


shiro 源码分析（五）CredentialsMatcher


Realm 在验证用户身份的时候，要进行密码匹配。最简单的情况就是明文直接匹配，然后就是加密匹配，这里的匹配工作则就是交给 CredentialsMatcher 来完成的。先看下它的接口方法：

Java 代码 

```
1. public interface CredentialsMatcher {
2.     boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info);
3. }
```

根据用户名获取 AuthenticationInfo ，然后就需要将用户提交的 AuthenticationToken 和 AuthenticationInfo 进行匹配。

AuthenticatingRealm 从第三篇文章知道是用来进行认证流程的，它有一个属性 CredentialsMatcher credentialsMatcher，使用如下：

Java 代码 

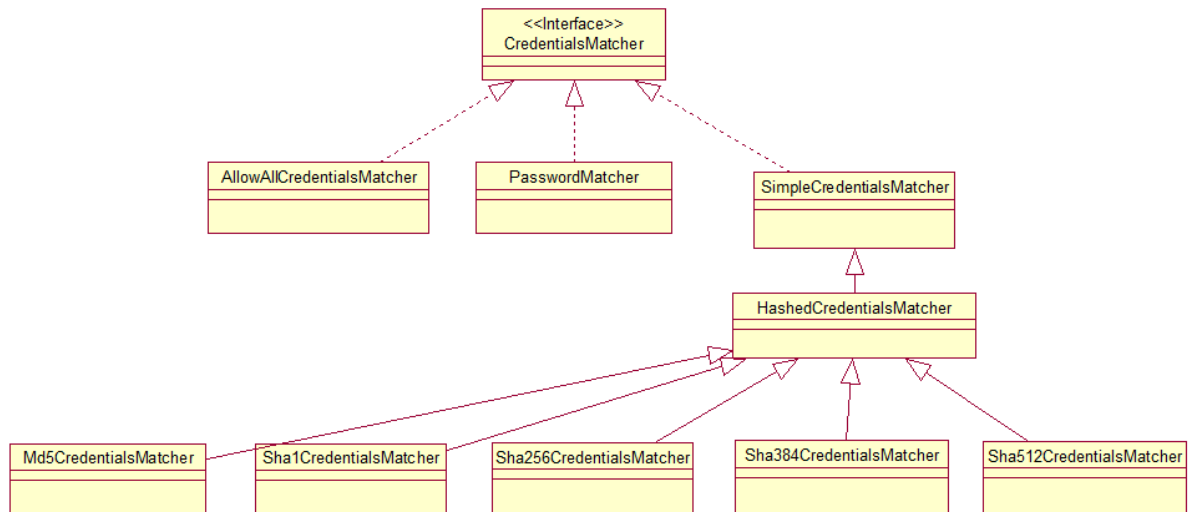
```
1. public final AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
2.
3.     AuthenticationInfo info = getCacheAuthenticationInfo(token);
4.     if (info == null) {
5.         //otherwise not cached, perform the lookup:
6.         info = doGetAuthenticationInfo(token);
7.         log.debug("Looked up AuthenticationInfo [{}] from doGetAuthenticationInfo", info);
8.         if (token != null && info != null) {
9.             cacheAuthenticationInfoIfPossible(token, info);
10.        }
11.    } else {
12.        log.debug("Using cached authentication info [{}] to perform credentials matching.", info);
13.    }
14.
15.    if (info != null) {
16.        //在这里进行认证密码匹配
```

```

17.         assertCredentialsMatch(token, info);
18.     } else {
19.         log.debug("No AuthenticationInfo found for submitted AuthenticationToken [{}]. Returning null.", token);
20.     }
21.
22.     return info;
23. }
24. protected void assertCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) throws AuthenticationException {
25.     CredentialsMatcher cm = getCredentialsMatcher();
26.     if (cm != null) {
27.         if (!cm.doCredentialsMatch(token, info)) {
28.             //not successful - throw an exception to indicate this:
29.             String msg = "Submitted credentials for token [" + token + "] did not match the expected credentials.";
30.             throw new IncorrectCredentialsException(msg);
31.         }
32.     } else {
33.         throw new AuthenticationException("A CredentialsMatcher must be configured in order to verify " +
34.             "credentials during authentication. If you do not wish for credentials to be examined, you " +
35.             "can configure an " + AllowAllCredentialsMatcher.class.getName() + " instance.");
36.     }
37. }

```

以上我们知道了 **CredentialsMatcher** 所处的认证的位置及作用，下面就要详细看看具体的匹配过程，还是接口设计图：



对于上图的三个分支，一个一个来说。

对于 AllowAllCredentialsMatcher:

Java 代码 ☆

```
1. public class AllowAllCredentialsMatcher implements CredentialsMatcher {
2.     public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
3.         return true;
4.     }
5. }
```

都返回 **true**，这意味着，只要该用户名存在即可，不用去验证密码是否匹配。

对于 PasswordMatcher:

Java 代码 ☆

```
1. public class PasswordMatcher implements CredentialsMatcher {
2.
3.     private PasswordService passwordService;
4.
5.     public PasswordMatcher() {
6.         this.passwordService = new DefaultPasswordService();
7.     }
8.     public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
9.         //确保有 PasswordService, 若没有抛异常
```

```

10. PasswordService service = ensurePasswordService();
11. //获取提交的密码
12. Object submittedPassword = getSubmittedPassword(token);
13. //获取服务器端存储的密码
14. Object storedCredentials = getStoredPassword(info);
15. //服务器端存储的密码必须是 String 或者 Hash 类型（待会详细介绍什么是 Hash），见该方法
16. assertStoredCredentialsType(storedCredentials);
17.
18. //对服务器端存储的密码分成两类来处理，一类是 String，另一类是 Hash
19. if (storedCredentials instanceof Hash) {
20.     Hash hashedPassword = (Hash)storedCredentials;
21.     HashingPasswordService hashingService = assertHashingPasswordService(service);
22.     return hashingService.passwordsMatch(submittedPassword, hashedPassword);
23. }
24. //otherwise they are a String (asserted in the 'assertStoredCredentialsType' method call above):
25. String formatted = (String)storedCredentials;
26. return passwordService.passwordsMatch(submittedPassword, formatted);
27. }
28. private void assertStoredCredentialsType(Object credentials) {
29.     if (credentials instanceof String || credentials instanceof Hash) {
30.         return;
31.     }
32.
33.     String msg = "Stored account credentials are expected to be either a " +
34.         Hash.class.getName() + " instance or a formatted hash String.";
35.     throw new IllegalArgumentException(msg);
36. }
37.
38. }

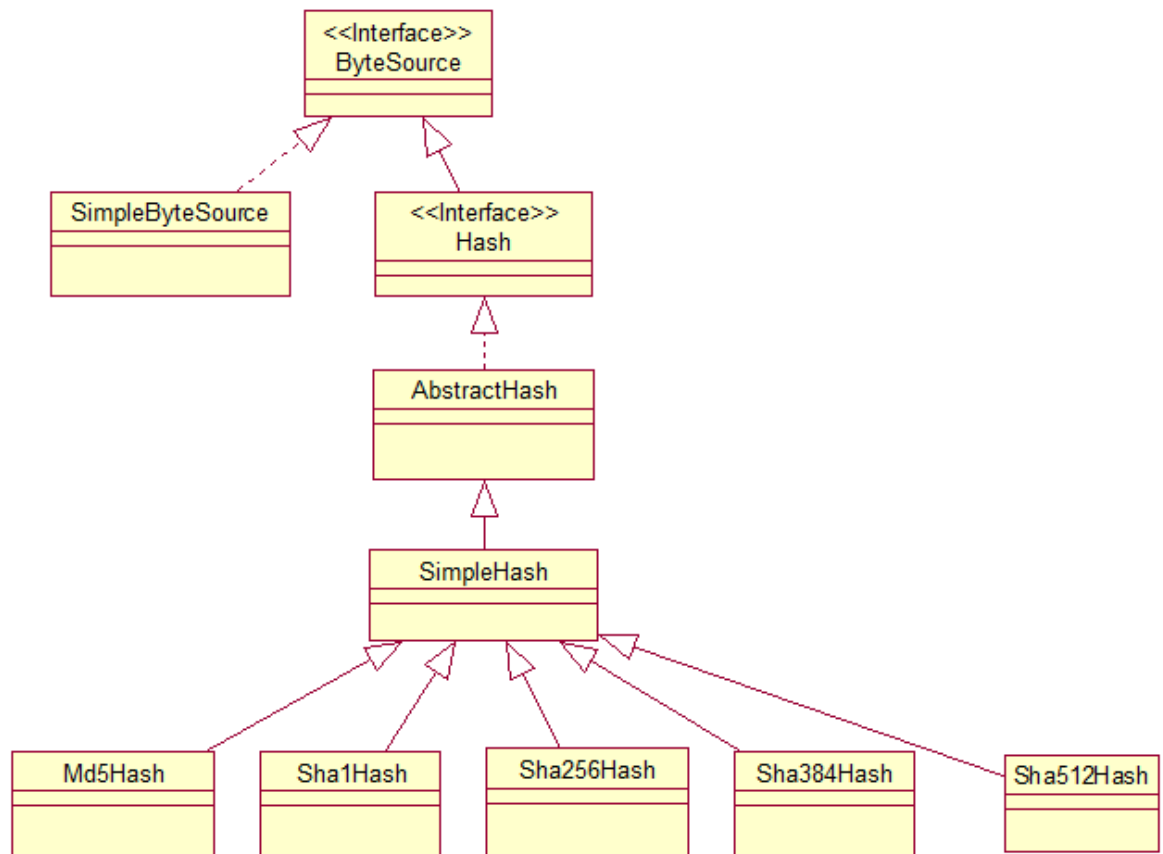
```

内部使用一个 **PasswordService** 来完成匹配。从上面的匹配过程中，我们了解到了，对于服务器端存储的密码分成 **String** 和 **Hash** 两种，然后由 **PasswordService** 来分别处理。所以 **PasswordMatcher** 也只

是完成了一个流程工作，具体的内容要到 **PasswordService** 来看。

到底什么是 **Hash** 呢？

先看下接口图：



看下 **ByteSource**:

Java 代码 ☆

```
1. public interface ByteSource {
2.     byte[] getBytes();
3.     String toHex();
4.     String toBase64();
5.     //略
6. }
```

就维护了一个 **byte[]** 数组。

看下 **SimpleByteSource** 的实现：

Java 代码 ☆

```
1. public class SimpleByteSource implements ByteSource {
2.
3.     private final byte[] bytes;
4.     private String cachedHex;
5.     private String cachedBase64;
6.
7.     public SimpleByteSource(byte[] bytes) {
8.         this.bytes = bytes;
9.     }
10.    public String toHex() {
11.        if ( this.cachedHex == null ) {
12.            this.cachedHex = Hex.encodeToString(getBytes());
13.        }
14.        return this.cachedHex;
15.    }
16.
17.    public String toBase64() {
18.        if ( this.cachedBase64 == null ) {
19.            this.cachedBase64 = Base64.encodeToString(getBytes());
20.        }
21.        return this.cachedBase64;
22.    }
23.    //略
24. }
```

toHex 就是将 byte 数组转换成 16 进制形式的字符串。toBase64 就是将 byte 数组进行 base64 编码。

Hex.encodeToString(getBytes()) 详情如下：

Java 代码 ☆

```
1. public class Hex {
2.
3.     /**
4.      * Used to build output as Hex
5.      */
6.     private static final char[] DIGITS = {
7.         '0', '1', '2', '3', '4', '5', '6', '7',
8.         '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
9.     };
10.    public static String encodeToString(byte[] bytes) {
```

```

11.         char[] encodedChars = encode(bytes);
12.         return new String(encodedChars);
13.     }
14.     public static char[] encode(byte[] data) {
15.
16.         int l = data.length;
17.
18.         char[] out = new char[l << 1];
19.
20.         // two characters form the hex value.
21.         for (int i = 0, j = 0; i < l; i++) {
22.             out[j++] = DIGITS[(0xF0 & data[i]) >>> 4];
23.             out[j++] = DIGITS[0x0F & data[i]];
24.         }
25.
26.         return out;
27.     }
28.     //略
29. }

```

对于一个 byte[] data 数组，byte 含有 8 位，(0xF0 & data[i]) >>> 4 表示取其高四位的值。如

当 data[i]=01001111 时，0xF0 & data[i]则为 01000000,然后右移四位则变成 00000100 即为值 4，所以 DIGITS[(0xF0 & data[i])>>>4]=DIGITS[4]=4，同理 data[i]的低四位变成 f。最终的结果为一个 byte 01001111 变成两个 char 4f。

Base64.encodeToString(getBytes()): 就稍微比较麻烦，这里不再详细说明。原理的话可以到网上搜下，有很多这样的文章。还是回到 ByteSource 的接口图，该轮到 Hash 了。

Java 代码 ☆

```
1. public interface Hash extends ByteSource {
2.     String getAlgorithmName();
3.     ByteSource getSalt();
4.     int getIterations();
5. }
```

多添加了三个属性,算法名、盐值、hash 次数。

继续看 Hash 的实现者 AbstractHash:

Java 代码 ☆

```
1. public AbstractHash(Object source, Object salt, int hashIterations) throws C
   odecException {
2.     byte[] sourceBytes = toBytes(source);
3.     byte[] saltBytes = null;
4.     if (salt != null) {
5.         saltBytes = toBytes(salt);
6.     }
7.     byte[] hashedBytes = hash(sourceBytes, saltBytes, hashIterations);
8.     setBytes(hashedBytes);
9. }
```

整个过程就是根据源 source 和 salt 和 hashIterations (hash 次数), 算出一个新的 byte 数组。

再来看下是如何生成新数组的:


Java 代码 ☆

```
1. protected byte[] hash(byte[] bytes, byte[] salt, int hashIterations) throw
   s UnknownAlgorithmException {
2.     MessageDigest digest = getDigest(getAlgorithmName());
3.     if (salt != null) {
4.         digest.reset();
5.         digest.update(salt);
6.     }
7.     byte[] hashed = digest.digest(bytes);
8.     int iterations = hashIterations - 1; //already hashed once above
9.     //iterate remaining number:
10.    for (int i = 0; i < iterations; i++) {
11.        digest.reset();
12.        hashed = digest.digest(hashed);
```



```
13.     }
14.     return hashed;
15. }
```

看到这里就明白了，`MessageDigest` 是 jdk 自带的 `java.security` 包中的工具，用于对数据进行加密。可以使用不同的加密算法，举个简单的例子，如用 `md5` 进行加密。`md5` 是对一个任意的 `byte` 数组进行加密变成固定长度的 128 位，即 16 个字节。然后这 16 个字节的展现有多种形式，这就与 `md5` 本身没关系了。展现形式如：把加密后的 128 位即 16 个字节进行 `Hex.encodeToString` 操作，即每个字节转换成两个字符（高四位一个字符，低四位一个字符）。到这个网址 <http://www.cmd5.com/> 中去输入字符串 "lg", 得到的 `md5 ("lg",32)` 的结果为 `a608b9c44912c72db6855ad555397470`，下面我们就来做出此结果

Java 代码 

```
1. public static void main(String[] args) throws NoSuchAlgorithmException, Unsu
   pportedEncodingException{
2.     MessageDigest md5=MessageDigest.getInstance("MD5");
3.     String str="lg";
4.     md5.reset();
5.     byte[] ret=md5.digest(str.getBytes("UTF-8"));
6.     System.out.println(Hex.encodeToString(ret));
7. }
```

`md5.reset()` 表示要清空要加密的源数据。`digest (byte[])` 表示将该数据填充到源数据中，然后加密。

`md5` 算出结果 `byte[] ret` 后，我们选择的展现形式是

`Hex.encodeToString(ret)` 即转换成 16 进制字符表示。这里的 `Hex` 就是借用 `shiro` 的 `Hex`。结果如下：

```
1. a608b9c44912c72db6855ad555397470
```

和上面的结果一样，也就是说该网址对 md5 加密后的结果也是采用转换成 16 进制字符的展现形式。该网址的 md5(lg,16) = 4912c72db6855ad5 则是取自上述结果的中间字符。

简单介绍完 md5 后，继续回到 AbstractHash 的 hash 方法中，就变得很简单。digest.update(salt)方法就是向源数据中继续添加要加密的数据，digest.digest(hash)内部调用了 update 方法即先填充数据，然后执行加密过程。

所以这里的过程为：

第一轮： salt 和 bytes 作为源数据加密得到 hashed byte 数组

第二轮： 如果传递进来的 hashIterations hash 次数大于 1 的话，要对上述结果继续进行加密
得到最终的加密结果。

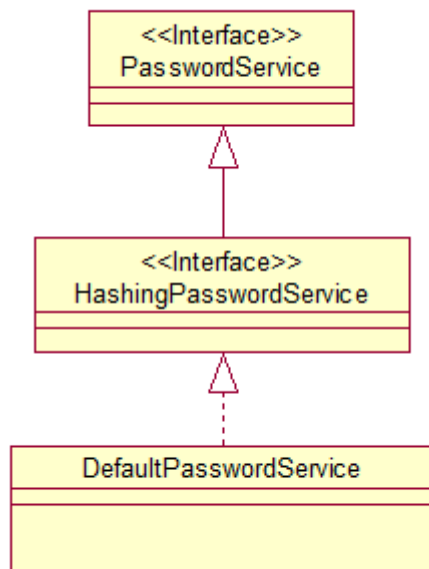
AbstractHash 对子类留了一个抽象方法 public abstract String getAlgorithmName()，用于获取加密算法名称。然而此类被标记为过时，推荐使用它的子类 SimpleHash，不过上述原理仍然没有变，不再详细去说，可以自己去查看，Hash 终于解释完了，总结一下，就是根据源字节数组、算法、salt、hash 次数得到一个加密的 byte 数组。


回到 CredentialsMatcher 的实现类 PasswordMatcher 中，在该类中，对服务器端存储的密码形式分成了两类，一类是 String，另一类就是

Hash, Hash 中包含了加密采用的算法、salt、hash 次数等信息。

PasswordMatcher 中的 PasswordService 来完成匹配过程。我们就可以试想匹配过程：若服务器端存储的密码为 Hash a，则我们就能知道加密过程所采用的算法、salt、hash 次数信息，然后对原密码进行这样的加密，算出一个 Hash b，然后比较 a b 的 byte 数组是否一致，这只是推想，然后来看下实际内容：

PasswordService 接口图如下：



Java 代码 

```
1. public interface PasswordService {
2.     String encryptPassword(Object plaintextPassword) throws IllegalArgumentException;
3.     boolean passwordsMatch(Object submittedPlaintext, String encrypted);
4. }
```

HashingPasswordService: 继承了 PasswordService ，加入了对 Hash 处理的功能

Java 代码 ☆

```
1. public interface HashingPasswordService extends PasswordService {
2.     //根据服务器端存储的 Hash 的采用的算法、salt、hash 次数和原始密码得到一个经过相同加密过程的 Hash
3.     Hash hashPassword(Object plaintext) throws IllegalArgumentException;
4.     boolean passwordsMatch(Object plaintext, Hash savedPasswordHash);
5. }
```

最终的实现类 DefaultPasswordService:

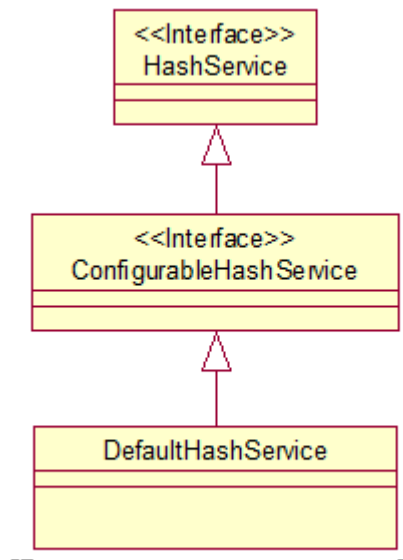
Java 代码 ☆

```
1. public class DefaultPasswordService implements HashingPasswordService {
2.
3.     public static final String DEFAULT_HASH_ALGORITHM = "SHA-256";
4.     public static final int DEFAULT_HASH_ITERATIONS = 500000; //500,000
5.
6.     private static final Logger log = LoggerFactory.getLogger(DefaultPasswordService.class);
7.
8.     private HashService hashService;
9.     private HashFormat hashFormat;
10.    private HashFormatFactory hashFormatFactory;
11.
12.    private volatile boolean hashFormatWarned; //used to avoid excessive logging noise
13.
14.    public DefaultPasswordService() {
15.        this.hashFormatWarned = false;
16.
17.        DefaultHashService hashService = new DefaultHashService();
18.        hashService.setHashAlgorithmName(DEFAULT_HASH_ALGORITHM);
19.        hashService.setHashIterations(DEFAULT_HASH_ITERATIONS);
20.        hashService.setGeneratePublicSalt(true); //always want generated salts for user passwords to be most secure
21.        this.hashService = hashService;
22.
23.        this.hashFormat = new Shiro1CryptFormat();
24.        this.hashFormatFactory = new DefaultHashFormatFactory();
25.    }
26.    //略
27. }
```

首先还是先了解属性，三个重要属性 HashService 、 HashFormat、

HashFormatFactory 。

HashService 接口类图：



Java 代码 ☆

```
1. public interface HashService {
2.     Hash computeHash(HashRequest request);
3. }
```

将一个 HashRequest 计算出一个 Hash。什么是 HashRequest？

Java 代码 ☆

```
1. public interface HashRequest {
2.     ByteSource getSource();
3.     ByteSource getSalt();
4.     int getIterations();
5.     String getAlgorithmName();
6.     //略
7. }
```

就是我们上述所说的那几个重要元素。原密码、salt、hash 次数、算法名称。这个计算过程也就是上述 AbstractHash 的过程。

再看 HashService 的子类 ConfigurableHashService：

Java 代码 ☆

```
1. public interface ConfigurableHashService extends HashService {
2.     void setPrivateSalt(ByteSource privateSalt);
3.     void setHashIterations(int iterations);
4.     void setHashAlgorithmName(String name);
5.     void setRandomNumberGenerator(RandomNumberGenerator rng);
6. }
```

就是可以对上述几个重要元素进行设置。`privateSalt` 和

`RandomNumberGenerator` 接下来再说，再看

`ConfigurableHashService` 的实现类 `DefaultHashService`:

Java 代码 ☆

```
1. public class DefaultHashService implements ConfigurableHashService {
2.     //主要是用来生成随机的 publicSalt
3.     private RandomNumberGenerator rng;
4.     private String algorithmName;
5.     private ByteSource privateSalt;
6.     private int iterations;
7.     //标志是否去产生 publicSalt
8.     private boolean generatePublicSalt;
9.     public DefaultHashService() {
10.         this.algorithmName = "SHA-512";
11.         this.iterations = 1;
12.         this.generatePublicSalt = false;
13.         this.rng = new SecureRandomNumberGenerator();
14.     }
15. }
```

来看下它是怎么实现将 `HashRequest` 变成 `Hash` 的:

Java 代码 ☆

```
1. public Hash computeHash(HashRequest request) {
2.     if (request == null || request.getSource() == null || request.getSource().isEmpty()) {
3.         return null;
4.     }
5.     //获取算法名字
6.     String algorithmName = getAlgorithmName(request);
7.     //获取原密码
8.     ByteSource source = request.getSource();
```


```

9.         //获取 hash 次数
10.        int iterations = getIterations(request);
11.        //获取 publicSalt
12.        ByteSource publicSalt = getPublicSalt(request);
13.        //获取 privateSalt
14.        ByteSource privateSalt = getPrivateSalt();
15.        //结合两者
16.        ByteSource salt = combine(privateSalt, publicSalt);
17.        //这就是之前始终强调的原理部分，就是根据算法、原始数据、salt、hash 次数进行
    加密
18.        Hash computed = new SimpleHash(algorithmName, source, salt, iterations);
19.
20.        //对于 computed 有很多信息，只想对外暴露某些信息。如 publicSalt
21.        SimpleHash result = new SimpleHash(algorithmName);
22.        result.setBytes(computed.getBytes());
23.        result.setIterations(iterations);
24.        //Only expose the public salt - not the real/combined salt that might have been used:
25.        result.setSalt(publicSalt);
26.
27.        return result;
28.    }

```

第一步：获取算法，先获取 `request` 本身的算法，如果没有，则使用 `DefaultHashService` 默认算法，在 `DefaultHashService` 的构造函数中默认使用 `SHA-512` 的加密算法。同理对于 `hash` 次数也是同样的逻辑。

第二步：获取 `publicSalt`

Java 代码 

```

1. protected ByteSource getPublicSalt(HashRequest request) {
2.
3.     ByteSource publicSalt = request.getSalt();
4.
5.     if (publicSalt != null && !publicSalt.isEmpty()) {
6.         //a public salt was explicitly requested to be used - go ahead and use it:
7.         return publicSalt;
8.     }

```

```

9.
10.         publicSalt = null;
11.
12.         //check to see if we need to generate one:
13.         ByteSource privateSalt = getPrivateSalt();
14.         boolean privateSaltExists = privateSalt != null && !privateSalt.isEmpty();
15.
16.         //If a private salt exists, we must generate a public salt to protect the integrity of the private salt.
17.         //Or generate it if the instance is explicitly configured to do so:
18.         if (privateSaltExists || isGeneratePublicSalt()) {
19.             publicSalt = getRandomNumberGenerator().nextBytes();
20.         }
21.
22.         return publicSalt;
23.     }

```

当 HashRequest request 本身有 salt 时，则充当 publicSalt 直接返回。当没有时，则需要去使用 RandomNumberGenerator 产生一个 publicSalt，当 DefaultHashService 的 privateSalt 存在或者 DefaultHashService 的 generatePublicSalt 标志为 true，都会去产生 publicSalt。

第三步：结合 publicSalt 和 privateSalt

第四步：Hash computed = new SimpleHash(algorithmName, source, salt, iterations)这就就是上文我们强调的加密核心，不再说明了，可以到上面去找。

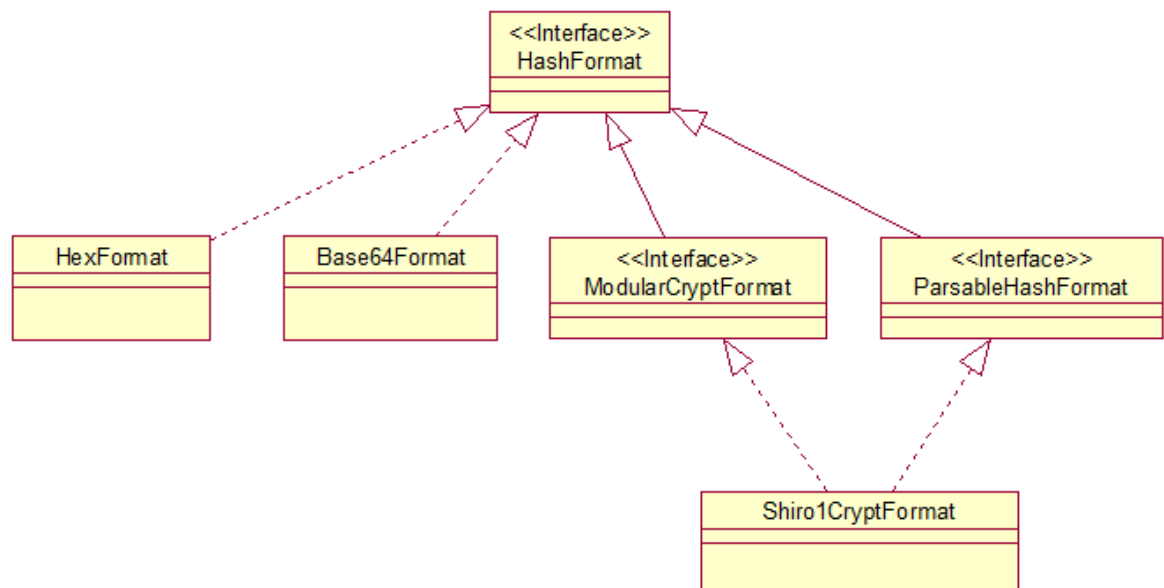
第五步：仅仅暴漏 Hash computed 中的某些属性，不把 privateSalt 暴漏出去。至此 DefaultHashService 的工作就全部完成了。

继续回到 DefaultPasswordService：看下一个类 HashFormat:

Java 代码 ☆

```
1. public interface HashFormat {  
2.     String format(Hash hash);  
3. }
```

这个就是对 Hash 进行格式化输出而已，看下接口设计：



HexFormat 如下

Java 代码 ☆

```
1. public class HexFormat implements HashFormat {  
2.     public String format(Hash hash) {  
3.         return hash != null ? hash.toHex() : null;  
4.     }  
5. }
```

就是调用 Hash 本身的 toHex 方法，同理 Hash 本身也有 String toBase64()方法，所以 Base64Format 也是同样的道理。


ModularCryptFormat 和 ParsableHashFormat 如下

Java 代码 ☆

```
1. public interface ModularCryptFormat extends HashFormat {  
2.     public static final String TOKEN_DELIMITER = "$";
```

```
3.     String getId();
4. }
5. public interface ParsableHashFormat extends HashFormat {
6.     Hash parse(String formatted);
7. }
```

他们的实现类 **Shiro1CryptFormat**，来看看是如何 **format** 的和如何 **parse** 的：


Java 代码 

```
1. public String format(Hash hash) {
2.     if (hash == null) {
3.         return null;
4.     }
5.
6.     String algorithmName = hash.getAlgorithmName();
7.     ByteSource salt = hash.getSalt();
8.     int iterations = hash.getIterations();
9.     StringBuilder sb = new StringBuilder(MCF_PREFIX).append(algorithmName)
10.        .append(TOKEN_DELIMITER).append(iterations).append(TOKEN_DELIMITER);
11.
12.     if (salt != null) {
13.         sb.append(salt.toBase64());
14.     }
15.
16.     sb.append(TOKEN_DELIMITER);
17.     sb.append(hash.toBase64());
18.
19.     return sb.toString();
20. }
```

format 就是将一些算法信息、**hash** 次数、**salt** 等进行字符串的拼接，**parse** 过程则是根据拼接的信息逆向获取算法信息、**hash** 次数、**salt** 等信息而已。这里就终于明白了，为什么 **PasswordMatcher** 对服务器端存储的密码分成 **Hash** 和 **String** 来处理了，他们都是存储算法、**salt**、**hash** 次数等信息的地方，**Hash** 直接是以结构化的类来存储，而 **String** 则是以格式化的字符串来存储，需要 **parse** 才能获取算法、**salt**

等信息。


HashFormat 则也完成了。DefaultPasswordService 还剩最后一个 HashFormatFactory 了，它则是用来生成不同的 HashFormat 的。

Java 代码 

```
1. public interface HashFormatFactory {  
2.     HashFormat getInstance(String token);  
3. }
```

根据 String 密码（格式化过的）来寻找对应的 HashFormat。这里不再详细介绍了，有兴趣的可以自己去看研究。

回到我们关注的重点，密码匹配过程：DefaultPasswordService

Java 代码 

```
1. public DefaultPasswordService() {  
2.     this.hashFormatWarned = false;  
3.  
4.     DefaultHashService hashService = new DefaultHashService();  
5.     hashService.setHashAlgorithmName(DEFAULT_HASH_ALGORITHM);  
6.     hashService.setHashIterations(DEFAULT_HASH_ITERATIONS);  
7.     hashService.setGeneratePublicSalt(true); //always want generated salts for user passwords to be most secure  
8.     this.hashService = hashService;  
9.  
10.    this.hashFormat = new Shiro1CryptFormat();  
11.    this.hashFormatFactory = new DefaultHashFormatFactory();  
12. }
```

使用了，DefaultHashService 和 Shiro1CryptFormat 和 DefaultHashFormatFactory。

先来看看是如何匹配加密密码是 String 的，后面再看看是如何匹配 Hash 的

Java 代码 ☆

```
1. public boolean passwordsMatch(Object submittedPlaintext, String saved) {
2.     ByteSource plaintextBytes = createByteSource(submittedPlaintext);
3.
4.     if (saved == null || saved.length() == 0) {
5.         return plaintextBytes == null || plaintextBytes.isEmpty();
6.     } else {
7.         if (plaintextBytes == null || plaintextBytes.isEmpty()) {
8.             return false;
9.         }
10.    }
11.
12.    //First check to see if we can reconstitute the original hash - thi
s allows us to
13.    //perform password hash comparisons even for previously saved passwo
rds that don't
14.    //match the current HashService configuration values. This is a ver
y nice feature
15.    //for password comparisons because it ensures backwards compatibilit
y even after
16.    //configuration changes.
17.    HashFormat discoveredFormat = this.hashFormatFactory.getInstance(sav
ed);
18.
19.    if (discoveredFormat != null && discoveredFormat instanceof Parsable
HashFormat) {
20.
21.        ParsableHashFormat parsableHashFormat = (ParsableHashFormat)disc
overedFormat;
22.        Hash savedHash = parsableHashFormat.parse(saved);
23.
24.        return passwordsMatch(submittedPlaintext, savedHash);
25.    }
26.
27.    //If we're at this point in the method's execution, We couldn't reco
nstitute the original hash.
28.    //So, we need to hash the submittedPlaintext using current HashServi
ce configuration and then
29.    //compare the formatted output with the saved string. This will cor
rectly compare passwords,
30.    //but does not allow changing the HashService configuration withou
t breaking previously saved
31.    //passwords:
32.
```


```

33.         //The saved text value can't be reconstituted into a Hash instance. We need to format the
34.         //submittedPlaintext and then compare this formatted value with the saved value:
35.         HashRequest request = createHashRequest(plaintextBytes);
36.         Hash computed = this.hashService.computeHash(request);
37.         String formatted = this.hashFormat.format(computed);
38.
39.         return saved.equals(formatted);
40.     }

```

分成了两个分支，第一个分支就是能将加密的 `String` 密码使用 `HashFormat` 解析成 `Hash`，然后调用 `public boolean passwordsMatch(Object plaintext, Hash saved)` 即 `Hash` 的匹配方式，第二个分支就是，不能解析的情况下，把原始密码封装成 `HashRequest`，然后使用 `HashService` 来讲 `HashRequest` 计算出一个 `Hash`，再用 `HashFormat` 来格式化它变成 `String` 字符串，两个字符串进行 `equals` 比较。

对于 `Hash` 的匹配方式：

Java 代码 

```

1. public boolean passwordsMatch(Object plaintext, Hash saved) {
2.     ByteSource plaintextBytes = createByteSource(plaintext);
3.
4.     if (saved == null || saved.isEmpty()) {
5.         return plaintextBytes == null || plaintextBytes.isEmpty();
6.     } else {
7.         if (plaintextBytes == null || plaintextBytes.isEmpty()) {
8.             return false;
9.         }
10.    }
11.
12.    HashRequest request = buildHashRequest(plaintextBytes, saved);
13.
14.    Hash computed = this.hashService.computeHash(request);
15.
16.    return saved.equals(computed);

```


```

17.     }
18.     protected HashRequest buildHashRequest(ByteSource plaintext, Hash saved) {
19.         //keep everything from the saved hash except for the source:
20.         return new HashRequest.Builder().setSource(plaintext)
21.             //now use the existing saved data:
22.             .setAlgorithmName(saved.getAlgorithmName())
23.             .setSalt(saved.getSalt())
24.             .setIterations(saved.getIterations())
25.             .build();
26.     }

```

这个过程就是我们之前设想的过程，就是很据已由的 **Hash saved** 的算法、salt、hash 次数对 **Object plaintext** 进行同样的加密过程，然后匹配 **saved.equals(computed)** 的信息是否一致。至此我们就走通了 **PasswordMatcher** 的整个过程。这是 **CredentialsMatcher** 的第二个分支，我们继续看 **CredentialsMatcher** 的第三个分支

SimpleCredentialsMatcher:

Java 代码 

```

1.     public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationI
    nfo info) {
2.         Object tokenCredentials = getCredentials(token);
3.         Object accountCredentials = getCredentials(info);
4.         return equals(tokenCredentials, accountCredentials);
5.     }
6.     protected Object getCredentials(AuthenticationToken token) {
7.         return token.getCredentials();
8.     }
9.     protected Object getCredentials(AuthenticationInfo info) {
10.        return info.getCredentials();
11.    }
12.    protected boolean equals(Object tokenCredentials, Object accountCredential
    s) {
13.        if (log.isDebugEnabled()) {
14.            log.debug("Performing credentials equality check for tokenCreden
    tials of type [" +
15.                tokenCredentials.getClass().getName() + " and accountCre
    dentials of type [" +
16.                accountCredentials.getClass().getName() + "]);

```


```

17.     }
18.     if (isByteSource(tokenCredentials) && isByteSource(accountCredential
s)) {
19.         if (log.isDebugEnabled()) {
20.             log.debug("Both credentials arguments can be easily converte
d to byte arrays. Performing " +
21.                 "array equals comparison");
22.         }
23.         byte[] tokenBytes = toBytes(tokenCredentials);
24.         byte[] accountBytes = toBytes(accountCredentials);
25.         return Arrays.equals(tokenBytes, accountBytes);
26.     } else {
27.         return accountCredentials.equals(tokenCredentials);
28.     }
29. }

```

它的实现比较简单，就是直接比较 `AuthenticationToken` 的 `getCredentials()` 和 `AuthenticationInfo` 的 `getCredentials()` 内容，若为 `ByteSource` 则匹配下具体的内容，否则直接匹配引用。

看下它的子类 `HashedCredentialsMatcher` 的匹配过程：

Java 代码 

```

1. public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationI
nfo info) {
2.     Object tokenHashedCredentials = hashProvidedCredentials(token, inf
o);
3.     Object accountCredentials = getCredentials(info);
4.     return equals(tokenHashedCredentials, accountCredentials);
5. }

```

其中 `equals` 方法仍然是调用父类的方法，即一旦为 `ByteSource` 则进行 `byte` 匹配，否则进行引用匹配。只是这里的 `tokenHashedCredentials` 和 `accountCredentials` 和父类的方式不一样，如下：

Java 代码

```
1. protected Object hashProvidedCredentials(AuthenticationToken token, AuthenticationInfo info) {
2.     Object salt = null;
3.     if (info instanceof SaltedAuthenticationInfo) {
4.         salt = ((SaltedAuthenticationInfo) info).getCredentialsSalt();
5.     } else {
6.         //retain 1.0 backwards compatibility:
7.         if (isHashSalted()) {
8.             salt = getSalt(token);
9.         }
10.    }
11.    return hashProvidedCredentials(token.getCredentials(), salt, getHashIterations());
12. }
13. protected Hash hashProvidedCredentials(Object credentials, Object salt, int hashIterations) {
14.     String hashAlgorithmName = assertHashAlgorithmName();
15.     return new SimpleHash(hashAlgorithmName, credentials, salt, hashIterations);
16. }
```

可以看到仍然是使用算法名称和 **credentials**（用户提交的未加密的）、**salt**、**hash** 次数构建一个 **SimpleHash**（构造时进行加密）。再看对于已加密的 **credentials** 则是也构建一个 **SimpleHash**，但是不再进行加密过程：

Java 代码

```
1. protected Object getCredentials(AuthenticationInfo info) {
2.     Object credentials = info.getCredentials();
3.
4.     byte[] storedBytes = toBytes(credentials);
5.
6.     if (credentials instanceof String || credentials instanceof char[]) {
7.         //account.credentials were a char[] or String, so
8.         //we need to do text decoding first:
9.         if (isStoredCredentialsHexEncoded()) {
10.            storedBytes = Hex.decode(storedBytes);
11.        } else {
```




```

12.         storedBytes = Base64.decode(storedBytes);
13.     }
14. }
15.     AbstractHash hash = newHashInstance();
16.     hash.setBytes(storedBytes);
17.     return hash;
18. }
19. protected AbstractHash newHashInstance() {
20.     String hashAlgorithmName = assertHashAlgorithmName();
21.     return new SimpleHash(hashAlgorithmName);
22. }

```

对于 HashedCredentialsMatcher 也就是说 AuthenticationToken token, AuthenticationInfo info 都去构建一个 SimpleHash，前者构建时执行加密过程，后者（已加密）不需要去执行加密过程，然后匹配这两个 SimpleHash 是否一致。然后就是 HashedCredentialsMatcher 的子类（全部被标记为已废弃），如 Md5CredentialsMatcher：

Java 代码 

```

1. public class Md5CredentialsMatcher extends HashedCredentialsMatcher {
2.
3.     public Md5CredentialsMatcher() {
4.         super();
5.         setHashAlgorithmName(Md5Hash.ALGORITHM_NAME);
6.     }
7. }

```

仅仅是将 HashedCredentialsMatcher 的算法改为 md5，所以 Md5CredentialsMatcher 本身就没有存在的价值。HashedCredentialsMatcher 其他子类都是同样的道理。

至此 CredentialsMatcher 的三个分支都完成了。

已经很长了，下一篇文章以具体的案例来使用上述原理。

作者：乒乓狂魔