# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/063\_如果Broker返回响应消息，在OP\_READ事件中是如何处理的？/笔记.doc**

你之前发送出去的请求，如果说broker给你返回了响应消息，那么你一定会感知到一个OP\_READ事件，在这里会使用while循环，针对一个broker的连接，反复的读，推测一下，因为是这样子的

你的一个broker是可以通过一个连接连续发送出去多个请求的，这个多个请求可能都没有收到响应消息，此时人家broker端可能会连续处理完多个请求然后连续返回多个响应给你，所以在这里，你一旦去读数据

可能会连续读到多个请求的响应

所以说在这里处理OP\_READ事件的时候，必须要通过一个while循环，连续不断的读，可能会读到多个响应消息，全部放到一个暂存的集合里，stagedReceives

底层的KafkaChannel.read -> TransportLayer.read -> SocketChannel.read，我是怎么区分开来不同的请求对应的响应的呢？我到底是怎么通过底层的NIO去进行 响应的读取的呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/073\_重新在内存缓冲里入队的Batch在什么时机下会判定可以重试？/笔记.doc**

对于这个处于重试状态的Batch

lastAttemptMs，是他重新入队的时间，retryBackoffMs其实就是重试的间隔，默认是100ms，他的意思是必须间隔一定的时间再进行重试，这个100ms一般来说建议保持默认值就可以了，但是重试的次数可以自己设置一下，一遍来说建议设置为3次

如果3次重试 都不行，那么一定是Kafka的集群出现问题了，此时人家就会回调你，通知你的回调函数说，重试之后还是异常

重新入队之后到现在必须已经过了100ms了，才能算做backingOff是true

lastAttemptMs + retryBackoffMs > now，意思是什么？上次重新入队的时间到现在还没超过100ms呢，如果说当前时间距离上次入队时间还没到100ms，此时backingOff就是true，如果是true的话，就不能重试

假如说：lastAttemptMs + retryBackoffMs <= now，就说明现在的时间距离上次重新入队的时间已经超过了100ms了，此时backingOff就是false，此时就说明这个要重试的Batch就可以再次发送了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/065\_工业级NIO编程实践：读取数据遇到粘包类问题怎么处理？/笔记.doc**

读取数据的时候，其实可能会遇到连续不断的多个响应粘在一起给你返回回来的，就是你在这里读取，可能会在不停的读取的过程中发现你读到了多个响应消息，这个就是类似于粘包的问题

发送请求，是不会出现粘包类的问题，你自己是可以控制一次只能把一个请求给人家发送过去，你只会出现拆包类的问题，就是一个请求一次没有发送完毕，就需要通过执行多次OP\_WRITE事件才能发送出去

有可能会出现拆包问题，有可能会出现粘包

先说粘包问题

区分哪段数据是一个响应，另外一段数据是下一个响应呢？人家通过短短的一块代码，就把粘包问题和拆包问题都解决了，工业级的代码，非常值得大家来学习，是任何书本、Demo程序都学习不到的

以后如果你自己研发中间件系统，要基于NIO进行网络通信，设计 客户端跟服务器来通信，服务器层的NIO网络通信跟客户端的是类似的，只不过是稍微有一些自己的特点而已，你甚至都可以完全把Kafka的这套网络通信的机制搬过去

如果要解决粘包问题，就是每个响应中间必须插入一个特殊的几个字节的分隔符，一般来说用作分隔符比如很经典的就是在响应消息前面先插入4个字节（integer类型的）代表响应消息自己本身数据大小的数字

响应消息1，199个字节；响应消息2,238个字节；响应消息3,355个字节

199响应消息（1）238响应消息（2）355响应消息（3）

此时会从channel中读取4个字节的数字，写入到size ByteBuffer（4个字节），就是如果已经读取到了4个字节，position就会变成4，就会跟limit是一样的，此时就代表着size ByteBuffer的4个字节已经读满了

ByteBuffer.rewind，把position设置为0，一个ByteBuffer写满之后，调用rewind，把position重置为0，此时就可以从ByteBuffer里读取数据了

ByteBuffer.getInt()，就会默认从ByteBuffer当前position的位置获取4个字节，转换为一个int类型的数字返回给你

接下来就会直接把channel里的一条响应消息的数据读取到一个跟他的大小一致的ByteBuffer中去，粘包问题的解决，就是完美的通过每条消息基于一个4个字节的int数字（他们自己的大小）来进行分割

拆包，假如说size是4个字节，你一次read就读取到了2个字节，连size都没有读取完毕，出现了拆包，此时怎么办呢？或者你读取到了一个size，199个字节，但是在读取响应消息的时候，就读取到了162个字节，拆包问题，响应消息没有读取完毕

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/058\_回头看看发送完请求之后是如何让Selector关注OP\_READ事件的？/笔记.doc**

发送完请求如何关注NIO里的OP\_READ事件呢？

key.interestOps(key.interestOps() & ~SelectionKey.OP\_CONNECT | SelectionKey.OP\_READ);

SeletionKey，里面封装了Selector对一个连接关注那个连接上的哪些事件，OP\_CONNECT，OP\_WRITE，OP\_READ，取消对OP\_CONNECT事件的关注，增加对OP\_READ事件的一个关注，主要都是通过二进制位运算来实现的

一旦建立好连接之后，天然的就会去监听这个连接的OP\_READ事件

要发送请求的时候，会把这个请求暂存到KafkaChannel里去，同时让Selector监视他的OP\_WRITE事件，增加一种OP\_WRITE事件，同时保留了OP\_READ事件，此时Selector会同时监听这个连接的OP\_WRITE和OP\_READ事件

发送完了请求之后，对事件的监听会怎么样呢？一旦写完请求之后，就会把OP\_WRITE事件取消监听，就是此时不关注这个写请求的事件了，此时仅仅保留关注OP\_READ事件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/060\_对于已经发送给Broker的请求会进行什么样的后续处理？/笔记.doc**

实际上就是在刚刚的那个poll方法里，对一个broker发送出去的request

expectResponse应该是通过acks计算出来的，如果说acks = 0的话，也就是不需要对一个请求接收响应，此时expectResponse应该就是false，这个时候直接就会把这个Request从inFlightRequests里面移出去

直接就可以返回一个响应了，其实就是做一个回调

如果说一次请求没有把所有的数据都发送出去的话，会怎么样？

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/075\_如果一个Batch一直停留在内存缓冲里，如何检测到他的超时？/笔记.doc**

如果说超时，一定会调用回调函数，必须去释放到这个batch的内存资源

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/064\_读取响应时在底层如何通过NIO编程实现数据的读取？/笔记.doc**

我们来初步来看看底层读取数据的时候，是怎么通过NIO来实现的

就是很简单的，真的不要把很多复杂的中间件的网络通信，并发编程，磁盘读写，内存管理，没有大家想象的那么的复杂，其实如果你把NIO -> Netty这块底层搞的很扎实的话，那么你看懂大量的开源项目的网络通信层的源码基本问题不大

磁盘读写，你只要把Java IO流体系搞清楚，磁盘读写这一块问题也不大

并发编程，如果你把我们的并发课都看明白，volatile、Atomic、ThreadLocal、synchronized、Lock、并发集合类、线程池、线程基本的操作（join、daemon），都搞明白了，看懂任何开源项目的并发这块的东西，都不是问题

内存管理，主要是搞明白NIO以及集合，JDK集合，List、队列、Map，你要搞明白

SocketChannel.write(ByteBuffer)

SocketChannel.read(ByteBuffer)

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/067\_对刚读取出来的暂存状态的响应消息是如何进行处理的？/笔记.doc**

刚刚读取出来的一些stagedReceives他是如何来进行处理的

如果一个连接一次OP\_READ读取出来多个响应消息的话，在这里仅仅只会把每个连接对应的第一个响应消息会放到completedReceives里面去，放到后面去进行处理，此时有可能某个连接的stagedReceives是不为空的

completedReceives他是如何进行处理的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/069\_如果一个请求被正确处理之后如何调用设置的回调函数？/笔记.doc**

主要就是对获取到的请求进行二进制字节数组的解析，把人家回传过来的数据给解析出来，把响应和请求一一匹配起来，一次请求是对应的每个Partition会有一个Batch放在这个请求里

所以说响应也是一样的，对每个Partition只有一个Batch是有对应的请求的

如果正常情况下，就会回调你的每条消息对应的一个回调函数

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/074\_对于失败的请求多次重试发送之后的结果又会怎么处理呢/笔记.doc**

如果一个Batch是重试发送出去的，成功了，没有什么特别的，直接就是回调函数，然后就是释放资源，那么如果在指定次数内，3次，都没成功，哪怕重试几次都失败了，一定会回调通知你的

在使用Kafka的时候，如果是走异步的消息发送，回调函数的编写是很有必要的

还是最终会释放掉这个Batch占用的内存资源的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/072\_深入看看请求处理异常之后的重试机制是如何实现的？/笔记.doc**

重试的Batch会放入到队列的头部，不是尾部，这样的话，下一次循环的时候就可以优先处理这个要重新发送的Batch了，attempts、lastAttemptMs这些参数都会进行设置，辅助判断这个Batch下一次是什么时候要进行重试发送

Batch的内存资源不会释放掉的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/071\_Batch被处理完毕之后是如何释放底层的内存块资源到缓冲池的？/笔记.doc**

就是应该释放这个Batch底层的内存块的资源，给还回到内存缓冲池里去，让下一个Batch可以重复利用内存块的资源，一个是把内存块的资源给还回去，另外一个就是做并发的通知的处理

如果之前内存已经被耗尽了，此时有线程使用了Condition阻塞在这里等待获取内存资源，一旦有内存资源还回去了，此时就会使用Condition的await方法，唤醒之前阻塞等待的线程，告诉他们说，可以来尝试获取锁，然后申请内存资源了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/066\_工业级NIO编程实践：一条消息无法读完的拆包类问题怎么处理？/笔记.doc**

199 响应消息 238响应消息352响应消息

在读取消息的时候，4个字节的size都没读完，2个字节，或者是199个字节的消息就读到了162个字节，拆包问题怎么来处理的呢？

position = 0，limit = 4

现在读取1个字节，position = 1；读取2个字节，position = 2，此时remaining是2，还剩下个2字节是可以读取的

这一次这个poll里面，对这个broker的读取事件的处理就完事儿了，就读到了2个字节，什么都没有，下一次如果再次执行poll，发现又有数据可以读取了，此时的话呢，就会再次运行到这里去

NetworkReceive还是停留在那里，所以呢可以继续读取

剩余只能读2个字节，所以最多就只能读取2个字节到里面去，4个字节凑满了，此时就说明size数字是可以读取出来了，解决了size的拆包的问题，第二次拆包问题发生了，199个字节的消息，只读取到了162个字节

37个字节是剩余可以读取的

下一次又发现这个broker有OP\_READ可以读取的时候，再次进来，继续读取数据

这段代码，你在外面绝对见不到的，完美的处理了发送请求和读取响应的粘包和拆包的问题，用NIO来编程，主要要自己考虑的其实就是粘包和拆包的问题

只要但凡是你已经有了stagedReceves

积压的响应消息是如何来进行处理

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/076\_如果一个inFlightRequest一直没有收到响应，如何检测他的超时？/笔记.doc**

如果说发现有节点对请求是超时响应的，过了60s还没响应，此时会关闭掉跟那个Broker的连接，认为那个Broker已经故障了 ，做很多内存数据结构的清理，再次标记为需要去重新拉取元数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/059\_工业级网络编程中的NIO实践：基于位运算控制事件的监听/笔记.doc**

梳理和总结一下，通过Kafka客户端源码的研究，对NIO的编程可以有非常好的认识和进步，就是完全掌握利用底层的NIO进行开发的技术，对不同事件的监听和取消监听，是通过二进制位运算的方式来实现的

但是其实人家NIO是支持同时监听一个连接上的多种事件的，就是通过位运算的

key.interestOps() & ~ SelectionKey.OP\_READ | SelectionKey.OP\_WRITE，底层的NIO网络编程里是非常有实践意义的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/061\_看看Kafka生产端的NIO编程是如何进行拆包类问题的处理的？/笔记.doc**

如果Kafka一个请求一次write操作没有把全部的数据都写到broker去，相当于出现了类似于拆包的问题，一个请求一次没法发送完毕，此时如何处理的呢？这个是非常工业级的一个问题的处理方案

如果说一个请求对应的ByteBuffer中的二进制字节数据一次write没有全部发送完毕，如果说一次请求没有发送完毕，此时肯定remaining是大于0，此时就不会取消对OP\_WRITE事件的监听

假设此时针对某个Broker是说，此时是可以再次发送一个Request了，必须得先判断一下，这个Broker上一次发送的Request请求是否发送完毕了，那个request中的数据是否发送完了呢？

即使发送完毕了，还得限制为最多只发送5个request是没有收到响应的

如果说上一次 request出现了类似拆包的问题，一次请求没有发送完毕，此时下次就不会继续往这个broker发送请求了，但是此时针对这个broker还是保持着OP\_WRITE的监听，下次调用poll，会发现对这个broker可以再次执行WRITABLE事件

大不了再次对SocketChannel调用write方法，把ByteBuffer里剩余的数据继续往Broker去写，上述的过程重复多次，一定会把这个请求发送完毕的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/068\_对于确认读取完毕的响应消息是如何解析二进制字节数组的？/笔记.doc**

确认读取完毕的响应消息放在completedReceives

从inFlightRequests中，移除掉一个request，腾出来一个位置，其中的一个请求是获取到了响应消息了，不管是不是成功，去解析他的响应，读取到的数据一定是一段二进制字节数组的一段数据

这段数据一定是按照人家的二进制协议来定义的，比如说返回什么什么东西，什么什么东西，把这段二进制的字节数组，一点一点从里面，先读取8个字节，代表了什么，再读取20个字节，代表了什么

放到一个Java对象里去，就代表了他的响应消息

correlation\_id，是全局唯一的，用来标识一次请求的，也就是说你发送请求的时候，就会带过去这个东东，读取到的响应，首先一定是可以读取到这个correlation\_id的，就知道对应的是哪一次请求

你一定是可以在inFlighRequest里面是知道他对饮的请求的

对于同一个broker，连续发送多个request出去，但是会在inFlighRequest里面排队

inFlighRequests -> <请求1，请求2，请求3，请求4，请求5>

此时对broker读取响应，响应1，响应2，都在stagedReceives -> 响应1放在completedReceives -> 只会获取到响应1

就是直接从inFlighRequests里面移除掉请求1，按照顺序，先发送请求1，那么就应该先获取到请求1对应的响应1，而不是响应2

其实在这里，仅仅是解析一个响应，还没有对响应进行处理呢！

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/077\_KafkaProducer源码精华总结：内存管理、缓存机制、NIO网络通信/笔记.doc**

KafkaProducer源码中的 精华总结

缓冲机制：数据结构，CopyOnWriteMap + Dequeu，Batch + Request

内存管理：内存块缓冲池，有很多空的内存块，可以循环的利用，大幅度减轻JVM GC的弊端，避免频繁的回收大量的内存块

网络通信：NIO封装自己的网络通信框架，KafkaSelector、KafkaChannel，一个客户端对多个Broker服务器建立长连接，缓存维护，IO多路复用，一个主线程完成跟多个客户端的网络通信，读写请求中的粘包和拆包的处理

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/062\_对一个Broker如何同时发送出去多个inFilghtRequest？/笔记.doc**

假设如果说一个Request已经发送完毕了，那么接下来是否可以在接收到响应之前，就继续发送下一个Reqeust呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/058~077资料/070\_从源码层面看看如果一个请求处理异常会进行什么样的处理？/笔记.doc**

一旦说某个请求的响应中，发现了其中某个Batch有异常，就会首当其冲 判断一下，这个Batch是否可以进行重试，首先一个Batch的重试次数（默认从0开始），必须得小于设置的重试次数

默认情况下，是不允许你重试的，异常就是异常，他会在回调函数里通知你，这条消息是有异常的，比如说在客户端缓存的元数据里，知道Partition的Leader在Broker01上，结果此时发送消息过去到那个Parititon Leader

但是Broker01上突然发现，这Leader之前做过一次切换，Leader已经转移到Broker02上去了，此时Broker01会给你一个异常，意思就是说LeaderNotExistException，在我这里找不到对应的Leader

是很常见的，完全是可以进行重试的，但是如果要重试，就必须得重新拉取一下这个Topic的对应的元数据，感知一下这个Partition的Leader现在已经转移到哪里去了，比如说已经到Broker02上去了

重试下一次就应该试试去发送请求到那个Broker上去

NetworkException，网络抖动，网络通信突然短暂的失败，也是可以进行重试的

推测这个重试应该就是说把这个RecordBatch给放回到Accumulator里的Queue里去，Batch的内存资源被释放的过程看一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/039\_内存缓冲中的Batch到底是如何被判定为可以发送出去的？（二）/笔记.doc**

exhuasted，内存是否已经耗尽，可能有人阻塞在写操作，无法申请到内存，在等待新的内存块空闲出来才可以创建新的Batch

backingOff，是跟请求重试有关系的，除非你的请求失败了，此时开始重试，然后就会在这里有一段判断的逻辑，重试是有一个间隔的，默认是100ms，如果进入了重试的阶段，上一次发送这个batch的时间 + 重试间隔的时间，是否大于了当前时间

如果一旦进入了重试阶段，每次发送这个Batch，都必须符合重试的间隔才可以，必须得是超过了重试间隔的时间之后，才可以再次发送这个Batch

刚开始的时候，默认情况下，发送一个Batch，肯定是不涉及到重试，attempts就一定是0，一定没有进入重试的状态

waitedTimeMs，当前时间减去上一次发送这个Batch的时间，假设一个Batch从来没有发送过，此时当前时间减去这个Batch被创建出来的那个时间，这个Batch从创建开始到现在已经等待了多久了

timeToWaitMs，这个Batch从创建开始算起，最多等待多久就必须去发送，如果是在重试的阶段，这个时间就是重试间隔，但是在非重试的初始阶段，就是linger.ms的时间（100ms），对于他的一些参数的含义就很清晰了

full，Batch是否已满，如果说Dequeue里超过一个Batch了，说明这个peekFirst返回的Batch就一定是已经满的，另外就是如果假设Dequeue里只有一个Batch，但是判断发现这个Batch达到了16kb的大小，也是已满的

expired，当前Batch已经等待的时间（120ms） >= Batch最多只能等待的时间（100ms），已经超出了linger.ms的时间范围了，否则呢，60ms < 100ms，此时就没有过期。如果linger.ms默认是0，就意味着说，只要Batch创建出来了，在这个地方一定是expired = true

sendable，综合上述所有条件来判断，这个Batch是否需要发送出去，如果Bach已满必须得发送，如果Batch没有写满但是expired也必须得发送出去，如果说Batch没有写满而且也没有expired，但是内存已经消耗完毕

如果上述条件都不满足，此时closed，当前客户端要关闭掉，此时就必须立马把内存缓冲的Batch都发送出去，就是当前强制必须把所有数据都flush出去到网络里面去，此时就必须得发送

但是如果说此时某个Batch还没有达到要发送的条件

比如说此时看到一个Partition的batch还没达到要发送的条件，batch没满，linger.ms也没到，但是linger.ms设置的是最多等待100ms，但是此时已经等待了60ms，但是剩余等待的时间40ms

40ms设置为nextReadyCheckDelayMs

接下来又有一个Partition的batch同样的情况，batch没满，linger.ms没到，此时已经等待了90ms，剩余等待的时间就是10ms

10ms会设置为nextReadyCheckDelayMs

他会算出来当前所有的Partition的Batch里，暂时不能发送的那些Batch，需要等待最少时间就能发送的那个Batch，他还需要等待的时间，就设置为nextReadyCheckDelayMs，下次再来检查是否有batch可以发送，起码要等nextReadyCheckDelayMs时间过了以后才可以

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/040\_内存缓冲中的Batch到底是如何被判定为可以发送出去的？（三）/笔记.doc**

现在10个Partition，每个Partition都有一个Dequeue，有的Dequeue里可能有多个Batch，但是这个算法一轮下来，每个Parititon只会查看他的first Batch，此时就会判断他的first Batch是否可以发送

如果这个Partiion的first Batch可以发送，此时就把这个Partition leader所在的Broker放入一个readyNodes集合里，他不是说对一个Partiton的Dequeu在这里会遍历，只看first Batch，非常的关键

假设有4个Partiton的first Batch可以发送，这4个Partiton Leader分别对应在2个Broker上，每个Broker有两个Partition Leader，此时readyNodes里就有两个Node，2个Broker会在里面

但是如果一个Partition的first Batch都不可以发送，此时会利用这个Batch来计算一下nextReadyCehckDelayMs，假设此时有6个Partitio的first Batch都不可以发送，会综合利用这个6个Partiton的firstBatch的timeToLeft（linger.ms - 已经等待的时间），取一个最小值，就代表说最快可以发送的那个batch的等待时间

下一次来检查是否有Batch可以发送起码要等待那个时间，比如说10ms

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/047\_Kafka封装的Selector是如何初始化与Broker的连接的？/笔记.doc**

Network、Selector、Channel他们是如何初始化的，kafka如何封装的，以及与原生Java NIO的Selector、Channel的关系是如何的。如果你把之前的Java NIO精讲的课程，NIO实战的课程，分布式文件系统的项目

keepalive的意思，主要是避免客户端和服务端任何一方如果断开连接之后，别人不知道，一直保持着网络连接的资源；所以设置这个之后，2小时内如果双方没有任何通信，那么发送一个探测包，根据探测包的结果保持连接、重新连接或者断开连接

观察一下，在这个工业级的网络通信框架的封装中，对底层的NIO是如何来使用的，一些参数是如何来设置的

需要去设置socket的发送和接收的缓冲区的大小，分别是128kb和32kb，这个缓冲区的大小一般都是在NIO编程里需要自己去设置的

TcpNoDelay，如果默认是设置为false的话，那么就开启Nagle算法，就是把网络通信中的一些小的数据包给收集起来，组装成一个大的数据包然后再一次性的发送出去，如果大量的小包在传递，会导致网络拥塞

如果设置为true的话，意思就是关闭Nagle，让你发送出去的数据包立马就是通过网络传输过去，所以这个参数大家也要注意下

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/054\_连接建立之后如何获取要发送到每个Broker去的所有Batch？/笔记.doc**

获取broker上所有的partition，遍历broker上的所有的partitions，对每个partition获取到dequeue里的first batch，放入待发送到broker的列表里，每个broker都有一个batches，最后有一个map，放了这些数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/049\_NIO中的SocketChannel.connect到底具备什么样的业务语义？/笔记.doc**

如果这个SocketChannel是被设置为非阻塞模式的话，那么对这个connect方法的调用，会初始化一个非阻塞的连接请求，如果这个发起的连接立马就成功了，比如说客户端跟要连接的服务端都在一台机器上

此时就会出现一个立马就连接成功的情况，然后就会返回一个true

否则只要不是那种立马可以连接成功的情况，就会返回一个false，接着就需要在后面去调用SocketChannel的finishConnect方法，去完成最终的连接

接下来我们再换一讲，连接之后是如何处理这个连接请求的缓存等待后续完成的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/038\_内存缓冲中的Batch到底是如何被判定为可以发送出去的？（一）/笔记.doc**

exhausted，内存耗尽，有人在排队等待申请内存

如果某个分区的Leader Broker还不知道是谁，此时就会设置一个标志位，后面会尝试进行元数据的拉取，但是对于我们来说，先假设他的Topic对应的元数据此时都应该已经有了，Leader Broker肯定是知道的

在往这个Dequeu里写入数据的时候，放一个一个的Batch的时候，也是会加锁的，在从Deque里读取数据的时候也是会加锁的，基于最最重量级的synchronized锁来进行的，锁死了，别人就不能操作了

如果此时判断出来这个Batch是可以发送出去的，此时就会将这个Batch对应的那个Partiton的Leader Broker给放入到一个Set里去，他在这里找的不是说找哪些Partition可以发送数据，也不是找Batch

他在这里找的是哪些Broker有数据可以发送过去，而且通过Set进行了去重，可能对于一个Broker而言，是有多个Partiton的Batch可以发送过去的

代码编写的技巧，如果你的方法要返回的是一个复杂的数据结构，此时可以定义一些Bean，里面封装你要返回的数据，哪些Broker可以发送数据过去，下一次来检查是否有Batch可以发送的时间间隔，是否有Partiton还不知道自己的Leader所在的Broke

如何判定一个Batch可以发送的？

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/056\_如何将要发送到每个Broker去的Request依托封装组件暂存起来？/笔记.doc**

接下来就是要一个一个的去发送请求了，看看依托于KafkaChannel和NIO selector多路复用的机制，是如何把这个请求给发送出去的，其实就是依托inFlightRequests去暂存了正在发送的Request

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/051\_给大家一个学习工业级NIO编程的提示：封装原生API以及缓存机制/笔记.doc**

原生NIO编程的时候，可以把他原生的 API和组件封装一下，就可以基于你自己的需求实现不同的功能了，你一定要学会进行一定的缓存机制的设计，比如说针对多个机器进行连接，那么对应的连接组件就需要进行缓存

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/046\_KafkaChannel是如何对原生Java NIO的SocketChannel进行封装的？/笔记.doc**

KafkaChannel

broker id对应一个网络连接，一个网络连接对应一个KafkaChannel，底层对应的是SocketChannel，SocketChannel对应的是最最底层的网络通信层面的一个Socket，套接字通信，Socket通信，TCP

Send，应该是说要交给这个底层的Channel发送出去的请求，可能会不断的变换的，因为发送完一个请求需要发送下一个请求

NetworkReceive，这个Channel最近一次读出来的响应，先暂存在这里，也是会不断的变换的，因为会不断的读取新的响应数据

TransportLayer是封装了底层的Java NIO的SocketChannel

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/057\_在不断轮询的万能poll方法中如何基于NIO将请求发送出去？/笔记.doc**

如果说已经发送完毕数据了，那么就可以取消对OP\_WRITE事件的关注，否则如果一个Request的数据都没发送完毕，此时还需要保持对OP\_WRITE事件的关注，而且如果发送完毕了，就会放到completedSends里面去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/053\_如何通过不断轮询的poll方法完成跟目标Broker的连接/笔记.doc**

Java NIO的Selector.select -> 他会负责去看看，注册到他这里的多个Channel，谁有响应过来可以接收，或者谁现在可以执行一个请求的发送，如果Channel可以准备执行IO读写操作，此时就把那个Channel的SelectionKey返回

接下来就会对获取到的一堆SelectionKeys进行处理，到这一步为止，我们就可以看到基于NIO来开发的很多企业级的一些功能，一个是SocketChannel如何构建，二个是一个客户端如何连接多个服务器，三个如何通过轮询调用Selector.select

select一般在这种场景里可以设置对应的超时时间，然后就可以获取到SelectionKeys

lruConnections，因为一般来说一个客户端不能放太多的Socket连接资源，否则会导致这个客户端的复杂过重，所以他需要采用lru的方式来不断的淘汰掉最近最少使用的一些连接，很多连接最近没怎么发送消息

比如说有一个连接，最近一次使用是在1个小时之前了，还有一个连接，最近一次使用是在1分钟之前，此时如果要淘汰掉一个连接，你会选择谁？LRU算法，明显是淘汰掉那个1小时之前才使用的连接

如果发现SelectionKey当前处于的状态是可以建立连接，isConnectable方法是true，接着其实就是调用到KafkaChannel最底层的SocketChannel的finishConnect方法，等待这个连接必须执行完毕

同时接下来就不要关注OP\_CONNECT事件了，对于这个Channel，接下来Selector就不要关注连接相关的事件了，也不是OP\_READ读取事件，肯定selector要关注的是OP\_WRITE事件，要针对这个连接写数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/037\_Kafka生产端唯一的一个IO线程到底在干什么？/笔记.doc**

暂时先默认，执行到这里的时候，在之前已经把Topic对应的元数据拉取到客户端来缓存了，所以说Topic -> Partitions -> Leader/Follower + ISR

（1）获取已经可以发送消息的那些Partition：哪些Partition有已经写满的batch（16kb），batch创建的时间已经超过了linger.ms，此时这个Partition就有可以发送出去的batch了，收集出来Partition Leader所在的Broker

（2）如果说有一些Partition对应的元数据都没拉取到，此时就必须标识一下，必须要在后面去尝试拉取元数据

（3）检查一下是否准备好可以向那些Borker发送数据了，就是说如果此时还没跟某个Broker建立好连接，必须在这里把长连接准备好，TCP连接，然后才可以把数据发送过去，直接就是基于最底层的NIO来开发的

（4）你有很多Partiton可以发送数据，有一些Partition Leader是在同一个Broker上，此时按照Broker对Partition进行分组，找到一个Broker对应的多个Partition的Batch，如果一个batch已经在内存缓冲里停留超过60s，超时不要了

（5）对每个Broker都创建一个ClientReqeust，包括了多个Batch，就是在这个Broker上的多个Leader Partition所对应的Batch，聚合起来组成一个ClientRequest，形成一个请求，发送到Broker上去

（6）通过NetWorkClient走底层的网络通信，把每个Broker的ClientRequest给发送过去就可以了，poll方法，他是负责实际的 进行网络IO通信操作的一个核心的方法，负责发送数据出去，也包括读取响应回来

（7）如果说请求超时了，如何判断和处理

（8）元数据加载的请求是如何通过网络通信来发送的，元数据加载的响应是如何来处理的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/044\_回头看看NetworkClient网络通信组件是如何初始化的？/笔记.doc**

Selector的组件来进行的，是怎么来的，就得先分析一下Network初始化的过程

针对多个Broker的网络连接，执行非阻塞的IO操作

NetworkClient初始化而言，大家就应该已经搞清楚了，NetworkClient主要是一个网络通信组件，底层核心的Selector负责最最核心的建立连接、发起请求、处理实际的网络IO，初始化的入口初步找到了

kafka自己封装的一些组件，但是呢，他的底层是有最最核心的Java NIO的Selector

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/045\_惊讶的发现：Kafka网络通信底层是基于原生的Java NIO开发的！/笔记.doc**

最最核心的一点，就是在KafkaSelector的底层，其实就是封装了原生的Java NIO的Selector，很关键的组件，就是一个多路复用组件，他会一个线程调用他直接监听多个网络连接的请求和响应

BIO，每个线程对一个网络连接监听他的请求和响应

maxReceiveSize，最大可以接收的数据量的大小

connectionsMaxIdle，每个网络连接最多可以空闲的时间的大小，就要回收掉

Map<String, KafkaChannel> channels，这里保存了每个broker id到Channel的映射关系，对于每个broker都有一个网络连接，每个连接在NIO的语义里，都有一个对应的SocketChannel，我们估计，KafkaChannel封装了SocketChannel

List<Send> completedSends，已经成功发送出去的请求

List<NetworkReceive> completedReceives，已经接收回来的响应而且被处理完了

Map<KafkaChannel, Dequeue<NetworkReceive>>，每个Broker的收到的但是还没有被处理的响应

conneted、disconnected、failedSends，已经成功建立连接的brokers，以及还没成功建立连接的brokers，发送请求失败的brokers

Selector内部的源码一定要带着大家深入到每个细节的研究，因为这是完全经历过全世界大量的、复杂的、大规模的场景考验的一套网络通信的框架，基于NIO封装的一套网络通信的框架

里面的涉及到的很多的细节和机制，都是代表了工业级、企业级的网络通信的设计

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/042\_如果跟Broker之间还没建立连接，如何检查是否可以建立连接？/笔记.doc**

要不然是已经建立好连接，底层的NIO Channel是ok的，inFlighRequests没有满5个，此时就可以针对这个Broker去发送一个请求过去了

但是如果上述条件不满足，假设是因为还没有建立连接，此时如何判断是否可以跟一个Broker建立连接呢？

先找到broker id对应的一个连接状态，如果此时这个连接状态是null，就说明之前从来没有建立过连接，此时就可以直接返回true，就说明可以跟这个broker建立连接；否则如果连接状态已经存在，如果当前broker的状态是断开连接，而且上一次跟这个broker尝试建立连接的时间到现在，已经超过了重试的时间了，默认100ms

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/041\_如何检查筛选出来的目标Broker可以发送数据过去？/笔记.doc**

筛选出来一些可以发送数据的Broker，现在源码就要走到，这些筛选出来的Broker到底是否可以发送数据过去呢？

当前不能处于元数据加载的过程，而且下一次要更新元数据的间隔时间为0，现在没有加载元数据，但是马上就应该要加载元数据了，如果对上述条件判断是非的话，要不然是正在加载元数据，或者是还没到加载元数据的时候

我们就认为现在还没到加载元数据的时候，就认为这个条件是false，满足了就可以了

为什么前面一定要有这个条件？假设此时必须要更新元数据了，就不能发送请求，必须要等待这个元数据被刷新了再次去发送请求

（1）有一个Broker连接状态的缓存，先查一下这个缓存，当前这个Broker是否已经建立了连接了，如果是的话，才可以继续判断其他的条件

（2）Selector，你大概可以认为底层封装的就是Java NIO的 Selector，但凡是看过我的NIO课程，跟着做NIO研发分布式文件系统，Selector上要注册很多Channel，每个Channel就代表了跟一个Broker建立的连接

（3）inFlightRequests，有一个参数可以设置这个东西，默认是对同一个Broker同一时间最多容忍5个请求发送过去但是还没有收到响应，所以如果对一个Broker已经发送了5个请求，都没收到响应，此时就不可以继续发送了

必须同时满足3个条件，才可以认为这个Broker可以发送数据过去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/052\_如果跟Broker的连接还没完成建立，那么会向他发送请求吗？/笔记.doc**

在poll方法里，会去执行跟目标broker节点完成最终的连接的建立

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/043\_深入底层网络通信的起点：通过哪个核心组件与Broker建立连接？/笔记.doc**

有一个broker的连接状态，是有一个设计模式在里面，状态机的模式，类似于我们一直说的那个状态模式，抽取和封装一个组件的多个状态，然后通过一个状态机管理组件，可以让这个状态可以互相流转

null，CONNECTING，CONNECTED，DISCONNECTED，针对不同的状态，还可以做不同的事情，如果是null就可以发起连接，如果连接成功，就可以进入已连接的状态，如果中间发生连接的故障，就进入连接失败

底层建立的都是Socket连接，发送请求也是通过底层的Socket来走的，收取数据也是通过Socket读取的，在工业级的网络通信的开发里面，两个核心的参数必须设置的，就是对于Socket的发送和接收的缓冲区

Selector的组件进行连接，如果我们学习过NIO的课程，NIO建立连接他其实就是在底层初始化一个SocketChannel发起一个连接的请求，就会把SocketChannel给注册到Selector上面去，让Selector监听他的建立连接的事件

如果Broker返回响应说可以建立连接，Selector就会告诉你，你就可以通过一个API的调用，完成底层的网络连接，TCP三次握手，双方都有一个Socket（Java，操作系统级别的概念，Socket代表了网络通信终端）

网络通信组件初始化的过程，涉及到哪些核心的网络通信组件，Selector肯定是里面非常核心的一个，NetworkClient也是一个

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/055\_针对每个目标Broker构建一个很多Batch组成的Request/笔记.doc**

发送出去的请求，需要按照kafka的二进制协议来定制数据的格式

他需要包含对应的请求头，api key，api version，acks，request timeout，接着才是请求体，里面就是包含了对应的多个batch的数据，最后的最后，一定是把刚才说的那些东西都给打成一个二进制的字节数组

ClientRequest里面就是封装了按照二进制协议的格式，放入了组装好的数据，发送到broker上去的有很多个Topic，每个Topic有很多Partition，每个Partitioin是对应就一个batch的数据发送过去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/048\_工业级组件中的NIO：KeepAlive、SocketBuffer、TcpNoDelay/笔记.doc**

NetworkClient、Selector、KafkaChannel、ConnectStates，这些东西是极为值得我们来研究的，对我们的技术底层的功底的夯实极为有好处，假设我们真的要去开发一个网络通信的程序，打算基于NIO来做

对于客户端而言，他的SocketChannel到底应该如何来设置呢？你就可以参考人家做法：KeepAlive、TcpNoDelay、SocketBuffer

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/037~057资料/050\_发起连接请求之后针对不同的情况是如何进行缓存的？/笔记.doc**

你直接初始化了一个SocketChannel然后就发起了一个连接请求，接着不管连接请求是成功还是暂时没成功，都需要把这个SocketChannel给缓存起来，接下来你才可以基于这个东西去完成连接，或者是发起读写请求

发起连接之后，直接就把这个SocketChannel给注册到Selector上去了，让Selector监视这个SocketChannel的OP\_CONNECT事件，就是是否有人同意跟他建立连接，会获取到一个SelectionKey

大概可以认为这个SelectionKey就是和SocketChannel是一一对应的

接着就是将SelectionKey、brokerid封装为了KafkaChannel，他是先把SelectionKey封装到TransportLayer里面去（SelectionKey底层是跟SocketChannel是一一对应起来），Authenticator，brokerid，直接封装一个KafkaChannel

大概可以认为是把一个核心的组件跟SelectionKey给关联起来，后续在通过SelectionKey进行网络请求和相应的处理的时候，就可以从SelectionKey里获取出来SocketChannel，可以获取出来之前attach过的一个核心组件，复制请求响应的处理

缓存起来立即建立好连接的SelectionKey

具体完成者连接需要到poll方法里才能实现，我们现在先不看

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/011\_集群元数据拉取组件的分析以及多个拉取触发时机的分析/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/001\_Kafka源码分析环境搭建：JDK、Scala以及Gradle的安装/笔记.doc**

kafka核心原理、生产部署、开发使用、线上规划，在上一个课程都已经讲完了，一般的场景使用kafka来开发都能搞定了，kafka异常报错 & 各种技术难题对应的解决方案

先阅读kafka的源码

kafka线上生产高阶故障处理、性能优化以及解决方案

我在自己本地很早都已经搭建好了kafka源码分析的环境了，但是我会把完整的步骤给出来，大家参照着就可以一步一步的把kafka源码的环境搭建起来，读源码就很方便了

在win上安装JDK 1.8，这个不用多说了

在win上安装Scala 2.10.6，上官网找到2.10.6版本对应的下载地址，kafka的服务器端的源码是scala写的，但是新版本的客户端的源码是java写的

HYPERLINK "https://www.scala-lang.org/download/2.10.6.html" ?https://www.scala-lang.org/download/2.10.6.html

然后就可以下载win上的安装包，scala.msi，下载好之后傻瓜式安装就可以了，接着必须配置SCALA\_HOME和PATH两个环境变量，首先必须得有Java和Scala两种编程语言的支持才可以

接着需要安装Gradle，现在国外很多知名的开源项目，项目构建（依赖管理、打包、远程部署、运行单元测试、静态代码检测，写好了java源代码，接着要把这个写好的代码打包存储起来，或者准备部署，项目构建），不是用maven了

都是用Gradle来进行项目的构建了，Kafka也是如此，所以需要安装Gradle来完成Kafka源码的构建，使用gradle 3.1，从官网下载gradle-3.1-bin.zip，解压缩即可，然后配置GRADLE\_HOME和PATH

就比如说你的kafka的源码写好了之后，现在要打包，打成一个压缩包提供给我们来下载，下载了之后就可以在本地安装和部署kafka了

https://gradle.org/releases/

验证三个基础的依赖都正确安装了

java -version

scala -version

gradle -version

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/007\_Kafka源码分析环境搭建：验证IntelliJ IDEA中启动的Kafka能否使用/笔记.doc**

接着就要验证一下启动的kafka能否正常使用，也就是能否正常的生产消息，接收消息，使用之前eclipse中的kafka-demo来验证即可，我们分析客户端源码的时候，其实直接在eclipse就可以，只要粘上去源码，打断点就可以开始调试了

如果是分析broker层面的一些源码，比如说请求的处理架构，磁盘读写，或者是consumer group coordinator的一些工作流程，我们在intellij idea里打断点来进行调试就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/010\_生产端的核心：KafkaProducer初始化时会涉及到哪些组件？/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/002\_Kafka源码分析环境搭建：在Windows上部署和启动ZooKeeper/笔记.doc**

要搭建分析kafka源码的环境，那么zookeeper是必须使用的，因为需要在win本地以源码的方式来启动kafka，那么kafka依赖zookeeper，所以直接用之前hadoop课程里的zk 3.4.9即可，在win上解压缩，修改zoo.cfg里的dataDir指向本地win目录

然后执行bin目录下的zkServer.cmd命令启动即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/012\_在源码中分析核心参数的含义：请求超时、缓冲大小、请求大小/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/009\_回顾一下Kafka生产端是如何进行开发的以及涉及哪些东西/笔记.doc**

既然要从kafka生产端开始研究他的源码，就得先看一下他的生产端涉及到哪些东西

KafkaProducer，里面包含了核心的资源，包括线程资源以及网络资源，他主要是通过一些线程实现了消息的异步发送，批处理机制，维护了跟 各个broker的网络连接，然后可以通过网络连接发送消息到broker去

可以整个系统全局就通过唯一的一个KafkaProducer来发送消息也是可以的，多线程并发安全的，发送的消息是通过ProducerRecord来进行封装的，代表了你要生产发送的一条消息，交给KafkaProducer来进行发送即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/004\_Kafka源码分析环境搭建：将构建好的Kafka源码导入IntelliJ IDEA中/笔记.doc**

构建好了kafka源码之后

首先你需要进入IntelliJ IDEA的这个界面：

在右下方有一个“Configure”，里面有个“Settings”，进入那个界面

左侧有一个“Plugins”，搜索scala相关的插件，此时一开始是找不到的，然后点击“search in repositories”，找到一个“Scala”插件，他的类别是“Language”，在线装即可，他会下载之后安装

安装完了在plugins里面就可以找到scala插件了，然后点击“ok”就会提示你重启intellij idea来激活安装好的插件

然后点击里面的那个Import Project按钮即可，选择你的kafka源码所在的目录，选择你构建项目的方式是“gradle”，导入的过程也需要不少的时间，需要耐心等待，会显示的是如下的图：

最后导入成功了，应该是如下图：

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/015\_数据发送线程是如何初始化以及acks参数在源码中的含义分析/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/008\_从一条消息的生产发送开始如何逐步探索Kafka运行的全流程/笔记.doc**

并不是说从broker入手来探索了，我们会从一条消息开始生产出去，首先是生产端的源码运行的流程，最核心的是两块，第一块要学习一下Kafka客户端是如何去设计一个非常优秀的生产级的保证高吞吐的一个缓冲机制

要深入的分析一下，生产端的Sender线程他的网络通信的模块，这个是我们绝对重点中的重点，必须要搞清楚kafka客户端是如何通过网络通信把一批消息发送到broker上去的，这个里面对于网络通信的很多细节，我们需要去深入的扣一下

网络通信的设置一些对应的参数，应对网络故障，人家是怎么来做的

kafka broker的集群架构的设计和实现，各个broker启动的时候，如何组成一个集群，集群的主控节点是如何选举出来的，后续如何监控集群里的各个broker是否正常运行的，或者是故障宕机，如何对故障的broker找备用的来进行替代，集群的元数据（topic、partition、leader/follower、isr）

消息会到kafka broker那块去，我们要学习很多重点的地方，首先就是网络通信里的server端的处理架构应该如何来设计，以及server端的网络通信的细节，包括里面的底层网络通信对应的一些参数的设置

深入的学习他的磁盘读写这块是如何来实现的，他的消息是如何写入磁盘的，磁盘的存储结构，怎么去使用os page cahe，怎么实现磁盘文件的顺序写，非常的优秀的，我们要来学习里面的这些东西

多副本冗余以及高可用的架构设计，leader和follower是如何同步的，副本是如何传输的，另外就是这个过程中，他的各种offset是如何变更的，如果leader所在的broker故障了，是如何进行leader和follower的切换的，高可用的架构

负载均衡以及伸缩架构，他是如何保证数据均匀的分布在集群的各个broker机器上的，负载均衡，如何进行topic的partition的扩容，让一个topic可以通过partition扩容来使用集群里更多的broker的机器资源，另外一个就是说broker扩容，如何通过加更多的broker机器来扩容集群的存储资源以及网络资源

消费端的原理，每个消费组的主控节点是如何来选择的，group coordinator如何选择，consumer group leader如何选择，分区分配的方法，分布式消费的实现机制，拉取消息的原理，offset提交的原理

kafka线上故障的处理，生产性能优化，以及各种生产问题的解决方案

kafka是一个非常难的技术，而且在使用的过程中，是经常遇到问题的，遇到问题的概率比hdfs之类的技术高很多，数据采集，flume、canal、分布式爬虫，kafka和hdfs都是比较难的技术

离线数仓，yarn、mapreduce、hive、spark，底层对应的是一些性能优化，故障处理，数据仓库的模型的设计，复杂ETL的开发和优化，数据治理

实时平台，自研一套复杂的平台系统

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/014\_底层的网络通信组件初探以及核心网络参数的分析/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/003\_Kafka源码分析环境搭建：使用Gradle来构建Kafka源码/笔记.doc**

从kafka官网下载kafka-0.10.0.1版本的源码，现在还是0.10.x版本用的人比较多，所以先分析这个版本的源码即可，其实核心的基本都是类似的，也算是比较新的版本了，下载的是kafka-0.10.0.1-src.tgz源码压缩包，解压缩

http://kafka.apache.org/downloads

通过win命令行进入kafka-0.10.0.1-src目录下，然后执行“gradle idea”为源码导入idea进行构建，如果要导入eclipse，那么就执行“gradle eclipse”，但是我们还是用idea来查看源码就可以了

这个过程会下载大量的依赖jar包，建议是可以连通国外的网，比如一些vpn，这样速度快，过程看起来如下图：

需要耐心等待

这个过程中可能会报错，比如下面的错误：

解决方案如下：

用编辑器打开build.gradle文件，加入以下内容：

ScalaCompileOptions.metaClass.daemonServer = true

ScalaCompileOptions.metaClass.fork = true

ScalaCompileOptions.metaClass.useAnt = false

ScalaCompileOptions.metaClass.useCompileDaemon = false

如下图：

最后成功了如下图：

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/013\_内存缓冲区的构建以及消息batch打包发送request的原理/笔记.doc**

我们先来看一看KafkaProducer初始化的时候会涉及到哪些内部的核心组件，默认情况下，一个jvm内部，如果你要是搞多个KafkaProducer的话，每个都默认会生成一个client.id，producer-自增长的数字，producer-1

（1）核心组件：Partitioner，后面用来决定，你发送的每条消息是路由到Topic的哪个分区里去的

（2）核心组件：Metadata，这个是对于生产端来说非常核心的一个组件，他是用来从broker集群去拉取元数据的Topics（Topic -> Partitions（Leader+Followers，ISR）），后面如果写消息到Topic，才知道这个Topic有哪些Partitions，Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据，metadata.max.age.ms，默认是5分钟，默认每隔5分钟一定会强制刷新一下

还有就是我们猜测，在发送消息的时候，如果发现你要写入的某个Topic对应的元数据不在本地，那么他是不是肯定会通过这个组件，发送请求到broker尝试拉取这个topic对应的元数据，如果你在集群里增加了一台broker，也会涉及到元数据的变化

（3）核心参数：每个请求的最大大小（1mb），缓冲区的内存大小（32mb），重试时间间隔（100ms），缓冲区填满之后的阻塞时间（60s），请求超时时间（30s）

（4）核心组件：RecordAccumulator，缓冲区，负责消息的复杂的缓冲机制，发送到每个分区的消息会被打包成batch，一个broker上的多个分区对应的多个batch会被打包成一个request，batch size（16kb）

默认情况下，如果光光是考虑batch的机制的话，那么必须要等到足够多的消息打包成一个batch，才能通过request发送到broker上去；但是有一个问题，如果你发送了一条消息，但是等了很久都没有达到一个batch大小

所以说要设置一个linger.ms，如果在指定时间范围内，都没凑出来一个batch把这条消息发送出去，那么到了这个linger.ms指定的时间，比如说5ms，如果5ms还没凑出来一个batch，那么就必须立即把这个消息发送出去

（5）核心行为：初始化的时候，直接调用Metadata组件的方法，去broker上拉取了一次集群的元数据过来，后面每隔5分钟会默认刷新一次集群元数据，但是在发送消息的时候，如果没找到某个Topic的元数据，一定也会主动去拉取一次的

（6）核心组件：网络通信的组件，NetworkClient，一个网络连接最多空闲多长时间（9分钟），每个连接最多有几个request没收到响应（5个），重试连接的时间间隔（50ms），Socket发送缓冲区大小（128kb），Socket接收缓冲区大小（32kb）

（7）核心组件：Sender线程，负责从缓冲区里获取消息发送到broker上去，request最大大小（1mb），acks（1，只要leader写入成功就认为成功），重试次数（0，无重试），请求超时的时间（30s），线程类叫做“KafkaThread”，线程名字叫做“kafka-producer-network-thread”，此处线程直接被启动

（8）核心组件：序列化组件，拦截器组件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/006\_Kafka源码分析环境搭建：直接在IntelliJ IDEA中启动Kafka/笔记.doc**

上一讲配置完毕了kafka之后，包括他的主类，配置文件，使用的是core module，等等，接着右上方会出现下图：

如果你要启动kafka，就点击这里的绿色箭头按钮即可，他就会启动kafka.Kafka这个主类，第一次启动，会重新编译整个项目，燃火才会正常启动

这里有很多的warning，不过是不要紧的，编译完成之后，kafka就会成功启动，我们来分析一下他的启动日志：

[2019-04-16 11:44:20,079] INFO KafkaConfig values:

advertised.host.name = null

metric.reporters = []

quota.producer.default = 9223372036854775807

offsets.topic.num.partitions = 50

log.flush.interval.messages = 9223372036854775807

auto.create.topics.enable = true

controller.socket.timeout.ms = 30000

log.flush.interval.ms = null

principal.builder.class = class org.apache.kafka.common.security.auth.DefaultPrincipalBuilder

replica.socket.receive.buffer.bytes = 65536

min.insync.replicas = 1

replica.fetch.wait.max.ms = 500

num.recovery.threads.per.data.dir = 1

ssl.keystore.type = JKS

sasl.mechanism.inter.broker.protocol = GSSAPI

default.replication.factor = 1

ssl.truststore.password = null

log.preallocate = false

sasl.kerberos.principal.to.local.rules = [DEFAULT]

fetch.purgatory.purge.interval.requests = 1000

ssl.endpoint.identification.algorithm = null

replica.socket.timeout.ms = 30000

message.max.bytes = 1000012

num.io.threads = 8

offsets.commit.required.acks = -1

log.flush.offset.checkpoint.interval.ms = 60000

delete.topic.enable = false

quota.window.size.seconds = 1

ssl.truststore.type = JKS

offsets.commit.timeout.ms = 5000

quota.window.num = 11

zookeeper.connect = localhost:2181

authorizer.class.name =

num.replica.fetchers = 1

log.retention.ms = null

log.roll.jitter.hours = 0

log.cleaner.enable = true

offsets.load.buffer.size = 5242880

log.cleaner.delete.retention.ms = 86400000

ssl.client.auth = none

controlled.shutdown.max.retries = 3

queued.max.requests = 500

offsets.topic.replication.factor = 3

log.cleaner.threads = 1

sasl.kerberos.service.name = null

sasl.kerberos.ticket.renew.jitter = 0.05

socket.request.max.bytes = 104857600

ssl.trustmanager.algorithm = PKIX

zookeeper.session.timeout.ms = 6000

log.retention.bytes = -1

log.message.timestamp.type = CreateTime

sasl.kerberos.min.time.before.relogin = 60000

zookeeper.set.acl = false

connections.max.idle.ms = 600000

offsets.retention.minutes = 1440

replica.fetch.backoff.ms = 1000

inter.broker.protocol.version = 0.10.0-IV1

log.retention.hours = 168

num.partitions = 1

broker.id.generation.enable = true

listeners = null

ssl.provider = null

ssl.enabled.protocols = [TLSv1.2, TLSv1.1, TLSv1]

log.roll.ms = null

log.flush.scheduler.interval.ms = 9223372036854775807

ssl.cipher.suites = null

log.index.size.max.bytes = 10485760

ssl.keymanager.algorithm = SunX509

security.inter.broker.protocol = PLAINTEXT

replica.fetch.max.bytes = 1048576

advertised.port = null

log.cleaner.dedupe.buffer.size = 134217728

replica.high.watermark.checkpoint.interval.ms = 5000

log.cleaner.io.buffer.size = 524288

sasl.kerberos.ticket.renew.window.factor = 0.8

zookeeper.connection.timeout.ms = 6000

controlled.shutdown.retry.backoff.ms = 5000

log.roll.hours = 168

log.cleanup.policy = delete

host.name =

log.roll.jitter.ms = null

max.connections.per.ip = 2147483647

offsets.topic.segment.bytes = 104857600

background.threads = 10

quota.consumer.default = 9223372036854775807

request.timeout.ms = 30000

log.message.format.version = 0.10.0-IV1

log.index.interval.bytes = 4096

log.dir = /tmp/kafka-logs

log.segment.bytes = 1073741824

log.cleaner.backoff.ms = 15000

offset.metadata.max.bytes = 4096

ssl.truststore.location = null

group.max.session.timeout.ms = 300000

ssl.keystore.password = null

zookeeper.sync.time.ms = 2000

port = 9092

log.retention.minutes = null

log.segment.delete.delay.ms = 60000

log.dirs = F:\development\kafka\logs

controlled.shutdown.enable = true

compression.type = producer

max.connections.per.ip.overrides =

log.message.timestamp.difference.max.ms = 9223372036854775807

sasl.kerberos.kinit.cmd = /usr/bin/kinit

log.cleaner.io.max.bytes.per.second = 1.7976931348623157E308

auto.leader.rebalance.enable = true

leader.imbalance.check.interval.seconds = 300

log.cleaner.min.cleanable.ratio = 0.5

replica.lag.time.max.ms = 10000

num.network.threads = 3

ssl.key.password = null

reserved.broker.max.id = 1000

metrics.num.samples = 2

socket.send.buffer.bytes = 102400

ssl.protocol = TLS

socket.receive.buffer.bytes = 102400

ssl.keystore.location = null

replica.fetch.min.bytes = 1

broker.rack = null

unclean.leader.election.enable = true

sasl.enabled.mechanisms = [GSSAPI]

group.min.session.timeout.ms = 6000

log.cleaner.io.buffer.load.factor = 0.9

offsets.retention.check.interval.ms = 600000

producer.purgatory.purge.interval.requests = 1000

metrics.sample.window.ms = 30000

broker.id = 0

offsets.topic.compression.codec = 0

log.retention.check.interval.ms = 300000

advertised.listeners = null

leader.imbalance.per.broker.percentage = 10

(kafka.server.KafkaConfig)

[2019-04-16 11:44:20,251] INFO starting (kafka.server.KafkaServer)

[2019-04-16 11:44:20,283] INFO Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)

[2019-04-16 11:44:20,423] INFO Starting ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)

[2019-04-16 11:44:29,485] INFO Client environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:host.name=DESKTOP-FV82PGK (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:java.version=1.8.0\_151 (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:java.vendor=Oracle Corporation (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:java.home=F:\development\Java\jdk1.8.0\_151\jre (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:java.class.path=F:\development\Java\jdk1.8.0\_151\jre\lib\charsets.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\deploy.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\access-bridge-64.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\cldrdata.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\dnsns.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\jaccess.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\jfxrt.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\localedata.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\nashorn.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\sunec.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\sunjce\_provider.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\sunmscapi.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\sunpkcs11.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\ext\zipfs.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\javaws.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\jce.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\jfr.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\jfxswt.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\jsse.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\management-agent.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\plugin.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\resources.jar;F:\development\Java\jdk1.8.0\_151\jre\lib\rt.jar;F:\development\kafka\kafka-0.10.0.1-src\core\out\production\classes;F:\development\kafka\kafka-0.10.0.1-src\clients\out\production\classes;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\net.sf.jopt-simple\jopt-simple\4.9\ee9e9eaa0a35360dcfeac129ff4923215fd65904\jopt-simple-4.9.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\com.yammer.metrics\metrics-core\2.2.0\f82c035cfa786d3cbec362c38c22a5f5b1bc8724\metrics-core-2.2.0.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\org.scala-lang\scala-library\2.10.6\421989aa8f95a05a4f894630aad96b8c7b828732\scala-library-2.10.6.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\org.slf4j\slf4j-log4j12\1.7.21\7238b064d1aba20da2ac03217d700d91e02460fa\slf4j-log4j12-1.7.21.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\com.101tec\zkclient\0.8\c0f700a4a3b386279d7d8dd164edecbe836cbfdb\zkclient-0.8.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\org.apache.zookeeper\zookeeper\3.4.6\1b2502e29da1ebaade2357cd1de35a855fa3755\zookeeper-3.4.6.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\net.jpountz.lz4\lz4\1.3.0\c708bb2590c0652a642236ef45d9f99ff842a2ce\lz4-1.3.0.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\org.xerial.snappy\snappy-java\1.1.2.6\48d92871ca286a47f230feb375f0bbffa83b85f6\snappy-java-1.1.2.6.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\org.slf4j\slf4j-api\1.7.21\139535a69a4239db087de9bab0bee568bf8e0b70\slf4j-api-1.7.21.jar;C:\Users\lixue\.gradle\caches\modules-2\files-2.1\log4j\log4j\1.2.17\5af35056b4d257e4b64b9e8069c0746e8b08629f\log4j-1.2.17.jar;F:\development\JetBrains\IntelliJ IDEA Community Edition 2018.1.4\lib\idea\_rt.jar (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,485] INFO Client environment:java.library.path=F:\development\Java\jdk1.8.0\_151\bin;C:\WINDOWS\Sun\Java\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files (x86)\Intel\iCLS Client\;C:\ProgramData\Oracle\Java\javapath;C:\Program Files\Intel\iCLS Client\;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files\MySQL\MySQL Utilities 1.6\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files\Intel\WiFi\bin\;C:\Program Files\Common Files\Intel\WirelessCommon\;F:\development\Git\cmd;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\IPT;C:\Program Files\Intel\Intel(R) Management Engine Components\IPT;C:\Users\lixue\AppData\Local\Microsoft\WindowsApps;F:\development\Java\jdk1.8.0\_151\bin;F:\development\Maven\apache-maven-3.5.2\bin;F:\development\gradle\gradle-2.10\bin;F:\development\hadoop-source-code;;. (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:java.io.tmpdir=C:\Users\lixue\AppData\Local\Temp\ (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:java.compiler=<NA> (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:os.name=Windows 10 (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:os.arch=amd64 (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:os.version=10.0 (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:user.name=zhonghuashishan (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:user.home=C:\Users\lixue (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,490] INFO Client environment:user.dir=F:\development\kafka\kafka-0.10.0.1-src (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:29,492] INFO Initiating client connection, connectString=localhost:2181 sessionTimeout=6000 watcher=org.I0Itec.zkclient.ZkClient@25359ed8 (org.apache.zookeeper.ZooKeeper)

[2019-04-16 11:44:30,546] INFO Waiting for keeper state SyncConnected (org.I0Itec.zkclient.ZkClient)

[2019-04-16 11:44:30,551] INFO Opening socket connection to server 0:0:0:0:0:0:0:1/0:0:0:0:0:0:0:1:2181. Will not attempt to authenticate using SASL (unknown error) (org.apache.zookeeper.ClientCnxn)

[2019-04-16 11:44:30,553] INFO Socket connection established to 0:0:0:0:0:0:0:1/0:0:0:0:0:0:0:1:2181, initiating session (org.apache.zookeeper.ClientCnxn)

[2019-04-16 11:44:30,722] INFO Session establishment complete on server 0:0:0:0:0:0:0:1/0:0:0:0:0:0:0:1:2181, sessionid = 0x16a23e965e60000, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)

[2019-04-16 11:44:30,724] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)

[2019-04-16 11:44:31,319] INFO Log directory 'F:\development\kafka\logs' not found, creating it. (kafka.log.LogManager)

[2019-04-16 11:44:31,338] INFO Loading logs. (kafka.log.LogManager)

[2019-04-16 11:44:31,357] INFO Logs loading complete. (kafka.log.LogManager)

[2019-04-16 11:44:31,482] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogManager)

[2019-04-16 11:44:31,487] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)

[2019-04-16 11:44:31,499] WARN No meta.properties file under dir F:\development\kafka\logs\meta.properties (kafka.server.BrokerMetadataCheckpoint)

[2019-04-16 11:44:31,637] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)

[2019-04-16 11:44:31,647] INFO [Socket Server on Broker 0], Started 1 acceptor threads (kafka.network.SocketServer)

[2019-04-16 11:44:31,696] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)

[2019-04-16 11:44:31,701] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)

[2019-04-16 11:44:31,803] INFO Creating /controller (is it secure? false) (kafka.utils.ZKCheckedEphemeral)

[2019-04-16 11:44:31,940] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)

[2019-04-16 11:44:31,942] INFO 0 successfully elected as leader (kafka.server.ZookeeperLeaderElector)

[2019-04-16 11:44:32,158] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)

[2019-04-16 11:44:32,163] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)

[2019-04-16 11:44:32,205] INFO [GroupCoordinator 0]: Starting up. (kafka.coordinator.GroupCoordinator)

[2019-04-16 11:44:32,206] INFO [GroupCoordinator 0]: Startup complete. (kafka.coordinator.GroupCoordinator)

[2019-04-16 11:44:32,221] INFO [Group Metadata Manager on Broker 0]: Removed 0 expired offsets in 18 milliseconds. (kafka.coordinator.GroupMetadataManager)

[2019-04-16 11:44:32,254] INFO [ThrottledRequestReaper-Produce], Starting (kafka.server.ClientQuotaManager$ThrottledRequestReaper)

[2019-04-16 11:44:32,258] INFO [ThrottledRequestReaper-Fetch], Starting (kafka.server.ClientQuotaManager$ThrottledRequestReaper)

[2019-04-16 11:44:32,272] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)

[2019-04-16 11:44:32,361] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)

[2019-04-16 11:44:50,339] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)

[2019-04-16 11:44:50,484] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)

[2019-04-16 11:44:59,494] INFO Registered broker 0 at path /brokers/ids/0 with addresses: PLAINTEXT -> EndPoint(DESKTOP-FV82PGK,9092,PLAINTEXT) (kafka.utils.ZkUtils)

[2019-04-16 11:44:59,496] WARN No meta.properties file under dir F:\development\kafka\logs\meta.properties (kafka.server.BrokerMetadataCheckpoint)

[2019-04-16 11:44:59,561] WARN Error while loading kafka-version.properties :null (org.apache.kafka.common.utils.AppInfoParser)

[2019-04-16 11:44:59,563] INFO Kafka version : unknown (org.apache.kafka.common.utils.AppInfoParser)

[2019-04-16 11:44:59,563] INFO Kafka commitId : unknown (org.apache.kafka.common.utils.AppInfoParser)

[2019-04-16 11:44:59,565] INFO [Kafka Server 0], started (kafka.server.KafkaServer)

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/001~015资料/005\_Kafka源码分析环境搭建：对IntelliJ IDEA中的Kafka进行正确配置/笔记.doc**

我们肯定是要看到log4j输出的日志的，所以必须把config目录下的log4j.properties给放到src/main/scala目录下去，这样才能看到服务端运行起来的程序打印出来的日志，另外 需要修改 config目录下的server.properties

这个文件中只有一个配置项是必须修改的，那就是log.dirs，这个是配置kafka的日志存储目录的，可以配置成win下的一个目录即可，别的不用修改，zk配置默认连接本机的2181端口的

如果我们要在IntelliJ IDEA里启动kafka，通过源码的方式来启动

此外，Kafka的启动类是“kafka.Kafka”，他是要读取“server.properties”文件的，必须给他指定这个文件的所在位置才可以，在上方的菜单栏里，有一个“Run”菜单，点击后，里面有一个“Edit Configuration”菜单，点击这个

出现上图之后，选择“+”号，然后选择“Application”，“Name”输入为“Kafka”，“Main Class”输入为“kafka.Kafka”，“Program arguments”输入为“config/server.properties”，“use classpath of module”输入为“core\_main”

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/030\_为什么要在内存缓冲写入算法中引入double-check模式？/笔记.doc**

tryAppend方法，其实是把消息尝试写入Dequeu的最近一个batch中，但是如果Dequeu是空的，这个方法会失败，并发课里讲解过，其实可能会有多个线程并发的执行，多个线程都可能分别拿到一个16kb的ByteBuffer

3个线程，线程1，线程2，线程3，这3个线程都会获取到一个16kb的ByteBuffer内存

假设线程2进入了synchronized代码块里面去，基于16kb的ByteBuffer构造一个batch，放入Dequeue中，就成功了

接着线程3进入了synchronized代码块里面去，直接把消息放入Dequeu中已有的一个batch里去，那么他手上的一个16kb的ByteBuffer怎么办？在这里就会把这个16kb的ByteBuffer给放入到BufferPool的池子里去，保证内存可以复用

万一说有人的消息是52kb，超出了16kb，分配的那个ByteBuffer就会是52kb，如果对52kb的ByteBuffer进行处理，他会直接释放掉这块内存，不去管他，让gc掉，avaialbeMemory给加回去

avaiableMemory = 32mb - 16kb \* 3

free（Dequeue） = ByteBuffer（16kb）-> ByteBuffer（16kb）

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/018\_初步窥探客户端发送消息时源码运行的大致流程/笔记.doc**

我们先大致来看一下KafkaProducer.send()方法发送消息的时候，他源码里大致的运行的流程，先来窥探一下

（1）回调自定义的拦截器

（2）同步阻塞等待获取topic元数据

如果你要往一个topic里发送消息，必须是得有这个topic的元数据的，你必须要知道这个topic有哪些分区，然后根据Partitioner组件去选择一个分区，然后知道这个分区对应的leader所在的broker，才能跟那个broker建立连接，发送消息

调用同步阻塞的方法，去等待先得获取到那个topic对应的元数据，如果此时客户端还没缓存那个topic的元数据，那么一定会发送网络请求到broker去拉取那个topic的元数据过来，但是下一次就可以直接根据缓存好的元数据来发送了

（3）序列化key和value

你的key和value可以是各种各样的类型，比如说String、Double、Boolean，或者是自定义的对象，但是如果要发送消息到broker，必须对这个key和value进行序列化，把那些类型的数据转换成byte[]字节数组的形式

基于获取到的topic元数据，使用Partitioner组件获取消息对应的分区

（5）检查要发送的这条消息是否超出了请求最大大小，以及内存缓冲最大大小

（8）设置好自定义的callback回调函数以及对应的interceptor拦截器的回调函数

（7）将消息添加到内存缓冲里去，RecordAccumulator组件负责的

（8）如果某个分区对应的batch填满了，或者是新创建了一个batch，此时就会唤醒Sender线程，让他来进行工作，负责发送batch

学习提示：

默认大家无论是搞java的，还是大数据的同学，来看这个课程的话，都默认几点，首先一定不建议大家跳看，Java大量的基础技术，分布式、JDK、并发、网络，Spring Cloud源码（分布式服务框架的源码），等等，有了一定的技术的基础

再来看这个kafka源码，会非常的关注很多的细节，包括了架构设计，运行流程，核心机制的设计，编码的细节，数据结构设计的细节，并发技术的运用，网络通信的细节，磁盘读写的细节，异常处理的细节

可以学习到非常多的东西

大家就可以学到一点，16讲大家可以学到一点，就是刚开始他没有去拉取集群的元数据，而是在后面根据你发送消息时候的需要，要给哪个topic发送消息，再去拉取那个topic对应的元数据，这就是懒加载的设计思想，按需加载思想

一个高并发、高吞吐、高性能的消息系统的客户端的设计，他的核心流程是如何来搭建和设计的，可扩展，通过拦截器的模式预留一些扩展点给其他人来扩展，在消息发送之前或者发送之后，都可以进行自定义的扩展

设计一个通用框架的时候，必须得有一个序列化的过程，因为key和value可能是各种各样的类型，但是必须要保证把key和value都转换成通用的byte[]字节数组的格式，才可以来进行跟broker的通信

基于一个独立封装的组件来进行分区的选择和路由，可以用默认的，也可以用自定义的分区器，留下给用户自己扩展的空间

对消息的大小，是否超出请求的最大大小，是否会填满 内存缓冲导致内存溢出，对一些核心的请求数据必然要进行严格的检查

异步发送请求，通过先进入内存缓冲，同时设置一个callback回调函数的思路，在发送完成之后来回调你的函数通知你消息发送的结果，异步运行的后台线程配合起来使用，基于异步线程来发送消息

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/017\_分析一下Kafka集群元数据在客户端缓存采用的数据结构/笔记.doc**

KafkaProducer在初始化的时候是不会去拉取集群的元数据的，做了一个最最基本的初始化，也就是仅仅把我们配置的那个broker的地址放了进去，在客户端缓存集群元数据的时候，采用了哪些数据结构

List<Node>，Kafka Broker节点，一台机器

unautorhizedTopics，没有被授权访问的Topic的列表，就是kafka是可以支持权限控制的，如果你的客户端没有被授权访问某个Topic，那么就会放在这个列表里

Map<TopicParittion, PartitionInfo>，TopicPartition就代表了一个分区，里面就是他的topic的名字，以及他在topic里的分区号；PartitioinInfo，就代表了分区的详细信息，属于哪个topic，分区号，每个分区都有多个副本，Leader在哪个broker上，followers在哪些broker上，ISR列表，都在里面

partitionsByTopic，每个topic有哪些分区

availablePartitionsByTopic，每个topic有哪些当前可用的分区，如果某个分区没有leader是存活的，此时那个分区就不可用了

partitionsByNode，每个broker上放了哪些分区

nodesById，broker.id -> Node

对集群元数据的客户端缓存，如何根据不同的需求、使用和场景，采用不同的数据结构来进行存放，是我们需要跟kafka客户端的源码设计学习的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/031\_如何基于申请的ByteBuffer构造Batch并放入队列？/笔记.doc**

compressor完全就是可以按照最终的一条数据的格式来进行写，他在写的时候这个里面的逻辑还是有点点复杂的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/029\_如果内存空间充足，那么如何基于NIO ByteBuffer分配内存？/笔记.doc**

一个batch就对应了一块内存空间，这里要放一堆的消息，batchSize默认的大小是16kb，如果你的消息最大的值是1mb，如果说你的消息大于了16kb的话，就会使用你的消息的大小来分配一块内存空间

否则如果你的消息是小于16kb的话，那么就会基于16kb来分配内存空间

你在实际生产环境，request.max.size，batch.size是必须要调优的，你必须要根据自己实际发送的消息的大小来设置request.max.size和batch.size，如果你的消息频繁的是超过了batch.sizse的话

一个batch就一条消息，batch打包的机制还有意义吗？每条消息都对应一次网络请求

ReentrantLock他比较好的地方就是可以通过API灵活的控制加锁和释放锁，在这里，BufferPool这里是需要这样灵活的加锁和释放锁的，synchronized效果是一样，代码块的范围来加锁和释放锁

进入synchronized代码块就加锁，出这个代码块就释放锁

BufferPool里是有一个Deque作为队列，缓存了一些ByteBuffer，也就是缓存了一批内存空间，可以用来复用的，就是说他会缓存一批ByteBuffer，每个ByteBuffer都是16kb，默认的batch大小

Dequeu里的ByteBuffer的数量 \* 16kb = 已经缓存的内存空间的大小，0

availableMemory就是剩余的还可以使用的内存空间的大小，32mb，此时需要使用掉一块内存空间，减去batchSize的大小，32mb - 16kb，接下来就直接返回ByteBuffer分配出来的一块16kb大小的内存空间

如果当前内存空间还可以分配新的ByteBuffer，那么就是上述的运行逻辑

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/019\_从未有过的细致源码研究：工业级的客户端如何进行异常处理？/笔记.doc**

这一讲是从来没有过的细致的源码级别的研究，在KafkaProduer.doSend()，最最核心的用于发送消息到broker的方法，他对应了N多个异常处理的try catch语句，异常处理到底应该怎么来做？

（1）自定义异常体系

有一个比较大的标准，就是底层的模块一般来说都应该要把自己的网络异常、IO异常，往上抛出来，在最最核心的上层的流程控制的逻辑里来捕获所有的异常，这样可以让你在核心流程的运行的过程中，根据异常来进行对应的处理

在底层的一些核心代码的编写过程中，你需要在编写代码的时候就考虑到，比如说你的负责网络通信或者磁盘读写，或者是干别的一些事情的代码，有没有可能在运行的过程中碰到一些异常，此时就需要自定义一套异常体系

必然会涉及到自己去自定义一套异常，在你写代码的时候如果感觉在某个地方运行的时候，可能会遇到某种异常，就需要在一个地方，try catch之后，抛出来一个自定义的异常，打印出来自定义异常的信息

严格、成熟的系统里，一定是有一套异常体系，就是针对你自己脑子里大概能思考到的系统可坑出现的一些问题，定义好一些预定义的异常类，主要看到一个异常类就知道在代码的哪个地方发生了大概什么样的异常，写好异常信息

各种底层处理过程中的异常都可以往上抛到核心流程控制的逻辑里去

（2）底层模块把自定义异常往上抛

（3）核心控制流程中对各种异常进行处理

处理技巧1：直接抛出异常到最上层的调用的地方去

处理技巧2：就是封装成一种错误的状态码，400，返回给最上层的调用者

处理技巧3：对异常的情况做一些统计

处理技巧4：在特殊的异常的记录类里，记录下来发生的异常

针对Java和中间件/大数据两种不同的系统，来说一下异常抛到最上层一般是怎么处理的，针对Java开发的一些业务系统，就是最上层，比如spring mvc的controller里最上层的业务逻辑控制的地方，发现了异常之后，一般来说就是封装一个有异常状态码和异常提示消息的响应对象

Result {

int status;

String message;

String exceptionType;

}

如果是中间件系统，或者是大数据系统，一般来说就是直接把异常抛到最上层你调用他的地方，如果是同步调用就是方法里异常直接抛出来，你要感知到这个异常；如果是异步调用，回调你的回调函数，告诉你异常信息

特别是如果是中间件系统，或者是大数据系统的话，可能会因为最上层获取到异常之后，就会导致中间件挂掉，停止运行，或者大数据系统挂掉，停止运行

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/034\_如果一个Batch被写满了，如何申请内存块构建下一个Batch？/笔记.doc**

假设一个batch被写满了之后，如何申请下一个ByteBuffer开辟下一块batch，如果你已经写入的数量已经大于了writeLimit，16kb，如果已经写入了15.9kb，15.9kb + 1条消息的大小 > 16kb，说明连1条消息都不能写了，此时batch已经满了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/022\_是否有必要现在就对底层的网络通信组件深入分析？/笔记.doc**

底层的网络通信组件，请求是如何发送的，响应是如何接收的，这些东西都对应了一套kafka底层的网络通信的一套东西，如果我们现在去琢磨和研究他的话，其实是不靠谱的，非常的复杂

没有必要去死扣底层的细节，只要知道在发送消息的时候，如果一个topic的元数据么有，此时会发送请求去broker拉取元数据以及缓存在客户端即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/028\_对队列加锁之后尝试将消息放入队列已有的batch中/笔记.doc**

CopyOnWriteMap

就是尝试将消息写入队列最近一个batch中，但是实际上，我们现在的Deque是空的，里面是没有batch的，在队列中的batch为空的情况下，源码是如何运行的，如果batch是存在的，就会将消息放入队列最近一个batch中

这个方法就可以返回了，你要实现的就是这么个效果

但是如果队列是空的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/024\_如何根据分区key将消息路由到同一个分区中去？/笔记.doc**

假设指定了分区key

此时，kafka会通过自己的工具类，murmur2，实现一个算法，将一个字节数组转换为一个hash值，你的分区key，比如说订单id就会被通过murmur2算法转换为一个int类型的hash值

只要你的分区key是一样的，比如说是同样的订单id，此时就一定会生成相同的hash值，接下来就用hash值对分区数量进行取模，就可以保证说只要是分区key相同，hash值一定也相同，路由到的分区一定是相同的

对于如果你要保证说发送出去的消息按照一定的规律严格是有序的，比如说mysql binlog就一定要严格按照这个模式来发送，就一定是要按照数据库里的表主键id来作为分区key进行发送

同一条数据的增删改的binlog都是进入到同一个分区的，才能拿到正确的顺序

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/023\_如果没有指定分区key是如何对消息负载均衡分发到分区的？/笔记.doc**

既然已经有了元数据了，接下来就可以进行分区路由了，把每个消息要路由到某一个分区去，默认有一个DefaultPartitioner，这个里面他实际上是会去负责默认的分区路由的策略，支持三种，指定分区，指定分区key，或者不指定分区key

AtomicInteger，初始值是一个随机的integer类型的数字，接下来默认是递增的，一定会保证是一个正整数，就是比如说topic有5个分区，就会对这个递增的数字（23），对topic的分区数量进行取模

就是根据这个递增的数字23，路由到5个分区中的1个，3

接下来一条消息，就会递增counter，比如说就是24，4；25 -> 0；26 -> 1；27 -> 2；28 -> 3；29 -> 4；30 -> 0，所以说采用了一个counter递增的方式，不断的用一个递增的数字来对分区的数量进行取模

保证在不指定分区key的情况下，所有的消息会均匀的分发到各个分区中去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/025\_快速过一下在将消息发送到内存缓冲区之前做的准备工作/笔记.doc**

他会按照他接下来要按照自己的二进制协议拼接出来这个发送消息的请求

4个字节：crc

1个字节：magic

1个字节：attribute

8个字节：timestamp

4个字节：key size

真正你的key是多少个字节：是通过之前的字节数组的长度推算出来的

4个字节：value size

真正你的value是多少个字节：是通过之前的字节数组的长度推算出来的，你的那个value的值转换成字节数组之后有多少个字节，就会显示在这里

一条消息对应到broker那儿是一条log，日志，会写入日志文件的，每条日志开头都有一个size，代表了这个日志的大小，4个字节；还会有offset，代表这个消息在日志文件里的offset偏移量，8个字节

检查，不能超过请求大小（1mb），也不能超过内存缓冲大小（32mb）

准备好时间戳、回调函数

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/032\_一条消息是如何按照二进制协议写入Batch的ByteBuffer的？/笔记.doc**

一条消息是如何按照二进制协议的规范写入到底层的ByteBuffer里去的

offset | size | crc | magic | attibutes | timestamp | key size | key | value size | value

是严格的按照二进制协议的规范，他规范里规定了，就是先是几个字节的offset，然后是几个字节的size，然后是几个字节的crc，接着是几个字节的magic，以此类推，他就是完全按照规范来写入ByteBuffer里去的

可以看到他最最底层的写入ByteBuffer的IO流的方式

ByteBufferOutputStream包裹了ByteBuffer，持有一个针对ByteBuffer的输出流，接着会把ByteBufferOutputStream给包裹在一个压缩流里，gzip、lz4、snappy，如果是包裹在压缩流里，写入的时候会先进入压缩流的缓冲区

压缩流会把一条消息放在缓冲区里，用压缩算法给压缩了，再写入底层的ByteBufferOutputStream里去

如果是非压缩的模式，最最普通的情况下，就是DataOutputStream包裹了ByteBufferOutputSteram，然后写入数据，Long、Byte、String，都会在底层转换为字节进入到ByteBuffer里去

完全就清楚了，一条消息是如何写入底层的ByteBuffer中的，值得学习的

内存空间管理的方式，包括他有内存缓冲的核心数据结构，内存缓冲池，ByteBuffer，如何通过IO流将数据写入ByteBuffer的，如何按照二进制协议规范来写一条消息的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/033\_频繁写入的消息是如何直接进入已有的分区batch中的？/笔记.doc**

如果已经有了batch之后，就会不停的往这个batch里去写

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/026\_大致浏览一下源码中将消息写入内存缓冲的运行流程/笔记.doc**

如何将消息写入内存缓冲里面，先大致浏览一下里面的流程，然后再逐个击破一点一点的去看他，这个里面一定要关注的一点就是说，kafka客户端设计是如何管理自己的内存的，如何基于内存里的数据结构构造一个缓冲区

如何基于缓冲区去承载写入进去的消息，以及batch批处理的机制，消息聚合成batch的机制，整个这套机制是如何来实现的

KafkaProducer设计的理念就是多线程并发安全的，可以让多个线程并发的来调用KafkaProducer还保证数据不会错乱的，所以说是可能会有多个线程并发的来调用他的send()方法的

他会从内存缓冲区里获取一个分区对应的Deque，这个Deque里是一个队列，放了很多的Batch，就是这个分区对应的多个batch，CopyOnWrite这个东西，我们在并发课程里，讲解过CopyOnWriteArrayList

就是说，适合的是读多写少的场景，每次更新的时候，都是copy一个副本，在副本里来更新，接着更新整个副本，好处就在于说写和读的操作互相之间不会有长时间的锁互斥，写的时候不会阻塞读

坏处在于说对内存的占用是很大的，适合的是读多写少的场景，大量读的场景就直接基于快照副本来进行读取的，CoypOnWriteMap也是类似的思路，一个分区创建一个Deque，其实是频次很低的写行为

大量的主要还是在读取，就是去大量的从map里读取一个分区对应的Deque，最后高并发频繁更新的就是分区对应的那个Deque，读的时候基于快照来读即可，所以这种场景非常适合使用CopyOnWrite系列的数据结构

如果说此时还没有创建对应的batch，此时会导致放入Deque会失败

他会基于BufferPool给这个batch分配一块内存出来，之所以说是Pool，就是因为这个batch代表的内存空间是可以复用的，用完一块内存之后会放回去下次给别人来使用，复用内存，避免了频繁的使用内存，丢弃对象，垃圾回收

已经可以往Deque队列里写入消息了，已经有一个新分配的batch了（对应了BufferPool分配的一块内存空间）

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/021\_为了搞清楚元数据加载的过程，回头看看Sender线程的初始化/笔记.doc**

Sender是如何拉取元数据的，起码先得看看Sender他是如何初始化的

public class KafkaThread extends Thread {

private final Logger log = LoggerFactory.getLogger(getClass());

public KafkaThread(final String name, Runnable runnable, boolean daemon) {

super(runnable, name);

setDaemon(daemon);

setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {

public void uncaughtException(Thread t, Throwable e) {

log.error("Uncaught exception in " + name + ": ", e);

}

});

}

}

在设计一些后台线程的时候，可以参照这种模式，把线程以及线程执行的逻辑给切分开来，Sender就是Runnable线程执行的逻辑，KafkaThread其实代表了这个线程本身，线程的名字，未捕获异常的处理，daemon线程的设置

后台线程和网络通信的组件要切分开来，线程负责业务逻辑，网络通信组件就专门进行网络请求和响应，封装NIO之类的东西

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/036\_不断申请内存空间的情况下导致可用内存耗尽了怎么办？/笔记.doc**

availableMemory = 32mb - Dequeu<ByteBuffer> - batch

实际上当前可用的内存应该是 = availableMemory + Dequeu<ByteBuffer> = 32mb - batch

此时已经申请不了新的ByteBuffer去开辟一个Batch了，会怎么样呢？会阻塞一段时间，maxBlockMs - 可能获取元数据耗费的时间，如果还是不行的话，就会抛异常了，但是这段时间里有可用内存腾出来了

有一些batch被发送出去了，获取到了响应，此时就可以释放那个batch底层对应的ByteBuffer，就会被放回到BufferPool里面去，此时就可以唤醒阻塞的线程，再次申请一个新的ByteBuffer构造一个Batch

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/035\_如何基于缓冲池中的ByteBuffer来复用内存空间？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/016\_KafkaProducer初始化的时候到底会不会去拉取集群元数据？/笔记.doc**

分析完毕了KafkaProducer初始化的时候，都涉及到了哪些组件，每个组件对应的一些核心参数在源码里的注释和默认值，他的作用和效果，都进行了分析，到底会不会真实的去拉取集群的元数据呢？

wait()，释放锁，然后进入一个休眠等待再次被人唤醒获取锁的状态

此时如果有人获取锁之后，调用notifyAll()，就会把之前调用wait()方法进入休眠的线程给唤醒，让他们再次尝试获取锁

在KafkaProducer初始化的时候，并没有真正的去某一个broker上去拉取元数据的，但是他肯定是对集群元数据做了一个初始化的，把你配置的那些broker地址转化为了Node，放在Cluster对象实例里

互联网教学里，学英语，背单词，阅读，碎片化，少儿英语启蒙

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/020\_如何对topic元数据进行细粒度的按需加载以及同步等待？/笔记.doc**

首先确保说topic的元数据是可以使用的

如果之前从来没有加载过topic的元数据，就会在这一步同步阻塞来等待就是说人家可以去连接到broker拉取元数据过来

maxBlockTimeMs，决定了你调用send()方法的时候，最多会被阻塞多长时间，所以这个方法决定了你的send在一些异常的情况下，比如说拉取topic的元数据，结果跟broker网络有问题，在一段时间后还是拉取不到

在你把数据放到内存缓冲的时候，如果内存缓冲满了，此时最多就只能阻塞这么长时间就必须返回了，如果你希望send()方法被阻塞的时间可以延长或者缩减，此时你可以自己去动手配置这个参数

在客户端的方法尝试等待获取topic元数据的过程中，核心的逻辑，就是说先必须唤醒Sender线程，然后呢就会通过一个while循环，直接去wait释放锁，尝试最多就是等待默认的60s的时间

topic元数据的拉取，是走的是异步的方式的，但是对异步的结果进行同步的阻塞的等待，他其实唤醒Sender线程，就是本质上就是在让那个Sender线程去从broker拉取对应的topic的元数据

如果拉取成功了，那么version版本号，集群元数据的版本号一定会累加，所以只要判断version版本号还没有累加，就说明此时Sender县城关还没有成功的拉取元数据，此时就是在主线程里，就是要wait阻塞等待最多60s即可

接下来肯定是分为两种情况：

（1）Sender线程成功的在60s内把topic元数据加载到了，然后缓存到了Metadata里去，更新了version版本号，而且此时一定会尝试把wait阻塞等待的主线程给唤醒，让主线程直接返回阻塞等待的时长

（2）如果wait(60s)一直超时了，你的Sender线程都没加载成功元数据，此时人家在60s后自动醒来了，此时会直接超时抛异常

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/016~036资料/027\_如何基于CopyOnWriteMap实现线程安全的分区队列构建/笔记.doc**

他内存中就是一个最最普通的，非线程安全的Map数据结构，但是他把这个数据结构定义为volatile类型，就可以保证可见性，就是只要有人更新了这个引用变量对应的实际的map对象的地址，就可以立即看到

读的时候是完全不用加锁的，多个线程并发进来，高并发的执行读的操作，在这里完全是没有任何的互相之间的影响的，可以实现高并发的读，没有锁在这里。如果队列已经存在了，直接返回即可

多个线程会并发的执行putIfAbsent方法，在这个方法里可以保证线程安全的，除非队列不存在才会设置进去，在put方法的时候是有synchronized，可以保证同一时间只有一个线程会来更新这个值

为什么说写数据的时候不会阻塞读的操作，针对副本进行kv设置，把副本通过volatile写的方式赋值给对应的变量，并发之类的课要学精，否则学大数据无从谈起，跳看，直接来看kafka源码，一定是搞不定的

保证多线程并发安全的，KafkaProducer最最核心的，会出现多线程并发访问的，就是内存缓冲区，是最核心的一块数据结果，专门放你缓冲的消息的，对这块东西，他的并发这块的使用也是非常好的

CopyOnWriteMap，自定义线程安全的数据结构

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/112\_写完数据之后如何更新分区对应的LogEndOffset/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/109\_如何基于FileChannel将数据写入底层的日志磁盘文件/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/107\_基于mmap机制实现的索引文件内存映射以及内存写入/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/116\_完成本地磁盘写入：将响应状态通过回调函数发送回去/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/104\_对分区目录表下的segment磁盘文件是如何构建的/笔记.doc**

0000 0000 0000 0000 0000.log

0000 0000 0000 0001 2358.log

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/100\_初探基于ReplicaManager写数据的入口处源码流程/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/102\_终于看到核心点：基于Parition将自己的Batch数据写入磁盘/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/113\_如何以固定的时间频率将os cache的数据flush到磁盘/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/111\_深入分析写入日志磁盘文件的每一条日志是什么格式/笔记.doc**

1个字节 -> 8位

000 0 0000

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/101\_继续追踪将数据写入本地磁盘文件的源码流程/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/115\_一个Request中的多个分区的响应状态如何进行封装/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/106\_深入探索.index文件被写入稀疏索引的具体细节/笔记.doc**

写入稀疏索引的原理就说清楚了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/103\_最底层的磁盘写入入口：基于分区目录对应的Log写入/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/105\_基于segment机制将Batch数据写入磁盘文件段里去/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/098\_真正的磁盘文件管理组件：LogMaanager的初始化过程/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/110\_回头看看JDK中的NIO注释：FileChannel优先写入os cache/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/099\_Kafka Broker启动时是如何扫描本地磁盘感知本地Log存储的/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/114\_对于每个分区写磁盘的过程如何判定成功或失败/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/108\_初步看看数据是如何被写入到.log日志文件中的/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/097~116资料/097\_用于将消息写入磁盘文件的ReplicaManager是什么？/笔记.doc**

磁盘文件写入：非常的复杂，一点都不比网路通信那块简单

分析kafka的源码，我是绝对不会跳过太多的细节的，少数的细节，无关紧要。但是呢，对于他的网络通信、磁盘读写、内存管理、并发控制，必须要搞明白他底层的一些原理的，这些东西其实是精华

LEO、HW，两种核心的offset

一个partition有多个副本，leader + 多个follower

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/142\_多个Broker是如何竞争选举成为Controller的/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/145\_Controller是如何感知到集群里的Broker的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/151\_Kafka集群是如何默默的进行rebalance的？（1）/笔记.doc**

他无非就是把你的副本均匀分散在指定的一些broker上

基于那个脚本去执行这个副本重新分配的方案的时候，是如何来做的

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/143\_Controller Broker宕机之后如何触发重新选举？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/154\_如果Broker突然宕机，Controller如何触发leader选举？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/149\_回头思考一下：Producer如何拉取到Topic元数据？/笔记.doc**

分区 -> [0,1] / [1,2]

是不是就可以把数据进行分区了，都是对应着 分区leader所在 的 broker

打包成Request，把一批在一个broker上的分区leader的Batch数据打包在一起，发送给 broker，broker本身再把每个分区的数据写入本地leader副本的磁盘文件

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/144\_选举完Controller之后，各个Broker如何进行注册？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/141\_集群管控组件之KafkaController是如何进行初始化的/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/146\_对于新注册加入集群的Broker是如何进行处理的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/138\_回顾一下已经讲解完毕的Kafka架构的运行流程/笔记.doc**

（1）Kafka Broker的启动流程

（2）Kafka Broker集群会选举出来一个Controller，是基于zk来实现的

（3）Controller如何感知Broker集群，维护一个集群元数据，以及下发集群元数据给所有的Broker机器

（4）创建Topic，如何负责划分分区，每个分区的leader和follower角色分布在哪些Broker上，核心数据

（5）Producer写消息 -> Broker接收消息 -> 副本同步，全部搞定了

（6）如果说某个Broker突然加入进来，Controller如何感知，以及如何实现数据的重平衡，如何进行元数据的变更和下发

（7）如果说某个Broker突然宕机，Controller如何感知，如何重新选举leader，如何进行元数据的变更和下发

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/155\_Producer在短暂故障之后如何向新的Leader发送消息？/笔记.doc**

Producer在发送消息出现异常的时候，会如何进行处理，一定是隔一段时间再次进行重试，重试的时候应该是要重新拉取Topic元数据，此时就应该感知到某个分区的leader可能已经在别的broker上了

再次发送，就可以发送消息过去了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/148\_Controller是如何感知到新Topic创建以及推送元数据的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/139\_找找Kafka Broker启动的源码入口在哪里/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/147\_创建Topic的时候是如何将分区均匀分配给Broker的？/笔记.doc**

第一件事情一定是去创建 一个Topic，指定有几个分区，每个分区有几个副本

对于你的Controller而言，他必须得决定，这个Topic的每个分区的leader和follower副本如何均匀的分配给各个Broker

有一点很重要的一点，你发起请求创建一个Topic，一定会指定几个Broker地址，此时一定是这样子的，会先随机发送一个请求给某个Broker，从那个Broker也提取出来集群所有的元数据

此时一定就可以感知到谁是Controller

另外一点，比如说，可以直接去读zk中的/controller节点，读取到brokerId，就知道谁是broker了，他是可以把请求发送给Controller

在客户端脚本中直接实现，读取zk元数据，均匀分配副本给broker，分配方案直接写入zk

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/152\_Kafka集群是如何默默的进行rebalance的？（2）/笔记.doc**

在同步的过程中，每次发送了fetch带上自己的LEO，一旦说leader发现follower的LEO大于了当前的HW，就会把新的 两个follower给加入到ISR列表里去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/138~157资料/140\_负责与ZK通信的组件是如何初始化的（附ZK自学提示）/笔记.doc**

ZooKeeper而言，他实际上来说，不是属于大数据的技术栈

分布式存储（HDFS），分布式资源调度（YARN），分布式计算（MapReduce/Spark），分布式数据仓库（Hive），分布式NoSQL数据库（HBase），分布式即席查询引擎（Presto/Impala），分布式OLAP分析引擎（Kylin），分布式流式计算（Storm/Spark Streaming/Flink），分布式消息系统（Kafka）

ZooKeeper这个东西他本身不是说大数据技术栈里的东西，分布式系统中的基石性的组件和系统，大数据本身是分布式系统，所以会大量的依赖于ZooKeepe；对于Java技术栈里很多开源的组件，微服务注册中心，依赖于ZooKeeper的

是放在Java架构课里的，所以在大数据架构课，我是不会深入分析他的，自研分布式系统的项目，分布式图片存储系统，分布式微服务注册中心，那些东西都要依赖于ZooKeeper，ZK我是放在Java架构课里讲解的

Elasticsearch，本质我是放在Java架构课里深入讲解的，源码，集群，优化

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/078\_在源码中找找Kafka服务端的网络通信模块/笔记.doc**

学了很多的东西，最后发现过了几个月全忘了，废话么

没有人是神仙，有多少人是神童，过目不忘？复习，重新看视频？就是看图，只要看到一张图，刷拉拉，全部想起来了，每过一段时间，就看一遍图

Kafka服务器：网络通信，直接基于NIO来设计实现的

四大块：网络通信，磁盘读写，副本同步，集群架构

消费端

主要是看里面的NIO网络通信的各种实现细节，在图片上而言，基本上来说就那么回事，里面很多网络通信的细节，一定要一次性给你打通

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/081\_负责监听客户端连接的Acceptor线程是如何启动的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/087\_从Kafka Broker启动源码中寻找处理请求的工作线程在哪里/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/080\_Kafka服务端是如何初始化网络通信模块准备接收连接的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/085\_客户端发送请求过来之后，Processor又是如何接收请求的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/091\_初步分析一下KafkaApis对生产消息的请求进行处理的逻辑/笔记.doc**

每一个请求，都包含了多个分区的一个Batch

所以在执行请求的时候，其实是将每个分区的Batch数据追加到那个分区对应的磁盘文件里去，所以说每个分区其实都对应了一个结果

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/093\_回过头看看Processor是如何处理队列中的响应的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/089\_再来看看处理请求的工作线程池是如何初始化以及运行的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/079\_Kafka源码都是Scala写的，你不懂Scala能看源码吗？/笔记.doc**

主要是Scala + Java一起写的

用Scala主要是一些语法糖，面向对象，方法，变量，集合，API

一些核心的部分，都是Java写的，真正遇到IO、并发、网络，都是用的Java。Spark是用Scala的，慢慢开始用Java来重构核心代码，基于Spark来开发一些作业

完全可以基于Java 8、Java 9，代码已经很优雅了，就喜欢用Scala来开发Spark作业

最多就是说他有一些语法糖，让可以代码写的更快一些，或者更加的方便一些，功能更强大一些

Kafka、Spark，作者为何要用Scala开发呢？Scala他也是一门JVM的语言

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/090\_Kafka服务器最核心的IO线程启动之后会干什么？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/082\_用于接收请求的Processor线程是如何启动的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/088\_先一起来看看奇怪的KafkaApis到底是个什么东西？/笔记.doc**

KafkaApis，封装了处理各种不同请求的业务逻辑

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/092\_处理完一条消息过后如何将结果放入Processor的响应队列中？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/083\_Acceptor线程是如何不停的轮询发现新的客户端连接请求的？/笔记.doc**

如果是跟9092端口来进行连接请求的通信，必须是客户端发送过来的连接的请求，你是在跟ServerSocketChannel进行通信，如果你是发送请求过来的话，其实是跟Processor那里的SocketChannel在通信

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/095\_对于发送完毕的响应消息会如何进行善后处理？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/084\_对于新的客户端连接，Processor线程如何准备监听请求？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/096\_从SocketChannel中读写数据的时候发现连接断开如何处理？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/094\_通过监听OP\_WRITE发送KafkaChannel中的响应消息/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/078~096资料/086\_对于接收到的请求，是如何转交给后续的工作线程的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/166\_如果Consumer挂掉了，此时如何rebalance？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/161\_Consumer Leader是如何制定分区分配方案的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/160\_Consumer Coordinator是如何选举出Consuemr Leader？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/163\_Consumer的poll方法是如何进行数据的消费的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/158\_先来重新回顾一遍Kafka Consumer的实现原理/笔记.doc**

Producer -> Broker -> Consumer

有一个consumer group的概念，每个consumer初始化启动，都会发送请求到某一个broker的consumer group coordinator上去，加入group，如果coordinator确认一个consumer group组建完毕

此时就会选举出来一个consumer leader

consumer leader接到通知说自己是leader了，接着就会制定一份分区消费的方案，每个分区只会分配给一个consumer，一个consumer可以有多个分区，返回给coordinator，下发给所有的consumer知道

然后大家就开始进行消费了，单线程，一次拉取一批数据过来，给你进行消费

默认是后台自动提交offset，写入内存\_\_consumer\_offsets里去，其实就知道每个consumer group对一个topic的各个分区都消费到了哪些地方了，假设你的consumer停机了，重启，此时会接着上一次提交的offset继续进行消费

如果某个consumer挂掉，coordinator感知到了之后，就会rebalance，把那个挂掉的consumer的分区分配给其他的consumer。如果在group中新增了一个consumer，此时会考虑将已有consumer的分区转移给新的consumer来进行消费

新的consumer直接会接着那个分区被提交过的offset继续进行消费即可

consumer这块，我不会无止尽的去剖析里面的源码细节了，涉及到的一些原理就已经给大家讲清楚了，过一遍源码流程

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/159\_来看看Consumer初始化的时候是如何加入Group的？/笔记.doc**

发送Join Group请求过去之后，在broker端的源码，大家都可以自己去看一下了，都没什么太大的难度，核心的原理，作业：Cooridnator接收到了gruop中的各个consumer发送过来的JoinGruop之后会怎么样

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/167\_如果新加入了一个Consumer，此时如何rebalance？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/162\_Cooridnator收到分区分配方案之后如何下发给全组/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/158~168资料/168\_给每个人留下的作业：自己仔细看Consumer源码细节/笔记.doc**

整个consumer，无非就是组建一个consumer group，跟coordiantor交互

leader制定分区分配方案，下发方法给所有的consumer

每个consumer通过poll开始拉取数据进行消费 -> broker层面是如何结合稀疏索引 + 二分查找来读取数据

consumer是如何自动提交offset的，如果是手动提交offset是怎么来做的

每个consumer是如何发送心跳给coordinator的

coordinaotr是如何通过延迟任务来监控consumer的心跳，对于挂掉的consumer重新让leader制定分配方案，再次下发；对于新加入group的consumer，如何让leader重新制定新的分配方案，再次下发

集群架构：contorller的选举，分区分配方案，变动，broker上线，broker挂掉

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/125\_如何结合稀疏索引实现从磁盘文件读取要同步的数据/笔记.doc**

NIO来说 ，主要知道一个文件读取的起始的position以及结尾的position，就可以了，此时就可以基于NIO读取文件的一段数据，变成ByteBuffer，MessageSet

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/127\_拉取到副本数据之后写入follower磁盘文件中/笔记.doc**

两边的LEO的更新其实是非常简单的

leader而言，如果有数据写入就更新LEO；follower而言，如果拉取到了数据，就更新LEO

第一个，leader而言是如何更新HW

第二个，ISR是如何更新的

第三个，fetch的时候如果刚好没有新的数据如何延迟执行

第四个，acks=all，在produce的时候，如何延迟返回响应 ，等待副本拉取

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/117\_负责主动为follower副本拉取数据的后台线程在哪里？/笔记.doc**

副本机制，每个分区可以有多个副本，其中一个leader负责读写请求，还有其他的follower是负责同步数据，做一个热备份，在其他机器上的broker会负责从leader来拉取数据同步到follower中

副本同步机制给好好讲解一下

从leader拉取数据的过程，以及写入follower本地磁盘文件

LEO的更新，随着数据不停的拉取，HW的更新

ISR列表的更新，如何监控各个副本的同步进度，维护ISR

acks=0的时候，如何等待多个副本同步成功再返回响应回去

每个Broker一定会有一个后台线程，他会感知到自己这里有哪些分区的follower，同时需要知道那些分区的leader在哪里，创建Topic的时候，指定分区数量以及副本的数量，此时一定会决定Topic有哪些分区

每个分区有哪些副本，leader副本在哪儿，follower副本在哪儿

这些都是元数据更新的机制，保证说broker自己知道自己负责的分区，leader/follower

肯定会有一个后台的线程

ReplicaManager，他呢是负责把数据写入本地磁盘文件，他内部就有后台线程负责从leader拉取数据过来

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/126\_将读取到的数据通过回调函数发送回follower副本/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/118\_拉取副本的ReplicaFetcherManager到底是个什么东西？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/120\_对一批follower分区是如何创建ReplicaFetcher线程的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/123\_如何利用NetworkClient组件将FetchRequest发送出去？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/137\_拉取数据的时候如何触发延迟返回的produce任务？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/128\_leader副本被拉取数据之后如何维护follower的LEO/笔记.doc**

每次follower发送一个fetch过去，带上自己的LEO，fetch的时候是接着当前的follower的LEO来继续拉取的

所以的话呢，leader是可以感知到当前follower的LEO

所以每次收到一个fetch请求，leader都要去更新自己本地维护的各个follower的LEO

然后leader才可以每次都判断一下，各个follower的LEO是否都超出了当前的HW了，如果是的话，那么就可以把自己的HW往前面推移了

只要一更新这个东西了以后，就可以根据各个follower LEO来判断，是否要推移leader HW了，如果推移了leader HW之后，就可以返回最新的HW给follower，进一步推动follower HW也会变化

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/130\_后台运行的ISR检查线程是如何踢出落后的follower的？/笔记.doc**

leader LEO=100，HW=90

follower LEO=90

follower LEO=80

如果说ISR收缩，第二个follower给干出去了，此时就需要更新 HW

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/129\_ReplicaManager启动时运行ISR检查线程的源码/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/124\_查阅KafkaApis对于FetchRequest的处理逻辑源码/笔记.doc**

先会尝试从本地磁盘文件中读取指定offset之后的数据

如果能读取到，那么就直接返回给人家就可以了

是不是要考虑更新一下HW，ISR，这些东西是否需要在这里来维护

如果读取不到任何新的数据，此时需要采用时间轮机制，延迟执行fetch

如果这个leader分区有新的数据写入，此时可以唤醒时间轮中等待的FetchRequest来执行数据拉取

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/133\_每次follower拉取数据之后是如何尝试更新HW的？/笔记.doc**

leader LEO=100,HW=90

follower LEO=90

follower LEO=95

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/136\_acks=all的时候是如何让produce任务延迟返回的/笔记.doc**

acks=all / -1，必须等待 所有的ISR里的follower都拉取到了这批数据，才可以返回，这个timeout就是你在producer端设置的一个值，发送消息 最大的超时时间，30s，在这里就是插入时间轮中的30s的时间格

如果时间轮正常走，走到30s自然就会返回一个响应

如果这个过程中判断发现 各个follower都对这次produce中的分区，都拉取到了数据，此时就可以提前唤醒时间轮中的任务，调用回调函数去返回就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/121\_ReplicaFetcher线程启动之后的入口源码逻辑分析/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/132\_被踢出ISR列表的follower是如何重新加入ISR的？/笔记.doc**

如果说一个follower他的LEO是大于等于leader的HW的，就可以把这个follower给加回到ISR列表里去

ISR踢出去的时候，超过10秒没发送fetch请求

ISR加回去的时候，LEO大于等于leader HW，就可以回去

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/122\_拉取数据的FetchRequest是如何构建出来的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/131\_ISR列表收缩之后是如何更新leader HW的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/134\_如果发送fetch请求的时候没有拉取到任何数据怎么办？/笔记.doc**

时间轮机制，轮子，把一个DelayedFetch任务插入到500毫秒之后的时间格里去，随着轮子的运转，最多就是500毫秒之后就会执行这个任务，如果在这个500毫秒之内，有新的数据进入到了leader中

是不是就是说，你的fetch任务就可以执行了呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/119\_初步分析一下创建ReplicaFetcher线程的入口方法的源码注释/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/04\_探索世界上最复杂的消息系统：Kafka内核源码分析unzip/117~137资料/135\_leader接收到新的数据之后触发延迟的fetch任务继续执行/笔记.doc**

DelayedFetch的时候，其实是有回调函数，如果一旦触发他完成，意思就是说调用回调函数，去发送空数据回去，人家立马可以再次发送一个fetch过来，此时一定就可以获取到数据了