtils.SAVED_REQUEST_KEY, savedRequest);
    }
    private String getDefaultBackUrl(HttpServletRequest request) {
        String scheme = request.getScheme();
        String domain = request.getServerName();
        int port = request.getServerPort();
        String contextPath = request.getContextPath();
        StringBuilder backUrl = new StringBuilder(scheme);
        backUrl.append("://");
        backUrl.append(domain);
        if("http".equalsIgnoreCase(scheme) && port != 80) {
            backUrl.append(":").append(String.valueOf(port));
        } else if("https".equalsIgnoreCase(scheme) && port != 443) {
            backUrl.append(":").append(String.valueOf(port));
        }
        backUrl.append(contextPath);
        backUrl.append(getSuccessUrl());
        return backUrl.toString();
    }
}
```

ClientAuthenticationFilter 是用于实现身份认证的拦截器（authc），当用户没有身份认证时；

1、首先得到请求参数 backUrl，即登录成功重定向到的地址；

2、然后保存保存请求到会话，并重定向到登录地址（server 模块）；

3、登录成功后，返回地址按照如下顺序获取：backUrl、保存的当前请求地址、defaultBackUrl（即设置的 successUrl）；

ClientShiroFilterFactoryBean

```
public class ClientShiroFilterFactoryBean extends ShiroFilterFactoryBean implements ApplicationContextAware {
    private ApplicationContext applicationContext;
    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }
    public void setFiltersStr(String filters) {
        if(StringUtils.isEmpty(filters)) {
            return;
        }
        String[] filterArray = filters.split(";");
        for(String filter : filterArray) {
            String[] o = filter.split("=");
            getFilters().put(o[0], (Filter)applicationContext.getBean(o[1]));
        }
    }
    public void setFilterChainDefinitionsStr(String filterChainDefinitions) {
        if(StringUtils.isEmpty(filterChainDefinitions)) {
            return;
        }
        String[] chainDefinitionsArray = filterChainDefinitions.split(";");
        for(String filter : chainDefinitionsArray) {
            String[] o = filter.split("=");
            getFilterChainDefinitionMap().put(o[0], o[1]);
        }
    }
}
```

1、setFiltersStr：设置拦截器，设置格式如“filterName=filterBeanName; filterName=filterBeanName”；多个之间分号分隔；然后通过 applicationContext 获取 filterBeanName 对应的 Bean 注册到拦截器 Map 中；

2、setFilterChainDefinitionsStr：设置拦截器链，设置格式如“url=filterName1[config],filterName2; url=filterName1[config],filterName2”；多个之间分号分隔；

Shiro 客户端配置 spring-client.xml

提供了各应用通用的 Shiro 客户端配置；这样应用只需要导入相应该配置即可完成 Shiro 的配置，简化了整个配置过程。

```
<context:property-placeholder location=
    "classpath:client/shiro-client-default.properties,classpath:client/shiro-client.properties"/>
```

提供给客户端配置的 properties 属性文件，client/shiro-client-default.properties 是客户端提供的默认的配置；classpath:client/shiro-client.properties 是用于覆盖客户端默认配置，各应用应该提供该配置文件，然后提供各应用个性配置。

```
<bean id="remoteRealm" class="com.github.zhangkaitao.shiro.chapter23.client.ClientRealm">
    <property name="cachingEnabled" value="false"/>
    <property name="appKey" value="${client.app.key}"/>
    <property name="remoteService" ref="remoteService"/>
</bean>
```

appKey：使用${client.app.key}占位符替换，即需要在之前的 properties 文件中配置。

```
<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
    <constructor-arg value="${client.session.id}"/>
    <property name="httpOnly" value="true"/>
    <property name="maxAge" value="-1"/>
    <property name="domain" value="${client.cookie.domain}"/>
    <property name="path" value="${client.cookie.path}"/>
</bean>
```

Session Id Cookie，cookie 名字、域名、路径等都是通过配置文件配置。

```
<bean id="sessionDAO"
  class="com.github.zhangkaitao.shiro.chapter23.client.ClientSessionDAO">
    <property name="sessionIdGenerator" ref="sessionIdGenerator"/>
    <property name="appKey" value="${client.app.key}"/>
    <property name="remoteService" ref="remoteService"/>
</bean>
```

SessionDAO 的 appKey，也是通过${ client.app.key }占位符替换，需要在配置文件配置。

```
<bean id="sessionManager"
  class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">
        <property name="sessionValidationSchedulerEnabled" value="false"/>//省略其他
</bean>
```

其他应用无须进行会话过期调度，所以 sessionValidationSchedulerEnabled=false。

```
<bean id="clientAuthenticationFilter"
   class="com.github.zhangkaitao.shiro.chapter23.client.ClientAuthenticationFilter"/>
```

应用的身份认证使用 ClientAuthenticationFilter，即如果没有身份认证，则会重定向到 Server 模块完成身份认证，身份认证成功后再重定向回来。

```
<bean id="shiroFilter"
   class="com.github.zhangkaitao.shiro.chapter23.client.ClientShiroFilterFactoryBean">
      <property name="securityManager" ref="securityManager"/>
      <property name="loginUrl" value="${client.login.url}"/>
      <property name="successUrl" value="${client.success.url}"/>
      <property name="unauthorizedUrl" value="${client.unauthorized.url}"/>
      <property name="filters">
            <util:map>
                  <entry key="authc" value-ref="clientAuthenticationFilter"/>
            </util:map>
      </property>
      <property name="filtersStr" value="${client.filters}"/>
      <property name="filterChainDefinitionsStr" value="${client.filter.chain.definitions}"/>
</bean>
```

ShiroFilter 使用我们自定义的 ClientShiroFilterFactoryBean，然后 loginUrl（登录地址）、successUrl（登录成功后默认的重定向地址）、unauthorizedUrl（未授权重定向到的地址）通过占位符替换方式配置；另外 filtersStr 和 filterChainDefinitionsStr 也是使用占位符替换方式配置；这样就可以在各应用进行自定义了。

默认配置 client/ shiro-client-default.properties

```
#各应用的 appKey
client.app.key=
#远程服务 URL 地址
client.remote.service.url=http://localhost/chapter23-server/remoteService
#登录地址
client.login.url=http://localhost/chapter23-server/login
#登录成功后，默认重定向到的地址
client.success.url=/
```

```
#未授权重定向到的地址
client.unauthorized.url=http://localhost/chapter23-server/unauthorized
#session id 域名
client.cookie.domain=
#session id 路径
client.cookie.path=/
#cookie 中的 session id 名称
client.session.id=sid
#cookie 中的 remember me 名称
client.rememberMe.id=rememberMe
#过滤器 name=filter-ref;name=filter-ref
client.filters=
#过滤器链 格式 url=filters;url=filters
client.filter.chain.definitions=/**=anon
```

在各应用中主要配置 client.app.key、client.filters、client.filter.chain.definitions。

shiro-example-chapter23-app*模块

继承 shiro-example-chapter23-pom 模块

```
<parent>
    <artifactId>shiro-example-chapter23-pom</artifactId>
    <groupId>com.github.zhangkaitao</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

依赖 shiro-example-chapter23-client 模块

```
<dependency>
    <groupId>com.github.zhangkaitao</groupId>
    <artifactId>shiro-example-chapter23-client</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

客户端配置 client/shiro-client.properties

配置 shiro-example-chapter23-app1

```
client.app.key=645ba612-370a-43a8-a8e0-993e7a590cf0
client.success.url=/hello
client.filter.chain.definitions=/hello=anon;/login=authc;/**=authc
```

client.app.key 是 server 模块维护的，直接拷贝过来即可；client.filter.chain.definitions 定义了拦截器链；比如访问/hello，匿名即可。

配置 shiro-example-chapter23-app2

```
client.app.key=645ba613-370a-43a8-a8e0-993e7a590cf0
client.success.url=/hello
client.filter.chain.definitions=/hello=anon;/login=authc;/**=authc
```

和 app1 类似，client.app.key 是 server 模块维护的，直接拷贝过来即可；client.filter.chain.definitions 定义了拦截器链；比如访问/hello，匿名即可。

web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:client/spring-client.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

指定加载客户端 Shiro 配置，client/spring-client.xml。

```
<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>
```

配置 ShiroFilter 拦截器。

控制器

shiro-example-chapter23-app1

```
@Controller
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        return "success";
    }
    @RequestMapping(value = "/attr", method = RequestMethod.POST)
    public String setAttr(
            @RequestParam("key") String key, @RequestParam("value") String value) {
        SecurityUtils.getSubject().getSession().setAttribute(key, value);
        return "success";
    }
    @RequestMapping(value = "/attr", method = RequestMethod.GET)
    public String getAttr(
            @RequestParam("key") String key, Model model) {
        model.addAttribute("value",
            SecurityUtils.getSubject().getSession().getAttribute(key));
        return "success";
    }
    @RequestMapping("/role1")
    @RequiresRoles("role1")
    public String role1() {
        return "success";
    }
}
```

shiro-example-chapter23-app2 的控制器类似，role2 方法使用@RequiresRoles("role2")注解，即需要角色 2。

其他配置请参考源码。

测试

1、安装配置启动 nginx

1、首先到 http://nginx.org/en/download.html 下载，比如我下载的是 windows 版本的；

2、然后编辑 conf/nginx.conf 配置文件，在 server 部分添加如下部分：

```
 location ~ ^/(chapter23-server)/ {
        proxy_pass http://127.0.0.1:8080;
        index /;
            proxy_set_header Host $host;
    }
    location ~ ^/(chapter23-app1)/ {
        proxy_pass http://127.0.0.1:9080;
        index /;
            proxy_set_header Host $host;
    }
    location ~ ^/(chapter23-app2)/ {
        proxy_pass http://127.0.0.1:10080;
        index /;
            proxy_set_header Host $host;
    }
```

3、最后双击 nginx.exe 启动 Nginx 即可。

已经配置好的 nginx 请到 shiro-example-chapter23-nginx 模块下下周 nginx-1.5.11.rar 即可。

2、安装依赖

1、首先安装 shiro-example-chapter23-core 依赖，到 shiro-example-chapter23-core 模块下运行 mvn install 安装 core 模块。

2、接着到 shiro-example-chapter23-client 模块下运行 mvn install 安装客户端模块。

3、启动 Server 模块

到 shiro-example-chapter23-server 模块下运行 mvn jetty:run 启动该模块；使用 http://localhost:8080/chapter23-server/即可访问，因为启动了 nginx，那么可以直接访问 http://localhost/chapter23-server/。

4、启动 App*模块

到 shiro-example-chapter23-app1 和 shiro-example-chapter23-app2 模块下分别运行 mvn

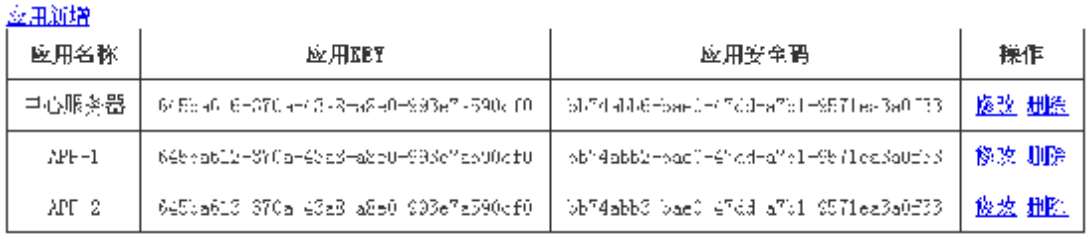
jetty:run 启动该模块；使用 http://localhost:9080/chapter23-app1/ 和 http://localhost:10080/chapter23-app2/即可访问，因为启动了 nginx，那么可以直接访问 http://localhost/chapter23-app1/和 http://localhost/chapter23-app2/。

5、服务器端维护

1、访问 http://localhost/chapter23-server/；

2、输入默认的用户名密码：admin/123456 登录

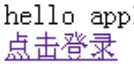
3、应用管理，进行应用的 CRUD，主要维护应用 KEY（必须唯一）及应用安全码；客户端就可以使用应用 KEY 获取用户对应应用的权限了。



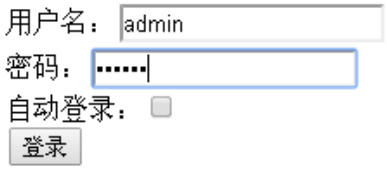
4、授权管理，维护在哪个应用中用户的角色列表。这样客户端就可以根据应用 KEY 及用户名获取到对应的角色/权限字符串列表了。

6、App*模块身份认证及授权

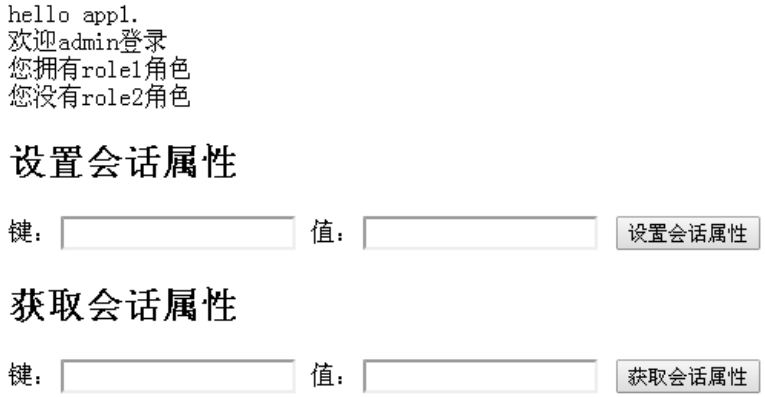
1、在未登录情况下访问 http://localhost/chapter23-app1/hello，看到下图：



2、登录地址是 http://localhost/chapter23-app1/login?backUrl=/chapter23-app1，即登录成功后重定向回 http://localhost/chapter23-app1（这是个错误地址，为了测试登录成功后重定向地址），点击登录按钮后重定向到 Server 模块的登录界面：

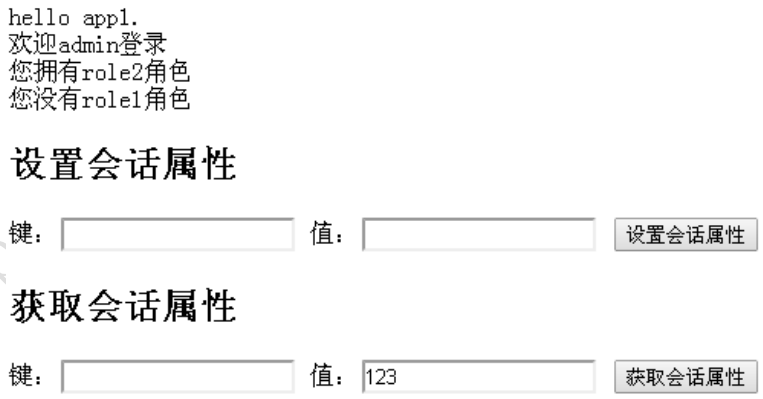


3、登录成功后，会重定向到相应的登录成功地址；接着访问 http://localhost/chapter23-app1/hello，看到如下图：



4、可以看到 admin 登录，及其是否拥有 role1/role2 角色；可以在 server 模块移除 role1 角色或添加 role2 角色看看页面变化；

5、可以在 http://localhost/chapter23-app1/hello 页面设置属性，如 key=123；接着访问 http://localhost/chapter23-app2/attr?key=key 就可以看到刚才设置的属性，如下图：



另外在 app2，用户默认拥有 role2 角色，而没有 role1 角色。

到此整个测试就完成了，可以看出本示例实现了：会话的分布式及权限的集中管理。

本示例缺点

1、没有加缓存；
2、客户端每次获取会话/权限都需要通过客户端访问服务端；造成服务端单点和请求压力大；单点可以考虑使用集群来解决；请求压力大需要考虑配合缓存服务器（如 Redis）来解决；即每次会话/权限获取时首先查询缓存中是否存在，如果有直接获取即可；否则再查服务端；降低请求压力；
3、会话的每次更新（比如设置属性/更新最后访问时间戳）都需要同步到服务端；也造成了请求压力过大；可以考虑在请求的最后只同步一次会话（需要对 Shiro 会话进行改造，通过如拦截器在执行完请求后完成同步，这样每次请求只同步一次）；
4、只能同域名才能使用，即会话 ID 是从同一个域名下获取，如果跨域请考虑使用 CAS/OAuth2 之实现。

所以实际应用时可能还是需要改造的，但大体思路是差不多的。

http://jinnianshilongnian.iteye.com/

第二十四章 在线会话管理

有时候需要显示当前在线人数、当前在线用户，有时候可能需要强制某个用户下线等；此时就需要获取相应的在线用户并进行一些操作。

本章基于《第十六章 综合实例》代码构建。

com/

会话控制器

```
@RequiresPermissions("session:*")
@Controller
@RequestMapping("/sessions")
public class SessionController {
    @Autowired
    private SessionDAO sessionDAO;
    @RequestMapping()
    public String list(Model model) {
        Collection<Session> sessions =  sessionDAO.getActiveSessions();
        model.addAttribute("sessions", sessions);
        model.addAttribute("sesessionCount", sessions.size());
        return "sessions/list";
    }
    @RequestMapping("/{sessionId}/forceLogout")
    public String forceLogout(@PathVariable("sessionId") String sessionId,
        RedirectAttributes redirectAttributes) {
        try {
            Session session = sessionDAO.readSession(sessionId);
            if(session != null) {
                session.setAttribute(
                    Constants.SESSION_FORCE_LOGOUT_KEY, Boolean.TRUE);
            }
        } catch (Exception e) {/*ignore*/}
        redirectAttributes.addFlashAttribute("msg", "强制退出成功！");
        return "redirect:/sessions";
    }
}
```

1、list 方法：提供了展示所有在线会话列表，通过 sessionDAO.getActiveSessions()获取所有

在线的会话。

2、forceLogout 方法：强制退出某一个会话，此处只在指定会话中设置 Constants.SESSION_FORCE_LOGOUT_KEY 属性，之后通过 ForceLogoutFilter 判断并进行强制退出。

此处展示会话列表的缺点是：sessionDAO.getActiveSessions()提供了获取所有活跃会话集合，如果做一般企业级应用问题不大，因为在线用户不多；但是如果应用的在线用户非常多，此种方法就不适合了，解决方案就是分页获取：

```
Page<Session> getActiveSessions(int pageNumber, int pageSize);
```

Page 对象除了包含 pageNumber、pageSize 属性之外，还包含 totalSessions（总会话数）、Collection<Session> （当前页的会话）。

分页获取时，如果是 MySQL 这种关系数据库存储会话比较好办，如果使用 Redis 这种数据库可以考虑这样存储：

```
session.id=会话序列化数据
session.ids=会话 id Set 列表（接着可以使用 LLEN 获取长度，LRANGE 分页获取）
```

会话创建时（如 sessionId=123），那么 redis 命令如下所示：

```
SET session.123 "Session 序列化数据"
LPUSH session.ids 123
```

会话删除时（如 sessionId=123），那么 redis 命令如下所示：

```
DEL session.123
LREM session.ids 123
```

获取总活跃会话：

```
LLEN session.ids
```

分页获取活跃会话：

```
LRANGE key 0 10 #获取到会话 ID
MGET session.1 session.2……  #根据第一条命令获取的会话 ID 获取会话数据
```

ForceLogoutFilter

```
public class ForceLogoutFilter extends AccessControlFilter {
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
        Session session = getSubject(request, response).getSession(false);
        if(session == null) {
            return true;
        }
        return session.getAttribute(Constants.SESSION_FORCE_LOGOUT_KEY) == null;
    }
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
        try {
            getSubject(request, response).logout();//强制退出
        } catch (Exception e) {/*ignore exception*/}
        String loginUrl = getLoginUrl() + (getLoginUrl().contains("?") ? "&" : "?") + "forceLogout=1";
        WebUtils.issueRedirect(request, response, loginUrl);
        return false;
    }
}
```

强制退出拦截器，如果用户会话中存在 Constants.SESSION_FORCE_LOGOUT_KEY 属性，表示被管理员强制退出了；然后调用 Subject.logout()退出，且重定向到登录页面（自动拼上 fourceLogout 请求参数）。

登录控制器

在 LoginController 类的 showLoginForm 方法中最后添加如下代码：

```
if(req.getParameter("forceLogout") != null) {
    model.addAttribute("error", "您已经被管理员强制退出，请重新登录");
}
```

即如果有请求参数 forceLogout 表示是管理员强制退出的，在界面上显示相应的信息。

Shiro 配置 spring-config-shiro.xml

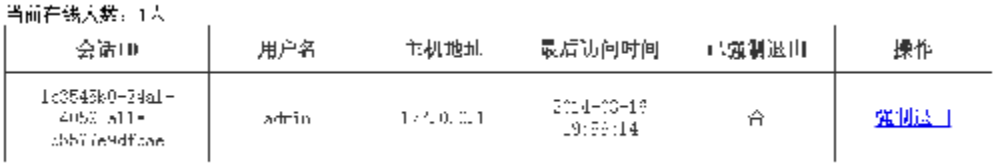
和之前的唯一区别是在 shiroFilter 中的 filterChainDefinitions 拦截器链定义中添加了 forceLogout 拦截器：

```
/** = forceLogout,user,sysUser
```

测试

1、首先输入 http://localhost:8080/chapter24/跳转到登录页面输入 admin/123456 登录；

2、登录成功后，点击菜单的“会话管理”，可以看到当前在线会话列表：



3、点击“强制退出”按钮，会话相应的用户再点击界面的话会看到如下界面，表示已经被强制退出了：

另外可参考我的 ES 中的在线会话管理功能：UserOnlineController.java，其使用数据库存储会话，并分页获取在线会话。

http://jinnianshilongnian.iteye.com/