本来只是想看下 metaq 的文档，但是发现深入到细节时，文档描述的有些模糊，现在 metaq 其实有两个大分支了，一个是庄晓丹维护的已开源的，另外一个是淘宝内部的，本质结构原理没太大区别，只不过开源的已经去掉了对淘系相关的依赖。然后淘系的 metaq 已经到 3.*版本了。干脆自己看 metaq 的源码，做个笔记记录下，怕我以后忘记了。有少量的章节和图片从内网拿来的，大部分是自己写的，记录下几个主要的点。

---- By 天向

一 metaq 是什么

metaq 是一个分布式消息中间件，消息中间件是典型的生产者-消费者模型，核心作用是解耦，生产者和消费者彼此没有直接依赖，同步化解成了异步。metaq 并没有遵循 jms 规范，jms 规范体现在系统层面和 api 层面。例如

- 消费模型

例如 jms 定义了两种消息传递方式：

1 基于队列的点对点消费模型

2 基于发布/订阅的消费模型

Metaq 只有发布订阅的消费方式。

- 消息类型

JMS 定义的消息类型有 TextMessage、MapMessage、BytesMessage、StreamMessage、ObjectMessage。Metaq 只有一种类型：Message。

- 消息持久性

JMS 定义两种持久性类型：

PERSISTENT 指示 JMS provider 持久保存消息，以保证消息不会因为 JMS provider 的失败而丢失。
NON_PERSISTENT 不要求 JMS provider 持久保存消息。

Metaq 的消息都是持久性的

- API

JMS 定义了消息中间件的生产端 api 和消费端 api，这些 api 都是约定的接口，都都被 metaq 无视了。

二　一些概念

摘自 metaq 百科

消息生产者

负责产生消息并发送消息到 meta 服务器

消息消费者

负责消息的消费，meta 采用 pull 模型，由消费者主动从 meta 服务器拉取数据并解析成消息并消费

Topic

消息的主题，由用户定义并在服务端配置。producer 发送消息到某个 topic 下，consumer 从某个 topic 下消费消息

分区

同一个 topic 下面还分为多个分区，如 meta-test 这个 topic 我们可以分为 10 个分区，分别有两台服务器提供，那么可能每台服务器提供 5 个分 区，假设服务器分别为 0 和 1，则所有分区为 0-0、0-1、0-2、0-3、0-4、1-0、1-1、1-2、1-3、1-4

Message

消息，负载用户数据并在生产者、服务端和消费者之间传输

Broker

就是 meta 的服务端或者说服务器，在消息中间件中也通常称为 broker。

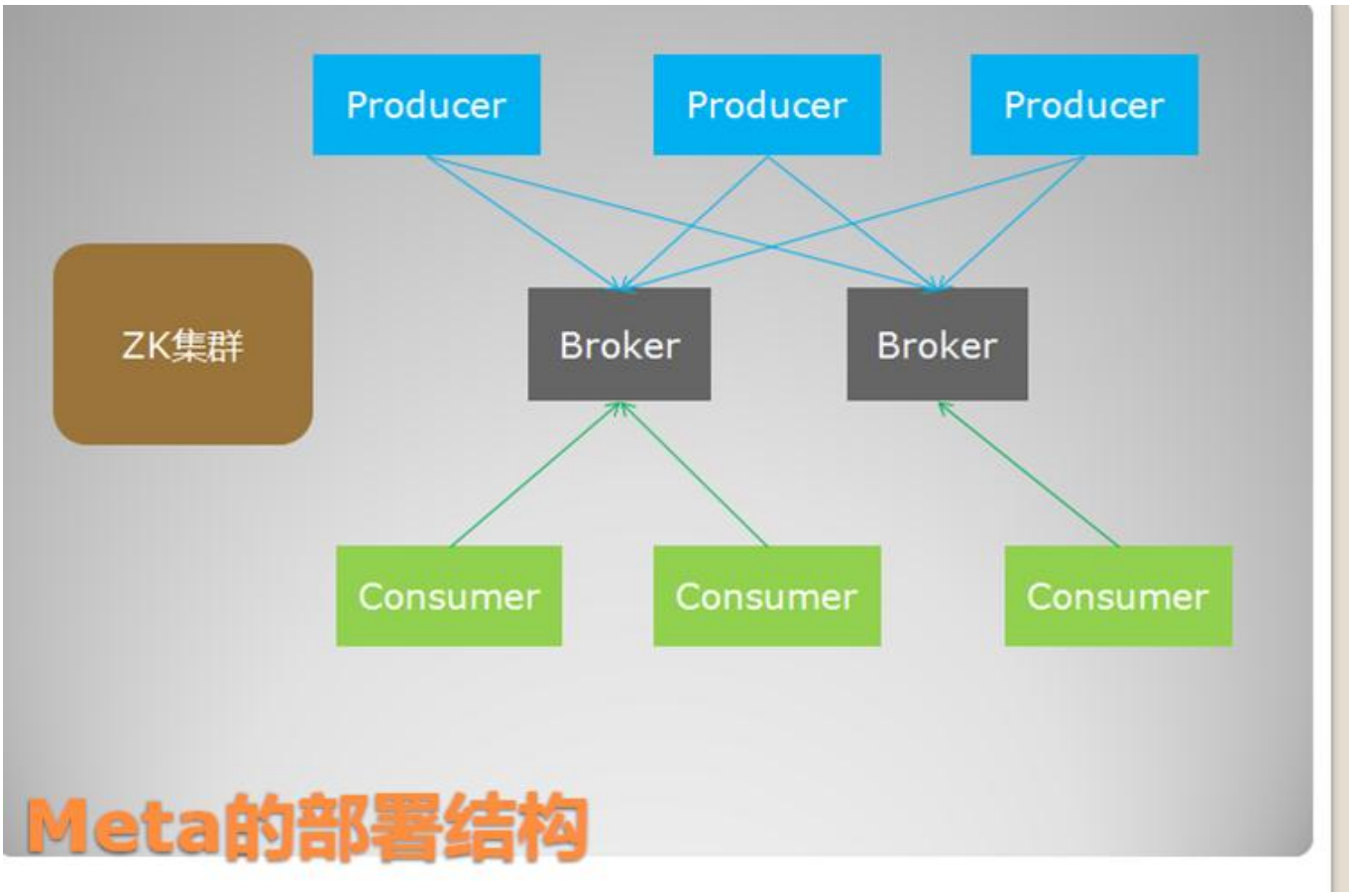
消费者分组(Group)

消费者可以是多个消费者共同消费一个 topic 下的消息，每个消费者消费部分消息。这些消费者就组成一个分组，拥有同一个分组名称,通常也称为消费者集群

Offset

消息在 broker 上的每个分区都是组织成一个文件列表，消费者拉取数据需要知道数据在文件中的偏移量，这个偏移量就是所谓 offset。Offset 是绝对偏移量，服务器会将 offset 转化为具体文件的相对偏移量

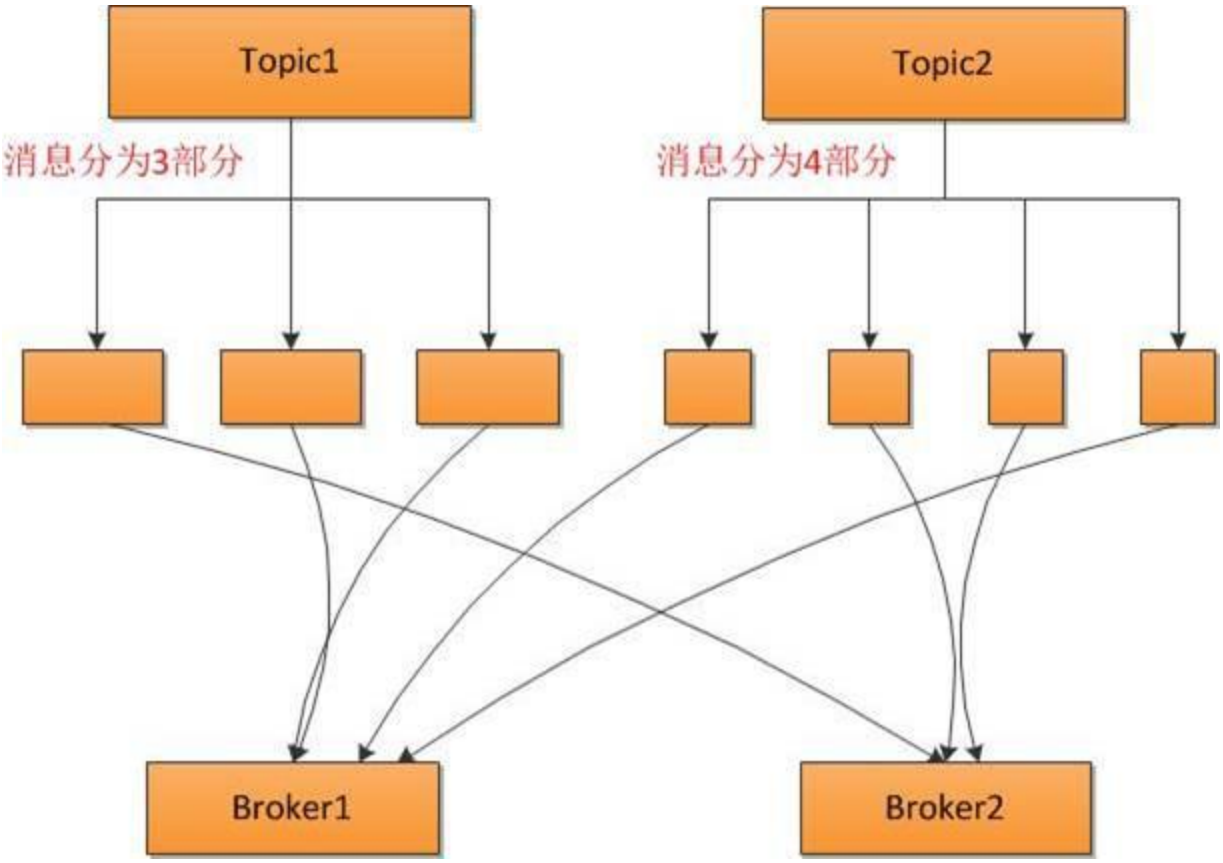
三　总体结构图



四　消息存储

消息中间件中消息堆积是很常见，这要求 broker 具有消息存储的能力，消息存储结构决定了消息的读写性能，对整体性能有很大影响，metaq 是分布式的，多个 borker 可以为一个 topic 提供服务，一个 topic 下的消息分散存储在多个 broker，它们是多对多关系。

如下图



消息定义

id

消息的唯一 id，系统自动产生，用户无法设置，在发送成功后由服务器返回，发送失败则为 0。

topic

消息的主题，订阅者订阅该主题即可接收发送到该主题下的消息，必须

data

消息的有效载荷，也就是消息内容，meta 永远不会修改消息内容，你发送出去是什么样子，接收到就是什么样子。

attribute

消息属性，一个字符串，可选。发送者可设置消息属性来让消费者过滤。

物理文件

metaq 将消息存储在本地文件中，每个文件最大大小为 1G，如果写入新的消息时，超过当前文件大小，则会自动新建一个文件。文件名称为起始字节大小，例如，假设文件最大尺寸为 1k，有三个文件，则文件名如

下(长度为 20 位，不足补 0)：

00000000000000000000

00000000000000001024

00000000000000002048

即使一个 broker 为多个 topic 服务，这些 topic 的消息都存储同一个文件组中，消息顺序写入，永远都是当前文件在写，其他文件只读。

索引文件

弄清消息的物理存储后，也许我们会有一个疑问：如何读取指定 topic 的当前消息？的确，仅仅只存储消息是无法做到这个的，所以 metaq 还有索引文件，类似数据库的索引，但是有很大区别。
broker 将消息存储到文件后，会将该消息在文件的物理位置，消息大小，消息类型封装成一个固定大小的数据结构，暂且称这个数据结构为索引单元吧，大小固定为 16k，消息在物理文件的位置称为 offset。

索引单元结构

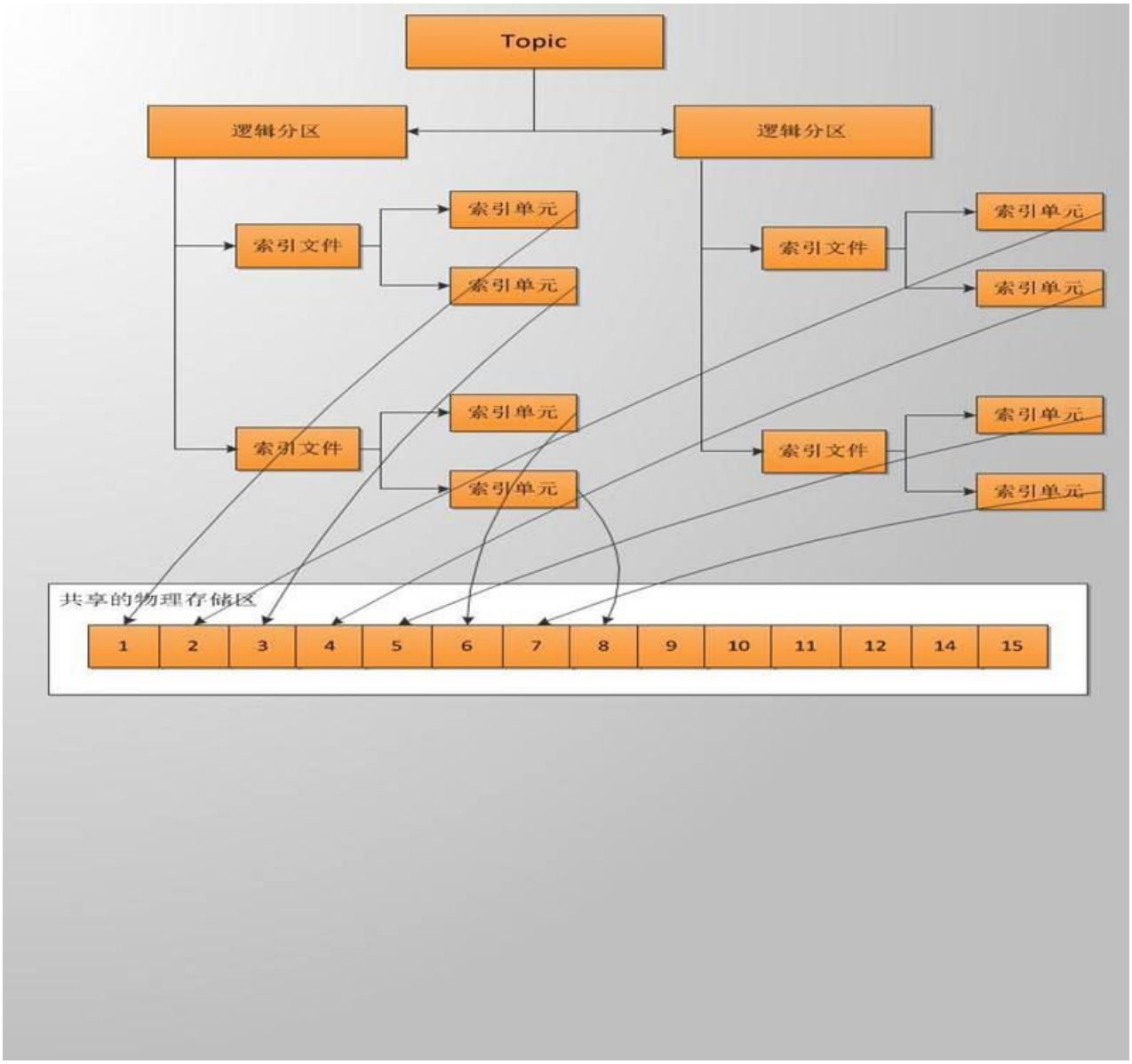
offset	size	messateType
8 字节	4 字节	4 字节

多个索引单元组成了一个索引文件，索引文件默认固定大小为 20M，和消息文件一样，文件名是起始字节位置，写满后，产生一个新的文件。

逻辑分区

一个逻辑分区实际上是一组索引文件。一个 topic 在一个 broker 上可以有多个逻辑分区，默认为 1，但可自由配置。为什么会有多个分区的情况？逻辑分区的作用不仅仅是通过索引提供快速定位消息的功能，它还跟整个 metaq 的集群有很大的关系。

逻辑结构图



五　集群与负载均衡

Topic 分布

一个 topic 可以分布在多台 broker 上，具体体现就是多个 broker 配置了这个 topic，并且最少有一个分区。假如有一个 topic 名为”t1”，两个 broker：b1,b2；每个 borker 都为 t1 配置了两个分区。那么 t1 一共有 4 个分区：b1-1,b1-2,b2-1,b2-2。生产者和消费者对 topic 发布消息或消费消息时，目的地都是以分区为单位。当一个 topic 消息量逐渐变大时，可以将 topic 分布在更多的 borker 上。

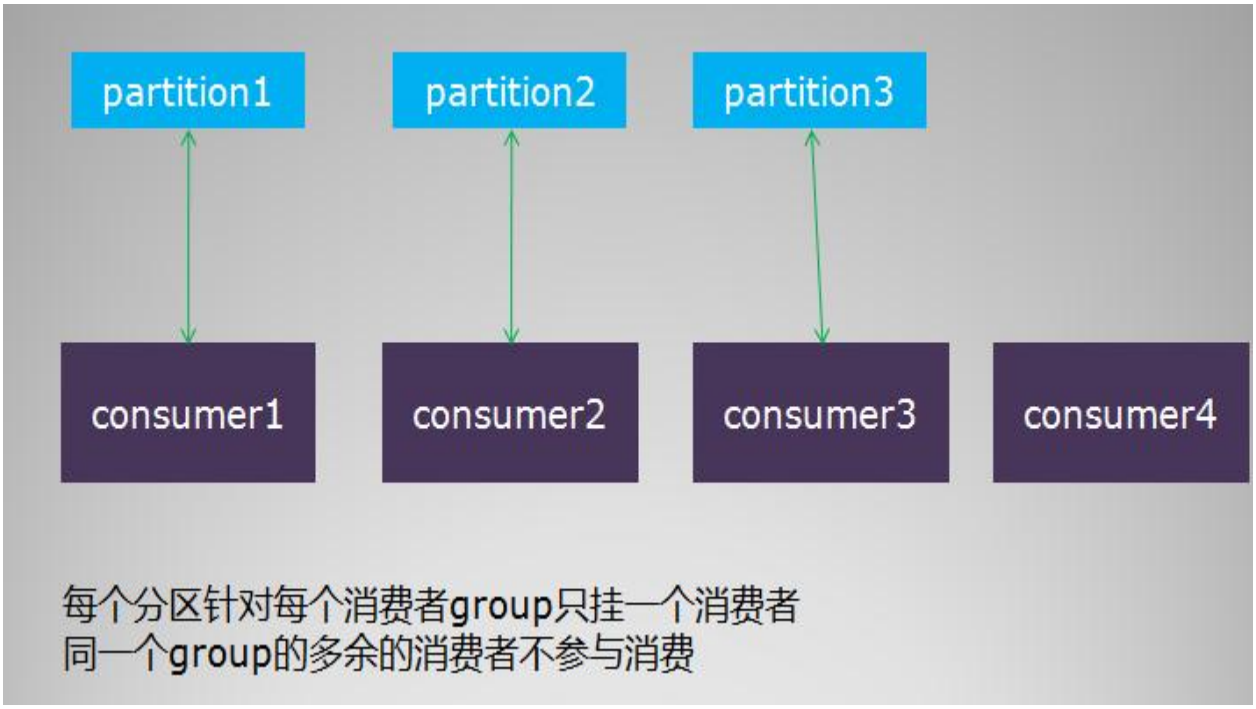
某个 broker 上的分区数越多，意味着该 borker 承担更繁重的任务，分区数可以认为是权重的表现形式。

生产者

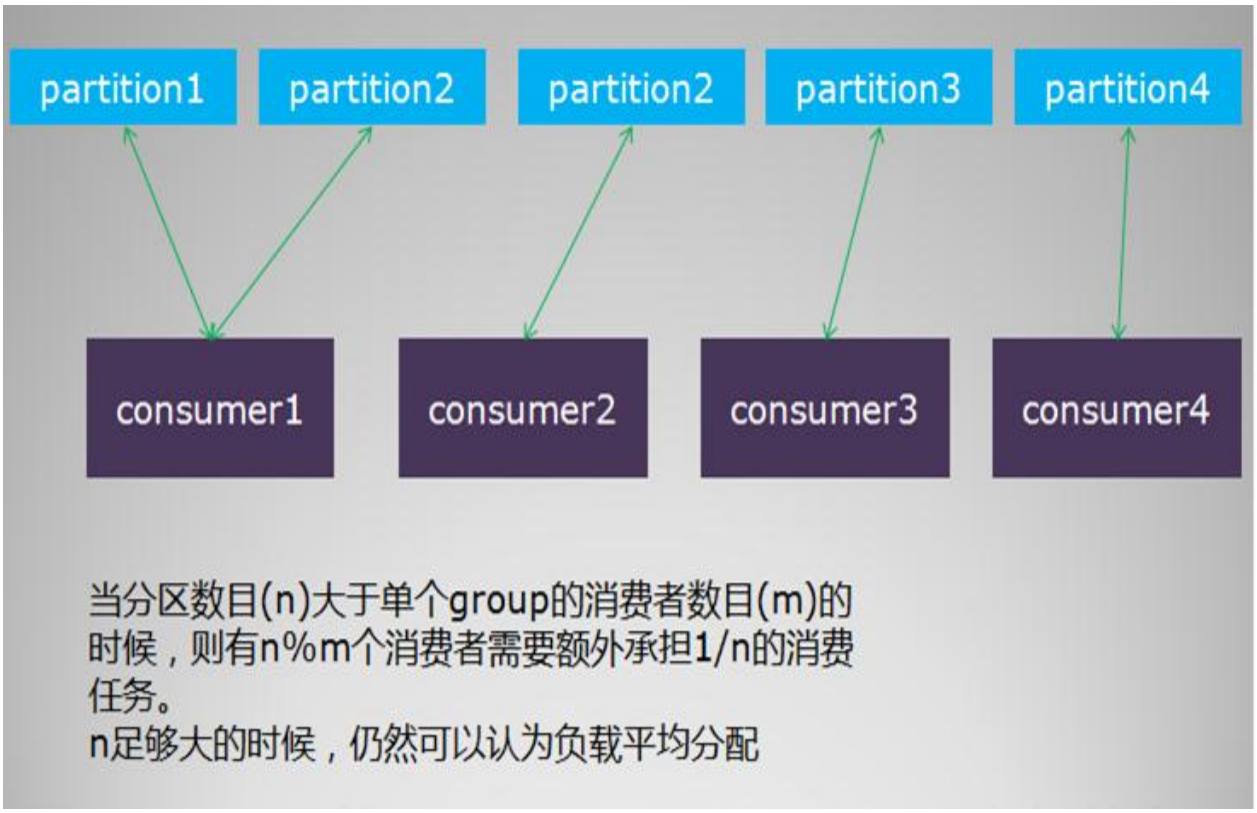
生产者在通过 zk 获取分区列表之后，会按照 brokerId 和分区号的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。这是默认的分区策略，考虑到我们的 broker 服务器软硬件配置基本一致，默认的轮询策略已然足够。如果你想实现自己的负载均衡策略，可以实现上文提到过的 PartitionSelector 接口，并在创建 producer 的时候传入即可。在 broker 因为重启或者故障等因素无法服务的时候，producer 通过 zookeeper 会感知到这个变化，将失效的分区从列表中移除做到 fail over。因为从故障到感知变化有一个延迟，可能在那一瞬间会有部分的消息发送失败。

消费者

消费者的负载均衡会相对复杂一些。我们这里讨论的是单个分组内的消费者集群的负载均衡，不同分组的负载均衡互不干扰，没有讨论的必要。 消费者的负载均衡跟 topic 的分区数目紧密相关，要考察几个场景。 首先是，单个分组内的消费者数目如果比总的分区数目多的话，则多出来的消费者不参与消费



其次，如果分组内的消费者数目比分区数目小，则有部分消费者要额外承担消息的消费任务，具体见示例图如下



六　文件读写

消息存储在文件中，如何保证性能？Metaq 使用了文件内存映射特性，对应的是 MappedByteBuffer 对象。 MappedByteBuffer 只是一种特殊的 ByteBuffer ，即是 ByteBuffer 的子类。 MappedByteBuffer 将文件直接映射到内存（这里的内存指的是虚拟内存，并不是物理内存）。通常，可以映射整个文件，如果文件比较大的话可以分段进行映射， 只要指定文件的那个部分就可以。而且，与 ByteBuffer 十分类似，没有构造函数（你不可 new MappedByteBuffer（ ）来构造一个 MappedByteBuffer），我们可以通过 java.nio.channels.FileChannel 的 map() 方法来获取 MappedByteBuffer 。其实说的通俗一点就是 Map 把文件的内容被映像到计算机虚拟内存的一块区域，这样就可以直接操作内存当中的数据而无需操作的时候每次都通过 I/O 去物理 硬盘读取文件，所以效率上有很大的提升。

映射方式

MappedByteBuffer map(int mode,long position,long size); 可以把文件的从 position 开始的 size 大小的区域映射为内存映像文件，mode 指出了可访问该内存映像文件的方式：

READ_ONLY,（只读）

试图修改将导致抛出异常

READ_WRITE（读/写）

对得到的缓冲区的更改最终将传播到文件；该更改对映射到同一文件的其他程序不一定是可见的。

PRIVATE（专用）

对得到的缓冲区的更改不会传播到文件，并且该更改对映射到同一文件的其他程序也不是可见的；相反，会创建缓冲区已修改部分的专用副本。

三个关键方法

fore()

缓冲区是 READ_WRITE 模式下，此方法对缓冲区内容的修改强行写入文件

load()

将缓冲区的内容载入内存，并返回该缓冲区的引用

isLoaded()

如果缓冲区的内容在物理内存中，则返回真，否则返回假

调用信道的 map()方法后，即可将文件的某一部分或全部映射到内存中，映射内存缓冲区是个直接缓冲区，继承自 ByteBuffer,但相对于 ByteBuffer,它有更多的优点： a. 读取快 b. 写入快 c. 随时随地写入

释放内存句柄

通过 FileChannel.map 方法可以得到一个 MappedByteBuffer，但 FileChannel 没有提供 unmap 方法，FileChannel 关闭后，不会释放映射的 MappedByteBuffer。导致的问题是一个 map 过的文件关闭后，却无法将其删除。根据 JAVADOC 的说明,是在垃圾收集的时候.而众所周知垃圾收集是程序根本无法控制的，有个土方：

```
AccessController.doPrivileged(new PrivilegedAction() {
 public Object run() {
  try {
   Method getCleanerMethod = buffer.getClass().getMethod("cleaner", new Class[0]);
   getCleanerMethod.setAccessible(true);
   sun.misc.Cleaner cleaner = (sun.misc.Cleaner)
   getCleanerMethod.invoke(byteBuffer, new Object[0]);
   cleaner.clean();
  } catch (Exception e) {
   e.printStackTrace();
```

```
    }

    return null;
  }
});
```

如果希望更加高效地处理映射到内存中的文件，把文件的内容加载到物理内存中是一个好办法。通过 MappedByteBuffer 类的 load 方法可以把该缓冲区所对应的文件内容加载到物理内存中，以提高文件操作时的性能。由于物理内存的容量受限，不太可能直接把一个大文件的全部内容一次性地加载到物理内存中。可以每次只映射文件的部分内容，把这部分内容完全加载到物理内存中进行处理。完成处理之后，再映射其他部分的内容。

在程序中对 MappedByteBuffer 做的修改不一定会立即同步到文件 系统中。如果在没有同步之前发生了程序错误，可能导致所做的修改丢失。因此，在执行完某些重要文件内容的更新操作之后，应该调用 MappedByteBuffer 类 的 force 方法来强制要求把这些更新同步到底层文件中。可以强制同步的更新有两类，一类是文件的数据本身的更新，另一类是文件的元数据的更新。在使用 force 方法时，可以通过参数来声明是否在同步数据的更新时也同步元数据的更新。

七 消息消费

metaq 的消费模型不是生产端推送，而是消费端不停拉取。但是注意，不停拉取不是指消费端定时拉取，而是拉取完一批消息，消费完毕，再去拉取下一批。这里有实时性和吞吐量之间的矛盾，如果每次批量拉取的消息数量过少，会增加实时性，但是减少吞吐量；反之，如果每次批量拉取的消息数量过大，则实时性会打折扣，但吞吐量上升。由于 metaq 的消息存储结构，消费端拉取消息时，至少需要以下几个参数：

- 消息主题
- 逻辑队列序号
- 索引起始位置
- 消息最大长度
- 当前请求序列号
- 消费者分组名称

Metaq 刚好也定义了这样的一个请求对象，刚好 6 个属性，分别对应前面所说的参数。

```
public class GetCommand{
    private final long offset;
```

```
    private final intmaxSize;
    private final int partition;
    private final String group;
    private Integer opaque;
    private String topic;
    ……
}
```

- 根据 topic 和 partition 找到逻辑队列：A
- 根据 offset 从 A 定位指定的索引文件：B
- 从 B 中读取所有的索引数据：C
- 遍历 C，根据索引单元的消息物理地址和消息长度，找到物理消息 D，将 D 放入集合，并计算消息的累加长度，若大于请求里消息最大长度 maxSize，则终止遍历，返回结果。

消息结果里有当前批次消息的索引读取结束位置（offset），消费端会将当前 offset 存储在本地，下次拉取消息时，要将结束位置作为参数放入消息拉取请求里。由于 metaq 是分布式结构，消费端和生产端的对应关系可能会经常变动，offset 不能仅仅只是保存到本地，必须保存在一个共享的存储里，比如 zookeeper，数据库，或共享的文件系统。默认情况下，metaq 将 offset 及时保存在本地，并定时写入 zookeeper。在某些情况下，会发生消息重复消费，比如某个 consumer 挂掉了，新的 consumer 将会接替它继续消费，但是 offset 是异步存储的，可能新的 consumer 起来后，从 zookeeper 上拿到的还是旧的 offset，导致当前批次重复，产生重复消费。

八：可靠性保证

生产者可靠性保证

消息生产者发送消息后返回 SendResult，如果 isSuccess 返回为 true,则表示消息已经确认发送到服务器并被服务器接收存储。整个发送过程是一个同步的过程。保证消息送达服务器并返回结果。

服务器可靠性保证

消息生产者发送的消息，meta 服务器收到后在做必要的校验和检查之后的第一件事就是写入磁盘，写入成功之后返回应答给生产者。因此，可以确认每条发送结果为成功的消息服务器都是写入磁盘的。写入磁盘，不意味着数据落到磁盘设备上，毕竟我们还隔着一层 os，os 对写有缓冲。Meta 有以下刷盘策略：

异步刷盘

每 1000 条（可配置），即强制调用一次 force 来写入磁盘设备。
每隔 10 秒（可配置），强制调用一次 force 来写入磁盘设备。

同步刷盘

此外，如果存储配置上的 groupCommitEnable 选项为 true，则会在写入消息后，立即强制刷盘。

消费者可靠性保证

消费者是一条接着一条地消费消息，只有在成功消费一条消息后才会接着消费下一条。如果在消费某条消息失败（如异常），则会尝试重试消费这条消 息（默认最大 5 次），超过最大次数后仍然无法消费，则将消息存储在消费者的本地磁盘，由后台线程继续做重试。而主线程继续往后走，消费后续的消息。因此， 只有在 MessageListener 确认成功消费一条消息后，meta 的消费者才会继续消费另一条消息。由此来保证消息的可靠消费。消费者的另一个可靠性的关键点是 offset 的存储，也就是拉取数据的偏移量。默认存储在 zoopkeeper 上，zookeeper 通过集群来保证数据的安全性。Offset 会定期保存，并且在每次重新负载均衡前都会强制保存一次，因此可能会存在极端情况下的消息的重复消费。

九 zookeeper 结构

Metaq 的集群信息，topic 分布/消费等，都记录在 zookeeper 上，如果很好的理解了 metaq 的 zk 结构，那对 metaq 的分布式会有很清楚的认识。结构如下：

/meta/brokers/ids

描述 broker 的注册信息

假如有 3 个 broker，id 分别为 m1,s1,s2,s1 和 s2 是 m1 的 slave(实际上这些 id 都是数字，不能有字母)。则结构为

/meta/brokers/ids/m1/master

/meta/brokers/ids/m1/slaves1

/meta/brokers/ids/m1/slaves2

m1 是 master brokerid，如果根据 m1 找 master brokerid，只需判断 m1/master 是否存在。如果寻找 m1 的 slave，只需找到 m1 下的 3 个节点，比对节点名称是否以"slave"字符串开头，若是，则截取 slave id 加入到 slave 节点集合。

/meta/brokers/topics

这个结构稍微有些复杂，还是举例说明吧。假如有以下 broker 信息:master m1,slave s1;master m2,slave s2;有一个 topic 名为”hello”,两组 broker 都配置了”hello”这个 topic。则目录如下：

/meta/brokers/topics/hello/m1-m

/meta/brokers/topics/hello/m2-m

/meta/brokers/topics/hello/s1-s

/meta/brokers/topics/hello/s2-s

-m 表示 master，-s 表示 slave，为什么要有这个结构呢？因为 producer 给某个 topic 推送消息时，需要知道哪些 broker 配置了该 topic。

根据 topic 获取 master 或者 slave，很简单，找到/meta/brokers/topics/hello 的子目录名称，然后判断是否以-m 或者-s 结尾，分别归类为 master 和 slave。不过拿到 master 或者 slave 的 brokeid 后，还需要按照 brokeid 检查 broker 是否存在。详情可以看 MetaZookeeper 的 getMasterBrokersByTopic 方法。

关于 topic 在 broker 上的分区信息，接着上面继续思考，仅仅知道哪些 borker 配置了某个 topic 还不够，因为 topic 在一个 broker 上还有分区信息。假如 hello 这个 topic 在 m1 上有 2 个分区，可以认为/meta/brokers/topics/hello 是一个目录，/meta/brokers/topics/hello/m1-m 是一个文件，那么 hello 这个 topic 在 m1 上的分区信息就是文件里的数据了。

/meta/brokers/topics/hello/m1-m 的数据是一个整数，某个 topic 在某个 broker 上的分区号是递增的，因此如果/meta/brokers/topics/hello/m1-m 的数据为 2，则表明 hello 在 m1 上的分区有 2 个。详情请看 MetaZookeeper 的 getPartitionsForTopicsFromMaster 方法。基于 /meta/brokers/topics 的结构，还可以查找某个 broker 发布了哪些 topic。假如存在以下目录

/meta/brokers/topics/hello1/m1-m
/meta/brokers/topics/hello1/m2-m
/meta/brokers/topics/hello1/s1-s
/meta/brokers/topics/hello1/s2-s
/meta/brokers/topics/hello2/m1-m
/meta/brokers/topics/hello2/m2-m
/meta/brokers/topics/hello2/s1-s
/meta/brokers/topics/hello2/s2-s

查找过程如下

- 找到/meta/brokers/topics 的所有子目录，得到 hello1 和 hello2，其实就是整个集群里有哪些 topic。
- 遍历每个 topic 的子目录，例如 hello1 的子目录为 m1-m，m2-m，s1-s，s2-s
- 遍历这些子目录，找到角色为 master 的 brokerid 是否和当前查找的 brokerid 一致，如果是，则将当前 topic 加入到指定 brokerid 发布的 topic 集合里。例如对于 m1 这个 brokerid，输出是 hello1，hello2。详情见 getTopicsByBrokerIdFromMaster 方法。

/meta/consumers/group/ids

存储某个分组的消费者注册信息，还有他们分别订阅了哪些 topic。group 是个变量，以消费者的实际分组为准。假设有一个消费者分组名为“hellogroup”，该分组有两个消费者，id 分别为"c1"和"c2"，c1 订阅了 topic "t1"和"t2",c3 订阅了"t3"和"t4"。则存在以下两个节点：

/meta/consumers/hellogroup/ids/hellogroup_c1　节点数据为“hello1,hello2”

/meta/consumers/hellogroup/ids/hellogroup_c2　节点数据为"hello2,hello3"

消费者 id 的计算规则

consumerId=所属分组名称+“_”+consumerUUID

如果构建一个消费端时，配置里指定了 consumerUUID，则以该 consumerUUID 为准，否则按照规则计算。见 ConsumerZookeeper 的 getConsumerUUID 方法：

```
 protected String getConsumerUUID(final ConsumerConfig consumerConfig) throws
Exception {
      String consumerUUID = null;
      if (consumerConfig.getConsumerId() != null) {
         consumerUUID = consumerConfig.getConsumerId();
      }else {
         consumerUUID =
               RemotingUtils.getLocalAddress() + "-" + System.currentTimeMillis() + "-"
                     + this.counter.incrementAndGet();
      }
      return consumerUUID;
 }
```

/meta/consumers/group/standby

group 是一个变量，以实际消费者分组名称为准，这个比较简单，存储的是一个数字，假设为 n，那么意思就是该分组的所有消费者都从第 n 个 slave 获取信息，禁止写入。默认情况下，该值为空，除非 master 挂掉，或者人工修改。有个问题待定：一个 topic 分布在多个 broker 上，每个 broker 的 slave 数量可能不一样，例如某个 broker 的 slave 数量 1，但是 n 却为 2。以此推测，这个配置可能是基于一个约定，就是每个 broker 的 slave 数量都是相同的。

/meta/consumers/group/offsets/topic

存储一个分组对某个 topic 不同分区的消费位置，group 和 topic 是变量，以实际值为准，假如一个 topic 名称
为 t1,部署在两台 broker：b1,b2；每个 broker 有两个分区。则一共有 4 个分区：b1-1,b1-2,b2-1,b2-2。一个
消费者分组“hellogroup”消费了这个 topic，b1-1,b1-2,b2-1,b2-2 的消费位置分别是 1,2,3,4；则有以下节点：

/meta/consumers/hellogroup/offsets/t1/b1-1　数据为 1
/meta/consumers/hellogroup/offsets/t1/b1-2　数据为 2
/meta/consumers/hellogroup/offsets/t1/b2-1　数据为 3
/meta/consumers/hellogroup/offsets/t1/b2-2　数据为 4

/meta/consumers/group/owners/topic

存储一个分组内，某个 topic 不同分区被哪个消费者消费了，group 和 topic 是变量，以实际值为准。假如一个 topic 名称为 t1,部署在 1 台 broker：b1；b1 有两个分区。则分区 id 为：b1-1,b1-2。一个分组“hellogroup
消费了这个 topic，消费者 id 分别为 c1,c2;c1 消费了 b1-1,c2 消费了 b1-2,则有以下节点：

/meta/consumers/hellogroup/owners/t1/b1-1　数据为 c1
/meta/consumers/hellogroup/owners/t1/b1-2　数据为 c2

十　通信框架

使用淘宝内部一个基于 nio 的通信框架 gecko，类似 tbremoting。实现方式和 api 使用都是类似的。不同的是 tbremoting 默认基于 mina 实现，而 gecko 全都是自己实现的。与 tbremoting 一样，gecko 也是基于 Handler 机制，向上提供 request/processor 方式进行业务处理。有关 mina 的资料介绍非常多，有兴趣可自己学习下，这里不做深入介绍。Gecko 的 hander 定义和 mina 很像。

```
public interface Handler {
    void onSessionCreated(Session session);
    void onSessionStarted(Session session);
    void onSessionClosed(Session session);
    void onMessageReceived(Session session, Object msg);
    void onMessageSent(Session session, Object msg);
    void onExceptionCaught(Session session, Throwablethrowable);
```

```
    void onSessionExpired(Session session);
    void onSessionIdle(Session session);
    void onSessionConnected(Session session, Object... args);
}
```

关注 void onMessageReceived(Session session, Object msg);当服务端或客户端收到消息后，就会触发这个方法。Session 为当前网络连接，msg 为收到的信息，网络中传输二进制数据，类似 mina，在过滤器链中，二进制数据与 java 对象之间会互相编码解码，不需要应用层关心。gecko 包装了 handler，对外只提供 request/processor 处理方式，意思是对于不同类型请求用相应的处理器处理。事实上 onMessageReceived 方法收到的 msg 只有两种对象:RequestCommand 和 ResponseCommand。分别代表了请求和响应。

```
void onMessageReceived(Session session, Object msg){
        ……
        if (message instanceofRequestCommand) {
            this.processRequest(session, message, defaultConnection);
        } else if (message instanceofResponseCommand) {
            this.processResponse(message, defaultConnection);
        }
        ……
}
```

看看 MetaMorphosisBroker 的 registerProcessors()就知道了。摘录片段如下：

```
this.remotingServer.registerProcessor(GetCommand.class, new
GetProcessor(this.brokerProcessor,
            this.executorsManager.getGetExecutor()));
this.remotingServer.registerProcessor(PutCommand.class, new
PutProcessor(this.brokerProcessor,
            this.executorsManager.getUnOrderedPutExecutor()));
this.remotingServer.registerProcessor(OffsetCommand.class, new
OffsetProcessor(this.brokerProcessor,
            this.executorsManager.getGetExecutor()));
```

以下是对应关系(不是全部的)，实际上，不同的 request 都有对应的通讯协议

GetCommand.class/GetProcessor;

PutCommand.class/PutProcessor;

OffsetCommand.class/OffsetProcessor

十一：通信协议

Meta 的协议是基于文本行的协议，类似 memcached 的文本协议。通用的协议格式如下

command params opaque\r\n body

其中 command 为协议命令，params 为参数列表，而 opaque 为协议的序列号，用于请求和应答的映射。客户端发送协议的时候需要自增此序列号， 而服务端将拷贝来自客户端的序列号并作为应答的序列号返回，客户端可根据应答的序列号将应答和请求对应起来。body 为协议体，可选，在协议头里需要有字 段指名 body 长度

Put 命令

参数

topic partition value-length flag [transactionKey]

说明

发送消息协议,topic 为发送的消息主题，partition 为发送的目的分区，value-length 为发送的消息体长度,flag 为消息标识位,transactionKey 为事务标识符，可选。

示例

put meta-test 0 5 0 1\r\nhello

get 命令

参数

topic group partition offset maxSize

说明

消费者拉取消息协议，topic 为拉取的消息主题，group 为消费者分组名称，partition 为拉取的目的分区，offset 为拉取的起始偏移量，maxSize 为本次拉取的最大数据量大小

示例

get meta-test example 0 1024 512 1\r\n

data 命令

参数

total-length

说明

get 请求返回的应答，total-length 返回的数据长度

示例
data 5 1\r\nhello

result 命令

参数
code length
说明
通用应答协议，如返回请求结果。code 为应答状态码，采用与 HTTP 应答状态码一样的语义。
length 为协议体长度
示例
result 200 0 1\r\n

offset 命令

参数
topic group partition offset
说明
查询离某个 offset 的最近有效的 offset,topic 为查询的消息主题，group 为消费者分组名称，
partition 为查询的分区,offset 为查询的 offset
示例
offset meta-test example 0 1024 1\r\n

stats 命令

参数
item(可选)
说明
查询服务器的统计情况，item 为查询的项目名称，如 realtime(实时统计),具体的某个 topic 等，可以为空
示例
stats 1\r\n

十二　异步复制

Meta 的 HA(High Availability)提供了在某些 Broker 出现故障时继续工作而不影响消息服务的可用性；跟 HA 关系紧密的就是 Failover，当故障 Server 恢复时能重新加入 Cluster 处理请求，这个过程对消息服务的使用者是透明的。Meta 基于 Master/Slave 实现 HA，Slave 以作为 Master 的订阅者（consumer）来跟踪消息记录，当消息发送到 Master 时候，Slave 会定时的获取此消息记录，并存储在自己的 Store 实现上；当 Master 出现故障无法继续使用了，消息还会在 Slave 上 Backup 的记录。

这种方式不影响原有的消息的记录，一旦 master 记录成功，就返回成功，不用等待在 slave 上是否记录；正因如此，slave 和 master 还有稍微一点的时间差异，在 Master 出故障 那一瞬间，或许有最新产生的消息，就无法同步到 slave；另外 Slave 可以作为 Consumer 的服务提供者，意思就是如果要写入必须通过 Master，消费时候可以从 Slave 上直接获取。

Failover 机制采用 client 端方式，Master 和 Slave 都需要注册到 ZK 上，一旦 Master 无法使用，客户端可使用与之对应的 Slave；当 Master 的故障恢复时候，这时候有两种方式处理：

1. 原来的 master 变成 Slave，Slave 变成 Master；恢复故障的 broker 作为 slave 去之前的 Slave 同步消息。优点简单，但是需要 slave 和 Master 有一样的配置和处理能力，这样就能取代 Master 的位置。（目前 Meta 采用此方式）
2. 需要自动把请求重新转移回恢复的 Master。实现复杂，需要再次把最新的消息从 Slave 复制会 Master，在复制期间还要考虑处理最新的消息服务（Producer 可以暂存消息在本地，等复制成功后再和 Broker 交互）。

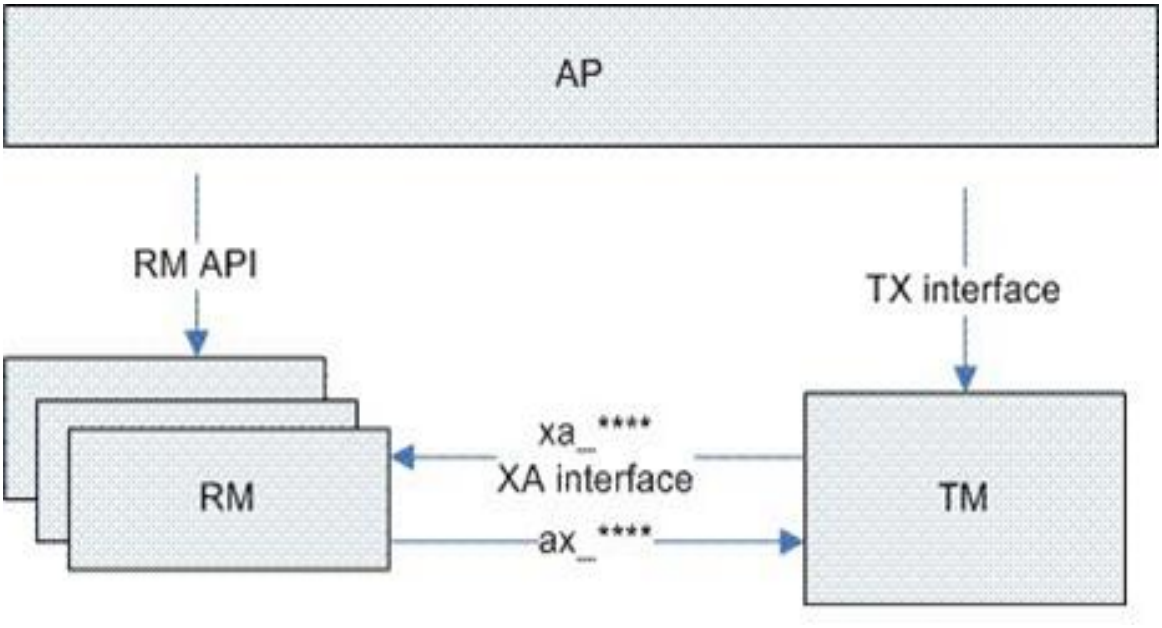
十三　分布式事务

metaq 提供分了布式事务的功能，说起分布式事务，就不能不提及 XA。X/Open 组织定义了分布式事务处理模型。

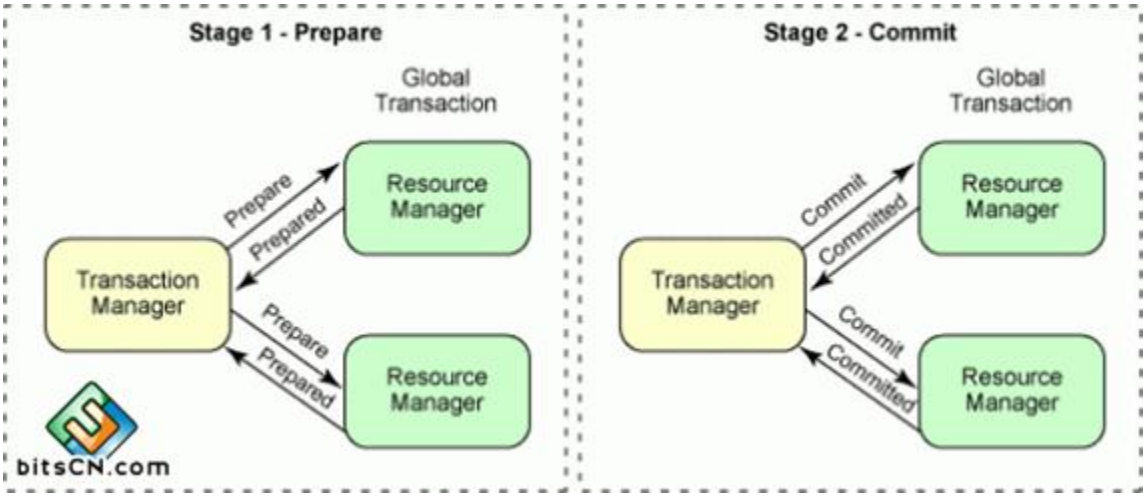
1. X/Open DTP 模型包括
2. 应用程序（ AP ）
3. 事务管理器（ TM ）
4. 资源管理器（ RM ）
5. 通信资源管理器（ CRM ）

一般，常见的资源管理器（ RM ）是数据库，常见的通信资源管理器（ CRM ）是消息中间件。

X/Open DTP 模型



二阶段提交示意图



XA 与 JTA 的关系

XA 是一个规范，JTA 也是一个规范，其实这两个规范是一样的，只不过 XA 跟语言无关，而 JTA 是 java 版的规范，进一步细化了 XA 规范，定义了明确清晰的接口。

JTA 的主要接口

UserTransaction 面向应用程序的接口，控制事务的开始、挂起、提交、回滚等

begin()
开始一个分布式事务，（在后台 TransactionManager 会创建一个 Transaction 事务对象并把此对象通过 ThreadLocale 关联到当前线程上）
commit()
提交事务（在后台 TransactionManager 会从当前线程下取出事务对象并把此对象所代表的事务提交）
rollback()
回滚事务（在后台 TransactionManager 会从当前线程下取出事务对象并把此对象所代表的事务回滚）
ugetStatus()
返回关联到当前线程的分布式事务的状态
usetRollbackOnly()
标识关联到当前线程的分布式事务将被回滚

Transaction
代表一个物理意义上的事务，UserTransaction 接口中的 commit()、rollback()，getStatus() 等方法都将最终委托给 Transaction 类的对应方法执行。
commit()　提交事务
rollback() 回滚事务
setRollbackOnly()　标识关联到当前线程的分布式事务将被回滚
getStatus()　返回关联到当前线程的分布式事务的状态
enListResource(XAResource xaRes, int flag)　将事务资源加入到当前的事务中
udelistResourc(XAResource xaRes, int flag)　将事务资源从当前事务中删除
uregisterSynchronization(Synchronization sync)　回调接口，在事务完成时得到通知从而触发一些处理工作。当事务成功提交后，回调程序将被激活。

TransactionManager
不承担实际事务处理功能，是用户接口和实现接口的桥梁。调用 UserTransaction.begin() 方法时 TransactionManager 会创建一个 Transaction 对象，并把此对象关联到当前线程上；同样 UserTransaction.commit() 会调用 TransactionManager.commit()，方法将从当前线程下取出事务对象 Transaction 并提交，即调用 Transaction.commit()。
begin()　开始事务
commit()　提交事务
rollback()　回滚事务
getStatus()　返回当前事务状态
setRollbackOnly()
getTransaction()　返回关联到当前线程的事务
setTransactionTimeout(int seconds)　设置事务超时时间
resume(Transaction tobj)　继续当前线程关联的事务

suspend() 挂起当前线程关联的事务

XAResource

这是一个非常重要的接口，是对底层事务资源的抽象，定义了分布式事务处理过程中事务管理器和资源管理器之间的协议。

commit() 提交事务

isSameRM(XAResource xares) 检查当前的 XAResource 与参数是否同一事务资源

prepare() 通知资源管理器准备事务的提交工作

rollback() 通知资源管理器回滚事务

消息提交和回滚

我们熟悉了前面的一些概念，分布式事务模型中有几个角色。metaq 和数据库一样其实是一个 RM，不过它没有遵守 JMS 的分布式事务标准，它对外呈现的就是一个 XAResource。可以粗略的讲，只有数据可能会发生修改，才需要事务来保证数据的完整性，如果只是读取数据，则不需要事务，因为事务需要成本（数据库读取数据也会有事务的，这个原因有很多方面，比如事务的隔离和 MVCC ）。所以，metaq 的事务主要发生在生产者，一个典型的场景示例如下：

- 应用程序向数据库写入一条记录
- 然后向 metaq 写入一条消息
- 然后再向数据库写入一条日志
- 如果日志写入失败，则前面步骤全部回滚
- 如果日志写入成功，则前面步骤全部提交

如果 metaq 调用处于分布式事务，则调用方式如下

```
XAMessageSessionFactory xaSF= new XAMetaMessageSessionFactory(new
MetaClientConfig());
XAMessageProducer xaProducer=xaSF.createXAProducer();
XAResource metaXares = producer.getXAResource();
/**
 *加入 JTA 事务  该接口最终会调用 XAResource.start 方法，即 metaXares.start(Xid,int)方法，
 *把该资源加入当前事务当中，发送一个带 XID 的事务命定，通知 Metaq 启动一个全局事务
 *分支,用 XID 标示该全局事务。
*/
tx.enlistResource(metaXares);
//事务中的业务操作  向 meta server 发送一条消息
String message="hello world!";
String topic="meta-test";
```

```
producer.sendMessage(new Message(topic, messate.getBytes());
```

看看两阶段提交和 XAResouce，XAMessageProducer 的 getXAResource()方法可得到一个 TransactionContext 对象，实现了 XAResource 接口。通过 UserTransaction. enListResource(XAResource xaRes, intflag)方法将当前 XAResource 加入到分布式事务里时，XAResource 的 start 方法将被调用。Start 方法向 metaq 的 broker 发送一个事务开始的命令，表示后续的操作都在分布式服务里，这些操作要暂存在事务文件里，不能直接写到消息队列里。TransactionContext 有 prepare()和 commit()方法，这两个方法对应着分布式事务提交的两个阶段。prepare 阶段，metaq 只是将生产者发送的消息暂存在本地的事务日志里，其实就是一个文件，commit 阶段才会从事务暂存文件里提取消息，写入到消息队列。