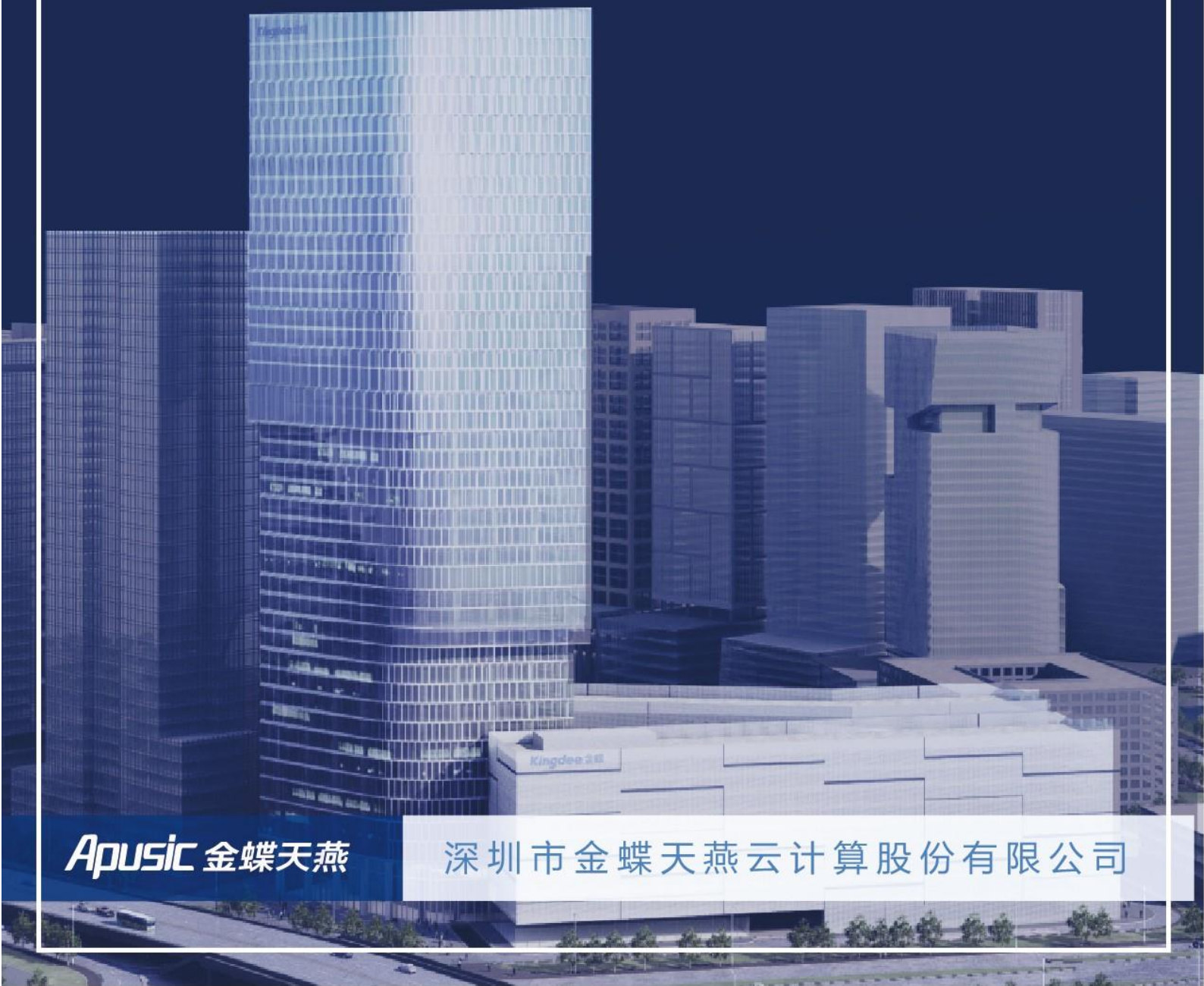
金蝶 Apusic 分布式消息队列 V2.0

用　户　手　册



Apusic 金蝶天燕

深圳市金蝶天燕云计算股份有限公司

版权声明

目录

部分I 快速开始指南

第1章 关于本快速开始指南

本快速入门指南介绍了金蝶 Apusic 分布式消息队列 v2.0 产品安装、启动、卸载、管理与使用等基本操作，为用户快速使用本产品提供指导。

1.1 基本介绍

随着业务系统复杂度的增加，业务拆分粒度越来越小，在一个大的系统中通常包含数十个、数百个小系统，各个子业务系统之间需要一种可靠的、稳定的和高效的通信方式。因此像 Kafka、RocketMQ、RabbitMQ 这些专注于消息通信的系统应运而生，提供了系统解耦、削峰填谷、异构集成、异步隔离和分布式存储等功能，在数据传输方面很大程度上提高了系统的稳定性和可靠性。

金蝶天燕在 2003 年左右推出了基于 JMS 规范的 Apusic 消息中间件，在推出之后不断升级和优化，已经成长成一个相对完善和高性能的系统。之后，为了更好的服务金融系统，金蝶天燕在 2020 年启动了以开源项目 Pulsar 为内核的云原生消息中间件，在高可靠、低延迟、高并发异地容灾和安全等方面进行了深入研究，能有效的应用于支付、电商等对系统稳定性和可靠性有严格要求的系统中。

金蝶 Apusic 分布式消息队列（简称 ADMQ）提供了一个云原生消息和事件流平台，使您能够为实时和历史事件构建一个实时应用和数据基础设施。金蝶 Apusic 分布式消息队列为您的公司轻松构建关键任务消息和流应用以及实时数据管道，让您专注于如何从实时数据中实现业务价值的最大化。

1.2 相关资源

针对不同的操作系统，金蝶 Apusic 分布式消息队列提供不同的安装包，也提供适合各个操作系统解压即用的 zip 格式的产品包。

更多信息，可以访问金蝶天燕官方网站 http://www.apusic.com 获取。

1.3 基本概念

在正确使用应用服务器来部署、管理应用之前，需要先理解以下几个基本概念：

1. ADMQ 集群

   一个 ADMQ 实例由一个或多个 ADMQ 集群组成。集群又由以下部分组成。

- 一个或者多个 brokers
- 一个 ZooKeeper 协调器，用于集群级别的配置和协调
- 一组 BookKeeper 的 Bookies 用于消息的持久化存储
- 一个管理控制台，用于管理用户和相关资源的创建和分配，也是 broker 和 bookeeper 启动的依赖。

2. Broker

broker 是一个无状态组件，主要负责运行另外的两个组件:

- 一个 HTTP 服务器，它暴露了 REST 系统管理接口以及在生产者和消费者之间进行 Topic 查找的 API。
- 一个调度分发器，它是异步的 TCP 服务器，通过自定义二进制协议应用于所有相关的数据传输。

为了支持全局 Topic 异地复制，Broker 会控制 Replicators 追踪本地发布的条目，并把这些条目用 Java 客户端重新发布到其他区域。

3. 元数据存储

ADMQ 使用 Zookeeper 进行元数据存储、集群配置和协调。在一个 ADMQ 实例中:

- 配置与仲裁存储: 存储租户，命名域和其他需要全局一致的配置项
- 每个集群有自己独立的 ZooKeeper 保存集群内部配置和协调信息，例如归属信息，broker 负载报告等等。

4. 持久化存储

ADMQ 为应用程序提供有保障的消息传递。如果一个消息成功地到达 ADMQ Broker，它将被送达其预定的目标。为了提供这种保证，未确认送达的消息需要持久化存储直到它们被确认送达。ADMQ 用 BookKeeper 作为持久化存储。 BookKeeper 是一个分布式的预写日志（WAL）系统。

5.管理控制台

管理控制台是一个基于 Web 的 GUI 管理和监控工具，用于管理用户、权限、租户、命名空间、主题、订阅、broker、集群，并支持多个环境的动态配置。

第2章 安装与使用

2.1 安装前准备

2.1.1 获取安装包

从 http://www.apusic.com/下载金蝶 Apusic 分布式消息队列 V2.0 安装包，或从金蝶

Apusic 分布式消息队列 V2.0 产品光盘中获得相应的安装包文件。

2.1.2 操作系统

平台类型	系统类型
Linux	国产操作系统:如银河麒麟系列、中标麒麟系列、普华、中科红旗、深度等
	RedHat 系列
	CentOS
	Suse Linux 系列
Unix	HP Unix 系列
	IBM AIX 系列
	Solaris 系列

2.1.3 支持的 Java 虚拟机

金蝶 Apusic 分布式消息队列所支持的 Java 虚拟机:

- Oracle JDK 8+
- Open JDK 8+

2.1.4 系统要求

系统组件	系统要求
Java 环境	JDK1.8 及以上版本
内存	2GB+
硬盘空间	10GB+
浏览器	IE8 及以上，FireFox,Chrome

2.1.5 设置环境变量

下载好压缩文件后，解压缩并使用 cd 命令进入文件所在位置

$ tar xvfz admq-V2.0-bin.tar.gz

$ cd admq-v2

设置 ADMQ 目录的环境变量

$ export ADMQ_HOME=<path-to-admq>

在路径中添加 ADMQ 的 bin 目录

$ export PATH=${ADMQ_HOME}/bin:$PATH

2.2 开始安装

安装金蝶 Apusic 分布式消息队列之前，需确保 Java 运行环境已安装并配置好了。

系统包含包含两部分，第一是管理控制台，提供可视化界面管理和配置系统信息；第二个是提供消息读写服务的后台程序，包含 broker、bookie 和 zookeeper 三个组件。

2.2.1 管理控制台安装

金蝶 Apusic 分布式消息队列管理控制台提供用于配置，管理和监视的界面化统一管理配置平台，支持本地及远程访问管理控制台。admq-manager 的数据存储使用 mysql，所以在此之前您必须先部署好 mysql，并使用${yourpath}/admq-manager/share/pulsar-manager/sql/mysql-schema.sql 脚本文件初始化数据库。

2.2.1.1 添加 license

将从金蝶天燕获取的 license 文件拷贝到 admq-manager 项目中，路径为 ${yourpath}/admq-manager/share/pulsar-manager/license-files,否则 admq-manager 将无法启动。

2.2.1.2 生成权限认证需要的 token

生成密钥

bin/admq tokens create-secret-key --output ${outpath}/my-secret.key

创建超管角色

bin/ admq tokens create --secret-key file://${outpath}/my-secret.key --subject admin

保存生成的 token 信息。

2.2.1.3 参数配置

admq-manager 只需要部署一个节点就行，下载 ADMQ 压缩包，解压重命名为 admq-manager，重命名非必须，只是为了更好区分几个组件，然后进到 config 目录，${yourpath}/admq-manager/share/pulsar-manager/config。修改 config 文件夹下的 application-prod.properties 配置文件,具体配置如下：

配置项：

spring.datasource.url=${yourMysqlUrl}

spring.datasource.username=${yourMysqlUserName}

spring.datasource.password=${yuorMysqlPassword}

#第一个 broker 集群

broker.jwtInfo[0].authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

broker.jwtInfo[0].authParams=token:${之前生成好的超管 token}

broker.jwtInfo[0].secretKey=file:///path/to/my-secret.key

broker.jwtInfo[0].url=每台 broker 节点的 url 地址，注意每个 url 都要带 http://

#第二个 broker 集群，如果有的话

broker.jwtInfo[1].authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

broker.jwtInfo[1].authParams=token:${之前生成好的超管 token}

broker.jwtInfo[1].secretKey=file:///path/to/my-secret.key

broker.jwtInfo[1].url=每台 broker 节点的 url 地址，注意每个 url 都要带 http://

参数说明：

spring.datasource.url：mysql 的 url

spring.datasource.username：mysql 的用户名

spring.datasource.password：mysql 的用户密码

broker.jwtInfo[i].authPlugin:鉴权插件

broker.jwtInfo[i].authParams:超管 token

broker.jwtInfo[i].secretKey:Secret key 文件地址

broker.jwtInfo[i].url:每台 broker 节点的 url 地址，注意每个 url 都要带 http://

示例：

spring.datasource.url=jdbc:mysql://localhost:3306/admq_manager?useUnicode=true&characterEncoding=utf-8&allowMultiQueries=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

broker.jwtInfo[0].authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

broker.jwtInfo[0].authParams=token:eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiJ9.NFdJbfENLdBohcsYffju9qhd474FvR_4yxJokz_pq-k

broker.jwtInfo[0].secretKey=file:///home/software/admq-manager/share/pulsar-manager/my-secret.key

broker.jwtInfo[0].url=http://172.18.100.185:8080,http://172.18.100.186:8080,http:

//172.18.100.187:8080

2.2.1.3 启动

进入 admq-manager 的根目录，执行以下命令：

bin/manager-start.sh

2.2.2 后台服务安装（单机模式）

2.2.2.1 参数配置和解释

进入到 admq-V2.0.0/share/pulsar/conf 目录，修改 standalone.conf 文件，主要包含如下修改项。

配置项

manageLicenseUrl=http://x.x.x.x:7750

clusterMode=standalone

authenticationEnabled=true

authorizationEnabled=true

authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationProviderToken

brokerClientAuthenticationPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

brokerClientAuthenticationParameters=token:${之前生成好的超管 token}

superUserRoles=admin

tokenSecretKey=file://${outpath}/my-secret.key

参数说明

manageLicenseUrl：manager 地址

clusterMode：单机模式

authenticationEnabled：是否开启身份认证

authorizationEnabled：是否开启权限认证

authenticationProviders：认证插件

brokerClientAuthenticationPlugin：认证插件

brokerClientAuthenticationParameters：超管 token

superUserRoles：超管标识

tokenSecretKey：Secret key 路径

2.2.2.2 启动服务

bin/standalone-start.sh

2.2.2.3 停止服务

bin/standalone-stop.sh

2.2.3 后台服务安装（集群模式）

一个 ADMQ 集群实例主要由以下三部分组成：

- 多个 Broker 组成的集群，负责将生产者生产的消息发送给消费者。
- 多个 Bookkeeper 组成的集群，负责消息的持久化存储。
- 多个 Zookeeper 组成的本地集群，用来存储 Broker 和 Bookkeeper 相关元数据。

需要依次安装并启动 zookeeper、bookkeeper 和 broker。

2.2.3.1 安装前准备

- JDK8 以上
- 最少 6 台服务器，3 台性能要求不高的机器部署 Zookeeper 集群和 admq-manager，3 台性能强劲的机器部署 Bookkeeper 和 Broker 集群。

2.2.3.2 本地 zookeeper 集群部署

在 3 个节点分别下载 ADMQ 压缩包，解压重命名为 admq-local-zk，重命名非必须，只是为了更好区分几个组件，然后进到 conf 目录，${yourpath}/admq-local-zk/share/pulsar/conf。

配置

修改 conf 文件夹下的 zookeeper.conf 配置文件，3 个节点都需要修改，具体配置如下：

dataDir=${yourDataDir}

dataLogDir=${dataLogDir}

server.1=node1 ip:port1:port2

server.2=node2 ip:port1:port2

...

server.n=node n ip:port1:port2

参数说明：

dataDir：当前 zookeeper 节点的数据存放目录，可自定义

dataLogDir：当前 zookeeper 节点的日志存放目录，可自定义

server.1~n：为 zookeeper 集群的各节点指定编号,两个端口可自定义,一般是 2888 和 3888

新建相关文件夹

在每个 zookeeper 节点的机器上，新建 dataDir 和 dataLogDir，跟第二步配的路径一致即可，${dataDir}和${dataLogDir}替换成具体路径。

创建 myid

为每个 Zookeeper 节点新建 myid，myid 必须放在${dataDir}路径下，然后写入第二步配置文件中指定的节点编号，echo serverid > ${dataDir}/myid，如：

节点 1: echo 1 >/home/software/admq-local-zk/share/pulsar/data/myid

节点 2: echo 2 >/home/software/admq-local-zk/share/pulsar/data/myid

节点 3: echo 3 >/home/software/admq-local-zk/share/pulsar/data/myid

开放端口

某些机器如 centos7.3 以上，即使关闭防火墙，也无法访问未对外开放的端口，如果遇到端口无法访问，将 3 个节点在第二步中指定的端口开放对外访问，如 2181,2888,3888 端口，具体操作如下：

先查看防火墙是否开启：systemctl status firewalld

如果没开启，要先开启：systemctl start firewalld

查看端口是否对外开放：firewall-cmd --query-port=端口号/tcp

如果没开放，就执行： firewall-cmd --add-port=端口号/tcp --permanent

重新加载防火墙的端口：firewall-cmd –reload

启动

分别启动 3 个节点的 zookeeper，进入 admq-local-zk 的根目录，执行相关命令。

前台方式启动：bin/admq zookeeper

前台方式关闭：ctrl+c

后台方式启动：bin/admq-daemon start zookeeper

后台方式关闭：bin/admq-daemon stop zookeeper

2.2.3.3 初始化元数据

```
/bin/admq initialize-cluster-metadata \
  --cluster ${clusterName} \
  --zookeeper local zookeeper server ip :port \
  --configuration-store global zookeeper server ip:port \
  --web-service-url http://broker1 ip:port \
```

```
--web-service-url-tls https://broker1 ip:port \
--broker-service-url pulsar://broker1 ip:port \
--broker-service-url-tls pulsar+ssl://broker1 ip:port \
--manager-url http://admq-manager ip:port
```

参数说明：

cluster：集群名称。

zookeeper：本地 zooKeeper 集群，仅需要包含 zooKeeper 集群中的一个节点即可。

configuration-store：admq 实例的配置全局存储 zookeeper 集群，多集群部署时才会发挥作用，只需要包含 zooKeeper 集群中的一个节点即可，单个 admq 集群时只需填写本地 zk 集群中的一个节点即可。

web-service-url：broker 集群 Web 服务的 URL+端口，URL 是一个标准的 DNS 名称，默认端口 8080，不建议修改。

web-service-url-tls：broker 集群 Web 提供 TLS 服务的 URL+端口，端口默认 8443，不建议修改。

broker-service-url：集群 brokers 服务 URL，URL 中 DNS 的名称和 Web 服务保持一致，URL 使用 pulsar 替代 http，端口默认 6650，不建议修改。

broker-service-url-tls：集群 brokers 提供 TLS 服务的 URL，默认端口 6551，不建议修改。

manager-url:上一步部署的 admq-manager 地址,默认端口是 7750，具体看部署时有无修改端口。

示例

```
bin/admq initialize-cluster-metadata \
--cluster admq-cluster \
--zookeeper 172.18.100.182:2181 \
--configuration-store 172.18.100.182:2181 \
--web-service-url http://172.18.100.185:8080 \
--web-service-url-tls https://172.18.100.185:8443 \
--broker-service-url pulsar://172.18.100.185:6650 \
--broker-service-url-tls pulsar+ssl://172.18.100.185:6651 \
--manager-url http://172.18.100.182:7750
```

2.2.3.4 全局 zookeeper 部署

多个 ADMQ 集群才需要安装全局 Zookeeper，单个集群可跳过此步骤。安装全局

Zookeeper 集群，如果机器充足，可安装在另外 3 台机器上，也可安装在其中一个本地 zookeeper 集群的机器上，安装步骤和本地 Zookeeper 基本一致，如果和本地 zookeeper 安装在同一批机器上，需要改步骤二中 zookeeper.conf 的 clientPort，server.1~server.n 的端口号，避免端口冲突，其余安装步骤一样。

2.2.3.5 bookkeeper 集群部署

下载 ADMQ 压缩包，解压重命名为 admq-Bookkeeper，然后进到 conf 目录，${yourPath}/admq- bookkeeper/share/pulsar/conf，修改 conf 文件夹下的 bookkeeper.conf 配置文件，3 个节点都需要修改。具体配置如下：

配置项

advertisedAddress=本机 ip

zkServers=local zk server1 ip:port,local zk server2 ip:port,local zk server3 ip:port

journalDirectory=${yourJournalDirectories}

ledgerDirectories=${yourLedgerDirectories}

prometheusStatsHttpPort=普罗米修斯端口

httpServerEnabled=true

参数说明

advertisedAddress：指定当前节点的主机名或 IP 地址。

zkServers：指定本地 Zookeeper 集群，用来将 bookkeeper 节点的元数据存放在 Zookeeper 集群。

journalDirectory：当前 bookkeeper 节点的 journal 数据存放目录。如果需要提高磁盘写入性能，可以指定多个目录用来存放 journal 数据，关键是每一个目录必须在不同的磁盘，不然反而会影响写入性能。

ledgerDirectories：当前 bookkeeper 节点的 ledger 存放目录，如果情况允许，跟 journalDirectories 放在不同的磁盘,可以提高性能。

prometheusStatsHttpPort:普罗米修斯端口号，默认 8000，跟 bookkeeper 自己的 httpServerPort 端口冲突，所以修改。

httpServerEnabled：开启 web 服务

节点 1 示例

advertisedAddress=172.18.100.185

zkServers=172.18.100.182:2181,172.18.100.184:2181,172.18.100.190:2181

journalDirectory=/home/software/admq-bookkeeper/share/pulsar/data/journal

ledgerDirectories=/data/admq-bookkeeper/share/pulsar/data/ledgers

prometheusStatsHttpPort=8100

新建相关文件夹

在各个节点创建相关的 journal 和 ledgers 目录。

mkdir -pv ${journalDirectory}

mkdir -pv ${ledgerDirectories}

初始化元数据

在任意节点执行元数据初始化，遇到提示按"y"，只需执行一次。

bin/admq-bookkeeper shell metaformat

开放端口

对外放开放 3181，8000 端口,这是 bookkeeper 默认运行的 tcp 和 http 端口。

firewall-cmd --add-port=3181/tcp --permanent

firewall-cmd --add-port=8000/tcp --permanent

firewall-cmd –reload

启动验证

在各个节点启动 bookkeeper

前台方式启动：bin/admq bookie

后台方式启动：bin/admq-daemon start bookie

后台方式关闭：bin/admq-daemon stop bookie

验证是否成功启动，出现 Bookie sanity test succeeded 代表成功。

bin/admq-bookkeeper shell bookiesanity

验证集群内的 bookkeeper 是否都能正常工作

bin/admq-bookkeeper shell simpletest --ensemble <num-bookies> --writeQuorum <num-bookies> --ackQuorum <num-bookies> --numEntries <num-entries>

例如：

bin/admq-bookkeeper shell simpletest --ensemble 3 --writeQuorum 3 --ackQuorum 3

--numEntries 3

2.2.3.6 broker 集群部署

下载 ADMQ 压缩包，解压重命名为 admq-broker,修改 conf 文件夹下的 broker.conf 配置文件，具体配置如下：

配置项

zookeeperServers=ip:port,ip2:port,ip3:port

configurationStoreServers=ip:port,ip2:port,ip3:port

advertisedAddress=本节点 ip

clusterName=集群名

authenticationEnabled=true

authorizationEnabled=true

authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationProviderToken

brokerClientAuthenticationPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

brokerClientAuthenticationParameters=token:${之前生成好的超管 token}

superUserRoles=admin

tokenSecretKey=file:///path/to/my-secret.key

参数说明

clusterMode：集群模式

zookeeperServers:本地 zk 集群，存储 broker 节点元数据。

configurationStoreServers：全局 zk 集群，若无搭建，则跟上面 zookeeperServers 填一样。

advertisedAddress：当前节点的主机名或者 ip。

clusterName：集群名称，必须与初始化 zk 元数据时的集群名一样。

authenticationEnabled：是否开启身份认证

authorizationEnabled：是否开启权限认证

authenticationProviders：认证插件

brokerClientAuthenticationPlugin：认证插件

brokerClientAuthenticationParameters：超管 token

superUserRoles：超管标示

tokenSecretKey：Secret key 路径

最终的配置文件需要修改的地方示例如下:

zookeeperServers=172.18.100.182:2181,172.18.100.184:2181,172.18.100.190:2181

configurationStoreServers=172.18.100.182:2184,172.18.100.184:2184,172.18.100.190:2184

advertisedAddress=172.18.100.185

clusterName=admq-cluster

authenticationEnabled=true

authorizationEnabled=true

authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationProviderToken

brokerClientAuthenticationPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

brokerClientAuthenticationParameters=token:eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiJ9. NFdJbfENLdBohcsYffju9qhd474FvR_4yxJokz_pq-k

superUserRoles=admin

tokenSecretKey=file:///home/software/admq-broker/share/pulsar/my-secret.key

还要修改 client.conf 配置文件，目的是让集群间的 broker 在开启鉴权之后也能相互访问

authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken

authParams=token:${之前生成好的超管 token}

开放 6650 端口

firewall-cmd --add-port=6650/tcp --permanent

firewall-cmd --reload

启动

回到 admq-broker 的根目录，启动 broker

bin/admq-daemon start broker

查看集群节点情况

${clusterName}替换成自己的集群名字，如果列表包含所有节点，则代表 broker 部署成功。

bin/admqctl brokers list ${clusterName}

2.3 使用管理控制台创建资源

使用管理控制台创建名为 example-tenant 的租户，在租户 example-tenant 中创建一个名 为 example-ns 的命名空间，在命名空间 example-ns 中创建一个有 1 个分区的主题 test-topic。具体创建教程请参考本文档“部分 II 管理控制台使用”。

第3章 Java 客户端的使用

你可以使用 Java 客户端来创建消息的 Java 生产者、消费者和读者，并执行管理任务。Java 客户端的生产者、消费者和 reader 中的所有方法都是线程安全的。客户端的 Javadoc 按包分为以下两个类型。

Package	描述
org.apache.pulsar.client.api	生产者和消费者 API
org.apache.pulsar.client.admin	管理 API

3.1 客户端安装

最新版本的 Java 客户端库可以通过 Maven 中心获得。要使用最新版本，请在构建配置中添加 pulsar-client 库。在 pom.xml 文件中添加以下信息。

```
<pulsar.version>2.8.0</pulsar.version>
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-client</artifactId>
  <version>${pulsar.version}</version>
</dependency>
```

3.2 连接 URL

要使用客户端库连接到服务器，你需要指定一个 Pulsar 协议 URL。

你可以将 Pulsar 协议 URL 分配给特定的集群，并使用 pulsar 方案。默认的端口是 6650。下面是一个 localhost 的例子。

```
pulsar://localhost:6650
```

如果你有多个 broker，URL 如下：

```
pulsar://localhost:6550,localhost:6651,localhost:6652
```

生产环境集群的 URL 如下：

pulsar://pulsar.cn-west.example.com:6650

如果你使用 TLS 认证，URL 如下：

pulsar+ssl://pulsar.cn-west.example.com:6651

3.3 获取权限验证的 token

超管的 token 从本文档“管理控制台安装-2.2.1.2 章节”中生成获取，普通用户通过管理控制台界面获取，具体操作参考下面章节：管理控制台的使用 – 管理权限。

3.4 创建客户端

你可以像这样只用一个集群的 URL 来实例化一个 PulsarClient 对象：

```
PulsarClient client = PulsarClient.builder()
        .serviceUrl("pulsar://localhost:6650")
        .authentication(AuthenticationFactory.token("eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJKb2UifQ.ipevRNuRP6HflG8cFKnmUPtypruRC4fb1DWtoLL62SY")) // 通过管理控制台获取到的 token
        .build();
```

如果你有多个 broker，你可以像这样启动一个 PulsarClient：

```
PulsarClient client = PulsarClient.builder()
        .serviceUrl("pulsar://localhost:6650,localhost:6651,localhost:6652")
        .authentication(AuthenticationFactory.token("eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJKb2UifQ.ipevRNuRP6HflG8cFKnmUPtypruRC4fb1DWtoLL62SY"))
        .build();
```

如果你创建一个客户端，可以使用 loadConf 配置。在 loadConf 中可以使用以下参数。

类型	名称	描述	缺省值
String	serviceUrl	服务的 URL 提供者	
String	authPluginClassName	认证插件的名称	
String	authParams	String 代表认证插件的参数例子 key1:val1,key2	

		:val2	
long	operationTimeoutMs	操作超时时间	30000
Long	statsIntervalSeconds	每条统计信息之间的间隔统计信息以正数 statsInterval 激活将 statsIntervalSeconds 设置为至少 1 秒	60
Int	numIoThreads	用于处理与 broker 连接的线程数量	1
int	numListenerThreads	用于处理消息监听器的线程数	1
boolean	useTcpNoDelay	是否在连接上使用 TCP no-delay 标志以禁用 Nagle 算法	True
Boolean	useTls	是否在连接上使用 TLS 加密	False
String	tlsTrustCertsFilePath	信任的 TLS 证书文件的路径	
Boolean	tlsAllowInsecureConnection	客户端是否接受来自 broker 的不受信任的 TLS 证书	False
Boolean	tlsHostnameVerificationEnable	是否启用 TLS 主机名验证	False

Int	concurrentLookupRequest	允许在每个 broker 连接上发送的并发查询请求的数量，以防止 broker 的过载	5000
int	maxLookupRequest	每个 broker 连接允许的最大查询请求数，以防止 broker 的过载	50000
int	maxNumberOfRejectedRequestPerConnection	在当前连接关闭后，客户端创建一个新的连接以连接到不同的 broker 后，在一定的时间范围内（30 秒），一个 broker 的最大拒绝请求数	50
int	keepAliveIntervalSeconds	每个客户端连接的存活间隔秒数	30
Int	connectionTimeoutMs	等待与 broker 建立连接的持续时间 如果时间过了，没有来自 broker 的响应，连接尝试	10000

		将被放弃	
int	requestTimeoutMs	完成一项请求的最长时间	60000
int	defaultBackoffIntervalNanos	默认的 backoff 间隔时间	TimeUnit.MILLISECONDS.toNanos(100);
long	maxBackoffIntervalNanos	一个 backoff 间隔的最大持续时间	TimeUnit.SECONDS.toNanos(30)

3.5 生产者

生产者向主题写入消息。一旦你实例化了一个 PulsarClient 对象，你就可以为一个特定的主题创建一个生产者。

```
Producer<byte[]> producer = client.newProducer()
        .topic("my-topic")
        .create();
producer.send("My message".getBytes());
```

默认情况下，生产者产生的消息是由字节数组组成的。你可以通过指定一个消息 schema 来产生不同的类型。

```
Producer<String> stringProducer = client.newProducer(Schema.STRING)
        .topic("my-topic")
        .create();
stringProducer.send("My message");
```

确保你在不需要时关闭你的生产者、消费者和客户端对象：

```
producer.close();
consumer.close();
client.close();
```

关闭操作也可以是异步的：

```
producer.closeAsync()
    .thenRun(() -> System.out.println("Producer closed"))
    .exceptionally((ex) -> {
        System.err.println("Failed to close producer: " + ex);
        return null;
```

});

3.5.1 配置生产者

如果你像上面的例子那样只指定一个主题名称来实例化一个生产者对象，则使用生产者的默认配置。

如果你创建一个生产者，可以使用 loadConf 配置。在 loadConf 中可以使用以下参数。

类型	名称	描述	缺省值
String	topicName	主题名称	
String	producerName	生产者名称	
Long	sendTimeoutMs	消息发送超时，单位是 ms 如果在 sendTimeout 过期前，消息没有被服务器确认，就会发生一个错误。	30000
Boolean	blockIfQueueFull	如果它被设置为 true，当传出的消息队列已满时，生产者的 Send 和 SendAsync 方法会被阻塞，而不是失败和抛出错误。 如果它被设置为 false，当传出的消息队列已满时，生产者的 Send 和 SendAsync 方法会失败，并出现 ProducerQueueIsFullError 异常。 MaxPendingMessages 参数决定了外	False

		发消息队列的大小。	
Int	maxPendingMessages	持有未决消息的队列的最大大小。 例如，一个等待从 broker 收到确认的消息。 默认情况下，当队列已满时，所有对 Send 和 SendAsync 方法的调用都会失败，除非你把 BlockIfQueueFull 设置为 true。	1000
Int	maxPendingMessagesAcrossPartitions	各个分区的最大待处理消息数。 如果总数超过了配置的值，使用设置来降低每个分区的最大待处理信息	50000
MessageRoutingMode	messageRoutingMode	分区主题上的生产者的消息路由逻辑。只在对消息设置无键时应用该逻辑。 可用的选项如下： - pulsar.RoundRobinDistribution：轮询 - pulsar.UseSinglePartition：将所有消息发布到一个分区上 -	RoundRobinDistribution

		pulsar.CustomPartition：一个自定义的分区方案	
HashingScheme	hashingScheme	确定你发布特定信息的分区的哈希函数(仅适用于分区的主题）。 可用的选项如下： - pulsar.JavaStringHash：相当于 Java 中的 String.hashCode() - pulsar.Murmur3_32Hash：应用 Murmur3 的散列函数。 - pulsar.BoostHash：应用 C++的 Boost 库中的散列函数。	HashingScheme.JavaStringHash
ProducerCryptoFailureAction	cryptoFailureAction	加密失败时，生产者应采取行动。 - FAIL：如果加密失败，未加密的信息发送失败。 - SEND：如果加密失败，未加密的信息被发送。	ProducerCryptoFailureAction.FAIL
long	batchingMaxPublishDelayMicros	分批发送信息的时间段	TimeUnit.MILLISECONDS.toMicros(1)

int	batchingMaxMessages	一个批次中允许的最大信息数量	1000
Boolean	batchingEnabled	启用信息批处理	True
CompressionType	compressionType	生产者使用的消息数据压缩类型。 可用的选项: LZ4 ZLIB ZSTD SNAPPY	

如果你不想使用默认配置，你可以配置参数。下面是一个例子:

```
Producer<byte[]> producer = client.newProducer()
        .topic("my-topic")
        .batchingMaxPublishDelay(10, TimeUnit.MILLISECONDS)
        .sendTimeout(10, TimeUnit.SECONDS)
        .blockIfQueueFull(true)
        .create();
```

3.5.2 消息路由

当使用分区主题时，你可以在使用生产者发布消息时指定路由模式。缺省的路由策略为轮询，下面是一个例子:

```
String pulsarBrokerRootUrl = "pulsar://localhost:6650";
String topic = "persistent://my-tenant/my-namespace/my-topic";

PulsarClient pulsarClient = PulsarClient.builder().serviceUrl(pulsarBrokerRootUrl).build();
Producer<byte[]> producer = pulsarClient.newProducer()
            .topic(topic)
            .messageRoutingMode(MessageRoutingMode. RoundRobinPartition)
            .create();
producer.send("Partitioned topic message".getBytes());
```

3.5.3 异步发送

你可以使用 Java 客户端异步地发布消息。通过异步发送，生产者将消息放在一个阻塞队列中并立即返回。然后客户端库在后台将消息发送到代理。如果队列已满（最大尺寸可配置），

生产者在调用 API 时就会被阻塞或立即失败，这取决于传递给生产者的参数。

下面是一个例子：

```
producer.sendAsync("my-async-message".getBytes()).thenAccept(msgId -> {
    System.out.printf("Message with ID %s successfully sent", msgId);
});
```

正如你在上面的例子中看到的，异步发送操作返回一个包裹在 CompletableFuture 中的 MessageId。

3.5.4 配置消息

除了值之外，你还可以在一个给定的信息上设置其他项目：

```
producer.newMessage()
    .key("my-message-key")
    .value("my-async-message".getBytes())
    .property("my-key", "my-value")
    .property("my-other-key", "my-other-value")
    .send();
```

你可以用 sendAsync()终止构建器链，并获得一个 future。

3.6 消费者

在 ADMQ 中，消费者订阅主题并处理生产者向这些主题发布的消息。你可以通过首先实例化一个 PulsarClient 对象并将 broker 的 URL 传递给它来实例化一个新的消费者。

一旦你实例化了一个 PulsarClient 对象，你就可以通过指定主题和订阅来创建一个消费者。

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscribe();
```

subscribe 方法自动将消费者订阅到指定的主题和订阅。让消费者监听主题的一种方法是设置一个 while 循环。在这个例子的循环中，消费者监听消息，打印任何收到的消息的内容，然后确认消息已被处理。如果处理逻辑失败了，你可以使用否定的确认来在以后重新传递消息。

```
while (true) {
  // Wait for a message
  Message msg = consumer.receive();
```

```
    try {
        // Do something with the message
        System.out.printf("Message received: %s", new String(msg.getData()));

        // Acknowledge the message so that it can be deleted by the message broker
        consumer.acknowledge(msg);
    } catch (Exception e) {
        // Message failed to process, redeliver later
        consumer.negativeAcknowledge(msg);
    }
}
```

3.6.1 配置消费者

如果你像上面的例子那样只指定主题和订阅名称来实例化一个消费者对象，那么消费者会使用默认配置。

当你创建一个消费者时，你可以使用 loadConf 配置。在 loadConf 中可以使用以下参数。

类型	名称	描述	缺省值
Set<String>	topicNames	主题名称	Sets.newTreeSet( )
Pattern	topicsPattern	主题模式	
String	subscriptionName	订阅名称	
Subscription Type	subscriptionType	订阅类型 有四种订阅类型可供选择： Exclusive Failover Shared Key_Shared	SubscriptionType .Exclusive
int	receiverQueueSize	一个消费者的接收队列的大小。例如，在应用程序调用 Receive 之前，消费者积累的消息数量。 一个高于默认值的值会增加消费者的吞吐量，尽管代价是更多的内存利用率。	1000

long	acknowledgementsGroupTimeMicros	对消费者的确认进行分组，并指定时间。 默认情况下，消费者使用 100ms 的分组时间来向经纪人发送确认。 设置一个 0 的分组时间可以立即发送确认信息。 较长的确认分组时间是更有效的，代价是失败后消息的重新传送会略有增加。	TimeUnit.MILLISECONDS.toMicros(100)
Long	negativeAckRedeliveryDelayMicros	在重新发送处理失败的消息之前所要等待的时间。 当应用程序使用 Consumer#negativeAcknowledge(Message)时，失败的消息会在一个固定的超时后重新交付。	TimeUnit.MINUTES.toMicros(1)
int	maxTotalReceiverQueueSizeAcrossPartitions	各个分区的最大总接收队列大小。 如果总的接收队列大小超过这个值，这个设置会减少单个分区的接收队列大小。	50000
String	consumerName	消费者名称	
long	ackTimeoutMillis	未确认信息的超时	0
Long	tickDurationMillis	ack-timeout 重新交付的粒度。 使用较高的 tickDurationMillis 可以减少在设置 ack-timeout 为较大值（例如 1 小时）时跟踪消息的内存开销。	1000
int	priorityLevel	消费者的优先级，broker 在共享订阅模式下调度消息时给予其更多的优先权。 broker 遵循递减的优先级。例如，	0

		0=最大优先级，1，2，...。 在共享订阅模式下，如果最大优先级的消费者有许可证，broker 首先将消息投递给他们。否则，broker 会考虑下一个优先级的消费者。	
ConsumerCryptoFailureAction	cryptoFailureAction	当消费者收到无法解密的消息时，应该采取行动。 - FAIL:这是默认的选项，在加密成功之前，消息都是失败的。 - DISCARD:默默地确认，不向应用程序传递消息。 - CONSUME：将加密的消息传递给应用程序。解密消息是应用程序的责任。 消息的解压缩失败。 如果消息包含批处理的消息，客户端就不能在批处理中检索单个消息。 交付的加密消息包含 EncryptionContext，其中包含加密和压缩信息，应用程序可以用它来解密消耗的消息有效载荷。	ConsumerCryptoFailureAction.FAIL
SortedMap<String, String>	properties	这个消费者的一个名称或属性。 属性是附加到消费者身上的应用程序定义的元数据。 当获得一个主题的统计信息时，将这个元数据与消费者的统计信息联系起来，以便于识别。	new TreeMap<>()
Boolean	readCompacted	如果启用 readCompacted，消费者会从一个压缩的主题中读取消息，而不是读取一个主题的全部	False

		消息积压。 消费者只看到压缩后的主题中每个键的最新值，直到到达主题消息中压缩积压的点。超过这一点，就像平常一样发送消息。 只在持久性话题的订阅上启用 readCompacted，这些话题有一个活跃的消费者（比如失败或排他性订阅）。 试图在非持久性主题的订阅或共享订阅上启用它，会导致订阅调用抛出一个 PulsarClientException。	
SubscriptionInitialPosition	subscriptionInitialPosition	第一次订阅一个主题时，设置订阅的初始位置。	SubscriptionInitialPosition.Latest
int	patternAutoDiscoveryPeriod	当使用主题的消费者模式时，主题的自动发现周期。 默认值和最小值是 1 分钟。	1
RegexSubscriptionMode	regexSubscriptionMode	当使用正则表达式订阅一个主题时，你可以选择某种类型的主题。 - PersistentOnly：只订阅持久性话题。 - NonPersistentOnly：只订阅非持久性话题。 - AllTopics：同时订阅持久性和非持久性主题。	RegexSubscriptionMode.PersistentOnly
DeadLetterPolicy	deadLetterPolicy	消费者的死信策略。 在默认情况下，一些消息可能会被多次重发，甚至到了从未停止的程度。 通过使用死信机制，消息有最大重传次数。当超过最大重传次数	

		时，消息被发送到死信主题并自动确认。 你可以通过设置 deadLetterPolicy 来启用死信机制 例如： client.newConsumer() .deadLetterPolicy(DeadLetterPolicy.builder().maxRedeliverCount(10).build()) .subscribe(); 缺省的死信队列的名称为 {TopicName}-{Subscription}-DLQ 可以设置一个自定义的死信主题名称 client.newConsumer() .deadLetterPolicy(DeadLetterPolicy.builder().maxRedeliverCount(10) .deadLetterTopic("your-topic-name").build()) .subscribe(); 当指定死信策略而不指定 ackTimeoutMillis 时，你可以将 ack 超时设置为 30000 毫秒。	
boolean	autoUpdatePartitions	如果启用了 autoUpdatePartitions，消费者会自动订阅分区增量。 注意：这只适用于分区的消费者。	True
boolean	replicateSubscriptionState	如果 replicateSubscriptionState 被启用，一个订阅状态会被复制	false

		到地理复制的集群中。	

如果你不想使用默认的配置，你可以配置参数。下面是一个例子。

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .ackTimeout(10, TimeUnit.SECONDS)
        .subscriptionType(SubscriptionType.Exclusive)
        .subscribe();
```

3.6.2 异步消息接收

接收方法同步地接收消息（消费者进程被阻塞，直到有消息可用）。你也可以使用异步接收，一旦有新的消息可用，它就立即返回一个 CompletableFuture 对象。

```
CompletableFuture<Message> asyncMessage = consumer.receiveAsync();
```

异步接收操作返回一个包裹在 CompletableFuture 中的消息。

3.6.3 批量消息接收

使用 batchReceive 来接收每次调用的多个信息。下面是一个例子。

```
Messages messages = consumer.batchReceive();
for (Object message : messages) {
  // do something
}
consumer.acknowledge(messages)
```

批量接收策略限制了单个批次中信息的数量和字节数。你可以指定一个超时来等待足够的消息。如果满足以下任何一个条件，批处理接收就完成了：足够的消息数量，消息的字节数，等待超时。

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .batchReceivePolicy(BatchReceivePolicy.builder()
                .maxNumMessages(100)
                .maxNumBytes(1024 * 1024)
                .timeout(200, TimeUnit.MILLISECONDS)
                .build())
        .subscribe();
```

默认的批量接收策略是：

```
BatchReceivePolicy.builder()
        .maxNumMessage(-1)
        .maxNumBytes(10 * 1024 * 1024)
        .timeout(100, TimeUnit.MILLISECONDS)
        .build();
```

3.6.4 多主题订阅

除了将消费者订阅到单一的主题外，你还可以使用多主题订阅同时订阅多个主题。要使用多主题订阅，你可以提供一个正则表达式（regex）或一个主题列表。如果你通过正则表达式选择主题，所有的主题必须是在同一个命名空间内。

```
import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.PulsarClient;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

ConsumerBuilder consumerBuilder = pulsarClient.newConsumer()
            .subscriptionName(subscription);

// Subscribe to all topics in a namespace
Pattern allTopicsInNamespace = Pattern.compile("public/default/.*");
Consumer allTopicsConsumer = consumerBuilder
            .topicsPattern(allTopicsInNamespace)
            .subscribe();

// Subscribe to a subsets of topics in a namespace, based on regex
Pattern someTopicsInNamespace = Pattern.compile("public/default/foo.*");
Consumer allTopicsConsumer = consumerBuilder
            .topicsPattern(someTopicsInNamespace)
            .subscribe();
```

在上面的例子中，消费者订阅的是符合主题名称模式的持久性主题。如果你想让消费者订阅

所有符合主题名称模式的持久性和非持久性主题，请将 subscriptionTopicsMode 设置为 RegexSubscriptionMode.AllTopics。

```
Pattern pattern = Pattern.compile("public/default/.*");
pulsarClient.newConsumer()
        .subscriptionName("my-sub")
        .topicsPattern(pattern)
        .subscriptionTopicsMode(RegexSubscriptionMode.AllTopics)
        .subscribe();
```

默认情况下，消费者的 subscriptionTopicsMode 是 PersistentOnly。subscriptionTopicsMode 的可用选项是 PersistentOnly、NonPersistentOnly 和 AllTopics。

你也可以订阅一个明确的主题列表（如果你愿意，可以跨命名空间）。

```
List<String> topics = Arrays.asList(
        "topic-1",
        "topic-2",
        "topic-3"
);

Consumer multiTopicConsumer = consumerBuilder
        .topics(topics)
        .subscribe();
// Alternatively:
Consumer multiTopicConsumer = consumerBuilder
        .topic(
            "topic-1",
            "topic-2",
            "topic-3"
        )
        .subscribe();
```

你也可以使用 subscribeAsync 方法异步订阅多个主题，而不是使用同步订阅方法。下面是一个例子。

```
Pattern allTopicsInNamespace = Pattern.compile("persistent://public/default.*");
consumerBuilder
        .topics(topics)
```

```
        .subscribeAsync()
        .thenAccept(this::receiveMessageFromConsumer);

private void receiveMessageFromConsumer(Object consumer) {
    ((Consumer)consumer).receiveAsync().thenAccept(message -> {
                // Do something with the received message
                receiveMessageFromConsumer(consumer);
            });
}
```

3.6.5 订阅模式

ADMQ 有各种订阅模式，以配合不同的场景。一个主题可以有多个具有不同订阅模式的订阅。然而，一个订阅一次只能有一个订阅模式。

一个订阅与订阅名称相同，一次只能指定一个订阅模式。你不能改变订阅模式，除非这个订阅的所有现有消费者都是离线的。

不同的订阅模式有不同的消息分发模式。本节描述了订阅模式的区别以及如何使用它们。

为了更好地描述它们的区别，假设你有一个名为 "my-topic "的主题，生产者已经发布了 10 条消息。

```
Producer<String> producer = client.newProducer(Schema.STRING)
            .topic("my-topic")
            .enableBatching(false)
            .create();
  // 3 messages with "key-1", 3 messages with "key-2", 2 messages with "key-3" and 2 messages with "key-4"
  producer.newMessage().key("key-1").value("message-1-1").send();
  producer.newMessage().key("key-1").value("message-1-2").send();
  producer.newMessage().key("key-1").value("message-1-3").send();
  producer.newMessage().key("key-2").value("message-2-1").send();
  producer.newMessage().key("key-2").value("message-2-2").send();
  producer.newMessage().key("key-2").value("message-2-3").send();
  producer.newMessage().key("key-3").value("message-3-1").send();
  producer.newMessage().key("key-3").value("message-3-2").send();
  producer.newMessage().key("key-4").value("message-4-1").send();
```

producer.newMessage().key("key-4").value("message-4-2").send();

3.6.5.1 Exclusive

创建一个新的消费者，用 Exclusive 订阅模式订阅。

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
            .subscriptionName("my-subscription")
            .subscriptionType(SubscriptionType.Exclusive)
            .subscribe()
```

只有第一个消费者被允许参与订阅，其他消费者会收到一个错误。第一个消费者收到所有的 10 条消息，消费的顺序与生产的顺序相同。

如果主题是一个分区的主题，第一个消费者会订阅所有分区的主题，其他消费者没有被分配到分区，并收到一个错误。

3.6.5.2 Failover

创建新的消费者，用 Failover 订阅模式订阅。

```
Consumer consumer1 = client.newConsumer()
            .topic("my-topic")
            .subscriptionName("my-subscription")
            .subscriptionType(SubscriptionType.Failover)
            .subscribe()
Consumer consumer2 = client.newConsumer()
            .topic("my-topic")
            .subscriptionName("my-subscription")
            .subscriptionType(SubscriptionType.Failover)
            .subscribe()
```

多个消费者可以附加到同一个订阅上，但只有第一个消费者是活动的，其他的是备用的。当活跃的消费者断开连接时，消息将被派发到一个备用的消费者身上，然后这个备用的消费者就成为活跃的消费者。

3.6.5.3 Shared

创建新的消费者，用共享订阅模式订阅。

```
Consumer consumer1 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
```

```
        .subscribe()

Consumer consumer2 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
        .subscribe()
```

在共享订阅模式下，多个消费者可以附加到同一个订阅上，信息在消费者之间以轮流分配的方式传递。

3.6.5.4 Key_shared

创建新的消费者并以 Key_Shared 订阅模式订阅。

```
Consumer consumer1 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Key_Shared)
        .subscribe()

Consumer consumer2 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Key_Shared)
        .subscribe()
```

Key_Shared 订阅和 Shared 订阅一样，所有消费者都可以附加到同一个订阅。但它与 Key_Shared 订阅不同，具有相同 key 的消息只依次传递给一个消费者。默认情况下，我们不会事先知道哪些 key 将被分配给一个消费者，但一个 key 只会在同一时间被分配给一个消费者。

consumer1 收到以下信息

("key-1", "message-1-1")

("key-1", "message-1-2")

("key-1", "message-1-3")

("key-3", "message-3-1")

("key-3", "message-3-2")

Consumer2 收到以下信息

("key-2", "message-2-1")

("key-2", "message-2-2")

("key-2", "message-2-3")

("key-4", "message-4-1")

("key-4", "message-4-2")

如果在生产者端启用了批处理，不同 key 的消息默认会被添加到一个批次中。broker 将把批处理派发给消费者，所以默认的批处理机制可能会破坏 Key_Shared 订阅保证的消息分发语义。生产者需要使用 KeyBasedBatcher。

```
Producer producer = client.newProducer()
        .topic("my-topic")
        .batcherBuilder(BatcherBuilder.KEY_BASED)
        .create();
```

或者生产者可以禁用批处理。

```
Producer producer = client.newProducer()
        .topic("my-topic")
        .enableBatching(false)
        .create();
```

3.7 Reader 接口

通过 Reader 接口，客户端可以在一个主题中 "手动定位"，从指定消息开始处理所有消息。Java 的 API 使你能够通过指定一个主题和一个 MessageId 来创建 Reader 对象，下面是一个例子。

```
byte[] msgIdBytes = // Some message ID byte array
MessageId id = MessageId.fromByteArray(msgIdBytes);
Reader reader = pulsarClient.newReader()
        .topic(topic)
        .startMessageId(id)
        .create();

while (true) {
    Message message = reader.readNext();
    // Process message
}
```

在上面的例子中，为一个特定的主题和消息（通过 ID）实例化了一个 Reader 对象；在消息被 msgIdBytes 识别后，Reader 对主题中的每条消息进行迭代。上面的示例代码展示了将 Reader 对象指向一个特定的消息（通过 ID），但你也可以使用 MessageId.earliest 指向主题上最早的可用消息，MessageId.latest 指向最近的可用消息。

当你创建一个 Reader 时，你可以使用 loadConf 配置。在 loadConf 中可以使用以下参数。

类型	名称	描述	缺省值
String	topicName	主题名称	
Int	receiverQueueSize	一个消费者的接收队列的大小。例如，在应用程序调用 Receive 之前，消费者可以积累的消息数量。一个高于默认值的值会增加消费者的吞吐量，尽管以更多的内存利用率为代价。	1000
ReaderListener<T>	readerListener	一个监听器，在收到消息时被调用。	
String	readerName	Reader 名称	
String	subscriptionRolePrefix	订阅角色前缀	
CryptoKeyReader	CryptoKeyReader	抽象出对密钥存储的访问的接口。	
ConsumerCryptoFailureAction	cryptoFailureAction	当消费者收到无法解密的消息时，应该采取行动。 - FAIL：这是默认选项，使消息失败，直到加密成	ConsumerCryptoFailureAction.FAIL

		功。 - DISCARD：默默地确认，不向应用程序传递消息。 - CONSUME：将加密的消息传递给应用程序。解密消息是应用程序的责任。 消息解压失败。 如果消息包含批处理的消息，客户端就不能在批处理中检索单个消息。 交付的加密消息包含{@link EncryptionContext}，其中包含加密和压缩信息，应用程序可以用它来解密消耗的消息有效载荷。	
boolean	readCompacted	如果启用readCompacted，消费者会从一个压缩的主题中读取消息，而不是一个主题的全部消息积压。 消费者只看到压	False

		缩后的主题中每个键的最新值，直到到达主题消息中压缩积压的点。超过这个点，就像正常一样发送消息。 readCompacted 只能在持久性话题的订阅上启用，这些订阅有一个活跃的消费者（例如，失败或排他性订阅）。 试图在非持久性主题的订阅或共享订阅上启用它，会导致订阅调用抛出一个 PulsarClientException。	
boolean	resetIncludeHead	如果设置为"true"，将返回的第一条信息是由 messageId 指定的那条。 如果设置为 false，要返回的第一条信息是 messageId 指定	false

		的信息旁边的那条。	

3.7.1 Sticky key 范围 Reader

在 Sticky key 范围 Reader 中，broker 将只发送消息 key 的哈希值包含在指定的 key 哈希范围内的消息。一个 Reader 上可以指定多个 key 哈希值范围。下面是一个创建 sticky key 范围 Reader 的例子。

```
pulsarClient.newReader()
        .topic(topic)
        .startMessageId(MessageId.earliest)
        .keyHashRange(Range.of(0, 10000), Range.of(20001, 30000))
        .create();
```

散列范围总大小为 65536，所以范围的最大端应该小于或等于 65535。

3.8 Schema

在 ADMQ 中，所有的消息数据都是由字节数组组成的。消息 Schema 使你在构建和处理消息时能够使用其他类型的数据（从简单的类型如字符串到更复杂的特定应用类型）。如果你构建了一个生产者，如果没有指定 schema，那么这个生产者只能产生 byte[]类型的消息。下面是一个例子。

```
Producer<byte[]> producer = client.newProducer()
        .topic(topic)
        .create();
```

如果你想对不同类型的数据使用生产者，你需要指定一个 schema，告知 ADMQ 哪种数据类型将在主题中传输。

3.8.1 Schema 示例

假设你有一个 SensorReading 类，你想通过一个主题来传输：

```
public class SensorReading {
    public float temperature;

    public SensorReading(float temperature) {
        this.temperature = temperature;
    }
```

```
    // A no-arg constructor is required
    public SensorReading() {
    }

    public float getTemperature() {
        return temperature;
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
    }
}
```

然后你可以创建一个 Producer<SensorReading>（或 Consumer<SensorReading>）：

```
Producer<SensorReading> producer = client.newProducer(JSONSchema.of(SensorReading.class))
        .topic("sensor-readings")
        .create();
```

以下是目前可用于 Java 的 schema：

- 无 schema 或字节数组 schema

```
Producer<byte[]> bytesProducer = client.newProducer(Schema.BYTES)
        .topic("some-raw-bytes-topic")
        .create();
```

或者等效于：

```
Producer<byte[]> bytesProducer = client.newProducer()
        .topic("some-raw-bytes-topic")
        .create();
```

- 对于 UTF-8 编码的字符串数据，使用 Schema.STRING：

```
Producer<String> stringProducer = client.newProducer(Schema.STRING)
        .topic("some-string-topic")
        .create();
```

- 使用 Schema.JSON 为 POJO 创建 JSON schema：

```
Producer<MyPojo> pojoProducer = client.newProducer(Schema.JSON(MyPojo.class))
        .topic("some-pojo-topic")
```

```
        .create();
```

- 使用 Schema.PROTOBUF 生成 Protobuf schema。下面的例子显示了如何创建 Protobuf schema 并使用它来实例化一个新的生产者:

```
Producer<MyProtobuf> protobufProducer = client.newProducer(Schema.PROTOBUF(MyProtobuf.class))
        .topic("some-protobuf-topic")
        .create();
```

- 用 Schema.AVRO 定义 Avro schema。下面的代码片断演示了如何创建和使用 Avro schema:

```
Producer<MyAvro> avroProducer = client.newProducer(Schema.AVRO(MyAvro.class))
        .topic("some-avro-topic")
        .create();
```

第4章 删除金蝶 Apusic 分布式消息队列

在删除金蝶 Apusic 分布式消息队列之前，请停止以下过程:

- 所有域名和其他相关流程
- 命令提示使用安装目录或其子目录

使用属于 Java 平台标准版（Java SE）的文件的任何应用程序

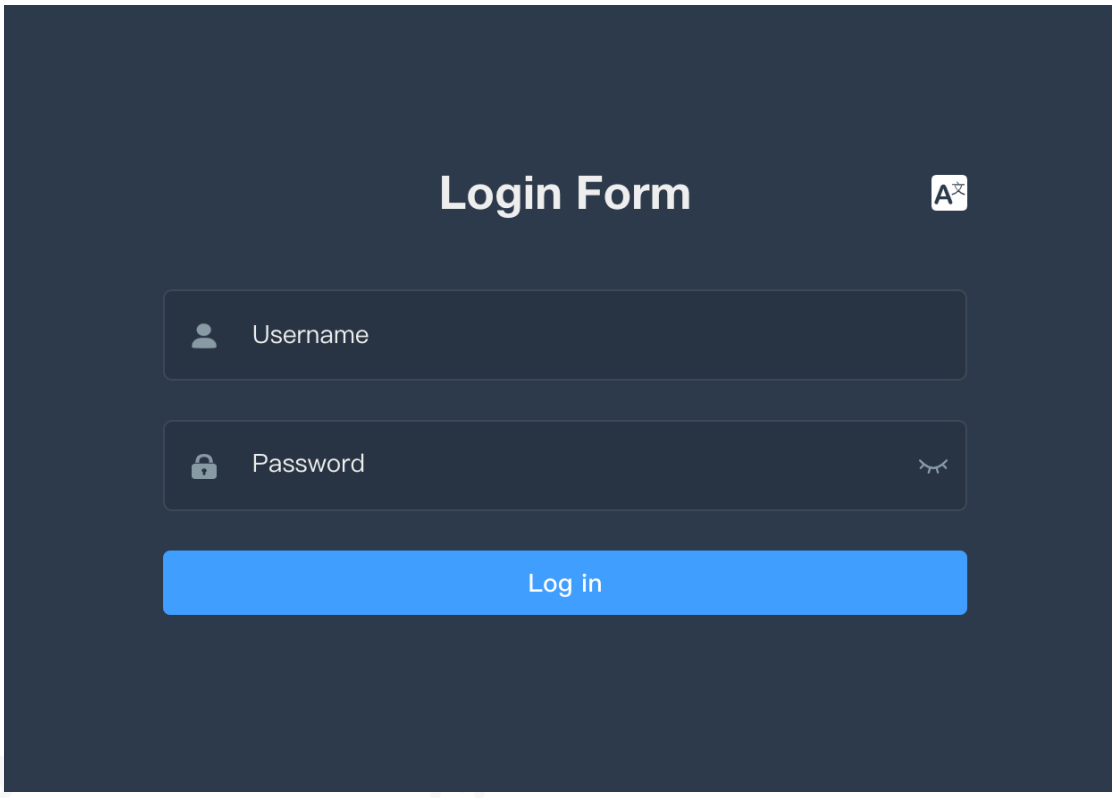
删除金蝶 Apusic 分布式消息队列安装包解压安装目录。

部分II管理控制台的使用

管理控制台是一个基于 Web 的 GUI 管理和监控工具，用于管理租户、命名空间、主题、订阅、broker、集群，并支持多个环境的动态配置。

第1章 登录管理控制台

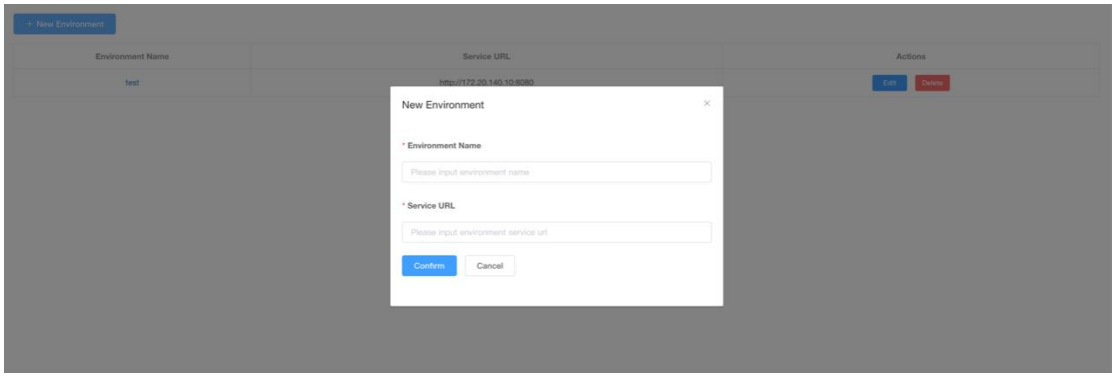
在浏览器中键入 URLhttps://ip:7750/ui/index.html，在登录界面输入用户名和密码进行登录，默认都是 admin。



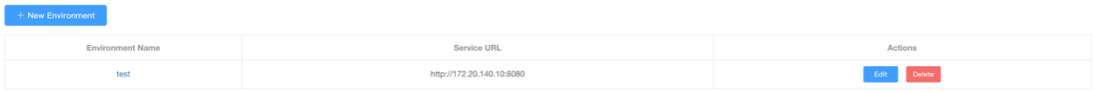
第2章 配置环境

管理控制台可以管理多个集群。

新建环境，需要输入集群环境名称和 ADMQ 集群中任意一台 BROKER 的 WEB 服务 URL

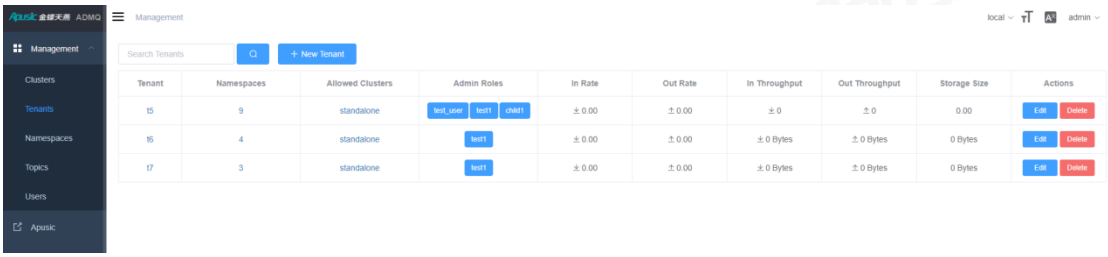


环境列表，列出当前的环境，可以进行编辑和删除。

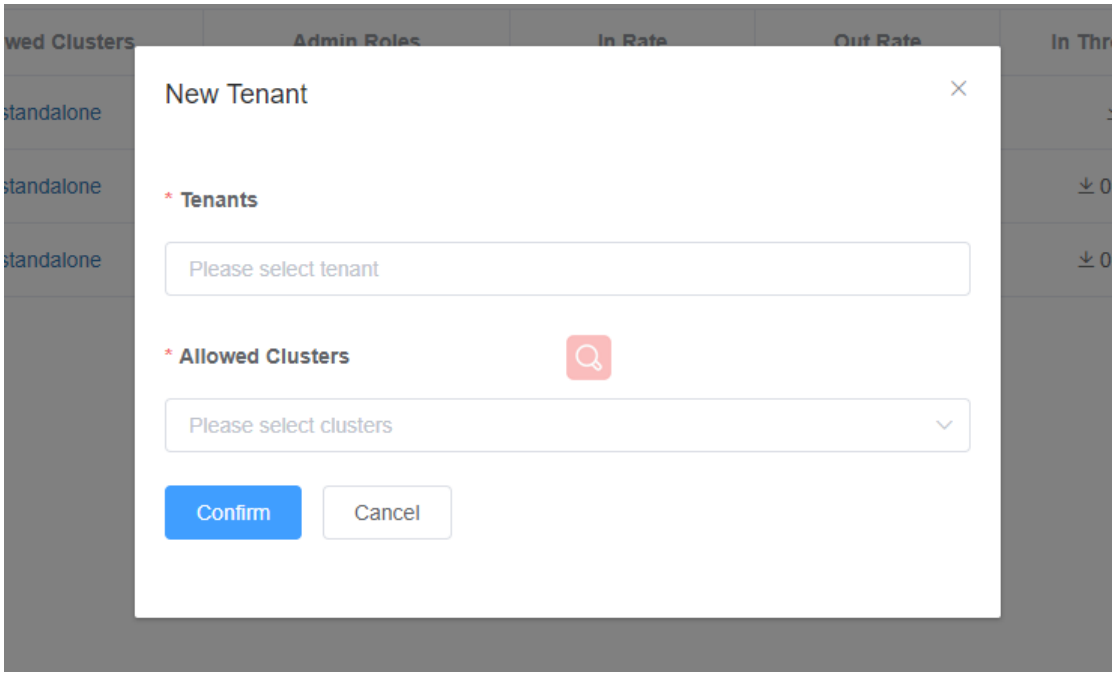


第3章 管理租户

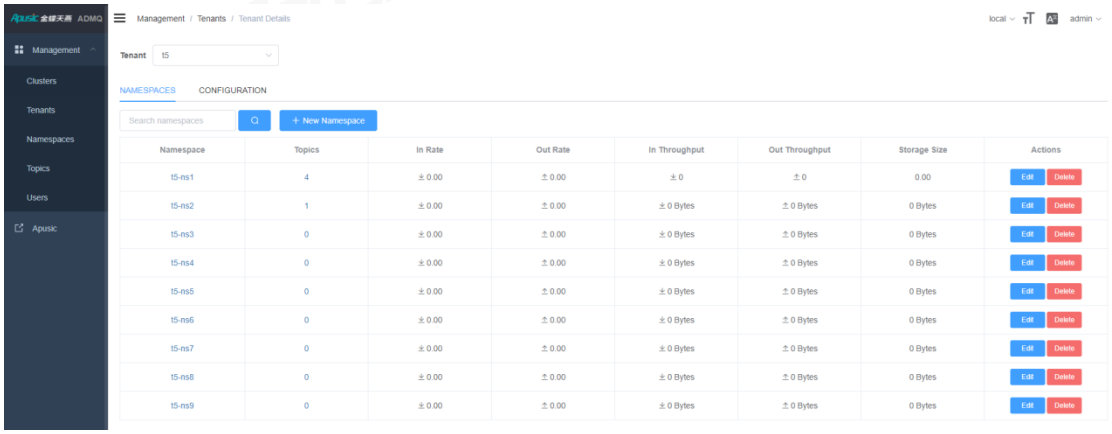
租户列表，列出当前环境的所有租户：



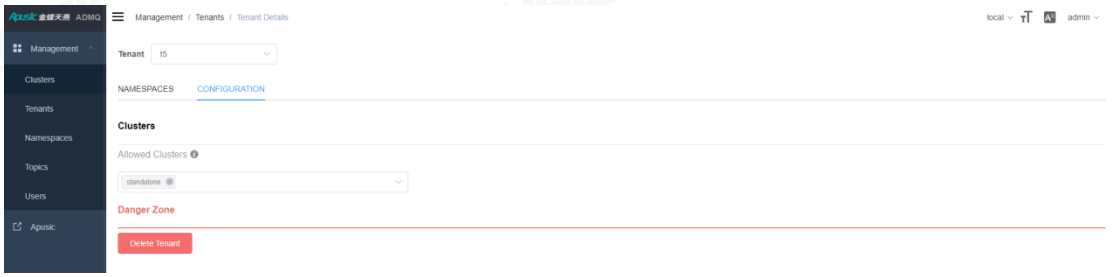
创建租户，指定租户名称、所属的集群：



租户包含的命名空间列表：

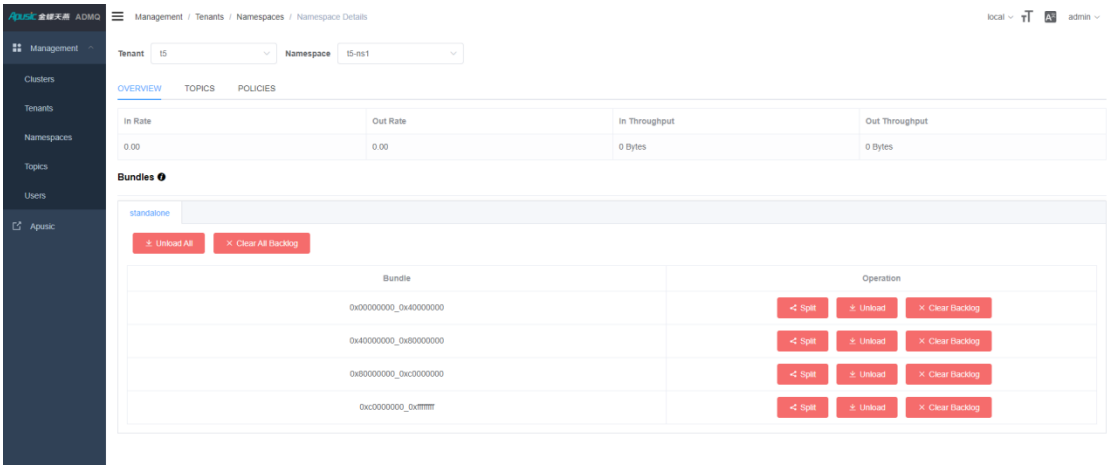


租户的配置信息：

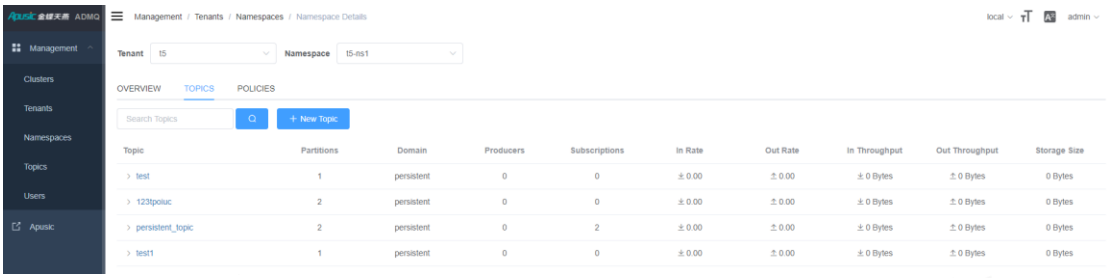


第4章 管理命名空间

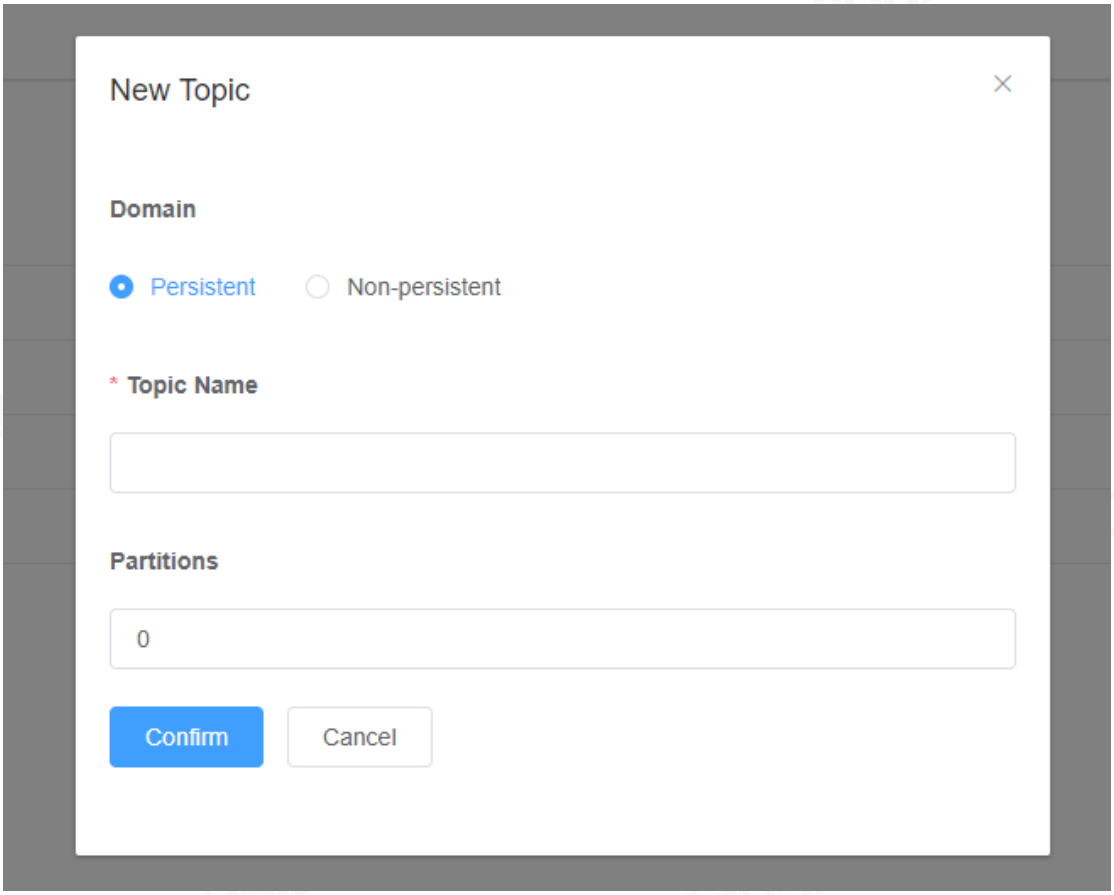
从命名空间列表，进入某个命名空间的概要信息，还可以对 bundle 进行卸载和拆分：



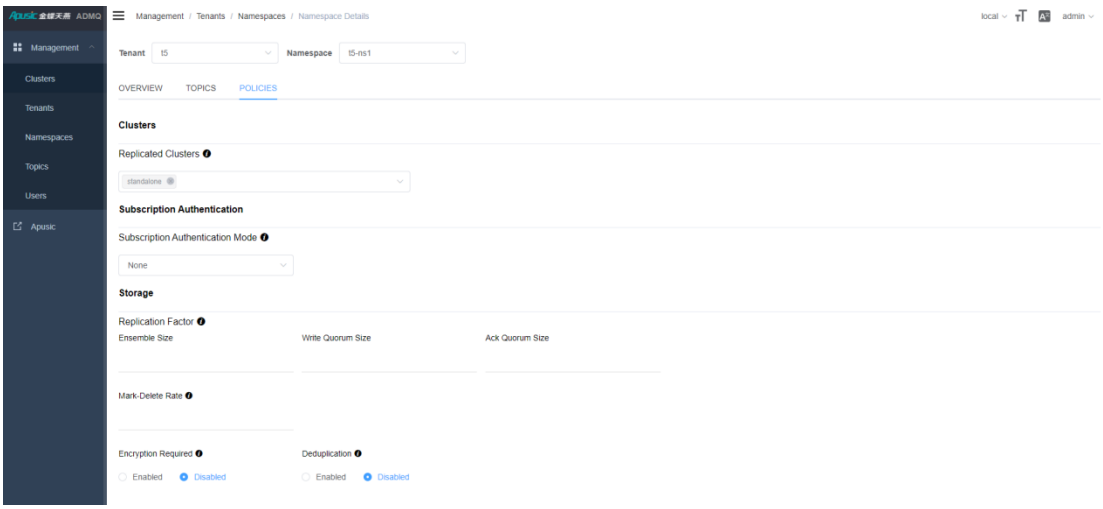
查看命名空间的主题列表：



在命名空间新建主题：



命名空间的策略配置：



Replicated Clusters（复制的集群）：这是有多个集群且开启了跨集群复制时，表示数据会在哪几个集群间进行复制。

Subscription Authentication Mode（订阅身份验证模式）：配置订阅身份验证，身份验证时限制订阅命名约定的模式已启用。可用选项：“NONE(无)”、“Perfix(前缀)”

- NONE:每个客户端角色都可以使用任何允许的订阅名称连接。
- Perfix:只允许客户端使用以其角色名称为前缀的订阅名称去订阅 例如，如果客户机被验证为“example”，则它只能使用前缀为“example”的订阅名称进行订阅。不允许使用其他订阅名称。

Ensemble Size（存储集合大小）：该命名空间使用多少台 bookie 来存储数据。

Write Quorum Size（写入 Quorum 大小）：该命名空间下每条消息使用多少台 bookie 来存储数据，必须小于等于 Ensemble Size 设置的数量。

Ack Quorum Size（Ack Quorum 大小）：多少台 bookie 完成消息存储就认为这条消息已经存储成功，必须小于等于 Write Quorum Size

Mark-Delete Rate（标记删除率）：每秒允许多少被标记为删除的速率限制，0 为无限制。

Encryption Required（消息加密）：对命名空间强制执行消息加密。

Deduplication（重复数据消除）：命名空间的重复数据消除。

Backlog Quota Size（backlog 配额大小）：命名空间允许的最大积压配额（以字节为单位）

Backlog Retention Policy（backlog 保留策略）：当达到 backlog 配额时要强制执行的保留策略。

有效选项为:

producer_request_hold:在资源可用（或等待超时）之前保留生产者发送的请求的策 略.

producer_exception: 拒绝生产者发送的请求的策略.

consumer_backlog_eviction: 从最慢的消费者的 backlog 中逐出最旧消息的策略.

AutoUpdate Strategy（schema 自动更新策略 ）:

兼容性检查策略	定义	更改	检查 Schema 版本	优先升级
ALWAYS_COMPATIBLE	禁用 schema 兼容性检查。	允许所有更改	所有旧版本	任何请求
ALWAYS_INCOMPATIBLE	禁用 schema 演化。	禁用更改	无	无

BACKWARD	使用 schema V3 的 consumer 可以处理 producer 使用 schema V3 或 V2 编写的数据。	• 添加可选字段 • 删除字段 •	最新版本	Consumers
BACKWARD_TRANSITIVE	使用 schema V3 的	•	所有旧版本	Consumers

	consumer 可以处理 producer 使用 schema V3、V2 或 V1 编写的数据。	添加可选字段 • • 删除字段 •		
FORWARD	使用 schema V3 或 V2 的 consumer	• 添加字段 • •	最新版本	Producers

	可以处理 producer 使用 schema V3 编写的数据。	删除可选字段 •		
FORWARD_TRANSITIVE	使用 schema V3、V2 或 V1 的 consumer 可以处理 producer 使	• 添加字段 • • 删除可选字段 •	所有旧版本	Producers

	用 schema V3 编写的数据。			
FULL	Schema V3 和 V2 之间向后和向前兼容。	• 修改可选字段	最新版本	任何请求
FULL_TRANSITIVE	Schema V3、V2 和 V1 之间向后和	• 修改可选字段	所有旧版本	任何请求

	向前兼容。			

Schema Validation Enforced （schema 强制验证）：对生产者强制 schema 验证。如果禁 用，则允许没有 schema 的生产者向具有 schema 的主题生成消息。

Message TTL (seconds)（消息 TTL（秒））：以秒为单位设置消息 TTL。如果订阅的任何使用者未使用这些消息，则在为该订阅配置的 TTL 期间之后，这些消息将标记为“已使用”。

Retention Size (megabytes)（保留大小（MB））：保留大小，只应用于消息被所有订阅 确认。

Retention Period (minutes)（保留期限（分钟））：保留期限，只应用于消息被所有订 阅确认

Compaction Threshold (bytes)（压缩阈值（字节））：当存储大小达到阈值时，将自动 触发压缩。

Offload Threshold (bytes) （卸载阈值（字节））：当未卸载消息的数据大小达到阈值时， 消息将自动卸载到分层存储。

Offload Deletion Lag (milliseconds) （卸载删除延迟（毫秒））：在删除从 bookie 卸载 的 ledger 前等待的毫秒数 。负值表示删除已完全禁用。

Max Producers Per Topic （每个主题的最大生产者）：每个主题允许的最大生产者数量。

Max Consumers Per Topic（每个主题最大消费者）：每个主题允许的最大消费者数量。

Max Consumers Per Subscription（每个订阅的最大消费者数）：每个订阅允许的最大消 费者数。

Dispatch Rate Per Topic（每个主题的调度率，根据每个主题限制 分派速率。）：

Throughput (bytes/second)：吞吐量（字节/秒）

Rate (messages/second)：速率（消息/秒）

Time Period (seconds)：时间段（秒）

Dispatch Rate Per Subscription （每个订阅的调度率，根据每个订阅限制分派速率。）：

Throughput (bytes/second)：吞吐量（字节/秒）

Rate (message/second)：速率（消息/秒）

Time Period (seconds)：时间段（秒）

Subscribe Rate Per Consumer（每个消费者的订阅率，限制消费者尝试订阅主题的速度。）：

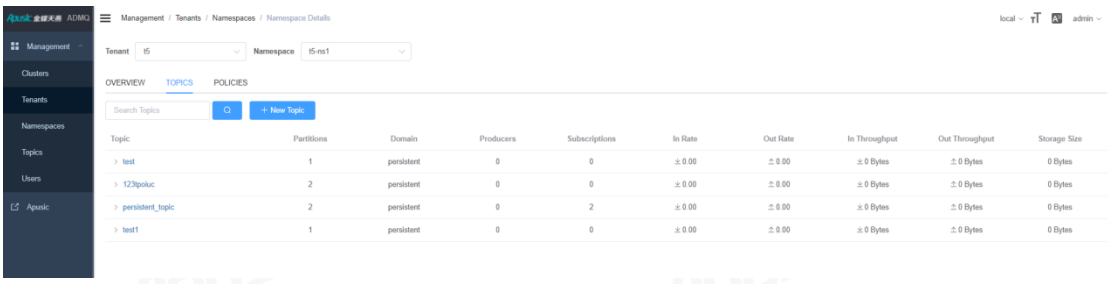
Rate (subscribes/second)：速率（消息/秒）

Time Period (seconds)：时间段（秒）

Anti-Affinity Group： 为此命名空间配置 Anti-Affinity 组。

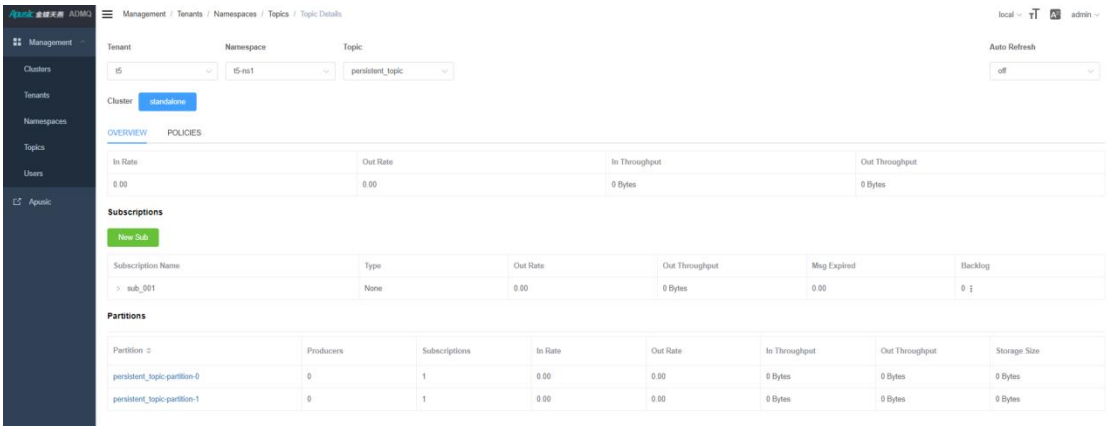
第5章 管理主题

主题的概要信息浏览：



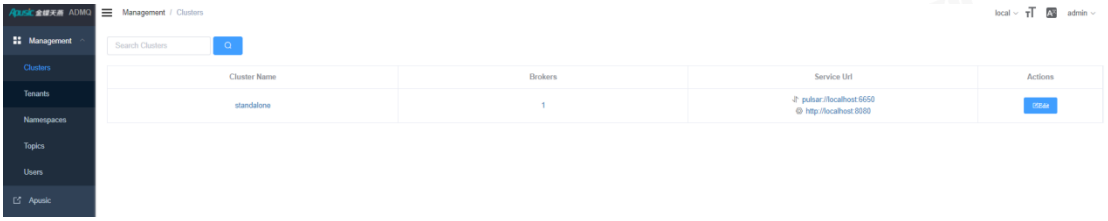
第6章 管理订阅

在主题概览页面，可以查看到该主题的所有订阅，点击某个订阅，可以对其进行管理：

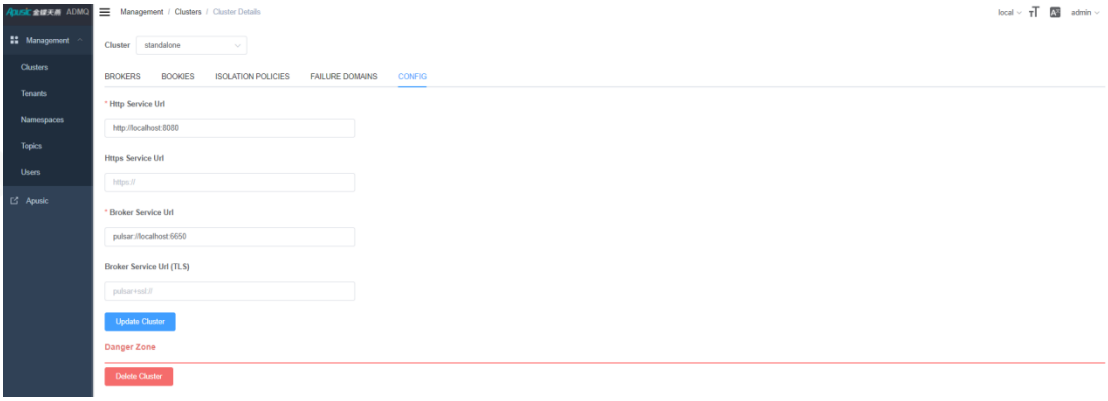


第7章 管理集群

集群列表



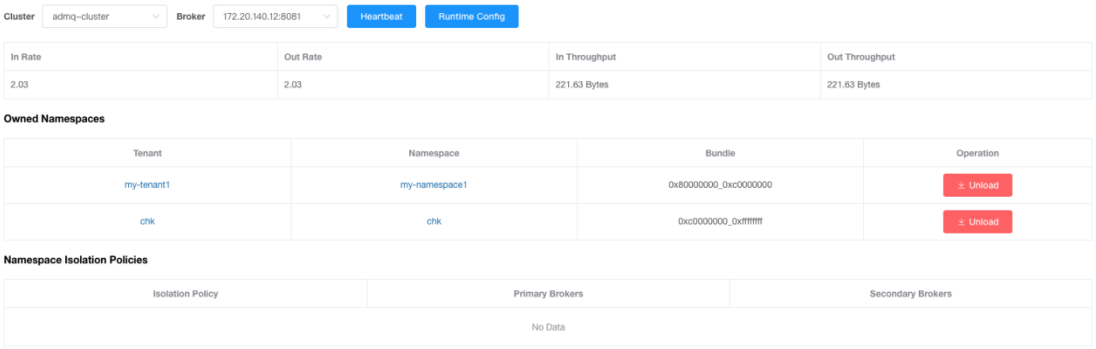
从集群列表，进入某个集群的配置信息



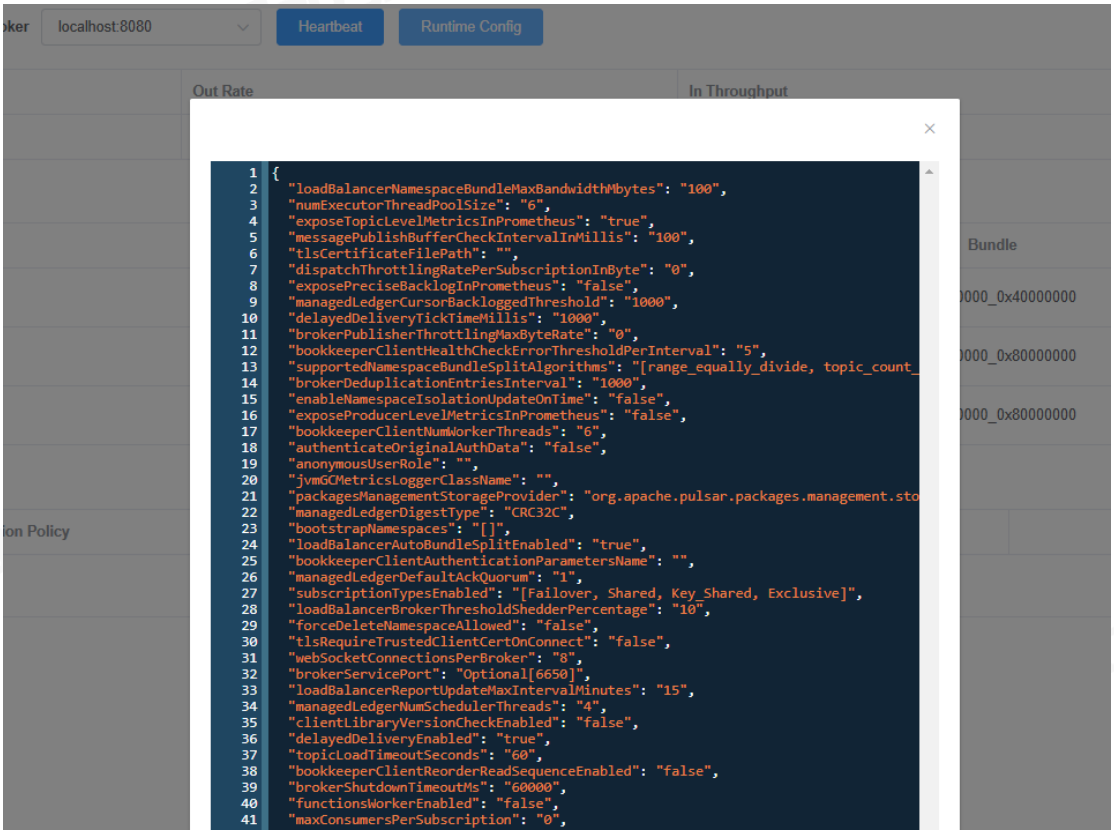
第8章 管理 broker/bookie

集群的 broker 列表，可以看到集群中 broker 的域名或者 ip 和端口号，拥有的命名空间，当前的消息收发速率和吞吐量。

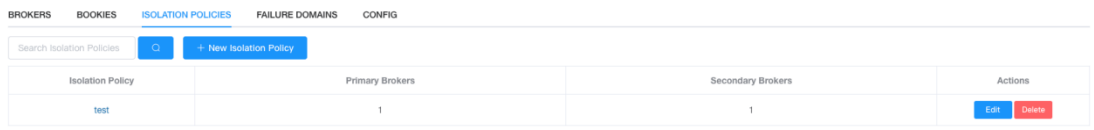
从集群列表，进入 broker 的概览信息页面：



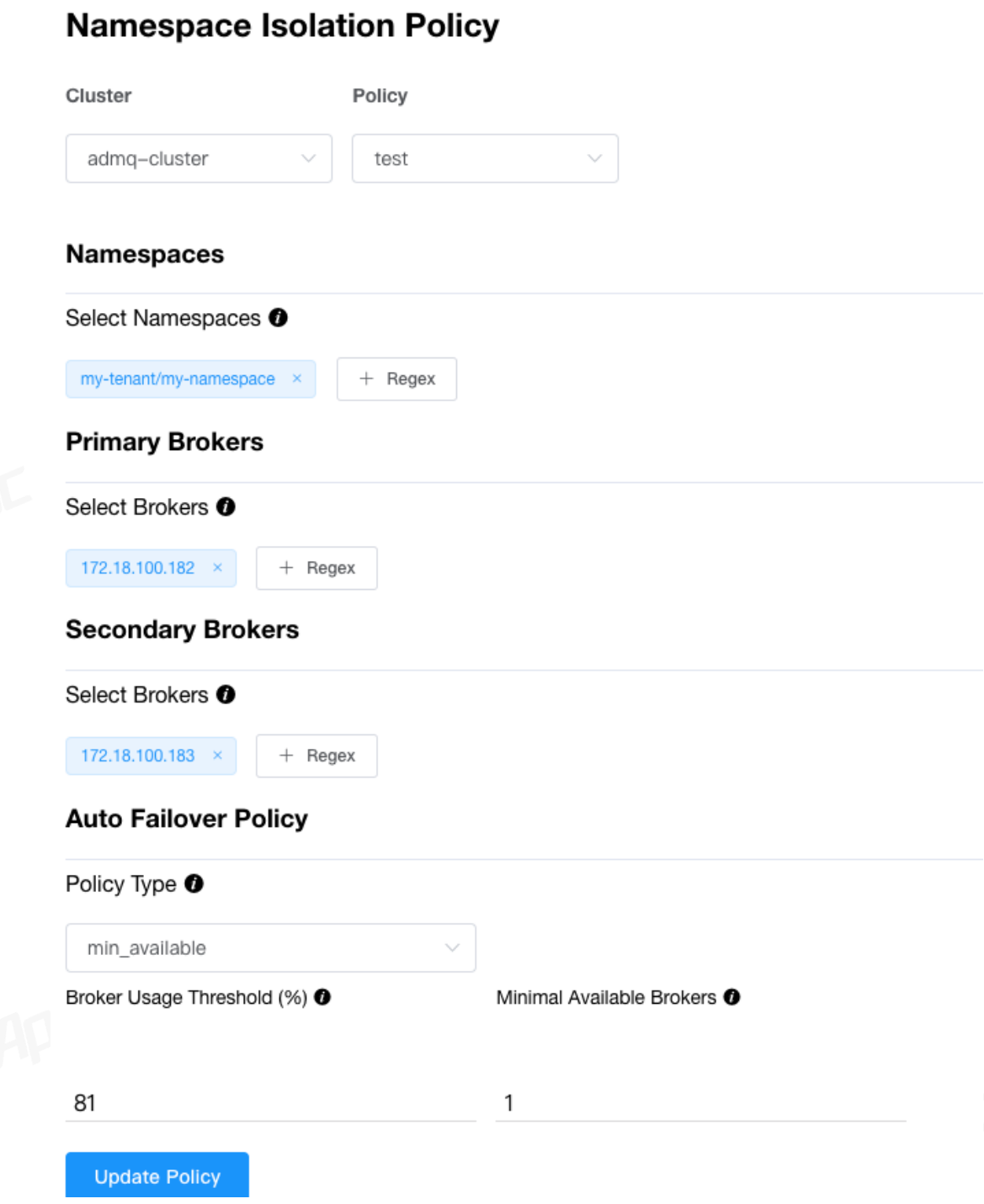
点击 Runtime Config，查看 broker 的运行时配置：



命名空间 broker 级别隔离策略列表：

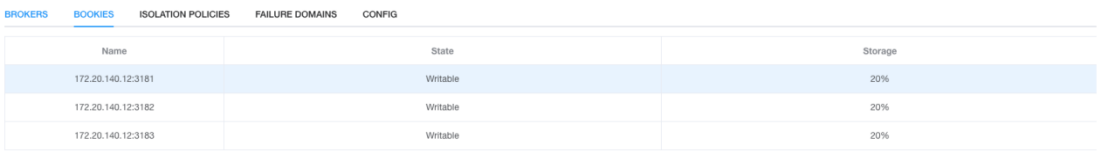


新建隔离策略：



配置好集群名称，命名空间，主要和次要 broker，自动故障转移策略等相关参数，该 namespaces 的数据只往 primary 的 broker 发，然果 primary broker 达到了自动故障转移策略的设置，就将数据往 secondary brokers 发。

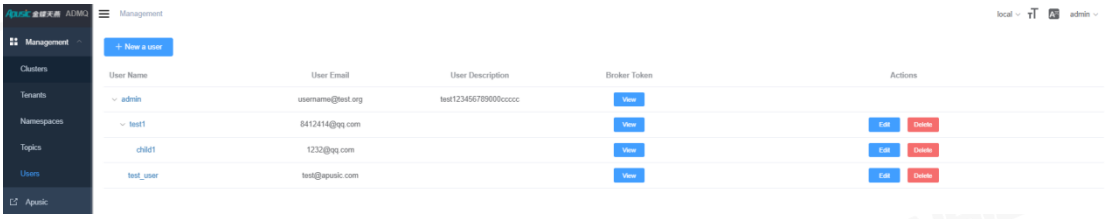
集群的 bookie 列表：



第9章 管理权限

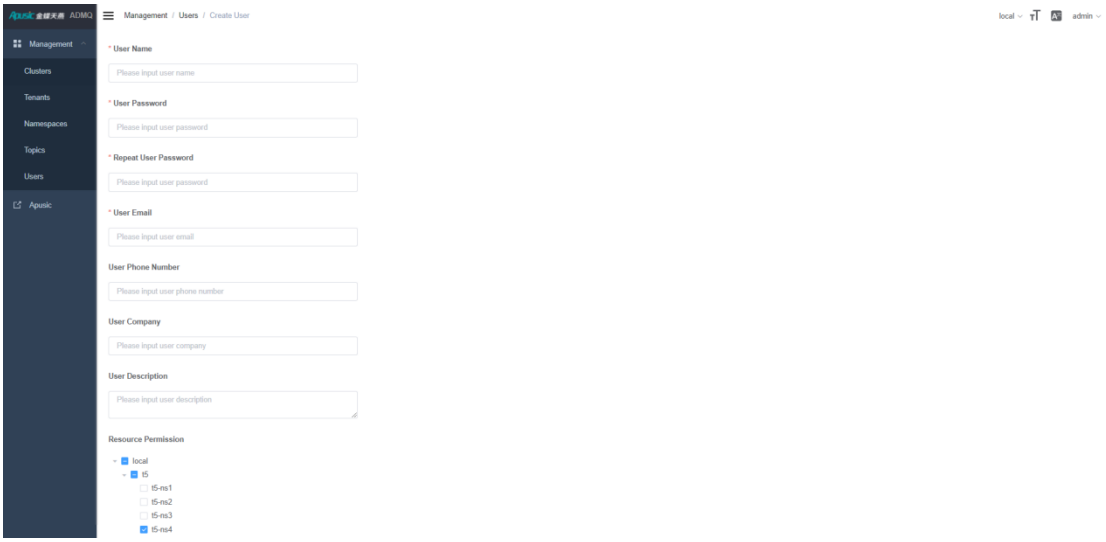
系统提供统一的权限管理界面，支持创建租户、命令空间和主题资源，并分配权限给指定的用户。

查看当前用户列表：

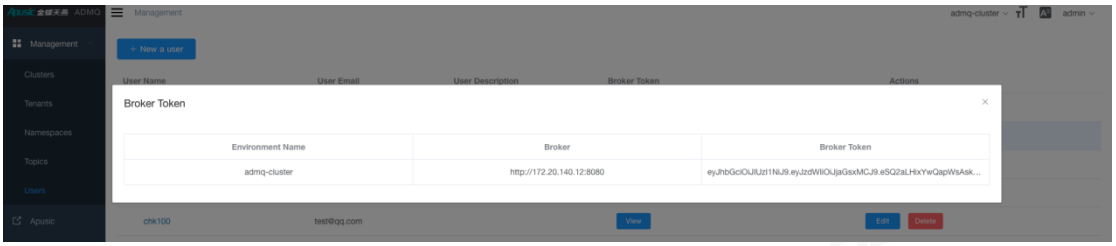


创建新用户并分配资源权限

输入用户名，密码，邮箱，勾选相应的权限，权限有 3 个层级，一级是集群环境，二级是租户，三级是命名空间，勾选上一级，则下一级的全部权限会自动获取，勾选下级某个数据权限，只会拥有该数据权限，比如现在有 my-namespace 的命名空间，下面有 my-topic 的主题，勾选 my-namespace 的话，则后续如果在 my-namespace 下创建新的主题，也会自动拥有该权限，如果只勾选 my-topic，则不管任何时候，只会有 my-topic 的权限。创建好用户后，后续该用户登陆管控台，只能看到拥有权限的数据，使用 admq 进行消息生产消费时，也只能对有权限的主题进行操作。



查看客户端认证时需要的 token 信息：



部分III 适配其他消息中间件接口

ADMQ 提供了适配其他消息中间件的 SDK，使得开发者们能利用 ADMQ 已有的基础架构，把 ADMQ 当作一个可靠、高效、稳定的流数据存储，可以利用它去开发一些可插拔消息协议。比如 KOA 协议、AOA 协议、JOA 协议等等。使用可插拔的消息协议后，那些从其他消息应用程序切换到 ADMQ 的用户，在做迁移的时候，就能允许在不改变代码的情况下将其使用的客户端应用程序切换到 ADMQ。

下面以单机部署模式说明各个插件的使用方式。

第1章 Kafka On ADMQ

KoA(Kafka on ADMQ) 将 Kafka 协议处理插件引入 ADMQ broker，从而实现 ADMQ 对原生 Apache Kafka 协议的支持。将 KoA 协议处理插件添加到现有 ADMQ 集群后，用户不用修改代码就可以将现有的 Kafka 应用程序和服务迁移到 ADMQ，从而使用 ADMQ 的强大功能，例如：

- 利用企业级多租户特性简化运营
- 避免数据搬迁，简化操作
- 利用 Apache BookKeeper 和分层存储持久保留事件流
- 利用 Pulsar Functions 进行无服务器化事件处理

1.1 参数配置

修改配置文件 /yourpath/admq-V1.0.0/share/pulsar/conf/standalone.conf

设置协议

messagingProtocols=kafka

protocolHandlerDirectory=/yourpath/admq-V1.0.0/lib

#allowAutoTopicCreationType 默认是不分片的，但是 KoA 必须设置分片 partitioned

allowAutoTopicCreationType=partitioned

设置监听器

kafkaListeners=PLAINTEXT://127.0.0.1:9092

kafkaAdvertisedListeners=PLAINTEXT://127.0.0.1:9092

设置偏移管理

brokerEntryMetadataInterceptors=org.apache.pulsar.common.intercept.AppendIndexMetadataInterceptor

关闭 topic 自动删除

ADMQ 会删除已分区主题的非活动分区，而不会删除已分区主题的元数据。在这种情况下，KoP 无法创建丢失的分区，所以这个设置非常重要。

brokerDeleteInactiveTopicsEnabled=false

1.2 测试

下载 kafka2.0 以上版本

运行生产者

bin/kafka-console-producer.sh --broker-list 127.0.0.1:9092 --topic test

运行消费者

bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic test --from-beginning

之后就可以持续看到消息的发送和接收。

第2章 MQTT On ADMQ

2.1 参数配置

修改配置文件 /yourpath/admq-V1.0.0/share/pulsar/conf/standalone.conf

messagingProtocols=mqtt

protocolHandlerDirectory=./protocols

mqttListeners=mqtt://127.0.0.1:1883

advertisedAddress=127.0.0.1

2.2 测试

使用 mqtt 客户端进行测试。

添加 pom 依赖:

```
<dependency>
    <groupId>org.fusesource.mqtt-client</groupId>
    <artifactId>mqtt-client</artifactId>
    <version>1.16</version>
</dependency>
```

测试代码:

```
// Java Code
MQTT mqtt = new MQTT();
mqtt.setHost("127.0.0.1", 1883);
BlockingConnection connection = mqtt.blockingConnection();
connection.connect();
Topic[] topics = { new Topic("persistent://public/default/my-topic", QoS.AT_LEAST_ONCE) };
connection.subscribe(topics);

// publish message
connection.publish("persistent://public/default/my-topic", "Hello MOP!".getBytes(), QoS.AT_LEAST_ONCE, false);

// receive message
Message received = connection.receive();
System.out.println(new String(received.getPayload()));
```

第3章 JMS On ADMQ

ADMQ 在 java 客户端实现了 JMS 2.0（Java 消息服务）。可以使用对应的 Java 库（pulsar-jms）来使用它。通过 JOA 协议，可以实现 JMS 到 ADMQ 的应用迁移，而无需修改代码，就可以使用 ADMQ 了。

3.1 参数配置

配置名称	是否必须	作用	示例

webServiceUrl	是	Pulsar HTTP endpoint	http://localhost:8080
brokerServiceUrl	是	连接 pulsar broker 或 proxy	pulsar://localhost:6650
enableTransaction	否	启动事物	false
jms.usePulsarAdmin	否	允许客户端使用 admin api	false
authPlugin	否	授权插件	
authParams	否	pulsar 生成的 token	生成的 token
producerConfig	否	producer 的配置	Map<String,Object>
consumerConfig	否	consumer 的配置	Map<String,Object>

其中 authPlugin 可以设置为 org.apache.pulsar.client.impl.auth.AuthenticationToken

3.2 测试

使用 java 程序测试，添加 pom 依赖:

```
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>pulsar-jms-all</artifactId>
    <version>1.0.0</version>
</dependency>
```

编写代码:

```
// Java Code
String topic = "persistent://public/default/example-topic";

Map<String, Object> properties = new HashMap<>();
properties.put("webServiceUrl","http://127.0.0.1:8080");
properties.put("brokerServiceUrl","pulsar://127.0.0.1:6650");

Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put("batchingEnabled", false);
```

```
Map<String, Object> consumerConfig = new HashMap<>();
consumerConfig.put("receiverQueueSize", 1);
configuration.put("consumerConfig", consumerConfig);
configuration.put("producerConfig", producerConfig);

try (PulsarConnectionFactory factory = new PulsarConnectionFactory(properties); ){
    JMSContext context = factory.createContext();
    Queue queue = context.createQueue(topic);

    // Listen for messages...
    context.createConsumer(queue).setMessageListener(new MessageListener() {
        @Override
        public void onMessage(Message message) {
            try {
                System.out.println("Received: " + message.getBody(String.class));
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    });

    for (int i = 0; i < 10; i++) {
        String message = "Hello world! " + i;
        System.out.println("Sending: "+message);
        context.createProducer().send(queue, message);
    }

    Thread.sleep(10000);
}
```

第4章 AMQP On ADMQ

AoA（ADMQ 上的 AMQP，如 RabbitMQ）通过在 ADMQ 代理上引入 AMQP 协议处理程序，为 ADMQ 带来了原生的 AMQP 协议支持。通过将 AoA 协议处理程序添加到您现有的 ADMQ 集群，可以实现 AMQP 到 ADMQ 的应用迁移，而无需修改代码，就可以使用 ADMQ 了。

目前，AoA 协议处理程序支持 AMQP0-9-1 协议，仅支持持久交换和持久队列。一个 VirtualHost 对应的 namespace 只能设置一个 bundle。并且需要提前为 VirtualHost 创建一个 namespace 。

4.1 参数配置

修改配置文件 /yourpath/admq-V1.0.0/share/pulsar/conf/standalone.conf

```
# Protocol handler configuration
messagingProtocols=amqp
protocolHandlerDirectory=/yourpath/admq-V1.0.0/lib

# Set AMQP service listeners
amqpListeners=amqp://127.0.0.1:5672
advertisedAddress=127.0.0.1

# 当前 connection 中，channel 的最大数量，默认是 64. 设置成和 rabbitMQ 同样的 channel 大小 2047
# 如果保持默认 64，在使用中可能因为客户端 channel 数量已满，造成创建不出新的 channel 而报错。
amqpMaxNoOfChannels=2047

# 当前 connection 中，最大帧的大小，默认是 4MB
amqpMaxFrameSize=4194304

# 默认心跳超时时间 60s
amqpHeartBeat=60
```

4.2 为 VirtualHost 添加命名空间

需要提前初始化 VirtualHost，保证客户端使用之前，它是存在的。如果客户端使用了不存在的 VirtualHost，那么这个 VirtualHost 就需要作废，AOA 协议暂时不能初始化已经在客户端使用的 VirtualHost。

```
# must set one bundle
/yourpath/admq-V1.0.0/bin/admqctl namespaces create -b 1 public/vhost1
```

4.3 测试

添加 pom 依赖

```
<dependency>
   <groupId>com.rabbitmq</groupId>
   <artifactId>amqp-client</artifactId>
   <version>5.8.0</version>
</dependency>
```

编写测试代码

```
// Java Code
// create connection
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.setVirtualHost("vhost1");
connectionFactory.setHost("127.0.0.1");
connectionFactory.setPort(5672);
Connection connection = connectionFactory.newConnection();
Channel channel = connection.createChannel();

String exchange = "ex";
String queue = "qu";

// exchage declare
channel.exchangeDeclare(exchange, BuiltinExchangeType.FANOUT, true, false, false, null);

// queue declare and bind
channel.queueDeclare(queue, true, false, false, null);
```

```
channel.queueBind(queue, exchange, "");

// publish some messages
for (int i = 0; i < 100; i++) {
    channel.basicPublish(exchange, "", null, ("hello - " + i).getBytes());
}

// consume messages
CountDownLatch countDownLatch = new CountDownLatch(100);
channel.basicConsume(queue, true, new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        System.out.println("receive msg: " + new String(body));
        countDownLatch.countDown();
    }
});
countDownLatch.await();

// release resource
channel.close();
connection.close();
```

使命

推动民族软件创新与发展

愿景

数字政府服务领航者

价值观

致良知、走正道、行王道

Apusic 金蝶天燕

深圳金蝶天燕云计算股份有限公司

电话：400-555-800

网站：www.apusic.com

北京总部：北京市顺义区金蝶软件园　深圳总部：深圳市南山区金蝶软件园　上海总部：上海市浦东区金蝶软件园