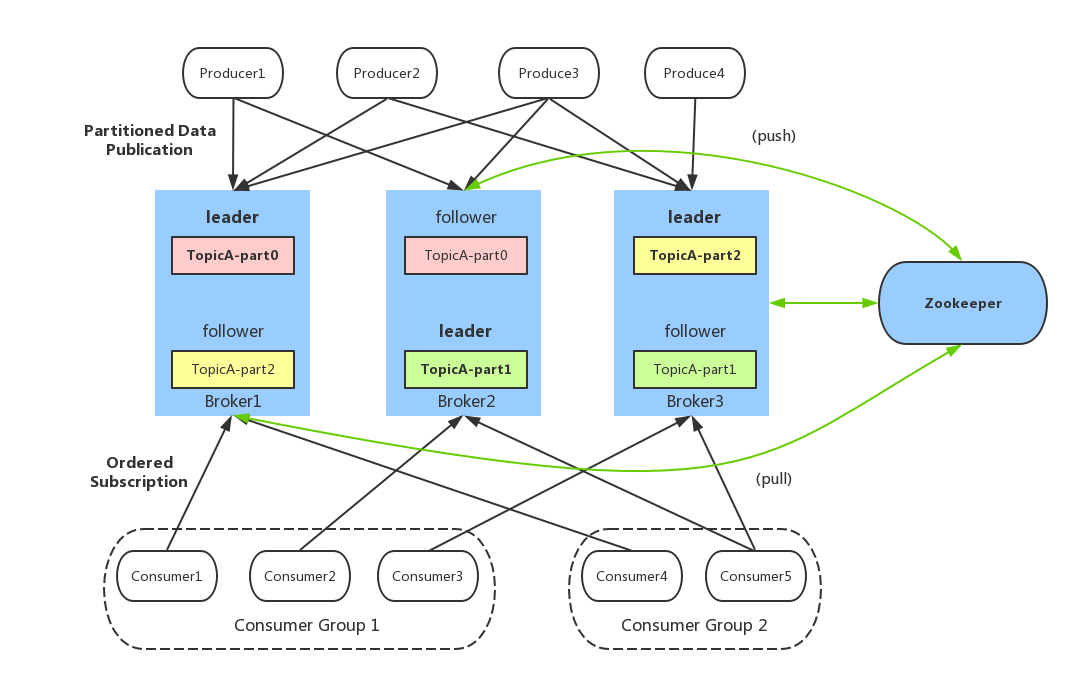
Kafka集群成员关系

Kafka的系统架构图如下所示：



它的架构包括以下组件：

1. Topic(话题)，特定类型的消息流，消息是字节的有效负载（Payload），话题是消息的分类名和种子（Feed）名
2. Producer(生产者)，能够发布消息到话题的任何对象，生产者可以选择序列化方法对消息内容编码
3. 已发布的消息保存在一组服务器中，它们被称为代理（Broker）或Kafka集群
4. Consumer(消费者)，可以订阅一个或者多个话题，并从Broker拉数据，从而消费这些已经发布的消息。
5. Consumer Group，在Kakfa中每个Consumer都属于一个Consumer Group，每条消息只能被Group中的一个Consumer消费，但是可以被多个Consumer Group消费
6. Patition，物理上的概念，每个Topic包含一个或者多个Partition
7. Replica，为了保证可用性，Partition可以有多个副本
8. Leader，Partition有多个副本时，选择其中一个作为Leader，Producer和Consumer只和Leader交互。Follower从Leader中复制数据。
9. Controller，Kafka其中一个服务器，用来进行Leader Election以及各种Failover。

在Kafka集群中，通常包括多个代理，为了负载均衡，将话题分成多个分区，每个代理存储一个或者多个分区。多个生产者和消费者同时生产和获取消息。

在Kafka中，使用Zookeeper来维护集群成员的信息，每个broker都有一个唯一标识符，可以在配置文件里指定，也可以自动生成。在broker启动的时候，通过创建临时节点把自己的ID注册到Zookeeper中。Kafka组件订阅Zookeeper的/brokers/ids路径，当有broker加入集群或退出集群时，这些组件就可以获得通知。

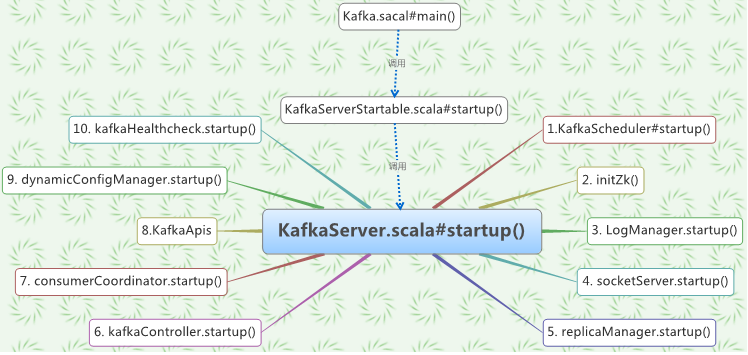
在broker停机，出现网络分区或者长时间垃圾回收停顿，broker会从zookeeper上断开连接，此时broker在启动时创建的临时节点会自动从Zookeeper上移除，监听broker列表的Kafak组件会被告知该broker已移除。在关闭broker时，对应的节点也会消失，不过它的ID会继续存在与其他数据结构中。例如，主题的副本列表里就可能包含这些ID，在完全关闭一个broker之后，如果使用相同的ID启动另一个全新的broker，就会立即加入集群，并拥有与旧broker相同的分区和主题。

# Kafka Broker启动

Kafka的main入口中调用KafkaServerStartable startup，其对KafakServer的封装，在使用命令行：

*./kafka-server-start.sh -daemon ../config/server.properties*

进行启动时，调用这个类。KafkaServer是一个broker，其startup中启动modules如下：



## 1.1 KafkaScheduler

KafkaScheduler是基于ScheduledThreadPoolExecutor的scheduler，在后台执行任务，其处理的任务，其核心方法为schedule。其他组件如LogManager，KafkaController及ReplicaManager均调用KafkaScheduler实现方法的周期性执行。

*def schedule(name: String, fun: () => Unit, delay: Long, period: Long, unit: TimeUnit) {*

*this synchronized { //仅需要传递fun方法，即可调用*

*ensureRunning() //创建线程*

*val runnable = CoreUtils.runnable {*

*try {*

*trace("Beginning execution of scheduled task '%s'.".format(name))*

*fun()*

*} ..... }*

*if(period >= 0) //通过Executor调用*

*executor.scheduleAtFixedRate(runnable, delay, period, unit)*

*else*

*executor.schedule(runnable, delay, unit) }}*

## 1.2 zk的初始化

Kafka是基于Zookeeper进行配置管理，因此需要创建zkclient和Zookeeper集群通信，ZK的初始化主要完成两件事情：

1. 连接到zk服务器

*\_zkClient = createZkClient(config.zkConnect, secureAclsEnabled)*

*\_zkClient.createTopLevelPaths() //创建zk paths*

1. 创建通用节点，通用节点包括：

*val PersistentZkPaths = Seq(*

*"/consumers", // old consumer path*

*BrokerIdsZNode.path,*

*TopicsZNode.path,*

*ConfigEntityChangeNotificationZNode.path,*

*DeleteTopicsZNode.path,*

*BrokerSequenceIdZNode.path,*

*IsrChangeNotificationZNode.path,*

*ProducerIdBlockZNode.path,*

*LogDirEventNotificationZNode.path*

*) ++ ConfigType.all.map(ConfigEntityTypeZNode.path)*

## 1.3 日志管理器LogManager

LogManager是Kafka的子系统，负责log的创建、检索和清理，所有的读写操作都由LogManager来代理，在Log对象中包含若干个LogSegment。

LogManager实例的生成依赖于KafkaScheduler的后台调度管理组件，管理各partition消息，包括index,log及segment等管理。在KafkaServer的startup函数中对LogManager初始化并启动：

*logManager = LogManager(config, initialOfflineDirs, zkClient, brokerState, kafkaScheduler, time, brokerTopicStats, logDirFailureChannel)*

*logManager.startup()*

其中相关的配置项：

|  |  |
| --- | --- |
| 配置项 | 描述 |
| log.cleaner.enable | 是否开启日志定时清理，默认是启用的 |
| log.cleaner.threads | 用于配置清理过期日志的线程个数，用于日志合并，默认值1 |
| log.cleaner.backoff.ms | 用于定时检查日志是否需要清理的时间间隔，默认15秒 |
| log.cleaner.dedupe.buffer.size | 用户清理过期数据的内存缓冲区，默认值128MB |
| log.cleaner.io.buffer.load.factor | 配置清理内存缓存区的数据装载因子，默认0.9 |
| log.cleaner.buffer.size | 用于清理过去数据的IO缓冲区大小，默认512KB |
| message.max.bytes | 设置单条数据的最大值，LogRetentionHours \* 60 \* 60 \* 1000L |
| log.flush.scheduler.interval.ms | 检查日志是否被flush到磁盘，默认不检查 |
| log.retention.check.interval.ms | 定期检查保留日志的时间间隔，默认5分钟 |
| num.recovery.threads.per.data.dir | 启动时，用于日志恢复的线程个数 |

### 1.3.1 LogManager初始化

1) 使用Pool管理所有的Log对象

private val currentLogs = new Pool[TopicAndPartition, Log]()

2)支持将本地多文件夹作为日志存储目录，创建和验证这些目录的有效性

createAndValidateLogDirs(logDirs, initialOfflineDirs)

判断是否有重复的log目录，不存在则创建。

1. 使用文件锁锁定目录：val dirLocks = lockLogDirs(liveLogDirs)
2. 创建每个目录中的recovery-point-offset-checkpoint文件的读取类对象，记录各个offset之前的数据是否已经落盘成功
3. 恢复并加载日志目录中的日志文件：loadLogs
4. 启动LogCleaner

### 1.3.2 加载partition日志的Segment

在LogManager实例生成时，会读取每个目录下的topic-partition目录，并生成Log实例，初始化时读取Segment信息

*private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] = new ConcurrentSkipListMap[java.lang.Long, LogSegment]*

调用loadSegments去加载对应的Partition的segments的信息

*for (file <- dir.listFiles.sortBy(\_.getName) if file.isFile) {*

*if (isIndexFile(file)) {*

*.....*

*val offset = offsetFromFile(file)*

*val logFile = Log.logFile(dir, offset)*

*} else if (isLogFile(file)) {*

*.....*

*val segment = LogSegment.open(dir = dir,*

*baseOffset = baseOffset,*

*config,*

*time = time,*

*fileAlreadyExists = true)*

*....*

*addSegment(segment)*

*}*

*}*

### 1.3.3 startup函数启动LogManager

*scheduler.schedule("kafka-log-retention",cleanupLogs \_,delay = InitialTaskDelayMs,*

*period = retentionCheckMs,TimeUnit.MILLISECONDS) //读取定时删除过去日志*

*scheduler.schedule("kafka-log-flusher",flushDirtyLogs \_,delay = InitialTaskDelayMs,*

*period = flushCheckMs,TimeUnit.MILLISECONDS)*

*//配置的执行间隔，定期对partition的segment的磁盘缓冲区进行flush操作*

*scheduler.schedule("kafka-recovery-point-checkpoint",checkpointLogRecoveryOffsets \_, delay = InitialTaskDelayMs,period = flushRecoveryOffsetCheckpointMs,TimeUnit.MILLISECONDS)*

*//定期对个partition中当前的offset进行checkpoint操作*

*scheduler.schedule("kafka-log-start-offset-checkpoint", checkpointLogStartOffsets \_,*

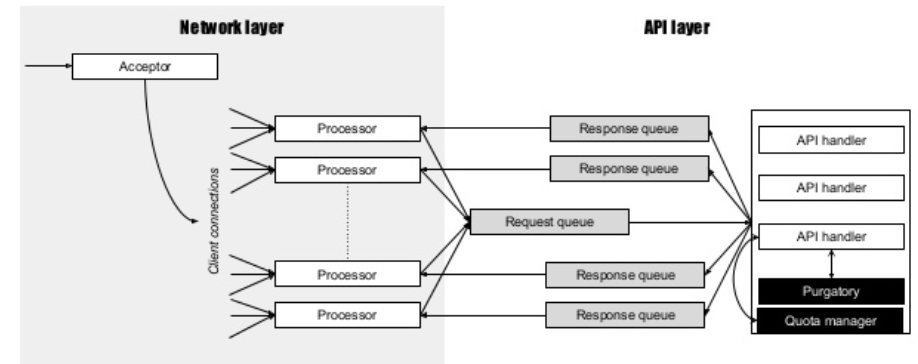
*delay = InitialTaskDelayMs,period = flushStartOffsetCheckpointMs,TimeUnit.MILLISECONDS)*

*//定期启动CleanerThread线程*

*scheduler.schedule("kafka-delete-logs", deleteLogs \_,delay = InitialTaskDelayMs,unit = TimeUnit.MILLISECONDS)*

## 1.4 SocketServer

SocketServer是基于JDK NIO的socket服务器，采用了Reactor模式，其线程模型如下图所示：



1. 一个Acceptor线程接收/处理所有新的连接
2. N个Processor线程，每个Processor都有自己的Selector，从每个连接中读取请求
3. M个Handler线程处理请求，并将产生的请求返回给Processor线程用于写回客户端

Handler和Processor之间通过队列来缓冲请求。S

### 1.4.1 SocketServer的启动

SocketServer在启动时会启动一个Acceptor和N个Processor

*private def createAcceptorAndProcessors(processorsPerListener: Int,*

*endpoints: Seq[EndPoint]): Unit = synchronized {*

*val sendBufferSize = config.socketSendBufferBytes*

*val recvBufferSize = config.socketReceiveBufferBytes*

*val brokerId = config.brokerId*

*endpoints.foreach { endpoint =>*

*val listenerName = endpoint.listenerName*

*val securityProtocol = endpoint.securityProtocol*

*val acceptor = new Acceptor(endpoint, sendBufferSize, recvBufferSize, brokerId, connectionQuotas)*

*addProcessors(acceptor, endpoint, processorsPerListener)*

*KafkaThread.nonDaemon(s"kafka-socket-acceptor-$listenerName-$securityProtocol-${endpoint.port}", acceptor).start()*

*acceptor.awaitStartup()*

*acceptors.put(endpoint, acceptor)*

*}*

*}*

相关配置：

|  |  |
| --- | --- |
| 配置 | 描述 |
| message.max.bytes | Server可以接收的消息最大值 |
| num.network.threads | Server用来处理网络请求的线程数目，默认3 |
| num.io.threads | Server用来处理请求的IO线程数目，至少等于硬盘个数 |
| queued.max.requests | 排队等待IO线程处理的最大请求个数 |
| max.connections.per.ip | 每个ip地址上每个broker可以被连接的最大数目 |
| socker.request.max.bytes | Socket请求允许的最大请求数据量 |
| listeners |  |

### 1.4.2 kafka.network.Acceptor

这个类继承AbstractServerThread，实现Runnable接口，主要职责是监听客户端的连接请求，并建立和客户端的数据传输通道，为客户端指定一个Proccessor，然后去响应下一个客户端的连接请求，在run方法的主要逻辑如下：

* 在ServerSocketChannel上注册OP\_ACCEPT事件

*serverChannel.register(nioSelector, SelectionKey.OP\_ACCEPT)*

* 等待客户端的连接请求

*val ready = nioSelector.select(500)*

* 如果有链接进来，将其分配给当前的Processor，并且把当前processor指向下一个processor（采用Round Robin）的方式来选择processor

*if (ready > 0) {*

*val keys = nioSelector.selectedKeys()*

*val iter = keys.iterator()*

*while (iter.hasNext && isRunning) {*

*try {*

*val key = iter.next*

*iter.remove()*

*if (key.isAcceptable) {*

*val processor = synchronized {*

*currentProcessor = currentProcessor % processors.size*

*processors(currentProcessor)*

*}*

*accept(key, processor)*

*} else*

*....*

*currentProcessor = currentProcessor + 1*

*}*

*}*

*}*

其中SelectionKey是Channel与Selector的注册关系，在Acceptor中的Selector，只有监听到客户端连接请求的ServerSocketChannel的OP\_ACCEPT事件在上面。

Acceptor的accept方法的处理逻辑为：首先通过SelectionKey来拿到对应的ServerSocketChannel，并调用其accept方法来建立和客户端的连接，然后拿到对应的SocketChannel并交给了processor。

### 1.4.3 kafka.network.Processor

Proccessor也是继承自AbstractServerThread并实现Runnable接口，也是线程类，主要职责是负责从客户端读取数据和将响应返回给客户端，其本身并不处理具体的业务逻辑，不处理从客户端读取的数据。每个Processor都有一个Selector，用来监听多个客户端，因此可以非阻塞的处理多个客户端的读写请求。

1. 处理新的连接，在Acceptor中将SocketChannel分配给Processor

*processor.accept(socketChannel)*

在processor中有一个队列来保存这些数据连接，processor的accept方法处理如下：

*def accept(socketChannel: SocketChannel) {*

*newConnections.add(socketChannel) 🡸 ConcurrentLinkedQueue[SocketChannel]*

*wakeup() <= selector.wakeup*

*}*

将SocketChannel放到队列中，然后唤醒selector.

2）读取客户端的数据

在processor的run方法中，调用Selector.select方法来监听客户的数据请求，代码如下：

*private def processCompletedReceives() {*

*selector.completedReceives.asScala.foreach { receive =>*

*try {*

*openOrClosingChannel(receive.source) match {*

*case Some(channel) =>*

*val header = RequestHeader.parse(receive.payload)*

*val context = new RequestContext(header, receive.source, channel.socketAddress,*

*channel.principal, listenerName, securityProtocol)*

*val req = new RequestChannel.Request(processor = id, context = context,*

*startTimeNanos = time.nanoseconds, memoryPool, receive.payload, requestChannel.metrics)*

*requestChannel.sendRequest(req)*

*selector.mute(receive.source)*

*.....*

*}*

*}*

大致流程：

* 从SelectionKey中取出对应SocketChannel，并获取在SelectionKey上的Receive对象，创建BoundedByteBufferReceive，处理具体的读数据的逻辑。在客户端都有一个Receive对象来读取数据
* 如果数据从客户端读取完毕：receive.complete，则将读取的对象封装成Request对象，添加到RequestChannle中

3) kafka.network.RequestChannel

RequestChannel是Processor和Handler交换数据的地方，包含一个队列RequestQueue存放processor加入的Request，Handle会从里面取出Request来处理。RequestChannel为每个Processor创建了一个ResponseQueue，用来存放Handler处理Request后给客户端的Response，如下所示：

*private val newConnections = new ConcurrentLinkedQueue[SocketChannel]()*

*private val inflightResponses = mutable.Map[String, RequestChannel.Response]()*

*private val responseQueue = new LinkedBlockingDeque[RequestChannel.Response]()*

Handle从requestQueue中取出Request

*def receiveRequest(): RequestChannel.BaseRequest =*

*requestQueue.take()*

1. 返回数据给客户端

processor不仅负责从客户端读取数据，还要将Handler的处理结果返回给客户端，在run方法中调用processNewResponse来处理Handler提供给客户端的Response，如下：

*private def processNewResponses() {*

*var curr: RequestChannel.Response = null*

*while ({curr = dequeueResponse(); curr != null}) {*

*val channelId = curr.request.context.connectionId*

*try {*

*curr.responseAction match {*

*case RequestChannel.NoOpAction =>*

*updateRequestMetrics(curr)*

*openOrClosingChannel(channelId).foreach(c => selector.unmute(c.id))*

*case RequestChannel.SendAction =>*

*val responseSend = curr.responseSend.getOrElse(*

*sendResponse(curr, responseSend)*

*case RequestChannel.CloseConnectionAction =>*

*updateRequestMetrics(curr)*

*trace("Closing socket connection actively according to the response code.")*

*close(channelId)*

*}*

*} ....*

*}*

*}*

把requestChannel中的responseQueue的Response取出来，然后将对应通道的OP\_WRITE事件注册到selector上。

### 1.4.4 KafkaRequestHandler

Handler的职责是从requestChannel中取出Request，处理以后将Response添加到responseQueue中，其处理具体业务，其run方法如下：

*def run() {*

*while (!stopped) {*

*req match {*

*case RequestChannel.ShutdownRequest =>*

*shutdownComplete.countDown()*

*return*

*case request: RequestChannel.Request =>*

*try {*

*request.requestDequeueTimeNanos = endTime*

*apis.handle(request)*

*}*

*}*

KafkaApis执行具体的业务逻辑。

https://www.zybuluo.com/jewes/note/59978

http://smartyml.com/2016/11/30/Kafka%E7%9A%84%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1%E6%9C%BA%E5%88%B6/

http://zqhxuyuan.github.io/2016/01/08/2016-01-08-Kafka\_SocketServer/

## 1.5 KafkaApis

Kafka目前支持的请求有几十种：

*request.header.apiKey match {*

*case ApiKeys.PRODUCE => handleProduceRequest(request)*

*case ApiKeys.FETCH => handleFetchRequest(request)*

*case ApiKeys.LIST\_OFFSETS => handleListOffsetRequest(request)*

*case ApiKeys.METADATA => handleTopicMetadataRequest(request)*

*case ApiKeys.LEADER\_AND\_ISR => handleLeaderAndIsrRequest(request)*

*case ApiKeys.STOP\_REPLICA => handleStopReplicaRequest(request)*

*case ApiKeys.UPDATE\_METADATA => handleUpdateMetadataRequest(request)*

*case ApiKeys.CONTROLLED\_SHUTDOWN => handleControlledShutdownRequest(request)*

*case ApiKeys.OFFSET\_COMMIT => handleOffsetCommitRequest(request)*

*case ApiKeys.OFFSET\_FETCH => handleOffsetFetchRequest(request)*

*case ApiKeys.FIND\_COORDINATOR => handleFindCoordinatorRequest(request)*

*case ApiKeys.JOIN\_GROUP => handleJoinGroupRequest(request)*

*case ApiKeys.HEARTBEAT => handleHeartbeatRequest(request)*

*case ApiKeys.LEAVE\_GROUP => handleLeaveGroupRequest(request)*

*case ApiKeys.SYNC\_GROUP => handleSyncGroupRequest(request)*

*case ApiKeys.DESCRIBE\_GROUPS => handleDescribeGroupRequest(request)*

*case ApiKeys.LIST\_GROUPS => handleListGroupsRequest(request)*

*case ApiKeys.SASL\_HANDSHAKE => handleSaslHandshakeRequest(request)*

*case ApiKeys.API\_VERSIONS => handleApiVersionsRequest(request)*

*case ApiKeys.CREATE\_TOPICS => handleCreateTopicsRequest(request)*

*case ApiKeys.DELETE\_TOPICS => handleDeleteTopicsRequest(request)*

*case ApiKeys.DELETE\_RECORDS => handleDeleteRecordsRequest(request)*

*case ApiKeys.INIT\_PRODUCER\_ID => handleInitProducerIdRequest(request)*

*case ApiKeys.OFFSET\_FOR\_LEADER\_EPOCH => handleOffsetForLeaderEpochRequest(request)*

*case ApiKeys.ADD\_PARTITIONS\_TO\_TXN => handleAddPartitionToTxnRequest(request)*

*case ApiKeys.ADD\_OFFSETS\_TO\_TXN => handleAddOffsetsToTxnRequest(request)*

*case ApiKeys.END\_TXN => handleEndTxnRequest(request)*

*case ApiKeys.WRITE\_TXN\_MARKERS => handleWriteTxnMarkersRequest(request)*

*case ApiKeys.TXN\_OFFSET\_COMMIT => handleTxnOffsetCommitRequest(request)*

*case ApiKeys.DESCRIBE\_ACLS => handleDescribeAcls(request)*

*case ApiKeys.CREATE\_ACLS => handleCreateAcls(request)*

*case ApiKeys.DELETE\_ACLS => handleDeleteAcls(request)*

*case ApiKeys.ALTER\_CONFIGS => handleAlterConfigsRequest(request)*

*case ApiKeys.DESCRIBE\_CONFIGS => handleDescribeConfigsRequest(request)*

*case ApiKeys.ALTER\_REPLICA\_LOG\_DIRS => handleAlterReplicaLogDirsRequest(request)*

*case ApiKeys.DESCRIBE\_LOG\_DIRS => handleDescribeLogDirsRequest(request)*

*case ApiKeys.SASL\_AUTHENTICATE => handleSaslAuthenticateRequest(request)*

*case ApiKeys.CREATE\_PARTITIONS => handleCreatePartitionsRequest(request)*

*case ApiKeys.CREATE\_DELEGATION\_TOKEN => handleCreateTokenRequest(request)*

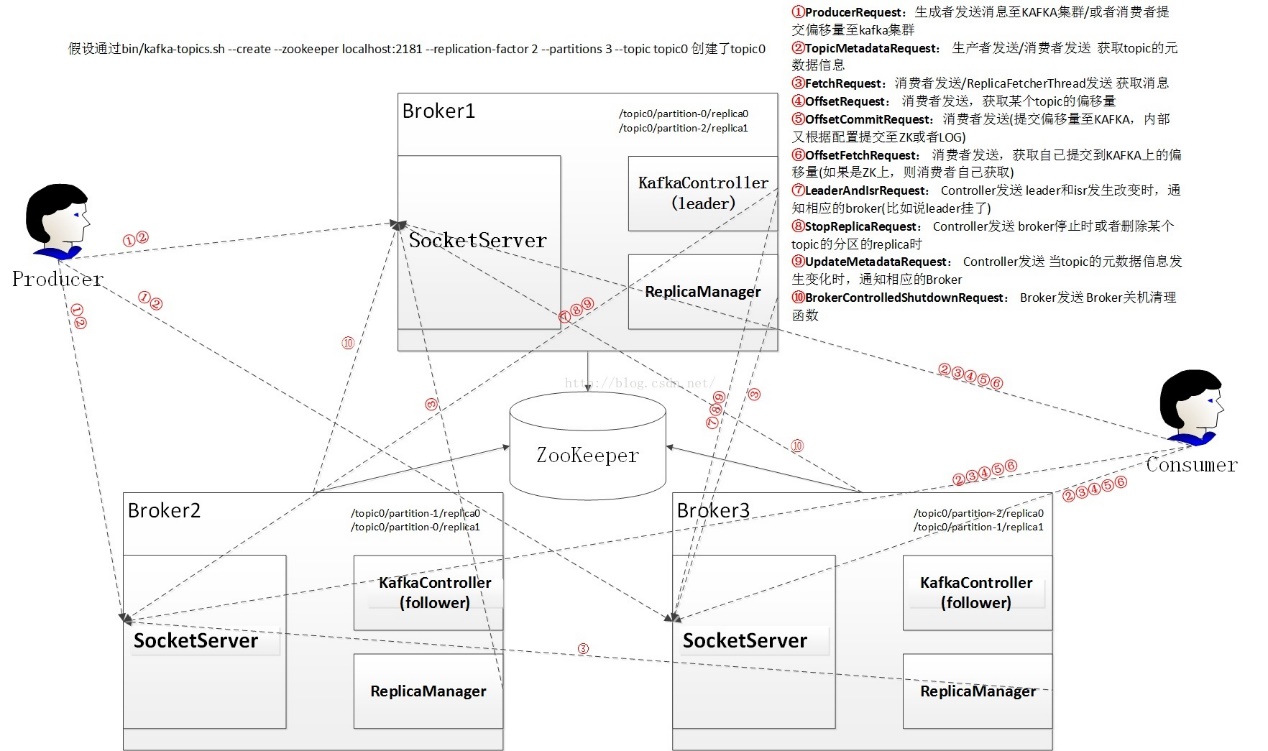
*case ApiKeys.RENEW\_DELEGATION\_TOKEN => handleRenewTokenRequest(request)*

*case ApiKeys.EXPIRE\_DELEGATION\_TOKEN => handleExpireTokenRequest(request)*

*case ApiKeys.DESCRIBE\_DELEGATION\_TOKEN => handleDescribeTokensRequest(request)*

*case ApiKeys.DELETE\_GROUPS => handleDeleteGroupsRequest(request)*

其中几种典型的Request的来源及处理如下图所示：



以Kafka Producer发送的RequestKeys.OFFSET\_FETCH，获取Kafka上的偏移量为例进行过程分析

*def handleOffsetFetchRequest(request: RequestChannel.Request) {*

*val header = request.header*

*val offsetFetchRequest = request.body[OffsetFetchRequest]*

*def createResponse(requestThrottleMs: Int): AbstractResponse = { //配置在ZK上，从zk上获取*

*val offsetFetchResponse =*

*... //zkClient，获取ConsumerOffset*

*val payloadOpt = zkClient.getConsumerOffset(offsetFetchRequest.groupId, topicPartition)*

*payloadOpt match {*

*case Some(payload) =>*

*(topicPartition, new OffsetFetchResponse.PartitionData(*

*payload.toLong, OffsetFetchResponse.NO\_METADATA, Errors.NONE))*

*case None =>*

*(topicPartition, OffsetFetchResponse.UNKNOWN\_PARTITION)*

*}*

*}*

*}.toMap.....*

*new OffsetFetchResponse(requestThrottleMs, Errors.NONE, (authorizedPartitionData ++ unauthorizedPartitionData).asJava)*

*} .....*

*offsetFetchResponse}*

*sendResponseMaybeThrottle(request, createResponse)*

*}*

## 1.6 KafkaController

Kafka集群中有多个Broker，有一个会被选举为controller leader，负责管理整个集群中分区和副本的状态，比如partition的leader副本故障时由Controller负责为该partition重新选举新的leader副本。在leader中维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，由leader动态维护。

Broker启动时会创建KafkaController对象，但是集群中只能由一个leader对外提供服务，每个节点上的KafkaController会在指定的Zookeeper路径下创建临时节点，只有第一个成功创建的节点的KafkaController才可以成为Leader，其余的都是Follower。当Leader故障后，所有的Follower会收到通知，再次竞争在该路径下创建节点从而选举新的Leader。

KafkaControoler内部由ZookeeperLeaderElector，用来通过ZK选举自己是否是Leader，选举路径为:

*/controller/\**

对zk集群建立一个会话超时的listener，启动时：

*case object Startup extends ControllerEvent {*

*def state = ControllerState.ControllerChange*

*override def process(): Unit =*

*{*

*zkClient.registerZNodeChangeHandlerAndCheckExistence(controllerChangeHandler)*

*elect()*

*}}*

当Broker故障再次触发的时候调用onResigningAsLeader，对应的2个函数分别对应：

onControllerFailover和onControllerResignation，其中核心逻辑如下：

*private def onControllerResignation() {*

*// 注销listener*

*zkClient.unregisterZNodeChildChangeHandler(isrChangeNotificationHandler.path)*

*zkClient.unregisterZNodeChangeHandler(partitionReassignmentHandler.path)*

*zkClient.unregisterZNodeChangeHandler(preferredReplicaElectionHandler.path)*

*zkClient.unregisterZNodeChildChangeHandler(logDirEventNotificationHandler.path)*

*unregisterBrokerModificationsHandler(brokerModificationsHandlers.keySet)*

*//重置topic deletion manager*

*topicDeletionManager.reset()*

*//关闭Leader相关服务*

*kafkaScheduler.shutdown*

*controllerContext.resetContext()*

*//重置*

*}*

## 1.7 ReplicaManager

ReplicaManager用于管理Kafka中Partition的副本信息，依赖于KafkaScheduler和LogManager，处理对消息的添加和读取的操作，实现副本间数据的同步等操作。

ReplicatManager的创建和启动如下：

*replicaManager = new ReplicaManager(config, metrics, time,*

*zkClient,*

*kafkaScheduler,*

*logManager,*

*isShuttingDown, quotaManagers,brokerTopicStats, metadataCache, logDirFailureChannel)*

ReplicatManager的主要作用是负责Controller的命令以完成replica的管理工作，其功能主要包括：

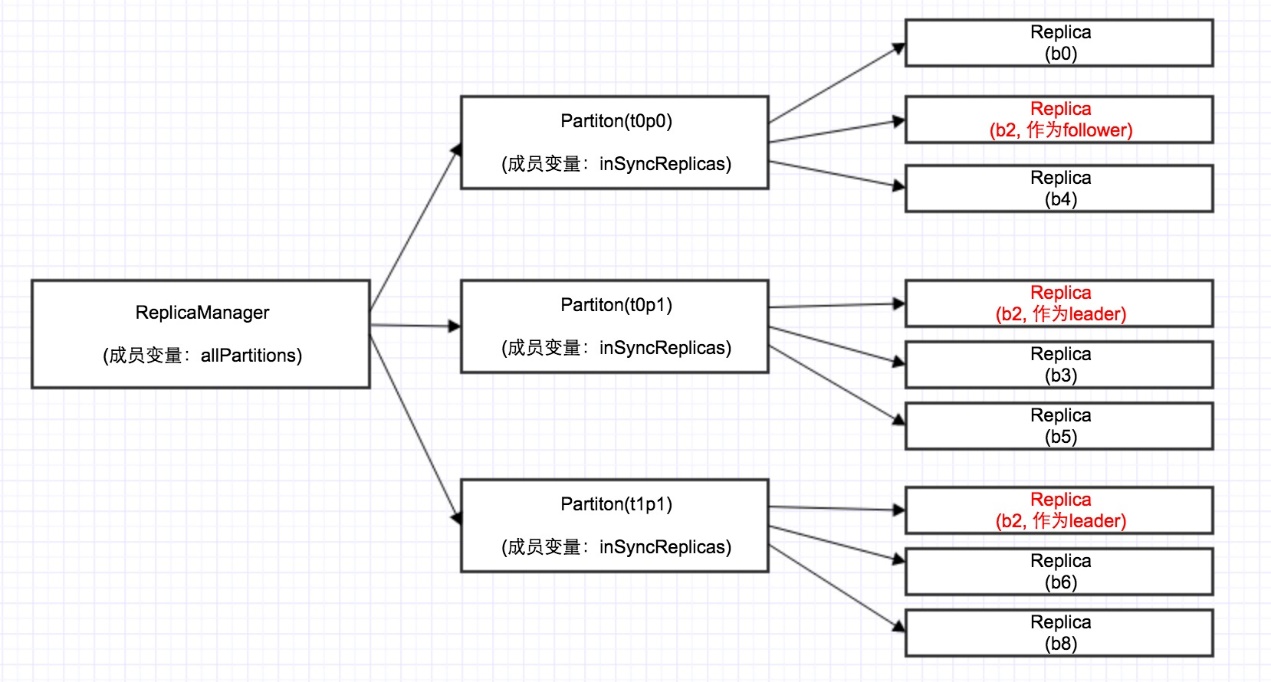
* maybeShrinkIsr，follower与leader的副本的数据同步时，定期检查这个心跳时间是否超过配置的时间，移出这个副本在isr上的选择
* maybePropagesIsrChanges，定期执行把isr的改变更新到zk中
* shutdownIdleReplicaAlterLogDirsThread，定期关闭无用的Replica

在ReplicatManager中核心的数据结构为：

*private val allPartitions = new Pool[TopicPartition, Partition](valueFactory =*

*Some(tp => new Partition(tp.topic, tp.partition, time, this)))*

架构图：



allPartitions存储了这台机器上所有的partition，对于每个Partition，其内部又存储了所有的Replica（ISR），如下所示：

*class Partition(val topic: String,*

*val partitionId: Int,*

*time: Time,*

*replicaManager: ReplicaManager,*

*val isOffline: Boolean = false) extends Logging with KafkaMetricsGroup {*

*val topicPartition = new TopicPartition(topic, partitionId)*

*private val localBrokerId = if (!isOffline) replicaManager.config.brokerId else -1*

*private val logManager = if (!isOffline) replicaManager.logManager else null*

*private val zkClient = if (!isOffline) replicaManager.zkClient else null*

*private val allReplicasMap = new Pool[Int, Replica]*

*private val leaderIsrUpdateLock = new ReentrantReadWriteLock*

*private var zkVersion: Int = LeaderAndIsr.initialZKVersion*

*@volatile private var leaderEpoch: Int = LeaderAndIsr.initialLeaderEpoch – 1*

*//核心变量，这个partition的Leader*

*@volatile var leaderReplicaIdOpt: Option[Int] = None*

*//核心变量：isr，除了leader外，其它所有活着的follower集合*

*@volatile var inSyncReplicas: Set[Replica] = Set.empty[Replica]*

*....*

*}*

对于每个Replica，其核心结构

*class Replica(val brokerId: Int,*

*val topicPartition: TopicPartition,*

*time: Time = Time.SYSTEM,*

*initialHighWatermarkValue: Long = 0L,*

*@volatile var log: Option[Log] = None) extends Logging {*

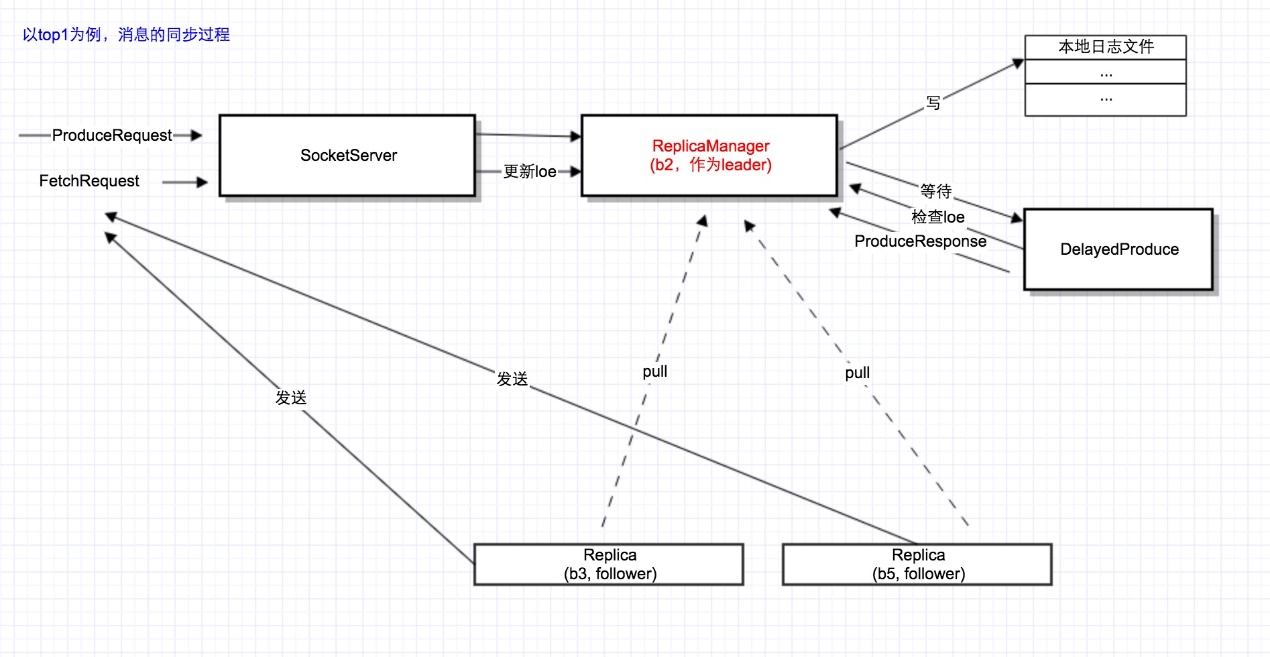
*@volatile private[this] var highWatermarkMetadata = new LogOffsetMetadata(initialHighWatermarkValue)*

*@volatile private[this] var logEndOffsetMetadata = LogOffsetMetadata.UnknownOffsetMetadata*

*......*

*}*

Replica的同步过程如下所示：



在Kafka集群中，对于Partition为例，b2为leader，b3和b5为follower，整个过程如下：

1. b2的SocketServer收到Producer的ProduceRequest请求，把这个请求交给ReplicaManager处理，调用appendMessages函数，把消息存在本地日志
2. ReplicatManager生成DelayedProduce对象，放入DelayedProducerPugatory中，等待follower来把该请求pul到自己的机器上
3. 2个followers和consumer一样，发送FetchRequest请求到SocketServer，ReplicatManager调用自己的fetchMessage函数返回日志，同时更新2个Follower的LOE，并且判断DelayedProducer是否可以complete，如果可以则发送ProducerResponse。

其他组件如：KafkaHealthChech、TopicConfigManager等不再介绍。

# Kafka Coordinator

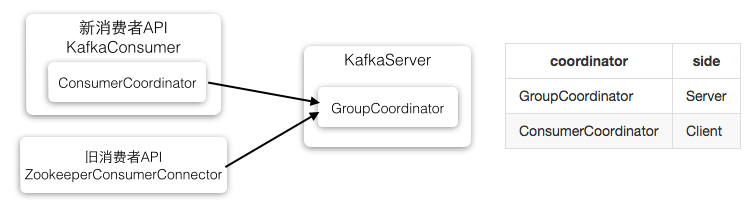
Kafka使用Coordinator实现了失败检测和Rebalance，包括以下类型：

* GroupCoordinator，Kafka Server实例，管理部分的Consumer Group和offset
* WorkerCoordinator，管理GroupCoordinator程序，主要管理Workers的分配
* ConsumerCoordinator，主要负责和服务端GroupCoordinator进行交互

在Kafka中Group Management protocol的动作序列：

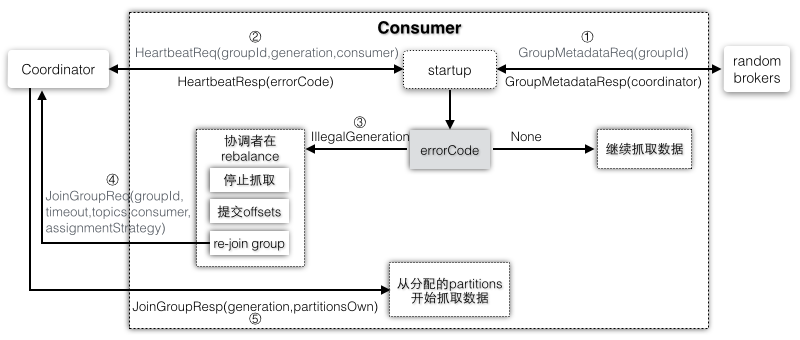
1. Group Registration，Group成员需要向Coordinator注册自己，并且提供关于成员自身的元数据
2. Group/Leader Selection，Coordinator确定Group包括哪些成员，并且选择其中一个作为Leader
3. State Assignment，Leader收集所有成员的metadata，并且给它们分配状态
4. Group Stabilization，每个成员收到leader分配的状态，并且开始处理

Coordinator的执行过程：



所有Consumer线程要先向Coordinator注册，由Coordinator选出leader，然后由leader来分配state。从Group Member中选出来一个作为Leader，由Leader来执行性能开销大的协调任务，这样把负载分配给Client端，从而减轻broker压力。所有Group Member（Consumer）都需要发送心跳给Coordinator。对于Kafka Consumer，实际上必须和Coordinator保存连接，还需要提交offset给Coordinator。

Consumer的消费者的工作过程如下图所示：



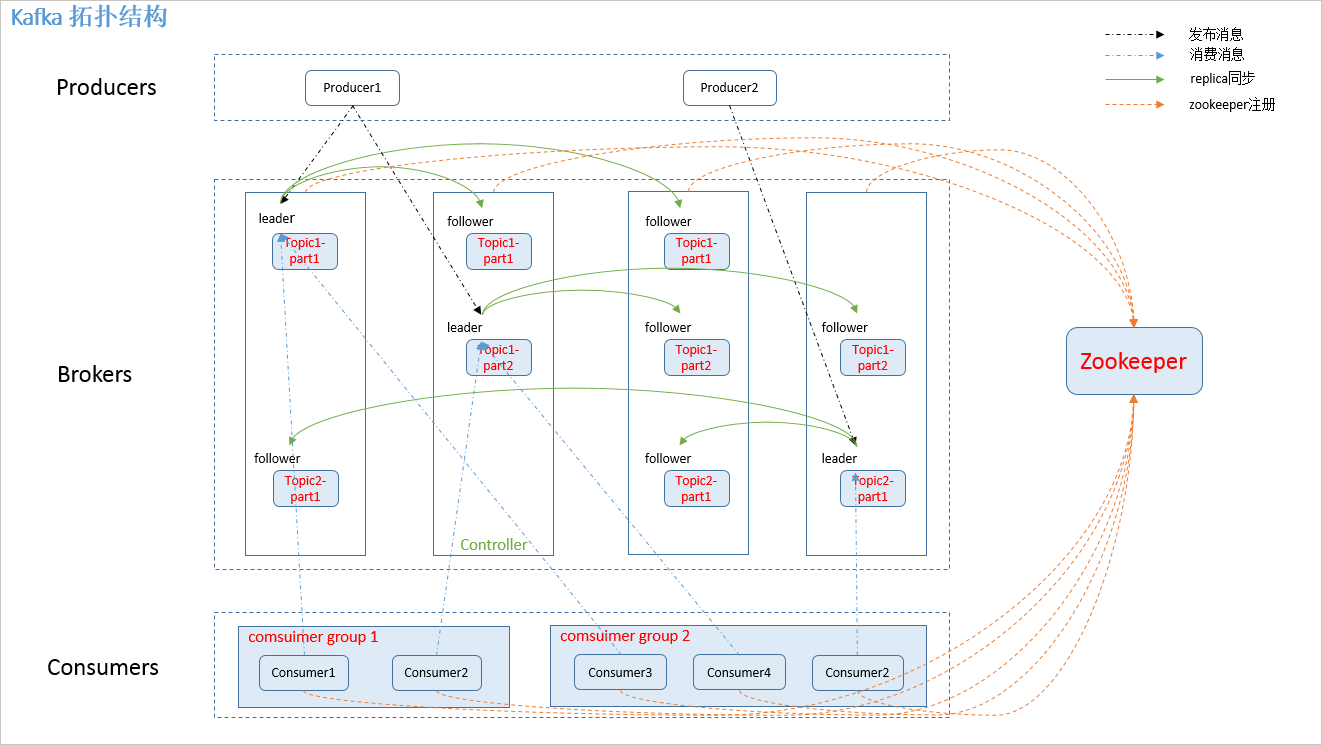
* 在Consumer启动时或者coordinator节点故障转移时，Consumer发送ConsumerMetadata

Request给任意一个brokers，在ConsumerMetadataResponse中，它接收对应的ConsumerGroup所属的Coordinator的位置信息

* Consumer连接Coordinator节点，并发送HeartbeatRequest，如果返回Response中返回IllegalGneration错误码，说明协调节点已经在初始化平衡。消费者就会停止抓取数据，提交offsets，发送JoinGroupRequest给协调节点。在JoinGroupResponse，它接收消费者应该拥有的topic-partitions列表以及当前Consumer Group的新的generation编号。这个时候Consumer Group管理完成，Consumer就可以开始fetch数据，并为它拥有的partitions提交offsets
* 如果HeartbeatResponse没有错误返回，Consumer会从它上次拥有的partition列表继续抓取数据。

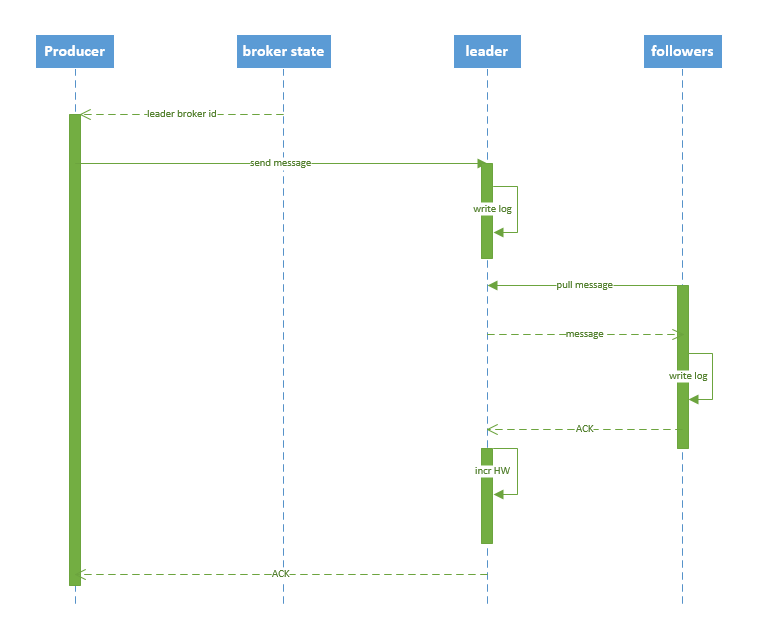
# Kafka执行流程

Kafka总拓扑结构：



## Producer发布消息

producer采用push模式将消息发布到broker，每条消息都被append到partition中，属于顺序写磁盘，写入消息序列图如下：

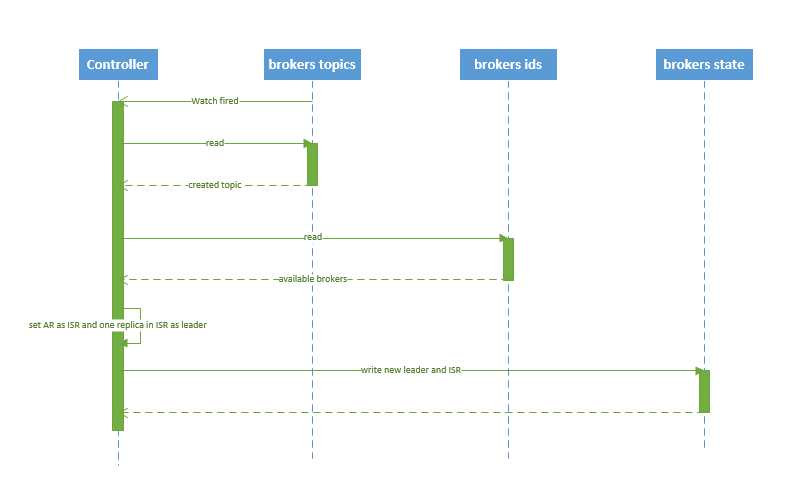


producer发送消息到broker时会根据分区算法选择将其存储到某个partition中，其流程如下：

1. 先根据partition获取其Broker Leader
2. producer将消息发送给该Leader
3. Leader将消息写入到本地log
4. Followers从leader中pull消息，写入本地log后，Leader发送ACK
5. Leader收到所有ISR中的replica的ACK后，增加High watermark并向producer发送ACK

## 创建Topic

物理上把topic分成一个或者多个partition，每个partition在物理上对应一个文件夹（存储该partition的所有消息和索引文件）。Topic的创建序列图如下所示：



1. Controller在ZK的/brokers/topic节点上注册watcher，当topic被创建则controller会通过watch得到该topic的partition/replica分配
2. Controller从/brokers/ids读取当前所有可用的broker列表
3. Controller通过RPC向相关的broker发送LeaderAndISRRequest