软件复用第二次讨论课方案

Metaq

**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。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，另外一个是在淘宝使用。

**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**：

 PERSISTEN: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\_PERSISTEN: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**

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

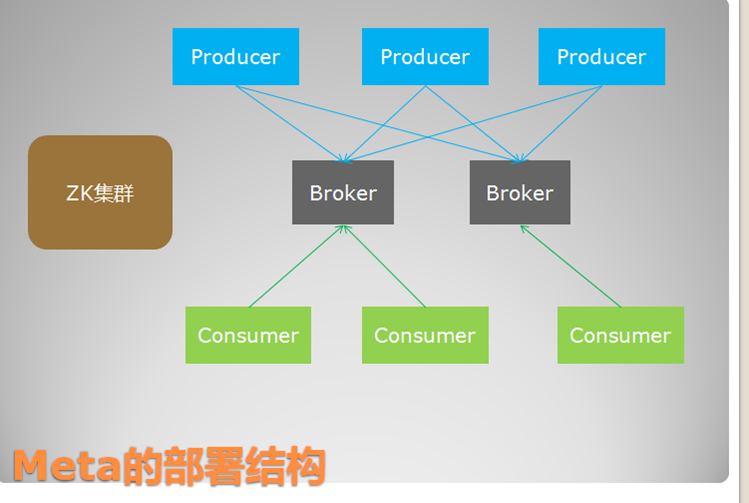
Message：消息，负载用户数据并在生产者、服务端和消费者之间传输

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

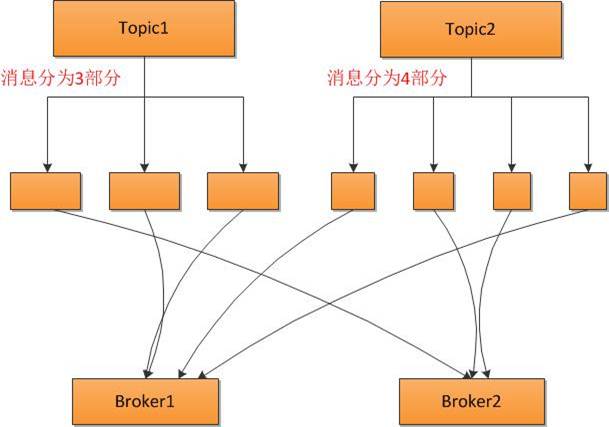
**三：总体结构图**



**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**

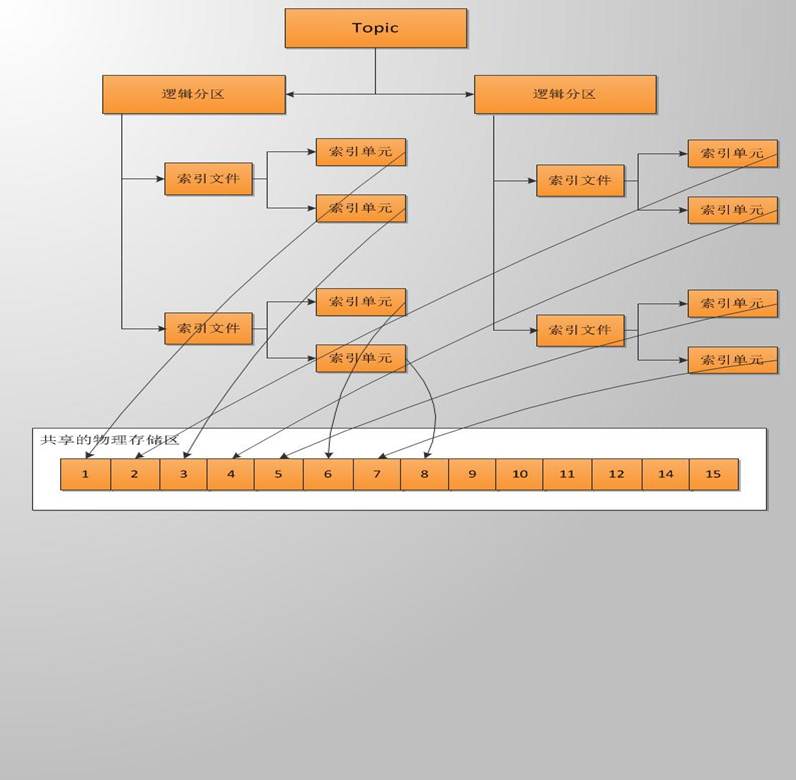
|  |  |  |
| --- | --- | --- |
| offset | size | messateType |
| 8字节 | 4字节 | 4字节 |

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**

T**opic分布**

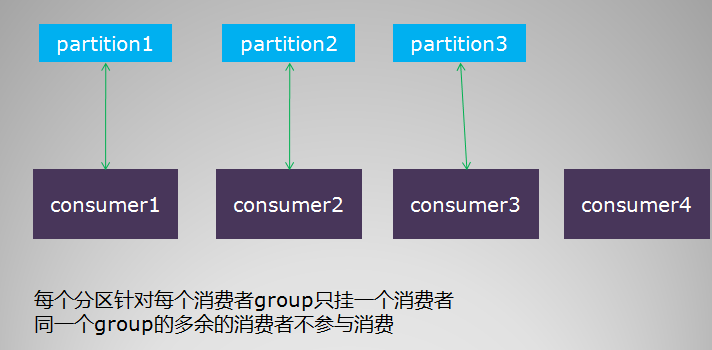
一个topic可以分布在多台broker上，具体体现就是多个broker配置了这个topic，并且最少有一个分区。假如有一个topic名为”t1”，两个broker：b1,b2；每个borker都为t1配置了两个分区。那么t1一共有4个分区：b1-1,b1-2,b2-1,b2-2。生产者和消费者对topic发布消息或消费消息时，目的地都是以分区为单位。当一个topic消息量逐渐变大时，可以将topic分布在更多的borker上。某个broker上的分区数越多，意味着该borker承担更繁重的任务，分区数可以认为是权重的表现形式。

**生产者**

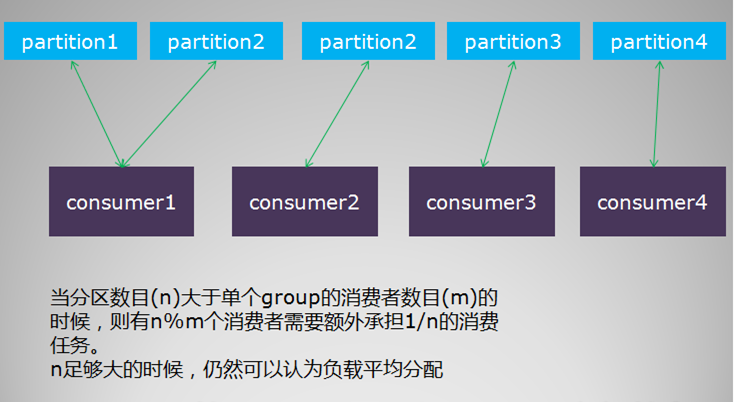
生产者在通过zk获取分区列表之后，会按照brokerId和分区号的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。这是默认的分区策略，考虑到我们的broker服务器软硬件配置基本一致，默认的轮询策略已然足够。如果你想实现自己的负载均衡策略，可以实现上文提到过的PartitionSelector接口，并在创建producer的时候传入即可。在broker因为重启或者故障等因素无法服务的时候，producer通过zookeeper会感知到这个变化，将失效的分区从列表中移除做到fail over。因为从故障到感知变化有一个延迟，可能在那一瞬间会有部分的消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

**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：

**Java代码**

1. AccessController.doPrivileged(**new** PrivilegedAction() {
2. **public** Object run() {
3. **try** {
4. Method getCleanerMethod = buffer.getClass().getMethod("cleaner", **new** Class[0]);
5. getCleanerMethod.setAccessible(**true**);
6. sun.misc.Cleaner cleaner = (sun.misc.Cleaner)
7. getCleanerMethod.invoke(byteBuffer, **new** Object[0]);
8. cleaner.clean();
9. } **catch** (Exception e) {
10. e.printStackTrace();
11. }
13. **return** **null**;
14. }
15. });

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。由于I/O操作一般比较耗时，出于性能考虑，很多操作在操作系统内部都是使用缓存的。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。

**Java代码**

1. **public** **class** GetCommand{
2. **private** **final** **long** offset;
3. **private** **final** intmaxSize;
4. **private** **final** **int** partition;
5. **private** **final** String group;
6. **private** Integer opaque;
7. **private** String topic;
8. ……
9. }

* 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存储结构**

**/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：

**Java代码**

1. **protected** String getConsumerUUID(**final** ConsumerConfig consumerConfig) **throws** Exception {
2. String consumerUUID = **null**;
3. **if** (consumerConfig.getConsumerId() != **null**) {
4. consumerUUID = consumerConfig.getConsumerId();
5. }**else** {
6. consumerUUID =
7. RemotingUtils.getLocalAddress() + "-" + System.currentTimeMillis() + "-"
8. + **this**.counter.incrementAndGet();
9. }
10. **return** consumerUUID;
11. }

**/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。

**Java代码**

1. **public** **interface** Handler {
2. **void** onSessionCreated(Session session);
3. **void** onSessionStarted(Session session);
4. **void** onSessionClosed(Session session);
5. **void** onMessageReceived(Session session, Object msg);
6. **void** onMessageSent(Session session, Object msg);
7. **void** onExceptionCaught(Session session, Throwablethrowable);
8. **void** onSessionExpired(Session session);
9. **void** onSessionIdle(Session session);
10. **void** onSessionConnected(Session session, Object... args);
11. }

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。

**Java代码**

1. **void** onMessageReceived(Session session, Object msg){
2. ……
3. **if** (message instanceofRequestCommand) {
4. **this**.processRequest(session, message, defaultConnection);
5. } **else** **if** (message instanceofResponseCommand) {
6. **this**.processResponse(message, defaultConnection);
7. }
8. ……
9. }

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

**Java代码  [收藏代码](javascript:void())**

1. **this**.remotingServer.registerProcessor(GetCommand.**class**, **new** GetProcessor(**this**.brokerProcessor,
2. **this**.executorsManager.getGetExecutor()));
3. **this**.remotingServer.registerProcessor(PutCommand.**class**, **new** PutProcessor(**this**.brokerProcessor,
4. **this**.executorsManager.getUnOrderedPutExecutor()));
5. **this**.remotingServer.registerProcessor(OffsetCommand.**class**, **new** OffsetProcessor(**this**.brokerProcessor,
6. **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）。

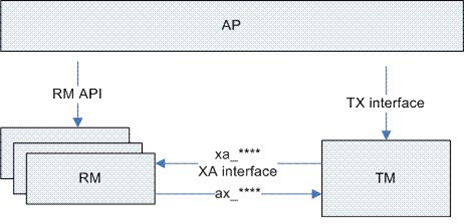
**十三：分布式事务**

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。

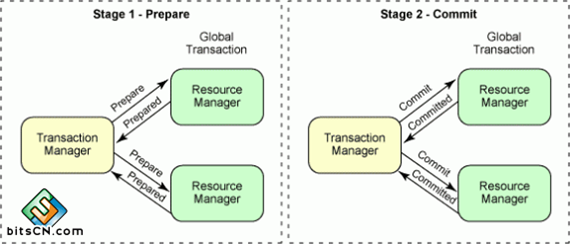
1. X/Open DTP 模型包括
2. 应用程序（ AP ）
3. 事务管理器（ TM ）
4. 资源管理器（ RM ）
5. 通信资源管理器（ CRM ）

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

**Java代码**

1. XAMessageSessionFactory xaSF= **new** XAMetaMessageSessionFactory(**new** MetaClientConfig());
2. XAMessageProducer xaProducer=xaSF.createXAProducer();
3. XAResource metaXares = producer.getXAResource();
4. /\*\*
5. \*加入JTA事务  该接口最终会调用XAResource.start方法，即metaXares.start(Xid,int)方法，
6. \*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
7. \*分支,用XID标示该全局事务。
8. \*/
9. tx.enlistResource(metaXares);
10. //事务中的业务操作  向meta server发送一条消息
11. String message="hello world!";
12. String topic="meta-test";
13. 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，这些操作要暂存是事务文件里，不能直接写到消息队列里。ransactionContext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。