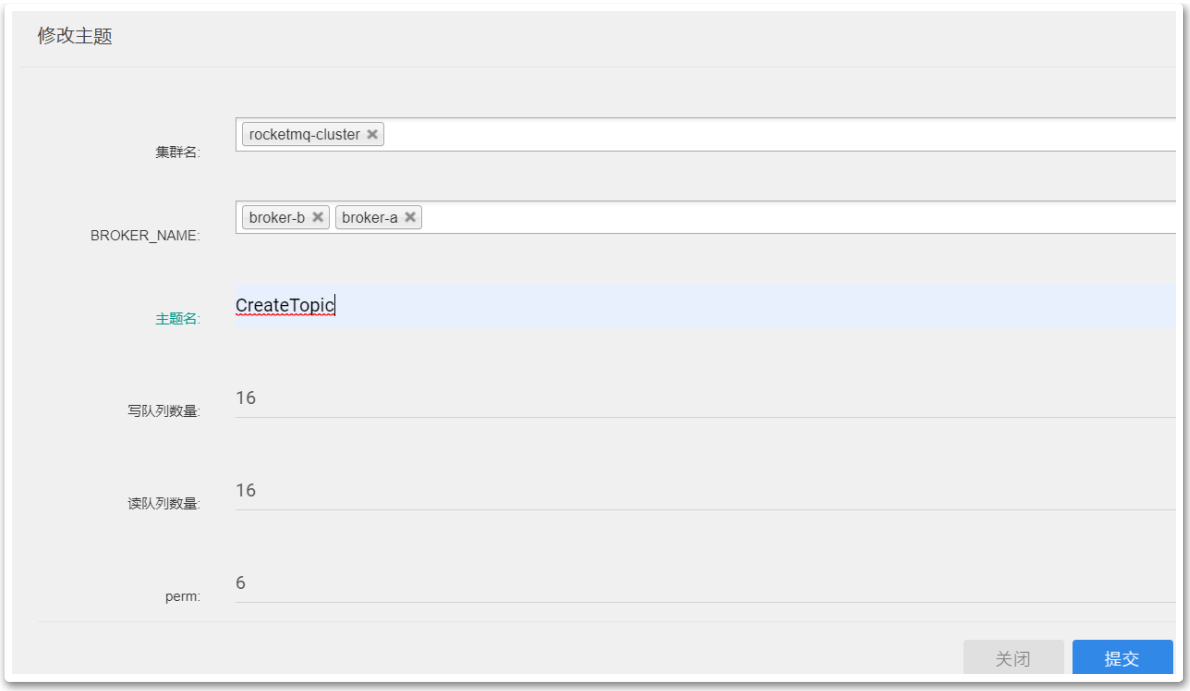
图灵：楼兰

你的神秘技术宝藏

前面的部分我们都是为了快速的体验RocketMQ的搭建和使用。这一部分，我们主要分享一些RocketMQ比较有特色的设计点。这些设计点正是RocketMQ高效性能的关键所在。同时，这些关键的设计点也可以带到后续的源码阅读章节中去验证，为枯燥的源码阅读找到一些目标。

一、读队列与写队列

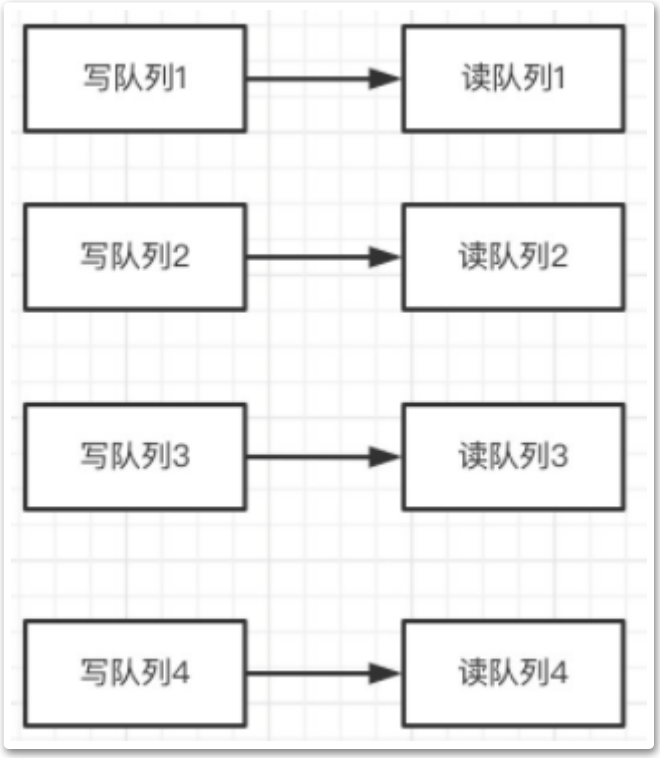
在RocketMQ的管理控制台创建Topic时，可以看到要单独设置读队列和写队列。通常在运行时，都需要设置读队列=写队列。



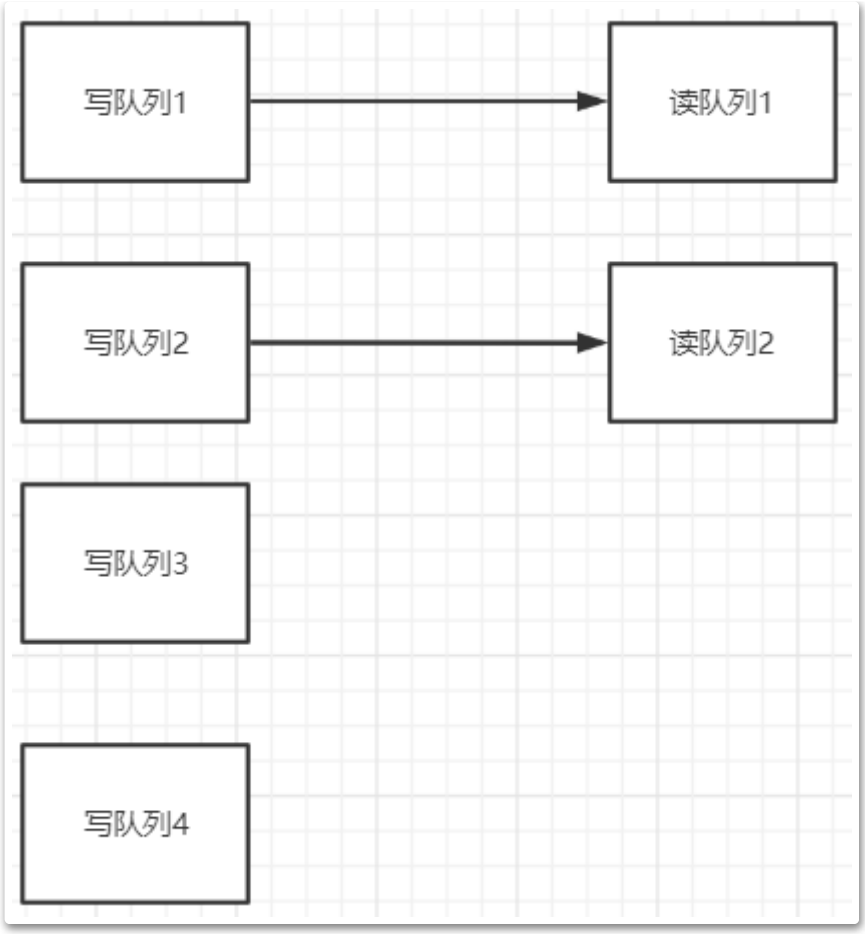
> perm字段表示Topic的权限。有三个可选项。 2：禁写禁订阅，4：可订阅，不能写，6：可写可订阅

这其中，写队列会真实的创建对应的存储文件，负责消息写入。而读队列会记录Consumer的Offset，负责消息读取。这其实是一种读写分离的思想。RocketMQ在最MessageQueue的路由策略时，就可以通过指向不同的队列来实现读写分离。

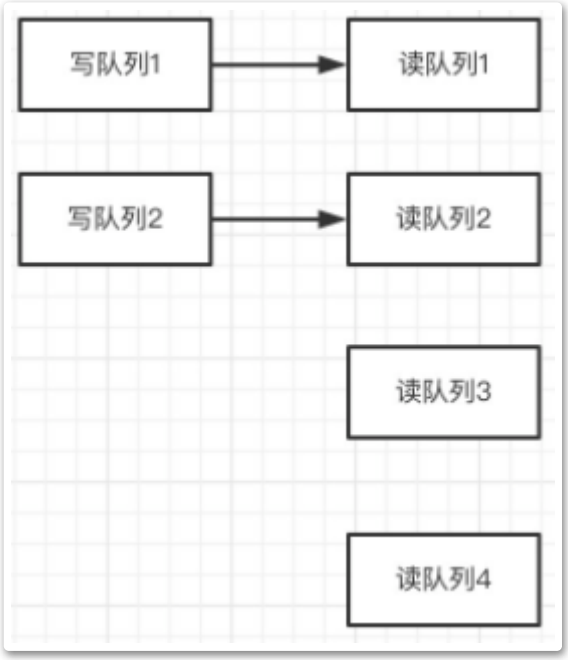
在往写队列里写Message时，会同步写入到一个对应的读队列中。



这时，如果写队列大于读队列，就会有一部分写队列无法写入到读队列中，这一部分的消息就无法被读取，就会造成消息丢失。--消息存入了，但是读不出来。



而如果反过来，写队列小于读队列，那就有一部分读队列里是没有消息写入的。如果有一个消费者被分配的是这些没有消息的读队列，那这些消费者就无法消费消息，造成消费者空转，极大的浪费性能。



从这里可以看到，写队列>读队列，会造成消息丢失，写队列<读队列，又会造成消费者空转。所以，在使用时，都是要求 写队列=读队列。

只有一种情况下可以考虑将读写队列设置为不一致，就是要对Topic的MessageQueue进行缩减的时候。例如原来四个队列，现在要缩减成两个队列。如果立即缩减读写队列，那么被缩减的MessageQueue上没有被消费的消息，就会丢失。这时，可以先缩减写队列，待空出来的读队列上的消息都被消费完了之后，再来缩减读队列，这样就可以比较平稳的实现队列缩减了。

二、消息持久化 -- 重点

RocketMQ消息直接采用磁盘文件保存消息，默认路径在${user_home}/store目录。这些存储目录可以在broker.conf中自行指定。

```
[oper@worker2 store]$ pwd
/app/rocketmq/store
[oper@worker2 store]$ ll
总用量 12
-rw-rw-r--.  1 oper oper 4096 4月  19 2021 checkpoint
drwxrwxr-x.  2 oper oper   34 10月 20 2020 commitlog
drwxrwxr-x.  2 oper oper  280 4月  19 2021 config
drwxrwxr-x. 13 oper oper 4096 12月  2 2020 consumequeue
drwxrwxr-x.  2 oper oper   31 12月 23 2020 index
-rw-rw-r--.  1 oper oper    4 4月  19 2021 lock
```

存储文件主要分为三个部分：

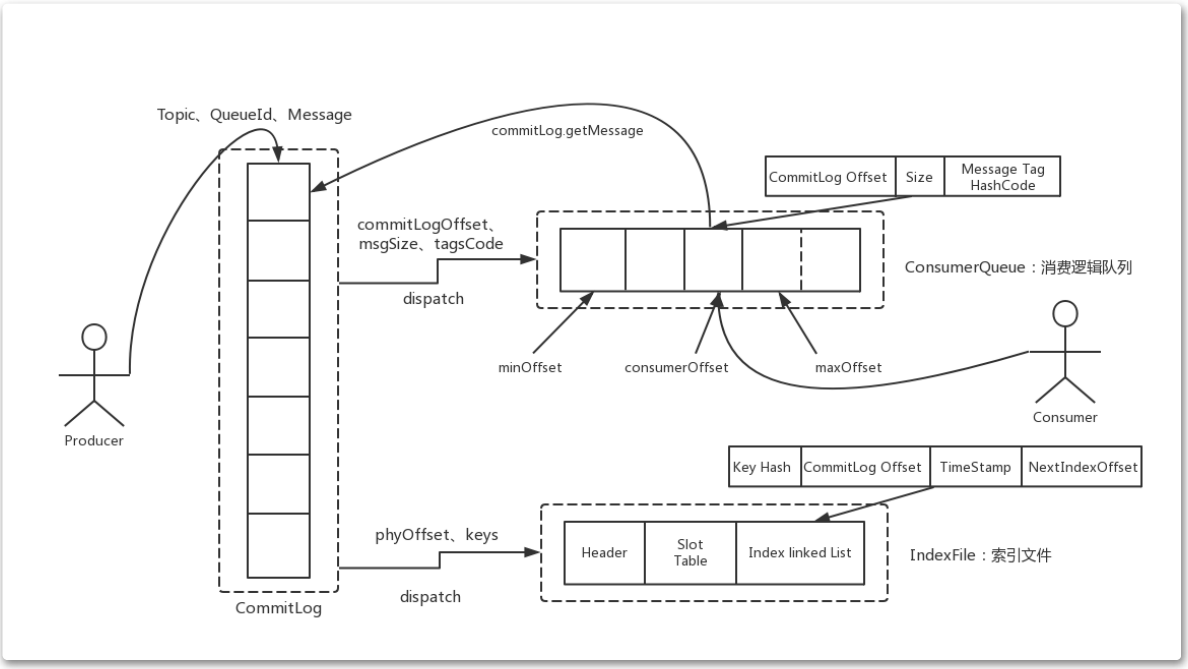
- CommitLog：存储消息的元数据。所有消息都会顺序存入到CommitLog文件当中。CommitLog由多个文件组成，每个文件固定大小1G。以第一条消息的偏移量为文件名。
- ConsumerQueue：存储消息在CommitLog的索引。一个MessageQueue一个文件，记录当前MessageQueue被哪些消费者组消费到了哪一条CommitLog。
- IndexFile：为了消息查询提供了一种通过key或时间区间来查询消息的方法，这种通过IndexFile来查找消息的方法不影响发送与消费消息的主流程

另外，还有几个辅助的存储文件：

- checkpoint：数据存盘检查点。里面主要记录commitlog文件、ConsumeQueue文件以及IndexFile文件最后一次刷盘的时间戳。
- config/*.json：这些文件是将RocketMQ的一些关键配置信息进行存盘保存。例如Topic配置、消费者组配置、消费者组消息偏移量Offset 等等一些信息。
- abort：这个文件是RocketMQ用来判断程序是否正常关闭的一个标识文件。正常情况下，会在启动时创建，而关闭服务时删除。但是如果遇到一些服务器宕机，或者kill -9这样一些非正常关闭服务的情况，这个abort文件就不会删除，因

此RocketMQ就可以判断上一次服务是非正常关闭的，后续就会做一些数据恢复的操作。

整体的消息存储结构如下图：



1、CommitLog文件存储所有消息实体。所有生产者发过来的消息，都会无差别的依次存储到Commitlog文件当中。这样的好处是可以减少查找目标文件的时间，让消息以最快的速度落盘。对比Kafka存文件时，需要寻找消息所属的Partition文件，再完成写入，当Topic比较多时，这样的Partition寻址就会浪费比较多的时间，所以Kafka不太适合多Topic的场景。而RocketMQ的这种快速落盘的方式在多Topic场景下，优势就比较明显。

文件结构：CommitLog的文件大小是固定的，但是其中存储的每个消息单元长度是不固定的，具体格式可以参考org.apache.rocketmq.store.CommitLog

```java
protected static int calMsgLength(int sysFlag, int bodyLength, int topicLength, int propertiesLength) {
    int bornhostLength = (sysFlag & MessageSysFlag.BORNHOST_V6_FLAG) == 0 ? 8 : 20;
    int storehostAddressLength = (sysFlag & MessageSysFlag.STOREHOSTADDRESS_V6_FLAG) == 0 ? 8 : 20;
    final int msgLen = 4 //TOTALSIZE
        + 4 //MAGICCODE
        + 4 //BODYCRC
        + 4 //QUEUEID
        + 4 //FLAG
        + 8 //QUEUEOFFSET
        + 8 //PHYSICALOFFSET
        + 4 //SYSFLAG
        + 8 //BORNTIMESTAMP
        + bornhostLength //BORNHOST
        + 8 //STORETIMESTAMP
        + storehostAddressLength //STOREHOSTADDRESS
        + 4 //RECONSUMETIMES
        + 8 //Prepared Transaction Offset
        + 4 + (bodyLength > 0 ? bodyLength : 0) //BODY
        + 1 + topicLength //TOPIC
        + 2 + (propertiesLength > 0 ? propertiesLength : 0) //propertiesLength
        + 0;
    return msgLen;
}
```

正因为消息的记录大小不固定，所以RocketMQ在每次存CommitLog文件时，都会去检查当前CommitLog文件空间是否足够，如果不够的话，就重新创建一个CommitLog文件。文件名为当前消息的偏移量。**在后面的源码中去验证。**

2、ConsumeQueue文件主要是加速消费者的消息索引。他的每个文件夹对应RocketMQ中的一个MessageQueue，文件夹下的文件记录了每个MessageQueue中的消息在CommitLog文件当中的偏移量。这样，消费者通过ComsumeQueue文件，就可以快速找到CommitLog文件中感兴趣的消息记录。而消费者在ConsumeQueue文件当中的消费进度，会保存在config/consumerOffset.json文件当中。

文件结构：每个ConsumeQueue文件固定由30万个固定大小20byte的数据块组成，数据块的内容包括：msgPhyOffset(8byte，消息在文件中的起始位置)+msgSize(4byte，消息在文件中占用的长度)+msgTagCode(8byte，消息的tag的Hash值)。

在ConsumeQueue.java当中有一个常量CQ_STORE_UNIT_SIZE=20，这个常量就表示一个数据块的大小。

3、IndexFile文件主要是辅助消息检索。消费者进行消息消费时，通过ConsumeQueue文件就足够完成消息检索了，但是如果要按照MeessageId或者MessageKey来检索文件，比如RocketMQ管理控制台的消息轨迹功能，ConsumeQueue文件就不够用了。IndexFile文件就是用来辅助这类消息检索的。他的文件名比较特殊，不是以消息偏移量命名，而是用的时间命名。但是其实，他也是一个固定大小的文件。

文件结构：他的文件结构由 indexHeader(固定40byte)+ slot(固定500W个，每个固定20byte) + index(最多500W*4个，每个固定20byte) 三个部分组成。

> indexFile的详细结构有大厂之前面试过，可以参考一下我的博文： https://blog.csdn.net/roykingw/article/details/120086520

这些文件的结构可以尝试到源码当中去验证。这里重点思考为什么这样设计，以及这样设计如何支撑上层的功能。

三、过期文件删除

消息既然要持久化，就必须有对应的删除机制。RocketMQ内置了一套过期文件的删除机制。

首先：如何判断过期文件：

RocketMQ中，CommitLog文件和ConsumeQueue文件都是以偏移量命名，对于非当前写的文件，如果超过了一定的保留时间，那么这些文件都会被认为是过期文件，随时可以删除。这个保留时间就是在broker.conf中配置的fileReservedTime属性。

注意，RocketMQ判断文件是否过期的唯一标准就是非当前写文件的保留时间，而并不关心文件当中的消息是否被消费过。所以，RocketMQ的消息堆积也是有时间限度的。

然后：何时删除过期文件：

RocketMQ内部有一个定时任务，对文件进行扫描，并且触发文件删除的操作。用户可以指定文件删除操作的执行时间。在broker.conf中deleteWhen属性指定。默认是凌晨四点。

另外，RocketMQ还会检查服务器的磁盘空间是否足够，如果磁盘空间的使用率达到一定的阈值，也会触发过期文件删除。所以RocketMQ官方就特别建议，broker的磁盘空间不要少于4G。

四、高效文件写

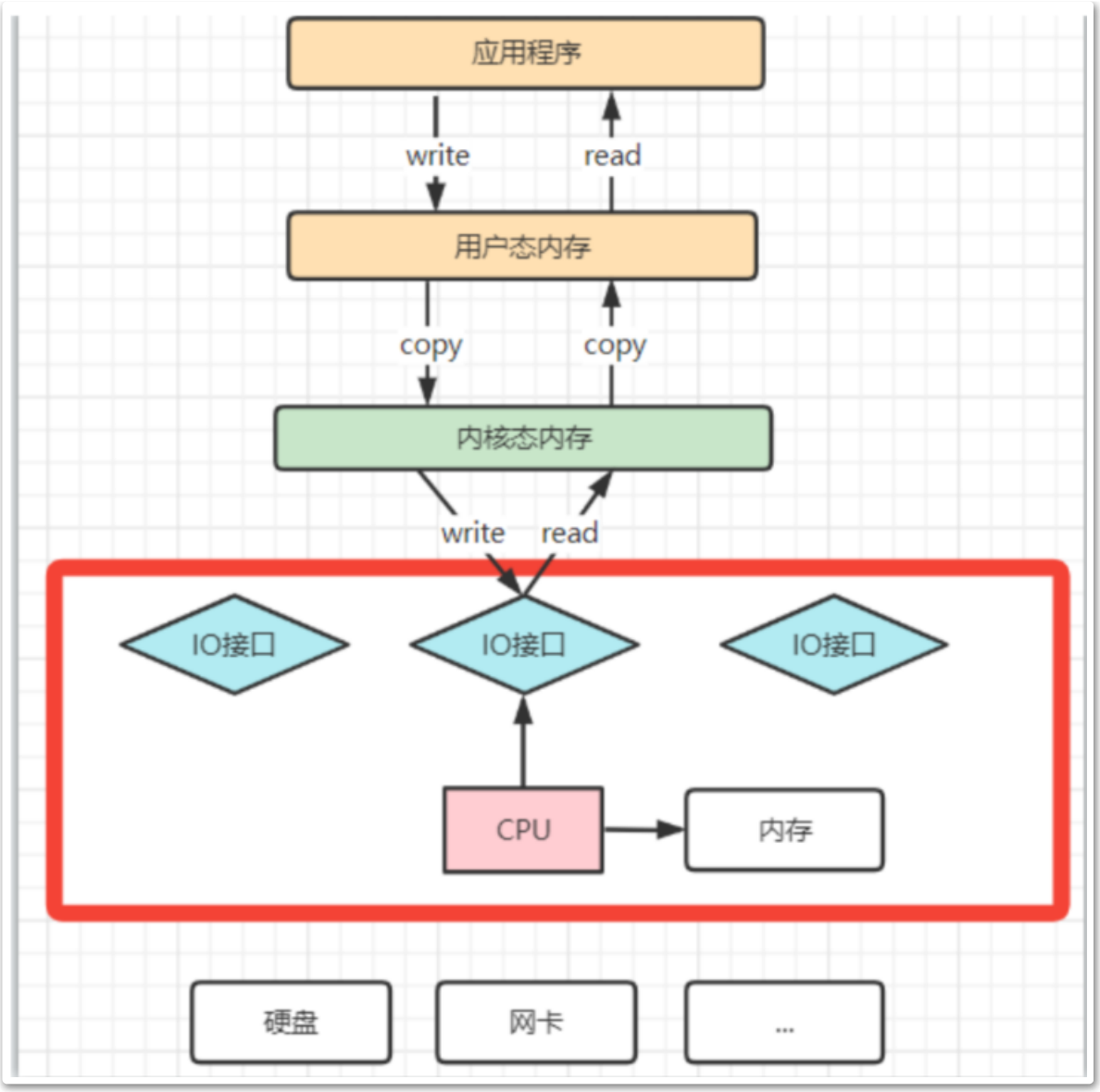
RocketMQ采用了类似于Kafka的文件存储机制，但是文件存储是一个比较重的操作，需要有非常多的设计才能保证频繁的文件读写场景下的高性能。

4.1 零拷贝技术加速文件读写

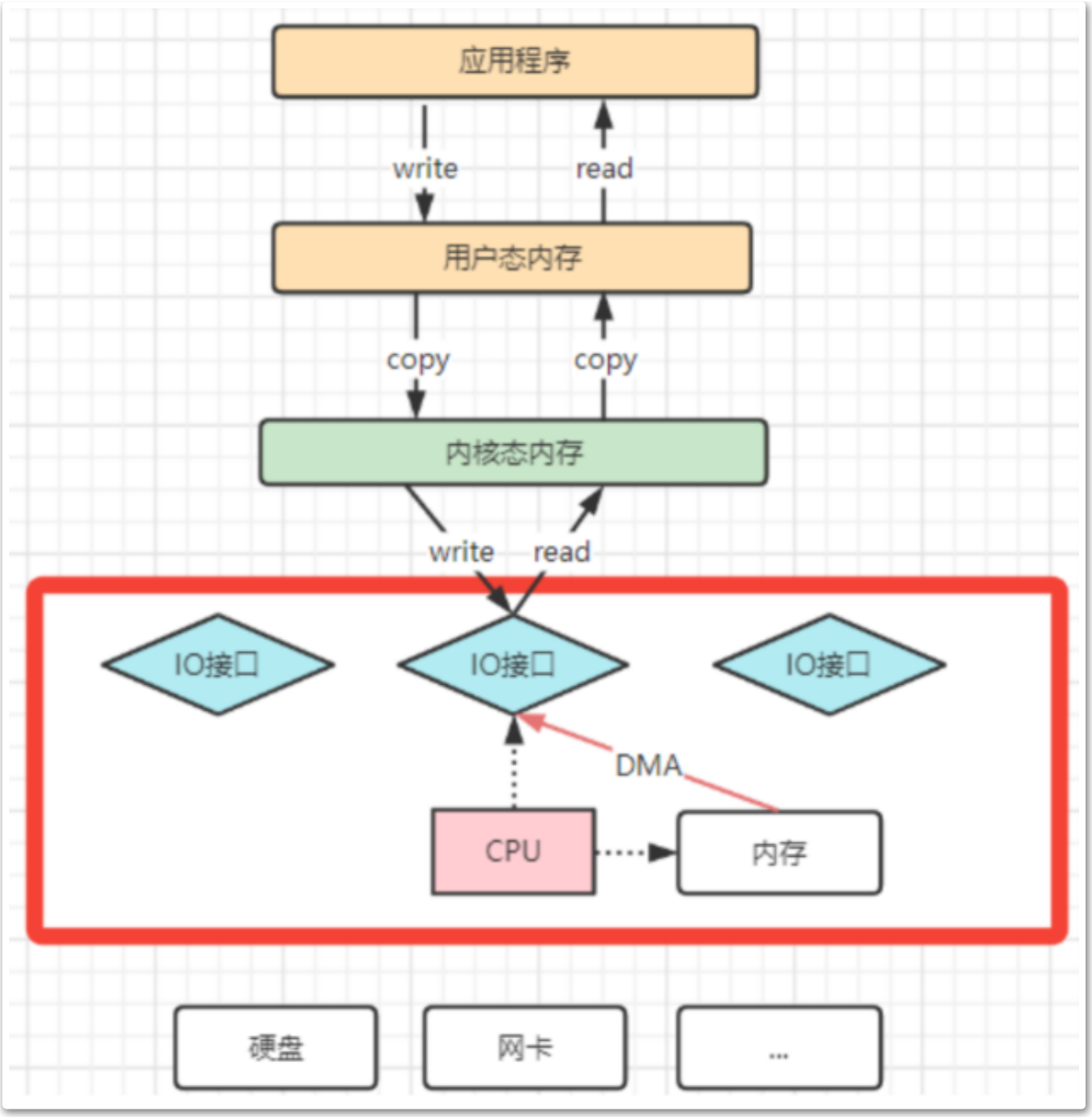
零拷贝(zero-copy)是操作系统层面提供的一种加速文件读写的操作机制，非常多的开源软件都在大量使用零拷贝，来提升IO操作的性能。对于Java应用层，对应着mmap和sendFile两种方式。接下来，咱们深入操作系统来详细理解一下零拷贝。

1：理解CPU拷贝和DMA拷贝

我们知道，操作系统对于内存空间，是分为用户态和内核态的。用户态的应用程序无法直接操作硬件，需要通过内核空间进行操作转换，才能真正操作硬件。这其实是为了保护操作系统的安全。正因为如此，应用程序需要与网卡、磁盘等硬件进行数据交互时，就需要在用户态和内核态之间来回的复制数据。而这些操作，原本都是需要由CPU来进行任务的分配、调度等管理步骤的，早先这些IO接口都是由CPU独立负责，所以当发生大规模的数据读写操作时，CPU的占用率会非常高。



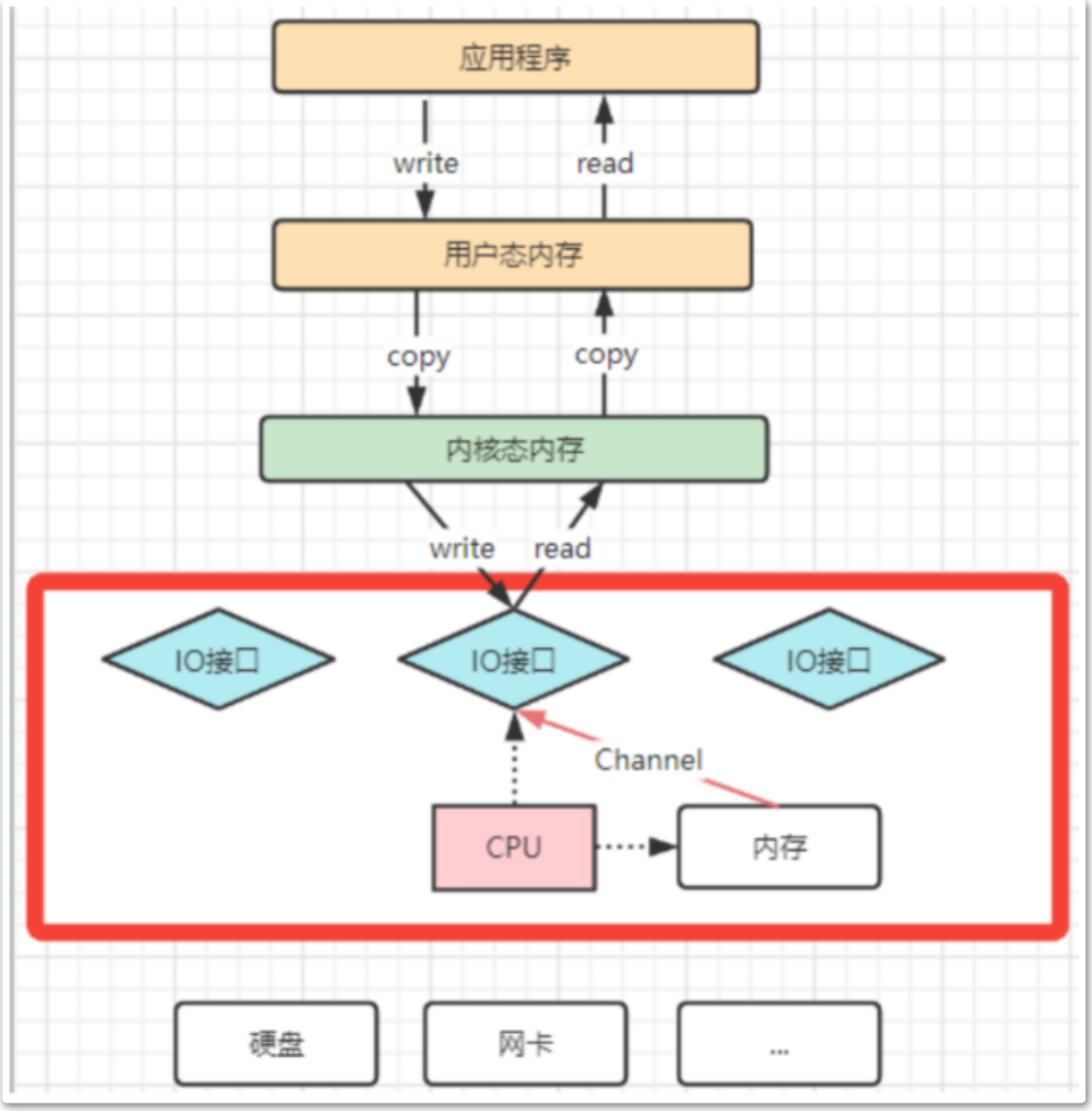
之后，操作系统为了避免CPU完全被各种IO调用给占用，引入了DMA(直接存储器存储)。由DMA来负责这些频繁的IO操作。DMA是一套独立的指令集，不会占用CPU的计算资源。这样，CPU就不需要参与具体的数据复制的工作，只需要管理DMA的权限即可。



DMA拷贝极大的释放了CPU的性能，因此他的拷贝速度会比CPU拷贝要快很多。但是，其实DMA拷贝本身，也在不断优化。

引入DMA拷贝之后，在读写请求的过程中，CPU不再需要参与具体的工作，DMA可以独立完成数据在系统内部的复制。但是，数据复制过程中，依然需要借助数据总进线。当系统内的IO操作过多时，还是会占用过多的数据总线，造成总线冲突，最终还是会影响数据读写性能。

为了避免DMA总线冲突对性能的影响，后来又引入了Channel通道的方式。Channel，是一个完全独立的处理器，专门负责IO操作。既然是处理器，Channel就有自己的IO指令，与CPU无关，他也更适合大型的IO操作，性能更高。



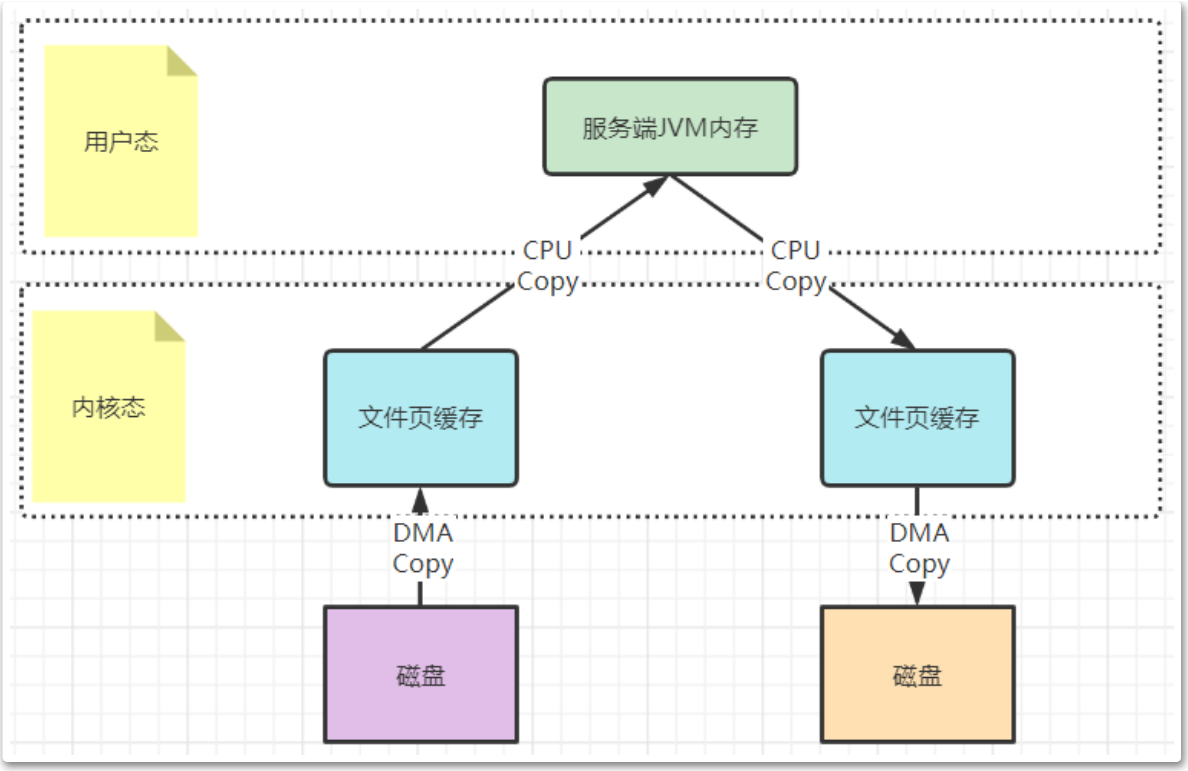
这也解释了，为什么Java应用层与零拷贝相关的操作都是通过Channel的子类实现的。这其实是借鉴了操作系统中的概念。

而所谓的零拷贝技术，其实并不是不拷贝，而是要尽量减少CPU拷贝。

2：再来理解下mmap文件映射机制是怎么回事。

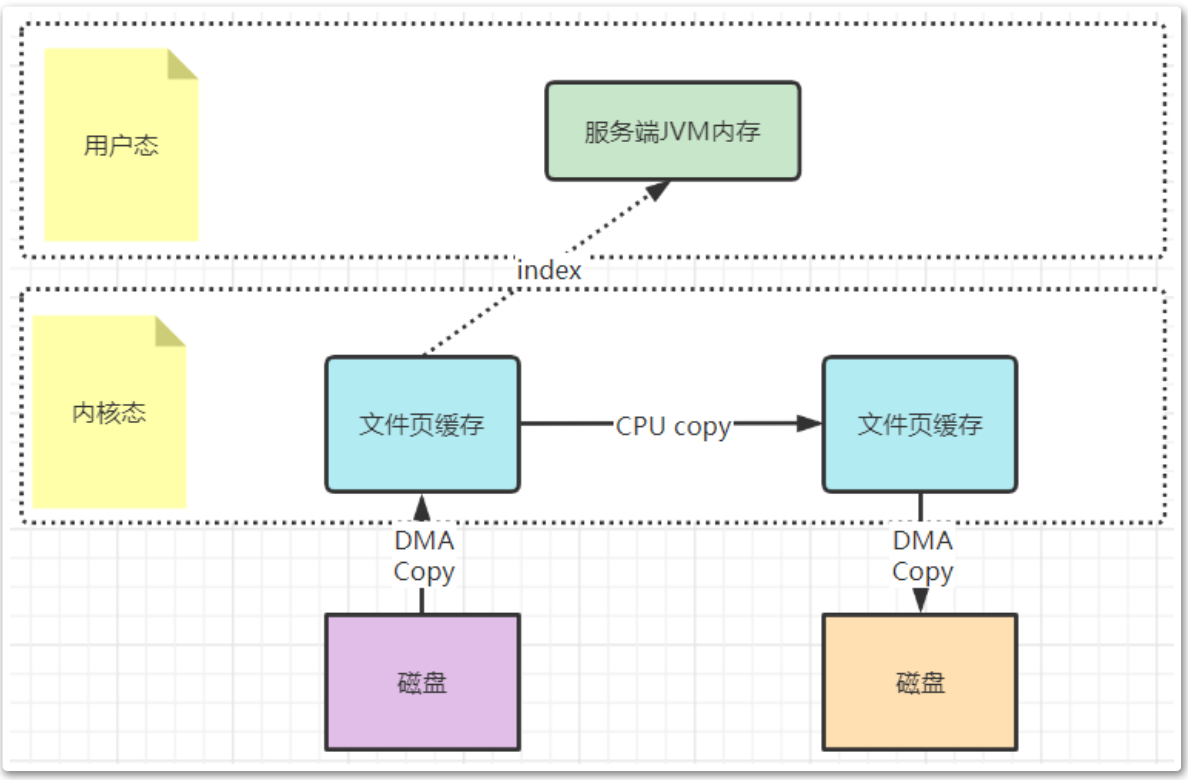
mmap机制的具体实现参见配套示例代码。主要是通过java.nio.channels.FileChannel的map方法完成映射。

以一次文件的读写操作为例，应用程序对磁盘文件的读与写，都需要经过内核态与用户态之间的状态切换，每次状态切换的过程中，就需要有大量的数据复制。



在这个过程中，总共需要进行四次数据拷贝。而磁盘与内核态之间的数据拷贝，在操作系统层面已经由CPU拷贝优化成了DMA拷贝。而内核态与用户态之间的拷贝依然是CPU拷贝。所以，在这个场景下，零拷贝技术优化的重点，就是内核态与用户态之间的这两次拷贝。

而mmap文件映射的方式，就是在用户态不再保存文件的内容，而只保存文件的映射，包括文件的内存起始地址，文件大小等。真实的数据，也不需要在用户态留存，可以直接通过操作映射，在内核态完成数据复制。



这个拷贝过程都是在操作系统的系统调用层面完成的，在Java应用层，其实是无法直接观测到的，但是我们可以去JDK源码当中进行间接验证。在JDK的NIO包中，java.nio.HeapByteBuffer映射的就是JVM的一块堆内内存，在HeapByteBuffer中，会由一个byte数组来缓存数据内容，所有的读写操作也是先操作这个byte数组。这其实就是没有使用零拷贝的普通文件读写机制。

```
HeapByteBuffer(int cap, int lim) {            // package-private
        super(-1, 0, lim, cap, new byte[cap], 0);
        /*
        hb = new byte[cap];
        offset = 0;
        */
    }
```

而NIO把包中的另一个实现类java.nio.DirectByteBuffer则映射的是一块堆外内存。在DirectByteBuffer中，并没有一个数据结构来保存数据内容，只保存了一个内存地址。所有对数据的读写操作，都通过unsafe魔法类直接交由内核完成，这其实就是mmap的读写机制。

mmap文件映射机制，其实并不神秘，我们启动任何一个Java程序时，其实都大量用到了mmap文件映射。例如，我们可以在Linux机器上，运行一下下面这个最简单不过的应用程序：

```
import java.util.Scanner;
public class BlockDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        final String s = scanner.nextLine();
        System.out.println(s);
    }
}
```

通过Java指令运行起来后，可以用jps查看到运行的进程ID。然后，就可以使用lsof -p {PID}的方式查看文件的映射情况。

```
[root@192-168-65-232 ~]# lsof -p 4870
lsof: WARNING: can't stat() fuse.gvfsd-fuse file system /run/user/988/gvfs
      Output information may be incomplete.
COMMAND  PID USER   FD   TYPE DEVICE  SIZE/OFF     NODE NAME
java    4870 root  cwd    DIR  253,0        51 75499781 /root/code
java    4870 root  rtd    DIR  253,0       268       64 /
java    4870 root  txt    REG  253,0      7734 27895626 /usr/java/jdk1.8.0_121/jre/bin/java
java    4870 root  mem    REG  253,0 106176928   171742 /usr/lib/locale/locale-archive
java    4870 root  mem    REG  253,0   3866653  4935851 /usr/java/jdk1.8.0_121/jre/lib/ext/cldrdata.jar
java    4870 root  mem    REG  253,0  65857716  4935826 /usr/java/jdk1.8.0_121/jre/lib/rt.jar
java    4870 root  mem    REG  253,0    124327 11532876 /usr/java/jdk1.8.0_121/jre/lib/amd64/libzip.so
java    4870 root  mem    REG  253,0     61560   185473 /usr/lib64/libnss_files-2.17.so
java    4870 root  mem    REG  253,0    225914  9971241 /usr/java/jdk1.8.0_121/jre/lib/amd64/libjava.so
java    4870 root  mem    REG  253,0     65672 11532875 /usr/java/jdk1.8.0_121/jre/lib/amd64/libverify.so
java    4870 root  mem    REG  253,0     43712   267211 /usr/lib64/librt-2.17.so
java    4870 root  mem    REG  253,0   1136944   171762 /usr/lib64/libm-2.17.so
java    4870 root  mem    REG  253,0  16989733 27895642 /usr/java/jdk1.8.0_121/jre/lib/amd64/server/libjvm.so
java    4870 root  mem    REG  253,0   2156344   171751 /usr/lib64/libc-2.17.so
java    4870 root  mem    REG  253,0     19248   171759 /usr/lib64/libdl-2.17.so
java    4870 root  mem    REG  253,0    102352 19961330 /usr/java/jdk1.8.0_121/jre/lib/amd64/jli/libjli.so
java    4870 root  mem    REG  253,0    142144   267207 /usr/lib64/libpthread-2.17.so
java    4870 root  mem    REG  253,0    163312   171744 /usr/lib64/ld-2.17.so
java    4870 root  mem    REG  253,0   2246277  4935822 /usr/java/jdk1.8.0_121/jre/lib/ext/localedata.jar
java    4870 root  mem    REG  253,0     32768 19361196 /tmp/hsperfdata_root/4870
java    4870 root    0u   CHR  136,1       0t0        4 /dev/pts/1
java    4870 root    1u   CHR  136,1       0t0        4 /dev/pts/1
java    4870 root    2u   CHR  136,1       0t0        4 /dev/pts/1
java    4870 root    3r   REG  253,0  65857716  4935826 /usr/java/jdk1.8.0_121/jre/lib/rt.jar
java    4870 root    4r   REG  253,0   3866653  4935851 /usr/java/jdk1.8.0_121/jre/lib/ext/cldrdata.jar
java    4870 root    5r   REG  253,0   2246277  4935822 /usr/java/jdk1.8.0_121/jre/lib/ext/localedata.jar
```

这里面看到的mem类型的FD其实就是文件映射。

> cwd 表示程序的工作目录。rtd 表示用户的根目录。 txt表示运行程序的指令。下面的1u表示Java应用的标准输出，2u表示Java应用的标准错误输出，默认的/dev/pts/1是linux当中的伪终端。通常服务器上会写 java xxx 1>text.txt 2>&1 这样的脚本，就是指定这里的1u，2u。

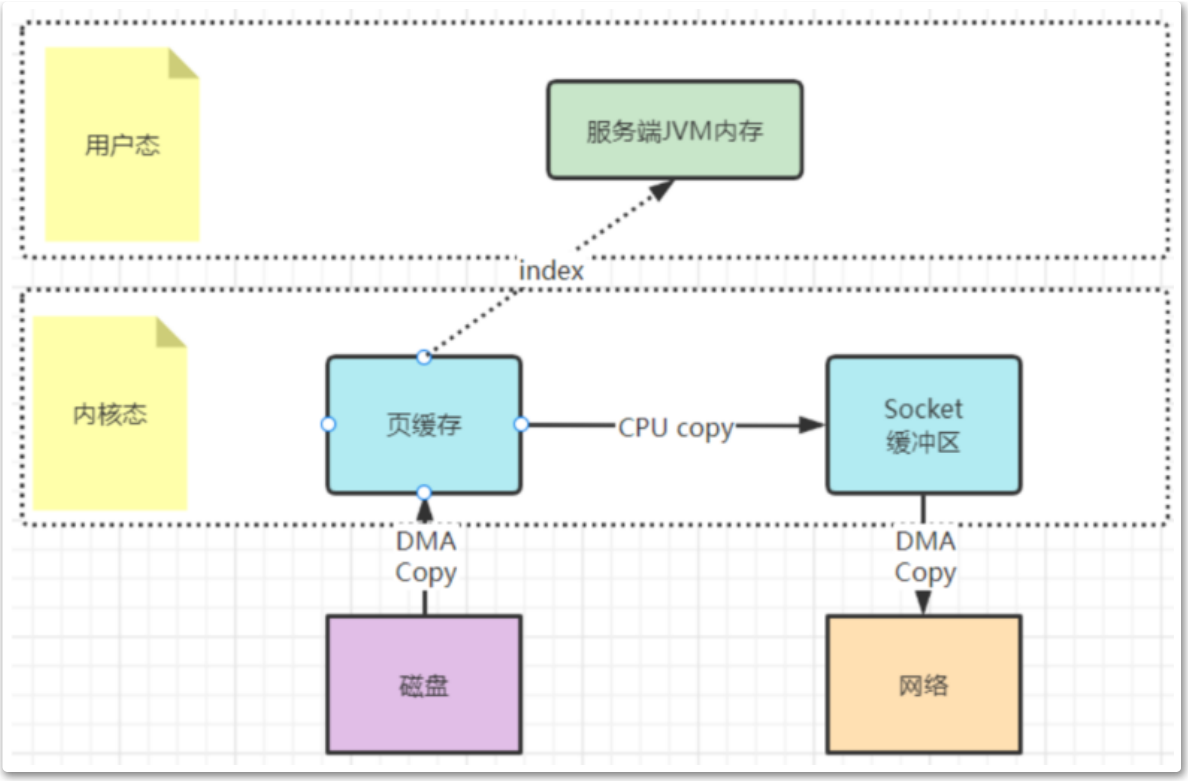
最后，这种mmap的映射机制由于还是需要用户态保存文件的映射信息，数据复制的过程也需要用户态的参与，这其中的变数还是非常多的。所以，**mmap机制适合操作小文件**，如果文件太大，映射信息也会过大，容易造成很多问题。通常mmap机制建议的映射文件大小不要超过2G 。而RocketMQ做大的CommitLog文件保持在1G固定大小，也是为了方便文件映射。

3：梳理下sendFile机制是怎么运行的。

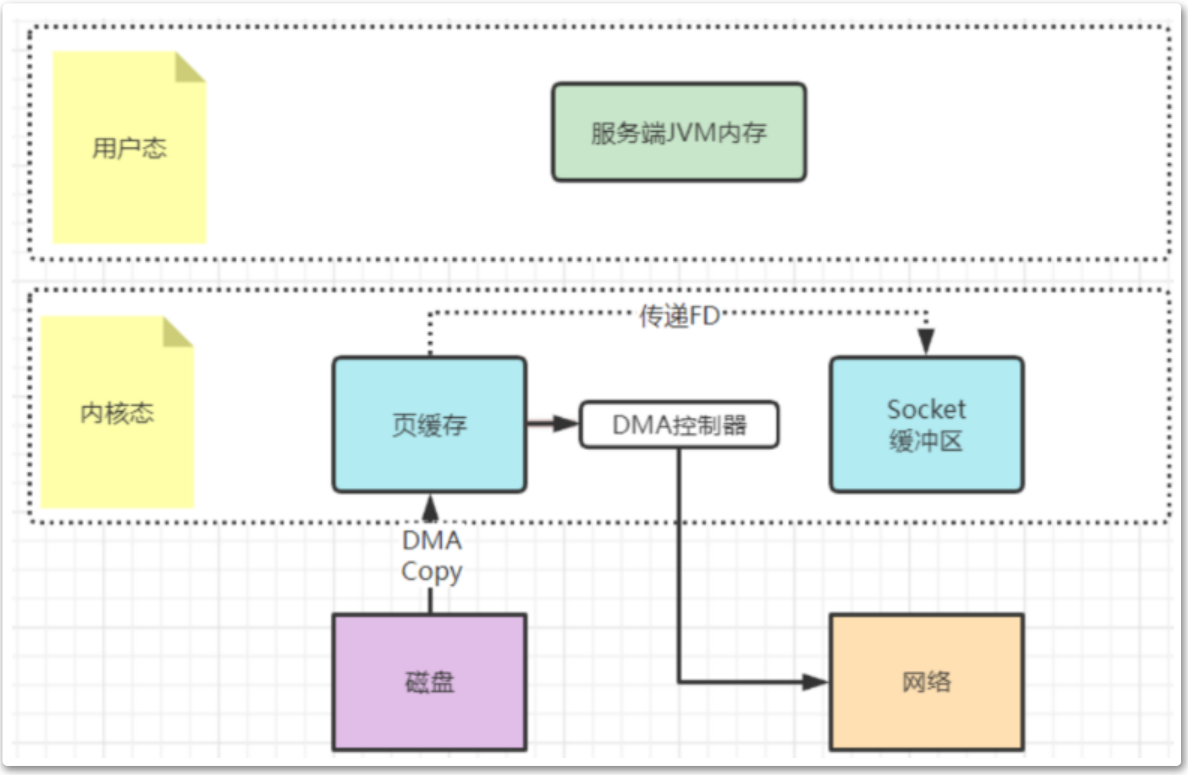
sendFile机制的具体实现参见配套示例代码。主要是通过java.nio.channels.FileChannel的transferTo方法完成。

```
1  sourceReadChannel.transferTo(0,sourceFile.length(),targetWriteChannel);
```

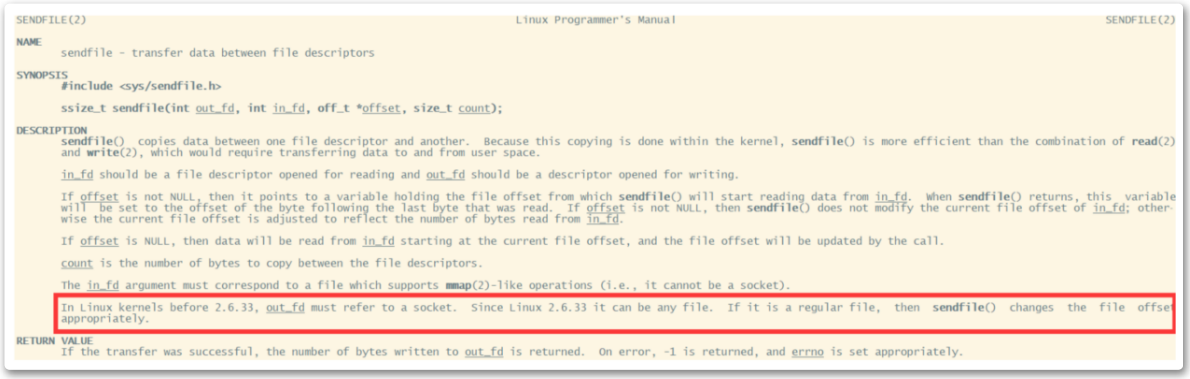
还记得Kafka当中是如何使用零拷贝的吗？你应该看到过这样的例子，就是Kafka将文件从磁盘复制到网卡时，就大量的使用了零拷贝。百度去搜索一下零拷贝，铺天盖地的也都是拿这个场景在举例。



早期的sendfile实现机制其实还是依靠CPU进行页缓存与socket缓存区之间的数据拷贝。但是，在后期的不断改进过程中，sendfile优化了实现机制，在拷贝过程中，并不直接拷贝文件的内容，而是只拷贝一个带有文件位置和长度等信息的文件描述符FD，这样就大大减少了需要传递的数据。而真实的数据内容，会交由DMA控制器，从页缓存中打包异步发送到socket中。



为什么大家都喜欢用这个场景来举例呢？其实我们去看下Linux操作系统的man帮助手册就能看到一部分答案。使用指令man 2 sendfile就能看到Linux操作系统对于sendfile这个系统调用的手册。



```
SENDFILE(2)                                Linux Programmer's Manual                                SENDFILE(2)

NAME
       sendfile - transfer data between file descriptors

SYNOPSIS
       #include <sys/sendfile.h>

       ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);

DESCRIPTION
       sendfile()  copies data between one file descriptor and another.  Because this copying is done within the kernel, sendfile() is more efficient than the combination of read(2)
       and write(2), which would require transferring data to and from user space.

       in_fd should be a file descriptor opened for reading and out_fd should be a descriptor opened for writing.

       If offset is not NULL, then it points to a variable holding the file offset from which sendfile() will start reading data from in_fd.  When sendfile() returns, this  variable
       will  be set to the offset of the byte following the last byte that was read.  If offset is not NULL, then sendfile() does not modify the current file offset of in_fd; other‐
       wise the current file offset is adjusted to reflect the number of bytes read from in_fd.

       If offset is NULL, then data will be read from in_fd starting at the current file offset, and the file offset will be updated by the call.

       count is the number of bytes to copy between the file descriptors.

       The in_fd argument must correspond to a file which supports mmap(2)-like operations (i.e., it cannot be a socket).

       In Linux kernels before 2.6.33, out_fd must refer to a socket.  Since Linux 2.6.33 it can be any file.  If it is a regular file,  then  sendfile()  changes  the  file  offset
       appropriately.

RETURN VALUE
       If the transfer was successful, the number of bytes written to out_fd is returned.  On error, -1 is returned, and errno is set appropriately.
```

2.6.33版本以前的Linux内核中，out_fd只能是一个socket，所以网上铺天盖地的老资料都是拿网卡来举例。但是现在版本已经没有了这个限制。

最后，sendfile机制在内核态直接完成了数据的复制，不需要用户态的参与，所以这种机制的传输效率是非常稳定的。**sendfile机制非常适合大数据的复制转移。**

4.2 顺序写加速文件写入磁盘

通常应用程序往磁盘写文件时，由于磁盘空间不是连续的，会有很多碎片。所以我们去写一个文件时，也就无法把一个文件写在一块连续的磁盘空间中，而需要在磁盘多个扇区之间进行大量的随机写。这个过程中有大量的寻址操作，会严重影响写数据的性能。而顺序写机制是在磁盘中提前申请一块连续的磁盘空间，每次写数据时，就可以避免这些寻址操作，直接在之前写入的地址后面接着写就行。

Kafka官方详细分析过顺序写的性能提升问题。Kafka官方曾说明，顺序写的性能基本能够达到内存级别。而如果配备固态硬盘，顺序写的性能甚至有可能超过写内存。而RocketMQ很大程度上借鉴了Kafka的这种思想。

例如可以看下org.apache.rocketmq.store.CommitLog#DefaultAppendMessageCallback中的doAppend方法。在这个方法中，会以追加的方式将消息先写入到一个堆外内存byteBuffer中，然后再通过fileChannel写入到磁盘。

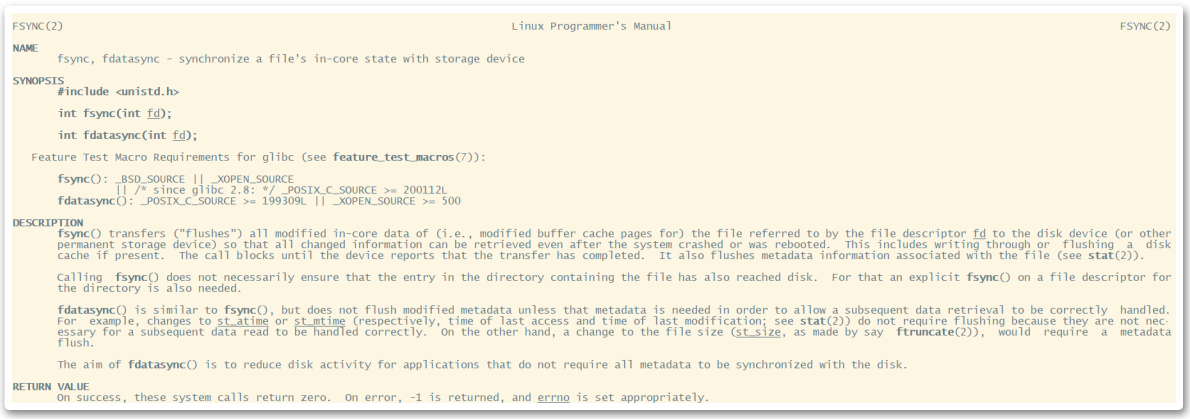
4.3 刷盘机制保证消息不丢失

在操作系统层面，当应用程序写入一个文件时，文件内容并不会直接写入到硬件当中，而是会先写入到操作系统中的一个缓存PageCache中。PageCache缓存以4K大小为单位，缓存文件的具体内容。这些写入到PageCache中的文件，在应用程序看来，是已经完全落盘保存好了的，可以正常修改、复制等等。但是，本质上，PageCache依然是内存状态，所以一断电就会丢失。因此，需要将内存状态的数据写入到磁盘当中，这样数据才能真正完成持久化，断电也不会丢失。这个过程就称为刷盘。

PageCache是源源不断产生的，而Linux操作系统显然不可能时时刻刻往硬盘写文件。所以，操作系统只会在某些特定的时刻将PageCache写入到磁盘。例如当我们正常关机时，就会完成PageCache刷盘。另外，在Linux中，对于有数据修改的PageCache，会标记为Dirty(脏页)状态。当Dirty Page的比例达到一定的阈值时，就会触发一次刷盘操作。例如在Linux操作系统中，可以通过/proc/meminfo文件查看到Page Cache的状态。

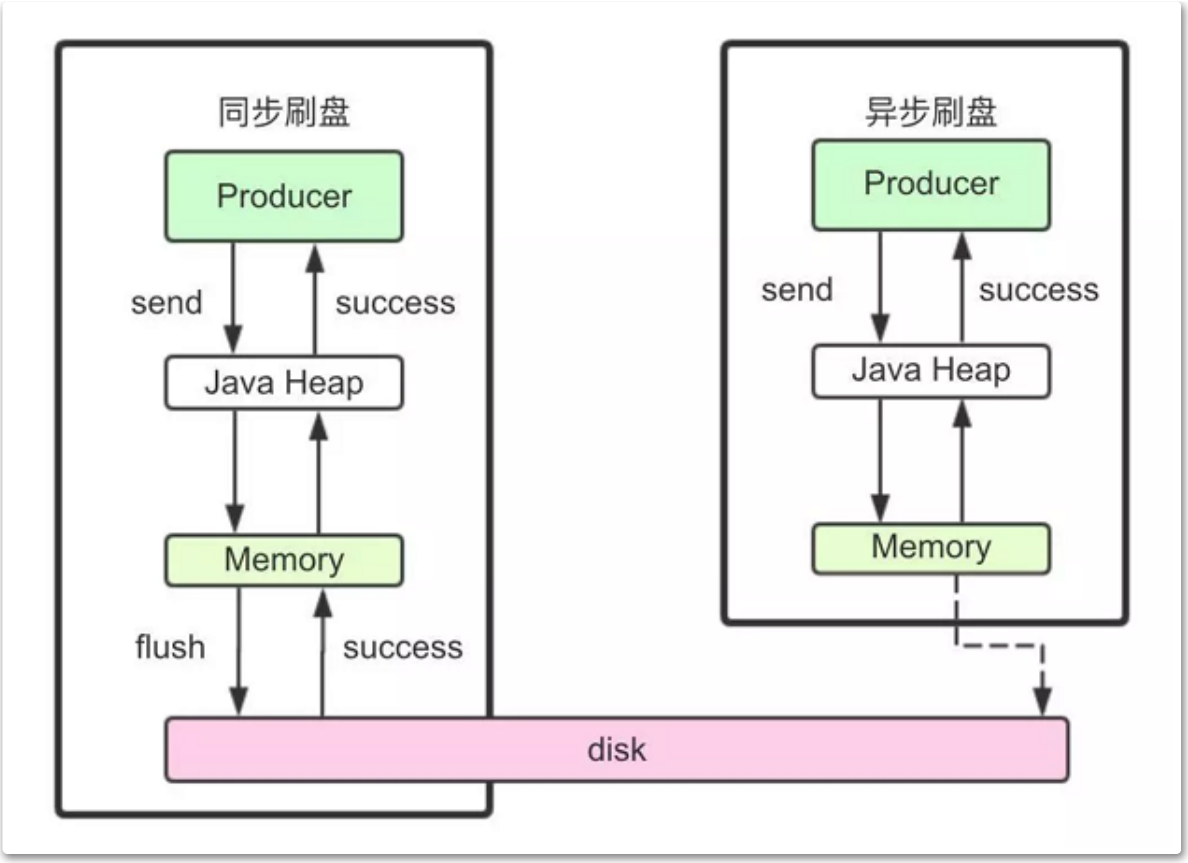
```
[root@192-168-65-174 ~]# cat /proc/meminfo
MemTotal:       16266172 kB
.....
Cached:           923724 kB
.....
Dirty:                32 kB
Writeback:             0 kB
.....
Mapped:           133032 kB
.....
```

但是，只要操作系统的刷盘操作不是时时刻刻执行的，那么对于用户态的应用程序来说，那就避免不了非正常宕机时的数据丢失问题。因此，操作系统也提供了一个系统调用，应用程序可以自行调用这个系统调用，完成PageCache的强制刷盘。在Linux中是fsync，同样我们可以用man 2 fsync 指令查看。



```
FSYNC(2)                                Linux Programmer's Manual                                FSYNC(2)

NAME
       fsync, fdatasync - synchronize a file's in-core state with storage device

SYNOPSIS
       #include <unistd.h>

       int fsync(int fd);

       int fdatasync(int fd);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       fsync(): _BSD_SOURCE || _XOPEN_SOURCE
                || /* since glibc 2.8: */ _POSIX_C_SOURCE >= 200112L
       fdatasync(): _POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500

DESCRIPTION
       fsync() transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor fd to the disk device (or other
       permanent storage device) so that all changed information can be retrieved even after the system crashed or was rebooted.  This includes writing through or  flushing  a  disk
       cache if present.  The call blocks until the device reports that the transfer has completed.  It also flushes metadata information associated with the file (see stat(2)).

       Calling  fsync() does not necessarily ensure that the entry in the directory containing the file has also reached disk.  For that an explicit fsync() on a file descriptor for
       the directory is also needed.

       fdatasync() is similar to fsync(), but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly  handled.
       For  example, changes to st_atime or st_mtime (respectively, time of last access and time of last modification; see stat(2)) do not require flushing because they are not nec-
       essary for a subsequent data read to be handled correctly.  On the other hand, a change to the file size (st_size, as made by say  ftruncate(2)),  would  require  a  metadata
       flush.

       The aim of fdatasync() is to reduce disk activity for applications that do not require all metadata to be synchronized with the disk.

RETURN VALUE
       On success, these system calls return zero.  On error, -1 is returned, and errno is set appropriately.
```

RocketMQ对于何时进行刷盘，也设计了两种刷盘机制，同步刷盘和异步刷盘。



- 同步刷盘：

  在返回写成功状态时，消息已经被写入磁盘。具体流程是，消息写入内存的PAGECACHE后，立刻通知刷盘线程刷盘， 然后等待刷盘完成，刷盘线程执行完成后唤醒等待的线程，返回消息写 成功的状态。

- 异步刷盘：

  在返回写成功状态时，消息可能只是被写入了内存的PAGECACHE，写操作的返回快，吞吐量大；当内存里的消息量积累到一定程度时，统一触发写磁盘动作，快速写入。

- 配置方式：

  刷盘方式是通过Broker配置文件里的flushDiskType 参数设置的，这个参数被配置成SYNC_FLUSH、ASYNC_FLUSH中的 一个。

  同步刷盘机制会更频繁的调用fsync，所以吞吐量相比异步刷盘会降低，但是数据的安全性会得到提高。

五、 消息主从复制

如果Broker以一个集群的方式部署，会有一个master节点和多个slave节点，消息需要从Master复制到Slave上。而消息复制的方式分为同步复制和异步复制。

- 同步复制：

同步复制是等Master和Slave都写入消息成功后才反馈给客户端写入成功的状态。

在同步复制下，如果Master节点故障，Slave上有全部的数据备份，这样容易恢复数据。但是同步复制会增大数据写入的延迟，降低系统的吞吐量。

- 异步复制：

异步复制是只要master写入消息成功，就反馈给客户端写入成功的状态。然后再异步的将消息复制给Slave节点。

在异步复制下，系统拥有较低的延迟和较高的吞吐量。但是如果master节点故障，而有些数据没有完成复制，就会造成数据丢失。

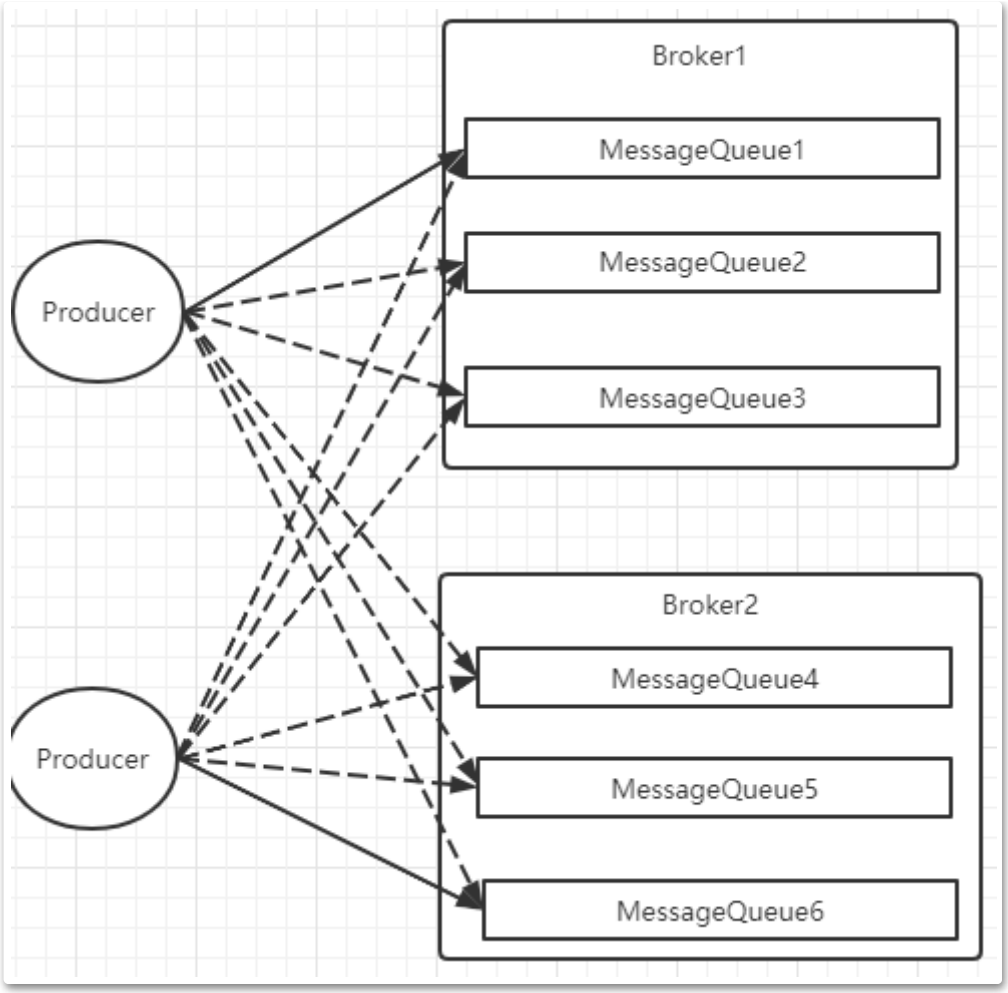
- 配置方式：

消息复制方式是通过Broker配置文件里的brokerRole参数进行设置的，这个参数可以被设置成ASYNC_MASTER、 SYNC_MASTER、SLAVE三个值中的一个。

六、负载均衡 --重点

6.1 Producer负载均衡

Producer发送消息时，默认会轮询目标Topic下的所有MessageQueue，并采用递增取模的方式往不同的MessageQueue上发送消息，以达到让消息平均落在不同的queue上的目的。而由于MessageQueue是分布在不同的Broker上的，所以消息也会发送到不同的broker上。



同时生产者在发送消息时，可以指定一个MessageQueueSelector。通过这个对象来将消息发送到自己指定的MessageQueue上。这样可以保证消息局部有序。

6.2 Consumer负载均衡

Consumer也是以MessageQueue为单位来进行负载均衡。分为集群模式和广播模式。

1、集群模式

在集群消费模式下，每条消息只需要投递到订阅这个topic的Consumer Group下的一个实例即可。RocketMQ采用主动拉取的方式拉取并消费消息，在拉取的时候需要明确指定拉取哪一条message queue。

而每当实例的数量有变更，都会触发一次所有实例的负载均衡，这时候会按照queue的数量和实例的数量平均分配queue给每个实例。

每次分配时，都会将MessageQueue和消费者ID进行排序后，再用不同的分配算法进行分配。内置的分配的算法共有六种，分别对应AllocateMessageQueueStrategy下的六种实现类，可以在consumer中直接set来指定。默认情况下使用的是最简单的平均分配策略。

- AllocateMachineRoomNearby： 将同机房的Consumer和Broker优先分配在一起。

这个策略可以通过一个machineRoomResolver对象来定制Consumer和Broker的机房解析规则。然后还需要引入另外一个分配策略来对同机房的Broker和Consumer进行分配。一般也就用简单的平均分配策略或者轮询分配策略。

> 感觉这东西挺鸡肋的，直接给个属性指定机房不是挺好的吗。

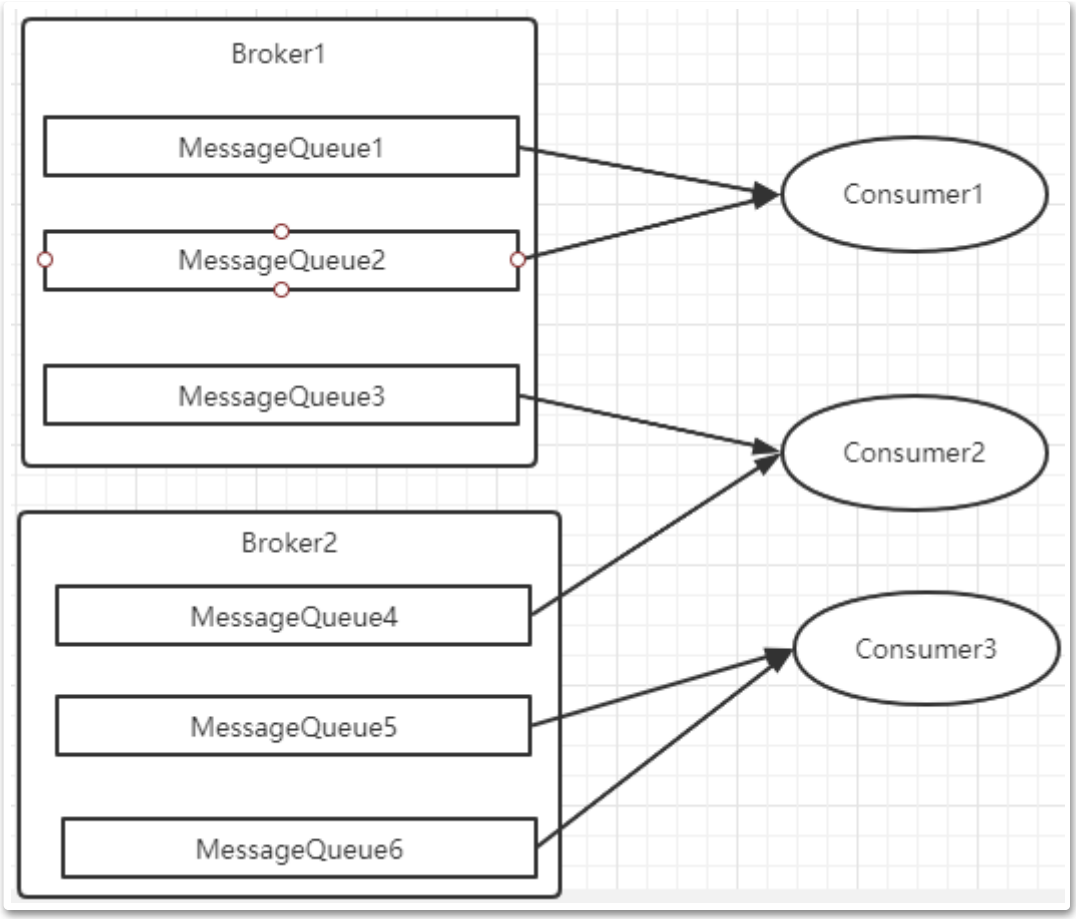
源码中有测试代码AllocateMachineRoomNearByTest。

在示例中：Broker的机房指定方式：messageQueue.getBrokerName().split("-")[0]，而Consumer的机房指定方式：clientID.split("-")[0]

clinetID的构建方式：见ClientConfig.buildMQClientId方法。按他的测试代码应该是要把clientIP指定为IDC1-CID-0这样的形式。

- AllocateMessageQueueAveragely：平均分配。将所有MessageQueue平均分给每一个消费者
- AllocateMessageQueueAveragelyByCircle： 轮询分配。轮流的给一个消费者分配一个MessageQueue。
- AllocateMessageQueueByConfig： 不分配，直接指定一个messageQueue列表。类似于广播模式，直接指定所有队列。
- AllocateMessageQueueByMachineRoom：按逻辑机房的概念进行分配。又是对BrokerName和ConsumerIdc有定制化的配置。
- AllocateMessageQueueConsistentHash。源码中有测试代码AllocateMessageQueueConsitentHashTest。这个一致性哈希策略只需要指定一个虚拟节点数，是用的一个哈希环的算法，虚拟节点是为了让Hash数据在换上分布更为均匀。

例如平均分配时的分配情况是这样的：



2、广播模式

广播模式下，每一条消息都会投递给订阅了Topic的所有消费者实例，所以也就没有消息分配这一说。而在实现上，就是在Consumer分配Queue时，所有Consumer都分到所有的Queue。

广播模式实现的关键是将消费者的消费偏移量不再保存到broker当中，而是保存到客户端当中，由客户端自行维护自己的消费偏移量。

七、消息重试

首先对于广播模式的消息， 是不存在消息重试的机制的，即消息消费失败后，不会再重新进行发送，而只是继续消费新的消息。而对于普通的消息，当消费者消费消息失败后，你可以通过设置返回状态达到消息重试的结果。

7.1、如何让消息进行重试

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置。可以有三种配置方式：

- 返回Action.ReconsumeLater-推荐

- 返回null
- 抛出异常

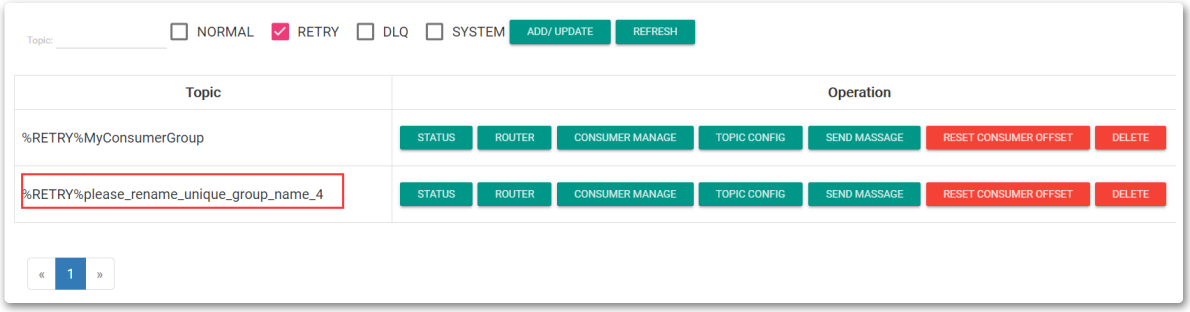
```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //处理消息
        doConsumeMessage(message);
        //方式1：返回 Action.ReconsumeLater，消息将重试
        return Action.ReconsumeLater;
        //方式2：返回 null，消息将重试
        return null;
        //方式3：直接抛出异常， 消息将重试
        throw new RuntimeException("Consumer Message exceotion");
    }
}
```

如果希望消费失败后不重试，可以直接返回Action.CommitMessage。

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            //捕获消费逻辑中的所有异常，并返回 Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回 Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

7.2、重试消息如何处理

重试的消息会进入一个 “%RETRY%” +ConsumeGroup 的队列中。



然后RocketMQ默认允许每条消息最多重试16次，每次重试的间隔时间如下：

重试次数	与上次重试的间隔时间	重试次数	与上次重试的间隔时间
1	10 秒	9	7 分钟
2	30 秒	10	8 分钟
3	1 分钟	11	9 分钟
4	2 分钟	12	10 分钟
5	3 分钟	13	20 分钟
6	4 分钟	14	30 分钟
7	5 分钟	15	1 小时
8	6 分钟	16	2 小时

> 这个重试时间跟延迟消息的延迟级别是对应的。不过取的是延迟级别的后16级别。
>
> messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
>
> 这个重试时间可以将源码中的org.apache.rocketmq.example.quickstart.Consumer里的消息监听器返回状态改为RECONSUME_LATER测试一下。

重试次数：

如果消息重试16次后仍然失败，消息将不再投递。转为进入死信队列。

另外一条消息无论重试多少次，这些重试消息的MessageId始终都是一样的。

然后关于这个重试次数，RocketMQ可以进行定制。例如通过consumer.setMaxReconsumeTimes(20);将重试次数设定为20次。当定制的重试次数超过16次后，消息的重试时间间隔均为2小时。

关于MessageId：

在老版本的RocketMQ中，一条消息无论重试多少次，这些重试消息的MessageId始终都是一样的。

但是在4.9.1版本中，每次重试MessageId都会重建。

配置覆盖：

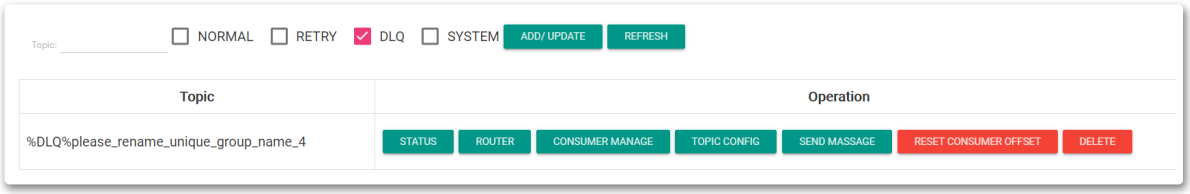
消息最大重试次数的设置对相同GroupID下的所有Consumer实例有效。并且最后启动的Consumer会覆盖之前启动的Consumer的配置。

八、死信队列

当一条消息消费失败，RocketMQ就会自动进行消息重试。而如果消息超过最大重试次数，RocketMQ就会认为这个消息有问题。但是此时，RocketMQ不会立刻将这个有问题的消息丢弃，而会将其发送到这个消费者组对应的一种特殊队列：死信队列。

> RocketMQ默认的重试次数是16次。见源码org.apache.rocketmq.common.subscription.SubscriptionGroupConfig中的retryMaxTimes属性。
>
> 这个重试次数可以在消费者端进行配置。 例如DefaultMQPushConsumer实例中有个setMaxReconsumeTimes方法指定重试次数。

死信队列的名称是%DLQ%+ConsumGroup



死信队列的特征：

- 一个死信队列对应一个ConsumGroup，而不是对应某个消费者实例。
- 如果一个ConsumeGroup没有产生死信队列，RocketMQ就不会为其创建相应的死信队列。
- 一个死信队列包含了这个ConsumeGroup里的所有死信消息，而不区分该消息属于哪个Topic。
- 死信队列中的消息不会再被消费者正常消费。
- 死信队列的有效期跟正常消息相同。默认3天，对应broker.conf中的fileReservedTime属性。超过这个最长时间的消息都会被删除，而不管消息是否消费过。

通常，一条消息进入了死信队列，意味着消息在消费处理的过程中出现了比较严重的错误，并且无法自行恢复。此时，一般需要人工去查看死信队列中的消息，对错误原因进行排查。然后对死信消息进行处理，比如转发到正常的Topic重新进行消费，或者丢弃。

> 注：默认创建出来的死信队列，他里面的消息是无法读取的，在控制台和消费者中都无法读取。这是因为这些默认的死信队列，他们的权限perm被设置成了2:禁读(这个权限有三种 2:禁读，4:禁写,6:可读可写)。需要手动将死信队列的权限配置成6，才能被消费(可以通过mqadmin指定或者web控制台)。

九、消息幂等

9.1、幂等的概念

在MQ系统中，对于消息幂等有三种实现语义：

- at most once 最多一次：每条消息最多只会被消费一次
- at least once 至少一次：每条消息至少会被消费一次
- exactly once 刚刚好一次：每条消息都只会确定的消费一次

这三种语义都有他适用的业务场景。

其中，at most once是最好保证的。RocketMQ中可以直接用异步发送、sendOneWay等方式就可以保证。

而at least once这个语义，RocketMQ也有同步发送、事务消息等很多方式能够保证。

而这个exactly once是MQ中最理想也是最难保证的一种语义，需要有非常精细的设计才行。RocketMQ只能保证at least once，保证不了exactly once。所以，使用RocketMQ时，需要由业务系统自行保证消息的幂等性。

> 关于这个问题，官网上有明确的回答：
>
> ### 4. Are messages delivered exactly once?

> RocketMQ ensures that all messages are delivered at least once.
> In most cases, the messages are not repeated.

但是，对于exactly once语义，阿里云上的商业版RocketMQ是明确有API支持的，至于如何实现的，就不得而知了。

9.2、消息幂等的必要性

在互联网应用中，尤其在网络不稳定的情况下，消息队列 RocketMQ 的消息有可能会出现重复，这个重复简单可以概括为以下情况：

- 发送时消息重复

  当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。 如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。
- 投递时消息重复

  消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。 为了保证消息至少被消费一次，消息队列 RocketMQ 的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。
- 负载均衡时消息重复（包括但不限于网络抖动、Broker 重启以及订阅方应用重启）

  当消息队列 RocketMQ 的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

9.3、处理方式

从上面的分析中，我们知道，在RocketMQ中，是无法保证每个消息只被投递一次的，所以要在业务上自行来保证消息消费的幂等性。

而要处理这个问题，RocketMQ的每条消息都有一个唯一的MessageId，这个参数在多次投递的过程中是不会改变的，所以业务上可以用这个MessageId来作为判断幂等的关键依据。

但是，这个MessageId是无法保证全局唯一的，也会有冲突的情况。所以在一些对幂等性要求严格的场景，最好是使用业务上唯一的一个标识比较靠谱。例如订单ID。而这个业务标识可以使用Message的Key来进行传递。

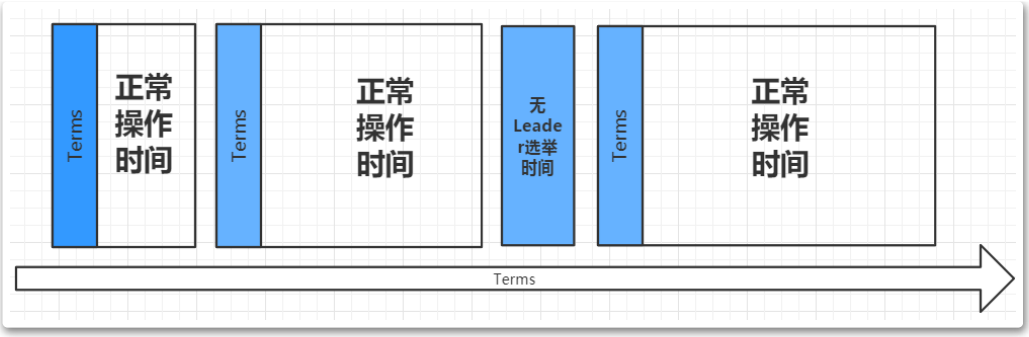
十、详解Dledger集群 -- 了解

Dledger是RocketMQ自4.5版本引入的实现高可用集群的一项技术。他基于Raft算法进行构建，在RocketMQ的主从集群基础上，增加了自动选举的功能。当master节点挂了之后，会在集群内自动选举出一个新的master节点。虽然Dledger机制目前还在不断验证改进的阶段，但是作为基础的Raft算法，已经是目前互联网行业非常认可的一种高可用算法了。Kafka目前也在基于Raft算法，构建摆脱Zookeeper的集群化方案。

RocketMQ中的Dledger集群主要包含两个功能：1、从集群中选举产生master节点。2、优化master节点往slave节点的消息同步机制。

先来看第一个功能：Dledger是使用Raft算法来进行节点选举的。

> 首先：每个节点有三个状态，Leader，follower和candidate(候选人)。正常运行的情况下，集群中会有一个leader，其他都是follower，follower只响应Leader和Candidate的请求，而客户端的请求全部由Leader处理，即使有客户端请求到了一个follower，也会将请求转发到leader。
>
> 集群刚启动时，每个节点都是follower状态，之后集群内部会发送一个timeout信号，所有follower就转成candidate去拉取选票，获得大多数选票的节点选为leader，其他候选人转为follower。如果一个timeout信号发出时，没有选出leader，将会重新开始一次新的选举。而Leader节点会往其他节点发送心跳信号，确认他的leader状态。然后会启动定时器，如果在指定时间内没有收到Leader的心跳，就会转为Candidate状态，然后向其他成员发起投票请求，如果收到半数以上成员的投票，则Candidate会晋升为Leader。然后leader也有可能会退化成follower。
>
> 然后，在Raft协议中，会将时间分为一些任意时间长度的时间片段，叫做term。term会使用一个全局唯一，连续递增的编号作为标识，也就是起到了一个逻辑时钟的作用。



在每一个term时间片里，都会进行新的选举，每一个Candidate都会努力争取成为leader。获得票数最多的节点就会被选举为Leader。被选为Leader的这个节点，在一个term时间片里就会保持leader状态。这样，就会保证在同一时间段内，集群中只会有一个Leader。在某些情况下，选票可能会被各个节点瓜分，形成不了多数派，那这个term可能直到结束都没有leader，直到下一个term再重新发起选举，这也就没有了Zookeeper中的脑裂问题。而在每次重新选举的过程中， leader也有可能会退化成为follower。也就是说，在这个集群中， leader节点是会不断变化的。

然后，每次选举的过程中，每个节点都会存储当前term编号，并在节点之间进行交流时，都会带上自己的term编号。如果一个节点发现他的编号比另外一个小，那么他就会将自己的编号更新为较大的那一个。而如果leader或者candidate发现自己的编号不是最新的，他就会自动转成follower。如果接收到的请求term编号小于自己的编号，term将会拒绝执行。

在选举过程中，Raft协议会通过心跳机制发起leader选举。节点都是从follower状态开始的，如果收到了来自leader或者candidate的心跳RPC请求，那他就会保持follower状态，避免争抢成为candidate。而leader会往其他节点发送心跳信号，来确认自己的地位。如果follower一段时间(两个timeout信号)内没有收到Leader的心跳信号，他就会认为leader挂了，发起新一轮选举。

选举开始后，每个follower会增加自己当前的term，并将自己转为candidate。然后向其他节点发起投票请求，请求时会带上自己的编号和term，也就是说都会默认投自己一票。之后candidate状态可能会发生以下三种变化：

- **赢得选举，成为leader**： 如果它在一个term内收到了大多数的选票，将会在接下的剩余term时间内称为leader，然后就可以通过发送

心跳确立自己的地位。(每一个server在一个term内只能投一张选票，并且按照先到先得的原则投出)

- **其他节点成为leader**： 在等待投票时，可能会收到其他server发出心跳信号，说明其他leader已经产生了。这时通过比较自己的term编号和RPC过来的term编号，如果比对方大，说明leader的term过期了，就会拒绝该RPC,并继续保持候选人身份; 如果对方编号不比自己小,则承认对方的地位,转为follower。
- **选票被瓜分,选举失败**: 如果没有candidate获取大多数选票, 则没有leader产生, candidate们等待超时后发起另一轮选举. 为了防止下一次选票还被瓜分,必须采取一些额外的措施, raft采用随机election timeout(随机休眠时间)的机制防止选票被持续瓜分。通过将timeout随机设为一段区间上的某个值, 因此很大概率会有某个candidate率先超时然后赢得大部分选票。

所以以三个节点的集群为例，选举过程会是这样的：

1. 集群启动时，三个节点都是follower，发起投票后，三个节点都会给自己投票。这样一轮投票下来，三个节点的term都是1，是一样的，这样是选举不出Leader的。
2. 当一轮投票选举不出Leader后，三个节点会进入随机休眠，例如A休眠1秒，B休眠3秒，C休眠2秒。
3. 一秒后，A节点醒来，会把自己的term加一票，投为2。然后2秒时，C节点醒来，发现A的term已经是2，比自己的1大，就会承认A是Leader，把自己的term也更新为2。实际上这个时候，A已经获得了集群中的多数票，2票，A就会被选举成Leader。这样，一般经过很短的几轮选举，就会选举出一个Leader来。
4. 到3秒时，B节点会醒来，他也同样会承认A的term最大，他是Leader，自己的term也会更新为2。这样集群中的所有Candidate就都确定成了leader和follower.
5. 然后在一个任期内，A会不断发心跳给另外两个节点。当A挂了后，另外的节点没有收到A的心跳，就会都转化成Candidate状态，重新发起选举。

然后，Dledger还会采用Raft协议进行多副本的消息同步

> 使用Dledger集群后，数据主从同步会分为两个阶段，一个是uncommitted阶段，一个是commited阶段。

> Leader Broker上的Dledger收到一条数据后，会标记为uncommitted状态，然后他通过自己的DledgerServer组件把这个uncommitted数据发给Follower Broker的DledgerServer组件。
>
> 接着Follower Broker的DledgerServer收到uncommitted消息之后，必须返回一个ack给Leader Broker的Dledger。然后如果Leader Broker收到超过半数的Follower Broker返回的ack之后，就会把消息标记为committed状态。
>
> 再接下来， Leader Broker上的DledgerServer就会发送committed消息给Follower Broker上的DledgerServer，让他们把消息也标记为committed状态。这样，就基于Raft协议完成了两阶段的数据同步。

> 有道云笔记链接地址：
>
> 文档：五期VIP03-深度解析RocketMQ高性能背后...
> 链接：http://note.youdao.com/noteshare?id=b1d25d692746087094a9eb2b2c3e4023&sub=2E3B45BA6D6C4D3E93C64FC0D8A7B4BF