shiro 源码分析（三）授权、认证、缓存的接口设计

前两篇文章主要说的是认证过程，这一篇来分析下授权的过程。还是开涛大神的案例（http://jinnianshilongnian.iteye.com/blog/2020017），如下：

Java 代码

```java
public class ShiroTest {

    @Test
    public void testHelloworld() {
        init();
        Subject subject=login("zhang","123");
        Assert.assertTrue(subject.hasRole("role1"));
        Assert.assertTrue(subject.hasRole("role2"));
        Assert.assertTrue(subject.hasRole("role3"));
    }

    private Subject login(String userName,String password){
         //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(userName,password);
        subject.login(token);
        return subject;
    }

    private void init(){
         //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);
    }
}
```

ini 配置文件如下：

Java 代码

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

从 subject.hasRole 开始入手，默认的 Subject 为 DelegatingSubject：

Java 代码

```
public boolean hasRole(String roleIdentifier) {
        return hasPrincipals() && securityManager.hasRole(getPrincipals(), roleIdentifier);
    }
```

首先就是该用户是否已登录，验证角色的地方在 securityManager 的 hasRole 方法中：

Java 代码

```
public boolean hasRole(PrincipalCollection principals, String roleIdentifier) {
        return this.authorizer.hasRole(principals, roleIdentifier);
    }
```

AuthorizingSecurityManager 实现了 Authorizer 接口，但是 AuthorizingSecurityManager 是通过内部 Authorizer 引用来完成具体的功能，默认采用的是 ModularRealmAuthorizer。如下：

Java 代码

```
public abstract class AuthorizingSecurityManager extends AuthenticatingSecurityManager {

    /**
     * The wrapped instance to which all of this <tt>SecurityManager</tt> authorization calls are delegated.
     */
```

```java
    private Authorizer authorizer;

    public AuthorizingSecurityManager() {
        super();
        this.authorizer = new ModularRealmAuthorizer();
    }
//略
}
```

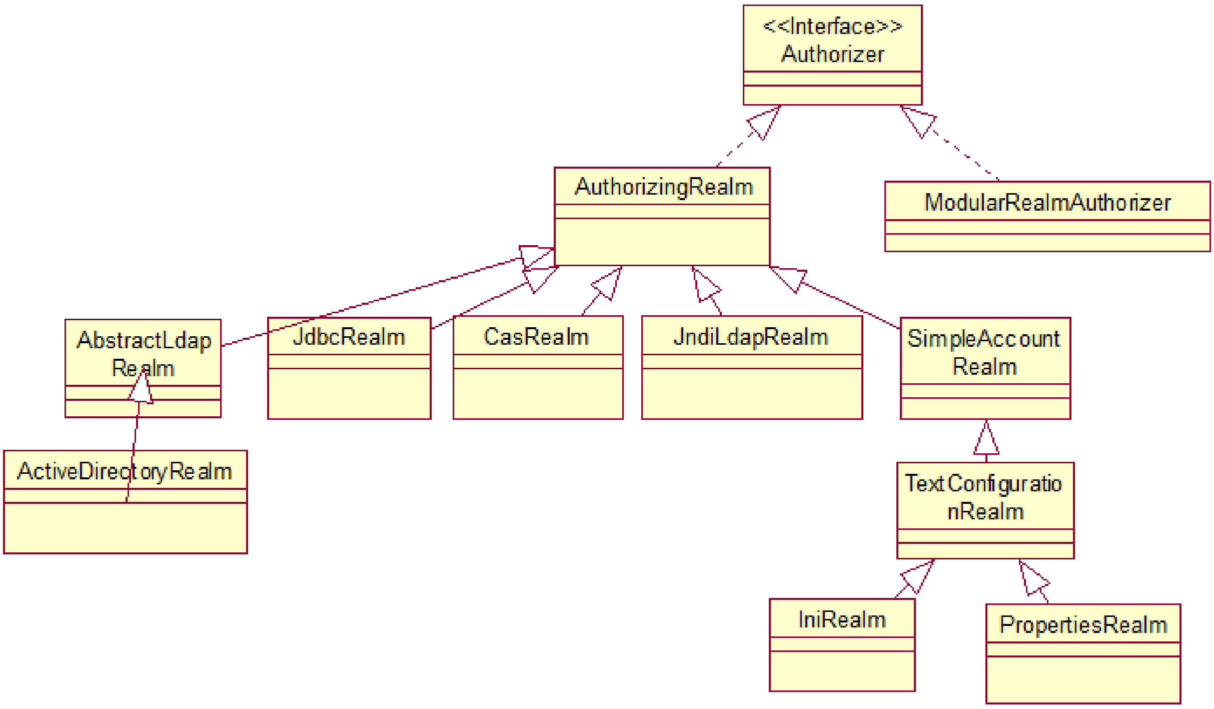
来看看这个 Authorizer 模块的接口设计：

Java 代码

```java
public interface Authorizer {
    boolean isPermitted(PrincipalCollection principals, String permission);
    boolean isPermitted(PrincipalCollection subjectPrincipal, Permission permission);
    boolean[] isPermitted(PrincipalCollection subjectPrincipal, String... permissions);
    boolean[] isPermitted(PrincipalCollection subjectPrincipal, List<Permission> permissions);
    boolean isPermittedAll(PrincipalCollection subjectPrincipal, String... permissions);
    boolean isPermittedAll(PrincipalCollection subjectPrincipal, Collection<Permission> permissions);

    void checkPermission(PrincipalCollection subjectPrincipal, String permission) throws AuthorizationException;
    void checkPermission(PrincipalCollection subjectPrincipal, Permission permission) throws AuthorizationException;
    void checkPermissions(PrincipalCollection subjectPrincipal, String... permissions) throws AuthorizationException;
    void checkPermissions(PrincipalCollection subjectPrincipal, Collection<Permission> permissions) throws AuthorizationException;

    boolean hasRole(PrincipalCollection subjectPrincipal, String roleIdentifier);
    boolean[] hasRoles(PrincipalCollection subjectPrincipal, List<String> roleIdentifiers);
    boolean hasAllRoles(PrincipalCollection subjectPrincipal, Collection<String> roleIdentifiers);

    void checkRole(PrincipalCollection subjectPrincipal, String roleIdentifier) throws AuthorizationException;
    void checkRoles(PrincipalCollection subjectPrincipal, Collection<String> roleIdentifiers) throws AuthorizationException;
    void checkRoles(PrincipalCollection subjectPrincipal, String... roleIdentifiers) throws AuthorizationException;

}
```

从上面的接口中，可以分成两大类，第一类是验证用户的某个或某些权限，第二类是验证用户的某个角色或某些角色。角色则是一组权限的集合，所以后者是粗粒度的验证，而前者是细粒度的验证。对于那些 check 方法则是验证不通过时抛出异常。

接口实现类图为：



可以看到很多的 Realm 都实现了该接口，即这些 Realm 不仅提供登陆验证，还提供权限验证。

先来看下默认使用的 ModularRealmAuthorizer：

Java 代码

```java
public class ModularRealmAuthorizer implements Authorizer, PermissionResolverAware, RolePermissionResolverAware {
    protected Collection<Realm> realms;

    protected PermissionResolver permissionResolver;

    protected RolePermissionResolver rolePermissionResolver;
    //略
}
```

可以看到，它有三个重要属性，Realm 集合和 PermissionResolver 、RolePermissionResolver 。

PermissionResolver 是什么呢？

Java 代码

```java
public interface PermissionResolver {
    Permission resolvePermission(String permissionString);
}
```

就是将权限字符串解析成 Permission 对象，同理 RolePermissionResolver 如下：

Java 代码

```java
public interface RolePermissionResolver {
    Collection<Permission> resolvePermissionsInRole(String roleString);
}
```

将角色字符串解析成 Permission 集合。

来看下这几个方法：

Java 代码

```java
public ModularRealmAuthorizer(Collection<Realm> realms) {
        setRealms(realms);
    }
public void setRealms(Collection<Realm> realms) {
        this.realms = realms;
        applyPermissionResolverToRealms();
        applyRolePermissionResolverToRealms();
    }
public void setPermissionResolver(PermissionResolver permissionResolver) {
        this.permissionResolver = permissionResolver;
        applyPermissionResolverToRealms();
    }
public void setRolePermissionResolver(RolePermissionResolver rolePermissionResolver) {
        this.rolePermissionResolver = rolePermissionResolver;
        applyRolePermissionResolverToRealms();
    }
protected void applyRolePermissionResolverToRealms() {
        RolePermissionResolver resolver = getRolePermissionResolver();
        Collection<Realm> realms = getRealms();
        if (resolver != null && realms != null && !realms.isEmpty()) {
            for (Realm realm : realms) {
                if (realm instanceof RolePermissionResolverAware) {
                    ((RolePermissionResolverAware) realm).setRolePermissionResolver(resolver);
                }
            }
        }
    }
protected void applyPermissionResolverToRealms() {
        PermissionResolver resolver = getPermissionResolver();
        Collection<Realm> realms = getRealms();
        if (resolver != null && realms != null && !realms.isEmpty()) {
            for (Realm realm : realms) {
                if (realm instanceof PermissionResolverAware) {
                    ((PermissionResolverAware) realm).setPermissionResolver(resolver);
                }
            }
        }
    }
```

看下这几个 set 方法，其目的都是如果哪些 Realm 想要 PermissionResolver 或 RolePermissionResolver 参数，则将 ModularRealmAuthorizer 的对应参数传给它。

再来看 ModularRealmAuthorizer 是如何实现 Authorizer 接口的：

Java 代码

```java
protected void assertRealmsConfigured() throws IllegalStateException {
        Collection<Realm> realms = getRealms();
        if (realms == null || realms.isEmpty()) {
            String msg = "Configuration error:  No realms have been configured!  One or more realms must be " +
                    "present to execute an authorization operation.";
            throw new IllegalStateException(msg);
        }
    }
public boolean isPermitted(PrincipalCollection principals, String permission) {
        assertRealmsConfigured();
        for (Realm realm : getRealms()) {
            if (!(realm instanceof Authorizer)) continue;
            if (((Authorizer) realm).isPermitted(principals, permission)) {
                return true;
            }
        }
        return false;
    }
```

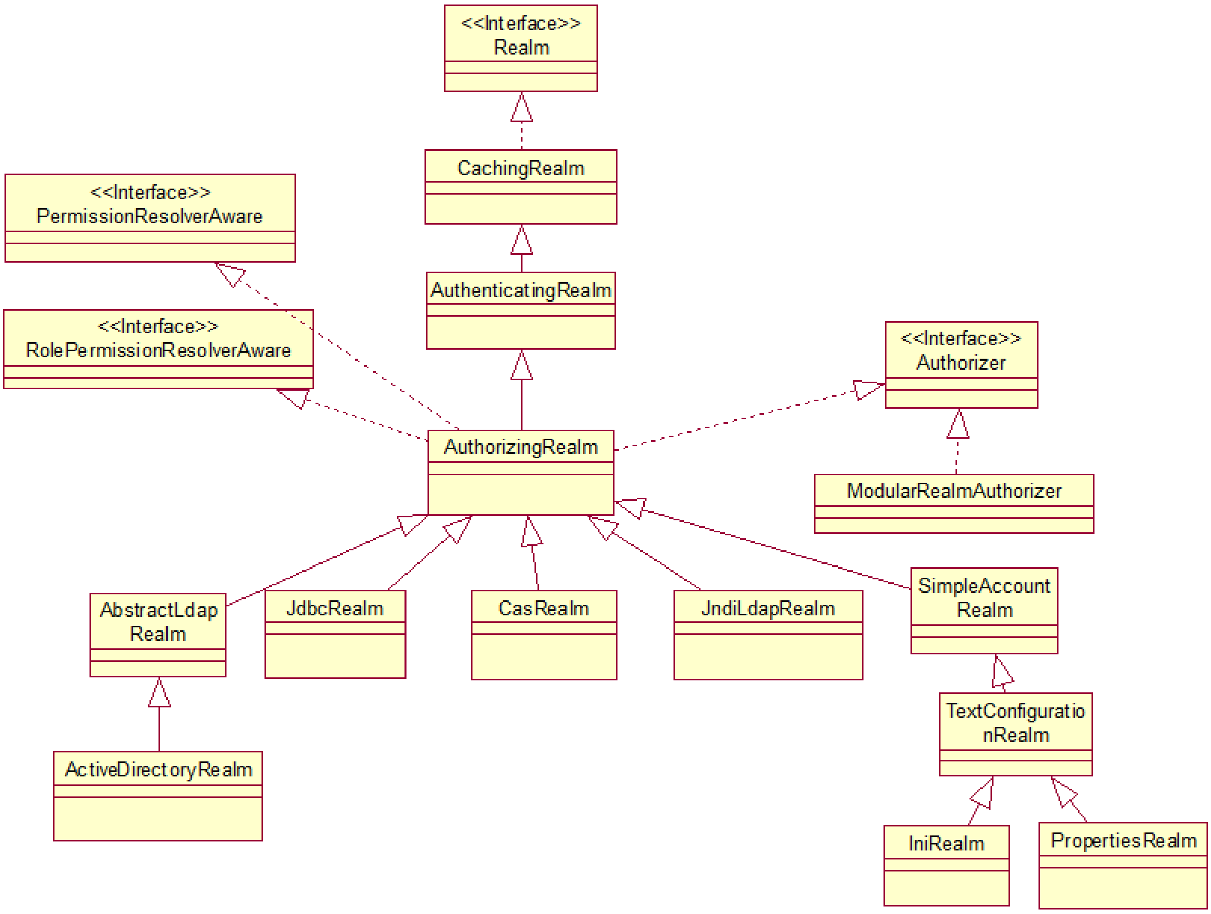
首先是判断 Collection<Realm> realms 集合是否为空，然后就是将那些实现了 Authorizer 接口的 Realm 来判断是否具有某个权限，也就是 ModularRealmAuthorizer 本身并不去权限验证，而是交给那些具有权限验证功能的 Realm 去验证（即那些 Realm 实现了 Authorizer 接口）。所以

ModularRealmAuthorizer 并不具有太多实际内容，我们转战那些实现了 Authorizer 接口的 Realm，去看看他们的验证过程。

这时候，就需要看 Authorizer 接口的另一个分支即下图

AuthorizingRealm 分支：

能：



AuthorizingRealm 涉及到 Realm，所以再把 Realm 说清楚。Realm 接口如下：

Java 代码 ☆

```java
public interface Realm {
    String getName();
    boolean supports(AuthenticationToken token);
    AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException;
}
```

Realm 本身只具有验证用户是否合法的功能，不具有授权的功能。再看它的实现者 CachingRealm，从名字上就可以知道加入了缓存功能：

Java 代码

```java
private static final AtomicInteger INSTANCE_COUNT = new AtomicInteger();

    private String name;
    private boolean cachingEnabled;
    private CacheManager cacheManager;
public CachingRealm() {
        this.cachingEnabled = true;
        this.name = getClass().getName() + "_" + INSTANCE_COUNT.getAndIncrement();
    }
```

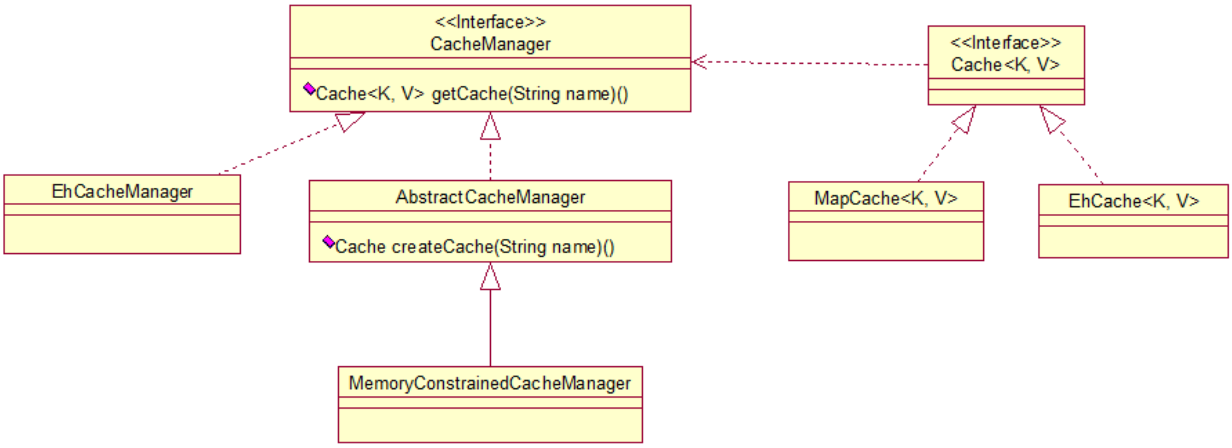
有 3 个对象属性和一个类属性，INSTANCE_COUNT 主要是用来计数 Realm 的个数的，同时追加到 name 属性中，cachingEnabled 对外提供 get、set 方法，这里的 cachingEnabled 就相当于一个总开关，它的子类都有子开关，共同决定着是否进行缓存，如它的子类 AuthenticatingRealm：

Java 代码

```java
private boolean authenticationCachingEnabled;
public boolean isAuthenticationCachingEnabled() {
        return this.authenticationCachingEnabled && isCachingEnabled();
    }
 public void setAuthenticationCachingEnabled(boolean authenticationCachingEnabled) {
        this.authenticationCachingEnabled = authenticationCachingEnabled;
        if (authenticationCachingEnabled) {
            setCachingEnabled(true);
        }
    }
```

从这里就可以看到两个 cacheEnabled 的作用。也对外提供 CacheManager 的 get、set 方法， CachingRealm 本身并没有做太多内容，就是把这几个参数收集起来，供子类去使用。

接下来看下 Cache 缓存的整体结构图：



我们要先看下 CacheManager 是干嘛的：

Java 代码

```java
public interface CacheManager {
  public <K, V> Cache<K, V> getCache(String name) throws CacheException;
}
```

根据 name 获取一个 Cache<K, V>这样的结构，看起来像 HashMap 的结构，这里的 name 到底是什么呢？

CachingRealm 的子类 AuthenticatingRealm 有一个 authenticationCacheName 属性，而这里的 authenticationCacheName 就是我们刚才要找的目标，证据如下：

Java 代码

```java
private Cache<Object, AuthenticationInfo> getAuthenticationCacheLazy() {

    if (this.authenticationCache == null) {

      log.trace("No authenticationCache instance set.  Checking for a cacheManager...");

      CacheManager cacheManager = getCacheManager();

      if (cacheManager != null) {
   //这里的 getAuthenticationCacheName()就是获取 authenticationCacheName
        String cacheName = getAuthenticationCacheName();
```

```java
                log.debug("CacheManager [{}] configured.  Building authentication cache '{}'", cacheManager, cacheName);
                this.authenticationCache = cacheManager.getCache(cacheName);
            }
        }

        return this.authenticationCache;
    }
```

再看下 authenticationCacheName 的构成：

Java 代码

```java
public AuthenticatingRealm(CacheManager cacheManager, CredentialsMatcher matcher) {
        authenticationTokenClass = UsernamePasswordToken.class;

        //retain backwards compatibility for Shiro 1.1 and earlier.  Setting to true by default will probably cause
        //unexpected results for existing applications:
        this.authenticationCachingEnabled = false;

        int instanceNumber = INSTANCE_COUNT.getAndIncrement();
        this.authenticationCacheName = getClass().getName() + DEFAULT_AUTHORIZATION_CACHE_SUFFIX;
        if (instanceNumber > 0) {
            this.authenticationCacheName = this.authenticationCacheName + "." + instanceNumber;
        }

        if (cacheManager != null) {
            setCacheManager(cacheManager);
        }
        if (matcher != null) {
            setCredentialsMatcher(matcher);
        }
    }
```

在创建 AuthenticatingRealm 时，authenticationCacheName 默认是当前类名+DEFAULT_AUTHORIZATION_CACHE_SUFFIX（为.authenticationCache）+数量。这个数量也是用来统计

AuthenticatingRealm 的个数的，这种方式仅仅是默认的，也可以去修改：

Java 代码

```java
public void setAuthenticationCacheName(String authenticationCacheName) {
        this.authenticationCacheName = authenticationCacheName;
    }
public void setName(String name) {
        super.setName(name);
        String authcCacheName = this.authenticationCacheName;
        if (authcCacheName != null && authcCacheName.startsWith(getClass().getName())) {
            //get rid of the default heuristically-created cache name.  Create a more meaningful one
            //based on the application-unique Realm name:
            this.authenticationCacheName = name + DEFAULT_AUTHORIZATION_CACHE_SUFFIX;
        }
    }
```

这两种方式都可以去修改。回到 CacheManager：

Java 代码

```java
public interface CacheManager {
    public <K, V> Cache<K, V> getCache(String name) throws CacheException;
}
```

然后就需要了解下 Cache<K, V>这个结构：

Java 代码

```java
public interface Cache<K, V> {
    public V get(K key) throws CacheException;
    public V put(K key, V value) throws CacheException;
    public V remove(K key) throws CacheException;
    public void clear() throws CacheException;
    public int size();
    public Set<K> keys();
    public Collection<V> values();
}
```

这基本上不就是 map 的结构吗？为什么还要单独设计这样的结构呢？来看下它的文档介绍就知道了：

Java 代码

```java
/**
 * A Cache efficiently stores temporary objects primarily to improve an application's performance.
 *
 * <p>Shiro doesn't implement a full Cache mechanism itself, since that is outside the core competency of a
 * Security framework.  Instead, this interface provides an abstraction (wrapper) API on top of an underlying
 * cache framework's cache instance (e.g. JCache, Ehcache, JCS, OSCache, JBossCache, TerraCotta, Coherence,
 * GigaSpaces, etc, etc), allowing a Shiro user to configure any cache mechanism they choose.
 *
 * @since 0.2
 */
```

Shiro 并不打算自己实现一个完整的缓存机制，因为这并不是安全框架的主要职责，相反它应该提供一个统一的 API 接口,可以加入不同缓存框架。而对于我们用户来说，只需针对这一层统一 API 进行编程，不再针对某个具体的缓存框架编程，这样就更加容易切换不同的缓存框架。

再看下，它的实现类 MapCache 和 EhCache，MapCache 很简单就是通过 Map 结构来实现

Java 代码

```java
public class MapCache<K, V> implements Cache<K, V> {
    private final Map<K, V> map;
    private final String name;

    public MapCache(String name, Map<K, V> backingMap) {
        if (name == null) {
            throw new IllegalArgumentException("Cache name cannot be null.");
        }
        if (backingMap == null) {
            throw new IllegalArgumentException("Backing map cannot be null.");
        }
        this.name = name;
        this.map = backingMap;
    }

    public V get(K key) throws CacheException {
        return map.get(key);
    }

    public V put(K key, V value) throws CacheException {
        return map.put(key, value);
    }

    public V remove(K key) throws CacheException {
        return map.remove(key);
    }
    //略
}
```

EhCache 则是通过 net.sf.ehcache.Ehcache 框架来来实现，不再涉及。

Cache<K, V>知道了，又有哪些 CacheManager 的实现呢？

AbstractCacheManager 如下：

Java 代码

```java
public abstract class AbstractCacheManager implements CacheManager, Destroyable {
    private final ConcurrentMap<String, Cache> caches;
    public AbstractCacheManager() {
        this.caches = new ConcurrentHashMap<String, Cache>();
    }

    public <K, V> Cache<K, V> getCache(String name) throws IllegalArgumentException, CacheException {
        if (!StringUtils.hasText(name)) {
            throw new IllegalArgumentException("Cache name cannot be null or empty.");
        }
        Cache cache;
        cache = caches.get(name);
        if (cache == null) {
            cache = createCache(name);
            Cache existing = caches.putIfAbsent(name, cache);
            if (existing != null) {
                cache = existing;
            }
        }
        return cache;
    }
    //略
}
```

也很简单，内部拥有一个 ConcurrentHashMap 集合，存取都是对该集合的操作，而把真正创建 Cache 的操作留给具体的子类来实现，即 createCache 方法。看下它的子类 MemoryConstrainedCacheManager 的 createCache 实现：

Java 代码

```java
public class MemoryConstrainedCacheManager extends AbstractCacheManager {
    @Override
    protected Cache createCache(String name) {
        return new MapCache<Object, Object>(name, new SoftHashMap<Object, Object>());
    }
}
```

就是创建了一个 MapCache 对象作为 Cache，至于 SoftHashMap 则需要单独去介绍其中的设计。

CachingRealm 就大致介绍完了，回到它的子类，看它的子类 AuthenticatingRealm 是怎么去使用 CacheManager。该子类主要完成

认证流程，首先是其的初始化，AuthenticatingRealm 及其子类都实现了 Initializable 接口，初始化的时候会首先获取其缓存，如下：

Java 代码

```java
public final void init() {
        //trigger obtaining the authorization cache if possible
        getAvailableAuthenticationCache();
        onInit();
    }
private Cache<Object, AuthenticationInfo> getAvailableAuthenticationCache() {
        Cache<Object, AuthenticationInfo> cache = getAuthenticationCache();
        boolean authcCachingEnabled = isAuthenticationCachingEnabled();
        if (cache == null && authcCachingEnabled) {
            cache = getAuthenticationCacheLazy();
        }
        return cache;
    }
private Cache<Object, AuthenticationInfo> getAuthenticationCacheLazy() {

        if (this.authenticationCache == null) {

            log.trace("No authenticationCache instance set.  Checking for a cacheManager...");

            CacheManager cacheManager = getCacheManager();

            if (cacheManager != null) {
                String cacheName = getAuthenticationCacheName();
                log.debug("CacheManager [{}] configured.  Building authentication cache '{}'", cacheManager, cacheName);
                this.authenticationCache = cacheManager.getCache(cacheName);
            }
        }

        return this.authenticationCache;
    }
```

首先会获取 Cache<Object, AuthenticationInfo> cache 属性，如果没

有，再判断是否允许缓存，如果允许，则通过 CacheManager 来获取，之前已分析过，如果还没有则会创建一个 Cache，然后返回。

再看下认证过程如下：

Java 代码

```java
public final AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {

        AuthenticationInfo info = getCachedAuthenticationInfo(token);
        if (info == null) {
            //otherwise not cached, perform the lookup:
            info = doGetAuthenticationInfo(token);
            log.debug("Looked up AuthenticationInfo [{}] from doGetAuthenticationInfo", info);
            if (token != null && info != null) {
                cacheAuthenticationInfoIfPossible(token, info);
            }
        } else {
            log.debug("Using cached authentication info [{}] to perform credentials matching.", info);
        }

        if (info != null) {
            assertCredentialsMatch(token, info);
        } else {
            log.debug("No AuthenticationInfo found for submitted AuthenticationToken [{}].  Returning null.", token);
        }

        return info;
    }
```

首先从缓存中尝试是否能找到 AuthenticationInfo ，如果找不到，则需要子类去完成具体的认证细节，然后再存储到缓存中，因为本类并没有具体的数据源，只有缓存源，所以本类只是搭建了认证流程，具体的认证细节则由具体的子类来完成，所以 doGetAuthenticationInfo(token)是一个 protected 的抽象方法，如下：

Java 代码

```java
protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException;
```

当缓存中存在或者子类进行具体的认证后，下一步的操作是要进行密码匹配的过程，AuthenticatingRealm 有一个属性 CredentialsMatcher credentialsMatcher，接口如下：

Java 代码

```java
public interface CredentialsMatcher {
    boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info);
}
```

就是匹配我们认证时的 AuthenticationToken 和刚才已找到的 AuthenticationInfo 是否匹配。有如下的实现类：AllowAllCredentialsMatcher、PasswordMatcher、SimpleCredentialsMatcher 等等。AuthenticatingRealm 的构造函数默认使用的是 SimpleCredentialsMatcher：

Java 代码

```java
public AuthenticatingRealm() {
        this(null, new SimpleCredentialsMatcher());
    }

    public AuthenticatingRealm(CacheManager cacheManager) {
        this(cacheManager, new SimpleCredentialsMatcher());
    }

    public AuthenticatingRealm(CredentialsMatcher matcher) {
        this(null, matcher);
    }
```

这一块内容先暂时不讲，后续文章再来详细说明。

当你匹配通过了，则就算认证成功了。认证流程就在

AuthenticatingRealm 中完成了。

我们再向它的子类 AuthorizingRealm 研究，这个就有涉及到授权的功能了。AuthenticatingRealm 是将整个认证流程框架化，AuthorizingRealm 则是将整个授权流程框架化，AuthorizingRealm 也有授权缓存，所以会通过父类 CachingRealm 来获取 CacheManager，同时也有一个子缓存开关 authorizationCachingEnabled，和 AuthenticatingRealm 基本类似，属性如下：

Java 代码 ☆

```java
public abstract class AuthorizingRealm extends AuthenticatingRealm
        implements Authorizer, Initializable, PermissionResolverAware, RolePermissionResolverAware {

    private static final String DEFAULT_AUTHORIZATION_CACHE_SUFFIX = ".authorizationCache";

    private static final AtomicInteger INSTANCE_COUNT = new AtomicInteger();

    private boolean authorizationCachingEnabled;
    private Cache<Object, AuthorizationInfo> authorizationCache;
    private String authorizationCacheName;

    private PermissionResolver permissionResolver;

    private RolePermissionResolver permissionRoleResolver;

    public AuthorizingRealm() {
        this(null, null);
    }

    public AuthorizingRealm(CacheManager cacheManager) {
        this(cacheManager, null);
    }

    public AuthorizingRealm(CredentialsMatcher matcher) {
        this(null, matcher);
    }

    public AuthorizingRealm(CacheManager cacheManager, CredentialsMatcher matcher) {
        super();
        if (cacheManager != null) setCacheManager(cacheManager);
        if (matcher != null) setCredentialsMatcher(matcher);

        this.authorizationCachingEnabled = true;
        this.permissionResolver = new WildcardPermissionResolver();

        int instanceNumber = INSTANCE_COUNT.getAndIncrement();
        this.authorizationCacheName = getClass().getName() + DEFAULT_AUTHORIZATION_CACHE_SUFFIX;
        if (instanceNumber > 0) {
            this.authorizationCacheName = this.authorizationCacheName + "." + instanceNumber;
        }
    }
//略
}
```

AtomicInteger 同样是用于对那些具有授权功能的 Realm 进行数量统计的，authorizationCachingEnabled 缓存子开关，authorizationCache 缓存，authorizationCacheName 缓存名字。PermissionResolver permissionResolver、RolePermissionResolver permissionRoleResolver 这两个则是对字符串进行解析对应的 Permission 和 Collection<Permission>的。我们来看下 AuthorizingRealm 的主要功能，对于授权接口 Authorizer 的实现：

Java 代码

```java
public boolean hasRole(PrincipalCollection principal, String roleIdentifier) {
        AuthorizationInfo info = getAuthorizationInfo(principal);
        return hasRole(roleIdentifier, info);
    }
```

首先就是获取授权信息，看下 getAuthorizationInfo：

Java 代码

```java
protected AuthorizationInfo getAuthorizationInfo(PrincipalCollection principals) {

        if (principals == null) {
            return null;
        }

        AuthorizationInfo info = null;

        if (log.isTraceEnabled()) {
            log.trace("Retrieving AuthorizationInfo for principals [" + principals + "]");
        }

        Cache<Object, AuthorizationInfo> cache = getAvailableAuthorizationCache();
        if (cache != null) {
            if (log.isTraceEnabled()) {
                log.trace("Attempting to retrieve the AuthorizationInfo from cache.");
            }
            Object key = getAuthorizationCacheKey(principals);
            info = cache.get(key);
            if (log.isTraceEnabled()) {
                if (info == null) {
                    log.trace("No AuthorizationInfo found in cache for principals [" + principals + "]");
                } else {
                    log.trace("AuthorizationInfo found in cache for principals [" + principals + "]");
                }
            }
        }


        if (info == null) {
            // Call template method if the info was not found in a cache
            info = doGetAuthorizationInfo(principals);
            // If the info is not null and the cache has been created, then cache the authorization info.
            if (info != null && cache != null) {
                if (log.isTraceEnabled()) {
                    log.trace("Caching authorization info for principals: [" + principals + "].");
                }
                Object key = getAuthorizationCacheKey(principals);
                cache.put(key, info);
            }
        }

        return info;
    }
```

同样很容易理解，先得到缓存，从缓存中去找有没有授权信息，如果没有，则需要子类去完成具体的授权细节即 doGetAuthorizationInfo，授权完成后放置缓存中。同样 doGetAuthorizationInfo 是 protected 的抽象方法，由子类去实现。PermissionResolver permissionResolver、RolePermissionResolver permissionRoleResolver 则是发挥如下作用：

Java 代码

```
private Collection<Permission> getPermissions(AuthorizationInfo info) {
        Set<Permission> permissions = new HashSet<Permission>();

        if (info != null) {
            Collection<Permission> perms = info.getObjectPermissions();
            if (!CollectionUtils.isEmpty(perms)) {
                permissions.addAll(perms);
            }
            perms = resolvePermissions(info.getStringPermissions());
            if (!CollectionUtils.isEmpty(perms)) {
                permissions.addAll(perms);
            }

            perms = resolveRolePermissions(info.getRoles());
            if (!CollectionUtils.isEmpty(perms)) {
                permissions.addAll(perms);
            }
        }
```

```java

        if (permissions.isEmpty()) {
            return Collections.emptySet();
        } else {
            return Collections.unmodifiableSet(permissions);
        }
    }
```

即有了授权信息 AuthorizationInfo 后，获取所有的权限 Permission，有三种途径来收集，第一种就是 info.getObjectPermissions() info 中直接含有 Permission 对象集合，第二种就是 info.getStringPermissions() info 中有字符串形式的权限表示，第三种就是 info.getRoles() info 中含有角色集合，角色也是一组权限的集合，看下 resolvePermissions(info.getStringPermissions()):

Java 代码 ☆

```java
private Collection<Permission> resolvePermissions(Collection<String> stringPerms) {
        Collection<Permission> perms = Collections.emptySet();
        PermissionResolver resolver = getPermissionResolver();
        if (resolver != null && !CollectionUtils.isEmpty(stringPerms)) {
            perms = new LinkedHashSet<Permission>(stringPerms.size());
            for (String strPermission : stringPerms) {
                Permission permission = getPermissionResolver().resolvePermission(strPermission);
                perms.add(permission);
            }
        }
        return perms;
    }
```

也很简单，对于每一个 strPermission 通过 PermissionResolver 转化成 Permission 对象，对于 resolveRolePermissions 也同理，不再说明。这里具体的转化细节先暂且不说，后续再将。

现在终于把认证流程和授权框架流程大致说完了，即

AuthenticatingRealm 和 AuthorizingRealm 的内容，他们分别留给子类 protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException; 具体的认证方法和 protected abstract AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)具体的授权方法。

作者：乒乓狂魔