权限验证框架

什么是认证（Authentication）

1. 通俗地讲就是**验证当前用户的身份**，证明“你是你自己”（比如：你每天上下班打卡，都需要通过指纹打卡，当你的指纹和系统里录入的指纹相匹配时，就打卡成功）
2. 互联网中的认证：
   - 用户名密码登录
   - 邮箱发送登录链接
   - 手机号接收验证码
   - 只要你能收到邮箱/验证码，就默认你是账号的主人

什么是授权（Authorization）

- 用户授予第三方应用访问该用户某些资源的权限
  - 你在安装手机应用的时候，APP 会询问是否允许授予权限（访问相册、地理位置等权限）
  - 你在访问微信小程序时，当登录时，小程序会询问是否允许授予权限（获取昵称、头像、地区、性别等个人信息）
- 实现授权的方式有：cookie、session、token、OAuth

什么是凭证（Credentials）

实现认证和授权的前提是**需要一种媒介（证书）来标记访问者的身份**

在互联网应用中，一般网站（如掘金）会有两种模式，游客模式和登录模式。游客模式下，可以正常浏览网站上面的文章，一旦想要点赞/收藏/分享文章，就需要登录或者注册账号。当用户登录成功后，服务器会给该用户使用的浏览器颁发一个令牌（token），这个令牌用来表明你的身份，每次浏览器发送请求时会带上这个令牌，就可以使用游客模式下无法使用的功能。

权限验证

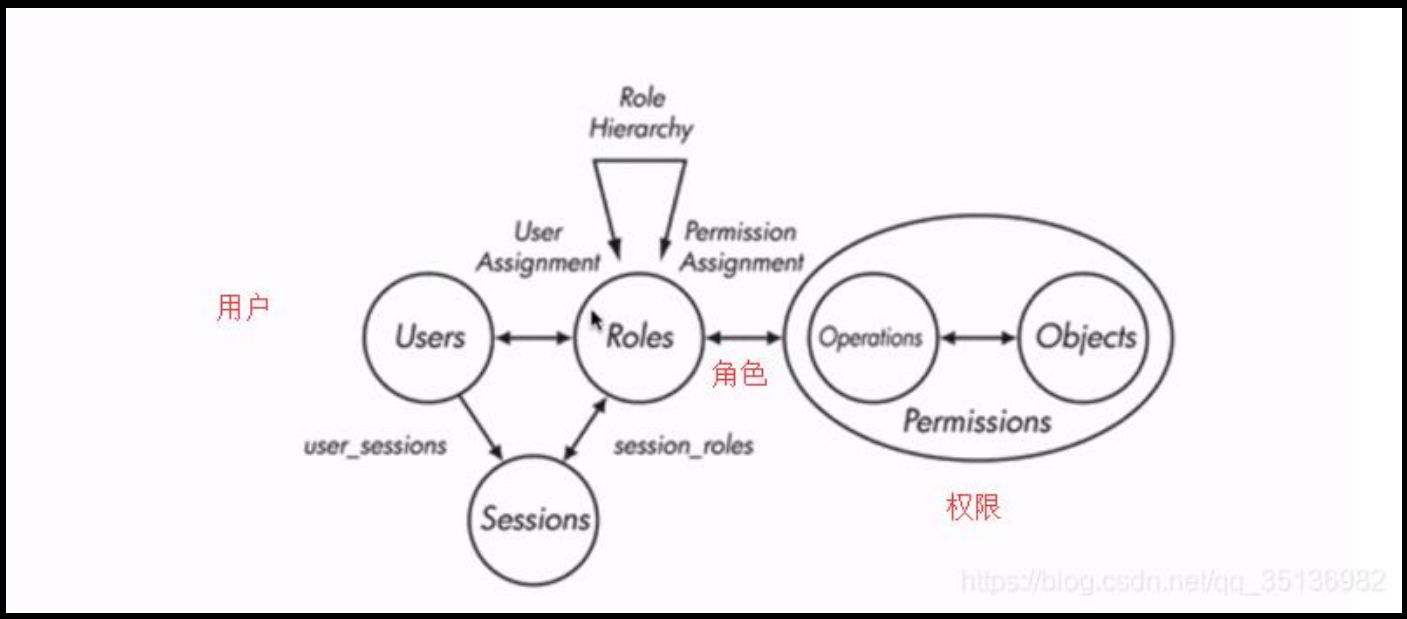
为什么需要权限管理？

1. 安全性：误操作，人为破坏，数据泄露等
2. 数据隔离：不同权限能看到及操作不同的数据
3. 明确职责：运营，客服等不同角色，等级不同

权限管理的核心是什么？

1.用户-权限：人员少，功能固定，或特别简单的系统

2.RBAC：用户-角色-权限 所有系统都适用



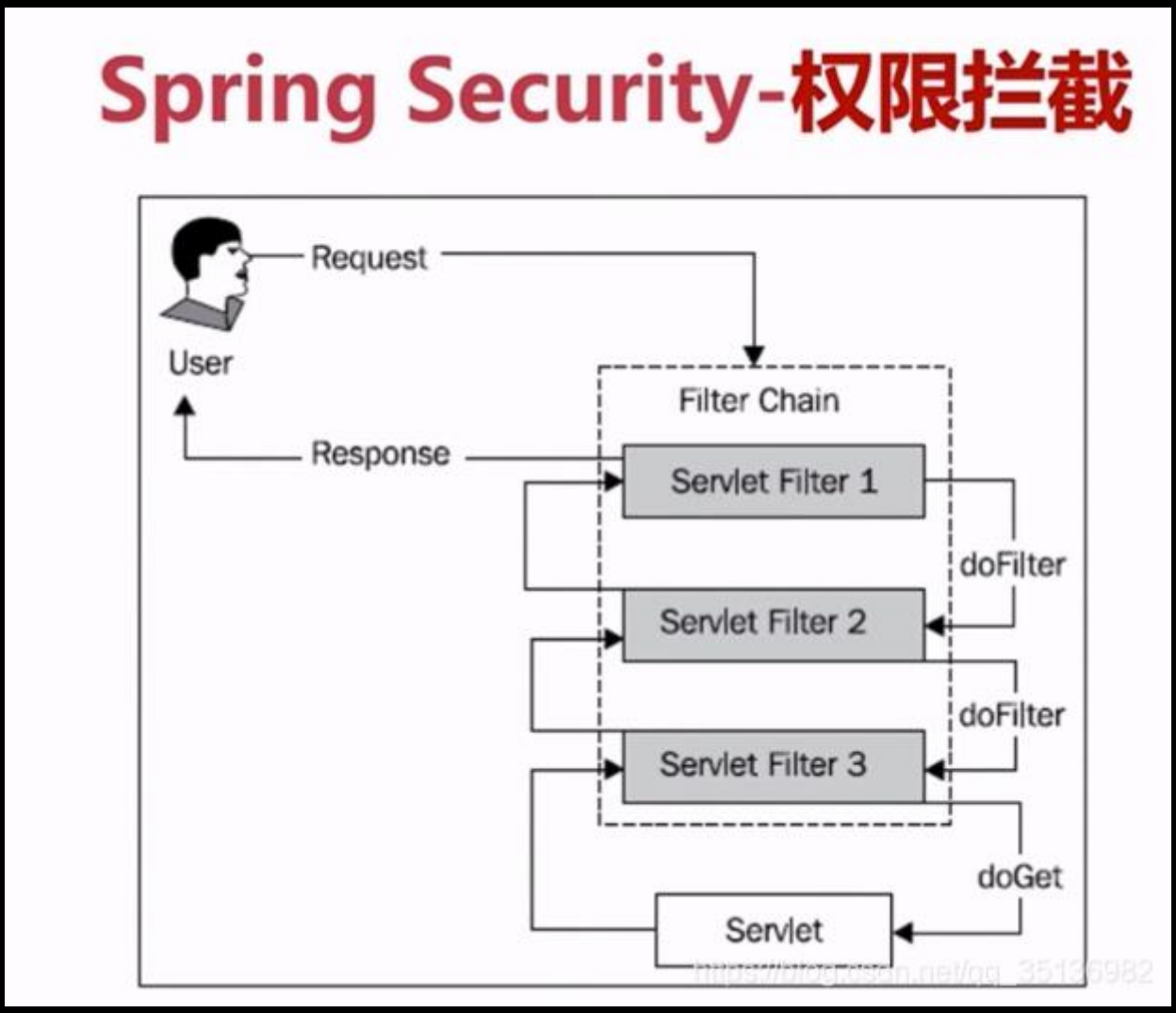
权限框架主要有？

1. Spring Security
2. apache shiro

Spring Security 权限框架

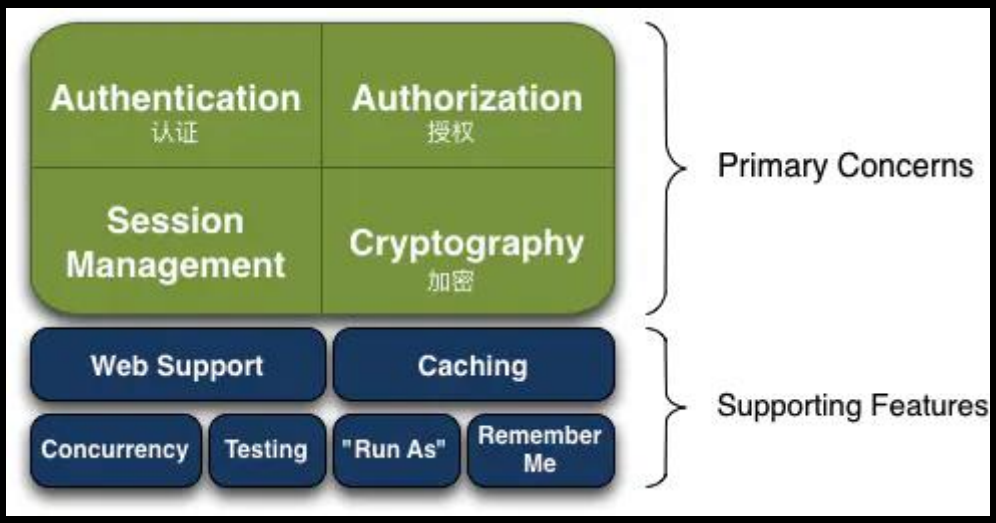
主要是认证和验证

1. Basic 认证：Basic 认证是 HTTP 中非常简单的认证方式，因为简单，所以不是很安全，不过仍然非常常用。
2. Digest 认证：客户端请求资源->服务器返回认证标示->客户端发送认证信息->服务器查验认证，如果成功则继续资源传送，否则直接断开连接。
3. x.509 认证：数字认证
4. LDAP 认证：LDAP 认证是通过 WSS3.0 加上轻量目录 LDAP 协议搭建的种认证方式，使用 https 加密传输，主要用于做文档管理。
5. From 认证：表单认证

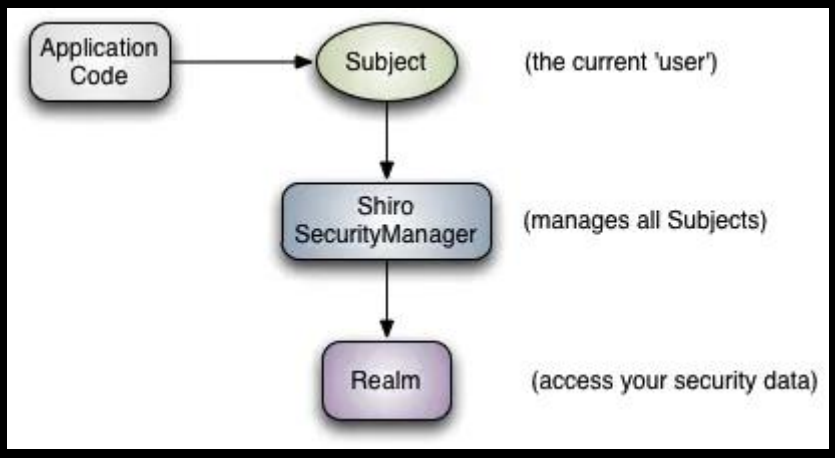


Shiro 权限框架

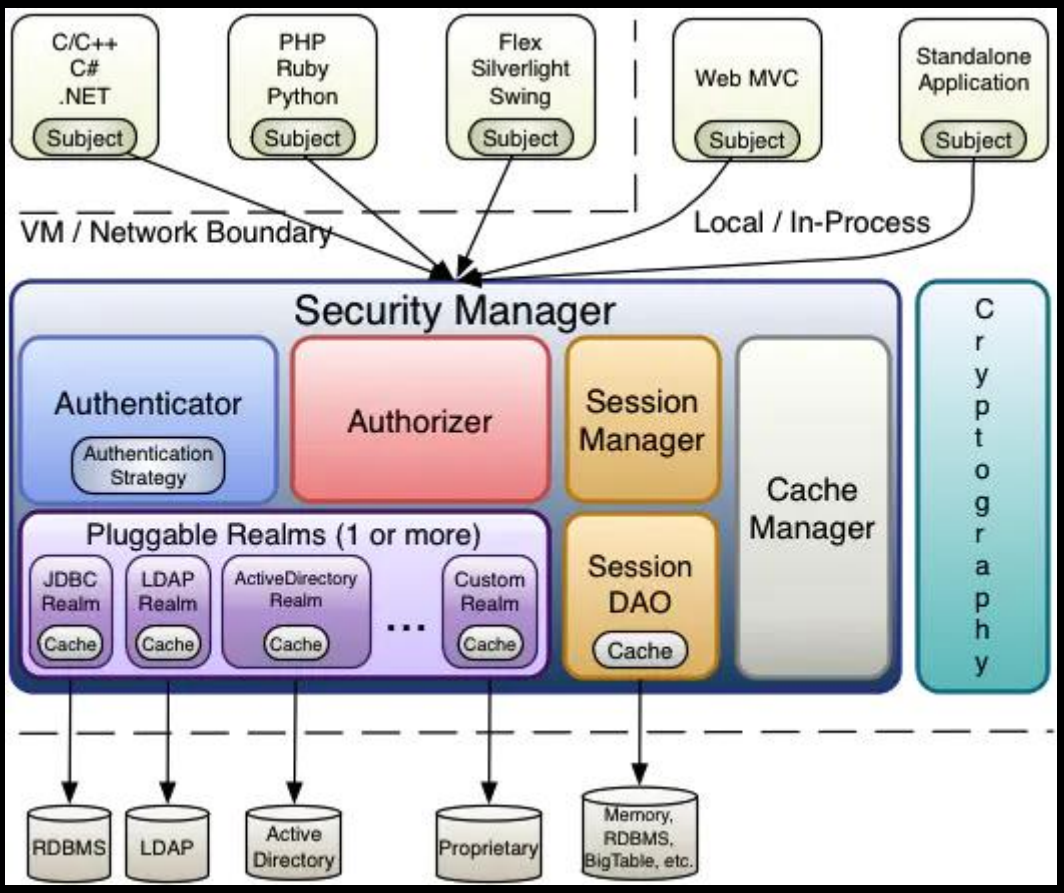
Shiro 是一个强大的简单易用的 Java 安全框架，主要用来更便捷的**认证，授权，加密，会话管理**。
Shiro 首要的和最重要的目标就是容易使用并且容易理解。



从大的角度来看，Shiro 有三个主要的概念：Subject，SecurityManager，Realms，下面这幅图可以看到这些原件之间的交互。



1. Subject：翻译为主角，当前参与应用安全部分的主角。可以是用户，可以是第三方服务，可以是 cron 任务，或者任何东西。主要指一个正在与当前软件交互的东西。
   所有 Subject 都需要 SecurityManager，当你与 Subject 进行交互，这些交互行为实际上被转换为与 SecurityManager 的交互
2. SecurityManager：安全管理员，Shiro 架构的核心，它就像 Shiro 内部所有原件的保护伞。然而一旦配置了 SecurityManager，SecurityManager 就用到的比较少，开发者大部分时间都花在 Subject 上面。请记得，当你与 Subject 进行交互的时候，实际上是 SecurityManager 在背后帮你举起 Subject 来做一些安全操作。
3. Realms：Realms 作为 Shiro 和你的应用的连接桥，当需要与安全数据交互的时候，像用户账户，或者访问控制，Shiro 就从一个或多个 Realms 中查找。
   Shiro 提供了一些可以直接使用的 Realms，如果默认的 Realms 不能满足你的需求，你也可以定制自己的 Realms



基于 Filter

案例如下：

Spring Boot 使用过滤器 Filter

过滤器是对数据进行过滤，预处理过程，当我们访问网站时，有时候会发布一些敏感信息，发完以后有的会用*替代，还有就是登陆权限控制等，一个资源，没有经过授权，肯定是不能让用户随便访问的，这个时候，也可以用到过滤器。过滤器的功能还有很多，例如实现 URL 级别的权限控制、压缩响应信息、编码格式等等。

过滤器依赖 servlet 容器。在实现上基于函数回调，可以对几乎所有请求进行过滤。下面简单的说说 Spring Boot 里面如何增加过滤器。

1. Pom: starter

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

2. 建立过滤器程序

   @Order(1): 表示过滤器的顺序，假设我们有多个过滤器，你如何确定过滤器的执行顺序？这个注解就是规定过滤器的顺序。

   @WebFilter: 表示这个 class 是过滤器。

里面的参数，filterName 为过滤器名字，urlPatterns 为过滤器的范围，initParams 为过滤器初始化参数。

过滤器里面的三个方法

a) init ： filter 对象只会创建一次，init 方法也只会执行一次。

b) doFilter ： 主要的业务代码编写方法，可以多次重复调用

c) destroy ： 在销毁 Filter 时自动调用（程序关闭或者主动销毁 Filter）。

```
@Order(1)
@WebFilter(filterName = "MyFilter",urlPatterns = "/*",initParams = {
        @WebInitParam(name = "URL",value = "http://localhost:8080/hello")
})

public class MyFilter implements Filter {
    private String url;
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        this.url = filterConfig.getInitParameter("URL");
        System.out.println("过滤 URL=" + this.url );
    }

    @Override
    public void destroy() {
        System.out.println("My Filter destroy!");
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException,
ServletException {
        System.out.println("我是过滤器的执行方法，客户端向 Servlet 发送的请求被我拦截到了");
        filterChain.doFilter(servletRequest, servletResponse);
        System.out.println("我是过滤器的执行方法，Servlet 向客户端发送的响应被我拦截到了");
    }
}
```

3. 建立 Contoller 类

```
@RestController
public class MyController {
    @RequestMapping(value = "/hello",method = RequestMethod.GET)
    public String hello(){
        return "被 Filter 拦截";
    }
    // /*这表示一层路由中的所有情况，/hi 是，/hi/hi 不是
    @RequestMapping(value = "/*",method = RequestMethod.GET)
    public String hi(){
        return "没有被 Filter 拦截";
    }
}
```

4. 启动类中增加注解，自动注册 Filter

@ServletComponentScan ： 在 SpringBootApplication 上使用 @ServletComponentScan 注解后，Servlet、Filter、Listener 可以直接通过@WebServlet、@WebFilter、@WebListener 注解自动注册，无需其他代码。

```
@SpringBootApplication

@ServletComponentScan
public class SpringFilterDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringFilterDemoApplication.class, args);
```

```
    }
}
```

Shiro 学习笔记

概述

注意:登录认证(身份认证)和权限验证是不一样的,登录认证是用户在登录的是否去验证用户的身份,为权限认证则是判断已经登录的用户是否具有某样操作的权限。

Session 不是 HTTP session，是 Shiro 自己的 session

Shiro 有三大核心组件：

Subject：即当前用户，在权限管理的应用程序里往往需要知道谁能够操作什么，谁拥有操作该程序的权利，shiro 中则需要通过 Subject 来提供基础的当前用户信息，Subject 不仅仅代表某个用户，与当前应用交互的任何东西都是 Subject，如网络爬虫等。所有的 Subject 都要绑定到 SecurityManager 上，与 Subject 的交互实际上是被转换为与 SecurityManager 的交互。

SecurityManager：即所有 Subject 的管理者，这是 Shiro 框架的核心组件，可以把他看做是一个 Shiro 框架的全局管理组件，用于调度各种 Shiro 框架的服务。作用类似于 SpringMVC 中的 DispatcherServlet，用于拦截所有请求并进行处理。

Realm：Realm 是用户的信息认证器和用户的权限人证器，我们需要自己来实现 Realm 来自定义的管理我们自己系统内部的权限规则。SecurityManager 要验证用户，需要从 Realm 中获取用户。可以把 Realm 看做是数据源。

最简单的一个 Shiro 应用：应用代码通过 Subject 来进行认证和授权，而 Subject 又委托给 SecurityManager；我们需要给 Shiro 的 SecurityManager 注入 Realm，从而让 SecurityManager 能得到合法的用户及其权限进行判断。

Subject：主体，可以看到主体可以是任何可以与应用交互的“用户”；

SecurityManager：相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2 中的 FilterDispatcher；是 Shiro 的心脏；所有具体的交互都通过 SecurityManager 进行控制；它管理着所有 Subject、且负责进行认证和授权、及会话、缓存的管理。

Authenticator：认证器，负责主体认证的，这是一个扩展点，如果用户觉得 Shiro 默认的不好，可以自定义实现；其需要认证策略（Authentication Strategy），即什么情况下算用户认证通过了；

Authrizer：授权器，或者访问控制器，用来决定主体是否有权限进行相应的操作；即控制着用户能访问应用中的哪些功能；

Realm：可以有 1 个或多个 Realm，可以认为是安全实体数据源，即用于获取安全实体的；可以是 JDBC 实现，也可以是 LDAP 实现，或者内存实现等等；由用户提供；注意：Shiro 不知道你的用户 / 权限存储在哪及以何种格式存储；所以我们一般在应用中都需要实现自己的 Realm；

SessionManager：如果写过 Servlet 就应该知道 Session 的概念，Session 呢需要有人去管理它的生命周期，这个组件就是 SessionManager；而 Shiro 并不仅仅可以用在 Web 环境，也可以用在如普通的 JavaSE 环境；所以 Shiro 就抽象了一个自己的 Session 来管理主体与应用之间交互的数据。

SessionDAO：DAO 大家都用过，数据访问对象，用于会话的 CRUD，比如我们想把 Session 保存到数据库，那么可以实现自己的 SessionDAO，通过如 JDBC 写到数据库；比如想把 Session 放到 Memcached 中，可以实现自己的 Memcached SessionDAO；另外 SessionDAO 中可以使用 Cache 进行缓存，以提高性能；

CacheManager：缓存控制器，来管理如用户、角色、权限等的缓存的；因为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能。

Cryptography：密码模块，Shiro 提高了一些常见的加密组件用于如密码加密 / 解密的。

Shiro 中通过 principals（身份）和 credentials（证明）给 shiro，

principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。一个主体可以有多个 principals，但只有一个 Primary principals，一般是用户名 / 密码 / 手

身份验证

身份验证，即在应用中谁能证明他就是他本人。一般提供如他们的身份 ID 一些标识信息来表明他就是他本人，如提供身份证，用户名 / 密码来证明。

在 shiro 中，用户需要提供 principals （身份）和 credentials（证明）给 shiro，从而应用能验证用户身份：

1. principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。一个主体可以有多个 principals，但只有一个 Primary principals，一般是用户名 / 密码 / 手机号。
2. credentials：证明 / 凭证，即只有主体知道的安全值，如密码 / 数字证书等。

最常见的 principals 和 credentials 组合就是用户名 / 密码了。接下来先进行一个基本的身份认证。

身份认证 demo：

```
public static void main(String[] args) {
    // 获取SecurityManager工厂，这里使用静态数据ini
    Factory<SecurityManager> factory = new IniSecurityManagerFactory( iniResourcePath: "classpath:demo/shiro-test-1.ini");
    // 获取一个SecurityManager工厂实例
    SecurityManager securityManager =  factory.getInstance();
    // 将SecurityManager绑定到SecurityUtils
    SecurityUtils.setSecurityManager(securityManager);
    // 获取一个Subject
    Subject subject = SecurityUtils.getSubject();
    // 创建一个用户登录数据:token
    UsernamePasswordToken token = new UsernamePasswordToken( username: "edw", password: "123");

    try {
        // 登录:身份验证
        subject.login(token);
        System.out.println("login successfully!");
    }catch (AuthenticationException e){
        e.printStackTrace();
        System.out.println("login failed!");
    }
    // 登出
    subject.logout();
    System.out.println("logout successfully!");
}
```

```
Ini:
[users]
edw=123
```

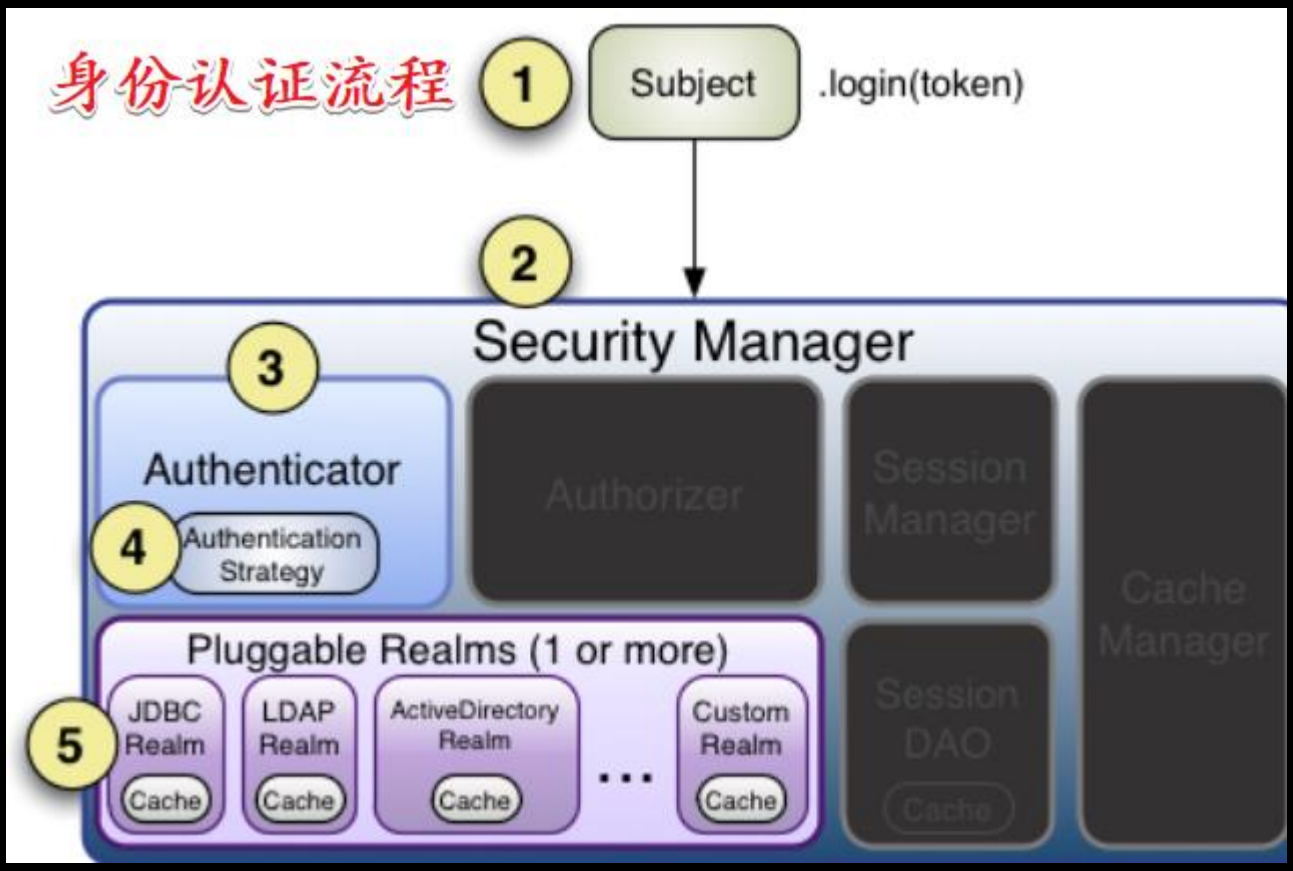
- 首先通过 new IniSecurityManagerFactory 并指定一个 ini 配置文件来创建一个 SecurityManager 工厂；
- 接着获取 SecurityManager 并绑定到 SecurityUtils，这是一个全局设置，设置一次即可；
- 通过 SecurityUtils 得到 Subject，其会自动绑定到当前线程；如果在 web 环境在请求结束时需要解除绑定；然后获取身份验证的 Token，如用户名 / 密码；
- 调用 subject.login 方法进行登录，其会自动委托给 SecurityManager.login 方法进行登录；
- 如果身份验证失败请捕获 AuthenticationException 或其子类，常见的

如：DisabledAccountException（禁用的帐号）、LockedAccountException（锁定的帐号）、UnknownAccountException（错误的帐号）、ExcessiveAttemptsException（登录失败次数过多）、IncorrectCredentialsException（错误的凭证）、ExpiredCredentialsException（过期的凭证）等，具体请查看其继承关系；对于页面的错误消息展示，最好使用如“用户名 / 密码错误”而不是“用户名错误”/“密码错误”，防止一些恶意用户非法扫描帐号库；

- 最后可以调用 subject.logout 退出，其会自动委托给 SecurityManager.logout 方法退出。

如上测试的几个问题：

1. **用户名 / 密码硬编码在 ini 配置文件，以后需要改成如数据库存储，且密码需要加密存储；**
2. **用户身份 Token 可能不仅仅是用户名 / 密码，也可能还有其他的，如登录时允许用户名 / 邮箱 / 手机号同时登录。**



1. 首先调用 Subject.login(token) 进行登录，其会自动委托给 Security Manager，调用之前必须通过 SecurityUtils.setSecurityManager() 设置；
2. SecurityManager 负责真正的身份验证逻辑；它会委托给 Authenticator 进行身份验证；
3. Authenticator 才是真正的身份验证者，Shiro API 中核心的身份认证入口点，此处可以自定义插入自己的实现；
4. Authenticator 可能会委托给相应的 AuthenticationStrategy 进行多 Realm 身份验证，默认 ModularRealmAuthenticator 会调用 AuthenticationStrategy 进行多 Realm 身份验证；
5. Authenticator 会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回 / 抛出异常表示身份验证失败了。此处可以配置多个 Realm，将按照相应的顺序及策略进行访问。

InI

Shiro 可以连接数据库，也可以使用静态数据，即在 ini 配置文件中直接指定数据。

INI 从一个根对象 SecurityManager 开始的

根对象 SecurityManager
从之前的 Shiro 架构图可以看出，Shiro 是从根对象 SecurityManager 进行身份验证和授权的；也就是所有操作都是自它开始的，这个对象是线程安全且**整个应用只需要一个即可**，因此 Shiro 提供了 SecurityUtils **让我们绑定它为全局的**，方便后续操作。

因为 Shiro 的类都是 POJO 的，因此都很容易放到任何 IoC 容器管理。但是和一般的 IoC 容器的区别在于，Shiro 从根对象 securityManager 开始导航；
Shiro 支持的依赖注入：public 无参构造器、setter 依赖注入。即**不支持有参构造器或者其他访问权限构造器。**

纯 Java 代码写法：

```
DefaultSecurityManager securityManager = new DefaultSecurityManager();
//设置 authenticator
ModularRealmAuthenticator authenticator = new ModularRealmAuthenticator();
authenticator.setAuthenticationStrategy(new AtLeastOneSuccessfulStrategy());
securityManager.setAuthenticator(authenticator);
//设置 authorizer
ModularRealmAuthorizer authorizer = new ModularRealmAuthorizer();
authorizer.setPermissionResolver(new WildcardPermissionResolver());
securityManager.setAuthorizer(authorizer);
//设置 Realm
DruidDataSource ds = new DruidDataSource();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3306/shiro");
ds.setUsername("root");
ds.setPassword("");
JdbcRealm jdbcRealm = new JdbcRealm();
jdbcRealm.setDataSource(ds);
jdbcRealm.setPermissionsLookupEnabled(true);
securityManager.setRealms(Arrays.asList((Realm) jdbcRealm));
//将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);
Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");
subject.login(token);
Assert.assertTrue(subject.isAuthenticated());
```

等价的 INI 配置：

```
[main]
\#authenticator
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
```

```
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy
authenticator.authenticationStrategy=$authenticationStrategy
securityManager.authenticator=$authenticator
\#authorizer
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
authorizer.permissionResolver=$permissionResolver
securityManager.authorizer=$authorizer
\#realm
dataSource=com.alibaba.druid.pool.DruidDataSource
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/shiro
dataSource.username=root
\#dataSource.password=
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
jdbcRealm.dataSource=$dataSource
jdbcRealm.permissionsLookupEnabled=true
securityManager.realms=$jdbcRealm
```

1. 对象名 = 全限定类名 相对于调用 public 无参构造器创建对象
2. 对象名.属性名 = 值 相当于调用 setter 方法设置常量值
3. 对象名.属性名 =$ 对象引用 相当于调用 setter 方法设置对象引用

使用 ini 配置:

```
Factory<org.apache.shiro.mgt.SecurityManager> factory =
        new IniSecurityManagerFactory("classpath:shiro-config.ini");
org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
//将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);
Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");
subject.login(token);
Assert.assertTrue(subject.isAuthenticated());
```

INI 配置文件的几个部分:

```
[main]
#提供了对根对象 securityManager 及其依赖的配置
securityManager=org.apache.shiro.mgt.DefaultSecurityManager
.....
securityManager.realms=$jdbcRealm
[users]
#提供了对用户/密码及其角色的配置，用户名=密码，角色 1，角色 2
username=password,role1,role2
[roles]
#提供了角色及权限之间关系的配置，角色=权限 1，权限 2
role1=permission1,permission2
[urls]
#用于 web，提供了对 web url 拦截相关的配置，url=拦截器[参数]，拦截器
/index.html = anon
/admin/** = authc, roles[admin], perms["permission1"]
```

[main 部分的注入]:

1. 创建对象

   securityManager=org.apache.shiro.mgt.DefaultSecurityManager

   其构造器必须是 public 空参构造器，通过反射创建相应的实例。

2. 常量值 setter 注入

```
dataSource.driverClassName=com.mysql.jdbc.Driver
jdbcRealm.permissionsLookupEnabled=true
```

会自动调用 jdbcRealm.setPermissionsLookupEnabled(true)，对于这种常量值会自动类型转换。

3. 对象引用 setter 注入

```
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy
authenticator.authenticationStrategy=$authenticationStrategy
securityManager.authenticator=$authenticator
```

   会自动通过 securityManager.setAuthenticator(authenticator) 注入引用依赖。

4. 嵌套属性 setter 注入

   securityManager.authenticator.authenticationStrategy=$authenticationStrategy

5. byte 数组 setter 注入

```
\#base64 byte[]
authenticator.bytes=aGVsbG8=
\#hex byte[]
authenticator.bytes=0x68656c6c6f&amp;nbsp;
```

   默认需要使用 Base64 进行编码，也可以使用 0x 十六进制。

6. Array/Set/List setter 注入

```
authenticator.array=1,2,3
authenticator.set=$jdbcRealm,$jdbcRealm
```

   多个之间通过 “,” 分割。

7. Map setter 注入

   authenticator.map=$jdbcRealm:$jdbcRealm,1:1,key:abc

[users] 部分:

配置用户名 / 密码及其角色，格式:**“用户名 = 密码，角色 1，角色 2”**，角色部分可省略。如:

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

密码一般生成其摘要 / 加密存储，

[roles] 部分

配置角色及权限之间的关系，**格式:“角色 = 权限 1，权限 2”**; 如:

```
[roles]
role1=user:create,user:update
role2=*
```

[urls] 部分

配置 url 及相应的拦截器之间的关系，格式:“url = 拦截器 [参数]，拦截器 [参数]，如:

```
[urls]
/admin/** = authc, roles[admin], perms["permission1"]
```

编码加密

在涉及到密码存储问题上，应该加密 / 生成密码摘要存储，而不是存储明文密码。

编码 / 解码

Shiro 提供了 **base64 和 16 进制字符串编码** / 解码的 API 支持，方便一些编码解码操作。

Shiro 内部的一些数据的存储 / 表示都使用了 base64 和 16 进制字符串。

```
base64 编码/解码操作:

String str = "hello";
String base64Encoded = Base64.encodeToString(str.getBytes());
String str2 = Base64.decodeToString(base64Encoded);

16 进制字符串编码/解码操作:

String str = "hello";
String base64Encoded = Hex.encodeToString(str.getBytes());
String str2 = new String(Hex.decode(base64Encoded.getBytes()));
```

散列算法:

散列算法一般用于生成数据的摘要信息，是一种不可逆的算法，一般适合存储密码之类的数据，常见的散列算法如 MD5、SHA 等。**一般进行散列时最好提供一个 salt(盐)**，比如加密密码“admin”，产生的散列值是 “21232f297a57a5a743894a0e4a801fc3”，可以到一些 md5 解密网站很容易的通过散列值得到密码 “admin”，即如果直接对密码进行散列相对来说破解更容易，此时我们可以加一些只有系统知道的干扰数据，如用户名和 ID(即盐)；这样散列的对象是 “密码 + 用户名 +ID”，这样生成的散列值相对来说更难破解。

```
String str = "hello";
String salt = "123";
String md5 = new Md5Hash(str, salt).toString();//还可以转换为 toBase64()/toHex()
```

散列时还可以指定散列次数，如 2 次表示:

```
md5(md5(str)): “new Md5Hash(str, salt, 2).toString()”。
```

```
String sha1 = new Sha256Hash(str, salt).toString();
```

使用 SHA256 算法生成相应的散列数据，另外还有如 SHA1、SHA512 算法。

Shiro 还提供了通用的散列支持:

```
String str = "hello";
String salt = "123";
//内部使用 MessageDigest
String simpleHash = new SimpleHash("SHA-1", str, salt).toString();
```

通过调用 SimpleHash 时指定散列算法，其内部使用了 Java 的 MessageDigest 实现。

为了方便使用，Shiro 提供了 HashService，默认提供了 DefaultHashService 实现。

```
DefaultHashService hashService = new DefaultHashService(); //默认算法 SHA-512
hashService.setHashAlgorithmName("SHA-512");
hashService.setPrivateSalt(new SimpleByteSource("123")); //私盐，默认无
hashService.setGeneratePublicSalt(true);//是否生成公盐，默认 false
hashService.setRandomNumberGenerator(new SecureRandomNumberGenerator());//用于生成公盐。默认就这个
```

```
hashService.setHashIterations(1); //生成 Hash 值的迭代次数
HashRequest request = new HashRequest.Builder()
          .setAlgorithmName("MD5").setSource(ByteSource.Util.bytes("hello"))
          .setSalt(ByteSource.Util.bytes("123")).setIterations(2).build();
String hex = hashService.computeHash(request).toHex();
```

1. 首先创建一个 DefaultHashService，默认使用 SHA-512 算法；
2. 以通过 hashAlgorithmName 属性修改算法；
3. 可以通过 privateSalt 设置一个私盐，其在散列时自动与用户传入的公盐混合产生一个新盐；
4. 可以通过 generatePublicSalt 属性在用户没有传入公盐的情况下是否生成公盐；
5. 可以设置 randomNumberGenerator 用于生成公盐；
6. 可以设置 hashIterations 属性来修改默认加密迭代次数；
7. 需要构建一个 HashRequest，传入算法、数据、公盐、迭代次数。

SecureRandomNumberGenerator 用于生成一个随机数：

```
SecureRandomNumberGenerator randomNumberGenerator =
     new SecureRandomNumberGenerator();
randomNumberGenerator.setSeed("123".getBytes());
String hex = randomNumberGenerator.nextBytes().toHex();
```

加密 / 解密

Shiro 还提供对称式加密 / 解密算法的支持，如 AES、Blowfish 等；当前还没有提供对非对称加密 / 解密算法支持

AES 算法实现：

```
AesCipherService aesCipherService = new AesCipherService();
aesCipherService.setKeySize(128); //设置 key 长度
//生成 key
Key key = aesCipherService.generateNewKey();
String text = "hello";
//加密
String encrptText =
aesCipherService.encrypt(text.getBytes(), key.getEncoded()).toHex();
//解密
String text2 =
 new String(aesCipherService.decrypt(Hex.decode(encrptText),
key.getEncoded()).getBytes());
Assert.assertEquals(text, text2);
```

PasswordService/CredentialsMatcher

Shiro 提供了 PasswordService 及 CredentialsMatcher 用于提供加密密码及验证密码服务。

```
public interface PasswordService {
    //输入明文密码得到密文密码
    String encryptPassword(Object plaintextPassword) throws
IllegalArgumentException;
}
public interface CredentialsMatcher {
    //匹配用户输入的 token 的凭证（未加密）与系统提供的凭证（已加密）
    boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo
info);
}
```

Shiro 默认提供了 PasswordService 实现 DefaultPasswordService；CredentialsMatcher 的实现有 PasswordMatcher 和 HashedCredentialsMatcher（更强大）

DefaultPasswordService 配合 PasswordMatcher 实现简单的密码加密与验证服务：

1. 定义 Realm

```
public class MyRealm extends AuthorizingRealm {
    private PasswordService passwordService;
    public void setPasswordService(PasswordService passwordService) {
        this.passwordService = passwordService;
    }
     //省略 doGetAuthorizationInfo，具体看代码
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
        return new SimpleAuthenticationInfo(
                "wu",              passwordService.encryptPassword("123"),
                getName());
    }
}
```

2. ini 配置

```
[main]
passwordService=org.apache.shiro.authc.credential.DefaultPasswordService
hashService=org.apache.shiro.crypto.hash.DefaultHashService
passwordService.hashService=$hashService
hashFormat=org.apache.shiro.crypto.hash.format.Shiro1CryptFormat
passwordService.hashFormat=$hashFormat
hashFormatFactory=org.apache.shiro.crypto.hash.format.DefaultHashFormatFactory
passwordService.hashFormatFactory=$hashFormatFactory
passwordMatcher=org.apache.shiro.authc.credential.PasswordMatcher
passwordMatcher.passwordService=$passwordService
myRealm=com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm
myRealm.passwordService=$passwordService
myRealm.credentialsMatcher=$passwordMatcher
securityManager.realms=$myRealm
```

a) passwordService 使用 DefaultPasswordService，如果有必要也可以自定义；

b) hashService 定义散列密码使用的 HashService，默认使用 DefaultHashService（默认 SHA-256 算法）；

c) hashFormat 用于对散列出的值进行格式化，默认使用 Shiro1CryptFormat，另外提供了 Base64Format 和 HexFormat，对于有 salt 的密码请自定义实现 ParsableHashFormat 然后把 salt 格式化到散列值中；

d) hashFormatFactory 用于根据散列值得到散列的密码和 salt；因为如果使用如 SHA 算法，那么会生成一个 salt，此 salt 需要保存到散列后的值中以便之后与传入的密码比较时使用；默认使用 DefaultHashFormatFactory；

e) passwordMatcher 使用 PasswordMatcher，其是一个 CredentialsMatcher 实现；

f) 将 credentialsMatcher 赋值给 myRealm，myRealm 间接继承了 AuthenticatingRealm，其在调用 getAuthenticationInfo 方法获取到 AuthenticationInfo 信息后，会使用 credentialsMatcher 来验证凭据是否匹配，如果不匹配将抛出 IncorrectCredentialsException 异常。

3.

HashedCredentialsMatcher 实现密码验证服务：

Shiro 提供了 CredentialsMatcher 的散列实现 HashedCredentialsMatcher，和之前的 PasswordMatcher 不同的是，它只用于密码验证，且可以提供自己的盐，而不是随机生成盐，且生成密码散列值的算法需要自己写，因为能提供自己的盐。

1. 生成密码散列值

此处我们使用 MD5 算法，“密码 + 盐（用户名 + 随机数）” 的方式生成散列值：

```
String algorithmName = "md5";
String username = "liu";
String password = "123";
String salt1 = username;
String salt2 = new SecureRandomNumberGenerator().nextBytes().toHex();
int hashIterations = 2;
SimpleHash hash = new SimpleHash(algorithmName, password, salt1 + salt2,
hashIterations);
String encodedPassword = hash.toHex();
```

如果要写用户模块，需要在新增用户 / 重置密码时使用如上算法保存密码，将生成的密码及 salt2 存入数据库

2. 生成 Realm

```
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)
throws AuthenticationException {
    String username = "liu"; //用户名及 salt1
    String password = "202cb962ac59075b964b07152d234b70"; //加密后的密码
    String salt2 = "202cb962ac59075b964b07152d234b70";
SimpleAuthenticationInfo ai =
        new SimpleAuthenticationInfo(username, password, getName());
    ai.setCredentialsSalt(ByteSource.Util.bytes(username+salt2)); //盐是用户名+
随机数
        return ai;
}
```

3. ini 配置

```
[main]
credentialsMatcher=org.apache.shiro.authc.credential.HashedCredentialsMatcher
credentialsMatcher.hashAlgorithmName=md5
credentialsMatcher.hashIterations=2
credentialsMatcher.storedCredentialsHexEncoded=true
myRealm=com.github.zhangkaitao.shiro.chapter5.hash.realm.MyRealm2
myRealm.credentialsMatcher=$credentialsMatcher
securityManager.realms=$myRealm
```

Realm

Realm：域，Shiro 从 Realm 获取安全数据（如用户、角色、权限），就是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的角色 / 权限进行验证用户是否能进行操作；可以把 Realm 看成 **DataSource**，即安全数据源。如我们之前的 ini 配置方式将使用

org.apache.shiro.realm.text.IniRealm。

单 Realm 配置

1. 自定义 Realm

   public class MyRealm1 implements Realm

2. Ini 配置指定 Realm 实现

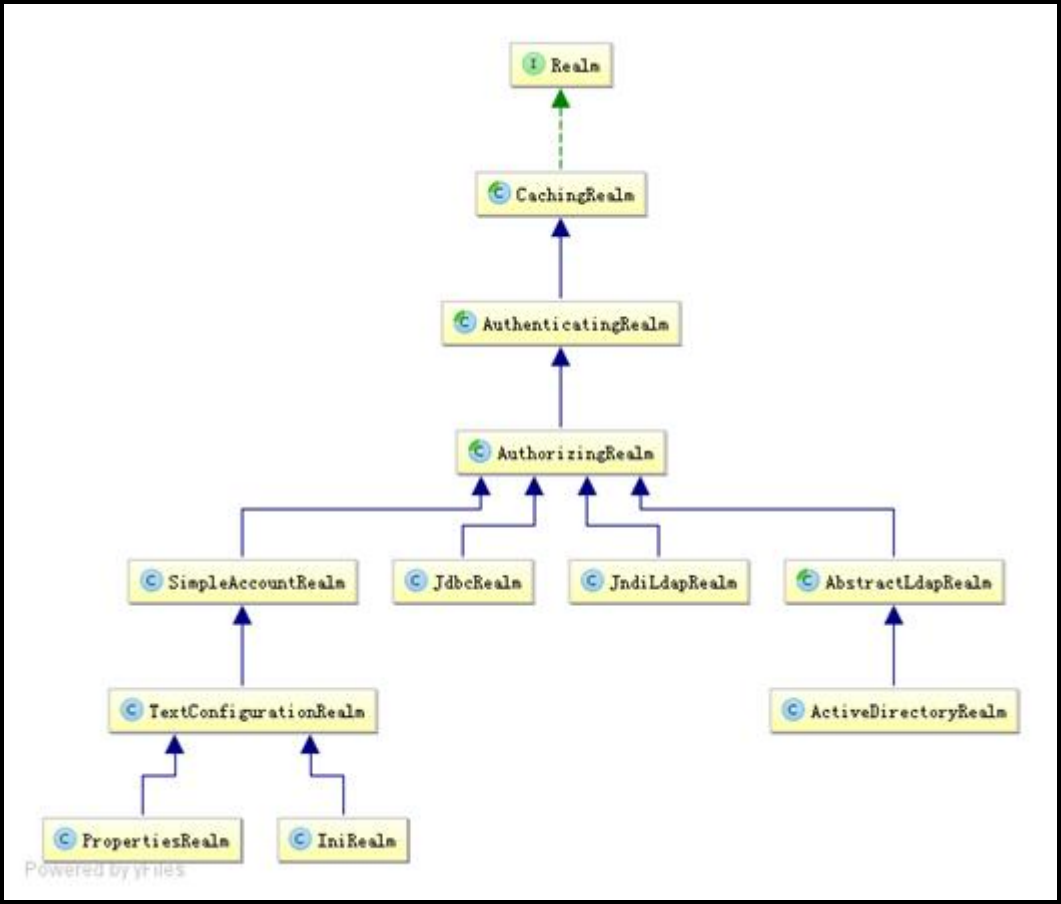
```
\#声明一个 realm
myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
\#指定 securityManager 的 realms 实现
securityManager.realms=$myRealm1
```

多 Realm 配置

```
\#声明一个 realm
myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
myRealm2=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm2
\#指定 securityManager 的 realms 实现
securityManager.realms=$myRealm1,$myRealm2
```

securityManager 会**按照 realms 指定的顺序进行身份认证**。此处我们使用显示指定顺序的方式指定了 Realm 的顺序，如果删除 “securityManager.realms=$myRealm1,$myRealm2”，那么 securityManager 会按照 realm 声明的顺序进行使用(即无需设置 realms 属性,其会自动发现)，当我们显示指定 realm 后，其他没有指定 realm 将被忽略，如“securityManager.realms=$myRealm1”，那么 myRealm2 不会被自动设置进去。

默认提供的 Realm



一般继承 **AuthorizingRealm**（授权）即可；其继承了 AuthenticatingRealm（即身份验证），而

且也间接继承了 CachingRealm（带有缓存实现）

1. IniRealm: [users] 部分指定用户名 / 密码及其角色；[roles] 部分指定角色即权限信息；
2. PropertiesRealm: user.username=password,role1,role2 指定用户名 / 密码及其角色；role.role1=permission1,permission2 指定角色及权限信息；
3. JDBC Realm

JDBC Realm:

- 数据库及依赖：mysql
- 到数据库 shiro 下建三张表：
  - users（用户名 / 密码）、
  - user_roles（用户 / 角色）、
  - roles_permissions（角色 / 权限）
- ini 配置：

```
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
dataSource=com.alibaba.druid.pool.DruidDataSource
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/shiro
dataSource.username=root
dataSource.password=
jdbcRealm.dataSource=$dataSource
securityManager.realms=$jdbcRealm
```

-

1. 变量名 = 全限定类名会自动创建一个类实例
2. 变量名.属性 = 值 自动调用相应的 setter 方法进行赋值
3. $ 变量名 引用之前的一个对象实例

没看完：
https://www.w3cschool.cn/shiro/hzlw1ifd.html

Authenticator 及 AuthenticationStrategy

Authenticator 的职责是验证用户帐号，是 Shiro API 中身份验证核心的入口点

```
public AuthenticationInfo authenticate(AuthenticationToken authenticationToken)
        throws AuthenticationException;
```

如果验证成功，将返回 AuthenticationInfo 验证信息；此信息中包含了身份及凭证；如果验证失败将抛出相应的 AuthenticationException 实现。

SecurityManager 接口继承了 Authenticator，另外还有一个 ModularRealmAuthenticator 实现，其委托给多个 Realm 进行验证，验证规则通过 AuthenticationStrategy 接口指定，默认提供的实现：

1. **FirstSuccessfulStrategy**：只要有一个 Realm 验证成功即可，只返回第一个 Realm 身份验证成功的认证信息，其他的忽略；
2. **AtLeastOneSuccessfulStrategy/** ：只要有一个 Realm 验证成功即可，和 FirstSuccessfulStrategy 不同，返回所有 Realm 身份验证成功的认证信息；

3. **AllSuccessfulStrategy**：所有 Realm 验证成功才算成功，且返回所有 Realm 身份验证成功的认证信息，如果有一个失败就失败了。

ModularRealmAuthenticator 默认使用 AtLeastOneSuccessfulStrategy 策略。

策略的配置：

```
\#指定 securityManager 的 authenticator 实现
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
securityManager.authenticator=$authenticator
\#指定 securityManager.authenticator 的 authenticationStrategy
allSuccessfulStrategy=org.apache.shiro.authc.pam.AllSuccessfulStrategy
securityManager.authenticator.authenticationStrategy=$allSuccessfulStrategy
```

另外还可以自定义策略。

授权

授权，也叫**访问控制**，即在应用中**控制谁能访问哪些资源**（如访问页面/编辑数据/页面操作等）。在授权中需了解的几个关键对象：主体（Subject）、资源（Resource）、权限（Permission）、角色（Role）。

1. 主体，即访问应用的用户，在 Shiro 中使用 Subject 代表该用户。用户只有授权后才允许访问相应的资源。
2. 资源 在应用中用户可以访问的 URL，比如访问 JSP 页面、查看/编辑某些数据、访问某个业务方法、打印文本等等都是资源。用户只要授权后才能访问。
3. 权限 安全策略中的原子授权单位，通过权限我们可以表示在应用中用户有没有操作某个资源的权力。即权限表示在应用中用户能不能访问某个资源，如： 访问用户列表页面 查看/新增/修改/删除用户数据（即很多时候都是 CRUD（增查改删）式权限控制）打印文档等。如上可以看出，权限代表了用户有没有操作某个资源的权利，即反映在某个资源上的操作允不允许，不反映谁去执行这个操作。所以后续还需要把权限赋予给用户，即定义哪个用户允许在某个资源上做什么操作（权限），Shiro 不会去做这件事情，而是由实现人员提供。

   Shiro 支持粗粒度权限（如用户模块的所有权限）和细粒度权限（操作某个用户的权限，即实例级别的）

4. 角色 角色代表了操作集合，可以理解为权限的集合，一般情况下我们会赋予用户角色而不是权限，即这样用户可以拥有一组权限，赋予权限时比较方便。典型的如：项目经理、技术总监、CTO、开发工程师等都是角色，不同的角色拥有一组不同的权限。

授权方式

Shiro 支持三种方式的授权：

1. 编程式：通过写 if/else 授权代码块完成：

```
Subject subject = SecurityUtils.getSubject();
if(subject.hasRole(“admin”)) {
    //有权限
} else {
    //无权限
}
```

2. 注解式

```
@RequiresRoles("admin")
public void hello() {
    //有权限
    // 没有权限将抛出相应的异常;
}
```

3. JSP/GSP 标签：在 JSP/GSP 页面通过相应的标签完成：

```
<shiro:hasRole name="admin">
<!— 有权限 —>
</shiro:hasRole>
```

还用不到，没看完

https://www.w3cschool.cn/shiro/skex1if6.html

Web 集成

Shiro 提供了与 Web 集成的支持，其通过一个 **ShiroFilter** 入口来拦截需要安全控制的 URL，然后进行相应的控制，ShiroFilter **类似于如 Strut2/SpringMVC 这种 web 框架的前端控制器**，其是安全控制的入口点，其负责读取配置（如 ini 配置文件），然后判断 URL 是否需要登录 / 权限等工作。

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-web</artifactId>
    <version>1.2.2</version>
</dependency>
```

基于角色的访问控制 vs. 于资源的访问控制

“RBAC”和“RBAC 新解”分别了解“基于角色的访问控制”“基于资源的访问控制(Resource-Based Access Control)”。

会话管理

shiro 提供了一个完整的企业级会话管理解决方案，不再依赖 web 容器。可以在 web 和非 web 环境下使用。

shiro 的 session 特性

1. 基于 POJO/J2SE：shiro 中 session 相关的类都是基于接口实现的简单的 java 对象（POJO），兼容所有 java 对象的配置方式，扩展也更方便，完全可以定制自己的会话管理功能 。
2. 简单灵活的**会话存储/持久化**：因为 shiro 中的 session 对象是基于简单的 java 对象的，所以

你可以将 session 存储在任何地方，例如，文件，各种数据库，内存中等。

3. 容器无关的集群功能：shiro 中的 session 可以很容易的集成第三方的缓存产品完成集群的功能。例如，Ehcache + Terracotta，Coherence，GigaSpaces 等。你可以很容易的实现会话集群而无需关注底层的容器实现。
4. 异构客户端的访问：可以实现 web 中的 session 和非 web 项目中的 session 共享。
5. 会话事件监听：提供对对 session 整个生命周期的监听。
6. 保存**主机地址**：在会话开始 session 会存用户的 ip 地址和主机名，以此可以判断用户的位置。
7. 会话失效/过期的支持：用户长时间处于不活跃状态可以使会话过期，调用 touch()方法，可以主动更新最后访问时间，让会话处于活跃状态。
8. 透明的 Web 支持：shiro 全面支持 Servlet 2.5 中的 session 规范。这意味着你可以将你现有的 web 程序改为 shiro 会话，而无需修改代码。
9. 单点登录的支持：shiro session 基于普通 java 对象，使得它更容易存储和共享，可以实现跨应用程序共享。可以根据共享的会话，来保证认证状态到另一个程序。从而实现单点登录。

使用会话：

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

获取 session 的 subject.getSession()方法等价于 currentUser.getSubject(true)。Suject.getSession(boolean create) 与 web 中的 HttpServletRequest.getSession(boolean create) 类似。

1. 如果 Subject 已经拥有一个 session，则方法中的 boolean 类型参数将会忽略，并直接返回已经存在的 session。
2. 如果 Subject 里没有拥有 session：
   a) 参数为 true，则创建一个新的 session 并返回。
   b) 参数为 false，则不会创建新的 session，并返回 null。

返回值	方法名	描述
Object	getAttribute(Object key)	根据key标识返回绑定到session的对象
Collection<Object>	getAttributeKeys()	获取在session中存储的所有的key
String	getHost()	获取当前主机ip地址，如果未知，返回null
Serializable	getId()	获取session的唯一id
Date	getLastAccessTime()	获取最后的访问时间
Date	getStartTimestamp()	获取session的启动时间
long	getTimeout()	获取session失效时间，单位毫秒
void	setTimeout(long maxIdleTimeInMillis)	设置session的失效时间
Object	removeAttribute(Object key)	通过key移除session中绑定的对象
void	setAttribute(Object key, Object value)	设置session会话属性
void	stop()	销毁会话
void	touch()	更新会话最后访问时间

SessionManager 管理所有 Subject 的 session，包括创建、维护、删除、失效、验证等工作。SessionManager 是顶层组件，由 SecurityManager 管理。

SecurityManager 的实现类 DefaultSecurityManager 及 DefaultWebSecurityManager 继承了 SessionsSecurityManager。

SessionsSecurityManager 可以把相应的会话管理委托给 SessionManager。

```
public Session start(SessionContext context) throws AuthorizationException {
    //委托给SessionManager
    return this.sessionManager.start(context);
}
public Session getSession(SessionKey key) throws SessionException {
    //委托给SessionManager
    return this.sessionManager.getSession(key);
}
```

SessionsSecurityManager 中的代码

委托给sessionManager

shiro 提供了三个 SessionManager 的实现

1. DefaultSessionManager：DefaultSecurityManager 使用的默认实现，用于非 web 环境。
2. ServletContainerSessionManager：DefaultWebSecurityManager 使用的默认实现，用于 Web 环境，其直接使用 Servlet 容器的会话。
3. DefaultWebSessionManager：用于 Web 环境的实现，可以替代 ServletContainerSessionManager 自己维护着会话，容器无关。

像 SecurityManager 其它组件一样，可以使用 getter/setter 方法获取和设置组件，同时也支持 ini 配置。

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

全局会话失效时间

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

也可以为每个 session 单独设置会话失效时间

调用 session 的 setTimeout(long maxIdleTimeInMillis)，参数为毫秒

会话监听：

需要自己实现监听器：

```
public class MySessionListener implements SessionListener {
    @Override
    public void onStart(Session session) {
        //会话创建时触发
```

```
        System.out.println("会话创建: " + session.getId());
    }
    @Override
    public void onExpiration(Session session) {
        //会话过期时触发
        System.out.println("会话过期: " + session.getId());
    }
    @Override
    public void onStop(Session session) {
        //退出/会话过期时触发
        System.out.println("会话停止: " + session.getId());
    }
}
```

会话存储/持久化

Session 可以存储在内存或者各种数据库中。 SessionManager 委托 SeesionDao 对 session 进行增删改查。

你可以为 SessionManager 配置 SessionDao

```
[main]
...
sessionDAO = com.foo.my.SessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
```

EHCache SessionDAO

EHCache SessionDAO 存储 Session 到内存，如果内存不够用的话，会保存到磁盘。Ehcache 配合 TerraCotta 可以实现容器无关的分布式集群。

```
<dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-ehcache</artifactId>
        <version>1.2.3</version>
    </dependency>
```

会话验证

Session 必须通过验证才可以将无效过过期的 session 删除，出于性能的考虑，只有在获取会话的时候去验证会话是否过期。如果用户不主动退出，是无法知道 session 是否失效或过期的。如果不定期清理，session 会越来越多。因此需要**定期清理**，shiro 提供了**会话验证调度器 SessionValidationScheduler 来定期完成清除 session** 的工作。

默认的调度器

默认的 SessionValidationScheduler 调度器实现是 ExecutorServiceSessionValidationScheduler （基于 JDKScheduledExecutorService 实现的）。默认的调度周期是 1 小时，也就是没小时都会执行一次 session 验证，并清除过期或无效的 session

```
[main]
...
```

```
sessionValidationScheduler =
org.apache.shiro.session.mgt.ExecutorServiceSessionValidationScheduler
# 默认是 3,600,000 毫秒 = 1 小时:
sessionValidationScheduler.interval = 3600000
securityManager.sessionManager.sessionValidationScheduler =
$sessionValidationScheduler
```

```
关闭调度器（默认是开启）:
[main]
...
securityManager.sessionManager.sessionValidationSchedulerEnabled = false

关闭无效 session 的删除（默认是开启）

main]
...
securityManager.sessionManager.deleteInvalidSessions = false
```

单点登录

集成验证码

Spring 集成

参考

https://www.w3cschool.cn/shiro/
https://www.bilibili.com/video/BV1Si4y1g7nZ?p=10

JWT

什么是 JWT

Json web token (JWT)，是为了在网络应用环境间**传递声明**而执行的一种**基于 JSON** 的开放标准((RFC 7519).该 token 被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT 的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该 token 也可直接被用于认证，也可被加密。

JSON Web Token（简称 JWT）是目前**最流行的跨域认证解决方案**。

官网：
https://jwt.io/

起源

基于 token 的认证和传统的 session 认证的区别：
传统的 session 认证：

http 协议本身是一种无状态的协议，而这就意味着如果用户向我们的应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行，因为根据 http 协议，我们并不能知道是哪个用户发出的请求，所以为了让我们的应用能识别是哪个用户发出的请求，我们只能在服务器存储一份用户登录的信息.

基于 session 的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于 session 认证应用的问题就会暴露出来.

缺点：

1. 认证用户的增多，服务端的开销会明显增大。
2. 分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。
3. CSRF：因为是基于 cookie 来进行用户识别的， cookie 如果被截获，用户就会很容易受到跨站请求伪造的攻击。

基于 token 的鉴权机制：
基于 token 的鉴权机制类似于 http 协议也是无状态的，它**不需要在服务端去保留用户的认证信息或者会话信息**。这就意味着基于 token 认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程上是这样的：

1. 用户使用用户名密码来请求服务器
2. 服务器进行验证用户的信息
3. 服务器通过验证发送给用户一个 token
4. 客户端存储 token，并在每次请求时附送上这个 token 值
5. 服务端验证 token 值，并返回数据

这个 token 必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持 CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了 Access-Control-Allow-Origin: *

JWT 长什么样？

JWT 是**由三段信息构成**的，将这三段信息文本用.链接一起就构成了 Jwt 字符串

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

- 第一部分我们称它为头部（header）

头部承载两部分信息：

a) 声明类型，这里是 jwt

b) 声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的 JSON：

```
{
  'typ': 'JWT',
  'alg': 'HS256'
}
```

然后将头部进行 base64 加密（该加密是可以对称解密的),构成了第一部分：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

- 第二部分我们称其为载荷（payload，类似于飞机上承载的物品）

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，

payload 的 json 结构并不像 header 那么简单，payload 用来承载要传递的数据，它的 json 结构实际上是对 JWT 要传递的数据的一组声明，这些声明被 JWT 标准称为 **claims**，它的一个“属性值对”其实就是一个 claim，每一个 claim 的都代表特定的含义和作用。比如上面结构中的 sub 代表这个 token 的所有人，存储的是所有人的 ID；name 表示这个所有人的名字；admin 表示所有人是否管理员的角色。当后面对 JWT 进行验证的时候，这些 claim 都能发挥特定的作用。

这些有效信息包含三个部分：

a) 标准中注册的声明：Reserved claims

- iss：jwt 签发者
- sub：jwt 所面向的用户
- aud：接收 jwt 的一方
- exp：jwt 的过期时间，这个过期时间必须要大于签发时间
- nbf：定义在什么时间之前，该 jwt 都是不可用的.
- iat：jwt 的签发时间
- jti：jwt 的唯一身份标识，主要用来作为一次性 token,从而回避重放攻击。

b) 公共的声明：Public claims

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.

c) 私有的声明：**Private claims**

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为 base64 是对称解密的，意味着该部分信息可以归类为明文信息。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

然后将其进行 base64 加密，得到 Jwt 的第二部分。

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
```

- 第三部分是签证（signature）

签证信息由三部分组成：

- header (base64 后的)
- payload (base64 后的)
- secret

这个部分需要 base64 加密后的 header 和 base64 加密后的 payload 使用.连接组成的字符串，然后通过 header 中声明的加密方式进行加盐 secret 组合加密，然后就构成了 jwt 的第三部分。

```
// javascript
var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);
```

```
var signature = HMACSHA256(encodedString, 'secret'); //
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

将这三部分用.连接成一个完整的字符串,构成了最终的 jwt:

注意：**secret 是保存在服务器端的**，jwt 的**签发生成也是在服务器端的**，secret 就是用来进行 jwt 的签发和 jwt 的验证，所以，它就是你服务端的**私钥**，在任何场景都不应该流露出去。一旦客户端得知这个 secret，那就意味着客户端是可以自我签发 jwt 了。

如何应用？

- 方式一

当用户希望访问一个受保护的路由或者资源的时候，可以把它放在 Cookie 里面自动发送，但是这样不能跨域

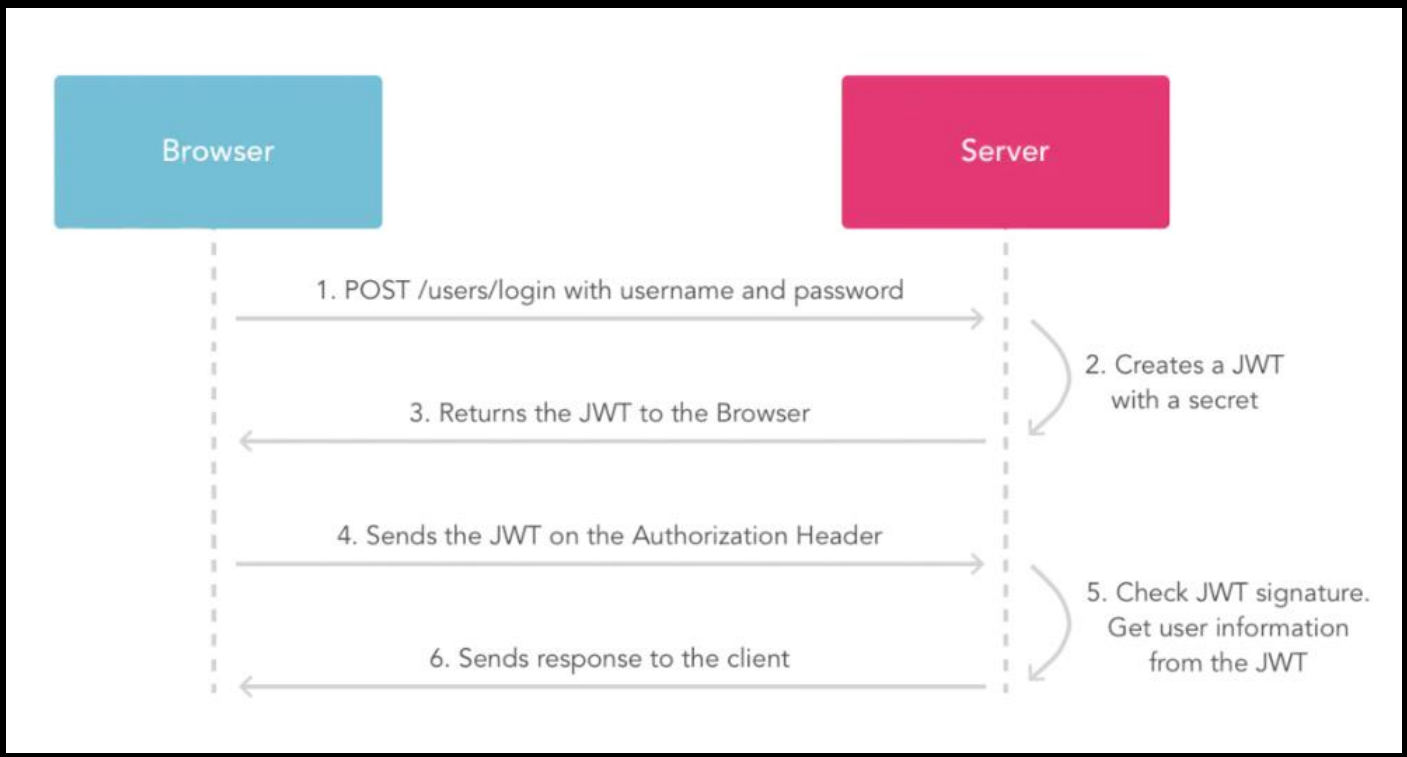
所以更好的做法是放在 HTTP 请求头信息的 **Authorization** 字段里，使用 **Bearer** 模式添加 JWT。

```
GET /calendar/v1/events
Host: http://api.example.com
Authorization: Bearer <token>
```

一般是在请求头里加入 Authorization，并加上 Bearer 标注：

```
fetch('api/user/1', {
  headers: {
    'Authorization': 'Bearer ' + token
  }
})
```

服务端会验证 token，如果验证通过就会返回相应的资源。整个流程就是这样的：



- 方式二

跨域的时候，可以把 JWT 放在 POST 请求的数据体里。

- 方式三

通过 URL 传输

http://www.example.com/user?token=xxx

优点

- 因为 json 的通用性，所以 JWT 是可以进行跨语言支持的，像 JAVA,JavaScript,NodeJS,PHP 等很多语言都可以使用。
- 因为有了 payload 部分，所以 JWT 可以在自身存储一些其他业务逻辑所必要的非敏感信息。
- 便于传输，jwt 的构成非常简单，字节占用很小，所以它是非常便于传输的。
- 它不需要在服务端保存会话信息，所以它易于应用的扩展
- 无状态：Session 会把状态维护在 server 端，浏览器只保存 sessionId（当然你可以保存更多额外的东西），服务端根据 sessionId 再去读取状态（用户权限之类），可以理解为双方被月老牵了红线，会话通信期间有个保持，使用 session-cookie 意味着 stateful；JWT 则可以把权限控制信息和更多的内容传递给浏览器或其他终端（mobile 等）使用，服务端（一般就是 auth 服务器喽）负责发放 JWT，类比保险箱给你发钥匙，你只要有钥匙，什么时候开都行，不需要在客户端和真正的资源服务器间绑定关系，也就是 stateless。
-

安全相关

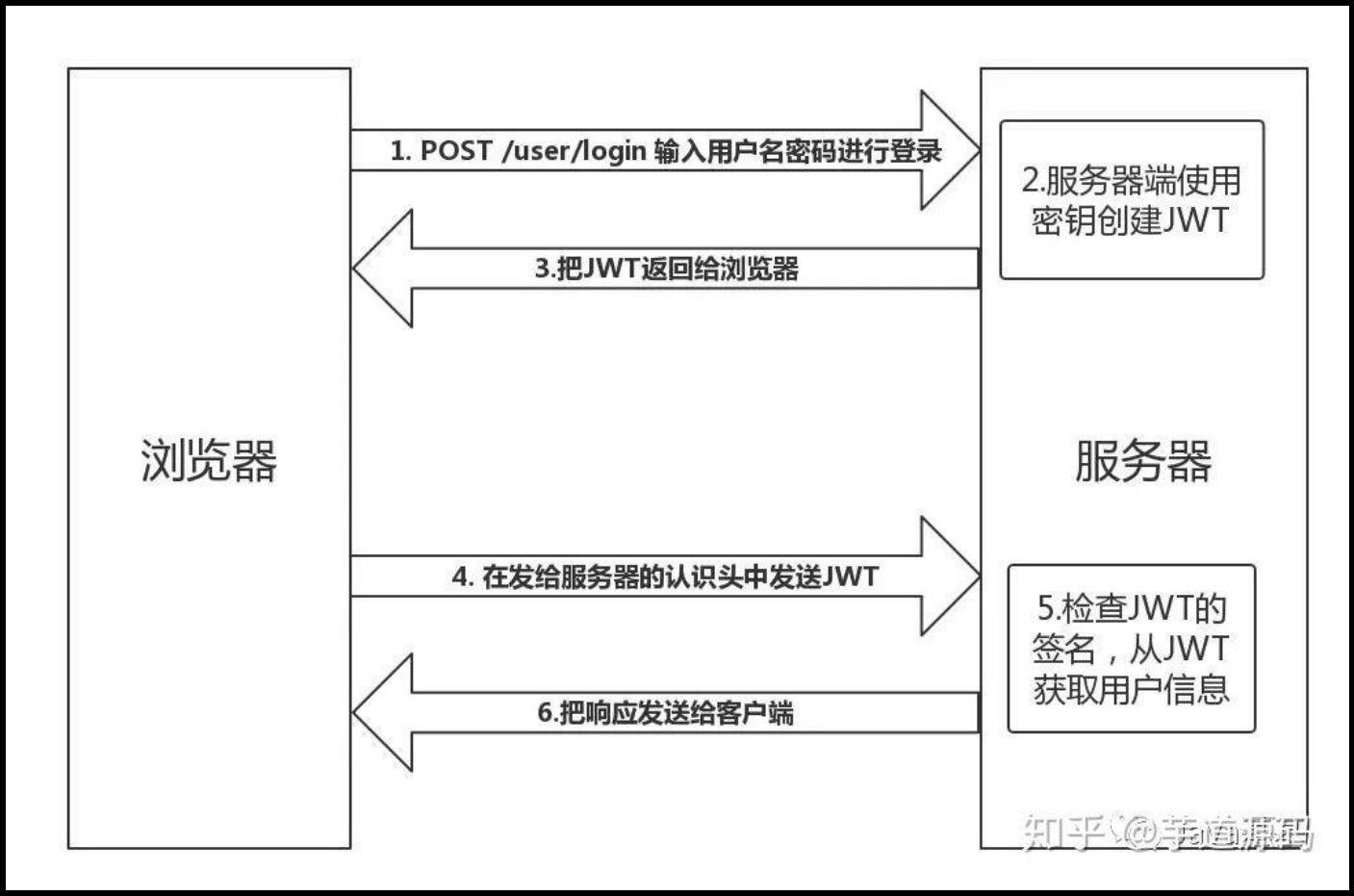
- 不应该在 jwt 的 payload 部分存放敏感信息，因为该部分是客户端可解密的部分。
- 保护好 secret 私钥，该私钥非常重要。
- 如果可以，请使用 https 协议

session 是应该被淘汰了，某些客户端不支持 cookie——这个无论如何都无法解决。

缺点

1. 因为 JWT 是无状态的，因此服务端无法控制已经生成的 Token 失效，这是不可控的
2. 获取到 JWT 也就拥有了登录权限，因此 JWT 是不可泄露的，网站最好使用 https，防止中间攻击偷取 JWT

原理



JWT 认证流程：

- 用户输入用户名/密码登录，服务端认证成功后，会返回给客户端一个 JWT
- 客户端将 token 保存到本地（通常使用 localstorage，也可以使用 cookie）
- 当用户希望访问一个受保护的路由或者资源的时候，需要请求头的 Authorization 字段中使用 Bearer 模式添加 JWT，其内容看起来是下面这样

```
Authorization: Bearer <token>
```

- 服务端的保护路由将会检查请求头 Authorization 中的 JWT 信息，如果合法，则允许用户的行为
- 因为 JWT 是自包含的（内部包含了一些会话信息），因此减少了需要查询数据库的需要
- 因为 **JWT 并不使用 Cookie** 的，所以你可以使用任何域名提供你的 API 服务而不**需要担心跨域资源共享问题（CORS）**
- 因为用户的状态不再存储在服务端的内存中，所以这是一种无状态的认证机制

适用场景

JWT 不适合会话管理？？？ https://www.cnblogs.com/JacZhu/p/9779975.html

使用 JWT，通常是为了前后端分离和兼容多元终端的需求？

JWT 的最佳用途是**一次性授权 Token**，这种场景下的 Token 的特性如下：

1. 有效期短

2. 只希望被使用一次

真实场景的例子——文件托管服务，由两部分组成：

1. Web 应用：这是一个可以被用户登录并维持状态的应用，用户在应用中挑选想要下载的文件。
2. 文件下载服务：无状态下载服务，只允许通过密钥下载。

Session 比较适用于 Web 应用的会话管理，其特点一般是：

1. 权限多，如果用 JWT 则其长度会很长，很有可能突破 Cookie 的存储限制。
2. 基本信息容易变动。如果是一般的后台管理系统，肯定会涉及到人员的变化，那么其权限也会相应变化，如果使用 JWT，那就需要服务器端进行主动失效，这样就将原本无状态的 JWT 变成有状态，改变了其本意。

我们使用 JWT，并不是说看到它新就用，而应该考虑其适用场景，如果需要进行管理，可以考虑使用 Session，毕竟其方案更加成熟。

应用

```
public class JwtUtil {
    /**
     * 秘钥
     * */
    private static final String SECRET = "sdfghjkkjhgferuiuxcbnmnbvc";
    /**
     * 有效期: 30 分钟
     * */
    private static final int EXPIRE_TIME = 30;

    /**
     * @description: 创建一个 Token
     * */
    public static String getToken(User user){
        JWTCreator.Builder builder = JWT.create();
        // 添加实体信息, 即 payload
        builder.withClaim("uid",user.getUid());
        builder.withClaim("username",user.getUsername());
        builder.withClaim("email",user.getEmail());
        builder.withClaim("phone",user.getPhone());
        builder.withClaim("password",user.getPassword());
        builder.withClaim("usertype",user.getUsertype());

        // 过期时间
        Calendar instance = Calendar.getInstance();
        instance.add(Calendar.MINUTE,EXPIRE_TIME);
        builder.withExpiresAt(instance.getTime());

        // 选择使用的算法 和 秘钥
        return builder.sign(Algorithm.HMAC256(SECRET));
    }


    /**
     * @description: 验证一个 token
     * */
    public static void verify(String token){
        JWT.require(Algorithm.HMAC256(SECRET)).build().verify(token);
    }

    /**
     * @description: 获取 token 包含的 payload 信息, 即 User
```

```
     * */
    public static User getTokenInfo(String token){
        DecodedJWT decodedJWT =
JWT.require(Algorithm.HMAC256(SECRET)).build().verify(token);
        User user = new User();
        user.setUid(decodedJWT.getClaim("uid").asInt());
        user.setUsername(decodedJWT.getClaim("username").asString());
        user.setEmail(decodedJWT.getClaim("email").asString());
        user.setPhone(decodedJWT.getClaim("phone").asString());
        user.setPassword(decodedJWT.getClaim("password").asString());
        user.setUsertype(decodedJWT.getClaim("usertype").asInt());
        return user;
    }

}
```

常见的前后端鉴权方式

1. **Session-Cookie**
2. **Token 验证（包括 JWT，SSO）**
3. **OAuth2.0（开放授权）**

cookies 是一种存储机制，而 JWT 是一种加密签名的令牌机制。

其他

HttpOnly

如果您在 cookie 中设置了 HttpOnly 属性，那么通过 js 脚本将无法读取到 cookie 信息，这样能有效的防止 XSS 攻击

补充 · 跨域

什么是跨域？

相同的协议，相同的主机和域名

```
http://www.xx.com
与 https://www.xx.com 不同源，因为协议不同
与 http://www.aa.com 不同源，因为主机不同
与 http://www.xx.com: 90 不同源，因为端口不同
```

哪些不能跨域访问

1.cookie,localstorage,indexdb

2.DOM 节点无法获取
3.ajax 请求不能发送

html 中哪些会出现跨域问题（即不允许跨域访问）

<script><img>等携带 src 属性的标签

html 中允许跨域写操作:
如表单提交和重定向请求
从不安全的域名下有表单提交过来，造成对本站数据的非法修改等问题，这就是常说的 CSRF 安全问题，即**跨站伪造请求**

当业务需要跨域访问时，我们如何做
RFC 中推荐使用 **CORS** 解决方案
如果跨域访问本站点，需要在 HTTP 响应中显示的告诉浏览器该站点被允许
即在响应头添加对应的允许头文件

即像 ajax 这种简单的跨域访问我们需要在响应中携带 **Access-Control-Allow-Origin** 头部，浏览器才会放行，*表示所有域

```
location /login {
    add_header Access-Control-Allow-Origin *;
    return 200 '{"hello":"world"}';
}
```

你这边慢慢来吧，自己多留意不要赔进去就好，创业嘛就这样，成功概率小。

补充 · jsonp

Jsonp(JSON with Padding)

在同源策略下，在某个服务器下的页面是无法获取到该服务器以外的数据的，但 **img、iframe、script** 等标签是个例外，这些标签可以通过 src 属性请求到其他服务器上的数据。 而 JSONP 就是通过 script 节点中的 src 属性调用跨域的请求。当我们通过 JSONP 模式请求跨域资源时，服务器返回给客户端一段 javascript 代码，这段 javascript 代码自动调用客户端回调函数。

Demo:
前端:

```
<!DOCTYPE html>
<head>
    <title>jsonp</title>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
</head>
<body>
<script>
    //动态创建 script 标签, 并请求
    function addScriptTag(src){
        var script = document.createElement('script');
        script.setAttribute('type', 'text/javascript');
        script.src = src;
        document.body.appendChild(script);
    };
    //在 onload 后, 跨域请求
    window.onload = function(){
        addScriptTag('http://127.0.0.1:8080/jsonp?callback=test');
    };
    //回调函数,必须为全局, 不然会报错
    function test(data){
        alert(data.name);
    };
</script>
</body>
</html>
```

Node server:

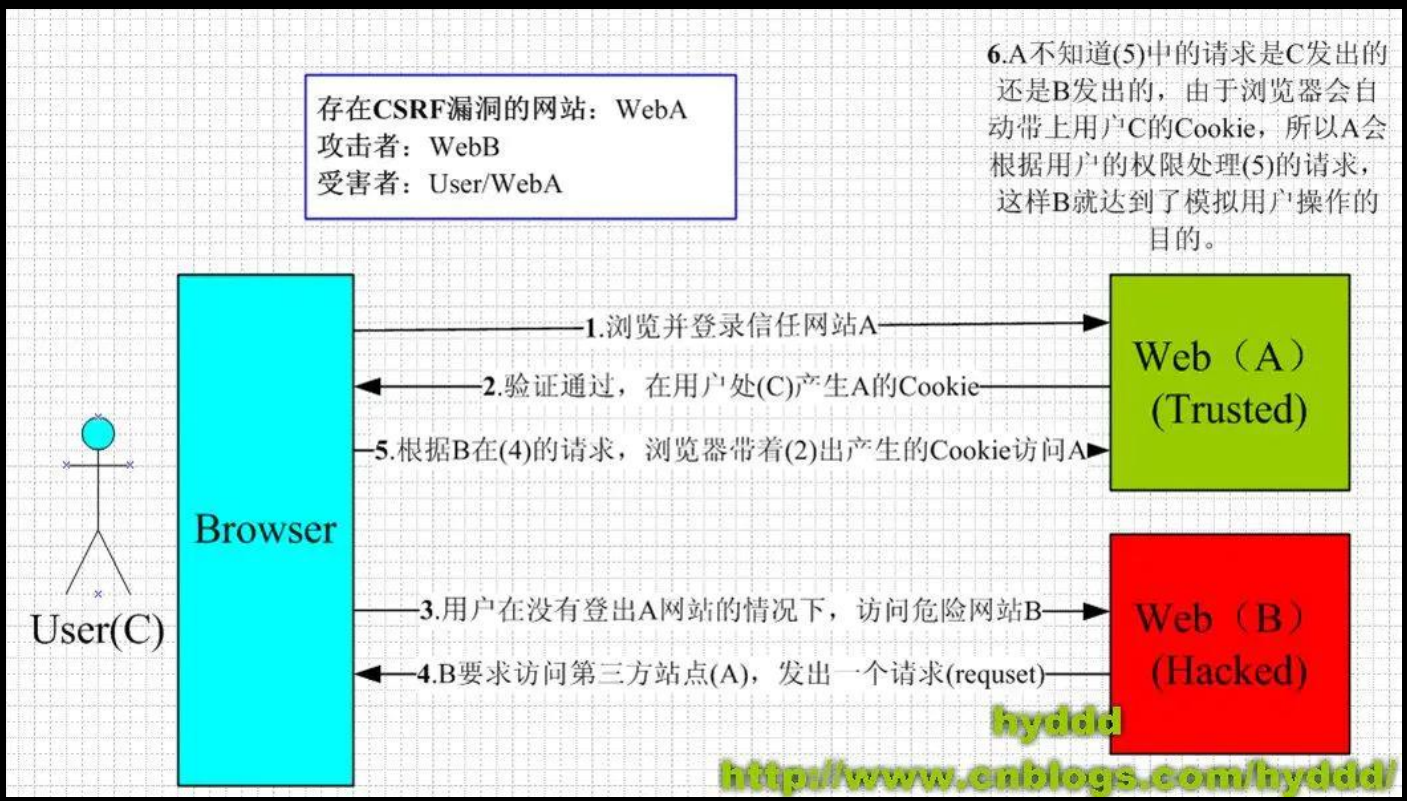
```
//告诉 Node.js 引入 http 模块给该服务器应用使用
var http = require('http');
//引入 url 模块解析 url 字符串
var url = require('url');
//引入 querystring 模块处理 query 字符串
var querystring = require('querystring');
//创建新的 HTTP 服务器
var server = http.createServer();
//通过 request 事件来响应 request 请求
server.on('request',function(req, res){
    var urlPath = url.parse(req.url).pathname;
    var qs = querystring.parse(req.url.split('?')[1]);
    if(urlPath === '/jsonp' && qs.callback){
        res.writeHead(200,{'Content-Type':'application/json;charset=utf-8'});
        var data = {
            "name": "Monkey"
        };
        data = JSON.stringify(data);
        var callback = qs.callback+'('+data+');';
        res.end(callback);
    }
    else{
        res.writeHead(200, {'Content-Type':'text/html;charset=utf-8'});
        res.end('Hell World\n');
    }
});
//监听 8080 端口
server.listen('8080');
//用于提示我们服务器启动成功
console.log('Server running!');
```

上面并没有使用其他请求，就是用的是 script 标签的。

补充 · CSRF

CSRF 是什么？

（Cross Site Request Forgery，跨站域请求伪造）是一种网络的攻击方式，它在 2007 年曾被列为互联网 20 大安全隐患之一，也被称为“One Click Attack”或者 Session Riding，通常缩写为 CSRF 或者 XSRF，是一种对网站的恶意利用。尽管听起来像跨站脚本（XSS），但它与 XSS 非常不同，并且攻击方式几乎相左。XSS 利用站点内的信任用户，而 CSRF 则通过伪装来自受信任用户的请求来利用受信任的网站。与 XSS 攻击相比，CSRF 攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比 XSS 更具危险性。



https://www.jianshu.com/p/7f33f9c7997b

同源策略

同源策略：同源策略是浏览器保证安全的基础，它的含义是指，A 网页设置的 Cookie，B 网页不能打开，除非这两个网页同源；同源指的是同协议、同域名、同端口；

PO VO DTO DO

VO（View Object）：视图对象，用于展示层，它的作用是把某个指定页面（或组件）的所有数据封装起来。

DTO（Data Transfer Object）：数据传输对象，这个概念来源于 J2EE 的设计模式，原来的目的是为了 EJB 的分布式应用提供粗粒度的数据实体，以减少分布式调用的次数，从而提高分布式调用的性能和降低网络负载，但在这里，我泛指用于展示层与服务层之间的数据传输对象。

DO（Domain Object）：领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。

PO（Persistent Object）：持久化对象，它跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么，数据表中的每个字段（或若干个）就对应 PO 的一个（或若干个）属性。

Base64

Base64 是一种最常见的二进制编码方法。
是一种基于**用 64 个可打印字符来表示二进制数据**的表示方法。

它通常用作存储、传输一些二进制数据编码方法！也是 MIME（多用途互联网邮件扩展，主要用作电子邮件标准）中一种可打印字符表示二进制数据的常见编码方法！它其实只是定义用可打印字符传输内容一种方法，并不会产生新的字符集！有时候，我们学习转换的思路后，我们其实也可以结合自己的实际需要，构造一些自己接口定义编码方式

用 64 个可打印字符表示二进制所有数据方法。由于 **2 的 6 次方等于 64，所以可以用每 6 个位元为一个单元**，对应某个可打印字符。我们知道三个字节有 24 个位元，就可以刚好对应于 4 个 Base64 单元，即 **3 个字节需要用 4 个 Base64 的可打印字符来表示。**在 Base64 中的可打印字符包括字母 A-Z、a-z、数字 0-9 ，这样共有 62 个字符，此外两个可打印符号在不同的系统中一般有所

不同。但是，我们经常所说的 Base64 另外 2 个字符是：“+/”。这 64 个字符，所对应表如下。

编号	字符		编号	字符		编号	字符		编号	字符
0	A		16	Q		32	g		48	w
1	B		17	R		33	h		49	x
2	C		18	S		34	i		50	y
3	D		19	T		35	j		51	z
4	E		20	U		36	k		52	0
5	F		21	V		37	l		53	1
6	G		22	W		38	m		54	2
7	H		23	X		39	n		55	3
8	I		24	Y		40	o		56	4
9	J		25	Z		41	p		57	5
10	K		26	a		42	q		58	6
11	L		27	b		43	r		59	7
12	M		28	c		44	s		60	8
13	N		29	d		45	t		61	9
14	O		30	e		46	u		62	+
15	P		31	f		47	v		63	/

转换的时候，将三个 byte 的数据，先后放入一个 24bit 的缓冲区中，先来的 byte 占高位。数据不足 3byte 的话，于缓冲区中剩下的 bit 用 0 补足。然后，每次取出 6 个 bit，按照其值选择 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ 中的字符作为编码后的输出。不断进行，直到全部输入数据转换完成。

如果最后剩下两个输入数据，在编码结果后加 1 个“=”；如果最后剩下一个输入数据，编码结果后加 2 个“=”；如果没有剩下任何数据，就什么都不要加，这样才可以保证资料还原的正确性。

编码后的数据比原始数据略长，为原来的 4/3。无论什么样的字符都会全部被编码，因此不像 Quoted-printable 编码，还保留部分可打印字符。所以，它的可读性不如 Quoted-printable 编码！

Session 的存储

SESSION 的数据保存在哪里呢?
PHP 中的 session 存储

SESSION 的数据保存在哪里呢?

当然是在服务器端，但不是保存在内存中，而是保存在文件或数据库中。

默认情况下，PHP.ini 中设置的 SESSION 保存方式是 files（session.save_handler =

files），即使用读写文件的方式保存 SESSION 数据，而 SESSION 文件保存的目录由 session.save_path 指定，文件名以 sess_ 为前缀，后跟 SESSION ID，如：sess_c72665af28a8b14c0fe11afe3b59b51b。文件中的数据即是序列化之后的 SESSION 数据了。

如果访问量大，可能产生的 SESSION 文件会比较多，这时可以设置分级目录进行 SESSION 文件的保存，效率会提高很多，设置方法为：session.save_path="N;/save_path"，N 为分级的级数，save_path 为开始目录。

当写入 SESSION 数据的时候，php 会获取到客户端的 SESSION_ID，然后根据这个 SESSION ID 到指定的 SESSION 文件保存目录中找到相应的 SESSION 文件，不存在则创建之，最后将数据序列化之后写入文件【3】。读取 SESSION 数据是也是类似的操作流程，对读出来的数据需要进行解序列化，生成相应的 SESSION 变量。

Java 中的 session 存储

sessionid 是一个会话的 key，浏览器第一次访问服务器会在服务器端生成一个 session，有一个 sessionid 和它对应。tomcat 生成的 sessionid 叫做 jsessionid。

session 在访问 tomcat 服务器 HttpServletRequest 的 getSession(true)的时候创建，tomcat 的 ManagerBase 类提供创建 sessionid 的方法：随机数+时间+jvmid。

存储在服务器的内存中，tomcat 的 StandardManager 类将 session 存储在内存中，也可以持久化到 file，数据库，memcache，redis 等。客户端只保存 sessionid 到 cookie 中，而不会保存 session，session 销毁只能通过 invalidate 或超时，关掉浏览器并不会关闭 session。

前后端分离

无论是 shiro 还是 secuitry，都是十几年前的产物，都是为了 jsp 那一套量身定做的，在前后端分离的大趋势下，其设计思想已经跟不上时代，很多功能点都需要进行二次封装，甚至找一大堆扩展插件才能集成，已经逐渐不太适合现代化项目的应用

secuitry 的二次开发是真的麻烦

没办法，设计思想上就是为了 jsp 那一套服务的，想要自己的功能，就得二次扩展

二次开发好难!! 不会弄

Edwin Xu