

【RocketMQ】

【主要内容】

1. RocketMQ 简介
2. RocketMQ 概念
3. RocketMQ 安装
4. RocketMQ 快速入门
5. RocketMQ 消息模式
6. RocketMQ 重试机制
7. RocketMQ 重复消费问题
8. RocketMQ 集成 SpringBoot

【学习目标】

知识点	要求
RocketMQ 简介	了解
RocketMQ 概念	了解
RocketMQ 安装	掌握
RocketMQ 快速入门	掌握
RocketMQ 消息模式	重点
RocketMQ 重试机制	重点
RocketMQ 重复消费问题	重点
RocketMQ 集成 SpringBoot	重点

1. RocketMQ 简介

MQ===Message Queue

官网: <http://rocketmq.apache.org/>

Apache RocketMQ

Apache RocketMQ™ is a unified messaging engine, lightweight data processing platform.

Latest release v4.9.3



16,751



9,392

🔗 Getting Started

RocketMQ 是阿里巴巴 2016 年 MQ 中间件，使用 Java 语言开发，RocketMQ 是一款开源的**分布式消息系统**，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。同时，广泛应用于多个领域，包括异步通信解耦、企业解决方案、金融支付、电信、电子商务、快递物流、广告营销、社交、即时通信、移动应用、手游、视频、物联网、车联网等。

具有以下特点：

1. 能够保证严格的消息顺序
2. 提供丰富的消息拉取模式
3. 高效的订阅者水平扩展能力
4. 实时的消息订阅机制
5. 亿级消息堆积能力

2. 为什么要使用 MQ

- 1, 要做到系统解耦，当新的模块进来时，可以做到代码改动最小； **能够解耦**
- 2, 设置流程缓冲池，可以让后端系统按自身**吞吐**能力进行消费，不被冲垮； **能够削峰，限流**
- 3, 强弱依赖梳理能把非关键调用链路的操作异步化并提升整体系统的吞吐能力；**能够异步**

Mq 的作用 削峰限流 异步 解耦合

2.1 定义

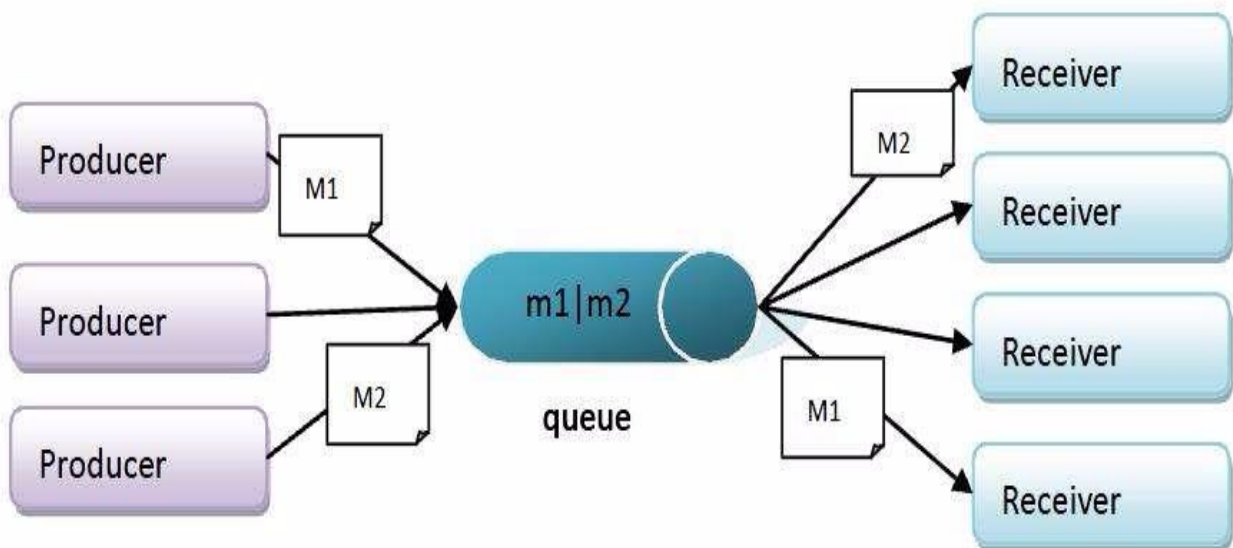
中间件（缓存中间件 redis memcache 数据库中间件 mycat canal 消息中间件 mq）面向消息的**中间件**(message-oriented middleware) MOM 能够很好的解决以上的问题。

是指利用**高效可靠的消息传递机制进行与平台无关（跨平台）的数据交流**，并基于数据通信来进行分布式系统的集成。

通过提供**消息传递和消息排队模型**在分布式环境下提供应用解耦，弹性伸缩，冗余存储，流量削峰，异步通信，数据同步等

大致流程

发送者把消息发给消息服务器[MQ]，消息服务器把消息存放在若干**队列/主题**中，在合适的时候，消息服务器会把消息转发给接受者。在这个过程中，发送和接受是异步的，也就是发送无需等待，发送者和接受者的生命周期也没有必然关系在发布 pub/订阅 sub 模式下，也可以完成一对多的通信，可以让一个消息有多个接受者[微信订阅号就是这样的]



2.2 特点

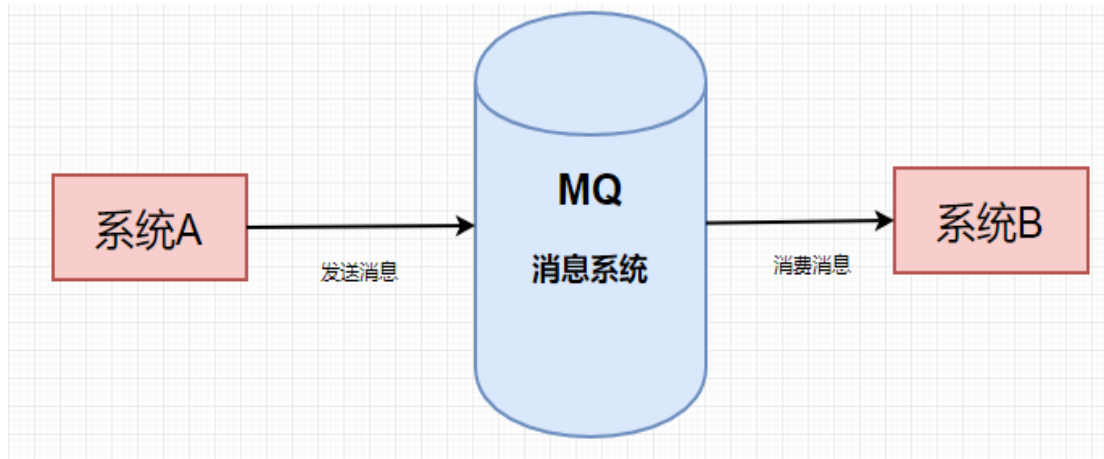
2.2.1 异步处理模式

消息发送者可以发送一个消息而无需等待响应。消息发送者把消息发送到一条虚拟的通道（主题或队列）上；

消息接收者则订阅或监听该通道。一条信息可能最终转发给一个或多个消息接收者，这些接收者都无需对消息发送者做出回应。整个过程都是异步的。

案例：

也就是说，一个系统和另一个系统间进行通信的时候，假如系统 A 希望发送一个消息给系统 B，让它去处理，但是系统 A 不关注系统 B 到底怎么处理或者有没有处理好，所以系统 A 把消息发送给 MQ，然后就不管这条消息的“死活”了，接着系统 B 从 MQ 里面消费出来处理即可。至于怎么处理，是否处理完毕，什么时候处理，都是系统 B 的事，与系统 A 无关。



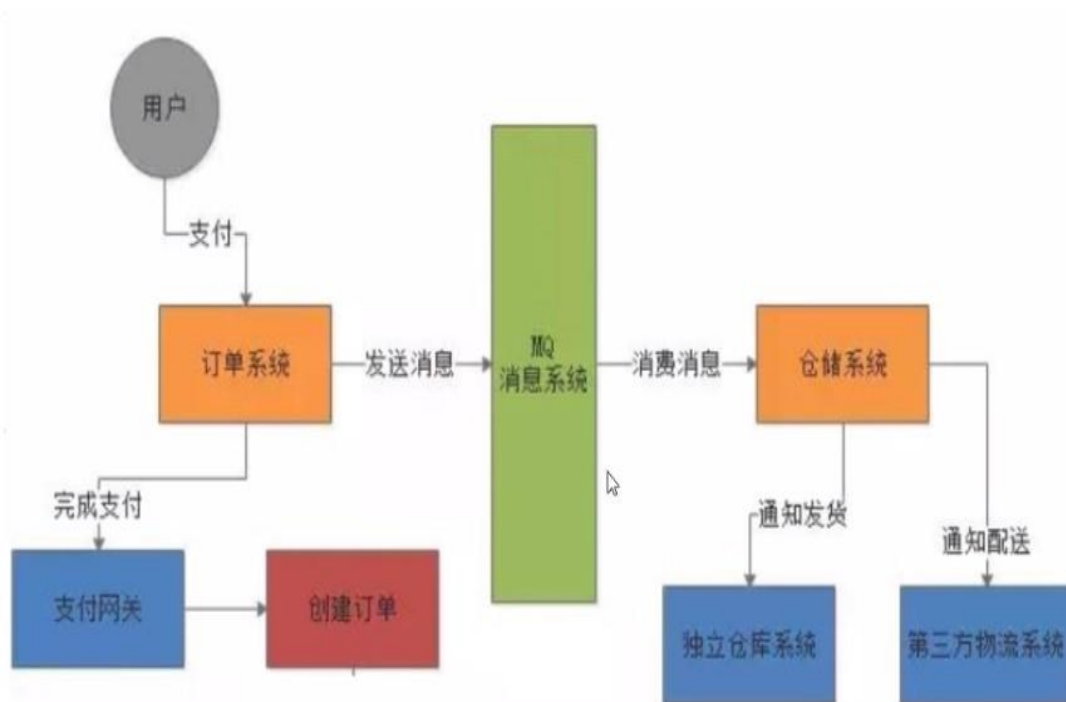
这样的一种通信方式，就是所谓的“异步”通信方式，对于系统 A 来说，只要把消息发给 MQ，然后系统 B 就会异步地去进行处理了，系统 A 不能“同步”的等待系统 B 处理完。这样的好处是什么呢？解耦

2.2.2 应用系统的解耦

发送者和接收者不必了解对方，只需要确认消息

发送者和接收者不必同时在线

2.2.3 现实中的业务



3. 各个 MQ 产品的比较

特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低；管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

4. RocketMQ 重要概念【重点】

Producer：消息的发送者，生产者；举例：发件人

Consumer：消息接收者，消费者；举例：收件人

Broker：暂存和传输消息的通道；举例：快递

NameServer：管理 Broker；举例：各个快递公司的管理机构 相当于 broker 的注册中心，保留了 broker 的信息

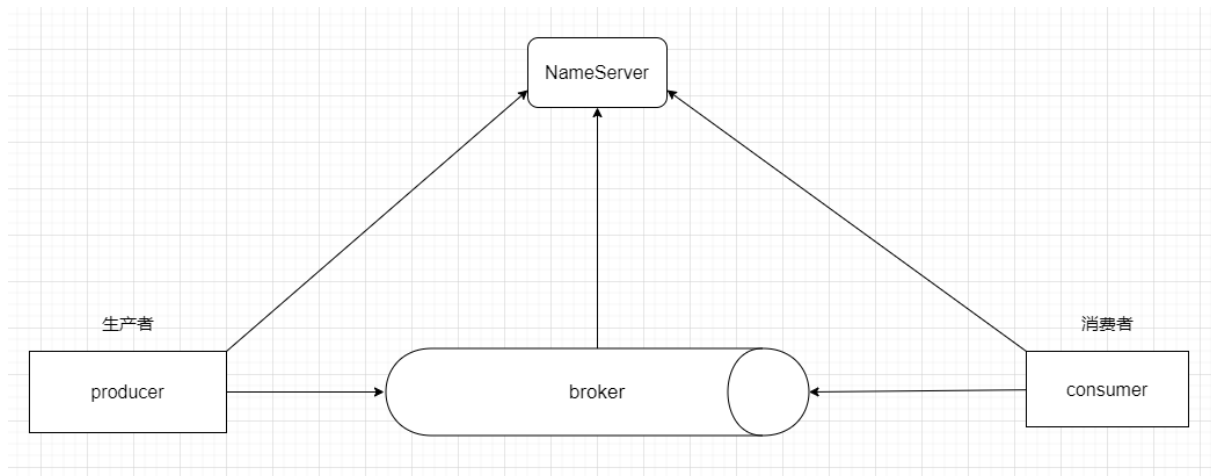
Queue：队列，消息存放的位置，一个 Broker 中可以有多个队列

Topic：主题，消息的分类

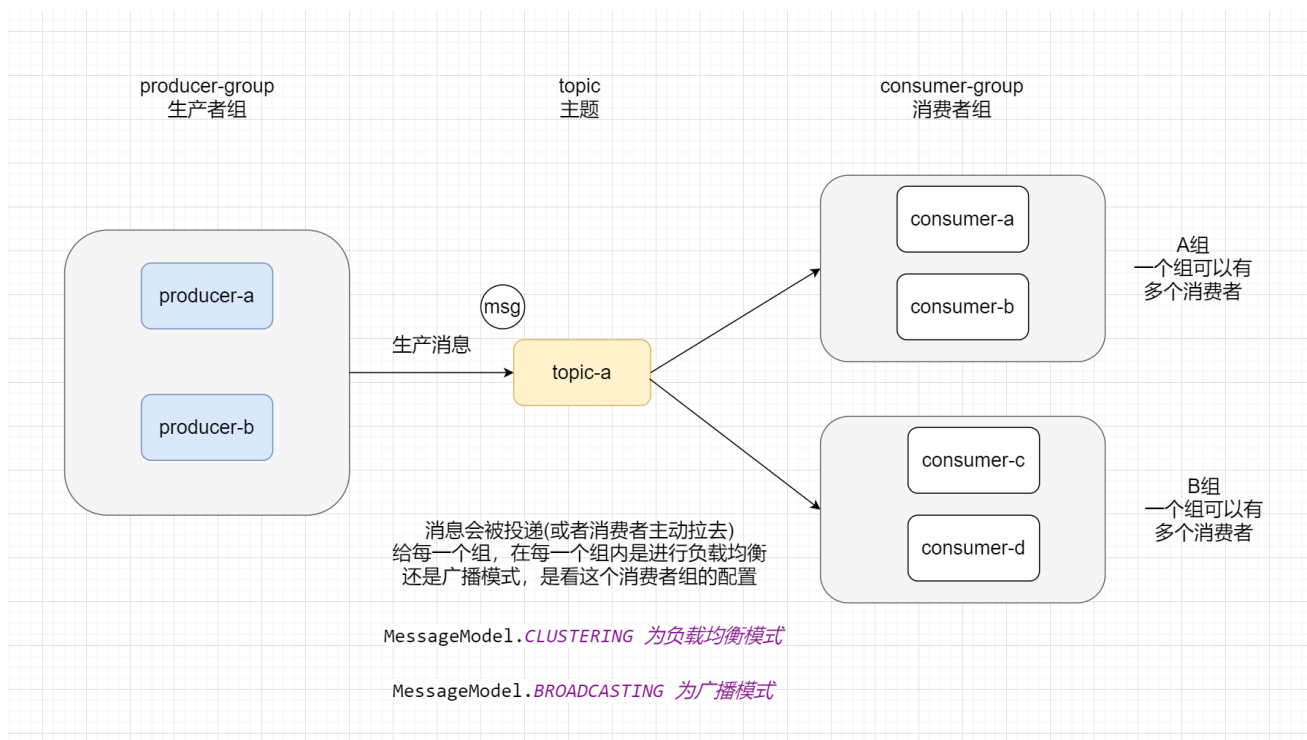
ProducerGroup：生产者组

ConsumerGroup: 消费者组, 多个消费者组可以同时消费一个主题的消息

消息发送的流程是, Producer 询问 NameServer, NameServer 分配一个 broker 然后 Consumer 也要询问 NameServer, 得到一个具体的 broker, 然后消费消息



5. 生产和消费理解【重点】



6. RocketMQ 安装

了解了 mq 的基本概念和角色以后，我们开始安装 rocketmq，建议在 linux 上

6.1 下载 RocketMQ

下载地址: <https://rocketmq.apache.org/downloading/releases/>

注意选择版本，这里我们选择 4.9.2 的版本，后面使用 alibaba 时对应

USER GUIDE

- Why RocketMQ
- Quick Start
- Simple Example
- Order Example
- Broadcasting Example
- Schedule Example
- Batch Example
- Filter Example
- Logappender Example
- OpenMessaging Example
- Transaction Example
- FAQ

RELEASE NOTES

[Download](#)

- [Release Notes](#)
- Source: [rocketmq-all-4.9.3-source-release.zip](#) [PGP] [SHA512]
- Binary: [rocketmq-all-4.9.3-bin-release.zip](#) [PGP] [SHA512]

4.9.2 release

- Released Oct 26, 2021
- [Release Notes](#)
- Source: [rocketmq-all-4.9.2-source-release.zip](#) [PGP] [SHA512]
- Binary: [rocketmq-all-4.9.2-bin-release.zip](#) [PGP] [SHA512]

下载地址:

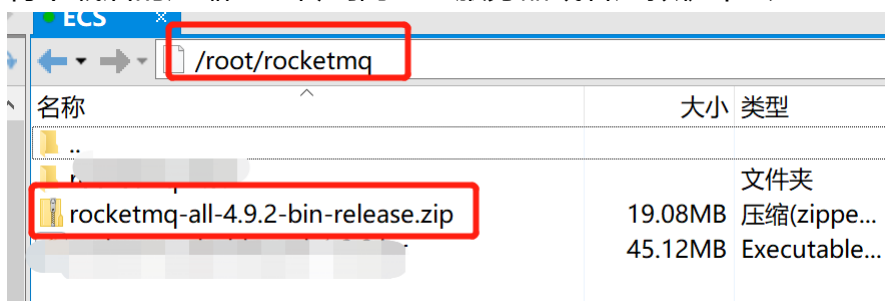
<https://archive.apache.org/dist/rocketmq/4.9.2/rocketmq-all-4.9.2-bin-release.zip>

6.2 上传服务器

在 root 目录下创建文件夹

```
mkdir rocketmq
```

将下载后的压缩包上传到阿里云服务器或者虚拟机中去



6.3 解压

```
unzip rocketmq-all-4.9.2-bin-release.zip
```

如果你的服务器没有 unzip 命令，则下载安装一个

```
yum install unzip
```

目录分析

```
22 13:56 benchmark
 9 11:07 bin
 9 13:10 conf
22 13:56 lib
22 13:41 LICENSE
 9 11:10 logs
 9 13:18 nohup.out
22 13:41 NOTICE
22 13:41 README.md
```

Benchmark：包含一些性能测试的脚本；

Bin：可执行文件目录；

Conf：配置文件目录；

Lib：第三方依赖；

LICENSE：授权信息；

NOTICE：版本公告；

6.4 配置环境变量

```
vim /etc/profile
```

在文件末尾添加

```
export NAMESRV_ADDR=阿里云公网 IP:9876
```

6.5 修改 nameServer 的运行脚本

进入 bin 目录下，修改 runserver.sh 文件，将 71 行和 76 行的 Xms 和 Xmx 等改小一点

```
vim runserver.sh
```

```

64
65 choose_gc_options()
66 {
67     # Example of JAVA_MAJOR_VERSION value : '1', '9', '10', '11', ...
68     # '1' means releases before Java 9
69     JAVA_MAJOR_VERSION=$(("$JAVA" -version 2>&1 | sed -r -n 's/.* version "([0-9]*).*/\1/p')
70     if [ -z "$JAVA_MAJOR_VERSION" ] || [ "$JAVA_MAJOR_VERSION" -lt "9" ]; then
71         JAVA_OPT="$JAVA_OPT" -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
72         JAVA_OPT="$JAVA_OPT" -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSInitiatingOccupancyFraction=7
73         -XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8 -XX:-UseParNewGC"
74         JAVA_OPT="$JAVA_OPT" -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps"
75     else
76         JAVA_OPT="$JAVA_OPT" -server -Xms256m -Xmx256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
77         JAVA_OPT="$JAVA_OPT" -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerMB=0"
78         JAVA_OPT="$JAVA_OPT" -Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,tags:filecount=5,filesize=30M"
79     fi

```

保存退出

6.6 修改 broker 的运行脚本

进入 bin 目录下，修改 runbroker.sh 文件，修改 67 行

```

64
65 choose_gc_log_directory
66
67 JAVA_OPT="$JAVA_OPT" -server -Xms256m -Xmx256m"
68 JAVA_OPT="$JAVA_OPT" -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerMB=0"
69 JAVA_OPT="$JAVA_OPT" -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_broker_gc_%p_%t.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintAdaptiveSizePolicy"
70 JAVA_OPT="$JAVA_OPT" -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=30m"
71 JAVA_OPT="$JAVA_OPT" -XX:-OmitStackTraceInFastThrow"
72 JAVA_OPT="$JAVA_OPT" -XX:+AlwaysPreTouch"
73 JAVA_OPT="$JAVA_OPT" -XX:MaxDirectMemorySize=15g"
74 JAVA_OPT="$JAVA_OPT" -XX:-UseLargePages -XX:-UseBiasedLocking"
75 JAVA_OPT="$JAVA_OPT" -Djava.ext.dirs=${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib:${JAVA_HOME}/lib/ext"
76 #JAVA_OPT="$JAVA_OPT" -Xdebug -Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
77 JAVA_OPT="$JAVA_OPT" ${JAVA_OPT_EXT}"

```

保存退出

6.7 修改 broker 的配置文件

进入 conf 目录下，修改 broker.conf 文件

```

brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
namesrvAddr=localhost:9876
autoCreateTopicEnable=true
brokerIP1=阿里云公网 IP

```

添加参数解释

namesrvAddr: nameSrv 地址 可以写 localhost 因为 nameSrv 和 broker 在一个服务器

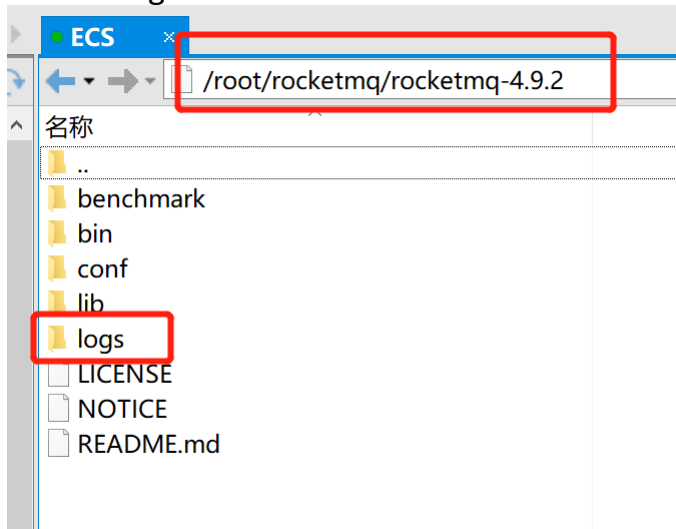
autoCreateTopicEnable: 自动创建主题，不然需要手动创建出来

brokerIP1: broker 也需要一个公网 ip，如果不指定，那么是阿里云的内网地址，我们再本地无法连接使用

6.8 启动

首先在安装目录下创建一个 logs 文件夹，用于存放日志

mkdir logs



一次运行两条命令

启动 nameSrv

```
nohup sh bin/mqnamesrv > ./logs/namesrv.log &
```

启动 broker 这里的 -c 是指定使用的配置文件

```
nohup sh bin/mqbroker -c conf/broker.conf > ./logs/broker.log &
```

查看启动结果

```
[root@cxsxjw conf]# jps
31815 rocketmq-dashboard-1.0.0.jar
4266 Jps
32766 BrokerStartup
31359 NamesrvStartup
[root@cxsxjw conf]# a
```

6.9 RocketMQ 控制台的安装 RocketMQ-Console

Rocketmq 控制台可以可视化 MQ 的消息发送!

旧版本源码是在 rocketmq-external 里的 rocketmq-console，新版本已经单独拆分成

dashboard

网址: <https://github.com/apache/rocketmq-dashboard>

下载地址:

<https://github.com/apache/rocketmq-dashboard/archive/refs/tags/rocketmq-dashboard-1.0.0.zip>

下载后解压出来, 在跟目录下执行

```
mvn clean package -Dmaven.test.skip=true
```

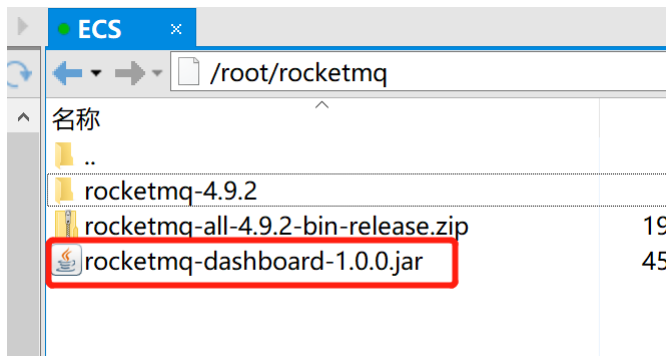
```
BASE.jar (154 kB at 238 kB/s)
Downloaded from nexus-aliyun: https://maven.aliyun.com/repository/public/org/slf4j/slf4j
Downloaded from nexus-aliyun: https://maven.aliyun.com/repository/public/org/springfram
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:58 min
[INFO] Finished at: 2022-03-19T12:00:46+08:00
[INFO] -----
```

vttools > rocketmq > rocketmq-dashboard-rocketmq-dashboard-1.0.0 > target >

名称

- classes
- generated-sources
- maven-archiver
- maven-shared-archive-resources
- maven-status
- .plxarc
- checkstyle-cachefile
- checkstyle-checker.xml
- checkstyle-result.xml
- rocketmq-dashboard-1.0.0.jar
- rocketmq-dashboard-1.0.0.jar.original

将 jar 包上传到服务器上去



然后运行

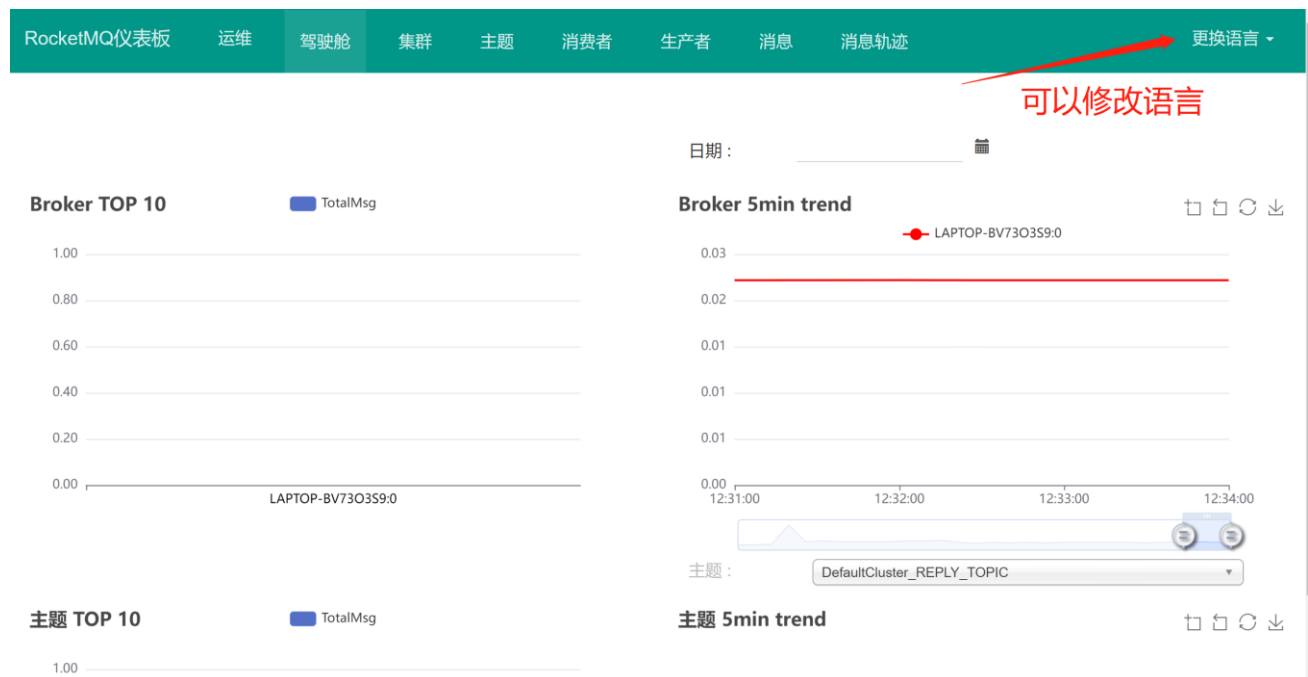
```
nohup java -jar ./rocketmq-dashboard-1.0.0.jar
rocketmq.config.namesrvAddr=127.0.0.1:9876 > ./rocketmq-4.9.3/logs/dashbo
ard.log &
```

命令拓展: --server.port 指定运行的端口

--rocketmq.config.namesrvAddr=127.0.0.1:9876 指定 namesrv 地址

访问: <http://localhost:8001>

运行访问端口是 8001, 如果从官网拉下来打包的话, 默认端口是 8080



7. RocketMQ 安装之 docker

7.1 下载 RockerMQ 需要的镜像

```
docker pull rocketmqinc/rocketmq
```

```
docker pull styletang/rocketmq-console-ng
```

7.2 启动 NameServer 服务

7.2.1 创建 NameServer 数据存储路径

```
mkdir -p /home/rocketmq/data/namesrv/logs /home/rocketmq/data/namesrv/store
```

7.2.2 启动 NameServer 容器

```
docker run -d --name rmqnamesrv -p 9876:9876 -v /home/rocketmq/data/namesrv/logs:/root/logs  
-v /home/rocketmq/data/namesrv/store:/root/store -e "MAX_POSSIBLE_HEAP=100000000"  
rocketmqinc/rocketmq sh mqnamesrv
```

7.3 启动 Broker 服务

7.3.1 创建 Broker 数据存储路径

```
mkdir -p /home/rocketmq/data/broker/logs /home/rocketmq/data/broker/store
```

7.3.2 创建 conf 配置文件目录

```
mkdir /home/rocketmq/conf
```

7.3.3 在配置文件目录下创建 broker.conf 配置文件

```
# 所属集群名称，如果节点较多可以配置多个  
brokerClusterName = DefaultCluster  
#broker 名称，master 和 slave 使用相同的名称，表明他们的主从关系  
brokerName = broker-a  
#0 表示 Master，大于 0 表示不同的 slave  
brokerId = 0  
#表示几点做消息删除动作，默认是凌晨 4 点  
deleteWhen = 04  
#在磁盘上保留消息的时长，单位是小时  
fileReservedTime = 48  
#有三个值：SYNC_MASTER，ASYNC_MASTER，SLAVE；同步和异步表示 Master 和 Slave 之间  
同步数据的机制；  
brokerRole = ASYNC_MASTER  
#刷盘策略，取值为：ASYNC_FLUSH，SYNC_FLUSH 表示同步刷盘和异步刷盘；SYNC_FLUSH  
消息写入磁盘后才返回成功状态，ASYNC_FLUSH 不需要；  
flushDiskType = ASYNC_FLUSH  
# 设置 broker 节点所在服务器的 ip 地址  
brokerIP1 = 你服务器外网 ip
```

7.3.4 启动 Broker 容器

```
docker run -d --name rmqbroker --link rmqnamesrv:namesrv -p 10911:10911 -p 10909:10909 -v /home/rocketmq/data/broker/logs:/root/logs -v /home/rocketmq/data/broker/store:/root/store -v /home/rocketmq/conf/broker.conf:/opt/rocketmq-4.4.0/conf/broker.conf --privileged=true -e "NAMESRV_ADDR=namesrv:9876" -e "MAX_POSSIBLE_HEAP=200000000" rocketmqinc/rocketmq sh mqbroker -c /opt/rocketmq-4.4.0/conf/broker.conf
```

7.4 启动控制台

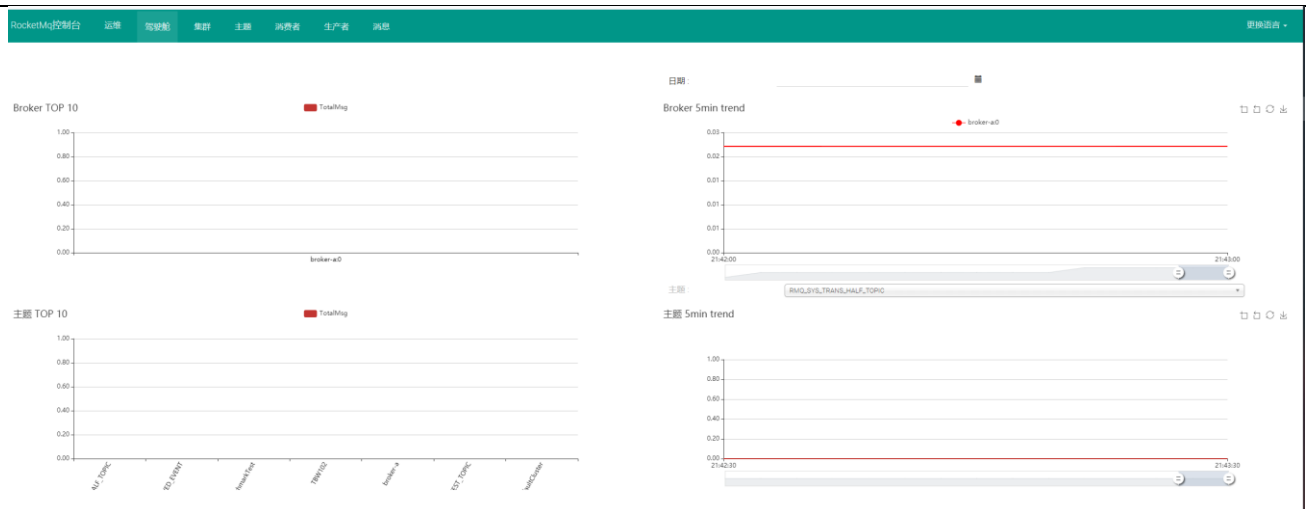
```
docker run -d --name rmqadmin -e "JAVA_OPTS=-Drocketmq.namesrv.addr=你的外网地址:9876 \ -Dcom.rocketmq.sendMessageWithVIPChannel=false \ -Duser.timezone='Asia/Shanghai'" -v /etc/localtime:/etc/localtime -p 9999:8080 styletang/rocketmq-console-ng
```

7.5 正常启动后的 docker ps

```
[root@192 conf]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED            STATUS              PORTS
cafbcb835da        rocketmqinc/rocketmq  "sh mqbroker -c /o..." 14 minutes ago    Up 14 minutes      0.0.0.0:10909->10909/tcp, 9876/tcp, 0.0.0.0:10911->10911/tcp
418cc78b8142        styletang/rocketmq-console-ng  "sh -c 'java $JAVA..." 29 minutes ago    Up 29 minutes      0.0.0.0:9999->8080/tcp
acf2ba259719        rocketmqinc/rocketmq  "sh mqnamesrv"          35 minutes ago    Up 35 minutes      10909/tcp, 0.0.0.0:9876->9876/tcp, 10911/tcp
```

7.6 访问控制台

<http://你的服务器外网 ip:9999/>



8. RocketMQ 快速入门

RocketMQ 提供了发送多种发送消息的模式，例如同步消息，异步消息，顺序消息，延迟消息，事务消息等，我们一一学习

8.1 消息发送和监听的流程

我们先搞清楚消息发送和监听的流程，然后我们在开始敲代码

8.1.1 消息生产者

1. 创建消息生产者 producer，并制定生产者组名
2. 指定 Nameserver 地址
3. 启动 producer
4. 创建消息对象，指定主题 Topic、Tag 和消息体等
5. 发送消息
6. 关闭生产者 producer

8.1.2 消息消费者

1. 创建消费者 consumer，制定消费者组名
2. 指定 Nameserver 地址
3. 创建监听订阅主题 Topic 和 Tag 等
4. 处理消息
5. 启动消费者 consumer

8.2 搭建 Rocketmq-demo

8.2.1 加入依赖

```
<dependencies>
  <dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.2</version>
    <!-- docker 的用下面这个版本 -->
    <version>4.4.0</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.22</version>
  </dependency>
</dependencies>
```


8.2.2 编写生产者

```
/**
 * 测试生产者
 *
 * @throws Exception
 */
@Test
public void testProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    for (int i = 0; i < 10; i++) {
        // 创建消息
        // 第一个参数: 主题的名字
        // 第二个参数: 消息内容
        Message msg = new Message("TopicTest", ("Hello RocketMQ " + i).getBytes());
        SendResult send = producer.send(msg);
        System.out.println(send);
    }
    // 关闭实例
    producer.shutdown();
}
```

8.2.3 编写消费者

```
/**
 * 测试消费者
 *
 * @throws Exception
 */
@Test
public void testConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 *表示没有过滤参数 表示这个主题的任何消息
    consumer.subscribe("TopicTest", "*");
    // 注册一个消费监听 MessageListenerConcurrently 是多线程消费, 默认 20 个线程, 可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {
            System.out.println(Thread.currentThread().getName() + "----" + msgs);
            // 返回消费的状态 如果是 CONSUME_SUCCESS 则成功, 若为 RECONSUME_LATER 则该条消息会被重回队列, 重新被投递
            // 重试的时间为 messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"
            // 也就是第一次1s 第二次5s 第三次10s .... 如果重试了18次 那么这个消息就会被终止发送给消费者
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    // 这个start 一定要写在registerMessageListener 下面
    consumer.start();
    System.in.read();
}
```

8.2.4 测试

启动生产者和消费者进行测试

9. 消费模式

MQ 的消费模式可以大致分为两种，一种是推 Push，一种是拉 Pull。

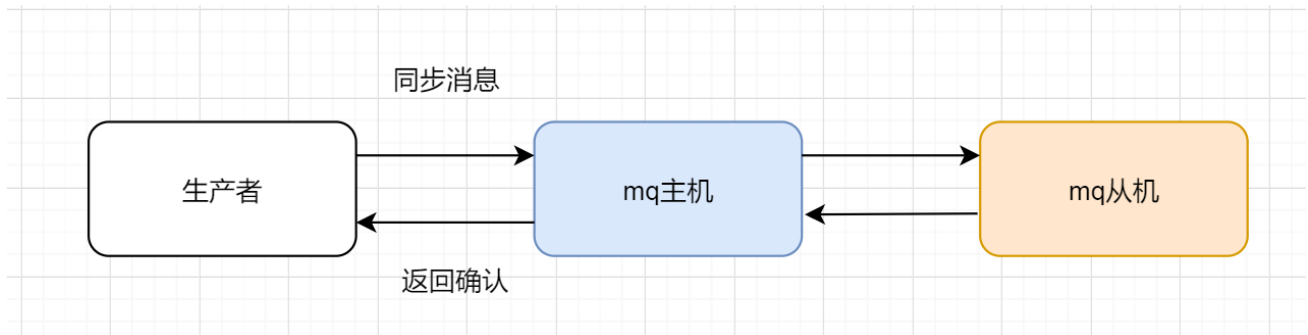
Push 是服务端【MQ】主动推送消息给客户端，优点是及时性较好，但如果客户端没有做好流控，一旦服务端推送大量消息到客户端时，就会导致客户端消息堆积甚至崩溃。

Pull 是客户端需要主动到服务端取数据，优点是客户端可以依据自己的消费能力进行消费，但拉取的频率也需要用户自己控制，拉取频繁容易造成服务端和客户端的压力，拉取间隔长又容易造成消费不及时。

Push 模式也是基于 pull 模式的，只能客户端内部封装了 api，一般场景下，上游消息生产量小或者匀速的时候，选择 push 模式。在特殊场景下，例如电商大促，抢优惠券等场景可以选择 pull 模式

10. RocketMQ 发送同步消息

上面的快速入门就是发送同步消息，发送过后会有一个返回值，也就是 mq 服务器接收到消息后返回的一个确认，这种方式非常安全，但是性能上并没有这么高，而且在 mq 集群中，也是要等到所有的从机都复制了消息以后才会返回，所以针对重要的消息可以选择这种方式



11. RocketMQ 发送异步消息

异步消息通常用在响应时间敏感的业务场景，即发送端不能容忍长时间地等待 Broker 的响应。发送完以后会有一个异步消息通知

11.1 异步消息生产者

```
@Test
public void testAsyncProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    Message msg = new Message("TopicTest", ("异步消息").getBytes());
    producer.send(msg, new SendCallback() {
        @Override
        public void onSuccess(SendResult sendResult) {
            System.out.println("发送成功");
        }
        @Override
        public void onException(Throwable e) {
            System.out.println("发送失败");
        }
    });
    System.out.println("看看谁先执行");
    // 挂起jvm 因为回调是异步的不然测试不出来
    System.in.read();
    // 关闭实例
    producer.shutdown();
}
```

11.2 异步消息消费者

```
@Test
public void testAsyncConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 *表示没有过滤参数 表示这个主题的任何消息
    consumer.subscribe("TopicTest", "*");
    // 注册一个消费监听 MessageListenerConcurrently 是并发消费
    // 默认是20 个线程一起消费, 可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {
            // 这里执行消费的代码 默认是多线程消费
            System.out.println(Thread.currentThread().getName() + "-----" + msgs);
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

12. RocketMQ 发送单向消息

这种方式主要用在不关心发送结果的场景，这种方式吞吐量很大，但是存在消息丢失的风险，例如日志信息的发送

12.1 单向消息生产者

```
@Test
public void testOnewayProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    Message msg = new Message("TopicTest", ("单向消息").getBytes());
    // 发送单向消息
    producer.sendOneway(msg);
    // 关闭实例
    producer.shutdown();
}
```

12.2 单向消息消费者

消费者和上面一样

13. RocketMQ 发送延迟消息

消息放入 mq 后，过一段时间，才会被监听到，然后消费
比如下订单业务，提交了一个订单就可以发送一个延时消息，30min 后去检查这个订单的状态，如果还是未付款就取消订单释放库存。

13.1 延迟消息生产者

```
@Test
public void testDelayProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
}
```

```
producer.start();
Message msg = new Message("TopicTest", ("延迟消息").getBytes());
// 给这个消息设定一个延迟等级
// messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"
msg.setDelayTimeLevel(3);
// 发送单向消息
producer.send(msg);
// 打印时间
System.out.println(new Date());
// 关闭实例
producer.shutdown();
}
```

13.2 延迟消息消费者

消费者和上面一样

这里注意的是 RocketMQ 不支持任意时间的延时

只支持以下几个固定的延时等级，等级 1 就对应 1s，以此类推，最高支持 2h 延迟

```
private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m
10m 20m 30m 1h 2h";
```

14. RocketMQ 发送顺序消息

消息有序指的是可以**按照消息的发送顺序来消费**(FIFO)。RocketMQ 可以严格的保证消息有序，可以分为：分区有序或者全局有序。

可能大家会有疑问，mq 不就是 FIFO 吗？

rocketMq 的 broker 的机制，导致了 rocketMq 会有这个问题

因为一个 broker 中对应了四个 queue

[powernode]状态

队列	最小位点	最大位点
MessageQueue [topic=powernode, brokerName=LAPTOP-BV73O3S9, queueId=3]	0	7
MessageQueue [topic=powernode, brokerName=LAPTOP-BV73O3S9, queueId=2]	0	4
MessageQueue [topic=powernode, brokerName=LAPTOP-BV73O3S9, queueId=1]	0	3
MessageQueue [topic=powernode, brokerName=LAPTOP-BV73O3S9, queueId=0]	0	4

顺序消费的原理解析，在默认的情况下消息发送会采取 Round Robin 轮询方式把消息发送到

不同的 queue(分区队列); 而消费消息的时候从多个 queue 上拉取消息, 这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个 queue 中, 消费的时候只从这个 queue 上依次拉取, 则就保证了顺序。当发送和消费参与的 queue 只有一个, 则是全局有序; 如果多个 queue 参与, 则为分区有序, 即相对每个 queue, 消息都是有序的。下面用订单进行分区有序的示例。一个订单的顺序流程是: 下订单、发短信通知、物流、签收。订单顺序号相同的消息会被先后发送到同一个队列中, 消费时, 同一个顺序获取到的肯定是同一个队列。

14.1 场景分析

模拟一个订单的发送流程, 创建两个订单, 发送的消息分别是

订单号 111 消息流程 下订单->物流->签收

订单号 112 消息流程 下订单->物流->拒收

14.2 创建一个订单对象

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Order {
    /**
     * 订单id
     */
    private Integer orderId;

    /**
     * 订单编号
     */
    private Integer orderNumber;

    /**
     * 订单价格
     */
    private Double price;

    /**
     * 订单号创建时间
     */
    private Date createTime;

    /**
```

```

        * 订单描述
        */
        private String desc;
    }

```

14.3 顺序消息生产者

```

@Test
public void testOrderlyProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    List<Order> orderList = Arrays.asList(
        new Order(1, 111, 59D, new Date(), "下订单"),
        new Order(2, 111, 59D, new Date(), "物流"),
        new Order(3, 111, 59D, new Date(), "签收"),
        new Order(4, 112, 89D, new Date(), "下订单"),
        new Order(5, 112, 89D, new Date(), "物流"),
        new Order(6, 112, 89D, new Date(), "拒收")
    );
    // 循环集合开始发送
    orderList.forEach(order -> {
        Message message = new Message("TopicTest", order.toString().getBytes());
        try {
            // 发送的时候 相同的订单号选择同一个队列
            producer.send(message, new MessageQueueSelector() {
                @Override
                public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                    // 当前主题有多少个队列
                    int queueNumber = mqs.size();
                    // 这个arg 就是后面传入的 order.getOrderNumber()
                    Integer i = (Integer) arg;
                    // 用这个值去%队列的个数得到一个队列
                    int index = i % queueNumber;
                    // 返回选择的这个队列即可，那么相同的订单号 就会被放在相同的队列里 实现FIFO了
                    return mqs.get(index);
                }
            }, order.getOrderNumber());
        } catch (Exception e) {
            System.out.println("发送异常");
        }
    });
    // 关闭实例
    producer.shutdown();
}

```

14.4 顺序消息消费者，测试时等一会即可有延迟

```

@Test
public void testOrderlyConsumer() throws Exception {
    // 创建默认消费者组
}

```

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
// 设置nameServer 地址
consumer.setNamesrvAddr("localhost:9876");
// 订阅一个主题来消费 *表示没有过滤参数 表示这个主题的任何消息
consumer.subscribe("TopicTest", "*");
// 注册一个消费监听 MessageListenerOrderly 是顺序消费 单线程消费
consumer.registerMessageListener(new MessageListenerOrderly() {
    @Override
    public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
        MessageExt messageExt = msgs.get(0);
        System.out.println(new String(messageExt.getBody()));
        return ConsumeOrderlyStatus.SUCCESS;
    }
});
consumer.start();
System.in.read();
}
```

15. RocketMQ 发送批量消息

Rocketmq 可以一次性发送一组消息，那么这一组消息会被当做一个消息消费

15.1 批量消息生产者

```
@Test
public void testBatchProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    List<Message> msgs = Arrays.asList(
        new Message("TopicTest", "我是一组消息的 A 消息".getBytes()),
        new Message("TopicTest", "我是一组消息的 B 消息".getBytes()),
        new Message("TopicTest", "我是一组消息的 C 消息".getBytes())
    );
    SendResult send = producer.send(msgs);
    System.out.println(send);
    // 关闭实例
    producer.shutdown();
}
```

15.2 批量消息消费者

```
@Test
public void testBatchConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 表达式，默认是*
```

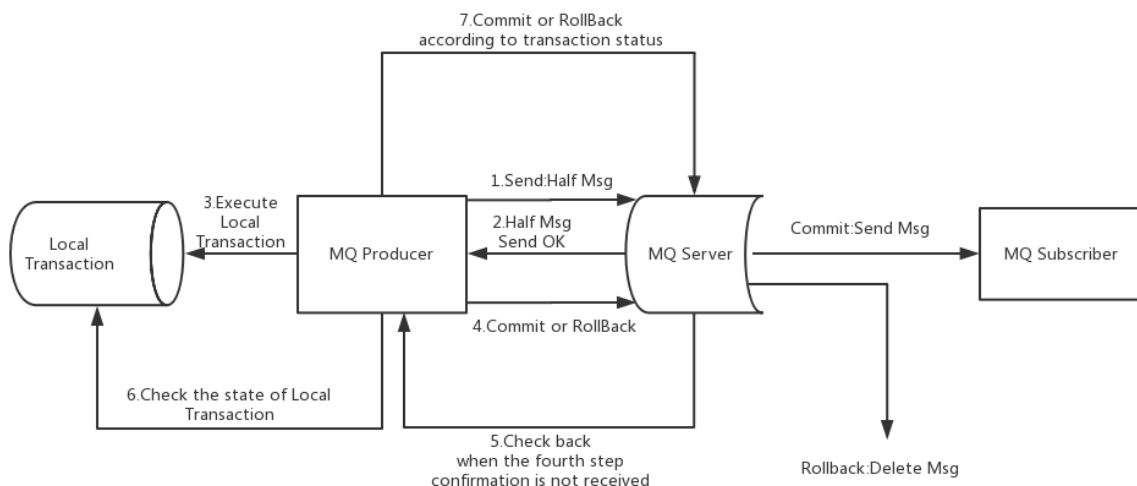


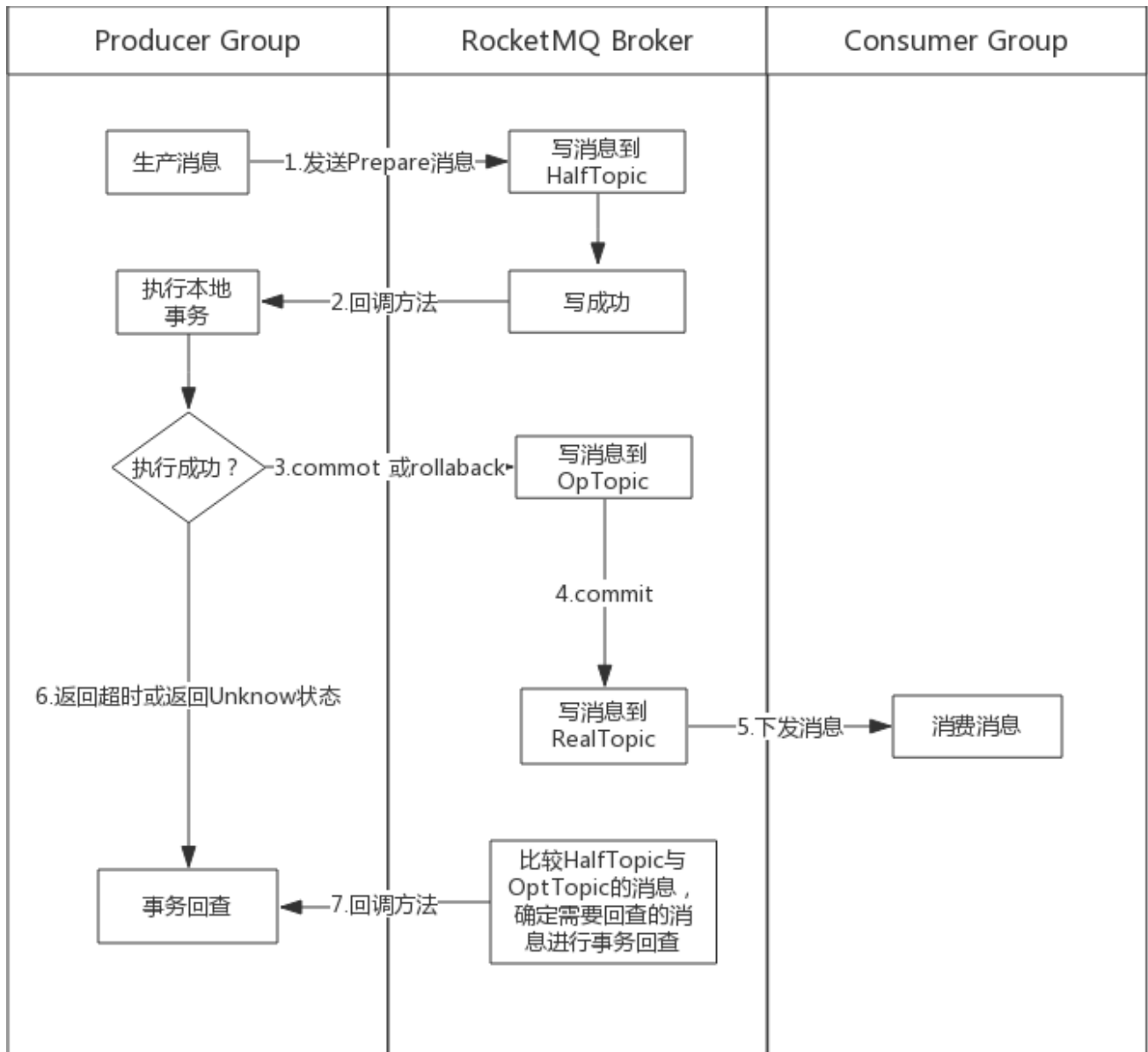
```
consumer.subscribe("TopicTest", "*");
// 注册一个消费监听 MessageListenerConcurrently 是并发消费
// 默认是 20 个线程一起消费，可以参看 consumer.setConsumeThreadMax()
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
        ConsumeConcurrentlyContext context) {
        // 这里执行消费的代码 默认是多线程消费
        System.out.println(Thread.currentThread().getName() + "----" + new String(msgs.get(0).getBody()));
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
consumer.start();
System.in.read();
}
```

16. RocketMQ 发送事务消息

16.1 事务消息的发送流程

它可以被认为是一个两阶段的提交消息实现，以确保分布式系统的最终一致性。事务性消息确保本地事务的执行和消息的发送可以原子地执行。





上图说明了事务消息的大致方案，其中分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程。

事务消息发送及提交

1. 发送消息 (half 消息) 。
2. 服务端响应消息写入结果。
3. 根据发送结果执行本地事务 (如果写入失败，此时 half 消息对业务不可见，本地逻辑不执行) 。
4. 根据本地事务状态执行 Commit 或 Rollback (Commit 操作生成消息索引，消息对消费者可见)

事务补偿

1. 对没有 Commit/Rollback 的事务消息（pending 状态的消息），从服务端发起一次“回查”
2. Producer 收到回查消息，检查回查消息对应的本地事务的状态
3. 根据本地事务状态，重新 Commit 或者 Rollback

其中，补偿阶段用于解决消息 UNKNOWN 或者 Rollback 发生超时或者失败的情况。

事务消息状态

事务消息共有三种状态，提交状态、回滚状态、中间状态：

- TransactionStatus.CommitTransaction：提交事务，它允许消费者消费此消息。
- TransactionStatus.RollbackTransaction：回滚事务，它代表该消息将被删除，不允许被消费。
- TransactionStatus.Unknown：中间状态，它代表需要检查消息队列来确定状态。

16.2 事务消息生产者

```
/**
 * TransactionalMessageCheckService 的检测频率默认 1 分钟，可通过在 broker.conf 文件中设置 transactionCheckInterval 的
 * 值来改变默认值，单位为毫秒。
 * 从 broker 配置文件中获取 transactionTimeOut 参数值。
 * 从 broker 配置文件中获取 transactionCheckMax 参数值，表示事务的最大检测次数，如果超过检测次数，消息会默认为丢弃，即回滚
 * 消息。
 *
 * @throws Exception
 */
@Test
public void testTransactionProducer() throws Exception {
    // 创建一个事务消息生产者
    TransactionMQProducer producer = new TransactionMQProducer("test-group");
    producer.setNamesrvAddr("localhost:9876");
    // 设置事务消息监听器
    producer.setTransactionListener(new TransactionListener() {
        // 这个是执行本地业务方法
        @Override
        public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
            System.out.println(new Date());
            System.out.println(new String(msg.getBody()));
            // 这个可以使用 try catch 对业务代码进行性包裹
            // COMMIT_MESSAGE 表示允许消费者消费该消息
            // ROLLBACK_MESSAGE 表示该消息将被删除，不允许消费
            // UNKNOWN 表示需要 MQ 回查才能确定状态 那么过一会 代码会走下面的 checkLocalTransaction(msg) 方法
            return LocalTransactionState.UNKNOWN;
        }
    });

    // 这里是回查方法 回查不是再次执行业务操作，而是确认上面的操作是否有结果
    // 默认是 1min 回查 默认回查 15 次 超过次数则丢弃打印日志 可以通过参数设置
    // transactionTimeOut 超时时间
    // transactionCheckMax 最大回查次数
    // transactionCheckInterval 回查间隔时间单位毫秒
    // 触发条件
    // 1. 当上面执行本地事务返回结果 UNKNOWN 时，或者下面的回查方法也返回 UNKNOWN 时 会触发回查
    // 2. 当上面操作超过 20s 没有做出一个结果，也就是超时或者卡主了，也会进行回查
}
```

```
@Override
public LocalTransactionState checkLocalTransaction(MessageExt msg) {
    System.err.println(new Date());
    System.err.println(new String(msg.getBody()));
    // 这里
    return LocalTransactionState.UNKNOW;
}
});
producer.start();
Message message = new Message("TopicTest2", "我是一个事务消息".getBytes());
// 发送消息
producer.sendMessageInTransaction(message, null);
System.out.println(new Date());
System.in.read();
}
```

16.3 事务消息消费者

```
@Test
public void testTransactionConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 *表示没有过滤参数 表示这个主题的任何消息
    consumer.subscribe("TopicTest2", "*");
    // 注册一个消费监听 MessageListenerConcurrently 是并发消费
    // 默认是20个线程一起消费, 可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {
            // 这里执行消费的代码 默认是多线程消费
            System.out.println(Thread.currentThread().getName() + "----" + new String(msgs.get(0).getBody()));
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

16.4 测试结果

Wed Mar 23 14:24:53 CST 2022

我是一个事务消息

Wed Mar 23 14:24:53 CST 2022

Wed Mar 23 14:25:19 CST 2022

我是一个事务消息

Wed Mar 23 14:26:19 CST 2022

我是一个事务消息

Wed Mar 23 14:27:19 CST 2022

我是一个事务消息

17. RocketMQ 发送带标签的消息，消息过滤

Rocketmq 提供消息过滤功能，通过 tag 或者 key 进行区分

我们往一个主题里面发送消息的时候，根据业务逻辑，可能需要区分，比如带有 tagA 标签的被 A 消费，带有 tagB 标签的被 B 消费，还有在事务监听的类里面，只要是事务消息都要走同一个监听，我们也需要通过过滤才区别对待

17.1 标签消息生产者

```
@Test
public void testTagProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    Message msg = new Message("TopicTest", "tagA", "我是一个带标记的消息".getBytes());
    SendResult send = producer.send(msg);
    System.out.println(send);
    // 关闭实例
    producer.shutdown();
}
```

17.2 标签消息消费者

```
@Test
public void testTagConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 表达式，默认是*,支持"tagA || tagB || tagC" 这样或者的写法 只要是符合任何一个标签都可以消费
    consumer.subscribe("TopicTest", "tagA || tagB || tagC");
    // 注册一个消费监听 MessageListenerConcurrently 是并发消费
    // 默认是 20 个线程一起消费，可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {
            // 这里执行消费的代码 默认是多线程消费
            System.out.println(Thread.currentThread().getName() + "----" + new String(msgs.get(0).getBody()));
            System.out.println(msgs.get(0).getTags());
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

17.3 什么时候该用 Topic, 什么时候该用 Tag?

总结：不同的业务应该使用不同的 Topic 如果是相同的业务里面有不同表的表现形式，那么我们要使用 tag 进行区分

可以从以下几个方面进行判断：

- 1.消息类型是否一致：如普通消息、事务消息、定时（延时）消息、顺序消息，不同的消息类型使用不同的 Topic，无法通过 Tag 进行区分。
- 2.业务是否相关联：没有直接关联的消息，如淘宝交易消息，京东物流消息使用不同的 Topic 进行区分；而同样是天猫交易消息，电器类订单、女装类订单、化妆品类订单的消息可以用 Tag 进行区分。
- 3.消息优先级是否一致：如同样是物流消息，盒马必须小时内送达，天猫超市 24 小时内送达，淘宝物流则相对会慢一些，不同优先级的消息用不同的 Topic 进行区分。
- 4.消息量级是否相当：有些业务消息虽然量小但是实时性要求高，如果跟某些万亿量级的消息使用同一个 Topic，则有可能会因为过长的等待时间而“饿死”，此时需要将不同量级的消息进行拆分，使用不同的 Topic。

总的来说，针对消息分类，您可以选择创建多个 Topic，或者在同一个 Topic 下创建多个 Tag。但通常情况下，不同的 Topic 之间的消息没有必然的联系，而 Tag 则用来区分同一个 Topic 下相互关联的消息，例如全集和子集的关系、流程先后的关系。

18. RocketMQ 中消息的 Key

在 rocketmq 中的消息，默认会有一个 messageId 当做消息的唯一标识，我们也可以给消息携带一个 key，用作唯一标识或者业务标识，包括在控制面板查询的时候也可以使用 messageId 或者 key 来进行查询



RocketMQ仪表盘 运维 驾驶舱 集群 主题 消费者 生产者 消息 消息轨迹

TOPIC MESSAGE KEY MESSAGE ID

Only Return 64 Messages

Topic: TopicTest Key: Q搜索

Message ID	Tag	Key	StoreTime
------------	-----	-----	-----------

18.1 带 key 消息生产者

```
@Test
public void testKeyProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    Message msg = new Message("TopicTest", "tagA", "key", "我是一个带标记和 key 的消息".getBytes());
    SendResult send = producer.send(msg);
    System.out.println(send);
    // 关闭实例
    producer.shutdown();
}
```

18.2 带 key 消息消费者

```
@Test
public void testKeyConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 表达式, 默认是*, 支持"tagA || tagB || tagC" 这样或者的写法 只要是符合任何一个标签都可以消费
    consumer.subscribe("TopicTest", "tagA || tagB || tagC");
    // 注册一个消费监听 MessageListenerConcurrently 是并发消费
    // 默认是 20 个线程一起消费, 可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {
            // 这里执行消费的代码 默认是多线程消费
            System.out.println(Thread.currentThread().getName() + "----" + new String(msgs.get(0).getBody()));
            System.out.println(msgs.get(0).getTags());
            System.out.println(msgs.get(0).getKeys());
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

TOPIC	MESSAGE KEY	MESSAGE ID
-------	-------------	------------

Only Return 64 Messages

Topic: TopicTest	Key: key	Q搜索
------------------	----------	-----

Message ID	Tag	Key	StoreTime	Operation
7F0000015DBC18B4AAC274902E120000	tagA	key	2022-03-23 15:13:26	MESSAGE DETAIL

19. RocketMQ 重试机制

19.1 生产者重试

```
// 失败的情况重发 3 次
producer.setRetryTimesWhenSendFailed(3);
// 消息在 1s 内没有发送成功，就会重试
producer.send(msg, 1000);
```

19.2 消费者重试

在消费者放 `return ConsumeConcurrentlyStatus.RECONSUME_LATER`; 后就会执行重试
上图代码中说明了，我们再实际生产过程中，一般重试 3-5 次，如果还没有消费成功，则可以把消息签收了，通知人工等处理

```
/**
 * 测试消费者
 *
 * @throws Exception
 */
@Test
public void testConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    // 设置 nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 *表示没有过滤参数 表示这个主题的任何消息
    consumer.subscribe("TopicTest", "*");
    // 注册一个消费监听
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                                                         ConsumeConcurrentlyContext context) {
            try {
                // 这里执行消费的代码
                System.out.println(Thread.currentThread().getName() + "-----" + msgs);
                // 这里制造一个错误
                int i = 10 / 0;
            } catch (Exception e) {
                // 出现问题 判断重试的次数
                MessageExt messageExt = msgs.get(0);
                // 获取重试的次数 失败一次消息中的失败次数会累加一次
                int reconsumeTimes = messageExt.getReconsumeTimes();
                if (reconsumeTimes >= 3) {
                    // 则把消息确认了，可以将这条消息记录到日志或者数据库 通知人工处理
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                } else {
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
            }
        }
    });
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
```



```
    }
  });
  consumer.start();
  System.in.read();
}
```

20. RocketMQ 死信消息

当消费重试到达阈值以后，消息不会被投递给消费者了，而是进入了死信队列

当一条消息初次消费失败，RocketMQ 会自动进行消息重试，达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息。此时，该消息不会立刻被丢弃，而是将其发送到该消费者对应的特殊队列中，这类消息称为死信消息（Dead-Letter Message），存储死信消息的特殊队列称为死信队列（Dead-Letter Queue），死信队列是死信 Topic 下分区数唯一的单独队列。如果产生了死信消息，那对应的 ConsumerGroup 的死信 Topic 名称为%DLQ%ConsumerGroupName，死信队列的消息将不会再被消费。可以利用 RocketMQ Admin 工具或者 RocketMQ Dashboard 上查询到对应死信消息的信息。我们也可以去监听死信队列，然后进行自己的业务上的逻辑

20.1 消息生产者

```
@Test
public void testDeadMsgProducer() throws Exception {
    DefaultMQProducer producer = new DefaultMQProducer("dead-group");
    producer.setNamesrvAddr("localhost:9876");
    producer.start();
    Message message = new Message("dead-topic", "我是一个死信消息".getBytes());
    producer.send(message);
    producer.shutdown();
}
```

20.2 消息消费者

```
@Test
public void testDeadMsgConsumer() throws Exception {
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("dead-group");
    consumer.setNamesrvAddr("localhost:9876");
    consumer.subscribe("dead-topic", "*");
    // 设置最大消费重试次数 2 次
    consumer.setMaxReconsumeTimes(2);
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
            System.out.println(msgs);
            // 测试消费失败
            return ConsumeConcurrentlyStatus.RECONSUME_LATER;
        }
    });
    consumer.start();
    System.in.read();
}
```

}

20.3 死信消费者

注意权限问题

修改主题

BROKER_NAME: broker-a X

主题名: %DLQ%dead-group

写队列数量: 1

读队列数量: 1

perm: 6

把2改成6

关闭 提交

```
@Test
public void testDeadMq() throws Exception{
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("dead-group");
    consumer.setNamesrvAddr("localhost:9876");
    // 消费重试到达阈值以后, 消息不会被投递给消费者了, 而是进入了死信队列
    // 队列名称 默认是 %DLQ% + 消费者组名
    consumer.subscribe("%DLQ%dead-group", "*");
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
            System.out.println(msgs);
            // 处理消息 签收
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

20.4 控制台显示

RocketMQ仪表盘 运维 驾驶舱 集群 主题 消费者 生产者 消息 消息轨迹

主题: ☒ 普通 ☐ 重试 ☒ 死信 ☐ 系统 新增/更新 刷新

主题	操作
%DLQ%dead-group	状态 路由 CONSUMER 管理 TOPIC 配置 发送消息 重置消费位点 跳过堆积 删除

%DLQ%dead-group订阅组						
订阅组	dead-group	延迟	0	最后消费时间	2022-03-21 21:02:23	
Broker	队列	消费者终端		代理者位点	消费者位点	上次时间
LAPTOP-BV73O3S9	0	192.168.199.165@18664#92974693177600		1	1	2022-03-21 21:02:23

关闭

21. RocketMQ 消息重复消费问题

21.1 为什么会出现重复消费问题呢？

BROADCASTING(广播)模式下，所有注册的消费者都会消费，而这些消费者通常是集群部署的一个个微服务，这样就会多台机器重复消费，当然这个是根据需要来选择。

CLUSTERING（负载均衡）模式下，如果一个 topic 被多个 consumerGroup 消费，也会重复消费。

即使是在 CLUSTERING 模式下，同一个 consumerGroup 下，一个队列只会分配给一个消费者，看起来好像是不会重复消费。但是，有个特殊情况：一个消费者新上线后，同组的所有消费者要重新负载均衡（反之一个消费者掉线后，也一样）。一个队列所对应的新的消费者要获取之前消费的 offset（偏移量，也就是消息消费的点位），此时之前的消费者可能已经消费了一条消息，但是并没有把 offset 提交给 broker，那么新的消费者可能会重新消费一次。虽然 orderly 模式是前一个消费者先解锁，后一个消费者加锁再消费的模式，比起 concurrently 要严格了，但是加锁的线程和提交 offset 的线程不是同一个，所以还是会出现极端情况下的重复消费。

还有在发送批量消息的时候，会被当做一条消息进行处理，那么如果批量消息中有一条业务处理成功，其他失败了，还是会被重新消费一次。

那么如果在 CLUSTERING（负载均衡）模式下，并且在同一个消费者组中，不希望一条消息被重复消费，改怎么办呢？我们可以想到去重操作，找到消息唯一的标识，可以是 msgId 也可以是你自定义的唯一的 key，这样就可以去重了

21.2 解决方案

使用去重方案解决，例如将消息的唯一标识存起来，然后每次消费之前先判断是否存在这个唯一标识，如果存在则不消费，如果不存在则消费，并且消费以后将这个标记保存。

想法很好，但是消息的体量是非常大的，可能在生产环境中会到达上千万甚至上亿条，那么我们该如何选择一个容器来保存所有消息的标识，并且又可以快速的判断是否存在呢？

我们可以选择布隆过滤器(BloomFilter)

布隆过滤器 (Bloom Filter) 是 1970 年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

在 hutool 的工具中我们可以直接使用，当然你自己使用 redis 的 bitmap 类型手写一个也是可以的 <https://hutool.cn/docs/#/bloomFilter/%E6%A6%82%E8%BF%B0>

The screenshot shows the Hutool documentation website. On the left is a sidebar menu with various tool categories. The '布隆过滤 (Hutool-bloomFilter)' item is highlighted with a red box. The main content area has a title '介绍' (Introduction) and a paragraph explaining the Bloom Filter concept. Below this is a '使用' (Usage) section with a code block showing Java code for initializing and using a Bloom Filter. A reference link is also provided.

特别赞助 by:

介绍

布隆过滤器 (英语: Bloom Filter) 是1970年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

布隆过滤器的原理是，当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点，把它们置为1。检索时，我们只要看看这些点是不是都是1就 (大约) 知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是布隆过滤器的基本思想。

参考: <https://www.cnblogs.com/z941030/p/9218356.html>

使用

```

// 初始化
BitMapBloomFilter filter = new BitMapBloomFilter(10);
filter.add("123");
filter.add("abc");
filter.add("ddd");

// 查找
filter.contains("abc")
    
```

21.3 测试生产者

@Test

```

public void testRepeatProducer() throws Exception {
    // 创建默认的生产者
    DefaultMQProducer producer = new DefaultMQProducer("test-group");
    // 设置nameServer 地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动实例
    producer.start();
    // 我们可以使用自定义key 当做唯一标识
    String keyId = UUID.randomUUID().toString();
    System.out.println(keyId);
}
    
```

```
Message msg = new Message("TopicTest", "tagA", keyId, "我是一个测试消息".getBytes());
SendResult send = producer.send(msg);
System.out.println(send);
// 关闭实例
producer.shutdown();
}
```

21.4 添加 hutool 的依赖

```
<dependency>
  <groupId>cn.hutool</groupId>
  <artifactId>hutool-all</artifactId>
  <version>5.7.11</version>
</dependency>
```

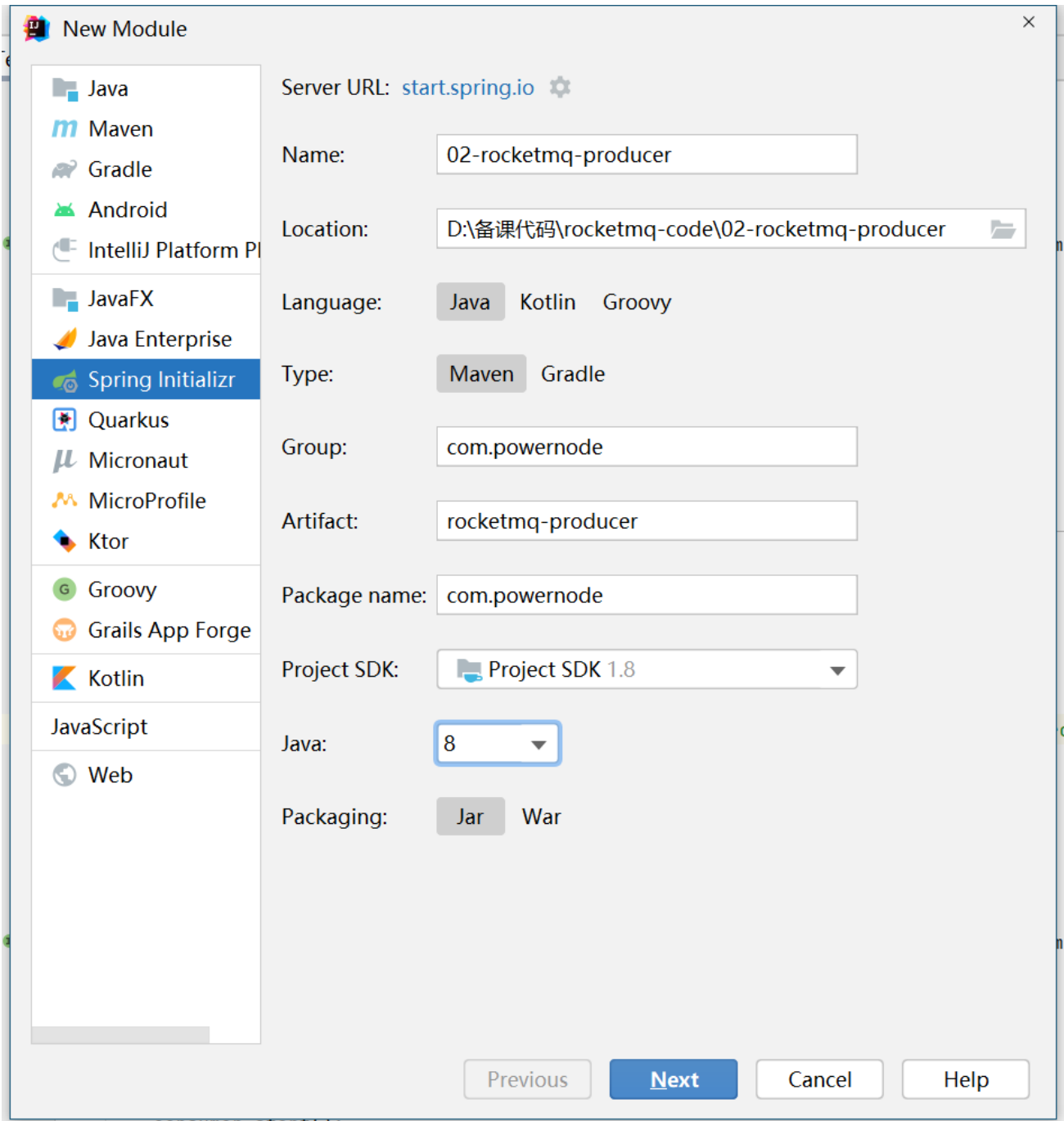
21.5 测试消费者

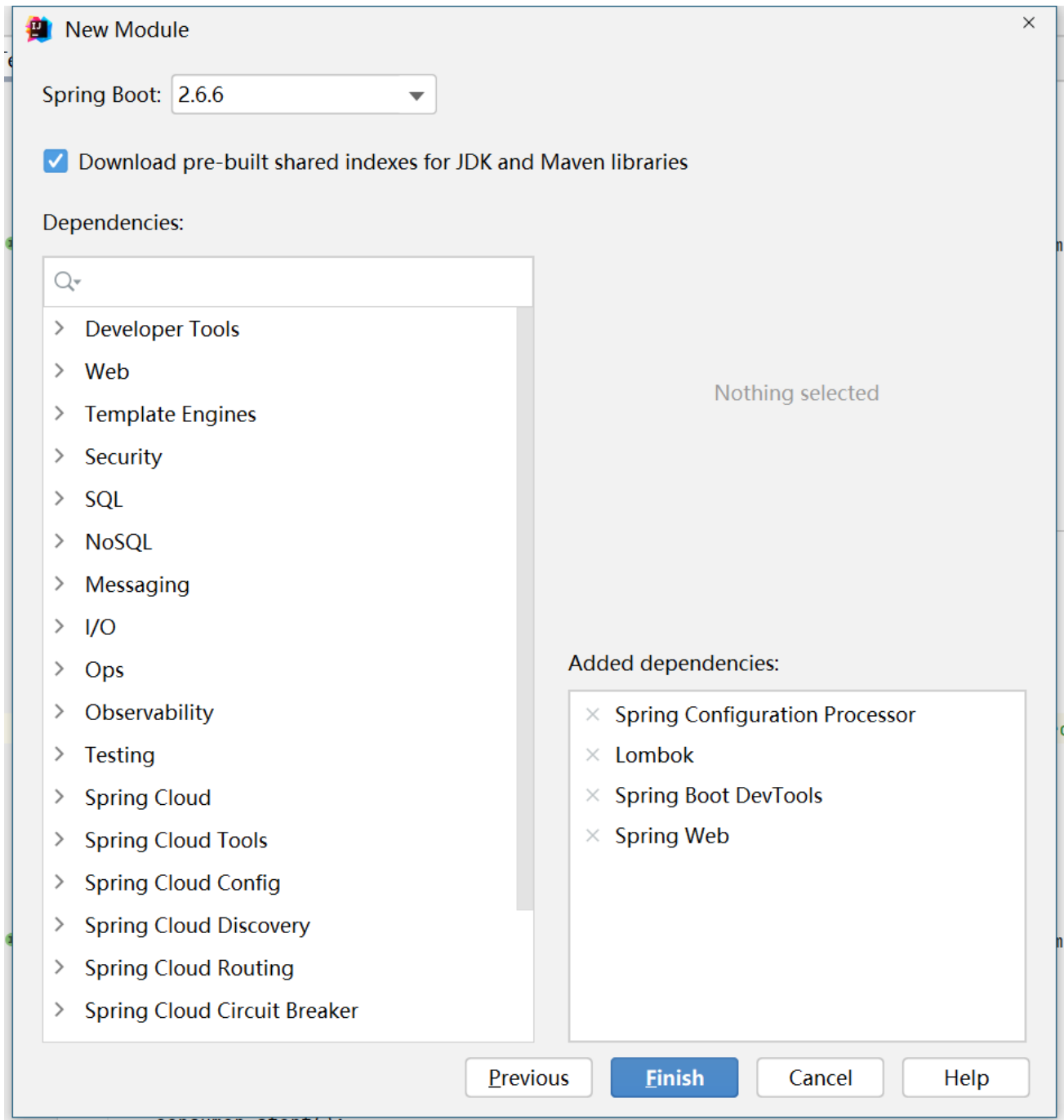
```
/**
 * 在 boot 项目中可以使用@Bean 在整个容器中放置一个单利对象
 */
public static BitMapBloomFilter bloomFilter = new BitMapBloomFilter(100);

@Test
public void testRepeatConsumer() throws Exception {
    // 创建默认消费者组
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("consumer-group");
    consumer.setMessageModel(MessageModel.BROADCASTING);
    // 设置 nameServer 地址
    consumer.setNamesrvAddr("localhost:9876");
    // 订阅一个主题来消费 表达式, 默认是*
    consumer.subscribe("TopicTest", "*");
    // 注册一个消费监听 MessageListenerConcurrently 是并发消费
    // 默认是 20 个线程一起消费, 可以参看 consumer.setConsumeThreadMax()
    consumer.registerMessageListener(new MessageListenerConcurrently() {
        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                                                         ConsumeConcurrentlyContext context) {
            // 拿到消息的 key
            MessageExt messageExt = msgs.get(0);
            String keys = messageExt.getKeys();
            // 判断是否存在布隆过滤器中
            if (bloomFilter.contains(keys)) {
                // 直接返回了 不往下处理业务
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
            // 这个处理业务, 然后放入过滤器中
            // do sth...
            bloomFilter.add(keys);
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    });
    consumer.start();
    System.in.read();
}
```

22. Rocketmq 集成 SpringBoot

22.1 搭建 rocketmq-producer (消息生产者)





22.1.1 创建项目，完整的 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.6.3</version>
<relativePath/> <!-- Lookup parent from repository -->
</parent>
<groupId>com.powernode</groupId>
<artifactId>01-rocketmq-producer</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>rocketmq-producer</name>
<description>Demo project for Spring Boot</description>
<properties>
  <java.version>1.8</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- rocketmq 的依赖 -->
  <dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>2.0.2</version>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```


22.1.2 修改配置文件 application.yml

```
spring:
  application:
    name: rocketmq-producer
rocketmq:
  name-server: 127.0.0.1:9876    # rocketMq 的 nameServer 地址
  producer:
    group: powernode-group      # 生产者组别
    send-message-timeout: 3000  # 消息发送的超时时间
    retry-times-when-send-async-failed: 2 # 异步消息发送失败重试次数
    max-message-size: 4194304   # 消息的最大长度
```

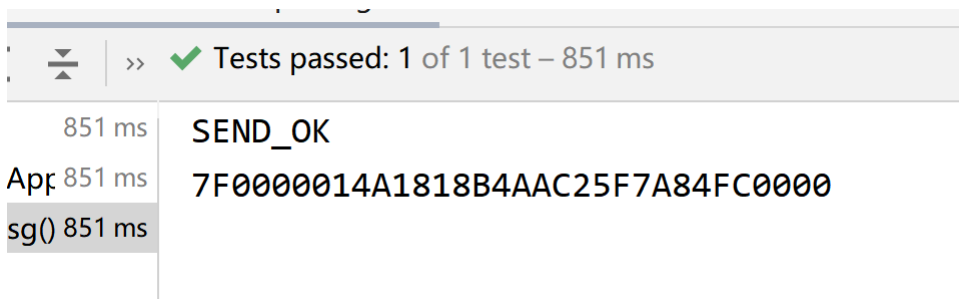
22.1.3 我们在测试类里面测试发送消息

往 powernode 主题里面发送一个简单的字符串消息

```
/**
 * 注入 rocketMQTemplate, 我们使用它来操作 mq
 */
@Autowired
private RocketMQTemplate rocketMQTemplate;

/**
 * 测试发送简单的消息
 *
 * @throws Exception
 */
@Test
public void testSimpleMsg() throws Exception {
    // 往 powernode 的主题里面发送一个简单的字符串消息
    SendResult sendResult = rocketMQTemplate.syncSend("powernode", "我是一个简单的消息");
    // 拿到消息的发送状态
    System.out.println(sendResult.getSendStatus());
    // 拿到消息的 id
    System.out.println(sendResult.getMsgId());
}
```

运行后查看控制台



22.1.4 查看 rocketMq 的控制台

RocketMQ仪表盘
运维
驾驶舱
集群
主题
消费者
生产者
消息
消息轨迹
更换语言

TOPIC
MESSAGE KEY
MESSAGE ID

Total 1 Messages

选择我们发送的主题

主题: powernode 开始: 2022-03-19 09:58 结束: 2022-03-19 12:58 搜索

Message ID	Tag	Key	StoreTime	Operation
7F0000014A1818B4AAC25F7A84FC0000			2022-03-19 12:57:45	消息详情

查看消息细节

Message ID:

7F0000014A1818B4AAC25F7A84FC0000

Topic:

powernode

Tag:

Key:

Storetime:

2022-03-19 12:57:45

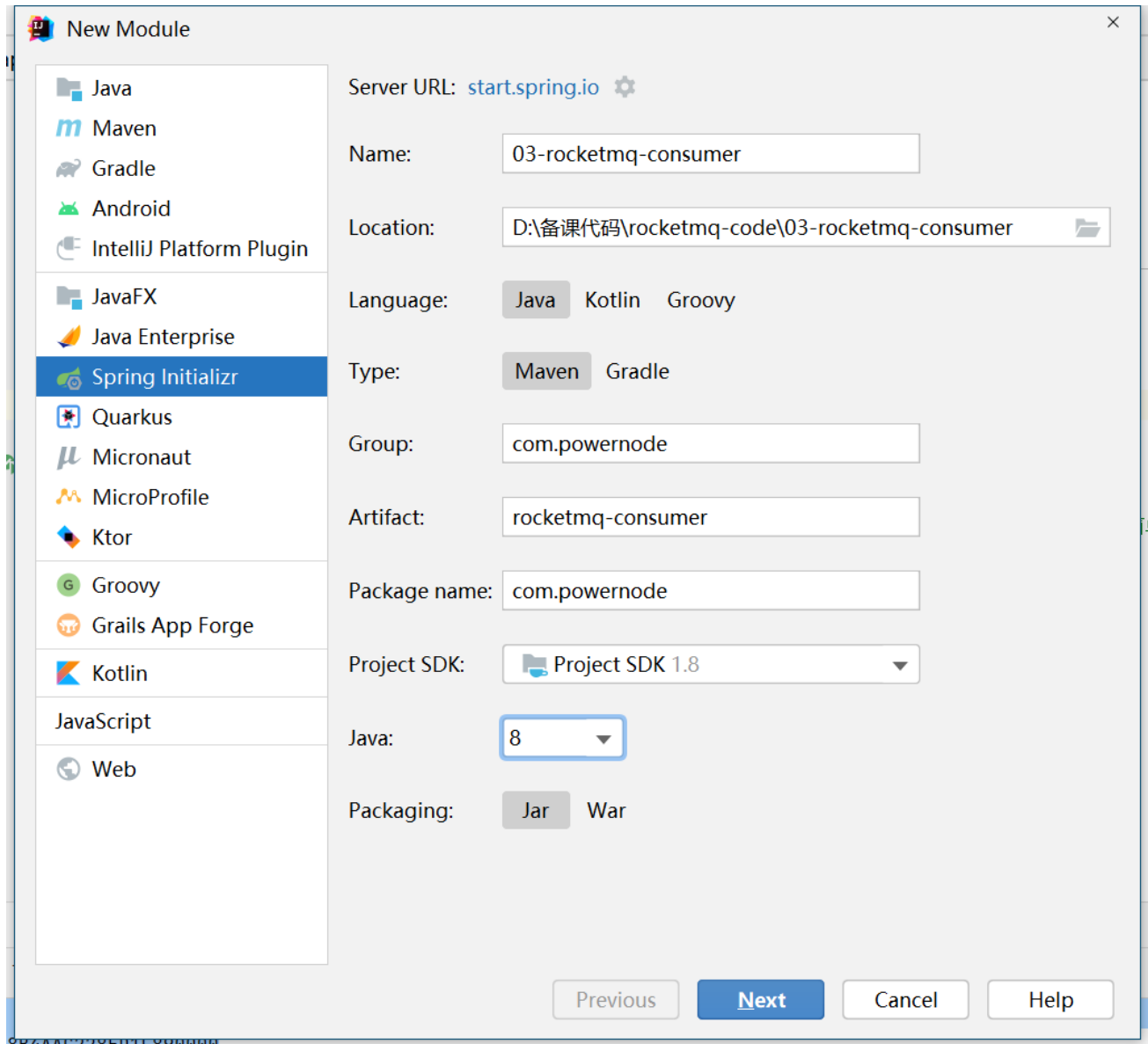
Message body:

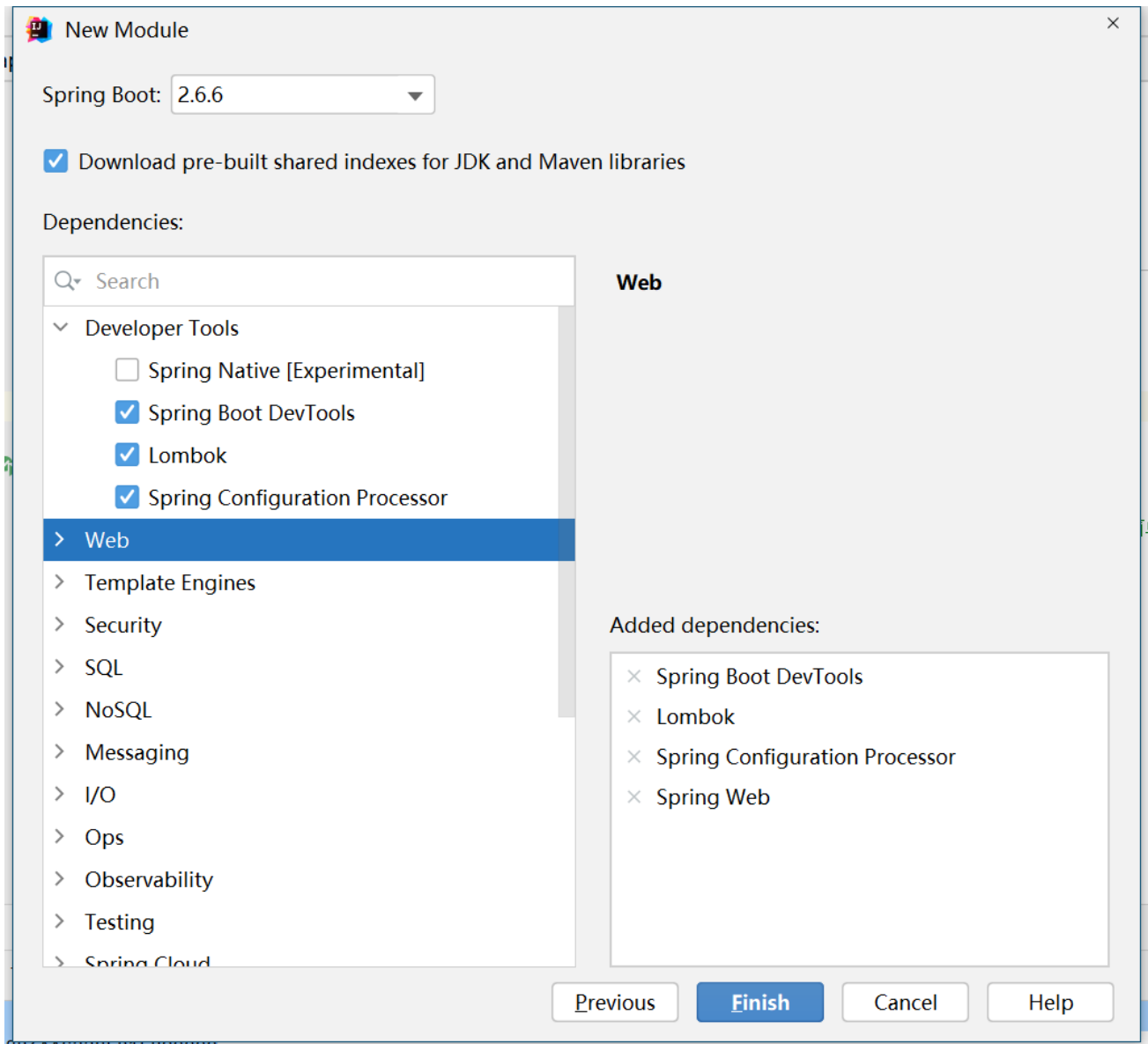
我是一个简单的消息

Operation

消息详情

22.2 搭建 rocketmq-consumer (消息消费者)





22.2.1 创建项目，完整的 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.3</version>
    <relativePath/> <!-- Lookup parent from repository -->
  </parent>
  <groupId>com.powernode</groupId>
  <artifactId>02-rocketmq-consumer</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<name>rocketmq-consumer</name>
<description>Demo project for Spring Boot</description>
<properties>
  <java.version>1.8</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- rocketmq 的依赖 -->
  <dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>2.0.2</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

22.2.2 修改配置文件 application.yml

```
spring:
  application:
```

```
name: rocketmq-consumer
rocketmq:
  name-server: 127.0.0.1:9876
```

22.2.3 添加一个监听的类 SimpleMsgListener

消费者要消费消息，就添加一个监听

```
package com.powernode.listener;

import org.apache.rocketmq.spring.annotation.MessageModel;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * 创建一个简单消息的监听
 * 1. 类上添加注解@Component 和@RocketMQMessageListener
 *    @RocketMQMessageListener(topic = "powernode", consumerGroup =
"powernode-group")
 *    topic 指定消费的主题, consumerGroup 指定消费组, 一个主题可以有多个消费者组,
一个消息可以被多个不同的组的消费者都消费
 * 2. 实现 RocketMQListener 接口, 注意泛型的使用, 可以为具体的类型, 如果想拿到消息
 * 的其他参数可以写成 MessageExt
 */
@Component
@RocketMQMessageListener(topic = "powernode", consumerGroup =
"powernode-group", messageModel = MessageModel.CLUSTERING)
public class SimpleMsgListener implements RocketMQListener<String> {

    /**
     * 消费消息的方法
     *
     * @param message
     */
    @Override
    public void onMessage(String message) {
        System.out.println(message);
    }
}
```

22.2.4 启动 rocketmq-consumer

查看控制台，发现我们已经监听到消息了

```
.RocketmqConsumerApplication : Sta
running for 3.898)
```

我是一个简单的消息

23. RocketMQ 发送对象消息和集合消息

我们接着在上面项目里面做

23.1 发送对象消息

主要是监听的时候泛型中写对象的类型即可

23.1.1 修改 rocketmq-producer 添加一个 Order 类

```
package com.powernode.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

/**
 * 订单对象
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Order {
    /**
     * 订单号
     */
    private String orderId;

    /**
     * 订单名称
     */
    private String orderName;

    /**
     * 订单价格
     */
    private Double price;

    /**
     * 订单号创建时间
     */
    private Date createTime;
}
```

```
/**
 * 订单描述
 */
private String desc;
}
```

23.1.2 修改 rocketmq-producer 添加一个单元测试

```
/**
 * 测试发送对象消息
 *
 * @throws Exception
 */
@Test
public void testObjectMsg() throws Exception {
    Order order = new Order();
    order.setOrderId(UUID.randomUUID().toString());
    order.setOrderName("我的订单");
    order.setPrice(998D);
    order.setCreateTime(new Date());
    order.setDesc("加急配送");
    // 往 powernode-obj 主题发送一个订单对象
    rocketMQTemplate.syncSend("powernode-obj", order);
}
```

23.1.3 发送此消息

23.1.4 修改 rocketmq-consumer 也添加一个 Order 类 (拷贝过来)

23.1.5 修改 rocketmq-consumer 添加一个 ObjMsgListener

```
package com.powernode.listener;

import com.powernode.domain.Order;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

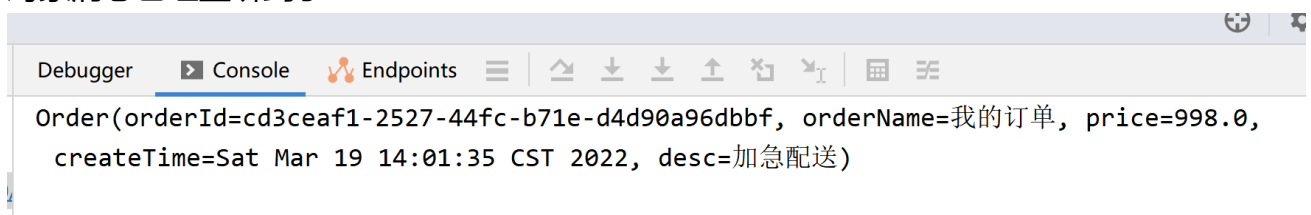
/**
 * 创建一个对象消息的监听
 * 1.类上添加注解@Component 和@RocketMQMessageListener
 * 2.实现 RocketMQListener 接口, 注意泛型的使用
 */
@Component
@RocketMQMessageListener(topic = "powernode-obj", consumerGroup =
"powernode-obj-group")
public class ObjMsgListener implements RocketMQListener<Order> {
```



```
/**
 * 消费消息的方法
 *
 * @param message
 */
@Override
public void onMessage(Order message) {
    System.out.println(message);
}
```

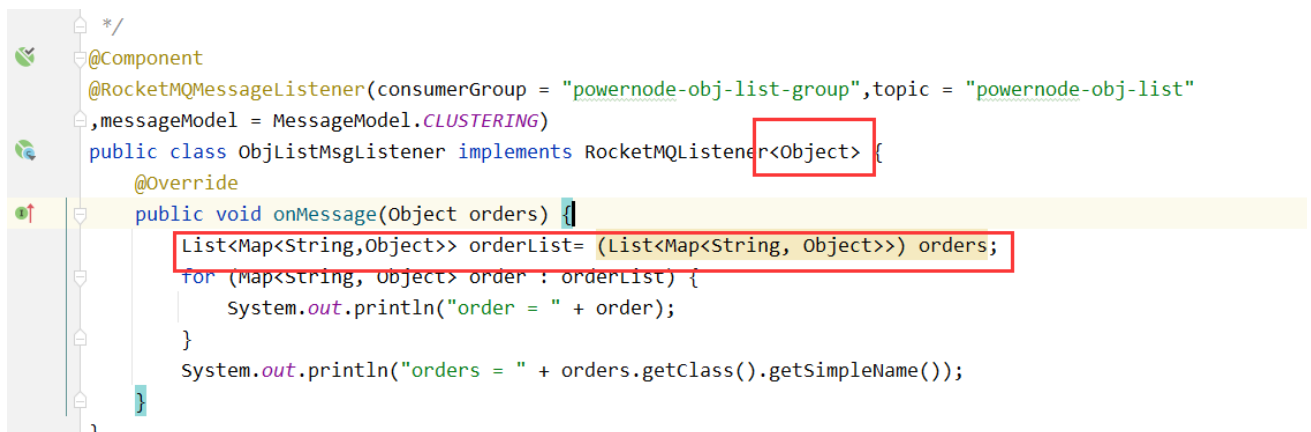
23.1.6 重启 rocketmq-consumer 后查看控制台

对象消息已经监听到了



23.2 发送集合消息

和对象消息同理, 创建一个 Order 的集合, 发送出去, 监听方注意修改泛型中的类型为 Object 即可, 这里就不做重复演示了



24. RocketMQ 集成 SpringBoot 发送不同消息模式

24.1 发送同步消息

理解为: 消息由消费者发送到 broker 后, 会得到一个确认, 是具有可靠性的
这种可靠性同步地发送方式使用的比较广泛, 比如: 重要的消息通知, 短信通知等。

我们在上面的快速入门中演示的消息，就是同步消息，即

```
rocketMQTemplate.syncSend()  
rocketMQTemplate.send()  
rocketMQTemplate.convertAndSend()
```

这三种发送消息的方法，底层都是调用 syncSend，发送的是同步消息

24.2 发送异步消息

```
rocketMQTemplate.asyncSend()
```

24.2.1 修改 rocketmq-producer 添加一个单元测试

```
/**  
 * 测试异步发送消息  
 *  
 * @throws Exception  
 */  
@Test  
public void testAsyncSend() throws Exception {  
    // 发送异步消息，发送完以后会有一个异步通知  
    rocketMQTemplate.asyncSend("powernode", "发送一个异步消息", new SendCallback()  
    {  
        /**  
         * 成功的回调  
         * @param sendResult  
         */  
        @Override  
        public void onSuccess(SendResult sendResult) {  
            System.out.println("发送成功");  
        }  
  
        /**  
         * 失败的回调  
         * @param throwable  
         */  
        @Override  
        public void onException(Throwable throwable) {  
            System.out.println("发送失败");  
        }  
    });  
    // 测试一下异步的效果  
    System.out.println("谁先执行");  
    // 挂起 jvm 不让方法结束  
    System.in.read();  
}
```

24.2.2 运行查看控制台效果

谁先发送打印在前面

```
谁先执行  
发送成功
```

24.3 发送单向消息

这种方式主要用在不关心发送结果的场景，这种方式吞吐量很大，但是存在消息丢失的风险，例如日志信息的发送

24.3.1 修改 rocketmq-producer 添加一个单元测试

```
/**  
 * 测试单向消息  
 *  
 * @throws Exception  
 */  
@Test  
public void testOnWay() throws Exception {  
    // 发送单向消息，没有返回值和结果  
    rocketMQTemplate.sendOnWay("powernode", "这是一个单向消息");  
}
```

24.4 发送延迟消息

24.4.1 修改 rocketmq-producer 添加一个单元测试

```
/**  
 * 测试延迟消息  
 *  
 * @throws Exception  
 */  
@Test  
public void testDelay() throws Exception {  
    // 构建消息对象  
    Message<String> message = MessageBuilder.withPayload("我是一个延迟消息").build();  
    // 发送一个延时消息，延迟等级为 4 级，也就是 30s 后被监听消费  
    SendResult sendResult = rocketMQTemplate.syncSend("powernode", message, 2000, 4);  
    System.out.println(sendResult.getSendStatus());  
}
```

24.4.2 运行后，查看消费者端，过了 30s 才被消费

这里注意的是 RocketMQ 不支持任意时间的延时

只支持以下几个固定的延时等级，等级 1 就对应 1s，以此类推，最高支持 2h 延迟

```
private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";
```

24.5 发送顺序消息

24.5.1 修改 Order 表添加一个顺序字段

```
/**
 * 订单的流程顺序
 */
private Integer seq;
```

24.5.2 修改 rocketmq-producer 添加一个单元测试

```
/**
 * 测试顺序消费
 * mq 会根据 hash 的值来存放到一个队列里面去
 *
 * @throws Exception
 */
@Test
public void testOrderly() throws Exception {
    List<Order> orders = Arrays.asList(
        new Order(UUID.randomUUID().toString().substring(0, 5), "张三的下订单", null, null, null, 1),
        new Order(UUID.randomUUID().toString().substring(0, 5), "张三的发短信", null, null, null, 1),
        new Order(UUID.randomUUID().toString().substring(0, 5), "张三的物流", null, null, null, 1),
        new Order(UUID.randomUUID().toString().substring(0, 5), "张三的签收", null, null, null, 1),

        new Order(UUID.randomUUID().toString().substring(0, 5), "李四的下订单", null, null, null, 2),
        new Order(UUID.randomUUID().toString().substring(0, 5), "李四的发短信", null, null, null, 2),
        new Order(UUID.randomUUID().toString().substring(0, 5), "李四的物流", null, null, null, 2),
        new Order(UUID.randomUUID().toString().substring(0, 5), "李四的签收", null, null, null, 2)
    );
    // 我们控制流程为 下订单->发短信->物流->签收 hash 的值为 seq, 也就是说 seq 相同的会放在同一个队列里面, 顺序消费
    orders.forEach(order -> {
        rocketMQTemplate.syncSendOrderly("powernode-obj", order, String.valueOf(order.getSeq()));
    });
}
```

24.5.3 发送消息

24.5.4 修改 rocketmq-consumer 的 ObjMsgListener

```
package com.powernode.listener;

import com.powernode.domain.Order;
```

```
import org.apache.rocketmq.spring.annotation.ConsumeMode;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * @Author 武汉动力节点
 * 创建一个对象消息的监听
 * 1.类上添加注解@Component 和@RocketMQMessageListener
 * 2.实现 RocketMQListener 接口, 注意泛型的使用
 * consumeMode 指定消费类型
 *     CONCURRENTLY 并发消费
 *     ORDERLY 顺序消费 messages orderly. one queue, one thread
 */
@Component
@RocketMQMessageListener(topic = "powernode-obj",
    consumerGroup = "powernode-obj-group",
    consumeMode = ConsumeMode.ORDERLY
)
public class ObjMsgListener implements RocketMQListener<Order> {

    /**
     * 消费消息的方法
     *
     * @param message
     */
    @Override
    public void onMessage(Order message) {
        System.out.println(message);
    }
}
```

24.5.5 重启 rocketmq-consumer

查看控制台，消息按照我们的放入顺序进行消费了

```
Debugger Console Endpoints
Order(orderId=e261e, orderName=李四的下订单, price=null, createTime=null, desc=null, seq=2)
Order(orderId=03cfe, orderName=张三的下订单, price=null, createTime=null, desc=null, seq=1)
INFO 39724 --- [MessageThread_2] a.r.s.s.DefaultRocketMQListenerCc
INFO 39724 --- [MessageThread_1] a.r.s.s.DefaultRocketMQListenerCc
Order(orderId=2e38a, orderName=李四的发短信, price=null, createTime=null, desc=null, seq=2)
Order(orderId=0068c, orderName=张三的发短信, price=null, createTime=null, desc=null, seq=1)
INFO 39724 --- [MessageThread_2] a.r.s.s.DefaultRocketMQListenerCc
INFO 39724 --- [MessageThread_1] a.r.s.s.DefaultRocketMQListenerCc
Order(orderId=10253, orderName=李四的物流, price=null, createTime=null, desc=null, seq=2)
Order(orderId=7d0fb, orderName=张三的物流, price=null, createTime=null, desc=null, seq=1)
INFO 39724 --- [MessageThread_2] a.r.s.s.DefaultRocketMQListenerCc
INFO 39724 --- [MessageThread_1] a.r.s.s.DefaultRocketMQListenerCc
Order(orderId=d789c, orderName=李四的签收, price=null, createTime=null, desc=null, seq=2)
Order(orderId=97c5c, orderName=张三的签收, price=null, createTime=null, desc=null, seq=1)
INFO 39724 --- [MessageThread_2] a.r.s.s.DefaultRocketMQListenerCc
INFO 39724 --- [MessageThread_1] a.r.s.s.DefaultRocketMQListenerCc
```

24.6 发送事务消息

24.6.1 修改 rocketmq-producer 添加一个单元测试

```
/**
 * 测试事务消息
 * 默认是 sync (同步的)
 * 事务消息会有确认和回查机制
 * 事务消息都会走到同一个监听回调里面, 所以我们需要使用 tag 或者 key 来区分过滤
 *
 * @throws Exception
 */
@Test
public void testTrans() throws Exception {
    // 构建消息体
    Message<String> message = MessageBuilder.withPayload("这是一个事务消息").build();
    // 发送事务消息 (同步的) 最后一个参数才是消息主题
    TransactionSendResult transaction = rocketMQTemplate.sendMessageInTransaction("powernode", message, "消息的参数");
    // 拿到本地事务状态
    System.out.println(transaction.getLocalTransactionState());
    // 挂起 jvm, 因为事务的回查需要一些时间
    System.in.read();
}
```

24.6.2 修改 rocketmq-producer 添加一个本地事务消息的监听(半消息)

```
/**
 * 事务消息的监听与回查
 * 类上添加注解@RocketMQTransactionListener 表示这个类是本地事务消息的监听类
 * 实现 RocketMQLocalTransactionListener 接口
 * 两个方法为执行本地事务, 与回查本地事务
 */
@Component
@RocketMQTransactionListener(corePoolSize = 4,maximumPoolSize = 8)
public class TmMsgListener implements RocketMQLocalTransactionListener {
```

```
/**
 * 执行本地事务，这里可以执行一些业务
 * 比如操作数据库，操作成功就 return RocketMQLocalTransactionState.COMMIT;
 * 可以使用 try catch 来控制成功或者失败;
 * @param msg
 * @param arg
 * @return
 */
@Override
public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg) {
    // 拿到消息参数
    System.out.println(arg);
    // 拿到消息头
    System.out.println(msg.getHeaders());
    // 返回状态 COMMIT, UNKNOWN
    return RocketMQLocalTransactionState.UNKNOWN;
}

/**
 * 回查本地事务，只有上面的执行方法返回 UNKNOWN 时，才执行下面的方法 默认是 1min 回查
 * 此方法为回查方法，执行需要等待一会
 * xxx.isSuccess() 这里可以执行一些检查的方法
 * 如果返回 COMMIT，那么本地事务就算是提交成功了，消息就会被消费者看到
 *
 * @param msg
 * @return
 */
@Override
public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {
    System.out.println(msg);
    return RocketMQLocalTransactionState.COMMIT;
}
}
```

24.6.3 测试发送事务，建议断点启动

1. 消息会先到事务监听类的执行方法，
2. 如果返回状态为 COMMIT，则消费者可以直接监听到
3. 如果返回状态为 ROLLBACK，则消息发送失败，直接回滚
4. 如果返回状态为 UNKNOWN，则过一会会走回查方法
5. 如果回查方法返回状态为 UNKNOWN 或者 ROLLBACK，则消息发送失败，直接回滚
6. 如果回查方法返回状态为 COMMIT，则消费者可以直接监听到

25. RocketMQ 集成 SpringBoot 的消息过滤


25.1 tag 过滤（常在消费者端过滤）

我们从源码注释得知，tag 带在主题后面用：来携带，感谢注释

```
/**
 * Same to {@link #syncSend(String, Message)}.
 *
 * @param destination formats: `topicName:tags`
 * @param payload the Object to use as payload
 * @return {@link SendResult}
 */
public SendResult syncSend(String destination, Object payload) {
    return syncSend(destination, payload, producer.getSendMsgTimeout());
}
```

我们往下去看源码，在

org.apache.rocketmq.spring.support.RocketMQUtil 的 getAndWrapMessage 方法里面看到了具体细节，我们也知道了 keys 在消息头里面携带



```
156 }
157
158 @private static Message getAndWrapMessage(String destination, MessageHeaders headers, byte[] payloads) {
159     if (destination == null || destination.length() < 1) {
160         return null;
161     }
162     if (payloads == null || payloads.length < 1) {
163         return null;
164     }
165     String[] tempArr = destination.split("regex: \":\", limit: 2); tag在topic后面用: 拼接
166     String topic = tempArr[0];
167     String tags = "";
168     if (tempArr.length > 1) {
169         tags = tempArr[1];
170     }
171     Message rocketMsg = new Message(topic, tags, payloads);
172     if (Objects.nonNull(headers) && !headers.isEmpty()) {
173         Object keys = headers.get(RocketMQHeaders.KEYS); keys在消息头里面
174         if (!StringUtils.isEmpty(keys)) { // if headers has 'KEYS', set rocketMQ message key
175             rocketMsg.setKeys(keys.toString());
176         }
177         Object flagObj = headers.getOrDefault(key: "FLAG", defaultValue: "0");
178         int flag = 0;
```

25.1.1 修改 rocketmq-producer 添加一个单元测试

```
/**
 * 发送一个带 tag 的消息
 *
 * @throws Exception
 */
@Test
public void testTagMsg() throws Exception {
    // 发送一个 tag 为 java 的数据
    rocketMQTemplate.syncSend("powernode-tag:java", "我是一个带 tag 的消息");
}
```



```
}
```

25.1.2 发送消息

25.1.3 修改 rocketmq-consumer 添加一个 TagMsgListener

```
package com.powernode.listener;

import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.annotation.SelectorType;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * @Author 武汉动力节点
 * 创建一个简单的标签消息的监听
 * 1.类上添加注解@Component 和@RocketMQMessageListener
 *     selectorType = SelectorType.TAG, 指定使用 tag 过滤。(也可以使用 sql92 需要在配置文件 broker.conf 中开启 enablePropertyFilter=true)
 *     selectorExpression = "java" 表达式, 默认是*,支持"tag1 || tag2 || tag3"
 * 2.实现 RocketMQListener 接口, 注意泛型的使用
 */
@Component
@RocketMQMessageListener(topic = "powernode-tag",
    consumerGroup = "powernode-tag-group",
    selectorType = SelectorType.TAG,
    selectorExpression = "java"
)
public class TagMsgListener implements RocketMQListener<String> {

    /**
     * 消费消息的方法
     */
    * @param message
    */
    @Override
    public void onMessage(String message) {
        System.out.println(message);
    }
}
```

25.1.4 重启 rocketmq-consumer 查看控制台

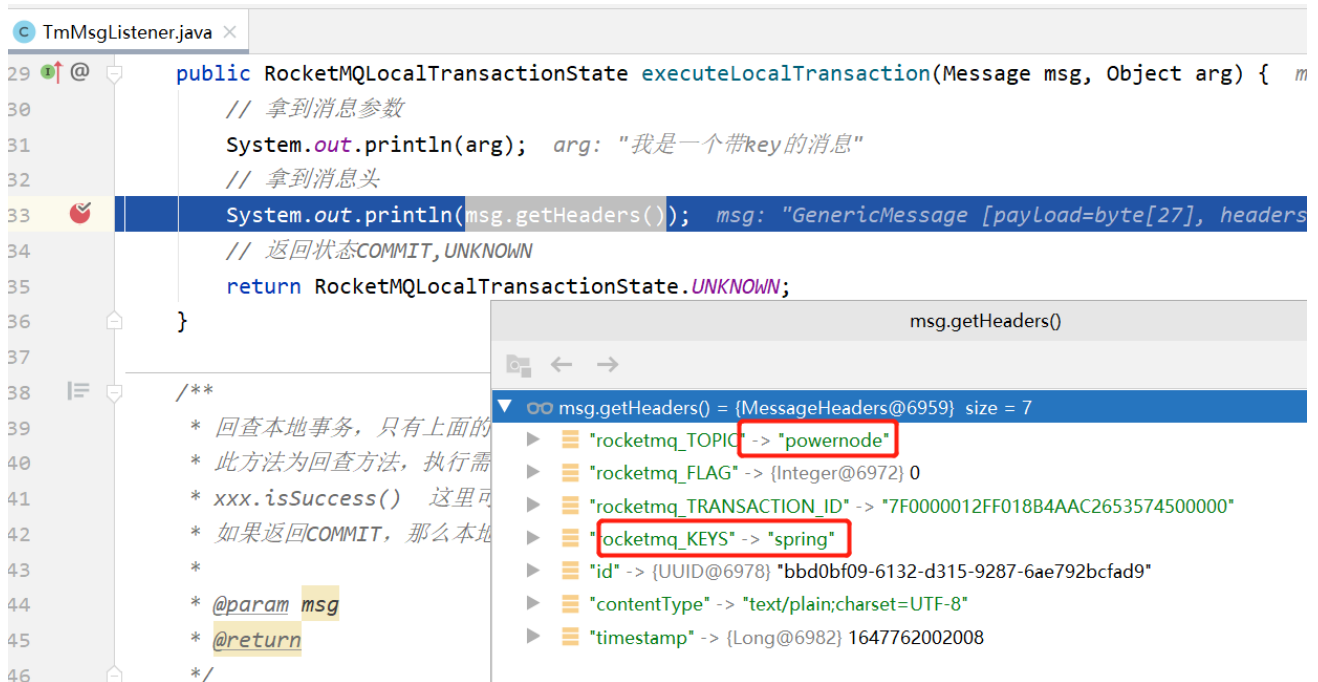
我是一个带tag的消息

25.2 Key 过滤（可以在事务监听类里面区分）

25.2.1 修改 rocketmq-producer 添加一个单元测试

```
/**
 * 发送一个带 key 的消息,我们使用事务消息 打断点查看消息头
 *
 * @throws Exception
 */
@Test
public void testKeyMsg() throws Exception {
    // 发送一个 key 为 spring 的事务消息
    Message<String> message = MessageBuilder.withPayload("我是一个带 key 的消息")
        .setHeader(RocketMQHeaders.KEYS, "spring")
        .build();
    rocketMQTemplate.sendMessageInTransaction("powernode", message, "我是一个带 key 的消息");
}
```

25.2.2 断点发送这个消息，查看事务里面消息头



The screenshot shows an IDE with a breakpoint set in the `executeLocalTransaction` method of `TmMsgListener.java`. The breakpoint is located at line 33, where `System.out.println(msg.getHeaders());` is called. The debug window shows the message headers for the message being sent. The headers are:

- `rocketmq_TOPIC` -> `"powernode"`
- `rocketmq_FLAG` -> `{Integer@6972} 0`
- `rocketmq_TRANSACTION_ID` -> `"7F0000012FF018B4AAC2653574500000"`
- `rocketmq_KEYS` -> `"spring"`
- `id` -> `{UUID@6978} "bbd0bf09-6132-d315-9287-6ae792bcfad9"`
- `contentType` -> `"text/plain;charset=UTF-8"`
- `timestamp` -> `{Long@6982} 1647762002008`

我们在 mq 的控制台也可以看到

主题:

powernode

▼

 开始: 2022-03-20 11:49 结束: 2022-03-20 14:49

Q搜索

Message ID	Tag	Key	StoreTime	Operation
7F0000015D2C18B4AAC264958D2E0000			2022-03-20 12:45:52	<div>消息详情</div>
7F0000011D2C18B4AAC264ACAA700000			2022-03-20 13:11:09	<div>消息详情</div>
7F0000012FF018B4AAC2653574500000		spring	2022-03-20 15:43:03	<div>消息详情</div>
7F000001302818B4AAC2649D0810000			2022-03-20 12:40:12	<div>消息详情</div>

26. RocketMQ 集成 SpringBoot 消息消费两种模式

Rocketmq 消息消费的模式分为两种：**负载均衡模式和广播模式**

负载均衡模式表示多个消费者交替消费同一个主题里面的消息

广播模式表示每个每个消费者都消费一遍订阅的主题的消息

26.1 再搭建一个消费者 rocketmq-consumer-b，依赖和配置文件和 rocketmq-consumer 一致，记住端口修改一下，避免占用

26.2 rocketmq-consumer-b 添加一个监听

```
package com.powernode.listener;

import org.apache.rocketmq.spring.annotation.MessageModel;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * messageModel 指定消息消费的模式
 * CLUSTERING 为负载均衡模式
 * BROADCASTING 为广播模式
 */
@Component
@RocketMQMessageListener(topic = "powernode",
    consumerGroup = "powernode-group",
    messageModel = MessageModel.CLUSTERING
)
public class ConsumerBListener implements RocketMQListener<String> {

    @Override
    public void onMessage(String message) {
        System.out.println(message);
    }
}
```

```
}  
}
```

26.3 修改 rocketmq-consumer 的 SimpleMsgListener

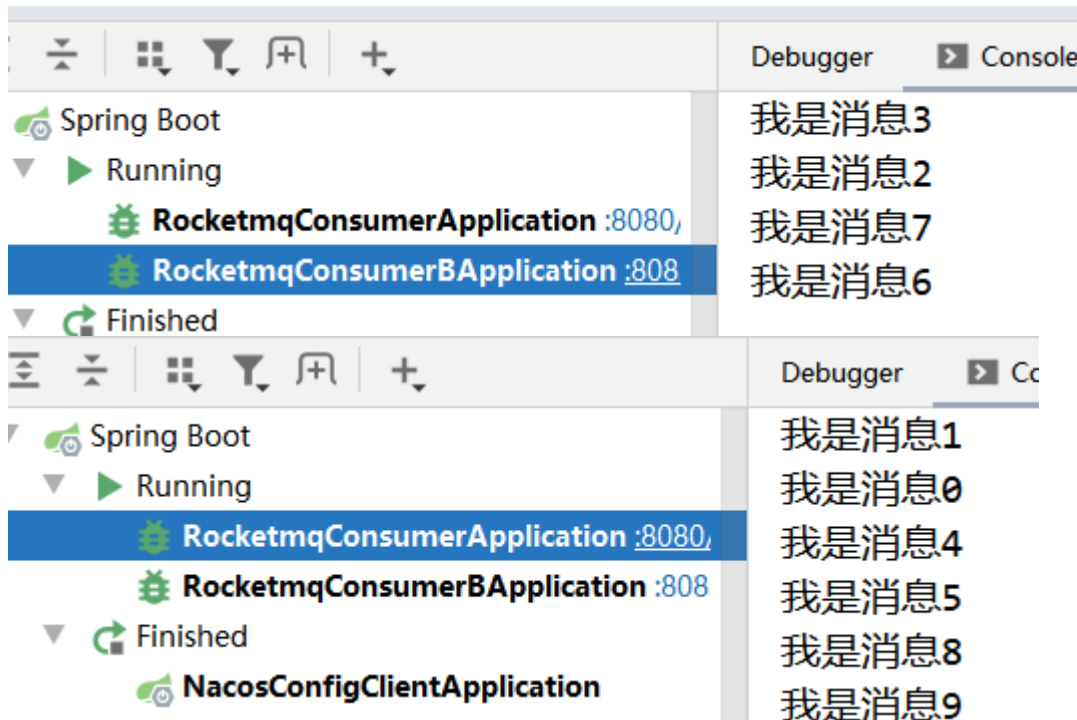
```
/**  
 * 创建一个简单消息的监听  
 * 1. 类上添加注解@Component 和@RocketMQMessageListener  
 *  
 * @RocketMQMessageListener(topic = "powernode", consumerGroup = "powernode-group")  
 * topic 指定消费的主题, consumerGroup 指定消费组, 一个主题可以有多个消费者组, 一个消息可以被多个不同的组的消费者都消费  
 * 2. 实现 RocketMQListener 接口, 注意泛型的使用  
 */  
@Component  
@RocketMQMessageListener(topic = "powernode",  
    consumerGroup = "powernode-group",  
    messageModel = MessageModel.CLUSTERING)  
public class SimpleMsgListener implements RocketMQListener<String> {  
  
    @Override  
    public void onMessage(String message) {  
        System.out.println(new Date());  
        System.out.println(message);  
    }  
}
```

26.4 启动两个消费者

26.5 在生产者里面添加一个单元测试并且运行

```
/**  
 * 测试消息消费的模式  
 *  
 * @throws Exception  
 */  
@Test  
public void testMsgModel() throws Exception {  
    for (int i = 0; i < 10; i++) {  
        rocketMQTemplate.syncSend("powernode", "我是消息" + i);  
    }  
}
```

26.6 查看两个消费者的控制台，发现是负载均衡的模式



26.7 修改两个消费者的模式为 BROADCASTING

重启测试，结果是广播模式，每个消费者都消费了这些消息

项目中 一般部署多态机器 消费者 2 - 3 根据业务可以选择具体的模式来配置

重置消费点位，将一个组的消费节点 设置在之前的某一个时间点上去 从这个时间点开始往后消费

跳过堆积 选择一个组 跳过堆积以后 这个组里面的的所有都不会被消费了

27. 如何解决消息堆积问题？

一般认为单条队列消息差值 $\geq 10w$ 时 算堆积问题

27.1 什么情况下会出现堆积

1. 生产太快了

生产方可以做业务限流

增加消费者数量,但是消费者数量 \leq 队列数量,适当的设置最大的消费线程数量(根据 $IO(2n)/CPU(n+1)$)

动态扩容队列数量,从而增加消费者数量

2. 消费者消费出现问题

排查消费者程序的问题

28. 如何确保消息不丢失?

1. 生产者使用同步发送模式,收到 mq 的返回确认以后 顺便往自己的数据库里面写
`msgId status(0) time`

2. 消费者消费以后 修改数据这条消息的状态 = 1

3. 写一个定时任务 间隔两天去查询数据 如果有 `status = 0 and time < day-2`

4. 将 mq 的刷盘机制设置为同步刷盘

5. 使用集群模式,搞主备模式,将消息持久化在不同的硬件上

6. 可以开启 mq 的 trace 机制,消息跟踪机制

1. 在 `broker.conf` 中开启消息追踪

`traceTopicEnable=true`

2. 重启 broker 即可

3. 生产者配置文件开启消息轨迹

`enable-msg-trace: true`

4. 消费者开启消息轨迹功能,可以给单独的某一个消费者开启

`enableMsgTrace = true`

在 rocketmq 的面板中可以查看消息轨迹

默认会将消息轨迹的数据存在 `RMQ_SYS_TRACE_TOPIC` 主题里面

29. 安全

1. 开启 acl 的控制 在 `broker.conf` 中开启 `aclEnable=true`
2. 配置账号密码 修改 `plain_acl.yml`
3. 修改控制面板的配置文件 放开 52/53 行 把 49 行改为 `true` 上传到服务器的 jar 包平级目录下即可