Shiro 源码分析（二）Subject 和 Session

继续上一篇文章的案例，第一次使用 SecurityUtils.getSubject()来获取 Subject 时

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;
    }
```

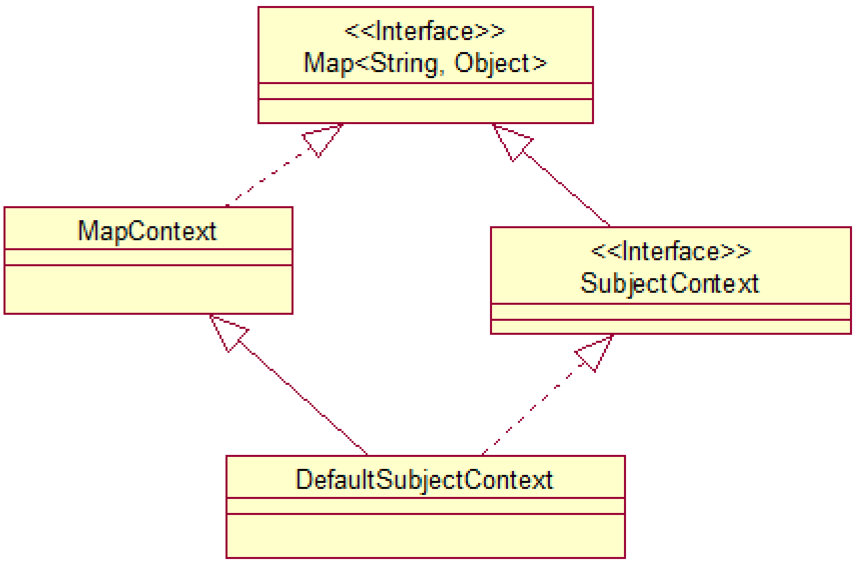
使用 ThreadLocal 模式来获取，若没有则创建一个并绑定到当前线程。此时创建使用的是 Subject 内部类 Builder 来创建的，Builder 会创建一个 SubjectContext 接口的实例 DefaultSubjectContext，最终会委托 securityManager 来根据 SubjectContext 信息来创建一个 Subject，下面详细说下该过程，在 DefaultSecurityManager 的 createSubject 方法中：

Java 代码

```java
public Subject createSubject(SubjectContext subjectContext) {
        SubjectContext context = copy(subjectContext);

        context = ensureSecurityManager(context);

        context = resolveSession(context);

        context = resolvePrincipals(context);

        Subject subject = doCreateSubject(context);

        save(subject);

        return subject;
    }
```

首先就是复制 SubjectContext，SubjectContext 接口继承了 Map<String, Object>，然后加入了几个重要的 SecurityManager、SessionId、Subject、PrincipalCollection、Session、boolean authenticated、boolean sessionCreationEnabled、Host、AuthenticationToken、AuthenticationInfo 等众多信息。

然后来讨论下接口设计：



讨论 1：首先是 SubjectContext 为什么要去实现 Map<String, Object>？

SubjectContext 提供了常用的 get、set 方法，还提供了一个 resolve 方法，以 SecurityManager 为例：

Java 代码

```java
SecurityManager getSecurityManager();

 void setSecurityManager(SecurityManager securityManager);

 SecurityManager resolveSecurityManager();
```

这些 get、set 方法则用于常用的设置和获取，而 resolve 则表示先调用 getSecurityManager，如果获取不到，则使用其他途径来获取，如 DefaultSubjectContext 的实现：

Java 代码

```java
public SecurityManager resolveSecurityManager() {
        SecurityManager securityManager = getSecurityManager();
        if (securityManager == null) {
            if (log.isDebugEnabled()) {
                log.debug("No SecurityManager available in subject context map.  " +
                        "Falling back to SecurityUtils.getSecurityManager() lookup.");
            }
            try {
                securityManager = SecurityUtils.getSecurityManager();
            } catch (UnavailableSecurityManagerException e) {
                if (log.isDebugEnabled()) {
                    log.debug("No SecurityManager available via SecurityUtils.  Heuristics exhausted.", e);
                }
            }
        }
        return securityManager;
    }
```

如果 getSecurityManager 获取不到，则使用 SecurityUtils 工具来获取。

再如 resolvePrincipals

Java 代码

```java
public PrincipalCollection resolvePrincipals() {
        PrincipalCollection principals = getPrincipals();

        if (CollectionUtils.isEmpty(principals)) {
            //check to see if they were just authenticated:
            AuthenticationInfo info = getAuthenticationInfo();
            if (info != null) {
                principals = info.getPrincipals();
            }
        }

        if (CollectionUtils.isEmpty(principals)) {
            Subject subject = getSubject();
            if (subject != null) {
                principals = subject.getPrincipals();
            }
        }

        if (CollectionUtils.isEmpty(principals)) {
            //try the session:
            Session session = resolveSession();
            if (session != null) {
                principals = (PrincipalCollection) session.getAttribute(PRINCIPALS_SESSION_KEY);
            }
        }

        return principals;
    }
```

普通的 getPrincipals()获取不到，尝试使用其他属性来获取。

讨论 2：此时就有一个问题，有必要再对外公开 getPrincipals 方法吗？什么情况下外界会去调用 getPrincipals 方法而不会去调用 resolvePrincipals 方法？

然后我们继续回到上面的类图设计上：

DefaultSubjectContext 继承了 MapContext，MapContext 又实现了 Map<String, Object>，看下此时的 MapContext 有什么东西：

Java 代码

```java
public class MapContext implements Map<String, Object>, Serializable {

    private static final long serialVersionUID = 5373399119017820322L;

    private final Map<String, Object> backingMap;

    public MapContext() {
        this.backingMap = new HashMap<String, Object>();
    }

    public MapContext(Map<String, Object> map) {
        this();
        if (!CollectionUtils.isEmpty(map)) {
            this.backingMap.putAll(map);
        }
    }
  //略
}
```

MapContext 内部拥有一个类型为 HashMap 的 backingMap 属性，大部分方法都由 HashMap 来实现，然后仅仅更改某些行为，MapContext 没有选择去继承 HashMap，而是使用了组合的方式，更加容易去扩展，如 backingMap 的类型不一定非要选择 HashMap，可以换成其他的 Map 实现，一旦 MapContext 选择继承 HashMap，如果想对其他的 Map 类型进行同样的功能增强的话，就需要另写一个类来继承它然后改变一些方法实现，这样的话就会有很多重复代码。这也是设计模式所强调的少用继承多用组合。但是 MapContext 的写法使得子类没法去替换 HashMap，哎，心塞 。

MapContext 又提供了如下几个返回值不可修改的方法：

Java 代码

```java
public Set<String> keySet() {
        return Collections.unmodifiableSet(backingMap.keySet());
    }

    public Collection<Object> values() {
        return Collections.unmodifiableCollection(backingMap.values());
    }

    public Set<Entry<String, Object>> entrySet() {
        return Collections.unmodifiableSet(backingMap.entrySet());
    }
```

有点扯远了。继续回到 DefaultSecurityManager 创建 Subject 的地方：

Java 代码

```java
public Subject createSubject(SubjectContext subjectContext) {
        //create a copy so we don't modify the argument's backing map:
        SubjectContext context = copy(subjectContext);

        //ensure that the context has a SecurityManager instance, and if not, add one:
        context = ensureSecurityManager(context);

        //Resolve an associated Session (usually based on a referenced session ID), and place it in the context before
        //sending to the SubjectFactory.  The SubjectFactory should not need to know how to acquire sessions as the
        //process is often environment specific - better to shield the SF from these details:
        context = resolveSession(context);

        //Similarly, the SubjectFactory should not require any concept of RememberMe - translate that here first
        //if possible before handing off to the SubjectFactory:
        context = resolvePrincipals(context);

        Subject subject = doCreateSubject(context);

        //save this subject for future reference if necessary:
```

```java
        //(this is needed here in case rememberMe principals were resolved and they need to be stored in the
        //session, so we don't constantly rehydrate the rememberMe PrincipalCollection on every operation).
        //Added in 1.2:
        save(subject);

        return subject;
    }
```

对于 context，把能获取到的参数都凑齐，SecurityManager、Session。resolveSession 尝试获取 context 的 map 中获取 Session，若没有则尝试获取 context 的 map 中的 Subject，如果存在的话，根据此 Subject 来获取 Session，若没有再尝试获取 sessionId,若果有了 sessionId 则构建成一个 DefaultSessionKey 来获取对应的 Session。

整个过程如下;

Java 代码

```java
protected SubjectContext resolveSession(SubjectContext context) {
        if (context.resolveSession() != null) {
            log.debug("Context already contains a session.  Returning.");
            return context;
        }
        try {
            //Context couldn't resolve it directly, let's see if we can since we have direct access to
            //the session manager:
            Session session = resolveContextSession(context);
            if (session != null) {
                context.setSession(session);
            }
        } catch (InvalidSessionException e) {
            log.debug("Resolved SubjectContext context session is invalid.  Ignoring and creating an anonymous " +
                    "(session-less) Subject instance.", e);
        }
        return context;
    }
```

先看下 context.resolveSession()：

Java 代码

```java
public Session resolveSession() {
  //这里则是直接从 map 中取出 Session
        Session session = getSession();
        if (session == null) {
            //try the Subject if it exists:
           //若果没有，尝试从 map 中取出 Subject
            Subject existingSubject = getSubject();
            if (existingSubject != null) {
                //这里就是 Subject 获取 session 的方法，需要详细看下
                session = existingSubject.getSession(false);
            }
        }
        return session;
    }
```

existingSubject.getSession(false)：通过 Subject 获取 Session 如下

Java 代码

```java
public Session getSession(boolean create) {
        if (log.isTraceEnabled()) {
            log.trace("attempting to get session; create = " + create +
                    "; session is null = " + (this.session == null) +
                    "; session has id = " + (this.session != null && session.getId() != null));
        }

        if (this.session == null && create) {

            //added in 1.2:
            if (!isSessionCreationEnabled()) {
                String msg = "Session creation has been disabled for the current subject.  This exception indicates " +
                        "that there is either a programming error (using a session when it should never be " +
                        "used) or that Shiro's configuration needs to be adjusted to allow Sessions to be created " +
                        "for the current Subject.  See the " + DisabledSessionException.class.getName() + " JavaDoc " +
                        "for more.";
                throw new DisabledSessionException(msg);
            }

            log.trace("Starting session for host {}", getHost());
            SessionContext sessionContext = createSessionContext();
            Session session = this.securityManager.start(sessionContext);
            this.session = decorate(session);
        }
        return this.session;
    }
```

getSession（）的参数表示是否创建 session，如果 Session 为空，并且传递的参数为 true，则会创建一个 Session。然而这里传递的是 false，也就是说不会在创建 Subject 的时候来创建 Session，所以把创建 Session 过程说完后，再回到此处是要记着不会去创建一个 Session。但是我们可以来看下是如何创建 Session 的，整体三大步骤，先创建一个 SessionContext ，然后根据 SessionContext 来创建 Session，最后是装饰 Session,由于创建 Session 过程内容比较多，先说说装饰 Session。

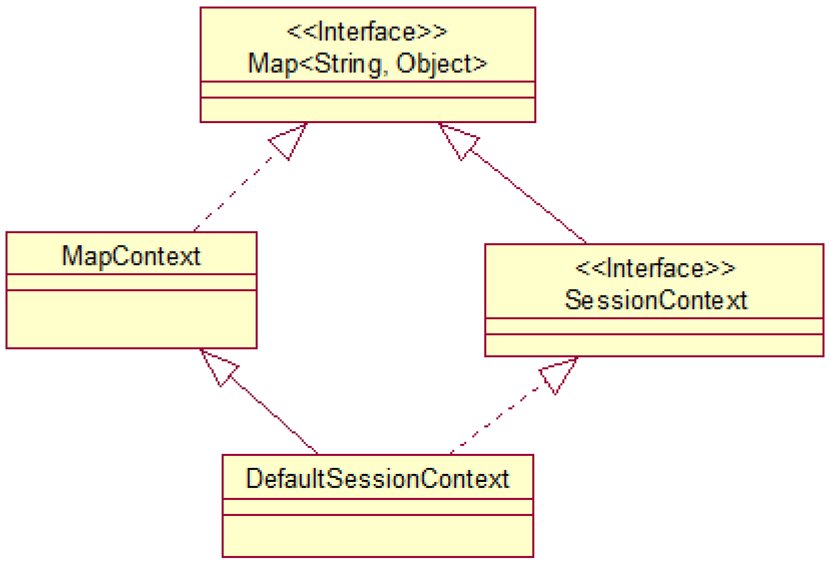
Java 代码

```java
protected Session decorate(Session session) {
        if (session == null) {
            throw new IllegalArgumentException("session cannot be null");
        }
        return new StoppingAwareProxiedSession(session, this);
    }
```

装饰 Session 就是讲 Session 和 DelegatingSubject 封装起来。

然后来说 Session 的创建过程，这和 Subject 的创建方式差不多。

同样是 SessionContext 的接口设计：



和 SubjectContext 相当雷同。

看下 SessionContext 的主要内容:

Java 代码

```java
void setHost(String host);
    String getHost();

    Serializable getSessionId();

    void setSessionId(Serializable sessionId);
```

主要两个内容，host 和 sessionId。

接下来看下如何由 SessionContext 来创建 Session:

Java 代码

```java
protected Session doCreateSession(SessionContext context) {
        Session s = newSessionInstance(context);
        if (log.isTraceEnabled()) {
            log.trace("Creating session for host {}", s.getHost());
        }
        create(s);
        return s;
    }

    protected Session newSessionInstance(SessionContext context) {
        return getSessionFactory().createSession(context);
    }
```

和 Subject 一样也是由一个 SessionFactory 根据 SessionContext 来创建出一个 Session，看下默认的 SessionFactory SimpleSessionFactory 的创建过程：

Java 代码

```java
public Session createSession(SessionContext initData) {
        if (initData != null) {
            String host = initData.getHost();
            if (host != null) {
                return new SimpleSession(host);
            }
        }
        return new SimpleSession();
    }
```

如果 SessionContext 有 host 信息，就传递给 Session，然后就是直接 new 一个 Session 接口的实现 SimpleSession，先看下 Session 接口有哪些内容：

Java 代码

```java
public interface Session {
    Serializable getId();
    Date getStartTimestamp();
    Date getLastAccessTime();
    long getTimeout() throws InvalidSessionException;
    void setTimeout(long maxIdleTimeInMillis) throws InvalidSessionException;
    String getHost();
    void touch() throws InvalidSessionException;
    void stop() throws InvalidSessionException;
    Collection<Object> getAttributeKeys() throws InvalidSessionException;
    Object getAttribute(Object key) throws InvalidSessionException;
    void setAttribute(Object key, Object value) throws InvalidSessionException;
    Object removeAttribute(Object key) throws InvalidSessionException;
}
```

id:Session 的唯一标识，创建时间、超时时间等内容。

再看 SimpleSession 的创建过程：

Java 代码

```java
public SimpleSession() {
        this.timeout = DefaultSessionManager.DEFAULT_GLOBAL_SESSION_TIMEOUT;
        this.startTimestamp = new Date();
        this.lastAccessTime = this.startTimestamp;
    }

    public SimpleSession(String host) {
        this();
        this.host = host;
    }
```

设置下超时时间为 DefaultSessionManager.DEFAULT_GLOBAL_SESSION_TIMEOUT 30 分钟，startTimestamp 和 lastAccessTime 设置为现在开始。就这样构建出了一个 Session 的实例，然后就是需要将该实例保存起来：

Java 代码

```java
protected Session doCreateSession(SessionContext context) {
        Session s = newSessionInstance(context);
        if (log.isTraceEnabled()) {
            log.trace("Creating session for host {}", s.getHost());
        }
        create(s);
        return s;
    }
protected void create(Session session) {
        if (log.isDebugEnabled()) {
            log.debug("Creating new EIS record for new session instance [" + session + "]");
        }
        sessionDAO.create(session);
    }
```

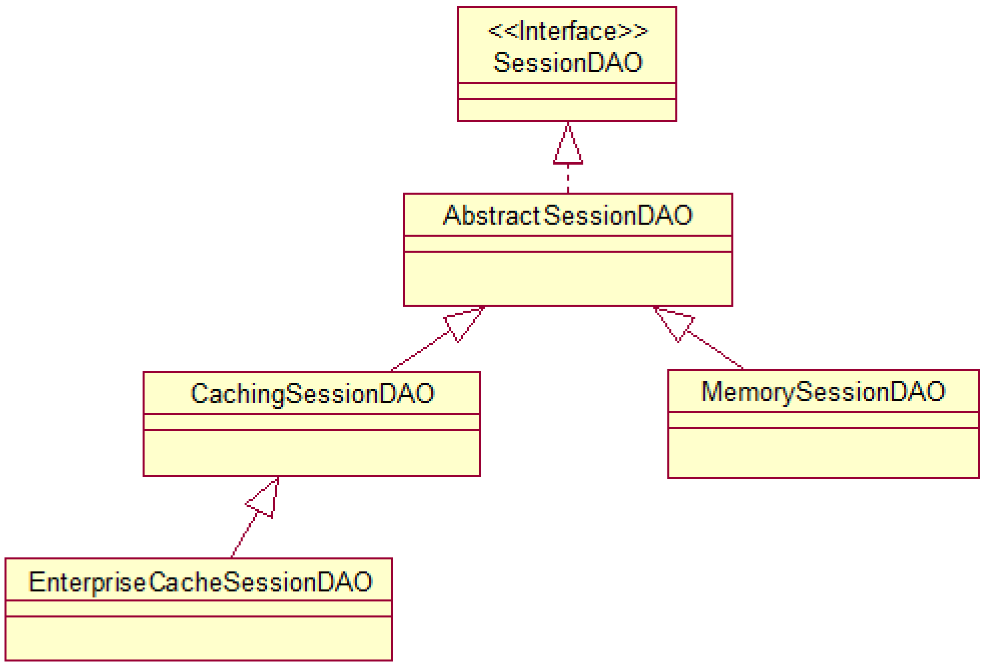
即该进行 create(s)操作了，又和 Subject 极度的相像，使用

sessionDAO 来保存刚才创建的 Session。再来看下 SessionDAO 接口：

Java 代码

```java
public interface SessionDAO {
    Serializable create(Session session);
    Session readSession(Serializable sessionId) throws UnknownSessionException;
    void update(Session session) throws UnknownSessionException;
    void delete(Session session);
    Collection<Session> getActiveSessions();
}
```

也就是对所有的 Session 进行增删该查，SessionDAO 接口继承关系如下：



AbstractSessionDAO：有一个重要的属性 SessionIdGenerator，它负责给 Session 创建 sessionId，SessionIdGenerator 接口如下：

Java 代码 ☆

```java
public interface SessionIdGenerator {
    Serializable generateId(Session session);
}
```

很简单，参数为 Session，返回 sessionId。SessionIdGenerator 的实现有两个 JavaUuidSessionIdGenerator、RandomSessionIdGenerator。而 AbstractSessionDAO 默认采用的是 JavaUuidSessionIdGenerator，如下：

Java 代码 ☆

```java
public AbstractSessionDAO() {
        this.sessionIdGenerator = new JavaUuidSessionIdGenerator();
    }
```

MemorySessionDAO 继承了 AbstractSessionDAO，它把 Session 存储在一个 ConcurrentMap<Serializable, Session> sessions 集合中，key 为 sessionId，value 为 Session。

CachingSessionDAO：主要配合在别的地方存储 session。先不介绍，之后的文章再详细说。

对于本案例来说 SessionDAO 为 MemorySessionDAO。至此整个 Session 的创建过程就走通了。

刚才虽然说了整个 Session 的创建过程，回到上文所说的，不会去创建 Session 的地方。在创建 Subject 搜集 session 信息时，使用的此时的 Subject 的 Session、sessionId 都为空，所以获取不到 Session。然后就是 doCreateSubject：

Java 代码 ☆

```java
protected Subject doCreateSubject(SubjectContext context) {
        return getSubjectFactory().createSubject(context);
    }
```

就是通过 SubjectFactory 工厂接口来创建 Subject 的，而 DefaultSecurityManager 默认使用的 SubjectFactory 是 DefaultSubjectFactory：

Java 代码

```java
public DefaultSecurityManager() {
        super();
        this.subjectFactory = new DefaultSubjectFactory();
        this.subjectDAO = new DefaultSubjectDAO();
    }
```

继续看 DefaultSubjectFactory 是怎么创建 Subject 的：

Java 代码

```java
public Subject createSubject(SubjectContext context) {
        SecurityManager securityManager = context.resolveSecurityManager();
        Session session = context.resolveSession();
        boolean sessionCreationEnabled = context.isSessionCreationEnabled();
        PrincipalCollection principals = context.resolvePrincipals();
        boolean authenticated = context.resolveAuthenticated();
        String host = context.resolveHost();

        return new DelegatingSubject(principals, authenticated, host, session, sessionCreationEnabled, securityManager);
    }
```

仍然就是将这些属性传递给 DelegatingSubject，也没什么好说的。创建完成之后，就需要将刚创建的 Subject 保存起来，仍回到：

Java 代码

```java
public Subject createSubject(SubjectContext subjectContext) {
        //create a copy so we don't modify the argument's backing map:
        SubjectContext context = copy(subjectContext);

        //ensure that the context has a SecurityManager instance, and if not, add one:
        context = ensureSecurityManager(context);

        //Resolve an associated Session (usually based on a referenced session ID), and place it in the context before
        //sending to the SubjectFactory.  The SubjectFactory should not need to know how to acquire sessions as the
        //process is often environment specific - better to shield the SF from these details:
        context = resolveSession(context);

        //Similarly, the SubjectFactory should not require any concept of RememberMe - translate that here first
        //if possible before handing off to the SubjectFactory:
        context = resolvePrincipals(context);

        Subject subject = doCreateSubject(context);

        //save this subject for future reference if necessary:
        //(this is needed here in case rememberMe principals were resolved and they need to be stored in the
        //session, so we don't constantly rehydrate the rememberMe PrincipalCollection on every operation).
        //Added in 1.2:
        save(subject);

        return subject;
    }
```

来看下 save 方法：

Java 代码 ☆

```java
protected void save(Subject subject) {
        this.subjectDAO.save(subject);
    }
```

可以看到又是使用另一个模块来完成的即 SubjectDAO，SubjectDAO 接口如下：

Java 代码 ☆

```java
public interface SubjectDAO {
    Subject save(Subject subject);
    void delete(Subject subject);
}
```

很简单，就是保存和删除一个 Subject。我们看下具体的实现类 DefaultSubjectDAO 是如何来保存的：

Java 代码 ☆

```java
public Subject save(Subject subject) {
        if (isSessionStorageEnabled(subject)) {
            saveToSession(subject);
        } else {
            log.trace("Session storage of subject state for Subject [{}] has been disabled: identity and " +
                    "authentication state are expected to be initialized on every request or invocation.", subject);
        }

        return subject;
    }
```

首先就是判断 isSessionStorageEnabled，是否要存储该 Subject 的 session 来

DefaultSubjectDAO：有一个重要属性 SessionStorageEvaluator，它

是用来决定一个 Subject 的 Session 来记录 Subject 的状态，接口如下

Java 代码

```java
public interface SessionStorageEvaluator {
    boolean isSessionStorageEnabled(Subject subject);
}
```

其实现为 DefaultSessionStorageEvaluator：

Java 代码

```java
public class DefaultSessionStorageEvaluator implements SessionStorageEvaluator {

    private boolean sessionStorageEnabled = true;

    public boolean isSessionStorageEnabled(Subject subject) {
        return (subject != null && subject.getSession(false) != null) || isSessionStorageEnabled();
    }
```

决定策略就是通过 DefaultSessionStorageEvaluator 的 sessionStorageEnabled 的 true 或 false 和 subject 是否有 Session 对象来决定的。如果允许存储 Subject 的 Session 的话，下面就说具体的存储过程：

Java 代码

```java
protected void saveToSession(Subject subject) {
        //performs merge logic, only updating the Subject's session if it does not match the current state:
        mergePrincipals(subject);
        mergeAuthenticationState(subject);
    }
protected void mergePrincipals(Subject subject) {
        //merge PrincipalCollection state:

        PrincipalCollection currentPrincipals = null;

        //SHIRO-380: added if/else block - need to retain original (source) principals
        //This technique (reflection) is only temporary - a proper long term solution needs to be found,
        //but this technique allowed an immediate fix that is API point-version forwards and backwards compatible
        //
        //A more comprehensive review / cleaning of runAs should be performed for Shiro 1.3 / 2.0 +
        if (subject.isRunAs() && subject instanceof DelegatingSubject) {
            try {
                Field field = DelegatingSubject.class.getDeclaredField("principals");
                field.setAccessible(true);
                currentPrincipals = (PrincipalCollection)field.get(subject);
            } catch (Exception e) {
                throw new IllegalStateException("Unable to access DelegatingSubject principals property.", e);
            }
        }
        if (currentPrincipals == null || currentPrincipals.isEmpty()) {
            currentPrincipals = subject.getPrincipals();
        }

        Session session = subject.getSession(false);

        if (session == null) {
           //只有当 Session 为空，并且 currentPrincipals 不为空的时候才会去创建 Session
           //Subject subject = SecurityUtils.getSubject()此时两者都是为空的，
           //不会去创建 Session
            if (!CollectionUtils.isEmpty(currentPrincipals)) {
                session = subject.getSession();
                session.setAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY, currentPrincipals);
            }
            //otherwise no session and no principals - nothing to save
        } else {
            PrincipalCollection existingPrincipals =
                    (PrincipalCollection) session.getAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY);

            if (CollectionUtils.isEmpty(currentPrincipals)) {
                if (!CollectionUtils.isEmpty(existingPrincipals)) {
                    session.removeAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY);
                }
                //otherwise both are null or empty - no need to update the session
            } else {
                if (!currentPrincipals.equals(existingPrincipals)) {
                    session.setAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY, currentPrincipals);
                }
                //otherwise they're the same - no need to update the session
            }
        }
    }
```

上面有我们关心的重点，当 subject.getSession(false)获取的 Session 为空时（它不会去创建 Session），此时就需要去创建 Session，subject.getSession()则默认调用的是 subject.getSession(true),则会进行 Session 的创建，创建过程上文已详细说明了。

在第一次创建 Subject 的时候

Java 代码

```java
Subject subject = SecurityUtils.getSubject();
```

虽然 Session 为空，但此时还没有用户身份信息，也不会去创建 Session。案例中的 subject.login(token)，该过程则会去创建 Session，具体看下过程：

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);
                }
            }
            throw ae; //propagate
        }
         //在该过程会进行 Session 的创建
        Subject loggedIn = createSubject(token, info, subject);

        onSuccessfulLogin(token, info, loggedIn);

        return loggedIn;
    }
```

对于验证过程上篇文章已经简单说明了，这里不再说明，重点还是在验证通过后，会设置 Subject 的身份，即用户名：

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);
    }
```

有了认证成功的 AuthenticationInfo 信息，SubjectContext 在 resolvePrincipals 便可以获取用户信息，即通过 AuthenticationInfo 的 getPrincipals()来获得。

Java 代码

```java
public PrincipalCollection resolvePrincipals() {
        PrincipalCollection principals = getPrincipals();

        if (CollectionUtils.isEmpty(principals)) {
            //check to see if they were just authenticated:
            AuthenticationInfo info = getAuthenticationInfo();
            if (info != null) {
                principals = info.getPrincipals();
            }
        }

        if (CollectionUtils.isEmpty(principals)) {
            Subject subject = getSubject();
            if (subject != null) {
                principals = subject.getPrincipals();
            }
        }

        if (CollectionUtils.isEmpty(principals)) {
            //try the session:
            Session session = resolveSession();
            if (session != null) {
                principals = (PrincipalCollection) session.getAttribute(PRINCIPALS_SESSION_KEY);
            }
        }

        return principals;
    }
```

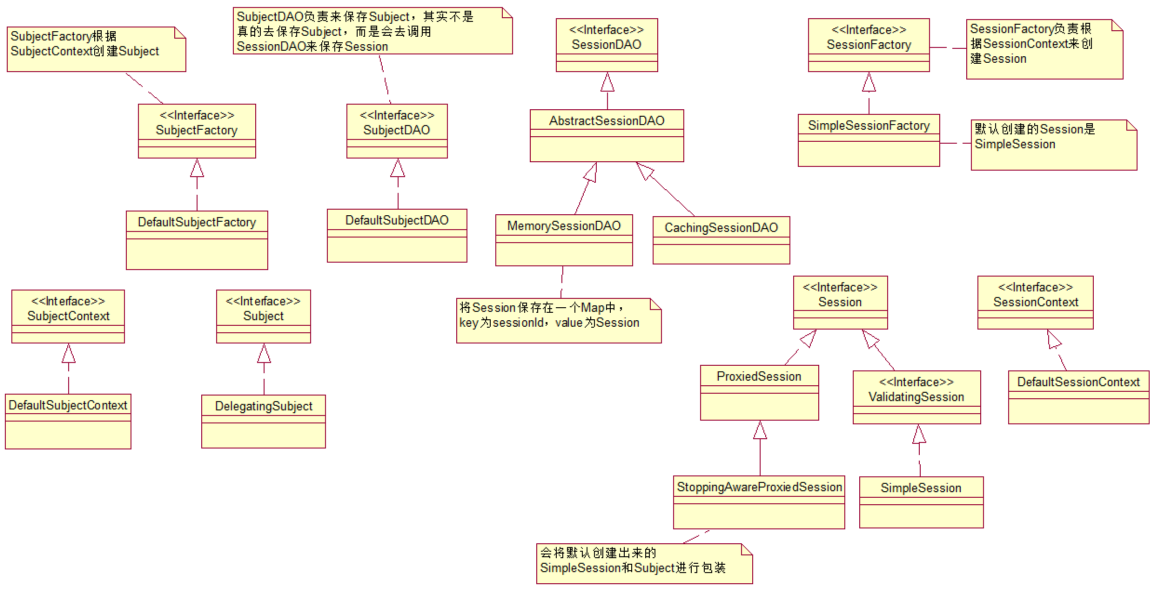
PrincipalCollection 不为空了，在 save(subject)的时候会得到 session 为空，同时 PrincipalCollection 不为空，则会执行 Session 的创建。也就是说在认证通过后，会执行 Session 的创建，Session 创建完成之后会进行一次装饰，即用 StoppingAwareProxiedSession 将创建出来的 session 和 subject 关联起来，然后又进行如下操作：

Java 代码

```java
public void login(AuthenticationToken token) throws AuthenticationException {
        clearRunAsIdentitiesInternal();
        //这里的 Subject 则是经过认证后创建的并且也含有刚才创建的 session，类型为
        //StoppingAwareProxiedSession，即是该 subject 本身和 session 的合体。
        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) {
            //在这里可以看到又进行了一次装饰
            this.session = decorate(session);
        } else {
            this.session = null;
        }
}
```

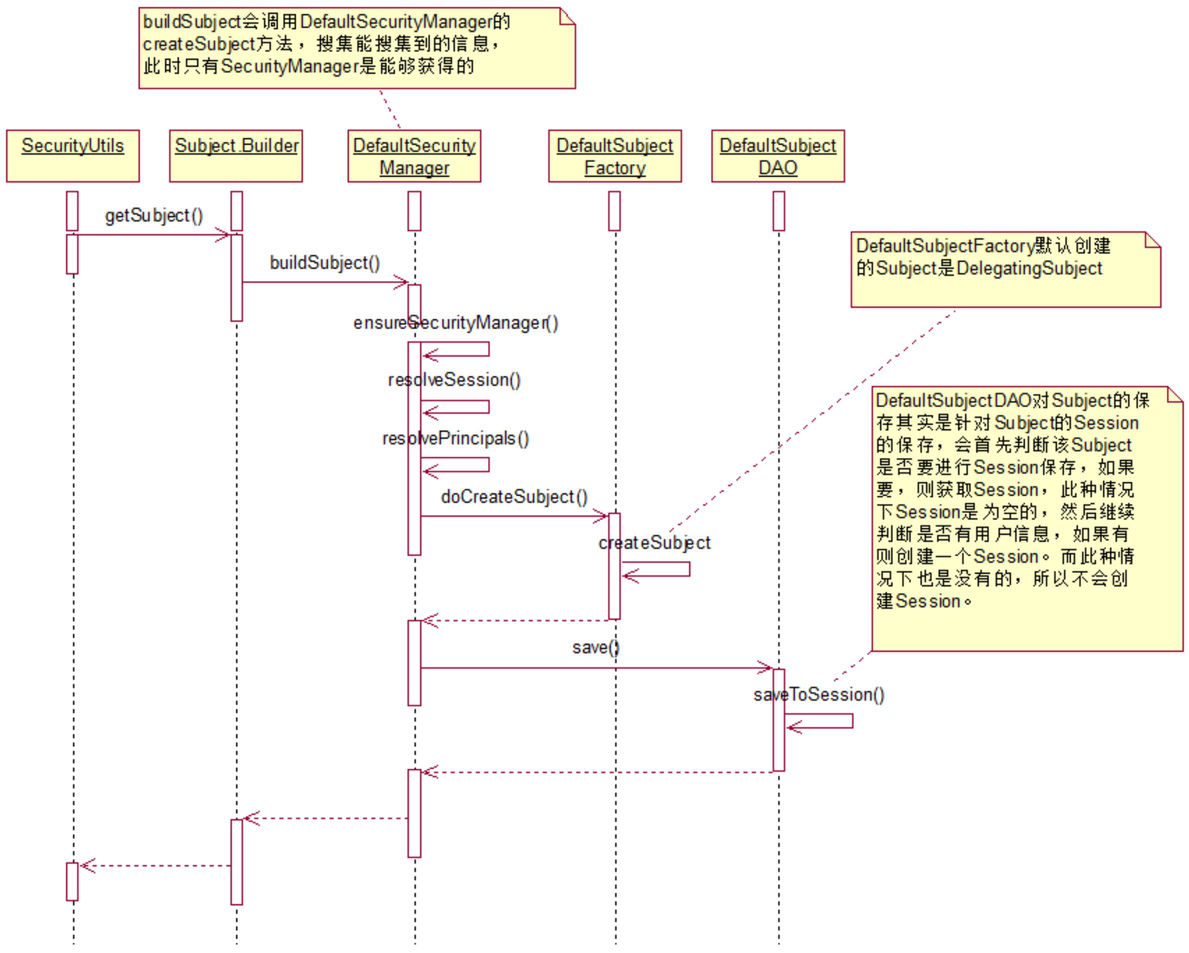
subject 创建出来之后，暂且叫内部 subject，就是把认证通过的内部 subject 的信息和 session 复制给我们外界使用的 subject.login(token) 的 subject 中，这个 subject 暂且叫外部 subject，看下 session 的赋值，又进行了一次装饰，这次装饰则把 session(类型为 StoppingAwareProxiedSession，即是内部 subject 和 session 的合体) 和外部 subject 绑定到一起。

最后来总结下，首先是 Subject 和 Session 的接口类图：

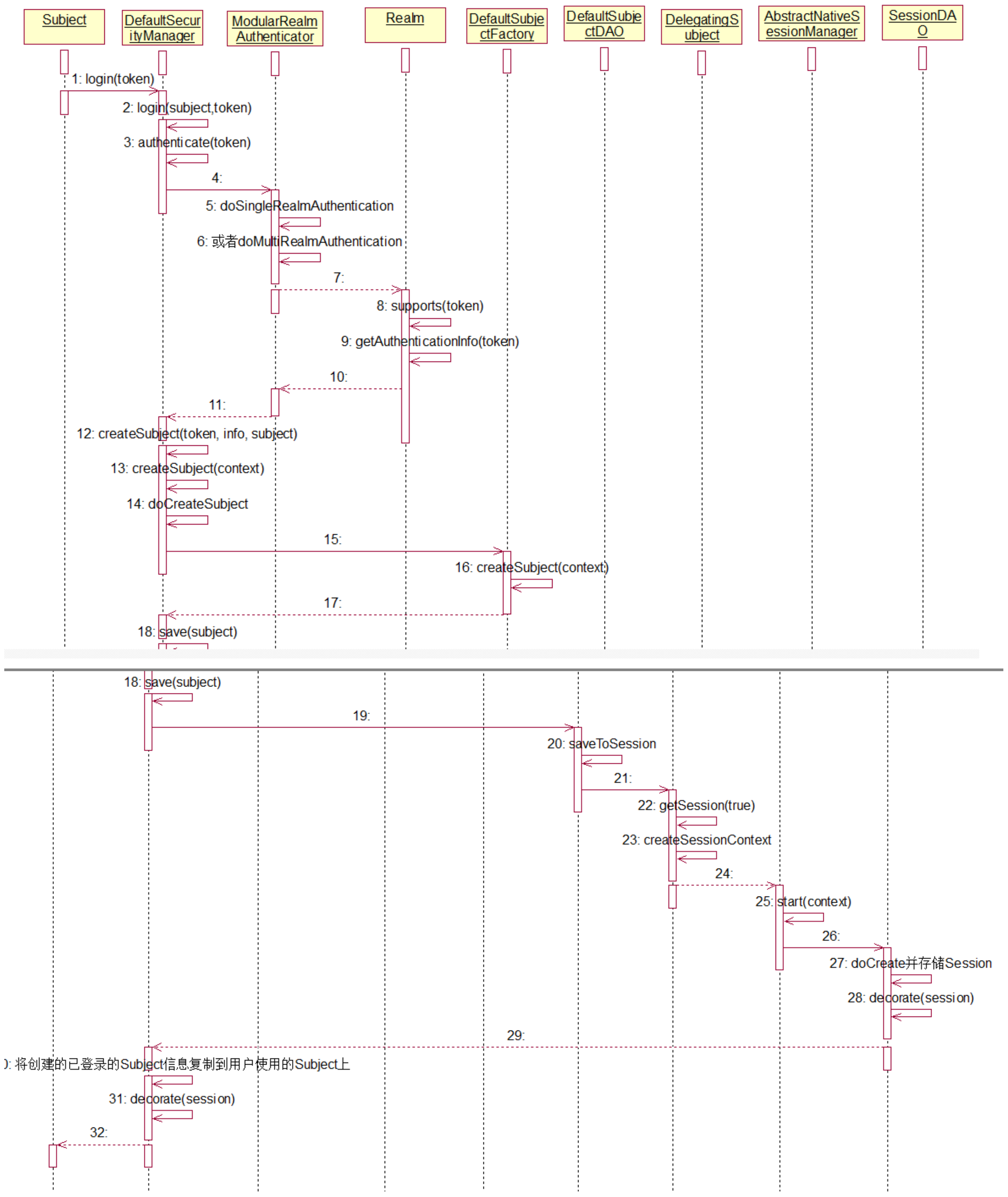


SubjectFactory根据SubjectContext创建Subject
SubjectDAO负责来保存Subject，其实不是真的去保存Subject，而是会去调用SessionDAO来保存Session
<<Interface>> SessionDAO
<<Interface>> SessionFactory
SessionFactory负责根据SessionContext来创建Session
<<Interface>> SubjectFactory
<<Interface>> SubjectDAO
AbstractSessionDAO
SimpleSessionFactory
默认创建的Session是SimpleSession
DefaultSubjectFactory
DefaultSubjectDAO
MemorySessionDAO
CachingSessionDAO
<<Interface>> SubjectContext
<<Interface>> Subject
将Session保存在一个Map中，key为sessionId，value为Session
<<Interface>> Session
<<Interface>> SessionContext
DefaultSubjectContext
DelegatingSubject
ProxiedSession
<<Interface>> ValidatingSession
DefaultSessionContext
StoppingAwareProxiedSession
SimpleSession
会将默认创建出来的SimpleSession和Subject进行包装

然后就是 Subject subject = SecurityUtils.getSubject()的一个简易的流程图：


buildSubject会调用DefaultSecurityManager的createSubject方法，搜集能搜集到的信息，此时只有SecurityManager是能够获得的
SecurityUtils
Subject.Builder
DefaultSecurity Manager
DefaultSubject Factory
DefaultSubject DAO
DefaultSubjectFactory默认创建的Subject是DelegatingSubject
getSubject()
buildSubject()
ensureSecurityManager()
resolveSession()
resolvePrincipals()
doCreateSubject()
createSubject
DefaultSubjectDAO对Subject的保存其实是针对Subject的Session的保存，会首先判断该Subject是否要进行Session保存，如果要，则获取Session，此种情况下Session是为空的，然后继续判断是否有用户信息，如果有则创建一个Session。而此种情况下也是没有的，所以不会创建Session。
save()
saveToSession()

最后是 subject.login(token)的简易流程图：



Subject
DefaultSecurityManager
ModularRealmAuthenticator
Realm
DefaultSubjectFactory
DefaultSubjectDAO
DelegatingSubject
AbstractNativeSessionManager
SessionDAO
1: login(token)
2: login(subject,token)
3: authenticate(token)
4:
5: doSingleRealmAuthentication
6: 或者doMultiRealmAuthentication
7:
8: supports(token)
9: getAuthenticationInfo(token)
10:
11:
12: createSubject(token, info, subject)
13: createSubject(context)
14: doCreateSubject
15:
16: createSubject(context)
17:
18: save(subject)
18: save(subject)
19:
20: saveToSession
21:
22: getSession(true)
23: createSessionContext
24:
25: start(context)
26:
27: doCreate并存储Session
28: decorate(session)
29:
): 将创建的已登录的Subject信息复制到用户使用的Subject上
31: decorate(session)
32:

作者：乒乓狂魔