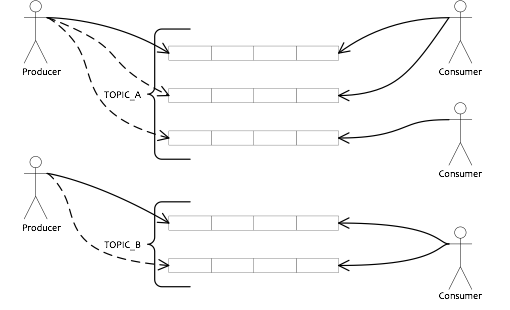
**RocketMQ**

## 一、RocketMQ 含义

**1 RocketMQ是阿里巴巴在2012年开源的分布式消息中间件**，目前已经捐赠给Apache基金会，已经于2016年11月成为 Apache 孵化项目，相信RocketMQ的未来会发挥着越来越大的作用，将有更多的开发者因此受益。

[](http://img3.tbcdn.cn/5476e8b07b923/TB1rdyvPXXXXXcBapXXXXXXXXXX)

上图是一个典型的消息中间件收发消息的模型，RocketMQ也是这样的设计，简单说来，RocketMQ具有以下特点：

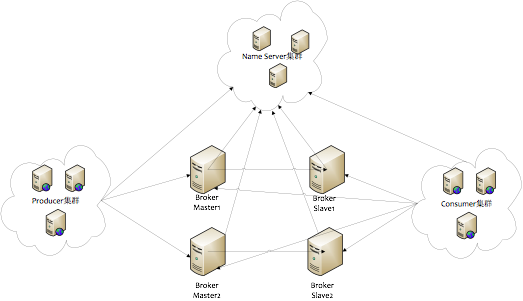
* 是一个队列模型的消息中间件，具有高性能、高可靠、高实时、分布式特点。
* Producer、Consumer、队列都可以分布式。
* Producer向一些队列轮流发送消息，队列集合称为Topic，Consumer如果做广播消费，则一个consumer实例消费这个Topic对应的所有队列，如果做集群消费，则多个Consumer实例平均消费这个topic对应的队列集合。
* 能够保证严格的消息顺序
* 提供丰富的消息拉取模式
* 高效的订阅者水平扩展能力
* 实时的消息订阅机制
* 亿级消息堆积能力
* 较少的依赖

**2 一些概念**

**--各个角色间的关系**

* RocketMq中每个Broker（master和slave）与Name Server集群中的所有节点建立长连接，定时注册Topic信息到所有Name Server。
* Name Server之间不会有任何信息交互，各自独立。
* producer和consumer随机从一个name server即可获得全部topic的路由信息。
* producer根据得到的路由信息，同master建立长连接。
* consumer根据路由信息，同master个salve都建立长连接。然后根据设置的订阅规则，选择从master或者slave订阅消息。
* **--Broker**
* 分为master和slave两个角色。master提供读写读物，salve只提供读服务。
* 为了保证可用性，需要部署多套broker，每套broker至少有1个master和1个以上的salve。
* 在同一套broker中，master和salve都是同样的brokerName，master的brokerId是0，salve的brokerId必须是非0的。
* **--同步刷盘和异步刷盘**
* 同步刷盘是说，broker在收到每个消息后，都是先要保存到硬盘上，然后再给producer确认。
* 异步刷盘就是先回复确认，然后批量保存到硬盘上。异步刷盘有更好的性能，当然也有更大的丢失消息的风险。
* **--同步复制和异步复制。**
* 是说在master和salve之间复制消息的方式。同步是说在salve也存储了消息后再答复producer。
* 异步复制是先答复producer，再去向salve复制。通过同步复制技术可以完全避免单点，同步复制势必会影响性能，适合对消息可靠性要求极高的场合，
* 例如与Money相关的应用。RocketMQ从3.0版本开始支持同步双写。
* **--MQPullConsumer和MQPushConsumer的区别**
* consumer被分为2类：MQPullConsumer和MQPushConsumer，其实本质都是拉模式（pull），即consumer轮询从broker拉取消息。
* 区别是：
* push方式里，consumer把轮询过程封装了，并注册MessageListener监听器，取到消息后，唤醒MessageListener的consumeMessage()来消费，
* 对用户而言，感觉消息是被推送（push）过来的。
* pull方式里，取消息的过程需要用户自己写，首先通过打算消费的Topic拿到MessageQueue的集合，遍历MessageQueue集合，然后针对每个
* MessageQueue批量取消息，一次取完后，记录该队列下一次要取的开始offset，直到取完了，再换另一个MessageQueue。
* 对RocketMQ使用长轮询Pull方式，可保证消息非常实时，消息实时性不低于Push的理解
* **--数据交互有两种模式**
* Push（推模式）、Pull（拉模式）。
* 推模式指的是客户端与服务端建立好网络长连接，服务方有相关数据，直接通过长连接通道推送到客户端。其优点是及时，一旦有数据变更，客户端立马能感知到；
* 拉模式指的是客户端主动向服务端发出请求，拉取相关数据。其优点是此过程由客户端发起请求，故不存在推模式中数据积压的问题。缺点是可能不够及时，对客户端来说需要考虑数据拉取相关逻辑。
* **--长轮询：**
* 轮询是说，每隔一定时间，客户端想服务端发起一次请求，服务端有数据就返回数据，没有数据就返回空，然后关闭请求。
* 长轮询，不同之处是，服务端如果此时没有数据，保持连接。等到有数据返回（相当于一种push），或者超时返回。
* 所以长轮询Pull的好处就是可以减少无效请求，保证消息的实时性，又不会造成客户端积压。
* 其他对于一个消息中间件来说，持久化部分的性能直接决定了整个消息中间件的性能。RocketMQ充分利用Linux文件系统内存cache来提高性能
* **--内存Buffer**
* RocketMq Broker的buffer不会满。原因是RocketMQ没有内存Buffer概念，RocketMQ的队列都是持久化磁盘，数据定期清除。
* 这是RocketMq和其他消息中间件的重要区别。对RocketMQ来说的内存Buffer抽象成一个无限长度的队列，不管有多少数据进来都能装得下，这个无限是有前提的，Broker会定期删除过期的数据。
* **--消息堆积。**
* 消息堆积的能力是评价一个消息中间件的重要方面。因为使用消息中间件有一部分功能是为了为后端系统挡住数据洪峰。在产生消息堆积时，消息中间件对外的服务能力至关重要。
* 因为RocketMq的消息都是持久化硬盘的，当消息不能在内存Cache命中时，要不可避免的访问磁盘，会产生大量读IO，读IO的吞吐量直接决定了消息堆积后的访问能力。
* **--分布式事务。**
* 分布式事务涉及到两阶段提交。分为预提交阶段和commit阶段。在commit阶段需要回去改消息的状态。
* RocketMq在这里没有使用KV存储来做。而是在commit阶段会拿到消息的offset，然后直接去找消息，修改其状态。这样的好处是设计更简单，速度更快。缺点是会产生过多的数据脏页。

### **3 RocketMQ 物理部署结构**

[](http://img3.tbcdn.cn/5476e8b07b923/TB18GKUPXXXXXXRXFXXXXXXXXXX)

**4 RocketMQ的部署结构有以下特点：**

* Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
* Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与Name Server集群中的所有节点建立长连接，定时注册Topic信息到所有Name Server。
* Producer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
* Consumer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，订阅规则由Broker配置决定。

**5 ZooKeeper是著名的分布式协作框架，提供了Master选举、分布式锁、数据的发布和订阅等诸多功能，为什么RocketMQ没有选择ZooKeeper，而是自己开发了NameServer，我们来具体看看NameServer在RocketMQ集群中的作用就明了了。**

结论一：NameServer用来保存活跃的broker列表，包括Master和Slave。

结论二：NameServer用来保存所有topic和该topic所有队列的列表

结论三：NameServer用来保存所有broker的Filter列表。

现在我们再回过头来看看RocketMQ为什么不使用ZooKeeper？

ZooKeeper可以提供Master选举功能，比如Kafka用来给每个分区选一个broker作为leader，

但对于RocketMQ来说，topic的数据在每个Master上是对等的，没有哪个Master上有topic上的全部数据，所以这里选举leader没有意义；

RockeqMQ集群中，需要有构件来处理一些通用数据，比如broker列表，broker刷新时间，虽然ZooKeeper也能存放数据，并有一致性保证，

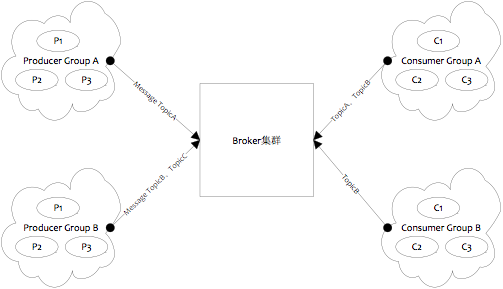
但处理数据之间的一些逻辑关系却比较麻烦，而且数据的逻辑解析操作得交给ZooKeeper客户端来做，如果有多种角色的客户端存在，自

己解析多级数据确实是个麻烦事情；既然RocketMQ集群中没有用到ZooKeeper的一些重量级的功能，只是使用ZooKeeper的数据一致性和发

布订阅的话，与其依赖重量级的ZooKeeper，还不如写个轻量级的NameServer，NameServer也可以集群部署，NameServer与NameServer之

间无任何信息同步，只有一千多行代码的NameServer稳定性肯定高于ZooKeeper，占用的系统资源也可以忽略不计。

### **6 RocketMQ 逻辑部署结构**

[](http://img3.tbcdn.cn/5476e8b07b923/TB1lEPePXXXXXX8XXXXXXXXXXXX)

如上图所示，RocketMQ的逻辑部署结构有Producer和Consumer两个特点。

Producer Group

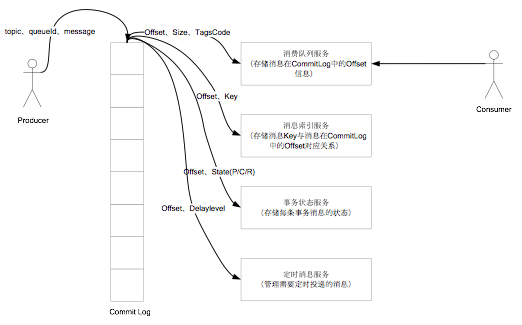
用来表示一个发送消息应用，一个Producer Group下**包含多个Producer实例，可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个Producer对象**。一个Producer Group可以发送多个Topic消息，Producer Group作用如下：

1. 标识一类Producer
2. 可以通过运维工具查询这个发送消息应用下有多个Producer实例
3. 发送分布式事务消息时，如果Producer中途意外宕机，Broker会主动回调Producer Group内的任意一台机器来确认事务状态。

Consumer Group

用来表示一个消费消息应用，一个Consumer Group下包含**多个Consumer实例，可以是多台机器，也可以是多个进程，或者是一个进程的多个Consumer对象**。一个Consumer Group下的多个Consumer以均摊方式消费消息，如果设置为广播方式，那么这个Consumer Group下的每个实例都消费全量数据。

### **7 RocketMQ 数据存储结构**

[](http://img3.tbcdn.cn/5476e8b07b923/TB1Ali2PXXXXXXuXFXXXXXXXXXX)

**如上图所示，RocketMQ采取了一种数据与索引分离的存储方法。有效降低文件资源、IO资源，内存资源的损耗。即便是阿里这种海量数据，高并发场景也能够有效降低端到端延迟，并具备较强的横向扩展能力。**

###### **8 RocketMQ的重复问题解决方式：**

a.MQ的消费端执行的操作具有幂等性，即无论多少次重复执行，其结果是一样的；

b.MQ的消费端做重复校验，比如将受到MQ消息的唯一编号保存到Redis中，即每次收到消息时，将检查唯一编号是否已经在Redis中，如果存在说明消息重复；否则将唯一编号放入到Redis中，可以根据系统需要设置唯一编号在Redis中的过期时间，以防止Redis溢出。

**nameServer顾名思义**，在系统中肯定是做命名服务，服务治理方面的工作，功能应该是和zookeeper差不多，据我了解，RocketMq的早期版本确实是使用的zookeeper,后来改为了自己实现的nameserver。现在来看一下nameServer在RocketMQ中的两个主要作用：

1 NameServer维护了一份Broker的地址列表和，broker在启动的时候会去NameServer进行注册，会维护Broker的存活状态.

2 NameServer维护了一份Topic和Topic对应队列的地址列表,broker每次发送心跳过来的时候都会把Topic信息带上.

**9 RocketMQ的Maven依赖**

<dependency>

<groupId>com.alibaba.rocketmq</groupId>

<artifactId>rocketmq-client</artifactId>

<version>3.2.6</version>

</dependency>

**10 MQ的消费类RocketMQConsumer.java：**

package com.lance.rocketMQ.RocketMQ;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;

import com.alibaba.rocketmq.client.consumer.listener.MessageListener;

import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;

import com.alibaba.rocketmq.client.exception.MQClientException;

import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;

import java.util.UUID;

public class RocketMQConsumer {

private DefaultMQPushConsumer consumer;

private MessageListener listener;

protected String nameServer;

protected String groupName;

protected String topics;

public RocketMQConsumer(MessageListener listener, String nameServer, String groupName, String topics) {

this.listener = listener;

this.nameServer = nameServer;

this.groupName = groupName;

this.topics = topics;

}

public void init() {

consumer = new DefaultMQPushConsumer(groupName);

consumer.setNamesrvAddr(nameServer);

try {

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

} catch (MQClientException e) {

e.printStackTrace();

}

consumer.setInstanceName(UUID.randomUUID().toString());

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

consumer.registerMessageListener((MessageListenerConcurrently) this.listener);

try {

consumer.start();

} catch (MQClientException e) {

e.printStackTrace();

}

System.out.println("RocketMQConsumer Started! group=" + consumer.getConsumerGroup() + " instance=" + consumer.getInstanceName()

);

}

}

**11 MQ消息的监听接口类RocketMQListener.java**

package com.lance.rocketMQ.RocketMQ;

import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;

import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;

import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;

import com.alibaba.rocketmq.common.message.MessageExt;

import java.util.List;

public class RocketMQListener implements MessageListenerConcurrently {

@Override

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

// System.out.println("get data from rocketMQ:" + msgs);

for (MessageExt message : msgs) {

String msg = new String(message.getBody());

System.out.println("msg data from rocketMQ:" + msg);

}

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

}

}

**12 MQ消息的生产者类RocketMQProducer.java**

package com.lance.rocketMQ.RocketMQ;

import com.alibaba.rocketmq.client.exception.MQClientException;

import com.alibaba.rocketmq.client.producer.DefaultMQProducer;

import com.alibaba.rocketmq.client.producer.SendResult;

import com.alibaba.rocketmq.client.producer.SendStatus;

import com.alibaba.rocketmq.common.message.Message;

import java.util.UUID;

public class RocketMQProducer {

private DefaultMQProducer sender;

protected String nameServer;

protected String groupName;

protected String topics;

public void init() {

sender = new DefaultMQProducer(groupName);

sender.setNamesrvAddr(nameServer);

sender.setInstanceName(UUID.randomUUID().toString());

try {

sender.start();

} catch (MQClientException e) {

e.printStackTrace();

}

}

public RocketMQProducer(String nameServer, String groupName, String topics) {

this.nameServer = nameServer;

this.groupName = groupName;

this.topics = topics;

}

public void send(Message message) {

message.setTopic(topics);

try {

SendResult result = sender.send(message);

SendStatus status = result.getSendStatus();

System.out.println("messageId=" + result.getMsgId() + ", status=" + status);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**13 测试RocketMQ的消费 RocketMQConsumerTest.java**

package com.lance.rocketMQ.RocketMQ;

public class RocketMQConsumerTest {

public static void main(String[] args) {

String mqNameServer = "172.10.254.2:9876";

String mqTopics = "MQ-MSG-TOPICS-TEST";

String consumerMqGroupName = "CONSUMER-MQ-GROUP";

RocketMQListener mqListener = new RocketMQListener();

RocketMQConsumer mqConsumer = new RocketMQConsumer(mqListener, mqNameServer, consumerMqGroupName, mqTopics);

mqConsumer.init();

try {

Thread.sleep(1000 \* 60L);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**run RocketMQConsumerTest.java 之后，控制台输出：**

RocketMQConsumer Started! group=CONSUMER-MQ-GROUP instance=1eb7d308-4414-4658-90b5-e2cae3b793eb

结果分析： 此时MQ对应的TOPIC中并没有响应的消息，故收不到消息，仅看到MQ消费者正常启动信息。

**14 MQ的生产者测试类:RocketMQProducerTest.java**

package com.lance.rocketMQ.RocketMQ;

import com.alibaba.rocketmq.common.message.Message;

public class RocketMQProducerTest {

public static void main(String[] args) {

String mqNameServer = "172.10.254.2:9876";

String mqTopics = "MQ-MSG-TOPICS-TEST";

String producerMqGroupName = "PRODUCER-MQ-GROUP";

RocketMQProducer mqProducer = new RocketMQProducer(mqNameServer, producerMqGroupName, mqTopics);

mqProducer.init();

for (int i = 0; i < 5; i++) {

Message message = new Message();

message.setBody(("I send message to RocketMQ " + i).getBytes());

mqProducer.send(message);

}

}

}

run RocketMQProducerTest.java 之后，RocketMQProducerTest.java 对应的控制台输出为：

messageId=0A71290100002A9F00000003D0BB0832, status=SEND\_OK

messageId=0A71290100002A9F00000003D0BB08BB, status=SEND\_OK

messageId=0A71290100002A9F00000003D0BB0944, status=SEND\_OK

messageId=0A71290100002A9F00000003D0BB09CD, status=SEND\_OK

messageId=0A71290300002A9F000000005440AEED, status=SEND\_OK

**结果分析：表明所有消息都已经正常发送，且被RocketMQ正常接收。**

此时查看RocketMQConsumerTest.java对应的控制台输出发生改变，输出内容变更如下:

RocketMQConsumer Started! group=CONSUMER-MQ-GROUP instance=1eb7d308-4414-4658-90b5-e2cae3b793eb

msg data from rocketMQ:I send message to RocketMQ 1

msg data from rocketMQ:I send message to RocketMQ 0

msg data from rocketMQ:I send message to RocketMQ 3

msg data from rocketMQ:I send message to RocketMQ 2

msg data from rocketMQ:I send message to RocketMQ 4