

shiro 源码分析（六）CredentialsMatcher 的案例分析

有了上一篇文章的原理分析，这一篇文章主要结合原理来进行使用。

shiro.ini 配置为：

Java 代码 ☆

```
1. [main]
2. #realm
3. dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
4. dataSource.driverClass=com.mysql.jdbc.Driver
5. dataSource.jdbcUrl=jdbc:mysql://localhost:3306/shiro
6. dataSource.user=XXXXXX
7. dataSource.password=XXXXX
8. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
9. jdbcRealm.dataSource=$dataSource
10. jdbcRealm.permissionsLookupEnabled=true
11. securityManager.realms=$jdbcRealm
```

代码为：

Java 代码 ☆

```
1. public class ShiroTest {
2.     @Test
3.     public void testHelloworld() {
4.         init();
5.
6.         Subject subject=login("lg","123");
7.         System.out.println(subject.hasRole("role1"));
8.         System.out.println(subject.hasRole("role2"));
9.         System.out.println(subject.hasRole("role3"));
10.    }
11.    private Subject login(String userName,String password){
12.        //3、得到 Subject 及创建用户名/密码身份验证 Token（即用户身份/凭证）
13.        Subject subject = SecurityUtils.getSubject();
14.        UsernamePasswordToken token = new UsernamePasswordToken(userName,password);
15.        subject.login(token);
16.        return subject;
17.    }
18.    private void init(){
```

```

19.         //1、获取 SecurityManager 工厂，此处使用 Ini 配置文件初始化 SecurityManager
20.         Factory<org.apache.shiro.mgt.SecurityManager> factory =
21.             new IniSecurityManagerFactory("classpath:shiro.ini");
22.         //2、得到 SecurityManager 实例 并绑定给 SecurityUtils
23.         org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
24.         SecurityUtils.setSecurityManager(securityManager);
25.     }
26. }

```

此案例，对于 `JdbcRealm` 并没有配置 `CredentialsMatcher`，它会使用默认的 `CredentialsMatcher` 即 `SimpleCredentialsMatcher`，如下：

Java 代码 

```

1. public AuthenticatingRealm() {
2.     this(null, new SimpleCredentialsMatcher());
3. }
4. public AuthenticatingRealm(CacheManager cacheManager) {
5.     this(cacheManager, new SimpleCredentialsMatcher());
6. }

```

从上一篇文章中知道 `SimpleCredentialsMatcher` 不进行加密，仅仅匹配密码对应的字节数组。

所以代码中用户 `lg` 的登陆密码为 `123`，则数据库中的密码也是明文 `123`。

下面我们就开始进行加密，首先从 `CredentialsMatcher` 的第三个分支来说，使用 `HashedCredentialsMatcher` 来进行加密。加密方式为 `salt` 自定义、`hash` 次数为 `2`、加密算法为 `md5` 的加密过程。配置文件如下所示：

Java 代码

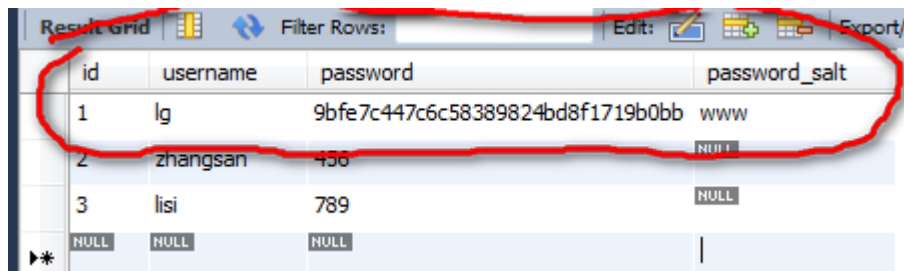
```
1. [main]
2. #realm
3. dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
4. dataSource.driverClass=com.mysql.jdbc.Driver
5. dataSource.jdbcUrl=jdbc:mysql://localhost:3306/shiro
6. dataSource.user=XXXX
7. dataSource.password=XXXX
8. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
9. jdbcRealm.dataSource=$dataSource
10. jdbcRealm.permissionsLookupEnabled=true
11. credentialsMatcher=org.apache.shiro.authc.credential.HashedCredentialsMatcher
12. credentialsMatcher.hashAlgorithmName=MD5
13. credentialsMatcher.hashIterations=2
14. jdbcRealm.credentialsMatcher=$credentialsMatcher
15. securityManager.realms=$jdbcRealm
```

如用户"lg"明文密码为"123", 假如 salt 为"www",hash 次数为 2, 加密算法为"MD5", 则密文可如下方式算出:

Java 代码

```
1. Hash hash=new SimpleHash("MD5", new SimpleByteSource("123"),new SimpleByteSource("www"),2);
2. System.out.println(hash.toHex());
```

经过 md5 加密后变成 byte 数组, 存在 Hash 的 bytes 属性中, 然后使用 hash.toHex()将 byte 数组转换成 16 进制的字符串, 最终结果为 "9bfe7c447c6c58389824bd8f1719b0bb",然后将该结果作为密码的密文存到数据库中, 同时我们把 salt 也存到数据库中, 则数据库中是如下记录:



	id	username	password	password_salt
1	1	lg	9bfe7c447c6c58389824bd8f1719b0bb	www
2	2	zhangsan	430	NULL
3	3	lisi	789	NULL
4	NULL	NULL	NULL	

数据库的数据准备完毕，然后就开始代码设置。第一个设置就是，读取用户"lg"的记录的时候要把 password_salt 读取出来，即 sql 语句应该为 DEFAULT_SALTED_AUTHENTICATION_QUERY =select password, password_salt from users where username = ?

然而默认的 sql 语句是：

DEFAULT_AUTHENTICATION_QUERY=select password from users where username = ?

如何才能达到上述替换结果呢？

Java 代码 ☆

```
1. public void setSaltStyle(SaltStyle saltStyle) {
2.     this.saltStyle = saltStyle;
3.     if (saltStyle == SaltStyle.COLUMN && authenticationQuery.equals(DEFAULT_AUTHENTICATION_QUERY)) {
4.         authenticationQuery = DEFAULT_SALTED_AUTHENTICATION_QUERY;
5.     }
6. }
```

从上面代码中可以看到，需要设置 JdbcRealm 的 saltStyle 为 SaltStyle.COLUMN。saltStyle 是一个枚举类型，然而在 ini 配置文件中，并不支持设置枚举类型，只能暂时在代码中如下解决：

Java 代码 ☆

```
1. Collection<Realm> realms=((RealmSecurityManager) securityManager).getRealms();
2.     JdbcRealm jdbcRealm=(JdbcRealm)realms.toArray()[0];
3.     jdbcRealm.setSaltStyle(SaltStyle.COLUMN);
```

或者开涛大神又给出另外一种解决方案：注册一个 Enum 转换器，这

个我准备在下一篇文章中给出 ini 配置文件的源码解析。

上述设置，就会使 JdbcRealm 从数据库中读出用户"lg"的

AuthenticationInfo 信息中含有密文和 salt。JdbcRealm 下一步就要进行 AuthenticationToken token（含有用户提交的明文密码"123"）

AuthenticationInfo info(含有密文密码

"9bfe7c447c6c58389824bd8f1719b0bb",和 salt "www")的匹配过程

Java 代码 ☆

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

hashProvidedCredentials 方法：HashedCredentialsMatcher 会将明文密码"123"进行如下类似的操作：

Java 代码 ☆

```
1. new SimpleHash("MD5", new SimpleByteSource("123"), new SimpleByteSource("www"), 2)
```

其中 md5 算法和 hash 次数 2 为我们所配置的，www 则是从数据库中读出来的。

得到 tokenHashedCredentials = 上述的结果。

getCredentials 方法：会将 AuthenticationInfo info 的密文密码先进行 decode,如下：

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. }
```

为什么呢？因为我们之前算出的密文是 **byte** 数组，然后进行了 16 进制转换变成字符串，所以这里要将密文密码

"9bfe7c447c6c58389824bd8f1719b0bb"先 decode 还原出 byte 数组
isStoredCredentialsHexEncoded()方法返回

HashedCredentialsMatcher 的 storedCredentialsHexEncoded 属性，默认为 true，即会进行 16 进制的 decode,正好符合我们的要求。如果设置为 false，则要进行 Base64 解码。

tokenHashedCredentials 和上述 getCredentials(AuthenticationInfo info)的结果的 byte 数组内容都是进过相同的算法和 salt 和 hash 次数，所以他们会匹配上，进而验证通过。

再来看下 `CredentialsMatcher` 的另一个分支 `PasswordMatcher` 的使用：

我们知道，根据上一篇文章的原理，`PasswordMatcher` 会对 `AuthenticationInfo` 的密码进行 `String` 和 `Hash` 的判断。而我们的 `JdbcRealm` 在创建获取用户的 `AuthenticationInfo` 时，默认采用的是 `char` 数组的形式存储的，如下：

Java 代码 ☆

```
1. protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token
   en) throws AuthenticationException {
2.
3.     UsernamePasswordToken upToken = (UsernamePasswordToken) token;
4.     String username = upToken.getUsername();
5.
6.     // Null username is invalid
7.     if (username == null) {
8.         throw new AccountException("Null usernames are not allowed by th
   is realm.");
9.     }
10.
11.    Connection conn = null;
12.    SimpleAuthenticationInfo info = null;
13.    try {
14.        conn = dataSource.getConnection();
15.
16.        String password = null;
17.        String salt = null;
18.        switch (saltStyle) {
19.            case NO_SALT:
20.                password = getPasswordForUser(conn, username)[0];
21.                break;
22.            case CRYPT:
23.                // TODO: separate password and hash from getPasswordForUser
   [0]
24.                throw new ConfigurationException("Not implemented yet");
25.                //break;
26.            case COLUMN:
27.                String[] queryResults = getPasswordForUser(conn, usernam
   e);
```

```

28.         password = queryResults[0];
29.         salt = queryResults[1];
30.         break;
31.     case EXTERNAL:
32.         password = getPasswordForUser(conn, username)[0];
33.         salt = getSaltForUser(username);
34.     }
35.
36.     if (password == null) {
37.         throw new UnknownAccountException("No account found for use
r [" + username + "]");
38.     }
39.
40. //重点在这里在这里在这里在这里在这里在这里 password.toCharArray()变成了 cha
r 数组
41.     info = new SimpleAuthenticationInfo(username, password.toCharArray(), getName());
42.
43.     if (salt != null) {
44.         info.setCredentialsSalt(ByteSource.Util.bytes(salt));
45.     }
46.
47. } catch (SQLException e) {
48.     final String message = "There was a SQL error while authenticati
ng user [" + username + "]";
49.     if (log.isDebugEnabled()) {
50.         log.error(message, e);
51.     }
52.
53.     // Rethrow any SQL errors as an authentication exception
54.     throw new AuthenticationException(message, e);
55. } finally {
56.     JdbcUtils.closeConnection(conn);
57. }
58.
59.     return info;
60. }

```

所以如果想让 `AuthenticationInfo` 存储的密码存储形式为 `Hash`，则需要我们来自定义 `JdbcRealm`。虽然是 `char` 数组，但 `PasswordMatcher` 对 `char` 数组转换成了 `String`，如下：

Java 代码

```
1. protected Object getStoredPassword(AuthenticationInfo storedAccountInfo) {
2.     Object stored = storedAccountInfo != null ? storedAccountInfo.getCredentials() : null;
3.     //fix for https://issues.apache.org/jira/browse/SHIRO-363
4.     if (stored instanceof char[]) {
5.         stored = new String((char[])stored);
6.     }
7.     return stored;
8. }
```

所以会调用 PasswordService 的 boolean passwordsMatch(Object submittedPlaintext, String encrypted)。

案例如下：

Java 代码


```
1. @Test
2.     public void testHelloworld() {
3.         init();
4.
5.         register("lisi", "456");
6.
7.         Subject subject=login("lisi", "456");
8.         System.out.println(subject.hasRole("role1"));
9.         System.out.println(subject.hasRole("role2"));
10.        System.out.println(subject.hasRole("role3"));
11.    }
12.
13.    public void register(String username,String password){
14.        JdbcRealm jdbcRealm=getJdbcRelam();
15.        PasswordMatcher passwordMatcher=(PasswordMatcher) jdbcRealm.getCredentialsMatcher();
16.        String encryptPassword=passwordMatcher.getPasswordService().encryptPassword(password);
17.        //保存用户名和密文到数据库，这里不再做
18.        System.out.println(encryptPassword);
19.    }
20.
21.    private Subject login(String userName,String password){
22.        //3、得到 Subject 及创建用户名/密码身份验证 Token（即用户身份/凭证）
23.        Subject subject = SecurityUtils.getSubject();
```

```

24.         UsernamePasswordToken token = new UsernamePasswordToken(userName,password);
25.         subject.login(token);
26.         return subject;
27.     }
28.
29.     private void init(){
30.         //1、获取 SecurityManager 工厂，此处使用 Ini 配置文件初始化 SecurityManager
31.         Factory<org.apache.shiro.mgt.SecurityManager> factory =
32.             new IniSecurityManagerFactory("classpath:shiro.ini");
33.         //2、得到 SecurityManager 实例 并绑定给 SecurityUtils
34.         org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
35.         SecurityUtils.setSecurityManager(securityManager);
36.         JdbcRealm jdbcRealm=getJdbcRelam();
37.         jdbcRealm.setSaltStyle(SaltStyle.COLUMN);
38.     }
39.
40.     public JdbcRealm getJdbcRelam(){
41.         Collection<Realm> realms=((RealmSecurityManager)SecurityUtils.getSecurityManager()).getRealms();
42.         return (JdbcRealm)realms.toArray()[0];
43.     }

```

ini 配置为:

Java 代码 

```

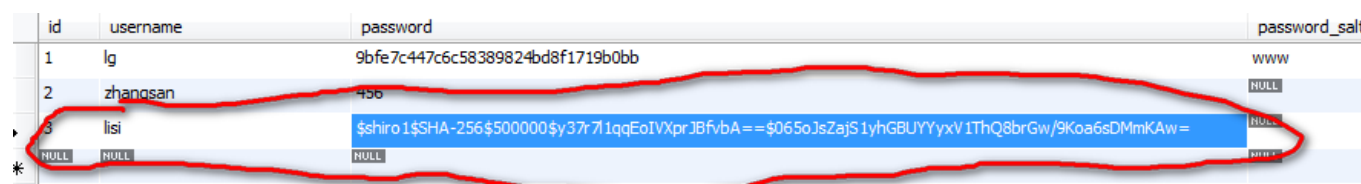
1. [main]
2. #realm
3. dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
4. dataSource.driverClass=com.mysql.jdbc.Driver
5. dataSource.jdbcUrl=jdbc:mysql://localhost:3306/shiro
6. dataSource.user=XXXX
7. dataSource.password=XXXX
8. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
9. jdbcRealm.dataSource=$dataSource
10. jdbcRealm.permissionsLookupEnabled=true
11. credentialsMatcher=org.apache.shiro.authc.credential.PasswordMatcher
12. jdbcRealm.credentialsMatcher=$credentialsMatcher
13. securityManager.realms=$jdbcRealm

```

第一个过程就是用户注册，对密码进行加密然后存到数据库的过程，

我们全部使用 PasswordMatcher 最简单的默认配置，获取密文过程即用户注册的过程，先根据 SecurityManager 拿到 JdbcRealm，再由 JdbcRealm 拿到 PasswordMatcher，再根据 PasswordMatcher 拿到 PasswordService，有了 PasswordService 就可以对明文密码进行加密了，打印的密文密码结果为：\$shiro1\$SHA-256\$500000\$y37r7l1qqEoIVXprJBfVbA==\$065oJsZajS1yhGBUYyXV1ThQ8brGw/9Koa6sDMmKAw=（每执行一次加密过程都会变，内部使用了随机生成 salt 的机制）

存到数据库中。如下截图：



id	username	password	password_salt
1	lg	9bfe7c447c6c58389824bd8f1719b0bb	WWW
2	zhangsan	456	NULL
3	lisi	\$shiro1\$SHA-256\$500000\$y37r7l1qqEoIVXprJBfVbA==\$065oJsZajS1yhGBUYyXV1ThQ8brGw/9Koa6sDMmKAw=	NULL
*	NULL	NULL	NULL

然后就可以直接用账号"lisi"和上述明文密码"456"来进行登陆，也可以登陆成功。

然后我们就分析下使用 PasswordService 对明文加密的过程和用户登录时的匹配过程（有了前一篇文章的原理分析，然后就能够更改默认配置，实现自己的需求）

先是 PasswordService 的各项默认配置：

我们 jdbcRealm 配置的 credentialsMatcher 是 org.apache.shiro.authc.credential.PasswordMatcher，它的配置如下：

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.     //内部使用了 DefaultPasswordService
9. }
```

再看 DefaultPasswordService 的配置：

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 HashService hashService;
7.     private HashFormat hashFormat;
8.     private HashFormatFactory hashFormatFactory;
9.
10.    private volatile boolean hashFormatWarned; //used to avoid excessive lo
    g noise
11.
12.    public DefaultPasswordService() {
13.        this.hashFormatWarned = false;
14.
15.        DefaultHashService hashService = new DefaultHashService();
16.        hashService.setHashAlgorithmName(DEFAULT_HASH_ALGORITHM);
17.        hashService.setHashIterations(DEFAULT_HASH_ITERATIONS);
18.        hashService.setGeneratePublicSalt(true); //always want generated sal
    ts for user passwords to be most secure
19.        this.hashService = hashService;
20.
21.        this.hashFormat = new Shiro1CryptFormat();
22.        this.hashFormatFactory = new DefaultHashFormatFactory();
23.    }
24.    //略
25. }
```

它内部使用的 HashService： 算法为 DEFAULT_HASH_ALGORITHM

即"SHA-256", hash 次数为 DEFAULT_HASH_ITERATIONS 即 500000, 是否产生 publicSalt 为 true, 即一定会产生 publicSalt (我们知道最终要参与计算的 salt 是 publicSalt 和 privateSalt 的合并, 这里默认并没有设置 DefaultHashService 的 privateSalt)。

它内部使用的 HashFormat: 为 Shiro1CryptFormat, 它的 format 方法能将一个 Hash 格式化成一个字符串, 它的 parse 方法能将一个上述格式化的字符串解析成一个 Hash。


它内部使用的 HashFormatFactory: 为经过 HashFormat 格式化的字符串找到对应的 HashFormat。

看完了 DefaultPasswordService 的基本配置, 然后就来看下对明文密码的加密过程:

Java 代码 

```
1. public String encryptPassword(Object plaintext) {
2.     Hash hash = hashPassword(plaintext);
3.     checkHashFormatDurability();
4.     return this.hashFormat.format(hash);
5. }
```

第一个过程, 即将明文密码通过 HashService 的算法等配置加密成一个 Hash

Java 代码 

```
1. public Hash hashPassword(Object plaintext) {
2.     ByteSource plaintextBytes = createByteSource(plaintext);
3.     if (plaintextBytes == null || plaintextBytes.isEmpty()) {
4.         return null;
5.     }
6.     HashRequest request = createHashRequest(plaintextBytes);
```

```
7.         return hashService.computeHash(request);
8.     }
9.     protected HashRequest createHashRequest(ByteSource plaintext) {
10.         return new HashRequest.Builder().setSource(plaintext).build();
11.     }
```

先创建一个 `HashRequest`，最终为 `new SimpleHashRequest(this.algorithmName, this.source, this.salt, this.iterations)`; 这个 `Request` 只有明文密码不为空，其他都为空，`iterations` 为 0。通过 `hashService.computeHash(request)` 过程来生成一个 `new SimpleHash(algorithmName, source, salt, iterations)`;

`algorithmName`: 来自 `hashService` 的算法名即 `SHA-256`

`source`: 即来自明文密码

`salt`: 是 `hashService` 的 `privateSalt`（从上文知道为空）和 `publicSalt` 的合并。由于 `hashService` 的 `generatePublicSalt` 属性为 `true` (从上文知道)，所以会生成 `publicSalt`，是如下方式随机生成的

Java 代码 

```
1. publicSalt = getRandomNumberGenerator().nextBytes();
```

`hash` 次数: 上述 `SimpleHashRequest` 的 `hash` 次数为 0，所以采用的是 `hashService` 的 `hash` 次数即 500000

综上所述，`hashService` 产生了一个 `new SimpleHashRequest("SHA-256", this.source, this.salt, 500000)` 的一个 `hash`。

接下来就是用 `hashFormat` 来格式化这个 `Hash`，过程如下:

Java 代码 ☆

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

MCF_PREFIX 为: \$shiro1\$, TOKEN_DELIMITER 为: \$

格式化的字符串为: \$shiro1\$算法名字\$hash 次数\$salt 的 base64 编码\$hash 的 base64 编码

所以得到的加密密文的结果为:

\$shiro1\$SHA-
256\$500000\$y37r7l1qqEoIVXprJBfvbA==\$065oJsZajS1yhGBUYyX
V1ThQ8brGw/9Koa6sDMmKAw=

所以这个密文每一部分都代表着一定的内容，从而可以实现 parse，得到采用的算法、hash 次数、salt 信息。所以在用户登陆的时候，就可以将用户的明文密码仍按照此配置进行一次加密来匹配，即可断定用户的密码是否正确，这其实就是密码匹配过程所采用的方式。

上述案例是使用 PasswordMatcher 默认配置，现在如果我们想更改算

法、salt、和 hash 次数来满足我们的需求。

PasswordMatcher 是依靠 PasswordService，默认的

PasswordService 是 DefaultPasswordService，


DefaultPasswordService 又是靠 HashService（默认是

DefaultHashService）的算法、salt、hash 次数等配置来加密的，所

以我们要更改算法、salt、hash 次数则要 DefaultHashService 进行设


置，如我们想用 md5 算法来加密、publicSalt 为随机生成，hash 次数

为 3 次，则 ini 配置文件要如下更改：

Java 代码 


```
1. [main]
2. #realm
3. dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
4. dataSource.driverClass=com.mysql.jdbc.Driver
5. dataSource.jdbcUrl=jdbc:mysql://localhost:3306/shiro
6. dataSource.user=root
7. dataSource.password=ligang
8. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
9. jdbcRealm.dataSource=$dataSource
10. jdbcRealm.permissionsLookupEnabled=true
11.
12. hashService=org.apache.shiro.crypto.hash.DefaultHashService
13. hashService.hashAlgorithmName=MD5
14. hashService.hashIterations=3
15. hashService.generatePublicSalt=true
16.
17. passwordService=org.apache.shiro.authc.credential.DefaultPasswordService
18. passwordService.hashService=$hashService
19.
20. credentialsMatcher=org.apache.shiro.authc.credential.PasswordMatcher
21. credentialsMatcher.passwordService=$passwordService
22.
23. jdbcRealm.credentialsMatcher=$credentialsMatcher
24. securityManager.realms=$jdbcRealm
```


PasswordMatcher 是依靠 PasswordService 来实现加密和匹配的，所以我们可以自定义一个 PasswordService 来按照我们自己约定的加密规则来实现加密，如下所示：

Java 代码 

```
1. public class MyPasswordService implements PasswordService{
2.
3.     private String algorithmName="MD5";
4.     private int iterations=5;
5.     private String salt="2014";
6.
7.     private HashFormat hashFormat=new Shiro1CryptFormat();
8.
9.     @Override
10.    public String encryptPassword(Object plaintextPassword)
11.        throws IllegalArgumentException {
12.        Hash hash=new SimpleHash(algorithmName,ByteSource.Util.bytes(plaintextPassword),ByteSource.Util.bytes(salt), iterations);
13.        return hashFormat.format(hash);
14.    }
15.
16.    @Override
17.    public boolean passwordsMatch(Object submittedPlaintext, String encrypted) {
18.        Hash hash=new SimpleHash(algorithmName,ByteSource.Util.bytes(submittedPlaintext),ByteSource.Util.bytes(salt), iterations);
19.        String password=hashFormat.format(hash);
20.        return encrypted.equals(password);
21.    }
22. }
```

加密过程和匹配过程都采用相同的步骤来实现匹配。以上便是一个简单的 PasswordService 实现，ini 配置更改为：

Java 代码 

```
1. [main]
2. #realm
3. dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
4. dataSource.driverClass=com.mysql.jdbc.Driver
```

```
5. dataSource.jdbcUrl=jdbc:mysql://localhost:3306/shiro
6. dataSource.user=root
7. dataSource.password=ligang
8. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
9. jdbcRealm.dataSource=$dataSource
10. jdbcRealm.permissionsLookupEnabled=true
11.
12. passwordService=com.lg.shiro.MyPasswordService
13.
14. credentialsMatcher=org.apache.shiro.authc.credential.PasswordMatcher
15. credentialsMatcher.passwordService=$passwordService
16.
17. jdbcRealm.credentialsMatcher=$credentialsMatcher
18. securityManager.realms=$jdbcRealm
```

作者： 乒乓狂魔