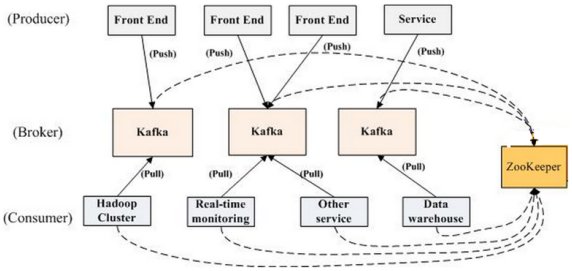
Base Part 1:Apache Kafka

定义:　　Kafka ==> 分布式发布-订阅消息系统
分布式，分区,可复制的,提交日志服务。(快速、可扩展)

功能:
1 .松耦和 2.异步处理 3.灵活性 & 峰值处理能力
4 . 可恢复性 ....

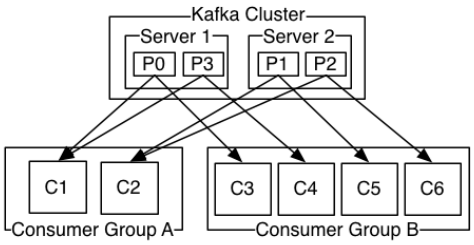
基本概念:
Topic : 是特定类型的消息流
Producer: 消息的生产者,发布消息到Topic的对象
broker: Kafka服务集群==>接受消息,保存消息,提供消息处理
consumers:消费者可以订阅一个或多个话题，
并从Broker拉数据，从而消费这些已发布的消息

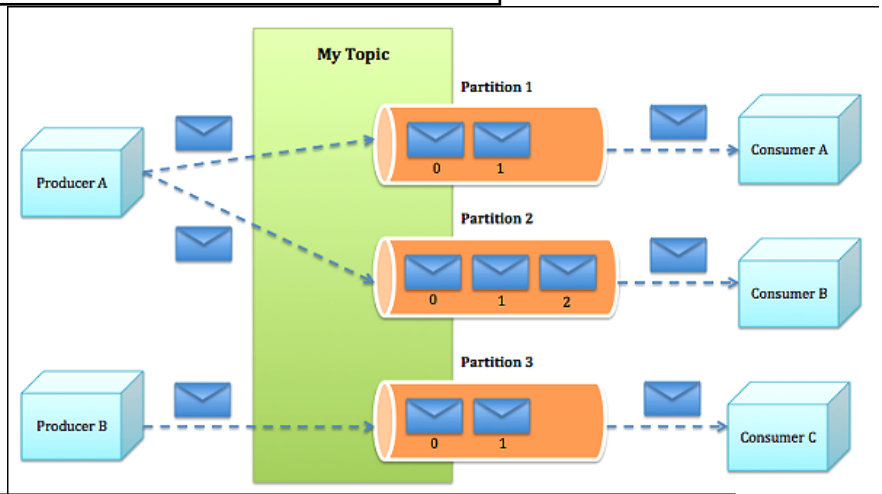
关系:
消息生产者（producer）发布关于某话题（topic）的消息
消息以一种物理方式被发送给了作为代理（broker）的服务器
消息使用者（consumer）订阅某个话题
，每条消息都会被发送给所有的使用者



消息类型:
queue:每个消息只被每个consumer消费一次
topic:每个消息被所有consumer消费
kafka实现:
1 按group分组,一个group有1个或多个consumer
2 多个consumer在同一个group内 == > queue
每个消息分配给一个consumer,均衡负载
3 每个消息会发送到所有group ==> topic
每个消息被所有group的一个consumer消费

kafka特性：
1 通过O(1)的磁盘数据结构提供消息的持久化
，这种结构对于即使数以TB的消息存储也能够保持长时间的稳定性能。
2 高吞吐量：即使是非常普通的硬件kafka也可以支持每秒数十万的消息。
3 支持同步和异步复制两种HA
4 Consumer客户端pull，随机读,利用sendfile系统调用，zero-copy ,批量拉数据
5 消费状态保存在客户端
6 消息存储顺序写
7 数据迁移、扩容对用户透明
8 支持Hadoop并行数据加载。
9 支持online和offline的场景。
10 持久化：通过将数据持久化到硬盘以及replication防止数据丢失。
11 scale out：无需停机即可扩展机器。
12 定期删除机制，支持设定partitions的segment file保留时间。





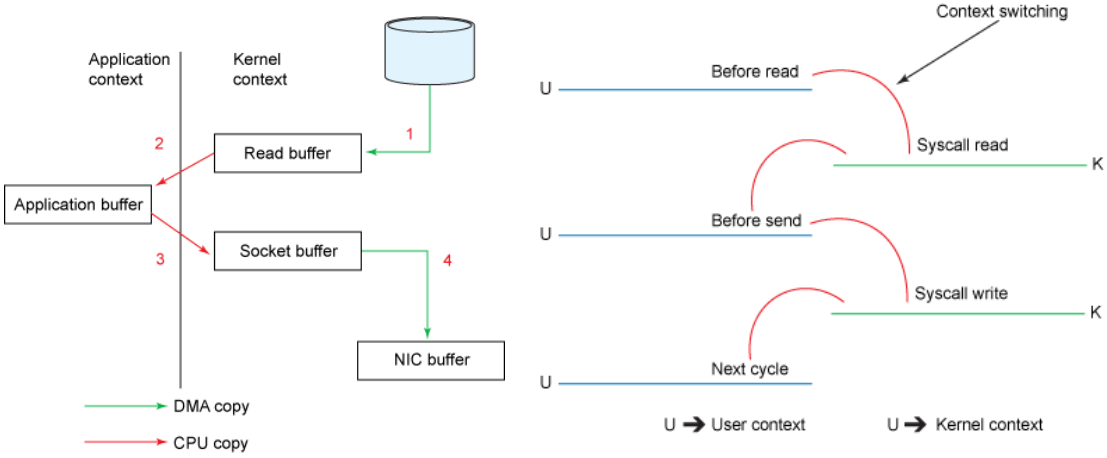
Topic & patition
1. partition :
　逻辑上指topic一个分区,物理上对应一个文件,保存在kafka集群
　本质 == >commit log :有序的,不可变的,连续的消息
　offset : 在这个partition中唯一标识某个message的sequential id
2. 一个Topic可以有多个partition, 并且可以指定副本数 <== 均衡负载，多台机器同时处理,容错性
3 .创建topic时可指定parition数量
4. 每个partition对应于一个文件夹，该文件夹下存储该partition的数据和索引文件

kafka 应用场景:
分布式应用的数据监控.
分布式应用的日志收集.
流处理(kafka+ storm)-实时处理
作为分布式系统额外的commitlog

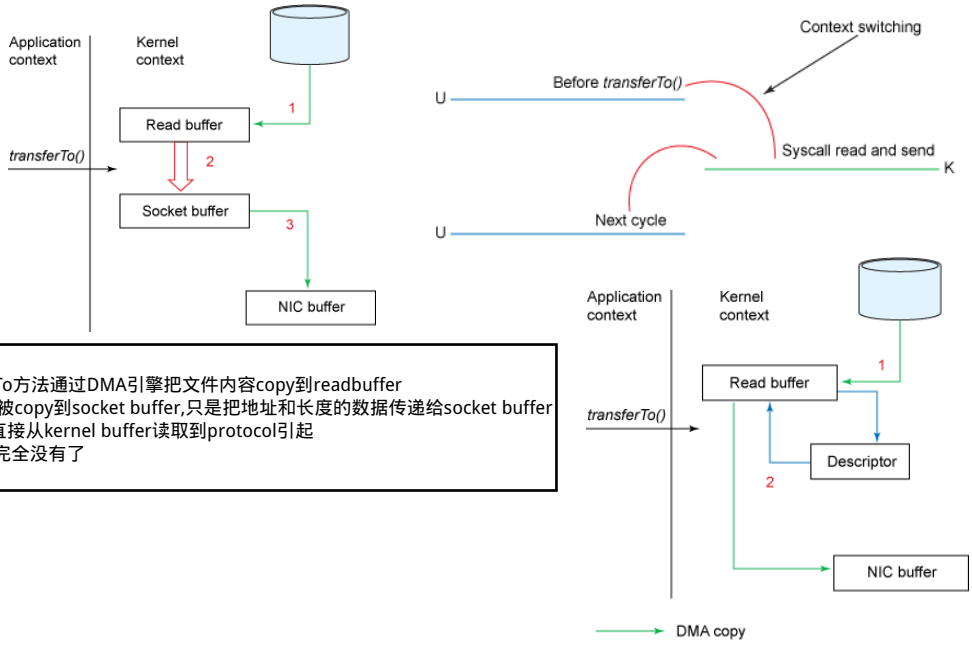
Base Part 2： 消息持久 & why fast

1. 线性写入 <== 6个7200rpm的SATA硬盘组成的RAID-5磁盘,300MB/秒
2.数据结构：简单地向文件中添加内容 <== 操作的复杂度都是O(1)， 读写操作互补影响
3.零拷贝: sendfile 传输 (Java中的API，FileChannel.transferTo)
4. 端到端的批量压缩 == > 数据在消息生产者发送之前先压缩一下，然后在服务器上一直保存压缩状态，只有到最终的消息使用者那里才需要将其解压缩 ----kafka支持GZIP和Snappy压缩
5. consumer 维持使用状态信息,数据通过pull方式从broker拉去

传统文件拷贝方式
1 . read() ==>从用户上下文到系统核心上下文,调用底层sys_read()从文件读取数据,第一次copy的执行通过Direct memory access引擎,把文件内容从硬盘保存到kernal address sapce buffer
2 请求的数据从kernel buffer 读取到 user buffer,给read()返回,这个过程引起有一次上下文转换
3. send() 方法再次触发上下文切换,第三次copy把数据放到kernel buffer(不同的buffer)
4. send()返回的时候,第四次上下文切换,通过DMA引擎把数据从kernel buffer 拷贝到 protocol 引擎



零拷贝方式
1. transferTo方法通过DMA引擎把文件内容copy到readbuffer,然后copy到关联的socket的kernel buffer
2. 最后DMA引擎吧socket buffer 拷贝到 protocol 引擎
减少两次上下文切换,减少一次cpu拷贝



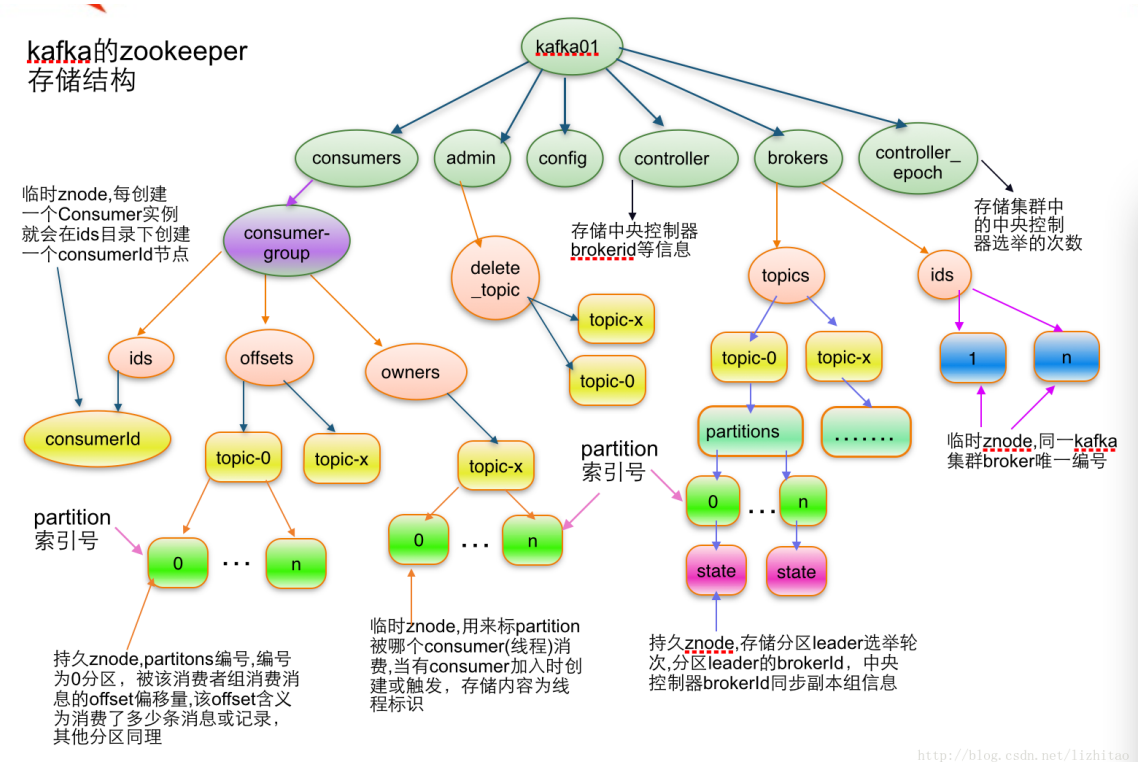
零拷贝方式
1. transferTo方法通过DMA引擎把文件内容copy到readbuffer
2 没有数据被copy到socket buffer,只是把地址和长度的数据传递给socket buffer,DMA引擎直接从kernel buffer读取到protocol引起
cpu的拷贝完全没有了

Base Part 3: kafka & 分布式 & ZooKeeper

Kafka ==> 分布式系统 —— 生产者、使用者和代理都可以运行在作为一个逻辑单位的、进行相互协作的集群中不同的机器上。

集群协调？ <== zookeeper
一个进程的多个线程之间需要同步操作[lock,synchronized]
多个不同服务器上的进程同步操作[zookeeper]
==========zookeeper基本作用==========
1 每个partition zookeeper从集群中选举出一台server的broker作为leader,其他做follower
2 leader负责处理所有对于这个partition的读写请求,follower负责复制到自己的副本 <== 容错性
3 每台server都会作为某些partition的leader <== 均衡负载
4 某台server出问题,会为partition选举新的leader负责读写请求 <== 容错性
==========
CAP？ == > AP

```
object ZkUtils extends Logging {
 val ConsumersPath = "/consumers"
 val BrokerIdsPath = "/brokers/ids"
 val BrokerTopicsPath = "/brokers/topics"
 val ControllerPath = "/controller"
 val ControllerEpochPath = "/controller_epoch"
 val ReassignPartitionsPath = "/admin/reassign_partitions"
 val DeleteTopicsPath = "/admin/delete_topics"
 val PreferredReplicaLeaderElectionPath = "/admin/preferred_replica_election"
 val BrokerSequenceIdPath = "/brokers/seqid"
 val IsrChangeNotificationPath = "/isr_change_notification"
 val EntityConfigPath = "/config"
 val EntityConfigChangesPath = "/config/changes"
```



1.topic注册信息
/brokers/topics/[topic] :
存储某个topic的partitions所有分配信息

Schema:

```
{
   "version": "版本编号目前固定为数字1",
   "partitions": {
      "partitionId编号": [
         同步副本组brokerId列表
      ],
      .....
   }
}
```

Example:

```
{
"version": 1,
"partitions": {
"0": [1, 2],
"1": [2, 1],
"2": [1, 2],
}
}
```

2.partition状态信息
/brokers/topics/[topic]/partitions/[0...N]
其中[0..N]表示partition索引号
/brokers/topics/[topic]/partitions/[partitionId]/state
Schema:

```
{
"controller_epoch": 表示kafka集群中的中央控制器选举次数,
"leader": 表示该partition选举leader的brokerId,
"version": 版本编号默认为1,
"leader_epoch": 该partition leader选举次数,
"isr": [同步副本组brokerId列表]
}
```

Example:

```
{
"controller_epoch": 1,
"leader": 2,
"version": 1,
"leader_epoch": 0,
"isr": [2, 1]
}
```

3. Broker注册信息
/brokers/ids/[0...N]
每个broker的配置文件中都需要指定一个数字类型的id(全局不可重复),此节点为临时znode(EPHEMERAL)
Schema:

```
{
"jmx_port": jmx端口号,
"timestamp": kafka broker初始启动时的时间戳,
"host": 主机名或ip地址,
"version": 版本编号默认为1,
"port": kafka broker的服务端端口号,
由server.properties中参数port确定
}
```

Example:

```
{
"jmx_port": 6061,

"timestamp":"1403061899859"
"version": 1,
"host": "192.168.1.148",
"port": 9092
}
```

4. Controller epoch:
/controller_epoch -> int (epoch)
此值为一个数字,kafka集群中第一个broker第一次启动时为1，
以后只要集群中center controller中央控制器所在broker变更或挂掉，就会重新选举新的center controller，每次center controller变更controller_epoch值就会 + 1;

5. Controller注册信息:
/controller -> int (broker id of the controller)
存储center controller中央控制器所在kafka broker的信息

Schema:

```
{
"version": 版本编号默认为1,
"brokerid": kafka集群中broker唯一编号,
"timestamp": kafka broker中央控制器变更时的时间戳
}
```

Example:

```
{
"version": 1,
"brokerid": 3,
"timestamp": "1403061802981"
}
```

6. Consumer注册信息:
每个consumer都有一个唯一的ID
(consumerId可以通过配置文件指定
,也可以由系统生成),此id用来标记消费者信息.
/consumers/[groupId]/ids/[consumerIdString]
是一个临时的znode,
即表示此consumer目前所消费的topic + partitions列表.
Schema:

{
"version": 版本编号默认为1,
"subscription": { //订阅topic列表
"topic名称": consumer中topic消费者线程数
},
"pattern": "static",
"timestamp": "consumer启动时的时间戳"
}

Example:
{
"version": 1,
"subscription": {
"open_platform_opt_push_plus1": 5
},
"pattern": "static",
"timestamp": "1411294187842"
}

7. Consumer owner:
/consumers/[groupId]/owners/[topic]/[partitionId] ->
consumerIdString + threadId索引编号
当consumer启动时,所触发的操作:
a) 首先进行"Consumer Id注册";
b) 然后在"Consumer id 注册"节点下注册一个watch
用来监听当前group中其他consumer的"退出"和"加入";
只要此znode path下节点列表变更,
都会触发此group下consumer的负载均衡.
(比如一个consumer失效
,那么其他consumer接管partitions).
c) 在"Broker id 注册"节点下,
注册一个watch用来监听broker的存活情况
;如果broker列表变更,
将会触发所有的groups下的consumer重新balance.

8. Consumer offset:
/consumers/[groupId]/offsets/[topic]/[partitionId]
-> long (offset)
用来跟踪每个consumer目前所消费的partition中最大的offset
此znode为持久节点,可以看出offset跟group_id有关
,以表明当消费者组(consumer group)中一个消费者失效,
重新触发balance,其他consumer可以继续消费.

Zookeeper 问题:
Customer实例之间可能会出现的 脑裂现象
脑裂就是集群内各节点间的心跳出现故障，但各节点还处于active状态，
多个节点分别接管服务并且写入共享文件资源导致数据损坏或者其它问题。

Base Part 4 : 项目使用

consumer

xxx.properties:
```
zookeeper.connect=122.144.134.67:2181
group.id=cassandra-logger-consumer
request.required.acks=1
zookeeper.session.timeout.ms=40000
zookeeper.sync.time.ms=2000
auto.commit.enable = true
auto.commit.interval.ms=1000
partition.assignment.strategy=range
```

xxxx.xml
```
<bean id="allLogConsumer"
  class="com.moneylocker.common.kafka.api.consumer.KafkaConsumer"
            init-method="start" destroy-method="close">
            <property name="properties">
                  <util:properties location="all-log-consumer.properties" />
            </property>
            <property name="messageHandlerBuilder" ref="logMessageHandlerBuilder" />
            <property name="topicMap">
                  <util:map map-class="java.util.HashMap">
                        <entry key="ml_obtain_credit_log" value="4" />
                        <entry key="ml_ad_right_slide_log" value="4"/>
                  </util:map>
            </property>
      </bean>
```

KafkaConsumer
----------start()----------
1. 连接zookeeper,注册consumer信息
 ConsumerConnector consumer = Consumer.createJavaConsumerConnector(new ConsumerConfig(properties));

2. 根据配置的topic名字和指定的consumer的数量,从zookeeper拉去对应topic的信息,建立stream
一个stream就是一个consumer和某个topic的partition的链接,一个group的同一个topic的consumer的
数量小于等于topic的partition数,
Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicMap);

3.为每个stream启动一个线程并新建一个msg handler处理消息
executor.submit(new ConsumerHandler(stream, messageHandlerBuilder.build(), topic));

4.messageHandler接受并处理消息
messageHandler.onMessage(messageAndMetadata, topic);

Producer

kafka-producer-async.properties
```
metadata.broker.list=122.144.134.82:9092,122.144.134.67:9092
serializer.class=kafka.serializer.StringEncoder
request.required.acks=1
#partitioner.class
#async config
producer.type=async
batch.num.messages=50
```

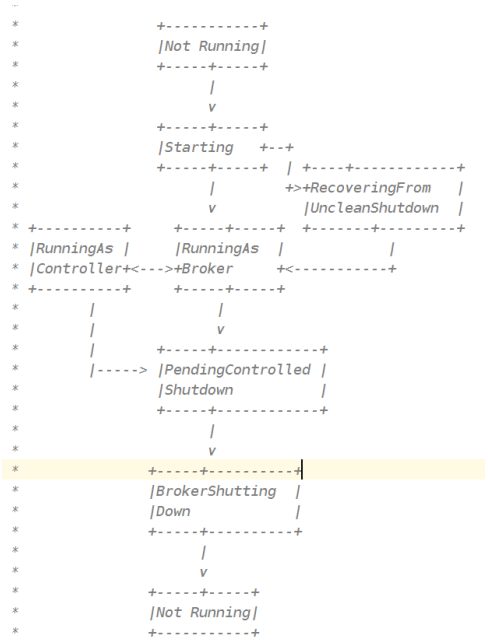
KafkaProducer
1.Producer producer = new Producer(new ProducerConfig(props));
2. send(String topic, String key, V value)

业务内容:
1.积分明细,左划,右划,曝光,安装,登录,消息验证日志收集到cassandra <== 日志收集
2.同一设备重复切换帐号检测 <== 异步处理
3.邀请作弊检测 <== 异步处理
4.根据手机号码查询用户区域 <== 异步处理
5.收益+任务更新 <== 异步处理

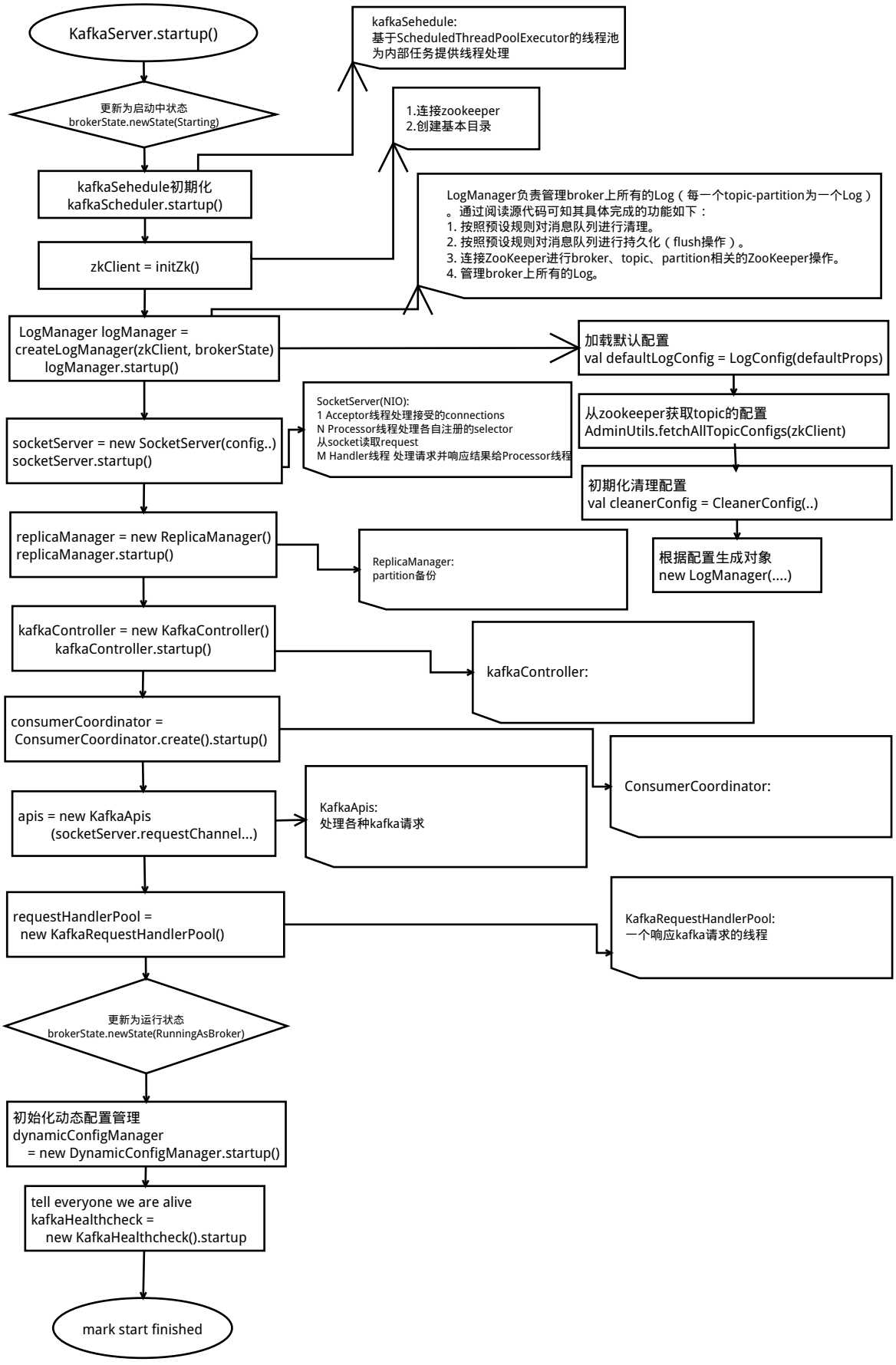
Source Part 1 : Kafka --------- broker

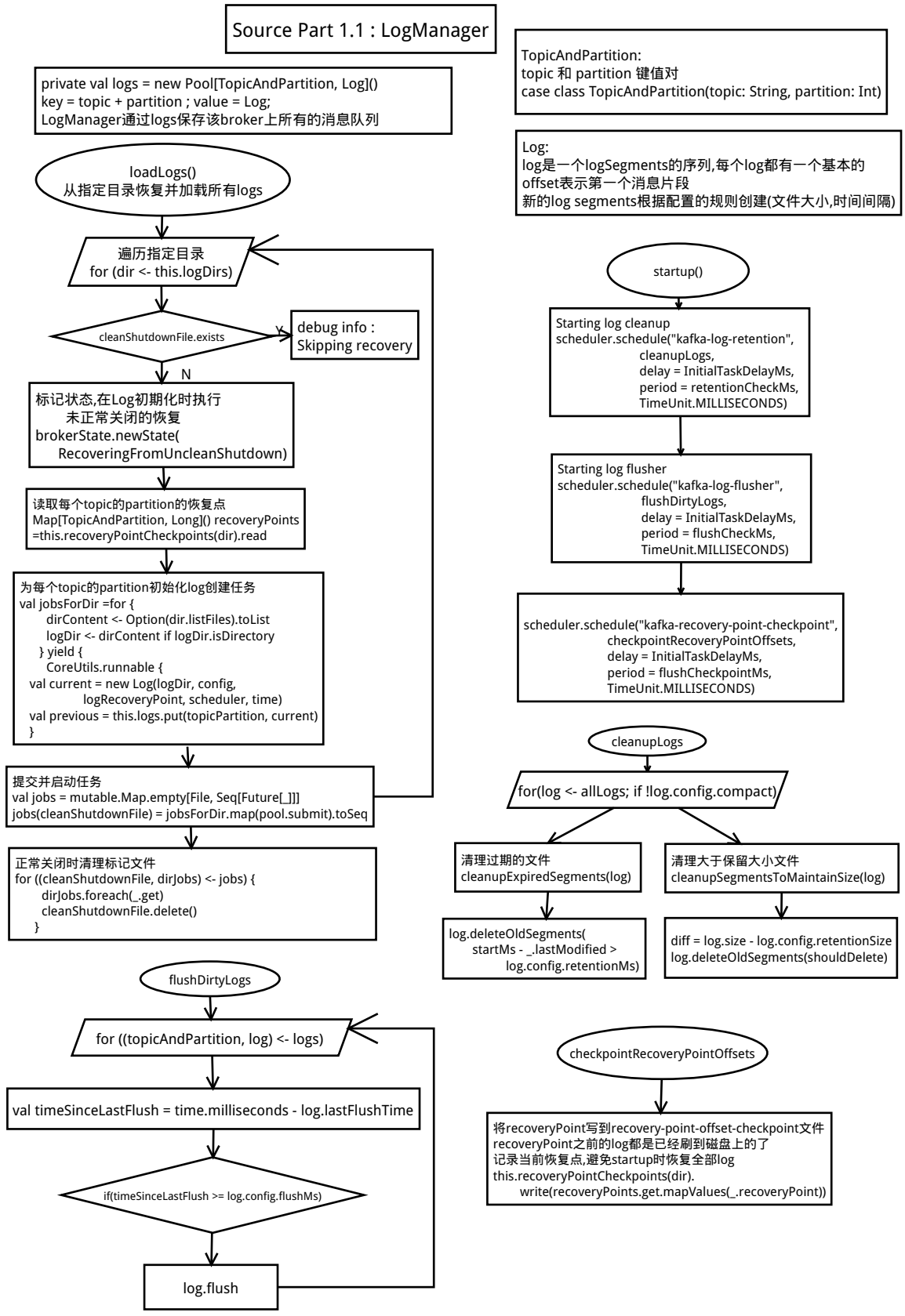
1.kafka server states

```
sealed trait BrokerStates { def state: Byte }
case object NotRunning extends BrokerStates { val state: Byte = 0 }
case object Starting extends BrokerStates { val state: Byte = 1 }
case object RecoveringFromUncleanShutdown extends BrokerStates { val state: Byte = 2 }
case object RunningAsBroker extends BrokerStates { val state: Byte = 3 }
case object RunningAsController extends BrokerStates { val state: Byte = 4 }
case object PendingControlledShutdown extends BrokerStates { val state: Byte = 6 }
case object BrokerShuttingDown extends BrokerStates { val state: Byte = 7 }
```



NotRunning : 未运行 初始状态
Starting : 开始运行 <---startup()


KafkaServer.startup()
更新为启动中状态
brokerState.newState(Starting)
kafkaSehedule初期化
kafkaScheduler.startup()
zkClient = initZk()
LogManager logManager =
createLogManager(zkClient, brokerState)
logManager.startup()
socketServer = new SocketServer(config..)
socketServer.startup()
replicaManager = new ReplicaManager()
replicaManager.startup()
kafkaController = new KafkaController()
kafkaController.startup()
consumerCoordinator =
ConsumerCoordinator.create().startup()
apis = new KafkaApis
(socketServer.requestChannel...)
requestHandlerPool =
new KafkaRequestHandlerPool()
更新为运行状态
brokerState.newState(RunningAsBroker)
初始化动态配置管理
dynamicConfigManager
= new DynamicConfigManager.startup()
tell everyone we are alive
kafkaHealthcheck =
new KafkaHealthcheck().startup
mark start finished
kafkaSehedule:
基于ScheduledThreadPoolExecutor的线程池
为内部任务提供线程处理
1.连接zookeeper
2.创建基本目录
LogManager负责管理broker上所有的Log（每一个topic-partition为一个Log）
。通过阅读源代码可知其具体完成的功能如下：
1. 按照预设规则对消息队列进行清理。
2. 按照预设规则对消息队列进行持久化（flush操作）。
3. 连接ZooKeeper进行broker、topic、partition相关的ZooKeeper操作。
4. 管理broker上所有的Log。
加载默认配置
val defaultLogConfig = LogConfig(defaultProps)
从zookeeper获取topic的配置
AdminUtils.fetchAllTopicConfigs(zkClient)
初期化清理配置
val cleanerConfig = CleanerConfig(..)
根据配置生成对象
new LogManager(....)
SocketServer(NIO):
1 Acceptor线程处理接受的connections
N Processor线程处理各自注册的selector
从socket读取request
M Handler线程 处理请求并响应结果给Processor线程
ReplicaManager:
partition备份
kafkaController:
ConsumerCoordinator:
KafkaApis:
处理各种kafka请求
KafkaRequestHandlerPool:
一个响应kafka请求的线程

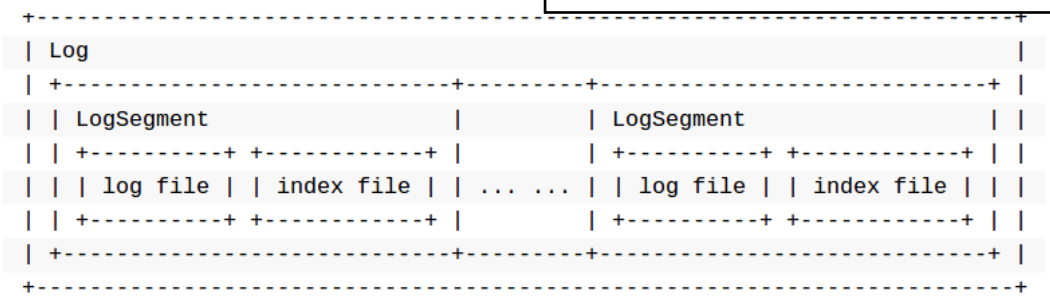

Source Part 1.1 : LogManager
TopicAndPartition:
topic 和 partition 键值对
case class TopicAndPartition(topic: String, partition: Int)
private val logs = new Pool[TopicAndPartition, Log]()
key = topic + partition ; value = Log;
LogManager通过logs保存该broker上所有的消息队列
Log:
log是一个logSegments的序列,每个log都有一个基本的offset表示第一个消息片段
新的log segments根据配置的规则创建(文件大小,时间间隔)
loadLogs()
从指定目录恢复并加载所有logs
遍历指定目录
for (dir <- this.logDirs)
cleanShutdownFile.exists
Y
debug info :
Skipping recovery
N
标记状态,在Log初期化时执行
未正常关闭的恢复
brokerState.newState(
RecoveringFromUncleanShutdown)
读取每个topic的partition的恢复点
Map[TopicAndPartition, Long]() recoveryPoints
=this.recoveryPointCheckpoints(dir).read
为每个topic的partition初始化log创建任务
val jobsForDir =for {
dirContent <- Option(dir.listFiles).toList
logDir <- dirContent if logDir.isDirectory
} yield {
CoreUtils.runnable {
val current = new Log(logDir, config,
logRecoveryPoint, scheduler, time)
val previous = this.logs.put(topicPartition, current)
}
提交并启动任务
val jobs = mutable.Map.empty[File, Seq[Future[_]]]
jobs(cleanShutdownFile) = jobsForDir.map(pool.submit).toSeq
正常关闭时清理标记文件
for ((cleanShutdownFile, dirJobs) <- jobs) {
dirJobs.foreach(_.get)
cleanShutdownFile.delete()
}
startup()
Starting log cleanup
scheduler.schedule("kafka-log-retention",
cleanupLogs,
delay = InitialTaskDelayMs,
period = retentionCheckMs,
TimeUnit.MILLISECONDS)
Starting log flusher
scheduler.schedule("kafka-log-flusher",
flushDirtyLogs,
delay = InitialTaskDelayMs,
period = flushCheckMs,
TimeUnit.MILLISECONDS)
scheduler.schedule("kafka-recovery-point-checkpoint",
checkpointRecoveryPointOffsets,
delay = InitialTaskDelayMs,
period = flushCheckpointMs,
TimeUnit.MILLISECONDS)
cleanupLogs
for(log <- allLogs; if !log.config.compact)
清理过期的文件
cleanupExpiredSegments(log)
清理大于保留大小文件
cleanupSegmentsToMaintainSize(log)
log.deleteOldSegments(
startMs - _.lastModified >
log.config.retentionMs)
diff = log.size - log.config.retentionSize
log.deleteOldSegments(shouldDelete)
flushDirtyLogs
for ((topicAndPartition, log) <- logs)
val timeSinceLastFlush = time.milliseconds - log.lastFlushTime
if(timeSinceLastFlush >= log.config.flushMs)
log.flush
checkpointRecoveryPointOffsets
将recoveryPoint写到recovery-point-offset-checkpoint文件
recoveryPoint之前的log都是已经刷到磁盘上的了
记录当前恢复点,避免startup时恢复全部log
this.recoveryPointCheckpoints(dir).
write(recoveryPoints.get.mapValues(_.recoveryPoint))

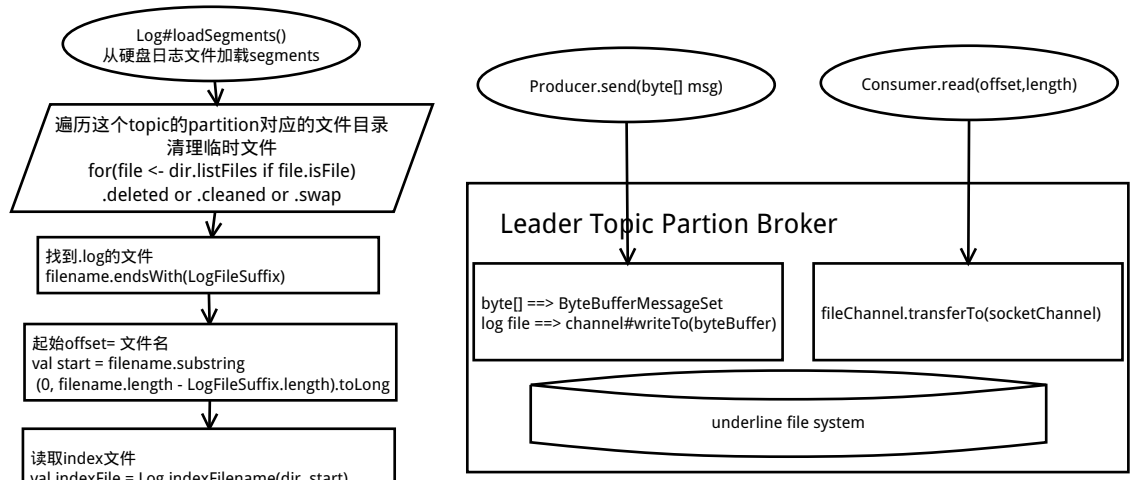
Source Part 1.1.1 Log#Field#init

```
/* the actual segments of the log */
private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment]
= new ConcurrentSkipListMap[java.lang.Long, LogSegment]
LogSegment==>
log的一个segment,每个segment有2个组件,log和index
log包含了实际的消息,index是一个offsetIndex,
映射逻辑offsets到物理文件位置
每个segment有一个基本的base_offset,
base_offset<=这个segment的任何message的offset
base_offset>任何之前的segment
base_offset会存储在2个文件中
[base_offset].index
[base_offset].log
```

```
/tmp/kafka-logs-1/
├── hello-1
│   ├── 00000000000000000000.index
│   └── 00000000000000000000.log
├── hello-2
│   ├── 00000000000000000000.index
│   └── 00000000000000000000.log
├── recovery-point-offset-checkpoint
└── replication-offset-checkpoint
```

recovery-point-offset-checkpoint:
记录已经刷新到磁盘的文件log位置
replication-offset-checkpoint:
存储每个replica的HighWatermark的，定时写由ReplicaManager执行
HW是last committed message的起始偏broker的recovery需要用到H





OffsetIndex#resize
重置内存映射和底层的文件大小
使用场景:
1.closing the segment or new segment being rolled
2.loading segments from disk or truncating back to
an old segment where a new log segment became active
目的:
reset the index size to maximum index
size to avoid rolling new segment

Log#append
添加message到当前活动log的segment,在需要的时候(大小,时间)rolling生成新的segment

Log#append

消息验证,封装AppendLogInfo
appendInfo = analyzeAndValidateMessageSet(messages)

validMessages = trimInvalidBytes(messages, appendInfo)

lock synchronized

根据当前正在使用的segment的offset设定消息的开始offset
appendInfo.firstOffset = nextOffsetMetadata.messageOffset
//messageOffset=>activeSegment.nextOffset()

消息offset验证,总长度验证

检查segment是否满了需要生成新的
val segment = maybeRoll(validMessages.sizeInBytes)

A log offset structure, including:
 * 1. the message offset
 * 2. the base message offset of the located segment
 * 3. the physical position on the located segment
nextOffsetMetadata = new LogOffsetMetadata(
  activeSegment.nextOffset(), activeSegment.baseOffset
    , activeSegment.size.toInt)

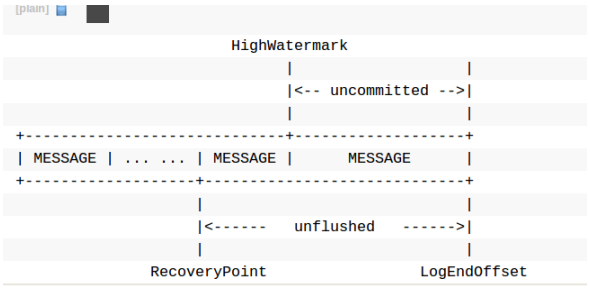
Log#maybeRoll
 The logSegment is full
 The maxTime has elapsed
 The index is full
生成一个新的segment

segment.append(appendInfo.firstOffset, validMessages)

备份的同步位置
highWatermark==>last committed message的起始偏移。broker的recovery需要用到HW
recoveryPoint==>channel中是否刷新到disk的分界点
logEndOffset==>这个channel最后一个entry的offset

updateLogEndOffset(appendInfo.lastOffset + 1)

def unflushedMessages() =
this.logEndOffset - this.recoveryPoin
  >= config.flushInterval



1 . if (offset <= this.recoveryPoint){
//将recoveryPoint到logEndOffset的LogSegment刷到磁盘
for(segment <- logSegments(this.recoveryPoint, offset))
    segment.flush()}
2.更新recoveryPoint为logEndOffset
if(offset > this.recoveryPoint) {
      this.recoveryPoint = offset
      lastflushedTime.set(time.milliseconds)
    }

Log#read
Read messages from the log

检查读取的offset有效性
找出LogSegment，也就是XXXXXXX.log跟XXXXXXX.index

//continue to read from successive segments
// until we get some messages or we reach the end of the log
while(entry != null) {
    val fetchInfo = entry.getValue.read(startOffset, maxOffset, maxLength)
    if(fetchInfo == null) {
      entry = segments.higherEntry(entry.getKey)
    } else {
      return fetchInfo
    }
  }

从开始位置到最后都没有读到任何消息(被删除了)返回空消息集合
FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)

Log#flush()
缓存在fileChannel中的data刷新到disk

for(segment <- logSegments(this.recoveryPoint, offset))
    segment.flush()

if(offset > this.recoveryPoint) {
      this.recoveryPoint = offset
      lastflushedTime.set(time.milliseconds)
    }

Source Part 1.1.2 : LogSegment#field#init

```
* @param log The message set containing log entries
* @param index The offset index
* @param baseOffset A lower bound on the offsets in this segment
* @param indexIntervalBytes The approximate number of bytes between entries in the index
* @param time The time instance
def this(dir: File, startOffset: Long, indexIntervalBytes: Int, maxIndexSize: Int, rollJitterMs: Long, time: Time
                  , fileAlreadyExists: Boolean = false, initFileSize: Int = 0, preallocate: Boolean = false) =
    this(new FileMessageSet(file = Log.logFilename(dir, startOffset), fileAlreadyExists = fileAlreadyExists
                  , initFileSize = initFileSize, preallocate = preallocate),
         new OffsetIndex(file = Log.indexFilename(dir, startOffset), baseOffset = startOffset, maxIndexSize = maxIndexSize),
         startOffset,
         indexIntervalBytes,
         rollJitterMs,
         time)
读取log,index文件
```

```
FileMessageSet#init
def this(file: File, fileAlreadyExists: Boolean, initFileSize: Int
  , preallocate: Boolean) =
    this(file,
      channel = FileMessageSet.openChannel(file, mutable = true
        , fileAlreadyExists, initFileSize, preallocate),
      start = 0,
      end = ( if ( !fileAlreadyExists && preallocate ) 0 else Int.MaxValue),
      isSlice = false)
```

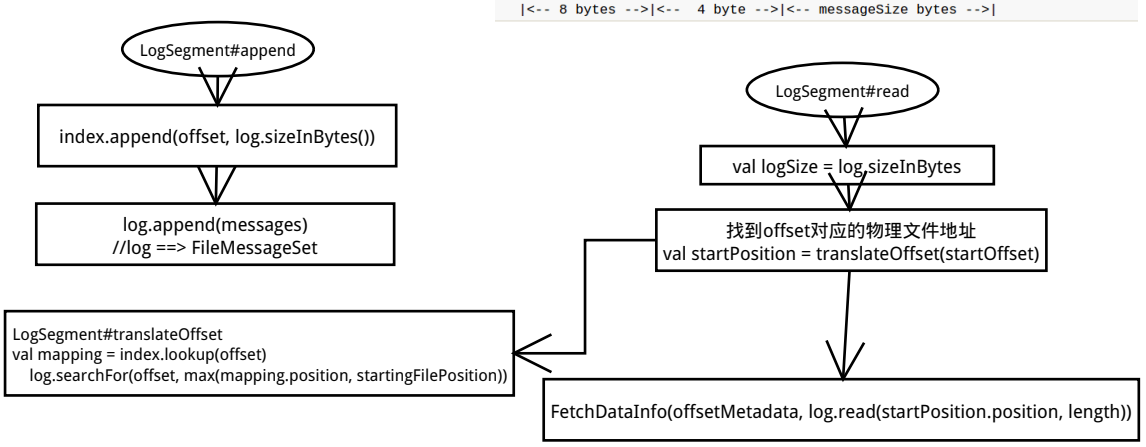
```
OffsetIndex#init
1 为某个log segment提供逻辑offsets到物理文件地址映射的索引
2 支持从文件的memory-map的二分查找offset/location pair
3 索引文件被打开的两种方式:
3.1 可变的允许追加的方式
3.2 不可变的只读的方式
4 这个文件的格式是一系列entries,物理存储格式:
4.1 4个字节的相对offset和4个字节这个message的文件location
```

```
FileMessageSet#openChannel
 new RandomAccessFile(file, "rw").getChannel()

 val MessageSizeLength = 4
 val OffsetLength = 8
 val LogOverhead = MessageSizeLength + OffsetLength
```

```
index entry format:
  +--------------------------------------------+
  |    relativeOffset=      |  physicalPosition  |
  | targetOffset-baseOffset |                    |  ... ...
  +-------------------------+--------------------+
  |<---  long, 8 bytes  --->|<-- int, 4 bytes -->|

log entry format:
  +-----------------------------------------------------------+
  |  targetOffset |  messageSize  |          MESSAGE          |  ... ...
  +---------------+---------------+---------------------------+
  |<-- 8 bytes -->|<--  4 byte -->|<-- messageSize bytes -->|
```



Source Part 1.1.3 ：FileMessageSet
@param file The file name for the underlying log data
 * @param channel the underlying file channel used
 * @param start A lower bound on the absolute position in the file from which the message set begins
 * @param end The upper bound on the absolute position in the file at which the message set ends
 * @param isSlice Should the start and end parameters be used for slicing?

FileMessageSet#append

```
ByteBufferMessageSet#writeTo(channel, 0, messages.sizeInBytes)
buffer.mark()
   var written = 0
   while(written < sizeInBytes)
     written += channel.write(buffer)
   buffer.reset()
   written
// val buffer = ByteBuffer.allocate(...)
```

FileMessageSet#read

在原来的FileMessageSet的基础是选取指定开始和长度的内容
产生新的FileMessageSet
new FileMessageSet(file, channel, start = this.start + position,
 end = math.min(this.start + position + size, sizeInBytes()))

FileMessageSet#flush
channel.force(true)

FileMessageSet#writeTo
zeroCopy

```
  val bytesTransferred = channel.transferTo(start + writePosition, math.min(size, sizeInBytes), destChannel).toInt
  trace("FileMessageSet " + file.getAbsolutePath + " : bytes transferred : " + bytesTransferred
    + " bytes requested for transfer : " + math.min(size, sizeInBytes))
  bytesTransferred
```

Java NIO中的FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。
FileChannel.read() ==>将数据从FileChannel读取到Buffer中
FileChannel.write()==>向FileChannel写数据
FileChannel.force()==>将通道里尚未写入磁盘的数据强制写到磁盘上
操作系统会将数据缓存在内存中，所以无法保证写入到FileChannel里的数据一定会即时写到磁盘上
FileChannel.map()==>将文件映射到内存

OffsetIndex#init

OffsetIndex#mmap
将文件映射到内存

```
val raf = new RandomAccessFile(file, "rw")
```

```
/* memory-map the file */
 val len = raf.length()
val idx = raf.getChannel.map(
FileChannel.MapMode.READ_WRITE, 0, len)
//MappedByteBuffer
```

```
/* the number of eight-byte entries currently in the index */
  private var size = new AtomicInteger(mmap.position / 8)
```

```
/* return the nth offset relative to the base offset */
 private def relativeOffset(buffer: ByteBuffer, n: Int): Int = buffer.getInt(n * 8)
```

```
/* return the nth physical position */
 private def physical(buffer: ByteBuffer, n: Int): Int = buffer.getInt(n * 8 + 4)
```

OffsetIndex#lookup
根据offset找对应的文件位置

```
val idx = mmap.duplicate
    val slot = indexSlotFor(idx, targetOffset)//二分查找
    if(slot == -1)
      OffsetPosition(baseOffset, 0)
    else
      OffsetPosition(baseOffset + relativeOffset(idx, slot), physical(idx, slot))
    }
```

OffsetIndex#append

```
this.mmap.putInt((offset - baseOffset).toInt)
this.mmap.putInt(position)
this.size.incrementAndGet()
this.lastOffset = offset
```

OffsetIndex#flush
MappedByteBuffer#force()

MappedByteBuffer#force()==> 刷新缓存到文件
MappedByteBuffer@buffer#putInt()==>写在缓存中

Source Part 1.2： SocketServer

Kafka SocketServer是基于Java NIO来开发的，采用了Reactor的模式，其中包含了1个Acceptor负责接受客户端请求，N个Processor负责读写数据，M个Handler来处理业务逻辑。在Acceptor和Processor，Processor和Handler之间都有队列来缓冲请求

```
private val processors = new Array[Processor](numProcessorThreads)
//多个processor线程通过单个connection并行处理所有请求
 private[network] var acceptors =  mutable.Map[EndPoint,Acceptor]()
//接受请求并配置生成新的connections的线程
 val requestChannel = new RequestChannel(numProcessorThreads, maxQueuedRequests)
//RequestChannel是Processor和Handler交换数据的地方
```

SocketServer#startup()

```
//创建并启动Processor线程
for (i <- 0 until numProcessorThreads) {
       processors(i) = new Processor(i,
   .....
         )
      Utils.newThread("kafka-network-thread-%d-%d"
         .format(brokerId, i)
       , processors(i), false).start()
  }
```

```
// register the processor threads for notification of responses
   requestChannel.addResponseListener
     ((id:Int) => processors(id).wakeup())
```

```
this.synchronized {
     endpoints.values.foreach(endpoint => {
       val acceptor = new Acceptor(endpoint.host...)
       acceptors.put(endpoint, acceptor)
       Utils.newThread("kafka-socket-acceptor-%s-%d"
         .format(endpoint.protocolType.toString, endpoint.port)
       , acceptor, false).start()
       acceptor.awaitStartup
     })
   }
```

Acceptor#init#field

```
val nioSelector = java.nio.channels.Selector.open()
 val serverChannel = openServerSocket(host, port)
 portToProtocol.put(serverChannel.socket().getLocalPort, protocol)
```

Acceptor#run

```
serverChannel.register(nioSelector, SelectionKey.OP_ACCEPT);
```

```
Record that the thread startup is complete
startupLatch.countDown()
//private val startupLatch = new CountDownLatch(1)
```

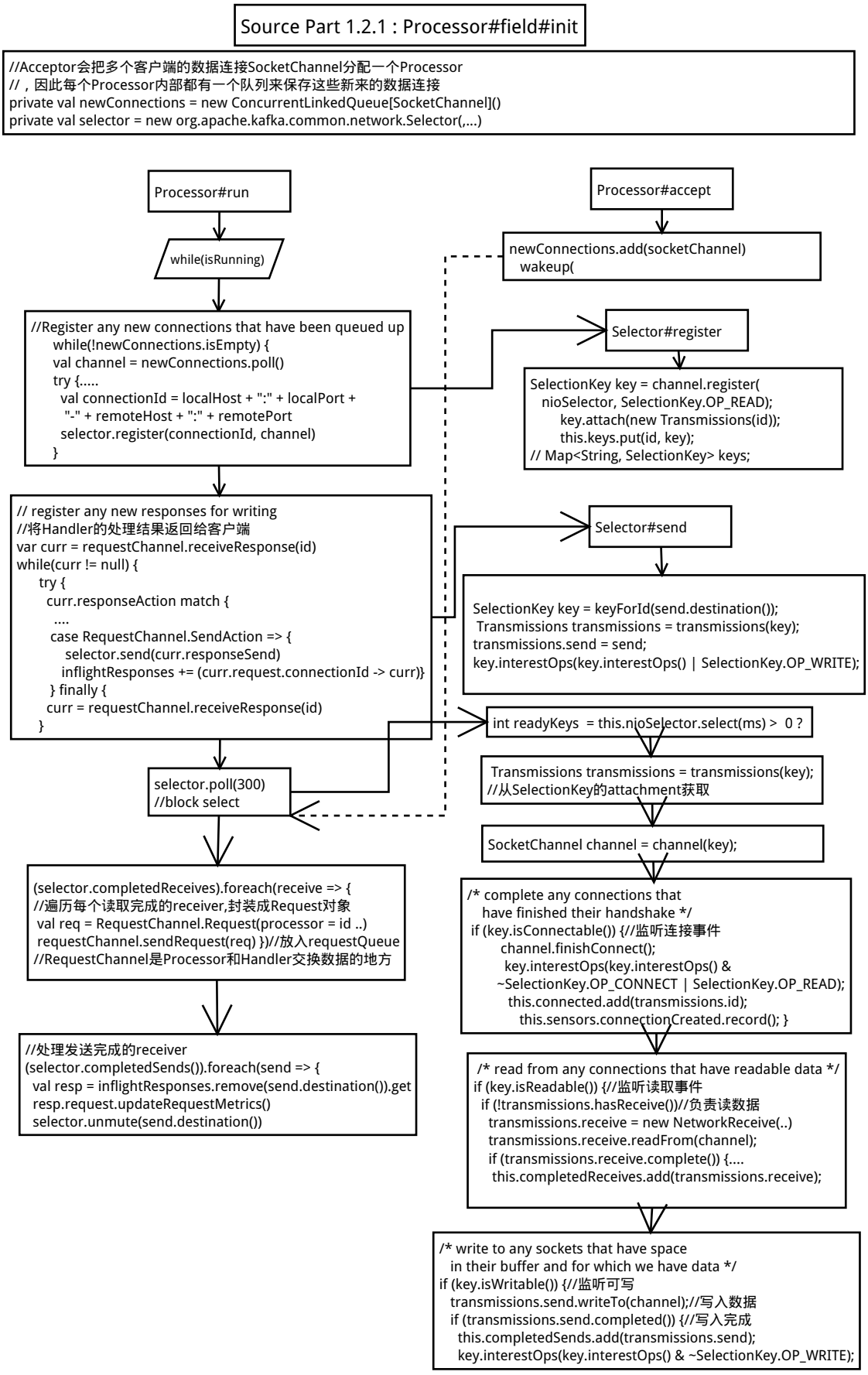
```
var currentProcessor = 0
 while(isRunning) {
     val ready = nioSelector.select(500)
     if(ready > 0) {
       val keys = nioSelector.selectedKeys()
       val iter = keys.iterator()
       while(iter.hasNext && isRunning) {
         var key: SelectionKey = null
         try {
           key = iter.next
           iter.remove()
           if(key.isAcceptable)
             accept(key, processors(currentProcessor))
           else
              throw new IllegalStateException("Unrecognized key.."
           // round robin to the next processor thread
           currentProcessor = (currentProcessor + 1) % processors.length
         } catch { ...}
```

Acceptor#accept

```
val serverSocketChannel = key.channel()
       .asInstanceOf[ServerSocketChannel]
   val socketChannel = serverSocketChannel.accept()
```

```
socketChannel.configureBlocking(false)
     socketChannel.socket().setTcpNoDelay(true)
     socketChannel.socket().setKeepAlive(true)
     socketChannel.socket().setSendBufferSize(sendBufferSize)
```

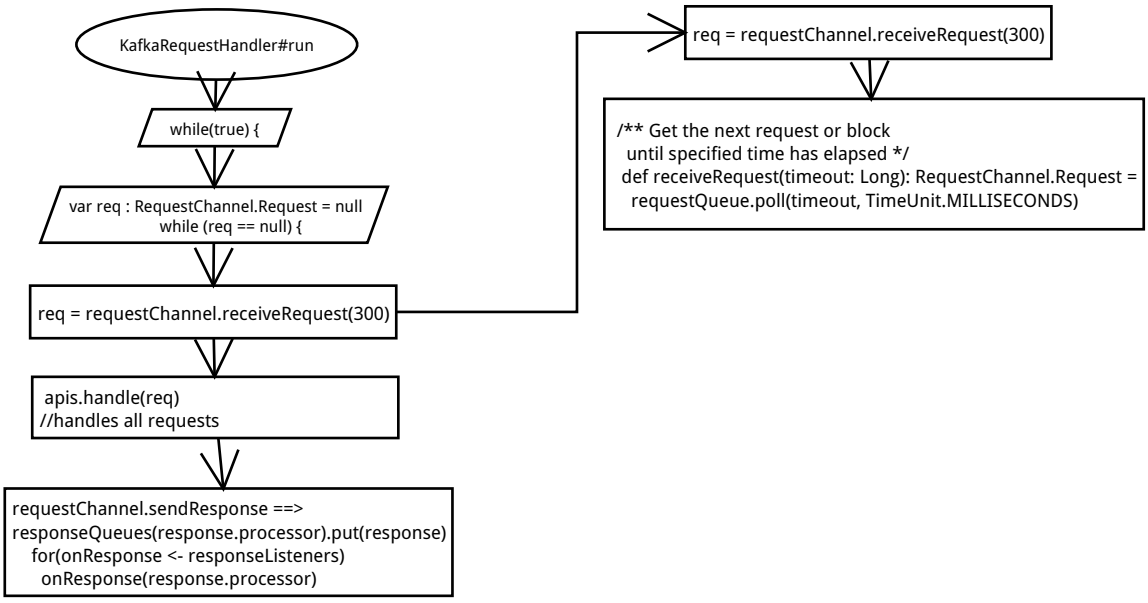
```
processor.accept(socketChannel)
```

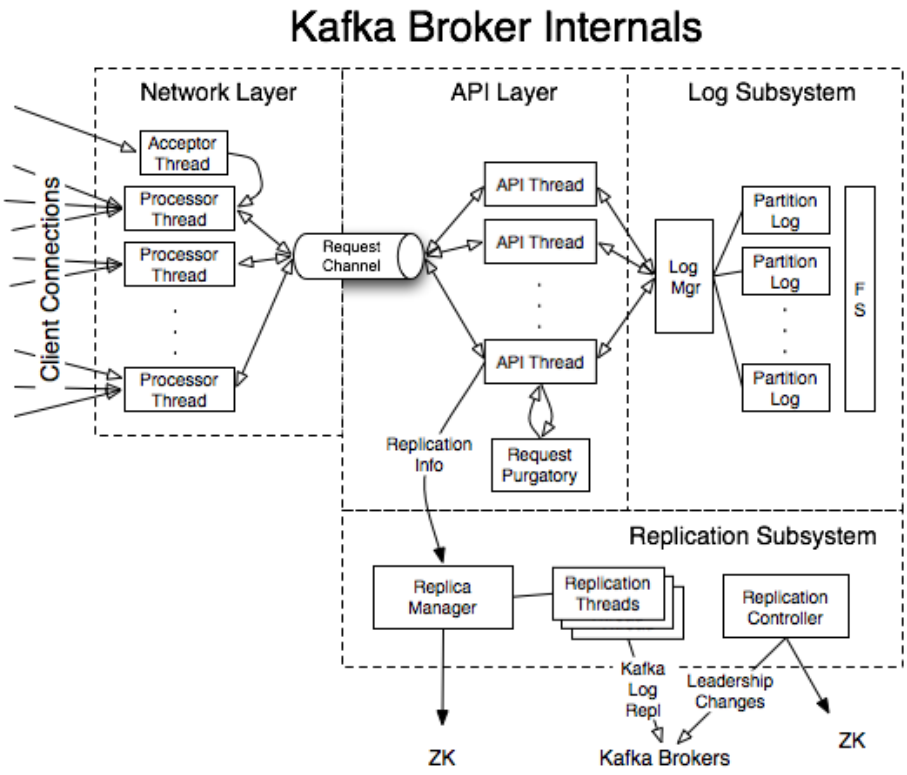

Source Part 1.2.1 : Processor#field#init
//Acceptor会把多个客户端的数据连接SocketChannel分配一个Processor
//，因此每个Processor内部都有一个队列来保存这些新来的数据连接
private val newConnections = new ConcurrentLinkedQueue[SocketChannel]()
private val selector = new org.apache.kafka.common.network.Selector(,...)
Processor#run
while(isRunning)
//Register any new connections that have been queued up
while(!newConnections.isEmpty) {
val channel = newConnections.poll()
try {.....
val connectionId = localHost + ":" + localPort +
"-" + remoteHost + ":" + remotePort
selector.register(connectionId, channel)
}
// register any new responses for writing
//将Handler的处理结果返回给客户端
var curr = requestChannel.receiveResponse(id)
while(curr != null) {
try {
curr.responseAction match {
....
case RequestChannel.SendAction => {
selector.send(curr.responseSend)
inflightResponses += (curr.request.connectionId -> curr)}
} finally {
curr = requestChannel.receiveResponse(id)
}
selector.poll(300)
//block select
(selector.completedReceives).foreach(receive => {
//遍历每个读取完成的receiver,封装成Request对象
val req = RequestChannel.Request(processor = id ..)
requestChannel.sendRequest(req) })//放入requestQueue
//RequestChannel是Processor和Handler交换数据的地方
//处理发送完成的receiver
(selector.completedSends()).foreach(send => {
val resp = inflightResponses.remove(send.destination()).get
resp.request.updateRequestMetrics()
selector.unmute(send.destination())
Processor#accept
newConnections.add(socketChannel)
wakeup(
Selector#register
SelectionKey key = channel.register(
nioSelector, SelectionKey.OP_READ);
key.attach(new Transmissions(id));
this.keys.put(id, key);
// Map<String, SelectionKey> keys;
Selector#send
SelectionKey key = keyForId(send.destination());
Transmissions transmissions = transmissions(key);
transmissions.send = send;
key.interestOps(key.interestOps() | SelectionKey.OP_WRITE);
int readyKeys = this.nioSelector.select(ms) > 0 ?
Transmissions transmissions = transmissions(key);
//从SelectionKey的attachment获取
SocketChannel channel = channel(key);
/* complete any connections that
have finished their handshake */
if (key.isConnectable()) {//监听连接事件
channel.finishConnect();
key.interestOps(key.interestOps() &
~SelectionKey.OP_CONNECT | SelectionKey.OP_READ);
this.connected.add(transmissions.id);
this.sensors.connectionCreated.record(); }
/* read from any connections that have readable data */
if (key.isReadable()) {//监听读取事件
if (!transmissions.hasReceive())//负责读数据
transmissions.receive = new NetworkReceive(..)
transmissions.receive.readFrom(channel);
if (transmissions.receive.complete()) {....
this.completedReceives.add(transmissions.receive);
/* write to any sockets that have space
in their buffer and for which we have data */
if (key.isWritable()) {//监听可写
transmissions.send.writeTo(channel);//写入数据
if (transmissions.send.completed()) {//写入完成
this.completedSends.add(transmissions.send);
key.interestOps(key.interestOps() & ~SelectionKey.OP_WRITE);

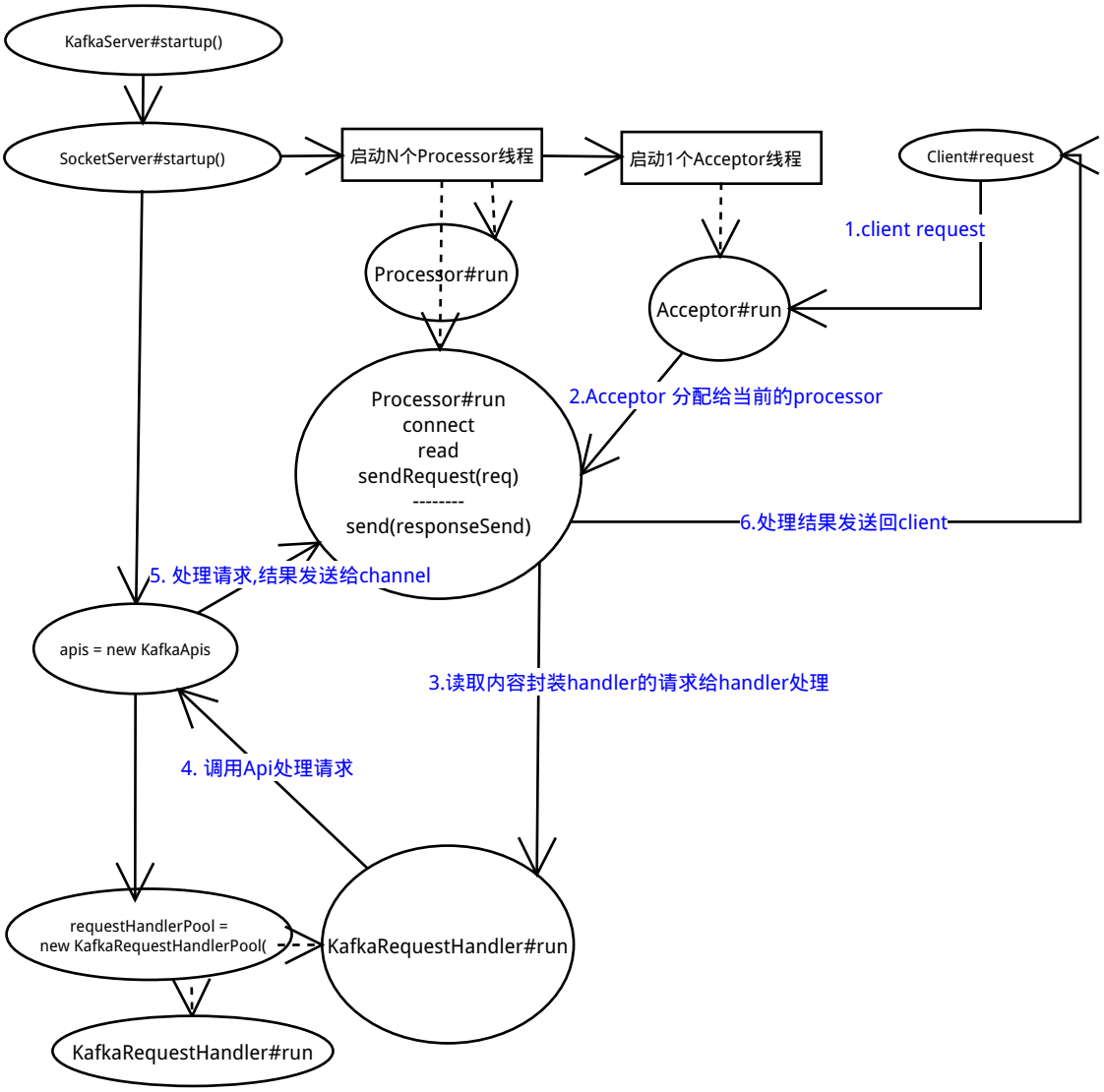
Source Part 1.2.2 : KafkaRequestHandlerPool#field#init

```
val runnables = new Array[KafkaRequestHandler](numThreads) //A thread that answers kafka requests
```

```
for(i <- 0 until numThreads) {
   runnables(i) = new KafkaRequestHandler(i, brokerId, aggregateIdleMeter, numThreads, requestChannel, apis)
   threads(i) = Utils.daemonThread("kafka-request-handler-" + i, runnables(i))
   threads(i).start()
 }
```







1 . Acceptor
1.1ServerSocketChannel上注册OP_ACCEPT事件
1.2 等待客户端的连接请求
1.3 如果有连接进来，则将其分配给当前的processor，并且把当前processor指向下一个processor

2.Processor
2.1 Processor的accpet方法（Acceptor会调用它），把一个SocketChannel放到队列中
，然后唤醒Processor的selector
2.2 如果有队列中有新的SocketChannel，则它首先将其OP_READ事情注册到该Processor的selector上面
2.3 在Processor的run方法中，它也是调用selector的select方法来监听客户端的数据请求
2.4 NetworkReceive.readFrom(channel) 读取数据
2.5 Processor的run方法（Processor是一个线程类），它会调用processNewResponses()
来处理Handler的提供给客户端的Response

3.RequestChannel
3.1 它包含了一个队列requestQueue用来存放Processor加入的Request，Handler会从里面取出Request来处理
3.2 它还为每个Processor开辟了一个respondQueue，用来存放Handler处理了Request后给客户端的Response

4.KafkaRequestHandler
4.1 Handler的职责是从requestChannel中的requestQueue取出Request
4.2 处理以后再将Response添加到requestChannel中的responseQueue中
4.3 所有的处理逻辑都交给了KafkaApis

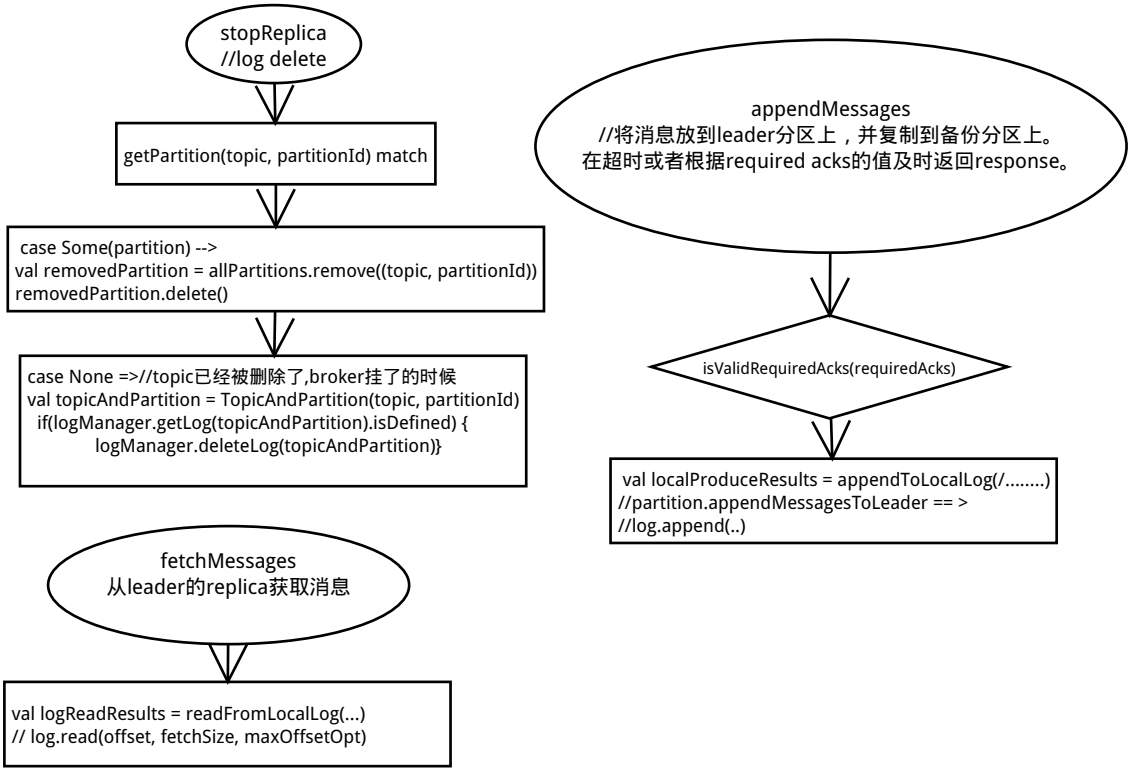
Source Part 1.3 ：KafkaController#field#init

val partitionStateMachine = new PartitionStateMachine(this)
partition的状态机:
1.NonExistentPartition--partiotion还么有创建或者已经被删除,上一个状态[如果存在过,offlinePartition]
2.NewPartition--创建完成后的状态,被指定了备份,但是还没有leader,上一个有效的状态[NonExistentPartition]
3.OnlinePartition--partition的leader被选举出来后的状态,上一个状态[NewPartition,offlinePartition]
4.offlinePartition--partition的leader挂了的状态,上一个状态[NewPartition,OnlinePartition]

val replicaStateMachine = new ReplicaStateMachine(this)
replicas的状态机:
1.NewReplica--新建的replica
2.OnlineReplica--replica开始,在这个状态可以成为leader或者follower
3.offlineReplica--replica挂了的状态
4.ReplicaDeletionStarted--replica删除开始的状态
5.ReplicaDeletionIneliqible--删除失败的状态
6.NonExistentReplica--删除成功的状态

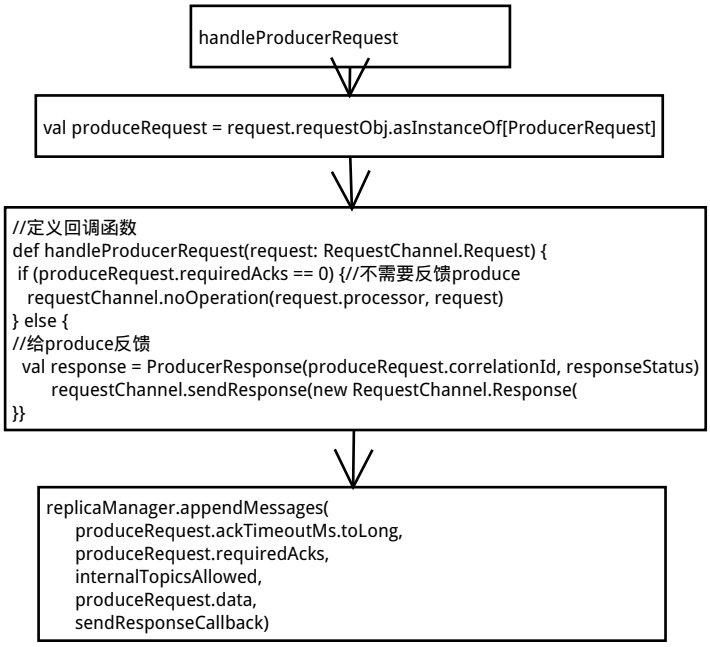
val controllerElector = new ZookeeperLeaderElector(..)
处理基于zookeeper的选举

Source Part 1.4 ：ReplicaManager#field#init

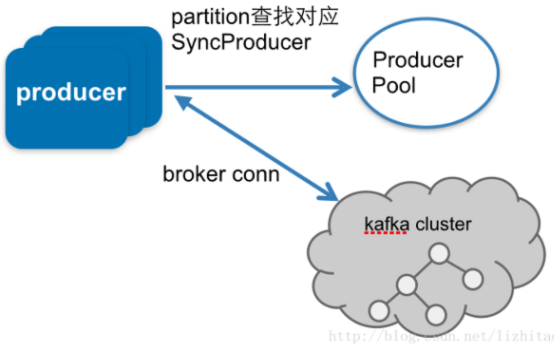


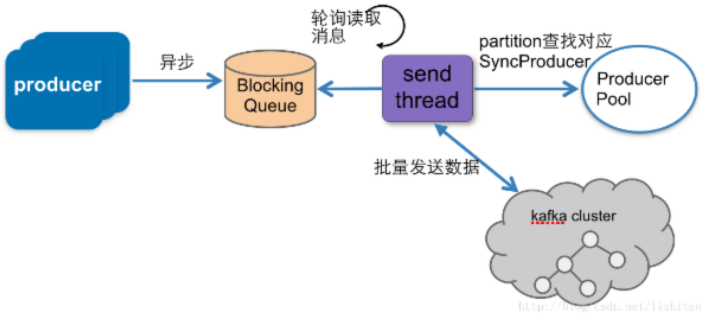
Source Part 1.5 ：KafkaApis#field#init

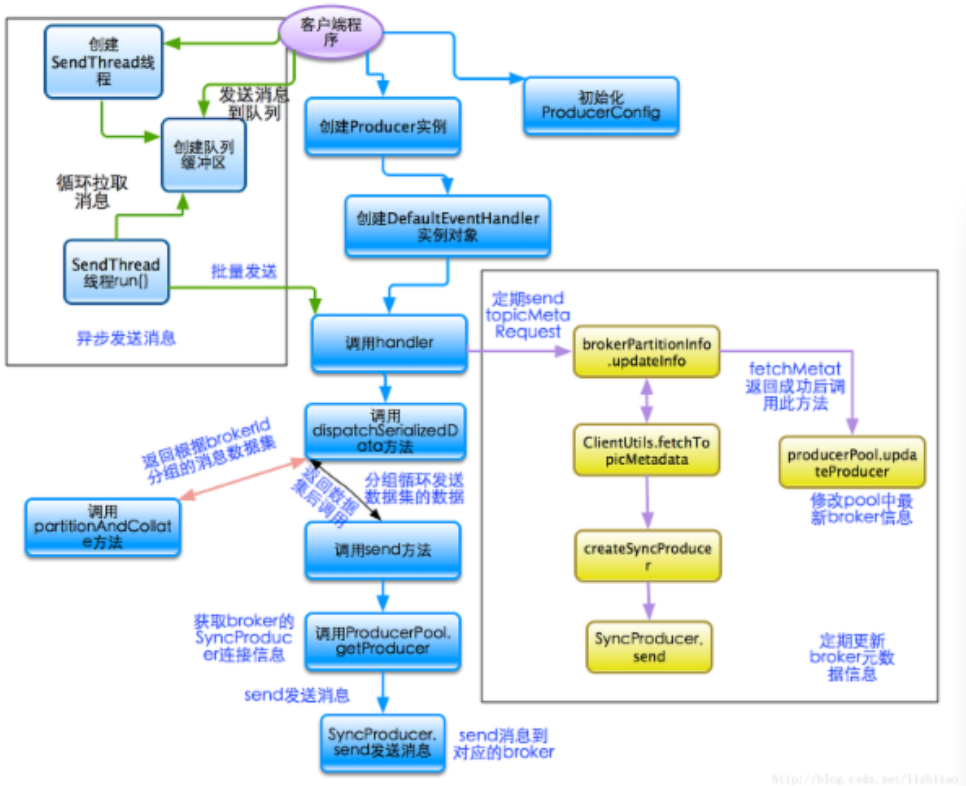
```
def handle(request: RequestChannel.Request) {
   try{
     trace("Handling request: " + request.requestObj + " from connection: " + request.connectionId)
     request.requestId match {
//produce追加message请求
       case RequestKeys.ProduceKey => handleProducerRequest(request)
//获取消息请求
       case RequestKeys.FetchKey => handleFetchRequest(request)
//
       case RequestKeys.OffsetsKey => handleOffsetRequest(request)
       case RequestKeys.MetadataKey => handleTopicMetadataRequest(request)
       case RequestKeys.LeaderAndIsrKey => handleLeaderAndIsrRequest(request)
       case RequestKeys.StopReplicaKey => handleStopReplicaRequest(request)
       case RequestKeys.UpdateMetadataKey => handleUpdateMetadataRequest(request)
       case RequestKeys.ControlledShutdownKey => handleControlledShutdownRequest(request)
       case RequestKeys.OffsetCommitKey => handleOffsetCommitRequest(request)
       case RequestKeys.OffsetFetchKey => handleOffsetFetchRequest(request)
       case RequestKeys.ConsumerMetadataKey => handleConsumerMetadataRequest(request)
       case RequestKeys.JoinGroupKey => handleJoinGroupRequest(request)
       case RequestKeys.HeartbeatKey => handleHeartbeatRequest(request)
       case requestId => throw new KafkaException("Unknown api code " + requestId)
     }
   }
```



Source Part 2 : Producer

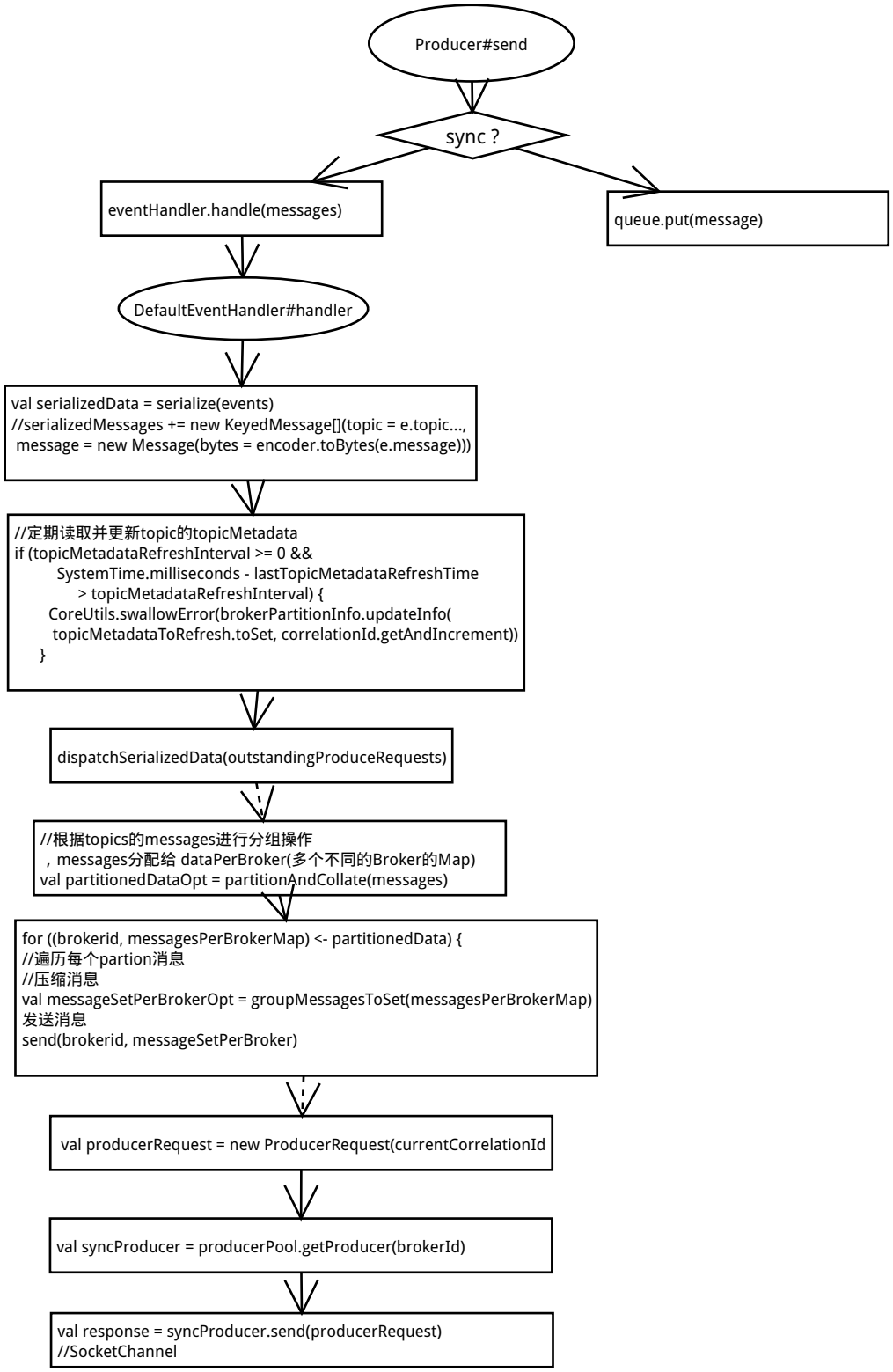






```
//异步消息队列
private val queue = new LinkedBlockingQueue[KeyedMessage[K,V]](config.queueBufferingMaxMessages)
  private var sync: Boolean = true
//异步发送线程
  private var producerSendThread: ProducerSendThread[K,V] = null
```

```
config.producerType match {
    case "sync" =>
    case "async" =>
      sync = false
      producerSendThread =
        new ProducerSendThread[K,V]("ProducerSendThread-" + config.clientId,
                                    queue,
                                    eventHandler,
                                    config.queueBufferingMaxMs,
                                    config.batchNumMessages,
                                    config.clientId)
      producerSendThread.start()//启动异步发送线程
  }
```


Producer#send
sync ?
eventHandler.handle(messages)
queue.put(message)
DefaultEventHandler#handler
val serializedData = serialize(events)
//serializedMessages += new KeyedMessage[](topic = e.topic...,
message = new Message(bytes = encoder.toBytes(e.message)))
//定期读取并更新topic的topicMetadata
if (topicMetadataRefreshInterval >= 0 &&
SystemTime.milliseconds - lastTopicMetadataRefreshTime
> topicMetadataRefreshInterval) {
CoreUtils.swallowError(brokerPartitionInfo.updateInfo(
topicMetadataToRefresh.toSet, correlationId.getAndIncrement))
}
dispatchSerializedData(outstandingProduceRequests)
//根据topics的messages进行分组操作
，messages分配给 dataPerBroker(多个不同的Broker的Map)
val partitionedDataOpt = partitionAndCollate(messages)
for ((brokerid, messagesPerBrokerMap) <- partitionedData) {
//遍历每个partion消息
//压缩消息
val messageSetPerBrokerOpt = groupMessagesToSet(messagesPerBrokerMap)
发送消息
send(brokerid, messageSetPerBroker)
val producerRequest = new ProducerRequest(currentCorrelationId
val syncProducer = producerPool.getProducer(brokerId)
val response = syncProducer.send(producerRequest)
//SocketChannel