课程结构目录

- IDEA源码调试环境构建
- NameSrv命名服务架构
- Broker架构
- 文件存储结构说明

Rocketmq源码调试环境构建

1、下载https://github.com/apache/rocketmq/ 对应工程版本文件

2、使用Idea 打开Rocketmq工程。打开工程后我们会看到多个模块，我们需要启动两个服务Namesrv与Broker服务

```
打开Namesrv服务入口类运行
org.apache.rocketmq.namesrv.NamesrvStartup#main

打开Broker服务入口类运行
org.apache.rocketmq.broker.BrokerStartup#main
```

直接运行会启动失败，原因是找不到启动配置文件

```
Please set the ROCKETMQ_HOME variable in your environment to match the location of the RocketMQ installation
```

不配环境变量，需要修改启动类

```
#修改Namesrv启动类
org.apache.rocketmq.namesrv.NamesrvStartup#createNamesrvController
//添加本行代码,路径为Rocketmq项目路径下的子项目distribution工程路径
namesrvConfig.setRocketmqHome("D:\\GitSource\\rocketmq-rocketmq-all-4.3.2\\distribution");
if (null == namesrvConfig.getRocketmqHome()) {
 System.out.printf("Please set the %s variable in your environment to match the location of the RocketMQ installation%n", MixAll.ROCKETMQ_HOME_ENV);
 System.exit(-2);
}
#修改Broker启动类
org.apache.rocketmq.broker.BrokerStartup#createBrokerController
//添加本行代码
```

```
12 brokerConfig.setRocketmqHome("D:\\GitSource\\rocketmq-rocketmq-all-4.3.2
\\distribution");
13 if (null == brokerConfig.getRocketmqHome()) {
14  System.out.printf("Please set the %s variable in your environment to ma
tch the location of the RocketMQ installation", MixAll.ROCKETMQ_HOME_ENV);
15  System.exit(-2);
16 }
```

运行服务

```
1 启动Namesrv
2 -n localhost:9876 &
3 启动Broker
4 -n localhost:9876 -c D:\GitSource\rocketmq-rocketmq-all-4.3.2\distributio
n\conf\broker.conf &
```

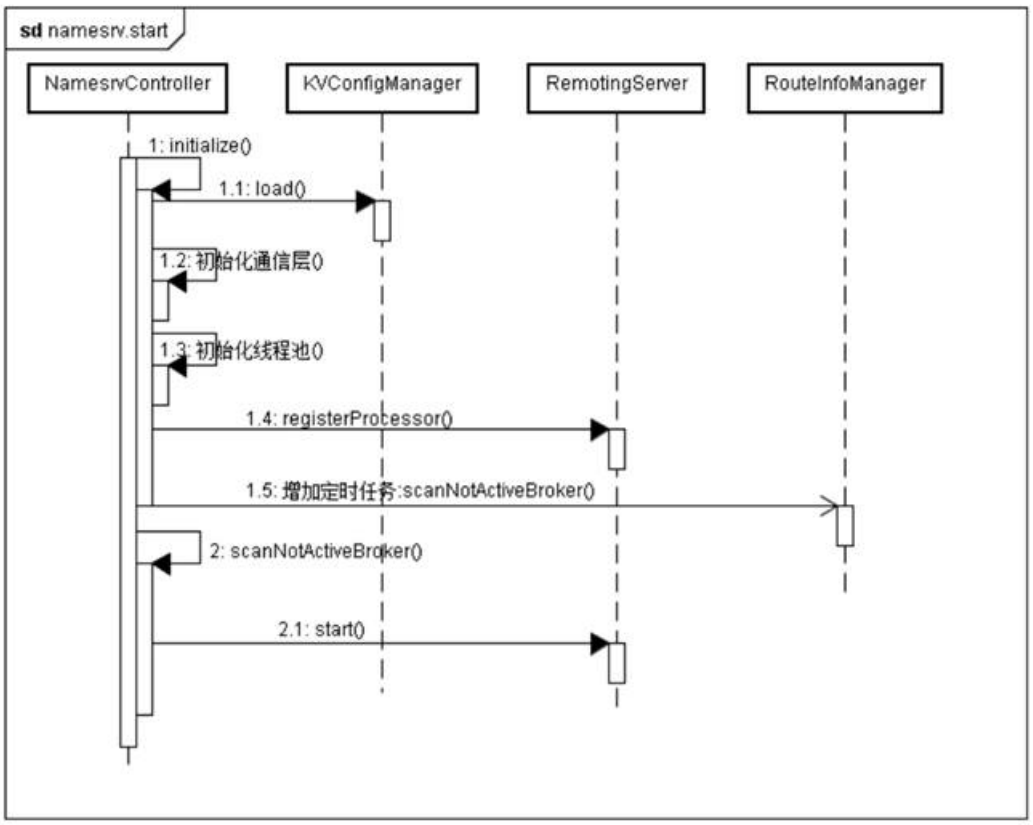
Namesrv架构

NameServer是一个非常简单的Topic路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。主要包括两个功能：

Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；

路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Conumser通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费。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。Broker是向每一台NameServer注册自己的路由信息，所以每一个NameServer实例上面都保存一份完整的路由信息。当某个NameServer因某种原因下线了，Broker仍然可以向其它NameServer同步其路由信息，Producer,Consumer仍然可以动态感知Broker的路由的信息。

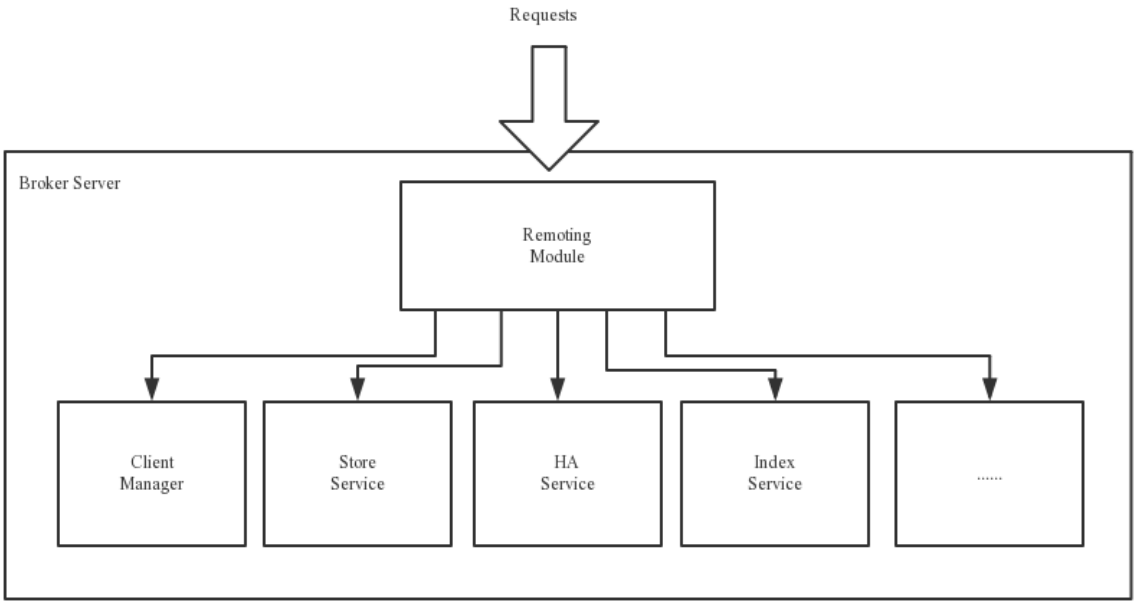
启动时序图



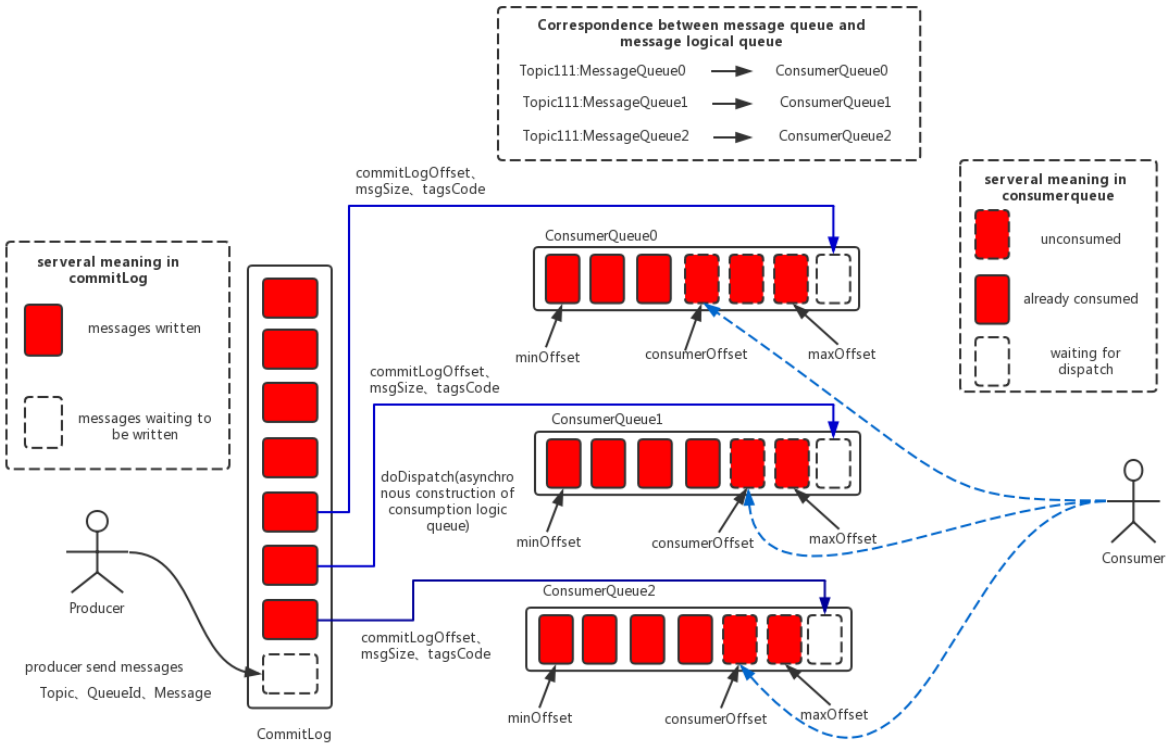
Broker架构

核心模块

- Remoting Module：整个Broker的实体，负责处理来自clients端的请求。
- Client Manager：负责管理客户端(Producer/Consumer)和维护Consumer的Topic订阅信息
- Store Service：提供方便简单的API接口处理消息存储到物理硬盘和查询功能。
- HA Service：高可用服务，提供Master Broker 和 Slave Broker之间的数据同步功能。
- Index Service：根据特定的Message key对投递到Broker的消息进行索引服务，以提供消息的快速查询。



消息存储整体架构



消息存储架构图中主要有下面三个跟消息存储相关的文件构成。

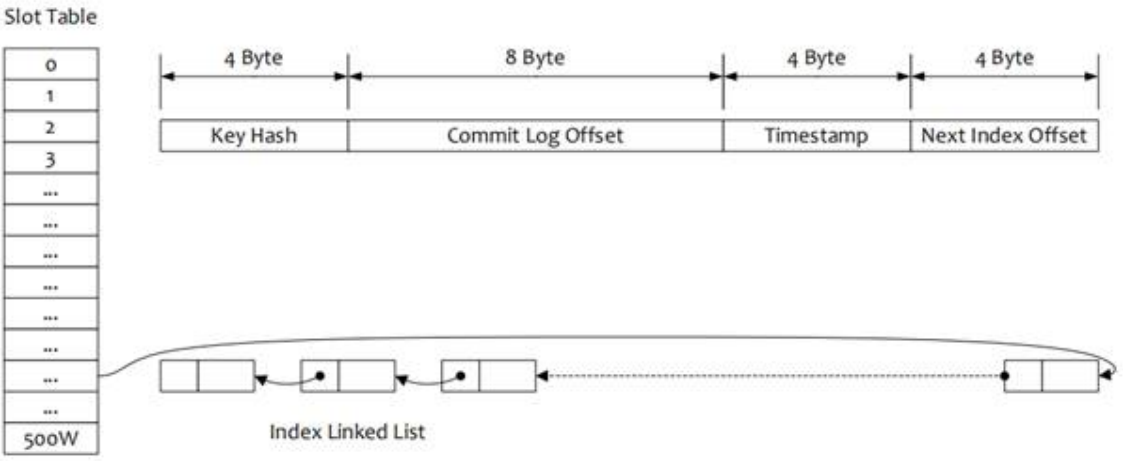
(1) CommitLog：消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容，消息内容不是定长的。单个文件大小默认1G ， 文件名长度为20位，左边补零，剩余为起始

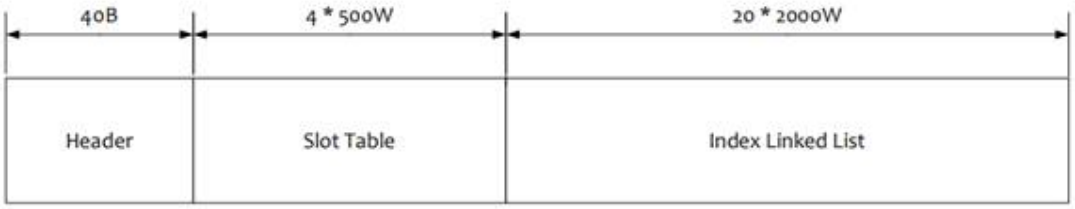
偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件；

(2) ConsumeQueue：消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。consumequeue文件可以看成是基于topic的commitlog索引文件，故consumequeue文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样consumequeue文件采取定长设计，每一个条目共20个字节，分别为8字节的commitlog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M；

(3) IndexFile：IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：$HOME \store\index${fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小：40+500W*4+2000W*20= 420000040个字节大小，约为400M，一个IndexFile可以保存 2000W个索引，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故rocketmq的索引文件其底层实现为hash索引。

索引文件由索引文件头（IndexHeader）+（槽位 Slot）+（消息的索引内容）三部分构成。



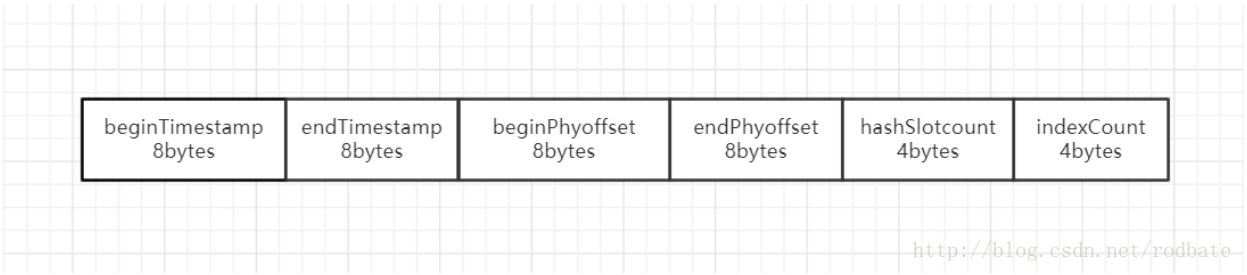



```
org.apache.rocketmq.store.index.IndexFile
#hashSlot占空间4bytes
private static int hashSlotSize = 4;
#index索引内容空间大小 20bytes
private static int indexSize = 20;
private static int invalidIndex = 0;
#槽位数量，默认500w
private final int hashSlotNum;
#索引数量 2000w
 private final int indexNum;
 #索引文件头信息
 private final IndexHeader indexHeader;

 org.apache.rocketmq.store.index.IndexHeader
 IndexHeader结构字段定义
 #信息头占用空间定长40 bytes，由6个部分构成
 public static final int INDEX_HEADER_SIZE = 40;
 #第一个索引消息落在Broker的时间戳 8bytes
 private AtomicLong beginTimestamp = new AtomicLong(0);
 #最后一个索引消息落在Broker的时间戳 8bytes
 private AtomicLong endTimestamp = new AtomicLong(0);
 #第一个索引消息在commitlog的偏移量 8bytes
 private AtomicLong beginPhyOffset = new AtomicLong(0);
```

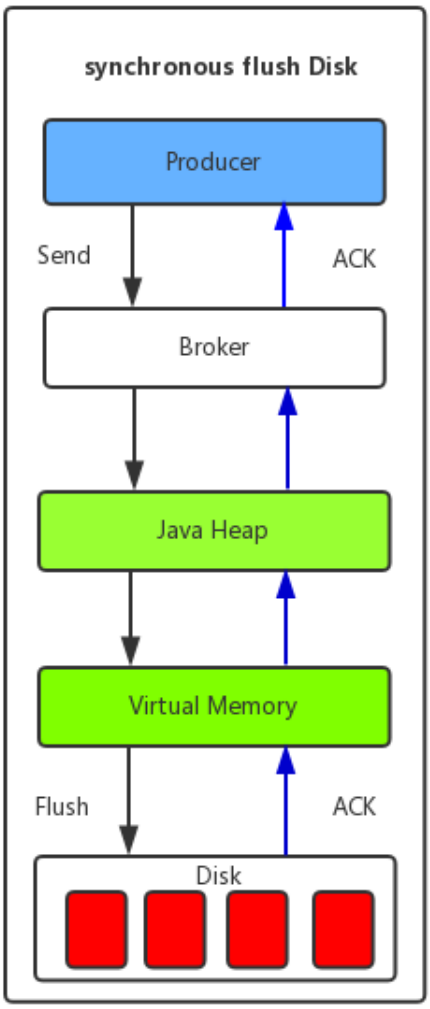
```
#最后一个索引消息在commitlog的偏移量 8bytes
private AtomicLong endPhyOffset = new AtomicLong(0);
#构建索引占用的槽位数 4bytes
private AtomicInteger hashSlotCount = new AtomicInteger(0);
#构建的索引个数，4bytes
private AtomicInteger indexCount = new AtomicInteger(1);
```

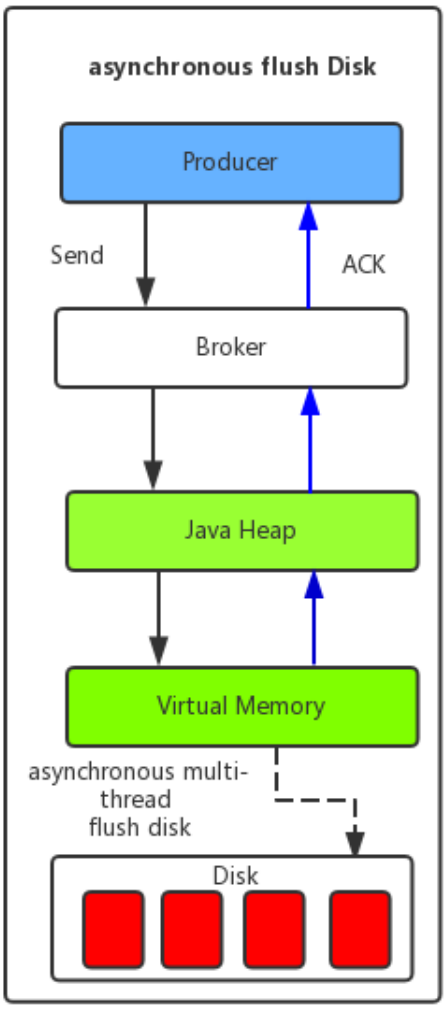
索引头文件存储结构：



在上面的RocketMQ的消息存储整体架构图中可以看出，RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。

消息刷盘

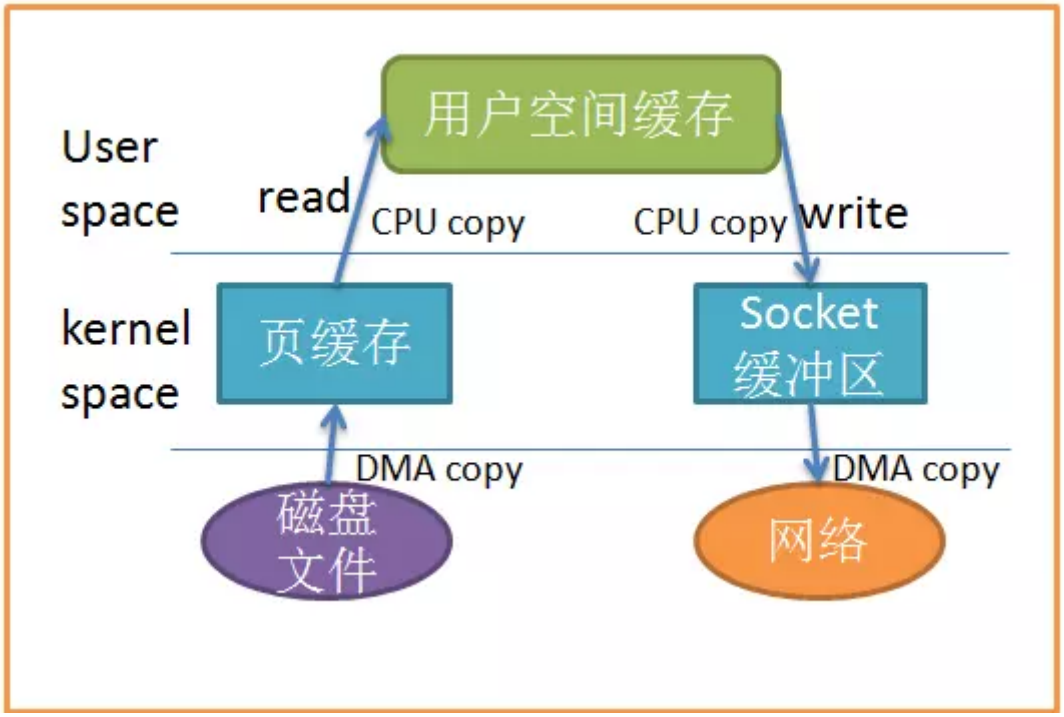




(1) 同步刷盘：如上图所示，只有在消息真正持久化至磁盘后RocketMQ的Broker端才会真正返回给Producer端一个成功的ACK响应。同步刷盘对MQ消息可靠性来说是一种不错的保障，但是性能上会有较大影响，一般适用于金融业务应用该模式较多。
(2) 异步刷盘：能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了MQ的性能和吞吐量。

零拷贝刷盘

以文件下载为例，服务端的主要任务是：**将服务端主机磁盘中的文件不做修改地从已连接的socket发出去。操作系统底层I/O过程如下图所示：**



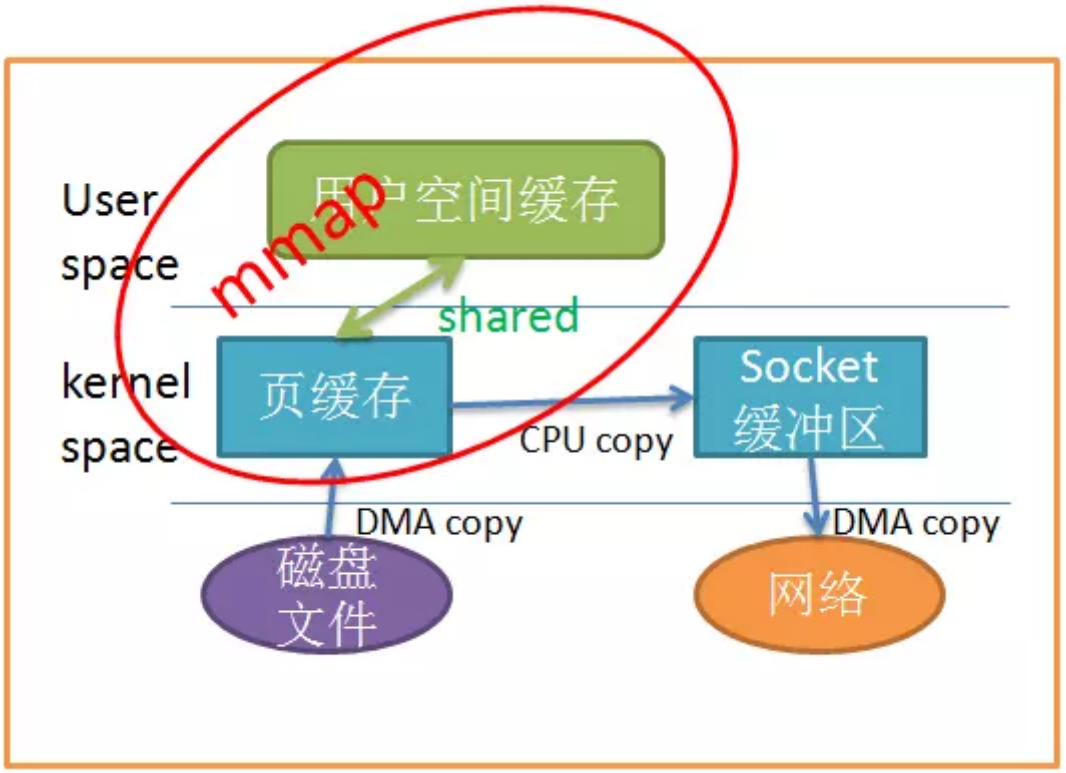
过程共产生了四次数据拷贝，在此过程中，我们没有对文件内容做任何修改，那么在内核空间和用户空间来回拷贝数据无疑就是一种浪费，而零拷贝主要就是为了解决这种低效性。

什么是零拷贝技术?

零拷贝主要的任务就是**避免**CPU将数据从一块存储拷贝到另外一块存储，主要就是利用各种零拷贝技术，避免让CPU做大量的数据拷贝任务，减少不必要的拷贝，或者让别的组件来做这一类简单的数据传输任务，让CPU解脱出来专注于别的任务。这样就可以让系统资源的利用更加有效。

原理是磁盘上的数据会通过DMA被拷贝的内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，这样就不需要把内核缓冲区的内容往用户空间拷贝。应用程序再调用write(),操作系统直接将内核缓冲区的内容拷贝到socket缓冲区中，这一切都发生在内核态，最后，socket缓冲区再把数据发到网卡去。

原理如下:


User space
用户空间缓存
mmap
shared
kernel space
页缓存
Socket 缓冲区
CPU copy
DMA copy
DMA copy
磁盘文件
网络