RocketMQ浅析

RocketMQ简介

介绍

RocketMQ是一个纯Java、分布式、队列模型的开源消息中间件，前身是MetaQ，是阿里参考Kafka特点研发的一个队列模型的消息中间件，后开源给apache基金会成为了apache的顶级开源项目，具有高性能、高可靠、高实时、分布式特点。

发展历程

RocketMQ在发展的过程中还是比较顺利的，先后取得了一些突破性的进展。在备战2016年双十一时，**RocketMq**团队重点做了**两件事情**，优化慢请求与统一存储引擎。

- **优化慢请求**：这里主要是解决在海量高并发场景下降低慢请求对整个集群带来的抖动，**毛刺问题**。这是一个极具挑战的技术活，团队同学经过长达1个多月的跟进调优，从双十一的复盘情况来看，99.996%的延迟落在了10ms以内，**而99.6%的延迟在1ms以内**。优化主要集中在**RocketMQ**存储层算法优化、JVM与操作系统调优。
- **统一存储引擎**：主要解决的消息引擎的高可用，成本问题。在多代消息引擎共存的前提下，我们对Notify的存储模块进行了全面移植与替换。

RocketMQ天生为金融互联网领域而生，追求高可靠、高可用、高并发、低延迟，是一个阿里巴巴由内而外成功孕育的典范，除了阿里集团上千个应用外，根据我们不完全统计，国内至少有上百家单位、科研教育机构在使用。**RocketMQ**在阿里集团也被广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理，**binglog**分发等场景。

它所拥有的功能

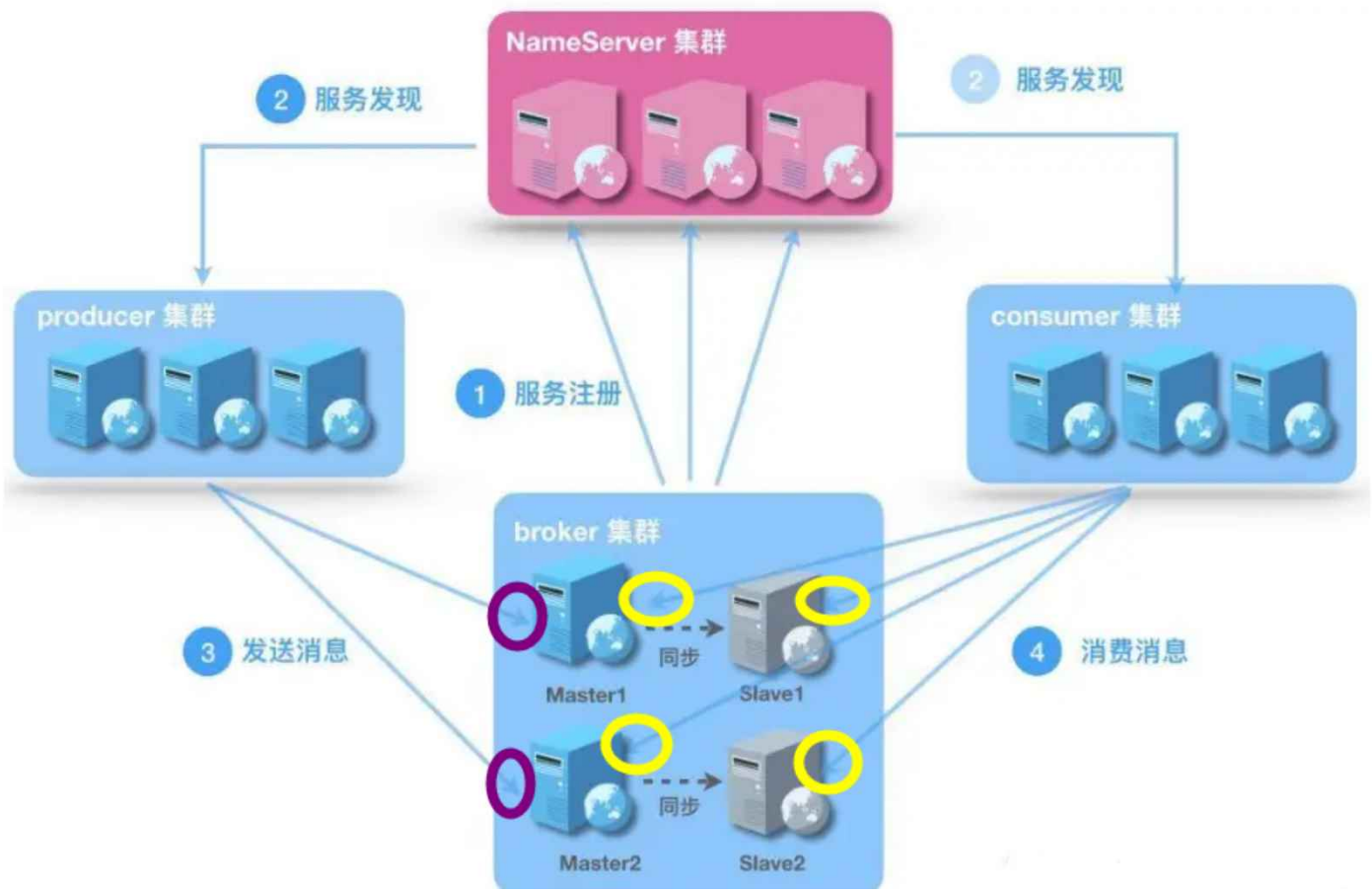
- 发布/订阅消息传递模型
- 财务级交易消息
- 各种跨语言客户端，例如Java，C / C ++，Python，Go
- 可插拔的传输协议，例如TCP，SSL，AIO
- 内置的消息跟踪功能，还支持开放式跟踪
- 多功能的大数据和流生态系统集成
- 按时间或偏移量追溯消息
- 可靠的FIFO和严格的有序消息传递在同一队列中
- 高效的推拉消费模型
- 单个队列中的百万级消息累积容量
- 多种消息传递协议，例如JMS和OpenMessaging
- 灵活的分布式横向扩展部署架构
- 快如闪电的批量消息交换系统

- 各种消息过滤器机制，例如SQL和Tag
- 用于隔离测试和云隔离群集的Docker映像
- 功能丰富的管理仪表板，用于配置，指标和监视
- 认证与授权、

它的核心模块

- rocketmq-broker：接受生产者发来的消息并存储（通过调用rocketmq-store），消费者从这里取得消息
- rocketmq-client：提供发送、接受消息的客户端API。
- rocketmq-namesrv：NameServer，类似于Zookeeper，这里保存着消息的TopicName，队列等运行时的元信息。
- rocketmq-common：通用的一些类，方法，数据结构等。
- rocketmq-remoting：基于Netty4的client/server + fastjson序列化 + 自定义二进制协议。
- rocketmq-store：消息、索引存储等。
- rocketmq-filtersrv：消息过滤器Server，需要注意的是，要实现这种过滤，需要上传代码到MQ！（一般而言，我们利用Tag足以满足大部分的过滤需求，如果更灵活更复杂的过滤需求，可以考虑filtersrv组件）。
- rocketmq-tools：命令行工具。

技术架构



分别介绍下各个集群组成部分

NameServer

主要负责对于源数据的管理，包括了对于**Topic**和路由信息的管理。

NameServer是一个功能齐全的服务器，其角色类似Dubbo中的Zookeeper，但NameServer与Zookeeper相比更轻量。主要是因为每个NameServer节点互相之间是独立的，没有任何信息交互。

NameServer压力不会太大，平时主要开销是在维持心跳和提供Topic-Broker的关系数据。

但有一点需要注意，Broker向NameServer发心跳时， 会带上当前自己所负责的所有**Topic**信息，如果**Topic**个数太多（万级别），会导致一次心跳中，就Topic的数据就几十M，网络情况差的话， 网络传输失败，心跳失败，导致NameServer误认为Broker心跳失败。

NameServer 被设计成几乎无状态的，可以横向扩展，节点之间相互之间无通信，通过部署多台机器来标记自己是一个伪集群。

每个 Broker 在启动的时候会到 NameServer 注册，Producer 在发送消息前会根据 Topic 到 **NameServer** 获取到 Broker 的路由信息，Consumer 也会定时获取 Topic 的路由信息。

所以从功能上看NameServer应该是和 ZooKeeper 差不多，据说 RocketMQ 的早期版本确实是使用的 ZooKeeper ，后来改为了自己实现的 NameServer 。

Producer

消息生产者，负责产生消息，一般由业务系统负责产生消息。

Producer由用户进行分布式部署，消息由**Producer**通过多种负载均衡模式发送到**Broker**集群，发送低延时，支持快速失败。

RocketMQ 提供了三种方式发送消息：同步、异步和单向。

- **同步发送**：同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。一般用于重要通知消息，例如重要通知邮件、营销短信。
- **异步发送**：异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包，一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。
- **单向发送**：单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。

Broker

消息中转角色，负责**存储消息**，转发消息。

Broker是具体提供业务的服务器，单个Broker节点与所有的NameServer节点保持长连接及心跳，并会定时将**Topic**信息注册到NameServer，顺带一提底层的通信和连接都是**基于Netty实现**的。

Broker负责消息存储，以Topic为纬度支持轻量级的队列，单机可以支撑上万队列规模，支持消息推拉模型。

官网上有数据显示：具有**上亿级消息堆积能力**，同时可**严格保证消息的有序性**。

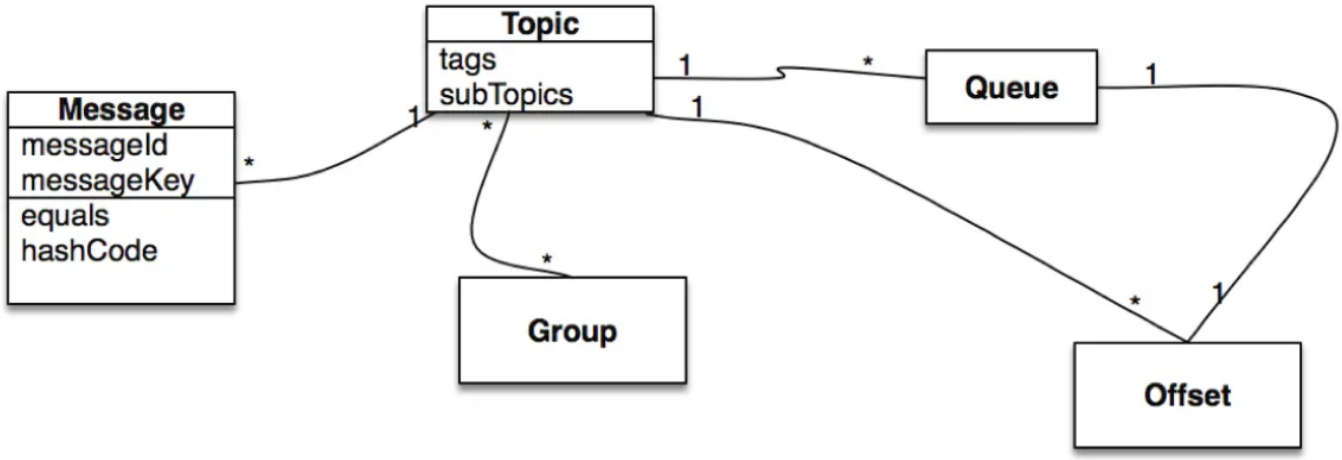
Consumer

消息消费者，负责消费消息，一般是后台系统负责异步消费。

Consumer也由用户部署，支持PUSH和PULL两种消费模式，支持**集群消费**和**广播消息**，提供**实时的消息订阅机制**。

- **Pull**：拉取型消费者（Pull Consumer）主动从消息服务器拉取信息，只要批量拉取到消息，用户应用就会启动消费过程，所以 Pull 称为主动消费型。
- **Push**：推送型消费者（Push Consumer）封装了消息的拉取、消费进度和其他的内部维护工作，将消息到达时执行的回调接口留给用户应用程序来实现。所以 Push 称为被动消费类型，但从实现上看还是从消息服务器中拉取消息，不同于 Pull 的是 Push 首先要注册消费监听器，当监听器处触发后才开始消费消息。

消息领域模型



Message

Message（消息）就是要传输的信息。

一条消息必须有一个主题（Topic），主题可以看做是你的信件要邮寄的地址。

一条消息也可以拥有一个可选的标签（Tag）和额处的键值对，它们可以用于设置一个业务 Key 并在 Broker 上查找此消息以便在开发期间查找问题。

Topic

Topic（主题）可以看做消息的规类，它是消息的第一级类型。比如一个电商系统可以分为：交易消息、物流消息等，一条消息必须有一个 Topic 。

Topic 与生产者和消费者的关系非常松散，一个 Topic 可以有0个、1个、多个生产者向其发送消息，一个生产者也可以同时向不同的 Topic 发送消息。

一个 Topic 也可以被 0个、1个、多个消费者订阅。

Tag

Tag（标签）可以看作子主题，它是消息的第二级类型，用于为用户提供额外的灵活性。使用标签，同一业务模块不同目的的消息就可以用相同 Topic 而不同的 **Tag** 来标识。比如交易消息又可以分为：交易创建消息、交易完成消息等，一条消息可以没有 **Tag** 。

标签有助于保持您的代码干净和连贯，并且还可以为 **RocketMQ** 提供的查询系统提供帮助。

Group

分组，一个组可以订阅多个Topic。

分为ProducerGroup，ConsumerGroup，代表某一类的生产者和消费者，一般来说同一个服务可以作为Group，同一个Group一般来说发送和消费的消息都是一样的

Queue

在**Kafka**中叫Partition，每个Queue内部是有序的，在**RocketMQ**中分为读和写两种队列，一般来说读写队列数量一致，如果不一致就会出现很多问题。

Message Queue

Message Queue（消息队列），主题被划分为一个或多个子主题，即消息队列。

一个 Topic 下可以设置多个消息队列，发送消息时执行该消息的 Topic ，RocketMQ 会轮询该 Topic 下的所有队列将消息发出去。

消息的物理管理单位。一个Topic下可以有多个Queue，Queue的引入使得消息的存储可以分布式集群化，具有了水平扩展能力。

Offset

在**RocketMQ** 中，所有消息队列都是持久化，长度无限的数据结构，所谓长度无限是指队列中的每个存储单元都是定长，访问其中的存储单元使用Offset 来访问，Offset 为 java long 类型，64 位，理论上在 100年内不会溢出，所以认为是长度无限。

也可以认为 Message Queue 是一个长度无限的数组，**Offset** 就是下标。

消息消费模式

消息消费模式有两种：**Clustering**（集群消费）和**Broadcasting**（广播消费）。

默认情况下就是集群消费，该模式下一个消费者集群共同消费一个主题的多个队列，一个队列只会被一个消费者消费，如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。

而广播消费消息会发给消费者组中的每一个消费者进行消费。

RocketMQ工作原理

RocketMQ消息存储和查询原理

从存储方式来看，主要有几个方面：

- 文件系统
- 分布式KV存储
- 关系型数据库

从效率上来讲，文件系统高于KV存储，KV存储又高于关系型数据库。因为直接操作文件系统肯定是最快的，那么业界主流的消息队列中间件，如 `RocketMQ 、RabbitMQ 、kafka` 都是采用文件系统的方式来存储消息。

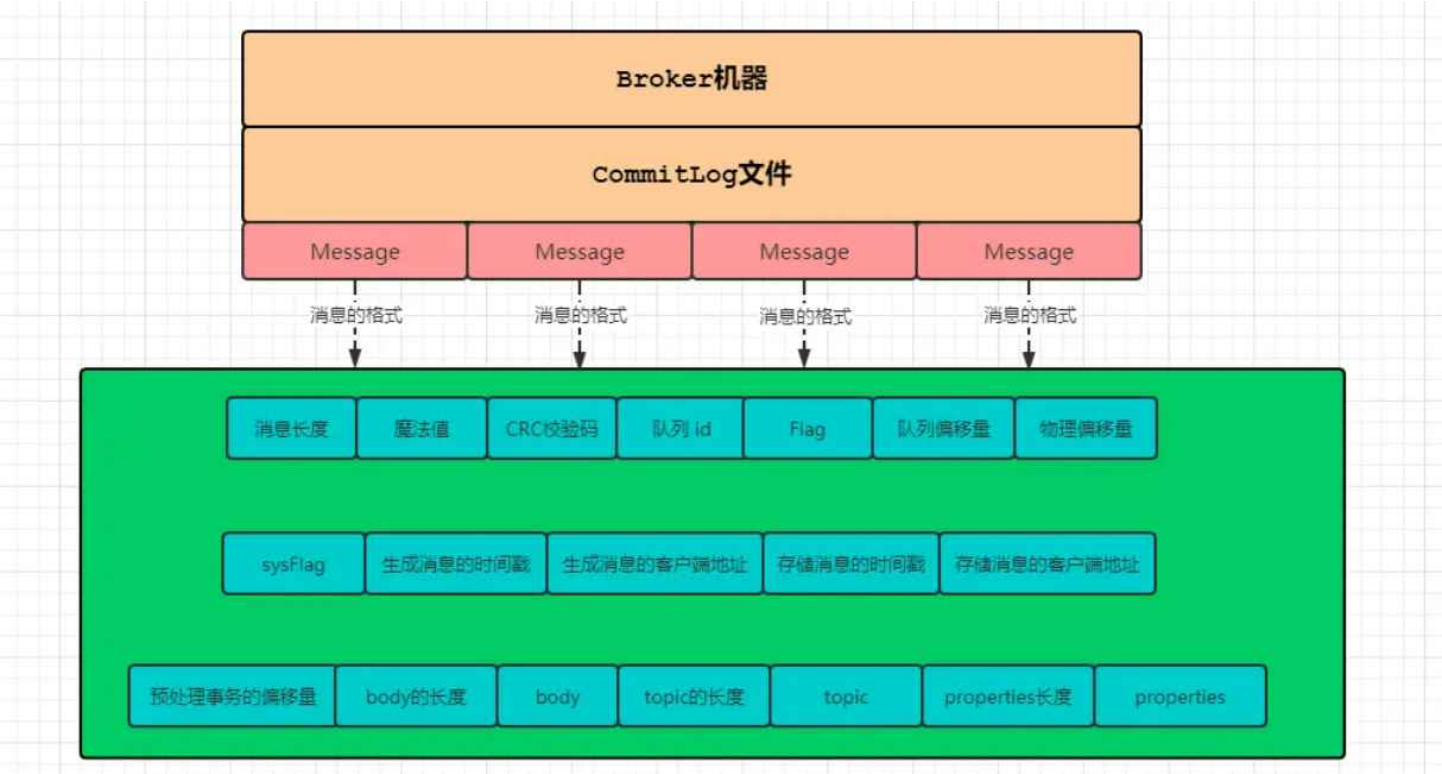
下面，我们就从它的存储文件入手，来探索一下 RocketMQ 消息存储的机制。

一、CommitLog

`CommitLog`，消息存储文件，所有主题的消息都存储在 `CommitLog` 文件中。

我们的业务系统向 `RocketMQ` 发送一条消息，不管在中间经历了多么复杂的流程，最终这条消息会被持久化到 `CommitLog` 文件。

我们知道，一台`Broker服务器`只有一个`CommitLog`文件(组)，`RocketMQ`会将所有主题的消息存储在同一个文件中，这个文件中就存储着一条条Message，每条Message都会按照顺序写入。



也许有时候，你会希望看看这个`CommitLog`文件中，存储的内容到底长什么样子？

1、消息发送

当然，我们需要先往`CommitLog`文件中写入一些内容，所以先来看一个消息发送的例子。

```
public static void main(String[] args) throws Exception {
    MQProducer producer = getProducer();
    for (int i = 0;i < 10;i++){
        Message message = new Message();
        message.setTopic("topic"+i);
        message.setBody(("Eternity的博客").getBytes());
        SendResult sendResult = producer.send(message);
    }
    producer.shutdown();
}
```

我们向10个不同的主题中发送消息，如果只有一台`Broker`机器，它们会保存到同一个`CommitLog`文件中。

2、读取文件内容

这个文件我们不能直接打开，因为它是一个二进制文件，所以我们需要通过程序来读取它的字节数组。

```
public static ByteBuffer read(String path)throws Exception{
    File file = new File(path);
    FileInputStream fin = new FileInputStream(file);
    byte[] bytes = new byte[(int)file.length()];
    fin.read(bytes);
    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    return buffer;
}
```

如上代码，可以通过传入文件的路径，读取该文件所有的内容。为了方便下一步操作，我们把读取到的字节数组转换为`java.nio.ByteBuffer`对象。

3、解析

在解析之前，我们需要弄明白两件事：

- 消息的格式，即一条消息包含哪些字段；
- 每个字段所占的字节大小。

在上面的图中，我们已经看到了消息的格式，包含了19个字段。关于字节大小，有的是 4 字节，有的是 8 字节，我们不再一一赘述，直接看代码。

```java
/**
 * commitlog 文件解析
 * @param byteBuffer
 * @return
 * @throws Exception
 */
public static MessageExt decodeCommitLog(ByteBuffer byteBuffer)throws Exception {

        MessageExt msgExt = new MessageExt();

        // 1 TOTALSIZE
        int storeSize = byteBuffer.getInt();
        msgExt.setStoreSize(storeSize);

        if (storeSize<=0){
            return null;
        }

        // 2 MAGICCODE
        byteBuffer.getInt();

        // 3 BODYCRC
        int bodyCRC = byteBuffer.getInt();
        msgExt.setBodyCRC(bodyCRC);

        // 4 QUEUEID
        int queueId = byteBuffer.getInt();
        msgExt.setQueueId(queueId);

        // 5 FLAG
        int flag = byteBuffer.getInt();
        msgExt.setFlag(flag);

        // 6 QUEUEOFFSET
        long queueOffset = byteBuffer.getLong();
        msgExt.setQueueOffset(queueOffset);

        // 7 PHYSICALOFFSET
        long physicOffset = byteBuffer.getLong();
        msgExt.setCommitLogOffset(physicOffset);

        // 8 SYSFLAG
        int sysFlag = byteBuffer.getInt();
        msgExt.setSysFlag(sysFlag);

        // 9 BORNTIMESTAMP
        long bornTimeStamp = byteBuffer.getLong();
        msgExt.setBornTimestamp(bornTimeStamp);

        // 10 BORNHOST
        int bornhostIPLength = (sysFlag & MessageSysFlag.BORNHOST_V6_FLAG) == 0 ? 4 : 16;
```

```
        byte[] bornHost = new byte[bornhostIPLength];
        byteBuffer.get(bornHost, 0, bornhostIPLength);
        int port = byteBuffer.getInt();
        msgExt.setBornHost(new InetSocketAddress(InetAddress.getByAddress(bornHost), por
t));

        // 11 STORETIMESTAMP
        long storeTimestamp = byteBuffer.getLong();
        msgExt.setStoreTimestamp(storeTimestamp);

        // 12 STOREHOST
        int storehostIPLength = (sysFlag & MessageSysFlag.STOREHOSTADDRESS_V6_FLAG) == 0
 ? 4 : 16;
        byte[] storeHost = new byte[storehostIPLength];
        byteBuffer.get(storeHost, 0, storehostIPLength);
        port = byteBuffer.getInt();
        msgExt.setStoreHost(new InetSocketAddress(InetAddress.getByAddress(storeHost), po
rt));

        // 13 RECONSUMETIMES
        int reconsumeTimes = byteBuffer.getInt();
        msgExt.setReconsumeTimes(reconsumeTimes);

        // 14 Prepared Transaction Offset
        long preparedTransactionOffset = byteBuffer.getLong();
        msgExt.setPreparedTransactionOffset(preparedTransactionOffset);

        // 15 BODY
        int bodyLen = byteBuffer.getInt();
        if (bodyLen > 0) {
            byte[] body = new byte[bodyLen];
            byteBuffer.get(body);
            msgExt.setBody(body);
        }

        // 16 TOPIC
        byte topicLen = byteBuffer.get();
        byte[] topic = new byte[(int) topicLen];
        byteBuffer.get(topic);
        msgExt.setTopic(new String(topic, CHARSET_UTF8));

        // 17 properties
        short propertiesLength = byteBuffer.getShort();
        if (propertiesLength > 0) {
            byte[] properties = new byte[propertiesLength];
            byteBuffer.get(properties);
            String propertiesString = new String(properties, CHARSET_UTF8);
            Map<String, String> map = string2messageProperties(propertiesString);
        }
        int msgIDLength = storehostIPLength + 4 + 8;
        ByteBuffer byteBufferMsgId = ByteBuffer.allocate(msgIDLength);

```

```
        String msgId = createMessageId(byteBufferMsgId, msgExt.getStoreHostBytes(), msgExt.getCommitLogOffset());
        msgExt.setMsgId(msgId);

        return msgExt;
    }
```

4、输出消息内容

```
public static void main(String[] args) throws Exception {
    String filePath = "C:\\Users\\yangshuo\\store\\commitlog\\00000000000000000000";
    ByteBuffer buffer = read(filePath);
    List<MessageExt> messageList = new ArrayList<>();
    while (true){
        MessageExt message = decodeCommitLog(buffer);
        if (message==null){
            break;
        }
        messageList.add(message);
    }
    for (MessageExt ms:messageList) {
        System.out.println("主题:"+ms.getTopic()+" 消息:"+
            new String(ms.getBody())+"队列ID:"+ms.getQueueId()+" 存储地址:"+ms.getStoreHost());
    }
}
```

运行这段代码，我们就可以直接看到 `CommitLog` 文件中的内容：

```
主题:topic0 消息:Eternity的博客 队列ID:1 存储地址:/192.168.44.1:10911
主题:topic1 消息:Eternity的博客 队列ID:0 存储地址:/192.168.44.1:10911
主题:topic2 消息:Eternity的博客 队列ID:1 存储地址:/192.168.44.1:10911
主题:topic3 消息:Eternity的博客 队列ID:0 存储地址:/192.168.44.1:10911
主题:topic4 消息:Eternity的博客 队列ID:3 存储地址:/192.168.44.1:10911
主题:topic5 消息:Eternity的博客 队列ID:1 存储地址:/192.168.44.1:10911
主题:topic6 消息:Eternity的博客 队列ID:2 存储地址:/192.168.44.1:10911
主题:topic7 消息:Eternity的博客 队列ID:3 存储地址:/192.168.44.1:10911
主题:topic8 消息:Eternity的博客 队列ID:2 存储地址:/192.168.44.1:10911
主题:topic9 消息:Eternity的博客 队列ID:0 存储地址:/192.168.44.1:10911
```

不用过多的文字描述，通过上面这些代码，相信你对 `CommitLog` 文件就有了更进一步的了解。

此时，我们再考虑另外一个问题：

`CommitLog` 文件保存了所有主题的消息，但我们消费时，更多的是订阅某一个主题进行消费。`RocketMQ` 是怎么样进行高效的检索消息的呢？

二、ConsumeQueue

为了解决上面那个问题， `RocketMQ` 引入了 `ConsumeQueue` 消费队列文件。

在继续往下说 `ConsumeQueue` 之前，我们必须先了解到另外一个概念，即 `MessageQueue` 。

1、MessageQueue

我们知道，在发送消息的时候，要指定一个Topic。那么，在创建Topic的时候，有一个很重要的参数 `MessageQueue` 。简单来说，就是你这个Topic对应了多少个队列，也就是几个 `MessageQueue` ，默认是4个。那它的作用是什么呢？

它是一个数据分片的机制。比如我们的Topic里面有100条数据，该Topic默认是4个队列，那么每个队列中大约25条数据。
然后，这些 `MessageQueue` 是和 `Broker` 绑定在一起的，就是说每个 `MessageQueue` 都可能处于不同的 `Broker` 机器上，这取决于你的队列数量和Broker集群。

既然 `MessageQueue` 是多个，那么在消息发送的时候，势必要通过某种方式选择一个队列。默认的情况下，就是通过轮询来获取一个消息队列。

```
1 public MessageQueue selectOneMessageQueue() {
2     int index = this.sendWhichQueue.getAndIncrement();
3     int pos = Math.abs(index) % this.messageQueueList.size();
4     if (pos < 0)
5         pos = 0;
6     return this.messageQueueList.get(pos);
7 }
```

当然， `RocketMQ` 还有一个故障延迟机制，在选择消息队列的时候会复杂一些，我们今天先不讨论。

2、ConsumeQueue

说完了 `MessageQueue` ，我们接着来看 `ConsumerQueue` 。上面我们说，它是为了高效检索主题消息的。

`ConsumerQueue` 也是一组组文件，它的位置在 `C:/Users/yangshuo/store/consumequeue` 。该目录下面是以Topic命名的文件夹，然后再下一级是以 `MessageQueue` 队列ID命名的文件夹，最后才是一个或多个文件。

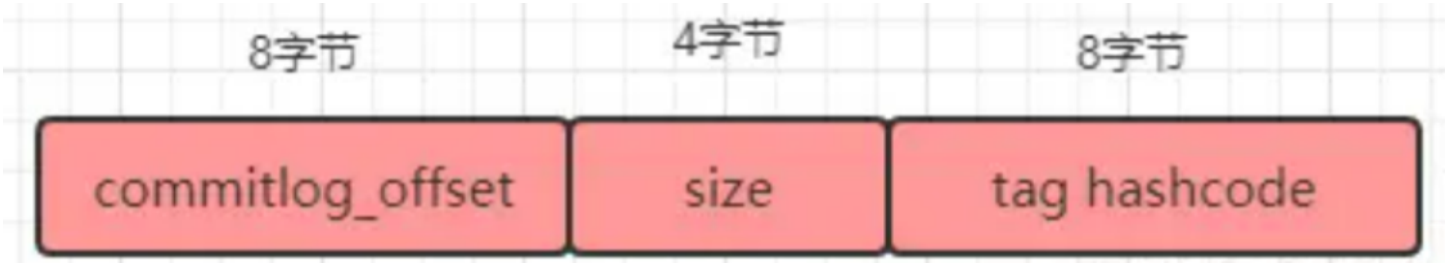
这样分层之后， `RocketMQ` 至少可以得到以下几个讯息：

- 先通过主题名称，可以定位到具体的文件夹；
- 然后根据消息队列ID找到具体的文件；
- 最后根据文件内容，找到具体的消息。

那么，这个文件里面存储的又是什么内容呢？

3、解析文件

为了加速 `ConsumerQueue` 的检索速度和节省磁盘空间，文件中不会存储消息的全量消息。其存储的格式如下：



同样的，我们先写一段代码，按照这个格式输出一下 `ConsumerQueue` 文件的内容。

```
public static void main(String[] args)throws Exception {
    String path = "C:\\Users\\yangshuo\\store\\consumequeue\\order\\0\\00000000000000000000";
    ByteBuffer buffer = read(path);
    while (true){
        long offset = buffer.getLong();
        long size = buffer.getInt();
        long code = buffer.getLong();
        if (size==0){
            break;
        }
        System.out.println("消息长度:"+size+" 消息偏移量:" +offset);
    }
    System.out.println("--------------------------");
}
```

在前面，我们已经向 `order` 这个主题中写了100条数据，所以在这里它的 `order#messagequeue#0` 里面有25条记录。

```
消息长度:173 消息偏移量:2003
消息长度:173 消息偏移量:2695
消息长度:173 消息偏移量:3387
消息长度:173 消息偏移量:4079
消息长度:173 消息偏移量:4771
消息长度:173 消息偏移量:5463
消息长度:173 消息偏移量:6155
消息长度:173 消息偏移量:6847
消息长度:173 消息偏移量:7539
消息长度:173 消息偏移量:8231
消息长度:173 消息偏移量:8923
消息长度:173 消息偏移量:9615
消息长度:173 消息偏移量:10307
消息长度:173 消息偏移量:10999
消息长度:173 消息偏移量:11691
消息长度:173 消息偏移量:12383
消息长度:173 消息偏移量:13075
消息长度:173 消息偏移量:13767
消息长度:173 消息偏移量:14459
消息长度:173 消息偏移量:15151
消息长度:173 消息偏移量:15843
消息长度:173 消息偏移量:16535
消息长度:173 消息偏移量:17227
消息长度:173 消息偏移量:17919
消息长度:173 消息偏移量:18611
--------------------------
```

上面输出的结果中，消息偏移量的差值等于 = 消息长度 * 队列长度。

4、查询消息

现在我们通过 `ConsumerQueue` 已经知道了消息的长度和偏移量，那么查找消息就比较容易了。

```
public static MessageExt getMessageByOffset(ByteBuffer commitLog,long offset,int size) throws Exception {
    ByteBuffer slice = commitLog.slice();
    slice.position((int)offset);
    slice.limit((int) (offset+size));
    MessageExt message = CommitLogTest.decodeCommitLog(slice);
    return message;
}
```

然后，我们可以依靠这种方法，来实现通过 `ConsumerQueue` 获取消息的具体内容。

```
public static void main(String[] args) throws Exception {

        //consumerqueue根目录
        String consumerPath = "C:\\Users\\yangshuo\\store\\consumequeue";
        //commitlog目录
        String commitLogPath = "C:\\Users\\yangshuo\\store\\commitlog\\00000000000000000000";
        //读取commitlog文件内容
        ByteBuffer commitLogBuffer = CommitLogTest.read(commitLogPath);

        //遍历consumerqueue目录下的所有文件
        File file = new File(consumerPath);
        File[] files = file.listFiles();
        for (File f:files) {
                if (f.isDirectory()){
                        File[] listFiles = f.listFiles();
                        for (File queuePath:listFiles) {
                                String path = queuePath+"/00000000000000000000";
                                //读取consumerqueue文件内容
                                ByteBuffer buffer = CommitLogTest.read(path);
                                while (true){
                                        //读取消息偏移量和消息长度
                                        long offset = (int) buffer.getLong();
                                        int size = buffer.getInt();
                                        long code = buffer.getLong();
                                        if (size==0){
                                                break;
                                        }
                                        //根据偏移量和消息长度，在commitloh文件中读取消息内容
                                        MessageExt message = getMessageByOffset(commitLogBuffer,offset,size);
                                        if (message!=null){
                                                System.out.println("消息主题:"+message.getTopic()+" MessageQueue:"+
                                                                message.getQueueId()+" 消息体:"+new String(message.getBody()));
                                        }
                                }
                        }
                }
        }
}

```

运行这段代码，就可以得到之前测试样例中，10个主题的所有消息。

```
消息主题:topic0 MessageQueue:1 消息体:Eternity的博客
消息主题:topic1 MessageQueue:0 消息体:Eternity的博客
消息主题:topic2 MessageQueue:1 消息体:Eternity的博客
消息主题:topic3 MessageQueue:0 消息体:Eternity的博客
消息主题:topic4 MessageQueue:3 消息体:Eternity的博客
消息主题:topic5 MessageQueue:1 消息体:Eternity的博客
消息主题:topic6 MessageQueue:2 消息体:Eternity的博客
消息主题:topic7 MessageQueue:3 消息体:Eternity的博客
消息主题:topic8 MessageQueue:2 消息体:Eternity的博客
消息主题:topic9 MessageQueue:0 消息体:Eternity的博客
```

5、消费消息

消息消费的时候，其查找消息的过程也是差不多的。不过值得注意的一点是，`ConsumerQueue`文件和`CommitLog`文件可能都是多个的，所以会有一个定位文件的过程，我们来看源码。

首先，根据消费进度来查找对应的`ConsumerQueue`，获取其文件内容。

```
public SelectMappedBufferResult getIndexBuffer(final long startIndex) {
    //ConsumerQueue文件大小
    int mappedFileSize = this.mappedFileSize;
    //根据消费进度，找到在consumerqueue文件里的偏移量
    long offset = startIndex * CQ_STORE_UNIT_SIZE;
    if (offset >= this.getMinLogicOffset()) {
        //返回ConsumerQueue映射文件
        MappedFile mappedFile = this.mappedFileQueue.findMappedFileByOffset(offset);
        if (mappedFile != null) {
            //返回文件里的某一块内容
            SelectMappedBufferResult result = mappedFile.selectMappedBuffer((int) (offset % mappedFileSize));
            return result;
        }
    }
    return null;
}
```

然后拿到消息在`CommitLog`文件中的偏移量和消息长度，获取消息。

```java
public SelectMappedBufferResult getMessage(final long offset, final int size) {
    //commitlog文件大小
    int mappedFileSize = this.defaultMessageStore.getMessageStoreConfig().getMappedFileSizeCommitLog();
    //根据消息偏移量，定位到具体的commitlog文件
    MappedFile mappedFile = this.mappedFileQueue.findMappedFileByOffset(offset, offset == 0);
    if (mappedFile != null) {
        //根据消息偏移量和长度，获取消息内容
        int pos = (int) (offset % mappedFileSize);
        return mappedFile.selectMappedBuffer(pos, size);
    }
    return null;
}
```

6、通过 Message Id 查询

上面我们看到了通过消息偏移量来查找消息的方式，但 `RocketMQ` 还提供了其他几种方式可以查询消息。

- 通过Message Key 查询；
- 通过Unique Key查询；
- 通过Message Id查询。

在这里， `Message Key和Unique Key` 都是在消息发送之前，由客户端生成的。我们可以自己设置，也可以由客户端自动生成， `Message Id` 是在 `Broker` 端存储消息的时候生成。

`Message Id` 总共 16 字节，包含消息存储主机地址和在 `CommitLog` 文件中的偏移量offset。有源码为证：

```java
/**
 * 创建消息ID
 * @param input
 * @param addr      Broker服务器地址
 * @param offset    正在存储的消息，在Commitlog中的偏移量
 * @return
 */
public static String createMessageId(final ByteBuffer input, final ByteBuffer addr, final long offset) {
    input.flip();
    int msgIDLength = addr.limit() == 8 ? 16 : 28;
    input.limit(msgIDLength);
    input.put(addr);
    input.putLong(offset);
    return UtilAll.bytes2string(input.array());
}
```

当我们根据 `Message Id` 向Broker查询消息时，首先会通过一个 `decodeMessageId` 方法，将Broker地址和消息的偏移量解析出来。

```java
public static MessageId decodeMessageId(final String msgId) throws Exception {
    SocketAddress address;
    long offset;
    int ipLength = msgId.length() == 32 ? 4 * 2 : 16 * 2;
    byte[] ip = UtilAll.string2bytes(msgId.substring(0, ipLength));
    byte[] port = UtilAll.string2bytes(msgId.substring(ipLength, ipLength + 8));
    ByteBuffer bb = ByteBuffer.wrap(port);
    int portInt = bb.getInt(0);
    //解析出来Broker地址
    address = new InetSocketAddress(InetAddress.getByAddress(ip), portInt);
    //偏移量
    byte[] data = UtilAll.string2bytes(msgId.substring(ipLength + 8, ipLength + 8 + 16));
    bb = ByteBuffer.wrap(data);
    offset = bb.getLong(0);
    return new MessageId(address, offset);
}

```

所以通过 `Message Id` 查询消息的时候，实际上还是直接从特定Broker上的 `CommitLog` 指定位置进行查询，属于精确查询。

这个也没问题，但是如果通过 `Message Key 和 Unique Key` 查询的时候， `RocketMQ` 又是怎么做的呢?

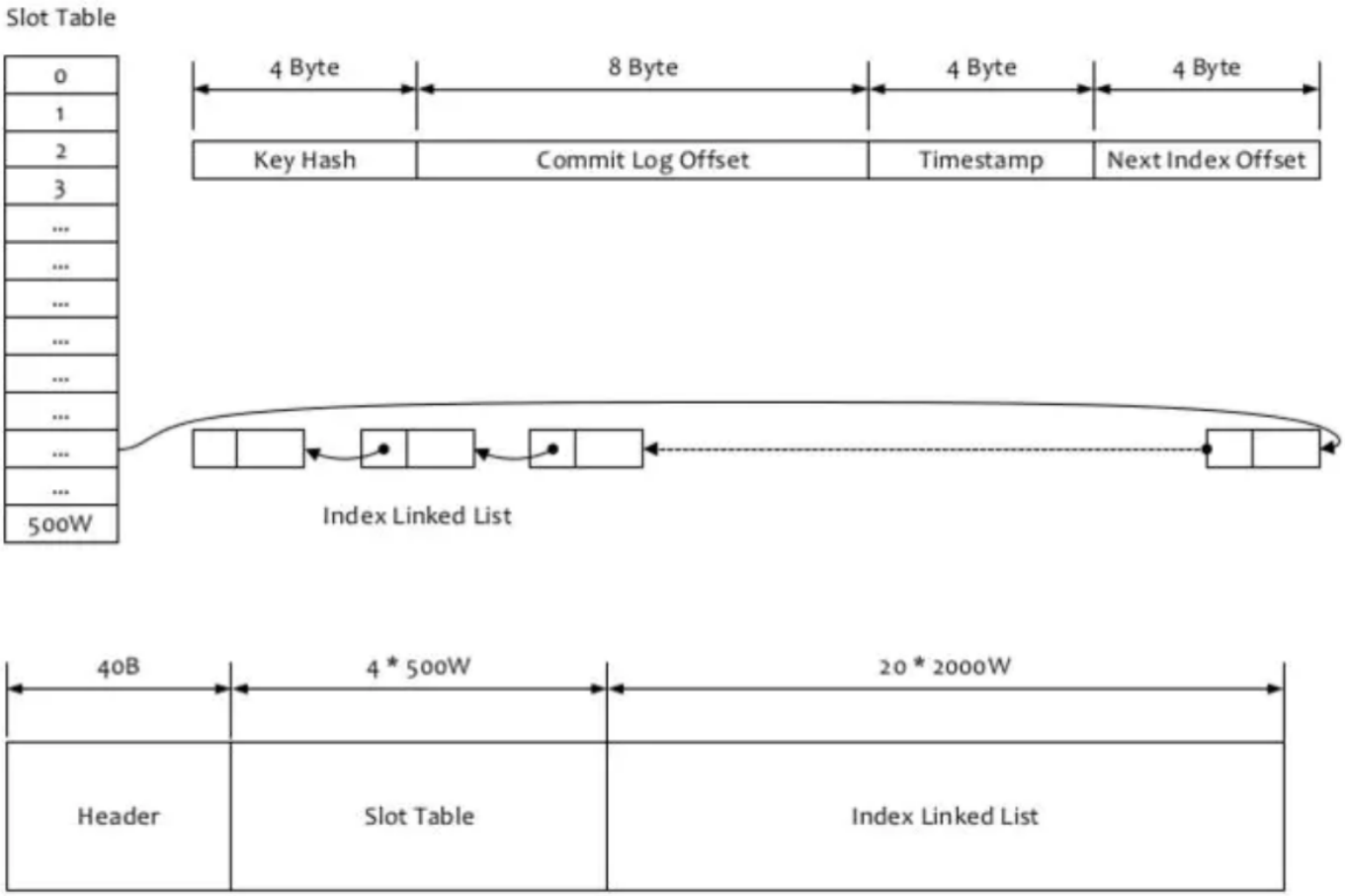
三、Index

1、index索引文件

`ConsumerQueue` 消息消费队列是专门为消息订阅构建的索引文件，提高根据主题与消息队列检索消息的速度。

另外， `RocketMQ` 引入Hash索引机制，为消息建立索引，它的键就是 `Message Key 和 Unique Key` 。

那么，我们先看看index索引文件的结构：



为了便于理解，我们还是以代码的方式，来解析这个文件。

```java
public static void main(String[] args) throws Exception {

    //index索引文件的路径
    String path = "C:\\Users\\yangshuo\\store\\index\\20200506224547616";
    ByteBuffer buffer = CommitLogTest.read(path);
    //该索引文件中包含消息的最小存储时间
    long beginTimestamp = buffer.getLong();
    //该索引文件中包含消息的最大存储时间
    long endTimestamp = buffer.getLong();
    //该索引文件中包含消息的最大物理偏移量(commitlog文件偏移量)
    long beginPhyOffset = buffer.getLong();
    //该索引文件中包含消息的最大物理偏移量(commitlog文件偏移量)
    long endPhyOffset = buffer.getLong();
    //hashslot个数
    int hashSlotCount = buffer.getInt();
    //Index条目列表当前已使用的个数
    int indexCount = buffer.getInt();

    //500万个hash槽，每个槽占4个字节，存储的是index索引
    for (int i=0;i<5000000;i++){
        buffer.getInt();
    }
    //2000万个index条目
    for (int j=0;j<20000000;j++){
        //消息key的hashcode
        int hashcode = buffer.getInt();
        //消息对应的偏移量
        long offset = buffer.getLong();
        //消息存储时间和第一条消息的差值
        int timedif = buffer.getInt();
        //该条目的上一条记录的index索引
        int pre_no = buffer.getInt();
    }
    System.out.println(buffer.position()==buffer.capacity());
}
```

我们看最后输出的结果为true，则证明解析的过程无误。

2、构建索引

我们发送的消息体中，包含`Message Key 或 Unique Key`，那么就会给它们每一个都构建索引。

这里重点有两个：

- 根据消息Key计算Hash槽的位置；
- 根据Hash槽的数量和Index索引来计算Index条目的起始位置。

将当前 **Index条目** 的索引值，写在Hash槽`absSlotPos`位置上；将**Index条目的具体信息**`(hashcode/消息偏移量/时间差值/hash槽的值)`，从起始偏移量`absIndexPos`开始，顺序按字节写入。

```java
public boolean putKey(final String key, final long phyOffset, final long storeTimestamp) {
    if (this.indexHeader.getIndexCount() < this.indexNum) {
        //计算key的hash
        int keyHash = indexKeyHashMethod(key);
        //计算hash槽的坐标
        int slotPos = keyHash % this.hashSlotNum;
        int absSlotPos = IndexHeader.INDEX_HEADER_SIZE + slotPos * hashSlotSize;
        //计算时间差值
        long timeDiff = storeTimestamp - this.indexHeader.getBeginTimestamp();
        timeDiff = timeDiff / 1000;
        //计算INDEX条目的起始偏移量
        int absIndexPos =
            IndexHeader.INDEX_HEADER_SIZE + this.hashSlotNum * hashSlotSize
                + this.indexHeader.getIndexCount() * indexSize;
        //依次写入hashcode、消息偏移量、时间戳、hash槽的值
        this.mappedByteBuffer.putInt(absIndexPos, keyHash);
        this.mappedByteBuffer.putLong(absIndexPos + 4, phyOffset);
        this.mappedByteBuffer.putInt(absIndexPos + 4 + 8, (int) timeDiff);
        this.mappedByteBuffer.putInt(absIndexPos + 4 + 8 + 4, slotValue);
        //将当前INDEX中包含的条目数量写入HASH槽
        this.mappedByteBuffer.putInt(absSlotPos, this.indexHeader.getIndexCount());
        return true;
    }
    return false;
}

```

这样构建完Index索引之后，根据 `Message Key 或 Unique Key` 查询消息就简单了。

比如我们通过 `RocketMQ` 客户端工具，根据 `Unique Key` 来查询消息。

```java
adminImpl.queryMessageByUniqKey("order", "FD88E3AB24F6980059FDC9C3620464741BCC18B4AAC220FDFE890007");
```

在 `Broker` 端，通过 `Unique Key` 来计算Hash槽的位置，从而找到Index索引数据。从Index索引中拿到消息的物理偏移量，最后根据消息物理偏移量，直接到 `CommitLog` 文件中去找就可以了。

总结

由于篇幅原因，并且自己对RocketMQ的理解还不是很到位，就写这么多，期间也查了很多技术博客。总之，在这次开源课程中收获了很多，继续加油。