

二

33 Kafka认证机制用哪家?

你好，我是胡夕。今天我要和你分享的主题是：Kafka 的认证机制。

什么是认证机制?

所谓认证，又称“验证”“鉴权”，英文是 authentication，是指通过一定的手段，完成对用户身份的确认。认证的主要目的是确认当前声称为某种身份的用户确实是所声称的用户。

在计算机领域，经常和认证搞混的一个术语就是授权，英文是 authorization。授权一般是指对信息安全或计算机安全相关的资源定义与授予相应的访问权限。

举个简单的例子来区分下两者：认证要解决的是你要证明你是谁的问题，授权要解决的则是你能做什么的问题。

在 Kafka 中，认证和授权是两套独立的安全配置。我们今天主要讨论 Kafka 的认证机制，在专栏的下一讲内容中，我们将讨论授权机制。

Kafka 认证机制

自 0.9.0.0 版本开始，Kafka 正式引入了认证机制，用于实现基础的安全用户认证，这是将 Kafka 上云或进行多租户管理的必要步骤。截止到目前最新的 2.3 版本，Kafka 支持基于 SSL 和基于 SASL 的安全认证机制。

基于 SSL 的认证主要是指 Broker 和客户端的双路认证（2-way authentication）。通常来说，SSL 加密（Encryption）已经启用了单向认证，即客户端认证 Broker 的证书（Certificate）。如果要做 SSL 认证，那么我们要启用双路认证，也就是说 Broker 也要认证客户端的证书。

对了，你可能会说，SSL 不是已经过时了吗？现在都叫 TLS（Transport Layer Security）了吧？但是，Kafka 的源码中依然是使用 SSL 而不是 TLS 来表示这类东西的。不过，今天出现的所有 SSL 字眼，你都可以认为它们是和 TLS 等价的。

Kafka 还支持通过 SASL 做客户端认证。**SASL 是提供认证和数据安全服务的框架。**Kafka 支持的 SASL 机制有 5 种，它们分别是在不同版本中被引入的，你需要根据你自己使用的 Kafka 版本，来选择该版本所支持的认证机制。

1. GSSAPI：也就是 Kerberos 使用的安全接口，是在 0.9 版本中被引入的。
2. PLAIN：是使用简单的用户名 / 密码认证的机制，在 0.10 版本中被引入。
3. SCRAM：主要用于解决 PLAIN 机制安全问题的新机制，是在 0.10.2 版本中被引入的。
4. OAUTHBEARER：是基于 OAuth 2 认证框架的新机制，在 2.0 版本中被引进。
5. Delegation Token：补充现有 SASL 机制的轻量级认证机制，是在 1.1.0 版本被引入的。

认证机制的比较

Kafka 为我们提供了这么多种认证机制，在实际使用过程中，我们应该如何选择合适的认证框架呢？下面我们就来比较一下。

目前来看，使用 SSL 做信道加密的情况更多一些，但使用 SSL 实现认证不如使用 SASL。毕竟，SASL 能够支持你选择不同的实现机制，如 GSSAPI、SCRAM、PLAIN 等。因此，我的建议是**你可以使用 SSL 来做通信加密，使用 SASL 来做 Kafka 的认证实现。**

SASL 下又细分了很多种认证机制，我们应该如何选择呢？

SASL/GSSAPI 主要是给 Kerberos 使用的。如果你的公司已经做了 Kerberos 认证（比如使用 Active Directory），那么使用 GSSAPI 是最方便的了。因为你不需要额外地搭建 Kerberos，只要让你们的 Kerberos 管理员给每个 Broker 和要访问 Kafka 集群的操作系统用户申请 principal 就好了。总之，**GSSAPI 适用于本身已经做了 Kerberos 认证的场景，这样的话，SASL/GSSAPI 可以实现无缝集成。**

而 SASL/PLAIN，就像前面说到的，它是一个简单的用户名 / 密码认证机制，通常与 SSL 加密搭配使用。注意，这里的 PLAIN 和 PLAINTEXT 是两回事。**PLAIN 在这里是一种认证机制，而 PLAINTEXT 说的是未使用 SSL 时的明文传输。**对于一些小公司而言，搭建公司级的 Kerberos 可能并没有什么必要，他们的用户系统也不复杂，特别是访问 Kafka 集群的用户可能不是很多。对于 SASL/PLAIN 而言，这就是一个非常合适的应用场景。**总体来说，SASL/PLAIN 的配置和运维成本相对较小，适合于小型公司中的 Kafka 集群。**

但是，SASL/PLAIN 有这样一个弊端：它不能动态地增减认证用户，你必须重启 Kafka 集群才能令变更生效。为什么呢？这是因为所有认证用户信息全部保存在静态文件中，所以只

能重启 Broker，才能重新加载变更后的静态文件。

我们知道，重启集群在很多场景下都是令人不爽的，即使是轮替式升级（Rolling Upgrade）。SASL/SCRAM 就解决了这样的问题。它通过将认证用户信息保存在 ZooKeeper 的方式，避免了动态修改需要重启 Broker 的弊端。在实际使用过程中，你可以使用 Kafka 提供的命令动态地创建和删除用户，无需重启整个集群。因此，**如果你打算使用 SASL/PLAIN，不妨改用 SASL/SCRAM 试试。不过要注意的是，后者是 0.10.2 版本引入的。你至少要升级到这个版本后才能使用。**

SASL/OAUTHBEARER 是 2.0 版本引入的新认证机制，主要是为了实现与 OAuth 2 框架的集成。OAuth 是一个开发标准，允许用户授权第三方应用访问该用户在某网站上的资源，而无需将用户名和密码提供给第三方应用。Kafka 不提倡单纯使用 OAUTHBEARER，因为它生成的不安全的 JSON Web Token，必须配以 SSL 加密才能用在生产环境中。当然，鉴于它是 2.0 版本才推出来的，而且目前没有太多的实际使用案例，我们可以先观望一段时间，再酌情将其应用于生产环境中。

Delegation Token 是在 1.1 版本引入的，它是一种轻量级的认证机制，主要目的是补充现有的 SASL 或 SSL 认证。如果要使用 Delegation Token，你需要先配置好 SASL 认证，然后再利用 Kafka 提供的 API 去获取对应的 Delegation Token。这样，Broker 和客户端在做认证的时候，可以直接使用这个 token，不用每次都去 KDC 获取对应的 ticket（Kerberos 认证）或传输 Keystore 文件（SSL 认证）。

为了方便你更好地理解 and 记忆，我把这些认证机制汇总在下面的表格里了。你可以对照着表格，进行一下区分。

认证机制	引入版本	推荐理由
SSL	0.9	适用于一般测试场景。
SASL / GSSAPI	0.9	适用于本身已经实现的Kerberos认证的场景。
SASL / PLAIN	0.10.2	适用于中小型公司的Kafka集群。
SASL / SCRAM	0.10.2	适用于中小型公司的Kafka集群，支持认证用户的动态增减。
SASL / OAUTHBEARER	2.0	适用于支持OAuth 2.0框架的场景。
Delegation Token	1.1	适用于Kerberos认证中出现TGT分发性能瓶颈的场景。

SASL/SCRAM-SHA-256 配置实例

接下来，我给出 SASL/SCRAM 的一个配置实例，来说明一下如何在 Kafka 集群中开启认证。其他认证机制的设置方法也是类似的，比如它们都涉及认证用户的创建、Broker 端以及 Client 端特定参数的配置等。

我的测试环境是本地 Mac 上的两个 Broker 组成的 Kafka 集群，连接端口分别是 9092 和 9093。

第 1 步：创建用户

配置 SASL/SCRAM 的第一步，是创建能否连接 Kafka 集群的用户。在本次测试中，我会创建 3 个用户，分别是 admin 用户、writer 用户和 reader 用户。admin 用户用于实现 Broker 间通信，writer 用户用于生产消息，reader 用户用于消费消息。

我们使用下面这 3 条命令，分别来创建它们。

```
$ cd kafka_2.12-2.3.0/
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-2
Completed Updating config for entity: user-principal 'admin'.
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-2
Completed Updating config for entity: user-principal 'writer'.
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-2
Completed Updating config for entity: user-principal 'reader'.
```

在专栏前面，我们提到过，kafka-configs 脚本是用来设置主题级别参数的。其实，它的功能还有很多。比如在这个例子中，我们使用它来创建 SASL/SCRAM 认证中的用户信息。我们可以使用下列命令来查看刚才创建的用户数据。

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --describe --entity-type users -
Configs for user-principal 'writer' are SCRAM-SHA-512=salt=Mwt60GplZHF6YnF5bmEyam9j
```

这段命令包含了 writer 用户加密算法 SCRAM-SHA-256 以及 SCRAM-SHA-512 对应的盐值 (Salt)、ServerKey 和 StoreKey。这些都是 SCRAM 机制的术语，我们不需要了解它们的含义，因为它们并不影响我们接下来的配置。

第 2 步：创建 JAAS 文件

配置了用户之后，我们需要为每个 Broker 创建一个对应的 JAAS 文件。因为本例中的两个 Broker 实例是在一台机器上，所以我只创建了一份 JAAS 文件。但是你要切记，在实际场景中，你需要为每台单独的物理 Broker 机器都创建一份 JAAS 文件。

JAAS 的文件内容如下：

```
KafkaServer {  
  org.apache.kafka.common.security.scram.ScramLoginModule required  
  username="admin"  
  password="admin";  
};
```

关于这个文件内容，你需要注意以下两点：

- 不要忘记最后一行和倒数第二行结尾处的分号；
- JAAS 文件中不需要任何空格键。

这里，我们使用 admin 用户实现 Broker 之间的通信。接下来，我们来配置 Broker 的 server.properties 文件，下面这些内容，是需要单独配置的：

```
sasl.enabled.mechanisms=SCRAM-SHA-256  
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256  
security.inter.broker.protocol=SASL_PLAINTEXT  
listeners=SASL_PLAINTEXT://localhost:9092
```

第 1 项内容表明开启 SCRAM 认证机制，并启用 SHA-256 算法；第 2 项的意思是为 Broker 间通信也开启 SCRAM 认证，同样使用 SHA-256 算法；第 3 项表示 Broker 间通信不配置 SSL，本例中我们不演示 SSL 的配置；最后 1 项是设置 listeners 使用 SASL_PLAINTEXT，依然是不使用 SSL。

另一台 Broker 的配置基本和它类似，只是要使用不同的端口，在这个例子中，端口是 9093。

第 3 步：启动 Broker

现在我们分别启动这两个 Broker。在启动时，你需要指定 JAAS 文件的位置，如下所示：

```
$KAFKA_OPTS=-Djava.security.auth.login.config=<your_path>/kafka-broker.jaas bin/kaf  
.....  
[2019-07-02 13:30:34,822] INFO Kafka commitId: fc1aaa116b661c8a (org.apache.kafka.c  
[2019-07-02 13:30:34,822] INFO Kafka startTimeMs: 1562045434820 (org.apache.kafka.c  
[2019-07-02 13:30:34,823] INFO [KafkaServer id=0] started (kafka.server.KafkaServer  
$KAFKA_OPTS=-Djava.security.auth.login.config=<your_path>/kafka-broker.jaas bin/kaf
```

```
.....  
[2019-07-02 13:32:31,976] INFO Kafka commitId: fc1aaa116b661c8a (org.apache.kafka.c  
[2019-07-02 13:32:31,976] INFO Kafka startTimeMs: 1562045551973 (org.apache.kafka.c  
[2019-07-02 13:32:31,978] INFO [KafkaServer id=1] started (kafka.server.KafkaServer
```

此时，两台 Broker 都已经成功启动了。

第 4 步：发送消息

在创建好测试主题之后，我们使用 kafka-console-producer 脚本来尝试发送消息。由于启用了认证，客户端需要做一些相应的配置。我们创建一个名为 producer.conf 的配置文件，内容如下：

```
security.protocol=SASL_PLAINTEXT  
sasl.mechanism=SCRAM-SHA-256  
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required u
```

之后运行 Console Producer 程序：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092,localhost:9093 --topic  
>hello, world  
>
```

可以看到，Console Producer 程序发送消息成功。

第 5 步：消费消息

接下来，我们使用 Console Consumer 程序来消费一下刚刚生产的消息。同样地，我们需要为 kafka-console-consumer 脚本创建一个名为 consumer.conf 的脚本，内容如下：

```
security.protocol=SASL_PLAINTEXT  
sasl.mechanism=SCRAM-SHA-256  
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required u
```

之后运行 Console Consumer 程序：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092,localhost:9093 --  
hello, world
```

很显然，我们是可以正常消费的。

第 6 步：动态增减用户

最后，我们来演示 SASL/SCRAM 动态增减用户的场景。假设我删除了 writer 用户，同时又添加了一个新用户：new_writer，那么，我们需要执行的命令如下：

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config 'SCRAM-SH
Completed Updating config for entity: user-principal 'writer'.
```

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config 'SCRAM-SH
Completed Updating config for entity: user-principal 'writer'.
```

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-2
Completed Updating config for entity: user-principal 'new_writer'.
```

现在，我们依然使用刚才的 producer.conf 来验证，以确认 Console Producer 程序不能发送消息。

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092,localhost:9093 --topic
>[2019-07-02 13:54:29,695] ERROR [Producer clientId=console-producer] Connection to
.....
```

很显然，此时 Console Producer 已经不能发送消息了。因为它使用的 producer.conf 文件指定的是已经被删除的 writer 用户。如果我们修改 producer.conf 的内容，改为指定新创建的 new_writer 用户，结果如下：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092,localhost:9093 --topic
>Good!
```

现在，Console Producer 可以正常发送消息了。

这个过程完整地展示了 SASL/SCRAM 是如何在不重启 Broker 的情况下增减用户的。

至此，SASL/SCRAM 配置就完成了。在专栏下一讲中，我会详细介绍一下如何赋予 writer 和 reader 用户不同的权限。

小结

好了，我们来小结一下。今天，我们讨论了 Kafka 目前提供的几种认证机制，我给出了它们各自的优劣势以及推荐使用建议。其实，在真实的使用场景中，认证和授权往往是结合在一起使用的。在专栏下一讲中，我会详细向你介绍 Kafka 的授权机制，即 ACL 机制，敬请期待。

Kafka的认证机制

- 所谓认证，是指通过一定的手段，完成对用户身份的确认。认证的主要目的是确认当前声称为某种身份的用户确实是所声称的用户。
- 截止到当前最新的2.3版本，Kafka支持基于SSL和基于SASL的安全认证机制，建议你使用SSL来做通信加密，使用SASL来做Kafka的认证实现。
- Kafka支持的SASL机制有5种，分别是GSSAPI、PLAIN、SCRAM、OAUTHBEARER和Delegation Token。它们是在不同的版本中被引入的，你需要根据自己使用的Kafka版本，来选择该版本所支持的认证机制。



极客时间

[上一页](#)

[下一页](#)