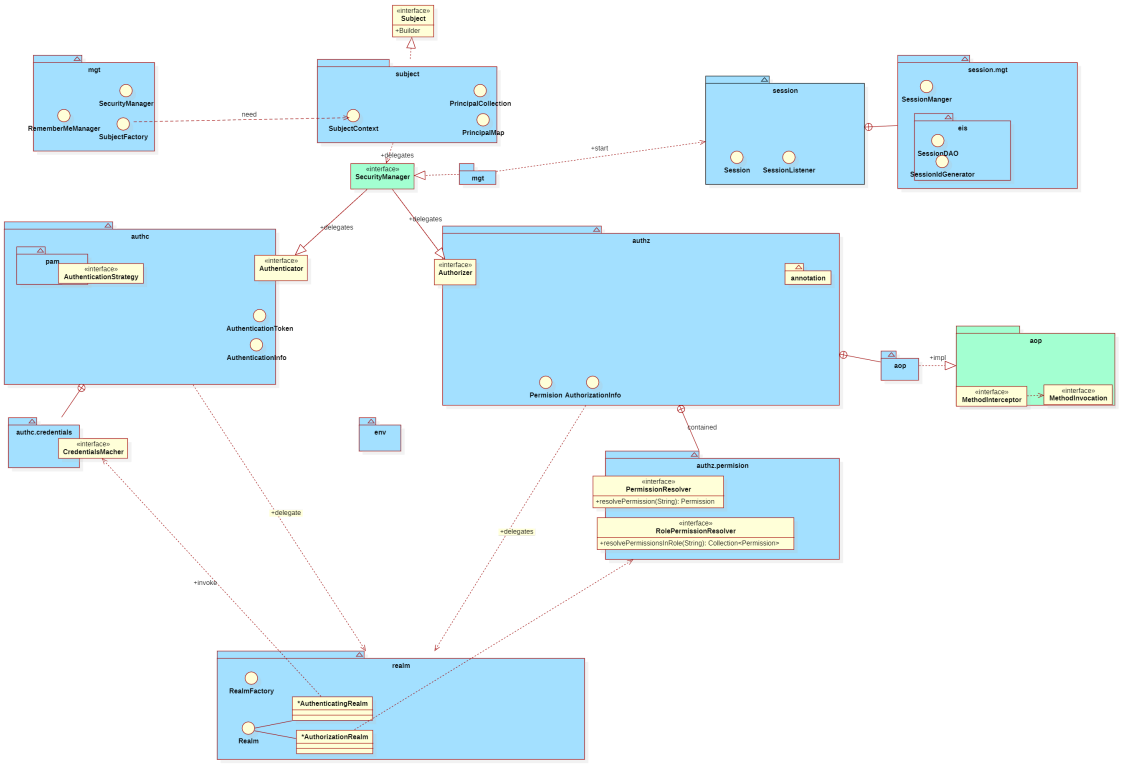
shiro-core

- Introduction to Apache Shiro | Apache Shiro： 【Abstraction： 概念层（抽象的功能描述， 对应需求分析阶段）】
- 整体架构：Apache Shiro Architecture | Apache Shiro 【Abstraction：设计 】【Process: 体系结构】
- 具体的包结构（仅包括最核心部分） 【Abstraction：设计 】【Process: 部署 + 接口】



- **整体初始化流程** 【Abstraction：实现】【Process: 流程】

> 所有的开始，都是从配置开始,配置的方式主要有两种：编程式配置，配置文件读取与解析（利用反射机制，调用构造函数，setter,getter),
>
> 初始化的对象，主要是SecurityManager
>
> Apache Shiro Configuration | Apache Shiro

Subject

设计模式

> Abstraction == 设计

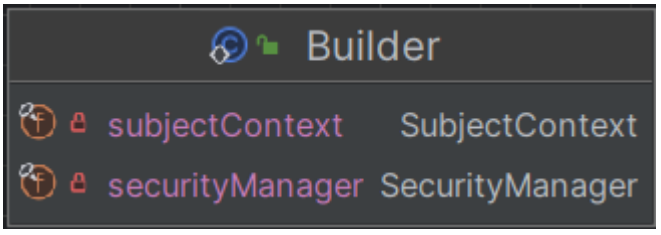
- builder pattern
- 工厂模式

实现

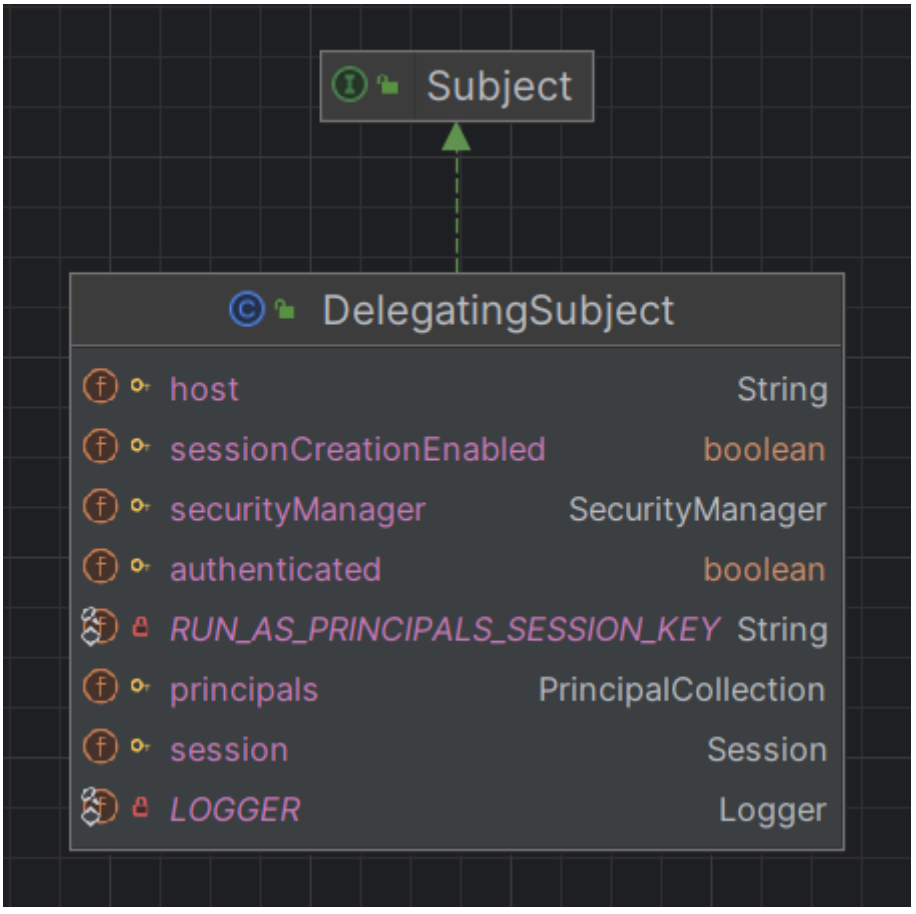
> [Abstraction] = 实现

- **数据结构**

  Subject.Builder



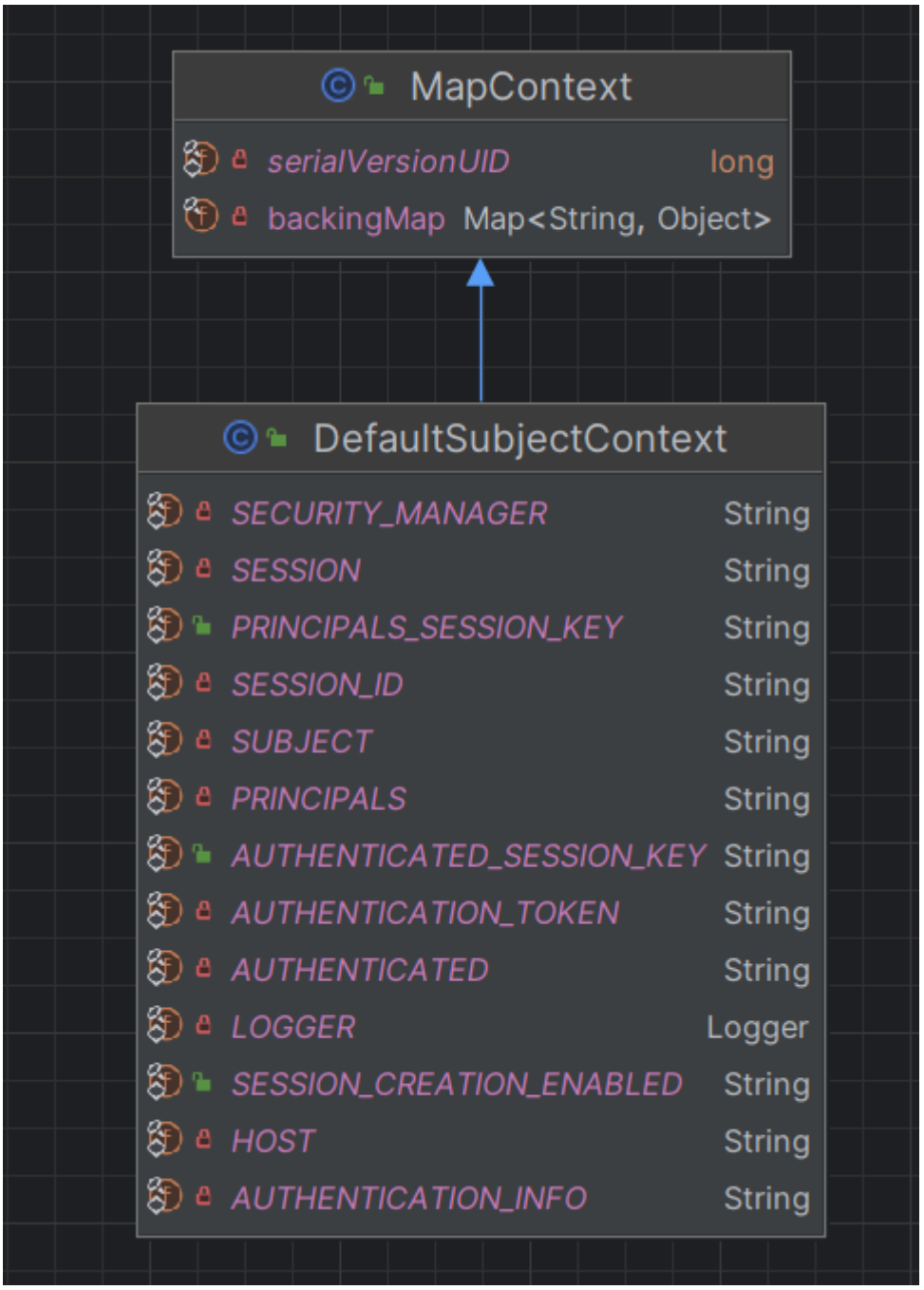
  DelegatingSubject



- **创建过程**

  1. SubjectContext： Subject == 1:1 ， SubjectFactory每生产一个subject都需要一个SubjectContext实例（本质是Map），

且已经定义好了key（常量）



2. 其创建由securityManager负责,故其建立在securtyManager初始化完成的基础上

    1. Subject.Builder

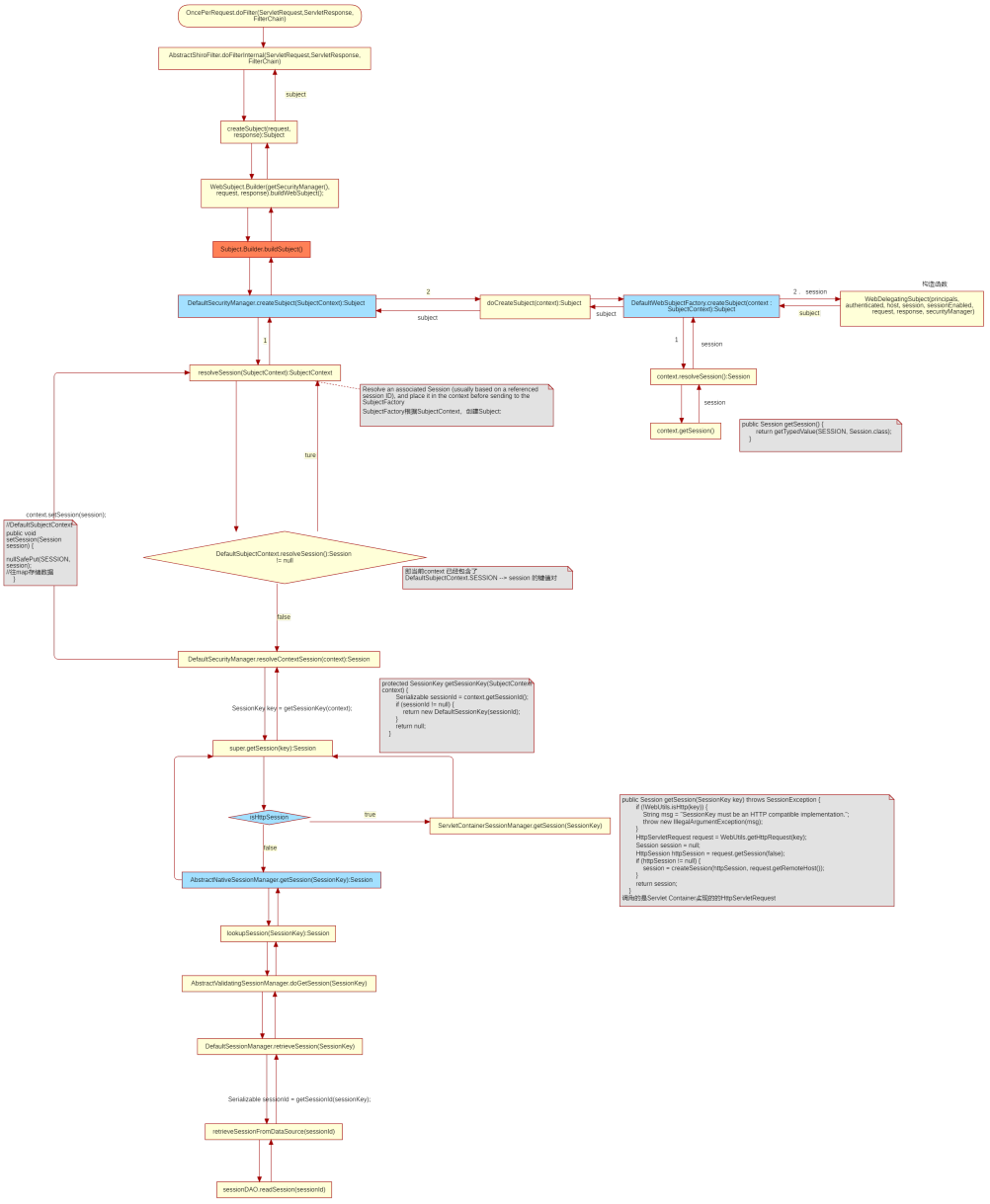
```
public Subject buildSubject() {
    return this.securityManager.createSubject(this.subjectContext);
}
```

    2. DefaultSecurityManager

```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:
    if (subjectContext.isSessionCreationEnabled()) {
        save(subject);
    }

    return subject;
}
```

```java
protected SubjectContext resolveSession(SubjectContext context) {
    if (context.resolveSession() != null) {
        LOGGER.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);  //假设查找不到，那么subject.getSession(true)再创建,惰性原则
        if (session != null) {
            context.setSession(session);
        }
    } catch (InvalidSessionException e) {
        LOGGER.debug("Resolved SubjectContext context session is invalid.  Ignoring and creating an anonymous "
                + "(session-less) Subject instance.", e);
    }
    return context;
}
```

后续过程非常复杂，其大概过程如下图（下图是shiro-web的过程，但核心部分并没变），主要是从sessionDAO中查询session,没有找到也没关系，调用subject.getSession(create:boolean)时还有机会创建



- **工作流程**

  The `Subject` instance, typically a `DelegatingSubject` (or a subclass) delegates to the application's `SecurityManager`

  login

```java
public void login(AuthenticationToken token) throws AuthenticationException {
    clearRunAsIdentitiesInternal();
    Subject subject = securityManager.login( subject: 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( create: false);
    if (session != null) {
        this.session = decorate(session);
    } else {
        this.session = null;
    }
}
```

checkPermission

```java
public void checkPermission(String permission) throws AuthorizationException {
    assertAuthzCheckPossible();
    securityManager.checkPermission(getPrincipals(), permission);
}
```

SecurityManager

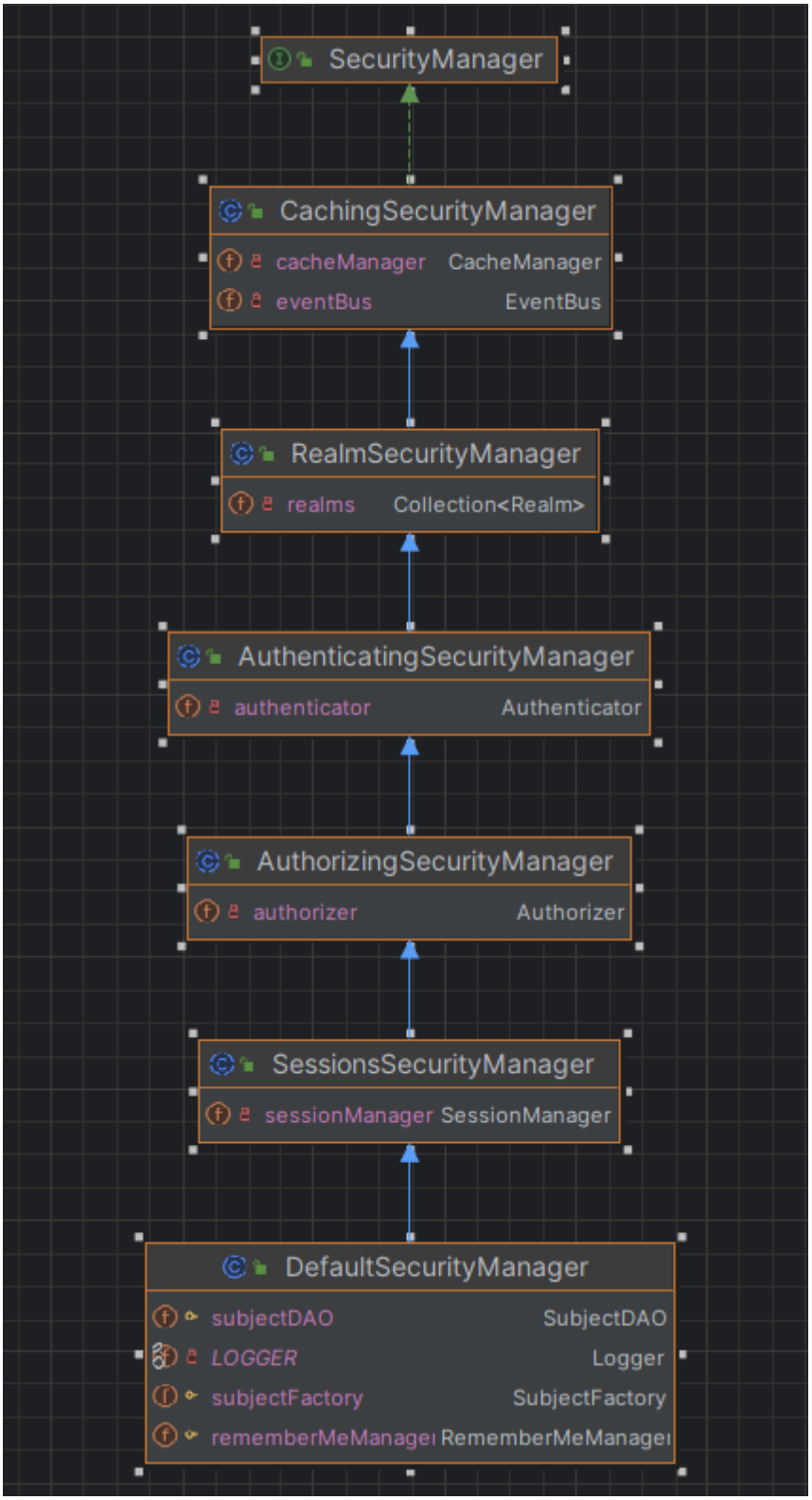
设计模式

- 工厂模式： SessionManagerFactory
- 委托：将验证,授权,会话管理都委托给Authenticator， Authorizer,SessionManager

实现

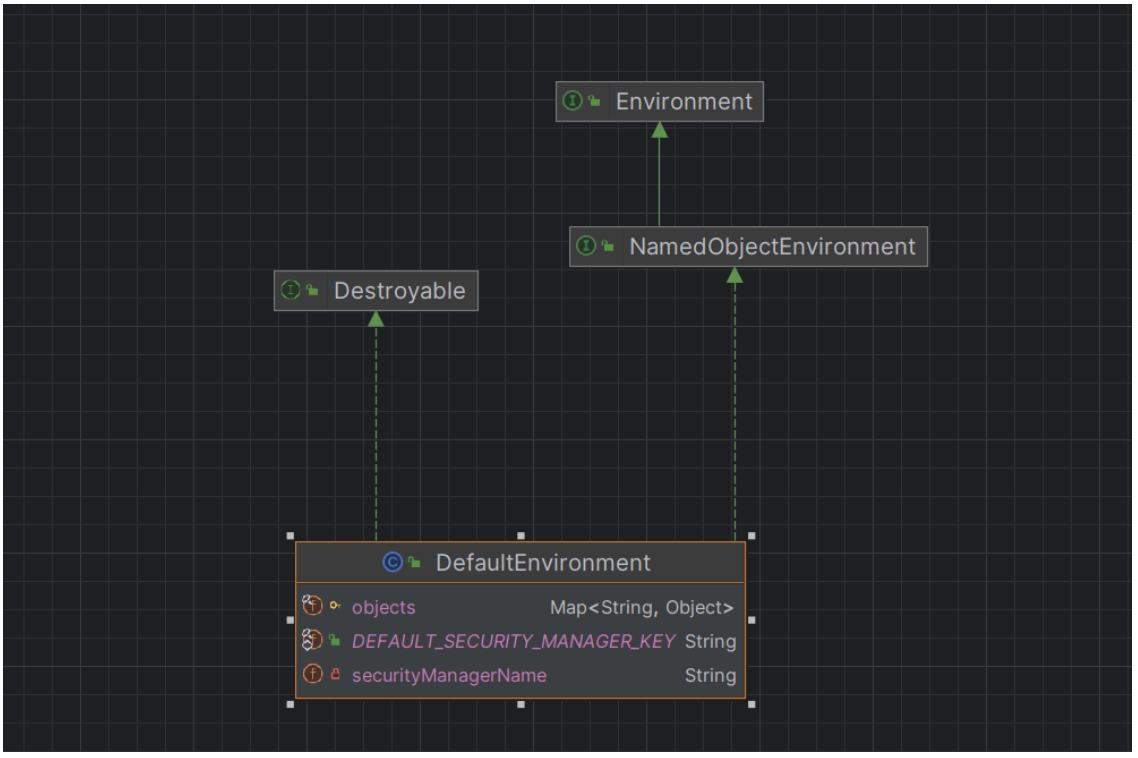
数据结构

> 每个实例其实包含了其所有父类的所有字段（不包括静态），包括私有字段，不过需要提供getter()



可以看出其包含（引用）了大部分其他组件，因此对于Environment（也就是"ShiroContext",shiro的**环境变量集**），最主要的环境变量是就SecurityManager

> **Environment**:

> An Environment instance encapsulates all of the objects that Shiro requires to function. **It is essentially a 'meta' object from which all Shiro components can be obtained for an application. An Environment instance is usually created as a result of parsing a Shiro configuration file.** The environment instance can be stored in any place the application deems necessary, and from it, can retrieve any of Shiro's components that might be necessary in implementing security behavior. For example, the most obvious component accessible via an Environment instance is the application's securityManager.
>
> 
>

创建过程

通过SessionManagerFactory创建，T 这里是SeesionManager

```java
public T createInstance() {  2 usages
    Ini ini = resolveIni();

    T instance;

    if (CollectionUtils.isEmpty(ini)) {
        LOGGER.debug("No populated Ini available.  Creating a default instance.");
        instance = createDefaultInstance();
        if (instance == null) {
            String msg = getClass().getName() + " implementation did not return a default instance in "
                    + "the event of a null/empty Ini configuration.  This is required to support the "
                    + "Factory interface.  Please check your implementation.";
            throw new IllegalStateException(msg);
        }
    } else {
        LOGGER.debug("Creating instance from Ini [" + ini + "]");
        instance = createInstance(ini);
        if (instance == null) {
            String msg = getClass().getName() + " implementation did not return a constructed instance from "
                    + "the createInstance(Ini) method implementation.";
            throw new IllegalStateException(msg);
        }
    }

    return instance;
}
```

默认情况下（不进行任何配置）SessionManagerFactory调用无参构造函数，在构造的过程中会连带着相关组件初始化（Authenticator，Authorizer,SessionManger，都是默认对象，Default***)

如果配置，无非也是利用反射，调用构造函数，setter,getter，进行具体设置

工作过程

- subject的创建（已经给出）
- 其他都委托给其他组件

Authenticator

设计模式

- **AOP（编程思想，而不是模式）** && **策略模式**

Filter、Interceptor、Spring AOP 的对比

Filter、Interceptor 和 Spring AOP 都体现了面向切面编程（AOP）的思想，本质上是为了解决**横切关注点**的问题。它们的主要区别在于**粒度（切入点的层次不同）**，但其核心思想一致：将业务逻辑和通用逻辑（如认证、日志、事务）解耦，实现关注点分离。

> AOP来源于 关注点分离原则

类型	切入点	
Filter	程序级别：Servlet	

类型ceptor	级别：Controller	
spring AOP	方法级别	

所以，我们可以说，**Filter、Interceptor 和 AOP 都是面向切面编程的不同层次实现**

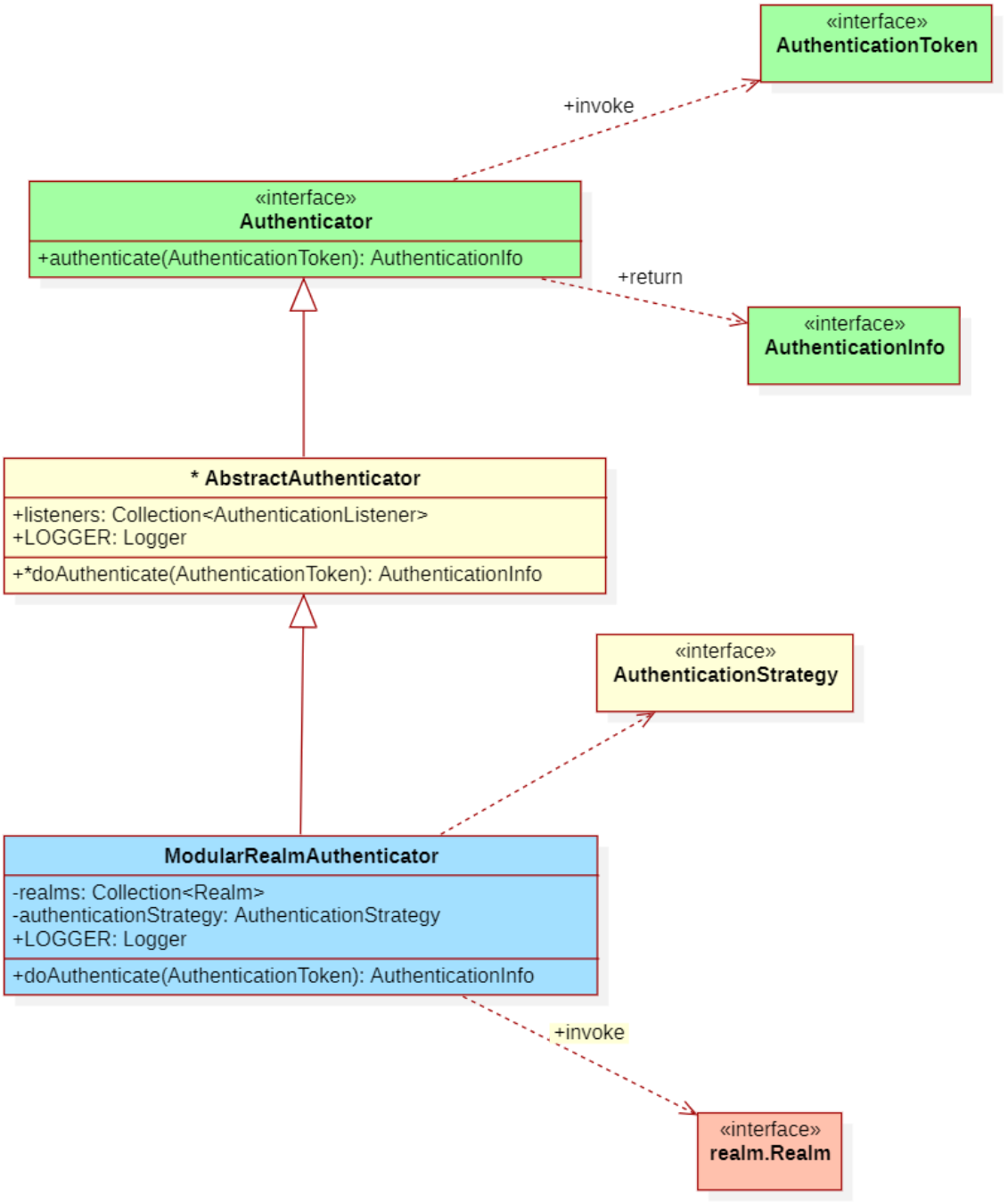
- 工厂模式

  RealmFatory

- 委托：将数据查询委托给Realm

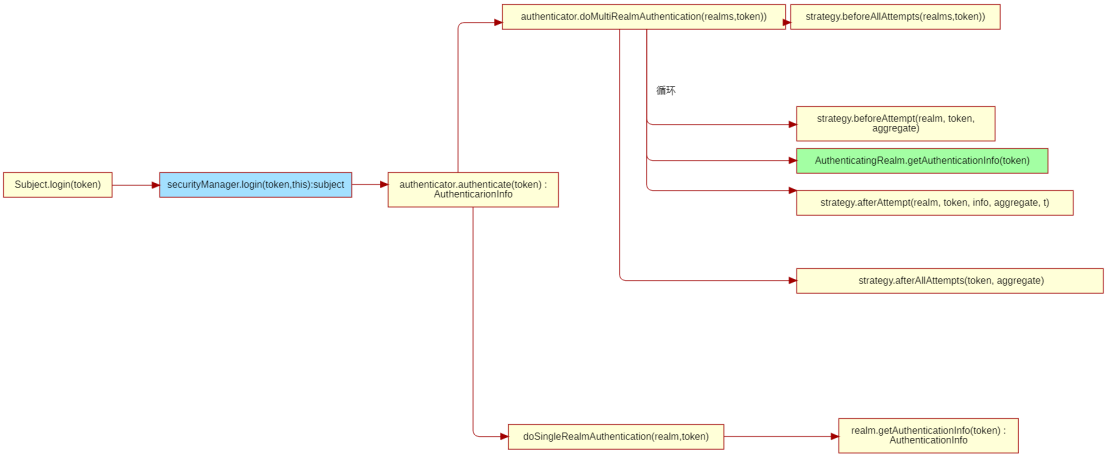
实现

- 数据结构/类图



- **验证流程**（工作流程）

> authenticator : ModularRealmAuthenticator



ModularRealmAuthenticator::

```java
protected AuthenticationInfo doMultiRealmAuthentication(Collection<Realm> realms, AuthenticationToken token) {  1 usage

    AuthenticationStrategy strategy = getAuthenticationStrategy();

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

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

    for (Realm realm : realms) {

        try {
            aggregate = strategy.beforeAttempt(realm, token, aggregate);
        } catch (ShortCircuitIterationException shortCircuitSignal) {
            // Break from continuing with subsequent realms on receiving
            // short circuit signal from strategy
            break;
        }

        if (realm.supports(token)) {

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

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

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

    aggregate = strategy.afterAllAttempts(token, aggregate);

    return aggregate;
}
```

很明显，采用了策略模式（具体策略分类由pam包实现），同时采用了 **静态的AOP** (即将advice()以源码的形式织入)

Authorizer

> Apache Shiro Authorization | Apache Shiro

Authorization Elements

- Permision = Resource + Actions
- Role 是权限的集合
- User

> As we've covered previously however, the `Subject` is really Shiro's 'User' concept.

设计模式

- AOP　:org.apache.shiro.aop.MethodInterceptor　--->对应authz.aop包

> **A MethodInterceptor intercepts a MethodInvocation to perform before or after logic (aka 'advice').**
> Shiro's implementations of this interface mostly have to deal with ensuring a current Subject has the ability to execute the method before allowing it to continue.

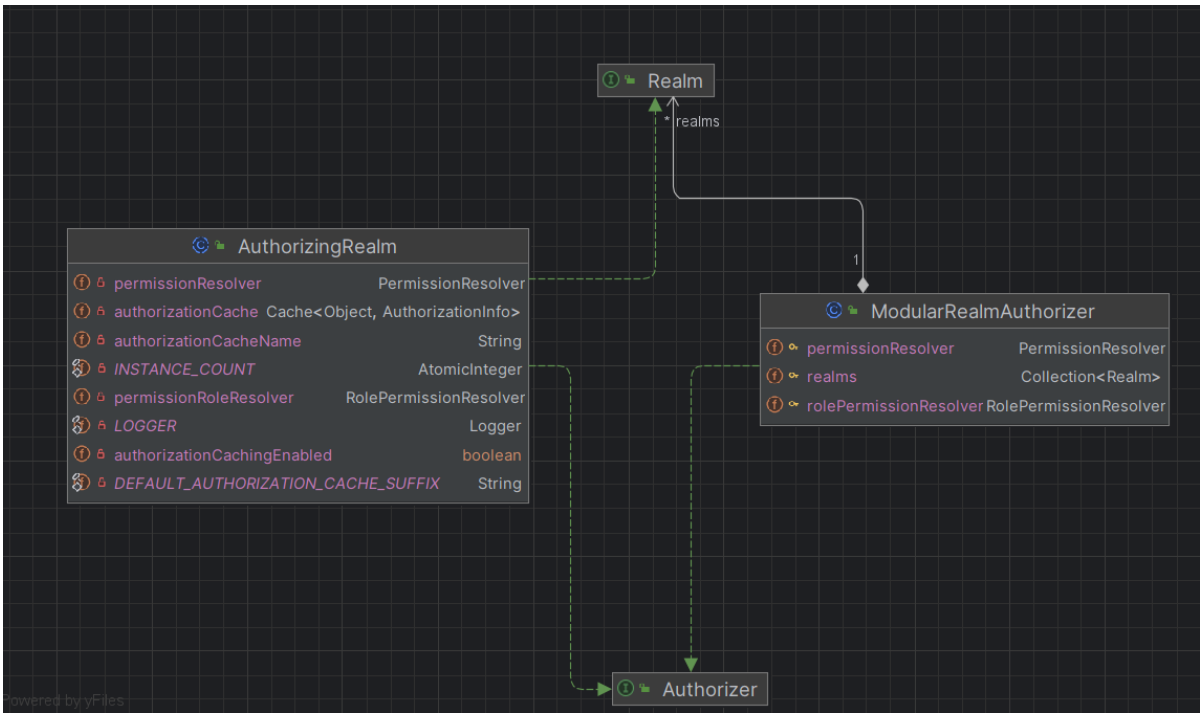
```
// org.apache.shiro.authz.aop.AuthorizingAnnotationMethodInterceptor
public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        assertAuthorized(methodInvocation);
        return methodInvocation.proceed();
    }
```

- 委托：将数据查询委托给Realm

实现

数据结构/类图

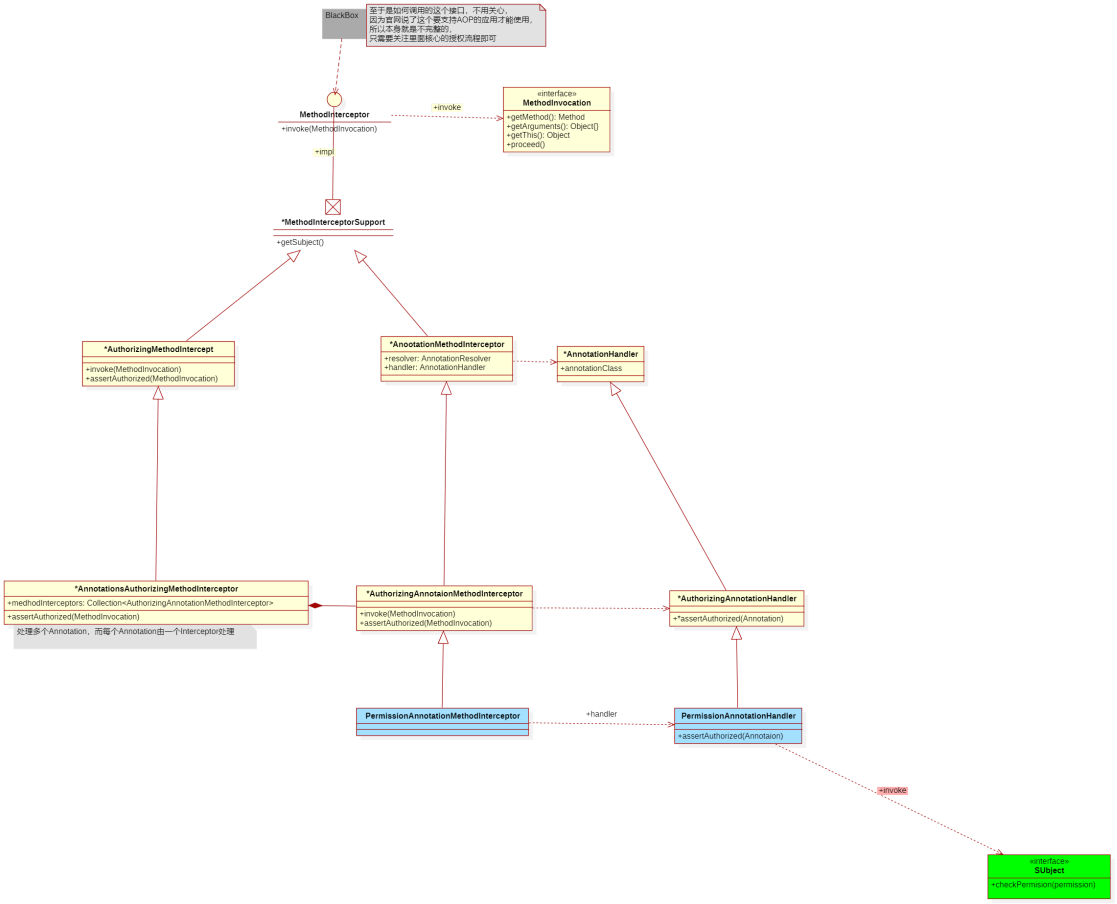
主要部分：



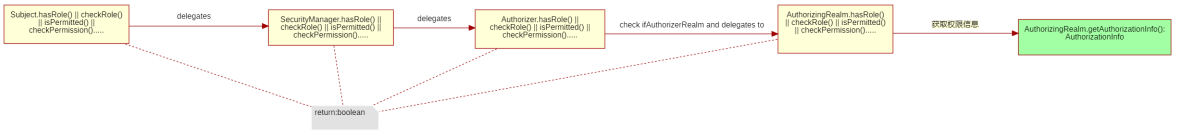
> 可以看出 委托模式和 代理模式（jdk) 在代码形式上是一致的，因为委托模式可以看作“没有任何代理逻辑的代理模式”

- aop部分 : 解决**以注解为基础的** 访问控制，算是扩展部分

> Before you can use Java annotations, you’ll need to enable AOP support in your application. There are a number of different AOP frameworks so, unfortunately, there is no standard way to enable AOP in an application.



授权流程（工作流程）



主要采用委托的方式：

```
//ModularRealmAuthorizer::


    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;
    }


    public boolean isPermitted(PrincipalCollection principals, Permission
permission) {
        assertRealmsConfigured();
        for (Realm realm : getRealms()) {
            if (!(realm instanceof Authorizer)) {
                continue;
            }
            if (((Authorizer) realm).isPermitted(principals, permission)) {
                return true;
            }
        }
        return false;
    }



//AuthorizingRealm::

public boolean isPermitted(PrincipalCollection principals, Permission permission)
{
        AuthorizationInfo info = getAuthorizationInfo(principals);
        return isPermitted(permission, info);
    }

    protected boolean isPermitted(Permission permission, AuthorizationInfo info)
{
        Collection<Permission> perms = getPermissions(info);
```

```
        if (perms != null && !perms.isEmpty()) {
            for (Permission perm : perms) {
                if (perm.implies(permission)) {//表示perm 包含 permission
                    return true;
                }
            }
        }
        return false;
    }
```

SessionManager

[Session Management | Apache Shiro]

> Like other core architectural components in Shiro, the `SessionManager` is a top-level component maintained by the `SecurityManager`

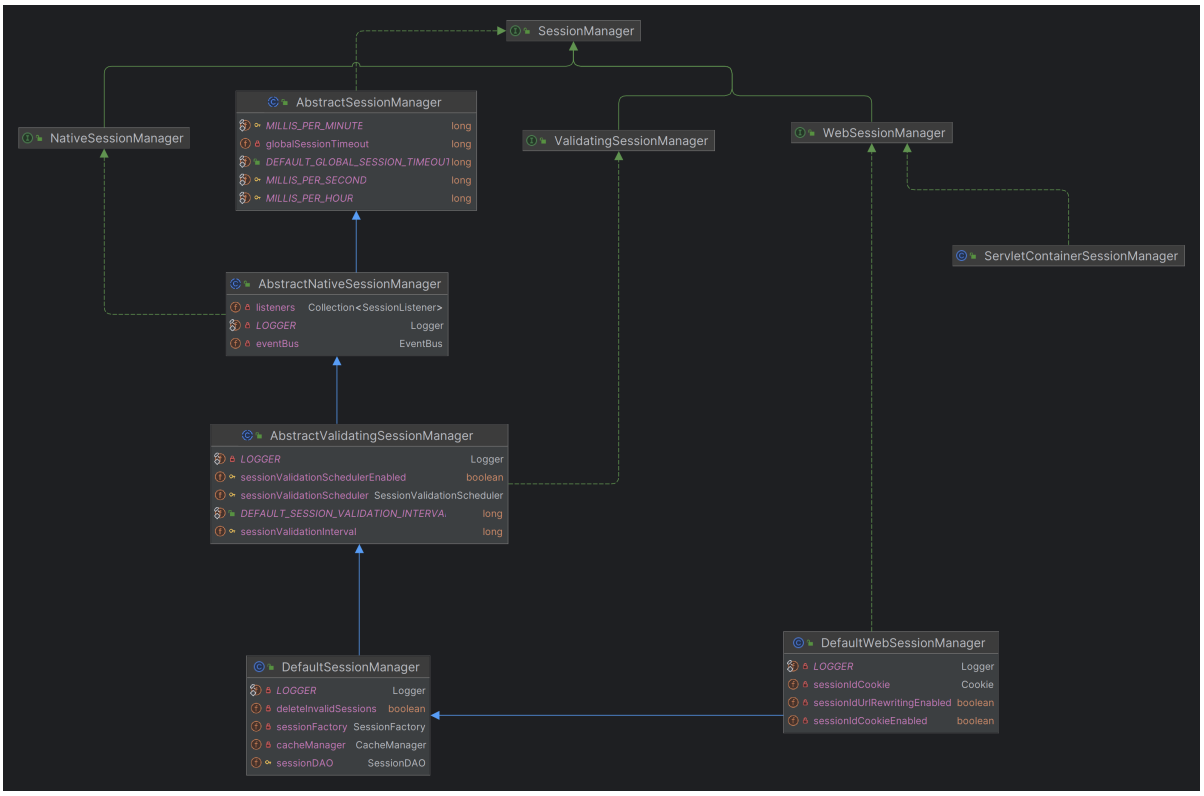
设计模式

1. 监听
2. DAO&委托

> `SessionManager` implementations delegate these Create/Read/Update/Delete (CRUD) operations to an internal component, the `SessionDAO`, which reflects the Data Access Object (DAO) design pattern.

实现

数据结构/类图



SessionDAO默认实现是用内存，由shiro实现，其他方式需要**由Application自己实现**

> shiro-web的WebsessionManager 的两个实现分别对应native（用shiro自己实现的会话管理） 和 serveltcontainer（调用ServletContainer实现的会话管理

```
//ServletContainerSessionManager::

public Session getSession(SessionKey key) throws SessionException {
     if (!WebUtils.isHttp(key)) {
         String msg = "SessionKey must be an HTTP compatible
implementation.";
         throw new IllegalArgumentException(msg);
     }

     HttpServletRequest request = WebUtils.getHttpRequest(key);

     Session session = null;

    //调用servlet container实现的接口
     HttpSession httpSession = request.getSession(false);

     if (httpSession != null) {
         session = createSession(httpSession, request.getRemoteHost());
     }

     return session;
 }
```

工作流程

在subject创建部分已给出

Realms

负责获取验证信息，访问控制（权限检查），shiro提供了一些简单实现，但**通常由Application根据业务实现**

[Apache Shiro Realms | Apache Shiro]

shiro-web

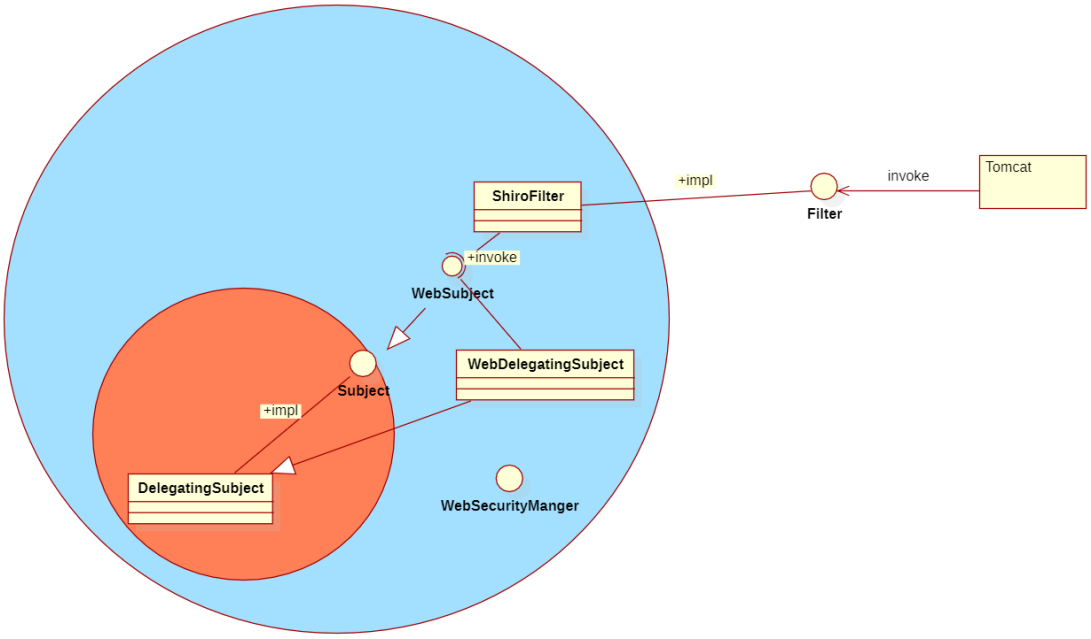
- shiro-web是shiro-core的**wrapper**：

> 其特征都是 ”实现一方，调用一方“ 注：调用不一定要显式的调用（代码上的），隐式调用也算，比如子类对于没有重写父类的方法，其字节码的函数引用与父类的一致，所以也可算作子类调用了父类的函数

比如shiro-web实现了Filter接口，从而使得Tomcat可以调用，同时ShiroFilter,又调用了WebSubject,

从抽象的接口的意义上讲，Shiro***Filter调用WebSubject相关验证和授权接口，但这些接口是继承自Subject的，所以会调用Subject接口，从而调用了shiro-core

> 如图：红色代表shiro-core，蓝色是shiro-web



> 从具体实现的意义过程来讲，当然是类调用类，但这是实现，不是"概念"，从抽象意义讨论调用过程比较契合概念.
>
> 同时由于其是软件级别的wrapper，shiro-core与其wrapper(shiro-web） 不是完全单向的调用关系,某些情况shiro-core会调用wrapper中的实现，比如SessionManager的两种实现

- 同时从上图可以看出其符合架构（Architecture）中的**分层结构（Layered）**
- **session管理冲突：**

  Session管理的使用，servlet容器（Tomcat) 已经实现了 的（Tomcat的Session管理(一) - coldridgeValley - 博客园 (cnblogs.com)），所以到底是使用 shrio提供的SessionManager,还是Tomcat已经实现的 是一个需要考虑的选择，shiro当然考虑到了这点

  > Apache Shiro Web Support | Apache Shiro| session_management

- subject 不在是"用户"级别 更确切的讲是"请求"级别，因为Servlet Container是以请求为单位进行处理，所以对它来说"主体" 是 请求
- 具体适配方式以后展开

Reference

什么是计算机软件设计中的 wrapper 技术 - 知乎 (zhihu.com)

Apache Shiro Reference Documentation | Apache Shiro

Tomcat的Session管理(一) - coldridgeValley - 博客园 (cnblogs.com)