Apache Shiro Reference Documentation

Introduction to Apache Shiro

What is Apache Shiro?

Apache Shiro 是一个强大而灵活的开源安全框架，它干净利落地处理身份认证，授权，企业会话管理和加密。

Apache Shiro 的首要目标是易于使用和理解。安全有时候是很复杂的，甚至是痛苦的，但它没有必要这样。框架应该尽可能掩盖复杂的地方，露出一个干净而直观的 API，来简化开发人员在使他们的应用程序安全上的努力。

以下是你可以用 Apache Shiro 所做的事情：

- 验证用户来核实他们的身份
- 对用户执行访问控制，如：
  - 判断用户是否被分配了一个确定的安全角色
  - 判断用户是否被允许做某事
- 在任何环境下使用 Session API，即使没有 Web 或 EJB 容器。
- 在身份验证，访问控制期间或在会话的生命周期，对事件作出反应。
- 聚集一个或多个用户安全数据的数据源，并作为一个单一的复合用户“视图”。
- 启用单点登录（SSO）功能。
- 为没有关联到登录的用户启用"Remember Me"服务

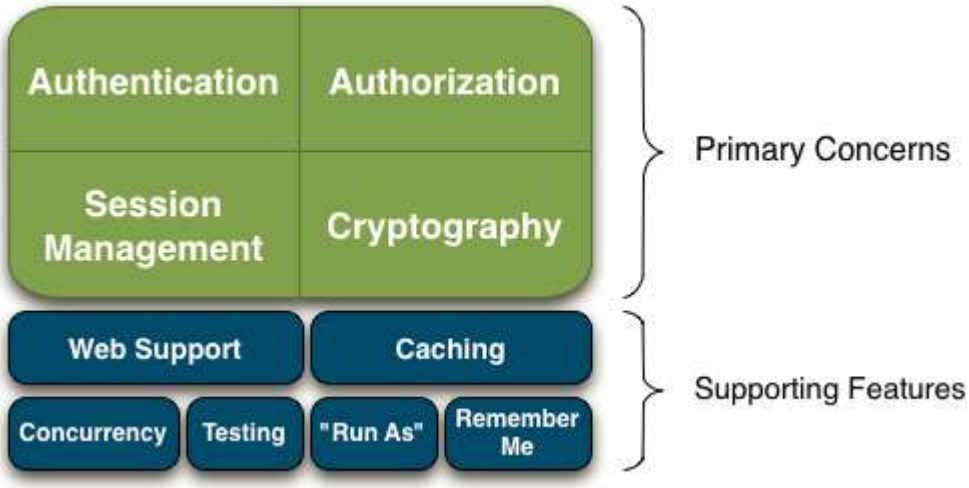
  …

  以及更多——全部集成到紧密结合的易于使用的 API 中。

Shiro 视图在所有应用程序环境下实现这些目标——从最简单的命令行应用程序到最大的企业应用，不强制依赖其他第三方框架，容器，或应用服务器。当然，该项目的目标是尽可能地融入到这些环境，但它能够在任何环境下立即可用。

Apache Shiro Features

Apache Shiro 是一个拥有许多功能的综合性的程序安全框架。下面的图表展示了 Shiro 的重点，并且这个参考手册也会与之类似的被组织起来：



Shiro 把 Shiro 开发团队称为“应用程序的四大基石”——身份验证，授权，会话管理和加密作为其目标。

- **Authentication**：有时也简称为“登录”，这是一个证明用户是他们所说的他们是谁的行为。
- **Authorization**：访问控制的过程，也就是绝对“谁”去访问“什么”。
- **Session Management**：管理用户特定的会话，即使在非 Web 或 EJB 应用程序。
- **Cryptography**：通过使用加密算法保持数据安全同时易于使用。

也提供了额外的功能来支持和加强在不同环境下所关注的方面，尤其是以下这些：

- Web Support：Shiro 的 web 支持的 API 能够轻松地帮助保护 Web 应用程序。
- Caching：缓存是 Apache Shiro 中的第一层公民，来确保安全操作快速而又高效。
- Concurrency：Apache Shiro 利用它的并发特性来支持多线程应用程序。
- Testing：测试支持的存在来帮助你编写单元测试和集成测试，并确保你的能够如预期的一样安全。
- "Run As"：一个允许用户假设为另一个用户身份（如果允许）的功能，有时候在管理脚本很有用。
- "Remember Me"：在会话中记住用户的身份，所以他们只需要在强制时候登录。

Apache Shiro Tutorial

Your First Apache Shiro Application

如果你从未使用过 Apache Shiro，这个简短的教程将会向您展示如何建立一个由 Apache Shiro 担保的初始的及非常简单的应用程序。一路上我们将讨论 Shiro 的核心概念来帮助你熟悉 Shiro 的设计和 API。

当你遵循本教程时，如果你确实不想编辑文件，你可以得到一个几乎相同的实例应用程序并按照你的意愿引用它。选择一个位置：

- 在 Apache Shiro 的版本控制库：https://svn.apache.org/repos/asf/shiro/trunk/samples/quickstart
- 在 Apache Shiro 的源代码的 samples/quickstart 目录。该源代码在 Download 页面提供下载。

Setup

在这个简单的示例中，我们将创建一个非常简单的命令行应用程序，它将会运行并迅速退出，这样你能够获得对 Shiro 的 API 的感受。

Any Application

Apache Shiro 从开始的那天起就被设计成能够支持任何应用程序——从最小的命令行应用程序到最大的群集 Web 应用程序。即使我们为该教程创建的是一个简单的应用，了解相同的使用模式适用于无论你的应用程序是怎样创建的及它被部署到哪里。

该教程需要 Java 1.5 及更高本。我们也使用 Apache Maven 作为我们的构建工具，但当然这不是使用 Apache Shiro 所必需的。你可以获取 Shiro 的 jar 包并按你喜欢的方式合并到你的应用程序，例如可能是一 Apache Ant 和 Ivy。

对于本教程，请确保你正在使用 Maven 2.2.1 或更高版本。你应该能够键入 mvn -version 命令行提示符，并看到与下面类似的东西：

Testing Maven Installation

```
hazlewood:~/shiro-tutorial$ mvn --version
Apache Maven 2.2.1 (r801777; 2009-08-06 12:16:01-0700)
Java version: 1.6.0_24
Java home: /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x" version: "10.6.7" arch: "x86_64" Family: "mac"
```

现在，在你的文件系统上创建一个新的目录，例如，shiro-tutorial 并在该目录下保存下面的 Maven pom.xml 文件：

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</moderVersion>
    <groupId>org.apache.shiro.tutorials</groupId>
    <artifactId>shiro-tutorial</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>First Apache Shiro Application</name>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                    <encoding>${project.build.sourceEncoding}</encoding>
                </configuration>
            </plugin>

            <!-- This plugin is only to test run our little application. It is not
                 needed in most Shiro-enabled applications: -->
            <plugin>
```

```
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>exec-maven-plugin</artifactId>
                        <version>1.1</version>
                        <executions>
                                <execution>
                                        <goals>
                                                <goal>java</goal>
                                        </goals>
                                </execution>
                        </executions>
                        <configuration>
                                <classpathScope>test</classpathScope>
                                <mainClass>Tutorial</mainClass>
                        </configuration>
                </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
                <groupId>org.apache.shiro</groupId>
                <artifactId>shiro-core</artifactId>
                <version>1.1.0</version>
        </dependency>
        <!-- Shiro use SLF4J for logging. We'll use the 'simple' binding
                in this example app. See http://ww.slf4j.org for more info. -->
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-simple</artifactId>
                <version>1.6.1</version>
                <scope>test</scope>
        </dependency>
    <dependencies>

</project>
```

The Tutorial class

我们将运行一个简单的命令行应用程序，因此，我们需要创建一个拥有 public static void main(String[] args)方法的 Java 类。

在包含你 pom.xml 文件的同样目录下，创建 src/main/java 子目录。在 src/main/java 目录下创建具有下面内容的 Tutorial.java 文件：

src/main/java/Tutorial.java

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.*;
import org.apache.shiro.config.IniSecurityManagerFactory;
```

```
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.session.Session;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Tutorial {
        private static final transient Logger log = LoggerFacutory.getLogger(Tutorial.class);

        public static void main(String[] args) {
                log.info("My First Apache Shiro Application");
                System.exit(0);
        }
}
```

现在不必担心 import 语句——我们不久将会涉及到它们。但现在，我们得有一个命令行应用程序“外形”。该程序所能够做到全部事情是打印出文本"My First Apache Shiro Application"并退出。

Test Run

要试用我们的教程应用程序，请在你的教程项目的根目录下执行以下的命令提示符，并键入以下内容：

```
mvn compile exec:java
```

然后你将看到我们的 little 教程“程序”运行并退出。你应该会看到与下面相似的一些东西（注意粗体文本，它显示了我们的输出）：

Run the Application

```
lhazlewood:~/projects/shiro-tutorial$ mvn compile exec:java

... a bunch of Maven output ...
1 [Tutorial.main()] INFO Tutorial - My First Apache Shiro Application
lhazlewood:~/projects/shiro-tutorial\$
```

我们已经验证了该程序运行成功——现在让我们启用 Apache Shiro。当我们继续本教程的时候，你可以在每次我们添加一些代码后运行 mvn compile exec:java 来观察我们变化后的结果。

Enable Shiro

在应用程序中启用 Shiro 最先要明白的事情是几乎在 Shiro 中的每个东西都与一个名为 SecurityManager 的主要的/核心的组件有关。对于那些熟悉 Java 安全的人来说，这是 Shiro 的 SecurityManager 概念——它不等同于 java.lang.SecurityManager。

虽然我们将在 Architecture 章节详细的涉及到 Shiro 的设计，但现在了解 Shiro 的 SecurityManager 是应用程序的 Shiro 环境的核心及每个应用程序中必须存在一个 SecurityManager 是很有益处的。因此，在我们的教程应用程序中第一件要做的事情就是配置 SecurityManager 实例。

Configuration

虽然我们能够直接实例化一个 SecurityManager 类，但 Shiro 的 SecurityManager 实现有足够的配置选项及内置组件使得在 Java 源代码做这件事情变得较为痛苦——如果使用一个灵活的基于文本的配置格式来配置 SecurityManager，那么这将是一件很容易的事情。

为此，Shiro 通过基于文本的 INI 配置文件提供了一个默认的"共性（common denominator）"解决方案。近来人们已经相当厌倦了使用笨重的 XML 文件，且 INI 文件易于阅读，使用简单，依赖性低。你稍后将会看到有了对象导航图的简单理解，INI 文件能够有效地被用来配置简单的对象图，如 SecurityManager。

Many Configuration Options
Shiro 的 SecurityManager 实现及所有支持组件都是兼容 JavaBean 的。这允许 Shiro 能够与几乎任何配置格式如 XML（Spring，JBoss，Guice 等等），YAML，JSON，Groovy Builder markup，以及更多配置被一起配置。INI 文件只是 Shiro 的“共性”格式，他它允许任何环境下的配置，除非其他选项不可用。

shiro.ini
因此，我们将为这个简单的应用程序使用 INI 文件来配置 Shiro SecurityManager。首先，在 pom.xml 所在的同一目录下创建 src/main/resources 目录。然后在新目录下创建包含以下内容的 shiro.ini 文件：

src/main/resources/shiro.ini

```
# =============================================================================
# Tutorial INI configuration
#
# Usernames/passwords are based on the classic Mel Brooks' film "Spaceballs" :)
# =============================================================================

# -----------------------------------------------------------------------------
# Users and their (optional) assigned roles
# username = password, role1, role2, ..., roleN
# -----------------------------------------------------------------------------
[users]
root = secret, admin
guest = guest, guest
presidentskroob = 12345, president
darkhelmet = ludicrousspeed, darklord, schwartz
lonestarr = vespa, goodguy, schwartz

# -----------------------------------------------------------------------------
# Roles with assigned permissions
# roleName = perm1, perm2, ..., permN
# -----------------------------------------------------------------------------
[roles]
admin = *
schwartz = lightsaber:*
goodguy = winnebago:drive:eagle5
```

如你所见，这个配置基本上建立了一小组静态用户帐户，对于我们的第一个应用程序已经足够了。在后面的章节中，你将看到我们如何使用更复杂的用户数据源，如关系数据库，LDAP 的 AcitveDirectory，以及更多。

Referencing the Configuration
现在我们已经定义好了一个 INI 文件，我们可以在我们的教程应用程序类中创建 SecurityManager 实例了。改变 main 方法来反映以下的更新内容：

```
public static void main(String[] args) {

    log.info("My First Apache Shiro Application");

    //1.
    Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini");

    //2.
    SecurityManager securityManager = factory.getInstance();

    //3.
    SecurityUtils.setSecurityManager(securityManager);

    System.exit(0);
}
```

好了，在仅仅添加了 3 行代码后，Shiro 就在我们的简单应用程序中启用了！很容易是吧？

轻松地运行 mvn compile exec:java，并看到这一切仍然运行成功（由于 Shiro 的默认调试日志或更低版本，你将不会看到任何的 Shiro 日志消息——如果在启动和运行没有报错，那么你知道一切仍然正常）。

这里是上面增加的代码所做的：

1. 我们使用 Shiro 的 IniSecurityManager 实现来提取我们的 shiro.ini 文件，它位于 classpath 的根目录。该实现反映了 Shiro 对工厂设计模式的支持。classpath: 前缀是一个资源定位符，用来告诉 shiro 去哪加载 ini 文件（其他前缀，如 url:和 file:也同样被支持）。
2. factory.getInstance()方法被调用，它来解析 INI 文件并返回反映该配置的 SecurityManager 实例。
3. 在这个简单的例子中，我们把 SecurityManager 设置为一个静态的（memory）单例，能够跨 JVM 访问。但请注意，这是不可取的，如果你在单个的 JVM 只中会有不只一个启用 Shiro 的应用程序。对于这个简单的例子而言，这是没有问题的，但更为复杂的应用程序环境通常将 SecurityManager 置于应用程序特定的存储中（如在 Web 应用中的 ServletContext 或 Spring，Guice 后 JBoss DI 容器实例）。

Using Shiro

现在我们的 SecurityManager 已经设置好并可以使用了，现在我们能够开始做一些我们真正关心的事情——执行安全操作。

当保护我们的应用程序时，我们对自己可能提出的最为相关的问题是“当前用户是谁”或“当前用户是否被允许做 XXX”。当我们编写代码或设计用户接口时，问这些问题是很常见的：应用程序通常是基于用户的背景情况建立的，且你想基于每个用户标准体现（保障）功能。因此，对于我们考虑应用程序安全的最自然的方式是基于当前用户。Shiro 的 API 使用它的 Subject 概念从根本上代表了“当前用户”的概念。

几乎在所有的环境中，你可以通过下面的调用获取当前正在执行的用户：

```
Subject currentUser = SecurityUtils.getSubject();
```

使用 SecurityUtils.getSubject()，我们可以获得当前正在执行的 Subject。Subject 是一个安全术语，它基本上的意思是“当前正在执行的用户的特定的安全视图”。它并没有被称为"User"是因为"User"一词通常和人类相关联。在安全界，术语"Subject"可以表示为人类，而且可是第三方进程，cron job，daemon account，或其他类似的东西。它仅仅意味着“该事物目前正与软件交互”。对于大多数的意图和目的，你可以把 Subject 看成是 Shiro 的"User"概念。

getSubject()在一个独立的应用程序中调用，可以返回一个在应用程序特定位置的基于用户数据的 Subject，并且在服务器环境中（例如，Web 应用程序），它获取的 Subject 是基于关联了当前线程或传入请求的用户数据的。

现在你拥有了一个 Subject，你能拿它来做什么？

如果你想在应用程序的当前会话中使事物对于用户可用，你可以获得他们的会话：

```
Session session = currentUser.getSession();
session.setAttribute( "someKey", "aValue" );
```

Session 是一个 Shiro 的特定实例，它提供了大部分你经常与 HttpSessoins 使用的东西，除了一些额外的好处以及一个巨大的区别：它不需要一个 HTTP 环境！

如果在一个 Web 应用程序内部部署，默认的 Session 将会是基于 HttpSession 的。但，在一个非 Web 环境中，像这个简单的教程应用程序，Shiro 将会默认自动地使用它的 Enterprise Session Management。这意味着你会使用相同的 API 在你的应用程序，在任何层，不论部署环境！这开辟了应用程序的新世界，由于任何需要会话的应用程序不必再被强制使用 HttpSession 或 EJB Stateful Session Beans。并且，任何客户端技术现在能够共享会话数据。

因此，现在你能获取一个 Subject 以及他们的 Session。如果他们被允许做某些事，如对角色和权限的检查，像“检查”真正有用的地方在哪呢？

嗯，我们只能为一个已知的用户做这些检查。我们上面的 Subject 实例代表了当前用户，但谁又是当前用户？呃，他们是匿名的——也就是说，直到直到他们至少登录一次。那么，让我像下面这样做：

```
if ( !currentUser.isAuthenticated() ) {
    //collect user principals and credentials in a gui specific manner
    //such as username/password html form, X509 certificate, OpenID, etc.
    //We'll use the username/password example here since it is the most common.
    UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");

    //this is all you have to do to support 'remember me' (no config - built in!):
    token.setRememberMe(true);

    currentUser.login(token);
}
```

这就是了！它再简单不过了。

但如果他们的登录尝试失败了会怎样？你能够捕获各种具体的异常来告诉你到底发生了什么，并允许你去处理并作出相应反应：

```
try {
    currentUser.login( token );
    //if no exception, that's it, we're done!
} catch ( UnknownAccountException uae ) {
    //username wasn't in the system, show them an error message?
} catch ( IncorrectCredentialsException ice ) {
    //password didn't match, try again?
} catch ( LockedAccountException lae ) {
    //account for that username is locked - can't login.  Show them a message?
}
    ... more types exceptions to check if you want ...
} catch ( AuthenticationException ae ) {
    //unexpected condition - error?
}
```

你能够检查到许多不同类型的异常，或抛出你自己的自定义条件的异常——Shiro 可能不提供的。请参见 AuthenticationException JavaDoc 获取更多。

Handy Hint
最安全的做法是给普通的登录失败消息给用户，因为你当然不想帮助试图闯入你系统的攻击者。

好了，到现在为止，我们已经有了一个登录用户。我们还能做些什么？

比方说，他们是是谁：

```
//print their identifying principal (in this case, a username):
log.info( "User [" + currentUser.getPrincipal() + "] logged in successfully." );
```

我们也可以测试他们是否有特定的角色：

```
if ( currentUser.hasRole( "schwartz" ) ) {
    log.info("May the Schwartz be with you!" );
} else {
    log.info( "Hello, mere mortal." );
}
```

我们还可以判断他们是否有权限在一个确定类型的实体上进行操作：

```
if ( currentUser.isPermitted( "lightsaber:weild" ) ) {
    log.info("You may use a lightsaber ring.  Use it wisely.");
} else {
    log.info("Sorry, lightsaber rings are for schwartz masters only.");
}
```

当然，我们可以执行极其强大的实例级权限检查——判断用户是否有能力访问某一类型的特定实例的能力：

```
if ( currentUser.isPermitted( "winnebago:drive:eagle5" ) ) {
    log.info("You are permitted to 'drive' the 'winnebago' with license plate (id) 'eagle5'.
             "Here are the keys - have fun!");
} else {
    log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
}
```

小菜一碟，对吧？

最后，当用户完成了对应用程序的使用，他们可以注销：

```
currentUser.logout(); //removes all identifying information and invalidates their session too.
```

Final Tutorial class
在添加上面的示例代码后，下面是我们的最终 Tutorial 类文件。请随意编辑和操作它，并按你喜欢的方式改变安全检查（以及 INI 配置）：

Final src/main/java/Tutorial.java

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.*;
```

```
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.session.Session;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Tutorial {
	private static final transient Logger log = LoggerFactory.getLogger(Tutorial.class);

	public static void main(String[] args) {
		log.info("My First Apache Shiro Application");

		Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro:ini");
		SecurityManager securityManager = factory.getInstance();
		SecurityUtils.setSecurityManager(securityManager);

		//get the currently executing user:
		Subject currentUser = SecurityUtils.getSubject();

		//Do some stuff with a Session (no need for a web or EJB container!!!)
		Session session = currentUser.getSession();
		session.setAttribute("someKey", "aValue");
		String value = (String) session.getAttribute("someKey");
		if (value.equals("aValue")) {
			log.info("Retrieved the correct vlaue! [" + vlaue + "]");
		}

		//let's login the current user so we can check against roles and permissions:
		if (!currentUser.isAuthenticated()) {
			UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");
			token.setRememberMe(true);
			try {
				currentUser.login(token);
			} catch (UnknownAccountException uae) {
				log.info("There is no user with username of " + token.getPrincipal());
			} catch(IncorrectCredentialsException ice) {
				log.info("Password for account " + token.getPrincipal() + " was incorrect!");
			} catch(LockedAccountException lae) {
				log.info("The account for username " + token.getPrincipal() + " is locked. " +
					"Please contact your administrator to unlock it.");
			}
			// … catch more exceptions here (maybe custom ones specific to your application?
			catch (AuthenticationException ae) {
				//unexpected condition? error?
			}
		}
```

```
            //say who they are:
            //print their identifying principal (in this case, a username):
            log.info("User [" + currentUser.getPrincipal() + ]" logged in successfully.");

            //test a role:
            if (currentUser.hasRole("schwartz")) {
                log.info("May the Schwartz be with you!");
            } else {
                log.info("Hello, mere mortal.");
            }

            //test a typed permission (not instance-level)
            if (currentUser.isPermitted("lightsaber:weild")) {
                log.info("You may use a lightsaber ring. Use it wisely.");
            } else {
                log.info("Sorry, lightsaber rings are for schwartz masters only.")
            }

            //a (very powerful) Instance Level permission:
            if (currentUser.isPermitted("winnebago:drive:eagle5")) {
                log.info("You are permitted to 'drive' the winnebago with license plate (id) 'eagle5' .  " +
                        "Here are the keys - have fun!");
            } else {
                log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
            }

            //all done - log out!
            currentUser.logout();

            System.exit(0);
        }
}
```

Summary

希望此次推出的教程帮助您了解如何在一个基本的应用程序设置 Shiro 以及 Shiro 的主要设计理念，Subject 和 SecurityManager。

但这是一个相当简单的应用程序。你可能已经问过你自己，“如果我不想使用 INI 用户帐户，而是要连接到一个更复杂的用户数数据源，该怎么办呢？”。

要回答这个问题，需要对 Shiro 的架构和支持的配置机制有更深一些的理解。我们下面将涉及到 Shiro 的架构。

Apache Shiro Architecture

Apache Shiro 的设计目标是通过直观和易于使用来简化应用程序安全。Shiro 的核心设计体现了大多数人们是如何考虑应用程序安全的——在某些人（或某些事）与应用程序交互的背景下。

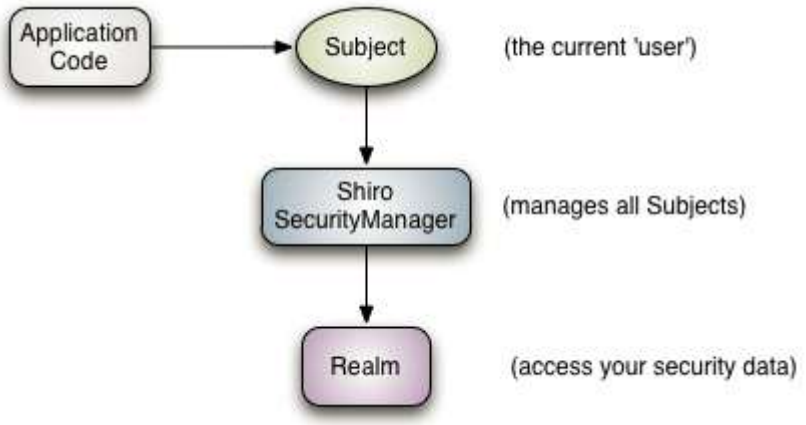
应用软件通常是基于用户背景情况设计的。也就是说，你将经常设计用户接口或服务 API，基于一个用户将要（或应该）如何与该软件交互。例如，你可能会说，“如果用户与我的应用程序交互的用户已经登录，我将显示一个他们能够点击的按钮来查看他们的帐户信息。如果他们没有登录，我将显示一个登录按钮。”

这个简单的陈述表明应用程序很大程度上的编写是为了满足用户的要求和需要。即使该“用户”是另一个软件系统而不是一个人类，你仍然得编写代码来响应行为，基于当前与你的软件进行交互的人或物。

Shiro 在它自己的设计中体现了这些概念。通过匹配那些对于软件开发人员来说已经很直观的东西，Apache Shiro 几乎在任何应用程序保持了直观和易用性。

High-Level Overview

在最高的概念层次，Shiro 的架构有 3 个主要的概念：Subject，SecurityManager 和 Realms。下面的关系图是关于这些组件是如何交互的高级概述，而且我们将会在下面讨论每一个概念：



- **Subject**：在我们的教程中已经提到，Subject 实质上是一个当前执行用户的特定的安全“视图”。鉴于"User"一词通常意味着一个人，而一个 Subject 可以是一个人，但它还可以代表第三方服务，daemon account，cron job，或其他类似的任何东西——基本上是当前正与软件进行交互的任何东西。

  所有 Subject 实例都被绑定到（且这是必须的）一个 SecurityManager 上。当你与一个 Subject 交互时，那些交互作用转化为与 SecurityManager 交互的特定 subject 的交互作用。

- **SecurityManager**：SecurityManager 是 Shiro 架构的心脏，并作为一种“保护伞”对象来协调内部的安全组件共同构成一个对象图。然而，一旦 SecurityManager 和它的内置对象图已经配置给一个应用程序，那么它单独留下来，且应用程序开发人员几乎使用他们所有的时间来处理 Subject API。

  我们稍后会更详细地讨论 SecurityManager，但重要的是要认识到，当你正与一个 Subject 进行交互时，实质上是幕后的 SecurityManager 处理所有繁重的 Subject 安全操作。这反映在上面的基本流程图。

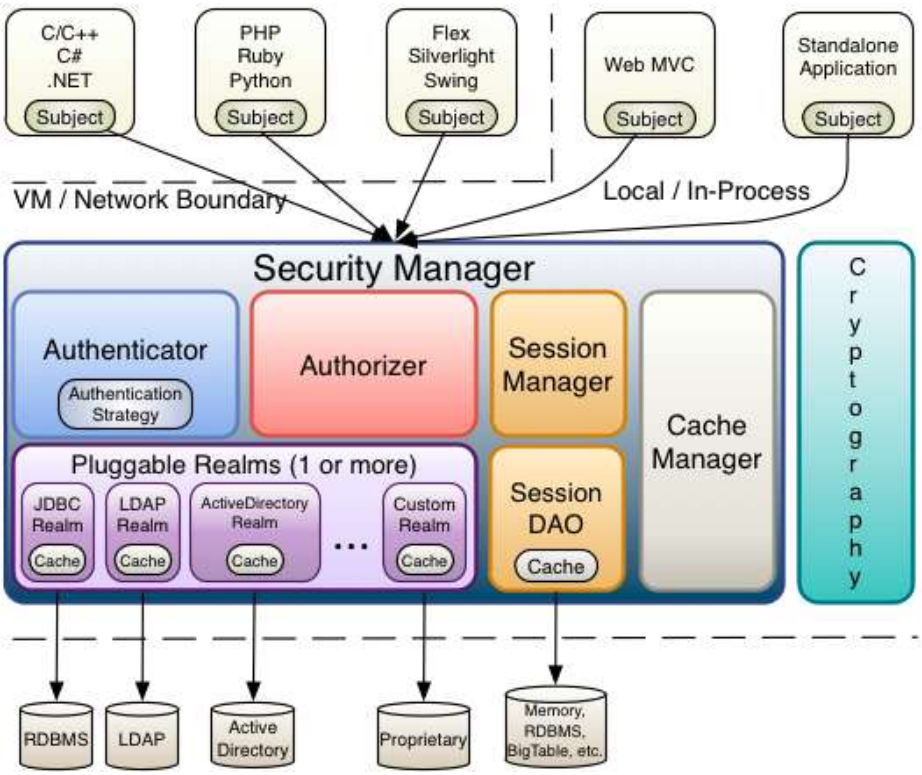
- **Realms**：Realms 担当 Shiro 和你的应用程序的安全数据之间的“桥梁”或“连接器”。当它实际上与安全相关的数据如用来执行身份验证（登录）及授权（访问控制）的用户帐户交互时，Shiro 从一个或多个为应用程序配置的 Realm 中寻找许多这样的东西。

  在这个意义上说，Realm 本质上是一个特定安全的 DAO：它封装了数据源的连接详细信息，使 Shiro 所需的相关的数据可用。当配置 Shiro 时，你必须指定至少一个 Realm 用来进行身份验证和/或授权。SecurityManager 可能配置多个 Realms，但至少有一个是必须的。

  Shiro 提供了立即可用的 Realms 来连接一些安全数据源（即目录），如 LDAP，关系数据库（JDBC），文本配置源，像 INI 及属性文件，以及更多。你可以插入你自己的 Realm 实现来代表自定义的数据源，如果默认地 Realm 不符合你的需求。

  像其他内置组件一样，Shiro SecurityManager 控制 Realms 是如何被用来获取安全和身份数据来代表 Subject 实例的。

  Detailed Architecture
  下图展示了 Shiro 的核心架构概念，紧跟其后的是每个的简短总结：



- **Subject**(org.apache.shiro.subject.Subject)
  当前与软件进行交互的实体（用户，第三方服务，cron job，等等）的安全特定“视图”。

- **SecurityManager**(org.apache.shiro.mgt.SecurityManager)
  如上所述，SecurityManager 是 Shiro 架构的心脏。它基本上是一个“保护伞”对象，协调其管理的组件以确保它们能够一起顺利的工作。它还管理每个应用程序用户的 Shiro 的视图，因此它知道如何执行每个用户的安全操作。

- **Authenticator**(org.apache.shiro.authc.Authenticator)

Authenticator 是一个对执行及对用户的身份验证（登录）尝试负责的组件。当一个用户尝试登录时，该逻辑被 Authenticator 执行。Authenticator 知道如何与一个或多个 Realm 协调来存储相关的用户/帐户信息。从这些 Realm 中获得的数据被用来验证用户的身份来保证用户确实是他们所说的他们是谁。

- **Authentication Strategy**(org.apache.shiro.authc.pam.AuthenticationStrategy)
  如果不止一个 Realm 被配置，则 AuthenticationStrategy 将会协调这些 Realm 来决定身份认证尝试成功或失败下的条件（例如，如果一个 Realm 成功，而其他的均失败，是否该尝试成功？ 是否所有的 Realm 必须成功？或只有第一个成功即可？）。

- **Authorizer**(org.apache.shiro.authz.Authorizer)
  Authorizer 是负责在应用程序中决定用户的访问控制的组件。它是一种最终判定用户是否被允许做某事的机制。与 Authenticator 相似，Authorizer 也知道如何协调多个后台数据源来访问角色恶化权限信息。Authorizer 使用该信息来准确地决定用户是否被允许执行给定的动作。

- **SessionManager**(org.apache.shiro.session.SessionManager)
  SessionManager 知道如何去创建及管理用户 Session 生命周期来为所有环境下的用户提供一个强健的 Session 体验。这在安全框架界是一个独有的特色——Shiro 拥有能够在任何环境下本地化管理用户 Session 的能力，即使没有可用的 Web/Servlet 或 EJB 容器，它将会使用它内置的企业级会话管理来提供同样的编程体验。SessionDAO 的存在允许任何数据源能够在持久会话中使用。

  - **SessionDAO**(org.apache.shiro.session.mgt.eis.SessionDAO)
    SesssionDAO 代表 SessionManager 执行 Session 持久化（CRUD）操作。这允许任何数据存储被插入到会话管理的基础之中。

- **CacheManager**(org.apahce.shiro.cache.CacheManager)
  CacheManager 创建并管理其他 Shiro 组件使用的 Cache 实例生命周期。因为 Shiro 能够访问许多后台数据源，由于身份验证，授权和会话管理，缓存在框架中一直是一流的架构功能，用来在同时使用这些数据源时提高性能。任何现代开源和/或企业的缓存产品能够被插入到 Shiro 来提供一个快速及高效的用户体验。

- **Cryptography**(org.apache.shiro.crypto.*)
  Cryptography 是对企业安全框架的一个很自然的补充。Shiro 的 crypto 包包含量易于使用和理解的 cryptographic Ciphers，Hasher（又名 digests）以及不同的编码器实现的代表。所有在这个包中的类都被精心地设计以易于使用和易于理解。任何使用 Java 的本地密码支持的人都知道它可以是一个难以驯服的具有挑战性的动物。Shiro 的 cryptoAPI 简化了复杂的 Java 机制，并使加密对于普通人也易于使用。

- **Realms**(org.apache.shiro.realm.Realm)
  如上所述，Realms 在 Shiro 和你的应用程序的安全数据之间担当“桥梁”或“连接器”。当它实际上与安全相关的数据如用来执行身份验证（登录）及授权（访问控制）的用户帐户交互时，Shiro 从一个或多个为应用程序配置的 Realm 中寻找许多这样的东西。你可以按你的需要配置多个 Realm（通常一个数据源一个 Realm），且 Shiro 将为身份验证和授权对它们进行必要的协调。

The SecurityManager

因为 Shiro 的 API 鼓励一个以 Subject 为中心的编程方式，大多数应用程序开发人员很少，如果真有，与 SecurityManager 直接进行交互（框架开发人员有时候会觉得它很有用）。即便如此，了解如何 SecurityManager 是如何工作的仍然是很重要的，尤其是在为应用程序配置一个 SecurityManager 的时候。

Design

如前所述，应用程序的 SecurityManager 执行安全操作并管理所有应用程序用户的状态。在 Shiro 的默认 SecurityManager 实现中，这包括：

- Authentication
- Authorization
- Session Management
- Cache Management
- Realm coordination
- Event propagation
- "Remember Me" Services
- Subject creation
- Logout
  以及更多。

但这是许多功能来尝试管理一个单一的组件。而且，使这些东西灵活而又可定制将会是非常困难的，如果一切都集中到一个单一的实现类。

为了简化配置并启用灵活配置/可插性，Shiro 的实现都是高度模块化设计——由于如此的模块化，SecurityManager 实现（以及它的类层次结构）并没有做很多事情。相反，SecurityManager 实现主要是作为一个轻量级的“容器”组件，委托计划所有的行为到嵌套/包裹的组件。这种“包装”的设计体现在上面的详细构架图。

虽然组件实际上执行逻辑，但 SecurityManager 实现知道何时以及如何协调组件来完成正确的行为。

SecurityManager 实现和组件都是兼容 JavaBean 的，它允许你（或某个配置机制）通过标准的 JavaBean 的 accessor/mutator 方法（get*/set*）轻松地自定义可拔插组件。这意味着 Shiro 的架构的组件性能够把自定义行为转化为非常容易的配置文件。

Easy Configuration
由于 JavaBeans 的兼容性，通过任何支持 JavaBean 风格的配置的机制可以很容易的用自定义组件配置 SecurityManager，如 Spring，Guice，JBoss，等等。

我们接下来将讨论 Configuration。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Apache Shiro Configuration

Shiro 被设计成能够在任何环境下工作，从最简单的命令行应用程序到最大的的企业群集应用。由于环境的多样性，使得许多配置机制适用于它的配置。本节仅讨论只被 Shiro 核心所支持的配置机制。

Many Configuration Options
Shiro 的 SecurityManager 实现及所支持的组件都是兼容 JavaBean 的。这使得 Shiro 几乎能使用任何配置格式，如 regular Java，XML(Spring, JBoss, Guice,等等)，YAML，JSON，Groovy Builder markup，以及更多的配置。

Programmatic Configuration

创建一个 SecurityManger 并把它提供给应用程序的绝对的最简单的方法是创建一个 org.apache.shiro.mgt.DefaultSecurityManager 并把它链到代码中。例如：

```
Realm realm = //instantiate or acquire a Realm instance.  We'll discuss Realms later.

SecurityManager securityManager = new DefaultSecurityManager(realm);

//Make the SecurityManager instance available to the entire application via static memory:
SecurityUtils.setSecurityManager(securityManager);
```

令人惊讶的是，仅在 3 行代码后，你马上就拥有了一个适合许多应用环境的功能全面的 Shiro 环境。这是多么的容易！？

SecurityManager Object Graph

正如在 Architecture 章节中讨论的一样，Shiro 的 SecurityManager 实现实质上是一个特定安全的嵌套组件中的模块化对象图。因为它们也是兼容 JavaBean 的，你可以调用任何嵌套组件的 getter 和 setter 方法来配置 SecurityManager 以及它的内部对象图。

例如，如果你想配置 SecurityManager 实例来使用自定义的 SessionDAO 来定制 Session Management，你可以通过嵌套的 SessionManager 的 setSessionDAO 方法直接设置 SessionDAO：

```
...
DefaultSecurityManager securityManager = new DefaultSecurityManager(realm);

SessionDAO sessionDAO = new CustomSessionDAO();

((DefaultSessionManager)securityManager.getSessionManager()).setSessionDAO(sessionDAO);
...
```

通过使用直接方法调用，你可以配置 SecurityManager 的对象图的任何部分。

但是，程式化定制过于简单，它并不能代表大多数现实世界的应用程序的理想配置。以下是程式化定制可能不适合你的几点原因：

- 它需要你了解和实例化一个直接实现。这将会更好，如果你不需要了解具体的实现和在哪里可以找到它们。
- 由于 Java 的类型安全性，你需要转换通过 get*方法获取的对象来得到它们具体的实现。如此多的转换是丑陋的，冗长的，并使你和实现类紧密连接起来。
- SecurityUtils.setSecurityManager 方法调用在一个 VM 静态单例中实例化 SecurityManager 实例，VM 静态单例在给许多应用程序带来好处的同时，也会引发一些问题，如果多个启用 Shiro 的应用程序在同一个 JVM 中运行。如果该实例是一个应用程序单例，而不是一个静态内存引用就再好不过了。
- 每当你想改变 Shiro 配置时，它需要你重新编译你的应用程序。

然而，即使有这么多警告，直接的编程操作方法在内存受限的环境中仍然是有价值的，如智能手机应用。若你的应用程序不在一个内存受限的环境下运行，你会发现基于文本的配置要更容易使用和阅读。

INI Configuration

大多数应用程序反而从基于文本的配置受益，能够独立地修改源代码，甚至让那些不熟悉 Shiro 的 API 的人能够更容易理解。

为了确保一个基于文本的通用标准配置机制能够在所有环境下以最小的第三方依赖工作，Shiro 支持 INI 格式来构建 SecurityManager 对象图及其支持的组件。INI 易于阅读，易于配置，且设置简单，适合大多数应用程序。

Creating a SecurityManager from INI

以下是两个关于如何基于 INI 配置构建 SecurityManager 的例子。

SecurityManager from an INI resource

我们能够从一个 INI 资源路径创建 SecurityManager 实例。我们可以从文件系统，classpath，或分别拥有前缀 file:，classpath:，或 url:的 URL 中获取资源。本例采用 Factory 提取 classpath 根目录下的 shiro.ini 文件并返回 SecurityManager 实例：

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini");
SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

SecurityManager from an INI instance

如果你需要，INI 配置也可以通过 org.apache.shiro.config.Ini 类使用编程方式创建。Ini 类的功能与 JDK java.util.Properties 类相似，但通过 section 名称，它同时还支持分割。

例如：

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.Ini;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Ini ini = new Ini();
//populate the Ini instance as necessary
...
Factory<SecurityManager> factory = new IniSecurityManagerFactory(ini);
SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

现在，我们知道如何从 INI 配置构建出一个 SecurityManager 了，我们看看到底是如何定义一个 Shiro INI 配置的。

INI Sections

INI 基本上是一个文本配置，包含了由唯一命名的 section 组织的键/值对。键只是每个 section 唯一，而不是在整个配置中（与 JDK 属性不同）。不过每个 section 都可以被看作单一的属性定义。

注释行能够以散列字符（# - 也就是"hash","pound"或"number"符号）或分号（";"）开始。

以下是 Shiro 能够理解的 section 例子：

```
# =======================
# Shiro INI configuration
# =======================

[main]
# Objects and their properties are defined here,
# Such as the securityManager, Realms and anything
# else needed to build the SecurityManager

[users]
# The 'users' section is for simple deployments
# when you only need a small number of statically-defined
# set of User accounts.

[roles]
# The 'roles' section is for simple deployments
# when you only need a small number of statically-defined
# roles.

[urls]
# The 'urls' section is used for url-based security
# in web applications.  We'll discuss this section in the
# Web documentation
```

[main]

[main] section 是你配置应用程序的 SecurityManager 实例及任何它的依赖组件（如 Realms）的地方。

配置对象实例，如 SecurityManager 或任何它的依赖组件，通过使用 INI 听起来是一件困难的事情，我们仅能使用名称/值对。但通过一点点约定和对对象图的理解，你将会发现你可以做很多事情。Shiro 使用这些条件来简单却相当简洁的配置机制。

我们往往喜欢把这种方法称为“穷人的”依赖注入，虽然没有完全成熟的 Spring/Guice/JBoss XML 文件强大，但你将会发现它能够完成许多事情却没有太多的复杂性。当然那些其他的配置机制也同样可用，但它们不是使用 Shiro 所必需的。

为了激起你的欲望，这里有一个有效的[main]配置的例子。我们将在下面讨论它的细节，但你可能会发现，仅凭直觉就能明白很多事情是怎么回事：

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher

myRealm = com.company.security.shiro.DatabaseRealm
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
myRealm.password = secret
myRealm.credentialsMatcher = $sha256Matcher

securityManager.sessionManager.globalSessionTimeout = 1800000
```

Defining an object

请思考下面[main] section 的摘要：

```
[main]
myRealm = com.company.shiro.realm.MyRealm
...
```

这一行实例化了 com.company.shiro.realm.MyRealm 类型的新的对象实例，并使该对象使用 myRealm 名称，以供进一步的引用和配置。

如果该实例化的对象实现了 org.apache.shiro.util.Nameable 接口，则 Nameable.setName 方法将会在拥有该名字（在本例中是 myRealm）的对象上调用。

Setting object properties

Primitive Values

简单的原始属性，可以通过使用等号来指定：

```
...
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
...
```

配置中的这些行转化为方法调用：

```
...
myRealm.setConnectionTimeout(30000);
myRealm.setUsername("jsmith");
...
```

这怎么可能呢？ 它假定所有的对象都是兼容 Java Bean 的 POJO。

在后台，当设置这些属性时，Shiro 默认使用 Apache Commons BeanUtils 来完成所有繁重的任务。所以，尽管 INI 值是文本的，BeanUtils 知道如何去转换字符串值到正确的原始类型，然后调用相应的 JavaBean 的 setter 方法。

Reference Values

如果你需要设置的值不是一个原始的，而是另一个对象呢？那么，你可以使用美元符号（$）来引用之前定义的实例。例如：

```
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
myRealm.credentialsMatcher = $sha256Matcher
...
```

这简单地定位通过名称 sha256Matcher 来定位定义好的对象，然后使用 BeanUtils 在 myRealm 实例上设置该对象（通过调用 myRealm.setCredentialsMatcher(sha256Matcher)方法）。

Nested Properties

在 INI 配置行的等号左侧使用点号，你可以遍历对象图来得到想设置的最终的对象/属性。例如，这个配置行：

```
...
securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

（通过 BeanUtils）转译成下面的逻辑：

```
securityManager.getSessionManager().setGlobalSessionTimeout(1800000);
```

对象图的遍历可以按需要来加大遍历深度：object.property1.property2….propertyN.value = blah

BeanUtils Property Support
任何 BeanUtils.setProperty 方法支持的属性赋值操作都会在 Shiro 的[main] section 中工作，包括 set/list/map 元素赋值。请参见 Apache Commons BeanUtils Website 及文档获取更多信息。

Byte Array Values
因为原始的字节数组本身不能使用文本格式，所以我们必须使用文本编码的字节数组。能够指定的值是一个 Base64 编码的字符串（默认），后一个 16 进制编码的字符串。默认是 Base64 是因为 Base64 编码只需较少的文本来表示值——它拥有一个较大的编码表，意味着你的 token 都是较短的（几个较好的文本配置）。

```
# The 'cipherKey' attribute is a byte array.    By default, text values
# for all byte array properties are expected to be Base64 encoded:

securityManager.rememberMeManager.cipherKey = kPH+bIxk5D2deZiIxcaaaA==
...
```

然而，如果你喜欢使用 16 进制编码，你必须在字符串 token 前加上 0x（"zero" "x"）前缀:

```
securityManager.rememberMeManager.cipherKey = 0x3707344A4093822299F31D008
```

Collection Properties
List，Set 和 Map 能够被设定任何属性——直接或作为一个嵌套属性。对于 Set 和 list 而言，只需指定一组由逗号分隔的值或对象的引用。

例如某些 SessionListener：

```
sessionListener1 = com.company.my.SessionListenerImplementation
...
sessionListener2 = com.company.my.other.SessionListenerImplementation
...
securityManager.sessionManager.sessionListeners = $sessionListener1, $sessionListener2
```

对于 Map，你指定一系列由逗号分隔的键-值对，每个键-值对通过冒号":"被限定：

```
object1 = com.company.some.Class
object2 = com.company.another.Class
...
anObject = some.class.with.a.Map.property

anObject.mapProperty = key1:$object1, key2:$object2
```

在上面的例子中，被$object1 引用的对象将属于在 map 中的字符串键值 key1，也就是，map.get("key1")返回 object1。你还可以依照键来使用其他对象：

```
anObject.map = $objectKey1:$objectValue1, $objectKey2:$objectValue2
...
```

Considerations

Order Matters

上面的 INI 格式和约定都非常便捷且易于理解，但它没有其他基于 text/XML 的配置机制强大。在使用上面的机制时最重要的问题是理解顺序问题！

Be Careful

每个对象的实例化以及每个值得分配都是按照它们在[main] section 中出现的顺序来执行的。这些配置行最终转化成一个 JavaBean 的 getter/setter 方法调用，因此，这些方法以同样的顺序被调用！

当你编写你的配置时，请记住这点。

Overriding Instances

任何对象能够被配置中后来新定义的实例覆盖。所以举例来说，第二个 myRealm 定义将会覆盖第一个：

```
...
myRealm = com.company.security.MyRealm
...
myRealm = com.company.security.DatabaseRealm
...
```

这将导致 myRealm 成为一个 com.company.security.DatebaseRealm 实例，且之前的实例将永不会被使用（同时被垃圾回收）。

Default SecurityManager

你可能已经注意到在上面的完整实例中，SecurityManager 实例的类并没有定义，我们仅在右边设定一个嵌套属性：

```
myRealm = ...

securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

这是因为 securityManager 实例是一个特殊的实例——它已经为你实例化并准备好使用，所以你不需要知道用来实例化的具体 SecurityManager 实例类。

当然，如果你确实想指定你自己的实例，你可以只定义你自己的实现，正如上面的"Overriding Instances"一节中所规定的：

```
...
securityManager = com.company.security.shiro.MyCustomSecurityManager
...
```

当然，这基本上没有必要——Shiro 的 SecurityManager 实现是可定制化的，且通常可以与任何必要的事物进行配置。如果你真想这么做的话，你得想问问你自己（或用户列表）。

[users]

[users] section 允许你定义一组静态的用户帐户。这在大部分拥有少数用户帐户或用户帐户不需要在运行时被动态地创建的环境下是很有用的。以下是一个例子：

```
[users]
admin = secret
lonestarr = vespa, goodguy, schwartz
darkhelmet = ludicrousspeed, badguy, schwartz
```

Automatic IniRealm

仅定义非空的[users]或[roles] section 将会自动地触发 org.apache.shiro.realm.text.IniRealm 实例的创建，并使它在[main] section 中可用且名为 iniRealm。你可以像上面所描述的其他对象一样来配置它。

Line Format

在[users] section 中的每行都必须保证是下面的格式：

username = password, roleName1, roleName2, …, roleNameN

- 在等号左边的值是用户名
- 在等号右边的第一个值是用户的秘密。密码是必须的。
- 任何在密码后用逗号分隔的值都是分配给该用户的角色名。角色名是可选的。

Encrypting Passwords

如果你不想[users] section 中密码是纯文本的，你可以使用你喜爱的散列算法（MD5，Sha1，Sha256，等等）来进行加密，并使用生产的字符串作为密码值。默认情况下，密码字符串是 16 进制编码，但可以使用 Base64 编码代替 16 进制编码来配置（见下面）。

一旦你指定了文本密码散列值，你得告诉 Shiro 这些都是加密的。你可以通过配置在[main] section 中隐式地创建 iniRealm 来使用合适的 CredentialsMatcher 实现来对应到你所指定的哈希算法：

```
[main]
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
iniRealm.credentialsMatcher = $sha256Matcher
...

[users]
# user1 = sha256-hashed-hex-encoded password, role1, role2, ...
user1 = 2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b, role1, role2, ...
```

你可以像任何其他对象一样在 CredentialsMatcher 上配置任何属性，以反映你哈希策略，例如，指定 salting 是否被使用或需要执行多少次哈希迭代。请参见 org.apache.shiro.authc.credential.HashedCredentialsMatcher 的 JavaDoc 来更好的理解哈希策略，如果它们对你可能有用的话。

例如，如果你的用户密码字符串是 Base64 编码而不是默认的 16 进制，你可以指定说明：

```
[main]
...
# true = hex, false = base64:
sha256Matcher.storedCredentialsHexEncoded = false
```

[roles]

[roles] section 允许你把定义在[users] section 中的角色与权限关联起来。另外，这在大部分拥有少数用户帐户或用户帐户不需要在运行时被动态地创建的环境下是很有用的。以下是一个例子：

```
[roles]
# 'admin' role has all permissions, indicated by the wildcard '*'
admin = *
# The 'schwartz' role can do anything (*) with any lightsaber:
schwartz = lightsaber:*
# The 'goodguy' role is allowed to 'drive' (action) the winnebago (type) with
# license plate 'eagle5' (instance specific id)
goodguy = winnebago:drive:eagle5
```

Line Format

在[roles] section 中每个配置行必须定义一个映射以下格式的角色到权限的键/值：

rolename = permissionDefinition1, permissionDefinition2, … , permissionDefinitionN

permissionDefinition 是一个任意的字符串，但大多数人将会使用符合 org.apache.shiro.authz.permission.WildcardPermission 格式的字符串，为了易用性和灵活性。请参见 Permissions 文档获取更多关于 Permissions 及你如何从它们受益的信息。

Internal commas

请注意，如果一个独立的 permissionDefinition 需要被内部逗号分隔（例如，printer:5thFloor:print,info），你需要用户双引号环绕该定义，以避免错误解析：

"printer:5thFloor:print,info"

Roles without Permissions

如果你有不需要权限关联的角色，你不需要在[roles] section 中间把他们列出来，如果你不想的话。只需定义在[user] section 中定义角色名就足以创建尚不存在的角色。

[urls]

该 section 及它的选项在 Web 章节被描述。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。

贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Authentication（身份验证）

Authentication 是指身份验证的过程——即证明一个用户实际上是不是他们所说的他们是谁。对于一个用户证明自己的身份来说，他们需要提供一些身份识别信息，以及某些你的系统能够理解和信任的身份证明。

这是通过提交用户的身份和凭证给 Shiro，以判断它们是否和应用程序预期的相匹配。

- **Principals**(身份)是 Subject 的‘identifying attributes(标识属性)’。Principals(身份)可以是任何能够证明 Subject 的东西，如名，姓氏，用户名，社会保险号（相当于我们的身份证号码）等等。当然像姓氏这样的用来标识 Subject 并不是很好，因此，最好的用来进行身份验证的 Principals(身份)是对应用程序来说应该是独一无二的——通常是用户名或电子邮件地址。

> **Primary Principal** 虽然 Shiro 可以代表任意数量的 Principals(身份)，但 Shiro 期望应用程序有一个确切的‘主要的’Principals(身份)——一个单一的值在应用程序内部唯一标识 Subject。这通常是一个用户名，电子邮件地址或者在大多数应用中的全球唯一用户 ID。

- **Credentials**(凭证)通常是只被 Subject 知道的秘密值，它用来作为一种起支持作用的证据，此证据事实上包含着所谓的身份证明。一些常见 credentials(凭证)的例子有密码，生物特征数据如指纹和视网膜扫描，以及 X.509 证书。

principal/credential 配对最常见的例子是用户名和密码。用户名是所声称的身份，密码是匹配所声称的身份的证明。如果提交的密码与应用程序期望的相匹配，应用程序可以很大程度上假设用户真的是他们说的他们是谁，因为其他人都应该不知道同样的密码。

Authenticating Subjects(验证 Subjects)

验证 Subjects 的过程中，可以有效地分解成三个不同的步骤：

1. 收集 Subjects 提交的 Principals(身份)和 Credentials(凭证)；
2. 提交 Principals(身份)和 Credentials(凭证)进行身份验证；
3. 如果提交成功，则允许访问，否则重新进行身份验证或者阻止访问。

Step 1：收集 Subject 的 Principals(身份)和 Credentials(凭证)

```
//Example using most common scenario of username/password pair:
UsernamePasswordToken token = new UsernamePasswordToken(username, password);
```

```
//"Remember Me" built-in:
token.setRememberMe(true);
```

在这种特殊情况下，我们使用 UsernamePasswordToken 来支持最常见的用户名/密码的身份验证方法。这是 Shiro 的 org.apache.shiro.authc.AuthenticationToken 的接口，是 Shiro 代表提交的 Principals(身份)和 Credentials(凭证)的身份验证系统所使用的基本接口的一个实现。

这里需要重要注意的是 Shiro 不关心你是如何获取此信息的：也许获得的数据是由用户提交的一个 HTML 表单，或者是从 HTTP 头中捕获，或者它是从一个 Swing 或 Flex GUI 密码表单，或者通过命令行参数。从终端用户收集信息的过程与 Shiro 的 Authentication Token 概念是不挂钩的。

你可以创建和实例化你喜欢的 Authentication Token 实例——它是与协议无关的。

这个例子也说明了我们希望 Shiro 为身份验证的尝试执行“记住我”的服务。这将确保在以后的日子，如果用户返回到应用程序时，Shiro 能够记得用户的身份。我们将在后面的章节讨论 Remember Me 服务。

Step 2：提交 Subject 的 Principals(身份)和 Credentials(凭证)

在 Principals(身份)和 Credentials(凭证)被收集以及被实例化为 AuthenticationToken 实例后，我们需要提交这个 token 给 Shiro 来执行真正的身份验证尝试：

```
Subject currentUser = SecurityUtils.getSubject();
currentUser.login(token);
```

在捕获到当前执行的 Subject 后，我们获得一个单一的 login 方法调用，并将之前获得的 AuthenticationToken 实例传递给它。

通过调用 login 方法，有效地体现了身份验证尝试。

Step3：处理成功或失败

如果 login 方法返回平静地，就是它——我们所做的一切！该 Subject 已通过验证。应用程序线程可以不受干扰地继续下去，而且所有进一步对 SecurityUtils.getSubject()的调用将返回认证后的 Subject 实例，同时任何对 subject.isAuthenticated()的调用将返回 true。

但是如果登录尝试失败会发生什么呢？例如，如果终端用户提供了不正确的密码，或这访问系统的次数太多，亦或是他们的帐户被锁定？

Shiro 拥有丰富的运行时 AuthenticationException 层次结构，可以指出尝试失败的确切原因。你可以用一个 try/catch 块将 login 方法包围起来，然后捕捉任何你期望的异常并进行相应的反应。例如：

```
try { currentUser.login(token);
} catch ( UnknownAccountException uae ) { …
} catch ( IncorrectCredentialsException ice ) { …
} catch (LockedAccountException lae ) { …
} catch (ExcessiveAttemptsException eae ) { …
} …  catch your own …
} catch ( AuthenticationException ae ) {
            //unexpected error?
}
```

```
//No problems, continue on as expected...
```

如果现有的异常类不符合您的要求，可以自定义 AuthenticationExceptions 来代表具体的异常情况。

Login Failure Tip
虽然你的代码可以以特定的异常作出反应，并执行必要的逻辑，最安全的做法是只显示通用的失败消息给终端用户，例如，“错误的用户名或密码。”。这样将确保具体的信息提供给黑客可能试图攻击的媒介。

Remembered vs. Authenticated(记住我对比认证)
如上面的例子所示，Shiro 支持除了普通的登录过程的所有“记住我”的概念。此时值得指出的是，Shiro 对记住我的 Subject 和通过验证的 Subject 作了精确的区分：

- **Remembered**(记住我)：一个记住我的 Subject 不是匿名的，而且有一个已知的身份 ID（也就是 subject.getPrincipals()是非空的）。但是这个被记住的身份 ID 是在之前的 session 中被认证的。如果 subject.isRemembered()返回 true，则 Subject 被认为是被记住的。
- **Authenticated**(已认证)：一个已认证的 Subject 是指在当前 Session 中被成功地验证过了（也就是说，login 方法被调用并且没有抛出异常）。如果 subject.isAuthenticated()返回 true 则认为 Subject 已通过验证。

Mutually Exclusive(互斥)
Remembered 和 Authenticated 是互斥的——若其中一个为真则另一个为假，反之亦然。

为何有这样的区别？

“身份验证”这个词有很强的证明的意思在里面。也就是说，有一个预期保证 Subject 已经证明他们是他们所说的谁。
当用户只记得之前与应用的交互时，认证将不复存在：被记住的身份 ID 使系统明白这个用户可能是谁，但在现实中没有办法绝对保证被记住的 Subject 代表期望的用户。一旦 Subject 通过验证，它们将不再仅仅被认为是被记住的，由于它们的身份已经在当前 session 中被证实。
尽管应用程序的许多部分仍然能够执行基于被记住身份 ID 的用户特定逻辑，像自定义视图，但它绝不应该执行高度敏感的操作，除非用户通过执行一个成功的认证尝试来合法地验证自己的身份。
例如，一个检查来判断一个 Subject 可以访问财务信息应该几乎总是取决于 isAuthenticated()，而不是 isRemembered()，以保证一个预期和核实的身份。

一个说明的例子

下面是一个相当普遍的情况，有助于说明 Remembered 和 Authenticated 之间区别的重要性。
比方说，你正在访问 Amazon.com。你已经登录成功并添加了几本书到你的购物车。但你心烦意乱地跑出去开会，却忘了注销。会议结束后，已经到了回家的时候，于是你离开了办公室。

第二天你工作的时候，你意识到你没有完成购买，于是你返回到 amazon.com。这一次，Amazon“记得”你是谁，给出了你的欢迎页面，并仍然为你提供一些个性化的建议书籍。对 Amazon 而言，subject.isRemembered()将返回 true。

但是，当你尝试访问你的帐户来更新你的信用卡信息为你书付账时会发生什么呢？尽管 Amazon“记住”你(isRemembered() = = true)，它不能保证你就是实际上的你（例如，也许一个同事正在使用你的计算机）。

所以，在你能够执行像更新信用卡信息等敏感行为之前，Amazon 将强制让你登录，使它们能够保证你的身份。在登录后，你的身份已经被核实，同时对 Amazon 而言，isAuthenticated()现在返回是 true。

这种情况在许多类型的应用中发生的是如此的频繁，所以这些功能被内置在 Shiro 中，这样你就能利用它来为你的应用服务了。现在，无论你使用的是 isRemembered()还是 isAuthenticated()来定制你的视图和工作流都由你来决定，但 Shiro 将维持这一基本情况以防你需要它。

注销

进行身份验证的反面是释放所有已知的的识别状态。当 Subject 完成了与应用程序的交互后，你可以调用 subject.logout()来释放所有的识别信息：

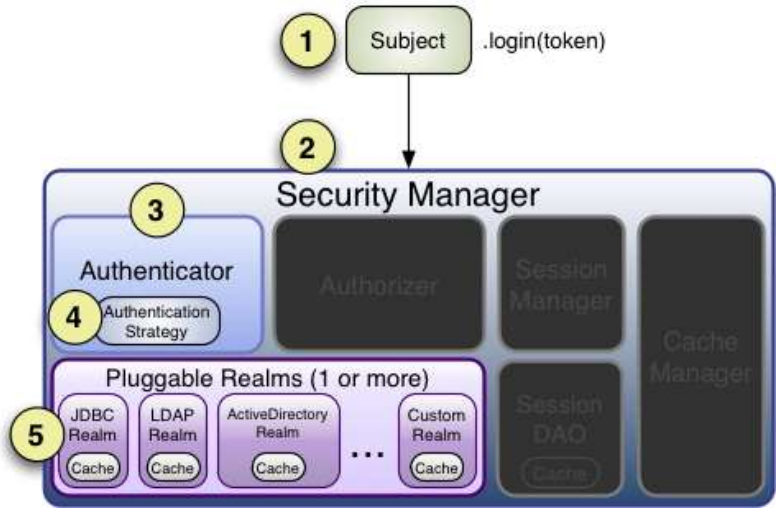
```
currentUser.logout();  //removes all identifying information an invalidates their session too.
```

当你调用 logout，任何现有的 Session 都将会失效，而且任何身份都将会失去关联（例如，在 Web 应用程序中，RememberMe cookie 也将被删除）。
在 Subject 注销后，该 Subject 的实例被再次认为是匿名的，当然，除了 Web 应用程序，它还可以重新用于 login 如果需要的话。

> Web Application Notice　由于在 Web 应用程序记住身份往往是依靠 Cookies，然而 Cookies 只能在 Response 被 committed 之前被删除，所以强烈建议在调用 subject.logout()后立即将终端用户重定向到一个新的视图或页面。这样能够保证任何与安全相关的 Cookies 都能像预期的一样被删除。这是 HTTP cookies 的功能限制，而不是 Shiro 的。

验证顺序

到现在为止，我们只了解了如何从应用程序代码中验证一个 Subject。现在我们将涉及到当一个认证尝试出现时 Shiro 内部会发什么。
我们采用了 Architecture 那一章的体系结构图，并只留下与 Authentication 有关的组件突出显示。每个数字代表认证尝试中的一个步骤：



Step 1：应用程序代码调用 Subject.login 方法，传递创建好的包含终端用户的 Principals(身份)和 Credentials(凭证)的 AuthenticationToken 实例。
Step 2：Subject 实例，通常是 DelegatingSubject（或子类）委托应用程序的 SecurityManager 通过调用 securityManager.login(token)开始真正的验证工作。
Step3：SubjectManager 作为一个基本的“保护伞”的组成部分，接收 token 以及简单地委托给内部的 Authenticator 实例通过调用 authenticator.authenticate(token)。这通常是一个 ModularRealmAuthenticator 实例，支持在身份验证中协调一个或多个 Realm 实例。ModularRealmAuthenticator 本质上为 Apache Shiro 提供了 PAM-style 范式（其中在 PAM 术语中每个 Realm 都是一个'module'）。

Step 4：如果应用程序中配置了一个以上的 Realm，ModularRealmAuthenticator 实例将利用配置好的 AuthenticationStrategy 来启动 Multi-Realm 认证尝试。在 Realms 被身份验证调用之前，期间和以后，AuthenticationStrategy 被调用使其能够对每个 Realm 的结果作出反应。我们马上就会涉及到 AuthenticationStrategies。

Single-Realm Application 如果只有一个单一的 Realm 被配置，它将被直接调用——没有必要为一个单一 Realm 的应用使用 AuthenticationStrategy。

Step 5：每个配置的 Realm 用来帮助看它是否支持提交的 AuthenticationToken。如果支持，那么支持 Realm 的 getAuthenticationInfo 方法将会伴随着提交的 token 被调用。getAuthenticationInfo 方法有效地代表一个特定 Realm 的单一的身份验证尝试。我们将在不久涉及到 Realm 验证行为。

Authenticator(认证器)

如前所述，Shiro SecurityManager 的实现默认使用一个 ModularRealmAuthenticator 实例。

ModularRealmAuthenticator 同样支持单一的 Realm 以及那些多个 Realm 的应用。

在一个单一的 Realm 应用中，ModularRealmAuthenticator 将直接调用这个单一的 Realm。如果有两个或两个以上的 Realm 配置，它将使用 AuthenticationStrategy 实例来调整这些尝试如何出现。下面我们将介绍 AuthenticationStrategies。

如果你想配置 SecurityManager 通过一个自定义的 Authenticator 实现，你可以在 shiro.ini 中做，例如：

```
[main]
…
authenticator = com.foo.bar.CustomAuthenticator
securityManager.authenticator = $authenticator
```

尽管在实践中，ModularRealmAuthenticator 很可能是适用于大多数需要的。

AuthenticationStrategy

当一个应用程序配置了两个或两个以上的 Realm 时，ModularRealmAuthenticator 依靠内部的 AuthenticationStrategy 组件来确定这些认证尝试的成功或失败条件。

例如，如果只有一个 Realm 验证一个 AuthenticationToken 成功，但所有其他的都失败，这被认为是成功的身份验证尝试吗？或者必须所有的 Realm 验证成功才被认为样子成功吗？或者，如果一个 Realm 验证成功，是否有必要进一步征询其他 Realm？AuthenticationStrategy 基于程序需要作出合适的决定。

AuthenticationStrategy 是一个无状态的组件，它在身份验证尝试中被询问 4 次（这 4 次交互所需的任何必要的状态将被作为方法参数）：

1. 在任何 Realm 被调用之前被询问；
2. 在一个单独的 Realm 的 getAuthenticationInfo 方法被调用之前立即被询问；
3. 在一个单独的 Realm 的 getAuthenticationInfo 方法被调用之后立即被询问；
4. 在所有的 Realm 被调用后询问。

另外，AuthenticationStrategy 负责从每一个成功的 Realm 汇总结果并将它们“捆绑”到一个单一的 AuthenticationInfo 再现。这最后汇总的 AuthenticationInfo 实例就是从 Authenticator 实例返回的值以及 Shiro 所用来代表 Subject 的最终身份 ID 的值（即 Principals(身份)）。

Subject Identity 'View' 如果在你的应用程序中使用多个 Realm 从多个数据源获取账户资料，AuthenticationStrategy 是最终为最后的“合并”能够被应用程序理解的 Subject 的身份的视图的负责人。

Shiro 有 3 个具体的 AuthenticationStrategy 实现：

AuthenticationStrategy class	描述
AtLeastOneSuccessfulStrategy	如果一个（或更多）Realm 验证成功，则整体的尝试被认为是成功的。如果没有一个验证成功，

	则整体尝试失败。
FirstSuccessfulStrategy	只有第一个成功地验证的 Realm 返回的信息将被使用。所有进一步的 Realm 将被忽略。如果没有一个验证成功，则整体尝试失败。
AllSucessfulStrategy	为了整体的尝试成功，所有配置的 Realm 必须验证成功。如果没有一个验证成功，则整体尝试失败。

ModularRealmAuthenticator 默认的是 AtLeastOneSuccessfulStrategy 实现，因为这是最常所需的方案。然而，如果你愿意的话，你可以配置一个不同的方案：

shiro.ini
[main] … authcStrategy = org.apache.shiro.authc.pam.FirstSuccessfulStrategy seurityManager.authenticator.authenticationStrategy = $authcStrategy …
Custom AuthenticationStrategy 如果你想创建你自己的 AuthenticationStrategy 来实现你自己，你可以使用 org.apache.shiro.authc.pam.AbstractAuthenticationStrategy 作为出发点。AbstractAuthenticationStrategy 类自动实现“捆绑”/聚集行为，从每一个 Realm 合并结果到一个单一的 AuthenticationInfo 实例。

Realm 的验证顺序

需要指出非常重要的一点是，ModularRealmAuthenticator 将与 Realm 实例以迭代的顺序进行交互。

在 SecurityManager 中已经配置好了 ModularRealmAuthenticator 对 Realm 实例的访问。当执行一个认证尝试时，它将会遍历该集合，并对每一个支持提交 AuthenticationToken 的 Realm 调用 Realm 的 getAuthenticationInfo 方法。

Implicit ordering(隐式排列)

当使用 Shiro 的 INI 配置文件格式时，你应该配置 Realm 处理 AuthenticationToken 的顺序，你想要的顺序。例如，在 shiro.ini 中，Realm 将会以它们在 INI 文件中定义好的顺序被请求到。也就是说，对于下面的 shiro.ini 示例：

```
blahRealm = com.company.blah.Realm
…
fooRealm = com.company.foo.Realm
…
barRealm = com.company.another.Realm
```

SecurityManager 根据这三个 Realm 来配置，在认证尝试期间，blahRealm，fooRealm 和 barRealm 将按照上面那个顺序调用。

这基本是相同的效果，就像下面的这一行定义：

```
securityManager.realms = $blahRealm, $fooRealm, $barRealm
```

使用这种方法，你并不需要设置 SecurityManager 的 Realm 属性——每个定义好的 realm 将会自动地被添加到 realm 的属性。

Explicit Ordering(显示排列)

如果你想明确地定义 Realm 的交互顺序，忽略它们是如何定义的，你可以设置 securityManager 的属性作为一个明确的集合属性。例如，如果使用上面的定义，但你想 blahRealm 最后被请求而不是第一个：

```
blahRealm = com.company.blah.Realm
…
fooRealm = com.company.foo.Realm
…
barRealm = com.company.another.Realm
securityManager.realms = $fooRealm, $barRealm,
$blahRealm
```

Explicit Realm Inclusion 当你显式地配置 securityManager.realms 的属性是，只有已引用的 Realm 将会在 SecurityManager 中被配置。这意味着你能够在 INI 文件中定义 5 个 realm，但是实际上只能使用 3 个如果只有这 3 个被引用到 realm 的属性中的话.这是和隐式 realm 顺序不同的，所有可用的隐式的 realm 都将被使用。

Realm 验证

本章涵盖了 Shiro 的主要工作流程，解释了一个认证尝试是如何产生的。内部工作流，指在验证过程中一个单一的 realm 被访问时发生的事情（也就是'Step 5'之上），已经包含在 Realm 章节的 Realm Authentication 部分。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而提高了 Shiro。

贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Authorization（授权）

Apache Shiro Authorization

授权，又称作为访问控制，是对资源的访问管理的过程。换句话说，控制谁有权限在应用程序中做什么。

授权检查的例子是：该用户是否被允许访问这个网页，编辑此数据，查看此按钮，或打印到这台打印机？这些都是决定哪些是用户能够访问的。

授权的要素

Apache Shiro 中的权限代表着安全政策中最基础的元素。它们从根本上作出了对行为的声明，并明确表示可以在应用程序中做什么。一个格式良好的权限声明基本上描述了资源以及当 Subject 与这些资源进行交互时可能出现的行为。
权限语句的一些例子：

- 打开一个文件；
- 查看'/user/list'网页；
- 打印文档；
- 删除用户'jsmith'。

大多数资源将支持典型的 CRUD（创建，读取，更新，删除）操作，但任何对特定资源有意义的行为都是可以的。基本的概念是，最小的许可声明是基于资源和行为的。
在查看权限时，最重要的可能是认识到许可声明没有谁可以执行代表行为的表现形式。它们仅仅只是在一个应用程序中能做什么的声明语句。

> Permissions represent behavior only
> 许可声明仅能够反映行为（与资源类型相关的行为）。它们不反映是谁能够执行这样的行为。

定义（用户）被允许做什么（权限），是一个以某种方式分配给用户权限的运用。这通常是由应用程序的数据模型来完成的，并且不同应用程序间变化很大。
例如，权限能够被集合到一个角色中，该角色可以与一个或多个用户对象相关联。或者某些应用程序可以有一组可以被分配一个角色的用户和组，传递的关联意味着该组中的所有用户都隐式地获得了该角色的权限。
如何授予用户权限可以有很多变化——应用程序决定如何基于应用的要求来建模。
我们稍后将讨论 Shiro 是如何确定一个 Subject 是否被允许做些什么。

权限粒度

以上所有权限例子详细说明了在某一资源类型（入口，文件，客户等等）的行为（打开，阅读，删除等等）。在某些情况下，它们甚至可以指定非常细粒度的实例级的行为——例如，“删除”（行为）用户名为"jsmith"的“用户”（资源类型）。在 Shiro，你有能力来定义这些声明能够达到的精确粒度。

我们将在 Shiro 的 Permission 文档中更加详细地讨论权限粒度和许可声明的“等级”。

角色

角色是一个命名的实体，通常代表一组行为或职责。这些行为演化为你在一个软件应用中能或者不能做的事情。角色通常是分配给用户帐户的，因此，通过分配，用户能够“做”的事情可以归属于各种角色。
有两种有效类型的角色，并且 Shiro 支持这两个概念：

- **隐式角色**：大多数人使用的角色作为一个隐式的构造：你的应用程序仅仅基于一个角色名就蕴含了一组行为（也就是权限）。有了隐式角色，在软件级别上没有说“角色 X 被允许执行行为 A，B 和 C”。行为已被一个单独的名字所蕴含。

> **Potentially Brittle Security**
> 虽然有比较简单和最常用的方法，隐式角色可能强加许多软件的维护和管理问题。
> 例如，如果你只是想添加或删除一个角色，或者重新定义一个角色的行为呢？你不得不返回到你的代码，并改写所有你所有的角色检查以反映你的安全模型的改变，每次像这样的改变都是必不可少的！且不说这会招致运营成本（重新测试，经过 QA，关闭应用程序，新的角色下检查升级软件，重启该应用程序等）。
> 这对非常简单的应用程序可能没什么问题（例如，也许有一个“admin”的角色和“其他人”的角色）。但对于更复杂的或可配置的应用程序，这可能是你的应用程序整个生命周期中最主要问题，并且为你的软件造成一大笔维护费用。

- **显式角色**：一个显式角色本质上是一个实际许可声明的命名集合。在这种形式下，应用程序（以及 Shiro）确切地知道有没有一个特定的角色意味着什么。因为它是已知能不能够被执行的确切行为，没有猜测或暗示一个特定的角色能或不能做什么。

Shiro 团队提倡使用权限和显式角色，而不是陈旧的隐式方法。你将会拥有更多的控制应用程序的安全经验。

Resource-Based Access Control(基于资源的访问控制)
请务必阅读 Les Hazlewood 的文章，新的 RBAC：基于资源的访问控制，其中包括深入使用权限和显式角色（以及它们在源代码上产生的积极影响）而不是陈旧的隐式方法的好处。

Users(用户)

用户实质上是指与应用程序有关的人。然而正如我们已经讨论的，Subject 才是 Shiro 的“用户”概念。
允许用户（Subjects）在你的应用程序中执行某些操作，是通过与他们的角色相关联或授予直接的权限。你的应用程序的数据模型定义了 Subject 是如何被允许做某事或不的。
例如，在你的数据模型中，也许你有一个实际的 User 类，而且你直接分配权限给 User 实例。或者，你也许只分配权限给角色，然后分配角色给用户，通过关联，用户延伸“有”的权限分配给自己的角色。或者你用"Group"的概念来代替这些东西。这些都随便你——使用什么使得你的程序有意义。
你的数据模型定义授权究竟是如和工作的。Shiro 依靠 Realm 来实现转换你的数据模型使其细节关联到一种 Shiro 能够理解的格式。
我们一会儿将讨论 Realms 是如何做到这一点的。

最终，你的 Realm 的实现是与你的数据源（RDBMS，LDAP 等）进行通信。所以，你的 realm 就是告诉 Shiro 是否存在角色或权限。在你的授权模型结构和定义上你有充分的控制权。

Authorizing Subjects(授权的 Subjects)

在 Shiro 中执行授权可以有 3 种方式：

- 编写代码——你可以在你的 Java 代码中用像 if 和 else 块的结构执行授权检查。
- JDK 的注解——你可以添加授权注解给你的 Java 方法。
- JSP/GSP 标签库——你可以控制基于角色和权限的 JSP 或者 GSP 页面输出。

Programmatic Authorization(编程授权)
也许最简单和最常见的方式来执行授权是直接以编程方式与当前 Subject 实例交互。
Role-Based Authorization(基于角色的授权)
如果你想进行基于简单/传统的隐式角色名来控制访问，你可以执行角色检查：
Role checks(角色检查)
如果你只是简单的想检查当前的 Subject 是否拥有一个角色，你可以在 Subject 实例上调用变体的 hasRole*方法。
例如，判断一个 Subject 是否拥有一个特别的（单一的）角色，你可以通过调用 subject.hasRole 方法，并作出相应的反应：

```
Subject currentUser = SecurityUtils.getSubject();

if(currentUser.hasRole("administrator")) {
        //show the admin button
} else {
        //don't show the button? Grey it out?
}
```

有几个面向角色的 Subject 方法可以调用，一起取决于你的需要：

Subject 方法	描述
hasRole(String roleName)	返回 true 如果 Subject 被分配了指定的角色，否则返回 false。
hasRole(List<String> roleNames)	返回 true 如果 Subject 被分配了所有指定的角色，否则返回 false。
hasAllRoles(Collection<String> roleNames)	返回一个与方法参数中目录一致的 hasRole 结果的数组。有性能的提高如果许多角色需要执行检查（例如，当自定义一个复杂的视图）。

Role Assertions(角色断言)

另一种方法通过检查布尔值来判断 Subject 是否拥有一个角色，你可以简单地断言它们有一个预期的角色在逻辑被执行之前。如果 Subject 没有预期的角色，AuthorizationException 将会被抛出。如果它们有预期的角色，断言将悄悄地执行，并且逻辑将如预期般继续。

例如：

```
Subject currentUser = SecurityUtils.getSubject();
//guarantee that the current user is a bank teller and
//therefore allowed to open the account:
currentUser.checkRole("bankTeller");
openBankAccount();
```

通过使用 hasRole*方法的一个好处就是代码可以变得清洁，由于你不需要创建你自己的 AuthorizationException 如果当前的 Subject 不符合预期条件（如果你不想的话）。

有几个你能调用的面向角色的 Subject 断言方法，取决于你的需要：

Subject 方法	描述
checkRole(String roleName)	安静地返回，如果 Subject 被分配了指定的角色，不然的话就抛出 AuthorizationException。
checkRoles(Collection<String> roleNames)	安静地返回，如果 Subject 被分配了所有的指定的角色，不然的话就抛出 AuthorizationException。
checkRoles(String… roleNames)	与上面的 checkRoles 方法的效果相同，但允许 Java5 的 var-args 类型的参数。

Permission-Based Authorization(基于权限的授权)

如前所述在我们所概述的角色，往往一个更好的方式执行访问控制是通过基于权限的授权。基于权限的授权，由于它与你的应用程序的原始功能（以及应用程序核心资源上的行为）紧密的关联在一起的，基于权限的授权源代码会在你的功能改变时改变，而不是在安全政策改变时改变。这意味着代码很少会被影响对比相似的基于角色的授权的代码。

Permission Checks(权限检查)

如果你想进行检查，看一个 Subject 是否被允许做某事，你可以调用各种 isPermitted*方法的变种。检查权限主要有两个方式——基于对象的权限实例或代表权限的字符串。

Object-based Permission Checks(基于对象的权限检查)

执行权限检查的一个可行方法是实例化 org.apache.shiro.authz.Permission 接口的一个实例，并把它传递给接收权限实例的*isPermitted 方法。

例如，请考虑以下情况：在办公室有一台打印机，具有唯一标识符 laserjet4400n。我们的软件需要检查当前用户是否被允许在该打印机上打印文档在我们允许他们按下“打印”按钮之前。上述情况的权限检查可以明确地像这样表达：

```
Permission printPermission = new PrinterPermission("laserjet4400n","print");
Subject currentUser = SecurityUtils.getSubject();
If (currentUser.isPermitted(printPermission)) {
        //show the Print button
} else {
        //don't show the button? Grey it out?
}
```

在这个例子中，我也看到了一个非常强大的实例级的访问控制检查的例子——限制行为的能力是基于个人的数据实例。

基于对象的权限是很有用的，如果：

- 你想编译时类型安全
- 你想保证权限被描述和使用是正确的
- 你想显式控制许可解析逻辑（被称作许可蕴含的逻辑，基于权限接口的 implies 方法）是如何执行的。

- 你想保证权限反映到应用程序资源是准确的（例如，也许权限类可以在能够基于项目的域模型的项目编译时自动生成）。

有几个你能调用的面向权限的 Subject 方法，取决于你的需要：

Subject 方法	描述
isPermitted(Permission p)	返回 true 如果该 Subject 被允许执行某动作或访问被权限实例指定的资源集合，否则返回 false。
isPermitted(List<Permission> perms)	返回一个与方法参数中目录一致的 isPermitted 结果的数组。有性能的提高如果需要执行许多检查（例如，当自定义一个复杂的视图）。
isPermittedAll(Collection<Permission> perms)	返回 true 如果该 Subject 被允许所有指定的权限，否则返回 false。

String-based permission checks(基于字符串的权限检查)

基于对象的权限可以是很有用的（编译时类型安全，保证行为，定制蕴含逻辑等），它们有时对应用程序来说会感到有点“笨手笨脚”的。另一种方法是使用正常的字符串来表示权限实例。

例如，基于上面的打印权限的例子上，我们可以重新制订与之前检查相同的基于字符串的权限检查：

```
Subject currentUser = SecurityUtils.getSubject();

if    (currentUser.isPermitted("printer:print:laserjet4400n")) {
            //show the Print button
} else {
            //don't show the button?    Grey it out?
}
```

这个例子还显示了相同的实例级权限检查，但权限的重要组成部分——打印机（资源类型），打印（行为），以及 laserjet4400n（实例 ID）——都用一个字符串表示。

这个特别的例子显示了一个特殊冒号分隔的格式，它由 Shiro 默认的 org.apache.shiro.authz.permission.WildcardPermission 实现来定义，其中大多数人会找到适合自己的格式。

也就是说，上面的代码块（大部分）是下面代码的简化：

```
Subject currentUser = SecurityUtils.getSubject();
Permission p = new WildcardPermission("printer:print:laserjet4400n");
if    (currentUser.isPermitted("printer:print:laserjet4400n")) {
            //show the Print button
} else {
            //don't show the button?    Grey it out?
}
```

WildcardPermission token 规定和构造操作的格式在 Shiro 的 Permission 文档中被深入的涉及到。

除了上面的字符串默认的 WildcardPermission 格式，你可以创建和使用自己的字符串格式如果你喜欢的话。我们将在 Realm Authorization 这一节讨论如何去做。

基于字符串的权限是很有帮助的，由于你不必被迫实现一个接口，而且简单的字符串易于阅读。其缺点是，你不具备类型安全，如果你需要更为复杂的行为将超出了字符串所能代表的范围，你就得实现你自己的基于权限接口的权限对象。在实际中，大部分的 Shiro 终端用户为了简洁选择基于字符串的方式，但最终你应用程序的需求会决定哪一个更好。

像基于对象的权限检查方法一样，也有字符串的变体来支持基于字符串的权限检查：

Subject 方法	描述
isPermitted(String perm)	返回 true 如果该 Subject 被允许执行某动作或访问被字符串权限指定的资源，否则返回 false。

isPermitted(String… perms)	返回一个与方法参数中目录一致的 isPermitted 结果的数组。有性能的提高如果许多字符串权限检查需要被执行（例如，当自定义一个复杂的视图）。
isPermittedAll(String… perms)	返回 true 如果该 Subject 被允许所有指定的字符串权限，否则返回 false。

Permission Assertions(权限断言)

作为检查一个布尔值来判断 Subject 是被允许做某事的一种替代，你可以在逻辑被执行之前简单地断言他们是否拥有预期的权限。如果该 Subject 是不被允许，AuthorizationException 异常将会被抛出。如果他们如预期的被允许，断言将安静地执行，逻辑也将如预期般继续。

例如：

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is permitted
//to open a bank account:
Permission p = new AccountPermission("open");
currentUser.checkPermission(p);
openBankAccount();
```

或者，同样的检查，使用字符串权限：

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is permitted
//to open a bank account:
currentUser.checkPermission("open");
openBankAccount();
```

通过使用 hasRole*方法的一个好处就是代码可以变得清洁，由于你不需要创建你自己的 AuthorizationException 如果当前的 Subject 不符合预期条件（如果你不想的话）。

有几个你能调用的面向权限的 Subject 断言方法，取决于你的需要：

Subject 方法	描述
checkPermission(Permission p)	安静地返回，如果 Subject 被允许执行某动作或访问被特定的权限实例指定的资源，不然的话就抛出 AuthorizationException 异常。
checkPermission(String perm)	安静地返回，如果 Subject 被允许执行某动作或访问被特定的字符串权限指定的资源，不然的话就抛出 AuthorizationException 异常。
checkPermissions(Collection<Permission> perms)	安静地返回，如果 Subject 被允许所有的权限，不然的话就抛出 AuthorizationException 异常。
checkPermissions(String… perms)	和上面的 checkPermissions 方法效果相同，但是使用的是基于字符串的权限。

Annotation-based Authorization(基于注解的授权)

除了 Subject API 的调用， Shiro 提供 Java 5+注解的集合，如果你喜欢以注解为基础的授权控制。

configuration(配置)

在你可以使用 Java 注释之前，你需要在你的应用程序中启用 AOP 支持。虽然现在有许多不同的 AOP 框架，但不幸的是，在应用程序中没有一个使用 AOP 的标准。

对于 AspectJ 而言，你你可以回顾我们的 AspectJ 的示例程序。

对于 Spring 应用而言，你可以看看我们的 Spirng 集成文档。

The RequiresAuthentication annotation(RequiresAuthentication 注解)

RequiresAuthentication 注解要求当前 Subject 已经在当前的 session 中被验证通过才能被注解的类/实例/方法访问或调用。

例如：

```
@RequiresAuthentication
public void updateAccount(Account userAccount) {
        //this method will only be invoked by a
        //Subject that is guaranteed authenticated
        …
}
```

这通常等同于接下来的基于 Subject 的逻辑：

```
public void updateAccount(Account userAccount) {
        if (!SecurityUtils.getSubject().isAuthenticated()) {
                throw new AuthorizationException(...);
        }

        //Subject is guaranteed authenticated here
        …
}
```

The RequiresGuest annotation(RequiresGuest 注解)

RequiresGuest 注解要求当前的 Subject 是一个"guest"，也就是说，他们必须是在之前的 session 中没有被验证或记住才能被注解的类/实例/方法访问或调用。

例如：

```
@RequiresGuest public void signUp(User newUser) {
        //this method will only be invoked by a
        //Subject that is unknown/anonymous
        …
}
```

这通常等同于接下来的基于 Subject 的逻辑：

```
public void signUp(User newUser) {
        Subject currentUser = SecurityUtils.getSubject();
        PrincipalCollection principals = currentUser.getPrincipals();
        if (principals != null && !principals.isEmpty()) {
                //known identity - not a guest:
                throw new AuthorizationException(...);
        }
        //Subject is guaranteed to be a 'guest' here
        …
}
```

The RequiresPermissions annotation(RequiresPermissions 注解)

RequiresPermissions 注解要求当前的 Subject 被允许一个或多个权限，以便执行注解的方法。

例如：

```
@RequiresPermissions("account:create")
public void createAccount(Account account) {
        //this method will only be invoked by a Subject
        //that is permitted to create an account
        …
}
```

这通常等同于接下来的基于 Subject 的逻辑：

```
public void createAccount(Account account) {
        Subject currentUser = SecurityUtils.getSubject();
        if (!subject.isPermitted("account:create")) {
                throw new AuthorizationException(...);
        }

        //Subject is guaranteed to be permitted here
        …
}
```

The RequiresRoles annotation(RequiresRoles 注解)

RequiresRoles 注解要求当前的 Subject 拥有所有指定的角色。如果他们没有，则该方法将不会被执行，而且 AuthorizationException 异常将会被抛出。

例如：

```
@RequiresRoles("administrator")
public void deleteUser(User user) {
        //this method will only be invoked by an administrator
        …
}
```

这通常等同于接下来的基于 Subject 的逻辑：

```
public void deleteUser(User user) {
        Subject currentUser = SecurityUtils.getSubject();
        if (!subject.hasRole("administrator")) {
                throw new AuthorizationException(...);
        }

        //Subject is guaranteed to be an 'administrator' here
        …
}
```

The RequiresUser annotation(RequiresUser 注解)

RequiresUser 注解需要当前的 Subject 是一个应用程序用户才能被注解的类/实例/方法访问或调用。一个“应用程序用户”被定义为一个拥有已知身份，或在当前 session 中由于通过验证被确认，或者在之前 session 中的'RememberMe'服务被记住。

```
@RequiresUser
public void updateAccount(Account account) {
        //this method will only be invoked by a 'user'
        //i.e. a Subject with a known identity
        …
}
```

这通常等同于接下来的基于 Subject 的逻辑：

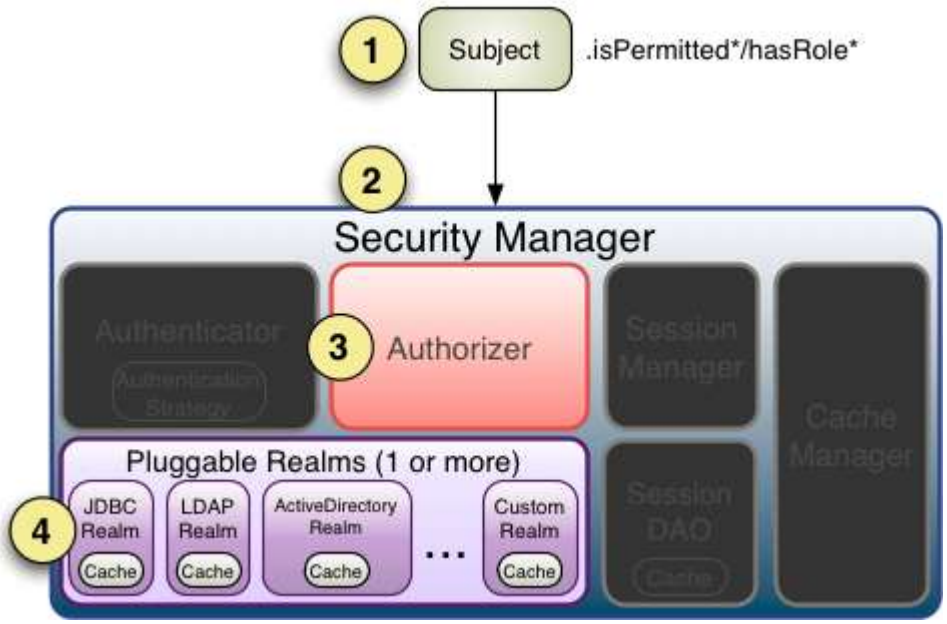
```
public void updateAccount(Account account) {
        Subject currentUser = SecurityUtils.getSubject();
        PrincipalCollection principals = currentUser.getPrincipals();
        if (principals == null || principals.isEmpty()) {
                //no identity - they're anonymous, not allowed:
                throw new AuthorizationException(...);
        }

        //Subject is guaranteed to have a known identity here
        …
}
```

JSP TagLib Authorization(JSP TagLib 授权)

Shiro 提供了一个用于控制 JSP/GSP 页面输出的基于 Subject 状态的标签库。这些包含在 Web 章节的 JSP/GSP 标签库部分。

Authorization Sequence(授权顺序)

现在我们已经知道了基于当前 Subject 上如何执行授权，让我们看看当授权调用时，Shiro 内部会发生什么。
我们采用了 Architecture 那一章的体系结构图，并只留下与 authorization 有关的组件突出显示。每个数字代表授权过程中的一个步骤：



Step 1：应用程序或框架代码调用任何 Subject 的 hasRole*, checkRole*, isPermitted*，或者 checkPermission*方法的变体，传递任何所需的权限或角色代表。
Step 2：Subject 的实例，通常是 DelegatingSubject（或子类）代表应用程序的 SecurityManager 通过调用 securityManager 的几乎各自相同的 hasRole*, checkRole*, isPermitted*，或 checkPermission*方法的变体（SecurityManager 实现 org.apache.shiro.authz.Authorizer 接口，他定义了所有 Subject 具体的授权方法）。
Step 3：SecurityManager，作为一个基本的“保护伞”组件，接替/代表它内部的 org.apache.shiro.authz.Authorizer 实例通过调用 authorizer 各自的 hasRole*, checkRole*, isPermitted*，或者 checkPermissions*方法。默认情况下，authorizer 实例是一个 ModularRealmAuthorizer 实例，它支持协调任何授权操作过程中的一个或多个 Realm 实例。
Step 4：每个配置好的 Realm 被检查是否实现了相同的 Authorizer 接口。如果是，Realm 各自的 hasRole*, checkRole*, isPermitted*，或 checkPermission*方法将被调用。

ModularRealmAuthorizer

如前所述，Shiro SecurityManager 的实现默认是使用一个 ModularRealmAuthorizer 实例。ModularRealmAuthorizer 同样支持单一的 Realm，以及那些与多个 Realm 的应用。

对于任何授权操作，ModularRealmAuthorizer 将遍历其内部的 Realm 集合，并按迭代顺序与每一个进行交互。每个 Realm 的交互功能如下：

1. 如果 Realm 自己实现了 Authorizer 接口，它的各个 Authorizer 方法（hasRole*, checkRole*, isPermitted*，或 checkPermission*）将被调用。
    1. 如果 Realm 的方法导致异常，该异常将会以 AuthorizationException 的形式传递给调用者。这将短路授权过程，同时任何剩余的 Realm 将不会被该授权操作所访问。
    2. 如果该 Realm 的方法是一个返回布尔值的 hasRole*或者 isPermitted*的变体，并且该返回值为 true，真值将会立即被返回，同时任何剩余的 Realm 都将被短路。这种行为作为提高性能的一种存在，如果该行为被一个 Realm 允许，这意味着该 Subject 也是被允许的。这有利于安全政策,每一处都是默认被禁止的情况下,一切都明确允许的,这是安全政策最安全的类型。
2. 如果 Realm 不实现 Authorizer 接口，它会被忽略。

Realm Authorization Order(Realm 的授权顺序)

需要重要指出的是，尤其是身份验证，ModularRealmAuthorizer 将以迭代顺序与 Realm 实例进行交互。

ModularRealmAuthorizer 根据 SecurityManager 的配置获得对 Realm 实例的访问。当执行授权操作时，它会遍历该集合，同时对于每一个自己实现 Authorizer 接口的 Realm，调用 Realm 各自的 Authorizer 方法（如 hasRole*, checkRole*, isPermitted*，或 checkPermission*）。

Configuring a global PermissionResolver(配置全局的 PermissionResolver)

当执行基于字符串的权限检查是，大多数 Shiro 的默认 Realm 实现首先将该字符串转换成一个实际的 Permission 实例，在执行权限 implication 逻辑之前。

这是因为 Permission 是基于 implication 逻辑评估的，而不是直接的 equality 检查（见 Permission 文档有关更多 implication 和 equality 的对比）。Implication 逻辑对比通过字符串比较能够更好的在代码中体现。因此，大多数 Realm 需要转换，或者将提交的权限字符串解析成相应的代表权限的实例。

为了帮助这种转换，Shiro 支持 PermissionResolver 的概念。大多数 Shiro Realm 的实现使用一个 PermissionResolver 以支持他们的基于字符串权限的 Authorizer 接口方法的实现：当其中一种方法在 Realm 上被调用是，它将使用 PermissionResolver 把该字符串转换成一个权限实例，并用这种方式来执行检查。

所有 Shiro Realm 的实现默认是内部的 WildcardPermissionResolver，它采用 Shiro 的 WildcardPermission 字符串格式。

如过你想创建自己的 PermissionResolver 的实现，也许是为了支持自己的权限字符串语法，而且你想要所有配置的 Realm 实例支持该语法，你可以将你的 PermissionResolver 设置为全局的，这样所有的 Realm 能够用一个配置。

例如，在 shiro.ini 中：

shiro.ini
globalPermissionResolver = com.foo.bar.authz.MyPermissionResolver … securityManager.authorizer.permissionResolver = $ globalPermissionResolver …

PermissionResolverAware

如果你想配置一个全局的 PermissionResolver，每个用来接收配置的 PermissionResolver 的 Realm 必须实现 PermissionResolverAware 接口。这样保证了配置的实例能够被每个支持该配置的 Realm 转发。

如果你想使用全局的 PermissionResolver 或你不想被 PermissionResolverAware 接口所困扰，你可以随时显式地配置一个拥有 PermissionResolver 实例的 Realm（假设有一个兼容 JavaBean 的 setPermissionResolver 的方法）：

```
permissionResolver = com.foo.bar.authz.MyPermissionResolver

realm = com.foo.bar.realm.MyCustomRealm
```

```
realm.permissionResolver = $permissionResolver
…
```

Configuring a global RolePermissionResolver(配置全局的 RolePermissionResolver)

与 PermissionResolver 在概念上相似，RolePermissionResolver 有能力代表需要的权限实例，通过一个 Realm 执行权限检查。

然而，与一个 RolePermissionResolver 的关键区别是输入的字符串是一个角色名，而不是一个权限字符串。

RolePermissionResolver 能够在 Realm 内部使用，当需要将一个角色名转换成一组具体的权限实例时。

这是一个特别有用的特征用来支持旧的或不灵活的，可能没有权限概念的数据源。

例如，许多 LDAP 目录存储了角色名（或组名），但是不支持关联角色名到具体的权限由于他们没有“权限”的概念。一个基于 Shiro 的应用程序能够使用存储在 LDAP 的角色名，还能实现一个 RolePermissionResolver 来转化 LDAP 名到一组显式的权限来执行首选的显式的访问控制。权限关联将会被存储在另一个数据仓库，可能是一个本地数据库。

由于这种转换角色名到权限的概念非常特定于应用程序，Shiro 默认 Realm 的实现并不使用它们。

然而，如果你想创建你自己的 RolePermissionResolver，并有多个你想配置的 Realm 的实现，你可以将你的 RolePermissionResolver 设置为全局的，这样所有的 Realm 都能够用一个配置。

shiro.ini

```
globalRolePermissionResolver = com.foo.bar.authz.MyPermissionResolver
…
securityManager.authorizer.RolePermissionResolver =
$ globalRolePermissionResolver
…
```

RolePermissionResolverAware

如果你想配置一个全局的 RolePermissionResolver，每个用来接收配置的 RolePermissionResolver 的 Realm 必须实现 RolePermissionResolverAware 接口。这样保证了配置的全局的 RolePermissionRosolver 实例能够被每个支持该配置的 Realm 转发。

如果你不希望使用全局的 RolePermissionResolver 或你不想被 RolePermissionResolverAware 接口所困扰，你可以随时显式地配置一个拥有 RolePermissionResolver 实例的 Realm（假设有一个兼容 JavaBean 的 setRolePermissionResolver 的方法）：

```
rolePermissionResolver = com.foo.bar.authz.MyRolePermissionResolver

realm = com.foo.bar.realm.MyCustomRealm
realm.RolePermissionResolver = $rolePermissionResolver
…
```

Custom Authorizer(自定义授权者)

如果你的应用程序使用多个 realm 来执行授权，并且 ModularRealmAuthorizer 默认基于简单的迭代，短路授权行为不符合你的要求，你很有可能想创建一个自定义的授权者，并配置相应的 SecurityManager。

例如，在 shiro.ini 中：

```
[main]
…
authorizer = com.foo.bar.authz.CustomAuthorizer
securityManager.authorizer = $authorizer
```

Understanding Permissions in Apache Shiro

Shiro 将权限定义为一个规定了明确行为或活动的声明。这是一个在应用程序中的原始功能语句，仅此而已。权限是在安全策略中最低级别的构造，且它们明确地定义了应用程序只能做“什么”。

它们从不描述“谁”能够执行这些动作。

一些权限的例子：

- 打开文件
- 浏览'/user/list'页面
- 打印文档
- 删除'jsmith'用户

规定“谁”（用户）允许做“什么”（权限）在某种程度上是分配用权限的一种习惯做法。这始终是通过应用程序数据模型来完成的，并且在不同应用程序之间差异很大。

例如，权限可以组合到一个角色中，且该角色能够关联一个或多个用户对象。或者某些应用程序能够拥有一组用户，且这个组可以被分配一个角色，通过传递的关联，意味着所有在该组的用户隐式地获得了该角色的权限。

如何授予用户权限可以有很多变化——应用程序基于应用需求来决定如何使其模型化。

Wildcard Permissions

上述的权限例子，“打开文件”、“浏览'/user/list'页面”等都是有效的权限语句。然而，将这些解释为自然语言字符串，并判断用户是否被允许执行该行为在计算上是非常困难的。

因此，为了使用易于处理且仍然可读的权限语句，Shiro 提供了强大而直观的语法，我们称之为 WildcardPermission。

Simple Usage

假设你想要保护到贵公司打印机的访问，使得某些人能够打印到特定的打印机，而其他人可以查询当前有哪些工作在队列中。

一个极其简单的方法是授予用户"queryPrinter"权限。然后你可以检查用户是否具有 queryPrinter 权限通过调用：

```
subject.isPermitted("queryPrinter")
```

这（很大程度）相当于

```
subject.isPermitted( new WildcardPermission("queryPrinter"))
```

但远不只这些。

简单的权限字符串可能在简单的应用程序中工作的很好，但它需要你拥有像"printPrinter"，"queryPrinter"，"managePrinter"等权限。你还可以通过使用通配符授予用户"*"权限（赋予此权限构造它的名字），这意味着他们在整个应用程序中拥有了所有的权限。

但使用这种方法不能说用户拥有“所有打印机权限”。由于这个原因，Wildcard Permissions（通配符权限）支持多层次的权限管理。

Multiple Parts

通配符权限支持多层次或部件（parts）的概念。例如，你可以通过授予用户权限来调整之前那个简单的例子。

```
printer:query
```

在这个例子中的冒号是一个特殊字符，它用来分隔权限字符串的下一部件。

在该例中，第一部分是权限被操作的领域（打印机），第二部分是被执行的操作（查询）。上面其他的例子将被改为：

```
printer:print
printer:manage
```

对于能够使用的部件是没有数量限制的，因此它取决于你的想象，依据你可能在你的应用程序中使用的方法。

Multiple Vaules
每个部件能够保护多个值。因此，除了授予用户"printer:print"和"printer:query"权限外，你可以简单地授予他们一个：

```
printer:print, query
```

它能够赋予用户 print 和 query 打印机的能力。由于他们被授予了这两个操作，你可以通过调用下面的语句来判断用户是否有能力查询打印机：

```
subject.isPermitted("print:query")
```

该语句将会返回 true。

All Values
如果你想在一个特定的部件给某一用户授予所有的值呢？这将是比手动列出每个值更为方便的事情。同样，基于通配符的话，我也可以做到这一点。若打印机域有 3 个可能的操作（query，print 和 manage），可以像下面这样：

```
printer:query, print, manage
```

简单点变成这样：

```
printer:*
```

然后，任何对"printer:XXX"的权限检查都将返回 true。以这种方式使用的通配符比明确地列出操作具有更好的尺度，如果你不久为应用程序增加了一个新的操作，你不需要更新使用通配符那部分的权限。

最后，在一个通配符权限字符串中的任何部分使用通配符 token 也是可以的。例如，如果你想对某个用户在所有领域（不仅仅是打印机）授予"view"权限，你可以这样做：

```
*:view
```

这样任何对"foo:view"的权限检查都将返回 true。

Instance-Level Access Control
另一种常见的通配符权限用法是塑造实例级的访问控制列表。在这种情况下，你使用三个部件——第一个是域，第二个是操作，第三个是被付诸实施的实例。

因此，像下面的例子：

```
printer:query:lp7200
printer:print:epsoncolor
```

第一个定义了查询拥有 ID lp7200 的打印机的行为。第二条权限定义了打印到拥有 ID epsoncolor 的打印机的行为。如果你授予这些权限给用户，那么他们能够在特定的实例上执行特定的行为。然后你可以在代码中做一个检查：

```
if (SecurityUtils.getSubject().isPermitted("printer:query:lp7200") {
        // Return the current jobs on printer lp 7200
}
```

这是体现权限的一个极为有效的方法。但同样，为所有的打印机定义多个实例 ID 能很好的扩展，尤其是当新的打印机添加到系统的时候。你可以使用通配符来代替：

```
printer:print:*
```

这个做到了扩展，因为它同时涵盖了任何新的打印机。你甚至可以运行访问所有打印机上的所有操作：

```
printer:*:*
```

或在一台打印机上的所有操作：

```
printer:*:lp7200
```

或甚至特定的操作：

```
printer:query, print:lp7200
```

"*"通配符，","子部件分离器可用于权限的任何部分。

Missing Parts

最后要注意的是权限分配：缺少的部件意味着用户可以访问所有与之匹配的值，换句话说，

```
printer:print
```

等价于

```
printer:print:*
```

并且

```
printer
```

等价于

```
printer:*:*
```

然而，你只能从字符串的结尾处省略部件，因此这样的：

```
printer:lp7200
```

并不等价于

```
printer:*:lp7200
```

Checking Permissions

虽然权限分配使用通配符较为方便且具有扩展性（"printer:print:*" = print to any printer），但在运行时的权限检查应该始终基于大多数具体的权限字符串。

例如，如果用户有一个用户界面，他们想打印一份文档到 lp7200 打印机，你应该通过执行这段代码来检查用户是否被允许这样做：

```
if ( SecurityUtils.getSubject().isPermitted("printer:print:lp7200")) {
        //print the document to the lp7200 printer
}
```

这种检查非常具体和明确地反映了用户在那一时刻试图做的事情。

然而，下面这个运行是检查是不为理想的：

```
if ( SecurityUtils.getSubject().isPermitted("printer:print")) {
        //print the document
}
```

为什么？因为第二个例子表明“对于下面的代码块的执行，你必须能够打印到任何打印机”。但请记住"pinter:print"是等价于"priner:print:*"的！

因此，这是一个不正确的检查。如果当前用户不具备打印到任何打印机的能力，仅仅只有打印到 lp7200 和 epsoncolor 的能力，该怎么办呢？那么上面的第二个例子也绝不允许他们打印到 lp7200 打印机，即使他们已被赋予了相应的能力！

因此，经验法则是在执行权限检查时，尽可能使用权限字符串。当然，上面的第二块可能是在应用程序中别处的一个有效检查，如果你真的想要执行该代码块，如果用户被允许打印到任何打印机（令人怀疑的，但有可能）。你的应用程序将决定检查哪些有意义，但一般情况下，越具体越好。

Implication, not Equality

为什么运行时权限检查应该尽可能的具体，但权限分配可以较为普通？这是因为权限检查是通过蕴含的逻辑来判断的——而不是通过相等检查。

也就是说，如果一个用户被分配了 user:*权限，这意味着该用户可以执行 user:view 操作。"user:*"字符串明显不等于"user:view"，但前者包含了后者。"user:*"描述了"user:view"所定义的功能的一个超集。

为了支持蕴含规则，所有的权限都被翻译到实现 org.apache.shiro.authz.Permission 接口的的对象实例中。这是以便蕴含逻辑能够在运行时执行，且蕴含逻辑通常比一个简单的字符串相等检查更为复杂。所有在本文档中描述的通配符行为实际上是由 org.apache.shiro.authz.permission.WildcardPermission 类实现的。下面是更多的一些通过蕴含逻辑访问的通配符权限字符串：

```
user:*
```

同时蕴含着删除一个用户的能力:

```
user:delete
```

同样地，

```
user:*:12345
```

同时蕴含着更新 ID 为 12345 的用户帐户的能力：

```
user:update:12345
```

而且

```
printer
```

蕴含着打印到任何打印机的能力

```
printer:print
```

Performance Considerations
权限检查比简单的相等比较要复杂得多，因此运行时的蕴含逻辑必须执行每个分配的权限。当使用像上面展示的权限字符串时，你正在隐式地使用 Shiro 默认的 WildcardPermission，它能够执行必要的蕴含逻辑。

Shiro 对 Realm 实现的默认行为是，对于每一个权限验证（例如，调用 subject.isPermitted），所有分配给该用户的权限（在他们的组，角色中，或直接分配给他们）需要为蕴含逻辑进行单独的检查。Shiro 通过首次成功检查立即返回来“短路”该进程以提高性能，但它不是一颗银弹。

这通常是极快的，当用户，角色和权限缓存在内存中且使用了一个合适的 CacheManager 时，在 Shiro 不支持的 Realm 实现中。只要知道使用此默认行为，当权限分配给用户或他们的角色或组增加时，执行检查的时间一定会增加。

如果一个 Realm 的实现者有一个更为高效的方式来检查权限并执行蕴含逻辑，尤其它如果是基于应用程序数据模型的，他们应该实现它作为 Realm isPermitted* 方法实现的一部分。默认的 Realm/WildcardPermission 存在的支持覆盖了大多数用例的 80~90%，但它可能不是在运行时拥有大量权限需要存储或检查的应用程序的最佳解决方案。

Lend a hand with documentation
我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Apache Shiro Realms

Realm 是一个能够访问应用程序特定的安全数据（如用户、角色及权限）的组件。Realm 将应用程序特定的数据转换成一种 Shiro 能够理解的格式，这样 Shiro 能够提供一个单一的易理解的 Subject 编程 API，无论有多少数据源存在或你应用程序特定的数据是怎样的。

Realm 通常和数据源是一对一的对应关系，如关系数据库，LDAP 目录，文件系统，或其他类似资源。因此，Realm 接口的实现使用数据源特定的 API 来展示授权数据（角色，权限等），如 JDBC，文件 IO，Hibernate 或 JPA，或其他数据访问 API。

Realm 实质上就是一个特定安全的 DAO

因为这些数据源大多通常存储身份验证数据（如密码的凭证）以及授权数据（如角色或权限），每个 Shiro Realm 能够执行身份验证和授权操作。

Realm configuration

如果使用 Shiro 的 INI 配置文件，你能够自定义及引用 Realm，就像在[main]项中的任何其他对象一样，但它们在 securityManager 中采用两种方法之一进行配置：显式或隐式。

Explicit Assignment

基于迄今的 INI 配置知识，这是一个显示的配置方法。在定义一个或多个 Realm 后，你将它们作为 securityManager 对象的集合属性。

例如：

```
fooRealm = com.company.foo.Realm
barRealm = com.company.another.Realm
bazRealm = com.company.baz.Realm

securityManager.realms = $fooRealm, $barRealm, $bazRealm
```

显式分配是确定的——你控制具体使用哪一个 Realm 及它们用于身份验证和授权的顺序。Realm 顺序的作用在 Authentication 章的 Authentication Sequence 节进行了详细的介绍。

Implicit Assignment

Not Preferred(不推荐)

这种方法可能引发意想不到的行为，如果你改变 realm 定义的顺序的话。建议你避免使用此方法，并使用显式分配，它拥有确定的行为。该功能很可能在未来的 Shiro 版本中被废弃或移除。

如果出于某些原因你不想显式地配置 securityManager.realms 的属性，你可以允许 Shiro 检测所有配置好的 realm 并直接将它们指派给 securityManager。

使用这种方法，realm 将会按照它们预先定义好的顺序来指派给 securityManager 实例。

也就是说，对于下面的 shiro.ini 示例：

```
blahRealm = com.company.blah.Realm
fooRealm =    com.company.foo.Realm
barRealm = com.company.another.Realm

# no securityManager.realms assignment here
```

基本上和下面这一行具有相同的效果：

```
securityManager.realms = $blahRealm, $fooRealm, $barRealm
```

然而，实现隐式分配，只是 realm 定义的顺序直接影响到了它们在身份验证和授权尝试中的访问顺序。如果你改变它们定义的顺序，你将改变主要的 Authenticaor 的 Authentication Sequence 是如何起作用的。

由于这个原因，以及保证明确的行为，我们推荐使用显式分配而不是隐式分配。

Realm Authentication

当你理解了 Shiro 的主要 Authentication 工作流后，了解在一个授权尝试中当 Authenticator 与 Realm 交互时到底发生了什么是很重要的。

Supporting AuthenticationTokens

在 authentication sequence 中已经提到，当在 Realm 被访问来执行一个授权尝试之前，它的 supports 方法被调用。如果返回值为 true，则只有这样它的 getAuthenticationInfo(token)方法才会被调用。

通常 realm 会检查提交的 token 的类型（接口或类）来判断它是否能够处理它。例如，一个能够处理生物数据的 realm 可能就一点也不理解 UsernamePasswordTokens，这样它将从 supports 方法返回 false。

Handling supported AuthenticationTokens

若 Realm 支持一个提交的 AuthenticationToken，那么 Authenticator 将会调用该 Realm 的 getAuthenticationInfo(token)方法。这有效地代表了一个与 Realm 的后备数据源的授权尝试。该方法按以下方法进行：

1. 为主要的识别信息（帐户识别信息）检查 token。
2. 基于 principal 在数据源中寻找相吻合的帐户数据。
3. 确保 token 支持的 credentials 匹配那些存储在数据源的。
4. 若 credentials 匹配，返回一个封装了 Shiro 能够理解的帐户数据格式的 AuthenticationInfo 实例。
5. 若 credentials 不匹配，则抛出 AuthenticationException 异常。

这是对所有 Realm getAuthenticationInfo 实现的最高级别的工作流。在此方法中，Realm 可以自由地做任何它们想做的，如记录在审计日志的尝试，更新数据记录，或任何其他可以对该数据存储的身份验证尝试有意义的东西。

唯一需要的东西就是，如果 credentials 匹配给予的 principal(s)，那么返回一个非空的 AuthenticationInfo 实例来代表来自于该数据源的 Subject 帐户信息。

Save Time

直接实现 Realm 接口可能导致时间消耗及错误。大多数人民选择 AuthorizingRealm 抽象类的子类而不是从头开始。这个类实现了常用的 authentication 及 authorization 工作流来节省你的时间和精力。

Credentials Matching

在上面的 realmauthentication 工作流中，Realm 不得不验证 Subject 提交的 credentials（如，，密码）必须匹配存储在数据存储中的 credentials。如果匹配，则被认为身份验证成功，同时系统还必须验证终端用户的身份。

Realm Credentials Matching

这是每个 Realm 的责任，去匹配提交的 credentials 和那些存储在 Realm 后备数据存储中的 credentials，而不是 Authenticator 的责任。每个 Realm 拥有有关私人信息的 credentials 格式，存储及能够执行详细的 credentials 匹配，然而 Authenticator 只是一个普通的工作量组件。

credentials 的匹配过程在所有应用程序中几乎一样，通常不一样的是进行比较的数据。为了确保该过程是可插入及可定制的如果需要的话，AuthenticatingRealm 及它的子类支持 CredentialsMatcher 来执行 credentials 对比的概念。

在发现帐户数据后，它以及提交的 AuthenticationToken 用来代表一个 CredentialsMatcher 来判断所提交的是否匹配所存储的。

Shiro 拥有某些可以让你立即使用的 CredentialsMatcher 实现，如 SimpleCredenticalsMatcher 和 HashedCredentialsMatcher，但如果你想为自定义的逻辑配置一个自定义的实现，你可以像下面一样直接做：

```
Realm myRealm = new com.company.shiro.realm.MyRealm();
CredentialsMatcher customMatcher = new com.company.shiro.realm.CustomCredentialsMatcher();
myRealm.setCredentialsMatcher(customMatcher);
```

或者，如果使用 Shiro 的 INI 配置文件：

```
[main]
…
customMatcher = com.company.shiro.realm.CustomCredentialsMatcher
myRealm = com.company.shiro.realm.MyRealm
myRealm.credentialsMatcher = $customMatcher
…
```

Simple Equality Check

所有 Shiro 立即可用的 Realm 的实现默认使用 SimpleCredentialsMatcher。SimpleCredentialsMatcher 执行一个普通的直接平等检查，关于存储的帐户 credentials 与在 AuthenticationToken 所提交的之间的检查。

例如，若一个 UsernamePasswordToken 被提交后，则 SimpleCredentialsMatcher 验证该密码实际上是否与存储在数据库中的密码相同。

SimpleCredentialsMatcher 不仅仅为字符串执行直接相等比较。它能够处理大多数常用的字节码，像字符串，字符数组，字节数组，文件及输入流。请参考它的 JavaDoc 获取更多。

Hashing Credentials

并非是存储 credentials 在其原始的 form 及执行原始/普通的比较，一个更安全的方式存储终端用户的 credentials（如，密码）是在存储它们到数据存储之前将它们单向散列化。

这确保终端用户的 credentials 绝不会以原始的 form 存储，而且没人会知道原始值。这是一个比纯文本或原始比较更为安全的机制，同时所有关注安全的应用程序应该较非哈希化的存储更为喜欢。

为了支持这些首选的加密哈希策略，Shiro 提供了 HashedCredentialsMatcher 的实现配置在 realm 上而不是上述 SimpleCredentialsMatcher。

哈希 credentials 及 salting 和多个哈希迭代的好处超出了该 Realm 文档的范围，但绝对要阅读 HashedCredentialsMatcher 的 JavaDoc，其中将详细介绍这些细节。

Hashing and Corresponding Matchers

那么，你如何很容易地配置一个启用 Shiro 的应用程序呢？

Shiro 提供了多个 HashedCredentialsMatcher 子类实现。你必须在你的 realm 中配置指定的实现来匹配你 hash 化你用户 credentials 时使用的哈希算法。

例如，假设你的应用程序为身份验证使用用户名/密码对。由于上文所述的哈希凭据的好处，假设当你创建一个用户帐户时，你想使用 SHA-256 算法单向散列用户的密码。你将哈希用户输入的纯文本密码并保持该值：

```
import org.apache.shiro.crypto.hash.Sha256Hash;
import org.apache.shiro.crypto.RandomNumberGenerator;
import org.apache.shiro.crypto.SecureRandomNumberGenerator;
…
```

```
//We'll use a Random Number Generator to generate salts. This
//is much more secure than using a username as a salt or not
//
//Note that a normal app would reference an attribute rather
//than create a new RNG every time:
RandomNumberGenerator rng = new SecureRandomNumberGenerator();
Object salt = rng.nextBytes();

//now hash the plain-text password with the random salt and multiple
//iterations and then Base64-encode the value (requires less space than Hex):
String hashedPasswordBase64 = new Sha256Hash(plainTextPassword, salt, 1024).toBase64();

User user =new User(username, hashedPasswordBase64);
//save the salt with the new account . The HashedCredentialsMatcher
//will need it later when handling login attempts:
user.setPasswordSalt(salt);
userDAO.create(user);
```

既然你使用 SHA-256 散列你用户的密码，你需要告诉 Shiro 使用合适的 HashedCredentialsMatcher 以匹配你的哈希参数选择。在这个例子中，我们创建了一个随机的 salt 并执行 1024 次哈希迭代，为了强大的安全性（请参见 HashedCredentialsMatcher 的 JavaDoc 获取原因）。这里是完成这项工作的 Shiro INI 配置：

```
[main]
…
credentialsMatcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
# base64 encoding, not hex in this example:
credentialsMatcher.storedCredentialsHexEncoded = false
credentialsMatcher.hashIterations = 1024
# This next property is only needed in Shiro 1.0. Remove it in 1.1 and later:
credentialsMatcher.hashSalted =true

…
myRealm = com.company……
myRealm.credentialsMatcher = $credentialsMathcer
...
```

SaltedAuthenticationInfo

为了确保这一工程，最后要做的事情是，你的 Realm 实现必须返回一个 SaltedAuthenticationInfo 实例而不是一个普通的 AuthenticationInfo 实例。SaltedAuthenticationInfo 接口确保在你创建用户帐户（如，user.setPasswordSalt(Salt); call above）时使用的 salt 能够被 HashedCredentialsMatcher 引用.

HashedCredentialsMatcher 需要该 salt 为了能够在提交的 AuthenticationToken 上执行相同的哈希技术来判断该 token 是否匹配你保存在数据存储中的东西。因此，如果你为用户密码使用 salting（而且你应该这样做！！！），确保你的 Realm 实现能够通过返回的 SaltedAuthenticationInfo 实例代表密码。

Disabling Authentication

如果出于某些原因，你不想用 Realm 对数据源执行身份验证（也许是由于你只想 Realm 执行授权），你可以彻底地禁用 Realm 对身份验证的支持通过从 Realm 的 support 方法返回 false。然后你的 realm 在身份验证尝试中永远不会被访问到。

当然，至少需要一个能够支持 AuthenticationTokens 且已配置的 Realm，如果你想验证 Subjects。

Realm Authorization

TBD(待定)

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Session Management

Apache Shiro 提供安全框架界独一无二的东西：一个完整的企业级 Session 解决方案，从最简单的命令行及智能手机应用到最大的集群企业 Web 应用程序。

这对许多应用有着很大的影响——直到 Shiro 出现，如果你需要 session 支持，你需要部署你的应用程序到 Web 容器或使用 EJB 有状态会话 Bean。Shiro 的 Session 支持比这两种机制的使用和管理更为简单，而且它在适用于任何程序，不论容器。

即使你在一个 Servlet 或 EJB 容器中部署你的应用程序，仍然有令人信服的理由来使用 Shiro 的 Session 支持而不是容器的。下面是一个 Shiro 的 Session 支持的最可取的功能列表：

Features

- **POJO/J2SE based(IoC friendly)** - Shiro 的一切（包括所有 Session 和 Session Management 方面）都是基于接口和 POJO 实现。这可以让你轻松地配置所有拥有任何 JavaBeans 兼容配置格式（如 JSON，YAML，Spring XML 或类似的机制）的会话组件。你也可以轻松地扩展 Shiro 的组件或编写你自己所需的来完全自定义 session management。
- **Easy Custom Session Storage** - 因为 Shiro 的 Session 对象是基于 POJO 的，会话数据可以很容易地存储在任意数量的数据源。这允许你自定义你的应用程序会话数据的确切位置——例如，文件系统，联网的分布式缓存，关系数据库，或专有的数据存储。
- **Container-Independent Clustering! -** Shiro 的会话可以很容易地聚集通过使用任何随手可用的网络缓存产品，像 Ehcache + Terracotta，Coherence，GigaSpaces，等等。这意味着你可以为 Shiro 配置会话群集一次且仅一次，无论你部署到什么容器中，你的会话将以相同的方式聚集。不需要容器的具体配置！
- **Heterogeneous Client Access** - 与 EJB 或 web 会话不同，Shiro 会话可以被各种客户端技术“共享”。例如，一个桌面应用程序可以“看到”和“共享”同一个被使用的物理会话通过在 Web 应用程序中的同一用户。我们不知道除了 Shiro 以外的其他框架能够支持这一点。
- **Event Listeners -** 事件监听器允许你在会话生命周期监听生命周期事件。你可以侦听这些事件和对自定义应用程序的行为作出反应——例如，更新用户记录当他们的会话过期时。
- **Host Address Retention** - Shiro Sessions 从会话发起地方保留 IP 地址或主机名。这允许你确定用户所在，并作出相应的反应（通常是在 IP 分配确定的企业内部网络环境）。

- **Inactivity/Expiration Support** - 由于不活动导致会话过期如预期的那样，但它们可以延续很久通过 touch()方法来保持它们“活着”，如果你希望的话。这在 RIA(富互联网应用)环境非常有用，用户可能会使用桌面应用程序，但可能不会经常与服务器进行通信，但该服务器的会话不应过期。
- **Transparent Web Use** - Shiro 的网络支持，充分地实现和支持关于 Sessions（HttpSession 接口和它的所有相关的 API）的 Servlet2.5 规范.这意味着你可以使用在现有 Web 应用程序中使用 Shiro 会话，并且你不需要改变任何现有的 Web 代码。
- **Can be used for SSO** - 由于 Shiro 会话是基于 POJO 的，它们可以很容易地存储在任何数据源，而且它们可以跨程序“共享”如果需要的话。我们称之为"poor man's SSO"，并它可以用来提供简单的登录体验，由于共享的会话能够保留身份验证状态。

Using Sessions

几乎与所有其他在 Shiro 中的东西一样，你通过与当前执行的 Subject 交互来获取 Session：

```
Subject currentUser = SecurityUtils.getSubject();
Session session = currentUser.getSession();
session.setAttribute("someKey", someValue);
```

subject.getSession()方法是调用 currentUser.getSubject(true)的快捷方式。

对于那些熟悉 HttpServletRequest API 的，Subject.getSession(boolean create)方法与 HttpServletRequest.getSession(boolean create)方法有着异曲同工之效。

- 如果该 Subject 已经拥有一个 Session，则 boolean 参数被忽略且 Session 被立即返回。
- 如果该 Subject 还没有一个 Session 且 create 参数为 true，则创建一个新的会话并返回该会话。
- 如果该 Subject 还没有一个 Session 且 create 参数为 false，则不会创建新的会话且返回 null。

Any Application

getSession 要求能够在任何应用程序工作，甚至是非 Web 应用程序。

当开发框架代码来确保一个 Session 没有被创建是没有必要的时候，subject.getSession(false)可以起到很好的作用。

当你获取了一个 Subject 的 Session 后，你可以用它来做许多事情，像设置或取得 attribute，设置其超时时间，以及更多。请参见 Session 的 JavaDoc 来了解一个单独的会话能够做什么。

The SessionManager

SessionManager，名如其意，在应用程序中为所有的 subject 管理 Session——
创建，删除，inactivity(失效)及验证，等等。如同其他在 Shiro 中的核心结构组件一样，SessionManager 也是一个由 SecurityManager 维护的顶级组件。

默认的 SecurityManger 实现是默认使用立即可用的 DefaultSessionManager。DefaultSessionManager 的实现提供一个应用程序所需的所有企业级会话管理，如 Session 验证，orphan cleanup，等等。这可以在任何应用程序中使用。

Web Applications

Web 应用程序使用不同 SessionManager 实现。请参见 Web 文档获取 web-specific Session Management 信息。

像其他被 SecurityManager 管理的组件一样，SessionManager 可以通过 JavaBean 风格的 getter/setter 方法在所有 Shiro 默认 SecurityManager 实现（getSessionManager()/setSessionManager()）上获取或设置值。或者例如，如果在使用 shiro.ini 配置：

Configuring a new SessionManager in shiro.ini

```
[main]
…
sessionManager = com.foo.my.SessionManagerImplementation
securityManager.sessionManager = $sessionManager
```

但从头开始创建一个 SessionManager 是一个复杂的任务且是大多数人不想亲自做的事情。Shiro 的立即可用的 SessionManager 实现是高度可定制的和可配置的，并满足大多数的需要。本文档的其余部分假定你将使用 Shiro 的默认 SessionManager 实现，当覆盖配置选项时。但请注意，你基本上可以创建或插入任何你想要的东西。

Session Timeout

默认地，Shiro 的 SessionManager 实现默认是 30 分钟会话超时。也就是说，如果任何 Session 创建后闲置（未被使用，它的上次访问时间未被更新）的时间超过了 30 分钟，那么该 Session 就被认为是过期的，且不允许再被使用。

你可以设置 SessionManager 默认实现的 globalSessionTimeout 属性来为所有的会话定义默认的超时时间。例如，如果你想超时时间是一个小时而不是 30 分钟：

Setting the Default Session Timeout in shiro.ini

```
[main]
…
# 3,600,000 milliseconds = 1 hour
securityManager.sessionManager.globalSessionTimeout = 3600000
```

Per-Session Timeout

上面的 globalSessionTimeout 值默认是为新建的 Session 使用的。你可以在每一个会话的基础上控制超时时间通过设置单独的会话超时时间值。与上面的 globalSessionTimeout 一样，该值以毫秒（不是秒）为时间单位。

Session Listeners

Shiro 支持 SessionListener 概念来允许你对发生的重要会话作出反应。你可以实现 SessionListener 接口（或扩展易用的 SessionListenerAdapter）并与相应的会话操作作出反应。

由于默认的 SessionManager sessionListeners 属性是一个集合，你可以对 SessionManager 配置一个或多个 listener 实现，就像其他在 shiro.ini 中的集合一样：

SessionListener Configuration in shiro.ini

```
[main]
…
aSessionListener = com.foo.my.SessionListener
anotherSessionListener = com.foo.my.OtherSessionListener

securityManager.sessionManager.sessionListeners = $aSessionListener, $anotherSessionListener, etc.
```

All Session Events

当任何会话发生事件时，SessionListeners 都会被通知——不仅仅是对一个特定的会话。

Session Storage

每当一个会话被创建或更新时，它的数据需要持久化到一个存储位置以便它能够被稍后的应用程序访问。同样地，当一个会话失效且不再被使用时，它需要从存储中删除以便会话数据存储空间不会被耗尽。SessionManager 实现委托这些 Create/Read/Update/Delete(CRUD)操作为内部组件，同时，SessionDAO，反映了数据访问对象（DAO）设计模式。

SessionDAO 的权力是你能够实现该接口来与你想要的任何数据存储进行通信。这意味着你的会话数据可以驻留在内存中，文件系统，关系数据库或 NoSQL 的数据存储，或其他任何你需要的位置。你得控制持久性行为。

你可以将任何 SessionDAO 实现作为一个属性配置在默认的 SessionManager 实例上。例如，在 shiro.ini 中：

Configuring a SessionDAO in shiro.ini
[main] … sessionDAO = com.foo.my.SessionDAO securityManager.sessionManager.sessionDAO = $sessionDAO

然而，正如你可能期望的那样，Shiro 已经有一些很好的 SessionDAO 实现，你可以立即使用或实现你需要的子类。

Web Applications

上述的 securityManager.sessionManager .sessionDAO = $sessionDAO 作业仅在使用一个本地的 Shiro 会话管理器时才工作。Web 应用程序默认不会使用本地的会话管理器，而是保持不支持 SessionDAO 的 Servlet Container 的默认会话管理器。如果你想基于 Web 应用程序启用 SessionDAO 来自定义会话存储或会话群集，你将不得不首先配置一个本地的 Web 会话管理器。例如：

```
[main]
…
sessionManager = org.apache.shiro.web.session.mgt.DefaultWebSessionManager
securityManager.sessionManager = $sessionManager

# Configure a SessionDAO and then set it:
securityManager.sessionManager.sessionDAO = $sessionDAO
```

Configure a SessionDAO!

Shiro 的默认配置本地 SessionManagers 使用仅内存 Session 存储。这是不适合大多数应用程序的。大多数生产应用程序想要配置提供的 EHCache（见下文）支持或提供自己的 SessionDAO 实现。

请注意 Web 应用程序默认使用基于 servlet 容器的 SessionManager，且没有这个问题。这也是使用 Shiro 本地 SessionManager 的唯一问题。

EHCache SessionDAO

EHCache 默认是没有启用的，但如果你不打算实现你自己的 SessionDAO，那么强烈地建议你为 Shiro 的 SessionManagerment 启用 EHCache 支持。EHCache SessionDAO 将会在内存中保存会话，并支持溢出到磁盘，若内存成为制约。这对生产程序确保你在运行时不会随机地“丢失”会话是非常好的。

Use EHCache as your default

如果你不准备编写一个自定义的 SessionDAO，则明确地在你的 Shiro 配置中启用 EHCache。EHCache 带来的好处远不止在 Sessions，缓存验证和授权数据方面。更多信息，请参见 Caching 文档。

Container-Independent Session Clustering

如果你急需独立的容器会话集群，EHCache 会是一个不错的选择。你可以显式地在 EHCache 之后插入 TerraCotta，并拥有一个独立于容器集群的会话缓存。不必再担心 Tomcat，JBoss，Jetty，WebSphere 或 WebLogic 特定的会话集群！

为会话启用 EHCache 是非常容易的。首先，确保在你的 classpath 中有 shiro-ehcache-<version>.jar 文件（请参见 Download 页面或使用 Maven 或 Ant+Ivy）。

当在 classpath 中后，这第一个 shiro.ini 实例向你演示怎样为所有 Shiro 的缓存需要（不只是会话支持）使用 EHCache：

Configuring EHCache for all of Shiro's caching needs in shiro.ini
[main] sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO securityManager.sessionManager.sessionDAO = $sessionDAO cacheManager = org.apache.shiro.cache.ehcache.EhcacheManager securityManager.cacheManager = $cacheManager

最后一行，securityManager.cacheManager = $cacheManager，为所有 Shiro 的需要配置了一个 CacheManager。该 CacheManager 实例会自动地直接传送到 SessionDAO（通过 EnterpriseCacheSessionDAO 实现 CacheManagerAware 接口的性质）。

然后，当 SessionManager 要求 EnterpriseCacheSessionDAO 去持久化一个 Session 时，它使用一个 EHCache 支持的 Cache 实现去存储 Session 数据。

Web Applications

当使用 Shiro 本地的 SessionManager 实现时不要忘了分配 SessionDAO 是一项功能。Web 应用程序默认使用基于容器的 SessionManager，它不支持 SessionDAO。如果你想在 Web 应用程序中使用基于 EHCache 的会话存储，配置一个如上所解释的 Web SessionManager。

EHCache Session Cache Configuration

默认地，EhCacheManager 使用一个 Shiro 特定的 ehcache.xml 文件来建立 Session 缓存区以及确保 Sessions 正常存取的必要设置。

然而，如果你想改变缓存设置，或想配置你自己的 ehcache.xml 或 EHCache net.sf.ehcache.CacheManager 实例，你需要配置缓存区来确保 Sessions 被正确地处理。

如果你查看默认的 ehcache.xml 文件，你会看到接下来的 shiro-activeSessionCache 缓存配置：

```
<cache name="shiro-activeSessionCache"
       maxElementsInMemory="10000"
       overflowToDisk="true"
       eternal="true"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
       diskPersistent="true"
       diskExpiryThreadIntervalSeconds="600"/>
```

如果你希望使用你自己的 ehcache.xml 文件，那么请确保你已经为 Shiro 所需的定义了一个类似的缓存项。很有可能你会改变 maxElementsInMemory 的属性值来吻合你的需要。然而，至少下面两个存在于你自己配置中的属性是非常重要的：

- overflowToDisk="true" - 这确保当你溢出进程内存时，会话不丢失且能够被序列化到磁盘上。
- eternal="true" - 确保缓存项（Session 实例）永不过期或被缓存自动清除。这是很有必要的，因为 Shiro 基于计划过程完成自己的验证。如果我们关掉这项，缓存将会在 Shiro 不知道的情况下清扫这些 Sessions，这可能引起麻烦。

EHCache Session Cache Name

默认地，EnterpriseCacheSessionDAO 向 CacheManager 寻求一个名为"shiro-activeSessionCache"的 Cache。该缓存的 name/region 将在 ehcache.xml 中配置，如上所述。

如果你想使用一个不同的名字而不是默认的，你可以在 EnterpriseCacheSessionDAO 上配置名字，例如：

Configuring the cache name for Shiro's active session cache in shiro.ini

```
[main]
…
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO.activeSessionsCacheName = myname
…
```

只要确保在 ehcahe.xml 中有一项与名字匹配且你已经配置好了如上所述的 overflowToDisk="true"和 eternal="true"。

Custom Session IDs

Shiro 的 SessionDAO 实现使用一个内置的 SessionIdGenerator 组件来产生一个新的 Session ID 当每次创建一个新的会话的时候。该 ID 生成后，被指派给新近创建的 Session 实例，然后该 Session 通过 SessionDAO 被保存下来。

默认的 SessionIdGenerator 是一个 JavaUuidSessionIdGenerator，它能产生基于 Java UUIDs 的 String IDs。该实现能够支持所有的生产环境。

如果它不符合你的需要，你可以实现 SessionIdGenerator 接口并在 Shiro 的 SessionDAO 实例上配置该实现。例如，在 shiro.ini 中：

Configuring a SessionIdGenerator in shiro.ini

```
[main]
…
sessionIdGenerator = com.my.session.SessionIdGenerator
securityManager.sessionManager.sessionDAO.sessionIdGenerator = $sessionIdGenerator
```

Session Validation & Scheduling

Sessions 必须被验证，这样任何无效(过期或停止)的会话能够从会话数据存储中删除。这保证了数据存储不会由于不能再次使用的会话而导致写入超时。

由于性能上的原因，仅仅在 Sessions 被访问（也就是 subject.getSession()）时验证它们是否停止或过期。这意味着，如果没有额外的定期验证，Session orphans(孤儿)将会开始填充会话数据存储。

一个常见的说明孤儿的例子是 Web 浏览器中的场景：比方说，用户登录到 Web 应用程序并创建了一个会话来保留数据（身份验证状态，购物车等）。如果用户不注销，并在应用程序不知道的情况下关闭了浏览器，则他们的会话实质上是“躺在”会话数据存储的（孤儿）。SessionManager 没有办法检测用户不再使用他们的浏览器，同时该会话永远不会被再次访问（它是孤儿了）。

会话孤儿，如果它们没有定期 清除，将会填充会话数据存储（这是很糟糕的）。因此，为了防止丢放孤儿，SessionManager 实现支持 SessionValidationScheduler 的概念。SessionValidationScheduler 负责定期地验证会话以确保它们是否需要清理。

Default SessionValidationScheduler

默认可用的 SessionValidationScheduler 在所有环境中都是 ExecutorServiceSessionValidationScheduler，它使用 JDK ScheduledExecutorService 来控制验证频率。

默认地，该实现每小时执行一次验证。你可以通过指定一个新的 ExecutorServiceSessionValidationScheduler 实例并指定不同的间隔（以毫秒为单位）改变速率来更改验证频率：

ExecutorServiceSessionValidationScheduler interval in shiro.ini

```
[main]
…
sessionValidationScheduler = org.apache.shiro.session.mgt.ExecutorServiceSessionValidationScheduler
# Default is 3,600,000 millis = 1 hour:
sessionValidationScheduler.interval = 3600000

securityManager.sessionManager.sessionValidationScheduler = ￥sessionValidationScheduler
```

Custom SessionValidationScheduler

如果你希望提供一个自定义的 SessionValidationScheduler 实现，你可以指定它作为默认的 SessionManager 实例的一个属性。例如，在 shiro.ini 中：

Configuring a custom SessionValidationScheduler in shiro.ini

```
[main]
…
sessionValidationScheduler = com.foo.my.SessionValidationScheduler
securityManager.sessionManager.sessionValidationScheduler = $sessionValidationScheduler
```

Disabling Session Validation

在某些情况下，你可能希望禁用会话验证项，由于你建立了一个超出了 Shiro 控制的进程来为你执行验证。例如，也许你正在使用一个企业的 Cache 并依赖于缓存的 Time To Live 设置来自动地去除旧的会话。或者也许你已经制定了一个计划任务来自动清理一个自定义的数据存储。在这些情况下你可以关掉 session validation scheduling：

Disabling Session Validation Scheduling in shiro.ini

```
[main]
…
securityManager.sessionManager.sessionValidationSchedulerEnabled = false
```

当会话从会话数据存储取回数据时它仍然会被验证，但这会禁用掉 Shiro 的定期验证。

Enable Session Validation *somewhere*
如果你关闭了 Shiro 的 session validation scheduler，你必须通过其他的机制（计划任务等）来执行定期的会话验证。这是保证会话孤儿不会填充数据存储的唯一方法。

Invalid Session Deletion
正如我们上面所说的，进行定期的会话验证主要目的是为了删除任何无效的（过期或停止）会话来确保它们不会填充会话数据存储。

默认地，某些应用程序可能不希望 Shiro 自动地删除会话。例如，如果一个应用程序已经提供了一个 SessionDAO 备份数据存储查询，也许是应用程序团队希望旧的或无效的会话在一定的时间内可用。这将允许团队对数据存储运行查询来判断，例如，在上周某个用户创建了多少个会话，或一个用户会话的持续时间，或与之类似报告类型的查询。

在这些情形中，你可以关闭 invalid session deletion 项。例如，在 shiro.ini 中：

Disabling Invalid Session Deletion ini shiro.ini
[main] … securityManager.sessionManager.deletInvalidSessions = false

请注意！如果你关闭了它，你得为确保你的会话数据存储不耗尽它的空间复杂。你必须自己从你的数据存储中删除无效的会话！

还要注意，即使你阻止了 Shiro 删除无效的会话，你仍然应该使用某种会话验证方式——要没通过 Shiro 的现有验证机制，要么通过一个你自己提供的自定义的机制（见上述的"Disabling Session Validation"获取更多）。验证机制将会更新你的会话记录以反映无效的状态（例如，什么时候它是无效的，它最后一次被访问是什么时候，等等），即使你在其他的一些时间将手动删除它们。

如果你配置 Shiro 来让它不会删除无效的会话，你得为确保你的会话数据存储不会耗尽它的空间负责。你必须亲自从你的数据存储删除无效的会话！
另外请注意，禁用会话删除并不等同于禁用 session validation schedule（会话验证调度）。你应该总是使用一个会话验证调度机制——无论是 Shiro 直接支持或者是你自己的。

Sessions and subject State

Stateful Applications(Sessions allowed)
默认地，Shiro 的 SecurityManager 实现使用一个 Subject 的 Session 作为一种策略来为接下来的引用存储 Subject 的身份 ID（PrincipalCollection）和验证状态（subject.isAuthenticated()）。这通常发生在一个 Subject 登录后或当一个 Subject 的身份 ID 通过 Remember 服务被发现后。

下面是使用这种默认方式的好处：

- 任何服务于请求，调用或消息的应用程序可以用请求/调用/消息的有效载荷关联会话 ID，且这是 Shiro 用入站请求关联用户所有所必须的。例如，如果使用 Subject.Builder，这是需要获取相关的 Subject 所需的一切：

```
Serializable sessionId = //get from the inbound request or remote method invocation payload
Subject requestSubject = new Subject.Builder().sessionId(sessionId),buildSubject();
```

  这给大多数 Web 应用程序及任何编写远程处理或消息框架的人带来了令人难以置信的方便(这事实上是 Shiro 的 Web 支持在自己的框架代码内关联 Subject 和 ServletRequest）。

- 任何"RememberMe"身份基于一个能够在第一次访问就能持久化到会话的初始请求。这确保了 Subject 被记住的身份可以跨请求保存而不需要反序列化及将它解释到每个请求。例如，在一个 Web 应用程序中，没有必要去读取每一个请求的加密 RememberMe Cookie，如果该身份在会话中是已知的。这可是一个很好的性能提升。

Stateless Applications(Sessionless)
虽然上述的默认策略对于大多数应用程序而言是很好的（通常是可取的），但这对于尝试尽可能无状态的应用程序来说是不合适的。许多无状态的架构规定在请求中不能存在持久状态，这种情况下的 Sessions 不会被允许（一个会话其本质代表了持久状态）。

但这一要求带来一个便利的代价——Subject 状态不能跨请求保留。这意味着有这一要求的应用程序必须确保 Subject 状态可以在每一个请求中以其他的方式代表。

这几乎总是通过验证每个由应用程序处理的请求/调用/消息来完成的。例如，大多数无状态 Web 应用程序通常支持这一点通过执行 HTTP 基本验证，允许浏览器验证每一个代表最终用户的请求。远程或消息框架必须确保 Subject 的身份和凭证连接到每一个调用或消息的有效载荷，通常是由框架代码执行。

Disabling Subject State Session Storage
在 Shiro 1.2 及以后开始，应用程序想禁用 Shiro 的内部实现策略——将 Subject 状态持久化到会话，可以禁用所有 Subject 的这一项，通过下面的操作：

在 shiro.ini 中，在 securityManager 上配置下面的属性：

shiro.ini
[main] … securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false

这将防止 Shiro 使用 Subject 的会话来存储所有跨请求/调用/消息的 Subject 状态。只要确保你对每个请求进行了身份验证，这样 Shiro 将会对给定的请求/调用/消息知道它的 Subject 是谁。

Shiro's Needs vs. Your Needs
使用 Sessions 作为存储策略将禁用 Shiro 本身的实现。它没有完全地禁用 Sessions。如果你的任何代码显式地调用 subject.getSession()或 subject.getSession(true)，一个 session 仍然会被创建。

A Hybrid Approach
上面的 shiro.ini 配置中的(securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false)这一行将会禁用 Shiro 为所有的 Subject 使用 Session 作为一种实现策略。

但，如果你想使用混合的方法呢？如果某些对象应该有会话而某些没有？这种混合法方法能够给许多应用程序带来好处。例如：

- 也许 human Subject（如 Web 浏览器用户）由于上面提供的好处能够使用 Session。
- 也许 non-human Subject（如 API 客户端或第三方应用程序）不应该创建 session 由于它们与软件的交互可能会间歇或不稳定。
- 也许所有某种确定类型的 Subject 或从某一确定位置访问系统的应该将状态保持在会话中，但所有其他的不应该。

如果你需要这个混合方法，你可以实现一个 SessionStorageEvaluator。

SessionStorageEvaluator

在你想究竟控制哪个 Subject 能够在它们的 Session 中保存它们的状态的情况下，你可以实现 org.apache.shiro.mgt.SessionStorageEvaluator 接口，并告诉 Shiro 哪个 Subject 支持会话存储。

该接口只有一个方法：

SessionStorageEvaluator

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

关于更详细的 API 说明，请参见 SessionStorageEvaluator 的 JavaDoc。

你可以实现这一接口，并检查 Subject，为了你可能做出这一决定的任何信息。

Subject Inspection

但实现 isSessionStorageEnabled(subject)接口方法时，你可以一直查看 Subject 并访问任何你需要用来作出决定的东西。当然所有期望的 Subject 方法都是可用的（gePrincipals()等），但特定环境的 Subject 实例也是有价值的。

例如，在 Web 应用程序中，如果该决定必须基于当前 ServletRequest 中的数据，你可以获取该 request 或该 response，因为运行时的 Subjce 实例实际上就是一个 WebSubject 实例：

```
…
public boolean isSessionStorageEnabled(Subject subject) {
        boolean enabled = false;
        if(WebUtils.isWeb(Subject)) {
                HttpServletRequest request = WebUtils.getHttpRequest(subject);
                //set 'enabled' based on the current request.
        } else {
                //not a web request - maybe a RMI or daemon invocation?
                //set 'enabled' another way …
        }
        return enabled;
}
```

N.B.框架开发人员应该考虑到这种类型的访问，并确保任何请求/调用/消息上下文对象可用是同过特定环境下的 Subject 实现的。联系 Shiro 用户邮件列表，如果你想帮助设置它，为了你的框架/环境。

Configuration

在你实现了 SessionStorageEvaluator 接口后，你可以在 shiro.ini 中配置它：

shiro.ini SessionStorageEvaluator configuration

```
[main]
…
sessionStorageEvaluator = com.mycompany.shiro.subject.mgt.MySessionStorageEvaluator
securityManager.subjectDAO.sessionStorageEvaluator = $sessionStorageEvaluator


…
```

Web Applications

通常 Web 应用程序希望在每一个请求的基础上容易地启用或禁用会话的创建，不管是哪个 Subject 正在执行请求。这经常在支持 REST 及 Messaging/RMI 构架上使用来产生很好的效果。例如，也许正常的终端用户（使用浏览器的人）被允许创建和使用会话，但远程的 API 客户端使用 REST 或 SOAP，不该拥有会话（因为它们在每一个请求上验证，常见 REST/SOAP 体系结构）。

为了支持这种 hybrid/per-request 的能力，noSessionCreation 过滤器被添加到 Shiro 的默认为 Web 应用程序启用的“池”。该过滤器将会阻止在请求期间创建新的会话来保证无状态的体验。在 shiro.ini 的[urls]项中，你通常定义该过滤器在所有其它过滤器之前来确保会话永远不会被使用。

例如：

shiro.ini - Disable Session Creation per request
[urls] … /rest/** = noSessionCreation, authcBasic, ...

这个过滤器允许现有会话的任何会话操作，但不允许在过滤的请求创建新的会话。也就是说，一个请求或没有会话存在的 Subject 调用下面四个方法中的任何一个时，将会自动地触发一个 DisabledSessionException 异常：

- httpServletRequest.getSession()
- httpServletRequest.getSession(true)
- subject.getSession()
- subject.getSession(true)

如果一个 Subject 在访问 noSessionCreation-protected-URL 之前已经有一个会话，则上述的四种调用仍然会如预期般工作。

最后，在所有情况下，下面的调用将始终被允许：

- httpServletRequest.getSession(false)
- subject.getSession(false)

Web

Configuration(配置)

将 Shiro 集成到任何 Web 应用程序的最简单的方法是在 web.xml 中配置 ContextListener 和 Filter，理解如何读取 Shiro 的 INI 配置文件。大部分的 INI 配置格式定义在 Configuration 页的 INI Sections 节，但我在这里我们将介绍一些额外的 Web 的特定部分。

Using Spring? Spring 框架用户　　将不执行此设置。如果你使用 Spring，你将要阅读关于 Spring 特定的 Web 配置。

Web.xml

Shiro 1.2 and later

在 Shiro 1.2 及以后版本，标准的 Web 应用程序通过添加下面的 XML 块到 web.xml 来初始化 Shiro：

```
<listener>
        <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>
</listener>

…

<filter>
        <filter-name>ShiroFilter</filter-name>
        <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>

<filter-mapping>
        <filter-name>ShiroFilter</filter-name>
        <url-pattern>*/</url-pattern>
</filter-mapping>
```

这假设一个 Shiro INI 配置文件在以下两个位置任意一个，并使用最先发现的那个：

1. /WEB-INF/shiro.ini
2. 在 classpath 根目录下 shiro.ini 文件

下面是上述配置所做的事情：

- EnvironmentLoaderListener 初始化一个 Shiro WebEnvironment 实例（其中包含 Shiro 需要的一切操作，包括 SecurityManager），使得它在 ServletContext 中能够被访问。如果你需要在任何时候获得 WebEnvironment 实例，你可以调用 WebUtils.getRequiredWebEnvironment（ServletContext）。
- ShiroFilter 将使用此 WebEnvironment 对任何过滤的请求执行所有必要的安全操作。
- 最后，filter-mapping 的定义确保了所有的请求被 ShiroFilter 过滤，建议大多数 Web 应用程序使用以确保任何请求是安全的。

ShiroFilter filter-mapping
它通常是可取的在任何其他 filter-mapping 声明之前定义 ShiroFilter filter-mapping，以确保 Shiro 也能在那些过滤器下工作的很好。

Custom WebEnvironment Class

默认情况下，EnvironmentLoaderListener 将创建一个 IniWebEnvironment 实例，呈现 Shiro 基于 INI 文件的配置。如果你愿意，你可以在 web.xml 中指定一个自定义的 ServletContext context-param：

```
<context-param>
        <param-name>shiroEnvironmentClass</param-name>
        <param-value>com.foo.bar.shiro.MyWebEnvironment</param-value>
</context-param>
```

这允许你自定义一个如何解析和代表 WebEnvironment 实例的配置格式。你可以为自定义的行为对现有的 IniWebEnvironment 创建子类，或完全支持不同的配置格式。例如，如果有人想在 XML 中配置 Shiro 而不是在 INI 中，他们可以创建一个基于 XML 的实现，如 com.foo.bar.shiro.XmlWebEnviroment。

Custom Configuration Locations

IniWebEnvironment 将会去读取和加载 INI 配置文件。默认情况下，这个类会自动地在下面两个位置寻找 Shiro.ini 配置（按顺序）。

1. /WEB-INF/shiro.ini
2. classpath:shiro.ini

它将使用最先发现的那个。

然而，如果你想把你的配置放在另一位置，你可以在 web.xml 中用 contex-param 指定该位置。

```
<context-param>
        <param-name>shiroConfigLocations</param-name>
        <param-value>YOUR_RESOURCE_LOCATION_HERE</param-value>
</context-param>
```

默认情况下，在 ServletContext.getResource 方法定义的规则下，param-value 是可以被解析的。例如，/WEB-INF/some/path/shiro.ini。

但你也可以指定具体的文件系统，如 classpath 或 URL 位置，通过使用 Shiro 支持的合适的资源前缀，例如：

- file://home/foobar/myapp/shiro.ini
- classpath:com/foo/bar/shiro.ini
- url:http://confighost.mycompany.com/myapp/shiro.ini

Shiro 1.1 and earlier

在 Web 应用程序中使用 Shiro 1.1 或更早版本的最简单的方法是定义 IniShiroFilter 并指定一个 filter-mapping:

```
<filter>
        <filter-name>ShiroFilter</filter-name>
        <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
</filter>

…

<!-- Make sure any request you want accessible to Shiro is filtered. /* catches all -->
<!-- requests.    Usually this filter mapping is defined first (before all others) to
-->
<!-- ensure that Shiro works in subsequent filters in the filter chain:
-->
<filter-mapping>
        <filter-name>ShiroFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

该定义期望你的 INI 配置是一个在 classpath 根目录的 Shiro.ini 文件（如：classpath:shiro.ini）。

Custom Path

如果你不想将你的 INI 配置放在/WEB-INF/shiro.ini 或 classpath:shiro.ini，你可以指定一个自定义的资源位置，如果必要的话。添加一个 configPath 的 init-param，并指定资源位置。

```
<filter>
        <filter-name>ShiroFilter</filter-name>
        <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
        <init-param>
                <param-name>configPath</param-name>
                <param-value>/WEB-INF/anotherFile.ini</param-value>
        <init-param>
</filter>
...
```

不合格的（不完整的组合或'non-prefixed'）configPath 值被假定为 ServletContext 的资源路径，通过 ServletContext.getResource 方法所定义的规则来解析。

ServletContext resource paths - Shiro 1.2+
ServletContext 资源路径是一个在 Shiro 1.2 即将推出的功能。在 1.2 被发布后，所有的 configPath 定义必须指定一个 classpath:，file:或 url:前缀。

通过分别地使用 classpath:，url:，或 file:前缀来指明 classpath，url，或 filesystem 位置，你也可以指定其他非 ServletContext 资源位置。例如：

```
...
<init-param>
        <param-name>configPath</param-name>
        <param-value>url:http://configHost/myApp/shiro.ini</param-value>
</init-param>
...
```

Inline Config
最后，也可以将你的 INI 配置嵌入到 web.xml 中而不使用一个独立的 INI 文件。你可以通过使用 init-param 做到这点，而不是 configPath：

```
<filter>
        <filter-name>ShiroFilter</filter-name>
        <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
        <init-param>
                <param-name>config</param-name>
                <param-value>

                        # INI Config Here

                      </param-value>
        </init-param>
</filter>
...
```

内嵌配置对于小型的或简单的应用程序通常是很好用的，但是由于以下原因一般把它具体化到一个专用的 Shiro.ini 文件中：

- 你可能编辑了许多安全配置，不希望为 web.xml 添加版本控制。
- 你可能想从余下的 web.xml 配置中分离安全配置。
- 你的安全配置可能变得很大，你想保持 web.xml 的苗条并易于阅读。
- 你有个负责的编译系统，相同的 shiro 配置可能需要在多个地方被引用。

这取决于你——使用什么使你的项目更有意义。

Web INI configuration

除了在主要的 Configuration 章节描述的标准的[main]，[user]和[roles]项外，你可以在 shiro.ini 文件中指定具有 web 特性的[urls]项：

```
# [main], [users] and [roles] above here
…
[urls]
…
```

[urls]项允许你做一些在我们已经见过的任何 Web 框架都不存在的东西：在你的应用程序中定义自适应过滤器链来匹配 URL 路径！

这将更为灵活，功能更为强大，比你通常在 web.xml 中定义的过滤器链更为简洁：即使你从未使用任何 Shiro 提供的其他功能并仅仅使用了这个，但它即使是单独使用也是值得的。

[urls]

在 urls 项的每一行格式如下：
URL_Ant_Path_Expression = Path_Specific_Filter_Chain

例如：

```
…
[urls]

/index.html = anon
/user/create = anon
/user/** = authc
/admin/** = authc, roles[administrator]
/rest/** = authc, rest
/remoting/rpc/** = authc, perms["remot:invoke"]
```

接下来我们将讨论这些行的具体含义。

URL Path Expressions

等号左边是一个与 Web 应用程序上下文根目录相关的 Ant 风格的路径表达式。

例如，假设你有如下的[urls]行：

```
/account/** = ssl, authc
```

此行表明，“任何对我应用程序的/accout 或任何它的子路径（/account/foo, account/bar/baz，等等）的请求都将触发'ssl, authc'过滤器链”。我们将在下面讨论过滤器链。

请注意，所有的路径表达式都是相对于你的应用程序的上下文根目录而言的。这意味着如果某一天你在某个位置部署了你的应用程序，如 www.somehost.com/myapp ，然后又将它部署到了 www.anotherhost.com （没有'myapp'子目录），这样的匹配模式仍将继续工作。所有的路径都是相对于 HttpServletRequest.getContextPath()的值来的。

Order Matters!
URL 路径表达式按事先定义好的顺序判断传入的请求，并遵循 FIRST MATCH WINS 这一原则。例如，让我们假设有如下链的定义：

```
/account/** = ssl, authc
/account/signup = anon
```

如果传入的请求旨在访问/account/signup/index.html（所有'anon'ymous 用户都能访问），那么它将永不会被处理！原因是因为/account/**的模式第一个匹配了传入的请求，“短路”了其余的定义。

始终记住基于 FIRST MATCH WINS 的原则定义你的过滤器链！

Filter Chain Definitions
等号右边是逗号隔开的过滤器列表，用来执行匹配该路径的请求。它必须符合以下格式：
filter1[optional_config1], filter2[optional_config2], ..., filterN[optional_configN]

并且：

- filterN 是一个定义在[main]项中的 filter bean 的名字
- [optional_configN]是一个可选的括号内的对特定的路径，特定的过滤器有特定含义的字符串（每个过滤器，每个路径的具体配置！）。若果该过滤器对该 URL 路径并不需要特定的配置，你可以忽略括号，于是 filteNr[] 就变成了 filterN.

因为过滤器标志符定义了链（又名列表），所以请记住顺序问题！请按顺序定义好你的逗号分隔的列表，这样请求就能够流通这个链。

最后，每个过滤器按照它期望的方式自由的处理请求，即使不具备必要的条件（例如，执行一个重定向，响应一个 HTTP 错误代码，直接渲染等）。否则，它有可能允许该请求继续通过这个过滤器链到达最终的视图。

确保能够对路径的具体配置作出反应，即[optional_configN]一个过滤器标志符的一部分，是一个对所有 Shiro 过滤器适用的独有的功能。
你也可以这样做，如果你想创建你自己的 javax.servlet.Filter 的实现的话，确保你的过滤器子类 org.apache.shiro.web.filter.PathMatchingFilter。

Available Filters
在过滤器链中能够使用的过滤器“池”被定义在[main]项。在[main]项中指派给它们的名字就是在过滤器链定义中使用的名字。例如：

```
[main]
```

```
…
myFilter = com.company.web.some.FilterImplementation
myFilter.property1 = value1
…
[urls]
…
/some/path/** = myFilter
```

Default Filters

当运行一个 Web 应用程序时，Shiro 将会创建一些有用的默认 Filter 实例，并自动地在[main]项中将它们置为可用。你可以在 main 中配置它们，当作在你的链的定义中你是否有任何其他的 bean 和 reference。例如：

```
[main]
…
# Notice how we didn't define the class for the FormAuthenticationFilter('authc')  - it is instantiated and available already:
authc.loginUrl = /login.jsp
…
[urls]
…
# make sure the end-user is authenticated. If not, redirect to the 'authc.loginUrl' above,
# and after successful authentication, redirect them back to the original account page they
# were trying to view:
/account/** = authc
...
```

自动地可用的默认的 Filter 实例是被 DefaultFilter 枚举定义的，枚举的名称字段是可供配置的名称。它们是：

Filter Name	Class
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter
perms	org.apache.shiro.web.filter.authz.PermissionAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authz.UserFilter

Shiro 1.2
'logout'和'noSessionCreation'过滤器仅仅在 SVN trunk/snapshot builds 使用。当 Shiro 1.2 发布的时候它们将会可用。

Enabling and Disabling Filters

Upcoming Feature, Not Yet Released
该功能还没有被发布。它现在仅在 Shiro 1.2.0-SNAPSHOT builds 中可用。如果你需要该功能，你需要从 Shiro 的 subversion trunk 编译，或者使用一个 SNAPSHOT 编译直到 1.2.0 final 发布。

由于这是与任何过滤器链定义机制（web.xml，Shiro 的 INI 等）相关的例子，你通过在过滤器链中包含它来启用过滤器，通过在过滤器链中移除它来禁用过滤器。

但在 Shiro 1.2 中新增的一个新功能是不通过从过滤器链中移除过滤器来启用或禁用过滤器。如果启用（默认设置），那么请求将如预期一样过滤。如果禁用，那么该过滤器将允许请求立即通过到 FilterChain 的下一个元素。你可以基于一般配置属性触发过滤器的启用状态，或者你甚至可以在每一个请求的基础上触发。

这是一个强大的概念，因为基于特定需求启用或禁用一个过滤器比更改静态过滤器链（这是永久的且固定的）定义更为方便。

Shiro 通过它的 OncePerRequestFilter 抽象父类来完成这点。所有 Shiro 的不受规范限制的过滤器实现子类实现这一点，因此不需要从过滤器链移除它们实现启用或禁用。如果你需要实现此功能，你可以为自己的过滤器实现继承这个类的子类。

SHIRO-224 将有望为任何过滤器使用这项功能，不仅仅只是那些 OncePerRequestFilter 的子类。如果这对你很重要，请为这个 issue 投票。

General Enabling/Disabling
OncePerRequestFilter（及其所有子类）支持 Enabling/Disabling 所有请求及 per-request 基础。

一般为所有的请求启用或禁用一个过滤器是通过设置其 enabled 属性为 true 或 false。默认的设置是 true 由于大多数过滤器本质上是需要执行的，如果他们被配置在一个过滤器链中。

例如，在 shiro.ini 中：

```
[main]
…
# configure Shiro's default 'ssl' filter to be disabled while testing:
ssl.enabled = false

[urls]
…
/some/path = ssl, authc
/another/path = ssl, roles[admin]
…
```

该例表明，许多潜在的 URL 路径都需要请求必须通过 SSL 连接保证。在开发中设置 SSL 是令人沮丧且费时的。在开发时，你可以禁用 ssl 过滤器。当部署产品时，你可以启用它通过一个配置属性——这比手动更改所有 URL 路径或维护两个 Shiro 配置要容易得多。

Request-specific Enabling/Disabling
OncePerRequestFilter 实际上决定过滤器启用或禁用是基于它的 isEnabled(request, response)方法。

该方法默认返回 enabled 属性的值，该属性通常是用来 enabling/disabling 上面提及的所有请求。如果你想启用或禁用一个基于特定标准的请求的过滤器，你可以通过覆盖 OncePerRequestFilter 的 isEnabled(request, response)方法来执行更多特定的检查。

Path-specific Enabling/Disabling
Shiro 的 PathMatchingFilter（一个 OncePerRequestFilter 的子类）能够对基于被过滤的特定路径的配置作出反应。这意味着你可以启用或禁用一个过滤器基于路径和特定路径配置，除了传入的 request 和 response。

如果你需要能够对匹配的路径和特定路径配置作出反应来判断一个过滤器是否是启用的或禁用的，而不是通过覆盖 OncePerRequestFilter 的 isEnabled(request, reponse)方法，你应该是覆盖 PathMatchingFilter 的 isEnabled(request, response)方法。

Remember Me Services
Shiro 将执行'rememberMe'服务如果 AuthenticationToken 实现了 org.apache.shiro.authc.RememberMeAuthenticationToken 接口。该接口指定了一个方法：

```
boolean isRememberMe();
```

如果该方法返回 true，Shiro 将会在整个会话中记住终端用户的身份 ID。

UsernamePasswordToken and RememberMe
经常使用的 UsernamePasswordToken 已经实现了 RememberMeAuthenticationToken 接口，并支持 rememberMe 登录。

Programmatic Support
要有计划性地使用 rememberMe，你可以在一个支持该配置的类上把它的值设为 true。例如，使用标准的 UsernamePasswordToken：

```
UsernamePasswordToken token = new UsernamePasswordToken(username, password);
token.setRememberMe(true);
SecurityUtils.getSubject().login(token);
…
```

Form-based Login
对于 Web 应用程序而言，authc 过滤器默认是 FormAuthenticationFilter。它支持将'rememberMe'的布尔值作为一个 form/request 参数读取。默认地，它期望该 request 参数被命名为 rememberMe。下面是一个支持这点的 shiro.ini 配置的例子：

```
[main]
authc.loginUrl = /login.jsp

[urls]
```

```
# your login form page here:
login.jsp = authc
```

同时在你的 web form 中有一个名为'rememberMe'的 checkbox。

```
<form ...>
        Username: <input type="text" name="username"/> <br/>
        Password:<input type="password" name="password"/>
        …
        <input type="checkbox" name="rememberMe" value="true"/>Remember Me?
        …
</form>
```

默认地，FormAuthenticationFilter 将会寻找名为 username，password 及 rememberMe 的 request 参数。如果这些不同于你使用的 form 中的表单域名，你可能想在 FormAuthenticationFilter 上配置这些参数名。例如，在 shiro.ini 中：

```
[main]
…
authc.loginUrl = /whatever.jsp
authc.usernameParam = somethingOtherThanUsername
authc.passswordParam = somthingOtherThanPassword
authc.rememberMeParam = somethingOtherThanRememberMe
…
```

Cookie configuration

你可以通过设定{{RememberMeManager}}的各方面的 cookie 属性来配置 rememberMe cookie 是如何工作的。例如，在 shiro.ini 中：

```
[main]
…
securityManager.rememberMeManager.cookie.name = foo
securityManager.rememberMeManager.cookie.maxAge = blah
…
```

请参见 CookieRememberMeManager 及 SimpleCookie 的 JavaDoc 支持来获取更多的配置属性。

Custom RememberMeManager

应该注意到，默认基于 cookie 的 RememberMeManager 实现不符合你的需求，你可以插入任何你喜欢的插件到 securityManager 当中，就像你配置任何其他对象的引用一样：

```
[main]
…
rememberMeManager = com.my.impl.RememberMeManager
securityManager.rememberMeManager = $rememberMeManager
```

JSP/GSP Tag Library

Apache Shiro 提供了一个 Subject-aware JSP/GSP 标签库，它允许你控制你的 JSP，JSTL 或 GSP 页面基于当前 Subject 的状态进行输出。这对于根据身份个性化视图及当前用户所浏览的页面授权状态是相当有用的。

Tag Library Configuration

标签库描述文件(TLD)被打包在 META-INF/shiro.tld 文件中的 shiro-web.jar 文件中。要使用任何标签，添加下面一行到你 JSP 页面（或任何你定义的页面指令）的顶部。

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
```

我们使用 shiro 前缀用以表明 shiro 标签库命名空间，当然你可以指定任何你喜欢的名字。

现在我们将讨论每一个标签，并展示它是如何用来渲染页面的。

The guest tag

guest 标签将显示它包含的内容，仅当当前的 Subject 被认为是'guest'时。'guest'是指没有身份 ID 的任何 Subject。也就是说，我们并不知道用户是谁，因为他们没有登录并且他们没有在上一次的访问中被记住（RememberMe 服务）。

例子：

```
<shiro:guest>
    Hi there! Please <a href="login.jsp">Login</a> or <a href="signup.jsp">Signup</a>today!
</shiro:guest>
```

guest 标签与 user 标签逻辑相反。

The user tag

user 标签将显示它包含的内容，仅当当前的 Subject 被认为是'user'时。'user'在上下文中被定义为一个已知身份 ID 的 Subject，或是成功通过身份验证及通过'RememberMe'服务的。请注意这个标签在语义上与 authenticated 标签是不同的，authenticated 标签更为严格。

例子：

```
<shiro:user>
    Welcome back John! Not John? Click <a href="login.jsp">here<a> to login.
</shiro:user>
```

usre 标签与 guest 标签逻辑相反。

The authenticated tag

仅仅只当当前用户在当前会话中成功地通过了身份验证 authenticated 标签才会显示包含的内容。它比'user'标签更为严格。它在逻辑上与'notAuthenticated'标签相反。

authenticated 标签只有当当前 Subject 在其当前的会话中成功地通过了身份验证才会显示包含的内容。它比 user 标签更为严格，authenticated 标签通常在敏感的工作流中用来确保身份 ID 是可靠的。

例子：

```
<shiro:authenticated>
        <a href="updateAccount.jsp">Update your contact information</a>.
</shiro:authenticated>
```

authenticated 标签与 notAuthenticated 标签逻辑相反。

The notAuthenticated tag

notAuthenticated 标签将会显示它所包含的内容，如果当前 Subject 还没有在其当前会话中成功地通过验证。

例子：

```
<shiro:notAuthenticated>
        Please <a href="login.jsp">login</a> in order to update your credit card information.
</shiro:notAuthenticated>
```

notAuthenticated 标签与 Authenticated 标签逻辑相反。

The principal tag

principal 标签将会输出 Subject 的主体（标识属性）或主要的属性。
若没有任何标签属性，则标签将使用 principal 的 toString()值来呈现页面。例如（假设 principal 是一个字符串的用户名）：

```
Hello, <shiro:principal/>, how are you today?
```

这（大部分地）等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipal().toString() %>, how are you today?
```

Typed principal

principal 标签默认情况下，假定该 principal 输出的是 subject.getPrincipal()的值。但若你想输出一个不是主要 principal 的值，而是属于另一个 Subject 的 principal collection，你可以通过类型来获取该 principal 并输出该值。

例如，输出 Subject 的用户 ID（并不是 username），假设该 ID 属于 principal collection：

```
User ID: <principal type="java.lang.Integer"/>
```

这（大部分地）等价于下面：

```
User ID: <%= SecurityUtils.getSubject().getPrincipals().oneByType(Integer.class).toString() %>
```

Principal property

但如果该 principal（是默认主要的 principal 或是上面的'typed' principal）是一个复杂的对象而不是一个简单的字符串，而且你希望引用该 principal 上的一个属性该怎么办呢？你可以使用 property 属性来来表示 property 的名称来理解（必须通过 JavaBeans 兼容的 getter 方法访问）。例如（假主要的 principal 是一个 User 对象）：

```
Hello, <shiro:principal property="firstName"/>, how are you today?
```

这（大部分地）等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipal().getFirstName().toString() %>, how are you today?
```

或者，结合类型属性：

```
Hello, <shiro:principal type="com.foo.User" property="firstName"/>, how are you today?
```

这也很大程度地等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipals.oneByType(com.foo.User.class).getFirstName().toString() %>, how are you today?
```

The hasRole tag

hasRole 标签将会显示它所包含的内容，仅当当前 Subject 被分配了具体的角色。

例如：

```
<shiro:hasRole name="administrator">
        <a href="admin.jsp">Administer the system</a>
</shiro:hasRole>
```

hasRole 标签与 lacksRole 标签逻辑相反。

The lacksRole tag

lacksRole 标签将会显示它所包含的内容，仅当当前 Subject 未被分配具体的角色。

例如：

```
<shiro:lacksRole name="administrator">
        Sorry, you are not allowed to administer the system.
</shiro:lacksRole>
```

lacksRole 标签与 hasRole 标签逻辑相反。

The hasAnyRole tag

hasAnyRole 标签将会显示它所包含的内容，如果当前的 Subject 被分配了任意一个来自于逗号分隔的角色名列表中的具体角色。

例如：

```
<shiro:hasAnyRole name="developer, project manager, administrator">
```

```
        You are either a developer, project manager, or administrater.
</shiro:hasAnyRole>
```

hasAnyRole 标签目前还没有与之逻辑相反的标签。

The hasPermission tag
hasPermission 标签将会显示它所包含的内容，仅当当前 Subject“拥有”（蕴含）特定的权限。也就是说，用户具有特定的能力。

例如：

```
<shiro:hasPermission name="user:create">
        <a href="createUser.jsp">Create a new User</a>
</shiro:hasPermission>
```

hasPermission 标签与 lacksPermission 标签逻辑相反。

The lacksPermission tag
lacksPermission 标签将会显示它所包含的内容，仅当当前 Subject 没有拥有（蕴含）特定的权限。也就是说，用户没有特定的能力。

例如：

```
<shiro:lacksPermission name="user:delete">
        Sorry, you are not allowed to deleted user accounts.
</shiro:hasPermission>
```

lacksPermission 标签与 hasPermission 标签逻辑相反。

Lend a hand with documentation
我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Caching

Shiro 开发团队明白在许多应用程序中性能是至关重要的。Caching 是从第一天开始第一个建立在 Shiro 中的一流功能，以确保安全操作保持尽可能的快。

然而，Caching 作为一个概念是 Shiro 的基本组成部分，实现一个完整的缓存机制是安全框架核心能力之外的事情。为此，Shiro 的缓存支持基本上是一个抽象的（包装）API，它将“坐”在一个基本的缓存机制产品（例如，Ehcache，OSCache，Terracotta，Coherence，GigaSpaces，JBossCache 等）之上。这允许 Shiro 终端用户配置他们喜欢的任何缓存机制。

Caching API

Shiro 有三个重要的缓存接口：

- CacheManager - 负责所有缓存的主要管理组件，它返回 Cache 实例。
- Cache - 维护 key/value 对。
- CacheManagerAware - 通过想要接收和使用 CacheManager 实例的组件来实现。

CacheManager 返回 Cache 实例，各种不同的 Shiro 组件使用这些 Cache 实例来缓存必要的数据。任何实现了 CacheManagerAware 的 Shiro 组件将会自动地接收一个配置好的 CacheManager，该 CacheManager 能够用来获取 Cache 实例。

Shiro 的 SecurityManager 实现及所有 AuthorizingRealm 实现都实现了 CacheManagerAware。如果你在 SecurityManager 上设置了 CacheManger，它反过来也会将它设置到实现了 CacheManagerAware 的各种不同的 Realm 上（OO delegation）。例如，在 shiro.ini 中：

example shiro.ini CacheManager configuration
securityManager.realms = $myRealm1, $myRealm2, … $myRealmN … cacheManager = my.implementation.of.CacheManager … # at this point, the securityManager and all CacheManagerAware # realms have been set with the cacheManager instance

我们拥有一个立即可用的 EhCacheManager 实现，因此，如果你想的话，今天都可以使用。相反地，你可以实现自己的 CacheManager（如使用 Coherence 等），并像上面那样配置它，你会取得很好的效果的。

Authorization Cache Invalidation

最后请注意 AuthorizingRealm 有一个 clearCachedAuthorizationInfo 方法能够被子类调用，用来清除特殊账户缓存的授权信息。它通常被自定义逻辑调用，如果与之匹配的账户授权数据发生了改变（来确保下次的授权检查能够捕获新数据）。

Concurrency & Multithreading

TODO:未翻译

Testing with Apache Shiro

文档的这一部分介绍了在单元测试中如何使用 Shiro。

What to know for tests

由于我们已经涉及到了 Subject reference，我们知道 Subject 是“当前执行”用户的特定安全视图，且该 Subject 实例绑定到一个线程来确保我们知道在线程执行期间的任何时间是谁在执行逻辑。

这意味着三个基本的东西必须始终出现，为了能够支持访问当前正在执行的 Subject：

1. 必须创建一个 Subject 实例
2. Subject 实例必须绑定当前执行的线程。
3. 在线程完成执行后（或如果该线程执行抛出异常），该 Subject 必须解除绑定来确保该线程在任何线程池环境中保持'clean'。

Shiro 拥有为正在运行的应用程序自动地执行绑定//解除绑定逻辑的建筑组件。例如，在 Web 应用程序中，当过滤一个请求时，Shiro 的根过滤器执行该逻辑。但由于测试环境和框架不同，我们需要自己选择自己的测试框架来执行此绑定/解除绑定逻辑。

Test Setup
我们知道在创建一个 Subject 实例后，它必须被绑定线程。在该线程（或在这个例子中，是一个 test）完成执行后，我们必须解除 Subject 的绑定来保持线程的'clean'.

幸运的是，现代测试框架如 JUnit 和 TestNG 已经能够在本地支持'setup'和'teardown'的概念。我们可以利用这一支持来模拟 Shiro 在一个“完整的”应用程序中会做些什么。我们已经在下面创建了一个你能够在你自己的测试中使用的抽象基类——随意复制和修改如果你觉得合适的话。它能够在单元测试和集成测试中使用（我在本例中使用 JUnit，但 TestNG 也能够工作得很好）：

AbstractShiroTest

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.subject.support.SubjectThreadState;
import org.apache.shiro.util.LifecycleUtils;
import org.apache.shiro.util.ThreadState;
import org.junit.AfterClass;

/**
 *Abstract test case enabling Shiro in test environments.
 */
public abstract class AbstractShiroTest {
        private static ThreadState subjectThreadState;
        public AbstractShiroTest() {
        }

        /**
         * Allows subclasses to set the currently executing {@link Subject} instance.
         *
         * @param subject the Subject instance
         */
        protected void setSubject(Subject subject) {
                clearSubject();
                subjectThreadState = createThreadState(subject);
                subjectThreadState.bind();
        }

        protected Subject getSubject() {
                return SecurityUtils.getSubject();
```

```
    }

    protected ThreadState createThreadState(Subject subject) {
        return new SubjectThreadState(subject);
    }

    /**
     * Clears Shiro's thread state, ensuring the thread remains clean for future test execution.
     */
    protected void clearSubject() {
        doClearSubject();
    }

    private static void doClearSubject() {
        if (subjectThreadState != null) {
            subjectThreadState.clear();
            subjectThreadState = null;
        }
    }

    protected static void setSecurityManager(SecurityManager securityManager) {
        SecurityUtils.setSecurityManager(securityManager);
    }

    protected static SecurityManager getSecurityManager() {
        return SecurityUtils.getSecurityManager();
    }

    @AfterClass
    public static void tearDownShiro() {
        doClearSubject();
        try {
            SecurityManager securityManager = getSecurityManager();
            LifecycleUtils.destroy(securityManager);
        } catch (UnavailableSecurityManagerException e) {
            //we don't care about this when cleaning up the test environment
            //(for example, maybe the subclass is a unit test and it didn't
            //need a SecurityManager instance because it was using only
            //mock Subject instances)
        }
        setSecurityManager(null);
    }
}
```

Testing & Frameworks

在 AbstractShiroTest 类中的代码使用 Shiro 的 ThreadState 概念及一个静态的 SecurityManager。这些技术在测试和框架代码中是很有用的，但几乎不曾在应用程序代码中使用。

大多数使用 Shiro 工作的需要确保线程的一致性的终端用户，几乎总是使用 Shiro 的自动管理机制，即 Subject.associateWith 和 Subject.execute 方法。这些方法包含在 Subject thread association 参考文献中。

Unit Testing

单元测试主要是测试你的代码，且你的代码是在有限的作用域内。当你考虑到 Shiro 时，你真正要关注的是你的代码能够与 Shiro 的 API 正确的运行——你不会想做关于 Shiro 的实现是否工作正常（这是 Shiro 开发团队在 Shiro 的代码库必须确保的东西）的必要测试。

测试 Shiro 的实现是否与你的实现协同工作是真实的集成测试（下面讨论）。

ExampleShiroUnitTest

由于单元测试适用于测试你的逻辑（而不是你可能调用的任何实现），这对于模拟你逻辑所依赖的任何 API 来说是个很好的主意。这能够与 Shiro 工作得很好——你可以模拟 Subject 实例，并使它反映任何情况下你所需的反应，这些反应是处于测试的代码做出的。

但正如上文所述，在 Shiro 测试中关键是要记住在测试执行期间任何 Subject 实例（模拟的或真实的）必须绑定到线程。因此，我们所需要做的是绑定模拟的 Subject 以确保如预期进行。

（这个例子使用 EasyMock，但 Mockito 也同样地工作得很好）：

```
import org.apache.shiro.subject.Subject;
import org.junit.After;
import org.junit.Test;

import static org.easymock.EasyMock.*;

/**
 * Simple example test class showing how one may perform unit tests for code that requires Shiro APIs
 */
public class ExampleShiroUnitTest extends AbstractShiroTest {
    @Test
    public void testSimple() {
        //1. Create a mock authenticated Subject instance for the test to run:
        Sujbect subjectUnderTest = createNiceMock(Subject.class);
        expect(subjectUnderTest.isAuthenticated()).andReturn(true);

        //2. Bind the subject to the current thread:
        setSubject(subjectUnderTest);

        //perform test logic here. Any call to
        //SecurityUtils.getSubject() directly (or nested in the
        //call stack) will work properly.
    }

    @After
    public void tearDownSubject() {
        //3. Unbind the subject from the current thread:
        clearSubject();
    }
}
```

正如你所看到的，我们没有设立一个 Shiro SecurityManager 实例或配置一个 Realm 或任何像这样的东西。我们简单地创建一个模拟 Subject 实例，并通过调用 setSubject 方法将它绑定到线程。这将确保任何在我们测试代码中的调用或在代码中我们正测试的 SecurityUtils.getSubject()正常工作。

请注意，setSubject 方法实现将绑定你的模拟 Subject 到线程，且它仍将存在，直到你通过一个不同的 Subject 调用 setSubject 或直到你明确地通过调用 clearSubject()将它从线程中清除。

保持 Subject 绑定到该线程多长时间（或在一个不同的测试中用来交换一个新的实例）取决于你及你的测试需求。

tearDownSubject()

在实例中的 tearDownSubject()方法使用了 Junit 4 的注释来确保该 Subject 在每个测试方法执行后被清除，不管发生什么。这要求你设立一个新的 Subject 实例并将它设置到每个需要执行的测试中。

然而这也不是绝对必要的。例如，你可以只每个测试开始时绑定一个新的 Subject 实例（通过 setSubject），也就是说，使用@Before-annotated 方法。但如果你将要这么做，你可以同时使用@After tearDownSubject() 方法来保持对称及'clean'。

你可以手动地在每个方法中混合及匹配该 setup/teardown 逻辑或使用@Before 和@After 注释只要你认为合适。所有测试完成后，AbstractShiroTest 超类在无论怎样都会将 Subject 从线程解除绑定，因为@After 注释在它的 tearDownShiro()方法中。

Integration Testing

现在我们讨论了单元测试的设置，让我们讨论一些关于集成测试的东西。集成测试是指测试跨 API 边界的实现。例如，测试当调用 B 实现时 A 实现是否工作，且 B 实现是否做它该做的事情。

你同样也可以在 Shiro 中轻松地执行集成测试。Shiro 的 SecurityManager 实例及它所包含的东西（如 Realms 和 SessionManager 等）都是占用很少内存的非常轻量级的 POJO。这意味着你可以为每一个你执行的测试类创建并销毁一个 SecurityManager 实例。当你的集成测试运行时，它们将使用“真实的”SecurityManager，且与你应用程序中相像的 Subject 实例将会在运行时使用。

ExampleShiroIntegrationTest

下面的实例代码看起来与上面的单元测试实例几乎相同，但这 3 个步骤却有些不同：

1. 现在有了 step '0'，它用来设立一个“真实的”SecurityManager 实例。
2. Step 1 现在通过 Subject.Builder 构造一个“真实的”Subject 实例，并将它绑定到线程。

线程的绑定与解除绑定（step 2 和 3）与单元测试实例中的作用一样。

```
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.junit.After;
import org.junit.BeforeClass;
import org.junit.Test;

public class ExampleShiroIntegrationTest extends AbstractShiroTest {
    @BeforeClass
    public static void beforeClass() {
        //0. Build and set the SecurityManager used to build Subject instances used in your tests
```

```
        //      This typically only needs to be done once per class if your shiro.ini doesn't change,
        //      otherwise, you'll need to do this logic in each test that is different
        Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:test.shiro.ini");
        setSecurityManager(factory.getInstance());
    }

    @Test
    public void testSimple() {
        //1. Build the Subject instance for the test to run:
        Subject subjectUnderTest = new Subject.Builder(getSecurityManager()).buildSubject();

        //2. Bind the subject to the current thread:
        setSubject(subjectUnderTest);

        //perform test logic here. Any call to
        //SecurityUtils.getSubject() directly (or nested in the
        //call stack) will work properly.
    }

    @After
    public void tearDownSubject() {
        //3. Unbind the subject from the current thread:
        clearSubject();
    }
}
```

正如你所看到的，一个具体的 SecurityManager 实现被实例化，并通过 setSecurityManager 方法使其余的测试能够对其进行访问。然后测试方法能够使用该 SecurityManager，当使用 Subject.Builder 后通过调用 getSecurityManager()方法。

还要注意 SecurityManager 实例在@BeforeClass 设置方法中只被设置一次——一个对于大多数测试类较为普遍的做法。如果你想，你可以创建一个新的 SecurityManager 实例并在任何时候从任何测试方法通过 setSerurityManager 来设置它——例如，你可能会引用两个不同的.ini 文件来构建一个根据你的测试需求而来的新 SecurityManager。

最后，与单元测试例子一样，AbstractShiroTest 超类将会清除所有 Shiro 产物（任何存在的 SecurityManager 及 Subject 实例）通过它的@AfterClass tearDownShiro()方法来确保该线程在下个测试类运行时是'clean'的。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Custom Subjects

Understanding Subjects in Apache Shiro(理解 Apache Shiro 中的 Subject)

毫无疑问，在 Apache Shiro 中最重要的概念就是 Subject。'Subject'仅仅是一个安全术语，是指应用程序用户的特定安全的“视图”。一个 Shiro Subject 实例代表了一个单一应用程序用户的安全状态和操作。

这些操作包括：

- authentication(login)
- authorization(access control)
- session access
- logout

我们原本希望把它称为"User"由于这样“很有意义”，但是我们决定不这样做：太多的应用程序现有的 API 已经有自己的 User classes/frameworks，我们不希望和这些起冲突。此外，在安全领域，"Subject"这一词实际上是公认的术语。

Shiro 的 API 为应用程序提供 Subject 为中心的编程范式支持。当编码应用程序逻辑时，大多数应用程序开发人员想知道谁才是当前正在执行的用户。虽然应用程序通常能够通过它们自己的机制（UserService 等）来查找任何用户，但涉及到安全性时，最重要的问题是“谁才是当前的用户？”。

虽然通过使用 SecurityManager 可以捕获任何 Subject，但只有基于当前用户/Subject 的应用程序代码更自然，更直观。

The Currently Executing Subject(当前执行的 Subject)

几乎在所有环境下，你能够获得当前执行的 Subject 通过使用 org.apache.shiro.SecurityUtils:

```
Subject currentUser = SecurityUtils.getSubject();
```

getSubject()方法调用一个独立的应用程序，该应用程序可以返回一个在应用程序特有位置上基于用户数据的 Subject，在服务器环境中（如，Web 应用程序），它基于与当前线程或传入的请求相关的用户数据上获得 Subject。

当你获得了当前的 Subject 后，你能够拿它做些什么？

如果你想在他们当前的 session 中使事情对用户变得可用，你可得的他们的 session:

```
Session session = currentUser.getSession();
session.setAttribute( "someKey", "aValue" );
```

Session 是一个 Shiro 的具体实例，它提供了大多数你经常要和 HttpSessions 用到的东西，但有一些额外的好处和一个很大的区别：它不需要一个 HTTP 环境！

如果在 Web 应用程序内部部署，默认的 Session 将会是基于 HttpSession 的。但是，在一个非 Web 环境中，像这个简单的 Quickstart，Shiro 将会默认自动地使用它的 Enterprise Session Management。这意味着你可以在你的应用程序中使用相同的 API，在任何层，无论部署环境。这打开了应用程序的全新世界，由于任何需要 session 的应用程序不再被强迫使用 HttpSession 或 EJB Stateful Session Beans。而且，任何客户端技术现在能够共享会话数据。

所以，你现在可以获取一个 Subject 以及他们的 Session。对于真正有用的东西像检查会怎么样呢，如果他们被允许做某些事——如对角色和权限的检查？

嗯，我只能对已知的用户做这些检查。我们的 Subject 实例代表了当前的用户，但谁又是实际上的当前用户呢？呃，他们都是匿名的——也就是说，直到他们至少登录一次。那么，让我们像下面这样做：

```
if ( !currentUser.isAuthenticated() ) {
        //collect user principals and credentials in a gui specific manner
        //such as username/password html form, X509 certificate, OpenID, etc.
        //We'll use the username/password example here since it is the most common.
        //(do you know what movie this is from? :)
        UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");
        //this is all you have to do to support 'remember me' (no config - built in!):
        token.setRememberMe(true);
        currentUser.login(token);
}
```

那就是了！它再简单不过了。

但如果他们的登录尝试失败了会怎么样？你可以捕获各种各样的具体的异常来告诉你到底发生了什么：

```
try {
        currentUser.login( token );
        //if no exception, that's it, we're done!
} catch ( UnknownAccountException uae ) {
        //username wasn't in the system, show them an error message?
} catch ( IncorrectCredentialsException ice ) {
        //password didn't match, try again?
} catch ( LockedAccountException lae ) {
        //account for that username is locked - can't login. Show them a message?
}
        … more types exceptions to check if you want …
} catch ( AuthenticationException ae ) {
        //unexpected condition -    error?
}
```

你，作为应用程序/GUI 开发人员，可以基于异常选择是否显示消息给终端用户（例如，“在系统中没有与该用户名对应的帐户。”）。有许多不同种类的异常供你检查，或者你可以抛出你自己自定义的异常，这些异常可能是 Shiro 还未提供的。有关详情，请查看 AuthenticationException 的 JavaDoc。

好了，现在，我们有了一个登录的用户，我们还有什么可以做的呢？

比方说，他们是谁：

```
//print their identifying principal (in this case, a username);
log.info( "User [" + currentUser.getPrincipal() + "] logged in successfully." );
```

我们还可以测试他们是否有特定的角色：

```
if ( currentUser.hasRole( "schwartz" ) ) {
        log.info( "May the Schwartz be with you!");
} else {
```

```
        log.info( "Hello, more mortal." );
}
```

我们还能够判断他们是否有权限对一个确定类型的实体进行操作：

```
if ( currentUser.isPermitted( "lightsaber:weild" ) ) {
        log.info("You may use a lightsaber ring. Use it wisely.");
} else {
        log.info("Sorry, lightsaber rings are for schwartz masters only.");
}
```

此外，我们可以执行一个非常强大的实例级权限检查——它能够判断用户是否能够访问一个类型的具体实例：

```
if (currentUser.isPermitted( "winnebago:drive:eagle5" )) {
        log.info("You are permitted to 'drive' the 'winnebago' with license plate (id) 'eagle5' .") +
                "Here are the keys -    have fun!");
} else {
        log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
}
```

小菜一碟，对吧？

最后，当用户完成了对应用程序的使用时，他们可以注销：

```
currentUser.logout();           //removes all identifying    information and invalidates their session
too.
```

这个简单的 API 包含了 90%的 Shiro 终端用户在使用 Shiro 时将会处理的东西。

Custom Subject Instances(自定义 Subject 实例)

Shiro 1.0 中添加了一个新特性，能够在特殊情况下构造自定义/临时的 subject 实例。

Special Use Only!
你应该总是通过调用 SubjectUtils.getSubject()来获得当前正在执行的 Subject；
创建自定义的 Subject 实例只应在特殊情况下进行。

当一些“特殊情况”是，这是可以很有用的：

- 系统启动/引导——当没有用户月系统交互时，代码应该作为一个'system'或 daemon 用户来执行。创建 Subject 实例来代表一个特定的用户是值得的，这样引导代码能够以该用户（如 admin）来执行。
  鼓励这种做法是由于它能保证 utility/system 代码作为一个普通用户以同样的方式执行，以确保代码是一致的。这使得代码更易于维护，因为你不必担心 system/daemon 方案的自定义代码块。
- 集成测试——你可能想创建 Subject 实例，在必要时可以在集成测试中使用。请参阅测试文档获取更多的内容。
- Daemon/background 进程的工作——当一个 daemon 或 background 进程执行时，它可能需要作为一个特定的用户来执行。

如果你已经有一个 Subject 的实例，并希望它提供给其他线程，你应该使用 Subject.associateWith*方法，而不是创建一个新的 Subject 实例。

好了，假设你仍然需要创建自定义的 Subject 实例的情况下，让我们看看如何来做：

Subject.Builder
Subject.Builder 被制定得非常容易创建 Subject 实例，而无需知道构造细节。

Builder 最简单的用法是构造一个匿名的，session-less 的实例。

```
Subject subject = new Subject.Builder.buildSubject()
```

上面所展示的默认的 Subject.Builder 无参构造函数将通过 SecurityUtils.getSubject()方法使用应用程序当前可访问的 SecurityManager。你也可以指定被额外的构造函数使用的 SecurityManager 实例，如果你需要的话：

```
SecurityManager securityManager = //acquired from somewhere
Subject subject = new Subject.Builder(securityManager).buildSubject();
```

所有其他的 Subject.Builder 方法可以在 buildSubject()方法之前被调用，它们来提供关于如何构造 Subject 实例的上下文。例如，假如你拥有一个 session ID ，想取得“拥有”该 session 的 Subject（假设该 session 存在且未过期）：

```
Serializable sessionId = //acquired from somewhere
Subject subject = new Subject.Builder().sessionId(sessionId).buildSubject();
```

同样地，如你想创建一个 Subject 实例来反映一个确定的身份：

```
Object userIdentity = //a long ID or String username, or whatever the "myRealm" requires
String realmName = "myRealm";
PrincipalCollection principals = new SimplePrincipalCollection(userIdentity, realmName);
Subject subject = new Subject.Builder().principals(principals).buildSubject();
```

然后，你可以使用构造的 Subject 实例，如预期一样对它进行调用。但请注意：

构造的 Subject 实例不会由于应用程序（线程）的进一步使用而自动地绑定到应用程序（线程）。如果你想让它对于任何代码都能够方便地调用 SecurityUtils.getSubject()，你必须确保创建好的 Subject 有一个线程与之关联。

Thread Association(线程关联)
如上所述，只是构建一个 Subject 实例，并不与一个线程相关联——一个普通的必要条件是在线程执行期间任何对 SecurityUtils.getSubject()的调用是否能正常工作。确保一个线程与一个 Subject 关联有三种途径：

- Automatic Association(自动关联)—— 通过 Sujbect.execute*方法执行一个 Callable 或 Runnable 方法会自动地绑定和解除绑定到线程的 Subject，在 Callable/Runnable 异常的前后。
- Manual Association(手动关联)——你可以在当前执行的线程中手动地对 Subject 实例进行绑定和解除绑定。这通常对框架开发人员非常有用。
- Different Thread(不同的线程)——通过调用 Subject.associateWith*方法将 Callable 或 Runnable 方法关联到 Subject，然后返回的 Callable/Runnable 方法在另一个线程中被执行。如果你需要为 Subject 在另一个线程上执行工作的话，这是首选的方法。

了解线程关联最重要的是，两件事情必须始终发生：

1. Subject 绑定到线程，所以它在线程的所有执行点都是可用的。Shiro 做到这点通过它的 ThreadState 机制，该机制是在 ThreadLocal 上的一个抽象。
2. Subject 将在某点解除绑定，即使线程的执行结果是错误的。这将确保线程保持干净，并在 pooled/reusable 线程环境中清除任何之前的 Subject 状态。

这些原则保证在上述三个机制中发生。接下来阐述它们的用法。

Automatic Association(自动关联)

如果你只需要一个 Subject 暂时与当前的线程相关联，同时你希望线程绑定和清理自动发生，Subject 的 Callable 或 Runnable 的直接执行正是你所需要的。在 Subject.execute 调用返回后，当前线程被保证当前状态与执行前的状态是一样的。这个机制是这三个中使用最广泛的。

例如，让我们假定你有一些逻辑在系统启动时需要执行。你希望作为一个特定用户执行代码块，但一旦逻辑完成后，你想确保线程/环境自动地恢复到正常。你可以通过调用 Subject.execute*方法来做到：

```
Subject subject = //build or acquire subject
subject.execute( new Runnable () {
        public void run() {
                //subject is 'bound' to the current thread now
                //any SecurityUtils.getSubject() calls in any
                //code called from here will work
        }
});
//At this point, the Subject is no longer associated
//with the current thread and everything is as it was before
```

当然，Callable 的实例也能够被支持，所以你能够拥有返回值并捕获异常：

```
Subject subject = //build or acquire subject
MyResult result = subject.execute( new Callable<MyResult>() {
        public MyResult call() throws Exception {
                //subject is 'bound' to the current thread now
                //any SecurityUtils.getSubject() calls in any
                //code called from here will work
                …
                //finish logic as this Subject
                …
                return myResult;
        }
});
//At this point, the Subject is no longer associated
//with the current thread and everything is as it was before
```

这种方法在框架开发中也是很有用的。例如，Shiro 对 secure Spring remoting 的支持确保了远程调用能够作为一个特定的 Subject 来执行：

```
Subject.Builder builder = new Subject.Builder();
//populate the builder's attributes based on the incoming RemoteInvocation
…
Subject subject = builder.buildSubject();

return subject.execute(new Callable() {
        public Object call() throws Exception {
                return invoke(invocation, targetObject);
        }
});
```

Manual Association(手动关联)
虽然 Subject.execute*方法能够在它们返回后自动地清理线程的状态，但有可能在一些情况下，你想自己管理 ThreadState。当结合 w/Shiro 时，这几乎总是在框架开发层次使用，但它很少在 bootstrap/daemon 情景下使用（上面 Subject.execute(callable)例子使用得更为频繁）。

Guarantee Cleanup
关于这一机制最重要的是，你必须一直保证当前的线程在逻辑执行完后被清理，以确保在一个可重复使用或线程池的环境中没有一个线程状态腐化。

最好的做法是在 try/finally 块保证清理：

```
Subject subject = new SubjectThreadState(subject);
ThreadState threadstate = new SubjectThreadState(subject);
threadState.bind();
try {
        //execute work as the built Subject
} finally {
        //ensure any state is cleaned to the thread won't be
        //corrupt in a reusable or pooled thread environment
        threadState.clear();
}
```

有趣的是，这正是 Subject.execute*方法实际上所做的——它们只是在 Callable 或 Runnable 执行前后自动地执行这个逻辑。Shiro 的 ShiroFilter 为 Web 应用程序执行几乎相同的逻辑（ShiroFilter 使用 Web 特定的 ThreadState 的实现，超出了本节的范围）。

Web Use
不要在一个处理 Web 请求的进程中使用上述 ThreadState 代码示例。 Web 特定的 ThreadState 的实现使用 Web 请求代替。相反，确保 ShiroFilter 拦截 Web 请求以确保 Subject 的 building/binding/cleanup 能够好好的完成。

A Different Thread
如果你有一个 Callable 或 Runnable 实例要以 Subject 来执行，你将自己执行 Callable 或 Runnable（或这将它移交给线程池或执行者或 ExcutorService），你应该使用 Subject.associateWith*方法。这些方法确保在最终执行的线程中保留 Subject，且该 Subject 是可访问的。

Callable 例子：

```
Subject subject = new Subject.Builder()…
Callable work = //build/acquire a Callable instance.
//associate the work with the built subject so SecurityUtils.getSubject() calls works properly:
work = subject.associateWith(work);
ExecutorService executorService = new java.util.concurrent.Executors.newCachedThreadPool();
//execute the work on a different thread as the built Subject:
executor.execute(work);
```

Runnable 例子：

```
Subject subject = new Subject.Builder()…
Runnable work = //build/acquire a Runnable instance.
//associate the work with the built subject so SecurityUtils.getSubject() calls works properly:
work = subject.associateWith(work);
Executor executor = new java.util.concurrent.Executors.newCachedThreadPool();
//execute the work on a different thread as the built Subject:
executor.execute(work);
```

Automatic Cleanup associateWith*方法自动执行必要的线程清理，以取保现在在线程池环境中的 clean。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Spring Framework

Integrating Apache Shiro into Spring based Applications

本页涵盖了将 Shiro 集成到基于 Spring 的应用程序的方法。

Shiro 的 JavaBean 兼容性使得它非常适合通过 Spring XML 或其他基于 Spring 的配置机制。Shiro 应用程序需要一个具有单例 SecurityManager 实例的应用程序。请注意，这不会是一个静态的单例，但应该只有一个应用程序能够使用的实例，无论它是否是静态单例的。

Standalone Applications

这里是在 Spring 应用程序中启用应用程序单例 SecurityManager 的最简单的方法：

```
<!-- Define the realm you want to use to connect to your back-end security datasource: -->
<bean id="myRealm" class="…">
…
</bean>
```

```
<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">
        <!-- Single realm app. If you have multiple realms, use the 'realms' property instead. -->
        <property name="realm" ref="myRealm"/>
</bean>
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

<!-- For simplest integration, so that all SecurityUtils.* methods work in all cases, -->
<!-- make the securityManager bean a static singleton. DO NOT do this in web          -->
<!-- applications - see the 'Web Applications' section below instead.                  -->
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="staticMethod" value="org.apache.shiro.SecurityUtils.setSercurityManager"/>
        <property name="arguments" ref="securityManager"/>
</bean>
```

Web Applications

Shiro 拥有对 Spring Web 应用程序的一流支持。在 Web 应用程序中，所有 Shiro 可访问的万恶不请求必须通过一个主要的 Shiro 过滤器。该过滤器本身是极为强大的，允许临时的自定义过滤器链基于任何 URL 路径表达式执行。

在 Shiro 1.0 之前，你不得不在 Spring web 应用程序中使用一个混合的方式，来定义 Shiro 过滤器及所有它在 web.xml 中的配置属性，但在 Spring XML 中定义 SecurityManager。这有些令人沮丧，由于你不能把你的配置固定在一个地方，以及利用更为先进的 Spring 功能的配置能力，如 PropertyPlaceholderConfigurer 或抽象 bean 来固定通用配置。

现在在 Shiro 1.0 及以后版本中，所有 Shiro 配置都是在 Spring XML 中完成的，用来提供更为强健的 Spring 配置机制。

以下是如何在基于 Spring web 应用程序中配置 Shiro：

web.xml

除了其他 Spring web.xml 中的元素（ContextLoaderListener，Log4jConfigListener 等等），定义下面的过滤器及过滤器映射：

```
<!-- The filter-name matches name of a 'shiroFilter' bean inside applicationContext.xml -->
<filter>
        <filter-name>shiroFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
        <init-param>
                <param-name>targetFilterLifecycle</param-name>
                <param-value>true</param-value>
        </init-param>
</filter>

…
<!-- Make sure any request you want accessible to Shiro is filtered. /*catches all -->
<!-- request. Usually this filter mapping is defined first (before all others) to -->
<!-- ensure that Shiro works in subsequent filters in the filter chain: -->
<filter-mapping>
        <filter-name>shiroFilter</filter-name>
        <url-pattern>/*<url-pattern>
</filter-mapping>
```

applicationContext.xml

在你的 applicationContext.xml 文件中，定义 web 支持的 SecurityManager 和'shiroFilter' bean 将会被 web.xml 引用。

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
        <property name="securityManager" ref="securityManager"/>
        <!-- override these for application-specific URLs if you like:
        <property name="loginUrl" value="/login.jsp"/>
        <property name="successUrl" value="/home.jsp"/>
        <property name="unauthorizedUrl" value="/unauthorized.jsp"/> -->
        <!-- The 'filters' property is not necessary since any declared javax.servlet.Filter bean -->
        <!-- defined will be automatically acquired and available via its beanName in chain         -->
        <!-- definitions, but you can perform instance overrides or name aliases here if you like: -->
        <!-- <propery name="filters">
                <util:map>
                        <entry key="anAlias" value-ref="someFilter"/>
                </util:map>
        </property> -->
        <propery name="filterChainDefinitions">
                <value>
                        # some example chain definitions
                        /admin/** = authc, roles[admin]
                        /docs/** = authc, perms[document:read]
                        /** = authc
                        # more URL-to-FilterChain definitions here
                </value>
        </property>
</bean>

<!-- Define any javax.servlet.Filter beans you want anywhere in this application context.         -->
<!-- They will automatically be acquired by the 'shiroFilter' bean above and made available -->
<!-- to the 'filterChainDefinitions' property. Or you can manully/explicitly add them    -->
<!-- to the shiroFilter's 'filter' Map if desired. See its JavaDoc for more details.      -->
<bean id="someFilter" class="…"/>
<bean id="anotherFilter" class="…"> … </bean>
…

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
        <!-- Single realm app. If you have multiple realms, use the 'realms' property instead. --->
        <!-- By default the servlet container sessions will be used. Uncomment this line
                to use shiro's native sessions (see the JavaDoc for more): -->
        <!-- <property name="sessionMode" value="native"/> -->
</bean>
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

<!-- Define the Shiro Realm implementation you want to use to connect to your back-end -->
<!-- security datasource: -->
<bean id="myRealm" class="…">
…
</bean>
```

Enabling Shiro Annotations

在独立应用程序和 Web 应用程序中，你可能想为安全检查使用 Shiro 的注释（例如，@RequiresRoles，@RequiresPermissions 等等）。这需要 Shiro 的 Spring AOP 集成来扫描合适的注解类以及执行必要的安全逻辑。

以下是如何使用这些注解的。只需添加这两个 bean 定义到 applicationContext.xml 中：

```
<!-- Enable Shiro Annotations for Spring-configured beans. Only run after -->
<!-- the lifecycleBeanProcessor has run: -->
<bean class="org.springframwork.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
depends-on="lifecycleBeanPostProcessor"/>
<bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">
        <property name="securityManager" ref="securityManager"/>
</bean>
```

Secure Spring Remoting

Shiro 的 Spring 远程支持有两部分：配置客户端远程调用和配置服务器接收及处理远程调用。

Server-side Configuration

当一个远程调用方法到达启用 Shiro 的服务器时，与该 RPC 调用关联的 Subject 在线程执行时必须绑定到访问的接收线程。这是通过在 applicationContext.xml 中定义 SecureRemotInvocationExecutor bean 来完成的：

```
<!-- Secure Spring remoting: Ensure any Spring Remoting method invocations -->
<!-- can be associated with a Subject for security checks. -->
<bean id="secureRemoteInvocationExecutor"
class="org.apache.shiro.spring.remoting.SecureRemoteInvocationExecutor">
        <property name="securityManager" ref="securityManager"/>
</bean>
```

当你定义这个 bean 之后，你必须将其插入到任何你正在用来 export/expose 你服务的远程 Exporter。Exporter 实现是根据使用的远程处理机制/协议来定义的。请参阅 Sping 的 Remoting 章节关于定义 Exporter bean 的内容。

例如，如果使用基于 HTTP 的远程调用（注意 secureRemoteInvocationExecutor bean 的相关属性）：

```
<bean name="/someService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <property name="service" ref="someService"/>
        <property name="serviceInterface" value="com.pkg.service.SomeService"/>
        <property name="remoteInvocationExecutor" ref="secureRemoteInvocationExecutor"/>
</bean>
```

Client-side Configuration

当远程调用被执行后，Subject 的识别信息必须附加到远程调用的负载上使服务器知道是谁作出了该调用。若客户端是一个基于 Spring 的客户端，该关联是通过 Shiro 的 SecureRemoteInvocationFactory 来完成的：

```
<bean id="secureRemoteInvocationFactory" class="org.apache.shiro.spring.remoting.SecureRemoteInvocationFactory">
```

在你定义好这个 bean 后，你需要将它插入到你正在使用的基于特定协议的 Spring remoting ProxyFactoryBean 中。

例如，如果你正在使用基于 HTTP 的远程调用（注意上面定义的 secureRemoteInvocationFactory bean 的相关属性）：

```
<bean id="someService" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
```

```
    <property name="serviceUrl" value="http://host:port/remoting/someService"/>
    <property name="serviceInterface" value="com.pkg.service.SomeService"/>
    <property name="remoteInvocationFactory" ref="secureRemoteInvocationFactory"/>
</bean>
```

Lend a hand with documentation
我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。

Guice

TODO：未翻译

Apache Shiro Terminology

请花 2 分钟来阅读和理解它——这很重要。真的。这里的术语和概念在文档的任何地方都被涉及到，它将在总体上大大简化你对 Shiro 和安全的理解。

由于所使用的术语使得安全可能令人困惑。我们将通过澄清一些核心概念使生活更容易，你将会看到 Shiro API 是如何很好地反映了它们：

- **Authentication**
  身份验证是验证 Subject 身份的过程——实质上是证明某些人是否真的是他们所说的他们是谁。当认证尝试成功后，应用程序能够相信该 subject 被保证是其所期望的。

- **Authorization**
  授权，又称为访问控制，是决定一个 user/Subject 是否被允许做某事的过程。它通常是通过检查和解释 Subject 的角色和权限（见下文），然后允许或拒绝到一个请求的资源或功能来完成的。

- **Cipher**
  密码是进行加密或解密的一种算法。该算法一般依赖于一块被称为 key 的信息。基于不同的 key 的加密算法也是不一样的，所有解密没有它是非常困难的。

  密码有不同的表现形式。分组密码致力于符号块，通常是固定大小的，而流密码致力于连续的符号流。对称性密码加密和解密使用相同的密钥（key），而非对称性加密使用不同的密钥。如果非对称性加密的密钥不能从其他地方得到，那么可以创建公钥/私钥对公开共享。

- **Credential**
  凭证是一块信息，用来验证 user/Subject 的身份。在认证尝试期间，一个（或多个）凭证与 Principals(s)被一同提交，来验证 user/Subject 所提交的确实是所关联的用户。证书通常是非常秘密的东西，只有特定的 user/Subject 才知道，如密码或 PGP 密钥或生物属性或类似的机制。

这个想法是为 principal 设置的，只有一个人会知道正确的证书来“匹配”该 principal。如果当前 user/Subject 提供了正确的凭证匹配了存储在系统中的，那么系统可以假定并信任当前 user/Subject 是真的他们所说的他们是谁。信任度随着更安全的凭证类型加深（如，生物识别签名 > 密码）。

- **Cryptography**
  加密是保护信息不受不希望的访问的习惯做法，通过隐藏信息或将它转化成无意义的东西，这样没人可以理解它。Shiro 致力于加密的两个核心要素：加密数据的密码，如使用公钥或私钥的邮件，以及散列表（也称消息摘要），它对数据进行不可逆的加密，如密码。

- **Hash**
  散列函数是单向的，不可逆转的输入源，有时也被称为消息，在一个编码的哈希值内部，有时也被称为消息摘要。它通常用于密码，数字指纹，或以字节数组为基础的数据。

- **Permission**
  权限，至少按照 Shiro 的解释，是在应用程序中描述原始功能的一份声明并没有更多的功能。权限是在安全策略中最低级别的概念。它们仅定义了应用程序能够做“什么”。它们没有说明“谁”能够执行这些操作。权限只是行为的声明，仅此而已。

  一些权限的例子：
  - 打开文件
  - 浏览'/user/list'页面
  - 打印文档
  - 删除'jsmith'用户

- **Principal**
  Principal 是一个应用程序用户（Subject）的任何标志属性。“标志属性”可以是任何对你应用程序有意义的东西——用户名，姓，名，社会安全号码，用户 ID 等。这就是它——没什么古怪的。

  Shiro 也引用一些我们称之为 Subject 的 primary principal 的东西。一个 primary principal 是在整个应用程序中唯一标识 Subject 的 principal。理想的 primary principal 是用户名或 RDBMS 用户表主键——用户 ID。对于在应用程序中的用户（Subject）来说，只有一个 primary principal

- **Realm**
  Realm 是一个能够访问应用程序特定的安全数据（如用户，角色和权限）的组件。它可以被看作是一个特定安全的 DAO（Data Access Object）。Realm 将这些应用程序特定的数据转换成 Shiro 能够理解的格式，这样 Shiro 反过来能够提供一个单一的易于理解的 Subject 编程 API，无论有多少数据源存在或无论你的数据是什么样的应用程序特定的格式。

  Realm 通常和数据源是一对一的对应关系，如关系数据库，LDAP 目录，文件系统，或其他类似资源。因此，Realm 接口的实现使用数据源特定的 API 来展示授权数据（角色，权限等），如 JDBC，文件 IO，Hibernate 或 JPA，或其他数据访问 API。

- **Role**
  基于你对话的对象，一个角色的定义是可以多变的。在许多应用程序中，它充其量是个模糊不清的概念，人们用它来隐式定义安全策略。Shiro 偏向于把角色简单地解释为一组命名的权限的集合。这就是它——一个应用程序的唯一名称，聚集一个或多个权限声明。

  这是一个比许多应用程序使用的隐式的定义更为具体的定义。如果你选择了你的数据模型反映 Shiro 的假设，你会发现将有更多控制安全策略的权力。

- **Session**
  会话是一个在一段时间内有状态的数据，其上下文与一个单一的与软件系统交互的 user/Subject 相关联。当 Subject 使用应用程序时，能够从会话中添加/读取/删除数据，并且应用程序稍后能够在需要的地方使用该数据。会话会被终止，由于 user/Subject 注销或会话不活动而超时。

  对于那些熟悉 HttpSession 的，Shiro Session 服务于同一目标，除了 Shiro 会话能够在任何环境下使用，甚至在没有 Servlet 容器或 EJB 容器的环境。

- **Subject**
  Subject 只是一个精挑细选的安全术语，基本上的意思是一个应用程序用户的安全特定的“视图”。然而 Subject 不总是需要反映为一个人——它可以代表一个调用你应用程序的外部进程，或许是一个系统帐户的守护进程，在一段时间内执行一些间歇性的东西（如一个 cron job）。它基本上是任何使用应用程序做某事的实体的一个代表。

Lend a hand with documentation

我们希望本文档可以帮助你及你用 Apache Shiro 所做的工作，我们的团体在不断改善和扩展该文档。如果你想帮助 Shiro 项目，请考虑修改，扩展，或者添加文档到你认为需要的地方。您的每一点帮助都壮大了我们的团体，从而改进了 Shiro。
贡献您的文档最简单的方法是将其发送到用户论坛或者用户的通讯录。