# 介绍

官网： <http://rocketmq.apache.org>

基础介绍：<https://mp.weixin.qq.com/s/uX2058WQmZbfzDQFvM7w5A>

## 什么是rocketmq

Github 上关于 RocketMQ 的介绍：

RcoketMQ 是一款低延迟、高可靠、可伸缩、易于使用的消息中间件。具有以下特性：

* 支持发布/订阅（Pub/Sub）和点对点（P2P）消息模型
* 在一个队列中可靠的先进先出（FIFO）和严格的顺序传递
* 支持拉（pull）和推（push）两种消息模式
* 单一队列百万消息的堆积能力（前提是足够的内存、硬盘空间）
* 支持多种消息协议，如 JMS、MQTT 等
* 分布式高可用的部署架构,满足至少一次消息传递语义
* 提供 docker 镜像用于隔离测试和云集群部署
* 提供配置、指标和监控等功能丰富的 Dashboard

RcoketMQ 是阿里巴巴开源的项目之一，被广泛的使用到实际当中。RcoketMQ 3+已经是高性能稳定的版本了。目前最新的版本是RcoketMQ 4.3，帶來了很多新特性。

## 使用mq的优势

消息队列作为高并发系统的核心组件之一，能够帮助业务系统解构提升开发效率和系统稳定性。主要具有以下优势：

* 削峰填谷（主要解决瞬时写压力大于应用服务能力导致消息丢失、系统奔溃等问题）
* 系统解耦（解决不同重要程度、不同能力级别系统之间依赖导致一死全死）
* 提升性能（当存在一对多调用时，可以发一条消息给消息系统，让消息系统通知相关系统）
* 蓄流压测（线上有些链路不好压测，可以通过堆积一定量消息再放开来压测）

## Rocketmq的优势

目前主流的MQ主要是Rocketmq、kafka、Rabbitmq，Rocketmq相比于Rabbitmq、kafka具有主要优势特性有：

* + 支持事务型消息（消息发送和DB操作保持两方的最终一致性，rabbitmq和kafka不支持）
  + 支持结合rocketmq的多个系统之间数据最终一致性（多方事务，二方事务是前提）
  + 支持18个级别的延迟消息（rabbitmq和kafka不支持）
  + 支持指定次数和时间间隔的失败消息重发（kafka不支持，rabbitmq需要手动确认）
  + 支持consumer端tag过滤，减少不必要的网络传输（rabbitmq和kafka不支持）
  + 支持重复消费（rabbitmq不支持，kafka支持）

## 基本概念

* Producer

消息生产者，生产者的作用就是将消息发送到 MQ，生产者本身既可以产生消息，如读取文本信息等。也可以对外提供接口，由外部应用来调用接口，再由生产者将收到的消息发送到 MQ。

* Producer Group

生产者组，简单来说就是多个发送同一类消息的生产者称之为一个生产者组。在编写程序的时候可以不用关心，只要知道有这么一个概念即可。Group主要是rocket自己维护使用。

* Consumer

消息消费者，简单来说，消费 MQ 上的消息的应用程序就是消费者，至于消息是否进行逻辑处理，还是直接存储到数据库等取决于业务需要。

* Consumer Group

消费者组，和生产者类似，消费同一类消息的多个 consumer 实例组成一个消费者组。在编写程序的时候可以不用关心，只要知道有这么一个概念即可。Group主要是rocket自己维护使用。

* Topic

Topic 是一种消息的逻辑分类，比如说你有订单类的消息，也有库存类的消息，那么就需要进行分类，一个是订单 Topic 存放订单相关的消息，一个是库存 Topic 存储库存相关的消息。

非广播模式下，一个生产者向一个topic中发送消息后，只会被一个消费者消费一次。

* Message

Message 是消息的载体。一个 Message 必须指定 topic，相当于寄信的地址。Message 还有一个可选的 tag 设置，以便消费端可以基于 tag 进行过滤消息。也可以添加额外的键值对，例如你需要一个业务 key 来查找 broker 上的消息，方便在开发过程中诊断问题。

* Tag

标签可以被认为是对 Topic 进一步细化。一般在相同业务模块中通过引入标签来标记不同用途的消息。

比如生产者的标签包括{"tagA", "tagB", "tagC", "tagD", "tagE"}。每次发送消息的时候，都会指定消息的标签。

场景有两个

第一：消费者可以订阅一部分，比如"tagA"，这样就只消费"tagA"的数据，其他四个标签的消息在进入消费者的时候自动被销毁。非tagA的消息不会进入我们自己编写的程序中

第一：消费者全部订阅，所有的消息都会进入我们编写的程序中，我们可以使用messageExt.getTags() 在程序中进行逻辑判断

* Broker

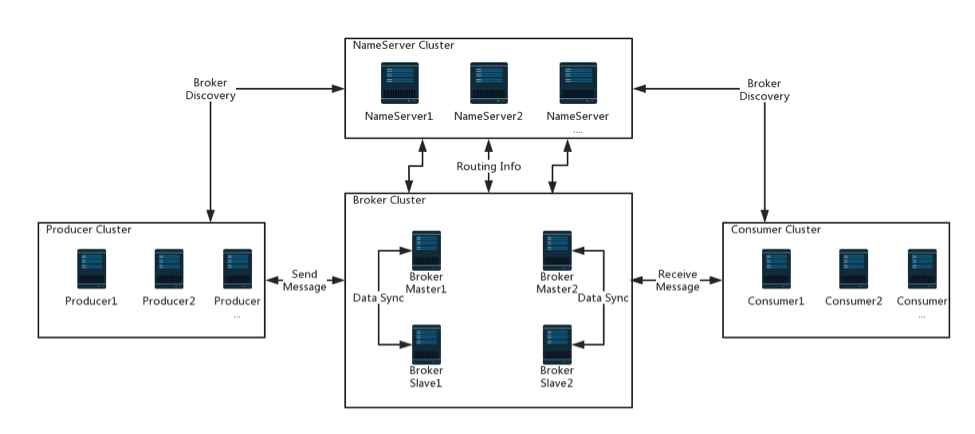
Broker 是 RocketMQ 系统的主要角色，其实就是前面一直说的 MQ。Broker 接收来自生产者的消息，储存以及为消费者拉取消息的请求做好准备。

我们所说的rocket集群就是启动多个Broker 实现的

* Name Server

Name Server 为 producer 和 consumer 提供路由信息。

## RocketMQ架构



由这张图可以看到有四个集群，分别是 NameServer 集群、Broker 集群、Producer 集群和 Consumer 集群：

NameServer: 提供轻量级的服务发现和路由。 每个 NameServer 记录完整的路由信息，提供等效的读写服务，并支持快速存储扩展。

Broker: 通过提供轻量级的 Topic 和 Queue 机制来处理消息存储,同时支持推（push）和拉（pull）模式以及主从结构的容错机制。

Producer：生产者，产生消息的实例，拥有相同 Producer Group 的 Producer 组成一个集群。

Consumer：消费者，接收消息进行消费的实例，拥有相同 Consumer Group 的

Consumer 组成一个集群。

简单说明一下图中箭头含义，从 Broker 开始，Broker Master1 和 Broker Slave1 是主从结构，它们之间会进行数据同步，即 Date Sync。同时每个 Broker 与

NameServer 集群中的所有节

点建立长连接，定时注册 Topic 信息到所有 NameServer 中。

Producer 与 NameServer 集群中的其中一个节点（随机选择）建立长连接，定期从 NameServer 获取 Topic 路由信息，并向提供 Topic 服务的 Broker Master 建立长连接，且定时向 Broker 发送心跳。Producer 只能将消息发送到 Broker master，但是 Consumer 则不一样，它同时和提供 Topic 服务的 Master 和 Slave

建立长连接，既可以从 Broker Master 订阅消息，也可以从 Broker Slave 订阅消息。

# 安装与部署

这里我们使用的是rocketmq-all-4.3.0-bin-release.zip

需要以下的依赖

64bit OS, Linux/Unix/Mac is recommended;

64bit JDK 1.8+;

Maven 3.2.x

官方提供安装的方式有两个。

## 源码安装

### 下载源码并编译

> unzip rocketmq-all-4.3.0-source-release.zip

> cd rocketmq-all-4.3.0/

> mvn -Prelease-all -DskipTests clean install -U

> cd distribution/target/apache-rocketmq

### 设置环境变量

Window： ROCKETMQ\_HOME

D:\java\rocketmq-all-4.3.0-bin-release

Linux ： /etc/profile

### Start Name Server

> nohup sh bin/mqnamesrv &

> tail -f ~/logs/rocketmqlogs/namesrv.log

The Name Server boot success...

### Start Broker

> nohup sh bin/mqbroker -n localhost:9876 &

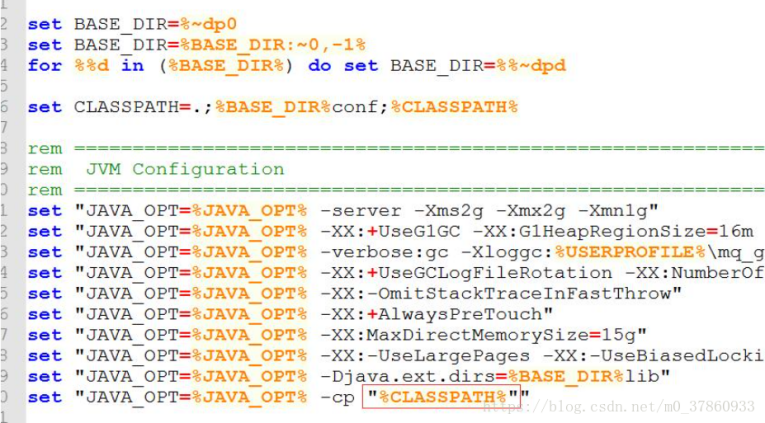
> tail -f ~/logs/rocketmqlogs/broker.log

The broker[%s, 172.30.30.233:10911] boot success...

自动创建topic：nohup sh bin/mqbroker -n localhost:9876 autoCreateTopicEnable=true &

Window:可能出现异常错误: 找不到或无法加载主类 xxxxxx

打开runbroker.cmd，然后将‘%CLASSPATH%’加上英文双引号。保存并重新执行start语句



### Send & Receive Messages

> export NAMESRV\_ADDR=localhost:9876

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer

SendResult [sendStatus=SEND\_OK, msgId= ...

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer

ConsumeMessageThread\_%d Receive New Messages: [MessageExt...

### Shutdown Servers

> sh bin/mqshutdown broker

The mqbroker(36695) is running...

Send shutdown request to mqbroker(36695) OK

> sh bin/mqshutdown namesrv

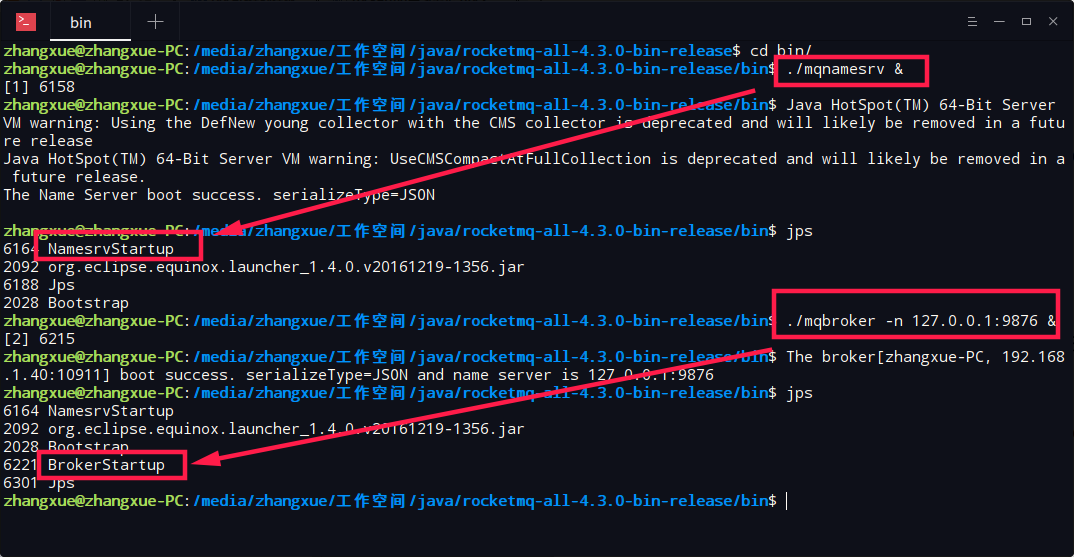
The mqnamesrv(36664) is running...

Send shutdown request to mqnamesrv(36664) OK

## 解压版安装

官方提供编译好的版本下载，依赖的环境与上面的一样。

除了编译之外其他的方式都是一样的。启动之后最好使用jps查询mq的进程是否正常启动。



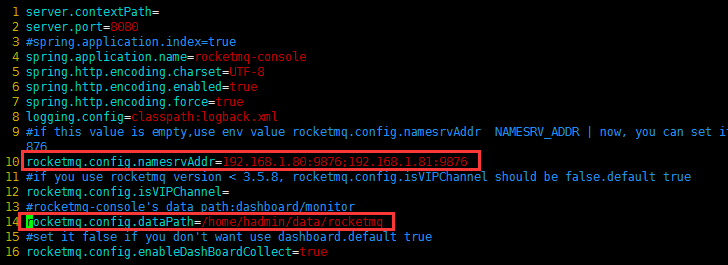
## RocketMQ插件部署

### 下载

地址：<https://github.com/apache/rocketmq-externals.git>

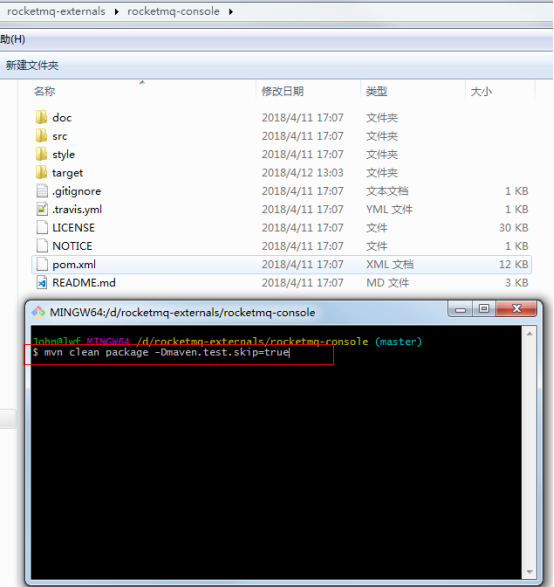
直接clone可能非常慢，建议在页面里下载。

下载完成之后，进入‘rocketmq-externals\rocketmq-console\src\main\resources’文件夹，打开‘application.properties’进行配置。



### 编译启动

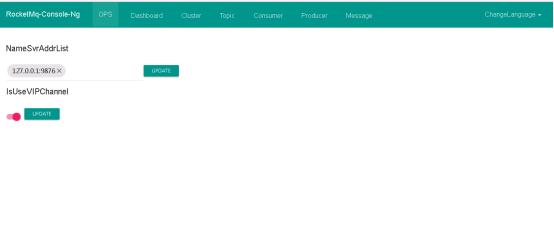
进入‘\rocketmq-externals\rocketmq-console’文件夹，执行‘mvn clean package -Dmaven.test.skip=true’，编译生成。



编译成功之后，Cmd进入‘target’文件夹，执行‘java -jar rocketmq-console-ng-1.0.0.jar’，启动‘rocketmq-console-ng-1.0.0.jar’。

### 测试

浏览器中输入‘127.0.0.1：配置端口’，成功后即可查看。



## 启动常见的问题

1. 报错如下：

: command not found

: command not found

mqnamesrv: line 35: syntax error: unexpected end of file

在bin目录下调用dos2unix \*将所有文件转化为unix格式，再次调用nohup sh mqnamesrv &

2.报错如下：

/home/hadoop/alibaba-rocketmq

Invalid initial heap size: -Xms4g

The specified size exceeds the maximum representable size.

Could not create the Java virtual machine.

由于安装的JDK版本为32位，4g超过了JDK所支持的最大内存，不过32位JDK也无法发挥出RocketMQ的优势，换成64位JDK

[hadoop@hadoop bin]$ jps

17682 NamesrvStartup

17800 Jps

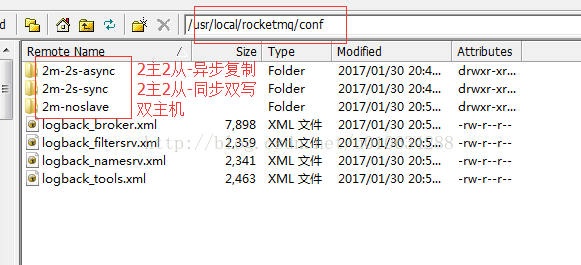
NameServer监听端口：9876

nettyServerConfig.setListenPort(9876);

如果服务器内存不够，可以修改runserver.sh脚本（mqnamesrv文件中通过runserver.sh脚本调用Name Server的主函数com.alibaba.rocketmq.namesrv.NamesrvStartup启动Name Server）中的JAVA\_OPT\_1参数

JAVA\_OPT\_1="-server -Xms4g -Xmx4g -Xmn2g -XX:PermSize=128m -XX:MaxPermSize=320m"

# 集群部署方式



异步复制和同步双写主要是主和从的关系。消息需要实时消费的，就需要采用主从模式部署

异步复制:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就算从producer端发送成功了，然后通过异步复制的方法将数据复制到从节点

同步双写:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就并不算从producer端发送成功了，需要通过同步双写的方法将数据同步到从节点后， 才算数据发送成功

## 单 master 模式

也就是只有一个 master 节点，称不上是集群，一旦这个 master 节点宕机，那么整个服务就不可用，适合个人学习使用。

## 多 master 模式

多个 master 节点组成集群，单个 master 节点宕机或者重启对应用没有影响。

优点：所有模式中性能最高

缺点：单个 master 节点宕机期间，未被消费的消息在节点恢复之前不可用，消息的实时性就受到影响。

注意：使用同步刷盘可以保证消息不丢失，同时 Topic 相对应的 queue 应该分布在集群中各个节点，而不是只在某各节点上，否则，该节点宕机会对订阅该 topic 的应用造成影响。

## 多 master 多 slave 异步复制模式

在多 master 模式的基础上，每个 master 节点都有至少一个对应的 slave。master

节点可读可写，但是 slave 只能读不能写，类似于 mysql 的主备模式。

优点： 在 master 宕机时，slave升级为master，消费者可以从 slave 读取消息，消息的实时性不会受影响，性能几乎和多 master 一样。

缺点：使用异步复制的同步方式有可能会有消息丢失的问题。

## 多 master 多 slave 同步双写模式

同多 master 多 slave 异步复制模式类似，区别在于 master 和 slave 之间的数据同步方式。

优点：同步双写的同步模式能保证数据不丢失。

缺点：发送单个消息 RT 会略长，性能相比异步复制低10%左右。

刷盘策略：同步刷盘和异步刷盘（指的是节点自身数据是同步还是异步存储）

同步方式：同步双写和异步复制（指的一组 master 和 slave 之间数据的同步）

注意：要保证数据可靠，需采用同步刷盘和同步双写的方式，但性能会较其他方式低

## 例子：部署一Master一Slave，集群采用异步复制方式

### Master：

[hadoop@hadoop bin]$ nohup sh mqbroker -n "192.168.58.163:9876" -c ../conf/2m-2s-async/broker-a.properties &

[2] 25493

[hadoop@hadoop bin]$ nohup: appending output to “nohup.out”

[hadoop@hadoop bin]$ cat nohup.out

load config properties file OK, ../conf/2m-2s-async/broker-a.properties

The broker[broker-a, 192.168.58.163:10911] boot success. and name server is 192.168.58.163:9876

[hadoop@hadoop bin]$ jps

25500 BrokerStartup

25545 Jps

17682 NamesrvStartup

### Slave

[hadoop@hadoop bin]$ nohup sh mqbroker -n "192.168.58.163:9876" -c ../conf/2m-2s-async/broker-a-s.properties &

[1] 1974

[hadoop@hadoop bin]$ nohup: appending output to “nohup.out”

[hadoop@hadoop bin]$ cat nohup.out

load config properties file OK, ../conf/2m-2s-async/broker-a-s.properties

The broker[broker-a, 192.168.58.164:10911] boot success. and name server is 192.168.58.163:9876

[hadoop@hadoop bin]$ jps

2071 Jps

1981 BrokerStartup

Broker监听端口：10911

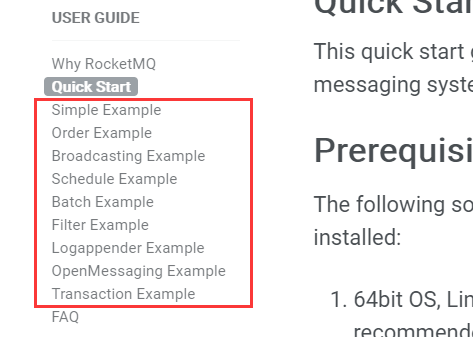
nettyServerConfig.setListenPort(10911);

如果服务器内存不够，可以修改runbroker.sh脚本（mqbroker文件中通过runbroker.sh脚本调用Broker的主函数com.alibaba.rocketmq.broker.BrokerStartup启动Broker）的JAVA\_OPT\_1参数

JAVA\_OPT\_1="-server -Xms4g -Xmx4g -Xmn2g -XX:PermSize=128m -XX:MaxPermSize=320m"

# 简单案例

现在最新版本，官方的案例新增了多种类型的消息



## 项目依赖

推荐使用jdk1.8

<dependencies>

<dependency>

<groupId>org.apache.rocketmq</groupId>

<artifactId>rocketmq-client</artifactId>

<version>4.3.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

## 三种方式发送消息

RocketMQ以三种方式发送消息：可靠的同步，可靠的异步和单向传输。

### 可靠的同步

/\*\*

\* @Project: 20180810\_rocketmq

\* @Title: SyncProducer

\* @Description: 同步传输

\* 应用：可靠的同步传输用于广泛的场景，如重要的通知消息，短信通知，短信营销系统等

\* @author: zhangxue

\* @date: 2018年8月11日上午10:24:26

\* @company: yooli

\* @Copyright: Copyright (c) 2015

\* @version v1.0

\*/

public class SyncProducer {

public static void main(String[] args) throws Exception {

//Instantiate with a producer group name.

DefaultMQProducer producer = new DefaultMQProducer("please\_rename\_unique\_group\_name");

//Launch the instance.

producer.start();

for (int i = 0; i < 100; i++) {

//Create a message instance, specifying topic, tag and message body.

Message msg = new Message("TopicTest" /\* Topic \*/,

"TagA" /\* Tag \*/,

("Hello RocketMQ " +

i).getBytes(RemotingHelper.DEFAULT\_CHARSET) /\* Message body \*/

);

//Call send message to deliver message to one of brokers.

SendResult sendResult = producer.send(msg);

System.out.printf("%s%n", sendResult);

}

//Shut down once the producer instance is not longer in use.

producer.shutdown();

}

}

### 可靠的异步传输

/\*\*

\* @Project: 20180810\_rocketmq

\* @Title: AsyncProducer

\* @Description: 异步传输

\* 应用：异步传输通常用于响应时间敏感的业务场景

\* @author: zhangxue

\* @date: 2018年8月11日上午10:24:48

\* @company: yooli

\* @Copyright: Copyright (c) 2015

\* @version v1.0

\*/

public class AsyncProducer {

public static void main(String[] args) throws Exception {

//Instantiate with a producer group name.

DefaultMQProducer producer = new DefaultMQProducer("ExampleProducerGroup");

//Launch the instance.

producer.start();

producer.setRetryTimesWhenSendAsyncFailed(0);

for (int i = 0; i < 100; i++) {

final int index = i;

//Create a message instance, specifying topic, tag and message body.

Message msg = new Message("TopicTest",

"TagA",

"OrderID188",

"Hello world".getBytes(RemotingHelper.DEFAULT\_CHARSET));

producer.send(msg, new SendCallback() {//回调函数

@Override

public void onSuccess(SendResult sendResult) {

System.out.printf("%-10d OK %s %n", index,

sendResult.getMsgId());

}

@Override

public void onException(Throwable e) {

System.out.printf("%-10d Exception %s %n", index, e);

e.printStackTrace();

}

});

}

//Shut down once the producer instance is not longer in use.

producer.shutdown();

}

}

### 单向传输

/\*\*

\* @Project: 20180810\_rocketmq

\* @Title: OnewayProducer

\* @Description: 单向传输

\* 应用：单向传输用于需要中等可靠性的情况，例如日志收集。

\* @author: zhangxue

\* @date: 2018年8月11日上午10:54:50

\* @company: yooli

\* @Copyright: Copyright (c) 2015

\* @version v1.0

\*/

public class OnewayProducer {

public static void main(String[] args) throws Exception{

//Instantiate with a producer group name.

DefaultMQProducer producer = new DefaultMQProducer("ExampleProducerGroup");

//Launch the instance.

producer.start();

for (int i = 0; i < 100; i++) {

//Create a message instance, specifying topic, tag and message body.

Message msg = new Message("TopicTest" /\* Topic \*/,

"TagA" /\* Tag \*/,

("Hello RocketMQ " +

i).getBytes(RemotingHelper.DEFAULT\_CHARSET) /\* Message body \*/

);

//Call send message to deliver message to one of brokers.

producer.sendOneway(msg);

}

//Shut down once the producer instance is not longer in use.

producer.shutdown();

}

}

## 模拟同步发送

这是生产中使用比较多的方式

### 生产者

/\*\*

\* @Project: 20180810\_rocketmq

\* @Title: OnewayProducer

\* @Description:

\* 下面的例子，模拟的是发送/接收全局和分区排序的消息

\* @author: zhangxue

\* @date: 2018年8月11日上午10:25:11

\* @company: yooli

\* @Copyright: Copyright (c) 2015

\* @version v1.0

\*/

public class OrderedProducer {

public static void main(String[] args) throws Exception {

//声明并初始化一个producer group name

//需要一个producer group名字作为构造方法的参数，这里为example\_group\_name

DefaultMQProducer producer = new DefaultMQProducer("example\_group\_name");

//设置NameServer地址,此处应改为实际NameServer地址，多个地址之间用；分隔

//NameServer的地址必须有，但是也可以通过环境变量的方式设置，不一定非得写死在代码里

//producer.setNamesrvAddr("10.1.54.121:9876;10.1.54.122:9876");

producer.setNamesrvAddr("127.0.0.1:9876");

//调用start()方法启动一个producer实例

producer.start();

// 给topic设置5个tag

String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};

// 发送10条消息到Topic为TopicTestZX，tag为TagA ... TagE， 消息内容为“Hello RocketMQ”拼接上i的值

for (int i = 0; i < 10; i++) {

int orderId = i % 10;

// 创建一条mq消息

Message msg = new Message("TopicTestZX", tags[i % tags.length], "KEY" + i,

("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT\_CHARSET));

//调用producer的send()方法发送消息

//这里调用的是同步的方式，所以会有返回结果

SendResult sendResult = producer.send(msg, new MessageQueueSelector() {

@Override

public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {

Integer id = (Integer) arg;

int index = id % mqs.size();

return mqs.get(index);

}

}, orderId);

System.out.printf("%s%n", sendResult);

}

//server shutdown

producer.shutdown();

}

}

### 消费者

public class OrderedConsumer {

public static void main(String[] args) throws Exception {

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("example\_group\_name");

consumer.setNamesrvAddr("127.0.0.1:9876");

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME\_FROM\_FIRST\_OFFSET);

consumer.subscribe("TopicTestZX", "TagA || TagC || TagD");

consumer.registerMessageListener(new MessageListenerOrderly() {

AtomicLong consumeTimes = new AtomicLong(0);

@Override

public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,

ConsumeOrderlyContext context) {

context.setAutoCommit(false);

System.out.printf(Thread.currentThread().getName() + " Receive New Messages: " + msgs + "%n");

this.consumeTimes.incrementAndGet();

if ((this.consumeTimes.get() % 2) == 0) {

return ConsumeOrderlyStatus.SUCCESS;

} else if ((this.consumeTimes.get() % 3) == 0) {

return ConsumeOrderlyStatus.ROLLBACK;

} else if ((this.consumeTimes.get() % 4) == 0) {

return ConsumeOrderlyStatus.COMMIT;

} else if ((this.consumeTimes.get() % 5) == 0) {

context.setSuspendCurrentQueueTimeMillis(3000);

return ConsumeOrderlyStatus.SUSPEND\_CURRENT\_QUEUE\_A\_MOMENT;

}

return ConsumeOrderlyStatus.SUCCESS;

}

});

consumer.start();

System.out.printf("Consumer Started.%n");

}

}

## 广播模式

广播正在向主题的所有订阅者发送消息。如果您希望所有订阅者都收到有关主题的消息，则广播是一个不错的选择。

### 生产者

public class BroadcastProducer {

public static void main(String[] args) throws Exception {

DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");

producer.setNamesrvAddr("127.0.0.1:9876");

producer.start();

for (int i = 0; i < 100; i++){

Message msg = new Message("TopicTest",

"TagA",

"OrderID188",

"Hello world".getBytes(RemotingHelper.DEFAULT\_CHARSET));

SendResult sendResult = producer.send(msg);

System.out.printf("%s%n", sendResult);

}

producer.shutdown();

}

}

### 消费者

public class BroadcastConsumer {

public static void main(String[] args) throws Exception {

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("example\_group\_name");

consumer.setNamesrvAddr("127.0.0.1:9876");

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME\_FROM\_FIRST\_OFFSET);

//set to broadcast mode

consumer.setMessageModel(MessageModel.BROADCASTING);

consumer.subscribe("TopicTest", "TagA || TagC || TagD");

consumer.registerMessageListener(new MessageListenerConcurrently() {

@Override

public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,

ConsumeConcurrentlyContext context) {

System.out.printf(Thread.currentThread().getName() + " Receive New Messages: " + msgs + "%n");

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

}

});

consumer.start();

System.out.printf("Broadcast Consumer Started.%n");

}

}

# 消费者-consumer

## 消息获取方式

系统接收消息方被称为消费者-consumer，获取消息的方式有两种

* DefultMQPushConsumerImpl：默认的方式，mq主动推送消息给客户端，提高了消息的实时处理速度
* DefaultMQPullConsumerImpl：由客户端主动拉取消息，相比默认的方式，存在一定的延迟性

## 消息消费模式

主要分为：集群模式 和 广播模式

集群模式:

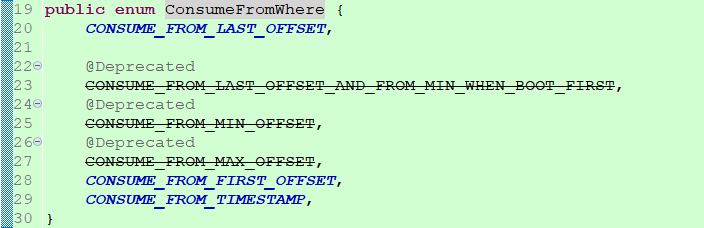
* 消费进度保存到broker上，也就是consumerOffset.json文件保存在mq的服务器上
* 订阅同一个topic的consumer（在同一个ConsumerGroup下）只能消费一次
* 消息消费失败了，consumer会把消息发送会broker，broker会根据失败的原因、次数，设置不同级别的delayLevel进行消息重发
* 相同的topic，不同的ConsumerGroup可以伪装成 广播模式，从而达到所有的consumer都能接受到消息的目的

广播模式

* 消费进度保存到consumer的服务器上，也就是这个consumerOffset.json文件
* 所有的consumer都会接受到topic的消息
* 消息消费失败后，这条消息会被丢弃，不会发送给broker重新投递

## 消费方式

Consumer可以设置自己怎么消费消息： 通过设置ConsumerFormWhere



DefaultMQPushConsumerImpl内部有一个类RebalancePushImpl，首先计算客户端需要拉取队列，然后到broker获取消费进度，获取消费offset代码在computePullFromWhere方法





首先看CONSUME\_FROM\_LAST\_OFFSET的逻辑，lastOffset >= 0，意味着broker端有消费进度了，说明之前已经启动且消费了一些消息，那么就从返回的offset开始消费。当-1 == lastOffset，如果为重试队列，从头消费，为普通队列则从最大offset处消费。

CONSUME\_FROM\_TIMESTAMP，如果为第一次启动，即-1 == lastOffset时，为普通队列的话，从设置的时间点开始消费，如果未设置时间点，默认从半小时前开始消费。

## 拉消息

Consumer的启动源码如下:DefaultMQPushConsumerImpl

public synchronized void start() throws MQClientException {

switch (this.serviceState) {

case CREATE\_JUST:

log.info("the consumer [{}] start beginning. messageModel={}, isUnitMode={}", this.defaultMQPushConsumer.getConsumerGroup(),

this.defaultMQPushConsumer.getMessageModel(), this.defaultMQPushConsumer.isUnitMode());

this.serviceState = ServiceState.START\_FAILED;

this.checkConfig();

this.copySubscription();

if (this.defaultMQPushConsumer.getMessageModel() == MessageModel.CLUSTERING) {

this.defaultMQPushConsumer.changeInstanceNameToPID();

}

this.mQClientFactory = MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQPushConsumer, this.rpcHook);

this.rebalanceImpl.setConsumerGroup(this.defaultMQPushConsumer.getConsumerGroup());

this.rebalanceImpl.setMessageModel(this.defaultMQPushConsumer.getMessageModel());

this.rebalanceImpl.setAllocateMessageQueueStrategy(this.defaultMQPushConsumer.getAllocateMessageQueueStrategy());

this.rebalanceImpl.setmQClientFactory(this.mQClientFactory);

this.pullAPIWrapper = new PullAPIWrapper(

mQClientFactory,

this.defaultMQPushConsumer.getConsumerGroup(), isUnitMode());

this.pullAPIWrapper.registerFilterMessageHook(filterMessageHookList);

if (this.defaultMQPushConsumer.getOffsetStore() != null) {

this.offsetStore = this.defaultMQPushConsumer.getOffsetStore();

} else {

switch (this.defaultMQPushConsumer.getMessageModel()) {

case BROADCASTING:

this.offsetStore = new LocalFileOffsetStore(this.mQClientFactory, this.defaultMQPushConsumer.getConsumerGroup());

break;

case CLUSTERING:

this.offsetStore = new RemoteBrokerOffsetStore(this.mQClientFactory, this.defaultMQPushConsumer.getConsumerGroup());

break;

default:

break;

}

this.defaultMQPushConsumer.setOffsetStore(this.offsetStore);

}

this.offsetStore.load();

if (this.getMessageListenerInner() instanceof MessageListenerOrderly) {

this.consumeOrderly = true;

this.consumeMessageService =

new ConsumeMessageOrderlyService(this, (MessageListenerOrderly) this.getMessageListenerInner());

} else if (this.getMessageListenerInner() instanceof MessageListenerConcurrently) {

this.consumeOrderly = false;

this.consumeMessageService =

new ConsumeMessageConcurrentlyService(this, (MessageListenerConcurrently) this.getMessageListenerInner());

}

this.consumeMessageService.start();

boolean registerOK = mQClientFactory.registerConsumer(this.defaultMQPushConsumer.getConsumerGroup(), this);

if (!registerOK) {

this.serviceState = ServiceState.CREATE\_JUST;

this.consumeMessageService.shutdown();

throw new MQClientException("The consumer group[" + this.defaultMQPushConsumer.getConsumerGroup()

+ "] has been created before, specify another name please." + FAQUrl.suggestTodo(FAQUrl.GROUP\_NAME\_DUPLICATE\_URL),

null);

}

mQClientFactory.start();

log.info("the consumer [{}] start OK.", this.defaultMQPushConsumer.getConsumerGroup());

this.serviceState = ServiceState.RUNNING;

break;

case RUNNING:

case START\_FAILED:

case SHUTDOWN\_ALREADY:

throw new MQClientException("The PushConsumer service state not OK, maybe started once, "

+ this.serviceState

+ FAQUrl.suggestTodo(FAQUrl.CLIENT\_SERVICE\_NOT\_OK),

null);

default:

break;

}

this.updateTopicSubscribeInfoWhenSubscriptionChanged();

this.mQClientFactory.checkClientInBroker();

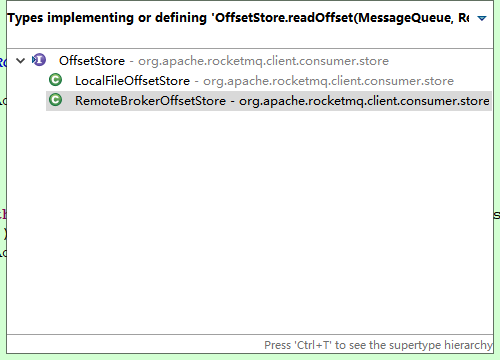
this.mQClientFactory.sendHeartbeatToAllBrokerWithLock();

this.mQClientFactory.rebalanceImmediately();

}

### 加载消费进度this.offsetStore.load();

根据consumer配置的消费模式，广播模式实例化LocalFileOffsetStore， 集群模式实例化RemoteBrokerOffsetStore。之后consumer对offset的读写操作都是直接操作加载到内存的实例化对象。Consumer会单独开启一个线程5S一次的频次将offset同步到磁盘上。



### 启动服务this.consumeMessageService.start();

启动消费者的业务服务，实现类是ConsumeMessageConcurrentlyService，多线程并发服务

### 连接broker mQClientFactory.start();

mQClientFactory的启动会建立和broker通道，定时任务，拉消息服务，负载均衡服务。由于拉消息的请求是由负载均衡发起，所以先说负载均衡服务。

负载均衡服务由RebalanceService线程每隔20s做一次，跟踪代码最终会调用到

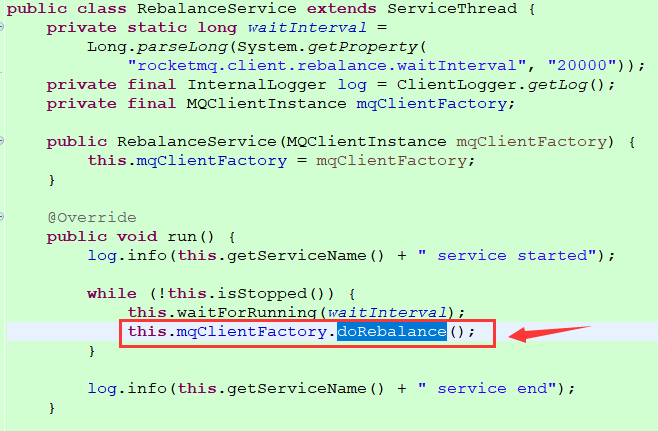
mQClientFactory.start();



this.rebalanceService.start();

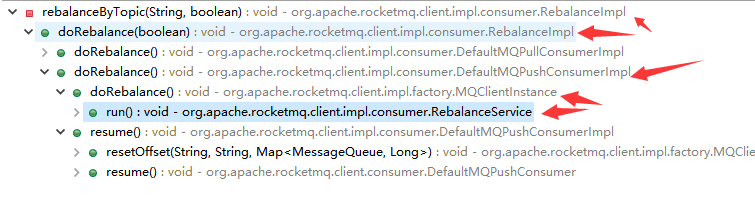
这个rebalanceService实现的RebalanceService： this.rebalanceService = new RebalanceService(this);

RebalanceService调用了doReblanceff1

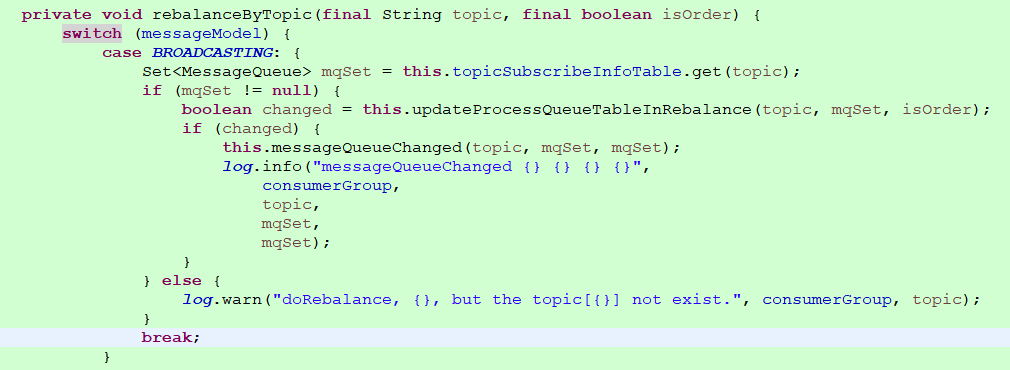


调用链如下：

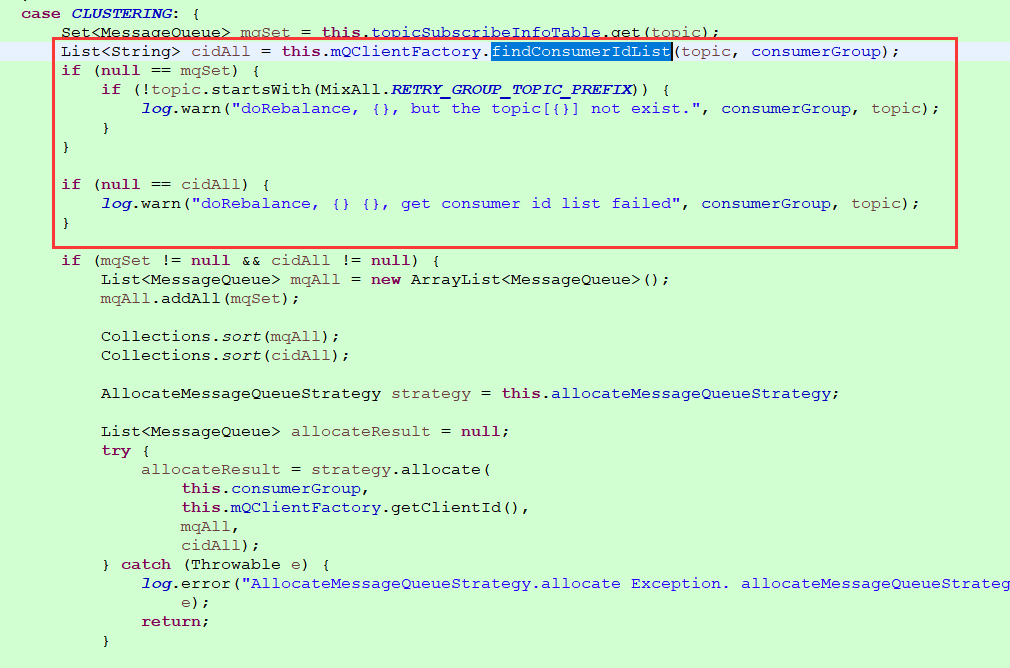
负载均衡服务由RebalanceService线程每隔20s做一次，最终调用到RebalanceImpl#rebalanceByTopic方法

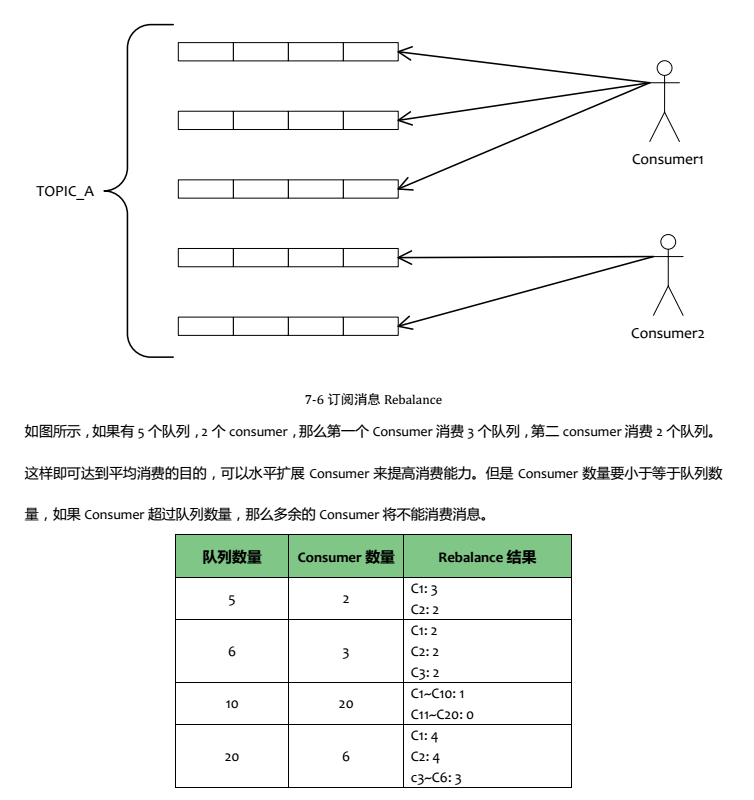


广播模式，由于所有consumer都需要收到消息，所以不存在负载均衡策略



集群模式，首先通过topic和consumerGroup获取consumer列表，然后分配拉取消息队列，默认为平均分配策略AllocateMessageQueueAveragely。

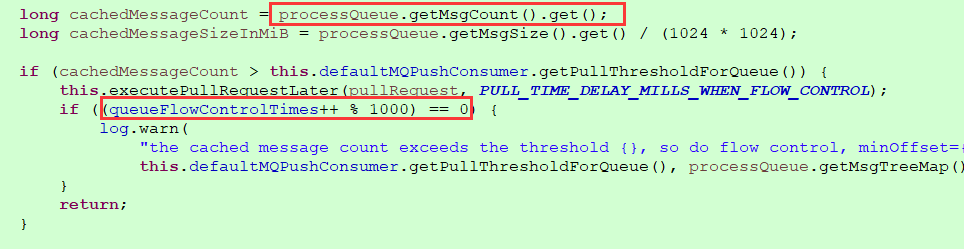




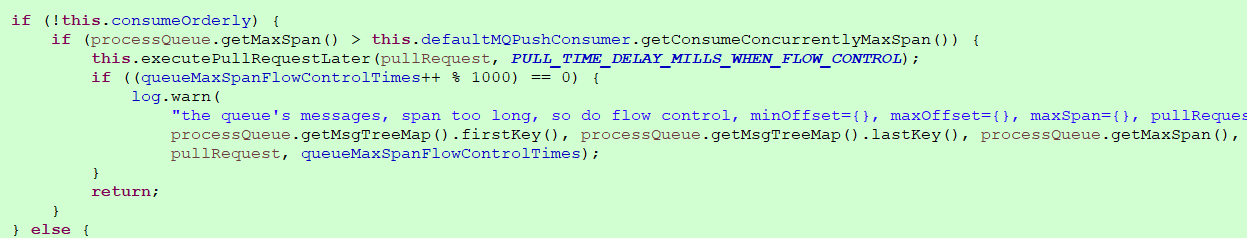
通过上面的测试数据，可以看到AllocateMessageQueueAveragely的allocate方法，就是以上结论。

## 拉取消息（DefaultMQPushConsumerImpl）

得到当前consumer需要拉取的消息队列后，到RebalanceImpl#updateProcessQueueTableInRebalance进行拉取数据请求PullRequest构造，到DefaultMQPushConsumerImpl#pullMessage进行拉取消息前的逻辑整合，最终通过mQClientFactory内部的mQClientAPIImpl通道到broker异步拉取数据。

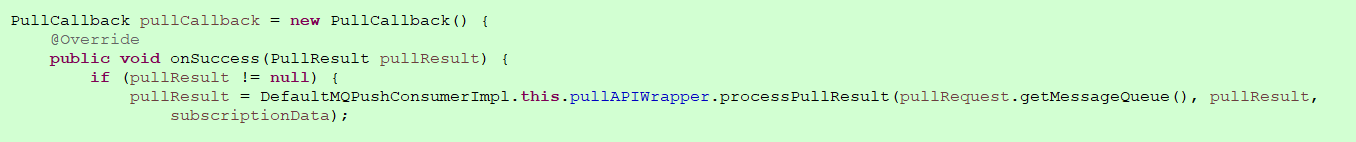


由于拉取的消息，会放到本地队列ProcessQueue进行处理，当发现本地队列大小超过1000，就延迟50ms再拉取。

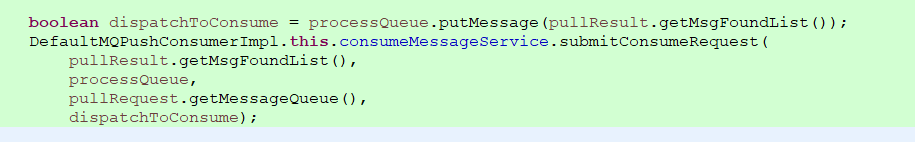


由于消息以offset为key，放入本地队列ProcessQueue的TreeMap中，所以这里有一步span检查。当span值(即this.msgTreeMap.lastKey() - this.msgTreeMap.firstKey())大于2000时，延迟拉取。 由于业务关系，消息消费快慢无法保证，如果offset大的消息处理的快，本地队列就会积压offset小的消息，所以span的值可能会越来越大。

由于是异步拉取消息，所以这里需要构造一个PullCallback对象。onSuccess方法内



对拉取结果进行处理，如果有消息，反序列化，再进行一步tag的比较去重。前面章节，有提到过，broker的ConsumeQueue中保存tag的hashcode值，所以consumer的此步去重是保证消息的准确性。



然后将消息放入本地队列，并通过submitConsumeRequest方法构造ConsumeRequest，由于consumeBatchSize为1，提交任务到consumeExecutor线程池(20个线程)，每线程每消息并发处理，ConsumeRequest会调用consumeMessageService触发MessageListener#consumeMessage，执行业务处理。

在ConsumeRequest#run业务处理完后，执行processConsumeResult方法

## 拉取消息（DefaultMQPullConsumerImpl）

consumer是长轮询拉消息，当consumer拉消息时，broker端如果没有新消息，broker会通过PullRequestHoldService服务hold住这个请求

if (brokerAllowSuspend && hasSuspendFlag) {

long pollingTimeMills = suspendTimeoutMillisLong;

if (!this.brokerController.getBrokerConfig().isLongPollingEnable()) {

pollingTimeMills = this.brokerController.getBrokerConfig().getShortPollingTimeMills();

}

String topic = requestHeader.getTopic();

long offset = requestHeader.getQueueOffset();

int queueId = requestHeader.getQueueId();

PullRequest pullRequest = new PullRequest(request, channel, pollingTimeMills,

this.brokerController.getMessageStore().now(), offset, subscriptionData);

this.brokerController.getPullRequestHoldService().suspendPullRequest(topic, queueId, pullRequest);

response = null;

break;

}

Broker通过ReputMessageService异步构建ConsumeQueue并通过注册的MessageArrivingListener通知PullRequestHoldService#notifyMessageArriving达到有消息，立即推送给consumer。ReputMessageService#doReput:

if (BrokerRole.SLAVE != DefaultMessageStore.this.getMessageStoreConfig().getBrokerRole()

&& DefaultMessageStore.this.brokerConfig.isLongPollingEnable()) {

DefaultMessageStore.this.messageArrivingListener.arriving(dispatchRequest.getTopic(),

dispatchRequest.getQueueId(), dispatchRequest.getConsumeQueueOffset() + 1,

dispatchRequest.getTagsCode());

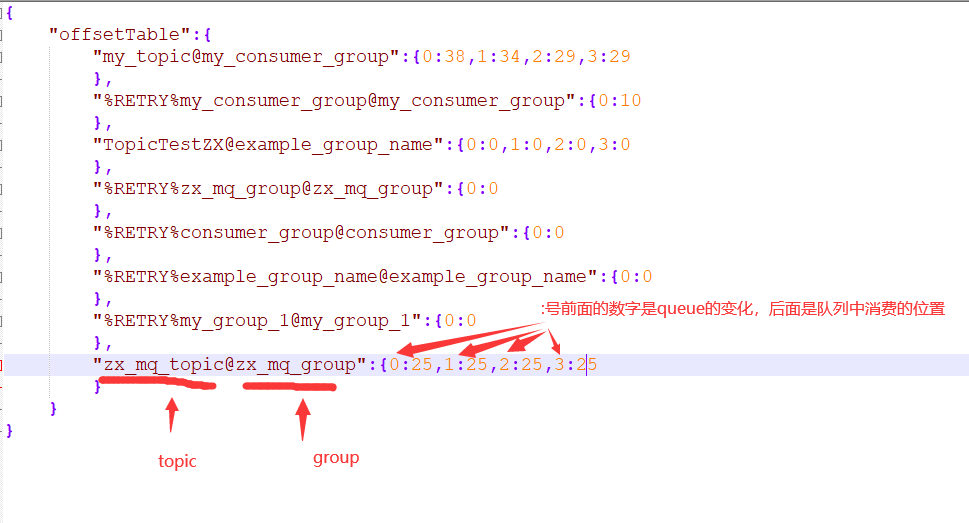
}

public void arriving(String topic, int queueId, long logicOffset, long tagsCode) {

this.pullRequestHoldService.notifyMessageArriving(topic, queueId, logicOffset, tagsCode);

}

## 消费进度consumerOffset



# 进阶

## 设置消息获取顺序

设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费。如果非第一次启动，那么按照上次消费的位置继续消费

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME\_FROM\_FIRST\_OFFSET);

## Consumer 批量消费

主要是设置在消费对，对于每次push消息的时候，设置最大消费数量。consumeMessageBatchMaxSize值默认是1。设置数量之后，在消息堆积的时候，可以在一定的程度上加速消息消费。

public static void main(String[] args) throws InterruptedException, MQClientException {

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please\_rename\_unique\_group\_name\_4");

consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");

consumer.setConsumeMessageBatchMaxSize(10);

/\*\*

\* 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>

\* 如果非第一次启动，那么按照上次消费的位置继续消费

\*/

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME\_FROM\_FIRST\_OFFSET);

consumer.subscribe("TopicTest", "\*");

consumer.registerMessageListener(new MessageListenerConcurrently() {

public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

try {

System.out.println("msgs的长度" + msgs.size());

System.out.println(Thread.currentThread().getName() + " Receive New Messages: " + msgs);

} catch (Exception e) {

e.printStackTrace();

return ConsumeConcurrentlyStatus.RECONSUME\_LATER;

}

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

}

});

consumer.start();

System.out.println("Consumer Started.");

}

## 消息重试机制

### Producer端重试

也就是Producer往MQ上发消息没有发送成功，我们可以设置发送失败重试的次数，重试的场景也是比较简单的。比如下面的：

DefaultMQProducer producer = new DefaultMQProducer("please\_rename\_unique\_group\_name");

producer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");

producer.setRetryTimesWhenSendFailed(10);//失败的 情况发送10次，默认是2次

### Consumer端重试

#### 程序抛出Exception

一般重复16次 10s、30s、1分钟、2分钟、3分钟等等。触发的机制如下。一般消费出现异常向外抛出的时候，客户端程序会自动判别异常，进行重试，不需要return 状态值。如果程序中出现异常后，可以我们进行try-catch进行判断，手动return状态值进行重试与否。

上面的代码中消费异常的情况返回

return ConsumeConcurrentlyStatus.RECONSUME\_LATER;//重试

正常则返回：

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;//成功

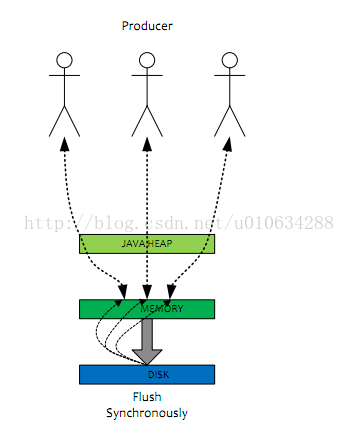
在消费者是多实例集群的时候，如果发生异常，触发的重试机制，消息是在同一个节点上重试的，不会再不同的机器上重试（这个地方是客户端程序进行重试的）。但是要是这个机器宕机了，会换到另一个节点上重试（重试次数从0开始）

#### 超时的情况

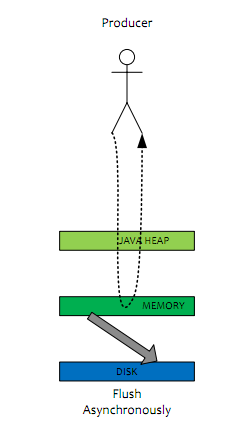
场景是程序消费消息的时间过长，这种情况MQ会无限制的发送给消费端。就是由于网络的情况，MQ发送数据之后，Consumer端并没有收到导致超时。也就是消费端没有给我返回return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;这样的就认为没有到达Consumer端。

## 刷盘方式

同步刷盘：在消息到达MQ后，RocketMQ需要将数据持久化，同步刷盘是指数据到达内存之后，必须刷到commitlog日志之后才算成功，然后返回producer数据已经发送成功。



异步刷盘：异步刷盘是指数据到达内存之后,返回producer说数据已经发送成功。然后再写入commitlog日志。



commitlog：commitlog就是来存储所有的元信息，包含消息体，类似于Mysql、Oracle的redolog,所以主要有CommitLog在，Consume Queue即使数据丢失，仍然可以恢复出来。

consumequeue：记录数据的位置,以便Consume快速通过consumequeue找到commitlog中的数据

# Mq配置

Mq的配置指的是配置broker

#broker所属的集群名字

brokerClusterName=rocketmq-cluster

#broker名字，同个集群中的每个broker应当具有它自己独有的名字

brokerName=broker-a

#设置主broker和从broker  其中0 表示 主机，>0 表示 从机

brokerId=0

#nameServer地址（地址为ip：端口），多个地址之间用分号分割

namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876

#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数

defaultTopicQueueNums=4

#是否允许 Broker 自动创建Topic，测试时可以开启，实用时关闭，生产建议关闭

autoCreateTopicEnable=true

#是否允许 Broker 自动创建订阅组，测试时可以开启，实用时关闭

#在pull形式消费时若设置了falsename会报subscription group not exist，且收不到消息，在push形式消费时没有影响

autoCreateSubscriptionGroup=true

#Broker 对外服务的监听端口

listenPort=10911

#haService中使用

haListenPort=10912

#主要用于slave同步master

fastListenPort=10909

#定时删除文件时间点，默认凌晨 4点

deleteWhen=04

#文件保留最长时间，默认 72 小时，也就是3天

fileReservedTime=72

#commitLog每个文件的大小默认1G

mapedFileSizeCommitLog=1073741824

#ConsumeQueue每个文件默认存30W条，根据业务情况调整

mapedFileSizeConsumeQueue=300000

#强制删除文件时间间隔（单位毫秒）

#destroyMapedFileIntervalForcibly=120000

#定期检查Hanged文件间隔时间（单位毫秒）

#redeleteHangedFileInterval=120000

#检测物理文件磁盘空间,磁盘空间使用率不能超过88%

diskMaxUsedSpaceRatio=88

#存储总路径

storePathRootDir=/usr/local/rocketmq/store

#commitLog 存储路径

storePathCommitLog=/usr/local/rocketmq/store/commitlog

#消费队列存储路径存储路径

storePathConsumeQueue=/usr/local/rocketmq/store/consumequeue

#消息索引存储路径

storePathIndex=/usr/local/rocketmq/store/index

#异常退出产生的文件存储路径

storeCheckpoint=/usr/local/rocketmq/store/checkpoint

#abort 文件存储路径

abortFile=/usr/local/rocketmq/store/abort

#限制的消息大小，4M

maxMessageSize=4194304

#Commitlog每次刷盘最少页数，每页4kb

flushCommitLogLeastPages=4

#ConsumeQueue每次刷盘最页数，每页4kb

#flushConsumeQueueLeastPages=2

#刷盘时间间隔（单位毫秒），此间隔时间优先级高于上面两个参数，即当时间间隔超过之后直接进行刷盘，不考虑页数问题

#flushCommitLogThoroughInterval=10000

#flushConsumeQueueThoroughInterval=60000

#Broker 的角色 （1） ASYNC\_MASTER 异步复制Master （2） SYNC\_MASTER 同步双写Master （3） SLAVE

brokerRole=ASYNC\_MASTER

#刷盘方式 （1） ASYNC\_FLUSH 异步刷盘  （2）SYNC\_FLUSH 同步刷盘

flushDiskType=ASYNC\_FLUSH

#是否开启事务check过程，消息体量大的时候可以不开启，默认为关闭状态

checkTransactionMessageEnable=false

#发消息线程池数量（如果不做配置，个数为16+（核\*线程）\*4）

#sendMessageThreadPoolNums=128

#拉消息线程池数量（如果不做配置，个数为16+（核\*线程）\*4）

#pullMessageThreadPoolNums=12