

RocketMQ

学习笔记

寿命齿轮

2024 年 1 月 20 日

作者的话

本文用于记录作者在学习 RocketMQ 过程中所记录的笔记。

学习资料来源：

1. RocketMQ 官网
2. 编程导航星球知识库：Yes 大佬的消息队列专栏
3. 网上各类与消息队列相关的博客

环境：

系统 Ubuntu22.04(云服务器)

Java 版本 17

目录

第一章 消息队列：初识	1
1.1 介绍	1
1.2 作用与优点	1
1.3 适用场景	2
1.3.1 异步场景举例：用户注册	2
1.3.2 解耦场景举例：订单-库存管理	2
1.3.3 削峰场景举例：秒杀活动	3
1.3.4 日志处理场景	3
1.3.5 消息通讯场景	4
1.4 常用消息队列框架	4
第二章 RocketMQ：启动	6
2.1 下载二进制文件包	6
2.2 启动 NameServer	7
2.3 启动 Broker+Proxy	9
2.4 关闭服务器	10
第三章 领域模型	11
3.1 架构总览	11
3.2 基本概念	12
3.2.1 消息的生命周期	12
3.2.2 通信方式	13
3.2.3 消息传输模型	14
3.3 主题（Topic）	17

3.3.1	模型关系：主题	17
3.3.2	内部属性	17
3.3.3	Admin 工具相关操作	18
3.3.4	使用建议	18
3.4	队列（MessageQueue）	18
3.4.1	模型关系：队列	19
3.4.2	内部属性	19
3.4.3	使用建议	20
3.5	消息（Message）	20
3.5.1	模型关系：消息	20
3.5.2	内部属性	21
3.5.3	使用建议	21
3.6	生产者（Producer）	21
3.6.1	模型关系：生产者	22
3.6.2	内部属性	22
3.6.3	使用建议	23
3.7	消费者分组（ConsumerGroup）	23
3.7.1	模型关系：消费者分组	23
3.7.2	内部属性	24
3.7.3	使用建议	24
3.8	消费者（Consumer）	24
3.8.1	模型关系：消费者	25
3.8.2	内部属性	25
3.8.3	使用建议	26
3.9	订阅关系（Subscription）	26
3.9.1	模型关系：订阅关系	26
3.9.2	订阅关系判断原则	27
3.9.3	内部属性	29
3.9.4	使用建议	29

第四章 Springboot 快速集成 RocketMQ	30
4.1 添加 Maven 依赖	30
4.2 添加配置文件属性	30
4.3 生产者	31
4.3.1 普通消息	31
4.3.2 顺序消息	33
4.3.3 延迟消息	34
4.3.4 批量消息	35
4.3.5 事务消息	36
4.4 消费者	36
4.4.1 Push 模式	36
4.4.2 Pull 模式	37

第一章 消息队列：初识

1.1 介绍

消息队列，顾名思义，存放消息（可类比为请求）的队列（一种先进先出的数据结构）。

其是一种常用于分布式系统的中间件，可以在不同的应用程序、服务或系统之间传递消息，并且常用于解耦合不同部分的系统，使得系统更加可扩展和灵活。

基本原理：发送者将消息放入队列，接收者从队列中获取消息并处理。

消息队列实质是一种方式，一种在不同组件之间传递消息的通信方式。发送者和接收者之间不需要直接通信，它们只需了解如何发送和接收消息即可。

1.2 作用与优点

由上述内容，可推断出消息队列的一些作用：

- **解耦：**发送者和接收者只需要关心发送消息和接受消息，不用关心彼此。
- **异步：**发送者不关心消息的处理，即不用等待消息的响应，故支持异步。
- **削锋：**某些活动的流量过大、请求过多，可能导致系统宕机；消息队列可以作为缓冲区，将这些请求暂时存储起来，以避免瞬时高流量，然后按照系统处理能力逐步消费，实现流量的平滑处理，从而降低系统的压力，避免宕机。

以及身为分布式系统的固有优点：

- **可扩展性：**在解耦后，可方便地单独对发送者或接收者或消息队列进行动态伸缩。
- **可靠性：**由于消息队列允许多个消费者和生产者，并且通常支持消息持久化和复制，因此即使其中一个组件出现故障，系统仍然可以继续运行并且消息也不会丢失。

1.3 适用场景

（真实适用场景还是需要多实践才能掌握，这里仅介绍一些常用场景）

1.3.1 异步场景举例：用户注册

需求

用户注册后需向其发送注册邮件和注册短信。

设计

用户注册后，将注册信息写入数据库；发送注册邮件；发送短信。

如不使用消息队列，不进行异步解耦，即注册服务器需要同步远程调用写入数据库、发送注册邮件、发送短信的三个函数，将与其他应用发生多次交互，同时还得等待响应，假设一个操作需要 0.5s，则该操作会占用注册服务器一个线程的 1.5s。

使用消息队列后，注册服务器直接向消息队列中写入三个消息（数据库写入消息、邮件发送消息、短信发送消息），并且是异步发送不用等待返回，假设一次发送消息为 0.1s，也仅需 0.3s。

1.3.2 解耦场景举例：订单-库存管理

需求

用户下订单后，库存系统需要减少相对数量。

设计

用户下单后，订单系统需要通知库存系统。

详细设计

原设计：订单系统调用库存系统的接口。存在缺陷：假如库存系统无法访问，则订单减库存将失败，从而导致订单失败；订单系统依赖库存系统接口，存在耦合。

改进后：订单系统发送订单消息（用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功），库存系统读取订单消息并自行处理（订阅订单消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作）。解决缺陷：假如库存

系统无法访问，订单系统仅需要发送消息，可保持运转；订单消息仅发送消息，消息解读由库存系统进行（发布-订阅或消息队列模式），降低耦合度。

1.3.3 削峰场景举例：秒杀活动

需求

在秒杀活动中，大量用户同时抢购商品，可能会导致系统压力激增。为了应对这一情况，需要一种机制来平稳处理激增的请求流量，避免系统崩溃或性能下降。

设计

传统的处理方式可能会导致系统崩溃或性能下降。为了解决这个问题，可以使用消息队列来削峰填谷。

详细设计

1. 秒杀活动开始：当秒杀活动开始时，用户可以提交秒杀请求。
2. 请求入队：订单系统接收到用户的秒杀请求后，将请求消息写入消息队列，而不是立即处理。
3. 消息处理：秒杀请求消息被消息队列按照一定的规则（如先进先出）分发给后端处理程序。
4. 后端处理：后端处理程序逐条处理消息，检查库存并进行相应的处理（如减少库存、生成订单等）。

以此消息队列可平滑处理激增的请求流量，避免系统因突发流量而崩溃。

1.3.4 日志处理场景

需求

需要一种解决大量日志传输和实时处理的方案，以便对日志数据进行分析 and 可视化展示。

设计

设计一个分布式日志处理系统，包括以下组件：

1. 日志采集客户端：负责从各个日志源采集日志数据，并将数据定期写入消息队列中。
2. 消息队列：接收来自日志采集客户端的日志数据，负责数据的存储和转发。

3. 日志处理应用：订阅并消费 Kafka 队列中的日志数据，进行实时处理和分析。
4. Logstash：作为日志处理应用的一部分，负责对原始日志进行解析和转换，统一输出为 JSON 格式的数据。
5. Elasticsearch：作为日志处理应用的核心数据存储服务，接收 Logstash 处理后的 JSON 格式日志数据，实现实时的数据索引和查询。
6. Kibana：基于 Elasticsearch 的数据可视化组件，用于将 Elasticsearch 中的数据进行可视化展示和分析。

1.3.5 消息通讯场景

需求

需要一种高效的消息通讯机制，可以用于点对点通讯或者创建聊天室等场景，以实现实时的消息传递和交流。

设计

设计一个基于消息队列的消息通讯系统，包括以下两种场景：

1. 点对点通讯：客户端 A 和客户端 B 使用同一队列进行消息通讯；消息队列负责接收和转发客户端 A 和客户端 B 的消息。
2. 客户端 A、客户端 B 等多个客户端订阅同一主题：当有客户端发布消息时，消息队列将消息广播给所有订阅了该主题的客户端，客户端收到消息后进行展示。

1.4 常用消息队列框架

1. **RabbitMQ**：RabbitMQ 是一个开源的消息队列系统，实现了高级消息队列协议（AMQP），它是一个可靠、高可用、可扩展的消息代理。RabbitMQ 提供了多种消息传递模式，如点对点、发布/订阅等，适用于各种场景的应用程序。
2. **RocketMQ**：RocketMQ 是阿里巴巴开源的分布式消息队列系统，具有高吞吐量、低延迟、高可用性等特点。它支持丰富的消息模型，包括顺序消息、事务消息等，适用于大规模分布式系统的消息通信。
3. **Kafka**：Kafka 是由 Apache 软件基金会开发的分布式流处理平台和消息队列系统。Kafka 设计用于支持大规模的消息处理，具有高吞吐量、持久性、分区等特点，广泛应用于大数据领域。

4. **ActiveMQ**: ActiveMQ 是一个开源的消息中间件，实现了 Java Message Service (JMS) 规范。它支持多种传输协议，如 TCP、UDP、SSL 等，提供了丰富的功能，包括消息持久化、事务支持等。
5. **Amazon SQS**: Amazon SQS (Simple Queue Service) 是亚马逊提供的消息队列服务，可帮助构建分布式应用程序。它具有高可用性、可扩展性、灵活性等特点，适用于构建在亚马逊云平台上的应用程序。

本文将使用 RabbitMQ。

第二章 RocketMQ：启动

2.1 下载二进制文件包

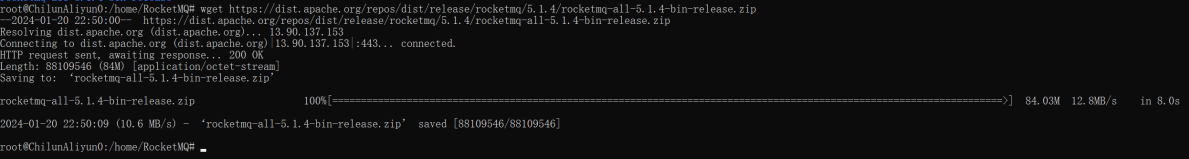
官网地址：<https://rocketmq.apache.org/zh/docs/quickStart/01quickstart>

获得二进制压缩包下载地址：

<https://dist.apache.org/repos/dist/release/rocketmq/5.1.4/rocketmq-all-5.1.4-bin-release.zip>

使用 `wget` 命令下载压缩包：

`wget https://dist.apache.org/repos/dist/release/rocketmq/5.1.4/rocketmq-all-5.1.4-bin-release.zip`



```
root@ChilunAliyun0:/home/RocketMQ# wget https://dist.apache.org/repos/dist/release/rocketmq/5.1.4/rocketmq-all-5.1.4-bin-release.zip
--2024-01-20 22:50:00-- https://dist.apache.org/repos/dist/release/rocketmq/5.1.4/rocketmq-all-5.1.4-bin-release.zip
Resolving dist.apache.org (dist.apache.org)... 13.90.137.153
Connecting to dist.apache.org (dist.apache.org) [13.90.137.153]:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 88109546 (84M) [application/octet-stream]
Saving to: 'rocketmq-all-5.1.4-bin-release.zip'

rocketmq-all-5.1.4-bin-release.zip 100%[=====] 84.03M 12.5MB/s in 8.0s

2024-01-20 22:50:09 (10.6 MB/s) - 'rocketmq-all-5.1.4-bin-release.zip' saved [88109546/88109546]
root@ChilunAliyun0:/home/RocketMQ#
```

图 2.1: 下载二进制压缩包

使用 `unzip` 命令解压二进制文件压缩包：

`unzip rocketmq-all-5.1.4-bin-release.zip`

```

root@ChilunAliyun0:/home/RocketMQ# ls
rocketmq-all-5.1.4-bin-release.zip
root@ChilunAliyun0:/home/RocketMQ# unzip rocketmq-all-5.1.4-bin-release.zip
Archive:  rocketmq-all-5.1.4-bin-release.zip
  inflating: rocketmq-all-5.1.4-bin-release/LICENSE
  inflating: rocketmq-all-5.1.4-bin-release/NOTICE
  inflating: rocketmq-all-5.1.4-bin-release/README.md
   creating: rocketmq-all-5.1.4-bin-release/benchmark/
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/consumer.sh
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/runclass.sh
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/tproducer.sh
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/batchproducer.sh
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/producer.sh
  inflating: rocketmq-all-5.1.4-bin-release/benchmark/shutdown.sh
   creating: rocketmq-all-5.1.4-bin-release/bin/
  inflating: rocketmq-all-5.1.4-bin-release/bin/tools.cmd
  inflating: rocketmq-all-5.1.4-bin-release/bin/runserver.sh
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqbroker.numanode0
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqproxy.cmd
  inflating: rocketmq-all-5.1.4-bin-release/bin/runbroker.cmd
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqbroker.numanodel
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqbrokercontainer
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqproxy
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqshutdown.cmd
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqcontroller
  inflating: rocketmq-all-5.1.4-bin-release/bin/runserver.cmd
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqadmin
  inflating: rocketmq-all-5.1.4-bin-release/bin/mqadmin.cmd
   creating: rocketmq-all-5.1.4-bin-release/bin/controller/

```

图 2.2: 解压二进制文件

2.2 启动 NameServer

进入目录 rocketmq-all-5.1.4-bin-release，执行命令：

nohup sh bin/mqnamesrv &

命令讲解：

- **nohup**：这代表“不挂起”。在终端中执行命令然后关闭终端时，与该命令相关联的进程通常也会终止。nohup 可以防止这种情况发生。
- **sh**：执行脚本文件的 shell 命令。
- **bin/mqnamesrv**：要运行的脚本路径。
- **&**：后台运行。

发现运行失败：

```

root@ChilunAliyun0:/home/RocketMQ/rocketmq-all-5.1.4-bin-release# nohup sh bin/mqnamesrv &
[1] 107550
root@ChilunAliyun0:/home/RocketMQ/rocketmq-all-5.1.4-bin-release# nohup: ignoring input and app
ending output to 'nohup.out'

[1]+  Exit 1                  nohup sh bin/mqnamesrv
root@ChilunAliyun0:/home/RocketMQ/rocketmq-all-5.1.4-bin-release#

```

图 2.3: 名字服务器启动失败

查看 nohup.out 文件, 发现报错: **OpenJDK 64-Bit Server VM warning: INFO: os::commit_memory(0x0000000700000000, 4294967296, 0) failed; error= 'Not enough space' (errno=12)**

操作系统内存不足 (由于 RocketMQ 对内存要求极高, 所以自己用云服务器运行基本都会报错), 进行修改:

进入 rocketmq-all-5.1.4-bin-release/bin 目录, 对 runserver.sh 和 runbroker.sh 以及 tools.sh 进行修改。(可使用 vim 的/+ 关键字进行查找)

1. runserver.sh:

```
JAVA_OPT="$JAVA_OPT -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m  
-XX:MaxMetaspaceSize=320m"
```

替换为

```
JAVA_OPT="$JAVA_OPT -server -Xms256m -Xmx256m -Xmn128m  
-XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

注意有两处。

2. runbroker.sh:

```
JAVA_OPT="$JAVA_OPT -server -Xms8g -Xmx8g"
```

替换为 **JAVA_OPT="\$JAVA_OPT -server -Xms256m -Xmx256m",**

```
JAVA_OPT="$JAVA_OPT -Xmn8G -XX:+UseConcMarkSweepGC
```

替换为

```
JAVA_OPT="$JAVA_OPT -Xmn256m -XX:+UseConcMarkSweepGC
```

3. tools.sh:

```
JAVA_OPT="$JAVA_OPT -server -Xms1g -Xmx1g -Xmn256m -XX:MetaspaceSize=128m  
-XX:MaxMetaspaceSize=128m"
```

替换为

```
JAVA_OPT="$JAVA_OPT -server -Xms256m -Xmx256m -Xmn128m  
-XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m"。
```

再次输入命令: **nohup sh bin/mqnamesrv &**

未报错, 查看 nohup.out 文件, 发现启动成功:

```
#
# There is insufficient memory for the Java Runtime Environment to continue.
# Native memory allocation (mmap) failed to map 4294967296 bytes for committing reserved memory
#
# An error report file with more information is saved as:
# /home/RocketMQ/rocketmq-all-5.1.4-bin-release/hs_err_pid107578.log
The Name Server boot success. serializeType=JSON, address 0.0.0.0:9876
```

图 2.4: 名称服务器启动成功

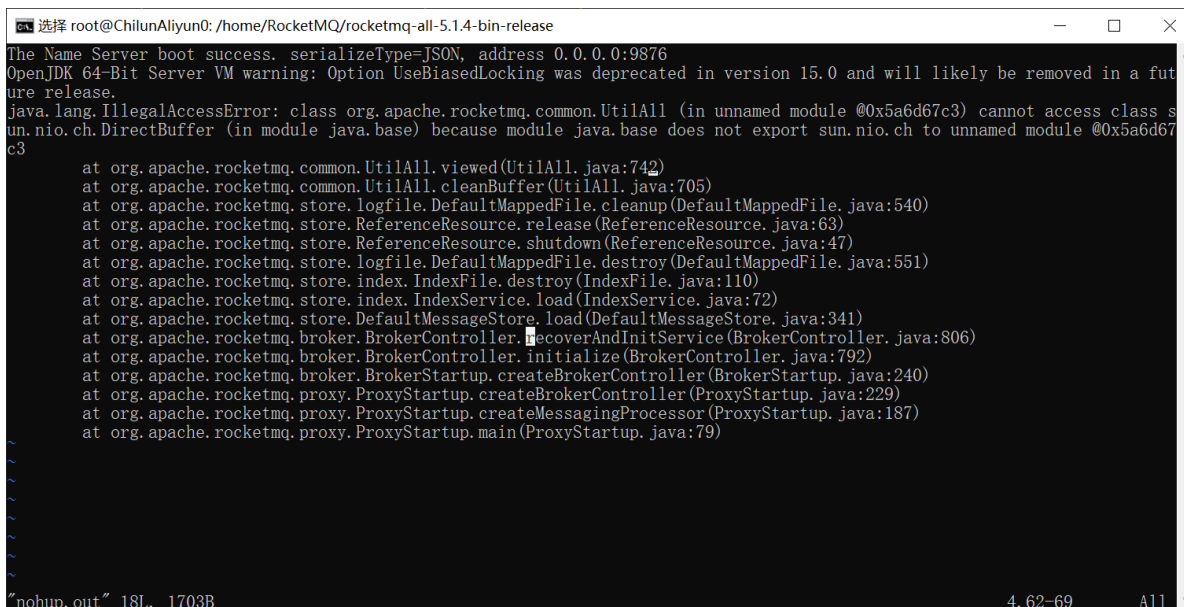
2.3 启动 Broker+Proxy

执行命令:

nohup sh bin/mqbroker -n localhost:9876 --enable-proxy &

可使用命令来查看是否启动成功: **tail -f ~/logs/rocketmqlogs/proxy.log**

未报错, 查看 nohup.out 文件, 发现启动还是失败:



```
选择 root@ChilunAliyun0: /home/RocketMQ/rocketmq-all-5.1.4-bin-release
The Name Server boot success. serializeType=JSON, address 0.0.0.0:9876
OpenJDK 64-Bit Server VM warning: Option UseBiasedLocking was deprecated in version 15.0 and will likely be removed in a future release.
java.lang.IllegalAccessException: class org.apache.rocketmq.common.UtilAll (in unnamed module @0x5a6d67c3) cannot access class sun.nio.ch.DirectBuffer (in module java.base) because module java.base does not export sun.nio.ch to unnamed module @0x5a6d67c3
    at org.apache.rocketmq.common.UtilAll.viewed(UtilAll.java:742)
    at org.apache.rocketmq.common.UtilAll.cleanBuffer(UtilAll.java:705)
    at org.apache.rocketmq.store.logfile.DefaultMappedFile.cleanup(DefaultMappedFile.java:540)
    at org.apache.rocketmq.store.ReferenceResource.release(ReferenceResource.java:63)
    at org.apache.rocketmq.store.ReferenceResource.shutdown(ReferenceResource.java:47)
    at org.apache.rocketmq.store.logfile.DefaultMappedFile.destroy(DefaultMappedFile.java:551)
    at org.apache.rocketmq.store.index.IndexFile.destroy(IndexFile.java:110)
    at org.apache.rocketmq.store.index.IndexService.load(IndexService.java:72)
    at org.apache.rocketmq.store.DefaultMessageStore.load(DefaultMessageStore.java:341)
    at org.apache.rocketmq.broker.BrokerController.recoverAndInitService(BrokerController.java:806)
    at org.apache.rocketmq.broker.BrokerController.initialize(BrokerController.java:792)
    at org.apache.rocketmq.broker.BrokerStartup.createBrokerController(BrokerStartup.java:240)
    at org.apache.rocketmq.proxy.ProxyStartup.createBrokerController(ProxyStartup.java:229)
    at org.apache.rocketmq.proxy.ProxyStartup.createMessagingProcessor(ProxyStartup.java:187)
    at org.apache.rocketmq.proxy.ProxyStartup.main(ProxyStartup.java:79)
"nohup.out" 18L, 1703B 4,62-69 A11
```

图 2.5: broker 启动失败

原因是 JAVA 版本过高, 进行修复。

修改 runbroker.sh 文件:

在 **numactl -interleave=all pwd > /dev/null 2>&1** 上方添加

\$JAVA \$JAVA_OPT --add-exports=java.base/sun.nio.ch=ALL-UNNAMED \$@

然后再次运行 **nohup sh bin/mqbroker -n localhost:9876 --enable-proxy &**

并查看 nohup.out, 发现为不断更新的日志文件, 推测运行成功。

查看/root/logs/rocketmqlogs, 发现运行成功。

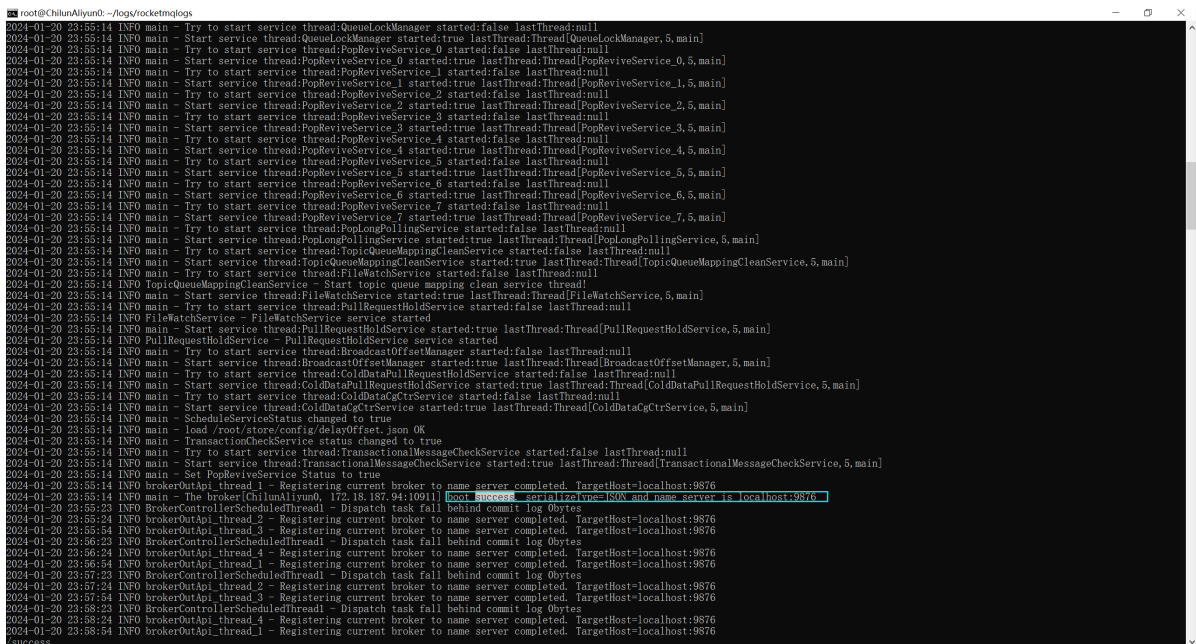


图 2.6: broker 启动成功

namesrv

至此，RocketMQ 启动成功。

2.4 关闭服务器

先关闭消息队列：**sh bin/mqshutdown broker**

再关闭名称服务：**sh bin/mqshutdown namesrv**

第三章 领域模型

3.1 架构总览

组件示意图：

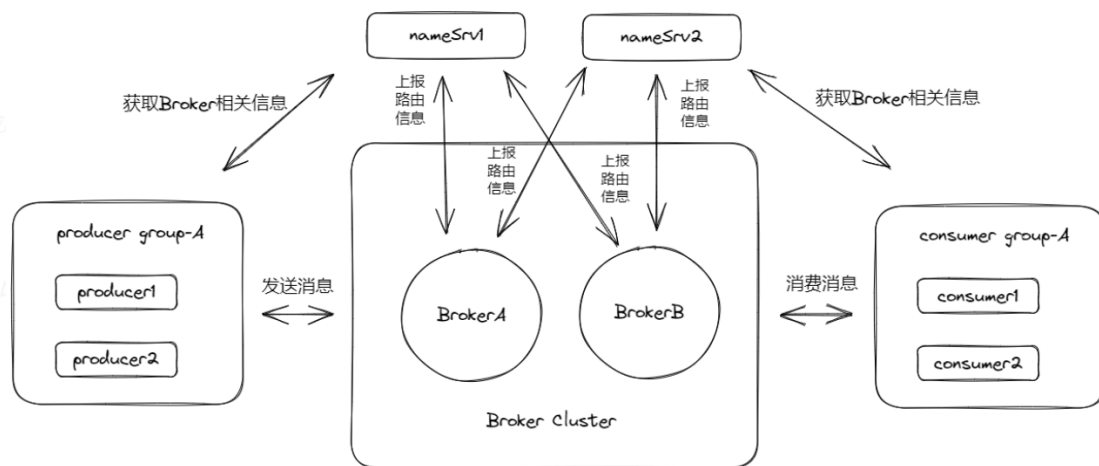


图 3.1: RocketMQ 全局图

- **Producer**：消息生产者。
- **NameSrv**：名称服务，即路由注册中心，可记录中转者提供给消费者和生产者。
- **Broker**：中转者（也可以认为是队列 Queue），负责接受、存储、转发消息。
- **Comsumer**：消息消费者。
- **Producer group**：生产者组。
- **Comsumer group**：消费者组。
- **Broker cluster**：中转者集群。

3.2 基本概念

3.2.1 消息的生命周期

RocketMQ 中消息的生命周期主要分为**消息生产**、**消息存储**、**消息消费**这三部分。

生产者生产消息并发送至 RocketMQ 服务端，消息被存储在服务端的主题中，消费者通过订阅主题消费消息。

消息生产

生产者 (Producer)：在 RocketMQ 中，生产者是用来发送消息的组件，通常它被集成到业务系统的前端。生产者是轻量级匿名无身份的。

消息存储

消息 (Message)：RocketMQ 的最小传输单元。消息具备不可变性，在初始化发送和完成存储后即不可变。

主题 (Topic)：RocketMQ 消息传输和存储的分组容器，主题内部由多个队列组成，消息的存储和水平扩展实际是通过主题内的队列实现的。

队列 (MessageQueue)：RocketMQ 消息传输和存储的实际单元容器，类比于其他消息队列中的分区。RocketMQ 使用流式特性的无限队列结构来存储消息，消息在队列内会按顺序进行存储。

消息消费

消费者分组 (ConsumerGroup)：RocketMQ 发布订阅模型中定义的独立的消费身份分组，用于统一管理底层运行的多个消费者 (Consumer)。同一个消费组的多个消费者必须保持消费逻辑和配置一致，共同分担该消费组订阅的消息，实现消费能力的水平扩展。

消费者 (Consumer)：RocketMQ 消费消息的运行实体，通常它被集成到业务系统的后端。消费者必须被指定到某一个消费组中。

订阅关系 (Subscription)：RocketMQ 的订阅关系是指在发布订阅模型中，用于配置消息过滤、重试以及消费进度的规则。订阅关系是以消费组为单位进行管理的，消费组通过定义订阅关系来控制该组下的消费者如何处理消息的过滤、重试以及消费进度恢复等操作。简单来说，订阅关系就是消费组对消息的一种规则设定，可以让我们更加灵活地控制消息的处理方式，确保系统能够按照我们预期的方式来消费和处理消息。

（RocketMQ 的订阅关系除过滤表达式之外都是持久化的，即服务端重启或请求断开，订阅关系依然保留。）

3.2.2 通信方式

分布式系统架构思想下，将复杂系统拆分为多个独立的子模块，例如微服务模块。此时就需要考虑子模块间的远程通信，典型的通信模式分为以下两种，一种是**同步的 RPC 远程调用**；一种是**基于中间件代理的异步通信**。

同步 RPC 调用模型

同步 RPC 调用模型下，不同系统之间直接进行调用通信，每个请求直接从调用方发送到被调用方，然后要求被调用方立即返回响应结果给调用方，以确定本次调用结果是否成功。

（注意：此处的同步并不代表 RPC 的编程接口方式，RPC 也可以有异步非阻塞调用的编程方式，但本质上仍然是需要在指定时间内得到目标端的直接响应。）

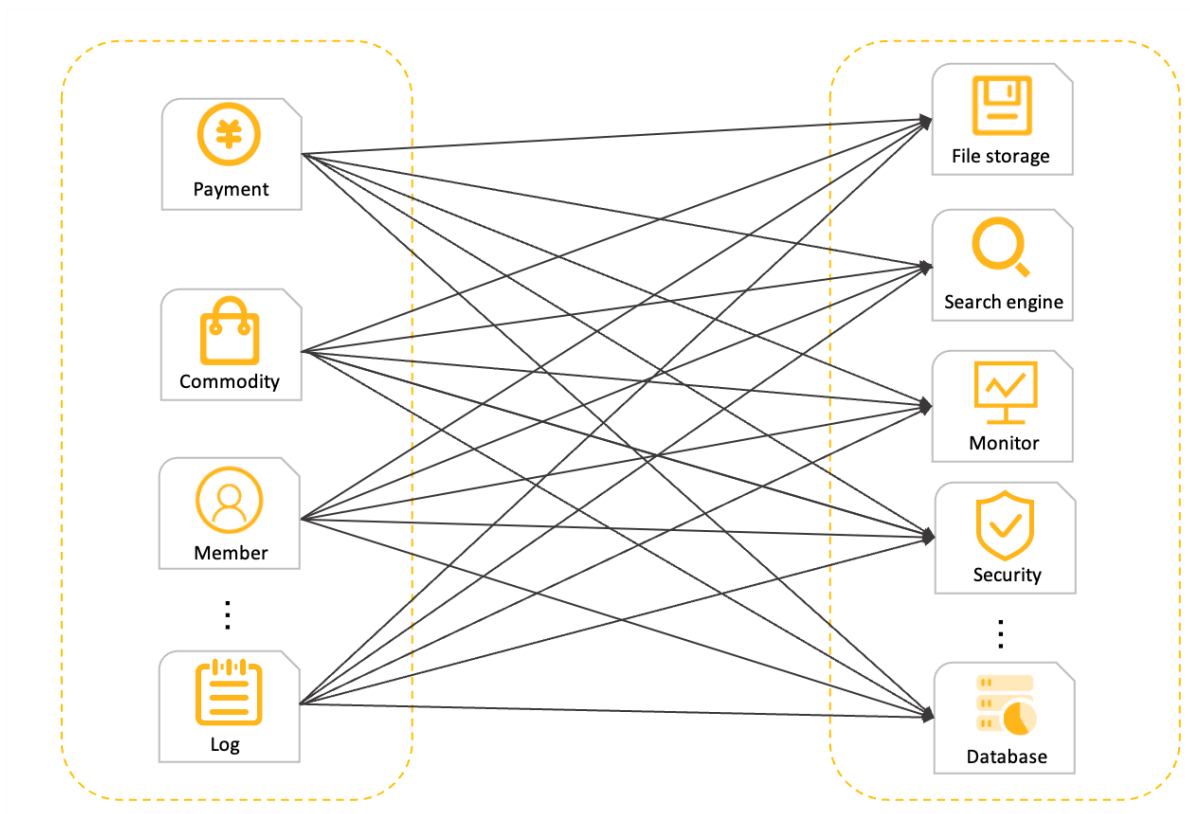


图 3.2: 同步 RPC 调用模型

异步通信模型

异步消息通信模式下，各子系统之间无需强耦合直接连接，调用方只需要将请求转化成异步事件（消息）发送给中间代理，发送成功即可认为该异步链路调用完成，剩下的工作中间代理会负责将事件可靠通知到下游的调用系统，确保任务执行完成。该中间代理一般就是消息中间件。

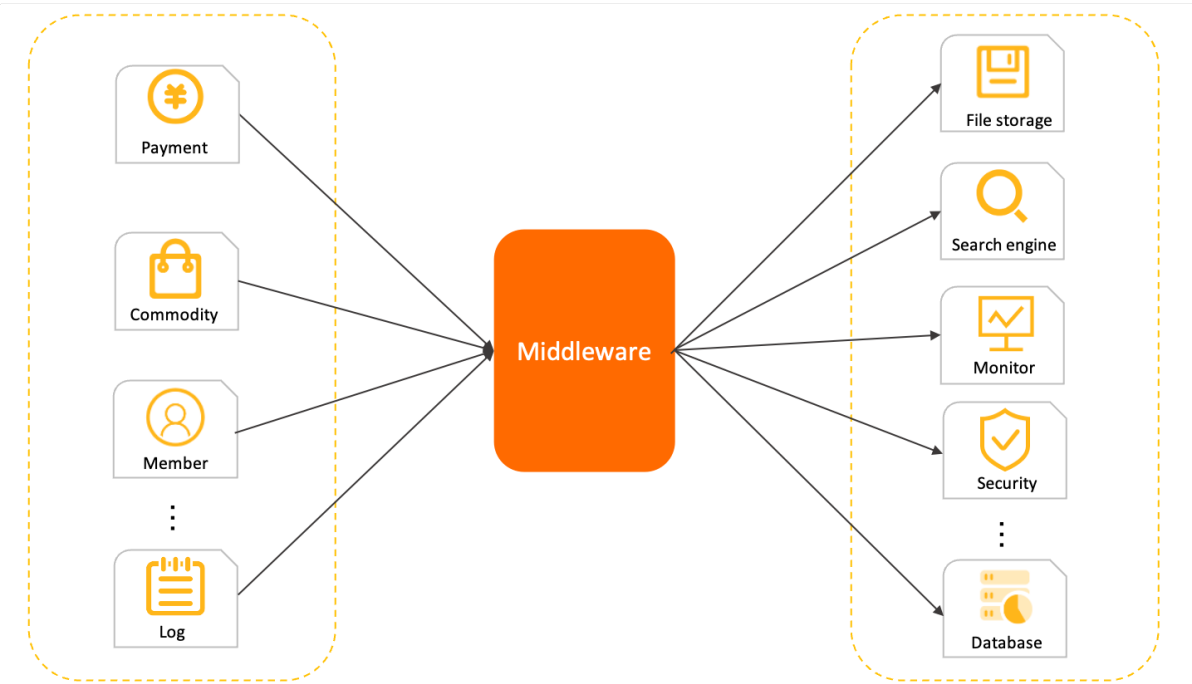


图 3.3: 异步通信模型

异步通信优势如下：

- 1. 星型结构系统，拓扑简单，易于维护和管理。
- 2. 上下游耦合性弱，可以独立升级和变更，不会互相影响。
- 3. 容量削峰填谷。基于消息的中间代理往往具备很强的流量缓冲和整形能力，业务流量高峰到来时不会击垮下游。

3.2.3 消息传输模型

主流的消息中间件的传输模型主要为点对点模型和发布订阅模型。

点对点模型

点对点模型也叫队列模型，具有如下特点：

1. 消费匿名：消息上下游沟通的唯一的身份就是队列，下游消费者从队列获取消息无法申明独立身份（即中间件对同一个队列中的所有消费者都一视同仁）。
2. 一对一通信：基于消费匿名特点，下游消费者即使有多个，但都没有自己独立的身份，因此共享队列中的消息，每一条消息都只会被唯一一个消费者处理。因此点对点模型只能实现一对一通信。

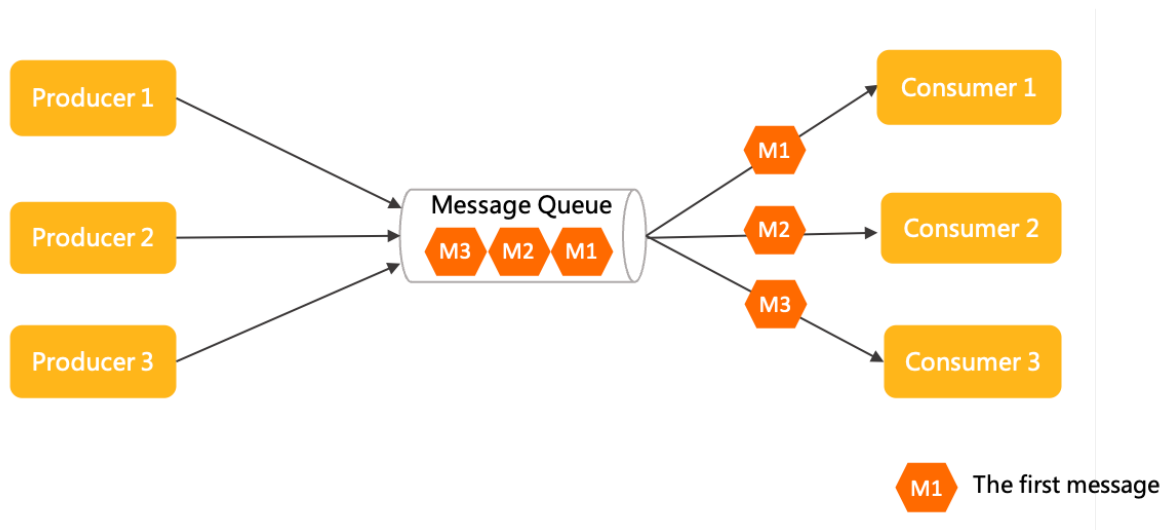


图 3.4: 点对点模型

发布订阅模型

发布订阅模型具有如下特点：

1. 消费独立：相比队列模型的匿名消费方式，发布订阅模型中消费方都会具备的身份，一般叫做订阅组（订阅关系），不同订阅组之间相互独立不会相互影响。
2. 一对多通信：基于独立身份的设计，同一个主题内的消息可以被多个订阅组处理，每个订阅组都可以拿到全量消息。因此发布订阅模型可以实现一对多通信。

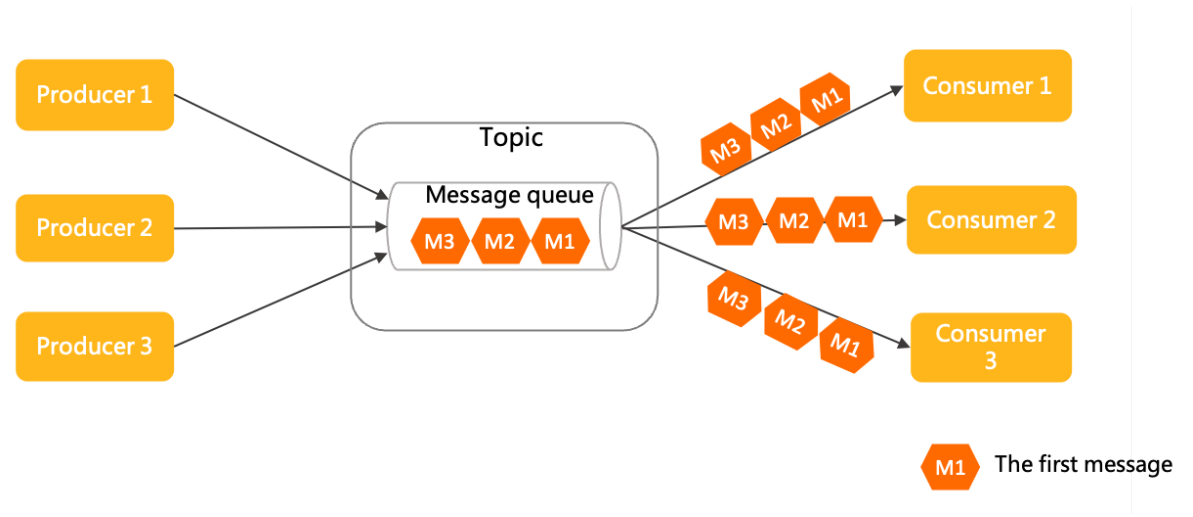


图 3.5: 发布订阅模型

对比

点对点模型和发布订阅模型各有优势，点对点模型更为简单，而发布订阅模型的扩展性更高。RocketMQ 使用的传输模型为发布订阅模型，因此也具有发布订阅模型的特点。

领域模型图

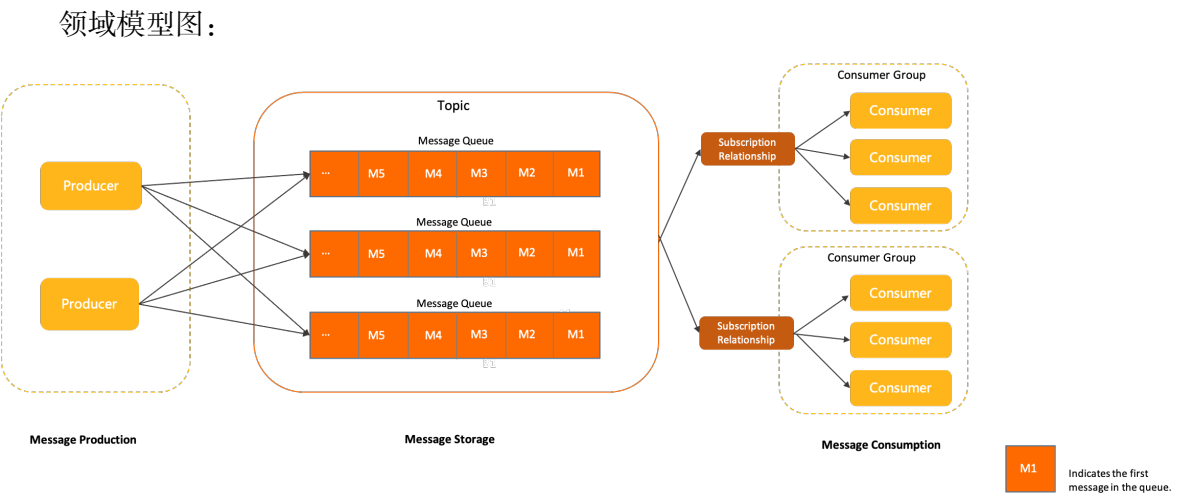


图 3.6: 领域模型图

3.3 主题 (Topic)

主题是 RocketMQ 中消息传输和存储的顶层容器，用于标识同一类业务逻辑的消息。主题的作用主要如下：

1. **数据分类**：在 RocketMQ 的方案设计中，建议将不同业务类型的数据拆分到不同的主题中管理，通过主题实现存储的隔离性和订阅隔离性。
2. **区分身份和权限**：RocketMQ 的消息本身是匿名无身份的，同一分类的消息使用相同的主题来做身份识别和权限管理。

3.3.1 模型关系：主题

在整个 RocketMQ 领域模型中，主题所处的位置如下（橙色区域）：

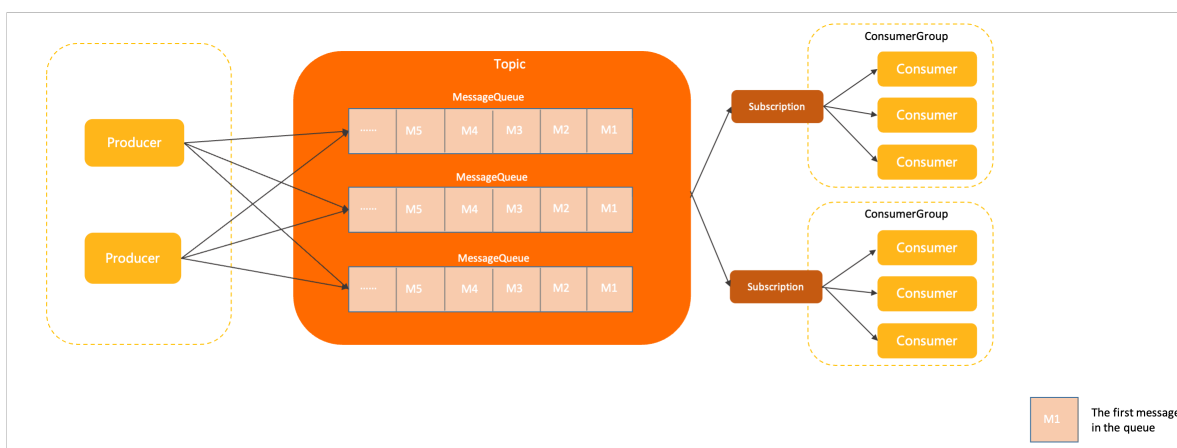


图 3.7: 模型关系：主题

主题是 RocketMQ 的顶层存储，所有消息资源的定义都在主题内部完成，但主题是一个**逻辑概念**，并不是实际的消息容器。

主题内部由多个队列组成，消息的存储和水平扩展能力最终是由队列实现的；并且针对主题的所有约束和属性设置，最终也是通过主题内部的队列来实现。

3.3.2 内部属性

1. **主题名称**：主题的名称，用于标识主题，主题名称集群内全局唯一，由用户创建主题时定义。
2. **队列列表**：队列作为主题的组成单元，是消息存储的实际容器，一个主题内包含一个或多个队列，消息实际存储在主题的各队列内；队列数量创建主题时定义（一个主题内至少包

含一个队列)。

3. 消息类型：主题所支持的消息类型（包括普通消息、顺序消息、定时/延时消息、事务消息），每个主题只允许发送一种消息类型的消息。

3.3.3 Admin 工具相关操作

创建主题操作命令：

```
sh mqadmin updateTopic -n <nameserver_address> -t <topic_name> -c <cluster_name> -a +message.type=<message_type>
```

各类操作（详情见RocketMQ 官网）：

1. updateTopic：创建/更新 Topic 配置。
2. deleteTopic：删除 Topic。
3. topicList：查看 Topic 列表信息。
4. topicRoute：查看 Topic 路由信息。
5. topicStatus：查看 Topic 消息队列 offset。
6. topicClusterList：查看 Topic 所在集群列表。
7. updateTopicPerm：更新 Topic 读写权限。
8. updateOrderConf：以平均负载算法计算消费者列表负载消息队列的负载结果。
9. statsAll：打印 Topic 订阅关系、TPS、积累量、24h 读写总量等信息。

3.3.4 使用建议

- 合理拆分主题，注意消息量级与消息时效性，避免将时效性要求高的业务消息与量级大的业务消息归到同一主题。
- 单一主题只收发一种类型消息，避免混用。
- 主题管理尽量避免自动化机制，并在生产环境需保证严格管理主题资源，不随意进行增、删、改、查操作。

3.4 队列（MessageQueue）

队列是 RocketMQ 中消息存储和传输的实际容器，也是 RocketMQ 消息的最小存储单元。

RocketMQ 的所有主题都是由多个队列组成，以此实现队列数量的水平拆分和队列内部的流式存储。

队列的作用如下：

- 1. 存储顺序性：队列天然具备顺序性，即消息按照进入队列的顺序写入存储。消息在队列中的位置通过位点（Offset）进行记录，队列头部为最早写入的消息（Offset=0），队列尾部为最新写入的消息（Offset 逐渐递增）。
- 2. 流式操作语义：RocketMQ 基于队列的存储模型可确保消息从任意位点（Offset）读取任意数量的消息，以此实现类似聚合读取、回溯读取等特性，这些特性是队列存储模型特有的。

3.4.1 模型关系：队列

在整个 RocketMQ 领域模型中，队列所处的位置如下（橙色区域）：

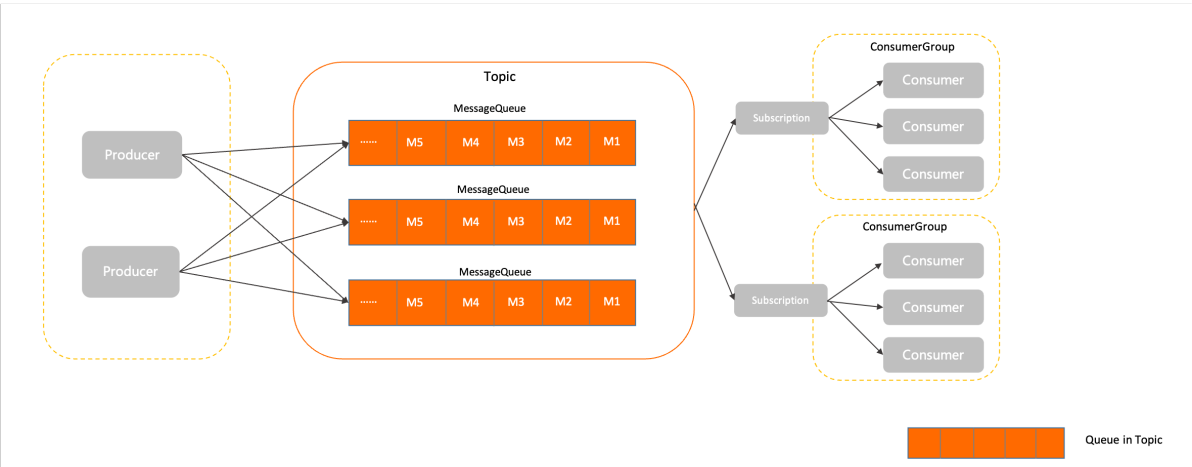


图 3.8: 模型关系：队列

RocketMQ 默认提供**可靠的消息存储机制**，所有发送成功的消息都被持久化存储到队列中，配合生产者和消费者客户端的调用保证至少投递一次。

队列属于主题的一部分，虽然所有的消息资源以主题粒度管理，但实际的操作实现是面向队列。例如，生产者指定某个主题，向主题内发送消息，但实际消息发送到该主题下的某个队列中。

RocketMQ 中可通过修改队列数量，以此实现横向的水平扩容和缩容。

3.4.2 内部属性

- 1. 读写权限：当前队列是否可以读写数据（队列的读写权限属于运维侧操作，不建议频繁修改）。

3.4.3 使用建议

- 队列数量的设置应遵循少用够用原则，否则可能导致集群元数据膨胀、系统负荷增加（队列越多，消费者的轮询越多）。
- 在集群水平扩容增加节点后，为了保证集群流量的负载均衡，建议在新的服务节点上新增队列，或将旧的队列迁移到新的服务节点上。
- 顺序消息的并发度会在一定程度上受队列数量的影响，当达到性能瓶颈时需要再增加队列。

3.5 消息（Message）

消息是 RocketMQ 中的**最小数据传输单元**。生产者将业务数据的负载和拓展属性包装成消息发送到 RocketMQ 服务端，服务端按照相关语义将消息投递到消费端进行消费。

消息的特点：

1. 不可变性：一旦产生后，消息的内容不会发生改变；消费端获取的消息都是只读消息视图。
2. 持久化：RocketMQ 会默认对消息进行持久化，即将接收到的消息存储到 RocketMQ 服务端的存储文件中，保证消息的可回溯性和系统故障场景下的可恢复性。

3.5.1 模型关系：消息

在整个 RocketMQ 领域模型中，消息所处的位置如下（橙色区域）：

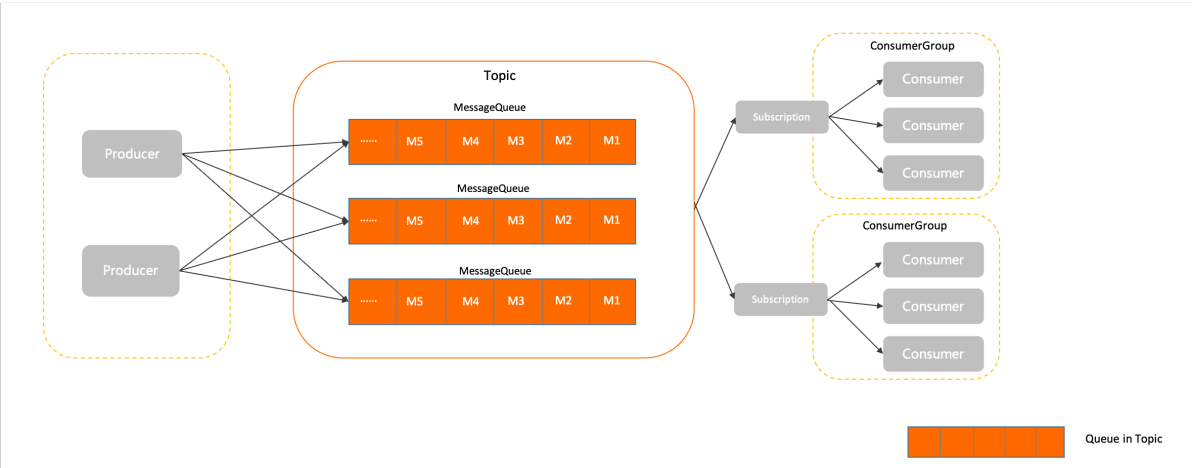


图 3.9: 模型关系：消息

消息由生产者初始化并发送到 RocketMQ 服务端，并按照到达 RocketMQ 服务端的顺序存储到队列中。消费者按照指定的订阅关系从 RocketMQ 服务端中获取消息并消费。

3.5.2 内部属性

1. 主题名称：当前消息所属的主题的名称。
2. 消息队列：实际存储当前消息的队列。
3. 消息位点：当前消息存储在队列中的位置（Offset）。
4. 消息 ID：消息的唯一标识，集群内每条消息的 ID 全局唯一，自动生成的 32 位字符串。
5. 消息类型：当前消息的类型，分为 Normal（普通消息）、FIFO（顺序消息）、Delay（定时消息）、Transaction（事务消息）。
6. 消息负载：业务消息的实际报文数据，由生产者负责序列化编码，按照二进制字节传输。
7. 消息发送时间戳：消息发送时，生产者客户端系统的本地毫秒级时间戳。
8. 消息保存时间戳：消息在 RocketMQ 服务端完成存储时，服务端系统的本地毫秒级时间戳。
9. 消费重试次数：消息消费失败后，RocketMQ 服务端重新投递的次数。每次重试后，重试次数加 1。
10. 其他自定义属性。
11. 索引 Key 列表（可选）：消息的索引键，可通过设置不同的 Key 区分消息和快速查找消息，由生产者定义。
12. 过滤标签 Tag（可选）：消息的过滤标签。消费者可通过 Tag 对消息进行过滤，仅接收指定标签的消息，由生产者定义。

3.5.3 使用建议

- 单条消息不建议传输超大负载。
- 由于消息的不可变性，在使用过程中对消息进行中转操作时，需要对消息重新初始化。

3.6 生产者（Producer）

生产者是 RocketMQ 系统中用来构建并传输消息到 RocketMQ 服务端的运行实体，生产者是匿名的。

在消息生产者中，可以定义如下传输行为：

1. 发送方式：生产者可通过 API 接口设置消息发送的方式，支持同步传输和异步传输。
2. 批量发送：生产者可通过 API 接口设置消息批量传输的方式。例如，批量发送的消息条数或消息大小。
3. 事务行为：Apache RocketMQ 支持事务消息，对于事务消息需要生产者配合进行事务检查

等行为保障事务的最终一致性。

生产者和主题的关系为**多对多关系**，即同一个生产者可以向多个主题发送消息，对于平台类场景如果需要发送消息到多个主题，并不需要创建多个生产者；同一个主题也可以接收多个生产者的消息，以此可以实现生产者性能的水平扩展和容灾。

3.6.1 模型关系：生产者

在整个 RocketMQ 领域模型中，生产者所处的位置如下（橙色区域）：

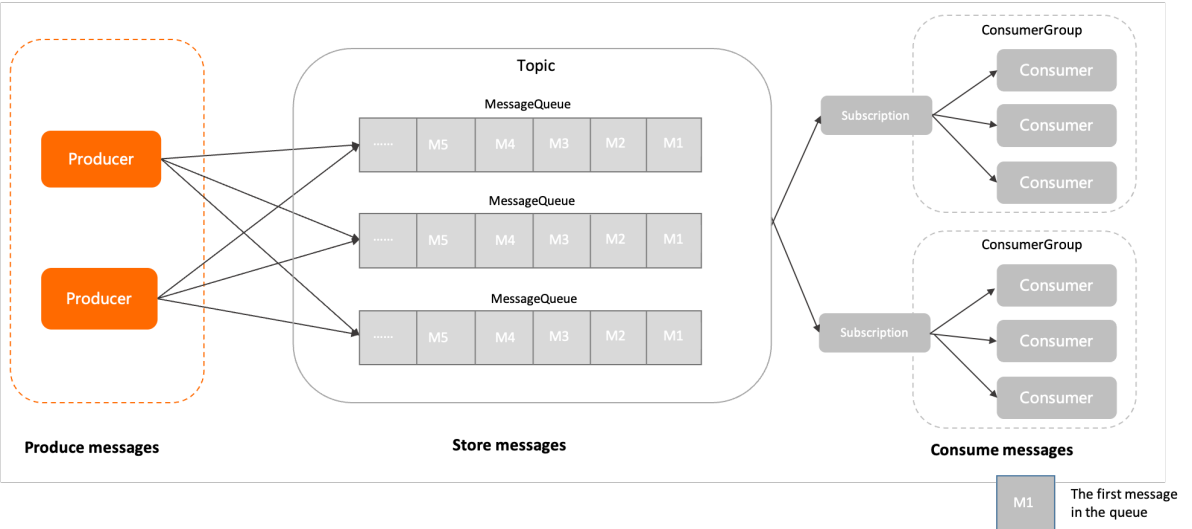


图 3.10: 模型关系：生产者

消息由生产者初始化并发送到 RocketMQ 服务端。

3.6.2 内部属性

1. 生产者客户端 ID：生产者客户端的标识，用于区分不同的生产者。集群内全局唯一。
2. 通信参数：接入点信息（必选，连接服务端的接入地址，用于识别服务端集群）、身份认证信息（可选）、请求超时时间（可选）。
3. 发送重试策略：生产者在消息发送失败时的重试策略。
4. 预绑定主题列表：生产者需要将消息发送到的目标主题列表。服务端会在生产者初始化时根据预绑定主题列表，检查目标主题的访问权限和合法性，而不需要等到应用启动后再检查。事务消息场景下，生产者在故障、重启恢复时，需要检查事务消息的主题中是否有未提交的事务消息，因此必须设置。

5. 事务检查器：事务消息场景下，生产者需要主动实现事务检查器的接口，以保证异常场景下事务的最终一致性。

3.6.3 使用建议

- 对于生产者的创建和初始化，建议遵循够用即可、最大化复用原则，如果有需要发送消息到多个主题的场景，无需为每个主题都创建一个生产者。
- 生产者是可以重复利用的底层资源，类似数据库的连接池中的连接一样，不建议频繁创建和销毁。

3.7 消费者分组 (ConsumerGroup)

消费者分组是 RocketMQ 系统中承载多个消费行为一致的消费者的负载均衡分组，并可通过消费者分组内初始化多个消费者实现消费性能的水平扩展以及高可用容灾。和消费者不同，消费者分组并不是运行实体，而是一个**逻辑资源**。

同一分组下的多个消费者将按照分组内统一的消费行为和负载均衡策略消费消息，包含以下消费行为：

1. 订阅关系：RocketMQ 以消费者分组的粒度管理订阅关系。
2. 投递顺序性：RocketMQ 的服务端将消息投递给消费者消费时，支持顺序投递和并发投递，投递方式在消费者分组中统一配置。
3. 消费重试策略：消费者消费消息失败时的重试策略，包括重试次数、死信队列设置等。

3.7.1 模型关系：消费者分组

在整个 RocketMQ 领域模型中，消费者分组所处的位置如下（橙色区域）：

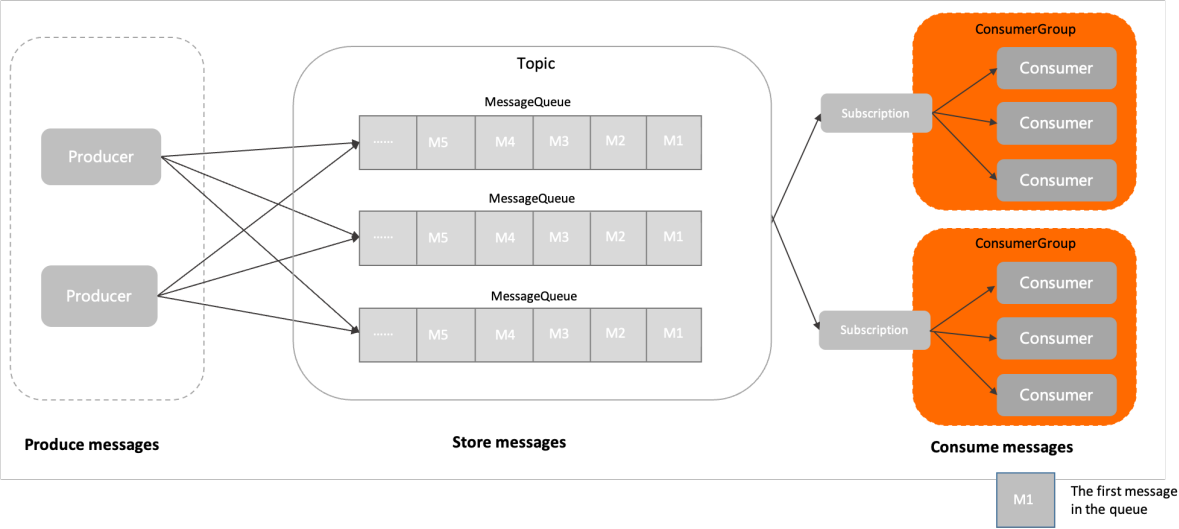


图 3.11: 模型关系：消费者分组

消费者按照指定的订阅关系从 RocketMQ 服务端中获取消息并消费。

3.7.2 内部属性

- 1. 消费者分组名称：消费者分组的名称，用于区分不同的消费者分组。集群内全局唯一。
- 2. 订阅关系：当前消费者分组关联的订阅关系集合。包括消费者订阅的主题，以及消息的过滤规则等。订阅关系由消费者动态注册到消费者分组中，RocketMQ 服务端会持久化订阅关系并匹配消息的消费进度。
- 3. 投递顺序性：消费者消费消息时，RocketMQ 向消费者客户端投递消息的顺序（支持顺序投递和并发投递）。
- 4. 消费重试策略：消费者消费消息失败时，系统的重试策略（将特定消息投递给消费者重新消费）。包括最大重试次数（超过则放入死信队列）、重试间隔。

3.7.3 使用建议

- 按照业务合理拆分分组。
- 消费者分组管理尽量避免自动化机制，请勿随意进行增、删、改、查操作。

3.8 消费者（Consumer）

消费者是 RocketMQ 中用来接收并处理消息的运行实体。消费者通常被集成在业务系统中，从 RocketMQ 服务端获取消息，并将消息转化成业务可理解的信息，供业务逻辑处理。

在消息消费端，可以定义如下传输行为：

1. 消费者身份：消费者必须关联一个指定的消费者分组，以获取分组内统一定义的行为配置和消费状态。
2. 消费者类型：RocketMQ 面向不同的开发场景提供了多样的消费者类型，包括 PushConsumer 类型、SimpleConsumer 类型、PullConsumer 类型（仅推荐流处理场景使用）等。
3. 消费者本地运行配置：消费者根据不同的消费者类型，控制消费者客户端本地的运行配置。例如消费者客户端的线程数，消费并发度等，实现不同的传输效果。

3.8.1 模型关系：消费者

在整个 RocketMQ 领域模型中，消费者所处的位置如下（橙色区域）：

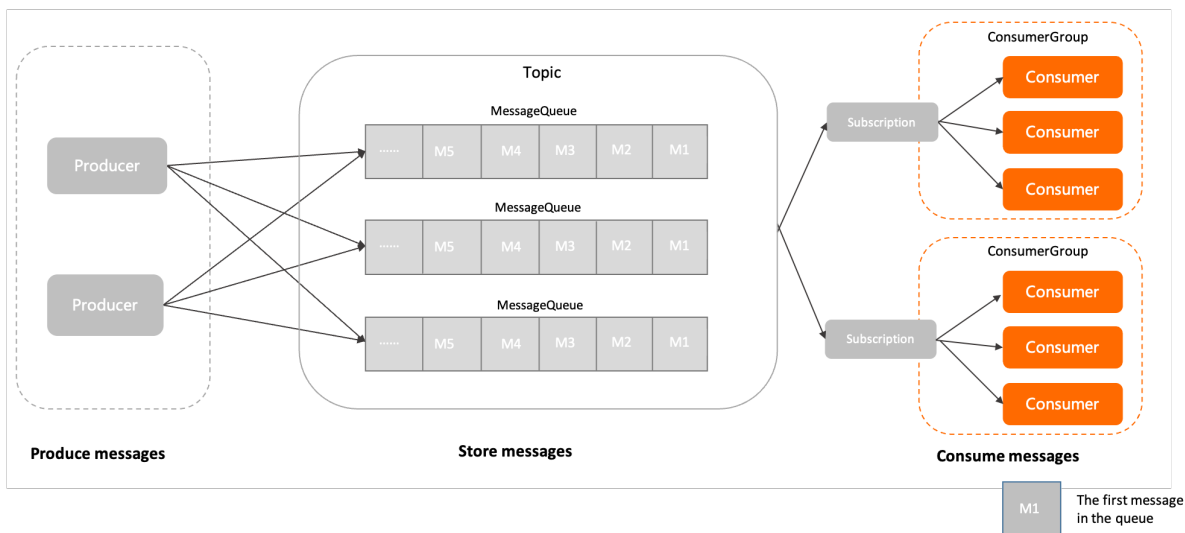


图 3.12: 模型关系：消费者

消费者按照指定的订阅关系从 RocketMQ 服务端中获取消息并消费。

3.8.2 内部属性

1. 客户端 ID：消费者客户端的标识，用于区分不同的消费者。集群内全局唯一。
2. 消费者分组名称：当前消费者关联的消费者分组名称，消费者必须关联到指定的消费者分组，通过消费者分组获取消费行为。
3. 预绑定订阅关系列表：指定消费者的订阅关系列表。RocketMQ 服务端可在消费者初始化阶段，根据预绑定的订阅关系列表对目标主题进行权限及合法性校验，无需等到应用启动后才能校验。

4. 通信参数：接入点信息（必选）、身份认证信息（可选）、请求超时时间（可选）。
5. 消费监听器：获得 RocketMQ 服务端的消息后，消费者调用消息消费逻辑的监听器。

3.8.3 使用建议

- 消费者是可以重复利用的底层资源，类似数据库的连接池中的连接一样，不建议频繁创建和销毁。
- 由于消费者在通信协议层面支持非阻塞传输模式，大部分场景下，单台主机内的一个消费分组只需要初始化一个唯一的消费者即可。

3.9 订阅关系 (Subscription)

订阅关系是 RocketMQ 系统中消费者获取消息、处理消息的规则和状态配置。订阅关系由消费者分组动态注册到服务端系统，并在后续的消息传输中按照订阅关系定义的过滤规则进行消息匹配和消费进度维护。

通过配置订阅关系，可控制如下传输行为：

1. 消息过滤规则：用于控制消费者在消费消息时，选择主题内的哪些消息进行消费，设置消费过滤规则可以高效地过滤消费者需要的消息集合，灵活根据不同的业务场景设置不同的消息接收范围。
2. 消费状态：Apache RocketMQ 服务端默认提供订阅关系持久化的能力，即消费者分组在服务端注册订阅关系后，当消费者离线并再次上线后，可以获取离线前的消费进度并继续消费。

3.9.1 模型关系：订阅关系

在整个 RocketMQ 领域模型中，订阅关系所处的位置如下（橙色区域）：

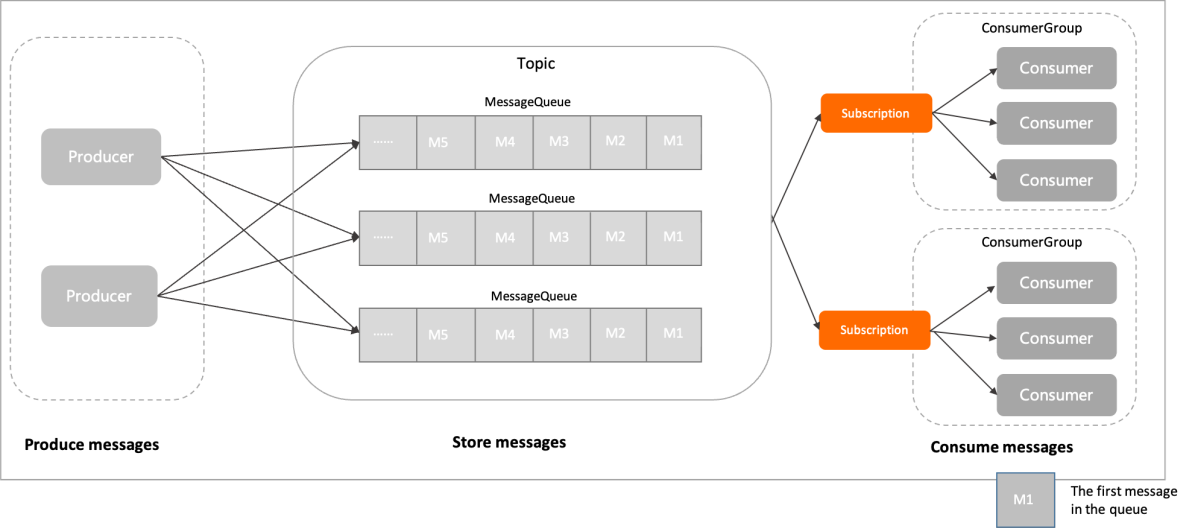


图 3.13: 模型关系：订阅关系

消费者按照指定的订阅关系从 RocketMQ 服务端中获取消息并消费。

3.9.2 订阅关系判断原则

RocketMQ 的订阅关系按照消费者分组和主题粒度设计，因此，一个订阅关系指的是指定某个消费者分组对于某个主题的订阅。

判断原则如下：

不同消费者分组对于同一个主题的订阅

不同消费者分组对于同一个主题的订阅是相互独立的。如图：

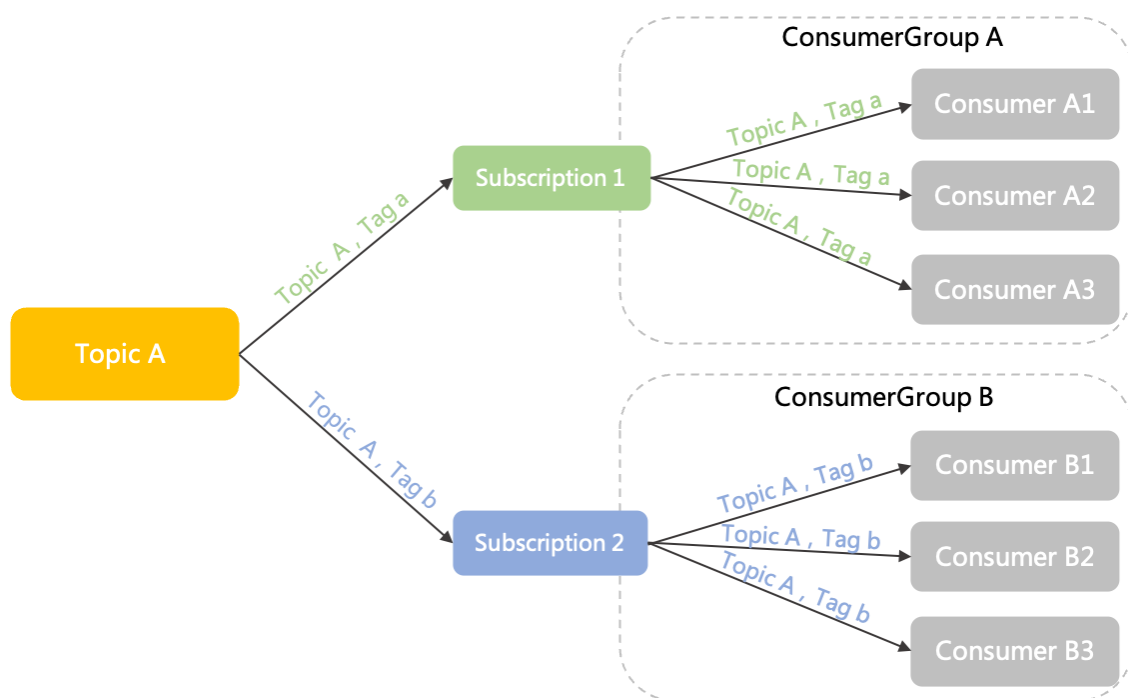


图 3.14: 不同消费者分组对于同一个主题的订阅

消费者分组 Group A 和消费者分组 Group B 分别以不同的订阅关系订阅了同一个主题 Topic A，这两个订阅关系互相独立，可以各自定义，不受影响。

同一个消费者分组对于不同主题的订阅

同一个消费者分组对于不同主题的订阅也是相互独立的。如图：

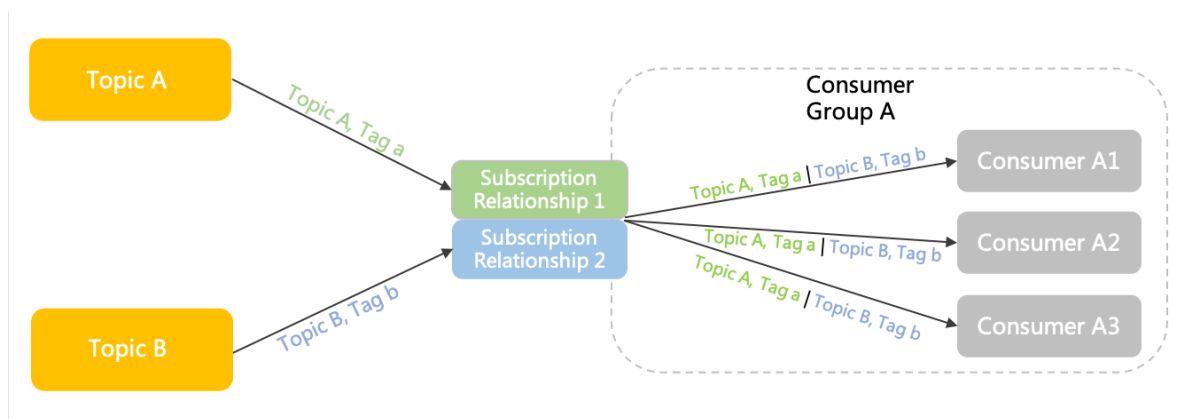


图 3.15: 同一个消费者分组对于不同主题的订阅

消费者分组 Group A 订阅了两个主题 Topic A 和 Topic B，对于 Group A 中的消费者来说，

订阅的 Topic A 为一个订阅关系，订阅的 Topic B 为另外一个订阅关系，且这两个订阅关系互相独立，可以各自定义，不受影响。

3.9.3 内部属性

1. 过滤类型：消息过滤规则的类型，分为 TAG 过滤（对 Tag 字符串进行全文过滤匹配）和 SQL92 过滤（按照 SQL 语法对消息属性进行过滤匹配）。
2. 过滤表达式：自定义的过滤规则表达式。

3.9.4 使用建议

- 不要频繁修改订阅关系，这样可能会导致客户端一直处于负载均衡调整和变更的过程，从而影响消息接收。

第四章 Springboot 快速集成 RocketMQ

资料来源:

rocketmq-spring 的 Github 地址

CSDN 上的一篇优质介绍文章

4.1 添加 Maven 依赖

作者使用的版本是 2.2.2, 也可以直接看rocketmq-spring-boot-starter 的 Maven 仓库来获得最新版本。

Listing 4.1: Maven 依赖

```
1 <dependency>
2   <groupId>org.apache.rocketmq</groupId>
3   <artifactId>rocketmq-spring-boot-starter</artifactId>
4   <version>2.2.2</version>
5 </dependency>
```

4.2 添加配置文件属性

Listing 4.2: 配置文件 application.yml

```
1 rocketmq:
2   name-server: 192.168.146.132:9876 # 名称服务访问地址
3   producer:
4     group: TEST_GROUP # 必须指定group
5     send-message-timeout: 3000 # 消息发送超时时长, 默认3s
```

```
6 retry-times-when-send-failed: 3 # 同步发送消息失败重试次数, 默认2
7 retry-times-when-send-async-failed: 3 # 异步发送消息失败重试次数, 默认2
```

4.3 生产者

4.3.1 普通消息

需要先注入 RocketMQTemplate 的 Bean, 用于进行消息的发送; topic 用于指定发送消息到的主题, sendMessage 为发送的具体消息; 可使用 topic:tag 的格式附带消息的 Tag; 可使用 setHeader 方法来设置消息的 key。

同步发送

阻塞当前线程, 等待 broker 响应发送结果。

Listing 4.3: 普通消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncSend(Object sendMessage,String topic,String tag,String
  key) {
4     return rocketMQTemplate.syncSend(topic+": "+tag, MessageBuilder.withPayload(
      sendMessage).setHeader(RocketMQHeaders.KEYS, key).build());
5 }
```

Listing 4.4: 同步发送方式简化版

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncSend(Object sendMessage,String topic) {
4     return rocketMQTemplate.syncSend(topic, MessageBuilder.withPayload(
      sendMessage).build());
5 }
```

异步发送

通过线程池执行发送到 broker 的消息任务，执行完后回调：在 SendCallback 中可处理相关成功失败时的逻辑。

Listing 4.5: 普通消息异步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public void asyncSend(Object sendMessage, String topic, String tag, String
  key) {
4     rocketMQTemplate.asyncSend(topic + ":" + tag, MessageBuilder.withPayload(
      sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(), new SendCallback
        () {
5         @Override
6         public void onSuccess(SendResult sendResult) {
7             //发送异步消息成功后的处理...
8         }
9
10        @Override
11        public void onException(Throwable throwable) {
12            //发送异步消息失败后的处理...
13        }
14    });
15 }
```

单向发送

只负责发送消息，不等待应答，不关心发送结果。

Listing 4.6: 普通消息单向发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
```

```
3 public void oneWaySend(Object sendMessage,String topic,String tag,String key)
  {
4   rocketMQTemplate.sendOneWay(topic+": "+tag, MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build());
5 }
```

4.3.2 顺序消息

topic 必须为顺序类型的主题，不允许将消息放到不同类型的主题。hashkey 用于确定发送到同一个主题中的哪个队列。要求顺序消费的多个消息必须使用同一个 hashkey 以保证进入同一个队列。

同步发送

Listing 4.7: 顺序消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncOrderlySend(Object sendMessage,String topic,String tag,
  String key,String hashkey) {
4   return rocketMQTemplate.syncSendOrderly(topic+": "+tag,MessageBuilder.
    withPayload(sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),
    hashkey);
5 }
```

异步发送

Listing 4.8: 顺序消息异步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public void asyncOrderlySend(Object sendMessage,String topic,String tag,
  String key,String hashkey) {
```

```
4    rocketMQTemplate.asyncSendOrderly(topic+":"+tag,MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),hashkey,new
    SendCallback() {
5        @Override
6        public void onSuccess(SendResult sendResult) {
7            System.out.println("发送异步消息成功" + JSON.toJSONString(sendResult));
8            //发送异步消息成功后的处理...
9        }
10
11        @Override
12        public void onException(Throwable throwable) {
13            System.out.println("发送异步消息失败" + JSON.toJSONString(throwable));
14            //发送异步消息失败后的处理...
15        }
16    });
17 }
```

4.3.3 延迟消息

延迟消息就是普通消息发送的后面加上一个 timeout 属性和 delayLevel 属性。timeout 是消息发送超时时长，为默认 3s。delayLevel 属性分为 18 个等级，分别为：1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h。

同步发送

Listing 4.9: 延迟消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncDelaySend(Object sendMessage,String topic,String tag,
    String key,long timeout, int delayLevel) {
4     return rocketMQTemplate.syncSend(topic+":"+tag,MessageBuilder.withPayload(
        sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),timeout,delayLevel
```

```
);  
}
```

异步发送

Listing 4.10: 延迟消息异步发送方式

```
1 @Resource  
2 private RocketMQTemplate rocketMQTemplate;  
3 public void asyncDelaySend(Object sendMessage, String topic, String tag,  
4 String key, long timeout, int delayLevel) {  
5     rocketMQTemplate.asyncSend(topic + ":" + tag, MessageBuilder.withPayload(  
6         sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(), new SendCallback  
7         () {  
8             @Override  
9             public void onSuccess(SendResult sendResult) {  
10                 System.out.println("发送异步消息成功" + JSON.toJSONString(sendResult));  
11                 //发送异步消息成功后的处理...  
12             }  
13  
14             @Override  
15             public void onException(Throwable throwable) {  
16                 System.out.println("发送异步消息失败" + JSON.toJSONString(throwable));  
17                 //发送异步消息失败后的处理...  
18             }  
19         }, timeout, delayLevel);  
20 }
```

4.3.4 批量消息

批量消息的发送方式即将原先普通消息中的 `sendMessage` 换成 `Collection<Message>` 的对象即可。

同步发送

Listing 4.11: 批量消息同步发送方式（简化版）

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public void syncBatchSend(List<Object> sendMessage, String topic) {
4     ArrayList<Message<Object>> list = new ArrayList<>();
5     for (Object message : sendMessage) {
6         list.add(MessageBuilder.withPayload(message).build());
7     }
8     rocketMQTemplate.syncSend(topic, list);
9 }
```

复杂版本和异步版本省略。

4.3.5 事务消息

暂不考虑。

4.4 消费者

在 RocketMQMessageListener 注解中，topic 用于指定接收的主题，selectorType 用于指定过滤的方式（默认为 Tag，可更改为 SQL92），selectorExpression 用于控制消息的过滤规则（Tag 模式下，* 代表全部 Tag），consumeMode 用于指定是即时接收消息还是顺序接收消息。（其他注解属性请自行了解）

4.4.1 Push 模式

消费消息仅通过消费监听器处理业务并返回消费结果。消息的获取、消费状态提交以及消费重试都通过 Apache RocketMQ 的客户端 SDK 完成。

Listing 4.12: Push 模式消费者

```
1 @Service
```

```
2 @RocketMQMessageListener(topic = "TEST_TOPIC", selectorExpression = "test",
  consumerGroup = "Group_One")
3 public class ConsumerSend implements RocketMQListener<User> {
4     // 监听到消息就会执行此方法
5     @Override
6     public void onMessage(Object message) {
7         //此处为处理消息的方法
8     }
9 }
```

4.4.2 Pull 模式

如果使用 Pull 模型，需要补充配置文件属性。（注意：如果不使用，就不要添加该属性。）

Listing 4.13: lite pull consumer 所需配置属性

```
1 rocketmq.pull-consumer.group=Group_One
2 rocketmq.pull-consumer.topic=TEST_TOPIC
```

Pull 模型通过调用 RocketMQTemplate 的 receive 方法实现。

Listing 4.14: Pull 模式消费者

```
1 public void Pull() throws Exception {
2     List<Object> messages = rocketMQTemplate.receive(Object.class);
3     //此处处理接收到的message集合
4 }
```