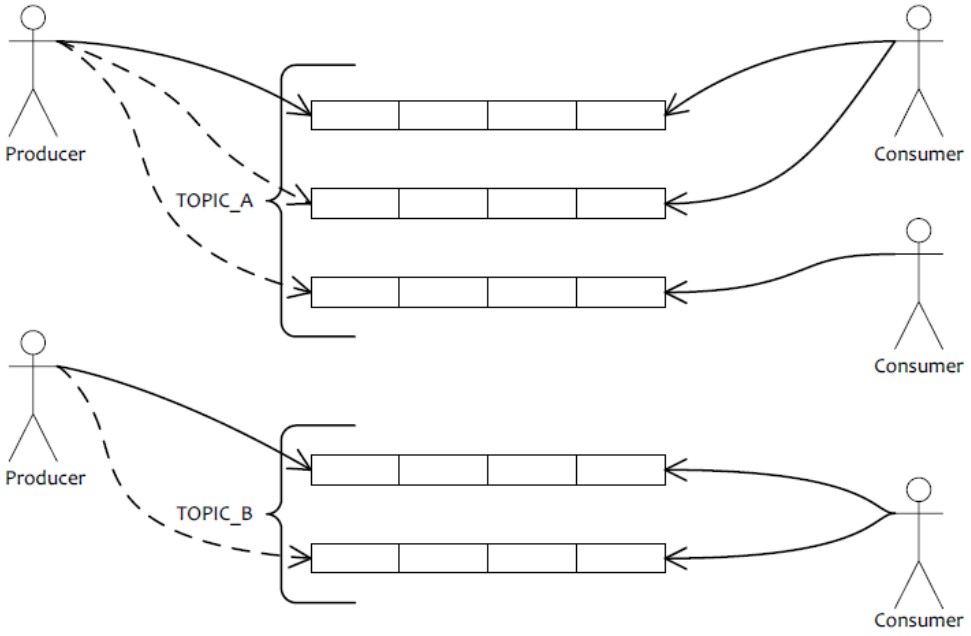
RocketMq 源码分析 ---存储层



什么是 rocketmq:

遵循 JMS 规范与 CORBA Notification 规范(消息队列服务规范)的分布式、队列模型的消息中间件。

特点：

Publish/Subscribe (发布订阅模型，区别于 p2p 模式)

消息顺序 (消息顺序性：分为普通顺序和严格顺序，顺序指的是按照单线程 producer 产生的顺序进行消费。普通顺序不能保证在宕机或者重启的情况下顺序的一致性，因为重启后队列的定位会出现短暂的错乱。严格顺序，牺牲了分布式的 failover 特性，也就是如果有一台宕机了，相当于集群都不可用，可以使用同步双写，但是在切换服务器的时候会造成短暂时间的服务不可用。)

消息 push/pull 模型 (push 底层是通过 pull 方式实现的)

高效的订阅者水平扩展能力

实时消息订阅机制（在消息不堆积情况下，消息到达Broker后，能立刻到达Consumer。

RocketMQ 使用长轮询 Pull 方式，可保证消息非常实时，消息实时性不低于Push。）

亿级消息堆积能力（数据持久化到硬盘）

消息持久化

Producer、Broker、Consumer 均可独立部署

支持消息广播和集群 广播

消费状态可以保存在客户端或者服务端(offset 本地管理和服务器端管理)

支持本地事务和 HA 事务(同步双写异步复制)

同类产品 :kafka、notify、timetunnel、MetaQ(B2B、天猫、淘宝)、RabbitMQ、Jafka

RocketMQ 安装及部署:

消息中间件需要解决的问题：

1. 发布订阅或者点对点
2. 消息优先级

注：rocketmq 其实没有实现消息优先级，由应用来实现。只要是通过设置不同的 topic 来实现的，可以将不同的 topic 在应用里分为高、中、低优先级，消费的时候通过之前的约定进行优先级消费。

3. 消息顺序

4. 消息过滤:

Broker 端消息过滤

在 Broker 中 ,按照 Consumer 的要求做过滤 ,优点是减少了对于 Consumer 无用消息的网络传输。

缺点是增加了 Broker 的负担，实现相对复杂。

(1). 淘宝 Notify 支持多种过滤方式，包含直接按照消息类型过滤，灵活的语法表达式过滤，几乎可以满足最苛刻的过滤需求。

(2). 淘宝 RocketMQ 只支持按照简单的 Message Tag 过滤。

(3). CORBA Notification 规范中也支持灵活的语法表达式过滤。

Consumer 端消息过滤

这种过滤方式可由应用完全自定义实现，但是缺点是很多无用的消息要传输到 Consumer 端。

这种过滤方式可由应用完全自定义实现，但是缺点是很多无用的消息要传输到 Consumer 端。

5. 消息持久化
6. 消息可靠性
7. 消息低延迟
8. 至少投递一次(At least Once)

9. Exactly Only Once

(1). 发送消息阶段，不允许发送重复的消息。

(2). 消费消息阶段，不允许消费重复的消息。

Rocketmq 没有实现这个功能

扫盲：

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。

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。

Push Consumer

Consumer的一种，应用通常向Consumer对象注册一个Listener接口，一旦收到消息，Consumer对象立刻回调Listener接口方法。

Pull Consumer

Consumer的一种，应用通常主动调用Consumer的拉消息方法从Broker拉消息，主动权由应用控制。

Producer Group

一类Producer的集合名称，这类Producer通常发送一类消息，且发送逻辑一致。

Consumer Group

一类Consumer的集合名称，这类Consumer通常消费一类消息，且消费逻辑一致。

Broker

消息中转角色，负责存储消息，转发消息，一般也称为Server。在JMS规范中称为Provider。

广播消费

一条消息被多个Consumer消费，即使这些Consumer属于同一个Consumer Group，消息也会被Consumer Group中的每个Consumer都消费一次，广播消费中的Consumer Group概念可以认为在消息划分方面无意义。

广播消费是offset保存在consumer本地的

集群消费才保存在broker上

一个Consumer Group中的Consumer实例平均分摊消费消息。例如某个Topic有9条消息，其中一个Consumer Group有3个实例（可能是3个进程，或者3台机器），那么每个实例只消费其中的3条消息。

顺序消息

消费消息的顺序要同发送消息的顺序一致，在RocketMQ中，主要指的是局部顺序，即一类消息为满足顺序性，必须Producer单线程顺序发送，且发送到同一个队列，这样Consumer就可以按照Producer发送的顺序去消费消息。

普通顺序消息

顺序消息的一种 ,正常情况下可以保证完全的顺序消息 ,但是一旦发生通信异常，Broker重启，由于队列总数发生变化，哈希取模后定位的队列会变化，产生短暂的消息顺序不一致。

如果业务能容忍在集群异常情况（如某个Broker宕机或者重启）下，消息短暂的乱序，使用普通顺序方式比较合适。

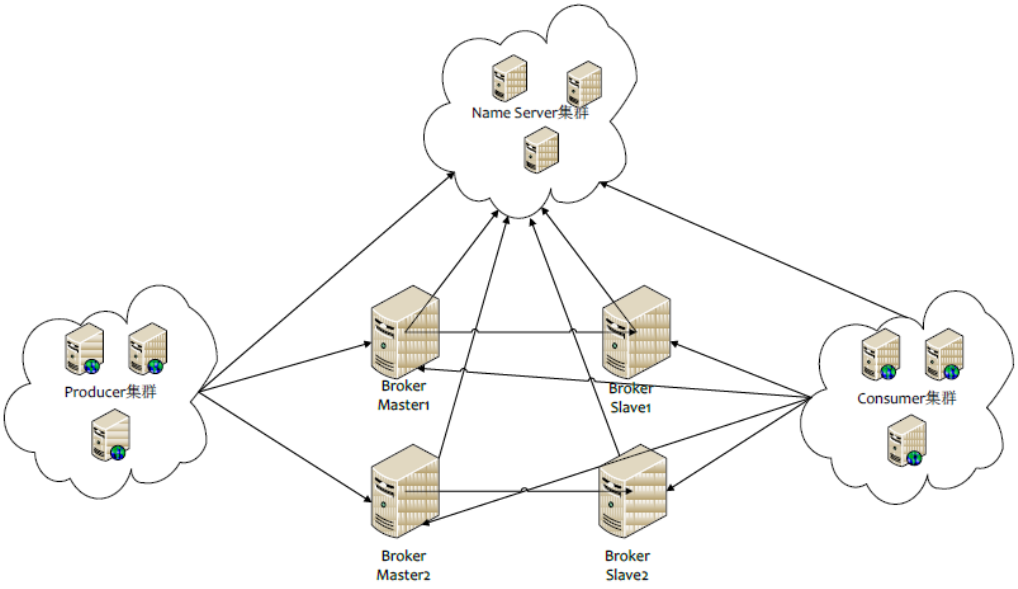
严格顺序消息

顺序消息的一种，无论正常异常情况都能保证顺序，但是牺牲了分布式Failover特性，即Broker集群中只要有一台机器不可用，则整个集群都不可用，服务可用性大大降低。

如果服务器部署为同步双写模式，此缺陷可通过备机自动切换为主避免，不过仍然会存在几分钟的服务不可用。（依赖同步双写，主备自动切换，自动切换功能目前还未实现）

目前已知的应用只有数据库binlog同步强依赖严格顺序消息 ,其他应用绝大部分都可以容忍短暂乱序，推荐使用普通的顺序消息。

网络拓扑结构：

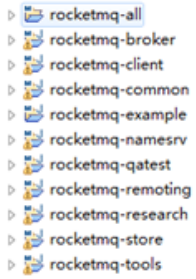


注：Producer、Consumer、Broker 可以分开部署，无相互依赖。Producer、Consumer 通过 Name Server 来寻找 Broker，Broker 启动之后，会定时发布 Topic 路由信息注册到 NameServer。

快速部署：

获取源码：

RocketMQ源码目录



Rocketmq-broker:队列服务端，上层接受consumer与producer的message request，并进行处理。HA的基本单元，支持同步双写和异步复制。下层主要调用store层服务进行数据的持久化(commitlog)
Rocketmq-client : consumer与producer的API层。提供了与broker交互的接口。应用开发时直接引用该包即可。
Rocketmq-common:公共类集合，工程使用的很多数据结构都在这里定义
Rocketmq-example : rocketmq提供的示例
Rocket-namesrv : rocketmq NameServer的实现
Rocketmq - qatest : QA测试类
Rocketmq-remoting : rocketmq 远程调用接口
Rocketmq-research:测试类
Rocketmq-store:rocketmq的存储层
Rocketmq-tools:mq集群管理和相关监控工具的实现

快速启动：

安装：

```
git clone https://github.com/alibaba/RocketMQ.git

cd rocketmq

sh install.sh

cd devenv
```

启动NameServer

nohup mqnamesrv &

启动Broker(指定NameServer地址)

mqbroker –n “IP1:9876;IP2 :9876“ 或者 export NAMESRV_ADDR=192.168.0.1:9876;192.168.0.2:9876

集群安装：

集群拓扑类型：

1. 单个 Master

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

nohup sh mqbroker -n 192.168.1.1:9876 –c

$ROCKETMQ_HOME/conf/2m-noslave/broker-a.properties &

2. 多 Master 模式

一个集群无 Slave，全是 Master，例如 2 个 Master 或者 3 个 Master

优点：配置简单，单个 Master 宕机或重启维护对应用无影响，在磁盘配置为 RAID10 时，即使机器宕机不可恢复情况下，由于 RAID10 磁盘非常可靠，消息也不会丢（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，消息实时性会受到受到影响。

3. 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。

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 ,因为 Master 宕机后，消费者仍然可以从 Slave 消费，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，会丢失少量消息。

4. 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，主备都写成功，向应用返回成功。

优点：数据与服务都无单点，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

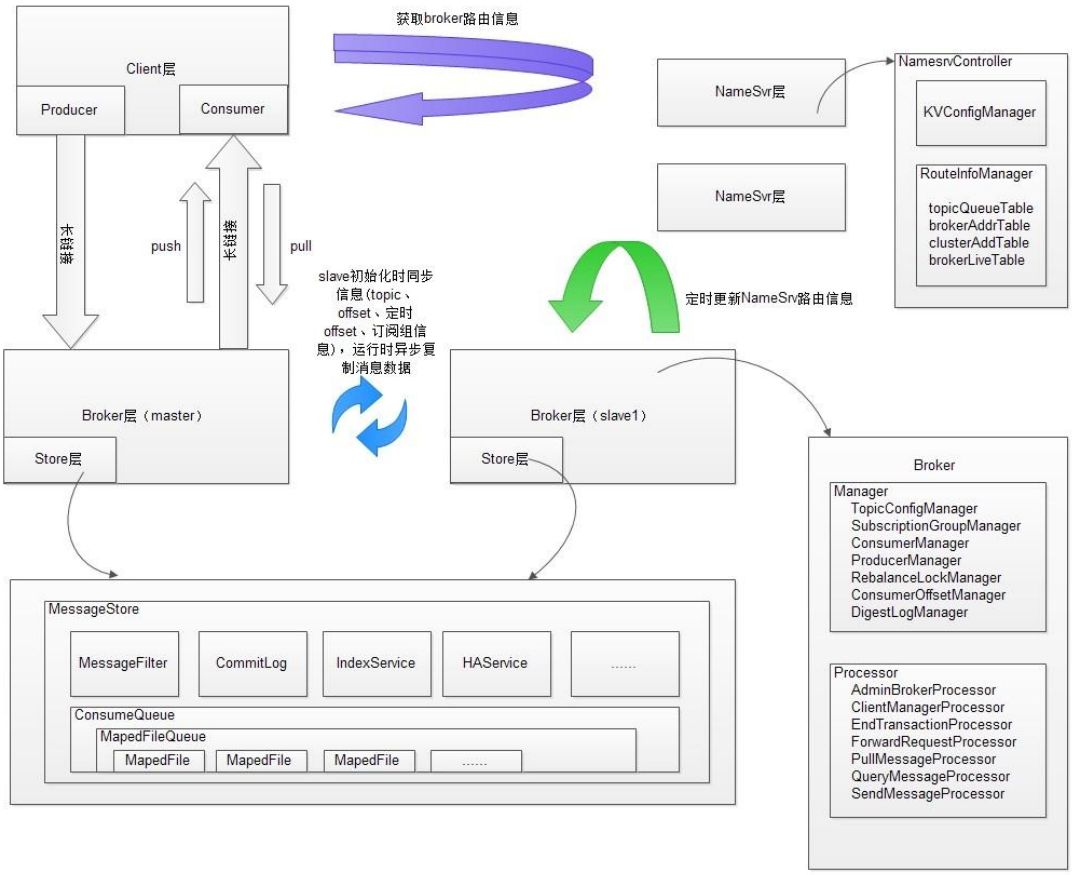
缺点 :性能比异步复制模式略低 ,大约低 10%左右 ,发送单个消息的 RT 会略高。

目前主宕机后，备机不能自动切换为主机，后续会支持自动切换功能。

部署时需要注意的问题：

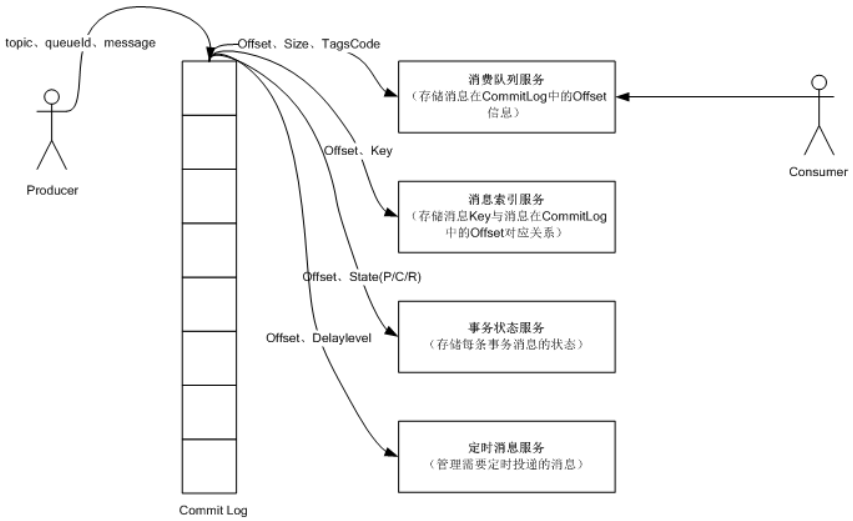
1. RocketMQ 必须运行在 JRE64 位操作系统上
2. RocketMQ 所有节点都支持分布式部署 ,broker 可以部署在不同机器上，需要分别启动
3. RocketMQ 支持 JMS 客户端 API
4. 源码目录下的 bin 是不能启动 broker 的，需要在编译后的目录下启动
5. Tools 目录下的有监控 MQ 状态的脚本
6. {user.home}目录下需要分配足够空间 ,Broker 的数据都会持久化到这个目录下

RocketMq 模块之间的关系：



Rocket FAQ:

存储层视图：



存储特点：

零拷贝方式

1.mmap+write 方式，这种方式适用于小块文件传输，不能很好的利用 DMA，所以也就消耗了 cpu 资 源，更多的需要程序控制内存，编程比较复杂，需要避免 jvm crash 问题。

RocketMq 如何避免 jvm crash：

由于 MapedByteBuffer 可以通过特殊方法人为释放掉 ,实际调用了 unmap 方法。此时之前映射到 JVM 的地址空间就非法，如果此后仍然对 MapedByteBuffer 进行读写 ,系统就会向 JVM 发送 SIGBUS 信号来通知进程此种操作非法。（这种一般是由于程序员没有处理好并发问题导致）

RocketMQ 如何避免？

采用引用计数方法，参考 C++智能指针实现方式。只要引用计数不为 0，MapedByteBuffer 对象就不会释放。

为什么不使用读写锁来避免？

采用引用计数使用的是原子变量，并发下要比读写锁性能更好

采用读写锁，每次对数据读写都要加锁，代码较冗余（个人看法）。

这种方式存在什么弊端？

如果不能正确操作引用计数，可能会导致文件无法删除，所以 RocketMQ 增加了一个补救措施，就是一旦关闭文件服务后，如果超过 2 分钟，引用计数还没有变为 0，则强制释放。

2. sendFile 方式，使用的是 DMA，消耗 cpu 比较少，适用大块文件传输，不需要考虑 jvm crash 问题。但是效率很低，而且无法使用异步 IO。

注：Server 在向 Consumer 返回消息时，没有直接使用 sendfile（transferTo）接口，但是使用了类似机制，直接将虚拟内存地址传给 socket，无论实际数据是在物理内存还是在文件，都由操作系统进行管理，也就是说消息数据不会重新 load 到 java 堆。

另外：将 pagecache 直接传输给 socket，在操作系统层面会做以下优化

因为 pagecache 内存本身是内核与应用共享的内存，所以不需要用户态向内核态内存拷贝。

Socket 在发现是 pagecache 时，会将 pagecache 直接传输，不需要将数据拷贝到 socket 缓冲区。（TCP 协议栈优化）

刷盘策略：

实践过程中需要注意的一些问题：

存储层类之间的关系：

RocketMQ 是否需要流控？

对于发送消息，接收消息不需要流控

因为性能测试中，千兆网卡上下行同时压满（流量都在 100M 以上），系统指标仍然正常。但是同时需要监控磁盘空间剩余量，因为在高 TPS 场景下，磁盘很快就会被写满。

Server 内部将消息消息位置信息派发至各个 Consume Queue 需要流控

在 1 万队列以下一般不需要流控，但是一旦超过 1 万个队列，则对队列的写性能会下降 ,此时前端请求过来 ,消息位置信息会在 java 堆中堆积，默认阀值是 40 万，超过则开始流控，对前端请求做 1 毫秒 sleep。

写入 Commit Log 成功，但是写入 Consume Queue 失败怎么办？

- JVM CRASH 情况下，消息位置信息未写入 Consume Queue，如何处理？

  JVM 重启后，从 Commit Log 恢复 Consume Queue，非全量恢复，只恢复当前可能丢失的数据

u 写入 Consume Queue 发生 IO 错误如何处理？

一旦发生 IO 错误，则认为可能是 IO 设备故障，停止对外写服务，但是数据仍然可读。

u JVM 发生 outofmemory

这种情况，可理解为 jvm 不能对外服务，Consume Queue 与 Commit Log 可能不一致。必须重启才能保证消息一致。

写完 Commit Log 后，消息位置信息是同步写入 Consume Queue 还是异步写入？

消息一旦写入 Commit Log ,则返回消息对应的 CommitLog offset ,size，等信息，将这类消息位置信息传递至另一个独立的 Dispatch 线程，然后主流程返回，并向发送方返回成功应答。

Consumer 拉消息是单个方式还是批量方式拉？

是批量方式拉消息，服务器可以配置一次最多拉多少条，最多多少字节，客户端也可以配置。

以最小的为主。

JVM CRASH 后，写入 PAGECACHE 的未刷盘数据是否会丢失？

不会。

为了避免 os 本身的刷盘机制与应用自己的刷盘机制冲突，设置了以下参数来抑制 os 刷盘。

sudo sysctl vm.dirty_writeback_centisecs=360000

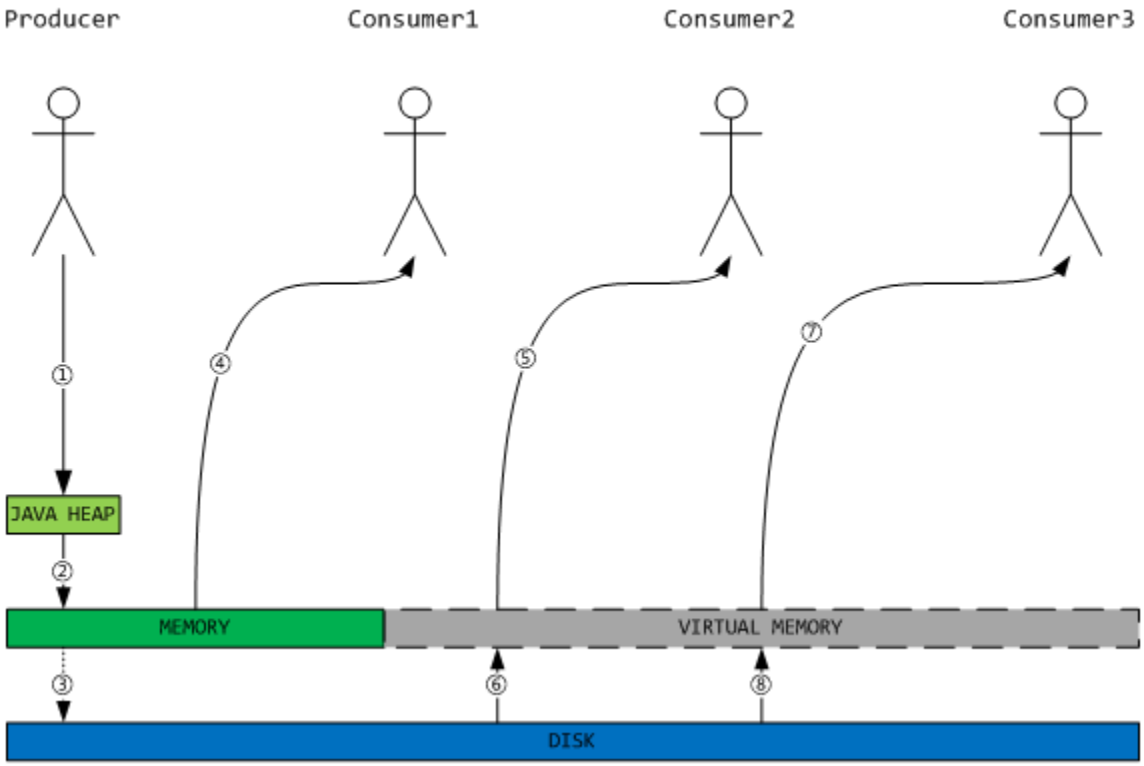
但是同时会导致 JVM CRASH 后，系统的 pagecache 不能及时刷盘，此时可以通过以下命令来刷盘

Sync

是否可以认为 Linux64 位可以无限制映射 PAGECACHE？

Pagecache 是由内核维护的，映射的越多，内核数据结构占用的物理内存越大，所以要映射更多的 pagecache，对机器的物理内存大小要求更高。

消息如何在 java 堆，物理内存，虚拟内存，磁盘之间流动？



图表 1 消息数据流图（机器视图）

(1). Producer 发送消息，消息从 socket 进入 java 堆。

(2). Producer 发送消息，消息从 java 堆转入 PAGACACHE，物理内存。

(3). Producer 发送消息，由异步线程刷盘，消息从 PAGECACHE 刷入磁盘。

(4). Consumer 拉消息（正常消费），消息直接从 PAGECACHE（数据在物理内存）转入 socket，到达 consumer，不经过 java 堆。

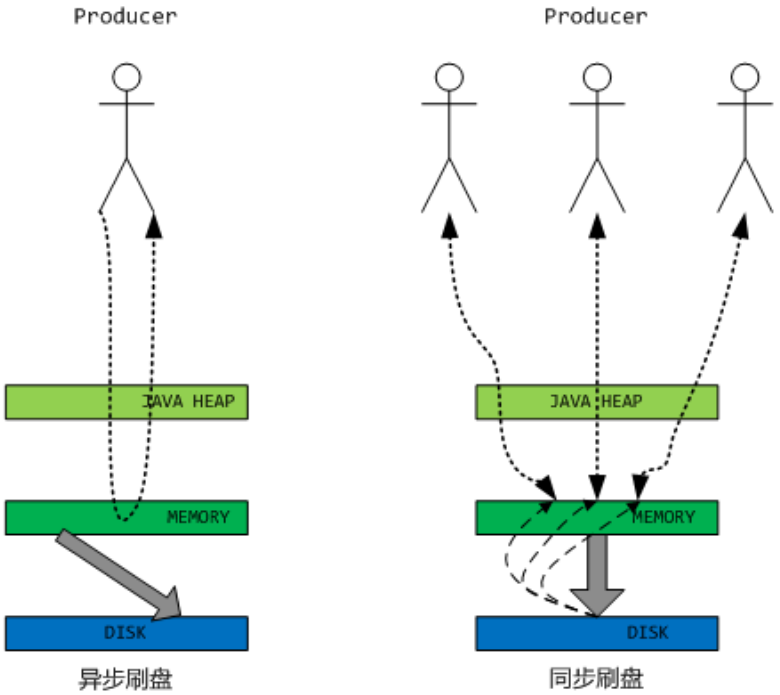
这种消费场景最多，线上 96G 物理内存，按照 1K 消息算，可以在物理内存缓存 1 亿条消息。

(5). Consumer 拉消息（异常消费），消息直接从 PAGECACHE（数据在虚拟内存）转入 socket。

(6). Consumer 拉消息（异常消费），由于 Socket 访问了虚拟内存，产生缺页中断，此时会产生磁盘 IO，从磁盘 Load 消息到 PAGECACHE，然后直接从 socket 发出去。

(7). 同 5 一致。

(8). 同 6 一致。



图表 2刷盘策略