**Zookeeper**

**什么是Zookeeper**

zookeeper是一个开源的分布式协调服务，由雅虎公司创建，是google chubby的开源实现。zookeeper的设计目标是将哪些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集（由若干条指令组成的，完成一定功能的一个过程），并且以一些列简单一用的接口提供给用户使用

**应用场景**

1. 数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到ZK节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。

1. 负载均衡

这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。

1. 命名服务(Naming Service)（注册中心）

命名服务也是分布式系统中比较常见的一类场景。在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等——这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用ZK提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

服务提供者在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。

服务消费者启动的时候，订阅/dubbo/${serviceName}/providers目录下的提供者URL地址， 并向/dubbo/${serviceName} /consumers目录下写入自己的URL地址。

注意，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。

1. 分布式通知/协调

ZooKeeper中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能够收到通知，并作出相应处理

1. 集群管理与Master选举
2. 分布式锁
3. 分布式队列

**配置**

http://apache.fayea.com/zookeeper/下载完成，通过tar -zxvf解压

单机安装

一般情况下，在开发测试环境，没有这么多资源的情况下，而且也不需要特别好的稳定性的前提下，我们可以使用单机部署；初 次 使 用zookeeper， 需要将conf目 录 下 的zoo\_sample.cfg文件copy一份重命名为zoo.cfg修改dataDir目录，dataDir表示日志文件存放的路径

集群环境安装

zookeeper集群中，各个节点总共有三种角色，分别是：leader，follower，observer

模拟3台机器来搭建zookeeper集群。分别复制安装包到三台机器上并解压，同时copy一份zoo.cfg。

修改配置文件：

server.A=B：C：D

A 是一个数字，表示这个是第几号服务器

B 是这个服务器的ip地址

C 表示的是这个服务器与集群中的Leader 服务器交换信息的端口

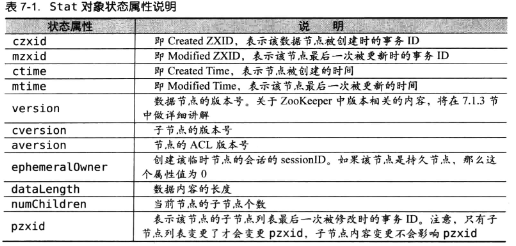
D 表示的是万一集群中的Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

新建datadir目录，设置myid

在每台zookeeper机器上，我们都需要在数据目录(dataDir)下创建一个myid文件，该文件只有一行内容，对应每台机器的Server ID数字；比如server.1的myid文件内容就是1。【必须确保每个服务器的myid文件中的数字不同，并且和自己所在机器的zoo.cfg中server.id的id值一致，id的范围是1~255】

**节点特性**

1. 同级节点的唯一性
2. 临时节点（会话结束会自动删除）和持久化节点
3. 有序节点
4. 临时节点下不能存在节点



**指令**

1. 创建

create [-s] [-e] path acl

s: 顺序节点

e: 临时节点

acl: 权限控制

create /zk-book 123

1. 读取

ls：ls path [watch] 例子：ls / 只能查看当前节点的下一层节点

get: get path [watch] 例子：get /zk-book 可以得到节点的value值（123）

1. 更新

set path data [version](节点有版本概念dataVersion)

set /zk-book 456

1. 删除

delete path [version]

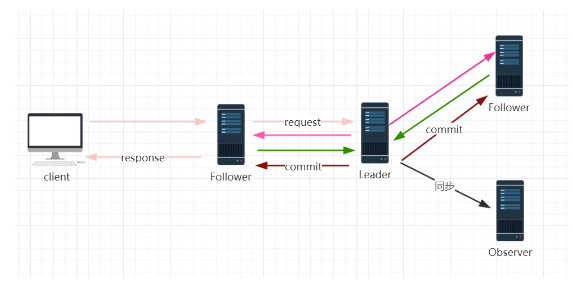
delete /zk-book (如果删除的节点下面有子节点，不能删除)

**常用命令**

1. 启动ZK服务: bin/zkServer.sh start
2. 查看ZK服务状态: bin/zkServer.sh status
3. 停止ZK服务: bin/zkServer.sh stop
4. 重启ZK服务: bin/zkServer.sh restart
5. 连接服务器：zkCli.sh -timeout 0 -r -server ip:port
6. 登陆客户端：sh zkCli.sh

**zookeeper的集群**

在zookeeper中，客户端会随机连接到zookeeper集群中的一个节点，如果是读请求，就直接从当前节点中读取数据，如果是写请求，那么请求会被转发给leader提交事务，然后leader会广播事务，只要有超过半数节点写入成功，那么写请求就会被提交（类2PC事务）



所有事务请求必须由一个全局唯一的服务器来协调处理，这个服务器就是Leader服务器，其他的服务器就是follower。leader服务器把客户端的失去请求转化成一个事务Proposal（提议），并把这个Proposal分发给集群中的所有Follower服务器。之后Leader服务器需要等待所有Follower服务器的反馈，一旦超过半数的Follower服务器进行了正确的反馈，那么Leader就会再次向所有的Follower服务器发送Commit消息，要求各个follower节点对前面的一个Proposal进行提交

**集群角色**

1. leader角色

Leader服务器是整个zookeeper集群的核心，主要的工作任务有两项

1. .事物请求的唯一调度和处理者，保证集群事物处理的顺序性
2. .集群内部各服务器的调度者
3. Follower角色
4. .处理客户端非事物请求、转发事物请求给leader服务器
5. .参与事物请求Proposal的投票（需要半数以上服务器通过才能通知leader commit数据; Leader发起的提案，要求Follower投票）
6. .参与Leader选举的投票
7. Observer角色

Observer是zookeeper3.3开始引入的一个全新的服务器角色，从字面来理解，该角色充当了观察者的角色。观察zookeeper集群中的最新状态变化并将这些状态变化同步到observer服务器上。Observer的工作原理与follower角色基本一致，而它和follower角色唯一的不同在于observer不参与任何形式的投票，包括事物请求Proposal的投票和leader选举的投票。简单来说，observer服务器只提供非事物请求服务，通常在于不影响集群事物处理能力的前提下提升集群非事物处理的能力（在不影响写的性能下提升了读的性能）

**集群组成**

通常zookeeper是由2n+1台server组成，每个server都知道彼此的存在。对于2n+1台server，只要有n+1台（大多数）server可用，整个系统保持可用。我们已经了解到，一个zookeeper集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信，基于这个特性，如果向搭建一个能够允许F台机器down掉的集群，那么就要部署2\*F+1台服务器构成的zookeeper集群。因此3台机器构成的zookeeper集群，能够在挂掉一台机器后依然正常工作。一个5台机器集群的服务，能够对2台机器怪调的情况下进行容灾。如果一台由6台服务构成的集群，同样只能挂掉2台机器。因此，5台和6台在容灾能力上并没有明显优势，反而增加了网络通信负担。系统启动时，集群中的server会选举出一台server为Leader，其它的就作为follower（这里先不考虑observer角色）。之所以要满足这样一个等式，是因为一个节点要成为集群中的leader，需要有超过及群众过半数的节点支持，这个涉及到leader选举算法。同时也涉及到事务请求的提交投票

**ZAB协议**

概念：ZAB(Zookeeper Atomic Broadcast) 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

模式：ZAB 协议包含两种基本模式，分别是崩溃恢复和原子广播。

1. 崩溃恢复

当整个集群在启动或者当 leader 节点出现网络中断、崩溃等情况时，ZAB 协议就会进入恢复模式并选举产生新的 leader，当 leader 服务器选举出来后并且集群中有过半的机器和该 leader 节点完成数据同步(用来保证集群中过半的机器能够和 leader 服务器的数据状态保持一致)后，ZAB 协议就会退出恢复模式。简单地说，就是 leader 挂了，然后需要选举出新的 leader（ZK是有中心化节点的中间件），完成新 leader 和 follower 和 observer 的数据同步。

原理：

1. 、选举出新的leader

基于下面的数据恢复的要点，必须要设计一个leader算法，能够确保已经被 leader 提交的事务 Proposal 能够提交、同时丢弃已经被跳过的事务 Proposal

选举阶段，集群间互传的消息称为投票，投票Vote主要包括二个维度的信息：ID、ZXID

1. 、被推举的leader的服务器ID，集群中的每个zk节点启动前就要配置好这个全局唯一的ID
2. 、被推举的leader的事务ID ，该值是从机器DataTree内存中取的，即事务已经在机器上被commit过了。

节点进入选举阶段后的大体执行逻辑如下：

1. 、设置状态为LOOKING，初始化内部投票Vote (id,zxid) 数据至内存，并将其广播到集群其它节点。节点首次投票都是选举自己作为leader，将自身的服务ID、处理的最近一个事务请求的ZXID（ZXID是从内存数据库里取的，即该节点最近一个完成commit的事务id）及当前状态广播出去。然后进入循环等待及处理其它节点的投票信息的流程中。
2. 、循环等待流程中，节点每收到一个外部的Vote信息，都需要将其与自己内存Vote数据进行PK，规则为取ZXID大的，若ZXID相等，则取ID大的那个投票。若外部投票胜选，节点需要将该选票覆盖之前的内存Vote数据，并再次广播出去；同时还要统计是否有过半的赞同者与新的内存投票数据一致，无则继续循环等待新的投票，有则需要判断leader是否在赞同者之中，在则退出循环，选举结束后根据选举结果及各自角色切换状态，leader切换成LEADING、follower切换到FOLLOWING、observer切换到OBSERVING状态。
3. 、数据恢复
4. 、已经被处理的消息不能丢：当 leader收到合法数量 follower的acks后，就向各个follower广播COMMIT命令，同时也会在本地执行COMMIT并向连接的客户端返回acks。但是如果在各个follower在收到COMMIT命令前leader就挂了，导致剩下的服务器并没有执行都这条消息。即leader在发commit命令前挂了（此时follower已经将该消息存入本地，只是没有commit），导致 follower 没有收到这条已经被 leader 处理的消息。
5. 、被丢弃的消息不能再次出现：当leader接收到消息请求生成 proposal后就挂了，其他follower并没有收到此proposal，因此经过恢复模式重新选了leader后，这条消息是被跳过的。 此时之前挂了的leader重新启动并注册成了follower，他保留了被跳过消息的proposal状态，与整个系统的状态是不一致的，需要将其删除。
6. 原子广播

当集群中已经有过半的 follower 节点完成了和 leader 状态同步以后，那么整个集群就进入了消息广播模式。这个时候 leader 节点正常工作，如果启动一台新的服务器加入到集群，那这个服务器会直接进入数据恢复模式，和 leader 节点进行数据同步。同步完成后即可正常对外提供非事务请求的处理。

消息广播的原理（zxid）：消息广播的过程实际上是一个简化版本的二阶段（2PC）提交过程。

1). leader 接收到消息请求后，将消息赋予一个全局唯一的64 位自增 id（叫:zxid），通过 zxid 的大小比较既可以实现因果有序这个特征。

2). leader 为每个 follower 准备了一个 FIFO 队列(通过 TCP协议来实现，以实现了全局有序这一个特点)将带有 zxid的消息作为一个提案(proposal)分发给所有的 follower。

3). 当 follower 接收到 proposal，先把 proposal 写到磁盘，写入成功以后再向 leader 回复一个 ack。

4). 当 leader 接收到合法数量(超过半数节点)的 ACK 后，leader 就会向这些 follower 发送 commit 命令，同时会在本地执行该消息。

5). 当 follower 收到消息的 commit 命令以后会提交该消息。

**数据存储**

事务日志:zoo.cfg的配置文件下面的dataDir配置项指定路径生成

快照日志

运行时日志: bin/zookeeper.out

**Java应用**

见com.jyh.middleware.zookeeper.ZookeeperTest

**Watcher事件机制**

绑定事件的方法：getData、Exists、getChildren

触发事件：事务操作增（create）、删（delete）、改（setData）

事件类型：

None (-1)：客户端链接状态发生变化的时候，会收到none的事件

NodeCreated (1)：创建节点的事件。比如zk-persis-micNode

Deleted (2): 删除节点的事件NodeData

Changed (3): 节点数据发生变更

NodeChildrenChanged (4): 子节点被创建、被删除、会发生事件触发

**Curator客户端使用**

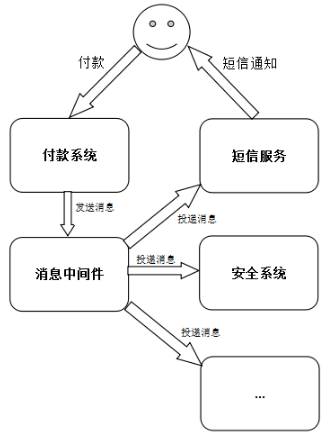
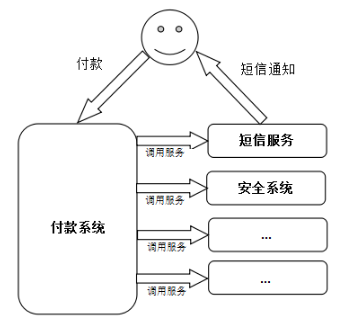
Curator是Netflix公司开源的一个ZooKeeper client library，用于简化ZooKeeper客户端编程

**MQ**

**初识消息中间件**

概念：消息中间件是在分布式系统中完成消息的发送和传递的基础软件

作用：异步和解耦



引入中间件之前：这会让付款系统变得非常复杂，每增加一个在付款成功之后需要调用的系统，就需要修改付款系统来进行相关的调用

引入中间件之后：通过消息中间件，付款系统就不需要关心到底有多少个系统需要知道付款成功这件事情了，也不用关心如何通知它们，只需要把登陆成功这件事情转化为一个消息发送到消息中间件就好了，这样，需要了解付款成功这件事情的系统自己去消息中间件订阅就行了，并且各个系统之间也是互不影响的

**JMS概述**

JMS是Java Message Service的缩写，即Java消息服务，它是Java EE中一个关于消息的规范，而Hornetq、ActiveMQ等产品是对这个规范的实现。

Java消息服务（Java Message Service）是java平台中关于面向消息中间件的API，用于在两个应用程序之间，或者分布式系统中发送消息，进行异步通信

JMS是一个与具体平台无关的API，绝大多数MOM（Message Oriented Middleware）（面向消息中间件）提供商都对JMS提供了支持。

在大型互联网中，我们采用消息中间件可以进行应用之间的解耦以及操作的异步，这是消息中间件两个最基础的特点，也正是我们所需要的。在此基础上，我们着重思考的是消息的顺序保证、扩展性、可靠性、业务操作与消息发送一致性，以及多集群订阅者等方面的问题。当然，这些我们要思考的东西，JMS都已经想到了，先看下JMS能帮开发者做什么：

1、定义一组消息公用概念和实用工具

所有Java应用程序都可以使用JMS中定义的API去完成消息的创建、接收与发送，任何实现了JMS标准的MOM都可以作为消息的中介，完成消息的存储转发

2、最大化消息应用程序的可移植性

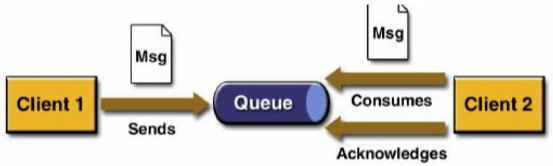
MOM提供了有保证的消息发送，应用程序开发人员无需了解远程过程调用（PRC）和网络/通信协议的细节，提供了程序的可移植性

3、最大化降低应用程序与应用程序之间的耦合度

由于MOM(Message Oriented Middleware)()的存在，各个应用程序只关心和MOM之间如何进行消息的接收与发送，而无须关注MOM的另一边，其他程序是如何接收和发送的

**JMS两种消息模型**

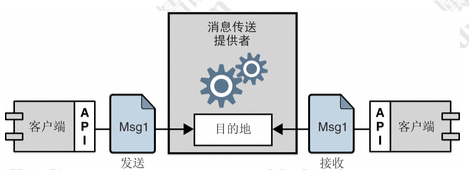
1. 点对点（point-to-point）(p2p)消息传递域
2. 、每个消息只能有一个消费者
3. 、消息的生产者和消费者之间没有时间上的相关性。无论消费者在生产者发送消息的时候是否处于运行状态，都可以提取消息



1. 发布/订阅(publish/subscribe)(pub/sub)消息传递域
2. 、每个消息可以有多个消费者
3. 、生产者和消费者之间有时间上的相关性。订阅一个主题的消费者只能消费自它订阅之后发布的消息。JMS规范允许客户创建持久订阅，这在一定程度上降低了时间上的相关性要求。持久订阅允许消费者消费它在未处于激活状态时发送的消息

**什么是MOM**

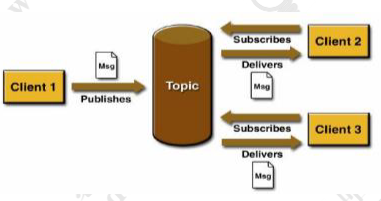
MOM是面向消息的中间件，使用消息传送提供者来协调消息传送操作。MOM需要提供API和管理工具。客户端使用api调用，把消息发送到由提供者管理的目的地。在发送消息之后，客户端会继续执行其他工作，并且在接收方收到这个消息确认之前，提供者一直保留该消息。



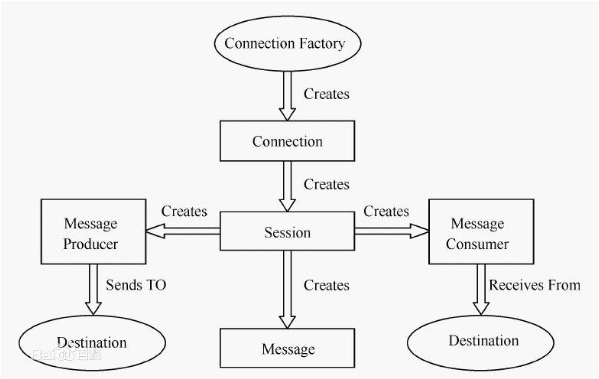
MOM的特点

1. 消息异步接收，发送者不需要等待消息接受者响应
2. 消息可靠接收，确保消息在中间件可靠保存。只有接收方收到后才删除消息

Java 消息传送服务规范最初的开发目的是为了使Java 应用程序能够访问现有MOM 系统。引入该规范之后，它已被许多现有的MOM 供应商采用并且已经凭借自身的功能实现为异步消息传送系统。



**JMS的体系结构**



**消息结构组成**

1. 消息头

消息头(Header) -消息头包含消息的识别信息和路由信息，消息头包含一些标准的属性如：

JMSDestination：消息发送的目的地，queue或者topic。

JMSDeliveryMode：传送模式，持久模式和非持久模式

JMSPriority：消息优先级（优先级分为10个级别，从0(最低)到9(最高). 如果不设定优先级，默认级别是4。需要注意的是， JMS provider并不一定保证按照优先级的顺序提交消息）

JMSMessageID 唯一识别每个消息的标识

1. 属性
2. 消息体

就是我们需要传递的消息内容，JMSAPI定义了5中消息体格式，可以使用不同形式发送接收数据，并可以兼容现有的消息格式，其中包括



**持久化订阅**

概念：持久订阅的概念，也很容易理解，比如还是以QQ为例，我们把QQ退出了，但是下次登录的时候，仍然能收到离线的消息。

特点：1.持久订阅者和非持久订阅者针对的Domain是Pub/Sub，而不是P2P

2.当Broker发送消息给订阅者时，如果订阅者处于未激活状态状态，持久订阅者可以收到消息，而非持久订阅者则收不到消息。

总结：当然这种方式也有一定的影响：当持久订阅者处于未激活状态时，Broker需要为持久订阅者保存消息；如果持久订阅者订阅的消息太多则会溢出。

**JMS消息的可靠性机制**

理论上来说，我们需要保证消息中间件上的消息，只有被消费者确认过以后才会被签收，相当于我们寄一个快递出去，收件人没有收到快递，就认为这个包裹还是属于待签收状态，这样才能保证包裹能够安全达到收件人手里。

消息中间件也是一样。消息的消费通常包含3个阶段：客户接收消息、客户处理消息、消息被确认首先，来简单了解JMS的事务性会话和非事务性会话的概念JMS Session接口提供了commit和rollback方法。事务提交意味着生产的所有消息被发送，消费的所有消息被确认；事务回滚意味着生产的所有消息被销毁，消费的所有消息被恢复并重新提交，除非它们已经过期。事务性的会话总是牵涉到事务处理中，commit或rollback方法一旦被调用，一个事务就结束了，而另一个事务被开始。关闭事务性会话将回滚其中的事务

1. 事务
2. 、在事务状态下进行发送操作，消息并未真正投递到中间件，而只有进行session.commit操作之后，消息才会发送到中间件，再转发到适当的消费者进行处理。如果是调用rollback操作，则表明，当前事务期间内所发送的消息都取消掉。通过在创建session的时候使用true or false来决定当前的会话是事务性还是非事务性connection.createSession(Boolean.TRUE,Session.AUTO\_ACKNOWLEDGE);在事务性会话中，消息的确认是自动进行，也就是通过session.commit()以后，消息会自动确认。

必须保证发送端和接收端都是事务性会话

1. 、消息何时被确认取决于创建会话时的应答模式(acknowledgement mode). 有三个可选项Session.AUTO\_ACKNOWLEDGE当客户成功的从receive方法返回的时候，或者从MessageListenner.onMessage方法成功返回的时候，会话自动确认客户收到消息。Session.CLIENT\_ACKNOWLEDGE客户通过调用消息的acknowledge方法确认消息。Session.DUPS\_ACKNOWLEDGE消息延迟确认。指定消息提供者在消息接收者没有确认发送时重新发送消息，这种模式不在乎接受者收到重复的消息。
2. 消息的持久化存储

消息的持久化存储也是保证可靠性最重要的机制之一，也就是消息发送到Broker上以后，如果broker出现故障宕机了，那么存储在broker上的消息不应该丢失。可以通过下面的代码来设置消息发送端的持久化和非持久化特性:producer.setDeliveryMode(DeliveryMode.PERSISTENT);

对于非持久的消息，JMS provider不会将它存到文件/数据库等稳定的存储介质中。也就是说非持久消息驻留在内存中，如果jms provider宕机，那么内存中的非持久消息会丢失

对于持久消息，消息提供者会使用存储-转发机制，先将消息存储到稳定介质中，等消息发送成功后再删除。如果jms provider挂掉了，那么这些未送达的消息不会丢失；jms provider恢复正常后，会重新读取这些消息，并传送给对应的消费者。

**ActiveMQ**

**消息同步发送和异步发送**

ActiveMQ支持同步、异步两种发送模式将消息发送到broker上。

同步发送：发送者发送一条消息会阻塞直到broker反馈一个确认消息，表示消息已经被broker处理。这个机制提供了消息的安全性保障，但是由于是阻塞的操作，会影响到客户端消息发送的性能

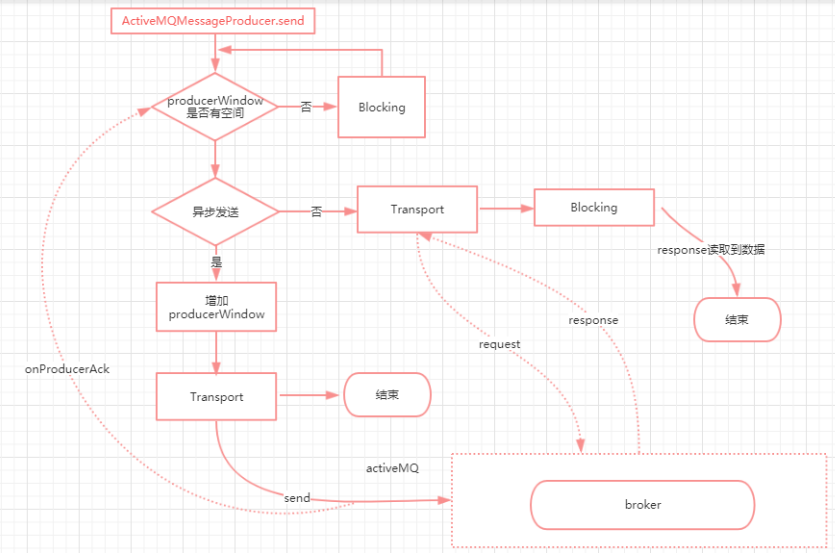
异步发送：异步发送的过程中，发送者不需要等待broker提供反馈，所以性能相对较高。但是可能会出现消息丢失的情况。所以使用异步发送的前提是在某些情况下允许出现数据丢失的情况

默认场景：

1. 非持久化消息是异步发送
2. 持久化消息并且是在非事务模式下是同步发送
3. 开启事务的情况下，消息都是异步发送

代码开启异步发送方式：

1. ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616?jms.useAsyncSend=true")
2. ((ActiveMQConnectionFactory) connectionFactory).setUseAsyncSend(true);
3. ((ActiveMQConnection)connection).setUseAsyncSend(true);



ProducerWindowsSize

含义：producer每发送一个消息，统计一下发送的字节数，当字节数达到ProducerWindowSize值时，需要等待broker的确认，才能继续发送

作用：主要用来约束在异步发送时producer端允许积压的(尚未ACK)的消息的大小，且只对异步发送有意义。每次发送消息之后，都将会导致memoryUsage大小增加(+message.size)，当broker返回producerAck时，memoryUsage尺寸减少(producerAck.size，此size表示先前发送消息的大小)

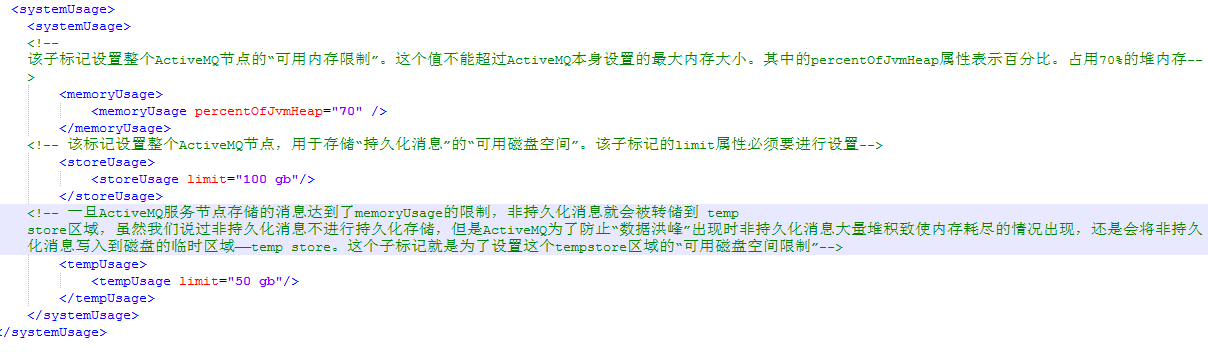
设置：

1. 在brokerUrl中设置: "tcp://localhost:61616?jms.producerWindowSize=1048576",这种设置将会对所有的producer生效。
2. 在destinationUri中设置: "test-queue?producer.windowSize=1048576",此参数只会对使用此Destination实例的producer失效，将会覆盖brokerUrl中的producerWindowSize值

影响：此值越大，意味着消耗Client端的内存就越大

**消息的持久化方案**

正常情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的。能够存储的最大消息数据在${ActiveMQ\_HOME}/conf/activemq.xml文件中的systemUsage节点



从上面的配置我们需要get到一个结论，当非持久化消息堆积到一定程度的时候，也就是内存超过指定的设置阀值时，ActiveMQ会将内存中的非持久化消息写入到临时文件，以便腾出内存。但是它和持久化消息的区别是，重启之后，持久化消息会从文件中恢复，非持久化的临时文件会直接删除

消息的持久化策略：

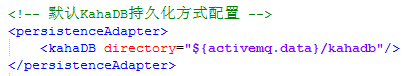
消息持久性对于可靠消息传递来说是一种比较好的方法，即时发送者和接受者不是同时在线或者消息中心在发送者发送消息后宕机了，在消息中心重启后仍然可以将消息发送出去。消息持久性的原理很简单，就是在发送消息出去后，消息中心首先将消息存储在本地文件、内存或者远程数据库，然后把消息发送给接受者，发送成功后再把消息从存储中删除，失败则继续尝试。接下来我们来了解一下消息在broker上的持久化存储实现方式

持久化支持类型：

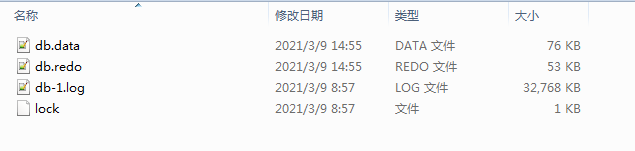
1. KahaDB存储（默认存储方式）

概要：KahaDB是目前默认的存储方式,可用于任何场景,提高了性能和恢复能力。消息存储使用一个事务日志和仅仅用一个索引文件来存储它所有的地址。KahaDB是一个专门针对消息持久化的解决方案,它对典型的消息使用模式进行了优化。在Kaha中,数据被追加到data logs中。当不再需要log文件中的数据的时候,log文件会被丢弃。

配置方式：



存储原理：



1. db.data:它是消息的索引文件，本质上是B-Tree（B树），使用B-Tree作为索引指向db-\*.log里面存储的消息
2. db.redo:用来进行消息恢复
3. db-\*.log:存储消息内容。新的数据以APPEND的方式追加到日志文件末尾。属于顺序写入，因此消息存储是比较快的。默认是32M，达到阀值会自动递增
4. lock:表示当前获得kahadb读写权限的broker
5. JDBC存储

概要：

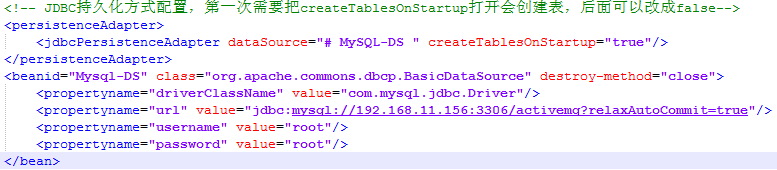
使用JDBC持久化方式，数据库会创建3个表：activemq\_msgs，activemq\_acks和activemq\_lock。

ACTIVEMQ\_MSGS 消息表，queue和topic都存在这个表中

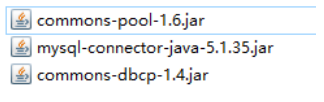
ACTIVEMQ\_ACKS 存储持久订阅的信息和最后一个持久订阅接收的消息ID

ACTIVEMQ\_LOCKS 锁表，用来确保某一时刻，只能有一个ActiveMQ broker实例来访问数据库

配置：

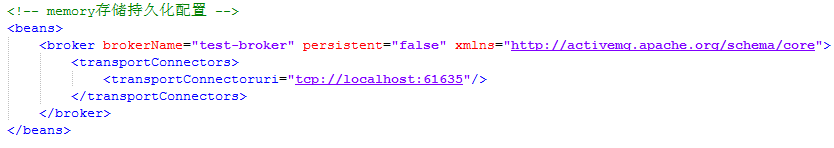


需要添加jar包依赖：



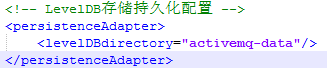
1. Memory存储

概要：基于内存的消息存储，内存消息存储主要是存储所有的持久化的消息在内存中。persistent=”false”,表示不设置持久化存储，直接存储到内存中



1. LevelDB存储

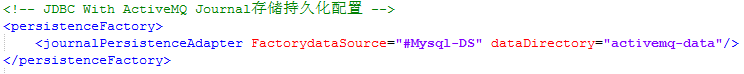
概要：LevelDB持久化性能高于KahaDB，虽然目前默认的持久化方式仍然是KahaDB。并且，在ActiveMQ 5.9版本提供了基于LevelDB和Zookeeper的数据复制方式，用于Master-slave方式的首选数据复制方案。



1. JDBC With ActiveMQ Journal

概要：这种方式克服了JDBC Store的不足，JDBC每次消息过来，都需要去写库和读库。ActiveMQ Journal，使用高速缓存写入技术，大大提高了性能。

当消费者的消费速度能够及时跟上生产者消息的生产速度时，journal文件能够大大减少需要写入到DB中的消息。举个例子，生产者生产了1000条消息，这1000条消息会保存到journal文件，如果消费者的消费速度很快的情况下，在journal文件还没有同步到DB之前，消费者已经消费了90%的以上的消息，那么这个时候只需要同步剩余的10%的消息到DB。如果消费者的消费速度很慢，这个时候journal文件可以使消息以批量方式写到DB。



**消费端消费消息的原理**

**关于PrefetchSize的优化**

概要：activemq的consumer端也有窗口机制，通过prefetchSize就可以设置窗口大小。不同的类型的队列，prefetchSize的默认值也是不一样的

默认值：

1. 、persistent queues (default value: 1000)
2. 、non-persistent queues (default value: 1000)
3. 、persistent topics (default value: 100)
4. 、non-persistent topics (default value: Short.MAX\_VALUE - 1)

消费端会根据 prefetchSize 的大小批量获取数据，比如说默认值是1000，那么消费者就会预先加载1000条数据到本地的内存中(unconsumedMessages队列中)

既然有批量加载，那么一定有批量确认，这样才算是彻底的优化

代码设置：

Destination destination = session.createTopic("testTopic?consumer.prefetchSize=10");

optimizeAcknowledge

1. 、概要：ActiveMQ提供了optimizeAcknowledge来优化确认，它表示是否开启“优化ACK”，只有在为true的情况下，prefetchSize以及optimizeAcknowledgeTimeout参数才会有意义，优化确认一方面可以减轻client负担（不需要频繁的确认消息）、减少通信开销，另一方面由于延迟了确认（默认ack了0.65\*prefetchSize个消息才确认），broker再次发送消息时又可以批量发送如果只是开启了prefetchSize，每条消息都去确认的话，broker在收到确认后也只是发送一条消息，并不是批量发布，当然也可以通过设置DUPS\_OK\_ACK来手动延迟确认
2. 、设置方式：ConnectionFactory connectionFactory=newActiveMQConnectionFactory("tcp://192.168.11.153:61616?jms.optimizeAcknowledge=true&jms.optimizeAcknowledgeTimeOut=10000");

总结：到目前为止，我们知道了optimizeAcknowledge和prefetchSize的作用，两者协同工作，通过批量获取消息、并延迟批量确认，来达到一个高效的消息消费模型。它不仅减少了客户端在获取消息时的阻塞次数，还能减少每次获取消息时的网络通信开销。 需要注意的是，如果消费端的消费速度比较高，通过这两者组合是能大大提升consumer的性能。如果consumer的消费性能本身就比较慢，设置比较大的prefetchSize反而不能有效的达到提升消费性能的目的。因为过大的prefetchSize不利于consumer端消息的负载均衡。因为通常情况下，我们都会部署多个consumer节点来提升消费端的消费性能。这个优化方案还会存在另外一个潜在风险，当消息被消费之后还没有来得及确认时，client端发生故障，那么这些消息就有可能会被重新发送给其他consumer，那么这种风险就需要client端能够容忍“重复”消息。

**消息确认过程**

消息确认机制：

optimizeAck表示是否开启“优化ACK”, 只有在为true的情况下，prefetchSize(下文中将会简写成prefetch)以及optimizeAcknowledgeTimeout参数才会有意义, "optimizeAcknowledgeTimeout"选项只能在brokerUrl中配置, 在destinationUri中指定prefetchSize(预获取)选项, 其中brokerUrl参数选项是全局的，如果同时指定，brokerUrl中的参数选项值将会被覆盖;

当consumer端使用MessageListener异步获取消息时: prefetch>=1

当consumer端使用receive()方法同步获取消息时: prefetch>=0

prefetch=0, receive()方法将会首先发送一个PULL指令并阻塞，直到broker端返回消息为止;

prefetch>0, broker端将会批量push给client 一定数量的消息(<= prefetch),client端会把这些消息(unconsumedMessage)放入到本地的队列中;

ACK\_MODE:

1. 、AUTO\_ACKNOWLEDGE = 1 自动确认
2. 、CLIENT\_ACKNOWLEDGE = 2 客户端手动确认
3. 、DUPS\_OK\_ACKNOWLEDGE = 3 自动批量确认
4. 、SESSION\_TRANSACTED = 0 事务提交并确认

虽然Client端指定了ACK模式,但是在Client与broker在交换ACK指令的时候,还需要告知ACK\_TYPE,ACK\_TYPE表示此确认指令的类型，不同的ACK\_TYPE将传递着消息的状态，broker可以根据不同的ACK\_TYPE对消息进行不同的操作。

ACK\_TYPE:

1. 、DELIVERED\_ACK\_TYPE = 0 消息"已接收"，但尚未处理结束
2. 、STANDARD\_ACK\_TYPE = 2 "标准"类型,通常表示为消息"处理成功"，broker端可以删除消息了
3. POSION\_ACK\_TYPE = 1 消息"错误",通常表示"抛弃"此消息，比如消息重发多次后，都无法正确处理时，消息将会被删除或者DLQ(死信队列)
4. 、REDELIVERED\_ACK\_TYPE = 3 消息需"重发"，比如consumer处理消息时抛出了异常，broker稍后会重新发送此消息
5. 、INDIVIDUAL\_ACK\_TYPE = 4 表示只确认"单条消息",无论在任何ACK\_MODE下
6. 、UNMATCHED\_ACK\_TYPE = 5 在Topic中，如果一条消息在转发给“订阅者”时，发现此消息不符合Selector过滤条件，那么此消息将不会转发给订阅者，消息将会被存储引擎删除(相当于在Broker上确认了消息)

**消息重发机制**

重发的情况：

1. 、在事务性会话中，没有调用session.commit确认消息或者调用session.rollback方法回滚消息
2. 、在非事务性会话中，ACK模式为CLIENT\_ACKNOWLEDGE的情况下，没有调用acknowledge或者调用了recover方法
3. 、一个消息被redelivedred超过默认的最大重发次数（默认6次）时，消费端会给broker发送一个”poison ack”(ActiveMQMessageConsumer#dispatch：1460行)，表示这个消息有毒，告诉broker不要再发了。这个时候broker会把这个消息放到DLQ（死信队列）。

死信队列：ActiveMQ中默认的死信队列是ActiveMQ.DLQ，如果没有特别的配置，有毒的消息都会被发送到这个队列。默认情况下，如果持久消息过期以后，也会被送到DLQ中。

死信队列配置：

**多节点高性能方案**

**KafKa**

**RabbitMQ**

**Redis**

**Nginx**