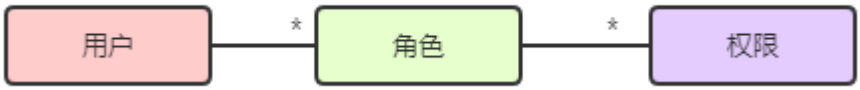
权限管理概述

权限管理，一般指根据系统设置的安全规则或者安全策略，用户可以访问而且只能访问自己被授权的资源，不多不少。权限管理几乎出现在任何系统里面，只要有用户和密码的系统。 很多人常将“用户身份认证”、“密码加密”、“系统管理”等概念与权限管理概念混淆。

在权限管理中使用最多的还是功能权限管理中的基于角色访问控制（RBAC，Role Based Access Control）。



当项目中需要使用权限管理的时候，我们可以选择自己去实现（前面的课程中所实现的RBAC系统），也可以选择使用第三方实现好的框架去实现，他们孰优孰劣这就需要看大家在项目中具体的需求了。

这里我们介绍两种常用的权限管理框架：

1. **Apache Shiro**

   Apache Shiro是一个强大且易用的Java安全框架,执行身份验证、授权、密码和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

2. **Spring Security**

   Spring Security以前叫做acegi，是后来才成为Spring的一个子项目，也是目前最为流行的一个安全权限管理框架，它与Spring紧密结合在一起。

   Spring Security关注的重点是在企业应用安全层为您提供服务，你将发现业务问题领域存在着各式各样的需求。银行系统跟电子商务应用就有很大的不同。电子商务系统与企业销售自动化工具又有很大不同。这些客户化需求让应用安全显得有趣，富有挑战性而且物有所值。Spring Security为基于J2EE的企业应用软件提供了一套全面的安全解决方案。

Shiro和Spring Security比较

1. Shiro比Spring更容易使用，实现和最重要的理解
2. Spring Security更加知名的唯一原因是因为品牌名称
3. “Spring”以简单而闻名，但讽刺的是很多人发现安装Spring Security很难
4. 然而，Spring Security却有更好的社区支持
5. Apache Shiro在Spring Security处理密码学方面有一个额外的模块
6. Spring-security 对spring 结合较好，如果项目用的springmvc ，使用起来很方便。但是如果项目中没有用到spring，那就不要考虑它了。
7. Shiro 功能强大、且 简单、灵活。是Apache 下的项目比较可靠，且不跟任何的框架或者容器绑定，可以独立运行

Shiro概述

什么是Apache Shiro?

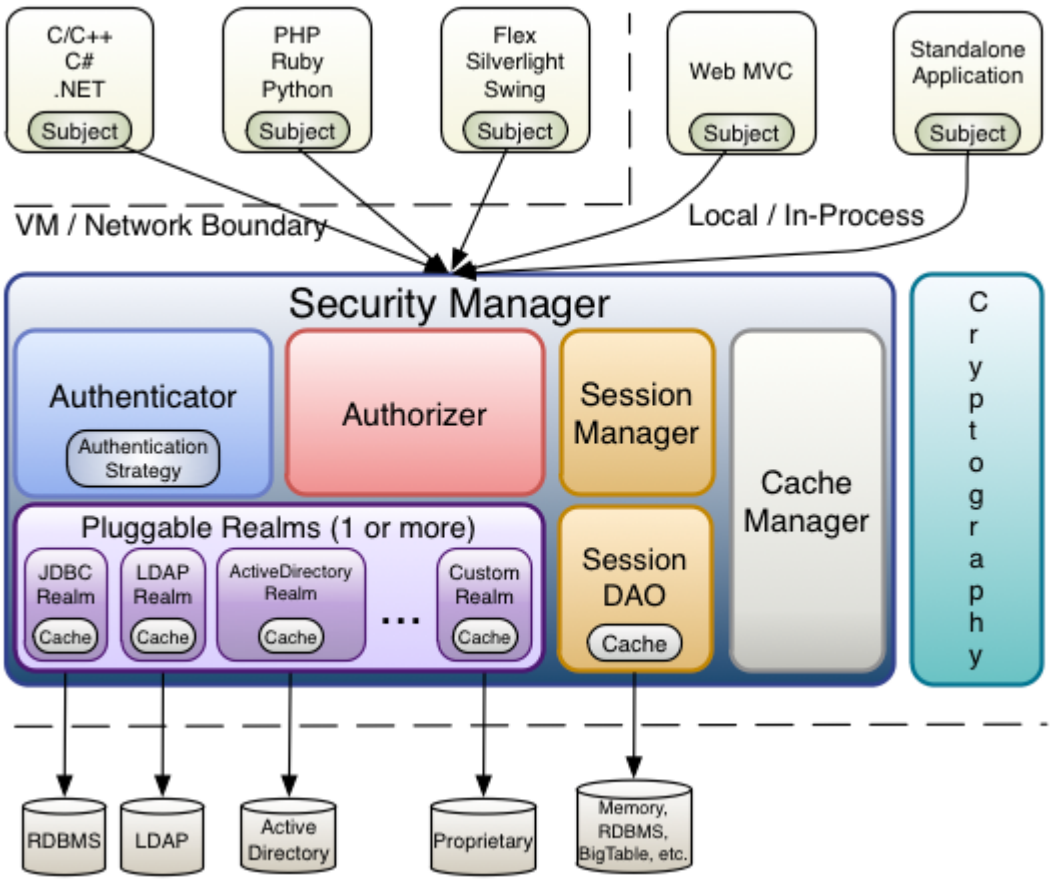
Apache Shiro是Java的一个安全框架。目前，使用Apache Shiro的人越来越多，因为它相当简单，对比Spring Security，可能没有Spring Security做的功能强大，但是在实际工作时可能并不需要那么复杂的东西，所以使用小而简单的Shiro就足够了。对于它俩到底哪个好，这个不必纠结，能更简单的解决项目问题就好了。

Shiro能做什么事情?

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。

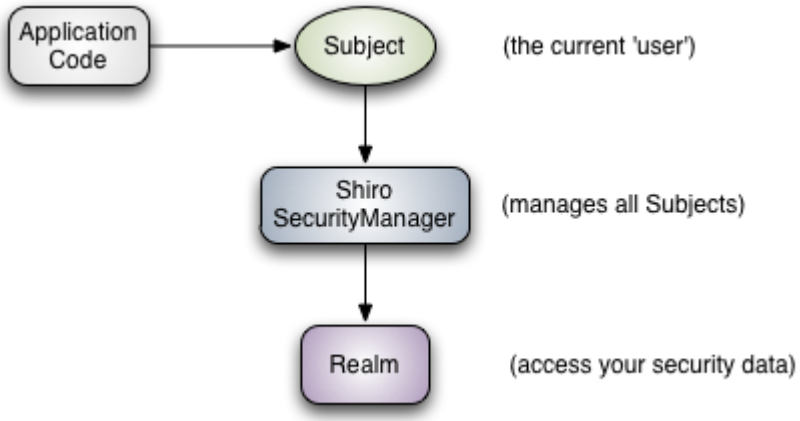
Shiro架构

Shiro主要组件包括：Subject，SecurityManager，Authenticator，Authorizer，SessionManager，CacheManager，Cryptography，Realms。



1. Subject（用户）： 访问系统的用户，主体可以是用户、程序等，进行认证的都称为主体；Subject一词是一个专业术语，其基本意思是“当前的操作用户”。它是一个抽象的概念，可以是人，也可以是第三方进程或其他类似事物，如爬虫，机器人等。 在程序任意位置：Subject currentUser = SecurityUtils.getSubject(); 类似 Employee user = UserContext.getUser() 一旦获得Subject，你就可以立即获得你希望用Shiro为当前用户做的90%的事情，如登录、登出、访问会话、执行授权检查等
2. SecurityManager（安全管理器） 安全管理器，它是shiro功能实现的核心，负责与后边介绍的其他组件(认证器/授权器/缓存控制器)进行交互，实现subject委托的各种功能。有点类似于SpringMVC中的DispatcherServlet前端控制器。
3. Realms（数据源） Realm充当了Shiro与应用安全数据间的“桥梁”或者“连接器”。；可以把Realm看成DataSource，即安全数据源。执行认证（登录）和授权（访问控制）时，Shiro会从应用配置的Realm中查找相关的比对数据。以确认用户是否合法，操作是否合理
4. Authenticator Authenticator用于认证，协调一个或者多个Realm，从Realm指定的数据源取得数据之后进行执行具体的认证。
5. Authorizer Authorizer用户访问控制授权，决定用户是否拥有执行指定操作的权限。
6. SessionManager Shiro与生俱来就支持会话管理，这在安全类框架中都是独一无二的功能。即便不存在web容器环境，shiro都可以使用自己的会话管理机制，提供相同的会话API。
7. CacheManager 缓存组件，用于缓存认证信息等。
8. Cryptography Shiro提供了一个加解密的命令行工具jar包，需要单独下载使用。

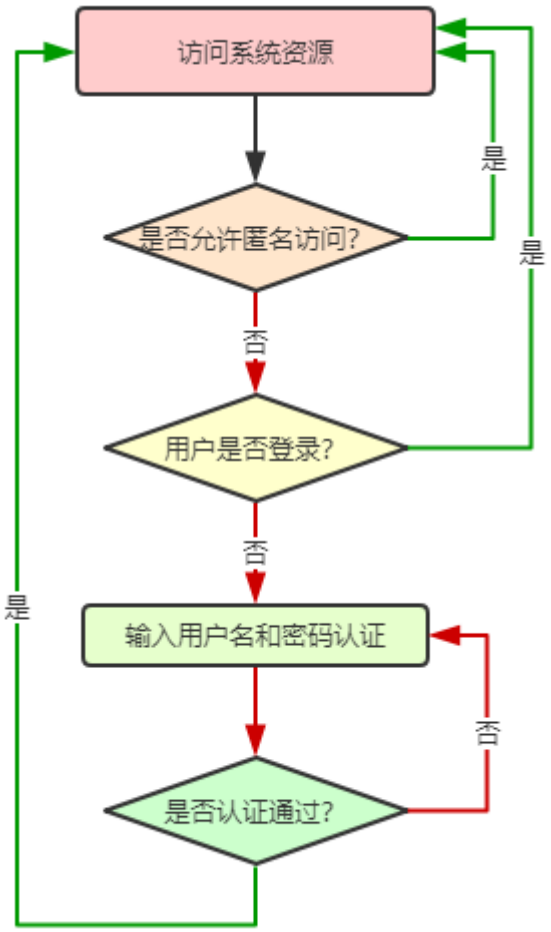
Shiro架构的核心（Subject，SecurityManager，Realms）



Shiro在应用中最常使用的两个功能：用户登录认证和访问授权

Shiro认证概述

认证的过程即为用户的身份确认过程，所实现的功能就是我们所熟悉的登录验证，用户输入账号和密码提交到后台，后台通过访问数据库执行账号密码的正确性校验。



上图表明了用户访问的认证流程，是否需要认证和是否已经通过认证，以及最后的如何完成认证是我们需要关心的，而这当中最关键的就是如何认证。

前面我们介绍过，Shiro不仅在web环境中可以使用，在JavaSE中一样可以完美的实现相关的功能，下面我们先来看看在JavaSES环境中它是如何实现认证功能的。

基于ini的Shiro认证

准备工作：

1. 导入基本的jar包

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</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>
```

2. 编写ini配置文件:shiro.ini

```
#用户的身份、凭据
[user]
zhangsan=555
lisi=666
```

使用Shiro相关的API完成身份认证

```
public void testShiro() throws Exception{
        //加载shiro.ini配置文件，得到配置中的用户信息（账号+密码）
        IniSecurityManagerFactory factory =
            new IniSecurityManagerFactory("classpath:shiro.ini");
        //创建Shiro的安全管理器
        SecurityManager manager = factory.getInstance();
        //将创建的安全管理器添加到运行环境中
        SecurityUtils.setSecurityManager(manager);
        //获取登录的用户主体对象
        Subject subject = SecurityUtils.getSubject();
        System.out.println("登录前的认证状态：+subject.isAuthenticated());//false
        //创建登录用户的身份凭证
        UsernamePasswordToken token = new UsernamePasswordToken("zhangsan",
"555");
        try {
```

```
            //登录认证
            subject.login(token);
        } catch (UnknownAccountException e) {
            e.printStackTrace();
            System.out.println("用户名错误");
        } catch (IncorrectCredentialsException e){
            e.printStackTrace();
            System.out.println("密码错误");
        }
        System.out.println("登录后的认证状态：+subject.isAuthenticated());//true
    }
```

步骤：

1. 加载shiro.ini配置文件，得到配置中的用户信息（账号+密码）
2. 创建Shiro的安全管理器，它在整个认证过程中担任控制器的角色
3. 将创建的安全管理器添加到运行环境中
4. 获取登录的用户主体对象，其中包含验证时需要的账号和密码
5. 创建登录用户的身份凭证，携带登录用户的账号和密码
6. 登录认证，将用户的和ini配置中的账号密码做匹配

如果输入的身份和凭证和ini文件中配置的能够匹配，那么登录成功，返回登录状态为true，反之登录状态为false。

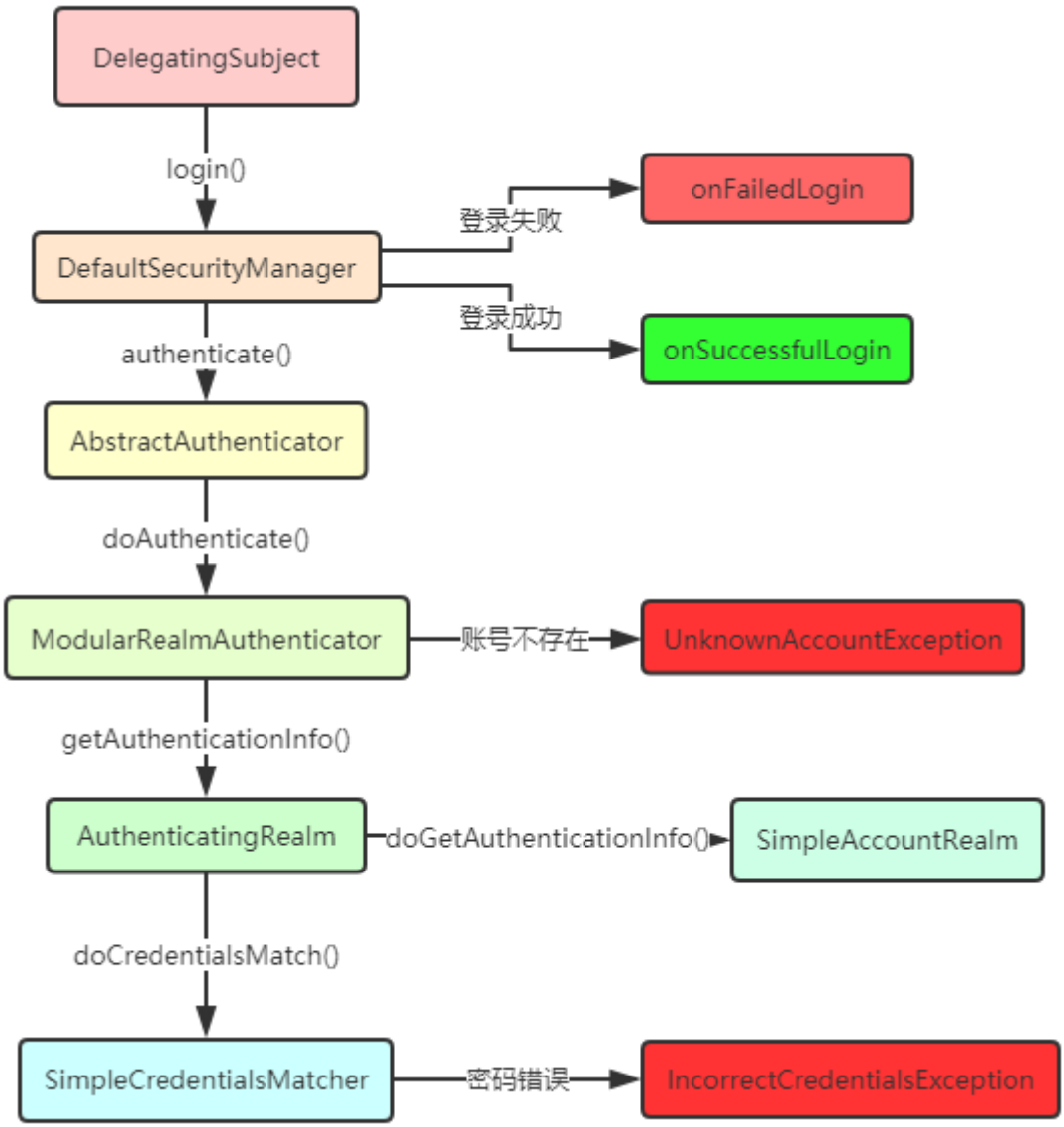
登录失败存在两种情况：

1. 账号错误

   org.apache.shiro.authc.**UnknownAccountException**
2. 密码错误

   org.apache.shiro.authc.**IncorrectCredentialsException**

Shiro认证流程分析（源码）



自定义Realm

自定义Realm在实际开发中使用非常多，应该我们需要使用的账户信息通常来自程序或者数据库中，而不是前面使用到的ini文件的配置。

在上面的例子中，我们通过断点调试的方式看到，我们默认使用的是SimpleAccountRealm来处理账户的校验。在自定义Reaml之前，我们先来看看他的实现方式。

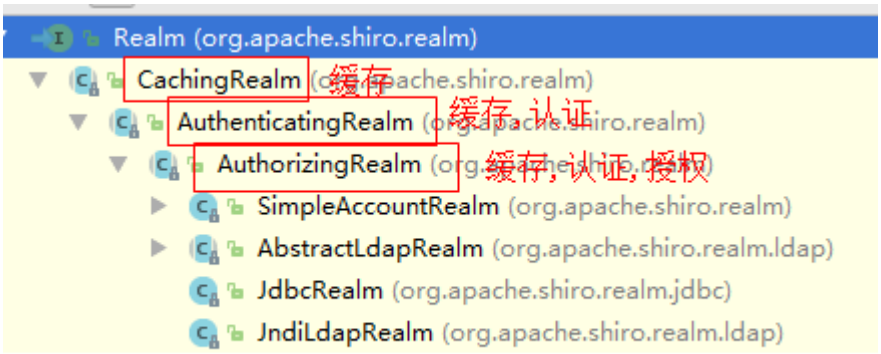
```
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)
throws AuthenticationException {
    //获取用户传入的账户信息
    UsernamePasswordToken upToken = (UsernamePasswordToken) token;
    //根据用户输入的账号到ini文件中获取对应的账户信息
    SimpleAccount account = getUser(upToken.getUsername());
    if (account != null) {
        if (account.isLocked()) {
            throw new LockedAccountException("Account [" + account + "] is
locked.");
```

```
        }
        if (account.isCredentialsExpired()) {
            String msg = "The credentials for account [" + account + "] are
expired";
            throw new ExpiredCredentialsException(msg);
        }
    }
    //返回获取到的账户信息
    return account;
}
```

在AuthenticatingRealm中调用该方法来判断账号是否正确，如果返回的account不等于空，说明账号存在，然后在进行密码校验。

所以，如果我们要自定义Reaml，应该覆写doGetAuthenticationInfo()方法，然后在该方法中实现账号的校验，并返回AuthenticationInfo对象给上层调用者AuthenticatingRealm做进一步的校验。

Reaml的继承体系



1. **自定义Reaml类**

在继承体系中的每个类所能够实现的功能不一样，在后面的开发中，我们通常需要使用到缓存，认证和授权所有的功能，所以选择继承AuthorizingRealm。

```
public class CRMRealm extends AuthorizingRealm {
    //获取认证信息
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
        //暂且使用假数据来模拟真实的账号和密码
        String username="zhangsan";
        String password="555";
        //如果账号正确，返回一个AuthenticationInfo对象
        if(username.equals(token.getPrincipal())){
            return new SimpleAuthenticationInfo(username,//账号
                                                password,//密码
                                                this.getName()//当前Realm的名称);
        }
        return null;
    }
```

```
    //获取授权信息
    protected AuthorizationInfo doGetAuthorizationInfo(
        PrincipalCollection principals) {
        return null;
    }
}
```

2. **通过配置修改SecurityManager中的默认Realm的使用**

shiro.ini

```
#自定义的Realm信息
crmRealm=cn.wolfcode.crm.shiro.CRMRealm
#将crmRealm设置到当前的环境中
securityManager.realms=$crmRealm
```

运行之前的测试代码，效果和前面的一样，但是现在使用的是我们自定义的Realm完成账号的校验。

在实际开发中，上面账户信息的假数据应该是从数据库中查询得到，下面我们将Shiro认证应用到CRM项目中

CRM中集成Shiro认证

首先思考下面几个问题：

1. Shiro认证是在访问系统资源的时候，对用户的身份等进行检查。那么身份检查操作应该在哪里实现呢?
2. 在SE环境中，我们使用的是DefaultSecurityManager来实现认证的控制，那么现在再EE环境中应该使用哪一个SecurityManager来实现认证控制呢?
3. 前面我们在ini配置文件中指定自定义的Realm，现在又是在哪里指定具体使用的Realm?
4. 前端页面需要根据后台认证的结果来处理页面的跳转，如何返回页面需要的数据呢?

以上几个问题就是我们继承Shiro认证功能时需要解决的问题，来吧，一个个搞定他!

1. 在访问的时候，需要做一系列的预处理操作，我们的最佳选择就是使用过滤器来实现了。

   ```
   <filter>
       <filter-name>shiroFilter</filter-name>
       <filter-class>
           org.springframework.web.filter.DelegatingFilterProxy
       </filter-class>
   </filter>
   <filter-mapping>
       <filter-name>shiroFilter</filter-name>
       <url-pattern>/*</url-pattern>
   </filter-mapping>
   ```

   shiro过虑器，DelegatingFilterProx会从spring容器中找shiroFilter，所以过滤器的生命周期还是交给Spring进行管理的。

Shiro中定义了多个过滤器来完成不同的预处理操作：

过滤器的名称	Java类
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter
user	org.apache.shiro.web.filter.authc.UserFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter

anon: 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”

authc: 表示需要认证(登录)才能使用;示例“/**=authc” 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: 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）；
roles:角色授权拦截器，验证用户是否拥有资源角色；示例“/admin/**=roles[admin]”
perms:权限授权拦截器，验证用户是否拥有资源权限；
示例“/employee/input=perms["user:update"]” **user:**用户拦截器，用户已经身份验证/记住我登录的都可；示例“/index=user” **logout:**注销拦截器，主要属性：redirectUrl：退出成功后重定向的地址（/）;示例“/logout=logout” **port:**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: 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:**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；

首先，为了单独对Shiro相关的配置进行管理，我们分离出一个shiro.xml配置文件，最后在mvc.xml中引入

mvc.xml

```
<import resource="classpath:shiro.xml"/>
```

shiro.xml：指定系统资源需要使用的具体过滤器

```
<bean id="shiroFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <!--引用指定的安全管理器-->
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="/login.html"/>
    <property name="filterChainDefinitions">
        <value>
            /js/**=anon
            /images/**=anon
            /css/**=anon
            /**=authc
        </value>
    </property>
</bean>
```

有了上面的配置，当我们的访问到达具体资源之前，会先进过指定的过滤器做预处理，在允许通过之后才能继续访问。

2. 解决安全管理器的指定问题，在JavaEE环境中，我们需要使用的安全管理器是：DefaultWebSecurityManager，并且在该安全管理器中指定我们自定义的Realm

```
<bean id="securityManager"
class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
        <property name="realm" ref="crmRealm"/>
</bean>
```

安全管理器要能够在Spring容器中找到指定的Realm，需要保证已经将Realm交给了Spring容器。

```
@Component("crmRealm")
public class CRMRealm extends AuthorizingRealm {

    @Autowired
    private EmployeeMapper mapper;
    //获取认证信息
    protected AuthenticationInfo
doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
        String username = token.getPrincipal().toString();
        Employee emp = mapper.selectEmployeeByUsername(username);
        if(emp == null){
            return null;
        }
        return new
SimpleAuthenticationInfo(emp,emp.getPassword(),this.getName());
    }

    //获取授权信息
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
        return null;
    }
}
```

有了这一步，Shiro就会通过自定义的Realm进行账号校验了。

3. 最后，当校验完成之后，需要告知前端页面校验的结果相关的信息。

   默认情况下会调用FormAuthenticationFilter中的onLoginFailure和onLoginSuccess方法对失败和成功的结果进行处理

```
protected boolean onLoginSuccess(AuthenticationToken token, Subject
subject,
            ServletRequest request, ServletResponse response) throws
Exception {
    issueSuccessRedirect(request, response);
    return false;
}
protected boolean onLoginFailure(AuthenticationToken token,
        AuthenticationException e,ServletRequest request, ServletResponse
response) {
    setFailureAttribute(request, e);
    return true;
}
```

前端页面中，登录请求相关的代码：

```
<script type="text/javascript">
    $(function(){
        $("#btn_submit").click(function () {
            $.post("/login.html", $("#loginForm").serialize(), function
(data) {
                if(data.success){
                    window.location.href="/employee/list.do";
                }else{
                    $.messager.alert("温馨提示",data.msg);
                }
            })
        })
    });
</script>
```

我们使用的ajax发送异步请求的方式来执行登录请求，最终想要获取到的是一个json格式的数据，所以我们必须修改FormAuthenticationFilter中两个方法的实现，修改源码中的实现，我们可以采用继承方式

```
@Component("crmFormAuthenticationFilter")
public class CRMFormAuthenticationFilter extends FormAuthenticationFilter {
    protected boolean onLoginFailure(AuthenticationToken token,
AuthenticationException e, ServletRequest request, ServletResponse
response) {
        String msg = "";
        if (e instanceof UnknownAccountException) {
            msg = "账号错误";
        } else if (e instanceof IncorrectCredentialsException) {
            msg = "密码错误";
        } else {
            msg = "未知错误";
        }
        e.printStackTrace();
        response.setContentType("text/json;charset=UTF-8");
        try {
            JSONResult result = new JSONResult();
            result.mark(msg);
            response.getWriter().print(JSON.toJSONString(result));
        } catch (Exception e1) {
            e1.printStackTrace();
        }
        return false;
    }

    protected boolean onLoginSuccess(AuthenticationToken token, Subject
subject, ServletRequest request, ServletResponse response) throws Exception
{
        response.setContentType("text/json;charset=UTF-8");
        response.getWriter().print(JSON.toJSONString(new JSONResult()));
        return false;
```

```
    }
}
```

然后将该过滤器设置为当前的认证过滤器

```
<bean id="shiroFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="/login.html"/>
    <property name="filterChainDefinitions">
        <value>
            /js/**=anon
            /images/**=anon
            /css/**=anon
            /**=authc
        </value>
    </property>
    <property name="filters">
        <map>
            <!--设置当前使用的认证过滤器-->
            <entry key="authc" value-ref="crmFormAuthenticationFilter"/>
        </map>
    </property>
</bean>
```

有了这个过滤器之后，我们在访问需要登录认证的资源时，完成登录认证后执行上面过滤器中的方法将认证结果返回给客户端。

注意：此时登录请求的url是/login.html，相信大家对这存在疑问。

以前我们的登录请求都是请求一个Controller，在Controller中完成登录的业务逻辑控制，现在怎么是一个html页面呢？而且在这个html中也没登录的业务逻辑处理啊。

结果分析如下：

1. 每个应用中的登录业务逻辑基本一致，所以Shiro将其做了抽取封装
2. Shiro是根据请求login.html的方式来判断是访问登录页面，还是做登录验证的请求

   get请求是访问登录页面，post请求是做登录验证

```
protected boolean onAccessDenied(
    ServletRequest request, ServletResponse response) throws Exception {
    if (isLoginRequest(request, response)) {
        if (isLoginSubmission(request, response)) {//post请求
            if (log.isTraceEnabled()) {
                log.trace(
                    "Login submission detected.  Attempting to execute
login.");
            }
            return executeLogin(request, response);
        } else {//get请求
```

```
            if (log.isTraceEnabled()) {
                log.trace("Login page view.");
            }
            //allow them to see the login page ;)
            return true;
        }
    } else {
        if (log.isTraceEnabled()) {
            log.trace(
                "Attempting to access a path which requires authentication.
Forwarding to the "
                + "Authentication url [" + getLoginUrl() + "]");
        }
        saveRequestAndRedirectToLogin(request, response);
        return false;
    }
}
```

到此，Shiro的认证功能已经成功的继承到了CRM项目中。

目前可以很方便的完成登录验证，登录拦截，注销等功能。

Shiro授权概述

系统中的授权功能就是为用户分配相关的权限，只有当用户拥有相应的权限后，才能访问对应的资源。

如果系统中无法管理用户的权限，那么将会出现非常客户信息泄露，数据没恶意篡改等问题，所以在绝大多数的应用中，我们都会有权限管理模块。

前面介绍过我们的权限管理系统是基于角色的权限管理，所以在系统中应该需要下面三个子模块：

1. 用户管理 2. 角色管理 3. 权限管理

这三个模块我们已经完成，同时可以很好的完成他们之间的关系管理。

那么目前我们所需要的就是将用户拥有的权限告知Shiro，供其在权限校验的时候使用。

Shiro授权方式分为三种：

1. 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
}
```

2. 注解式 通过在执行的Java方法上放置相应的注解完成

```
@RequiresRoles("admin")
@RequiresPermissions("user:create")
public void hello() {
    //有权限
}
```

3.JSP标签(shiro自带)或freemarker的标签(第三方) 在JSP页面通过相应的标签完成

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

基于ini的授权

在学习授权之前，我们需要先对权限表达式的格式做一个了解。

权限表达式的作用主要是用来在权限校验的时候使用，表达式中包含有当前访问资源的相关信息，应该具有唯一性。

权限表达式 在ini文件中用户、角色、权限的配置规则是：“用户名=密码，角色1，角色2...”“角色=权限1，权限2...”，首先根据用户名找角色，再根据角色找权限，角色是权限集合。

权限字符串的规则是：“资源标识符：操作：资源实例标识符”，意思是对哪个资源的哪个实例具有什么操作，“:”是资源/操作/实例的分割符，权限字符串也可以使用*通配符。

例子： 用户创建权限：user:create，或user:create:* 用户修改实例001的权限：user:update:001 用户实例001的所有权限：user：*：001

步骤：

1. 在ini文件中进行用户角色权限的关系配置

```
[users]
zhangsan=555,role1,role2
lisi=666,role2

[roles]
role1=user:list,user:delete
role2=user:update
```

2. 执行测试代码，验证当前登录的用户是否拥有指定的权限或者角色

```
public void testShiro() throws Exception{
    IniSecurityManagerFactory factory = new
IniSecurityManagerFactory("classpath:shiro.ini");
    SecurityManager manager = factory.getInstance();
```

```
        SecurityUtils.setSecurityManager(manager);
        Subject subject = SecurityUtils.getSubject();
        System.out.println("登录前的认证状态：+subject.isAuthenticated());//false
        UsernamePasswordToken token = new UsernamePasswordToken("zhangsan",
    "555");
        try {
            //登录认证
            subject.login(token);
            System.out.println("认证成功");
        } catch (UnknownAccountException e) {
            e.printStackTrace();
            System.out.println("用户名错误");
        } catch (IncorrectCredentialsException e){
            e.printStackTrace();
            System.out.println("密码错误");
        }
        System.out.println("登录后的认证状态：+subject.isAuthenticated());//true
        System.out.println(subject.isPermitted("user:delete"));
        System.out.println(Arrays.asList(subject.isPermitted("user:list",
    "user:update")[1]));
        System.out.println(subject.hasRole("role2"));
        System.out.println(subject.hasRoles(Arrays.asList("role1", "role2")));
        System.out.println(subject.hasAllRoles(Arrays.asList("role1",
    "role2")));
    }
```

首先保证登录成功，然后再观察权限和角色相关的执行结果

isPermitted(String permission):是否拥有当前指定的一个权限

isPermitted(String permission,...):是否拥有当前指定的一个或者多个权限，以数组形式返回每个判断的结果

hasRole(String role):是否拥有当前指定的一个角色

hasRoles(List roles):是否拥有当前指定的多个角色，以数组形式返回每个判断的结果

hasAllRoles(Collection roles):是否拥有指定的多个角色，只有当都有的时候才返回true，反之返回false

基于Realm的授权

和Realm的认证流程一致：

1. 自定义Realm
2. 在自定义的Realm中实现授权操作
3. 在ini文件中指定当前使用的Realm

```
public class CRMRealm extends AuthorizingRealm {
    @Autowired
    private EmployeeMapper mapper;
    //获取认证信息
```

```
    protected AuthenticationInfo doGetAuthenticationInfo(
            AuthenticationToken token) throws AuthenticationException {
        /*String username = token.getPrincipal().toString();
        Employee emp = mapper.selectEmployeeByUsername(username);
        if(emp == null){
            return null;
        }*/
        return new SimpleAuthenticationInfo("zhangsan","555",this.getName());
    }
    //获取授权信息
    protected AuthorizationInfo doGetAuthorizationInfo(
            PrincipalCollection principals) {
        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
        List<String> roles = new ArrayList<>();
        roles.add("role1");
        roles.add("role2");
        List<String> permissions = new ArrayList<>();
        permissions.add("user:list");
        permissions.add("user:delete");
        permissions.add("user:update");
        info.addRoles(roles);
        info.addStringPermissions(permissions);
        return info;
    }
}
```

在doGetAuthorizationInfo()方法中处理授权的业务逻辑，上面通过假数据测试授权的基本流程。

基于Shiro的权限加载

在权限管理模块中，我们采用自定义权限注解贴在需要的方法上，然后再扫描对应类的方法，获取对应的注解生成权限表达式。

其中的注解是我们自定义的，很明显，Shiro权限框架并不认识这个注解，在校验的是否自然也无法完成权限的校验功能，所以我们需要使用Shiro自身提供的一套注解来完成。

1. 在Controller的方法上贴上Shiro提供的权限注解（@RequiresPermissions）

```
@RequiresPermissions({"employee:list", "员工列表"},logical = Logical.OR)
@RequestMapping("list")
public String list(Model model, @ModelAttribute("qo") EmployeeQueryObject
qo) throws Exception {
    model.addAttribute("result", employeeService.query(qo));
    model.addAttribute("depts", departmentService.list());
    return "employee/list";
}
```

value属性：可以传递多个参数，利用这个特点，我们可以同时设置权限表达式和权限的名称

logical属性：Shiro会将value中的数据都作为权限表达式使用，校验的时候可以是都存在或者是拥有其中一个即可，对应的值分别为Logical.AND和Logical.OR

2. 开启Shiro注解扫描器，当扫描到Controller中有使用@RequiresPermissions注解时，会使用动态代理为当前Controller生成代理对象，增强对应方法的权限校验功能。

```
<bean class="org.apache.shiro.

spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
    <property name="securityManager" ref="securityManager"/>
</bean>
```

3. 扫描Controller类中的方法，生成权限信息到数据库中

```
public void reload() {
    //0:获取所有权限表达式准备排重
    List<String> expessions = permissionMapper.selectAllExpessions();
    //1: 获取所有的controller
    Map<String, Object> ctrlsMap =
ctx.getBeansWithAnnotation(Controller.class);
    Collection<Object> ctrls = ctrlsMap.values();
    //2: 遍历controller对象，获取controller中所有的带有RequiredPermission标签方
法
    for (Object ctrl : ctrls) {
        //3: 如果Controller方法上贴了RequiresPermissions注解，那么该Controller会
被代理
        //我们需要的是代理类的父类中对应的方法，所以这里需要特别注意
        Method[] methods =
ctrl.getClass().getSuperclass().getDeclaredMethods();
        for (Method method : methods) {
            if(method.isAnnotationPresent(RequiresPermissions.class)){
                RequiresPermissions ann = method
                    .getDeclaredAnnotation(RequiresPermissions.class);
                //注解中指定的权限表达式
                String expression = ann.value()[0];
                //排重
                if(expessions.contains(expression)){
                    continue; //结束本次循环
                }
                //注解中指定的权限名称
                String name = ann.value()[1];
                Permission p = new Permission();
                p.setName(name);
                p.setExpression(expression);
                //4: 保存对象
                permissionMapper.insert(p);
            }
        }
    }
}
```

不同之处：

1. 从Spring容器中获取到的Controller对象是代理对象，该对象的类继承自原始的Controller，而注解是贴在原始的Controller类方法上的，所以需要获取到Controller对象（代理对象）的父类（原始Controller）的字节码对象，再获取方法。
2. 权限表达式是直接在注解中指定的，不需要执行拼接操作，所以获取到注解value属性值（数组），在获取对应的表达式和名称即可。

web环境的Shiro授权

```
//获取授权信息
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
    SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
    //获取到当前登录的用户
    Employee employee = (Employee) principals.getPrimaryPrincipal();
    //如果是超级管理员，授予其admin的角色和所有权限
    if(employee.isAdmin()){
        info.addRole("admin");
        info.addStringPermission("*:*");
    }else{
        //根据登录用户的id查询到其拥有的所有角色的编码
        List<String> roleSns =
roleMapper.selectRoleSnsByEmpId(employee.getId());
        //根据登录用户的id查询到其拥有的所有权限表达式
        List<String> expressions = permissionMapper
            .selectExpressionsByEmpId(employee.getId());
        //将用户拥有的角色和权限添加到授权信息对象中，供Shiro权限校验时使用
        info.addRoles(roleSns);
        info.addStringPermissions(expressions);
    }
    return info;
}
```

注意：

: 表示所有权限，即不做任何限制

有了上面的授权控制，用户如果是超级管理员，可以访问系统中的任意资源；

如果用户不是超级管理员，只能访问分配给他的相关资源，如果访问了没有权限的资源，会抛出下面的异常。

org.apache.shiro.authz.UnauthorizedException:

Subject does not have permission [department:list]] with root cause

此时，我们只需要想办法捕获到该异常，然后进行处理即可。我们选择使用SpringMVC的统一异常处理。

```
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <!-- 定义默认的异常处理页面，当该异常类型的注册时使用 -->
    <property name="defaultErrorView" value="common/error"/>
    <!-- 定义异常处理页面用来获取异常信息的变量名，默认名为exception -->
    <property name="exceptionAttribute" value="ex"/>
    <!-- 定义需要特殊处理的异常，用类名或完全路径名作为key，异常也页名作为值 -->
    <property name="exceptionMappings">
        <!-- 这里还可以继续扩展不同异常类型的异常处理 -->
        <value>
            org.apache.shiro.authz.UnauthorizedException=common/nopermission
        </value>
    </property>
</bean>
```

在exceptionMappings中以键值对的格式设置异常需要跳转的错误页面。当系统抛出指定异常时，会跳转到等号后面指定的页面

异步请求没有权限异常处理

```
@ControllerAdvice
public class UnauthorizedExceptionUtil {
    @ExceptionHandler(UnauthorizedException.class)
    public void handler(HttpServletResponse response, HandlerMethod
method,UnauthorizedException e) throws IOException {
        if(method.getMethod().isAnnotationPresent(ResponseBody.class)){
            response.setContentType("text/json;charset=UTF-8");
            JSONResult result = new JSONResult();
            result.mark("对不起，您没有权限执行该操作");
            response.getWriter().print(JSON.toJSONString(result));
        }else{
            throw e;
        }
    }
```

```
}
```

@ControllerAdvice:Controller增强器

通常和@ExceptionHandler、@InitBinder、@ModelAttribute注解配合使用，其中最常用的是@ExceptionHandler，对Controller中的异常作特定处理。

@ExceptionHandler:异常处理器

贴在方法上，当Controller中配置指定异常时，会执行贴了该注解的方法。

异常方法的参数：

- 异常参数：包括一般的异常或特定的异常（即自定义异常），如果注解没有指定异常类，会默认进行映射。
- 请求或响应对象 (Servlet API or Portlet API)： 你可以选择不同的类型，

  如：ServletRequest/HttpServletRequest/ServletResponse/HttpServletResponse
- Session对象： HttpSession
- WebRequest或NativeWebRequest
- Locale
- InputStream/Reader
- OutputStream/Writer
- Model

Shiro标签

在前端页面上，我们通常可以根据用户拥有的权限来显示具体的页面，如：用户拥有删除员工的权限，页面上就把删除按钮显示出来，否则就不显示删除按钮，通过这种方式来细化权限控制。

要能够实现上面的控制，需要使用Shiro中提供的相关标签，标签的使用步骤如下：

1. 引入相关的jar包

```
<dependency>
    <groupId>net.mingsoft</groupId>
    <artifactId>shiro-freemarker-tags</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. 前端页面我们选择的是freemarker，而默认freemarker是不支持shiro标签的，所以需要对其功能做拓展

   可以理解为注册shiro的标签，达到在freemarker页面中使用的目的。

```
public class MyFreeMarkerConfig extends FreeMarkerConfigurer {
    @Override
    public void afterPropertiesSet() throws IOException, TemplateException
{
        //继承之前的属性配置，这不不能省
        super.afterPropertiesSet();
        Configuration cfg = this.getConfiguration();
        cfg.setSharedVariable("shiro", new ShiroTags());//shiro标签
    }
}
```

3. 在mvc.xml中将MyFreeMarkerConfig设置成当前环境中使用的配置对象

```
<bean class="cn.wolfcode.crm.util.MyFreeMarkerConfig">
    <!-- 配置freemarker的文件编码 -->
    <property name="defaultEncoding" value="UTF-8"/>
    <!-- 配置freemarker寻找模板的路径 -->
    <property name="templateLoaderPath" value="/WEB-INF/views/"/>
</bean>
```

有了上面的准备工作后，我们就可以在freemarker页面中使用shiro相关的标签来对页面显示做控制了。

shiro的freemarker常用标签：

1. authenticated标签：已认证通过的用户。`<@shiro.authenticated></@shiro.authenticated>`
2. notAuthenticated标签：未认证通过的用户。与authenticated标签相对。`<@shiro.notAuthenticated></@shiro.notAuthenticated>`
3. principal标签：
   输出当前用户信息，通常为登录帐号信息 `<@shiro.principal property="name" />`
4. hasRole标签：验证当前用户是否属于该角色，`<@shiro.hasRole name=”admin”>Hello admin!</@shiro.hasRole>`
5. hasAnyRoles标签：验证当前用户是否属于这些角色中的任何一个，角色之间逗号分隔，`<@shiro.hasAnyRoles name="admin,user,operator">Hello admin!</@shiro.hasAnyRoles>`
6. hasPermission标签：验证当前用户是否拥有该权限，`<@shiro.hasPermission name="/order:*">订单/@shiro.hasPermission>`

```
<@shiro.hasPermission name="employee:delete">
    <a href="#" class="btn btn-danger btn-xs btn-delete"
       data-href="/employee/delete.do?id=${entity.id}">
        <span class="glyphicon glyphicon-trash"></span> 删除
    </a>
</@shiro.hasPermission>
```

如果当前登录的用户没有员工删除的权限，就不在页面中显示删除超链接。

如果需要在页面中显示当前登录的用户，可以使用下面的代码实现

```
<@shiro.principal property="name"/>
```

后台是直接将整个员工对象作为身份信息的，所以这里可以直接访问他的name属性得到员工的姓名：

new SimpleAuthenticationInfo(**emp**,emp.getPassword(),this.getName());

MD5加密

加密的目的是从系统数据的安全考虑，如，用户的密码，如果我们不对其加密，那么所有用户的密码在数据库中都是明文，只要有权限查看数据库的都能够得知用户的密码，这是非常不安全的。所以，对用用户来说非常机密的数据，我们都应该想到使用加密技术，这里我们采用的是MD5加密。

在Shiro中继承了MD5的算法，所以可以直接使用它来实现对密码进行加密。

```
@Test
public void testMD5() throws Exception{
    Md5Hash hash = new Md5Hash("1");
    System.out.println(hash);//c4ca4238a0b923820dcc509a6f75849b
}
```

使用Md5Hash直接对数据进行加密

MD5加密的数据如果一样，那么无论在什么时候加密的结果都是一样的，所以，相对来说还是不够安全，但是别怕，我们可以对数据加“盐”。同样的数据加不同的“盐”之后就是千变万化的，因为我们不同的人加的“盐”都不一样。这样得到的结果相同率也就变低了。

```
@Test
public void testMD5() throws Exception{
    Md5Hash hash = new Md5Hash("1","admin");
    System.out.println(hash);//e00cf25ad42683b3df678c61f42c6bda
}
```

Md5Hash()构造方法中的第二个参数就是对加密数据添加的“盐”，加密之后的结果也和之前不一样了。

如果还觉得不够安全，我们还可以通过加密次数来增加MD5加密的安全性。

```
@Test
public void testMD5() throws Exception{
    Md5Hash hash = new Md5Hash("1","admin",3);
    System.out.println(hash);//f3559efea469bd6de83d27d4284b4a7a
}
```

上面指定对密码进行3次MD5加密，在开发中可以根据实际情况来指定加密次数。

CRM中使用MD5加密

在知道Shiro如何使用MD5加密之后，接下来我们来看看如何将其使用到我们的CRM项目中来。

1. 在添加用户的时候，需要对添加的用户密码进行加密

```
public void save(Employee entity, Long[] ids) {
    //密码加密,使用用户名作为加密的"盐"
    entity.setPassword(
        new Md5Hash(entity.getPassword(),entity.getName()).toString());
    employeeMapper.insert(entity);
    //维护与角色关系
    if(ids != null){
        for (Long id : ids) {
            employeeMapper.insertRelation(entity.getId(), id);
        }
    }
}
```

2. 登录认证过程中，密码匹配需要用到的密码应该和添加用户时的加密规则一致。

   所以和上面一样，指定加密使用的盐为当前用户的用户名。

```
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
    String username = token.getPrincipal().toString();
    Employee emp = mapper.selectEmployeeByUsername(username);
    if(emp == null){
        return null;
    }
    return new SimpleAuthenticationInfo(
        emp,//身份信息
        emp.getPassword(),//凭证
        ByteSource.Util.bytes(emp.getName()),//盐
        this.getName()//Realm名称);
}
```

3. 最后，最重要的是告诉Shiro我们选择使用哪一种加密算法

```
<!--指定当前需要使用的凭证匹配器-->
<bean class="org.apache.shiro.authc.credential.HashedCredentialsMatcher">
    <!--指定加密算法-->
    <property name="hashAlgorithmName" value="MD5"/>
</bean>
```

4. 将使用的凭证匹配器设置到当前使用的Realm中

```
@Component("crmRealm")
public class CRMRealm extends AuthorizingRealm {
    @Autowired
```

```
    private EmployeeMapper mapper;
    @Autowired
    private PermissionMapper permissionMapper;
    @Autowired
    private RoleMapper roleMapper;
    //将容器中的配置的凭证匹配器注入给当前的Realm对象
    //在该Realm中使用指定的凭证匹配器来完成密码匹配的操作
    @Autowired
    public void setCredentialsMatcher(CredentialsMatcher
credentialsMatcher) {
        super.setCredentialsMatcher(credentialsMatcher);
    }
}
```

测试之前使用命令对数据库中的密码进行加密

```
-- 先将所有的密码设置为1
update employee set password = 1;
-- 使用MD5函数对密码进行加密，其中name作为盐使用
update employee set password = MD5(concat(name,password));
```

有了以上操作之后，其实就两个地方用到了MD5加密：

1. 添加用户的时候，对用户的密码加密
2. 登录时，按照指定的方式对密码加密然后再和数据库中的加密过的数据进行匹配

集成EhCache

在请求中一旦进行权限的控制,如:

```
@RequiresPermissions("employee:view")
<shiro:hasPermission name="employee:input">
```

都去到Realm中的doGetAuthorizationInfo方法进行授权,我们授权信息应该要从数据库中查询的。 如果每次授权都要去查询数据库就太频繁了,性能不好. 而且用户登陆后,授权信息一般很少变动,所有我们可以在第一次授权后就把这些授权信息存到缓存中,下一次就直接从缓存中获取,避免频繁访问数据库。

Shiro中没有实现自己的缓存机制，只提供了一个可以支持具体缓存实现（如：Hazelcast, Ehcache, OSCache, Terracotta, Coherence, GigaSpaces, JBossCache等）的抽象API接口，这样就允许Shiro用户根据自己的需求灵活地选择具体的CacheManager。这里我们选择使用EhCache。

实现步骤：

1. 添加依赖

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-ehcache</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache-core</artifactId>
        <version>2.6.8</version>
</dependency>
```

2. 配置缓存管理器并引用缓存管理器

```
<!-- 缓存管理器 -->
<bean id="cacheManager"
class="org.apache.shiro.cache.ehcache.EhCacheManager">
    <!-- 设置配置文件 -->
    <property name="cacheManagerConfigFile" value="classpath:shiro-
ehcache.xml"/>
</bean>

<bean id="securityManager"
      class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="crmRealm"/>
    <property name="cacheManager" ref="cacheManager"/>
</bean>
```

3. 添加ehcache配置文件：shiro-ehcache.xml

```
<ehcache>
    <defaultCache
            maxElementsInMemory="1000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

Ehcache配置属性说明：

name: 缓存名称。

maxElementsInMemory： 缓存最大个数。 **eternal**:对象是否永久有效，一但设置了，timeout将不起作用。 **timeToIdleSeconds**： 设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。 **timeToLiveSeconds**：设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。 **overflowToDisk**：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中。 diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓

存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。 maxElementsOnDisk：硬盘最大缓存个数。 diskPersistent：是否缓存虚拟机重启期数据 Whether the disk store persists between restarts of the Virtual Machine. The default value is false. diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒。 memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。 clearOnFlush：内存数量最大时是否清除。

配置结束，登录之后，多次访问需要权限控制的代码时，不再反复查询权限数据，而是直接使用缓存中的权限数据。

清空缓存

1. 如果用户正常退出，缓存自动清空。
2. 如果用户非正常退出，缓存自动清空。
3. 如果修改了用户的权限，而用户不退出系统，修改的权限无法立即生效。
4. 当用户权限修改后，用户再次登陆shiro会自动调用realm从数据库获取权限数据，如果在修改权限后想立即清除缓存则可以调用realm的clearCache方法清除缓存。

在realm中定义该方法:

```
public void clearCached() {
     PrincipalCollection principals =
SecurityUtils.getSubject().getPrincipals();
     super.clearCache(principals);
}
```

在角色或权限service中,delete或者update方法去调用realm的清除缓存方法.

总结：Shiro为我们做了哪些事情?

1. 检查用户是否通过身份认证
2. 身份认证业务逻辑的实现
3. 系统注销
4. 权限校验
5. 密码加密
6. 权限数据的缓存