Shiro 源码分析（一）入门

最近闲来无事，准备读个框架源码，经别人推荐 shiro，那就准备读读其中的设计。开涛大神已经有了跟我学 Shiro 系列，那我就跟着这个系列入门然后再深入源代码，所以我的侧重点就是源码分析。

话不多说，上开涛大神的入门案例 地址

http://jinnianshilongnian.iteye.com/blog/2019547：

Java 代码

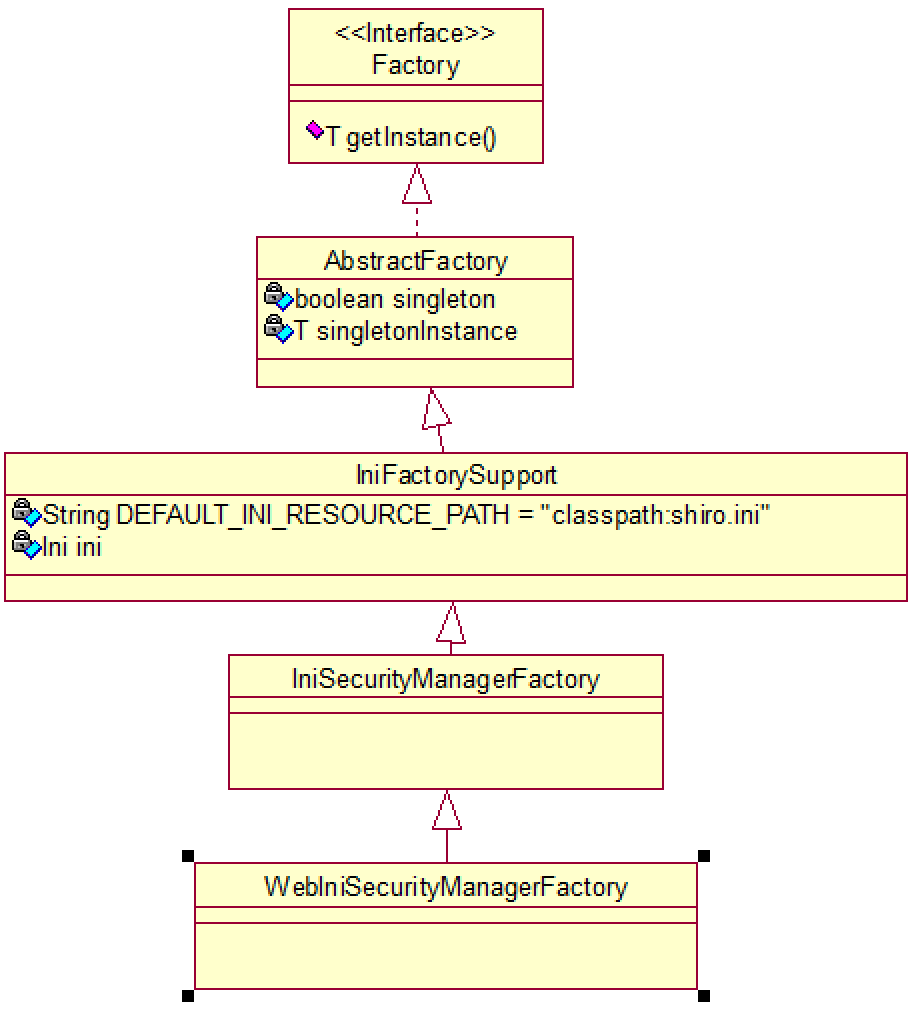
```java
@Test
    public void testHelloworld() {
        //1、获取 SecurityManager 工厂，此处使用 Ini 配置文件初始化 SecurityManager
        Factory<org.apache.shiro.mgt.SecurityManager> factory =
                new IniSecurityManagerFactory("classpath:shiro.ini");
        //2、得到 SecurityManager 实例 并绑定给 SecurityUtils
        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", "123232");

        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();
    }
```

1：使用工厂模式来得到 SecurityManager，由于可以通过不同工厂创建出不同的 SecurityManager，如通过配置文件的形式来创建的 IniSecurityManagerFactory 工厂。类图如下：



Factory 接口：通过泛型定义了一个 T getInstance()方法

AbstractFactory 抽象类：对于 getInstance 返回的对象加入单例或者非单例的功能，而把真正创建实例对象的 createInstance 功能留给子类去实现

Java 代码

```java
public T getInstance() {
        T instance;
        if (isSingleton()) {
            if (this.singletonInstance == null) {
                this.singletonInstance = createInstance();
            }
            instance = this.singletonInstance;
        } else {
            instance = createInstance();
        }
        if (instance == null) {
            String msg = "Factory 'createInstance' implementation returned a null object.";
            throw new IllegalStateException(msg);
        }
        return instance;
    }

    protected abstract T createInstance();
```

IniFactorySupport：加入了 Ini ini 属性，同过该对象来创建出一个实例，IniFactorySupport 对于 ini 的获取给出了两种方式，方式一：在构造 IniFactorySupport 时传入 Ini 对象，另一种就是加载类路径下默认的 Ini，如下：

Java 代码

```java
public static Ini loadDefaultClassPathIni() {
        Ini ini = null;
        if (ResourceUtils.resourceExists(DEFAULT_INI_RESOURCE_PATH)) {
            log.debug("Found shiro.ini at the root of the classpath.");
            ini = new Ini();
            ini.loadFromPath(DEFAULT_INI_RESOURCE_PATH);
            if (CollectionUtils.isEmpty(ini)) {
                log.warn("shiro.ini found at the root of the classpath, but it did not contain any data.");
            }
        }
        return ini;
    }
```

其中 DEFAULT_INI_RESOURCE_PATH 为 classpath:shiro.ini。然而 IniFactorySupport 并不负责通过 ini 配置文件来创建出什么样的对象，它仅仅负责获取 ini 配置文件，所以它要留出了两个方法让子类实现：

Java 代码

```java
protected abstract T createInstance(Ini ini);

protected abstract T createDefaultInstance();
```

第一个方法就是通过 ini 配置文件创建出什么对象，第二个方法就是当获取不到 ini 配置文件时，要创建默认的对象。

IniSecurityManagerFactory：通过 Ini 配置文件可以创建出 SecurityManager 对象，也可以通过 ini 配置文件创建 FilterChainResolver 对象，而 IniSecurityManagerFactory 则是通过 ini 配置文件来创建 SecurityManager 的，所以对于泛型的实例化是在该类完成的，如下：

Java 代码

```java
public class IniSecurityManagerFactory extends IniFactorySupport<SecurityManager>
public class IniFilterChainResolverFactory extends IniFactorySupport<FilterChainResolver>
```

IniSecurityManagerFactory 还不具有 web 功能，WebIniSecurityManagerFactory 则加入了 web 功能。

可以看到，有很多的类继承关系，每一个类都完成了一个基本功能，把职责划分的更加明确，而不是一锅粥把很多功能放到一个类中，导致很难去复用某些功能。

2 ：将创建的 SecurityManager 放到 SecurityUtils 类的静态变量中，供所有对象来访问。

3 ：创建一个 Subject 实例，接口 Subject 的文档介绍如下：

Java 代码

```
A {@code Subject} represents state and security operations for a <em>single</em> application user.These operations include authentication (login/logout), authorization (access control), and session access
```

及外界通过 Subject 接口来和 SecurityManager 进行交互，该接口含有登录、退出、权限判断、获取 session,其中的 Session 可不是平常我们所使用的 HttpSession 等，而是 shiro 自定义的，是一个数据上下文，与一个 Subject 相关联的。

先回到创建 Subject 的地方：

Java 代码

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

一看就是使用的是 ThreadLocal 设计模式，获取当前线程相关联的 Subject 对象，如果没有则创建一个，然后绑定到当前线程。然后我们来看下具体实现：

ThreadContext 是 org.apache.shiro.util 包下的一个工具类，它是用来操作和当前线程绑定的 SecurityManager 和 Subject，它必然包含了一个 ThreadLocal 对象如下：

Java 代码

```java
public abstract class ThreadContext {

    public static final String SECURITY_MANAGER_KEY = ThreadContext.class.getName() + "_SECURITY_MANAGER_KEY";
    public static final String SUBJECT_KEY = ThreadContext.class.getName() + "_SUBJECT_KEY";

    private static final ThreadLocal<Map<Object, Object>> resources = new InheritableThreadLocalMap<Map<Object, Object>>();

 //略

}
```

ThreadLocal 中所存放的数据是一个 Map 集合，集合中所存的 key 有两个 SECURITY_MANAGER_KEY 和 SUBJECT_KEY ，就是通过这两个 key 来存取 SecurityManager 和 Subject 两个对象的。具体的 ThreadLocal 设计模式分析可以详见我的另一篇博客 http://lgbolgger.iteye.com/blog/2117216。

当前线程还没有绑定一个 Subject 时，就需要通过 Subject.Builder 来创建一个然后绑定到当前线程。Builder 是 Subject 的一个内部类，它拥有两个重要的属性，SubjectContext 和 SecurityManager，创建 Builder 时使用 SecurityUtils 工具来获取它的全局静态变量 SecurityManager，SubjectContext 则是使用 newSubjectContextInstance 创建一个 DefaultSubjectContext 对象：

Java 代码

```java
public Builder() {
            this(SecurityUtils.getSecurityManager());
        }

        public Builder(SecurityManager securityManager) {
            if (securityManager == null) {
                throw new NullPointerException("SecurityManager method argument cannot be null.");
            }
            this.securityManager = securityManager;
            this.subjectContext = newSubjectContextInstance();
            if (this.subjectContext == null) {
                throw new IllegalStateException("Subject instance returned from 'newSubjectContextInstance' " +
                        "cannot be null.");
            }
            this.subjectContext.setSecurityManager(securityManager);
        }

protected SubjectContext newSubjectContextInstance() {
            return new DefaultSubjectContext();
        }
```

Builder 准备工作完成后，调用 buildSubject 来创建一个 Subject：

Java 代码

```java
public Subject buildSubject() {
            return this.securityManager.createSubject(this.subjectContext);
        }
```

最终还是通过 securityManager 根据 subjectContext 来创建一个 Subject。最终是通过一个 SubjectFactory 来创建的，SubjectFactory 是一个接口，接口方法为 Subject createSubject(SubjectContext context)，默认的 SubjectFactory 实现是 DefaultSubjectFactory，DefaultSubjectFactory 创建的 Subject 是 DelegatingSubject。至此创

建 Subject 就简单说完了。

4 继续看登陆部分

登陆方法为：void login(AuthenticationToken token)，

AuthenticationToken 接口如下：

Java 代码

```java
public interface AuthenticationToken extends Serializable {

    Object getPrincipal();

    Object getCredentials();

}
```

Principal 就相当于用户名，Credentials 就相当于密码，AuthenticationToken 的实现 UsernamePasswordToken 有四个重要属性，即 username、char[] password、boolean rememberMe、host。认证过程是由 Authenticator 来完成的，先来看下 Authenticator 的整体：

Java 代码

```java
public interface Authenticator {
    public AuthenticationInfo authenticate(AuthenticationToken authenticationToken)
            throws AuthenticationException;
}
```

很简单，就是根据 AuthenticationToken 返回一个 AuthenticationInfo ，如果认证失败会抛出 AuthenticationException 异常。

AbstractAuthenticator 实现了 Authenticator 接口，它仅仅加入了对认

证成功与失败的监听功能，即有一个

Collection<AuthenticationListener>集合：

Java 代码

```java
private Collection<AuthenticationListener> listeners;
```

对于认证过程：

Java 代码

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

        if (token == null) {
            throw new IllegalArgumentException("Method argumet (authentication token) cannot be null.");
        }

        log.trace("Authentication attempt received for token [{}]", token);

        AuthenticationInfo info;
        try {
            info = doAuthenticate(token);
            if (info == null) {
                String msg = "No account information found for authentication token [" + token + "] by this " +
                        "Authenticator instance.  Please check that it is configured correctly.";
                throw new AuthenticationException(msg);
            }
        } catch (Throwable t) {
            AuthenticationException ae = null;
            if (t instanceof AuthenticationException) {
                ae = (AuthenticationException) t;
            }
            if (ae == null) {
                //Exception thrown was not an expected AuthenticationException.  Therefore it is probably a little more
                //severe or unexpected.  So, wrap in an AuthenticationException, log to warn, and propagate:
                String msg = "Authentication failed for token submission [" + token + "].  Possible unexpected " +
                        "error? (Typical or expected login exceptions should extend from AuthenticationException).";
                ae = new AuthenticationException(msg, t);
            }
            try {
                notifyFailure(token, ae);
            } catch (Throwable t2) {
                if (log.isWarnEnabled()) {
                    String msg = "Unable to send notification for failed authentication attempt - listener error?.  " +
                            "Please check your AuthenticationListener implementation(s).  Logging sending exception " +
                            "and propagating original AuthenticationException instead...";
                    log.warn(msg, t2);
                }
            }


            throw ae;
        }

        log.debug("Authentication successful for token [{}].  Returned account [{}]", token, info);

        notifySuccess(token, info);

        return info;
    }

protected abstract AuthenticationInfo doAuthenticate(AuthenticationToken token)
            throws AuthenticationException;
```

从上面可以看到实际的认证过程 doAuthenticate 是交给子类来实现的，AbstractAuthenticator 只对认证结果进行处理，认证成功时调用 notifySuccess(token, info)通知所有的 listener，认证失败时调用 notifyFailure(token, ae)通知所有的 listener。

具体的认证过程就需要看 AbstractAuthenticator 子类对于 doAuthenticate 方法的实现，ModularRealmAuthenticator 继承了 AbstractAuthenticator，它有两个重要的属性如下

Java 代码

```java
private Collection<Realm> realms;
private AuthenticationStrategy authenticationStrategy;
```

首先就是 Realm 的概念：就是配置各种角色、权限和用户的地方，即提供了数据源供 shiro 来使用，它能够根据一个 AuthenticationToken 中的用户名和密码来判定是否合法等，文档如下：

Java 代码

```
A <tt>Realm</tt> is a security component that can access application-specific security entities such as users, roles, and permissions to determine authentication and authorization operations
```

接口如下：

Java 代码

```java
public interface Realm {

    String getName();

    boolean supports(AuthenticationToken token);

    AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException;

}
```

Realm 首先有一个重要的 name 属性，全局唯一的标示。supports、getAuthenticationInfo 方法就是框架中非常常见的一种写法，ModularRealmAuthenticator 拥有 Collection<Realm> realms 集合，在判定用户合法性时，会首先调用每个 Realm 的 supports 方法，如果

支持才会去掉用相应的 getAuthenticationInfo 方法。

关于 Realm 的详细接口设计之后再给出详细说明，此时先继续回到 ModularRealmAuthenticator 认证的地方

Java 代码

```java
protected AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken) throws AuthenticationException {
        assertRealmsConfigured();
        Collection<Realm> realms = getRealms();
        if (realms.size() == 1) {
            return doSingleRealmAuthentication(realms.iterator().next(), authenticationToken);
        } else {
            return doMultiRealmAuthentication(realms, authenticationToken);
        }
    }
```

代码很简单，当只有一个 Realm 时先调用 Realm 的 supports 方法看是否支持，若不支持则抛出认证失败的异常，若支持则调用 Realm 的 getAuthenticationInfo(token)方法如下：

Java 代码

```java
protected AuthenticationInfo doSingleRealmAuthentication(Realm realm, AuthenticationToken token) {
        if (!realm.supports(token)) {
            String msg = "Realm [" + realm + "] does not support authentication token [" +
                    token + "].  Please ensure that the appropriate Realm implementation is " +
                    "configured correctly or that the realm accepts AuthenticationTokens of this type.";
            throw new UnsupportedTokenException(msg);
        }
        AuthenticationInfo info = realm.getAuthenticationInfo(token);
        if (info == null) {
            String msg = "Realm [" + realm + "] was unable to find account data for the " +
                    "submitted AuthenticationToken [" + token + "].";
            throw new UnknownAccountException(msg);
        }
        return info;
    }
```

若有多个 Realm 时怎样才算是认证成功的呢？这就需要 ModularRealmAuthenticator 的认证策略 AuthenticationStrategy 来指定，对于 AuthenticationStrategy 目前有三种实现

AllSuccessfulStrategy：即所有的 Realm 都验证通过才算是通过

AtLeastOneSuccessfulStrategy：只要有一个 Realm 验证通过就算通过

FirstSuccessfulStrategy：这个刚开始不太好理解，和 AtLeastOneSuccessfulStrategy 稍微有些区别。AtLeastOneSuccessfulStrategy 返回了所有 Realm 认证成功的信息，FirstSuccessfulStrategy 只返回了第一个 Realm 认证成功的信息。

试想一下，如果让你来设计，你会怎么设计？

然后来具体看下 AuthenticationStrategy 的接口设计：

Java 代码

```java
public interface AuthenticationStrategy {

    AuthenticationInfo beforeAllAttempts(Collection<? extends Realm> realms, AuthenticationToken token) throws AuthenticationException;

    AuthenticationInfo beforeAttempt(Realm realm, AuthenticationToken token, AuthenticationInfo aggregate) throws AuthenticationException;

    AuthenticationInfo afterAttempt(Realm realm, AuthenticationToken token, AuthenticationInfo singleRealmInfo, AuthenticationInfo aggregateInfo, Throwable t)
            throws AuthenticationException;

    AuthenticationInfo afterAllAttempts(AuthenticationToken token, AuthenticationInfo aggregate) throws AuthenticationException;
}
```

验证过程是这样的，每一个 Realm 验证 token 后都会返回一个当前 Realm 的验证信息 AuthenticationInfo singleRealmInfo，然后呢会有一个贯穿所有 Realm 验证过程的验证信息 AuthenticationInfo aggregateInfo，每一个 Realm 验证过后会进行 singleRealmInfo 和 aggregateInfo 的合并，这是大体的流程

对于 AllSuccessfulStrategy 来说：它要确保每一个 Realm 都要验证成功，所以必然

（1）要在 beforeAttempt 中判断当前 realm 是否支持 token，如不支持抛出异常结束验证过程

（2）要在 afterAttempt(Realm realm, AuthenticationToken token, AuthenticationInfo singleRealmInfo, AuthenticationInfo aggregateInfo, Throwable t)中判断是否验证通过了，即异常 t 为空，并且 singleRealmInfo 不为空，则表示验证通过了，然后将 singleRealmInfo 和 aggregateInfo 合并，所以最终返回的 aggregateInfo 是几个 Realm 认证信息合并后的结果

AllSuccessfulStrategy 就会在这两处进行把关，一旦不符合抛出异常，认证失败，如下：

Java 代码

```java
public AuthenticationInfo beforeAttempt(Realm realm, AuthenticationToken token, AuthenticationInfo info) throws AuthenticationException {
        if (!realm.supports(token)) {
            String msg = "Realm [" + realm + "] of type [" + realm.getClass().getName() + "] does not support " +
                    " the submitted AuthenticationToken [" + token + "].  The [" + getClass().getName() +
                    "] implementation requires all configured realm(s) to support and be able to process the submitted " +
                    "AuthenticationToken.";
            throw new UnsupportedTokenException(msg);
        }

        return info;
    }

public AuthenticationInfo afterAttempt(Realm realm, AuthenticationToken token, AuthenticationInfo info, AuthenticationInfo aggregate, Throwable t)
            throws AuthenticationException {
        if (t != null) {
            if (t instanceof AuthenticationException) {
                //propagate:
                throw ((AuthenticationException) t);
            } else {
                String msg = "Unable to acquire account data from realm [" + realm + "].  The [" +
                        getClass().getName() + " implementation requires all configured realm(s) to operate successfully " +
                        "for a successful authentication.";
                throw new AuthenticationException(msg, t);
            }
        }
        if (info == null) {
            String msg = "Realm [" + realm + "] could not find any associated account data for the submitted " +
                    "AuthenticationToken [" + token + "].  The [" + getClass().getName() + "] implementation requires " +
                    "all configured realm(s) to acquire valid account data for a submitted token during the " +
                    "log-in process.";
            throw new UnknownAccountException(msg);
        }

        log.debug("Account successfully authenticated using realm [{}]", realm);

        // If non-null account is returned, then the realm was able to authenticate the
        // user - so merge the account with any accumulated before:
```

```
        merge(info, aggregate);

        return aggregate;
    }
```

对于 AtLeastOneSuccessfulStrategy 来说：它只需确保在所有 Realm 验证完成之后，判断下 aggregateInfo 是否含有用户信息即可，若有则表示有些 Realm 是验证通过了，此时 aggregateInfo 也是合并后的信息，如下

Java 代码

```java
public AuthenticationInfo afterAllAttempts(AuthenticationToken token, AuthenticationInfo aggregate) throws AuthenticationException {
        //we know if one or more were able to succesfully authenticate if the aggregated account object does not
        //contain null or empty data:
        if (aggregate == null || CollectionUtils.isEmpty(aggregate.getPrincipals())) {
            throw new AuthenticationException("Authentication token of type [" + token.getClass() + "] " +
                    "could not be authenticated by any configured realms.  Please ensure that at least one realm can " +
                    "authenticate these tokens.");
        }

        return aggregate;
    }
```

对于 FirstSuccessfulStrategy 来说：它只需要第一个 Realm 验证成功的信息，不需要去进行合并，所以它必须在合并上做手脚，即不会进行合并，一旦有一个 Realm 验证成功，信息保存到 aggregateInfo 中，之后即使再次验证成功也不会进行合并，如下

Java 代码

```java
protected AuthenticationInfo merge(AuthenticationInfo info, AuthenticationInfo aggregate) {
        if (aggregate != null && !CollectionUtils.isEmpty(aggregate.getPrincipals())) {
            return aggregate;
        }
        return info != null ? info : aggregate;
    }
```

验证策略分析完成之后，我们来看下 ModularRealmAuthenticator 的真个验证的代码过程：

Java 代码

```java
protected AuthenticationInfo doMultiRealmAuthentication(Collection<Realm> realms, AuthenticationToken token) {

        AuthenticationStrategy strategy = getAuthenticationStrategy();

        AuthenticationInfo aggregate = strategy.beforeAllAttempts(realms, token);

        if (log.isTraceEnabled()) {
            log.trace("Iterating through {} realms for PAM authentication", realms.size());
        }

        for (Realm realm : realms) {

            aggregate = strategy.beforeAttempt(realm, token, aggregate);

            if (realm.supports(token)) {

                log.trace("Attempting to authenticate token [{}] using realm [{}]", token, realm);

                AuthenticationInfo info = null;
                Throwable t = null;
                try {
                    info = realm.getAuthenticationInfo(token);
                } catch (Throwable throwable) {
                    t = throwable;
                    if (log.isDebugEnabled()) {
```

```java
            String msg = "Realm [" + realm + "] threw an exception during a multi-realm authentication attempt:";
            log.debug(msg, t);
          }
        }

        aggregate = strategy.afterAttempt(realm, token, info, aggregate, t);

      } else {
        log.debug("Realm [{}] does not support token {}.  Skipping realm.", realm, token);
      }
    }

    aggregate = strategy.afterAllAttempts(token, aggregate);

    return aggregate;
  }
```

有了之前的分析，这个过程便变的相当容易了。

再回到我们的入门案例中，有了 AuthenticationInfo 验证信息，之后进行了那些操作呢？

回到 DefaultSecurityManager 的如下 login 方法中：

Java 代码

```java
public Subject login(Subject subject, AuthenticationToken token) throws AuthenticationException {
    AuthenticationInfo info;
    try {
      info = authenticate(token);
    } catch (AuthenticationException ae) {
      try {
        onFailedLogin(token, ae, subject);
      } catch (Exception e) {
        if (log.isInfoEnabled()) {
          log.info("onFailedLogin method threw an " +
              "exception.  Logging and propagating original AuthenticationException.", e);
        }
      }
```

```java
            throw ae; //propagate
        }

        Subject loggedIn = createSubject(token, info, subject);

        onSuccessfulLogin(token, info, loggedIn);

        return loggedIn;
    }
```

Subject loggedIn = createSubject(token, info, subject)会根据已有的 token、认证结果信息 info、和 subject 从新创建一个已登录的 Subject，含有 Session 信息，创建过程如下：

Java 代码

```java
protected Subject createSubject(AuthenticationToken token, AuthenticationInfo info, Subject existing) {
        SubjectContext context = createSubjectContext();
        context.setAuthenticated(true);
        context.setAuthenticationToken(token);
        context.setAuthenticationInfo(info);
        if (existing != null) {
            context.setSubject(existing);
        }
        return createSubject(context);
    }
```

就是填充 SubjectContext，然后根据 SubjectContext 来创建 Subject，此 Subject 的信息是经过 SubjectDAO 保存的，再回到登陆方法：

Java 代码

```java
public void login(AuthenticationToken token) throws AuthenticationException {
        clearRunAsIdentitiesInternal();
        Subject subject = securityManager.login(this, token);

        PrincipalCollection principals;

        String host = null;

        if (subject instanceof DelegatingSubject) {
            DelegatingSubject delegating = (DelegatingSubject) subject;
            //we have to do this in case there are assumed identities - we don't want to lose the 'real' principals:
            principals = delegating.principals;
            host = delegating.host;
        } else {
            principals = subject.getPrincipals();
        }

        if (principals == null || principals.isEmpty()) {
            String msg = "Principals returned from securityManager.login( token ) returned a null or " +
                    "empty value.  This value must be non null and populated with one or more elements.";
            throw new IllegalStateException(msg);
        }
        this.principals = principals;
        this.authenticated = true;
        if (token instanceof HostAuthenticationToken) {
            host = ((HostAuthenticationToken) token).getHost();
        }
        if (host != null) {
            this.host = host;
        }
        Session session = subject.getSession(false);
        if (session != null) {
            this.session = decorate(session);
        } else {
            this.session = null;
        }
    }
```

最后的这些操作就是将刚才创建出来的 Subject 信息复制到我们所使用的 Subject 上，即

Java 代码

```java
subject.login(token)
```

中的 subject 中。至此已经太长了，先告一段落，如 SubjectDAO 和

Session 的细节后面再详细说明。

作者：乒乓狂魔