分布式消息系统作为实现分布式系统可扩展、可伸缩性的关键组件，需要具有高吞吐量、高可用等特点。而谈到消息系统的设计，就回避不了两个问题：

> 1. 消息的顺序问题
> 2. 消息的重复问题

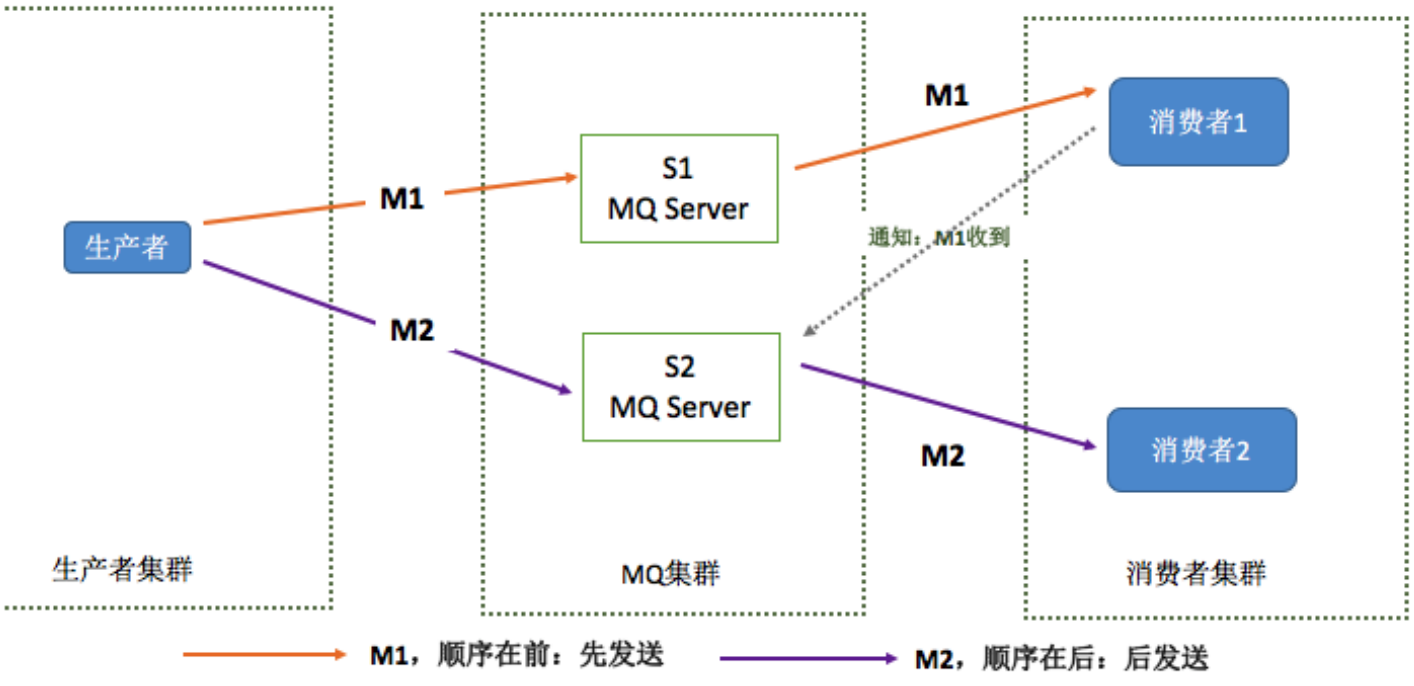
RocketMQ作为阿里开源的一款高性能、高吞吐量的消息中间件，它是怎样来解决这两个问题的？RocketMQ 有哪些关键特性？其实现原理是怎样的？

关键特性以及其实现原理

一、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。例如：一笔订单产生了 3 条消息，分别是订单创建、订单付款、订单完成。消费时，要按照顺序依次消费才有意义。与此同时多笔订单之间又是可以并行消费的。首先已两笔消息来看如下示例：

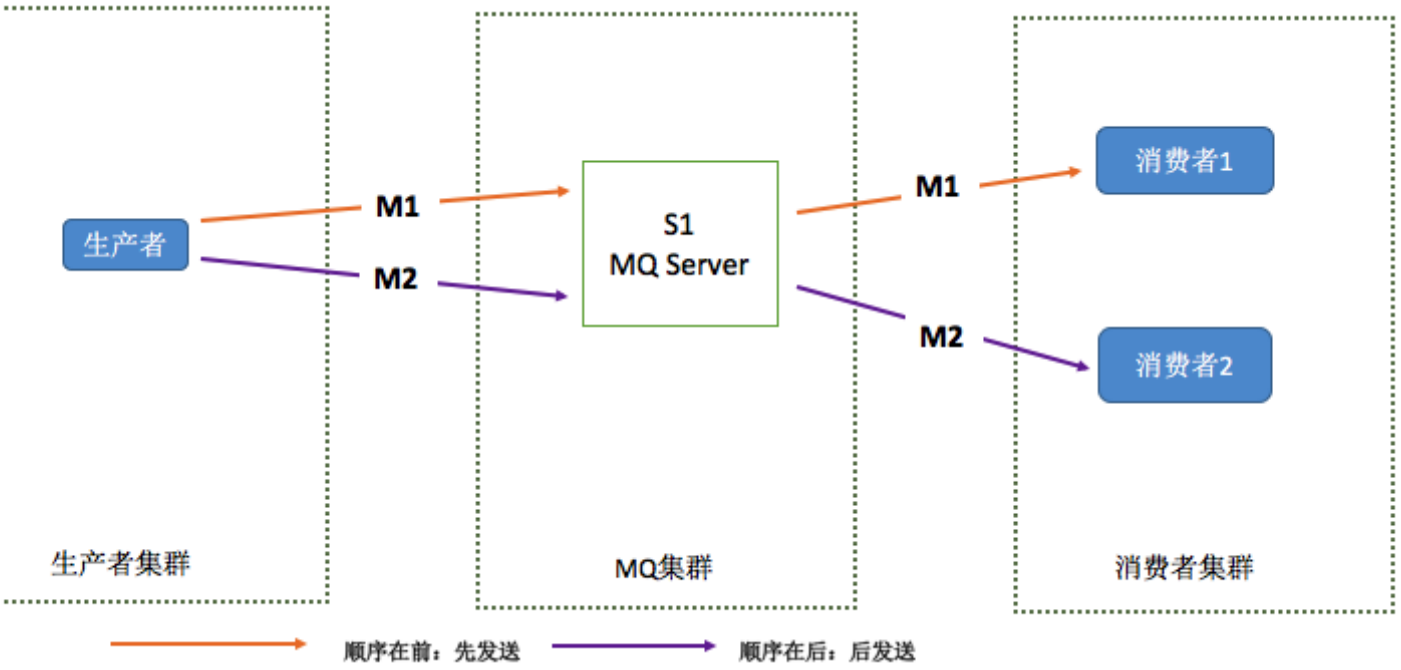
假如生产者产生了2条消息：M1、M2，要保证这两条消息的顺序，应该怎样做？你脑中想到的可能是这样：



你可能会采用这种方式保证消息顺序

如果M1发送到S1，M2发送到S2，如果要保证M1先于M2被消费，那么需要M1到达消费端被消费后，通知S2，然后S2再将M2发送到消费端。

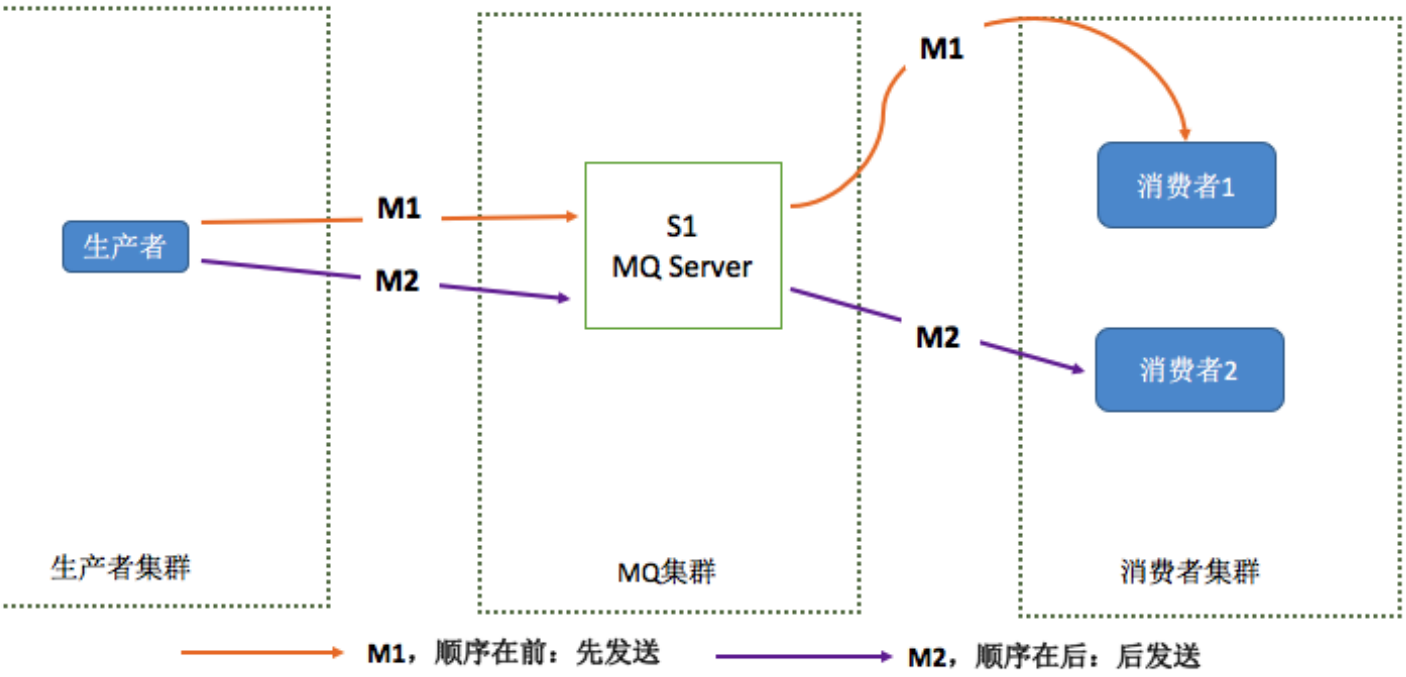
这个模型存在的问题是，如果M1和M2分别发送到两台Server上，就不能保证M1先达到MQ集群，也不能保证M1被先消费。换个角度看，如果M2先于M1达到MQ集群，甚至M2被消费后，M1才达到消费端，这时消息也就乱序了，说明以上模型是不能保证消息的顺序的。如何才能在MQ集群保证消息的顺序？一种简单的方式就是将M1、M2发送到同一个Server上：



保证消息顺序，你改进后的方法

这样可以保证M1先于M2到达MQServer（生产者等待M1发送成功后再发送M2），根据先达到先被消费的原则，M1会先于M2被消费，这样就保证了消息的顺序。

但这个模型也仅仅是理论上可以保证消息的顺序，但在实际运用中你应该会遇到下面的问题：

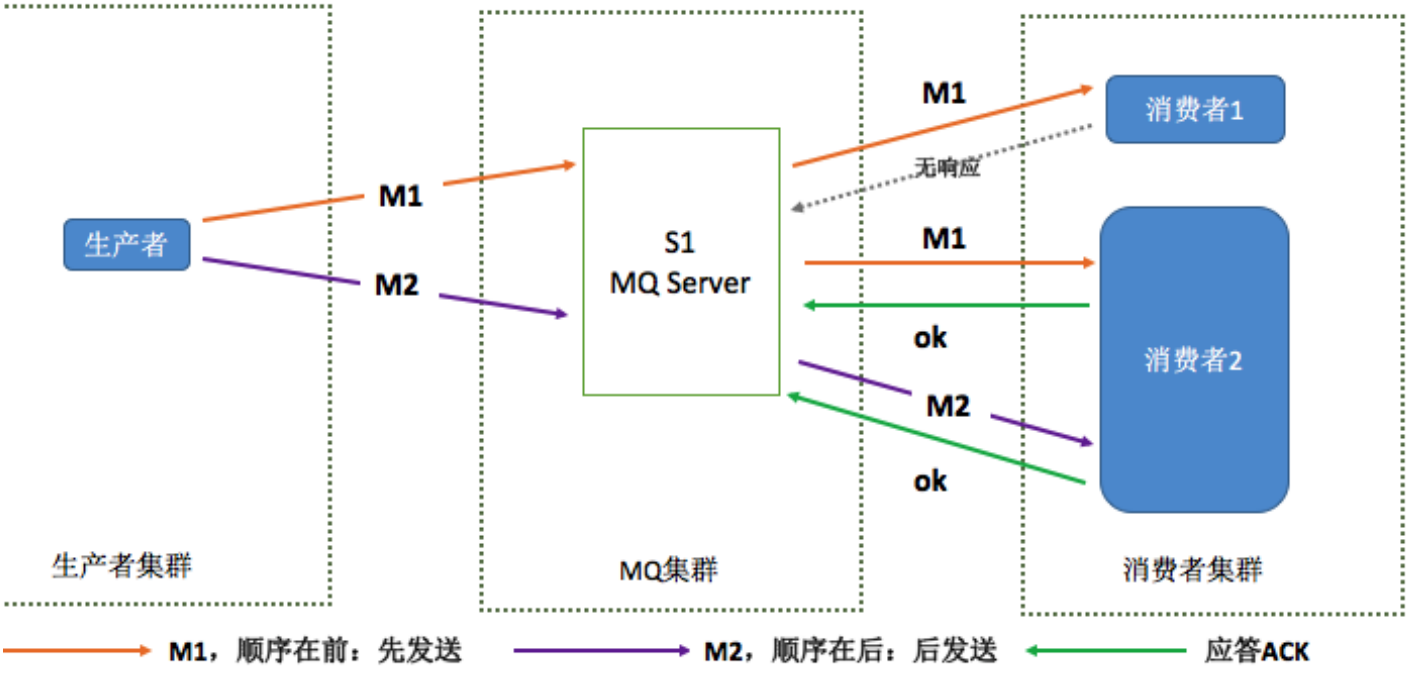


网络延迟问题

只要将消息从一台服务器发往另一台服务器，就会存在网络延迟问题。如上图所示，如果发送M1耗时大于发送M2的耗时，那么M2就仍将被先消费，仍然不能保证消息的顺序。即使M1和M2同时到达消费端，由于不清楚消费端1和消费端2的负载情况，仍然有可能出现M2先于M1被消费的情况。

那如何解决这个问题？将M1和M2发往同一个消费者，且发送M1后，需要消费端响应成功后才能发送M2。

聪明的你可能已经想到另外的问题：如果M1被发送到消费端后，消费端1没有响应，那是继续发送M2呢，还是重新发送M1？一般为了保证消息一定被消费，肯定会选择重发M1到另外一个消费端2，就如下图所示。



保证消息顺序的正确姿势

这样的模型就严格保证消息的顺序，细心的你仍然会发现问题，消费端1没有响应Server时有两种情况，一种是M1确实没有到达(数据在网络传送中丢失)，另外一种消费端已经消费M1且已经发送响应消息，只是MQ Server端没有收到。如果是第二种情况，重发M1，就会造成M1被重复消费。也就引入了我们要说的第二个问题，消息重复问题，这个后文会详细讲解。

回过头来看消息顺序问题，严格的顺序消息非常容易理解，也可以通过文中所描述的方式来简单处理。总结起来，要实现严格的顺序消息，简单且可行的办法就是：

> 保证 生产者 - *MQServer* - 消费者 是一对一对一的关系

这样的设计虽然简单易行，但也会存在一些很严重的问题，比如：

> 1. 并行度就会成为消息系统的瓶颈（吞吐量不够）
> 2. 更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。

但我们的最终目标是要集群的高容错性和高吞吐量。这似乎是一对不可调和的矛盾，那么阿里

是如何解决的？

> 世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！—— 沈询

有些问题，看起来很重要，但实际上我们可以通过合理的设计或者将问题分解来规避。如果硬要把时间花在解决问题本身，实际上不仅效率低下，而且也是一种浪费。从这个角度来看消息的顺序问题，我们可以得出两个结论：

> 1. 不关注乱序的应用实际大量存在
> 2. 队列无序并不意味着消息无序

所以从业务层面来保证消息的顺序而不仅仅是依赖于消息系统，是不是我们应该寻求的一种更合理的方式？

最后我们从源码角度分析RocketMQ怎么实现发送顺序消息。

RocketMQ通过轮询所有队列的方式来确定消息被发送到哪一个队列（负载均衡策略）。比如下面的示例中，订单号相同的消息会被先后发送到同一个队列中：

```
// RocketMQ通过MessageQueueSelector中实现的算法来确定消息发送到哪一个队列上
// RocketMQ默认提供了两种MessageQueueSelector实现：随机/Hash
// 当然你可以根据业务实现自己的MessageQueueSelector来决定消息按照何种策略发送到消息队列中
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);
```

在获取到路由信息以后，会根据 `MessageQueueSelector` 实现的算法来选择一个队列，同一个OrderId获取到的肯定是同一个队列。

```
private SendResult send()  {
```

```
    // 获取topic路由信息
    TopicPublishInfo topicPublishInfo = this.tryToFindTopicPublishInfo(msg.getTopic());
    if (topicPublishInfo != null && topicPublishInfo.ok()) {
        MessageQueue mq = null;
        // 根据我们的算法，选择一个发送队列
        // 这里的arg = orderId
        mq = selector.select(topicPublishInfo.getMessageQueueList(), msg, arg);
        if (mq != null) {
            return this.sendKernelImpl(msg, mq, communicationMode, sendCallback, timeout);
        }
    }
}
```

二、消息重复

上面在解决消息顺序问题时，引入了一个新的问题，就是消息重复。那么RocketMQ是怎样解决消息重复的问题呢？还是“恰好”不解决。

造成消息重复的根本原因是：网络不可达。只要通过网络交换数据，就无法避免这个问题。所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

> 1. 消费端处理消息的业务逻辑保持幂等性
> 2. 保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现

第1条很好理解，只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。第2条原理就是利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么就不再处理这条消息。

第1条解决方案，很明显应该在消费端实现，不属于消息系统要实现的功能。第2条可以消息系统实现，也可以业务端实现。正常情况下出现重复消息的概率其实很小，如果由消息系统来实现的话，肯定会对消息系统的吞吐量和高可用有影响，所以最好还是由业务端自己处理消息重复的问题，这也是RocketMQ不解决消息重复的问题的原因。

RocketMQ不保证消息不重复，如果你的业务需要保证严格的不重复消息，需要你自己在业务端去重。

三、事务消息

RocketMQ除了支持普通消息，顺序消息，另外还支持事务消息。首先讨论一下什么是事务消息以及支持事务消息的必要性。我们以一个转帐的场景为例来说明这个问题：Bob向Smith转账100块。

在单机环境下，执行事务的情况，大概是下面这个样子：

单机环境下转账事务示意图

当用户增长到一定程度，Bob和Smith的账户及余额信息已经不在同一台服务器上了，那么上面的流程就变成了这样：

集群环境下转账事务示意图

这时候你会发现，同样是一个转账的业务，在集群环境下，耗时居然成倍的增长，这显然是不能够接受的。那如何来规避这个问题？

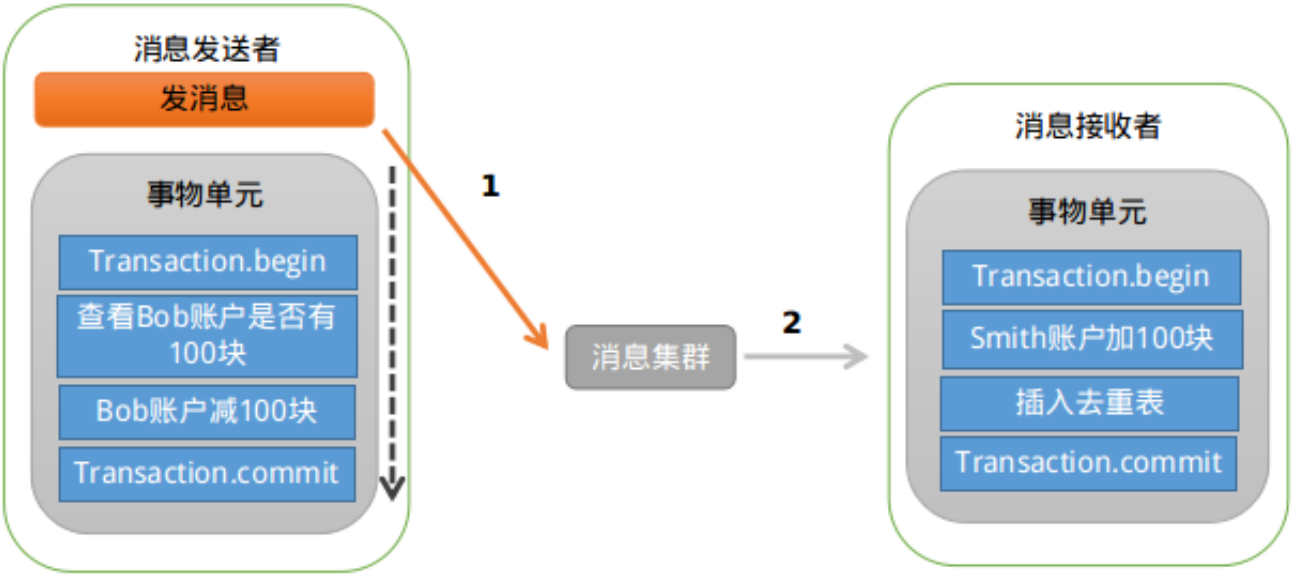
> 大事务 = 小事务 + 异步

将大事务拆分成多个小事务异步执行。这样基本上能够将跨机事务的执行效率优化到与单机一致。转账的事务就可以分解成如下两个小事务：

小事务+异步消息

图中执行本地事务（Bob账户扣款）和发送异步消息应该保证同时成功或者同时失败，也就是扣款成功了，发送消息一定要成功，如果扣款失败了，就不能再发送消息。那问题是：我们是先扣款还是先发送消息呢？

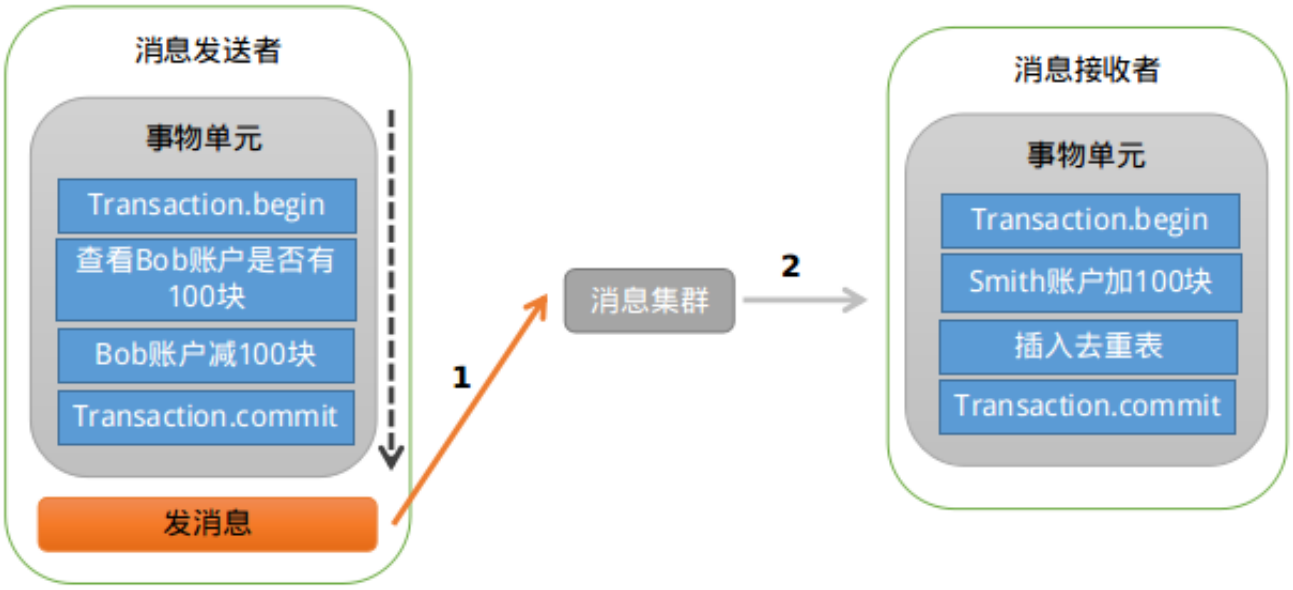
首先看下先发送消息的情况，大致的示意图如下：



事务消息：先发送消息

存在的问题是：如果消息发送成功，但是扣款失败，消费端就会消费此消息，进而向Smith账户加钱。

先发消息不行，那就先扣款吧，大致的示意图如下：



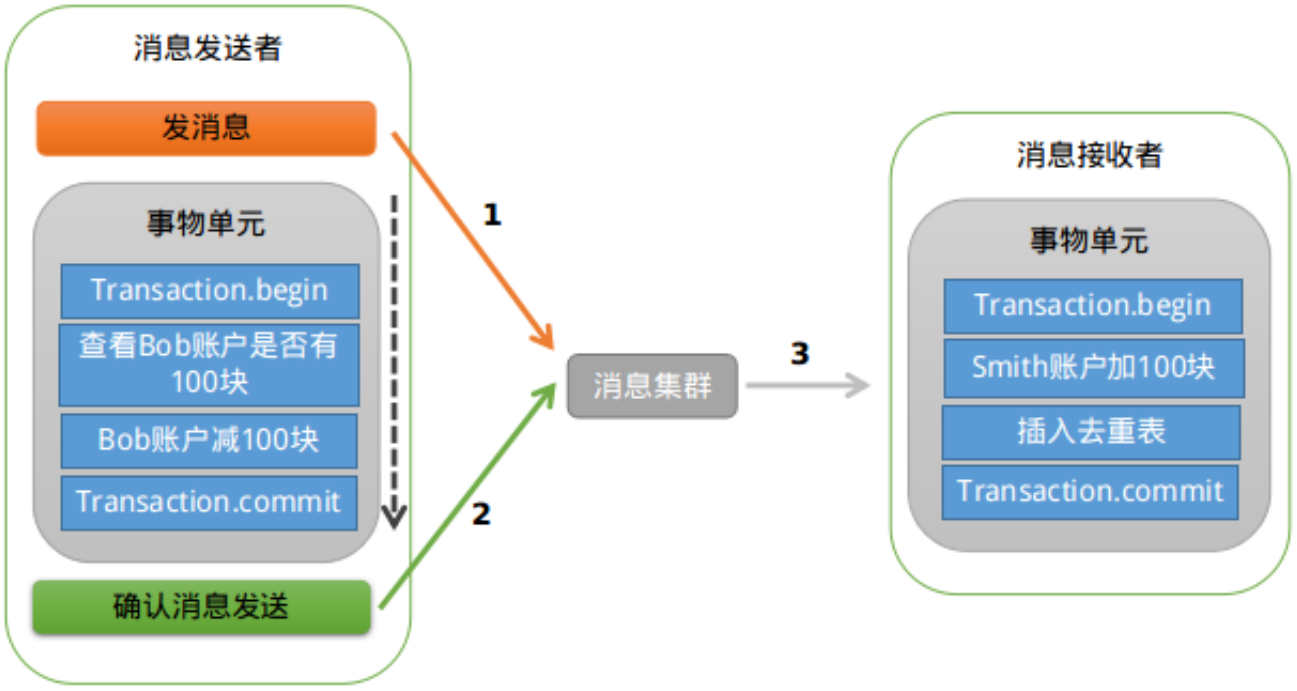
事务消息-先扣款

存在的问题跟上面类似：如果扣款成功，发送消息失败，就会出现Bob扣钱了，但是Smith账户未加钱。

可能大家会有很多的方法来解决这个问题，比如：直接将发消息放到Bob扣款的事务中去，如果发送失败，抛出异常，事务回滚。这样的处理方式也符合“恰好”不需要解决的原则。

> 这里需要说明一下：如果使用Spring来管理事物的话，大可以将发送消息的逻辑放到本地事物中去，发送消息失败抛出异常，Spring捕捉到异常后就会回滚此事物，以此来保证本地事物与发送消息的原子性。

RocketMQ支持事务消息，下面来看看RocketMQ是怎样来实现的。



RocketMQ实现发送事务消息

RocketMQ第一阶段发送 *Prepared*消息 时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改消息的状态。

细心的你可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，如果发现了 *Prepared*消息 ，它会向消息发送端(生产者)确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

那我们来看下RocketMQ源码，是如何处理事务消息的。客户端发送事务消息的部分（完整代码请查看： *rocketmq-example* 工程下的 *com.alibaba.rocketmq.example.transaction.TransactionProducer* ）：

```
// ==============================发送事务消息的一系列准备工作=========================================
// 未决事务，MQ服务器回查客户端
// 也就是上文所说的，当RocketMQ发现`Prepared消息`时，会根据这个Listener实现的策略来决断事务
TransactionCheckListener transactionCheckListener = new TransactionCheckListenerImpl();
// 构造事务消息的生产者
TransactionMQProducer producer = new TransactionMQProducer("groupName");
// 设置事务决断处理类
producer.setTransactionCheckListener(transactionCheckListener);
// 本地事务的处理逻辑，相当于示例中检查Bob账户并扣钱的逻辑
TransactionExecuterImpl tranExecuter = new TransactionExecuterImpl();
producer.start()
// 构造MSG，省略构造参数
```

```
Message msg = new Message(.....);
// 发送消息
SendResult sendResult = producer.sendMessageInTransaction(msg, tranExecuter, null);
producer.shutdown();
```

接着查看 *sendMessageInTransaction* 方法的源码，总共分为3个阶段：发送 *Prepared*消息 、执行本地事务、发送确认消息。

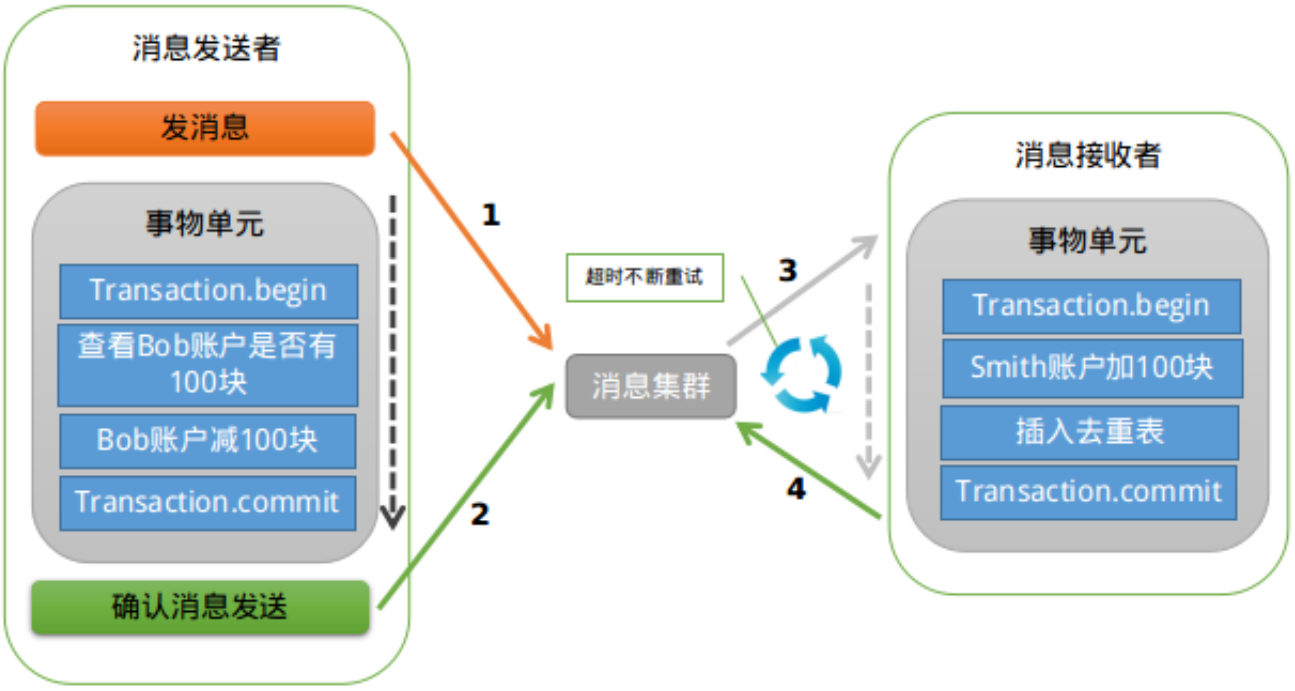
```
//  ================================事务消息的发送过程==========================================
public TransactionSendResult sendMessageInTransaction(.....)  {
    // 逻辑代码，非实际代码
    // 1.发送消息
    sendResult = this.send(msg);
    // sendResult.getSendStatus() == SEND_OK
    // 2.如果消息发送成功，处理与消息关联的本地事务单元
    LocalTransactionState localTransactionState = tranExecuter.executeLocalTransactionBranch(msg,
    // 3.结束事务
    this.endTransaction(sendResult, localTransactionState, localException);
}
```

endTransaction 方法会将请求发往 *broker(mq server)* 去更新事务消息的最终状态：

1. 根据 *sendResult* 找到 *Prepared*消息 ， *sendResult* 包含事务消息的ID
2. 根据 *localTransaction* 更新消息的最终状态

如果 *endTransaction* 方法执行失败，数据没有发送到 *broker* ，导致事务消息的 状态更新失败， *broker* 会有回查线程定时（默认1分钟）扫描每个存储事务状态的表格文件，如果是已经提交或者回滚的消息直接跳过，如果是 *prepared*状态 则会向 *Producer* 发起 *CheckTransaction* 请求， *Producer* 会调用 *DefaultMQProducerImpl.checkTransactionState()* 方法来处理 *broker* 的定时回调请求，而 *checkTransactionState* 会调用我们的事务设置的决断方法来决定是回滚事务还是继续执行，最后调用 *endTransactionOneway* 让 *broker* 来更新消息的最终状态。

再回到转账的例子，如果Bob的账户的余额已经减少，且消息已经发送成功，Smith端开始消费这条消息，这个时候就会出现消费失败和消费超时两个问题，解决超时问题的思路就是一直重试，直到消费端消费消息成功，整个过程中有可能会出现消息重复的问题，按照前面的思路解决即可。



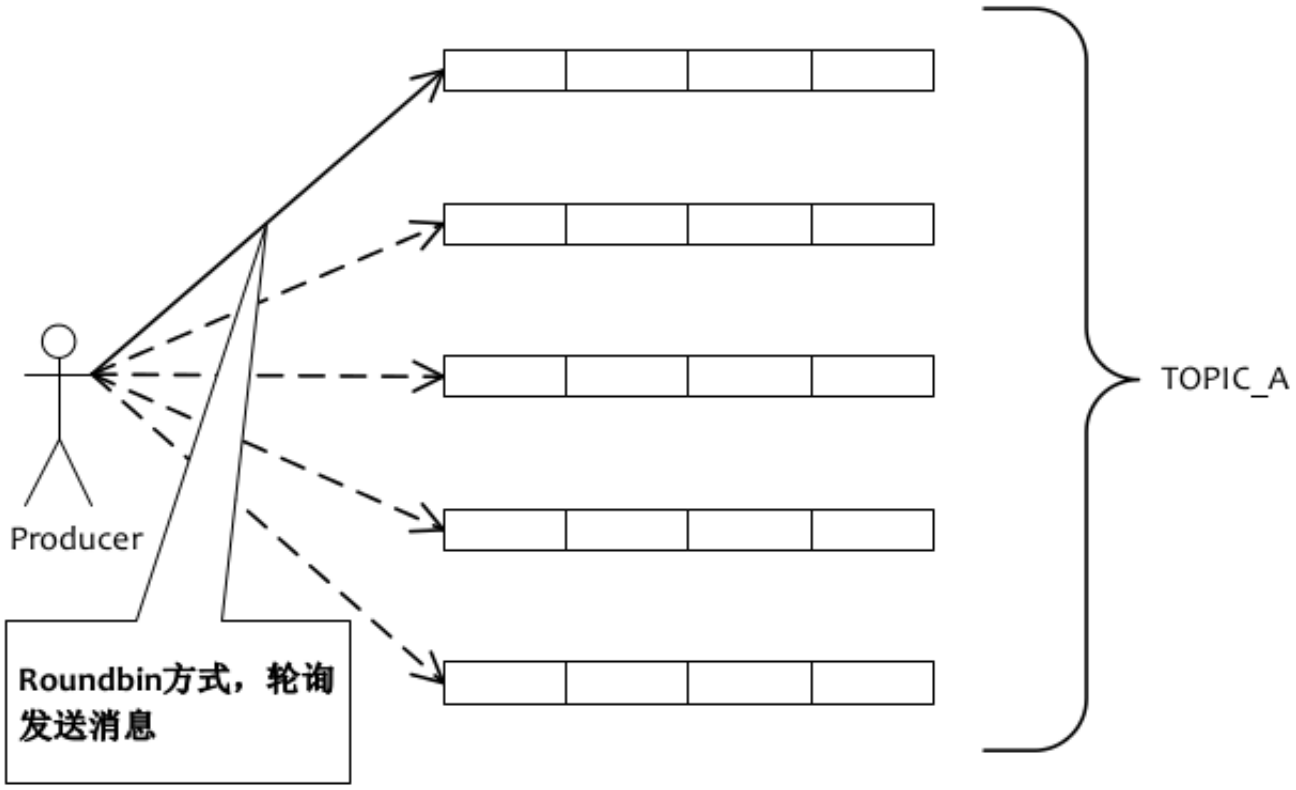
消费事务消息

这样基本上可以解决消费端超时问题，但是如果消费失败怎么办？阿里提供给我们的解决方法是：人工解决。大家可以考虑一下，按照事务的流程，因为某种原因Smith加款失败，那么需要回滚整个流程。如果消息系统要实现这个回滚流程的话，系统复杂度将大大提升，且很容易出现Bug，估计出现Bug的概率会比消费失败的概率大很多。这也是RocketMQ目前暂时没有解决这个问题的原因，在设计实现消息系统时，我们需要衡量是否值得花这么大的代价来解决这样一个出现概率非常小的问题，这也是大家在解决疑难问题时需要多多思考的地方。

> 20160321补充：在3.2.6版本中移除了事务消息的实现，所以此版本不支持事务消息，具体情况请参考rocketmq的issues：
> https://github.com/alibaba/RocketMQ/issues/65
> https://github.com/alibaba/RocketMQ/issues/138
> https://github.com/alibaba/RocketMQ/issues/156

四、Producer如何发送消息

Producer 轮询某topic下的所有队列的方式来实现发送方的负载均衡，如下图所示：



producer发送消息负载均衡

首先分析一下RocketMQ的客户端发送消息的源码：

```
// 构造Producer
DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
// 初始化Producer，整个应用生命周期内，只需要初始化1次
producer.start();
// 构造Message
Message msg = new Message("TopicTest1",// topic
                          "TagA",// tag：给消息打标签,用于区分一类消息，可为null
                          "OrderID188",// key：自定义Key，可以用于去重，可为null
                          ("Hello MetaQ").getBytes());// body：消息内容
// 发送消息并返回结果
SendResult sendResult = producer.send(msg);
// 清理资源，关闭网络连接，注销自己
producer.shutdown();
```

在整个应用生命周期内，生产者需要调用一次start方法来初始化，初始化主要完成的任务有：

> 1. 如果没有指定 *namesrv* 地址，将会自动寻址
> 2. 启动定时任务：更新namesrv地址、从namsrv更新topic路由信息、清理已经挂掉的

> broker、向所有broker发送心跳...
>
> 3. 启动负载均衡的服务

初始化完成后，开始发送消息，发送消息的主要代码如下：

```
private SendResult sendDefaultImpl(Message msg,.....) {
    // 检查Producer的状态是否是RUNNING
    this.makeSureStateOK();
    // 检查msg是否合法：是否为null、topic,body是否为空、body是否超长
    Validators.checkMessage(msg, this.defaultMQProducer);
    // 获取topic路由信息
    TopicPublishInfo topicPublishInfo = this.tryToFindTopicPublishInfo(msg.getTopic());
    // 从路由信息中选择一个消息队列
    MessageQueue mq = topicPublishInfo.selectOneMessageQueue(lastBrokerName);
    // 将消息发送到该队列上去
    sendResult = this.sendKernelImpl(msg, mq, communicationMode, sendCallback, timeout);
}
```

代码中需要关注的两个方法 *tryToFindTopicPublishInfo* 和 *selectOneMessageQueue* 。前面说过在producer初始化时，会启动定时任务获取路由信息并更新到本地缓存，所以 *tryToFindTopicPublishInfo* 会首先从缓存中获取topic路由信息，如果没有获取到，则会自己去 *namesrv* 获取路由信息。 *selectOneMessageQueue* 方法通过轮询的方式，返回一个队列，以达到负载均衡的目的。

如果Producer发送消息失败，会自动重试，重试的策略：

1. 重试次数 < retryTimesWhenSendFailed（可配置）
2. 总的耗时（包含重试n次的耗时） < sendMsgTimeout（发送消息时传入的参数）
3. 同时满足上面两个条件后，Producer会选择另外一个队列发送消息

五、消息存储

RocketMQ的消息存储是由 *consume queue* 和 *commit log* 配合完成的。

1、Consume Queue

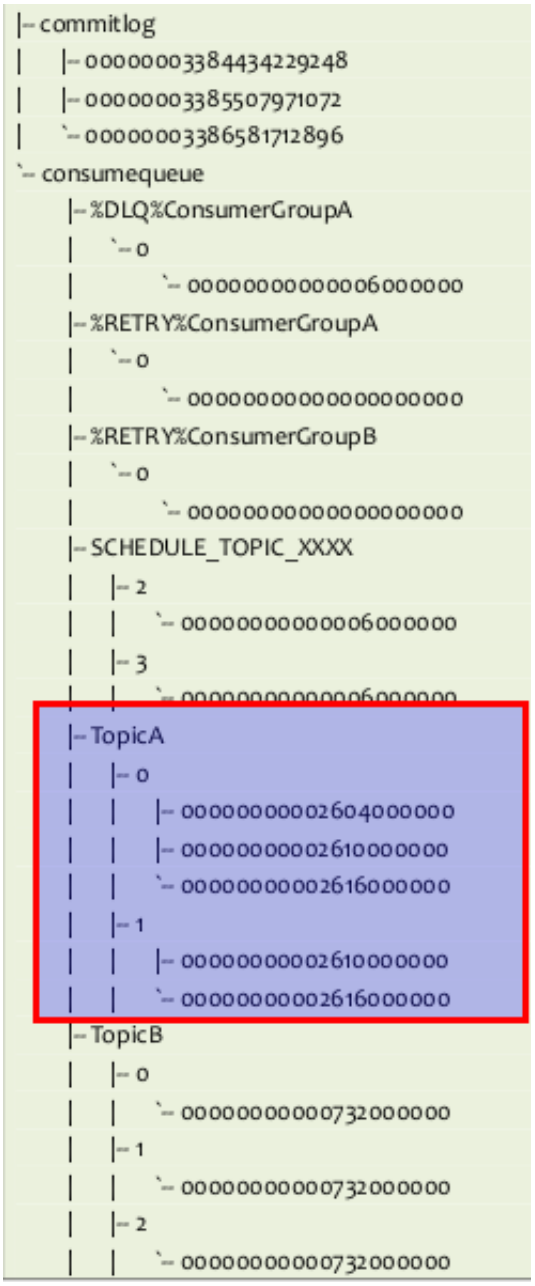
consume queue 是消息的逻辑队列，相当于字典的目录，用来指定消息在物理文件 *commit log* 上

的位置。

我们可以在配置中指定 *consumequeue* 与 *commitlog* 存储的目录
每个 *topic* 下的每个 *queue* 都有一个对应的 *consumequeue* 文件，比如：

```
${rocketmq.home}/store/consumequeue/${topicName}/${queueId}/${fileName}
```

Consume Queue文件组织，如图所示：



Consume Queue文件组织示意图

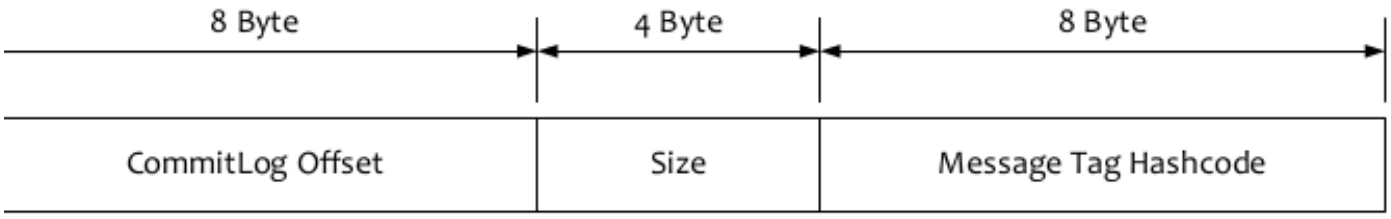
1. 根据 *topic* 和 *queueId* 来组织文件，图中TopicA有两个队列0,1，那么TopicA和QueueId=0组

成一个ConsumeQueue，TopicA和QueueId=1组成另一个ConsumeQueue。

2. 按照消费端的 *GroupName* 来分组重试队列，如果消费端消费失败，消息将被发往重试队列中，比如图中的 *%RETRY%ConsumerGroupA* 。
3. 按照消费端的 *GroupName* 来分组死信队列，如果消费端消费失败，并重试指定次数后，仍然失败，则发往死信队列，比如图中的 *%DLQ%ConsumerGroupA* 。

> 死信队列（Dead Letter Queue）一般用于存放由于某种原因无法传递的消息，比如处理失败或者已经过期的消息。

Consume Queue中存储单元是一个20字节定长的二进制数据，顺序写顺序读，如下图所示：



consumequeue文件存储单元格式

1. CommitLog Offset是指这条消息在Commit Log文件中的实际偏移量
2. Size存储中消息的大小
3. Message Tag HashCode存储消息的Tag的哈希值：主要用于订阅时消息过滤（订阅时如果指定了Tag，会根据HashCode来快速查找到订阅的消息）

2、Commit Log

CommitLog：消息存放的物理文件，每台 *broker* 上的 *commitlog* 被本机所有的 *queue* 共享，不做任何区分。
文件的默认位置如下，仍然可通过配置文件修改：

```
${user.home} \store\${commitlog}\${fileName}
```

CommitLog的消息存储单元长度不固定，文件顺序写，随机读。消息的存储结构如下表所示，

按照编号顺序以及编号对应的内容依次存储。

序号	消息存储结构	备注	长度（字节数）
1	TOTALSIZE	消息大小	4
2	MAGICCODE	消息的MAGIC CODE: daa320a7	4
3	BODYCRC	消息体BODY CRC，当broker重启时会校验	4
4	QUEUEID	队列编号，queueId	4
5	FLAG	不处理	4
6	QUEUEOFFSET	自增值，不是真正的consume queue的偏移量，可以代表这个队列中消息的个数，要通过这个值查找到consume queue中数据，QUEUEOFFSET * 20才是偏移地址	8
7	PHYSICALOFFSET	消息在commitLog中的物理起始地址偏移量	8
8	SYSFLAG	消息标志，指明消息是事物事物状态等等消息特征	4
9	BORNTIMESTAMP	消息生产端(producer)的时间戳	8
10	BORNHOST（IP+PORT）	生产者地址	8
11	STORETIMESTAMP	存储时间戳	8
12	STOREHOST（IP+PORT）	消息存储到broker的地址	8
13	RECONSUMETIMES	消息被某个订阅组重新消费了几次（订阅组之间独立计数）	8
14	Prepared Transaction Offset	表示该消息是prepared状态的事物消息	8
15	BODY	前4个字节存放消息体大小值， 后bodylength大小空间存储了消息体内容	4 + bodyLength
16	TOPIC	前1个字节存放topic名称能容大小， 后存放了topic的内容	1 + topicLength
17	properties	前2个字节（short）存放属性值大小， 后存放propertiesLength大小的属性数据	2 + propertiesLen

Commit Log存储单元结构图

3、消息存储实现

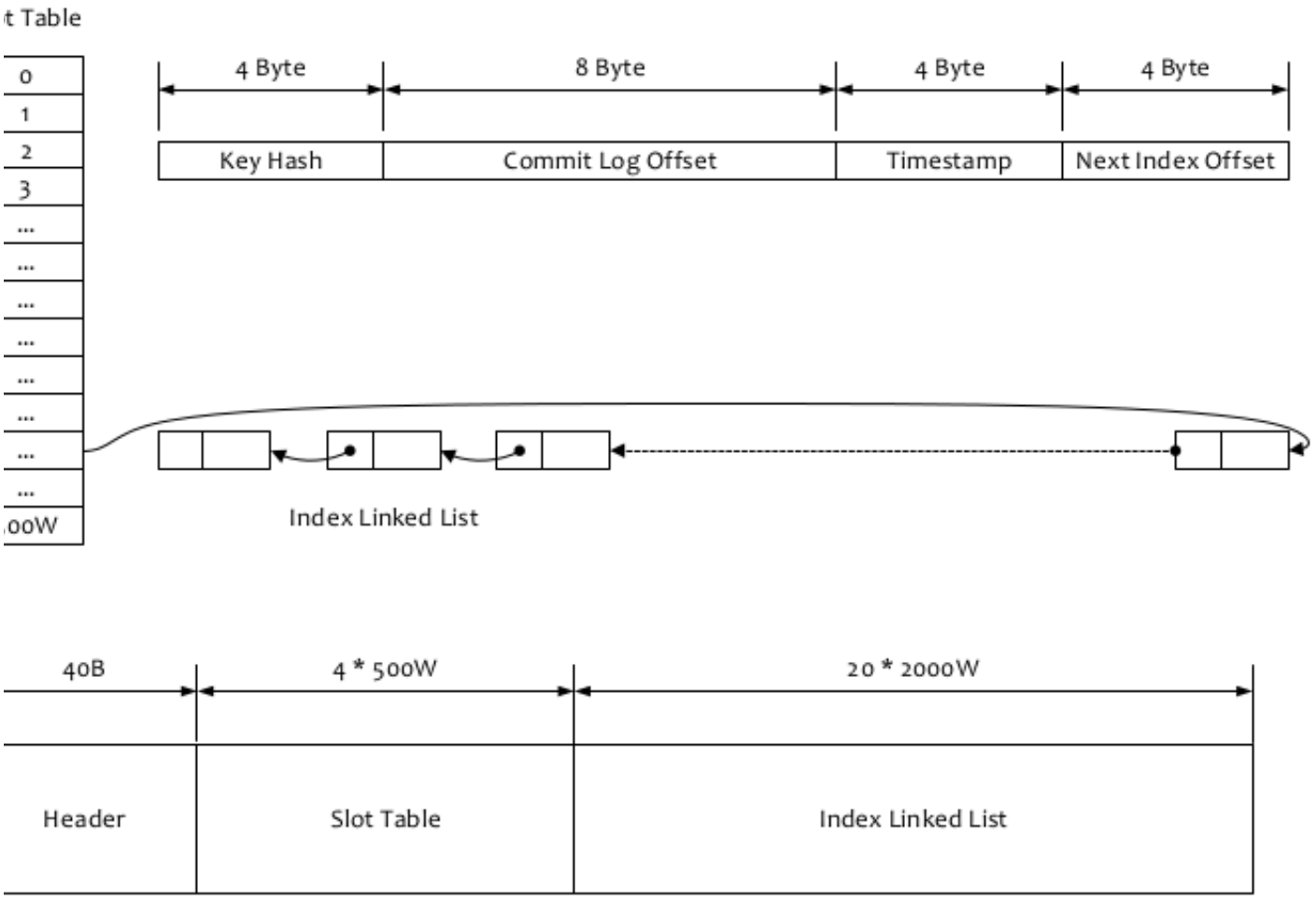
消息存储实现，比较复杂，也值得大家深入了解，后面会单独成文来分析(目前正在收集素材)，这小节只以代码说明一下具体的流程。

```
// Set the storage time
msg.setStoreTimestamp(System.currentTimeMillis());
// Set the message body BODY CRC (consider the most appropriate setting
msg.setBodyCRC(UtilAll.crc32(msg.getBody()));
StoreStatsService storeStatsService = this.defaultMessageStore.getStoreStatsService();
synchronized (this) {
    long beginLockTimestamp = this.defaultMessageStore.getSystemClock().now();
    // Here settings are stored timestamp, in order to ensure an orderly global
    msg.setStoreTimestamp(beginLockTimestamp);
    // MapedFile: 操作物理文件在内存中的映射以及将内存数据持久化到物理文件中
    MapedFile mapedFile = this.mapedFileQueue.getLastMapedFile();
    // 将Message追加到文件commitlog
    result = mapedFile.appendMessage(msg, this.appendMessageCallback);
    switch (result.getStatus()) {
    case PUT_OK:break;
```

```
    case END_OF_FILE:
         // Create a new file, re-write the message
         mapedFile = this.mapedFileQueue.getLastMapedFile();
         result = mapedFile.appendMessage(msg, this.appendMessageCallback);
     break;
     DispatchRequest dispatchRequest = new DispatchRequest(
                topic,// 1
                queueId,// 2
                result.getWroteOffset(),// 3
                result.getWroteBytes(),// 4
                tagsCode,// 5
                msg.getStoreTimestamp(),// 6
                result.getLogicsOffset(),// 7
                msg.getKeys(),// 8
                /**
                 * Transaction
                 */
                msg.getSysFlag(),// 9
                msg.getPreparedTransactionOffset());// 10
    // 1.分发消息位置到ConsumeQueue
    // 2.分发到IndexService建立索引
    this.defaultMessageStore.putDispatchRequest(dispatchRequest);
}
```

4、消息的索引文件

如果一个消息包含key值的话，会使用IndexFile存储消息索引，文件的内容结构如图：



消息索引

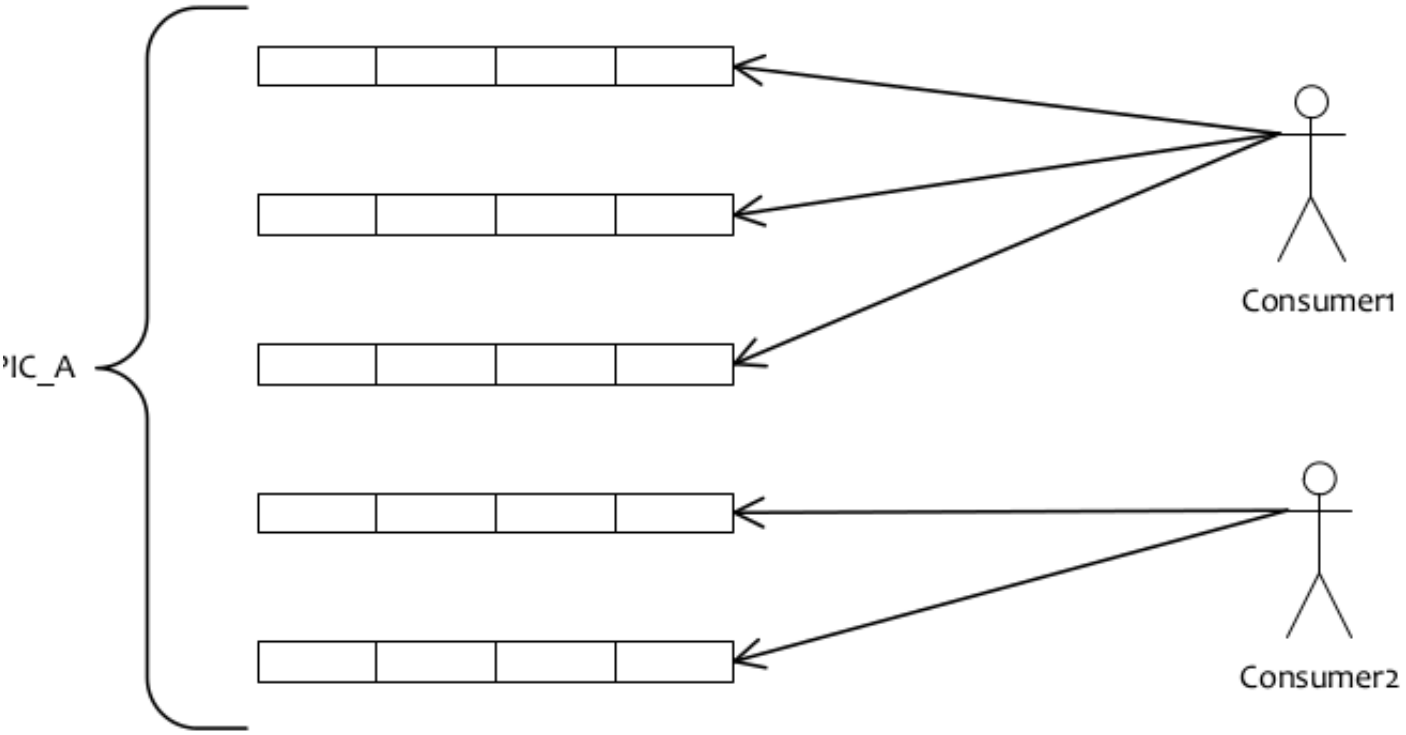
索引文件主要用于根据key来查询消息的，流程主要是：

1. 根据查询的 key 的 hashcode%slotNum 得到具体的槽的位置(slotNum 是一个索引文件里面包含的最大槽的数目，例如图中所示 slotNum=5000000)
2. 根据 slotValue(slot 位置对应的值)查找到索引项列表的最后一项(倒序排列,slotValue 总是指向最新的一个索引项)
3. 遍历索引项列表返回查询时间范围内的结果集(默认一次最大返回的 32 条记录)

六、消息订阅

RocketMQ消息订阅有两种模式，一种是Push模式，即MQServer主动向消费端推送；另外一种是Pull模式，即消费端在需要时，主动到MQServer拉取。但在具体实现时，Push和Pull模式都是采用消费端主动拉取的方式。

首先看下消费端的负载均衡：



消费端负载均衡

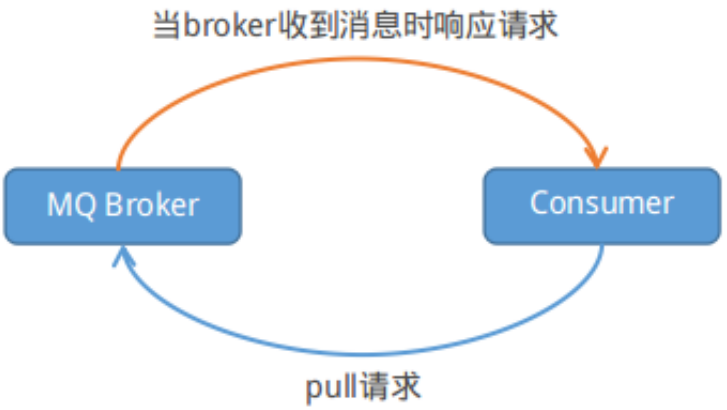
消费端会通过RebalanceService线程，10秒钟做一次基于topic下的所有队列负载：

1. 遍历Consumer下的所有topic，然后根据topic订阅所有的消息
2. 获取同一topic和Consumer Group下的所有Consumer
3. 然后根据具体的分配策略来分配消费队列，分配的策略包含：平均分配、消费端配置等

如同上图所示：如果有 5 个队列，2 个 consumer，那么第一个 Consumer 消费 3 个队列，第二 consumer 消费 2 个队列。这里采用的就是平均分配策略，它类似于分页的过程，TOPIC下面的所有queue就是记录，Consumer的个数就相当于总的页数，那么每页有多少条记录，就类似于某个Consumer会消费哪些队列。

通过这样的策略来达到大体上的平均消费，这样的设计也可以很方面的水平扩展Consumer来提高消费能力。

消费端的Push模式是通过长轮询的模式来实现的，就如同下图：



Push模式示意图

Consumer端每隔一段时间主动向broker发送拉消息请求，broker在收到Pull请求后，如果有消息就立即返回数据，Consumer端收到返回的消息后，再回调消费者设置的Listener方法。如果broker在收到Pull请求时，消息队列里没有数据，broker端会阻塞请求直到有数据传递或超时才返回。

当然，Consumer端是通过一个线程将阻塞队列 `LinkedBlockingQueue<PullRequest>` 中的 `PullRequest` 发送到broker拉取消息，以防止Consumer一致被阻塞。而Broker端，在接收到Consumer的 `PullRequest` 时，如果发现没有消息，就会把 `PullRequest` 扔到ConcurrentHashMap中缓存起来。broker在启动时，会启动一个线程不停的从ConcurrentHashMap取出 `PullRequest` 检查，直到有数据返回。

七、RocketMQ的其他特性

前面的6个特性都是基本上都是点到为止，想要深入了解，还需要大家多多查看源码，多多在实际中运用。当然除了已经提到的特性外，RocketMQ还支持：

1. 定时消息
2. 消息的刷盘策略
3. 主动同步策略：同步双写、异步复制
4. 海量消息堆积能力
5. 高效通信
6. .....

其中涉及到的很多设计思路和解决方法都值得我们深入研究：

1. 消息的存储设计：既要满足海量消息的堆积能力，又要满足极快的查询效率，还要保证写入的效率。
2. 高效的通信组件设计：高吞吐量，毫秒级的消息投递能力都离不开高效的通信。
3. .....

RocketMQ最佳实践

一、Producer最佳实践

1、一个应用尽可能用一个 Topic，消息子类型用 tags 来标识，tags 可以由应用自由设置。只有发送消息设置了tags，消费方在订阅消息时，才可以利用 tags 在 broker 做消息过滤。
2、每个消息在业务层面的唯一标识码，要设置到 keys 字段，方便将来定位消息丢失问题。由于是哈希索引，请务必保证 key 尽可能唯一，这样可以避免潜在的哈希冲突。
3、消息发送成功或者失败，要打印消息日志，务必要打印 sendresult 和 key 字段。
4、对于消息不可丢失应用，务必要有消息重发机制。例如：消息发送失败，存储到数据库，能有定时程序尝试重发或者人工触发重发。
5、某些应用如果不关注消息是否发送成功，请直接使用 *sendOneWay* 方法发送消息。

二、Consumer最佳实践

1、消费过程要做到幂等（即消费端去重）
2、尽量使用批量方式消费方式，可以很大程度上提高消费吞吐量。
3、优化每条消息消费过程

三、其他配置

线上应该关闭 *autoCreateTopicEnable* ，即在配置文件中将其设置为 *false* 。

RocketMQ在发送消息时，会首先获取路由信息。如果是新的消息，由于MQServer上面还没有创建对应的 *Topic* ，这个时候，如果上面的配置打开的话，会返回默认TOPIC的（RocketMQ会在每台 *broker* 上面创建名为 *TBW102* 的TOPIC）路由信息，然后 *Producer* 会选择一台 *Broker* 发送消息，选中的 *broker* 在存储消息时，发现消息的 *topic* 还没有创建，就会自动创建 *topic* 。后果就是：以后所有该TOPIC的消息，都将发送到这台 *broker* 上，达不到负载均衡的目的。

所以基于目前RocketMQ的设计，建议关闭自动创建TOPIC的功能，然后根据消息量的大小，手动创建TOPIC。

RocketMQ设计相关

RocketMQ的设计假定：

> 每台PC机器都可能宕机不可服务
> 任意集群都有可能处理能力不足
> 最坏的情况一定会发生
> 内网环境需要低延迟来提供最佳用户体验

RocketMQ的关键设计：

> 分布式集群化
> 强数据安全
> 海量数据堆积
> 毫秒级投递延迟（推拉模式）

这是RocketMQ在设计时的假定前提以及需要到达的效果。我想这些假定适用于所有的系统设计。随着我们系统的服务的增多，每位开发者都要注意自己的程序是否存在单点故障，如果挂了应该怎么恢复、能不能很好的水平扩展、对外的接口是否足够高效、自己管理的数据是否足够安全..... 多多规范自己的设计，才能开发出高效健壮的程序。

参考资料

1. RocketMQ用户指南
2. RocketMQ原理简介
3. RocketMQ最佳实践
4. 阿里分布式开放消息服务(ONS)原理与实践2
5. 阿里分布式开放消息服务(ONS)原理与实践3

6. RocketMQ原理解析