# ActiveMQ--3.消息持久化

ActiveMQ的消息持久化机制有JDBC，AMQ，KahaDB和LevelDB，还有一种内存存储的方式，由于内存不属于持久化范畴，而且如果使用内存队列，可以考虑使用更合适的产品，如ZeroMQ。所以内存存储不在讨论范围内。

无论使用哪种持久化方式，消息的存储逻辑都是一致的。

消息分为Queue和Topic两种，Queue是点对点消费，发送者发送一条消息，只有一个且唯一的一个消费者能对其进行消费。

Topic是订阅式消费，一个消息可以被很多的订阅者消费，其中定阅者又分为持久化订阅和非持久化订阅。持久化订阅是指即使订阅者当前不在线，其订阅之后，发送方发到Broker的消息，也会在持久化订阅者再次上线的时候完成消费，不会丢失消息。而非持久化订阅者，只有订阅者在线时才会消费，不在线时，即使Broker收到新的消息，当其再次上线时，也不会收到错过的消息。

ActiveMQ的持久化机制，对于Queue类型的消息，将存储在Broker，但是一旦其中一个消费者完成消费，则立即删除这条消息。对于Topic类型的消息，即使所有的订阅者都完成了消费，Broker也不一定会马上删除无用消息，而是保留推送历史，之后会异步清除无用消息。而每个订阅者消费到了哪条消息的offset会记录在Broker，以免下次重复消费。因为消息是顺序消费，先进先出，所以只需要记录上次消息消费到哪里就可以了。

配置持久化的方式，都是修改%ACTIVEMQ\_HOME%conf/acticvemq.xml文件。

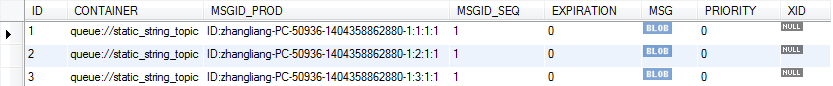
下面分别介绍几种持久化方式的特点：

**JDBC：**很多企业级应用比较喜欢这种存储方式。优点是大多数企业都有专门的DBA，以数据库作为存储介质，会让有这方面人才的公司比较放心。另外，数据库的存储方式，可以看到消息是如何存储的，可以通过SQL查询消息消费状态，可以查看消息内容，这是其他持久化方式所不具备的。还有一个优点就是数据库可以支持强一致性事务，支持两阶段提交的分布式事务。缺点是性能问题，数据库持久化是性能最低的一种方式。

之所以最先介绍数据库的持久化方式，是因为我们可以通过表结构很好的理解ActiveMQ是怎么存储和消费消息的。

数据库会创建3个表：activemq\_msgs，activemq\_acks和activemq\_lock。

activemq\_msgs用于存储消息，Queue和Topic都存储在这个表中。



下面介绍一下主要的数据库字段：

ID：自增的数据库主键

CONTAINER：消息的Destination

MSGID\_PROD：消息发送者客户端的主键

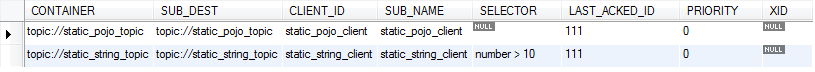
MSG\_SEQ：是发送消息的顺序，MSGID\_PROD+MSG\_SEQ可以组成JMS的MessageID

EXPIRATION：消息的过期时间，存储的是从1970-01-01到现在的毫秒数

MSG：消息本体的Java序列化对象的二进制数据

PRIORITY：优先级，从0-9，数值越大优先级越高

activemq\_acks用于存储订阅关系。如果是持久化Topic，订阅者和服务器的订阅关系在这个表保存。



主要的数据库字段如下：

CONTAINER：消息的Destination

SUB\_DEST：如果是使用Static集群，这个字段会有集群其他系统的信息

CLIENT\_ID：每个订阅者都必须有一个唯一的客户端ID用以区分

SUB\_NAME：订阅者名称

SELECTOR：选择器，可以选择只消费满足条件的消息。条件可以用自定义属性实现，可支持多属性AND和OR操作

LAST\_ACKED\_ID：记录消费过的消息的ID。

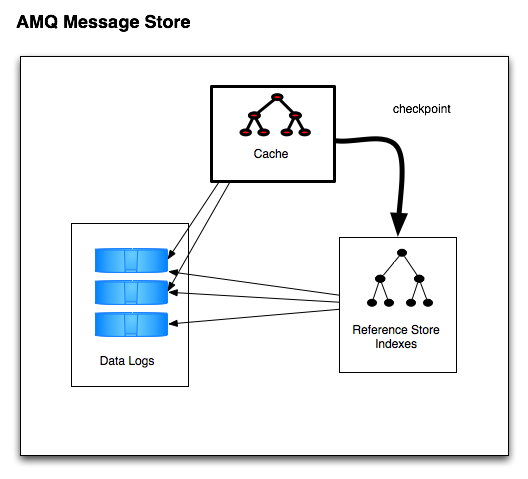
表activemq\_lock在集群环境中才有用，只有一个Broker可以获得消息，称为Master Broker，其他的只能作为备份等待Master Broker不可用，才可能成为下一个Master Broker。这个表用于记录哪个Broker是当前的Master Broker。

配置如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | < beans >       < broker  brokerName = "test-broker"  persistent = "true"  xmlns = "<http://activemq.apache.org/schema/core>" >           < persistenceAdapter >               < jdbcPersistenceAdapter  dataSource = "#mysql-ds" />           </ persistenceAdapter >       </ broker >         < bean  id = "mysql-ds"  class = "org.apache.commons.dbcp.BasicDataSource"  destroy-method = "close" >           < property  name = "driverClassName"  value = "com.mysql.jdbc.Driver" />           < property  name = "url"  value = "jdbc:mysql://localhost/activemq?relaxAutoCommit=true" />           < property  name = "username"  value = "activemq" />           < property  name = "password"  value = "activemq" />           < property  name = "maxActive"  value = "200" />           < property  name = "poolPreparedStatements"  value = "true" />       </ bean >  </ beans > |

首先定义一个mysql-ds的MySQL数据源，然后在persistenceAdapter节点中配置jdbcPersistenceAdapter并且引用刚才定义的数据源。

**AMQ：**性能高于JDBC，写入消息时，会将消息写入日志文件，由于是顺序追加写，性能很高。为了提升性能，创建消息主键索引，并且提供缓存机制，进一步提升性能。每个日志文件的大小都是有限制的（默认32m，可自行配置）。当超过这个大小，系统会重新建立一个文件。当所有的消息都消费完成，系统会删除这个文件或者归档（取决于配置）。主要的缺点是AMQ Message会为每一个Destination创建一个索引，如果使用了大量的Queue，索引文件的大小会占用很多磁盘空间。而且由于索引巨大，一旦Broker崩溃，重建索引的速度会非常慢。



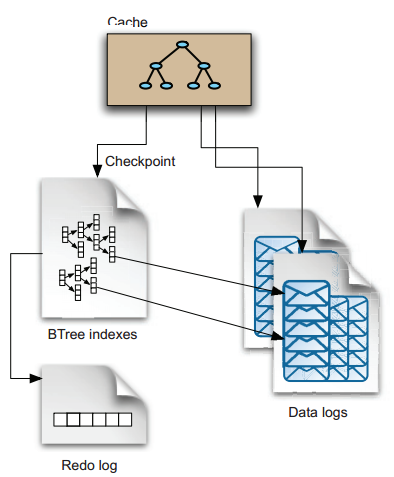
配置片段如下：

[?](http://my.oschina.net/u/719192/blog/287434)

|  |  |
| --- | --- |
| 1  2  3 | < persistenceAdapter >        < amqPersistenceAdapter  directory = "${activemq.data}/activemq-data"  maxFileLength = "32mb" />  </ persistenceAdapter > |

虽然AMQ性能略高于Kaha DB，但是由于其重建索引时间过长，而且索引文件占用磁盘空间过大，所以已经不推荐使用。这里就不在详细介绍AMQ持久化的数据结构。**在新版本的ActiveMQ中，AMQ已经被删除。**

**KahaDB：**从ActiveMQ 5.4开始默认的持久化插件，KahaDb恢复时间远远小于其前身AMQ并且使用更少的数据文件，所以可以完全代替AMQ。kahaDB的持久化机制和AMQ非常像。同样是基于日志文件，索引和缓存。和AMQ不同，KahaDB所有的Destination都使用一个索引文件。《ActiveMQ In Action》表示其可以支持10000个连接，每个连接都是一个独立的Queue，足以满足大部分应用场景。



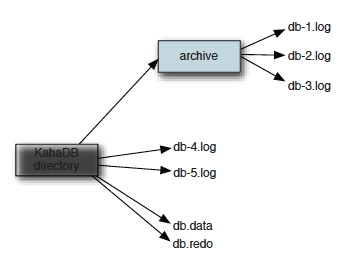
Data logs用于存储消息日志，消息的全部内容都在Data logs中。同AMQ一样，一个Data logs文件大小超过规定的最大值，会新建一个文件。同样是文件尾部追加，写入性能很快。每个消息在Data logs中有计数引用，所以当一个文件里所有的消息都不需要了，系统会自动删除文件或放入归档文件夹。

缓存用于存放在线消费者的消息。如果消费者已经快速的消费完成，那么这些消息就不需要再写入磁盘了。

Btree索引会根据MessageID创建索引，用于快速的查找消息。这个索引同样维护持久化订阅者与Destination的关系，以及每个消费者消费消息的指针。

Redo log用于系统崩溃后，重建Btree索引。因为Redo log的存在，使得重建索引时不需要读取Data logs的全量数据，大大提升性能。

KahaDB的目录结构：

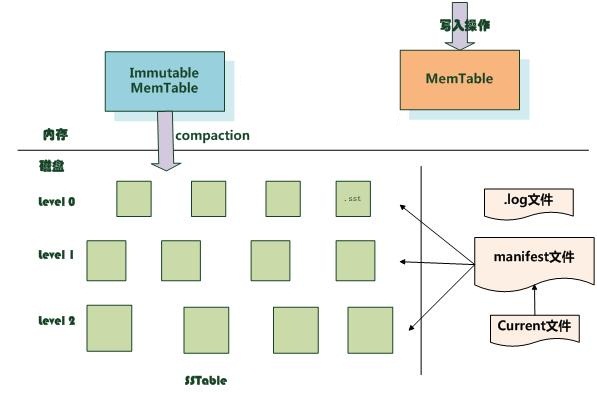


db log文件，以db-<Number>.log命名。archive目录用于存档归档的数据。db.data和db.redo分别是Btree索引和redo log。

由于是ActiveMQ的默认持久化机制，所以不需要修改配置文件就可以使用KahaDB，但是还是贴出配置片段：

|  |  |
| --- | --- |
| 1  2  3 | < persistenceAdapter >       < kahaDB  directory = "${activemq.data}/activemq-data"  journalMaxFileLength = "16mb" />  </ persistenceAdapter > |

**LevelDB：**从ActiveMQ 5.6版本之后，又推出了LevelDB的持久化引擎。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，但是LevelDB是将来的趋势。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。LevelDB使用自定义的索引代替常用的BTree索引。



通过上图可以看出LevelDB主要由6部分组成：内存中的MemTable和ImmutableMemTable，还有硬盘上的log文件，manifest文件，current文件和SSTable文件。还有一些其他的辅助文件，暂时不做说明。

每写入一次数据，需要写入log文件，和MemTable，也就是说，只需要一次硬盘的顺序写入，和一个内存写入，如果系统崩溃，可以通过log文件恢复数据。每次写入会先写log文件，后写MemTable来保证不丢失数据。

当MemTable到达内存阀值，LevelDB会创建一个新的MemTable和log文件，而旧的MemTable会变成ImmutableMemTable，ImmutableMemTable的内容是只读的。然后系统会定时的异步的把ImmutableMemTable的数据写入新的SSTable文件。

SSTable文件和MemTable，ImmutableMemTable的数据结构相同，都是key，value的数据，按照key排序。

manifest文件用于记录每个SSTable的key的起始值和结束值，有点类似于B-tree索引。而manifest同样会生成新文件，旧的文件不再使用。current文件就是指定哪个manifest文件是现在正在使用的。

配置片段如下：

|  |  |
| --- | --- |
| 1  2  3 | < persistenceAdapter >       < levelDB  directory = "${activemq.data}/activemq-data" />  </ persistenceAdapter > |

**在目前的ActiveMQ 5.10版本中，直接使用LevelDB会导致服务不能启动，抛出java.io.IOException: com.google.common.base.Objects.firstNonNull(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;**

原因是有两个Guava cache导致版本冲突，解决的办法是：

删除%ACTIVEMQ\_HOME%lib下面的pax-url-aether-1.5.2.jar

注释掉%ACTIVEMQ\_HOME%conf/activemq.xml的下面几行：

|  |  |
| --- | --- |
| 1  2  3  4 | < bean  id = "logQuery"  class = "org.fusesource.insight.log.log4j.Log4jLogQuery"  lazy-init = "false"  scope = "singleton"  init-method = "start"  destroy-method = "stop" >  </ bean > |

这个BUG地址是<https://issues.apache.org/jira/browse/AMQ-5225>，希望可以在下个版本顺利解决。

下面是跑在我机器上的性能测试，实际数据意义不大，因为每个环境的配置都不同，但是可以通过对比看出几种持久化方式的性能对比。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 发送1000条消息(毫秒) | 发送10000条消息(毫秒) | 消费1000条消息的时间(毫秒) | 消费10000条消息的时间(毫秒) |
| JDBC-Mysql | 43009 | 369802 | 610 | 509338 |
| KahaDB | 34227 | 360493 | 208 | 2224 |
| LevelDB | 34032 | 347712 | 220 | 2877 |

通过这个表格可以看出来，发送消息LevelDB最快，KahaDB稍微慢点，JDBC最慢，但是也不会慢太多，是一个数量级。消费消息，KahaDB最快，LevelDB稍微慢点，JDBC慢的让人不能忍受，差好几个数量级。LevelDB并没有显现出比KahaDB更多速度上的优势。但是由于LevelDB支持高可用的复制数据，所以首选肯定还是LevelDB。