RocketMQ设计原理分析

2020-11-26

MPOOM

核心内容

01 RocketMQ架构设计

02 RocketMQ工作流程

03 Mmap原理及OS内存分配

RocketMQ设计原理分析

目录
CONTENTS

01

架构设计

1、技术架构
2、基本概念
3、特性
4、集群部署
5、消息分片

MPOOM

1.1 架构设计

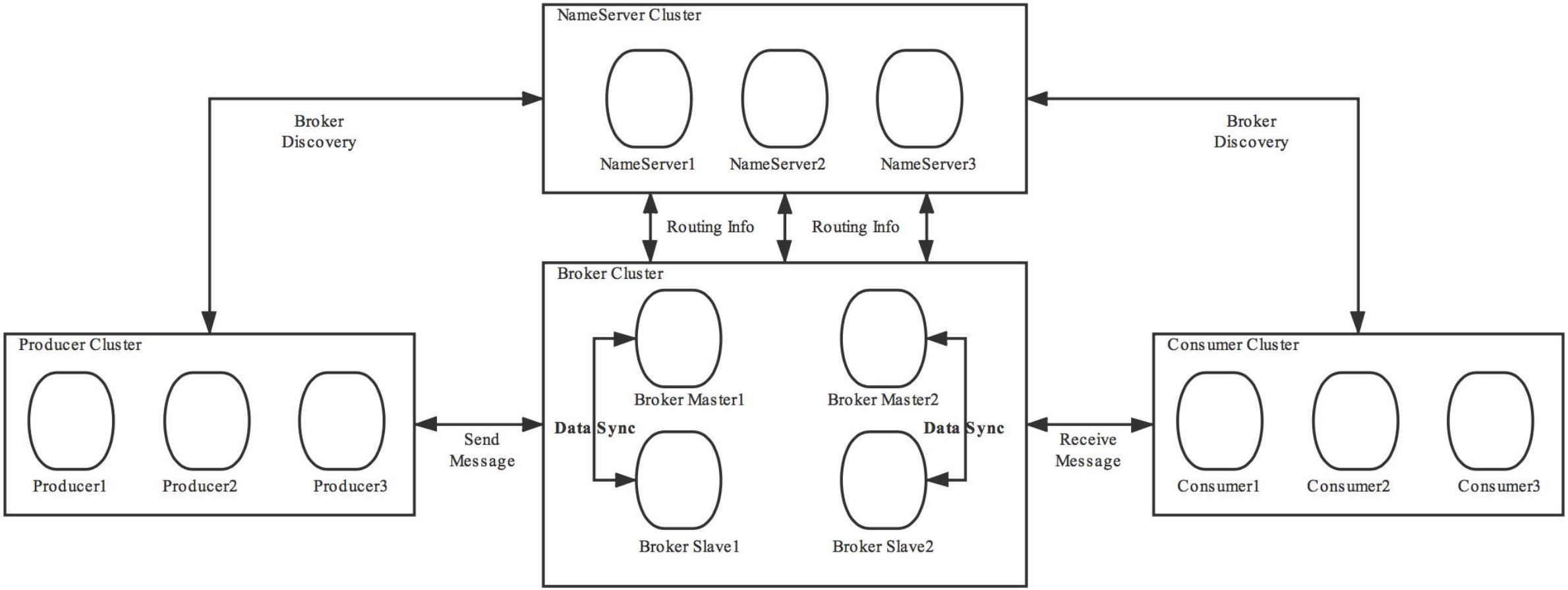


图1-1 RocketMQ架构

MPOOM

1.2.1 基本概念

Producer

消息发布的角色，支持分布式集群方式部署。Producer通过MQ的负载均衡模块选择相应的Broker集群队列进行消息投递，投递的过程支持快速失败并且低延迟。

Consumer

消息消费的角色，支持分布式集群方式部署。支持以push推，pull拉两种模式对消息进行消费。同时也支持集群方式和广播方式的消费，它提供实时消息订阅机制，可以满足大多数用户的需求。

NameServer

NameServer是一个非常简单的Topic路由注册中心，支持Broker的动态注册与发现。主要功能为Broker管理和路由信息管理。NameServer通常也是集群的方式部署，各实例间相互不进行信息通讯。

BrokerServer

Broker主要负责消息的存储、投递和查询以及服务高可用保证。

MPOOM

1.2.2 基本概念

主题（Topic）

表示一类消息的集合，每个主题包含若干条消息，每条消息只能属于一个主题，是RocketMQ进行消息订阅的基本单位。

消息（Message）

消息系统所传输信息的物理载体，生产和消费数据的最小单位，每条消息必须属于一个主题。RocketMQ中每个消息拥有唯一的Message ID，且可以携带具有业务标识的Key。系统提供了通过Message ID和Key查询消息的功能。

标签（Tag）

为消息设置的标志，用于同一主题下区分不同类型的消息。来自同一业务单元的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性，并优化RocketMQ提供的查询系统。消费者可以根据Tag实现对不同子主题的不同消费逻辑，实现更好的扩展性。

MPOOM

1.3 特性1

消息中间件	客户端 SDK	协议和规范	有序消息	定时消息	批量消息	广播消息	消息过滤	消息重新投递
Kafka	Java, Scala etc.	Pull model, support TCP	topic单分区有序	不支持	支持，在Producer缓存消息进行批量发送	不支持	Supported, you can use Kafka Streams to filter messages	不支持
RocketMQ	Java, C++, Go	Pull model, support TCP, JMS, OpenMessaging	topic单队列有序	支持固定延时等级的延时消息	支持，同步发送避免消息丢失	支持	属性过滤及SQL92	支持

MPOOM

1.3 特性2

消息存储	回溯消费	消息优先级	高可用及故障转移	消息追踪	配置	管理及操作工具
每个topic的每个partition对应一个文件。顺序写入，定时刷盘	支持偏移量回溯消费	不支持	支持，需要ZooKeeper server	不支持	采用key-value的配置方式	支持，终端命令
单个broker所有topic在CommitLog中顺序写	支持时间戳和偏移量回溯消费	不可用	支持，主从模式	支持	开箱即用，只需关注少量配置	支持，web和终端

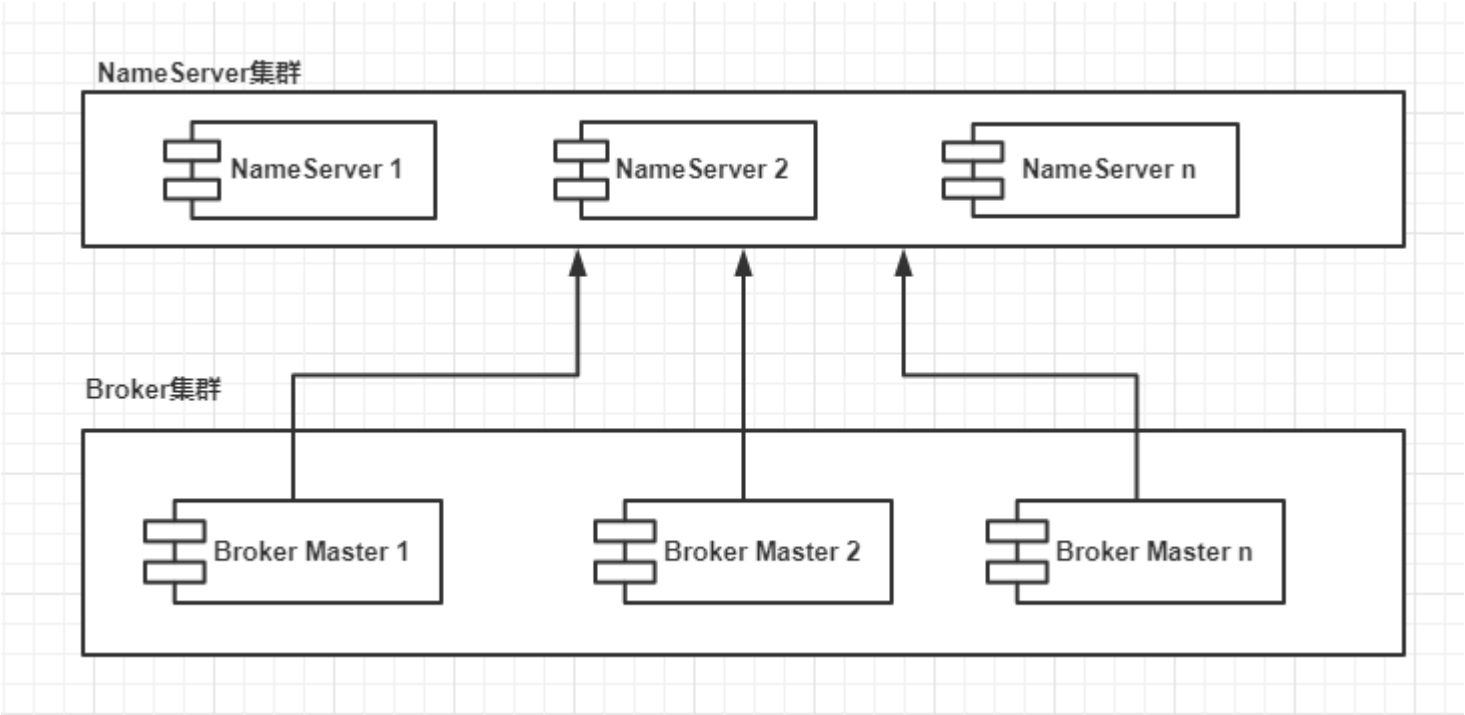
MPOOM

1.4 集群部署

Broker集群模式	说明
单Master模式	这种方式风险较大，一旦Broker重启或者宕机时，会导致整个服务不可用。不建议线上环境使用,可以用于本地测试。
多Master模式	一个集群无Slave，全是Master，例如2个Master或者3个Master
多Master多Slave模式-异步复制	每个Master配置一个Slave，有多对Master-Slave，HA采用异步复制方式，主备有短暂消息延迟（毫秒级）
多Master多Slave模式-同步双写	每个Master配置一个Slave，有多对Master-Slave，HA采用同步双写方式，即只有主备都写成功，才向应用返回成功

MPOOM

1.4.1 多Master模式

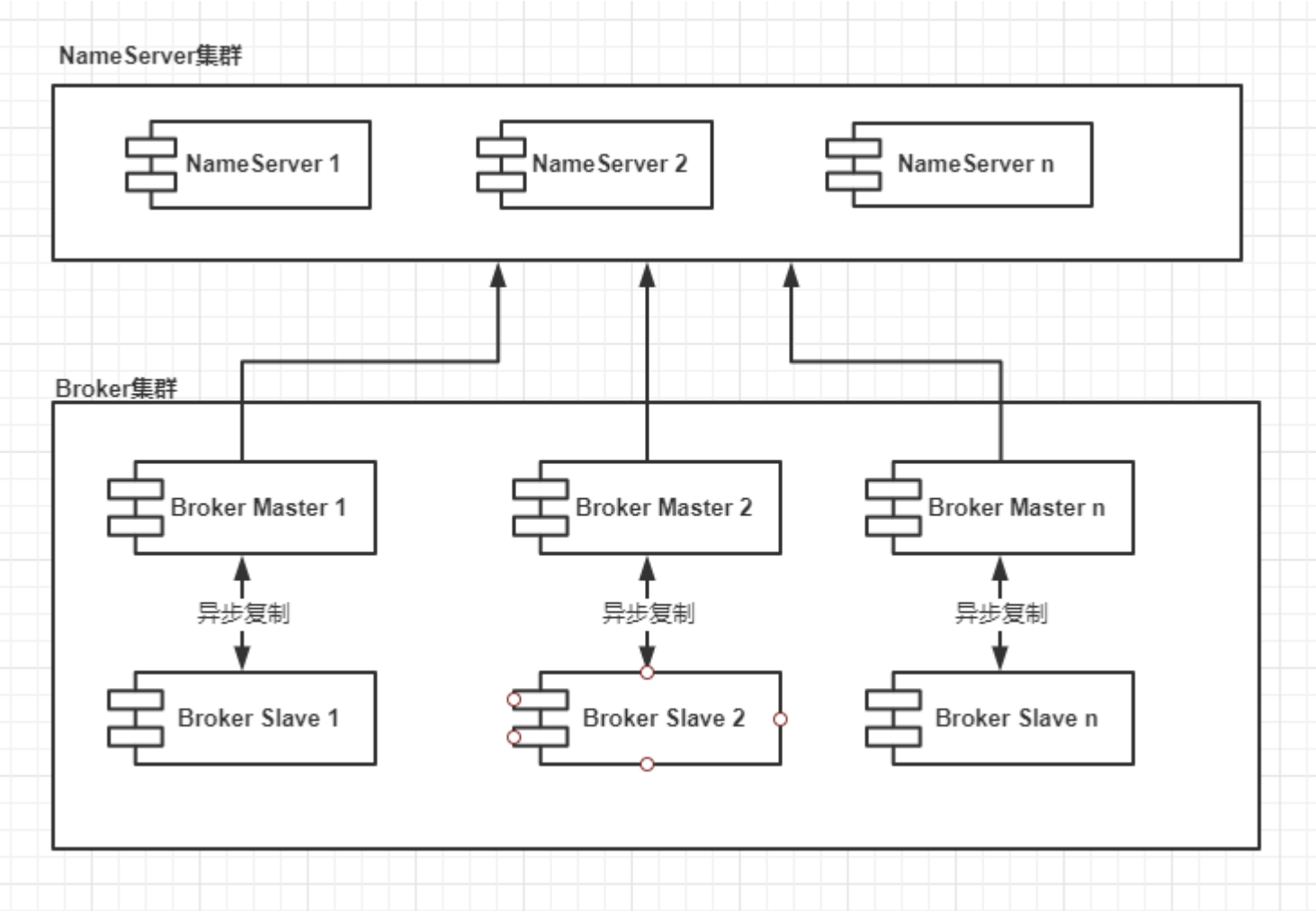


•优点：配置简单，单个Master宕机或重启维护对应用无影响，异步刷盘丢失少量消息，同步刷盘一条不丢，性能最高；

•缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响。

MPOOM

1.4.2 多Master多Slave模式-异步复制

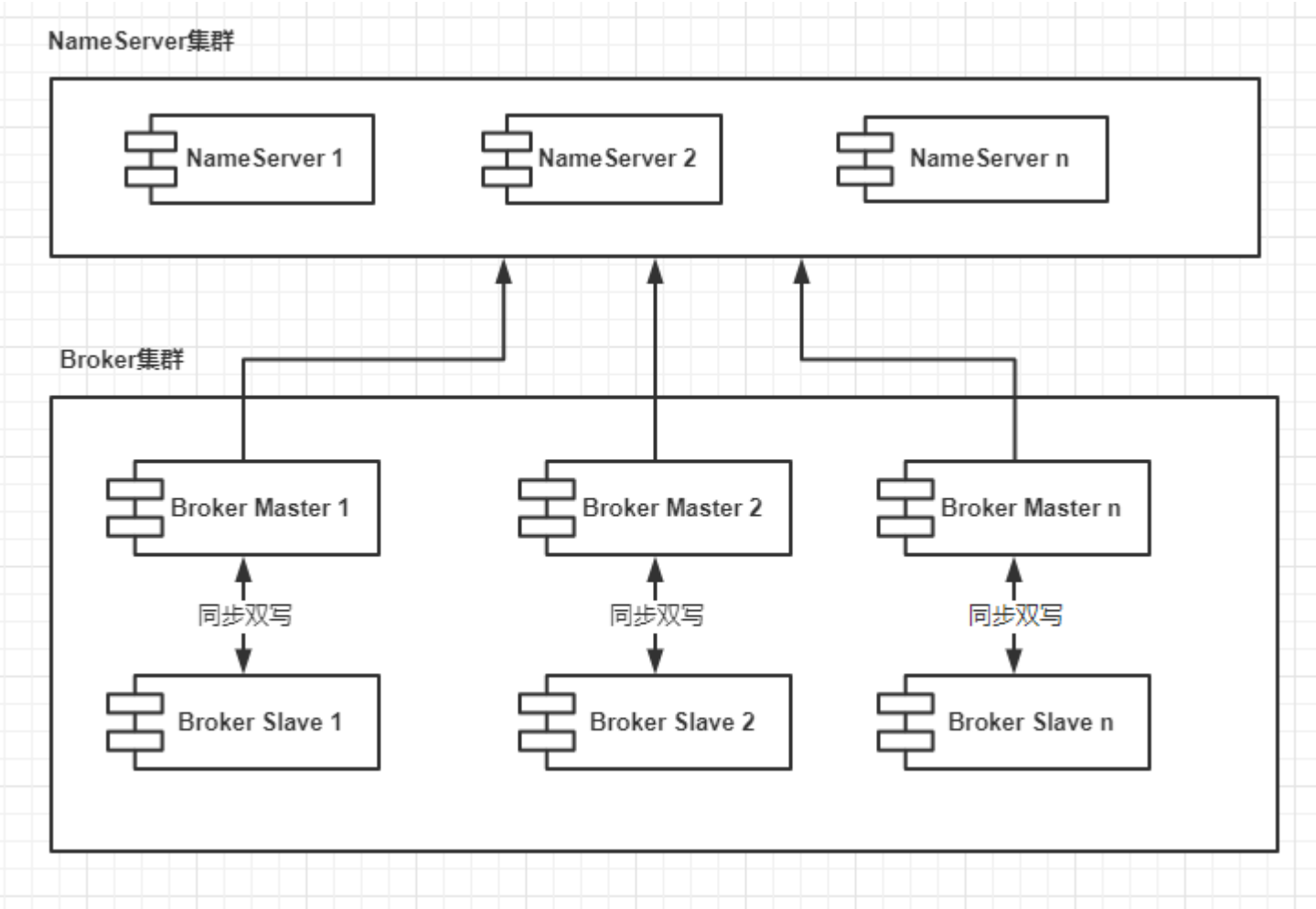


•优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，同时**Master宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干预**，性能同多Master模式几乎一样；

•缺点：Master宕机，磁盘损坏情况下会丢失少量消息。

MPOOM

1.4.3 多Master多Slave模式-同步双写



•优点：数据与服务都无单点故障，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高；

•缺点：性能比异步复制模式略低（大约低10%左右），发送单个消息的RT会略高，且**目前版本在主节点宕机后，备机不能自动切换为主机**

MPOOM

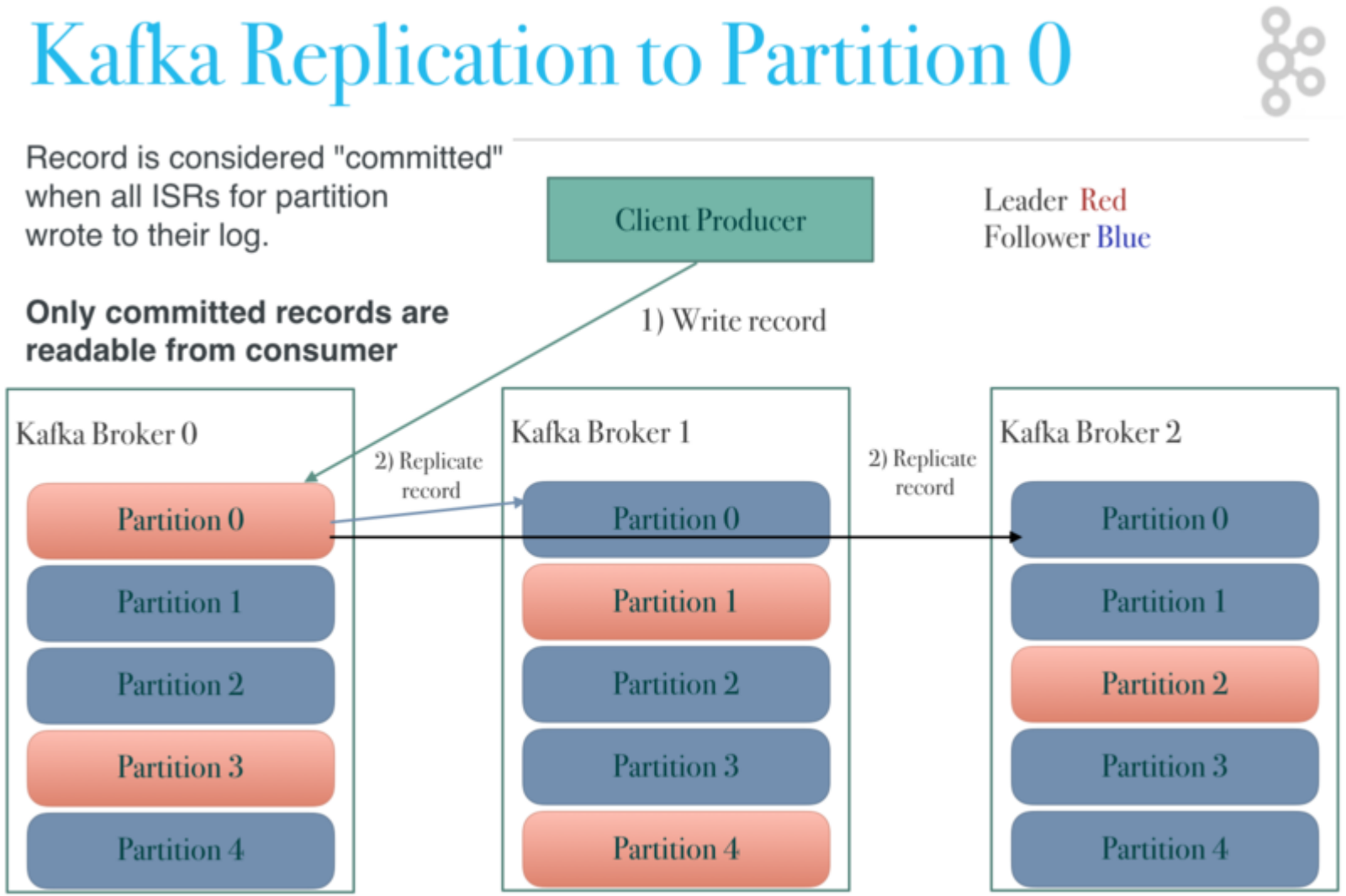
1.4.4 开启从Slave读数据功能

在某些情况下，Consumer需要将消费位点重置到1-2天前，这时在内存有限的Master Broker上，CommitLog会承载比较重的IO压力，影响到该Broker的其它消息的读与写。

可以开启slaveReadEnable=true，当Master Broker发现Consumer的消费位点与CommitLog的最新值的差值的容量超过该机器内存的百分比（accessMessageInMemoryMaxRatio=40%），会推荐Consumer从Slave Broker中去读取数据，降低Master Broker的IO。

MPOOM

1.5.1 kafka消息分区



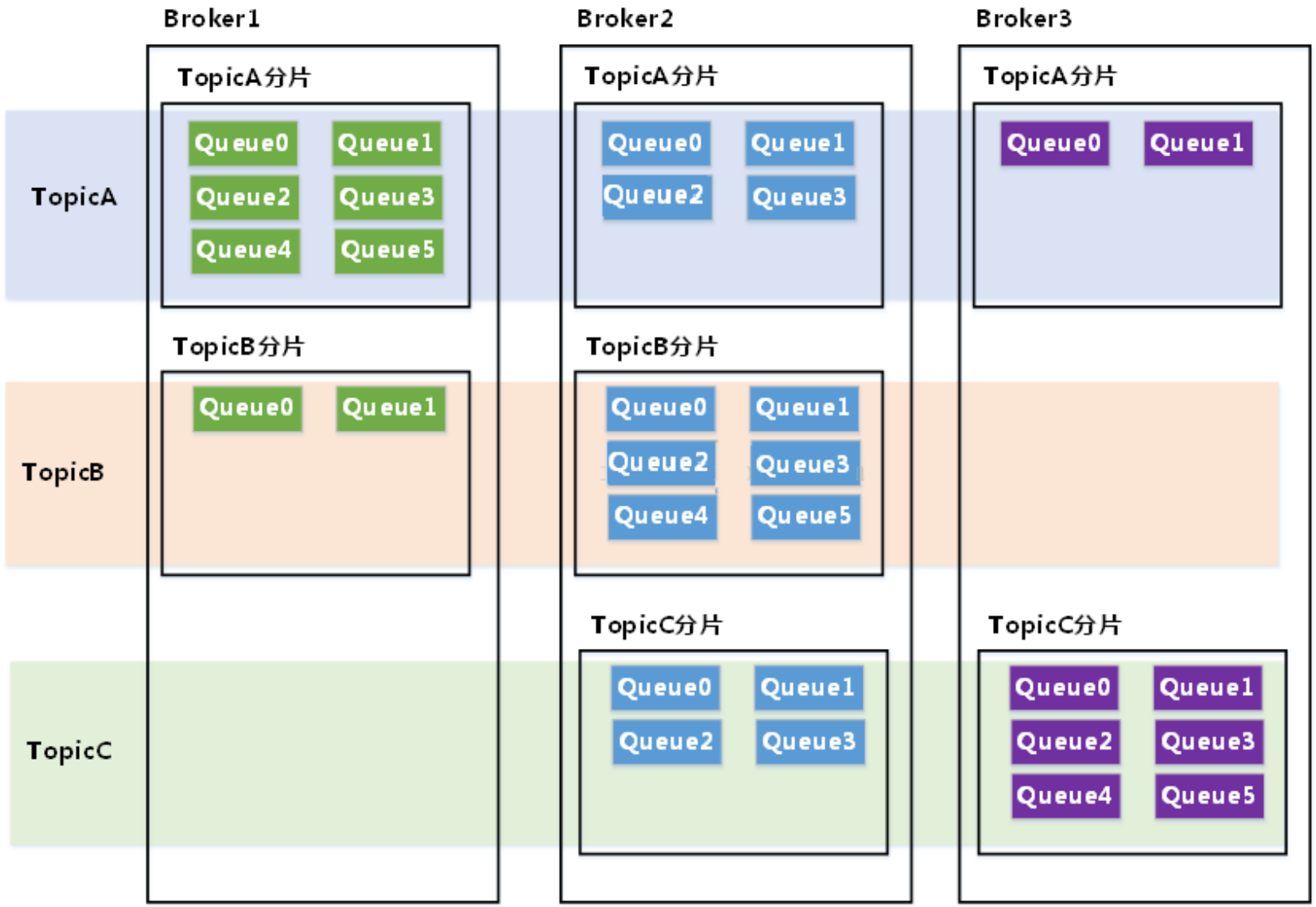
Kafka Broker集群中有3个Broker节点

在该集群下创建Topic，该Topic有5个分区，每个分区有3个副本（1个leader，2个follower）。

每个leader和follower都是一个broker。Kafka会把所有partition的leader平均分配到broker上，**所有的读写都只由leader来完成**，follower只从leader同步消息，并不对外服务。

MPOOM

1.5.2 RocketMQ消息分片



Broker集群中有3个Broker节点

TopicA在Broker1创建了6个读写队列；在Broker2创建了4个读写队列；在Broker3创建了2个读写队列

RocketMQ在创建Topic时需要指定Broker及队列数量。

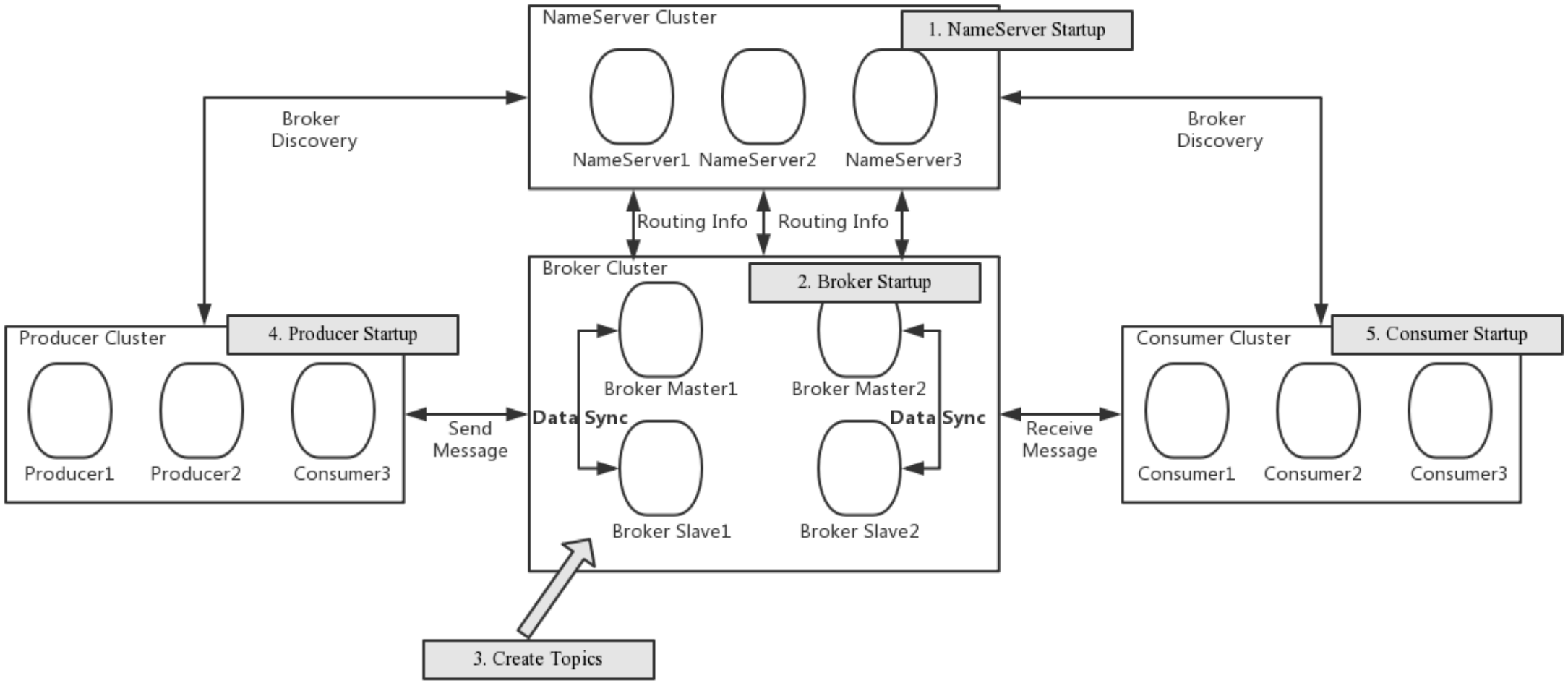
02

工作流程

1、创建Topic
2、消息发送
3、消息存储
4、消息消费

MPOOM

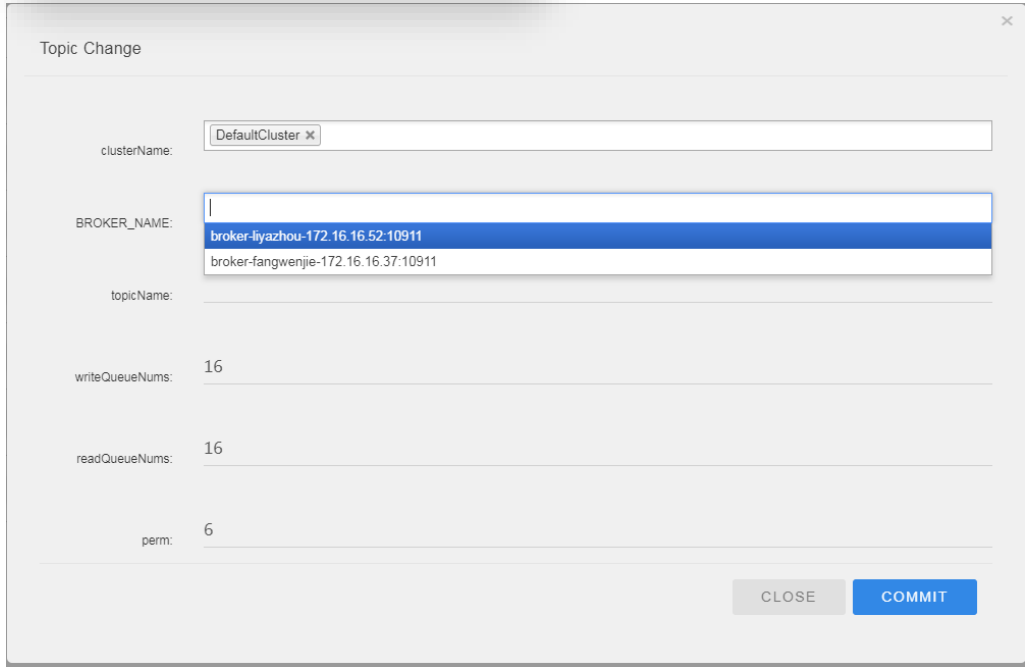
2. 工作流程


NameServer Cluster
1. NameServer Startup
NameServer1
NameServer2
NameServer3
Broker Discovery
Broker Discovery
Routing Info
Routing Info
Broker Cluster
2. Broker Startup
Broker Master1
Broker Master2
Broker Slave1
Broker Slave2
Data Sync
Data Sync
Producer Cluster
4. Producer Startup
Producer1
Producer2
Consumer3
Send Message
Receive Message
Consumer Cluster
5. Consumer Startup
Consumer1
Consumer2
Consumer3
3. Create Topics

MPOOM

2.1 创建Topic

创建Topic的三种方式



（1）通过RocketMQ-Console创建

```
root@liyazhou-All-Series:/home/liuxu/logs/rocketmqlogs# tail -n200 broker.log
2020-11-09 16:44:55 INFO main - brokerIP1=172.16.16.52
2020-11-09 16:44:55 INFO main - brokerIP2=172.17.0.1
2020-11-09 16:44:55 INFO main - brokerName=broker-liyazhou-172.16.16.52:10911
2020-11-09 16:44:55 INFO main - brokerClusterName=DefaultCluster
2020-11-09 16:44:55 INFO main - brokerId=0
2020-11-09 16:44:55 INFO main - brokerPermission=6
2020-11-09 16:44:55 INFO main - defaultTopicQueueNums=8
2020-11-09 16:44:55 INFO main - autoCreateTopicEnable=true
2020-11-09 16:44:55 INFO main - clusterTopicEnable=true
2020-11-09 16:44:55 INFO main - brokerTopicEnable=true
2020-11-09 16:44:55 INFO main - autoCreateSubscriptionGroup=true
2020-11-09 16:44:55 INFO main - messageStorePlugIn=
2020-11-09 16:44:55 INFO main - msgTraceTopicName=RMQ_SYS_TRACE_TOPIC
2020-11-09 16:44:55 INFO main - traceTopicEnable=false
2020-11-09 16:44:55 INFO main - sendMessageThreadPoolNums=1
2020-11-09 16:44:55 INFO main - pullMessageThreadPoolNums=24
2020-11-09 16:44:55 INFO main - processReplyMessageThreadPoolNums=24
2020-11-09 16:44:55 INFO main - queryMessageThreadPoolNums=12
2020-11-09 16:44:55 INFO main - adminBrokerThreadPoolNums=16
2020-11-09 16:44:55 INFO main - clientManageThreadPoolNums=32
2020-11-09 16:44:55 INFO main - consumerManageThreadPoolNums=32
2020-11-09 16:44:55 INFO main - heartbeatThreadPoolNums=4
2020-11-09 16:44:55 INFO main - endTransactionThreadPoolNums=16
2020-11-09 16:44:55 INFO main - flushConsumerOffsetInterval=5000
2020-11-09 16:44:55 INFO main - flushConsumerOffsetHistoryInterval=60000
```

（2）Broker开启自动创建（默认开启，生产环境建议关闭）

```
liuxu@fangwenjie-All-Series:/media/fangwenjie/data/boy/component/rocketmq-4.7.0/bin$ sh mqadmin updateTopic -c DefaultCluster -n 172.16.16.37:9876 -t cluster-broker-topic-hyk -r 8 -w 8
RocketMQLog:WARN No appenders could be found for logger (io.netty.util.internal.PlatformDependent0).
RocketMQLog:WARN Please initialize the logger system properly.
create topic to 172.16.16.37:10911 success.
create topic to 172.16.16.52:10911 success.
TopicConfig [topicName=cluster-broker-topic-hyk, readQueueNums=8, writeQueueNums=8, perm=RW-, topicFilterType=SINGLE_TAG, topicSysFlag=0, order=false]liuxu@fangwenjie-All-Series:/media/fangwenjie/
ponent/rocketmq-4.7.0/bin$
```

（3）通过命令行工具mqadmin创建

MPOOM

2.2 Producer发送消息

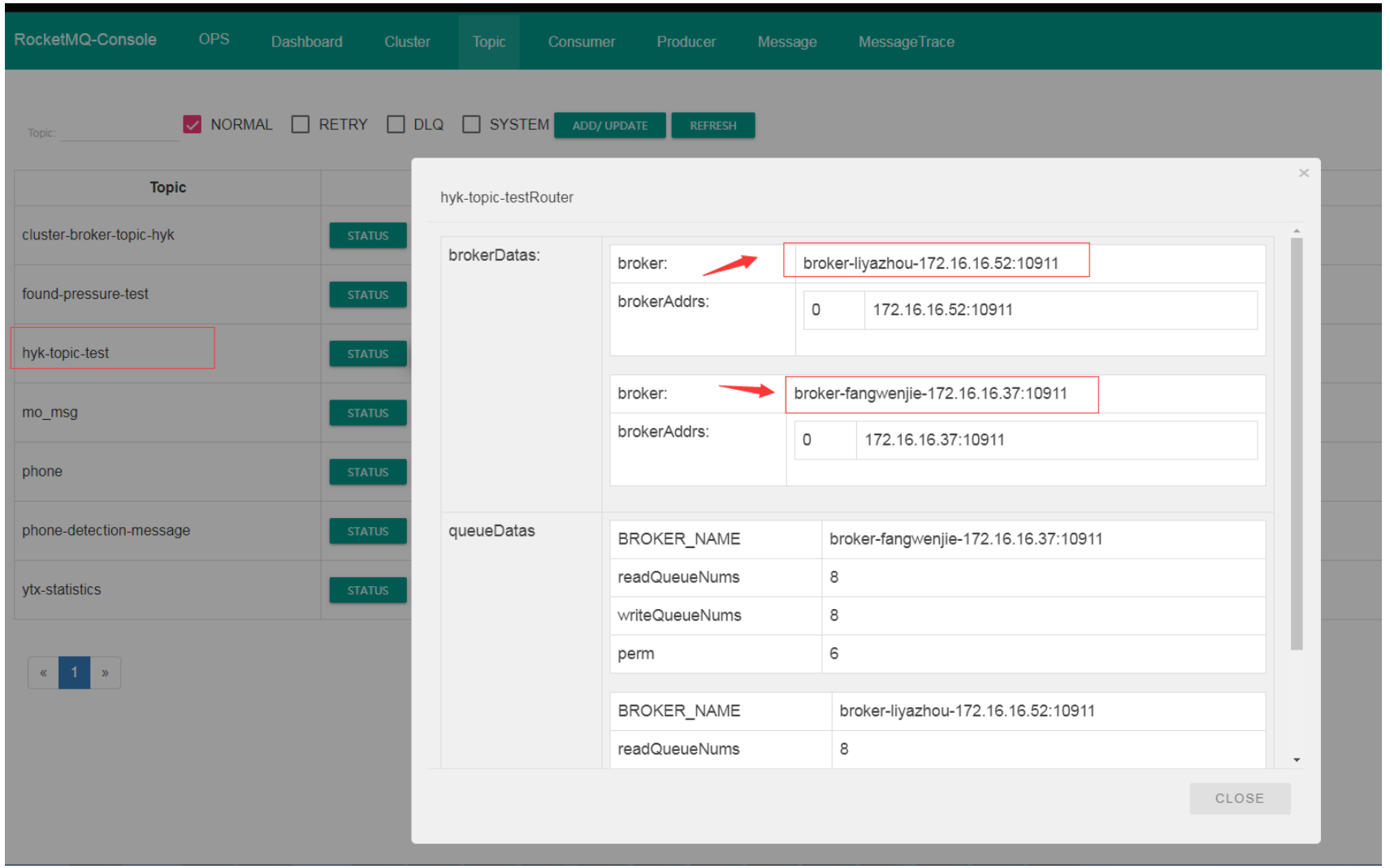
1、消费队列选择

2、失败重试

3、故障Broker规避

MPOOM

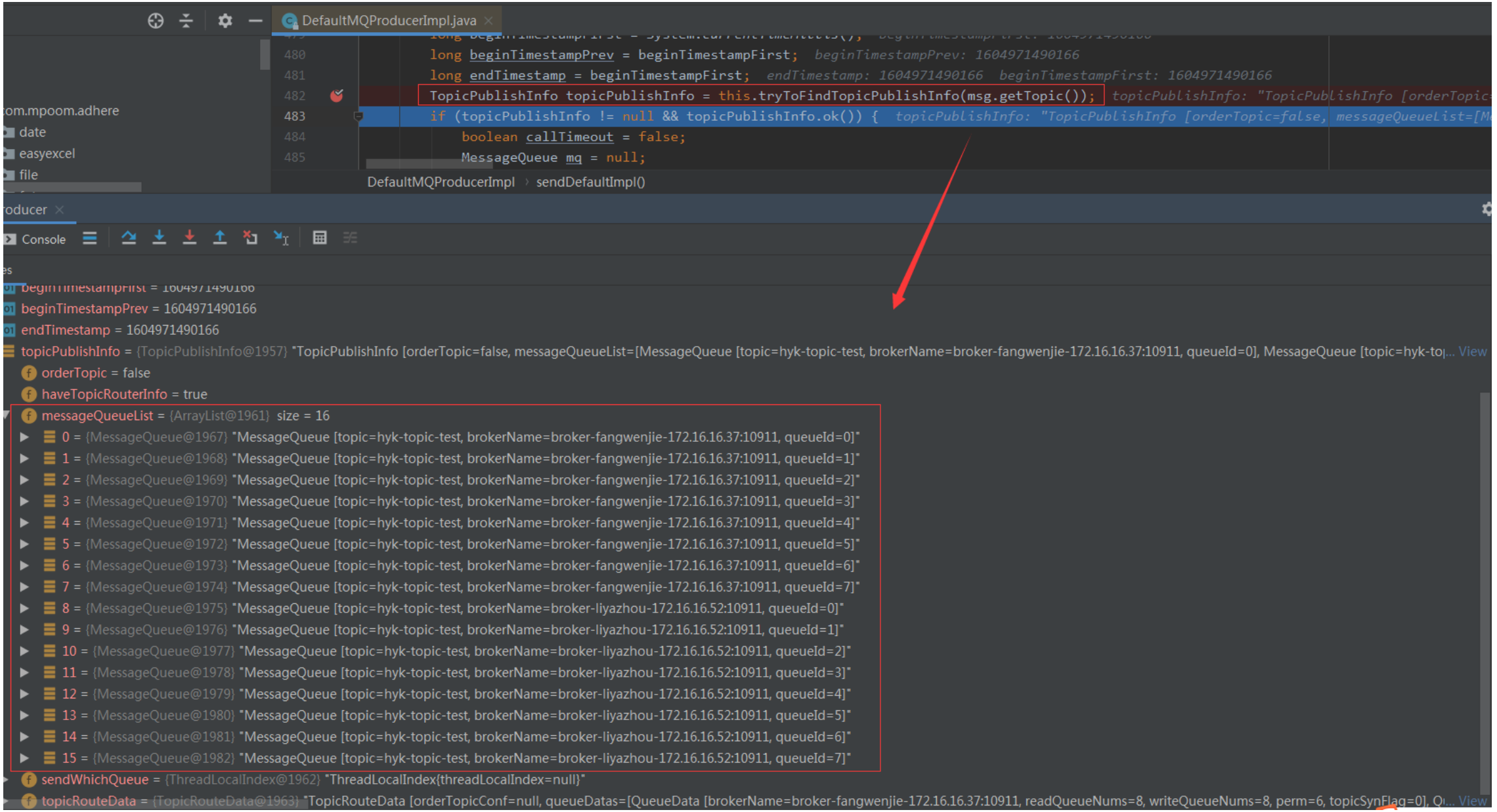
通过RocketMQ-Console查看Topic路由信息



Topic: hyk-topic-test路由信息

MPOOM

Producer端获取到的路由信息



MPOOM

Producer端失败重试

DefaultMQProducerImpl.java

```
469 @       private SendResult sendDefaultImpl(
470             Message msg,
471             final CommunicationMode communicationMode,
472             final SendCallback sendCallback,
473             final long timeout
474         ) throws MQClientException, RemotingException, MQBrokerException, InterruptedException {
475             this.makeSureStateOK();
476             Validators.checkMessage(msg, this.defaultMQProducer);
477
478             final long invokeID = random.nextLong();
479             long beginTimestampFirst = System.currentTimeMillis();
480             long beginTimestampPrev = beginTimestampFirst;
481             long endTimestamp = beginTimestampFirst;
482             TopicPublishInfo topicPublishInfo = this.tryToFindTopicPublishInfo(msg.getTopic());
483             if (topicPublishInfo != null && topicPublishInfo.ok()) {
484                 boolean callTimeout = false;
485                 MessageQueue mq = null;
486                 Exception exception = null;
487                 SendResult sendResult = null;
488                 int timesTotal = communicationMode == CommunicationMode.SYNC ? 1 + this.defaultMQProducer.getRetryTimesWhenSendFailed() : 1;
489                 int times = 0;
490                 String[] brokersSent = new String[timesTotal];
491                 for (; times < timesTotal; times++) {
492                     String lastBrokerName = null == mq ? null : mq.getBrokerName();
493                     MessageQueue mqSelected = this.selectOneMessageQueue(topicPublishInfo, lastBrokerName);
494                     if (mqSelected != null) {
495                         mq = mqSelected;
496                         brokersSent[times] = mq.getBrokerName();
497                         try {
498                             beginTimestampPrev = System.currentTimeMillis();
499                             long costTime = beginTimestampPrev - beginTimestampFirst;
500                             if (timeout < costTime) {
501                                 callTimeout = true;
502                                 break;
503                             }
```

MPOOM

Producer未开启故障延迟

sendLatencyFaultEnable = false

MPOOM

Producer默认未开启故障延迟

MQFaultStrategy.java

```
1   /.../
17
18  package org.apache.rocketmq.client.latency;
19
20  import ...
24
25  public class MQFaultStrategy {
26      private final static InternalLogger log = ClientLogger.getLog();
27      private final LatencyFaultTolerance<String> latencyFaultTolerance = new LatencyFaultToleranceImpl();
28
29      private boolean sendLatencyFaultEnable = false;
30
31      private long[] latencyMax = {50L, 100L, 550L, 1000L, 2000L, 3000L, 15000L};
32      private long[] notAvailableDuration = {0L, 0L, 30000L, 60000L, 120000L, 180000L, 600000L};
33
34      public long[] getNotAvailableDuration() { return notAvailableDuration; }
37
38      public void setNotAvailableDuration(final long[] notAvailableDuration) {
39          this.notAvailableDuration = notAvailableDuration;
40      }
41
```

latencyMax：消息发送延迟级别
notAvavilabelDuartion：Broker不可用时长

MPOOM

sendLatencyFaultEnable = false

MQFaultStrategy.java

```
55              this.sendLatencyFaultEnable = sendLatencyFaultEnable;
56          }
57
58      @   public MessageQueue selectOneMessageQueue(final TopicPublishInfo tpInfo, final String lastBrokerName) {
59              if (this.sendLatencyFaultEnable) {
60                  try {...} catch (Exception e) {
86                      log.error("Error occurred when selecting message queue", e);
87                  }
88
89                  return tpInfo.selectOneMessageQueue();
90              }
91
92              return tpInfo.selectOneMessageQueue(lastBrokerName);
93          }
94
```

MPOOM

循环遍历获取下一个队列，同时避开上一次发送失败的Broker

TopicPublishInfo.java

```
65      public void setHaveTopicRouterInfo(boolean haveTopicRouterInfo) { this.haveTopicRouterInfo = haveTopicRouterInfo; }
68
69  @   public MessageQueue selectOneMessageQueue(final String lastBrokerName) {
70          if (lastBrokerName == null) {
71              return selectOneMessageQueue();
72          } else {
73              int index = this.sendWhichQueue.getAndIncrement();
74              for (int i = 0; i < this.messageQueueList.size(); i++) {
75                  int pos = Math.abs(index++) % this.messageQueueList.size();
76                  if (pos < 0)
77                      pos = 0;
78                  MessageQueue mq = this.messageQueueList.get(pos);
79                  if (!mq.getBrokerName().equals(lastBrokerName)) {
80                      return mq;
81                  }
82              }
83              return selectOneMessageQueue();
84          }
85      }
86
87      public MessageQueue selectOneMessageQueue() {
88          int index = this.sendWhichQueue.getAndIncrement();
89          int pos = Math.abs(index) % this.messageQueueList.size();
90          if (pos < 0)
91              pos = 0;
92          return this.messageQueueList.get(pos);
93      }
94
```

MPOOM

Producer开启故障延迟

sendLatencyFaultEnable = true

MPOOM

循环遍历获取下一个可用队列

循环便利选取下一个队列。
如果队列所在Broker可用，
则返回当前队列

MQFaultStrategy.java

```
58 @    public MessageQueue selectOneMessageQueue(final TopicPublishInfo tpInfo, final String lastBrokerName) {
59          if (this.sendLatencyFaultEnable) {
60              try {
61                  int index = tpInfo.getSendWhichQueue().getAndIncrement();
62                  for (int i = 0; i < tpInfo.getMessageQueueList().size(); i++) {
63                      int pos = Math.abs(index++) % tpInfo.getMessageQueueList().size();
64                      if (pos < 0)
65                          pos = 0;
66                      MessageQueue mq = tpInfo.getMessageQueueList().get(pos);
67                      if (latencyFaultTolerance.isAvailable(mq.getBrokerName())) {
68                          if (null == lastBrokerName || mq.getBrokerName().equals(lastBrokerName))
69                              return mq;
70                      }
71                  }
72
73                  final String notBestBroker = latencyFaultTolerance.pickOneAtLeast();
74                  int writeQueueNums = tpInfo.getQueueIdByBroker(notBestBroker);
75                  if (writeQueueNums > 0) {
76                      final MessageQueue mq = tpInfo.selectOneMessageQueue();
77                      if (notBestBroker != null) {
78                          mq.setBrokerName(notBestBroker);
79                          mq.setQueueId(tpInfo.getSendWhichQueue().getAndIncrement() % writeQueueNums);
80                      }
81                      return mq;
82                  } else {
83                      latencyFaultTolerance.remove(notBestBroker);
84                  }
85              } catch (Exception e) {
86                  log.error("Error occurred when selecting message queue", e);
87              }
88
89              return tpInfo.selectOneMessageQueue();
90          }
91
92          return tpInfo.selectOneMessageQueue(lastBrokerName);
```

MPOOM

如何判断一个Broker是否可用?

LatencyFaultToleranceImpl.java

```
98
99      class FaultItem implements Comparable<FaultItem> {
100         private final String name;
101         private volatile long currentLatency;
102         private volatile long startTimestamp;
103
104 @       public FaultItem(final String name) { this.name = name; }
107
108         @Override
109 @       public int compareTo(final FaultItem other) {...}
132
133         public boolean isAvailable() {
134             return (System.currentTimeMillis() - startTimestamp) >= 0;
135         }
136
137         @Override
138         public int hashCode() {
139             int result = getName() != null ? getName().hashCode() : 0;
140             result = 31 * result + (int) (getCurrentLatency() ^ (getCurrentLatency() >>> 32));
141             result = 31 * result + (int) (getStartTimestamp() ^ (getStartTimestamp() >>> 32));
142             return result;
143         }
```

MPOOM

更新Broker可用性预测

LatencyFaultToleranceImpl.java

```
1   /.../
17
18  package org.apache.rocketmq.client.latency;
19
20  import ...
26
27  public class LatencyFaultToleranceImpl implements LatencyFaultTolerance<String> {
28      private final ConcurrentHashMap<String, FaultItem> faultItemTable = new ConcurrentHashMap<String, FaultItem>( initialCapacity: 16);
29
30      private final ThreadLocalIndex whichItemWorst = new ThreadLocalIndex();
31
32      @Override
33      public void updateFaultItem(final String name, final long currentLatency, final long notAvailableDuration) {
34          FaultItem old = this.faultItemTable.get(name);
35          if (null == old) {
36              final FaultItem faultItem = new FaultItem(name);
37              faultItem.setCurrentLatency(currentLatency);
38              faultItem.setStartTimestamp(System.currentTimeMillis() + notAvailableDuration);
39
40              old = this.faultItemTable.putIfAbsent(name, faultItem);
41              if (old != null) {
42                  old.setCurrentLatency(currentLatency);
43                  old.setStartTimestamp(System.currentTimeMillis() + notAvailableDuration);
44              }
45          } else {
46              old.setCurrentLatency(currentLatency);
47              old.setStartTimestamp(System.currentTimeMillis() + notAvailableDuration);
48          }
49      }
50
```

BrokerName

消息发送延迟时间

Broker不可用时间

MPOOM

消息发送成功或失败更新Broker的可用性

DefaultMQProducerImpl.java

```
496                 brokersSent[times] = mq.getBrokerName();
497                 try {
498                     beginTimestampPrev = System.currentTimeMillis();
499                     long costTime = beginTimestampPrev - beginTimestampFirst;
500                     if (timeout < costTime) {
501                         callTimeout = true;
502                         break;
503                     }
504
505                     sendResult = this.sendKernelImpl(msg, mq, communicationMode, sendCallback, topicPublishInfo,  timeout: timeout - costTime);
506                     endTimestamp = System.currentTimeMillis();
507                     this.updateFaultItem(mq.getBrokerName(),  currentLatency: endTimestamp - beginTimestampPrev,  isolation: false);
508                     switch (communicationMode) {...}
524                 } catch (RemotingException e) {
525                     endTimestamp = System.currentTimeMillis();
526                     this.updateFaultItem(mq.getBrokerName(),  currentLatency: endTimestamp - beginTimestampPrev,  isolation: true);
527                     log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
528                     log.warn(msg.toString());
529                     exception = e;
530                     continue;
531                 } catch (MQClientException e) {
532                     endTimestamp = System.currentTimeMillis();
533                     this.updateFaultItem(mq.getBrokerName(),  currentLatency: endTimestamp - beginTimestampPrev,  isolation: true);
534                     log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
535                     log.warn(msg.toString());
536                     exception = e;
537                     continue;
538                 } catch (MQBrokerException e) {
539                     endTimestamp = System.currentTimeMillis();
540                     this.updateFaultItem(mq.getBrokerName(),  currentLatency: endTimestamp - beginTimestampPrev,  isolation: true);
541                     log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
542                     log.warn(msg.toString());
543                     exception = e;
544                     switch (e.getResponseCode()) {...}
559                 } catch (InterruptedException e) {
560                     endTimestamp = System.currentTimeMillis();
561                     this.updateFaultItem(mq.getBrokerName(),  currentLatency: endTimestamp - beginTimestampPrev,  isolation: false);
```

MPOOM

Broker可用时间

```
public class MQFaultStrategy {
    private final static InternalLogger log = ClientLogger.getLog();
    private final LatencyFaultTolerance<String> latencyFaultTolerance = new LatencyFaultToleranceImpl();

    private boolean sendLatencyFaultEnable = false;

    private long[] latencyMax = {50L, 100L, 550L, 1000L, 2000L, 3000L, 15000L};
    private long[] notAvailableDuration = {0L, 0L, 30000L, 60000L, 120000L, 180000L, 600000L};
```

```
    public void updateFaultItem(final String brokerName, final long currentLatency, boolean isolation) {
        if (this.sendLatencyFaultEnable) {
            long duration = computeNotAvailableDuration(isolation ? 30000 : currentLatency);
            this.latencyFaultTolerance.updateFaultItem(brokerName, currentLatency, duration);
        }
    }

    private long computeNotAvailableDuration(final long currentLatency) {
        for (int i = latencyMax.length - 1; i >= 0; i--) {
            if (currentLatency >= latencyMax[i])
                return this.notAvailableDuration[i];
        }

        return 0;
    }
```

MPOOM

Broker不可用时队列选择策略

LatencyFaultToleranceImpl.java

```
61  public void remove(final String name) { this.faultItemTable.remove(name); }
64
65      @Override
66      public String pickOneAtLeast() {
67          final Enumeration<FaultItem> elements = this.faultItemTable.elements();
68          List<FaultItem> tmpList = new LinkedList<~>();
69          while (elements.hasMoreElements()) {
70              final FaultItem faultItem = elements.nextElement();
71              tmpList.add(faultItem);
72          }
73
74          if (!tmpList.isEmpty()) {
75              Collections.shuffle(tmpList);
76
77              Collections.sort(tmpList);
78
79              final int half = tmpList.size() / 2;
80              if (half <= 0) {
81                  return tmpList.get(0).getName();
82              } else {
83                  final int i = this.whichItemWorst.getAndIncrement() % half;
84                  return tmpList.get(i).getName();
85              }
86          }
87
88          return null;
89      }
90
```

对Broker打乱排序，在排序后的前半部分Broker集合中随机选取一个Broker返回。并使用该Broker中的一个队列进行发送。

MPOOM

总结

- 在不开启容错的情况下，轮询队列进行发送，如果失败了，重试的时候过滤失败的Broker
- 如果开启了容错策略，会通过RocketMQ的预测机制来预测一个Broker是否可用
- 如果上次失败的Broker可用那么还是会选择该Broker的队列
- 如果上述情况失败，则对Broker进行打乱排序，从前半部分中随机选取一个Broker
- 在发送消息的时候会记录一下调用的时间与是否报错，根据该时间去预测broker的可用时间

MPOOM

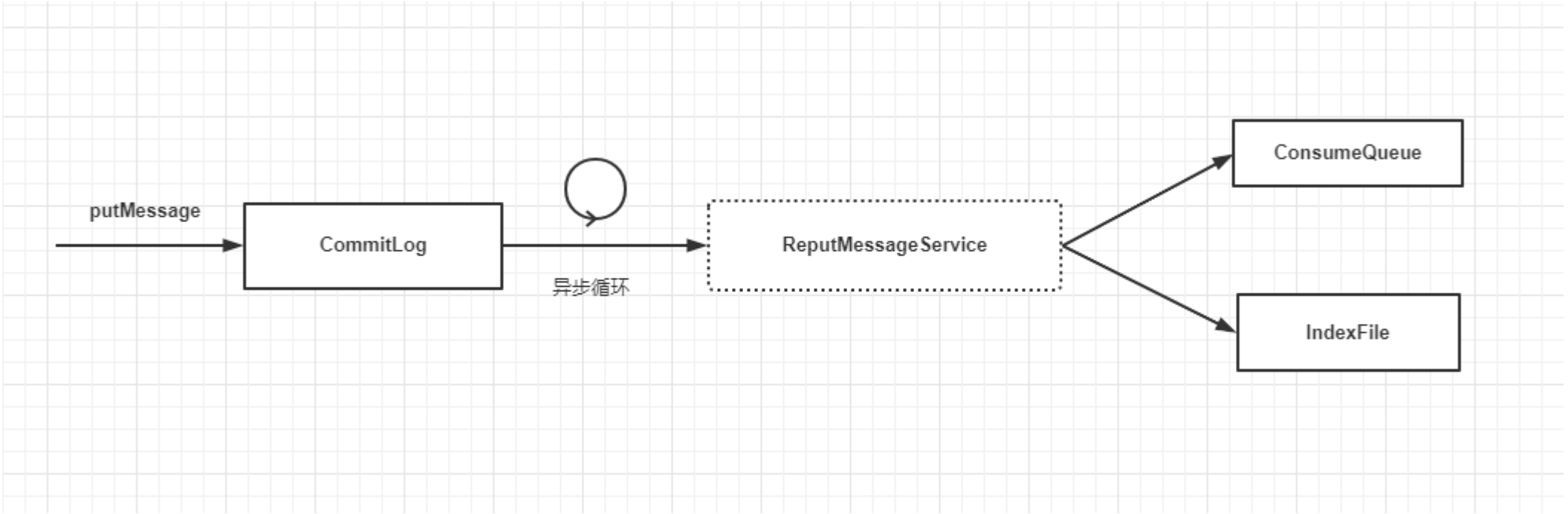
消息重复发送的问题?

DefaultMQProducerImpl.java

```
500                 if (timeout < costTime) {
501                     callTimeout = true;
502                     break;
503                 }
504
505                 sendResult = this.sendKernelImpl(msg, mq, communicationMode, sendCallback, topicPublishInfo, timeout: timeout - costTime);
506                 endTimestamp = System.currentTimeMillis();
507                 this.updateFaultItem(mq.getBrokerName(), currentLatency: endTimestamp - beginTimestampPrev, isolation: false);
508                 switch (communicationMode) {...}
524             } catch (RemotingException e) {
525                 endTimestamp = System.currentTimeMillis();
526                 this.updateFaultItem(mq.getBrokerName(), currentLatency: endTimestamp - beginTimestampPrev, isolation: true);
527                 log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
528                 log.warn(msg.toString());
529                 exception = e;
530                 continue;
531             } catch (MQClientException e) {
532                 endTimestamp = System.currentTimeMillis();
533                 this.updateFaultItem(mq.getBrokerName(), currentLatency: endTimestamp - beginTimestampPrev, isolation: true);
534                 log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
535                 log.warn(msg.toString());
536                 exception = e;
537                 continue;
538             } catch (MQBrokerException e) {
539                 endTimestamp = System.currentTimeMillis();
540                 this.updateFaultItem(mq.getBrokerName(), currentLatency: endTimestamp - beginTimestampPrev, isolation: true);
541                 log.warn(String.format("sendKernelImpl exception, resend at once, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginTi
542                 log.warn(msg.toString());
543                 exception = e;
544                 switch (e.getResponseCode()) {...}
559             } catch (InterruptedException e) {
560                 endTimestamp = System.currentTimeMillis();
561                 this.updateFaultItem(mq.getBrokerName(), currentLatency: endTimestamp - beginTimestampPrev, isolation: false);
562                 log.warn(String.format("sendKernelImpl exception, throw exception, InvokeID: %s, RT: %sms, Broker: %s", invokeID, endTimestamp - beginT
```

MPOOM

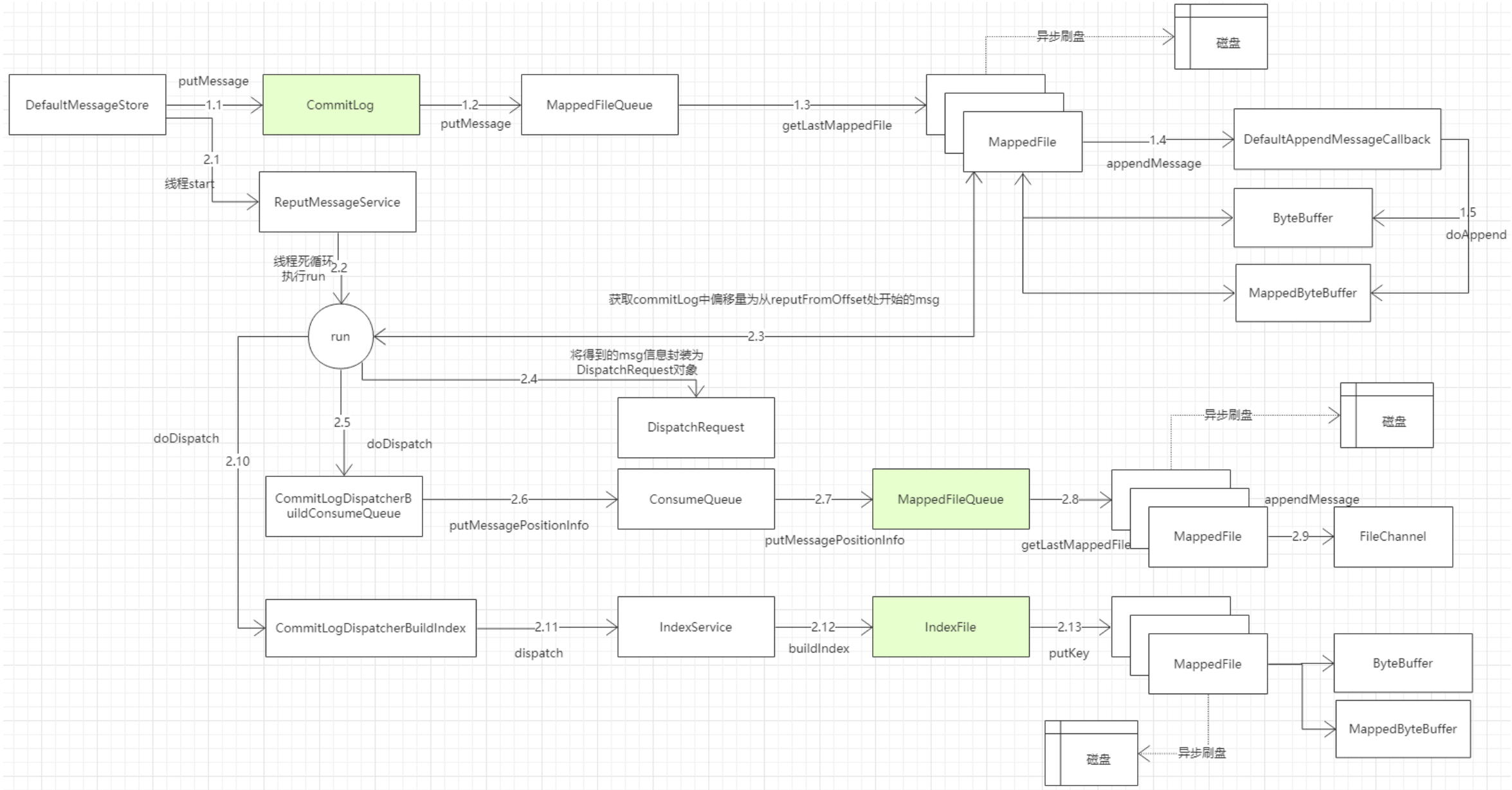
2.3 消息存储



消息存储过程

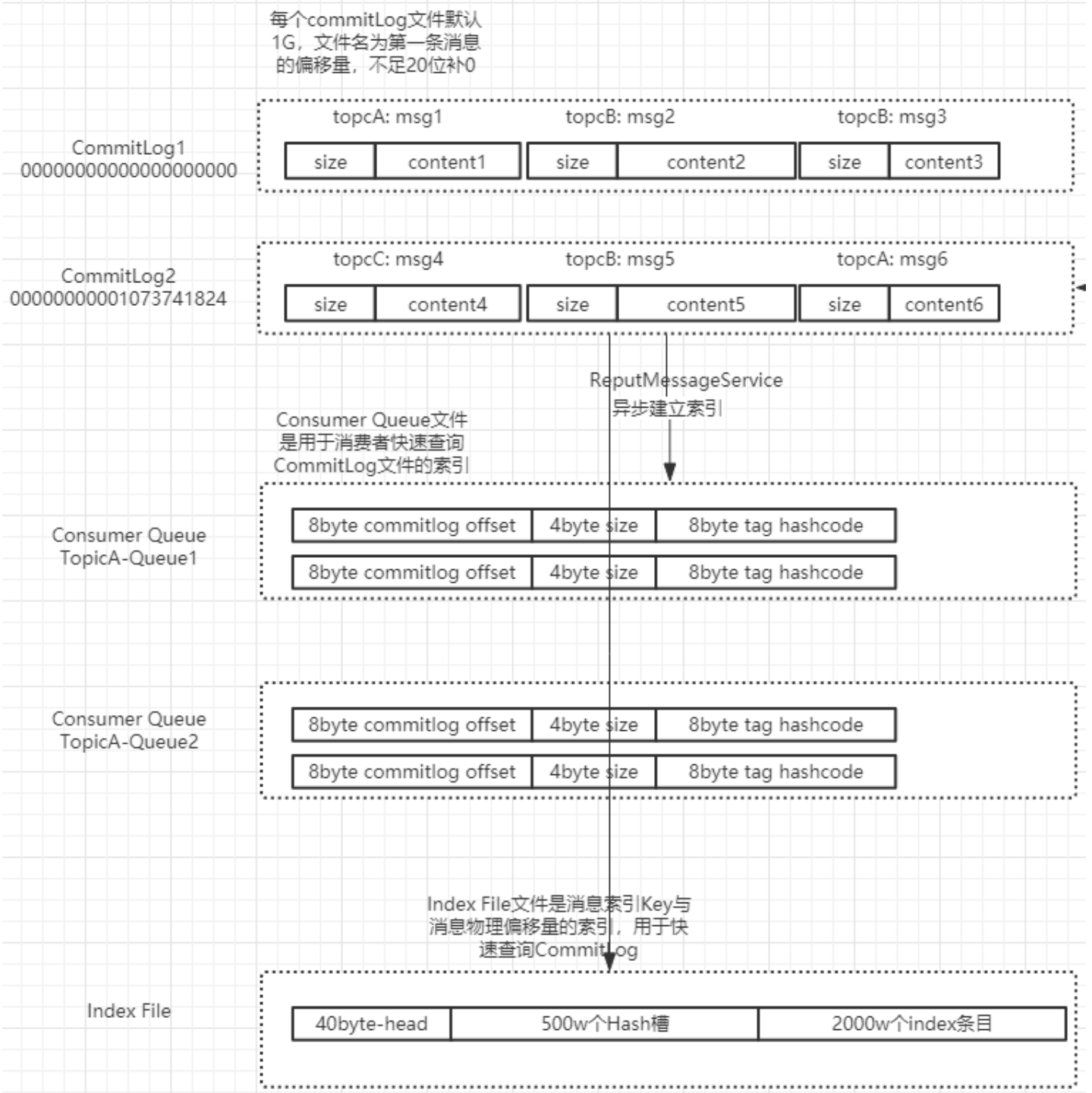
MPOOM

消息存储详细过程



MPOOM

CommitLog、ConsumeQueue、IndexFile 数据结构



MPOOM

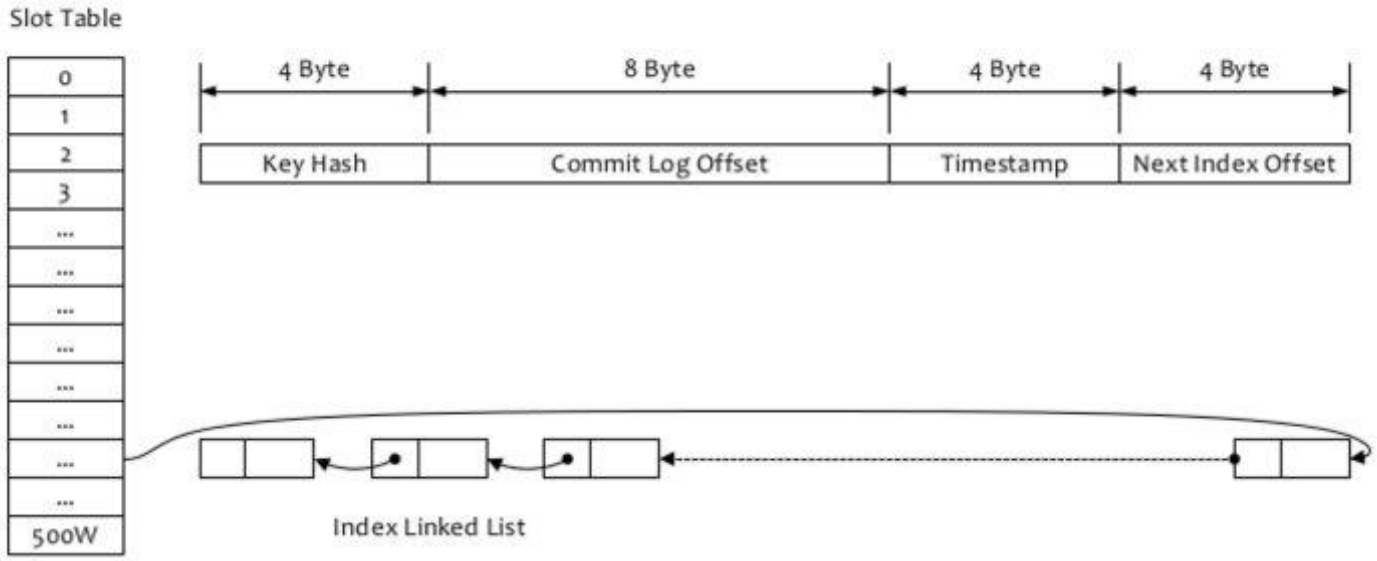
CommitLog数据结构

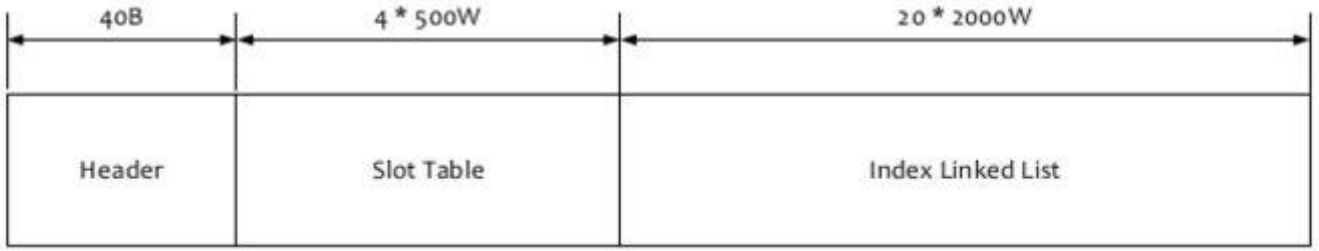
CommitLog.java

```
1836            // 1 TOTALSIZE
1837            this.msgBatchMemory.putInt(msgLen);
1838            // 2 MAGICCODE
1839            this.msgBatchMemory.putInt(CommitLog.MESSAGE_MAGIC_CODE);
1840            // 3 BODYCRC
1841            this.msgBatchMemory.putInt(bodyCrc);
1842            // 4 QUEUEID
1843            this.msgBatchMemory.putInt(messageExtBatch.getQueueId());
1844            // 5 FLAG
1845            this.msgBatchMemory.putInt(flag);
1846            // 6 QUEUEOFFSET
1847            this.msgBatchMemory.putLong(0);
1848            // 7 PHYSICALOFFSET
1849            this.msgBatchMemory.putLong(0);
1850            // 8 SYSFLAG
1851            this.msgBatchMemory.putInt(messageExtBatch.getSysFlag());
1852            // 9 BORNTIMESTAMP
1853            this.msgBatchMemory.putLong(messageExtBatch.getBornTimestamp());
1854            // 10 BORNHOST
1855            this.resetByteBuffer(bornHostHolder, bornHostLength);
1856            this.msgBatchMemory.put(messageExtBatch.getBornHostBytes(bornHostHolder));
1857            // 11 STORETIMESTAMP
1858            this.msgBatchMemory.putLong(messageExtBatch.getStoreTimestamp());
1859            // 12 STOREHOSTADDRESS
1860            this.resetByteBuffer(storeHostHolder, storeHostLength);
1861            this.msgBatchMemory.put(messageExtBatch.getStoreHostBytes(storeHostHolder));
1862            // 13 RECONSUMETIMES
1863            this.msgBatchMemory.putInt(messageExtBatch.getReconsumeTimes());
1864            // 14 Prepared Transaction Offset, batch does not support transaction
1865            this.msgBatchMemory.putLong(0);
1866            // 15 BODY
1867            this.msgBatchMemory.putInt(bodyLen);
1868            if (bodyLen > 0)
1869                this.msgBatchMemory.put(messagesByteBuff.array(), bodyPos, bodyLen);
1870            // 16 TOPIC
1871            this.msgBatchMemory.put((byte) topicLength);
```

MPOOM

IndexFile数据结构





MPOOM

2.4 Consumer负载均衡

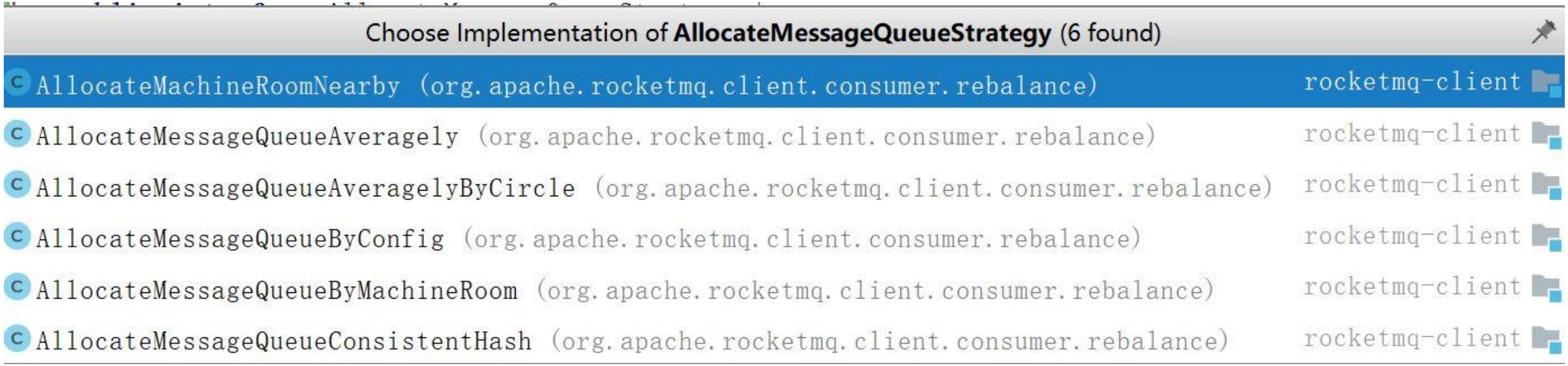
(1) 从rebalanceImpl实例的本地缓存变量—topicSubscribeInfoTable中，获取该Topic主题下的消息消费队列集合（mqSet）；

(2) 根据topic和consumerGroup为参数调用mQClientFactory.findConsumerIdList()方法向Broker端发送获取该消费组下消费者Id列表的RPC通信请求（Broker端基于前面Consumer端上报的心跳包数据而构建的consumerTable做出响应返回，业务请求码：GET_CONSUMER_LIST_BY_GROUP）；

(3) 先对Topic下的消息消费队列、消费者Id排序，然后用消息队列分配策略算法（默认为：消息队列的平均分配算法），计算出待拉取的消息队列。

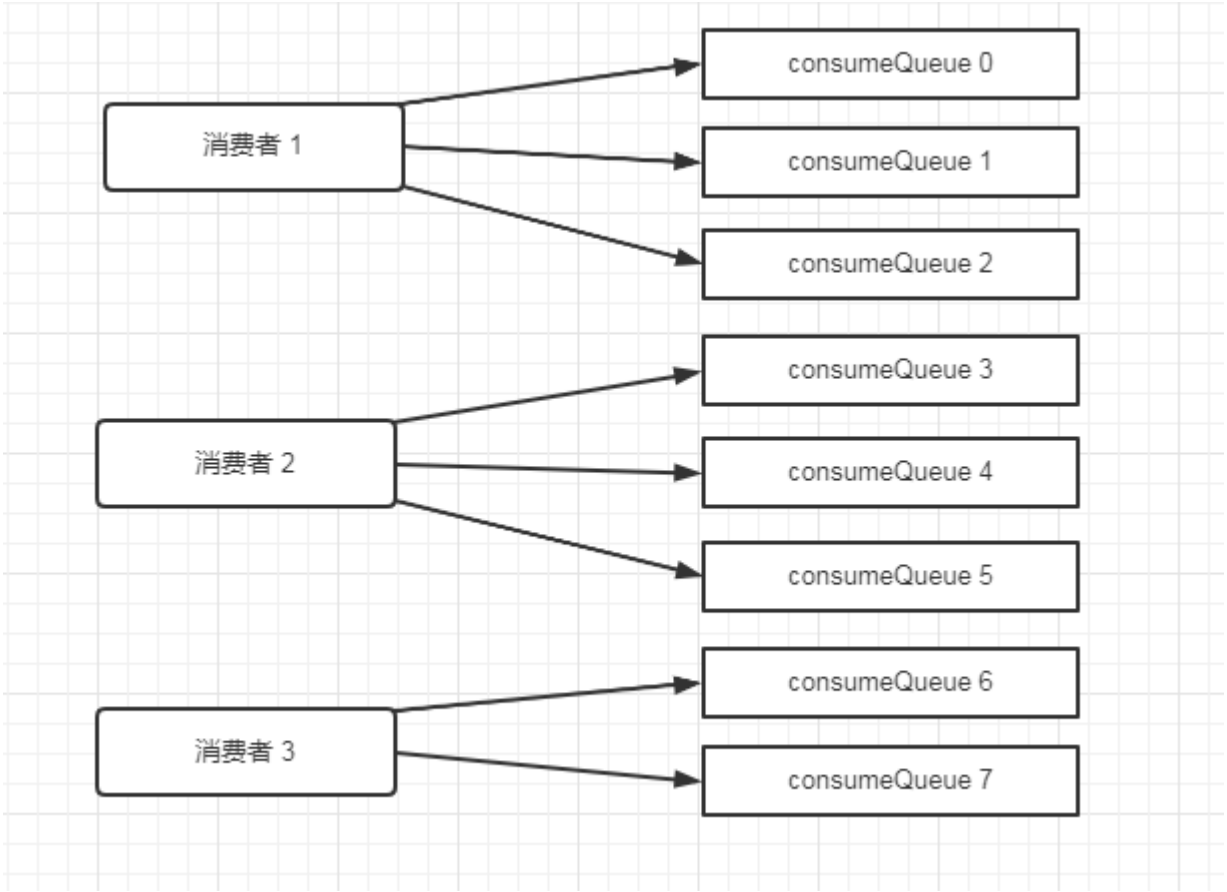
MPOOM

2.4.1 Consumer负载均衡算法



MPOOM

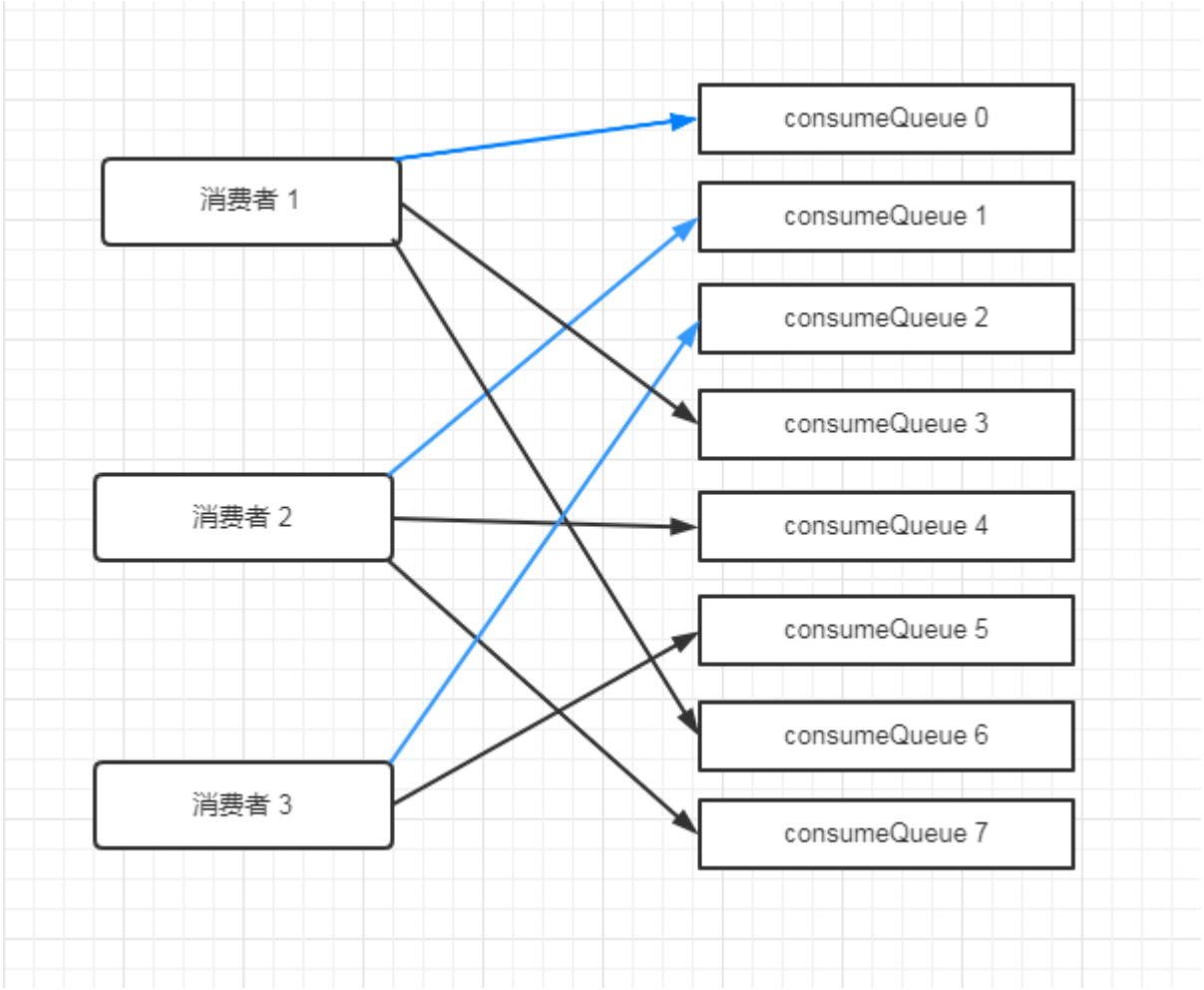
2.4.2 AllocateMessageQueueAveragely分配算法



这里的平均分配算法，类似于分页的算法，将所有MessageQueue排好序类似于记录，将所有消费端Consumer排好序类似页数，并求出每一页需要包含的平均size和每个页面记录的范围range，最后遍历整个range而计算出当前Consumer端应该分配到的记录（这里即为：MessageQueue）

MPOOM

2.4.3 AllocateMessageQueueAveragelyByCircle分配算法



MPOOM

2.4.4 Broker本地文件consumerOffset.json中的消费队列偏移量

```
root@fangwenjie-All-Series:/media/fangwenjie/data/boy/component/data/rocketmq-4.7.0/store/config# vim consumerOffset.json

{
        "offsetTable":{
                "found-pressure-test@ytx-statistics-group":{0:5076795,1:5076794,2:5076798,3:5076791
                },
                "hyk-mq-test@g-hyk-mq-test":{0:52,1:52,2:51,3:51,4:51,5:47,6:48,7:48,8:50,9:50,10:50,11:51,12:52,13:51,14:53,15:53
                },
                "mo_msg@mo-ticket-consumer-g":{0:2,1:2,2:2,3:2,4:2,5:2,6:2,7:2
                },
                "%RETRY%g_order_consumer@g_order_consumer":{0:0
                },
                "%RETRY%consumer-rattrap-mq@consumer-rattrap-mq":{0:0
                },
                "hyk-mq-test@huyaoke-first-consumer":{0:25,1:24,2:24,3:24,4:24,5:23,6:24,7:24,8:27,9:26,10:26,11:25,12:26,13:26,14:26,15:26
                },
                "RMQ_SYS_TRANS_HALF_TOPIC@CID_RMQ_SYS_TRANS":{0:76
                },
                "hyk-mq-test@g_hyk_scheduled_test":{0:58,1:58,2:57,3:60,4:57,5:53,6:54,7:54,8:56,9:59,10:57,11:58,12:59,13:58,14:59,15:63
                },
                "phone-detection-message@ytx-statistics-group":{0:1421120,1:1416914,2:1415521,3:1414114
                },
                "%RETRY%g_message_filter_consumer@g_message_filter_consumer":{0:0
                },
                "%RETRY%huyaoke-first-consumer@huyaoke-first-consumer":{0:0
                },
                "ytx-statistics@ytx-statistics-group":{0:109,1:109,2:109,3:109
                },
                "%RETRY%short-message@short-message":{0:6208
                },
                "%RETRY%mo-ticket-consumer-g@mo-ticket-consumer-g":{0:0
                },
                "phone@consumer-rattrap-mq":{0:779383,1:779385,2:779385,3:779387
                },
                "%RETRY%g-hyk-mq-test@g-hyk-mq-test":{0:0
                },
                "%RETRY%ytx-statistics-group@ytx-statistics-group":{0:6104
                },
                "phone-detection-message@short-message":{0:1421120,1:1416914,2:1415521,3:1414114
                },
                "%RETRY%g_hyk_scheduled_test@g_hyk_scheduled_test":{0:0
```

03

实现原理

1、Mmap实现原理

MPOOM

3. Mmap原理分析

1. 页缓存与内存映射
2. 常规读取文件流程分析
3. Mmap方式读取文件流程分析
4. Mmap在java中的使用
5. Linux Overcommit特性

MPOOM

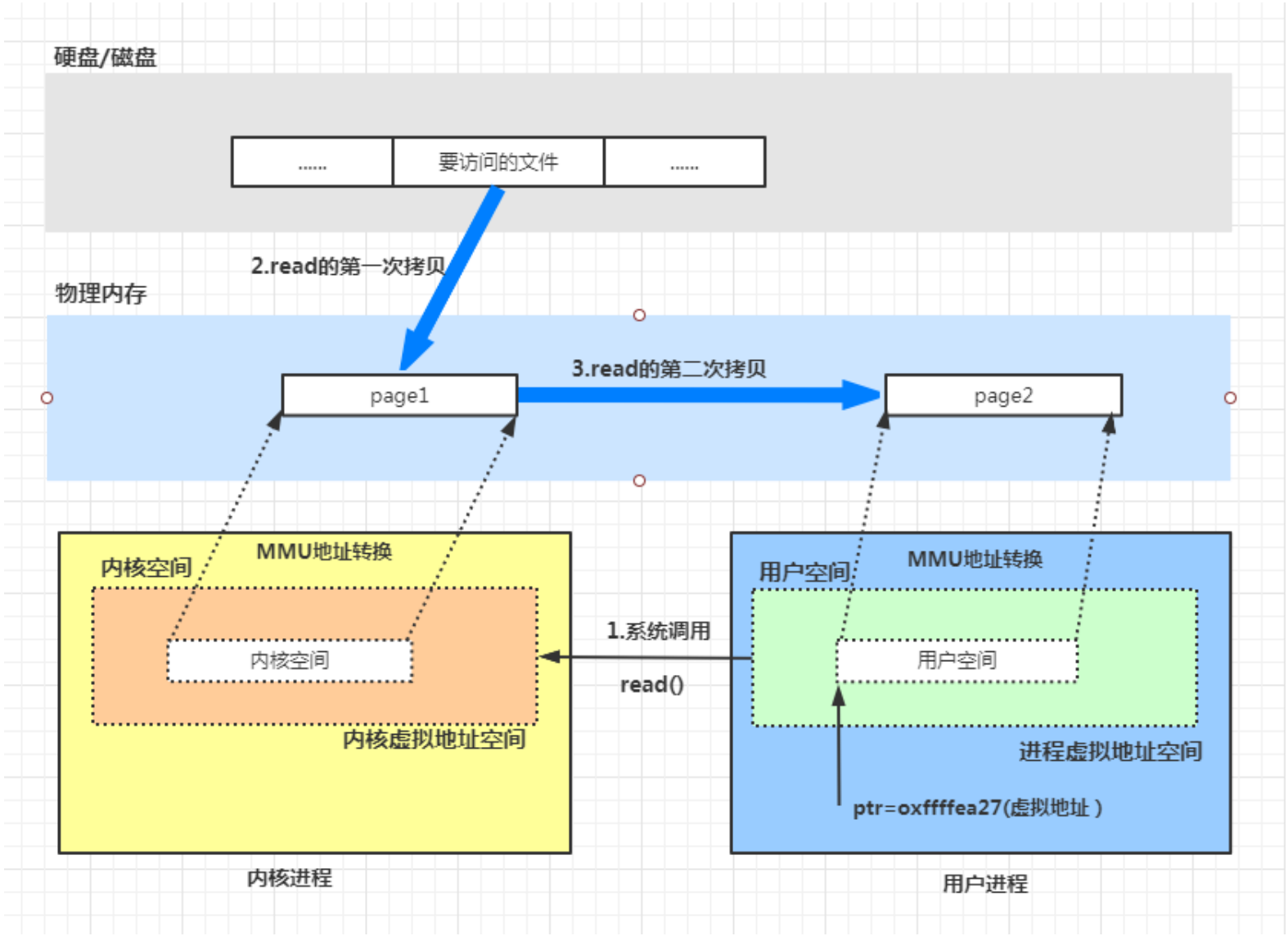
3.1 页缓存与内存映射

页缓存（PageCache)是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”（此时块存储采用SSD的话），随机读的性能也会有所提升。

MPOOM

3.2 常规文件读写流程



1. 进程发起读文件请求， 调用read()函数。

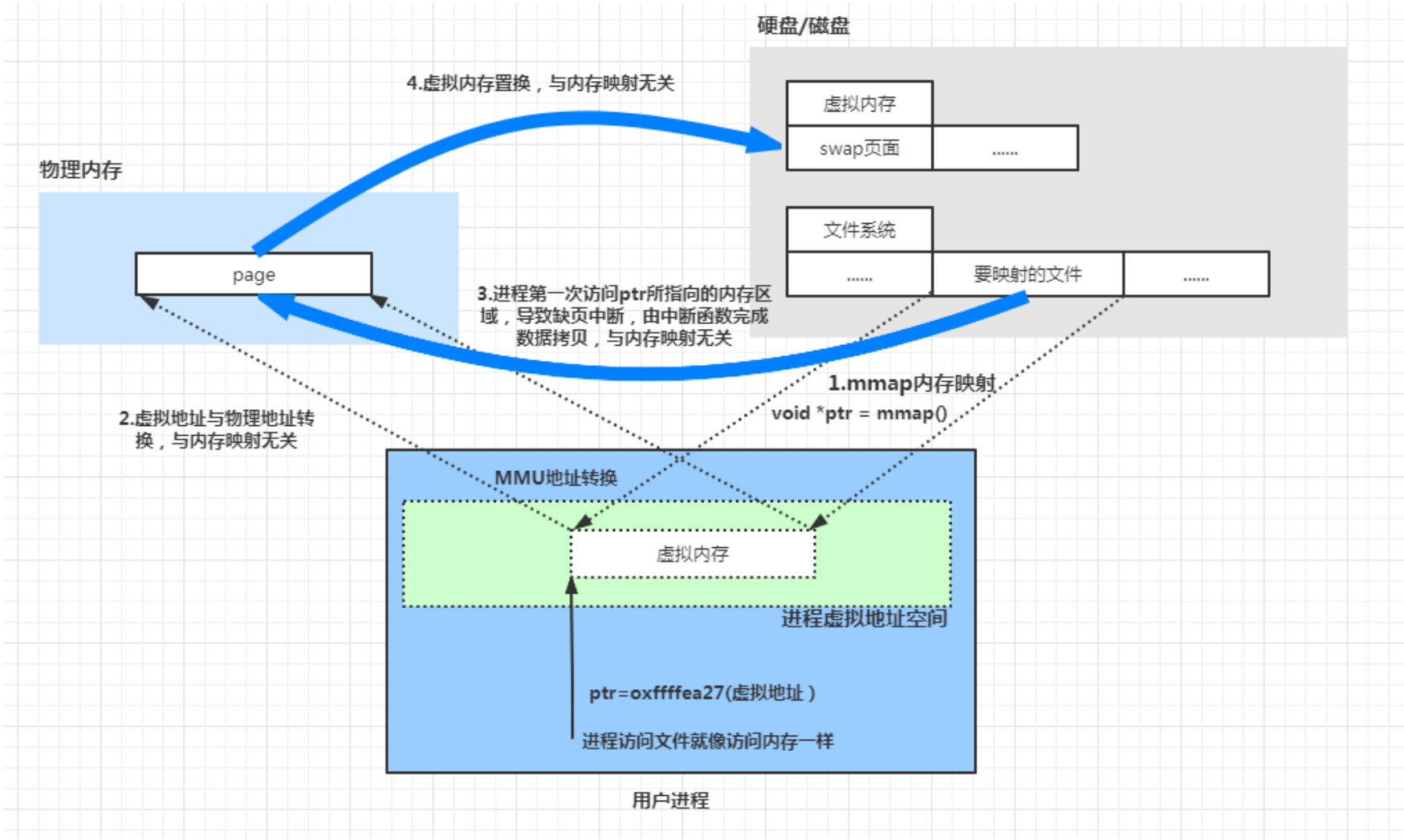
2. 内核通过查找进程文件符表，定位到内核已打开文件集上的文件信息，从而找到此文件的inode。

3. inode在address_space上查找要请求的文件页是否已经缓存在页缓存中。如果存在，则直接返回这片文件页的内容。

4. 如果不存在，则通过inode定位到文件磁盘地址，将数据从磁盘复制到页缓存。之后再次发起读页面过程，进而将页缓存中的数据发给用户进程。

MPOOM

3.3 Mmap读写文件流程



1.进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域。

2. 调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系。

3. 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝。

MPOOM

RocketMQ Broker占用内存分析

```
root@fangwenjie-All-Series:/home/huyaoke# jmap -heap 9748
Attaching to process ID 9748, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.221-b11

using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 40
   MaxHeapFreeRatio         = 70
   MaxHeapSize              = 4294967296 (4096.0MB)
   NewSize                  = 2147483648 (2048.0MB)
   MaxNewSize               = 2147483648 (2048.0MB)
   OldSize                  = 5452592 (5.1999969482421875MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 16777216 (16.0MB)

Heap Usage:
G1 Heap:
   regions  = 256
   capacity = 4294967296 (4096.0MB)
   used     = 445153024 (424.531005859375MB)
   free     = 3849814272 (3671.468994140625MB)
   10.364526510238647% used
G1 Young Generation:
Eden Space:
   regions  = 23
   capacity = 2231369728 (2128.0MB)
   used     = 385875968 (368.0MB)
   free     = 1845493760 (1760.0MB)
   17.293233082706767% used
Survivor Space:
   regions  = 2
   capacity = 33554432 (32.0MB)
   used     = 33554432 (32.0MB)
   free     = 0 (0.0MB)
```

```
root@fangwenjie-All-Series:/home/huyaoke# cat /proc/9748/status
Name:   java
Umask:  0002
State:  S (sleeping)
Tgid:   9748
Ngid:   0
Pid:    9748
PPid:   9745
TracerPid:      0
Uid:    998     998     998     998
Gid:    998     998     998     998
FDSize: 256
Groups: 994 998
NStgid: 9748
NSpid:  9748
NSpgid: 9740
NSsid:  4631
VmPeak:  9979796 kB
VmSize:  9979796 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:   4706680 kB
VmRSS:   4506280 kB
RssAnon:         4495784 kB
RssFile:           10496 kB
RssShmem:              0 kB
VmData:  4859152 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:     20212 kB
VmPTE:     10120 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
Threads:        157
```

MPOOM

3.3 Mmap在Java中的使用

```java
public static void mmapOperateFile(String content) throws IOException  {
    RandomAccessFile  raf = new RandomAccessFile( name: "C:\\Users\\0959\\Desktop\\mmap-test.txt",  mode: "rw");
    MappedByteBuffer mappedByteBuffer = raf.getChannel().map(FileChannel.MapMode.READ_WRITE,  position: 0, FILE_SIZE);
    raf.close();
    long start = System.currentTimeMillis();
    mappedByteBuffer.put(content.getBytes());
    mappedByteBuffer.force();
    System.out.println("time:" + (System.currentTimeMillis() - start));
}
```

```java
/**
* 加载该缓存的内容到物理内存中。这是因为mapp完成后，OS并没有直接读取文件的内容，当真正要访问的时候，通过缺页异常来进行读磁盘操作。
*/
public final MappedByteBuffer load() {
}
```

```java
/**
* 强制将修改后的的内容写入到存储设备上。
* 需要注意的是：如果是本地设备，那么该方法返回时，确保自从该缓存区创建后或该方法最后一次调用后，变更的内容一定写入了设备，如果是网络文件则没有该保证。
* 如果不是通过`MapMode.READ_WRITE`模式映射的，调用该方法没有任何影响。
*/
public final MappedByteBuffer force() {
}
```

MPOOM

3.3.1 Mmap堆外内存释放

DirectByteBuffer.java

```
114
115     // Primary constructor
116     //
117     DirectByteBuffer(int cap) {                   // package-private
118
119         super( mark: -1,  pos: 0, cap, cap);
120         boolean pa = VM.isDirectMemoryPageAligned();
121         int ps = Bits.pageSize();
122         long size = Math.max(1L, (long)cap + (pa ? ps : 0));
123         Bits.reserveMemory(size, cap);
124
125         long base = 0;
126         try {
127             base = unsafe.allocateMemory(size);
128         } catch (OutOfMemoryError x) {
129             Bits.unreserveMemory(size, cap);
130             throw x;
131         }
132         unsafe.setMemory(base, size, (byte) 0);
133         if (pa && (base % ps != 0)) {
134             // Round up to page boundary
135             address = base + ps - (base & (ps - 1));
136         } else {
137             address = base;
138         }
139         cleaner = Cleaner.create( o: this, new Deallocator(base, size, cap));
140         att = null;
141
142
143
144     }
145
```

DirectByteBuffer.java

```
69
70
71
72          private static class Deallocator
73              implements Runnable
74          {
75
76              private static Unsafe unsafe = Unsafe.getUnsafe();
77
78              private long address;
79              private long size;
80              private int capacity;
81
82      @       private Deallocator(long address, long size, int capacity) {
83                  assert (address != 0);
84                  this.address = address;
85                  this.size = size;
86                  this.capacity = capacity;
87              }
88
89              public void run() {
90                  if (address == 0) {
91                      // Paranoia
92                      return;
93                  }
94                  unsafe.freeMemory(address);
95                  address = 0;
96                  Bits.unreserveMemory(size, capacity);
97              }
98
99          }
100
```

MPOOM

3.3.2 DirectByteBuffer中的Cleaner

Cleaner.class

Decompiled .class file, bytecode version: 52.0 (Java 8)

```
1   /.../
5
6   package sun.misc;
7
8   import ...
12
13  public class Cleaner extends PhantomReference
14      private static final ReferenceQueue<Objec
15      private static Cleaner first = null;
16      private Cleaner next = null;
17      private Cleaner prev = null;
18      private final Runnable thunk;
19
20  @   private static synchronized Cleaner add(C
21          if (first != null) {
22              var0.next = first;
23              first.prev = var0;
24          }
25
26          first = var0;
27          return var0;
28      }
```

Cleaner.class

Decompiled .class file, bytecode version: 52.0 (Java 8)

```
58              this.thunk = var2;
59      }
60
61  @   public static Cleaner create(Object var0, Runnable var1) {
62          return var1 == null ? null : add(new Cleaner(var0, var1));
63      }
64
65      public void clean() {
66          if (remove( var0: this)) {
67              try {
68                  this.thunk.run();
69              } catch (final Throwable var2) {
70                  AccessController.doPrivileged(run() -> {
72                          if (System.err != null) {
73                              (new Error( message: "Cleaner terminated abnormally", var2)).printStackTrace();
74                          }
75
76                          System.exit( status: 1);
77                          return null;
78                  });
80              }
81
82          }
83      }
84  }
85
```

MPOOM

3.4 Linux Overcommit特性

Linux OverCommit特性

谢谢观看!