ACID--CAP--BASE理论

BASE：basically available基本可用, soft state软状态, eventually consistent最终一致性。

最终一致性5个变种：causal consistency因果关系

Read your writes 读己之所写

Session consistency会话一致性

Monotonic read consistency单调读一致性

Monotonic write consistency单调写一致性

最终一致性在关系型数据库中体现：主从复制，同步时是强一致性，，异步时存在延迟，即最终一致性。

二阶段提交协议

优点：原理简单，容易实现

缺点：同步阻塞，单点问题，脑裂（数据不一致），太过保守的容错机制

三阶段提交协议：CanCommit, PreCommit, Do Commit

Paxos算法

 数据的一致性协议有：

        1.  两阶段提交协议  2PC

        2.  三阶段提交协议  3PC

        3.  RWN协议

        4.  raft协议

        5.  Paxos协议

分布式系统的两个重要协议：

1. Paxos选举协议：Paxos 协议用于多个节点之间达成一致，往往用于实现总控节点选举。关于Paxos可以参考我的另外文章[从分布式数据复制一致性问题到paxos算法的理解（第一部分）](http://kaimingwan.com/post/fen-bu-shi/cong-fen-bu-shi-shu-ju-fu-zhi-zhi-xing-wen-ti-dao-paxossuan-fa-de-li-jie-di-bu-fen)

<http://kaimingwan.com/post/fen-bu-shi/cong-fen-bu-shi-shu-ju-fu-zhi-zhi-xing-wen-ti-dao-paxossuan-fa-de-li-jie-di-bu-fen>

和

[从分布式数据复制一致性问题到paxos算法的理解（第二部分）](http://kaimingwan.com/post/fen-bu-shi/cong-fen-bu-shi-shu-ju-fu-zhi-zhi-xing-wen-ti-dao-paxossuan-fa-de-li-jie-di-er-bu-fen)

<http://kaimingwan.com/post/fen-bu-shi/cong-fen-bu-shi-shu-ju-fu-zhi-zhi-xing-wen-ti-dao-paxossuan-fa-de-li-jie-di-er-bu-fen>

2. 两阶段提交协议(2PC): 保证跨多个节点操作的原子性，这些操作要么全部成功，要么全部失败。

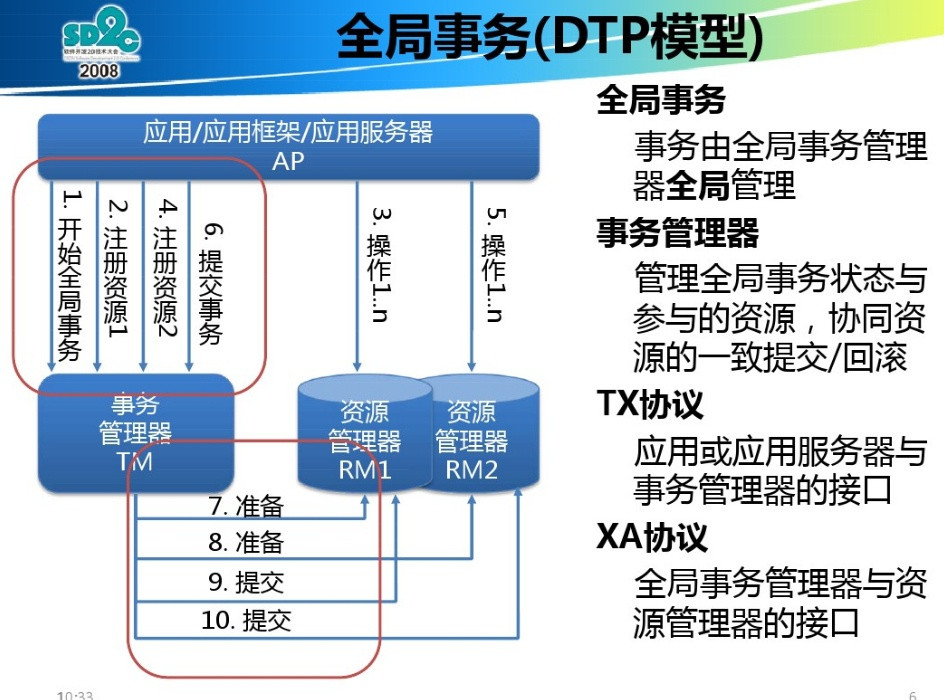
# 5. DTP事务模型

X/Open Distributed Transaction Processing (DTP) 事务模型实现了XA接口。XA事务指的是实现了XA接口符合DTP事务模型的事务。  
DTP事务模型是X/Open 这个组织定义的一套分布式事务的标准，也就是了定义了规范和API接口，由这个厂商进行具体的实现。  
DTP事务模型主要用于分布式事务

XA接口规范定义了以下几个组件：

1. 应用程序(AP):也就是应用程序，可以理解为使用DTP的程序 2.资源管理器(RM):资源管理器，这里可以理解为一个DBMS系统，或者消息服务器管理系统，应用程序通过资源管理器对资源进行控制。资源必须实现XA定义的接口 3.事务管理器(TM):事务管理器，负责协调和管理事务，提供给AP应用程序编程接口以及管理资源管理器

AP、RM、TM三者直接互相的协作关系用下图表示：



可以看到，其中TM 和 RM 通过XA接口进行双向通信。AP和TM、RM的通信接口规范没有约定，可以自己实现。

DTP定义的事务概念：

1. 事务：一个事务是一个完整的工作单元，由多个独立的计算任务组成，这多个任务在逻辑上是原子的。
2. 全局事务：对于一次性操作多个资源管理器的事务，就是全局事务
3. 分支事务：在全局事务中，某一个资源管理器有自己独立的任务，这些任务的集合作为这个资源管理器的分支任务

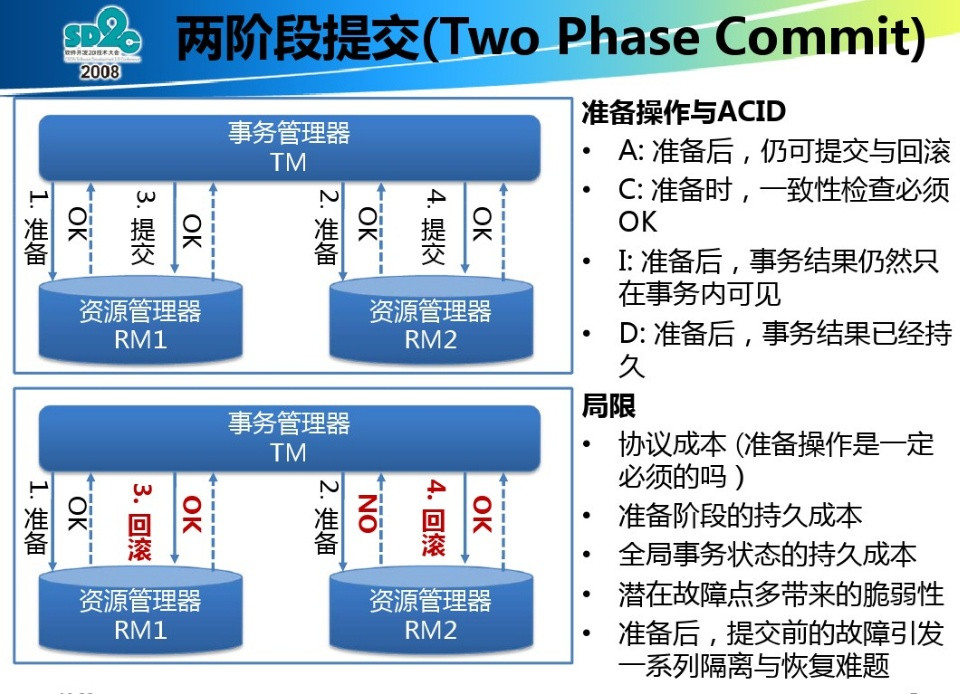
控制线程：用来表示一个工作线程，主要是关联AP,TM,RM三者的一个线程，也就是事务上下文环境。简单的说，就是需要标识一个全局事务以及分支事务的关系。

很多数据库都实现了DTP事务模型，用于支持分布式事务。

## 5.1 两阶段提交(2PC)

两阶段提交是DTP事务模型的一种实现方式。一个TM控制多个RM，协调资源。

两阶段主要指的是：  
第一阶段（准备阶段）：事务管理器通知资源管理器准备分支事务，资源管理器告之事务管理器准备结果  
第二阶段(提交阶段)：事务管理器通知资源管理器提交分支事务，资源管理器告之事务管理器结果

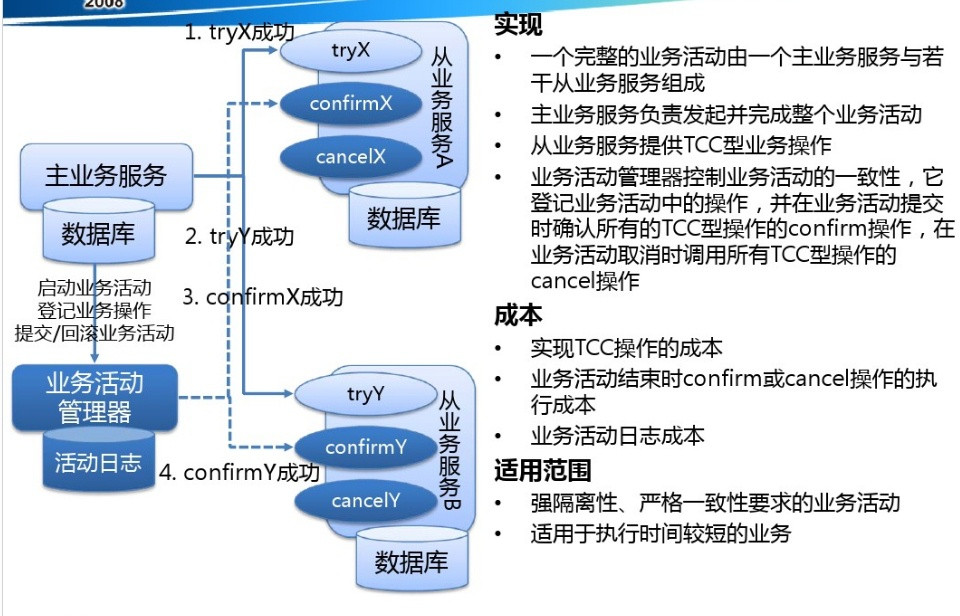


## 5.2 TCC

TCC是一种基于事务补偿（属于事后控制的策略，进行回滚）的模式，注意区别2PC

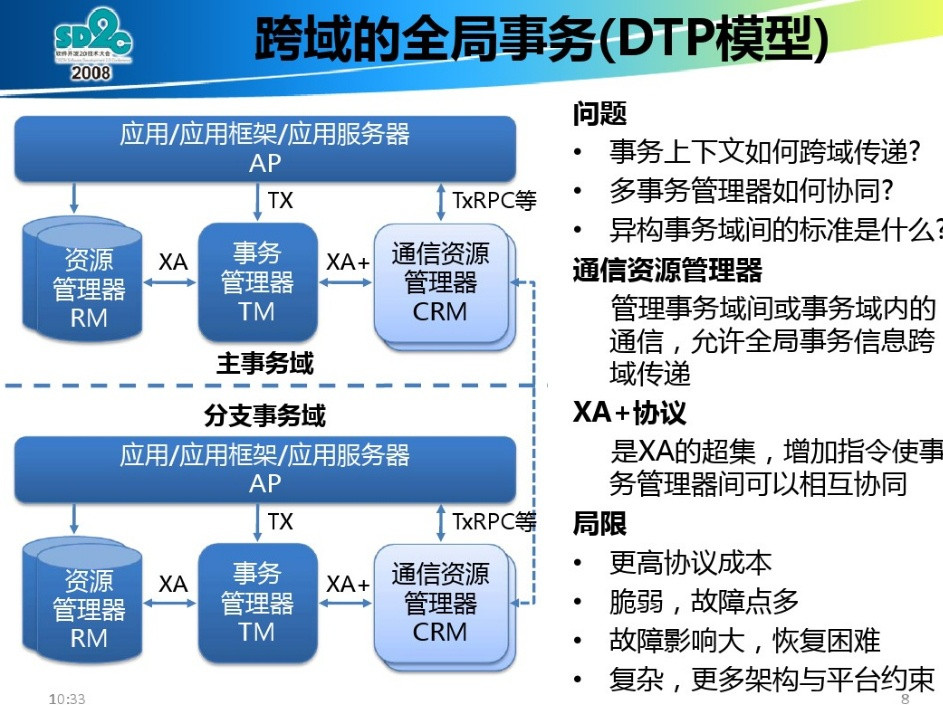


TCC的执行过程



# 6 跨域的DTP模型

需要RPC调用，有一些额外的问题。需要引入通信资源管理器。具体看下图：



集群伸缩性：分为应用服务器集群伸缩性和数据服务器集群伸缩性。

而数据服务器集群伸缩性又包含缓存数据服务器集群和存储数据服务器集群。

应用服务器集群的伸缩性设计：

事实上，大型网站总是部分使用DNS域名解析，利用域名解析作为第一级负载均衡手段，即域名解析得到的一组服务器并不是实际提供Web服务的物理服务器，而是同样提供负载均衡服务的内部服务器，这组内部负载均衡服务器再进行负载均衡，将请求分发到真实的web服务器上。

**负载均衡技术：**

1. HTTP重定向负载均衡：现实中并不多见，利用了一个专门的重定向服务器

优点：简单；

缺点：两次访问，性能差，重定向服务器自身的处理能力存在瓶颈。

1. DNS域名解析负载均衡：利用了DNS服务器

优点：工作移交给DNS，省了网站方管理重定向服务器的麻烦；支持基于地理位置的请求分发，DNS服务器会返回一个距离用户最近的IP地址给用户，加快用户访问速度，改善性能；

缺点：网站方不能做更强管理的控制，DNS多级解析给下线服务器更改带来延迟。并且DNS负载均衡的控制权在域名服务商那里，网站无法做更多改善和管理。

1. 反向代理负载均衡：

优点：与反向代理服务器功能集成在一起，部署简单

缺点：反向代理服务器是所有请求和响应的中转站，存在性能瓶颈

1. IP负载均衡：在网络层通过修改请求目标地址进行负载均衡，请求到专门的负载均衡服务器，服务器在操作系统内核进程获取网络数据包，根据负载均衡算法计算得到一台真实的web服务器ip地址，然后修改数据包目的ip地址，不需要通过用户进程处理。得到响应后，服务器再将数据包 源地址改回原来的ip地址。这个过程需要进行地址转换（真实处理的web服务器要发送数据给负载均衡服务器）

优点：在内核中完成分发，性能好

缺点：请求和响应都经过它，存在瓶颈，例如网卡宽带不好。

1. 数据链路层负载均衡：又叫做直接路由方式（DR）；指在通信协议的数据链路层修改MAC地址进行负载均衡。三角传输模式，负载均衡服务器只修改MAC地址，不修改IP地址，因此真实的WEB服务器处理完后可以直接发送响应到用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈。

优点：

缺点：

**使用三角传输模式的链路层负载均衡是目前大型网站使用最广的一种负载均衡手段。在Linux平台上最好的链路层负载均衡开源产品是LVS（Linux Virtual Server）。**

**负载均衡算法：**

1. 轮询 Round Robin, RR
2. 加权轮询 Weighted Round Robin, WRR：根据服务器硬件性能加权
3. 随机 Random
4. 最少连接 Least Connections：记录每个服务器连接数（请求数），新请求连到最少连接的机器上。
5. 源地址散列 Source Hashing：根据请求来源IP地址进行Hash计算，得到应用服务器，这样来自同一个IP地址的请求总在用同一个服务器上处理，请求上下文信息可以存储在这台机器上，在一个会话周期内重复使用，从而实现会话粘滞。

## 分布式锁的实现

针对分布式锁的实现，目前比较常用的有以下几种方案：

1. 基于数据库实现分布式锁 ：总结一下使用数据库来实现分布式锁的方式，这两种方式都是依赖数据库的一张表，一种是通过表中的记录的存在情况确定当前是否有锁存在，另外一种是通过数据库的排他锁来实现分布式锁。

****数据库实现分布式锁的优点****

直接借助数据库，容易理解。

****数据库实现分布式锁的缺点****

会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。

操作数据库需要一定的开销，性能问题需要考虑。

使用数据库的行级锁并不一定靠谱，尤其是当我们的锁表并不大的时候。

1. 基于缓存（redis，memcached，tair）实现分布式锁 ：关于缓存有两种实现吧：

基于SetNX实现：setNX是Redis提供的一个原子操作，如果指定key存在，那么setNX失败，如果不存在会进行Set操作并返回成功。我们可以利用这个来实现一个分布式的锁，主要思路就是，set成功表示获取锁，set失败表示获取失败，失败后需要重试。

3. 基于Zookeeper实现分布式锁

## 中间件定义

维基百科的定义：中间件为软件应用提供了操作系统所提供的服务之外的服务，可以把中间件描述为“软件胶水”。中间件不是操作系统的一部分，不是数据库管理系统，也不是软件应用的一部分，而是能够让软件开发者方便地处理通信、输入和输出，能够专注在他们自己应用的部分。

通俗描述：中间件不是最上层的应用，也不是最底层的支撑系统，是处于中间位置的组件。中间件起到的是桥梁作用，是应用与应用之间的桥梁，也是应用与服务之间的桥梁。特定中间件是解决特定场景问题的组件，它能够让软件开发人员专注于自己应用的开发。

## 消息中间件

1. 如何实现系统解耦：

消息提供者只管发送消息，不用管被谁消费，或者是否消费成功。实现了应用之间的解耦。

1. 如何实现业务操作和消息发送的一致性

　XA事物规范：

1. 引入分布式事务会带来一些开销并增加复杂
2. 对于业务操作有限制，要求业务操作的资源必须支持XA协议，才能够与发送消息一起来做分布式事务。

解决一致性的方案：

1. 发送消息给消息中间件

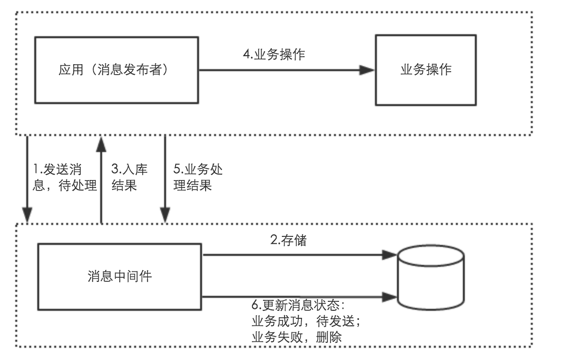
2. 消息中间件入库消息

3. 消息中间件返回结果

4. 业务操作

5. 发送业务操作结果给消息中间件

6. 更改存储中消息状态



异常分析：

1. 业务应用发送消息给消息中间件。如果失败，业务操作没有做，消息也没有存储在消息中间件，业务操作和消息的状态是一致的，没有问题。
2. 消息中间件把消息入库。如果失败，可能情况：消息中间件失效，应用收不到返回结果；插入消息失败，收到返回失败结果给应用。
3. 业务应用收到消息中间件结果异常。如果有业务应用正常，业务应用不知道在消息中间件的处理结果，按照失败来处理，如果入库成功，就会造成不一致；  
   如果业务应用自身有问题，消息入库成功，也会造成不一致；如果入库失败，则还是一致的。
4. 业务应用进行业务操作。
5. 业务应用发送业务操作结果给消息中间件。如果出现问题，消息中间件不知道如何操作存储的消息，可能会造成不一致。
6. 消息中间件更新消息状态。出现问题同上。
7. 如何解决消息中间件与使用者的强依赖问题

三种思路：

1. 提供消息中间件系统的可靠性，但是没办法保证百分之百可靠。

2. 对于消息中间件系统中影响业务操作进行的部分，使其可靠性与业务自身的可靠性相同。

其实就是保证如果业务操作能够成功，就需要消息能够入库成功。可以把消息中间件所需要的消息表与业务数据表放到同一个业务数据库中，这样业务应用就可以把业务操作和写入消息作为一个本地事务来完成，然后再让消息中间件轮询消息表，看是否有消息可以发送，这样就解决了一致性问题。出现的问题就是需要业务数据库承载消息数据，还要让消息中间件去访问业务数据库，还需要业务操作的对象是一个数据库，或者说支持事务的存储，并且这个存储必须支持消息中间件的需求。又或者优化一下，不要让消息中间件轮询消息表，而是让业务应用轮询消息表，消息中间件不直接跟业务数据库打交道。这两种方式虽然解决了大部分问题，但都要求业务操作必须是支持事务的数据库操作，具有一定的限制。再改进优化下，不把消息表放进业务数据库中，而是放在本地磁盘中，等待消息中间件回复后，再把消息发送到消息中间件。

3. 可以提供弱依赖的支持，能够较好的保证一致性。

**主要是从业务数据上进行消息补发，这才是最彻底的容灾手段，因为这样才能保证只要业务数据在，就一定可以有办法恢复消息**

1. JMS消息模型

1. JMS Queue模型（点对点）:如果queue里面的消息被一个应用处理了，那么其他应用也就不会再接收到这条消息了。消息从发送端发送出来时不能确定最终会被哪个应用消费，但是可以明确的是只有一个应用会去消费这条消息。

2. JMS Topic模型（发布订阅）：所有订阅过某种topic消息的应用都可以接受到提供者发出来的消息并消费。

具体来说，我们可以把集群和集群之间对消息的消费当做Topic模型来处理，而集群内部的各个具体应用实例对消息的消费当做queue模型来处理。

1. 消息订阅的方式

1. 持久订阅：消费者应用结束了，订阅关系不存在，消息不会保留

2. 非持久订阅：即使消费者应用结束了，消息会保留，应用下次启动时会再收到消息，除非显示的取消订阅关系。

1. 保证消息可靠性的做法

从消息开始从发送端应用到接收端应用，有三个阶段需要保证可靠，分别是：消息发送者把消息发送到消息中间件，消息中间件把消息存入消息存储，消息中间件把消息投递给消息接收者。

1. 消息发送端可靠性保证：发送之后返回结果处理
2. 消息存储的可靠性保证：实现基于文件的消息存储；实现基于数据库的消息存储；实现基于双机内存的消息存储（双机内存存储，提高吞吐量，如果一个机器出现故障，则停止另一台机器的数据写操作，并把当前数据落磁盘，这适合消息到了消息中间件后大部分消息能够被及时消费的情况，可以很好的提升性能）。例如把消息存到数据库表中，每条消息一条单行数据记录，表字段可以是消息的header信息（消息Id，创建时间，投递次数，优先级，自定义的键值对属性），消息的body（消息具体内容），消息的投递对象（集群Id:ClusterId）。消息进入数据库时，一般还会生成投递表，消息投递有结果时，去更新投递表，投递成功，删除投递表记录，投递失败，则更新投递次数以及下次投递时间，一般投递时间间隔越来越长。
3. 消息投递的可靠性保证：中间件需要显示的收到接收者确认消息处理完毕的信号才能删除，一定要从应用层的响应入手。
4. 消息系统的扩容处理：分为消息中间件自身的扩容和存储的扩容。如何在同一个存储中区分存储的消息是来自哪个消息中间件应用的，解决方案就是给每条消息增加一个server表示的字段。
5. JMS 中消息接收者对收到的消息进行确认的方式：

1. AUTO\_ACKNOWLEDGE 自动确认，接收到消息就确认，可能消息还未处理，所以不可靠。

2. CLIENT\_ACKONWLEDGE 接收端自己确认，主动调用Message接口的acknowledge()方法进行消息成功的确认。

3. DUPS\_OK\_ACKNOWLEDGE 消息接收方消息处理函数执行结束后自动确认，不需要用户主动调acknowledge（）方法。

以上三种确认方式是通过JMS的Connection在创建Queue或者Topic时设置的。

消息接收者对于消息的接收会出现at least once（至少一次） 和 at most once（至多一次）两种情况

消息中间件的两个最基础特点: 应用之间的解耦 和 操作的异步.

## 5 消息重复投递

分布式事务：实现复杂

接收者消息处理幂等性：降低消息中间件复杂，增加接受者门槛

## push/pull 模型

对于消费者而言有两种方式从消息中间件获取消息：

①Push方式：由消息中间件主动地将消息推送给消费者，采用Push方式，可以尽可能快地将消息发送给消费者；②Pull方式：由消费者主动向消息中间件拉取消息，会增加消息的延迟，即消息到达消费者的时间有点长

但是，Push方式会有一个坏处：如果消费者的处理消息的能力很弱(一条消息需要很长的时间处理)，而消息中间件不断地向消费者Push消息，消费者的缓冲区可能会溢出。

|  |  |  |
| --- | --- | --- |
|  | push | pull |
| 数据传输状态 | 保存在服务器端 | 保存在消费端 |
| 传输失败，重试 | 服务器需要维护每次传输状态，遇到失败情况重试 | 不需要 |
| 数据传输实时性 | 非常实时 | 默认的短轮询方式的实时性依赖于pull间隔时间，间隔越大实时性越低 |
| 流控控制 | 服务器需要依据订阅者的消费能力做流控 | 消费者可以根据自身消费能力决定是否去pull消息 |
|  |  |  |

### 消息中间件的有序性保证

有一个很重要的前提：要保持多个消息之间的时间顺序，首先它们要有一个全局的时间顺序。因此，每个消息在被创建时，都将被赋予一个全局唯一的、单调递增的、连续的序列号（Serial Number，SN）。可以通过一个全局计数器来实现这一点。通过比较两个消息的SN，确定其先后顺序。

方案一：通过某种算法，将需要保持先后顺序的消息放到同一个消息队列中。此方案有可能导致一个队列不堪重负，另一个队列为空，负载不均衡。

方案二：在将M1加入消息队列A时，同时对其他每一个队列加入一个特殊消息（Block Message, BM）。当某个消息队列处理BM时，检查M1是否已经被处理。如果M1已经被处理，则处理后面的消息，否则等待直到M1被处理。

严格地说，方案二不仅仅保证了M1和M2的时间顺序，更多地是保证了M1和所有M1之后的消息的时间顺序，因此它更适用于保证某个影响范围较大的操作的时间顺序。例如，M1对应创建一个用户的操作，而所有与此用户有关的操作都必须在创建此用户后才能执行。

方案二也有问题，那就是，在将M1加入队列时，必须明确知道其后会有依赖于M1的操作进行。对本例而言，评论微博以后，不一定会有删除微博的操作，而为了M1而block所有其它的消息队列，无疑会造成很大的性能浪费。那么又有

方案三：每一个消息队列都记录自己处理的最后一个消息的SN，称为Last SN，LSN。假设消息M1的SN=100，消息M2的SN=110。当处理到M2时，检查所有其它消息队列的LSN是否大于等于100：

如果所有消息队列的LSN都大于100，那么M1肯定已经被处理了。

如果有一个或多个消息队列的LSN小于100，那么，M1可能还没有被处理，但一定已经被放到了某个队列中（因为M2已经在当前队列中了）。在这种情况下，block所有LSN大于100的队列，等待所有LSN小于100队列继续运行，直到这些队列要么LSN大于等于100，要么队列为空。

有一个优化条件是，任何时候，如果某个消息队列的LSN等于100，那么M1肯定已经被处理了，可以直接结束等待。

当然，这肯定会导致性能上的额外开销。事实上，方案一和方案二的结合已经足够应付现实中的大多数情况了

## JMS规范

消息的消费   
在JMS中，消息的产生和消息是异步的。对于消费来说，JMS的消息者可以通过两种方式来消费消息。   
○ 同步   
订阅者或接收者调用receive方法来接收消息，receive方法在能够接收到消息之前（或超时之前）将一直阻塞   
○ 异步   
订阅者或接收者可以注册为一个消息监听器。当消息到达之后，系统自动调用监听器的onMessage方法。

### JMS编程模型

(1) ConnectionFactory

创建Connection对象的工厂，针对两种不同的jms消息模型，分别有QueueConnectionFactory和TopicConnectionFactory两种。可以通过JNDI来查找ConnectionFactory对象。

(2) Destination

Destination的意思是消息生产者的消息发送目标或者说消息消费者的消息来源。对于消息生产者来说，它的Destination是某个队列（Queue）或某个主题（Topic）;对于消息消费者来说，它的Destination也是某个队列或主题（即消息来源）。

所以，Destination实际上就是两种类型的对象：Queue、Topic可以通过JNDI来查找Destination。

(3) Connection

Connection表示在客户端和JMS系统之间建立的链接（对TCP/IP socket的包装）。Connection可以产生一个或多个Session。跟ConnectionFactory一样，Connection也有两种类型：QueueConnection和TopicConnection。

(4) Session

Session是我们操作消息的接口。可以通过session创建生产者、消费者、消息等。Session提供了事务的功能。当我们需要使用session发送/接收多个消息时，可以将这些发送/接收动作放到一个事务中。同样，也分QueueSession和TopicSession。

(5) 消息的生产者

消息生产者由Session创建，并用于将消息发送到Destination。同样，消息生产者分两种类型：QueueSender和TopicPublisher。可以调用消息生产者的方法（send或publish方法）发送消息。

(6) 消息消费者

消息消费者由Session创建，用于接收被发送到Destination的消息。两种类型：QueueReceiver和TopicSubscriber。可分别通过session的createReceiver(Queue)或createSubscriber(Topic)来创建。当然，也可以session的creatDurableSubscriber方法来创建持久化的订阅者。

(7) MessageListener

消息监听器。如果注册了消息监听器，一旦消息到达，将自动调用监听器的onMessage方法。EJB中的MDB（Message-Driven Bean）就是一种MessageListener。