Memorphosis是一个消息中间件，它是linkedin开源MQ——kafka的Java版本，针对淘宝内部应用做了定制和优化。Metamorphosis的设计原则

* 消息都是持久的，保存在磁盘
* 吞吐量第一
* 消费状态保存在客户端
* 分布式，生产者、服务器和消费者都可分布

Metamorphosis的部署结构

**[] Metamorphosis的特点**

除了完整实现kafka的功能之外，我们还为meta加入了额外的功能，使得meta成为一个更为强大的通用消息中间件，包括

* 彻底用java重写的实现，高效的协议和通讯框架
* 发送端的负载均衡
* Master/Slave异步和同步复制的高可用方案
* 专门用于广播消息的客户端实现
* 与diamond结合使用的顺序发送消息功能
* 支持事务，包括本地事务和分布式事务，实现JTA规范。

**[] Getting started**

我们在日常已经部署了metamorhposis环境，因此你可以直接在本地测试，如果你想部署一个自己的服务器，可以参照#服务器部署节。

前面提到，meta是一个消息中间件。消息中间件中有两个角色：消息生产者和消息消费者。Meta里同样有这两个概念，消息生产者负责创建消息并发送到meta服务器，meta服务器会将消息持久化到磁盘，消息消费者从meta服务器拉取消息并提交给应用消费。

**[] 消息会话工厂类**

在使用消息生产者和消费者之前，我们需要创建它们，这就需要用到消息会话工厂类——MessageSessionFactory，由这个工厂帮你创建生产者或者消费者。除了这些，MessageSessionFactory还默默无闻地在后面帮你做很多事情，包括

1. 服务的查找和发现，通过diamond和zookeeper帮你查找日常的meta服务器地址列表
2. 连接的创建和销毁，自动创建和销毁到meta服务器的连接，并做连接复用，也就是到同一台meta的服务器在一个工厂内只维持一个连接。
3. 消息消费者的消息存储和恢复，后续我们会谈到这一点。
4. 协调和管理各种资源，包括创建的生产者和消费者的。

因此，我们首先需要创建一个会话工厂类，MessageSessionFactory仅是一个接口，它的实现类是MetaMessageSessionFactory

MessageSessionFactory sessionFactory = new MetaMessageSessionFactory(new MetaClientConfig());

**请注意,MessageSessionFactory应当全局共用一个**

**[] 消息生产者**

翠花，上代码

package com.taobao.metamorphosis.example;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import com.taobao.metamorphosis.Message;

import com.taobao.metamorphosis.client.MessageSessionFactory;

import com.taobao.metamorphosis.client.MetaClientConfig;

import com.taobao.metamorphosis.client.MetaMessageSessionFactory;

import com.taobao.metamorphosis.client.producer.MessageProducer;

import com.taobao.metamorphosis.client.producer.SendResult;

public class Producer {

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

// New session factory,强烈建议使用单例

MessageSessionFactory sessionFactory = new MetaMessageSessionFactory(new MetaClientConfig());

// create producer,强烈建议使用单例

MessageProducer producer = sessionFactory.createProducer();

// publish topic

final String topic = "meta-test";

producer.publish(topic);

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

String line = null;

while ((line = reader.readLine()) != null) {

// send message

SendResult sendResult = producer.sendMessage(new Message(topic, line.getBytes()));

// check result

if (!sendResult.isSuccess()) {

System.err.println("Send message failed,error message:" + sendResult.getErrorMessage());

}

else {

System.out.println("Send message successfully,sent to " + sendResult.getPartition());

}

}

}

}

消息生产者的接口是MessageProducer，你可以通过它来发送消息。创建生产者很简单，通过MessageSessionFactory的createProducer方法即可以创建一个生产者。在Meta里，每个消息对象都是Message类的实例，Message表示一个消息对象，它包含这么几个属性：

|  |  |
| --- | --- |
| **属性** | **值** |
| 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。消息内容限制在1M以内，我的建议是最好不要发送超过上百K的消息，必须 |
| 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。 |

细心的朋友可能注意到，我们在sendMessage之前还调用了MessageProducer的publish(topic)方法

producer.publish(topic);

这一步在发送消息前是**必须**的，你必须发布你将要发送消息的topic，这是为了让会话工厂帮你去查找接收这些topic的meta服务器地址并初始化连接。这个步骤针对每个topic只需要做一次，多次调用无影响。

总结下这个例子，从标准输入读入你输入的数据，并将数据封装成一个Message对象，发送到topic为meta-test下。

请注意，**MessageProducer是线程安全的，完全可重复使用，因此最好在应用中作为单例来使用，一次创建，到处使用，配置为spring里的singleton bean。MessageProducer创建的代价昂贵，每次都需要通过zk查找服务器并创建tcp长连接。**

**[] 消息消费者**

发送消息后，消费者可以接收消息了，下面的代码创建消费者并订阅meta-test这个主题，等待消息送达并打印消息内容

package com.taobao.metamorphosis.example;

import java.util.concurrent.Executor;

import com.taobao.metamorphosis.Message;

import com.taobao.metamorphosis.client.MessageSessionFactory;

import com.taobao.metamorphosis.client.MetaClientConfig;

import com.taobao.metamorphosis.client.MetaMessageSessionFactory;

import com.taobao.metamorphosis.client.consumer.ConsumerConfig;

import com.taobao.metamorphosis.client.consumer.MessageConsumer;

import com.taobao.metamorphosis.client.consumer.MessageListener;

public class AsyncConsumer {

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

// New session factory,强烈建议使用单例

MessageSessionFactory sessionFactory = new MetaMessageSessionFactory(new MetaClientConfig());

// subscribed topic

final String topic = "meta-test";

// consumer group

final String group = "meta-example";

// create consumer,强烈建议使用单例

MessageConsumer consumer = sessionFactory.createConsumer(new ConsumerConfig(group));

// subscribe topic

consumer.subscribe(topic, 1024 \* 1024, new MessageListener() {

public void recieveMessages(Message message) {

System.out.println("Receive message " + new String(message.getData()));

}

public Executor getExecutor() {

// Thread pool to process messages,maybe null.

return null;

}

});

// complete subscribe

consumer.completeSubscribe();

}

}

通过createConsumer方法来创建MessageConsumer，注意到我们传入一个ConsumerConfig参数，这是消费者的配置对象。每个消息者都必须有一个ConsumerConfig配置对象，我们这里只设置了group属性，这是消费者的分组名称。Meta的Producer、Consumer和Broker都可以为集群。消费者可以组成一个集群共同消费同一个topic，发往这个topic的消息将按照一定的负载均衡规则发送给集群里的一台机器。同一个消费者集群必须拥有同一个分组名称，也就是同一个group，这个概念跟notify里的订阅者组名是一样的。我们这里将分组名称设置为meta-example。

订阅消息通过subscribe方法，这个方法接受三个参数

* topic，订阅的主题
* maxSize，因为meta是一个消费者主动拉取的模型，这个参数规定每次拉取的最大数据量，单位为字节，这里设置为1M，最大为1M。
* MessageListener，消息监听器，负责消息消息。

MessageListener的接口方法如下

public interface MessageListener {

/\*\*

\* 接收到消息列表，只有messages不为空并且不为null的情况下会触发此方法

\*

\* @param messages

\*/

public void recieveMessages(Message message);

/\*\*

\* 处理消息的线程池

\*

\* @return

\*/

public Executor getExecutor();

}

消息的消费过程可以是一个并发处理的过程，getExecutor返回你想设置的线程池，每次消费都会在这个线程池里进行。recieveMessage方法用于实际的消息消费处理，message参数即为消费者收到的消息，它必不为null。

我们这里简单地打印收到的消息内容就完成消费。如果在消费过程中抛出任何异常，该条消息将会在一定间隔后重新尝试提交给MessageListener消费。在多次消费失败的情况下，该消息将会存储到消费者应用的本次磁盘，并在后台自动恢复重试消费。

细心的你一定还注意到，在调用subscribe之后，我们还调用了completeSubscribe方法来完成订阅过程。请注意，**subscribe仅是将订阅信息保存在本地，并没有实际跟meta服务器交互，要使得订阅关系生效必须调用一次completeSubscribe，completeSubscribe仅能被调用一次，多次调用将抛出异常**。 为什么需要completeSubscribe方法呢，原因有二：

* 首先，subscribe方法可以被调用多次，也就是一个消费者可以消费多种topic
* 其次，如果每次调用subscribe都跟zk和meta服务器交互一次，代价太高

因此completeSubscribe一次性将所有订阅的topic生效，并处理跟zk和meta服务器交互的所有过程。

同样,**MessageConsumer也是线程安全的，创建的代价不低，因此也应该尽量复用。**

**[] 例子小结**

上面的例子可以直接在您的机器上跑起来，因为我们在日常已经部署了几台meta机器。不过我们建议您测试的时候使用自己的topic和消费者组名group，防止跟其他测试的开发者产生冲突，如有疑问，可以联系伯岩(boyan@taobao.com),无花(wuhua@taobao.com),

此例子的代码可以在metamorphosis-example工程下找到，

你可以在这里找到所有meta的例子源码。

**[] 事务**

Metamorphosis 1.2开始支持事务，包括发送端和消费端事务。发送端同时支持本地事务和分布式事务，可以在一个事务内发送多条消息，要么同时成功，要么同时失败；可以使用XA事务，在事务内操作其他XA资源，例如操作数据库，与此同时发送meta消息，可以保证这些操作和发送消息要么一起成功，要么一起失败。

在消费消息的时候，可以批量消费一批消息，要么一起消费成功，要么失败重试。

**[] 发送消息的本地事务**

事务跟线程关联，启动一个事务将会关联该事务到当前线程，在此线程和此事务内发送的消息，将作为一个整体发送，同时成功，或者同时失败，对外界看来是一个原子操作。发起一个本地事务很简单，参见代码：

try {

// 开始事务

producer.beginTransaction();

// 在事务内发送两条消息

if (!producer.sendMessage(new Message(topic, line.getBytes())).isSuccess()) {

// 发送失败，立即回滚

producer.rollback();

continue;

}

if (!producer.sendMessage(new Message(topic, line.getBytes())).isSuccess()) {

producer.rollback();

continue;

}

// 提交

producer.commit();

}

catch (final Exception e) {

producer.rollback();

}

beginTransaction方法启动一个事务并关联到当前线程，commit方法提交事务，而rollback则回滚当前事务。

**[] 发送消息的分布式事务**

如果你要在发送消息的同时操作数据库，比如同时将消息插入某张表，例如下订单的时候同时发送消息通知卖家并将订单插入数据库，这时候因为涉及到两个Resource（数据库和meta)，就需要使用分布式事务来保证ACID。分布式事务一般采用两阶段提交协议，在java里就是使用JTA规范API的XA部分。

在这种情形下，你需要使用数据库的XADatasource和meta的XAMessageProducer类，并使用一个开源JTA实现来支持事务管理器做协调者。例如我们在metamorphosis-example里的XATransactionProducer例子使用了atomikos这个开源JTA实现，具体不在这里讲解，请直接参考源码并尝试运行。

**[] 消费者的事务**

参见 TransactionalConsumer 例子。

**[] 概念和API详解**

**[] 术语解释**

**[] 消息生产者**

也称为Message Producer，一般简称为producer，负责产生消息并发送消息到meta服务器。

**[] 消息消费者**

也称为Message Consumer，一般简称为consumer，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(partition)**

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。

分区跟消费者的负载均衡机制有很大关系，具体见#消费者的负载均衡。

**[] 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。详细内容参见#消息的存储结构

**[] 客户端API**

**[] MessageSessionFactory接口**

参见javadoc MessageSessionFactory

主要实现类 MetaMessageSessionFactory

**[] MessageProducer接口**

参见javadoc MessageProducer

主要实现类 SimpleMessageProducer

**[] PartitionSelector接口**

分区选择器，用于从分区列表中选中将要发送消息的分区，参见javadoc PartitionSelector

主要实现类，轮询分区选择器 RoundRobinPartitionSelector

客户端可自定义分区选择器，并在创建生产者的时候注入。

**[] SendResult类**

发送结果信息,参见javadoc SendResult

**[] MessageConsumer接口**

参见javadoc MessageConsumer主要实现类 SimpleMessageConsumer

**[] MessageListener接口**

消息监听器，处理消费消息，参见javadoc MessageListener

**[] OffsetStorage接口**

Offset存储器，参见javadoc OffsetStorage，用户可自定义实现自己的存储器，默认提供下列三种存储器

1. ZkOffsetStorage,存储在zookeeper
2. MysqlOffsetStorage，存储在mysql数据库
3. LocalOffsetStorage，存储在本地文件，适合消费者分组只有一个消费者的情况，无需共享offset信息。

**[] 可靠性、顺序和重复**

**[] 可靠性**

Metamorphosis的可靠性保证贯穿客户端和服务器。

**[] 生产者的可靠性保证**

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。

**[] 服务器的可靠性保证**

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有两个特性来保证数据落到磁盘上

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

因此,Meta通过配置可保证在异常情况下（如磁盘掉电）10秒内最多丢失1000条消息。

服务器通常组织为一个集群，一条从生产者过来的消息可能按照路由规则存储到集群中的某台机器。Meta还正在实现高可用的HA方案，类似mysql的异步复制，将一台meta服务器的数据完整复制到另一台slave服务器，并且slave服务器还提供消费功能，本方案正在实现中，敬请期待。

**[] 消费者的可靠性保证**

消息的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。我们目前提供了以下几种存储方案

1. zookeeper，默认存储在zoopkeeper上，zookeeper通过集群来保证数据的安全性。
2. mysql，可以连接到您使用的mysql数据库，只要建立一张特定的表来存储。完全由数据库来保证数据的可靠性。
3. file，文件存储，将offset信息存储在消费者的本地文件中。

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。

**[] 顺序**

很多人关心的消息顺序，希望消费者消费消息的顺序跟消息的发送顺序是一致的。比如，我发送消息的顺序是A、B、C，那么消费者消费的顺序也应该是A、B、C。乱序对某些应用可能是无法接受的。

Metamorphosis对消息顺序性的保证是有限制的，默认情况下，消息的顺序以谁先达到服务器并写入磁盘，则谁就在先的原则处理。并且，发往同一个分区的消息保证按照写入磁盘的顺序让消费者消费，这是因为消费者针对每个分区都是按照从前到后递增offset的顺序拉取消息。

Meta可以保证，在**单线程内使用该producer发送的消息按照发送的顺序达到服务器并存储，并按照相同顺序被消费者消费，前提是这些消息发往同一台服务器的同一个分区**。为了实现这一点，你还需要实现自己的**PartitionSelector**用于固定选择分区

package com.taobao.metamorphosis.client.producer;

import java.util.List;

/\*\*

\* 分区选择器

\*

\* @author boyan

\* @Date 2011-4-26

\*

\*/

public interface PartitionSelector {

/\*\*

\* 根据topic、message从partitions列表中选择分区

\*

\* @param topic

\* topic

\* @param partitions

\* 分区列表

\* @param message

\* 消息

\* @return

\* @throws MetaClientException

\* 此方法抛出的任何异常都应当包装为MetaClientException

\*/

public Partition getPartition(String topic, List<Partition> partitions, Message message) throws MetaClientException;

}

选择分区可以按照一定的业务逻辑来选择，如根据业务id来取模。或者如果是传输文件，可以固定选择第n个分区使用。当然，如果传输文件，通常我们会建议你只配置一个分区，那也就无需选择了。

消息的顺序发送我们在1.2这个版本提供了OrderedMessageProducer，利用diamond来管理分区信息，并提供故障情况下的本地存储功能。

**[] 消息重复**

消息的重复包含两个方面，生产者重复发送消息以及消费者重复消费消息。

针对生产者来说，有可能发生这种情况，生产者发送消息，等待服务器应答，这个时候发生网络故障，服务器实际已经将消息写入成功，但是由于网络故障没有返回应答。那么生产者会认为发送失败，则再次发送同一条消息，如果发送成功，则服务器实际存储两条相同的消息。这种由故障引起的重复，meta是无法避免的，因为meta不判断消息的data是否一致，因为它并不理解data的语义，而仅仅是作为载荷来传输。

针对消费者来说也有这个问题，消费者成功消费一条消息，但是此时断电，没有及时将前进后的offset存储起来，则下次启动的时候或者其他同个分组的消费者owner到这个分区的时候，会重复消费该条消息。这种情况meta也无法完全避免。

Meta对消息重复的保证只能说在正常情况下保证不重复，异常情况无法保证，这些限制是由远程调用的语义引起的，要做到完全不重复的代价很高，meta暂时不会考虑。

**[] 集群和负载均衡**

**[] 集群**

Meta假定producer、broker和consumer都是分布式的集群系统。

Producer可以是一个集群，多台机器上的producer可以往同一个topic发送消息。

Meta的服务器broker一般也是一个集群，多台broker组成一个集群提供一些topic服务，生产者按照一定的路由规则往集群里某台broker发送消息，消费者按照一定的路由规则拉取某台broker上的消息。

Consumer也可以组织成一个集群来消费同一个topic，发往这个topic的消息按照一定的路由规则发送到consumer集群里的某一台机器。Consumer集群每个consumer必须拥有相同的分组名称。

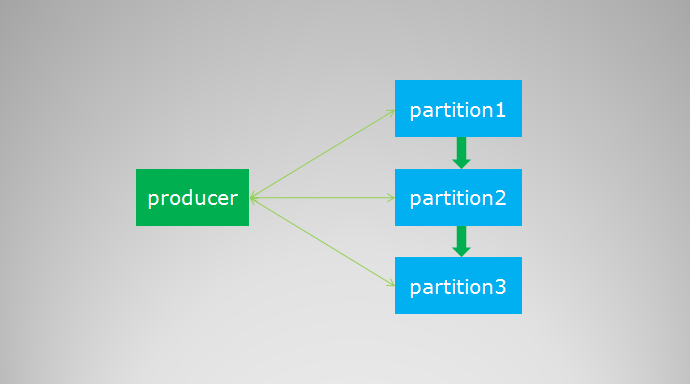
**[] 负载均衡**

负载均衡和failover分不开，我们将分别讨论下生产者和消费者的负载均衡策略。我们先假定broker是一个集群，这样每个topic必定有多个分区。

**[] 生产者的负载均衡和failover**

每个broker都可以配置一个topic可以有多少个分区，但是在生产者看来，一个topic在所有broker上的的所有分区组成一个分区列表来使用。

在创建producer的时候，客户端会从zookeeper上获取publish的topic对应的broker和分区列表，生产者在发送消息的时候必须选择一台broker上的一个分区来发送消息，默认的策略是一个轮询的路由规则，一张图来表示

[](http://baike.corp.taobao.com/index.php/File:Meta-producer-lb.png)

生产者在通过zk获取分区列表之后，会按照brokerId和partition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。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。

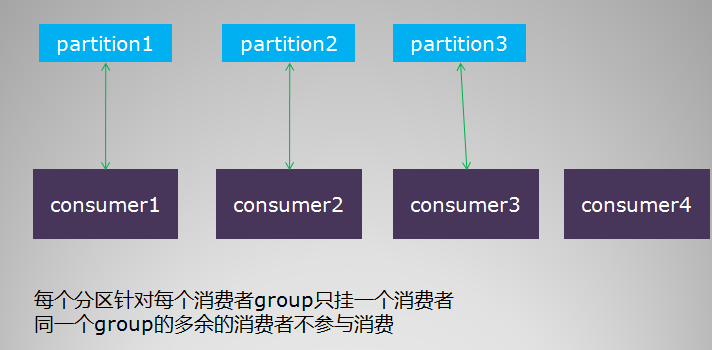
如果你想实现自己的负载均衡策略，可以实现上文提到过的PartitionSelector接口，并在创建producer的时候传入即可。

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。

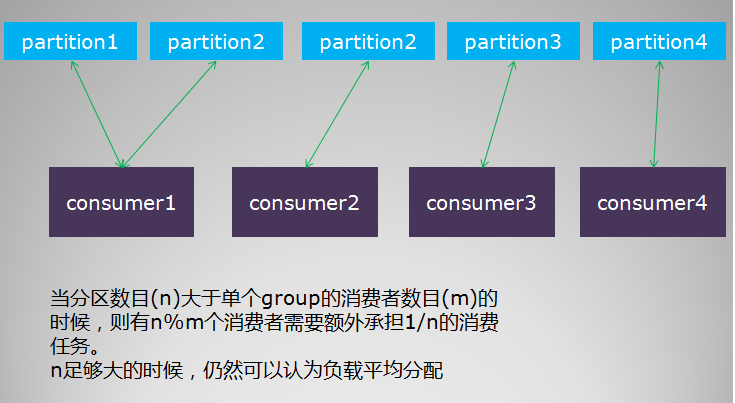
**[] 消费者的负载均衡**

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，如图

[](http://baike.corp.taobao.com/index.php/File:Meta-consumer-lb1.png)

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

[](http://baike.corp.taobao.com/index.php/File:Meta-consumer-lb2.png)

综上所述，单个分组内的消费者集群的负载均衡策略如下

1. 每个分区针对同一个group只挂载一个消费者
2. 如果同一个group的消费者数目大于分区数目，则多出来的消费者将不参与消费
3. 如果同一个group的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务

Meta的客户端会自动帮处理消费者的负载均衡，它会将消费者列表和分区列表分别排序，然后按照上述规则做合理的挂载。

从上述内容来看，合理地设置分区数目至关重要。如果分区数目太小，则有部分消费者可能闲置，如果分区数目太大，则对服务器的性能有影响。

在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch消费者列表），然后重新进行负载均衡，保证所有的分区都有消费者进行消费。

**[] 服务器部署**

### 前提

* **安装zookeeper**

首先你需要搭建自己的zookeeper集群，meta利用zookeeper做服务的注册和发现，以及默认情况下offset的存储。

* 安装java运行环境

### 第一步：下载安装包

从这里下载最新的metamorphosis服务器可运行包并在某个目录解压缩（或者checkout源码按上面的步骤打包出来）。解压出来的结构大概是这样

-你的目录

-taobao

-metamorphosis-server

-bin

-meta-server-start.sh

-meta-server-stop.sh

-conf

-server.properties

-logs

metaServer.log

-lib

其中bin目录包含启动脚本（目前只有linux下的启动脚本），conf下为配置文件，lib下为meta服务器的所有jar包,logs为日志所在目录。

### 第二步：配置server.ini

利用文本编辑器编辑conf/server.ini，这是meta服务器的配置文件，主要关注这几个配置项：

* brokerId 服务器ID，必须是集群内唯一
* numPartitions 默认每个topic的分区数目
* dataPath 数据文件的存放路径，默认在user.home/meta下
* zookeeper配置：

;以下为zk配置，可以为空，为空将从diamond获取（目前不提供），不为空则优先使用下列配置

zk.zkConnect=localhost:2181

;zk心跳超时，单位毫秒，默认30秒

zk.zkSessionTimeoutMs=30000

;zk连接超时时间，单位毫秒，默认30秒

zk.zkConnectionTimeoutMs=30000

;zk数据同步时间，单位毫秒，默认5秒

zk.zkSyncTimeMs=5000

zookeeper的地址也可以通过diamond管理，如果本地不明确配置zookeeper，则设置diamond的dataId和group即可自动从diamond获取zookeeper配置（目前不提供，要注释掉）：

;zk在diamond中配置存储的dataId

;diamondZKDataId=metamorphosis.zkConfig

;zk在diamond中配置存储的group

;diamondZKGroup=DEFAULT\_GROUP

配置topic列表

一份默认的文件如下：

;系统属性

[system]

;必须，服务器唯一标志

brokerId=0

;服务器hostname，可以为空，默认将取本机IP

hostName=

;默认每个topic的分区数目，默认为1

numPartitions=1

;服务器端口，必须

serverPort=8123

;数据文件路径，默认在user.home/meta下

dataPath=

;日志数据文件路径，默认跟dataPath一样

dataLogPath=

;最大允许的未flush消息数，超过此值将强制force到磁盘，默认1000

unflushThreshold=1000

;最大允许的未flush间隔时间，毫秒，默认10秒

unflushInterval=10000

;单个文件的最大大小,实际会超过此值，默认1G

maxSegmentSize=1073741824

;传输给客户端每次最大的缓冲区大小，默认1M

maxTransferSize=1048576

;处理get请求的线程数,默认cpus\*10

getProcessThreadCount=80

;处理put请求线程数，默认cpus\*10

putProcessThreadCount=80

;数据删除策略，默认超过7天即删除,这里的168是小时，10s表示10秒，10m表示10分钟，10h表示10小时，默认为小时

deletePolicy=delete,168

;删除策略的执行时间，cron表达式

deleteWhen=0 0 6,18 \* \* ?

;事务相关配置

;最大保存事务checkpoint数目，默认为3

maxCheckpoints=3

;事务checkpoint时间间隔，单位毫秒，默认1小时

checkpointInterval=3600000

;最大事务超时事件数，用于监控事务超时

maxTxTimeoutTimerCapacity=30000

;最大事务超时时间，单位秒

maxTxTimeoutInSeconds=60

;事务日志的刷盘设置，0表示让操作系统决定，1表示每次commit都刷盘，2表示每隔1秒刷盘一次

flushTxLogAtCommit=1

;zk配置

[zookeeper]

;是否注册到zk，默认为true

;zk.zkEnable=true

;以下为zk配置，可以为空，为空将从diamond获取，不为空则优先使用下列配置

;zk的服务器列表

zk.zkConnect=localhost:2181

;zk心跳超时，单位毫秒，默认30秒

zk.zkSessionTimeoutMs=30000

;zk连接超时时间，单位毫秒，默认30秒

zk.zkConnectionTimeoutMs=30000

;zk数据同步时间，单位毫秒，默认5秒

zk.zkSyncTimeMs=5000

;zk在diamond中配置存储的dataId

;diamondZKDataId=metamorphosis.zkConfig

;zk在diamond中配置存储的group

;diamondZKGroup=DEFAULT\_GROUP

;topic列表

[topic=test]

;是否启用统计

stat=true

;这个topic指定分区数目，如果没有设置，则使用系统设置

numPartitions=10

;topic的删除策略，默认使用系统策略

deletePolicy=

unflushInterval=

unflushThreshold=

;删除策略的执行时间，cron表达式

deleteWhen=0 0 6,18 \* \* ?

[topic=wuhua-test]

stat=true

numPartitions=10

unflushInterval=50

unflushThreshold=10

### 第三步：启动服务器

cd bin

sh meta-server-start.sh -f ../conf/server.properties

其中-f选项用于指定配置文件所在完整路径。 启动meta服务器后，你可以telnet到8123端口测试

telnet localhost 8123

stats

8123是meta服务器的默认端口，我们telnet上去并敲一个stats命令看看。敲quit命令可以退出telnet交互。

启动后，可以查看metaServer.log。

### 第四步：关闭服务器

关闭服务器通过meta-server-stop.sh脚本即可关闭

sh meta-server-stop.sh

## 其他

### 服务器的快速启动

Meta服务器在启动的时候会一个一个地校验所有文件，如果文件数目较多，那么这个启动过程会非常慢，如果想加快启动过程，可以使用fast\_boot选项，在meta-run-class.sh脚本里添加环境变量

META\_OPTS="-Xmx512m -server -Dcom.sun.management.jmxremote -Dlog4j.configuration=$base\_dir/bin/log4j.properties -Dmeta.fast\_boot=true"

将meta.fast\_boot设置为true即可跳过校验环节快速启动。

### 消息的删除策略

目前服务端支持两种删除策略：

* 定期删除，保存消息一定时间，超过指定时间就无条件删除。例如

deletePolicy=delete,72

的配置就是指使用删除策略，保存至少72个小时，超过即删除。

* 定期压缩归档，保存消息一定时间，超过指定时间就将消息压缩归档，例如

deletePolicy=archive,72

的配置就是指使用归档策略，保存至少72个小时，超过即归档。归档后的文件名前缀不变（也就是start offset)，后缀变为arc。归档策略还可以指定是否压缩：

deletePolicy=archive,72,true

第三个参数true指定归档策略使用压缩，meta使用zip压缩算法，压缩后的文件后缀即为zip，前缀不变。

### 不重启新增topic

新增一个topic或重新配置一些参数以后，运行tools里面的reloadconfig.sh，前提是已经部署了tools

Tools工程的打包部署：假设已经打包好了其他所有子工程，进入tools目录运行mvn -U -Dtest -DfailIfNoTests=false clean install package assembly:assembly，然后在tools的target目录里把跟服务端差不多结构的这个目录拷过去就可以了

或者直接使用jconsole的方式reload配置文件

### 如何启动异步复制slave

前提是使用metamorphosis-server-wrapper这个server。一个master可以挂上多个异步复制slave

1. 配置slave文件。编辑conf/async\_slave.properties

#异步复制的slave节点配置

#有问题联系无花 ([wq163@163.com](mailto:wq163@163.com))

#slave编号,大于等于0表示作为slave启动,同一个master下的slave编号应该设不同值.

slaveId=0

#作为slave启动时向master订阅消息的group,如果没配置则默认为meta-slave-group

#建议使用默认值

slaveGroup=meta-slave-group

#slave数据同步的最大延时,单位毫秒

slaveMaxDelayInMills=500

1. 确保conf/server.ini这个文件的brokerId跟master相同，这个表示作为哪个master的slave
2. 启动slave

sh meta-server-start.sh -f ../conf/server.properties -Fmetaslave=../conf/async\_slave.properties

**如何启用同步复制**

前提是使用metamorphosis-server-wrapper这个server。同步复制的master和slave是一对一的关系

1. 进入slave机器，确保conf/ gregor\_slave.properties文件的存在，不需要配置
2. 启动同步复制slave

sh meta-server-start.sh -f ../conf/server.properties -Fgregor =../conf/gregor\_slave.propertie

1. 进入master机器，配置conf/samsa\_master.properties

#slave节点ip，配置异步复制slave的地址

slave=localhost:8124

#应答回调线程池大小，默认3\*cpus

callbackThreadCount=10

#是否recover offset,只有在slave作为master启动的时候需要设置为true

recoverOffset=false

1. 启动同步复制master

sh meta-server-start.sh -f ../conf/server.properties -Fsamsa =../conf/samsa\_master.properties

**[] 最佳实践**

**[] 客户端最佳实践**

* 复用MessageSessionFactory，最好作为全局单例使用

**[] 生产者最佳实践**

* 尽量复用MessageProducer，可以单个MessageProducer发送多种topic，或者多个MessageProducer每个发送一种topic，前提是不要重复创建。
* 消息data的序列化方式建议不要使用特定于语言的序列化方式（如java序列化），可考虑自定义协议、json、protobufs、hessian都序列化协议，以便跨语言消费。
* 实现发送顺序所需要的ParitionSelector，我们会推荐使用业务id，如交易订单id来取模分区列表选择固定分区发送。
* 单条消息大小最好限制在百k以下。
* 如无顺序等特殊要求，不要实现自己的PartitionSelector,默认的轮询策略足够。

**[] 消费者的最佳实践**

* 尽量复用MessageConsumer，可以单个MessageConsumer订阅多种topic，或者多个MessageConsumer每个订阅一种topic，前提是不要重复创建。
* 单次拉取的数据不宜过大，对消息实时性要求较高的应用，应将单次拉取的数据缩小，但至少大于单条消息的大小。如对吞吐量要求较高，可将该值设大。
* 如消费过程非常轻量级(比如只是打印)，可不设置MessageListener线程池，减少资源耗费。
* 如消息发送量巨大，消费能力不高，可适当提高拉取消息线程数fetchRunnerCount和MessageListener的线程池大小。
* 尽量在消息消费过程中捕捉所有异常，减少消息在本地的堆积和恢复，前提是不要遗漏消息。如确实无法处理，请主动抛出异常以便重试。

**[] 原理和实现**

**[] 设计原理**

整体的设计思路与kafka是完全一致的，kafka的设计文档可以作为meta的参考文档

http://sna-projects.com/kafka/design.php

实现上大体介绍下。

**[] 网络通讯和协议**

采用notify-remoting做为通讯模块，实现meta的协议。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。

协议命令包括

|  |  |  |  |
| --- | --- | --- | --- |
| **命令** | **参数** | **说明** | **示例** |
| put | topic partition value-length flag [transactionKey] | 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。 | put meta-test 0 5 0 1\r\nhello |
| get | topic group partition offset maxSize | 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。 | get meta-test example 0 1024 512 1\r\n |
| data | total-length | get请求返回的应答，total-length返回的数据长度。 | data 5 1\r\nhello |
| result | code length | 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 | result 200 0 1\r\n |
| offset | topic group partition offset | 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 | offset meta-test example 0 1024 1\r\n |
| stats | item(可选) | 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。 | stats 1\r\n |

整个协议采用文本方式，非常适合其他语言实现客户端。

**[] 消息的存储结构**

消息在服务器的是按照顺序连续append在一起的，具体的单个消息的存储结构如下：

* message length(4 bytes),包括消息属性和payload data
* checksum(4 bytes)
* message id(8 bytes)
* message flag(4 bytes)
* attribute length(4 bytes) + attribute，可选
* payload

其中checksum采用CRC32算法计算，计算的内容包括消息属性长度+消息属性+data，消息属性如果不存在则不包括在内。消费者在接收到消息后会检查checksum是否正确。

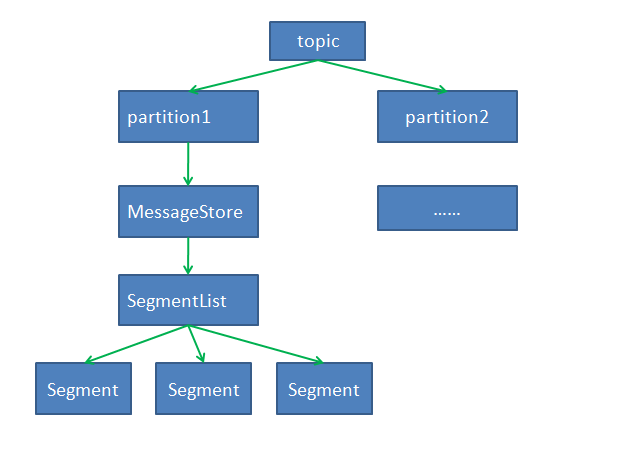
同一个topic下有不同分区，每个分区下面会划分为多个文件，只有一个当前文件在写，其他文件只读。当写满一个文件（写满的意思是达到设定值）则切换文件，新建一个当前文件用来写，老的当前文件切换为只读。文件的命名以起始偏移量来命名。看一个例子，假设meta-test这个topic下的0-0分区可能有以下这些文件：

* 00000000000000000000000000000000.meta
* 00000000000000000000000000001024.meta
* 00000000000000000000000000002048.meta
* ……

其中00000000000000000000000000000000.meta表示最开始的文件，起始偏移量为0。第二个文件00000000000000000000000000001024.meta的起始偏移量为1024，同时表示它的前一个文件的大小为1024-0=1024。同样，第三个文件00000000000000000000000000002048.meta的起始偏移量为2048，表明00000000000000000000000000001024.meta的大小为2048-1024=1024。

以起始偏移量命名并排序这些文件，那么当消费者要抓取某个起始偏移量开始位置的数据变的相当简单，只要根据传上来的offset二分查找文件列表，定位到具体文件，然后将绝对offset减去文件的起始节点转化为相对offset，即可开始传输数据。例如，同样以上面的例子为例，假设消费者想抓取从1536开始的数据1M，则根据1536二分查找，定位到00000000000000000000000000001024.meta这个文件（1536在1024和2048之间），1536-1024=512，也就是实际传输的起始偏移量是在00000000000000000000000000001024.meta文件的512位置。因为1024.meta的大小才1K，比1M小多了，实际传输的数据只有2048-1536=512字节。

这些文件在meta里命名为Segment，每个Segment对应一个FileMessageSet。文件组织成SegmentList，整体成为一个MessageStore，一个topic下的一个分区对应一个MessageStore。一张图如下

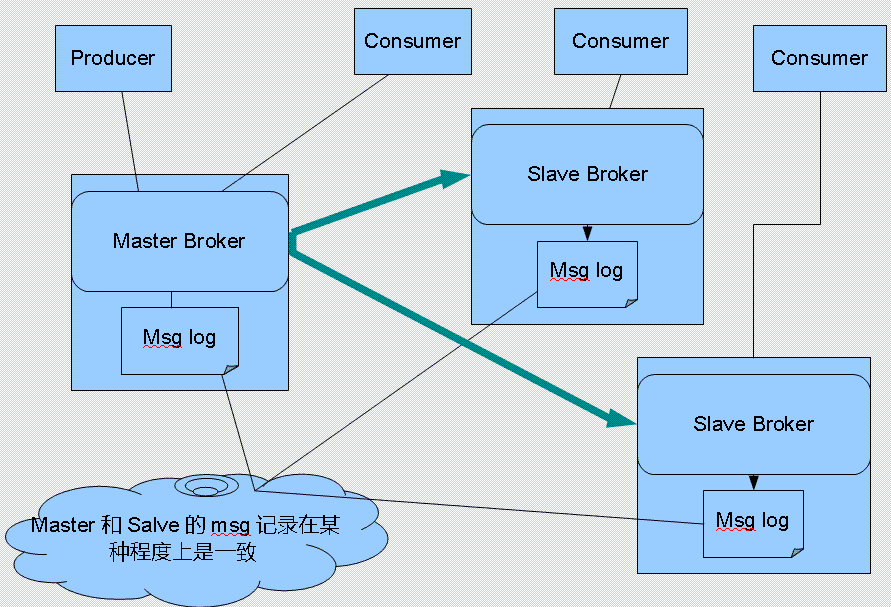
[](http://baike.corp.taobao.com/index.php/File:Meta-store.png)

**[] 消息的恢复和重试**

当消费者无法正常消费某条消息的适合，meta客户端会将消息存储在消费者本地磁盘，并在后台线程重试。存储是采用notify-store4j，notify-store4j的存储是按照插入顺序存储的的，因此可以保证按照顺序做消息的recover。notify-store4j的具体实现请看notify源码。

**[] HA异步复制方案**

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。如下图。

[](http://baike.corp.taobao.com/index.php/File:HA.GIF)

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：

* 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）
* 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）。

**[] FAQ**

**[] 采用pull模型，消息的实时性有保证吗？**

Metamorphosis在消费端采用pull的模型，consumer主动去broker拉取数据，而不是类似notify那样由broker主动push数据给消费者。可能很多人担心采用pull模型后，会不会消息的实时性降低了，从发送到消费的整个时间周期拉长了。

实际上，meta中消息的实时性受很多因素影响，不能简单地说实时性一定会降低，主要影响因素如下

1. broker上配置的批量force消息的阈值，默认是1000条force一次。这个值越大，则实时性越低。
2. 消费者每次抓取的数据大小，这个值越大，则实时性越低，但是吞吐量越高。
3. Topic的分区数目对实时性也有较大影响，分区数目越多，则磁盘压力越大，导致消息投递的实时性降低。
4. 消费者重试抓取的时间间隔，越长则延迟越严重。
5. 消费者抓取数据的线程数

可见，消息实时性在meta里受到很多因素的影响，meta可以让用户自己决定如何在响应性和吞吐量之间做平衡，通过配置来合理设置参数，达到应用方需要的实时性，实际测试，消息消费的延迟可以在几毫秒到几秒之间。

**[] Metamorphosis怎么做到环境隔离？**

有的朋友可能想搭建自己的meta开发和测试环境，不想使用日常的。这也完全可以，meta环境隔离的主要问题是zookeeper环境的隔离，你可以搭建一台自己的zookeeper，然后配置下meta的broker使用你自己的zookeeper，就可以完全跟日常环境隔离开。这是在服务端server.properties中配置

#以下为zk配置，可以为空，为空将从diamond获取，不为空则优先使用下列配置

#zk的服务器列表

zk.zkConnect=localhost:2181

#zk心跳超时，单位毫秒，默认30秒

zk.zkSessionTimeoutMs=30000

#zk连接超时时间，单位毫秒，默认30秒

zk.zkConnectionTimeoutMs=30000

#zk数据同步时间，单位毫秒，默认5秒

zk.zkSyncTimeMs=5000

如果你不想写死zk配置在服务端，也可以在diamond里配置上述的这些参数，然后在broker的server.properties配置下使用的diamond的dataId和group，meta会使用diamond client获取zk配置

#zk在diamond中配置存储的dataId

diamondZKDataId=metamorphosis.zkConfig

#zk在diamond中配置存储的group

diamondZKGroup=DEFAULT\_GROUP