

offset类似游标记录每次消费位置

## ISR机制的理解

- 1、AR (Assigned Replicas) 一个partition的所有副本（就是replica，不区分leader或follower）
- 2、ISR (In-Sync Replicas) 能够和 leader 保持同步的 follower + leader本身组成的集合。（10s或者设置落后的offset数值）
- 3、OSR (Out-Sync Relipcas) 不能和 leader 保持同步的 follower 集合
- 4、公式：AR = ISR + OSR

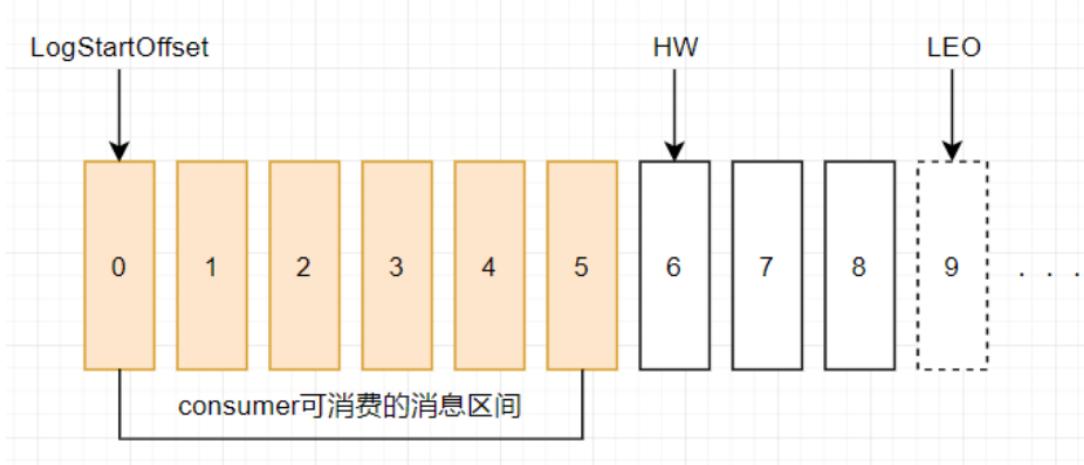
## 数据的处理过程是

Producer在发布消息到某个Partition时，先通过ZooKeeper找到副本Leader，leader读写。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。

Follower在收到该消息并写入其Log后，向Leader发送ACK。一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经commit了，Leader将增加HW(HighWatermark)并且向Producer发送ACK。

**LEO**：即日志末端位移(log end offset)，记录了该副本底层日志(log)中下一条消息的位移值。注意是下一条消息！也就是说，如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。另外，leader LEO和follower LEO的更新是有区别的。

**HW**：即上面提到的水位值 (Hight Water)。对于同一个副本对象而言，其HW值不会大于LEO值。小于等于HW值的所有消息都被认为是“已备份”的(replicated)。同理，leader副本和follower副本的HW更新是有区别的。通过下面这幅图来表达LEO、HW的含义，随着follower副本不断和leader副本进行数据同步，follower副本的LEO会主键后移并且追赶到leader副本，这个追赶上上的判断标准是当前副本的LEO是否大于或者等于leader副本的HW，这个追赶上也会使得被踢出的follower副本重新加入到ISR集合中。



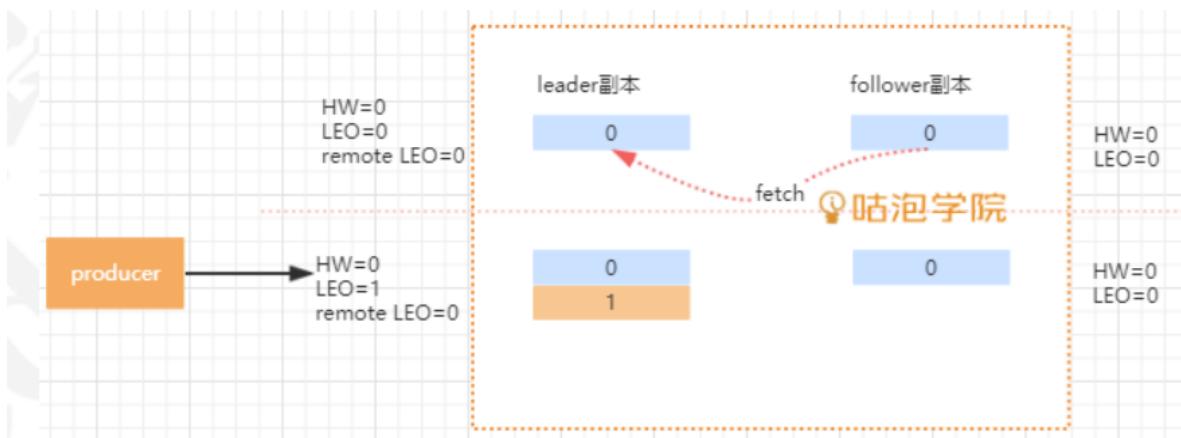
## 初始状态

初始状态下，leader和follower的HW和LEO都是0，leader副本会保存remote LEO，表示所有follower LEO，也会被初始化为0，这个时候，producer没有发送消息。follower会不断地向leader发送FETCH请求，但是因为没有数据，这个请求会被leader寄存，当在指定的时间之后会强制完成请求，这个时间配置是(replica.fetch.wait.max.ms)，如果在指定时间内producer有消息发送过来，那么kafka会唤醒fetch请求，让leader继续处理数据的同步处理会分两种情况，这两种情况下处理方式是不一样的 第一种是leader处理完producer请求之后，follower发送一个fetch请求过来 第二种是follower阻塞在leader指定时间之内，leader副本收到producer的请求。

## 第一种情况

### 生产者发送一条消息

leader处理完producer请求之后，follower发送一个fetch请求过来。状态图如下



leader副本收到请求以后，会做几件事情

1. 把消息追加到log文件，同时更新leader副本的LEO

2. 尝试更新leader HW值。这个时候由于follower副本还没有发送fetch请求，那么leader的remote LEO仍然是0。leader会比较自己的LEO以及remote LEO的值发现最小值是0，与HW的值相同，所以不会更新HW

follower fetch消息follower 发送fetch请求，leader副本的处理逻辑是：

1. 读取log数据、更新remote LEO=0(follower还没有写入这条消息，这个值是根据follower的fetch 请求中的offffset来确定的)
2. 尝试更新HW，因为这个时候LEO和remote LEO还是不一致，所以仍然是HW=0
3. 把消息内容和当前分区的HW值发送给follower副本

follower副本收到response以后

1. 将消息写入到本地log，同时更新follower的LEO
2. 更新follower HW，本地的LEO和leader返回的HW进行比较取小的值，所以仍然是0

第一次交互结束以后，HW仍然还是0，这个值会在下一次follower发起fetch请求时被更新 follower发第二次fetch请求，leader收到请求以后

1. 读取log数据
2. 更新remote LEO=1，因为这次fetch携带的offffset是1.
3. 更新当前分区的HW，这个时候leader LEO和remote LEO都是1，所以HW的值也更新为1
4. 把数据和当前分区的HW值返回给follower副本，这个时候如果没有数据，则返回为空follower副本收到response以后
5. 如果有数据则写本地日志，并且更新LEO
6. 更新follower的HW值

到目前为止，数据的同步就完成了，意味着消费端能够消费offffset=1这条消息。

## 第二种情况

前面说过，由于leader副本暂时没有数据过来，所以follower的fetch会被阻塞，直到等待超时或者 leader接收到新的数据。当leader收到请求以后会唤醒处于阻塞的fetch请求。处理过程基本上和前面说的一致

1. leader将消息写入本地日志，更新Leader的LEO
2. 唤醒follower的fetch请求
3. 更新HW

kafka使用HW和LEO的方式来实现副本数据的同步，本身是一个好的设计，但是在某个地方会存在一个数据丢失的问题，当然这个丢失只出现在特定的背景下。我们回想一下，HW的值是在新一轮FETCH中才会被更新。我们分析下这个过程为什么会出现数据丢失

## 数据丢失的问题

### 前提

`min.insync.replicas=1` //\*\*设定ISR中的最小副本数是多少，默认值为1（在server.properties中配置），并且acks参数设置为-1（表示需要所有副本确认）时，此参数才生效

表达的含义是，至少需要多少个副本同步才能表示消息是提交的，所以，当`min.insync.replicas=1`的时候，一旦消息被写入leader端log即被认为是“已提交”，而延迟一轮FETCH RPC更新HW值的设计使得follower HW值是异步延迟更新的，倘若在这个过程中leader发生变更，那么成为新leader的follower的HW值就有可能是过期的，使得clients端认为是成功提交的消息被删除。

**producer的ackacks**配置表示producer发送消息到broker上以后的确认值。有三个可选项

0：表示producer不需要等待broker的消息确认。这个选项时延最小但同时风险最大（因为当server宕机时，数据将会丢失）。

1：表示producer只需要获得kafka集群中的leader节点确认即可，这个选择时延较小同时确保了leader节点确认接收成功。

all(-1)：需要ISR中所有的Replica给予接收确认，速度最慢，安全性最高，但是由于ISR可能会缩小到仅包含一个Replica，所以设置参数为all并不能一定避免数据丢失，

## 数据丢失的解决方案

在kafka0.11.0.0版本之后，引入了一个leader epoch来解决这个问题，所谓的leader epoch实际上是一对值（epoch, offset），epoch代表leader的版本号，从0开始递增，当leader发生变更，epoch就+1，而offset则是对应这个epoch版本的leader写入第一条消息的offset，比如(0,0), (1,50)，表示第一个leader从offset=0开始写消息，一共写了50条。第二个leader版本号是1，从offset=50开始写，这个信息会持久化在对应的分区的本地磁盘上，文件名是/tmp/kafka-log/topic/leader-epoch-checkpoint。

leader broker中会保存这样一个缓存，并且定期写入到checkpoint文件中。当leader写log时它会尝试更新整个缓存：如果这个leader首次写消息，则会在缓存中增加一个条目；否则就不做更新。而每次副本重新成为leader时会查询这部分缓存，获取出对应leader版本的offset。我们基于同样的情况来分析，follower宕机并且恢复之后，有两种情况，如果这个时候leader副本没有挂，也就是意味着没有发生leader选举，那么follower恢复之后并不会去截断自己的日志，而是先发送一个OffsetsForLeaderEpochRequest请求给到leader副本，leader副本收到请求之后返回当前的 LEO。

如果follower副本的leaderEpoch和leader副本的epoch相同，leader的leo只可能大于或者等于 follower副本的leo值，所以这个时候不会发生截断。

如果follower副本和leader副本的epoch值不同，那么leader副本会查找follower副本传过来的 epoch+1 在本地文件中存储的StartOffset 返回给 follower 副本，也就是新leader副本的LEO。这样也避免了数据丢失的问题。

如果leader副本宕机了重新选举新的leader，那么原本的follower副本就会变成leader，意味着epoch 从0变成1，使得原本follower副本中LEO的值的到了保留。

## 选举机制

### 控制器 (Broker) 选举

controller的选举是通过broker在zookeeper的"/controller"节点下创建临时节点来实现的，并在该节点中写入当前broker的信息 {"version":1,"brokerid":1,"timestamp":"1512018424988"}，利用zookeeper的强一致性特性，一个节点只能被一个客户端创建成功，创建成功的broker即为controller，即"先到先得"。其他broker启动时也会在zookeeper中创建临时节点，创建watch对象。

### 分区副本选举机制

1. KafkaController会监听ZooKeeper的/brokers/ids节点路径，一旦发现有broker挂了，执行下面的逻辑
2. leader副本在该broker上的分区就要重新进行leader选举，目前的选举策略是
  - a) 优先从isr列表中选出第一个作为leader副本，这个叫优先副本，理想情况下有限副本就是该分区的leader副本
  - b) 如果isr列表为空，则查看该topic的unclean.leader.election.enable配置。

`unclean.leader.election.enable`: 为true则代表允许选用非isr列表的副本作为leader，那么此时就意味着数据可能丢失，为false的话，则表示不允许，直接抛出NoReplicaOnlineException异常，造成leader副本选举失败。

c) 如果上述配置为true，则从其他副本中选出一个作为leader副本，并且isr列表只包含该leader副本。一旦选举成功，则将选举后的leader和isr和其他副本信息写入到该分区的对应的zk路径上。

1. (epoch, offset)

## 消费组选主

第一个加入消费组的消费者即为消费组的leader，重选根据key的hash随机一个

## 消息的存储

消息发送端发送消息到broker上以后，消息是如何持久化的呢？那么接下来去分析下消息的存储。首先我们需要了解的是，kafka是使用**日志文件的方式来保存**生产者和发送者的消息，每条消息都有一个offset值来表示它在分区中的偏移量。Kafka中存储的一般都是海量的消息数据，为了避免日志文件过大，Log并不是直接对应在一个磁盘上的日志文件，而是**对应磁盘上的一个目录**，这个目录的命名规则是`<topic_name>_<partition_id>`

## 消息的文件存储机制

一个topic的多个partition在物理磁盘上的保存路径，路径保存在`/tmp/kafka-logs/topic_partition`，包含日志文件、索引文件和时间索引文件。kafka是通过分段的方式将Log分为多个LogSegment，LogSegment是一个逻辑上的概念，一个LogSegment对应磁盘上的一个日志文件和一个索引文件，其中日志文件是用来记录消息的。索引文件是用来保存消息的索引。那么这个LogSegment是什么呢？

## LogSegment

假设kafka以partition为最小存储单位，那么我们可以想象当kafka producer不断发送消息，必然会引起partition文件的无线扩张，这样对于消息文件的维护以及被消费的消息的清理带来非常大的挑战，所以kafka以**segment为单位又把partition进行细分**。每个partition相当于一个巨型文件被平均分配到多个大小相等的segment数据文件中（每个segment文件中的消息不一定相等），这种特性方便已经被消费的消息的清理，提高磁盘的利用率。

`log.segment.bytes=107370` (设置分段大小)，默认是1gb，我们把这个值调小以后，可以看到日志

## 分段的效果

抽取其中3个分段来进行分析segment file由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和".log"分别表示为**segment索引文件、数据文件**. segment文件命名规则：partition全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值进行递增。数值最大为64位long大小，20位数字字符串长度，没有数字用0填充

## 查看segment文件命名规则

通过下面这条命令可以看到kafka消息日志的内容 假如第一个log文件的最后一个offset为:5376,所以下一个segment的文件命名为:

00000000000000005376.log。对应的index为  
00000000000000005376.index

segment中index和log的对应关系从所有分段中，找一个分段进行分析 为了提高查找消息的性能，为每一个日志文件添加2个索引索引文件：OffsetIndex 和 TimeIndex，分别对应.index以及.timeindex, TimeIndex索引文件格式：它是映射时间戳和相对offset

## 查看索引内容：

```
sh kafka-run-class.sh kafka.tools.DumpLogSegments --files /tmp/kafka-logs/test- 0/00000000000000000000.log --print-data-log
```

```
sh kafka-run-class.sh kafka.tools.DumpLogSegments --files /tmp/kafka-logs/test- 0/00000000000000000000.index --print-data-log
```

如图所示，index 中存储了索引以及物理偏移量。log 存储了消息的内容。索引文件的元数据执行对应数据文件中 message 的物理偏移地址。举个简单的案例来说，以 [4053,80899] 为例，在 log 文件中，对应的是第 4053 条记录，物理偏移量 (position) 为 80899. position 是 ByteBuffer 的指针位置

## 在partition中如何通过offset查找message

查找的算法是

1. 根据offset的值，查找segment段中的index索引文件。由于索引文件命名是以上一个文件的最后一个offset进行命名的，所以，使用二分查找算法能够根据offset快速定位到指定的索引文件。
2. 找到索引文件后，根据offset进行定位，找到索引文件中的符合范围的索引。 (kafka采用稀疏索引的方式来提高查找性能)

3. 得到position以后，再到对应的log文件中，从position出开始查找offset对应的消息，将每条消息的offset与目标offset进行比较，直到找到消息

比如说，我们要查找offset=2490这条消息，那么先找到00000000000000000000.index，然后找到[2487,49111]这个索引，再到log文件中，根据49111这个position开始查找，比较每条消息的offset是否大于等于2490。最后查找到对应的消息以后返回

### Log\*\*文件的消息内容分析\*\*

前面我们通过kafka提供的命令，可以查看二进制的日志文件信息，一条消息，会包含很多的字段。 offset和position这两个前面已经讲过了、 createTime表示创建时间、 keysize和valuesize表示key和 value的大小、 compresscodec表示压缩编码、 payload:表示消息的具体内容

## 日志的清除策略以及压缩策略

```
offset: 5371 position: 102124 CreateTime: 1531477349286 isValid: true  
keysize: -1 valuesize: 12 magic: 2 compresscodec: NONE producerId: -1  
producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: []  
payload: message_5371
```

**日志清除策略** 前面提到过，日志的分段存储，一方面能够减少单个文件内容的大小，另一方面，方便kafka进行日志清理。日志的清理策略有两个

1. **根据消息的保留时间**，当消息在kafka中保存的时间超过了指定的时间，就会触发清理过程
2. **根据topic存储的数据大小**，当topic所占的日志文件大小大于一定的阀值，则可以开始删除最旧的消息。kafka会启动一个后台线程，定期检查是否存在可以删除的消息 通过log.retention.bytes和log.retention.hours这两个参数来设置，当其中任意一个达到要求，都会执行删除。默认的保留时间是：7天

## 日志压缩策略

Kafka还提供了“日志压缩（Log Compaction）”功能，通过这个功能可以有效的减少日志文件的大小，缓解磁盘紧张的情况，在很多实际场景中，消息的key和value的值之间的对应关系是不断变化的，就像数据库中的数据会不断被修改一样，消费者只关心key对应的最新的value。因此，我们可以开启kafka的日志压缩功能，服务端会在后台启动启动Cleaner线程池，定期将相同的key进行合并，只保留最新的value值。日志的压缩原理是

# 磁盘存储的性能问题

## 磁盘存储的性能优化

我们现在大部分企业仍然用的是机械结构的磁盘，如果把消息以随机的方式写入到磁盘，那么磁盘首先要做的就是寻址，也就是定位到数据所在的物理地址，在磁盘上就要找到对应的柱面、磁头以及对应的扇区；这个过程相对内存来说会消耗大量时间，为了规避随机读写带来的时间消耗，kafka采用顺序写的方式存储数据。即使是这样，但是频繁的I/O操作仍然会造成磁盘的性能瓶颈

## 零拷贝

消息从发送到落地保存，broker维护的消息日志本身就是文件目录，每个文件都是二进制保存，生产者和消费者使用相同的格式来处理。在消费者获取消息时，服务器先从硬盘读取数据到内存，然后把内存中的数据原封不动的通过socket发送给消费者。虽然这个操作描述起来很简单，但实际上经历了很多步骤。操作系统将数据从磁盘读入到内核空间的页缓存·应用程序将数据从内核空间读入到用户空间缓存中

- 应用程序将数据写回到内核空间到socket缓存中
  - 操作系统将数据从socket缓冲区复制到网卡缓冲区，以便将数据经网络发出
- 通过“零拷贝”技术，可以去掉这些没必要的数据复制操作，同时也会减少上下文切换次数。现代的unix操作系统提供一个优化的代码路径，用于将数据从页缓存传输到socket；**在Linux中，是通过sendfile系统调用完成的。**Java提供了访问这个系统调用的方法：FileChannel.transferTo API使用sendfile，只需要一次拷贝就行，**允许操作系统将数据直接从页缓存发送到网络上。**所以在这个优化的路径中，只有最后一步将数据拷贝到网卡缓存中是需要的

**页缓存**页缓存是操作系统实现的一种主要的磁盘缓存，但凡设计到缓存的，基本都是为了提升i/o性能，所以页缓存是用来减少磁盘I/O操作的。磁盘高速缓存有两个重要因素：第一，访问磁盘的速度要远低于访问内存的速度，若从处理器L1和L2高速缓存访问则速度更快。第二，数据一旦被访问，就很有可能短时间内再次访问。正是由于基于访问内存比磁盘快的多，所以磁盘的内存缓存将给系统存储性能带来质的飞跃。当一个进程准备读取磁盘上的文件内容时，操作系统会先查看待读取的数据所在的页(page)是否在页缓存(pagecache)中，如果存在（命中）则直接返回数据，从而避免了对物理磁盘的I/O操作；如果没有命中，则操作系统会向磁盘发起读取请求并将读取的数据页存入页缓存，之后再将数据返回给进程。同样，如果一个进程需要将数据写入磁盘，那么操作系统也会检测数据对应的页是否在页缓存中，如果不存在，则会先在

页缓存中添加相应的页，最后将数据写入对应的页。被修改过后的页也就变成了脏页，操作系统会在合适的时间把脏页中的数据写入磁盘，以**保持数据的一致性**。Kafka中大量使用了**页缓存**，这是Kafka实现高吞吐的重要因素之一。虽然消息都是先被写入页缓存，然后由操作系统负责具体的刷盘任务的，但在Kafka中同样提供了**同步刷盘及间断性强制刷盘(fsync)**，可以通过log.flush.interval.messages 和 log.flush.interval.ms 参数来控制。**同步刷盘能够保证消息的可靠性**，避免因为宕机导致页缓存数据还未完成同步时造成的数据丢失。但是实际使用上，我们没必要去考虑这样的因素以及这种问题带来的损失，消息可靠性可以由多副本解决，同步刷盘会带来性能的影响。**刷盘的操作由操作系统去完成即可**

## Kafka消息的可靠性

没有一个中间件能够做到百分之百的完全可靠，可靠性更多的还是基于几个9的衡量指标，比如4个9、5个9。软件系统的可靠性只能无限接近100%，但不可能达到100%。所以kafka如何是实现最大可能的可靠性呢？

分区副本，你可以创建更多的分区来提升可靠性，但是分区数过多也会带来性能上的开销，一般来说，3个副本就能满足对大部分场景的可靠性要求acks，生产者发送消息的可靠性，也就是我要保证我这个消息一定是到了broker并且完成了多副本的持久化，但这种要求也同样会带来性能上的开销。它有几个可选项

1，生产者把消息发送到leader副本，leader副本在成功写入到本地日志之后就告诉生产者消息提交成功，但是如果ISR集合中的follower副本还没来得及同步leader副本的消息，leader挂了，就会造成消息丢失-1，消息不仅仅写入到leader副本，并且被ISR集合中所有副本同步完成之后才告诉生产者已经提交成功，这个时候即使leader副本挂了也不会造成数据丢失。

0：表示producer不需要等待broker的消息确认。这个选项时延最小但同时风险最大（因为当server宕机时，数据将会丢失）。保障消息到了broker之后，消费者也需要有一定的保证，因为消费者也可能出现某些问题导致消息没有消费到

kafka服务器：自定义ISR副本数，与底层页缓存机制保护

**消费端**enable.auto.commit默认为true，也就是自动提交offset，自动提交是批量执行的，有一个时间窗口，这种方式会带来重复提交或者消息丢失的问题，所以对于高可靠性要求的程序，**要使用手动提交。对于高可靠要求的应用来说，宁愿重复消费也不应该因为消费异常而导致消息丢失**

**生产者端：**使用带回调方法的API / **acks=all(指定多少分区收到确认)** / **retries=MAX(指定重试次数)**

## 应用场景

**行为跟踪：**kafka可以用于跟踪用户浏览页面、搜索及其他行为。通过发布-订阅模式实时记录到对应的topic中，通过后端大数据平台接入处理分析，并做更进一步的实时处理和监控

**日志收集：**日志收集方面，有很多比较优秀的产品。对分布式系统统一日志管理；很多公司的套路都是把应用日志集中到kafka上，然后分别导入到es和hdfs上，用来做实时检索分析和离线统计数据备份等

## RabbitMQ

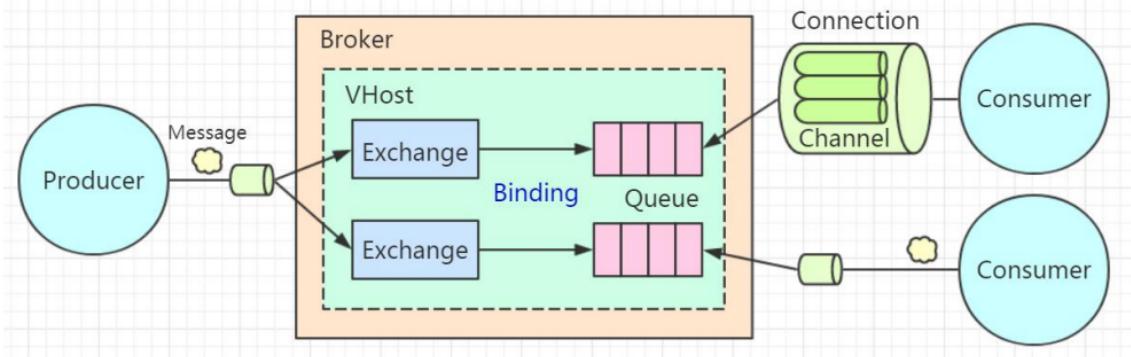
### 概述

AMQP协议应用层协议，可以做到跨语言跨平台，主要用作**实现异步通信**，**实现系统解耦**，**实现流量削峰**。



RabbitMQ用Erlang语言编写，实现AMQP

### 工作模型



## Broker

要使用 RabbitMQ 来收发消息，必须要安装一个 RabbitMQ 的服务，可以安装在 Windows 上面也可以安装在 Linux 上面，默认是 5672 的端口。这台 RabbitMQ 的服务器我们把它叫做 Broker，中文翻译是代理/中介，因为 MQ 服务器帮助我们做的事情就是**存储、转发消息**。

## Connection

无论是生产者发送消息，还是消费者接收消息，都必须要跟 Broker 之间建立一个连接，这个**连接是一个 TCP 的长连接**。

## Channel

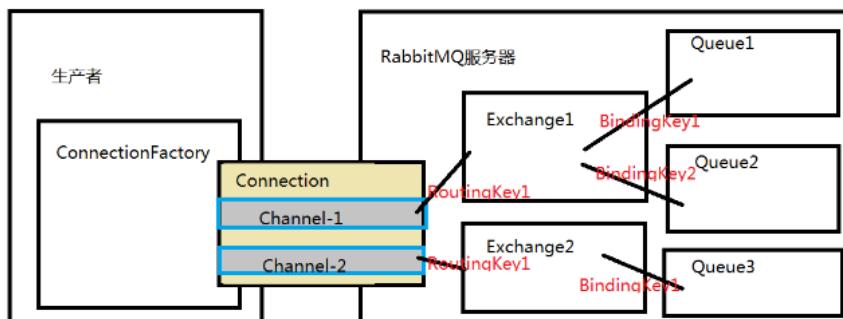
信道，在长连接开辟的，创建和释放 Channel，大大减少了资源消耗

## Queue

消息队列，真正储存消息的地方，FIFO 的特性先进先出

## Exchange

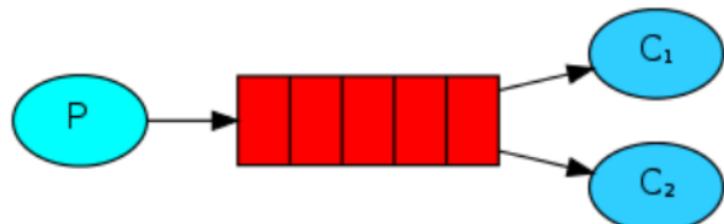
在 RabbitMQ 里面永远不会出现消息直接发送到队列的情况，引入了交换机（Exchange）的概念，用来实现消息的灵活路由



## Vhost

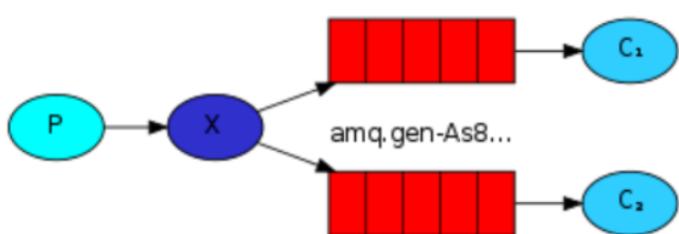
默认 /，可以隔离出多个，设置用户权限，多个共存，类似 docker 虚拟化

hello



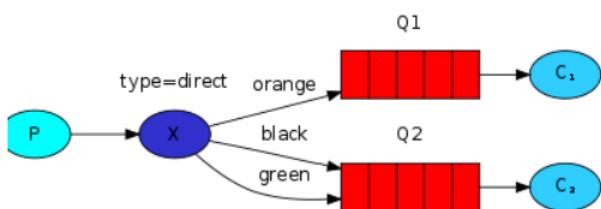
[https://blog.csdn.net/weixin\\_40877204](https://blog.csdn.net/weixin_40877204)

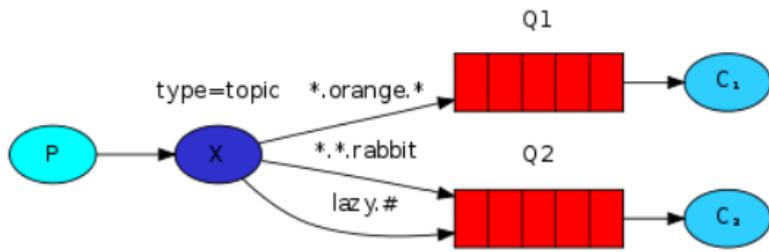
amq.gen-RQ6...



[https://blog.csdn.net/weixin\\_40877204](https://blog.csdn.net/weixin_40877204)

Q1



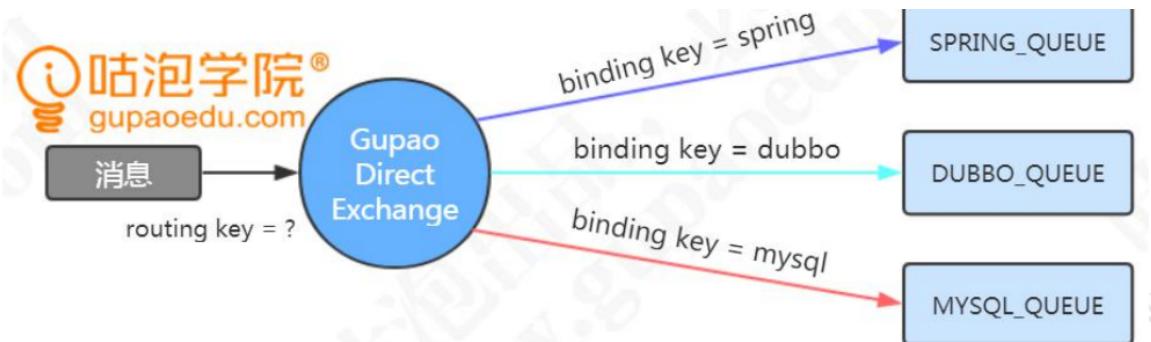


## 交换机类型

1. Direct exchange (直连交换机)
2. Fanout exchange (广播交换机)
3. Topic exchange (主题交换机)
4. Headers exchange (头交换机)

### 直连 Direct

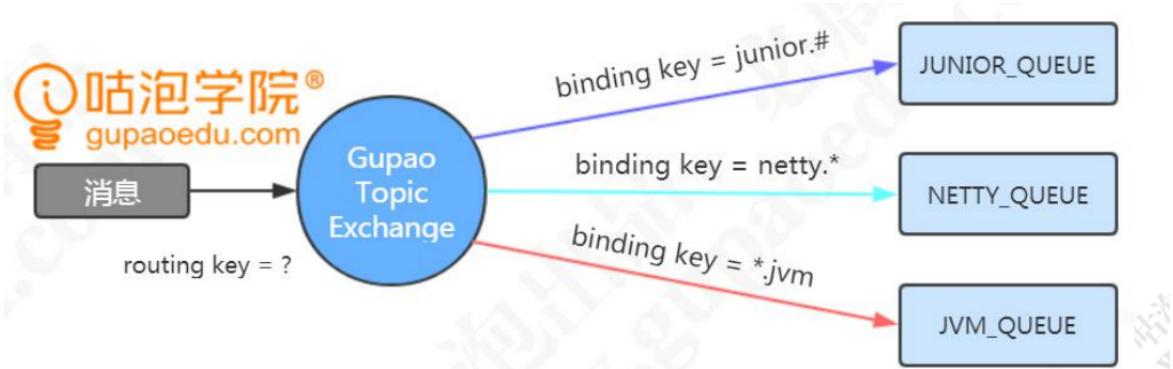
队列与直连类型的交换机绑定，需指定一个精确的绑定键



### 主题 Topic

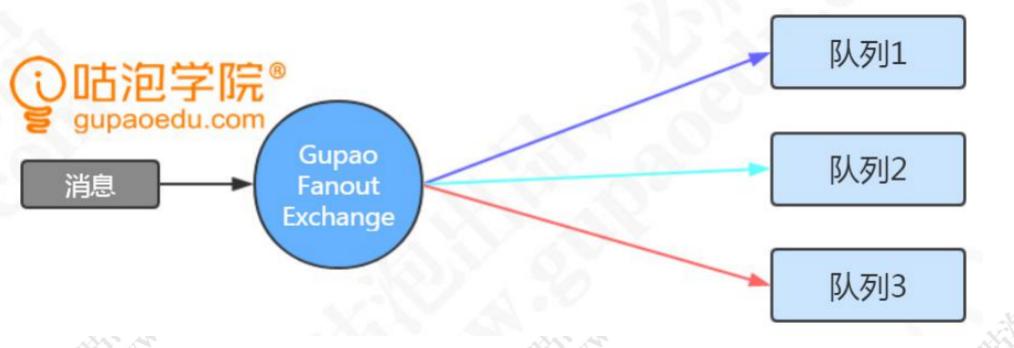
队列与主题类型的交换机绑定时，可以在绑定键中使用通配符。两个通配符：  
# 0个或者多个单词, \*不多不少一个单词

单词 (word) 指的是用英文的点“.”隔开的字符。例如 abc.def 是两个单词。



## 广播 Fanout

主题类型的交换机与队列绑定时，不需要指定绑定键



## 安装

<https://blog.csdn.net/spl545056/article/details/81392015>

## RabbitMQ 进阶知识

### TTL(Time To Live) 消息的过期时间

```
messageProperties.setExpiration("4000"); // 消息的过期属性，单位 ms
```

```
@Bean("ttlQueue")
```

如果同时指定了 Message TTL 和 Queue TTL，则小的那个时间生效

## 死信队列

队列在创建的时候可以指定一个死信交换机 DLX (Dead Letter Exchange), 死信交换机绑定的队列被称为死信队列 DLQ (Dead Letter Queue), DLX 实际上也是普通的交换机, DLQ 也是普通的队列 (例如替补球员也是普通球员)

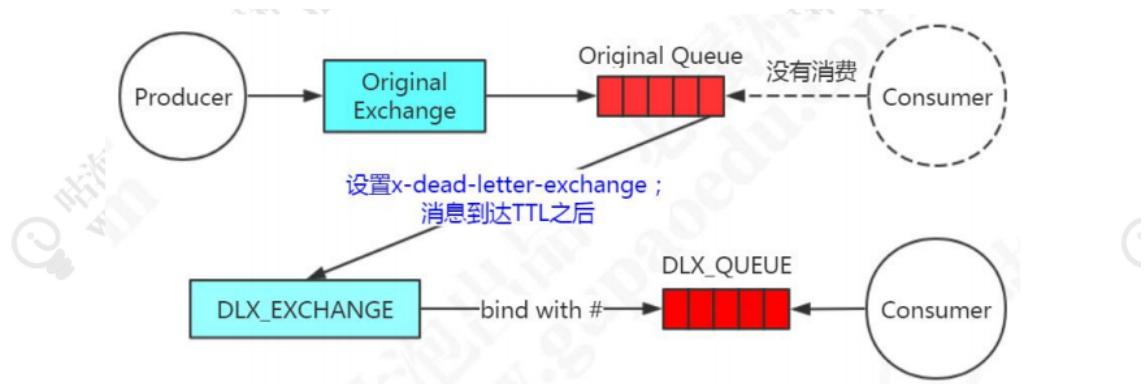
- 1) 消息被消费者拒绝并且未设置重回队列: (NACK || Reject ) && requeue == false

## 2) 消息过期

3) 队列达到最大长度，超过了 Max length (消息数) 或者 Max length bytes (字节数)，最先入队的消息会被发送到 DLX。

## 死信队列如何使用？

- 1、声明原交换机 (GP\_ORI\_USE\_EXCHANGE) 、原队列 (GP\_ORI\_USE\_QUEUE) 相互绑定,
- 2、声明死信交换机 (GP\_DEAD\_LETTER\_EXCHANGE) 、死信队列 (GP\_DEAD\_LETTER\_QUEUE) , 相互绑定
- 3、最终消费者监听死信队列
- 4、生产者发送消息。



## 延迟队列

在实际业务中有一些需要延时发送消息的场景，例如：

- 1、家里有一台智能热水器，需要在 30 分钟后启动
- 2、未付款的订单，15 分钟后关闭

RabbitMQ 本身不支持延迟队列，总的来说有三种实现方案：

- 1、先存储到数据库，用定时任务扫描
- 2、利用 RabbitMQ 的死信队列 (Dead Letter Queue) 实现
- 3、利用 rabbitmq-delayed-message-exchange 插件消息的流转流程：

TTL+DLX总体步骤：

- 1) 创建一个交换机
- 2) 创建一个队列，与上述交换机绑定，并且通过属性指定队列的死信交换机。
- 3) 创建一个死信交换机
- 4) 创建一个死信队列
- 5) 将死信交换机绑定到死信队列
- 6) 消费者监听死信队列

生产者——原交换机——原队列（超过 TTL 之后）——死信交换机——死信队列——最终消费者

## 服务端流控

解决生产速度远大于消费速度,消息堆积

x-max-length: 队列中最大存储最大消息数，超过这个数量，队头的消息会被丢弃。

x-max-length-bytes: 队列中存储的最大消息容量（单位 bytes），超过这个容量，队头的消息会被丢弃

ps:设置队列长度只在消息堆积的情况下有意义，而且会删除先入队的消息，不能真正地实现服务端限流

.**内存控制** ,默认物理内存40%阻塞连接,可修改[{"rabbit", [{"vm\_memory\_high\_watermark, 0.4}]}].

.**磁盘控制** 磁盘空间小于50M,disk\_free\_limit.relative = 3.0  
disk\_free\_limit.absolute = 2GB

## 消费端限流

消费速度跟不上,消费者会在本地缓存消息，如果消息数量过多，可能会导致 OOM 或者影响其他进程的正常运行。

**消费端限流**基于 Consumer 或者 channel 设置 prefetch count 的值，含义为 Consumer 端的最大的 unacked messages 数目。当超过这个数值的消息未被确认，RabbitMQ 会停止投递新的消息给该消费者。

```
1 channel.basicQos(2); // 如果超过 2 条消息没有发送 ACK, 当前消费者不再接受队列消息
2 channel.basicConsume(QUEUE_NAME, false, consumer);
3
4 SimpleMessageListenerContainer
5 container.setPrefetchCount(2);
6
7 Spring Boot 配置:
8 spring.rabbitmq.listener.simple.prefetch=2
```

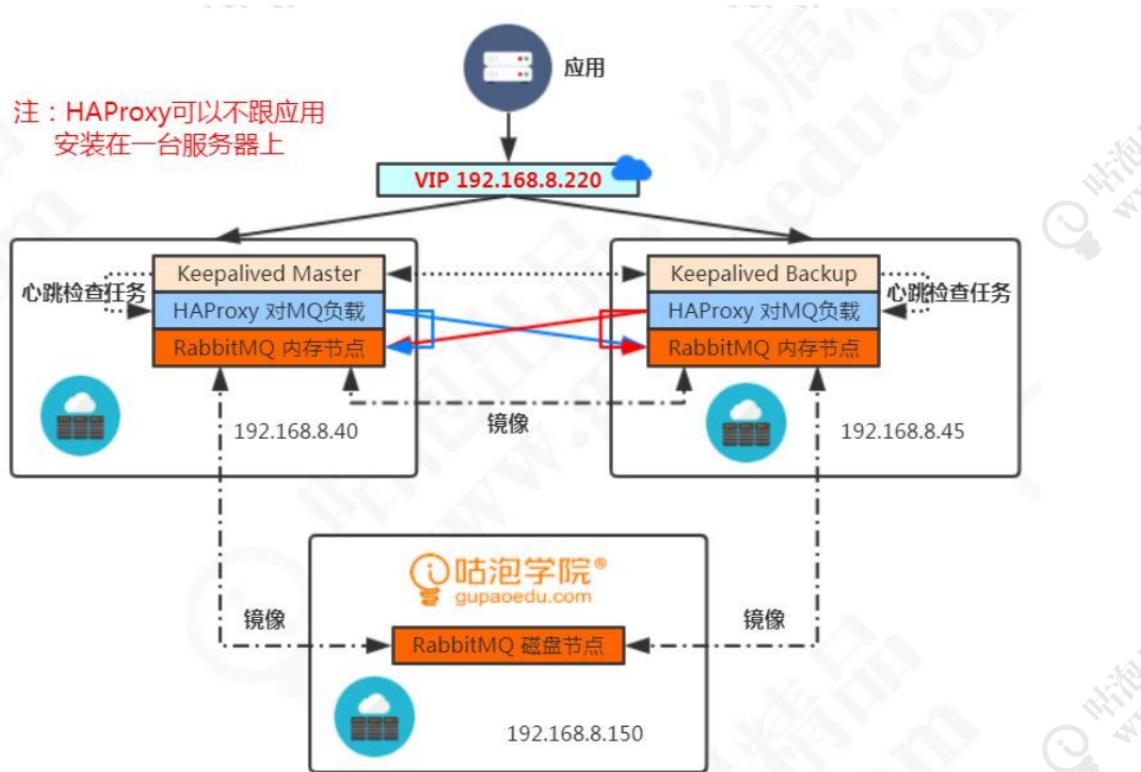
## Spring AMQP

# 集群和高可用

RabbitMQ 有两种集群模式：普通集群模式(相互同步元数据)和镜像队列模式(在镜像节点间同步，可用性更高)

高可用:用到这个协议叫做 VRRP 协议 (虚拟路由冗余协议)

## 基于 Docker 安装 HAProxy 负载+Keepalived 高可用



规划:

内存节点 1: 192.168.8.40

内存节点 2: 192.168.8.45

磁盘节点: 192.168.8.150

VIP: 192.168.8.220

1、我们规划了两个内存节点，一个磁盘节点。所有的节点之间通过镜像队列的方式同步数据。内存节点用来给应用访问，磁盘节点用来持久化数据。

2、为了实现对两个内存节点的负载，我们安装了两个 HAProxy，监听两个 5672 和 15672 的端口

3、安装两个 Keepalived，一主一备。两个 Keepalived 抢占一个

VIP192.168.8.220。谁抢占到这个 VIP，应用就连接到谁，来执行对 MQ 的负载。这种情况下，我们的 Keepalived 挂了一个节点，没有影响，因为 BACKUP 会变成 MASTER，抢占 VIP。HAProxy 挂了一个节点，没有影响，我们的 VIP 会自动路由的可用的 HAProxy 服务。RabbitMQ 挂了一个节点，没有影响，因为 HAProxy 会自动负载到可用的节点。

## 使用建议

RabbitMQ 可以通过 Firehose 功能来记录消息流入流出的情况，用于调试，排错

建议单条消息不要超过 4M (4096KB)，一次发送的消息数需要合理地控制。

## 如何保证消息队列消费的幂等性

问题:重复消费

解决办法

消费数据为了单纯的写入数据库，可以先根据主键查询数据是否已经存在，利用主键的唯一性来保证数

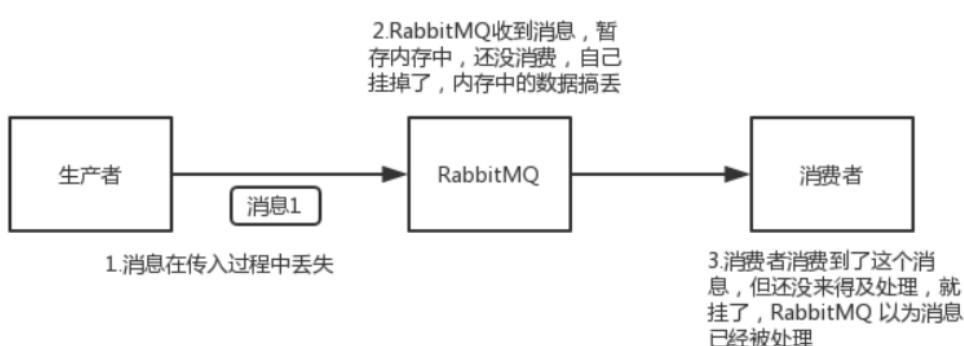
消费数据只是为了缓存到redis当中，这种情况就是直接往redis中set value了，天然的幂等性。

针对复杂的业务情况，可以在生产消息的时候给每个消息加一个全局唯一ID，消费者消费消息时根据这个ID去redis当中查询之前是否消费过。如果没有消费过，就进行消费并将这个消息的ID写入到redis当中。如果已经消费过了，就无需再次消费了。

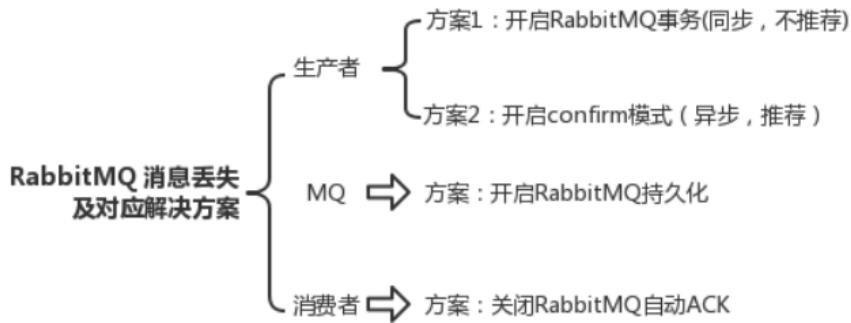
## 如何保证消息的可靠性

问题:消息丢失

### RabbitMQ 消息丢失的 3 种情况



## 解决方案:



异步确认模式需要添加一个 ConfirmListener，并且用一个 SortedSet 来维护没有被确认的消息

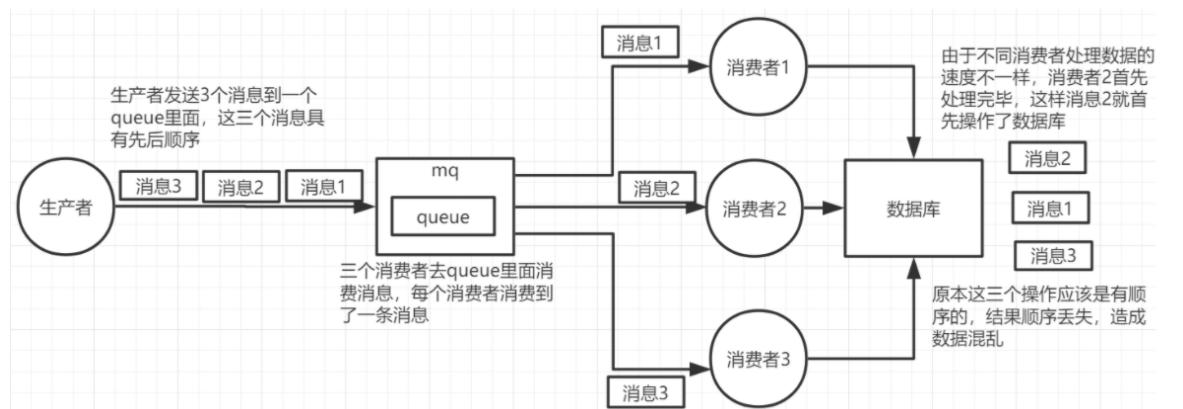
消费者消费可以掉生产者api操作,补偿操作,限制时间和频次

## 如何保证消息的顺序性

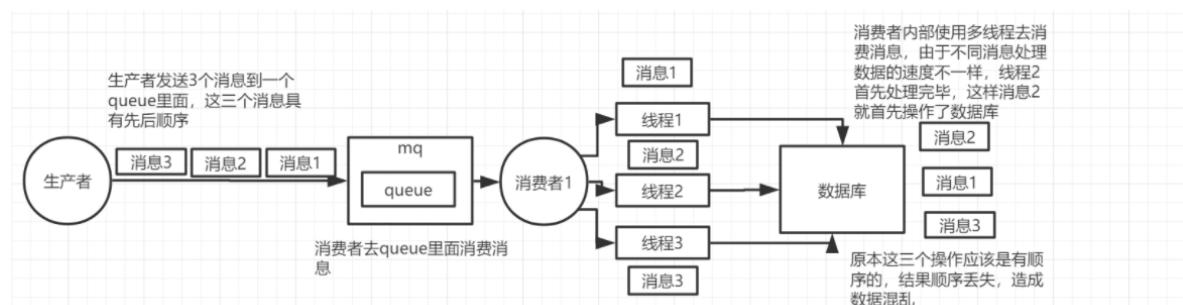
问题:出现消费顺序错乱的情况

只有一个队列仅有一个消费者的情况才能保证顺序消费

p1,为了提高处理效率,一个queue存在多个consumer

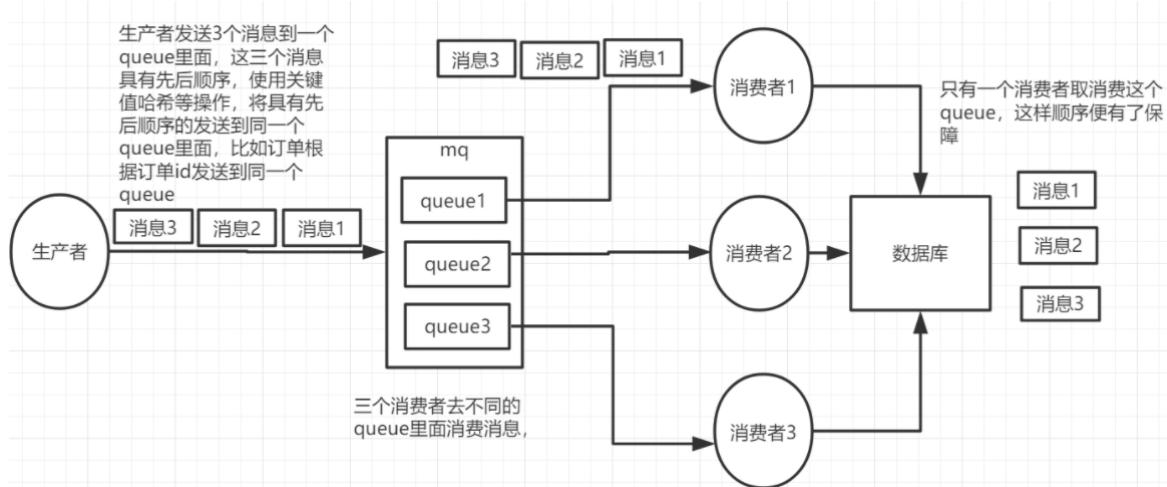


p2一个queue只存在一个consumer,但是为了提高处理效率, consumer中使用了多线程进行处理

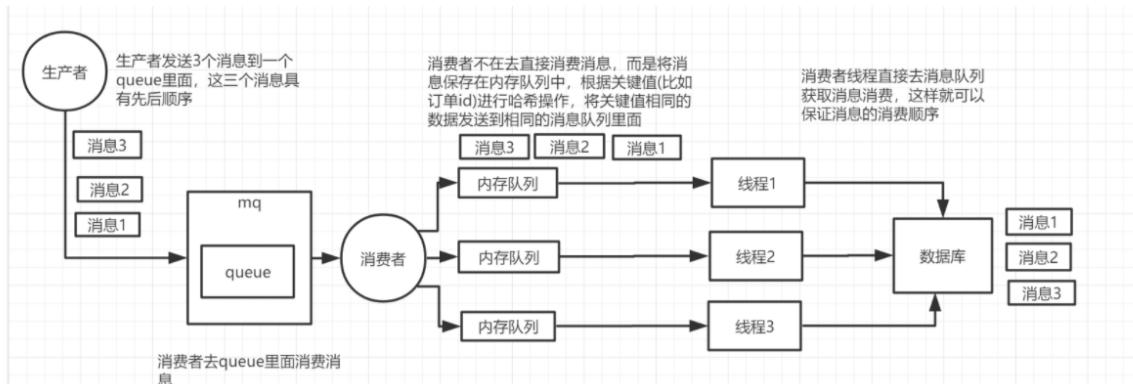


## 解决方案:

key1:将原来的一个queue拆分成多个queue，每个queue都有一个自己的consumer。该种方案的核心是生产者在投递消息的时候**根据业务数据关键值（例如订单ID哈希值对订单队列数取模）来将需要保证先后顺序的同一类数据（同一个订单的数据）发送到同一个queue当中**



key2:一个queue就一个consumer，在consumer中维护**多个内存队列**，根据**业务数据关键值（例如订单ID哈希值对内存队列数取模）**将消息加入到不同的**内存队列中**，然后多个真正负责处理消息的线程去各自对应的内存队列当中获取消息进行消费。



## RocketMQ

阿里捐献给Apache基金会,构思源自于kafka,

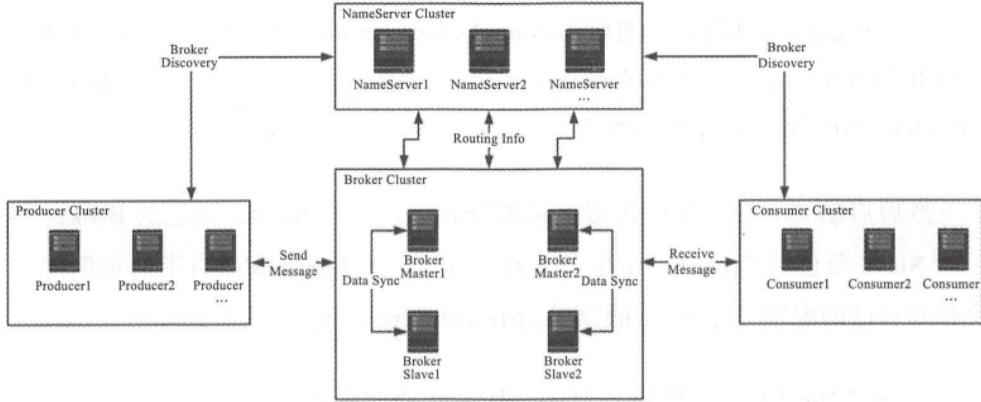


图 2-1 RocketMQ 各个角色间关系

RocketMQ由四部分组成

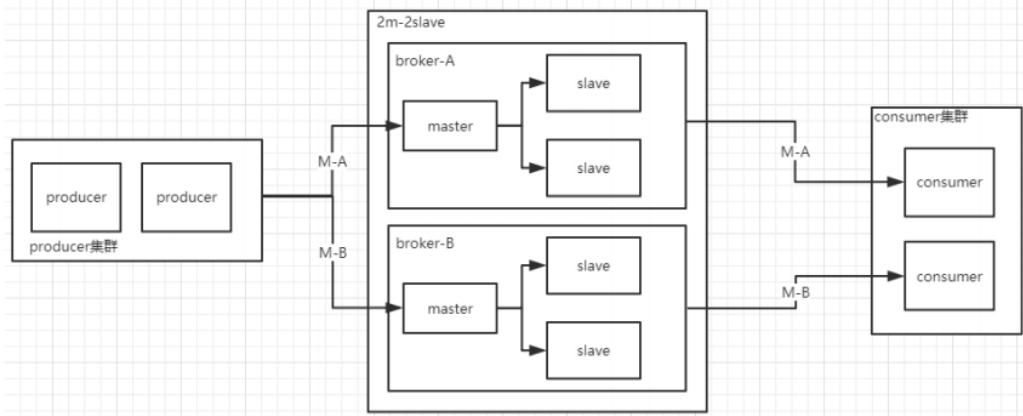
- 1) **Name Server**(替代zk) 可集群部署，节点之间无任何信息同步。提供轻量级的服务发现和路由
- 2) **Broker**(消息中转角色，负责存储消息，转发消息) 部署相对复杂，Broker 分为Master 与Slave，一个Master 可以对应多个Slave，但是一个Slave 只能对应一个Master，Master 与Slave 的对应关系通过 指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0 表示Master，非0 表示Slave。 Master 也可以部署多个。
- 3) **Producer**，生产者，拥有相同 Producer Group 的 Producer 组成一个集群，与Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 取Topic 路由信息，并向提供Topic服务的Master 建立长连接，且定时向Master 发送心跳。Producer 完全无状态，可集群部署。
- 4) **Consumer**，消费者，接收消息进行消费的实例，拥有相同 Consumer Group 的 Consumer 组成一个集群，与Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从Name Server 取 Topic 路由信息，并向提供 Topic 服务的Master、Slave 建立长连接，且定时向Master、Slave 发送心跳。Consumer既可以从Master 订阅消息，也可以从Slave 订阅消息，订阅规则由 Broker 配置决定。要使用rocketmq，至少需要启动两个进程，nameserver、broker，前者是各种topic注册中心，后者 是真正的broker

### rocketmq控制台安装

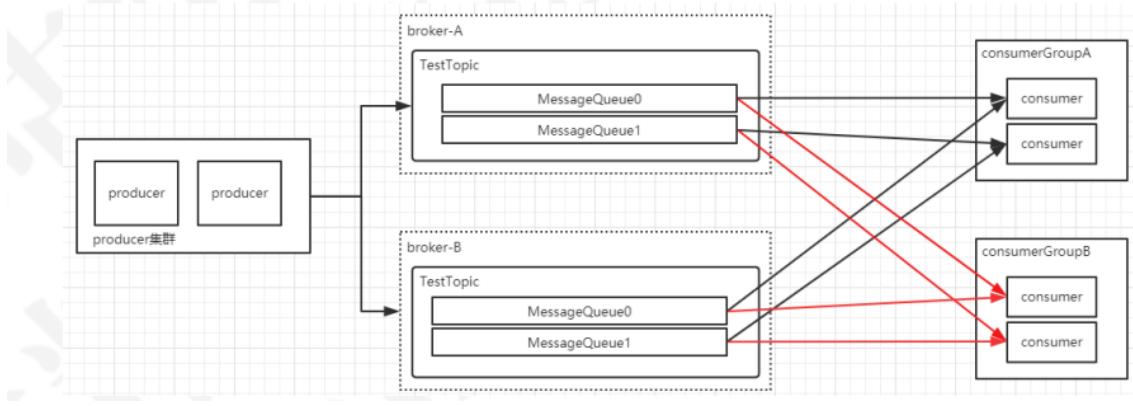
在RocketMQ中，是基于多个Message Queue来实现类似于kafka的效果。如果一个Topic 要发送 和接收的数据量非常大， 需要能支持增加并行处理的机器来提高处理速度，这时候一个Topic 可以根据需求设置一个或多个 Message Queue。Topic 有了多个Message Queue 后，消息可以并行地向各个 Message Queue 发送，消费者也可以并行地从多个Message Queue 读取消息并消费。

# RocketMQ消息发送及消费的基本原理

RocketMQ中并没有master选举功能，所以通过配置多个master节点来保证rocketMQ的高可用。和所有的集群角色定位一样，master节点负责接受事务请求、slave节点只负责接收读请求，并且接收master同步过来的数据和slave保持一致。当master挂了以后，如果当前rocketmq是一主多从，就意味着无法接受发送端的消息，但是消费者仍然能够继续消费。所以配置多个主节点后，可以保证当其中一个master节点挂了，另外一个master节点仍然能够对外提供消息发送服务。当存在多个主节点时，一条消息只会发送到其中一个主节点，rocketmq对于多个master节点的消息发送，会做负载均衡，使得消息可以平衡的发送到多个master节点上。一个消费者可以同时消费多个master节点上的消息，在下面这个架构图中，两个master节点恰好可以平均分发到两个消费者上，如果此时只有一个消费者，那么这个消费者会消费两个master节点的数据。由于每个master可以配置多个slave，所以如果其中一个master挂了，消息仍然可以被消费者从slave节点消费到。可以完美的实现rocketmq消息的高可用



接下来，站在topic的角度来看看消息是如何分发和处理的，假设有两个master节点的集群，创建了一个TestTopic，并且对这个topic创建了两个队列，也就是分区。消费者定义了两个分组，分组的概念也是和kafka一样，通过分组可以实现消息的广播



## 集群支持

RocketMQ天生对集群的支持非常友好

### 1) 单Master

优点：除了配置简单没什么优点

缺点：不可靠，该机器重启或宕机，将导致整个服务不可用

### 2) 多Master

优点：配置简单，性能最高

缺点：可能会有少量消息丢失（配置相关），单台机器重启或宕机期间，该机器下未被消费的消息在机器恢复前不可订阅，影响消息实时性

### 3) 多Master多Slave，每个Master配一个Slave，有多对Master-Slave，集群采用异步复制方式，主备有短暂消息延迟，毫秒级

优点：性能同多Master几乎一样，实时性高，主备间切换对应用透明，不需人工干预

缺点：Master宕机或磁盘损坏时会有少量消息丢失

### 4) 多Master多Slave，每个Master配一个Slave，有多对Master-Slave，集群采用同步双写方式，主备都写成功，向应用返回成功

优点：服务可用性与数据可用性非常高

缺点：性能比异步集群略低，当前版本主宕备不能自动切换为主

需要注意的是，在RocketMQ里面，1台机器只能要么是Master，要么是Slave。这个在**初始的机器配置里面，就定死了**。不会像kafka那样存在master动态选举的功能。其中Master的broker id = 0，Slave 的broker id > 0。

## 无法确保消息顺序性

### 什么时候触发负载均衡

消费者启动之后 消费者数量发生变更 每10秒会触发检查一次rebalance

## 消息的可靠性

RocketMQ提供了ack机制，以保证消息能够被正常消费

## 分布式事务

在上面这种场景中，本地事务无法解决，所以才引入了分布式事务，所谓的分布式事务是指分布式架构中多个服务的节点的数据一致性。

### 经典的X/OpenDTP事务模型

X/Open DTP(X/Open Distributed Transaction Processing Reference Model)是X/Open这个组织定义的一套分布式事务的标准，也就是定义了规范和API接口，由各个厂商进行具体的实现。这个标准提出了使用二阶段提交(2PC – Two-Phase-Commit)来保证分布式事务的完整性。后来J2EE也遵循了X/OpenDTP规范，设计并实现了java里的分布式事务编程接口规范-JTA

### X/OpenDTP角色

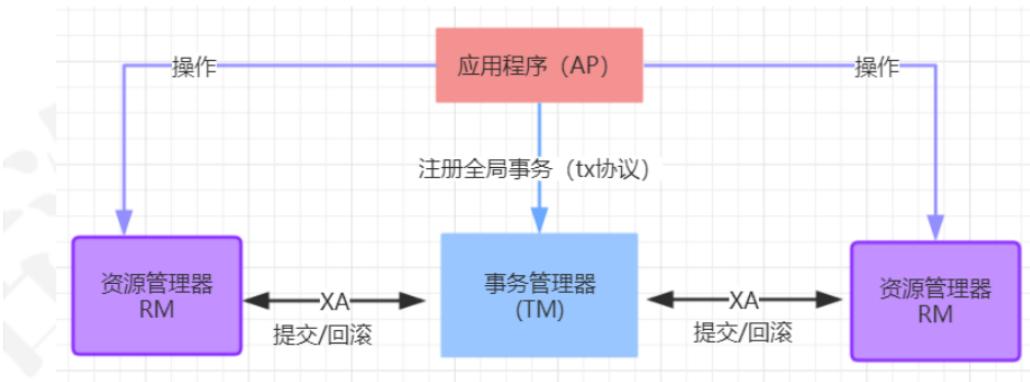
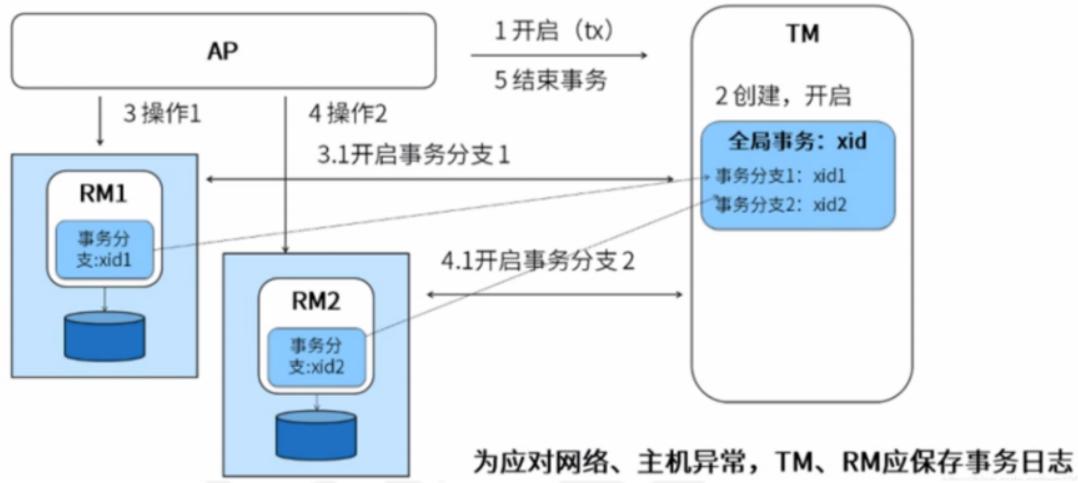
在X/OpenDTP事务模型中，定义了三个角色

AP: application, 应用程序，也就是业务层。哪些操作属于一个事务，就是AP定义的

RM: Resource Manager, 资源管理器。一般是数据库，也可以是其他资源管理器，比如消息队列，文件系统

TM: Transaction Manager , 事务管理器、事务协调者，负责接收来自用户程序 (AP) 发起的XA事务

指令，并调度和协调参与事务的所有RM (数据库)，确保事务正确完成  
在分布式系统中，每一个机器节点虽然都能够明确知道自己在进行事务操作过程中的结果是成功还是失败，但却无法直接获取到其他分布式节点的操作结果。因此当一个事务操作需要跨越多个分布式节点的时候，为了保持事务处理的ACID特性，就需要引入一个“协调者” (TM) 来统一调度所有分布式节点的执行逻辑，这些被调度的分布式节点被称为AP。TM负责调度AP的行为，并最终决定这些AP是否要把事务真正进行提交到 (RM)



## CAP理论

CAP的含义：

C: Consistency 一致性 同一数据的多个副本是否实时相同。

A: Availability 可用性 可用性：一定时间内 & 系统返回一个明确的结果 则称为该系统可用。

P: Partition tolerance 分区容错性 将同一服务分布在多个系统中，从而保证某一个系统宕机，仍然有其他系统提供相同的服务。

CAP理论告诉我们，在分布式系统中，C、A、P三个条件中我们最多只能选择两个。那么问题来了，究竟选择哪两个条件较为合适呢？对于一个业务系统来说，可用性和分区容错性是必须要满足的两个条件，并且这两者是相辅相成的。业务系统之所以使用分布式系统，主要原因有两个：

提升整体性能 当业务量猛增，单个服务器已经无法满足我们的业务需求的时候，就需要使用分布式系统，使用多个节点提供相同的功能，从而整体上提升系统的性能，

实现分区容错性 单一节点 或 多个节点处于相同的网络环境下，那么会存在一定的风险，万一该机房断电、该地区发生自然灾害，那么业务系统就全面瘫痪了。为了防止这一问题，采用分布式系统，将多个子系统分布在不同的地域、不同的机房中，从而保证系统高可用性。

这说明分区容错性是分布式系统的根本，如果分区容错性不能满足，那使用分布式系统将失去意义。此外，可用性对业务系统也尤为重要。在大谈用户体验的今天，如果业务系统时常出现“系统异常”、响应时间过长等情况，这使得用户对系统的好感度大打折扣，在互联网行业竞争激烈的今天，相同领域的竞争者不甚枚举，系统的间歇性不可用会立马导致用户流向竞争对手。因此，我们只能通过牺牲一致性来换取系统的**可用性和分区容错**

## Base理论

“牺牲一致性”并不是完全放弃数据一致性，而是牺牲**强一致性**换取**弱一致性**

BA：Basic Available 基本可用 整个系统在某些不可抗力的情况下，仍然能够保证“可用性”，即一定时间内仍然能够返回一个明确的结果。只不过“基本可用”和“高可用”的区别是：

“一定时间”可以适当延长 当举行大促时，响应时间可以适当延长 给部分用户返回一个降级页面 给部分用户直接返回一个降级页面，从而缓解服务器压力。但要注意，返回降级页面仍然是返回明确结果。

S：Soft State：柔性状态 同一数据的不同副本的状态，可以不需要实时一致。

E：Eventual Consistency：最终一致性 同一数据的不同副本的状态，可以不需要实时一致，但

一定要保证经过一定时间后仍然是一致的。

CP：优先保证一致性和分区容错性，**放弃可用性**。在数据一致性要求比较高的场合(譬如:zookeeper,Hbase)是比较常见的做法，一旦发生网络故障或者消息丢失，就会牺牲用户体验，等恢复之后用户才逐渐能访问。

AP：优先保证可用性和分区容错性，**放弃一致性**。NoSQL中的Cassandra就是这种架构。跟CP一样，放弃一致性不是说一致性就不保证了，而是逐渐的变得一致。**最终一致**

## 消息的存储结构

RocketMQ就是采用文件系统的方式来存储消息，消息的存储是由 ConsumeQueue和CommitLog配合完成的。 CommitLog是消息真正的物理存储文件。 ConsumeQueue是消息的逻辑队列，有点类似于数据库的索引文件，里面存储的是指向CommitLog文件中消息存储的地址。 每个Topic下的每个 Message Queue都会对应一个ConsumeQueue文件，文件的地址是：

`${store_home}/consumequeue/${topicName}/${queueId}/${fileName}`, 默认路径: /root/store

## CommitLog

CommitLog是用来存放消息的物理文件，每个broker上的commitLog本当前机器上的所有 consumerQueue共享，不做任何的区分。

CommitLog中的文件默认大小为1G，可以动态配置；当一个文件写满以后，会生成一个新的commitlog文件。 Topic数据是顺序写入在CommitLog文中的。

文件名的长度为20位，左边补0，剩余未起始偏移量，比如 000000000000000000 表示第一个文件，文件大小为102410241024，当第一个文件写满之后，生成第二个文件00000000001073741824 表示第二个文件，起始偏移量为1073741824

## ConsumeQueue

consumeQueue表示消息消费的逻辑队列，这里面包含MessageQueue在 commitlog中的其实物理位置偏移量offset，消息实体内容的大小和Message Tag的hash值。对于实际物理存储来说，consumeQueue对应每个topic和 queueid下的文件，每个consumeQueue类型的文件也是有大小，每个文件默认大小约为600W个字节，如果文件满了后会也会生成一个新的文件

## IndexFile

索引文件，如果一个消息包含Key值的话，会使用IndexFile存储消息索引。 Index索引文件提供了对CommitLog进行数据检索，提供了一种通过key或者时间区间来查找CommitLog中的消息的方法。在物理存储中，文件名是以创建的时间戳分明，固定的单个IndexFile大小大概为400M，一个IndexFile可以保存2000W个索引

## abort

broker在启动的时候会创建一个空的名为abort的文件，并在shutdown时将其删除，用于标识进程是否正常退出，如果不正常退出，会在启动时做故障恢复

# 清理物理消息文件

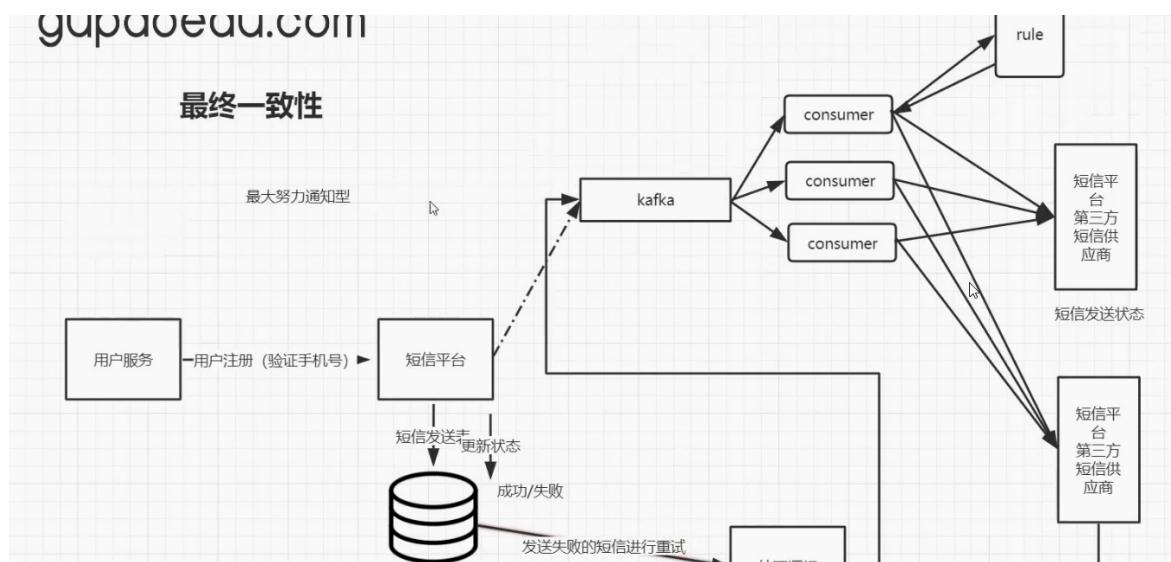
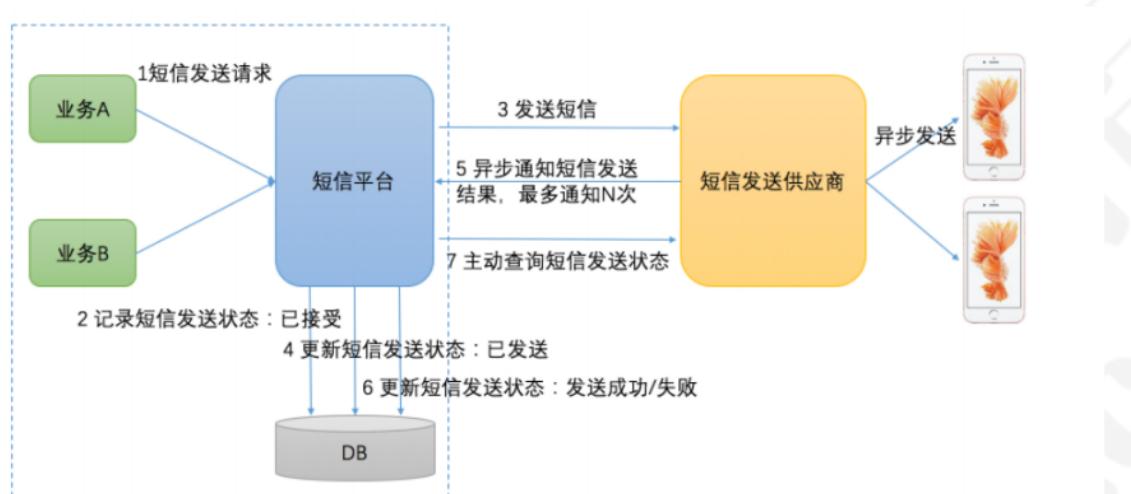
消息存储在CommitLog之后，的确是会被清理的，但是这个清理只会在以下任一条件成立才会批量删除消息文件（CommitLog）：

1. 消息文件过期（默认72小时），且到达清理时点（默认是凌晨4点），删除过期文件。
2. 消息文件过期（默认72小时），且磁盘空间达到了水位线（默认75%），删除过期文件。
3. 磁盘已经达到必须释放的上限（85%水位线）的时候，则开始批量清理文件（无论是否过期），直到空间充足。

注：若磁盘空间达到危险水位线（默认90%），出于保护自身的目的，broker会拒绝写入服务。

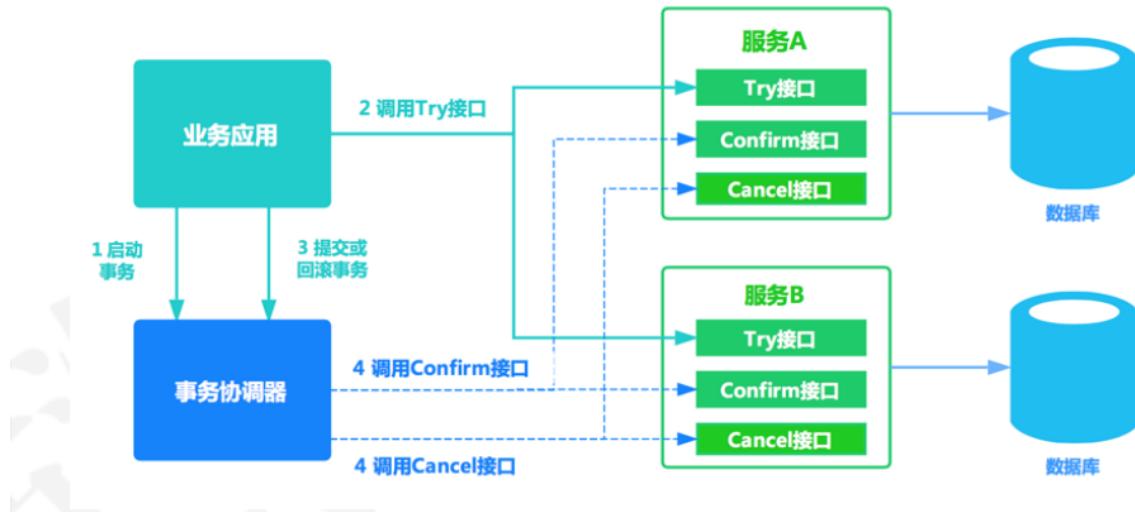
## 分布式事务常见解决方案

### 最大努力通知方案



### TCC两阶段补偿方案(借鉴2pc)

TCC是Try-Confirm-Cancel，比如在支付场景中，先冻结一笔资金，再去发起支付。如果支付成功，则讲冻结资金进行实际扣除；如果支付失败，则取消资金冻结



## Try阶段

完成所有业务检查（一致性），预留业务资源（准隔离性）

## Confirm阶段

确认执行业务操作，不做任何业务检查，只使用Try阶段预留的业务资源。

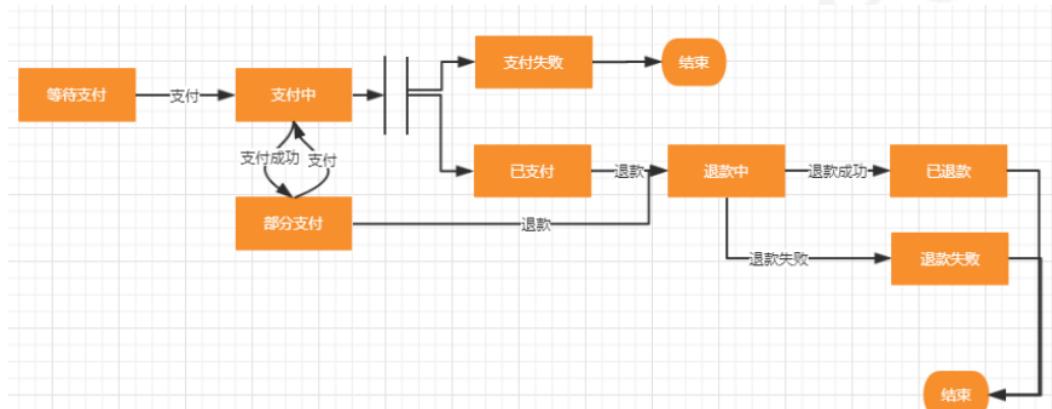
## Cancel阶段

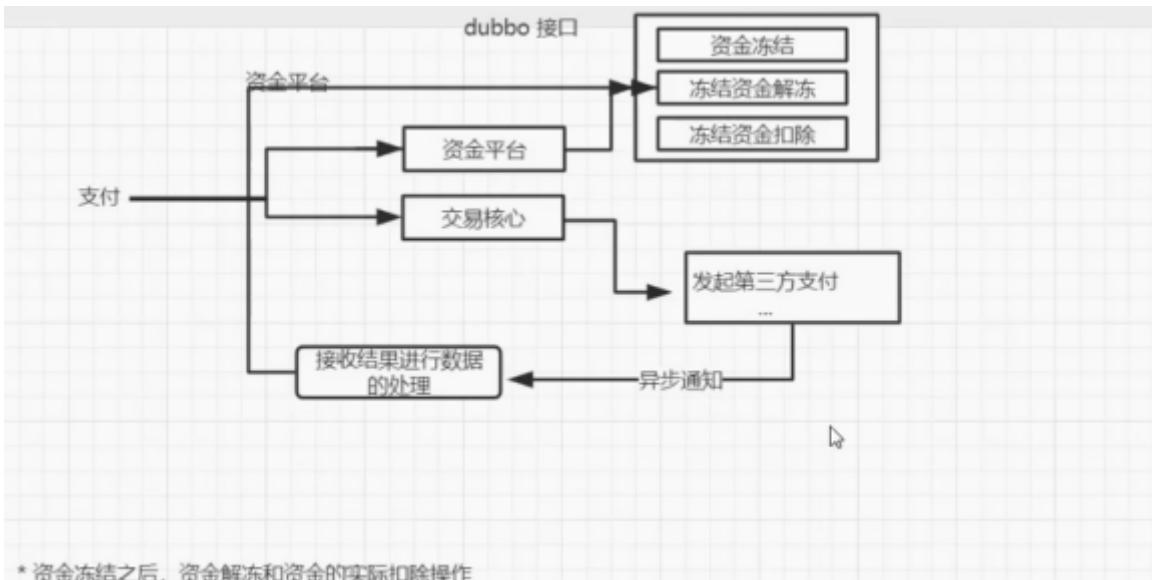
取消Try阶段预留的业务资源。Try阶段出现异常时，取消所有业务资源预留请求

状态机:确保对象随时都知道自己所处的状态以及所能做的操作

## 作用

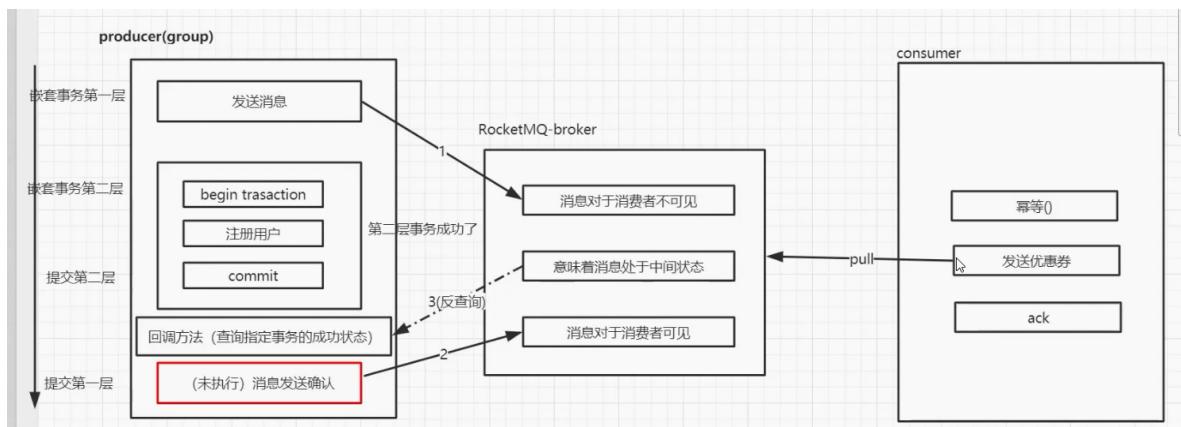
1. 实现幂等 (重复调用多次产生的业务结果与调用一次产生的业务结果相同)
2. 通过状态驱动数据的变化
3. 业务流程以及逻辑更加清晰，特别是应对复杂的业务场景





## 基于可靠性消息队列的最终一致性

### RocketMQ最终一致性

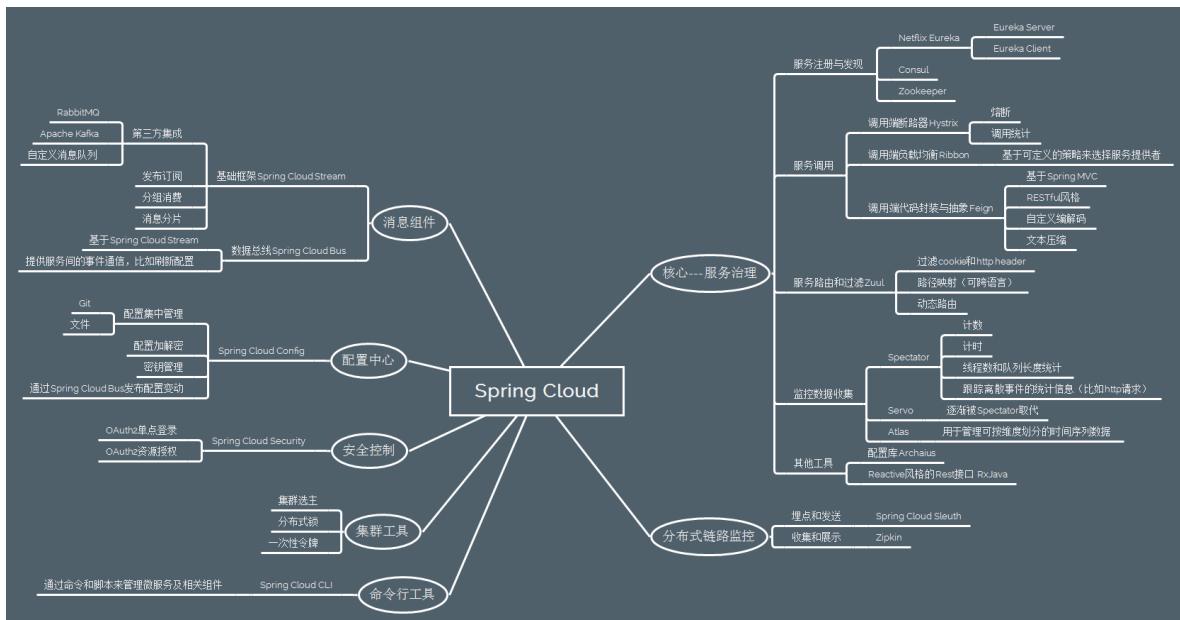
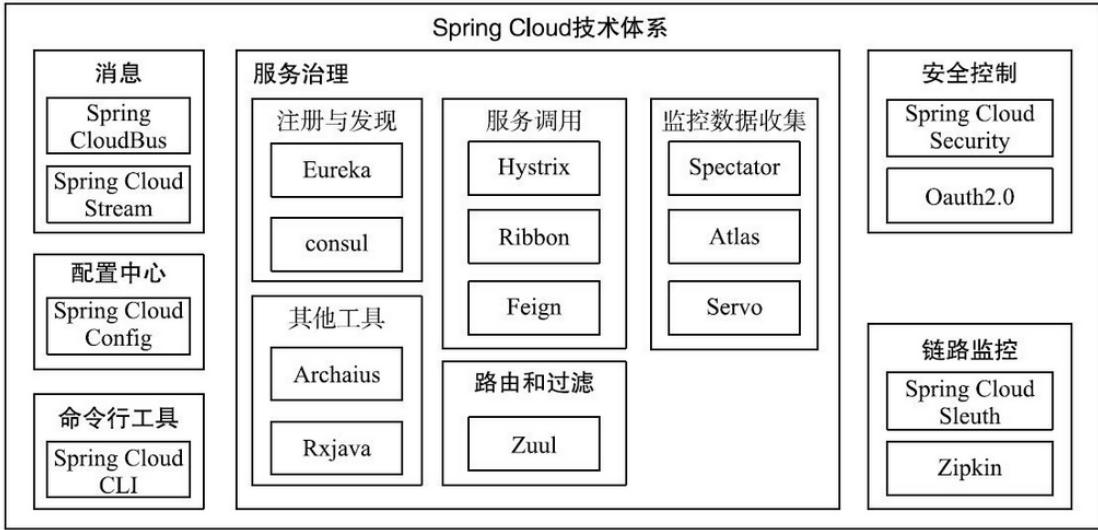


从主流的几种MQ消息队列采用的存储方式来看，主要会有三种

1. 分布式KV存储，比如ActiveMQ中采用的levelDB、Redis，这种存储方式对于消息读写能力要求不高的情况下可以使用
2. 文件系统存储，常见的比如kafka、RocketMQ、RabbitMQ都是采用消息刷盘到所部署的机器上的文件系统来做持久化，这种方案适合对于有高吞吐量要求的消息中间件，因为消息刷盘是一种高效率，高可靠、高性能的持久化方式，除非磁盘出现故障，否则一般是不会出现无法持久化的问题
3. 关系型数据库，比如ActiveMQ可以采用mysql作为消息存储，关系型数据库在单表数据量达到千万级的情况下IO性能会出现瓶颈，所以ActiveMQ并不适合于高吞吐量的消息队列场景。

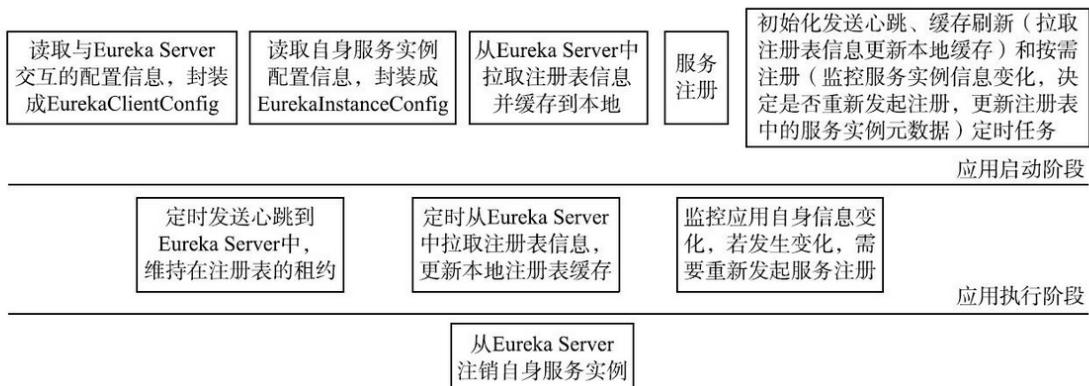
总的来说，对于存储效率，文件系统要优于分布式KV存储，分布式KV存储要优于关系型数据库

# Spring-cloud



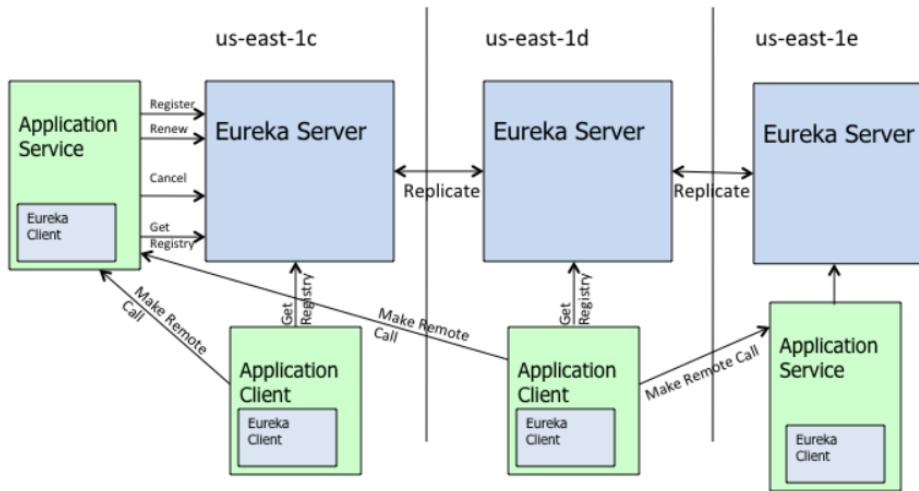
## Eureka介绍和部署

客户端运行过程



作为注册中心使用,服务端提供监控页面, 客户端就是consumer和provide

## 集群架构



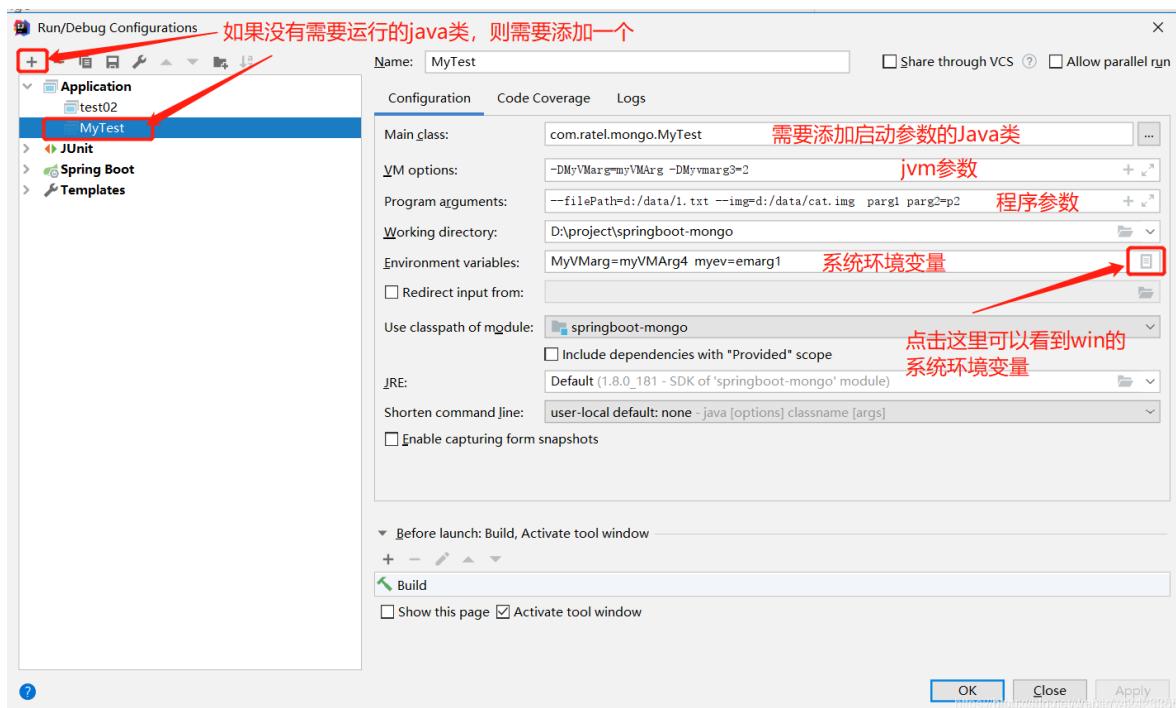
客户端初始化40秒后注册到服务端,发送自己的相关配置,第一次获取全部信息并缓存到本地,每30秒保持心跳一次,90未收到会被服务端剔除

服务端内容全部多点备份,挂掉的服务恢复的时候会从最近的节点全量获取

集群中Eureka server采用同步复制,每个节点一段时间同步一次,采用定时批量任务减少网络开销,**属于弱一致性**

<http://www.heartthinkdo.com/?p=1933>

多实例启动



	Nacos	Eureka	Consul	CoreDNS	Zookeeper
一致性协议	CP+AP	AP	CP	—	CP
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	—	Keep Alive
负载均衡策略	权重/metadata/Selector	Ribbon	Fabio	RoundRobin	—
雪崩保护	有	有	无	无	无
自动注销实例	支持	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心同步	支持	不支持	支持	不支持	不支持
SpringCloud集成	支持	支持	支持	不支持	支持
Dubbo集成	支持	不支持	支持	不支持	支持
K8S集成	支持	不支持	支持	支持	不支持

## Spring-could Open Feigh

<https://blog.csdn.net/chengqiuming/article/details/80713471>

Feign是Netflix开发的声明式、模板化的HTTP客户端， Feign可以帮助我们更快捷、优雅地调用HTTP API。

Spring Cloud Feign定义和实现依赖服务接口的定义。在Spring Cloud feign的实现下，只需要创建一个接口并用注解方式配置它，即可完成服务提供方的接口绑定，简化了在使用Spring Cloud Ribbon时自行封装服务调用客户端的开发量。Spring Cloud Feign具备可插拔的注解支持，支持Feign注解、JAX-RS注解和Spring MVC的注解

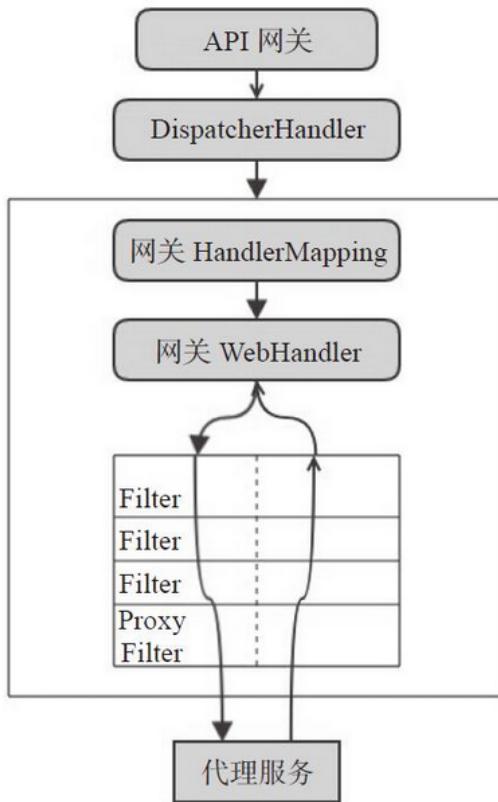
REST 框架	使用场景	请求映射注解	请求参数
Feign	客户端声明	@RequestLine	@Param
Spring Cloud Open Feign	客户端声明	@RequestMapping	@RequestParam
JAX-RS	客户端、服务端声明	@Path	@PathParam
Spring Web MVC	服务端声明	@RequestMapping	@RequestParam

## hystrix 熔断

<https://github.com/Netflix/Hystrix/wiki/Configuration>

限流原理基于线程池与信号量

# zuul 网关



## 服务器端负载均衡 Nginx

nginx 是客户端所有请求统一交给 nginx，由 nginx 进行实现负载均衡请求转发，属于服务器端负载均衡。

既请求由 nginx 服务器端进行转发。

## 客户端负载均衡 Ribbon

Ribbon 是从 eureka 注册中心服务器端上获取服务注册信息列表，缓存到本地，然后在本地实现轮询负载均衡策略。

既在客户端实现负载均衡。

### 1 | 应用场景的区别：

(1) Nginx适合于服务器端实现负载均衡比如 Tomcat，Ribbon适合与在微服务中RPC远程调用实现本地服务负载均衡，比如 Dubbo、SpringCloud 中都是采用本地负载均衡。

spring cloud的Netflix中提供了两个组件实现软负载均衡调用：ribbon和feign。

### (2) Ribbon

是一个基于 HTTP 和 TCP 客户端的负载均衡器

它可以在客户端配置 ribbonServerList (服务端列表) , 然后轮询请求以实现均衡负载。

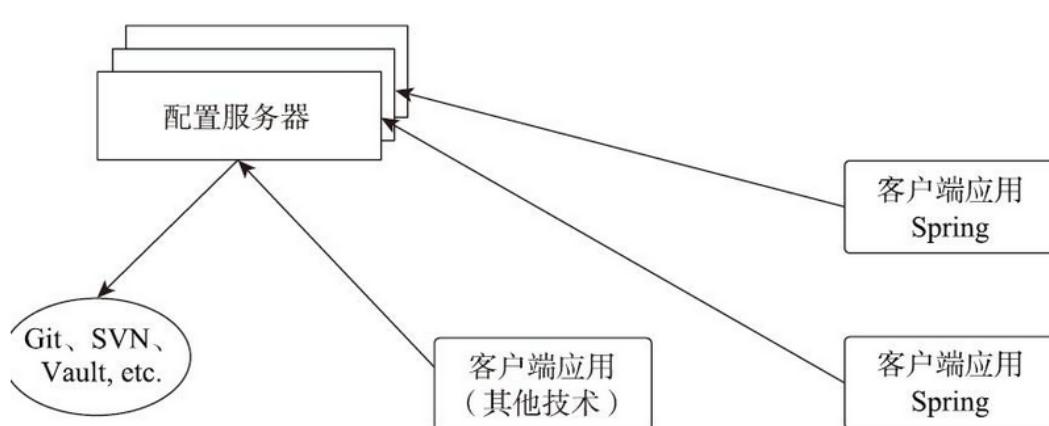
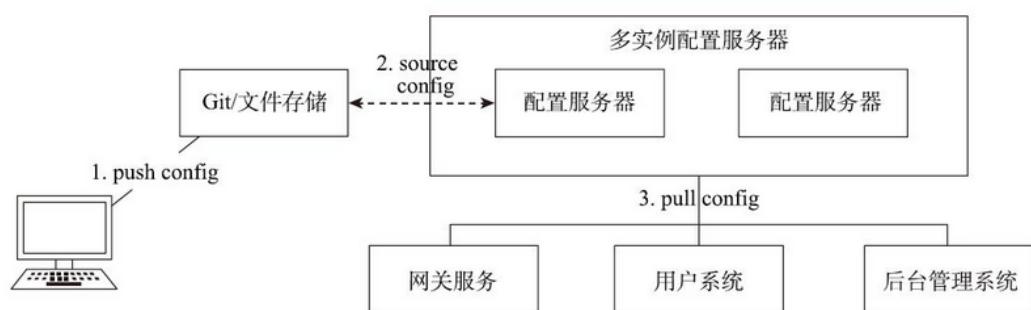
### 1 | springcloud的ribbon和nginx有什么区别？哪个性能好？

nginx性能好，但ribbon可以剔除不健康节点，nginx剔除节点比较复杂。  
ribbon还可以配合熔断器一起工作

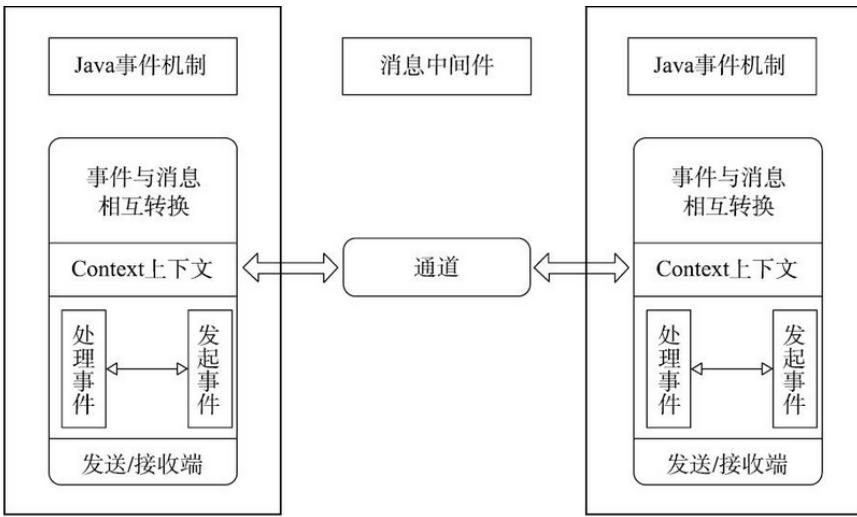
ribbon是客户端负载均衡，nginx是服务端负载均衡。客户端负载均衡，所有客户端节点都维护自己要访问的服务端清单。服务端负载均衡的软件模块会维护一个可用的服务清单

ribbon 是一个客户端负载均衡器，可以简单的理解成类似于 nginx的负载均衡模块的功能。

## config 配置中心



## bus



## Security

### Spring Security 对 Servlet 实现

传统 Servlet 3.0+ 容器实现 Web 自动装配实现 -

AbstractSecurityWebApplicationInitializer Spring Web  
WebApplicationInitializer

小知识：Servlet 3.0+ javax.servlet.ServletContainerInitializer 所有它的实现配置在 META-INF/services/javax.servlet.ServletContainerInitializer 后，均会被 Servlet 3.0+ 容器回调方法 -

javax.servlet.ServletContainerInitializer#onStartup 代表实现类 -

SpringServletContainerInitializer 当实现类标注

@javax.servlet.annotation.HandlesTypes 时，并且制定类的范围，被指定类的所有派生类会出现在onStartup 方法的首参 - Set<Class<?>> classes

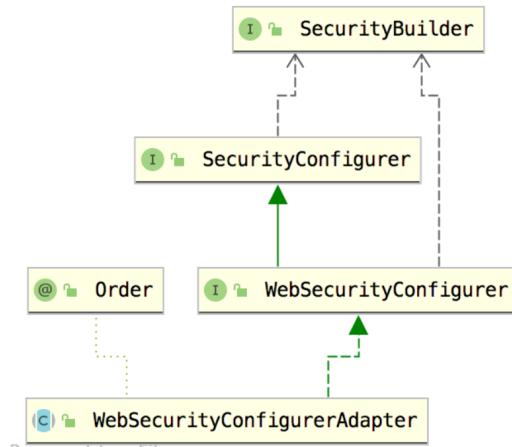
Spring Security 利用 Servlet 3.0+ Web 自动装配能力来引导 Root ApplicationContext 装配，并且将一个名为 "springSecurityFilterChain" 动态地（编码）加入（注册）到ServletContext 中

小知识："springSecurityFilterChain" 使通过 org.springframework.web.filter.DelegatingFilterProxy 动态读取，并且 DelegatingFilterProxy 是一个 Filter，它通过配置形成获取 Spring 应用上下文中的 Filter Bean

执行逻辑

AbstractSecurityWebApplicationInitializer -> DelegatingFilterProxy -> "springSecurityFilterChain" Filter

springSecurityFilterChain简单实现,注入 List<SecurityConfigifurer<Filter, WebSecurity>> SecurityConfigifurers



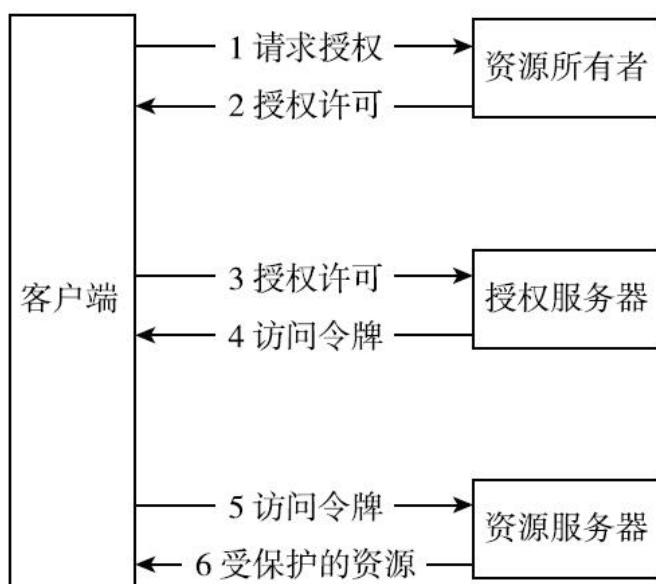
自定义操作是继承WebSecurityConfigurerAdapter,重写三个config方法

WebMvcConfigurer,sping5后推荐使用,

- 1 里面每个属性都有对应的类形成过滤链,会排序,多个重复链条会被覆盖掉
- 2 AuthenticationManagerBuilder
- 3 HttpSecurity
- 4 webSecurity

## Oauth2

OAuth2的认证流程图如图12-1所示。



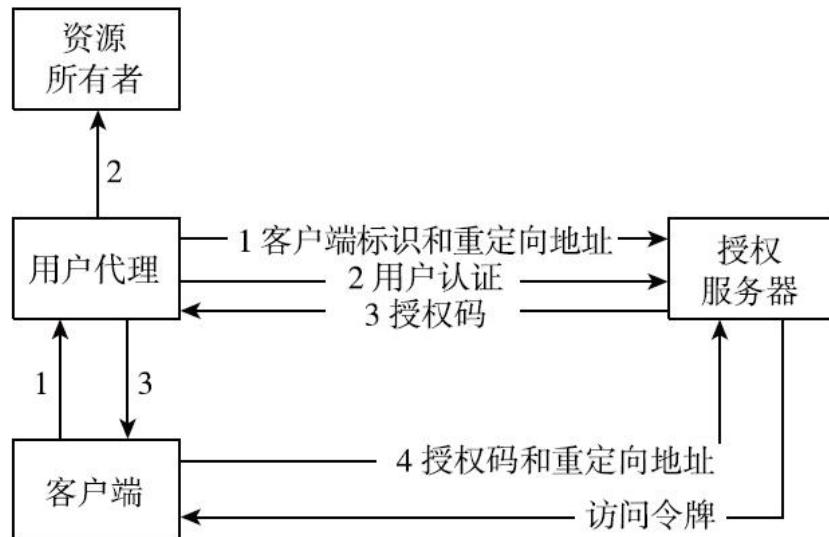


图12-2 授权码类型流程图

### Web 客户端安全功能比较

方案	XSS	JSON	Cache Control	Content Type Options	HTTP Strict Transport Security (HSTS)	HTTP Public Key Pinning (HPKP)	X-Frame-Options	X-XSS	Content Security Policy (CSP)
Servlet	✗	✗	✗	✗	✗	✗	✗	✗	✗
Apache Shiro	✗	✗	✗	✗	✗	✗	✗	✗	✗
Spring Security	✗	✗	○	○	○	○ <sup>1</sup>	○	○	○

### Web 服务端安全功能比较

方案	Cross Site Request Forgery (CSRF)	Authentication	Authorization	Cryptography	Session Management	Referer	Domain
Servlet	✗	○	○	✗	✗	✗	✗
Apache Shiro	✗	○	○	○	○	✗	✗
Spring Security	○	○	○	○	○	✗	✗

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests() ExpressionInterceptUrlRegistry
        .antMatchers("/**") ExpressionAuthorizationConfigurer<HttpSecurity>.AuthorizedUrl
        .authenticated() ExpressionUrlAuthorizationConfigurer<HttpSecurity> ExpressionInterceptUrlRegistry
        .and() HttpSecurity
        .csrf() CsrfConfigurer<HttpSecurity>
        .requireCsrfProtectionMatcher(request -> "POST".equalsIgnoreCase(request.getMethod()))      配置所保护资源列表
        .csrfTokenRepository(csrfTokenRepository)      将换成 HttpSession 的方式保存
        .disable() // 关闭 CSRF 功能
        .and() // 配置其他特性
        .formLogin() // Form
        .logout() // 登录登出页面的处理动作
}

```

## 资源保护配置

```
@Configuration  
@EnableResourceServer  
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {  
  
    @Override  
    public void configure(ResourceServerSecurityConfigurer configurer) throws Exception {  
        configurer.resourceId("1")  
    }  
}
```

授权

```
* 指定配置  
*/  
  
@Configuration  
@EnableAuthorizationServer  
public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter {  
  
    @Override  
    public void configure(AuthorizationServerSecurityConfigurer configurer) throws Exception {  
    }  
  
    @Override  
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
        clients.inMemory()  
            .withClient("zsxq") // ClientId = "zsxq"  
            .secret("123456") ClientDetailsServiceBuilder<InMemoryClientDetailsServiceBuilder>.ClientBuilder  
            .authorizedGrantTypes("password") // 授权类型密码 - "password" long duration  
            .scopes("read", "write", "trust") ClientDetailsServiceBuilder<InMemoryClientDetailsServiceBuilder>.ClientBuilder  
            .accessTokenValiditySeconds((int)TimeUnit.MINUTES.toSeconds(duration: 60))  
    }  
}
```

# session单点登录解决

## 关于 cookie 和 session 的联系

cookie 中会包含哪些信息

名字、值、过期时间、路径、域

qit.qupaoedu.com/blog/

cookie 会带到 http 请求头中发送给服务器端

如果 cookie 没有设置过期时间的话，那么 cookie 的默认生命周期是浏览器的会话。

session 机制

1. session 是容器对象，客户端在请求服务端的时候，服务端会根据客户端的请求判断是否包含了 sessionid 的标识
  2. 如果已经包含了，说明该客户端之前已经建立了会话。sessionid 是一个唯一的值
  3. 如果 sessionid 不存在，那么服务端会为这个客户端生成一个 sessionid。 JSESSIONID

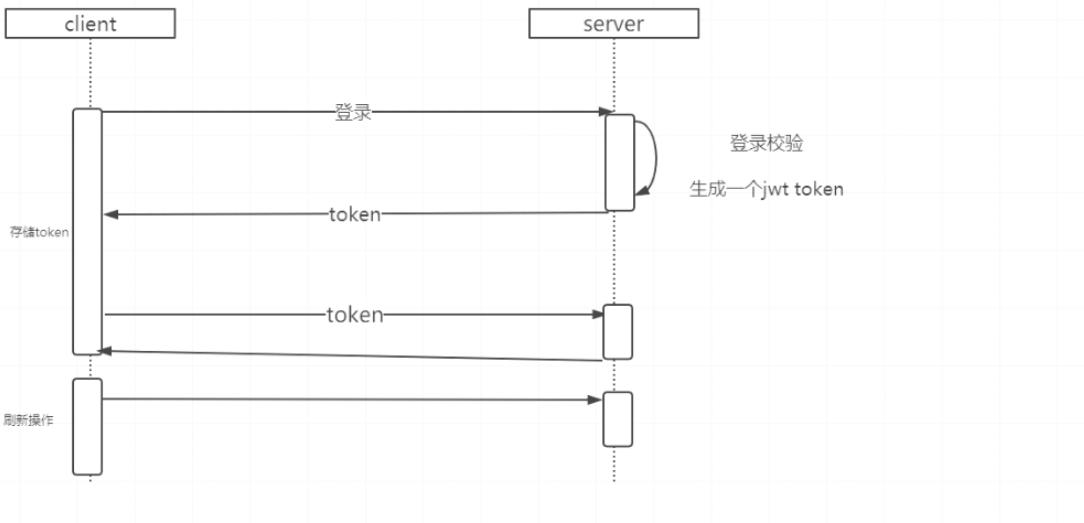
## 基于jwt的解决方案

## jwt组成部分

.heard {typ:"jwt",alg:"hs256"(算法)} payload{iss签发着} signature签名

heard和payload通过base64加密组成一个字符串  
base64(heard).Base(payload)生成JWT的token

str.签名字符串



```

public class JwtTokenUtil {

    private static Key getKeyInstance() {
        SignatureAlgorithm signatureAlgorithm=SignatureAlgorithm.HS256;
        byte[] dc=DatatypeConverter.parseBase64Binary( lexicalXSDBase64Binary: "gupao-user");
        return new SecretKeySpec(dc,signatureAlgorithm.getJcaName());
    }

    //生成token的方法
    public static String generatorToken(JwtInfo jwtInfo, int expire) {

        return Jwts.builder().claim(JwtConstants.JWT_KEY_USER_ID, jwtInfo.getUid())
            .setExpiration(DateTime.now().plusSeconds(expire).toDate())
            .signWith(SignatureAlgorithm.HS256, getKeyInstance()).compact();
    }
}

```

## 跨域问题：

浏览器的安全策略不许跨域操作，

使用nginx将项目部署到同一域中,配置请求头

### 1、添加响应头

- Access-Control-Allow-Origin: 支持哪些来源的请求跨域
- Access-Control-Allow-Methods: 支持哪些方法跨域
- Access-Control-Allow-Credentials: 跨域请求默认不包含cookie, 设置为true可以包含cookie
- Access-Control-Expose-Headers: 跨域请求暴露的字段
  - CORS请求时, XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段: Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, Pragma。如果想拿到其他字段, 就必须在Access-Control-Expose-Headers里面指定。
- Access-Control-Max-Age: 表明该响应的有效时间为多少秒。在有效时间内, 浏览器无须为同一请求再次发起预检请求。请注意, 浏览器自身维护了一个最大有效时间, 如果该首部字段的值超过了最大有效时间, 将不会生效。[https://blog.csdn.net/qq\\_35419179](https://blog.csdn.net/qq_35419179)

gateway网关中

分布式中解决办法, CorsWebFilter来解决跨域问题我们只需要将CorsWebFilter组件自定义并且注入到容器中即可解决跨域问题

```
1 @Configuration
2 public class WwmallCorsConfiguration {
3
4     @Bean
5     public CorsWebFilter corsWebFilter(){
6         UrlBasedCorsConfigurationSource
7         corsConfigurationSource = new
8         UrlBasedCorsConfigurationSource();
9         CorsConfiguration corsConfiguration =new
10        CorsConfiguration();
11        //允许哪些头跨域
12        corsConfiguration.addAllowedHeader("*");
13        //允许哪些方法跨域
14        corsConfiguration.addAllowedMethod("*");
15        //允许那些请求来源跨域
16        corsConfiguration.addAllowedOrigin("*");
17        //设置是否允许携带cookie跨域,否则跨域会丢失cookie信息
18        corsConfiguration.setAllowCredentials(true);
19
20        corsConfigurationSource.registerCorsConfiguration("/**",c
21        orsConfiguration);
22
23        return new
24        CorsWebFilter(corsConfigurationSource);
25    }
26}
```

# Redis

## 特性

官网介绍: <https://redis.io/topics/introduction>

中文网站: <http://www.redis.cn>

- 1) 更丰富的数据类型
- 2) 进程内与跨进程; 单机与分布式
- 3) 功能丰富: 持久化机制、过期策略
- 4) 支持多种编程语言
- 5) 高可用, 集群,

## 6)c语言开发的

ps:有 16 个库 (0-15) ,没有隔离性

## redis Linux安装步骤

[https://blog.csdn.net/Java\\_Fly1/article/details/109789305](https://blog.csdn.net/Java_Fly1/article/details/109789305)

```
1 拉取安装包
2 wget http://download.redis.io/releases/redis-5.0.5.tar.gz
3 编译redis需要gcc环境
4 yum -y install gcc
5 进入redis目录下的src目录, 执行以下命令
6 cd /usr/local/redis/src
7 make install
8
9 mkdir bin
10 mkdir etc
11 将 redis 里的主配置文件 redis.conf 移动到刚创建的 etc 文件夹
12 cd /usr/local/redis/src
13 mv mkrleasehdr.sh redis-benchmark redis-check-aof
   redis-check-rdb redis-cli redis-sentinel redis-server
   redis-trib.rb /usr/local/redis/bin/
14 修改配置文件
15 注释掉 bind 127.0.0.1 这一行
16 protected-mode 属性改为 no (关闭保护模式, 不然会阻止远程访问)
17 将 daemonize 属性改为 yes (这样启动时就在后台启动)
18 设置密码 123456 requirepass 123456
19 启动
20 ./bin/redis-server /usr/local/redis/etc/redis.conf
21 查看进程
22 ps -ef | grep redis
23 进入客户端
24 ./bin/redis-cli
25 auth 123456
26
```

## 常见五中类型

最基本的数据类型,外层hashtable,dictEntry,结构:数组加链表,redisObject 来存储

数据类型,整型,浮点,字符串(内部编码三种,int,embstr,raw,依次递增),异常珍惜内存,优化数据结构

int: 8个字节的长整型; embstr: 小于等于44个字节长度的字符串(简单动态字符串); raw: 大于44个字节的字符串。

ps:c语言字符串是char[]实现的,sds动态扩容不用担心溢出,字符串时间复杂度为 O(1),有预分配和多行分配特性

## String 升热点数据的访问速度

字符串类型是Redis中最为基础的数据存储类型,是一个由字节组成的序列,他在Redis中是二进制安全的,这便意味着该类型可以接受任何格式的数据,如JPEG图像数据和Json对象描述信息等,是标准的key-value,一般来存字符串,整数和浮点数。Value最多可以容纳的数据长度为512MB

应用场景:

## 分布式锁

STRING 类型 setnx 方法,只有不存在时才能添加成功,返回 true。

## 全局 ID

INT 类型, INCRBY, 利用原子性,(分库分表的场景,一次性拿一段) incrby  
userid 1000

## 计数器,

INT 类型, INCR 方法

例如: 文章的阅读量,微博点赞数,允许一定的延迟,先写入 Redis 再定时同步到数据库

## 限流

INT 类型, INCR 方法

以访问者的 IP 和其他信息作为 key, 访问一次增加一次计数, 超过次数则返回 false。

▶ 1. 字符串	1. set(set key value)
	2. get(get key)
	3. setnx(setnx key value)
	4. setrange(setrange key startIndex value) 替换字符串
	5. mset(mset key1 value1 key2 value2 ...) 批量设置键值对
	6. msetnx(msetnx key1 value1 key2 value2 ...) 如果key已存在那么设置失败
	7. getset(getset key newValue) 获取key的值然后设置新的值
	8. getrange(getrange key startIndex endIndex) 获取数据
	9. mget(mget key1 key2 key3 ...) 批量获取
	10. incr(incr key) 自增1
	11. incrby(incrby key num) 指定增减的数量
	12. decr(decr key) 自减1
	13. decrby(decrby key num) 指定自增减的数量
	14. append append(key value) 给指定的字符串追加 value 的值
	15. strlen(strlen key) 获取指定的key对应的值得长度

## list

Redis的列表允许用户从序列的两端推入或者弹出元素，列表由多个字符串值组成的有序可重复的序列，是链表结构，所以向列表两端添加元素的时间复杂度为O(1)，获取越接近两端的元素速度就越快。这意味着即使是一个有几千万个元素的列表，获取头部或尾部的10条记录也是极快的。List中可以包含的最大元素数量是4294967295。

**存储（实现）原理**:3.2之后quicklist（快速列表）是ziplist和linkedlist的结合体

应用场景：

### 用户消息时间线 timeline

**消息队列**:List 提供了两个阻塞的弹出操作：BLPOP/BRPOP，可以设置超时时间

过队列的 rpush 和 lpop 可以实现消息队列

▶ 2. List类型(类似于Java中的List)	1. lpush(lpush key value1 value2 ...) 往list集合中压入元素
	2. linsert(linsert key before/after value newValue) 在指定的元素前或者后插入元素
	3. lset(lset key index newValue) 设置指定下标的值
	4. lrem(lrem key count value) 删除 count 个与 value 相同的元素, count > 0 从开始位置进行删除, count < 0 从末尾开始删除, count = 0 删除所有的
	5. ltrim(ltrim key startIndex endIndex) 删除指定范围内以外的元素, 保留指定范围内的元素
	6. lpop(lpop key) 从list的头部删除元素
	7. lindex(lindex key index) 返回指定索引处的元素
	8. llen(llen key) 返回列表的长度
	9. rpush(rpush key value) 从末尾压入元素
	10. rpop(rpop key) 从末尾删除元素
	11. rpoplpush(rpoplpush key1 key2) 从key1链表中弹出最后一个元素然后压入到key2链表中

## hash

Redis中的散列可以看成具有String key和String value的map容器，可以将多个key-value存储到一个key中。每一个Hash可以存储4294967295个键值对。

value 只能是字符串，不能嵌套其他类型

## 同样是存储字符串， Hash 与 String 的主要区别？

- 1、把所有相关的值聚集到一个 key 中，节省内存空间
- 2、只使用一个 key，减少 key 冲突
- 3、当需要批量获取值的时候，只需要使用一个命令，减少内存/IO/CPU 的消耗

应用:存储、读取、修改用户属性 (name, age, pwd等)

key: 用户 id; field: 商品 id; value: 商品数量。

+1: hincr。 -1: hdecr。删除: hdel。全选: hgetall。商品数: hlen。



```
1 7 public class Hash_hset_operation {  
2     8     public static void main(String[] args) {  
3         9         jedis = new jedis("127.0.0.1",6379);  
4         10        /**  
5             * 示例1: hset hash的key 项的key 项的值  
6             */  
7         11         jedis.hset("myhash","id","3");  
8         12         jedis.hset("myhash","name","xiaohong");  
9         13         jedis.hset("myhash","age","13");  
10        14     }  
11    }
```

## set

### 存储（实现）原理

Redis 用 intset 或 hashtable 存储 set。如果元素都是整数类型，就用 intset 存储。如果不是整数类型，就用 hashtable （数组+链表的存来储结构）。

Redis的集合是无序不可重复的，和列表一样，在执行插入和删除和判断是否存在某元素时，效率是很高的。集合最大的优势在于可以进行交集并集差集操作。Set可包含的最大元素数量是4294967295。

应用场景：1.利用交集求共同好友。2.利用唯一性，可以统计访问网站的所有独立IP。3.好友推荐的时候根据tag求交集，大于某个threshold（临界值的）就可以推荐。抽奖 随机获取元素

4. Set(类似于Java中的Set)		
1. sadd(sadd key member1 member2...)添加元素		
2. scard(scard key)获取成员的数量		
3. sismember(sismember key member)判断是否存在member这个成员		
4. smembers(smembers key)获取所有的成员		
5. spop(spop key)随机弹出一个成员		
6. srandmember(srandmember key [count])随机获取一个或者多个成员		
7. srem(srem key member1 member2 ...)删除一个或者多个成员,如果成员不存在则忽略		
8. smove(smove source desition member)移动一个成员到指定的set中		
9. sdiff(sdiff first-key key1 key2...)返回给定集合之间的差集。不存在的集合 key 将视为空集		
10. sdiffstore(sdiffstore destionset key1 key2 ...)把获取到的差集保存到目标set中		
11. sinter(sinter key1 key2...)获取交集		
12. sinterstore(sinterstore destionset key1 key2....)把获取到的交集存储到目标set中		
13. sunion(sunion key1 key2...)获取并集		
14. sunionstore(sunionstore destionset key1 key2...)把获取到的并集存储到目标set中		

## zset

### 存储（实现）原理

同时满足以下条件时使用 ziplist 编码：

元素数量小于 128 个 ,所有 member 的长度都小于 64 字节

和set很像，都是字符串的集合，都不允许重复的成员出现在一个set中。他们之间差别在于有序集合中每一个成员都会有一个分数(score)与之关联，Redis正是通过分数来为集合中的成员进行从小到大的排序。尽管有序集合中的成员必须是唯一的，但是分数(score)却可以重复。

应用场景：可以用于一个大型在线游戏的积分排行榜，每当玩家的分数发生变化时，可以执行zadd更新玩家分数(score)，此后在通过zrange获取几分top ten的用户信息。

5. SortedSet(和Set类似但是可以排序)		
1. zadd(zadd key score1 member1 score2 member2...)添加成员		
2. zcard(zcard key)计算元素个数		
3. zincrby(zincrby key number member)给指定的member的分数添加或者减去number这个值		
4. zcount(zcount key min max)获取分数在min和max之间的成员和数量；默认是闭区间；想不包含可以：(min (max		
5. zrange(zrange key start stop [WITHSCORES])返回指定排名之间的成员(结果是分数由低到高)		
6. zrevrange(zrevrange key start stop [WITHSCORES])返回指定排名之间的成员(结果是分数由高到低)		
7. zrangebyscore(zrangebyscore key min max [withscores] [limit offset count]) 根据分数的范围获取成员(按照分数：从低到高)		
8. zrevrangebyscore(zrevrangebyscore key max min [withscores] [limit offset count]) 根据分数的范围获取成员(从高到低)		
9. zrank(zrank key member)返回一个成员的排名(从低到高的顺序)		
10. zrevrank(zrevrank key member)返回一个成员的排名(从高到低)		
11. zscore(zscore key member)获取一个成员的分数		
12. zrem(zrem key member1 member2...)删除指定的成员		
13. zremrangebyrank(zremrangebyrank key start stop)根据排名进行删除		
14. zremrangebyscore(zremrangebyscore key min max)根据分数的范围进行删除		

对象	对象 type 属性值	type 命令输出	底层可能的存储结构	object encoding
字符串对象	OBJ_STRING	"string"	OBJ_ENCODING_INT OBJ_ENCODING_EMBSTR OBJ_ENCODING_RAW	int embstr raw
列表对象	OBJ_LIST	"list"	OBJ_ENCODING_QUICKLIST	quicklist
哈希对象	OBJ_HASH	"hash"	OBJ_ENCODING_ZIPLIST OBJ_ENCODING_HT	ziplist hashtable
集合对象	OBJ_SET	"set"	OBJ_ENCODING_INTSET OBJ_ENCODING_HT	intset hashtable
有序集合对象	OBJ_ZSET	"zset"	OBJ_ENCODING_ZIPLIST OBJ_ENCODING_SKIPLIST	ziplist skipiplist (包含 ht)

对象	原始编码	升级编码	
字符串对象	INT	embstr	raw
	整数并且小于 long 2^63-1	超过 44 字节, 被修改	
哈希对象	ziplist	hashtable	
	键和值的长度小于 64byte, 键值对个数不超过 512 个, 同时满足		
列表对象	quicklist		
集合对象	intset	hashtable	
	元素都是整数类型, 元素个数小于 512 个, 同时满足		
有序集合对象	ziplist	skipiplist	
	元素数量不超过 128 个, 任何一个 member 的长度小于 64 字节, 同时满足。		

## 其他数据类型

### Hyperloglogs

Hyperloglogs：提供了一种不太准确的基数统计方法，比如统计网站的 UV，存在一定的误差。HyperLogLogTest.java

### Streams

5.0 推出的数据类型。支持多播的可持久化的消息队列，用于实现发布订阅功能，借鉴了 kafka 的设计

## 高级特性

### 发布订阅

队列存在的问题 (的 rpush 和 lpop 可以实现消息队列)

- 如果生产者生产消息的速度远大于消费者消费消息的速度，List 会占用大量的内存。
- 消息的实时性降低。list 还提供了一个阻塞的命令：blpop，没有任何元素可以弹出的时候，连接会被阻塞。

首先，我们有很多的频道（channel），我们也可以把这个频道理解成 queue。订阅者可以订阅一个或者多个频道。消息的发布者（生产者）可以给指定的频道发布消息。只要有消息到达了频道，所有订阅了这个频道的订阅者都会收到这条消息。

比如这个客户端订阅了 3 个频道。

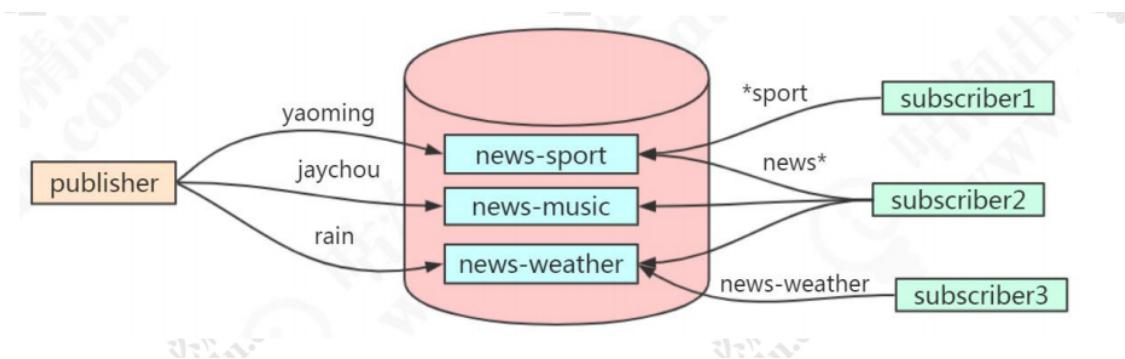
```
subscribe channel-1 channel-2 channel-3
```

发布者可以向指定频道发布消息（并不支持一次向多个频道发送消息）：

```
publish channel-1 2673
```

取消订阅（不能在订阅状态下使用）：

```
unsubscribe channel-1
```



## Redis 事务

Redis 的事务有两个特点：

- 1、按进入队列的顺序执行。
- 2、不会受到其他客户端的请求的影响。

Redis 的事务涉及到四个命令：multi（开启事务），exec（执行事务），discard（取消事务），watch（监视）

- ```
1 demo
2 案例场景：tom 和 mic 各有 1000 元，tom 需要向 mic 转账 100 元。
  tom 的账户余额减少 100 元，mic 的账户余额增加 100 元
3 127.0.0.1:6379> set tom 1000 OK
127.0.0.1:6379> set mic 1000 OK
127.0.0.1:6379> multi OK
127.0.0.1:6379> decrby tom 100 QUEUED
127.0.0.1:6379> incrby mic 100 QUEUED
127.0.0.1:6379> exec
4 1) (integer) 900 2) (integer) 1100
127.0.0.1:6379> get tom "900"
127.0.0.1:6379> get mic "1100"
```

**注意:redis事务无法嵌套使用,无法回滚**

## watch 命令

基于乐观锁CAS实现对多个key监控,如果开启事务之后, 至少有一个被监视 key 键在 exec 执行之前被修改了, 那么整个事务都会被取消 (key 提前过期除外)

用 unwatch 取消。

## Lua 脚本

使用Lua脚本对redis命令批量操作,基于C语言

脚本直接操作redis命令 eval "return redis.call('set',KEYS[1],ARGV[1])" 1  
gupao 2673 ,()内为redis命令

执行script load命令可以查看缓存

避免脚本超时设置了时间lua-time-limit,默认5秒

## Redis 为什么这么快?

1) 纯内存结构、2) 单线程、3) 多路复用 ;根据官方的数据, Redis 的 QPS 可以达到 10 万左右 (每秒请求数)

## 内存回收

### 定时过期 (主动淘汰)

每个设置过期时间的 key 都需要创建一个定时器, 到过期时间就会立即清除。该策略可以立即清除过期的数据, 对内存很友好; 但是会占用大量的 CPU 资源去处理过期的数据, 从而影响缓存的响应时间和吞吐量。

### 惰性过期 (被动淘汰)

只有当访问一个 key 时, 才会判断该 key 是否已过期, 过期则清除。该策略可以最大化地节省 CPU 资源, 却对内存非常不友好。极端情况可能出现大量的过期 key 没有再次被访问, 从而不会被清除, 占用大量内存

### 定期过期

每隔一定的时间, 会扫描一定数量的数据库的 expires 字典中一定数量的 key, 并清除其中已过期的 key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时, 可以在不同情况下使得 CPU 和内存资源达到最优的平衡效果。默认1秒

## 淘汰策略

设置 maxmemory 或者设置为 0, 64 位系统不限制内存, 32 位系统最多使用 3GB 内存

LRU, Least Recently Used 最近最少使用

传统实现

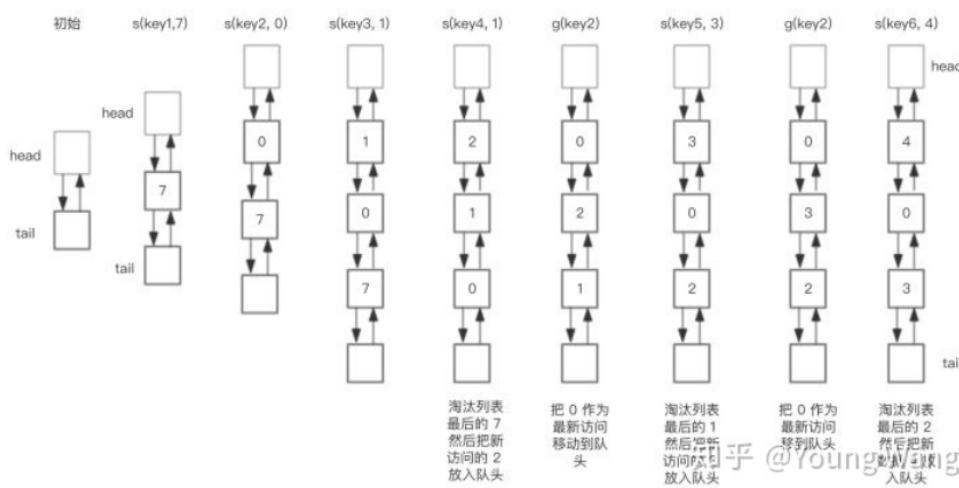
1. save(key, value):

2. 1. 首先在 HashMap 找到 Key 对应的节点, 如果节点存在, 更新节点的值, 并把这个节点移动队头。
2. 如果不存在, 需要构造新的节点, 并且尝试把节点塞到队头。
3. 如果 LRU 空间不足, 则通过 tail 淘汰掉队尾的节点, 同时在 HashMap 中移除 Key。

3. get(key):

4. 1. 通过 HashMap 找到 LRU 链表节点, 因为根据 LRU 原理, 这个节点是最新访问的, 所以要把节点插入到队头, 然后返回缓存的值。
5. Redis LRU 对传统的 LRU 算法进行了改良, **通过随机采样来调整算法的精度。**

使用HashMap + 双向链表数据结构实现的LRU操作双向链表部分的轨迹如下。



LFU, Least Frequently Used, 最不常用, 4.0 版本新增。

| 策略              | 含义                                                                                                                              |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------|
| volatile-lru    | 根据 LRU 算法删除设置了超时属性（ <code>expire</code> ）的键，直到腾出足够内存为止。如果没有可删除的键对象，回退到 <code>noeviction</code> 策略。                              |
| allkeys-lru     | 根据 LRU 算法删除键，不管数据有没有设置超时属性，直到腾出足够内存为止。                                                                                          |
| volatile-lfu    | 在带有过期时间的键中选择最不常用的。                                                                                                              |
| allkeys-lfu     | 在所有的键中选择最不常用的，不管数据有没有设置超时属性。                                                                                                    |
| volatile-random | 在带有过期时间的键中随机选择。                                                                                                                 |
| allkeys-random  | 随机删除所有键，直到腾出足够内存为止。                                                                                                             |
| volatile-ttl    | 根据键值对象的 <code>ttl</code> 属性，删除最近将要过期数据。如果没有，回退到 <code>noeviction</code> 策略。                                                     |
| noeviction      | 默认策略，不会删除任何数据，拒绝所有写入操作并返回客户端错误信息 ( <code>error</code> ) <code>OOM command not allowed when used memory</code> ，此时 Redis 只响应读操作。 |

## 持久化机制

一种是 RDB 快照 (Redis DataBase) RDB 是 Redis 默认的持久化方案。当满足一定条件的时候，会把当前内存中的数据写入磁盘，生成一个快照文件 `dump.rdb`。Redis 重启会通过加载 `dump.rdb` 文件恢

### 触发机制

`redis.conf`, `SNAPSHOTTING`, 其中定义了触发把数据保存到磁盘的触发频率 `save` 或 `bgsave`(异步), 生产使用 `bgsave`,

执行 `bgsave` 时，Redis 会在后台**异步进行快照操作**，具体操作是 Redis 进程执行 `fork` 操作创建子进程 (copy-on-write)，RDB 持久化过程由子进程负责，完成后自动结束。它**不会记录 fork 之后后续的命令**。阻塞只发生在 `fork` 阶段，**一般时间很短**

弊端:没办法做到实时持久化/秒级持久化,丢失最后一次快照之后的所有修改  
(数据有丢失)

一种是 AOF (Append Only File)

Redis 默认不开启。AOF 采用日志的形式来记录每个写操作，并**追加到文件中**。开启后，执行更改 Redis 数据的命令时，就会把命令写入到 AOF 文件中。

Redis 重启时会根据日志文件的内容把写指令从前到后执行一次以完成数据的恢复 .文件达到阀值,自动压缩

优点,高频次保存,不必新开线程,文件大恢复慢

高并发还是选用RDB

# redis集群

**主从复制原理** redis自带

1.执行命令slaveof ip 选定主节点

2.socket连接主节点,建立网路连接

3.第一次全量复制,master bgsave生成备份RDB回传到slave(设置超时时间),

--在生成RDB文件的时候新的命令会缓存在内存,后续再复制到slave node

--slave node 断掉,重入会先删点本地再重新复制

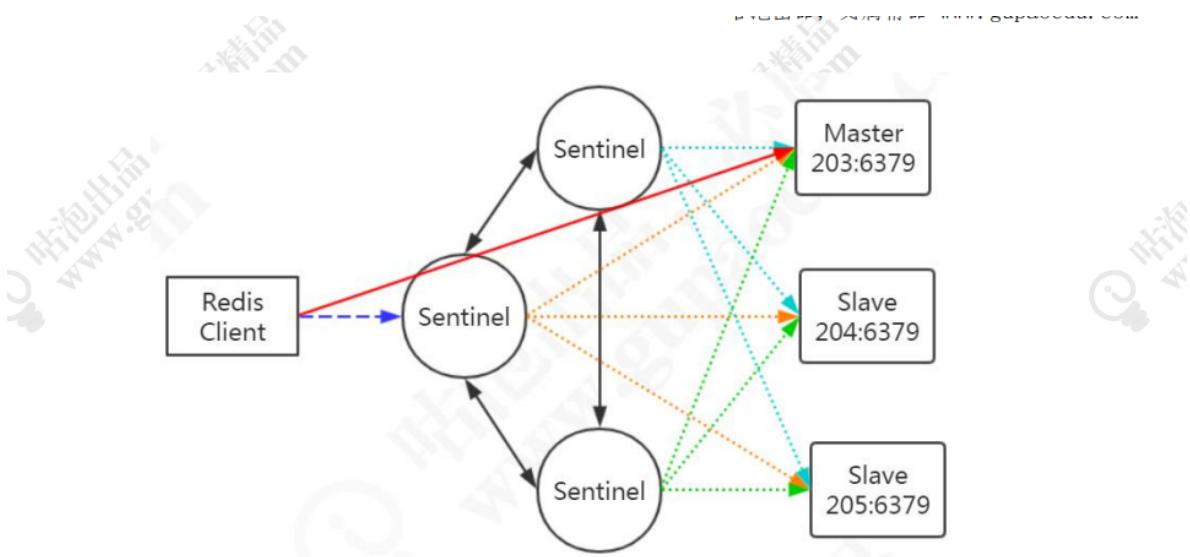
低延迟,高效率的选择

不足:

RDB 文件过大的情况下, 同步非常耗时

在一主一从或者一主多从的情况下, 如果主服务器挂了, 对外提供的服务就不可用了, 单点问题没有得到解决。如果每次都是手动把之前的从服务器切换成主服务器, 这个比较费时费力, 还会造成一定时间的服务不可用。

**可用性保证 sentinel(哨兵)**



Sentinel 的 Raft 算法( 共识算法 )

开启一个或多个sentinel,特殊的redis服务启动,需要专门的配置文件支持

sentinel每秒ping一次服务,超过未连上判断服务下线,大于半数sentinel,确定下线

确认master下线,现在sentinel里面选出lead,lead再选master,采用Raft算法实现 Sentinel 选举

Raft 的核心思想：先到先得，少数服从多数

redis-sentinel是一个单独的进程在和redis安装包一同下载了

### 问题：怎么让一个原来的 slave 节点成为主节点？

- 1、选出 Sentinel Leader 之后，由 Sentinel Leader 向某个节点发送 slaveof no one 命令，让它成为独立节点。
- 2、然后向其他节点发送 slaveof x.x.x.x xxxx（本机服务），让它们成为这个节点的子节点，故障转移完成。

### 问题：这么多从节点，选谁成为主节点？

关于从节点选举，一共有四个因素影响选举的结果，分别是**断开连接时长、优先级排序、复制数量、进程 id**。

如果与哨兵连接断开的比较久，超过了某个阈值，就直接失去了选举权。如果拥有选举权，那就看谁的优先级高，这个在配置文件里可以设置（replica-priority 100），数值越小优先级越高。

如果优先级相同，就看谁从 master 中复制的数据最多（复制偏移量最大），选最多的那个，如果复制数量也相同，就选择进程 id 最小的那个

安装使用

<https://blog.csdn.net/u010416101/article/details/79661468>

### 哨兵机制的不足

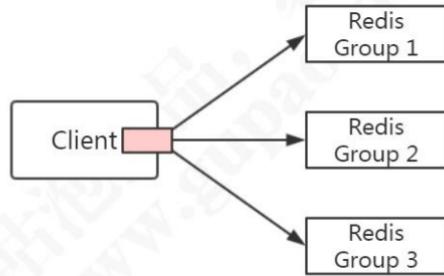
**主从切换的过程中会丢失数据**，因为只有一个 master。只能单点写，没有解决水平扩容的问题。

如果数据量非常大，这个时候我们需要多个 master-slave 的 group，把数据分布到不同的 group 中

## Redis 分布式方案(Redis 数据的分片)

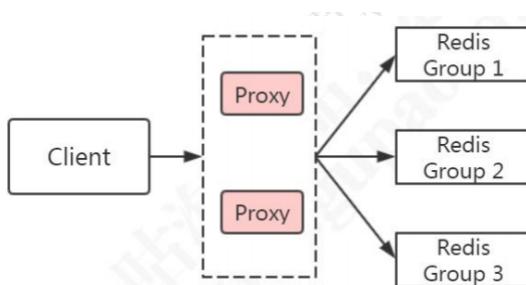
客户端

用取模或者一致性哈希对 key 进行分片，查询和修改都先判断 key 的路由  
Jedis 客户端提供了 Redis Sharding 的方案，并且支持连接池

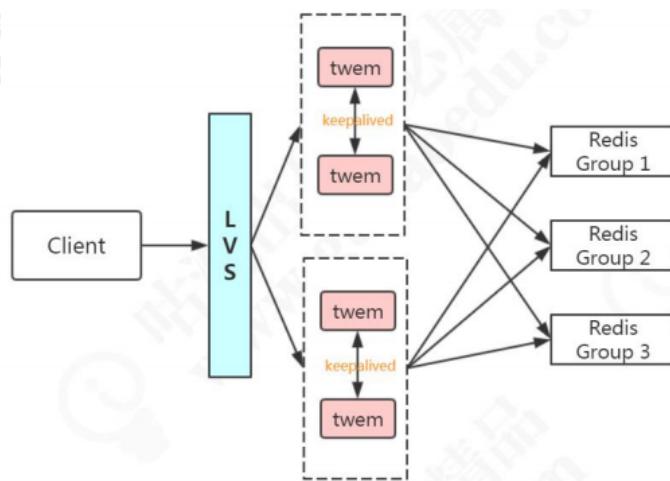


## 独立服务

运行一个独立的代理服务，客户端连接到这个代理服务，代理服务做请求的转发典型的代理分区方案有 Twitter 开源的 Twemproxy 和国内的豌豆荚开源的 Codis



## Twemproxy



优点：比较稳定，可用性高。

不足： 1、出现故障不能自动转移，架构复杂，需要借助其他组件  
(LVS/HAProxy +Keepalived) 实现 HA

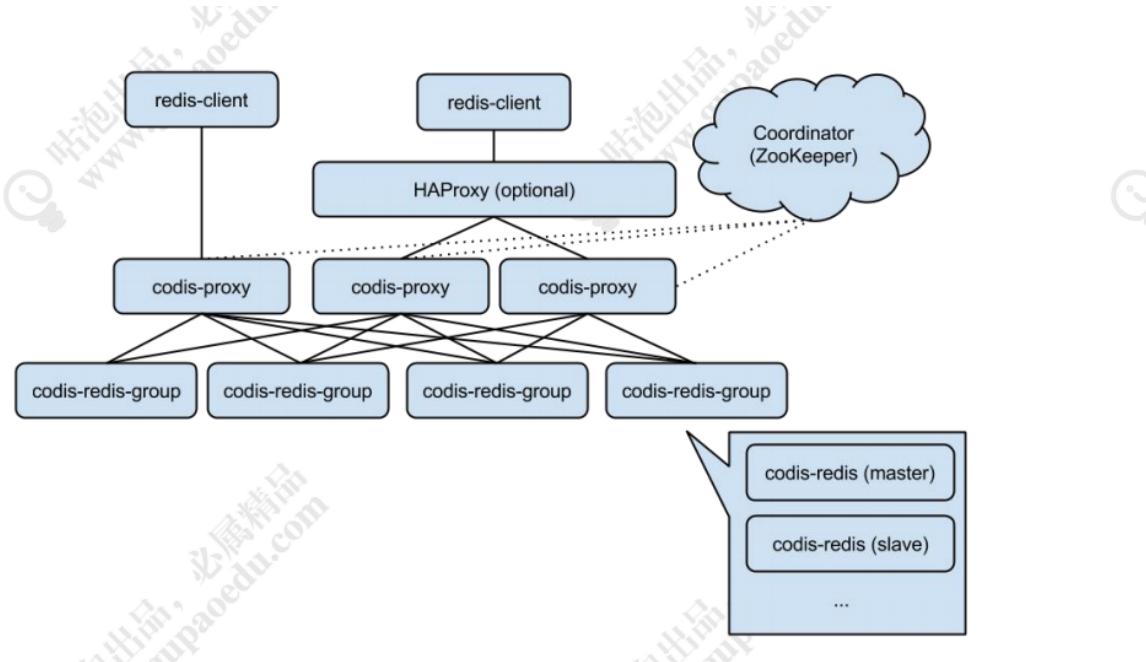
2、扩缩容需要修改配置，不能实现平滑地扩缩容（需要重新分布数据）。

## Codis

Codis 是一个代理中间件，用 Go 语言开发的

功能：客户端连接 Codis 跟连接 Redis 没有区别。

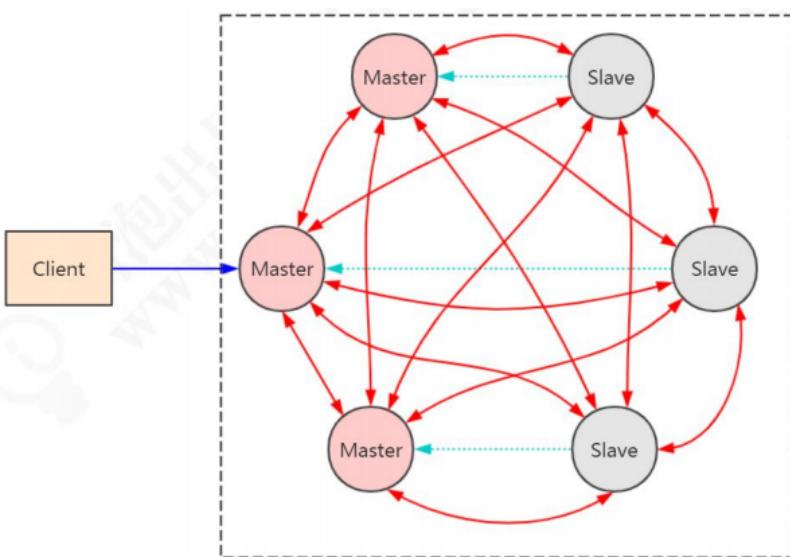
|                        | Codis | Twmproxy | Redis Cluster     |
|------------------------|-------|----------|-------------------|
| 重新分片不需要重启              | Yes   | No       | Yes               |
| pipeline               | Yes   | Yes      |                   |
| 多 key 操作的 hash tags {} | Yes   | Yes      | Yes               |
| 重新分片时的多 key 操作         | Yes   | -        | No                |
| 客户端支持                  | 所有    | 所有       | 支持 cluster 协议的客户端 |



分片原理：Codis 把所有的 key 分成了 N 个槽（例如 1024），每个槽对应一个分组，一个分组对应于一个或者一组 Redis 实例。Codis 对 key 进行 CRC32 运算，得到一个32 位的数字，然后模以 N（槽的个数），得到余数，这个就是 key 对应的槽，槽后面就是 Redis 的实例

## Redis Cluster

由多个 Redis 实例组成的数据集合。客户端不需要关注数据的子集到底存储在哪个节点，只需要关注这个集合整体服务端



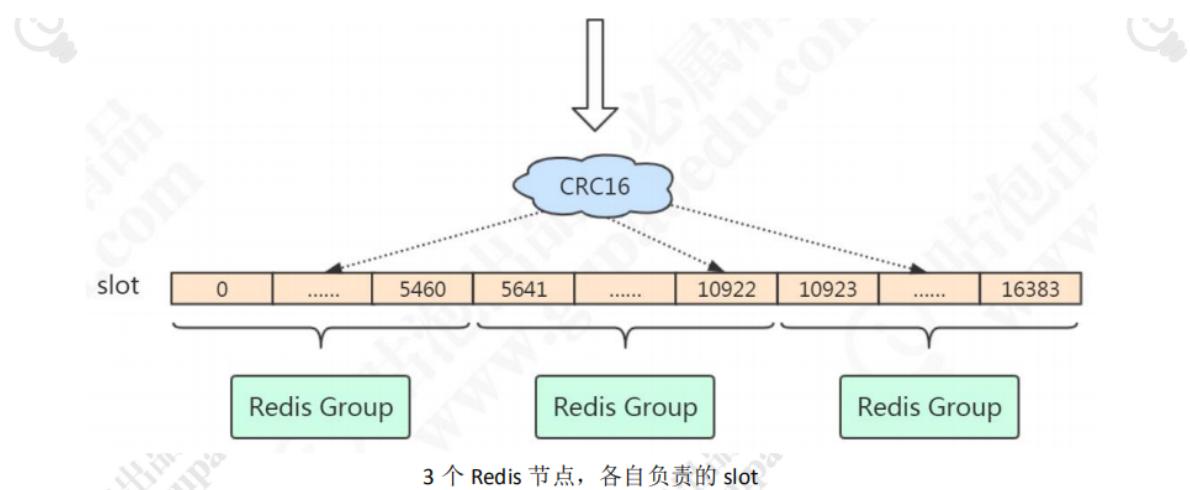
数据分片有几个关键的问题需要解决：

- 1、数据怎么相对均匀地分片
- 2、客户端怎么访问到相应的节点和数据
- 3、重新分片的过程，怎么保证正常服务

普通的事key的hash值取模,谷歌推出一致性哈希,hash值环形,增加节点最多影响一个

缺点,节点少的时候不均匀

引入虚拟机弥补



Redis 的每个 master 节点维护一个 **16384 位** ( $2048\text{bytes}=2\text{KB}$ ) 的位序列, 比如: 序列的第 0 位是 1, 就代表第一个 slot 是它负责; 序列的第 1 位是 0, 代表第二个 slot 不归它负责。对象分布到 Redis 节点上时, 对 key 用 CRC16 算法计算再 $\%16384$ , 得到一个 slot 的值, 数据落到负责这个 slot 的 Redis 节点上。

## 应用场景总结

缓存——提升热点数据的访问速度

共享数据——数据的存储和共享的问题

全局 ID —— 分布式全局 ID 的生成方案 (分库分表) 基于 Redis 的 INCR 命令实现序列号的生成基本能满足全局唯一与单调递增的特性

分布式锁——进程间共享数据的原子操作保证,借助于redis中的命令setnx(key, value), key不存在就新增, 存在就什么都不做

实现幂等性

为需要保证幂等性的每一次请求创建一个唯一标识token, 先获取token, 并将此token存入redis, 请求接口时, 将此token放到header或者作为请求参数请求接口, 后端接口判断redis中是否存在此token:

在线用户统计和计数

队列、栈——跨进程的队列/栈

消息队列——异步解耦的消息机制

服务注册与发现 —— RPC 通信机制的服务协调中心 (Dubbo 支持 Redis)

购物车

新浪/Twitter 用户消息时间线

抽奖逻辑 (礼物、转发)

点赞、签到、打卡

商品标签

用户 (商品) 关注 (推荐) 模型

电商产品筛选

排行榜

session共享

## 常见问题

### 缓存雪崩

大多数系统设计者考虑用加锁 (最多的解决方案) 或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效

时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开。

### 缓存穿透 高并发下命中率问题

### 解决办法\*\*;\*\*

最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个bitmap拦截掉，从

而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空 (不管是数据不存在，还是系统故

障)，我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次

到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴。

## 双写不一致问题,三种缓存模式根据业务自行选择

### 旁路缓存模式 适合读请求比较多的场景

同时维系 DB 和 cache，并且是以 DB 的结果为准。

写:先更新DB,然后删除cache

读:从cache中读取数据,读到就直接返回, cache中读取不到的话,就从DB中读取数据返回,在把数据放到cache中,

无法百分之百操作,只能降低风险,首次cache可以人工手动进行预热

读写穿透:

写先查cache,cache中不存在,直接更新Db

cache中存在, 则更新cache, 然后cache服务自己更新Db

读: 从cache 中读取数据, 读取到就直接返回, 独具不到的话, 就从DB加载, 写入到cache后返回响应,

异步缓存写入

批量操作考虑到脚本

[https://blog.51cto.com/u\\_15080000/2601956](https://blog.51cto.com/u_15080000/2601956)

# Docker操作

## 基本命令

- 1 利用Dockerfile 自定义镜像
- 2 docker pull [软件名]:[版本号]
- 3 docker search [软件名]:[版本号]
- 4 docker images 查看所有镜像
- 5 docker ps -a 查看容器运行状态
- 6 docker start/stop [容器id]或者[别名]
- 7 docker rm [容器id] (-f \$(docker ps -aq) 删除所容器)

```
8 docker rmi [镜像id](-f $(docker images)) //先
9   删除容器才能删除镜像
10  docker exec -it [容器id] /bin/bash
11  exit 退出
12  docker commit [别名] 提交名
13  查看容器资源占用
14  docker stats [容器名称]
15
16  docker run -d --link mongodb:db -p 80:9080 --name myapp
17    myapp
18    -d 后台运转,
19    --link 通信,直接连通两个容器的网络,后面接容器名称,建议自定义网
20    卡
21    -p端口映射,
22    --name 别名
23    --memory 内存      --cpu 权重
24    -e MYSQL_ROOT_PASSWORD=123456      mysql设置root账号密码
25    日志查看
26    docker logs --tail="10" mytest
27
28
29 //搭建镜像私服
30 阿里云,harbor,先登录
31
32 weaveworks 图形化界面监控
33 sudo curl -L git.io/scope -o /usr/local/bin/scope
34 sudo chmod a+x /usr/local/bin/scope
35 scope launch 39.100.39.63
36 # 停止
37 scope scope stop
38 # 同时监控两台机器, 在两台机器中分别执行如下命令
39 scope launch
40
41 docker compes
42
43 更多命令参看
44 https://docs.docker.com/engine/reference/commandline/docker/
45 可以参考每一个镜像的文档
46
```

```
47  
48  
49 jdk安装  
50 创建文件夹 mkdir /usr/local/java  
51 移动安装包到指定文件 mv jdk-8u144-linux-x64.tar.gz  
/usr/local/java/  
52 cd /usr/local/java/ 目录 解压 tar -zxvf jdk-8u144-  
linux-x64.tar.gz  
53 tar -zxvf jdk-8u144-linux-x64.tar.gz -C /usr/local/java/  
安装到指定目录  
54 修改文件 vi /etc/profile  
55 #set java environment  
56 export JAVA_HOME=/usr/local/java/jdk1.8.0_144  
57 export PATH=$JAVA_HOME/bin:$PATH  
58 export  
CLASSPATH=.:${JAVA_HOME}/lib/rt.jar:${JAVA_HOME}/lib/tools.j  
ar  
59 保存退出让命令生效  
60 source /etc/profile  
61 检测是否装好  
62 java -version  
63  
64 CentOS 7.0默认使用的是firewall作为防火墙  
65  
66 查看防火墙状态  
67  
68 firewall-cmd --state  
69 1  
70 停止firewall  
71  
72 systemctl stop firewalld.service  
73 1  
74 禁止firewall开机启动  
75  
76 systemctl disable firewalld.service  
77  
78 docker-compose  
79 编排操作,起集群  
80 https://www.cnblogs.com/minseo/p/11548177.html  
81  
82  
83  
84
```

```
85
86
87 利用Dockerfile 自定义镜像
88 docker pull [软件名]:[版本号]
89 docker search [软件名]:[版本号]
90 docker images 查看所有镜像
91 docker ps -a 查看容器运行状态
92 docker start/stop [容器id]或者[别名]
93 docker rm [容器id] (-f $(docker ps -aq))    删除所容器)
94 docker rmi [镜像id](-f $(docker images))    删除所有镜像)//先
删除容器才能删除镜像
95 docker exec -it [容器id] /bin/bash
96 修改容器反向生成镜像
97 docker commit [别名] 提交名
98 查看容器资源占用
99 docker stats [容器名称]
100 exit 退出
101 docker run -d --link mongodb:db -p 80:9080 --name myapp
myapp
102 -d 后台运转,
103 --link 通信,直接连通两个容器的网络,后面接容器名称,建议自定义网
卡
104 -p端口映射,
105 --name 别名
106 --memory 内存      --cpu 权重
107 -e MYSQL_ROOT_PASSWORD=123456   mysql设置root账号密码
108 -v 持久化,接宿主机目录,和容器数据生成目录,
109
110 查看容器持久化内容
111 docker volume ls
112 docker volume inspect [id]查看详情
113 日志查看
114 docker logs --tail="10" mytest
115
116
117 //搭建镜像私服
118 阿里云,harbor,先登录
119
120 weaveworks 图形化界面监控
121 sudo curl -L git.io/scope -o /usr/local/bin/scope
122 sudo chmod a+x /usr/local/bin/scope
123 scope launch 39.100.39.63
124 # 停止
```

```
125 scope scope stop
126 # 同时监控两台机器，在两台机器中分别执行如下命令
127 scope launch
128
129
130 Replication 弱一致性，主节点写入数据库就可以了
131 PXC Percona XtraDB Cluster 强一致性
132
133 PXCPERCONAPERCONA XtraDB Cluster 强一致性
134
135 单机管理，多容器
136 docker-compose
137 多机:k8s, docker-swarm
138
139 docker-swarm 内置于docker之中
140 docker swarm init 初始化
141 docker stack 编排
142 docker service scope xxx=[n]
143 集群每个节点都可以访问，overlay(跨节点访问，类似路由跨网段)
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166 04 创建一个单独的网段，给mysql数据库集群使用
167 (1) docker network create --subnet=172.18.0.0/24 pxc-net
```

```
168 (2)docket network inspect pxc-net [查看详情]
169 (3)docker network rm pxc-net [删除]
170
171 5 创建和删除volume 创建:
172 docker volume create --name v1
173 删除: docker volume rm v1
174 查看详情: docker volume inspect v1
175
176 docker持久化数据库数据
177 容器启动时创建一个容器卷 volume
178 docker run -itd --name ubuntu_test1 -v
    /container_data:/data ubuntu
179 -v 连接宿主机目录和容器数据存储目录,可以用多个数据卷
180
```

## 创建Dockerfile文件,

```
//引入其他基础镜像
FROM openjdk:8
//作者信息
MAINTAINER itcrazy2016
//指定元数据,避免中间镜像产生
LABEL name="dockerfile-demo" version="1.0" author="itcrazy2016"
//复制jar包到指定位置,供镜像使用
COPY dockerfile-demo-0.0.1-SNAPSHOT.jar dockerfile-image.jar
//镜像启动后的运行命令
CMD ["java","-jar","dockerfile-image.jar"]
```

运行

```
docker build -t test-docker-image .
```

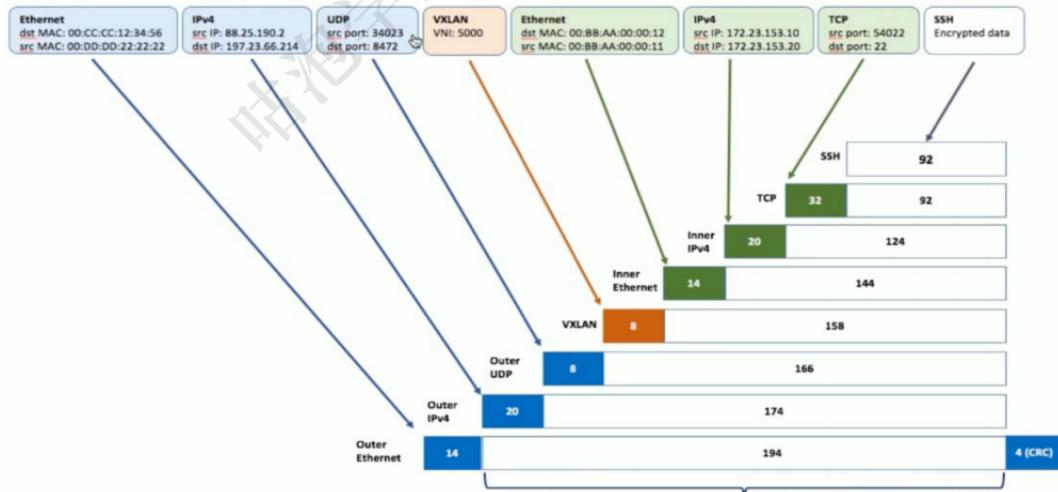
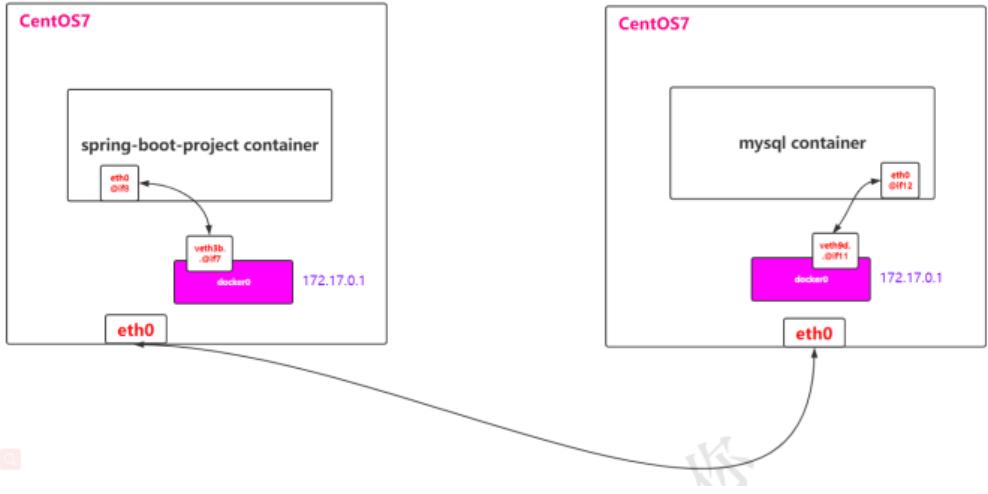
## 更多参考

<https://www.cnblogs.com/panwenbin-logs/p/8007348.html>

多机之间的container通信

多机网络通信的问题，底层一个实现技术是: overlay ---->vxlan

bridge host null ---->docker多机网络通信 overlay



## Docker 数据持久化

<https://github.com/apache/zookeeper/tags?after=release-3.4.14-rc0>

[https://blog.csdn.net/qq\\_33612228/article/details/106424541?ops\\_request\\_misc=%25257B%252522request%25255Fid%252522%25253A%252522161061174416780255280224%252522%25252C%252522scm%252522%25253A%25252220140713.130102334..%252522%25257D&request\\_id=161061174416780255280224&biz\\_id=0&utm\\_medium=distribute.pc\\_search\\_result.none-task-blog-2~all~first\\_rank\\_v2~rank\\_v29-1-106424541.pc\\_search\\_result\\_hbase\\_insert&utm\\_term=zookeeper%E9%9B%86%E7%BE%A4%E5%A4%89%A3%85](https://blog.csdn.net/qq_33612228/article/details/106424541?ops_request_misc=%25257B%252522request%25255Fid%252522%25253A%252522161061174416780255280224%252522%25252C%252522scm%252522%25253A%25252220140713.130102334..%252522%25257D&request_id=161061174416780255280224&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_v2~rank_v29-1-106424541.pc_search_result_hbase_insert&utm_term=zookeeper%E9%9B%86%E7%BE%A4%E5%A4%89%A3%85)

```
dubbo://169.254.94.134:20880/com.cainiao.lili.SayHello?  
anyhost=true&application=springboot-dubbo-  
server&bean.name=ServiceBean:com.cainiao.lili.SayHello&deprecated=fal  
se&dubbo=2.0.2&dynamic=true&generic=false&interface=com.cainiao.lili.  
SayHello&methods=sayHello&pid=17464&register=true&release=2.7.2&si  
de=provider&timestamp=1610634749101 to registry  
192.168.150.128:2181, cause: Failed to register  
dubbo://169.254.94.134:20880/com.cainiao.lili.SayHello?  
anyhost=true&application=springboot-dubbo-  
server&bean.name=ServiceBean:com.cainiao.lili.SayHello&deprecated=fal  
se&dubbo=2.0.2&dynamic=true&generic=false&interface=com.cainiao.lili.  
SayHello&methods=sayHello&pid=17464&register=true&release=2.7.2&si  
de=provider&timestamp=1610634749101 to zookeeper  
zookeeper://192.168.150.128:2181/org.apache.dubbo.registry.RegistrySer  
vice?application=springboot-dubbo-  
server&dubbo=2.0.2&interface=org.apache.dubbo.registry.RegistryService  
&pid=17464&qos-  
enable=false&release=2.7.2&timestamp=1610634749099, cause:  
KeeperErrorCode = Unimplemented for /dubbo  
at  
org.apache.dubbo.registry.support.FailbackRegistry.register(FailbackRegist  
ry.java:249) ~[dubbo-2.7.2.jar:2.7.2]
```

## Kubernetes

### k8s文件解读

```
1 kubernetes创建pod的yaml文件，参数说明  
2  
3 apiVersion: v1 #指定api版本，此值必须在kubectl apiVersion中  
4 kind: Pod #指定创建资源的角色/类型  
5 metadata: #资源的元数据/属性  
6   name: web04-pod #资源的名字，在同一个namespace中必须唯一  
7   labels: #设定资源的标签，详情请见  
     http://blog.csdn.net/liyingke112/article/details/77482384
```

```
8   k8s-app: apache
9     version: v1
10    kubernetes.io/cluster-service: "true"
11    annotations:          #自定义注解列表
12      - name: String      #自定义注解名字
13  spec:#specification of the resource content 指定该资源的内容
14    restartPolicy: Always #表明该容器一直运行， 默认k8s的策略，在此容器退出后，会立即创建一个相同的容器
15    nodeSelector:         #节点选择，先给主机打标签kubectl label
16      nodes kube-node1 zone=node1
17      zone: node1
18    containers:
19      - name: web04-pod #容器的名字
20        image: web:apache #容器使用的镜像地址
21        imagePullPolicy: Never #三个选择Always、Never、
22          IfNotPresent，每次启动时检查和更新（从registry）images的策
23          略，                                     # Always，每次都检查
24          # Never，每次都不检查（不管本地是
25          否有）                                     # IfNotPresent，如果本地有就不检
26          查，如果没有就拉取
27        command: ['sh'] #启动容器的运行命令，将覆盖容器中的
28          Entrypoint，对应Dockerfile中的ENTRYPOINT
29        args: ["$(str)"] #启动容器的命令参数，对应Dockerfile中CMD
30          参数
31        env: #指定容器中的环境变量
32          - name: str #变量的名字
33            value: "/etc/run.sh" #变量的值
34        resources: #资源管理，请求请见
35          http://blog.csdn.net/liyingke112/article/details/77452630
36          requests: #容器运行时，最低资源需求，也就是说最少需要多少
37          资源容器才能正常运行
38          cpu: 0.1 #CPU资源（核数），两种方式，浮点数或者是整数
39          +m，0.1=100m，最少值为0.001核（1m）
40          memory: 32Mi #内存使用量
41        limits: #资源限制
42          cpu: 0.5
43          memory: 32Mi
44        ports:
45          - containerPort: 80 #容器开发对外的端口
46            name: httpd #名称
```

```
39      protocol: TCP
40      livenessProbe: #pod内容器健康检查的设置，详情请见
41          httpGet: #通过httpget检查健康，返回200-399之间，则认为容
42             器正常
43                  path: / #URI地址
44                  port: 80
45                  #host: 127.0.0.1 #主机地址
46                  scheme: HTTP
47          initialDelaySeconds: 180 #表明第一次检测在容器启动后多
48             长时间后开始
49          timeoutSeconds: 5 #检测的超时时间
50          periodSeconds: 15 #检查间隔时间
51          #也可以用这种方法
52          #exec: 执行命令的方法进行监测，如果其退出码不为0，则认为容
53             器正常
54          # command:
55          #     - cat
56          #     - /tmp/health
57          #也可以用这种方法
58          #tcpSocket: //通过tcpSocket检查健康
59          # port: number
60      lifecycle: #生命周期管理
61      postStart: #容器运行之前运行的任务
62          exec:
63              command:
64                  - 'sh'
65                  - 'yum upgrade -y'
66      preStop:#容器关闭之前运行的任务
67          exec:
68              command: ['service httpd stop']
69      volumeMounts: #详情请见
70          http://blog.csdn.net/liyingke112/article/details/76577520
71          - name: volume #挂载设备的名字，与volumes[*].name 需要对
72             应
73          mountPath: /data #挂载到容器的某个路径下
74          readOnly: True
75      volumes: #定义一组挂载设备
76          - name: volume #定义一个挂载设备的名字
77              #emptyDir: {}
78          hostPath:
79              path: /opt #挂载设备类型为hostPath，路径为宿主机下
80                  的/opt，这里设备类型支持很多种
```

```
apiVersion: v1      版本号
kind: Pod        当前k8s资源的类型 Pod
metadata:       元数据
  name: nginx    pod的名字 nginx
  labels:         label   app: nginx
    app: nginx
spec:           与pod容器相关的详情定义
  containers:    拥有一个container
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

## 搭建K8s集群[无需科学上网]

官网：<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#installing-kubeadm-kubelet-and-kubectl>

GitHub：<https://github.com/kubernetes/kubeadm>

课程中：使用kubeadm搭建一个3台机器组成的k8s集群，1台master节点，2台worker节点

如果大家机器配置不够，也可以使用在线的，或者minikube的方式或者1个master和1个worker

配置要求：

- One or more machines running one of:
  - Ubuntu 16.04+
  - Debian 9+
  - CentOS 7 【课程中使用】
  - Red Hat Enterprise Linux (RHEL) 7
  - Fedora 25+

- HypriotOS v1.0.1+
- Container Linux (tested with 1800.6.0)
- 2 GB or more of RAM per machine (any less will leave little room for your apps)
- 2 CPUs or more
- Full network connectivity between all machines in the cluster (public or private network is fine)
- Unique hostname, MAC address, and product\_uuid for every node. See here for more details.
- Certain ports are open on your machines. See here for more details.
- Swap disabled. You **MUST** disable swap in order for the kubelet to work properly.

## 1.1 版本统一

```
1 Docker      18.09.0
2 ---
3 kubeadm-1.14.0-0
4 kubelet-1.14.0-0
5 kubectl-1.14.0-0
6 ---
7 k8s.gcr.io/kube-apiserver:v1.14.0
8 k8s.gcr.io/kube-controller-manager:v1.14.0
9 k8s.gcr.io/kube-scheduler:v1.14.0
10 k8s.gcr.io/kube-proxy:v1.14.0
11 k8s.gcr.io/pause:3.1
12 k8s.gcr.io/etcd:3.3.10
13 k8s.gcr.io/coredns:1.3.1
14 ---
15 calico:v3.9
```

## 1.2 准备3台centos

大家根据自己的情况来准备centos7的虚拟机。

要保证彼此之间能够ping通，也就是处于同一个网络中，虚拟机的配置要求上面也描述咯。

## 1.3 更新并安装依赖

3台机器都需要执行

```
1 | yum -y update
2 | yum install -y conntrack ipvsadm ipset jq sysstat curl
  | iptables libseccomp
```

## 1.4 安装Docker

根据之前学习的Docker方式[Docker第一节课的笔记中也有这块的说明]

在每一台机器上都安装好Docker，版本为18.09.0

```
1 | 01 安装必要的依赖
2 |     sudo yum install -y yum-utils \
3 |     device-mapper-persistent-data \
4 |     lvm2
5 |
6 |
7 | 02 设置docker仓库
8 |     sudo yum-config-manager --add-repo
9 |     http://mirrors.aliyun.com/docker-
10 |       ce/linux/centos/docker-ce.repo
11 |
12 | 【设置要设置一下阿里云镜像加速器】
13 |     sudo mkdir -p /etc/docker
14 |     sudo tee /etc/docker/daemon.json <<- 'EOF'
15 |     {
16 |     "registry-mirrors": ["这边替换成自己的实际地址"]
17 |     }
18 |     EOF
19 |     sudo systemctl daemon-reload
20 |
21 |
22 | 03 安装docker
23 |
24 |
25 | 04 启动docker
26 |     sudo systemctl start docker && sudo systemctl
  |       enable docker
```

## 1.5 修改hosts文件

(1)master

```
1 # 设置master的hostname，并且修改hosts文件
2 sudo hostnamectl set-hostname m
3
4 vi /etc/hosts
5 192.168.8.51 m
6 192.168.8.61 w1
7 192.168.8.62 w2
```

(2)两个worker

```
1 # 设置worker01/02的hostname，并且修改hosts文件
2 sudo hostnamectl set-hostname w1
3 sudo hostnamectl set-hostname w2
4
5 vi /etc/hosts
6 192.168.8.51 m
7 192.168.8.61 w1
8 192.168.8.62 w2
```

(3)使用ping测试一下

## 1.6 系统基础前提配置

```
1 # (1)关闭防火墙
2 systemctl stop firewalld && systemctl disable firewalld
3
4 # (2)关闭selinux
5 setenforce 0
6 sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/'
7 /etc/selinux/config
8
9 # (3)关闭swap
10 swapoff -a
11 sed -i '/swap/s/^.*$/#/g' /etc/fstab
12
13 # (4)配置iptables的ACCEPT规则
14 iptables -F && iptables -X && iptables -F -t nat &&
15 iptables -X -t nat && iptables -P FORWARD ACCEPT
```

```
15 # (5)设置系统参数
16 cat <<EOF > /etc/sysctl.d/k8s.conf
17 net.bridge.bridge-nf-call-ip6tables = 1
18 net.bridge.bridge-nf-call-iptables = 1
19 EOF
20
21 sysctl --system
```

## 1.7 Installing kubeadm, kubelet and kubectl

### (1)配置yum源

```
1 cat <<EOF > /etc/yum.repos.d/kubernetes.repo
2 [kubernetes]
3 name=Kubernetes
4 baseurl=http://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-e17-x86_64
5 enabled=1
6 gpgcheck=0
7 repo_gpgcheck=0
8 gpgkey=http://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
9 http://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
10 EOF
```

### (2)安装kubeadm&kubelet&kubectl

```
1 yum install -y kubeadm-1.14.0-0 kubelet-1.14.0-0 kubectl-1.14.0-0
```

### (3)docker和k8s设置同一个cgroup

```
1 # docker
2 vi /etc/docker/daemon.json
3   "exec-opts": ["native.cgroupdriver=systemd"], 
4
5 systemctl restart docker
6
7 # kubelet, 这边如果发现输出directory not exist, 也说明是没问题的, 大家继续往下进行即可
8 sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g"
9 /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
10 systemctl enable kubelet && systemctl start kubelet
```

## 1.8 proxy/pause/scheduler等国内镜像

(1)查看kubeadm使用的镜像

kubeadm config images list

可以发现这里都是国外的镜像

```
1 k8s.gcr.io/kube-apiserver:v1.14.0
2 k8s.gcr.io/kube-controller-manager:v1.14.0
3 k8s.gcr.io/kube-scheduler:v1.14.0
4 k8s.gcr.io/kube-proxy:v1.14.0
5 k8s.gcr.io/pause:3.1
6 k8s.gcr.io/etcd:3.3.10
7 k8s.gcr.io/coredns:1.3.1
```

(2)解决国外镜像不能访问的问题

- 创建kubeadm.sh脚本，用于拉取镜像/打tag/删除原有镜像

```
1#!/bin/bash
2
3 set -e
4
5 KUBE_VERSION=v1.14.0
6 KUBE_PAUSE_VERSION=3.1
7 ETCD_VERSION=3.3.10
8 CORE_DNS_VERSION=1.3.1
9
10 GCR_URL=k8s.gcr.io
```

```
11 ALIYUN_URL=registry.cn-
hangzhou.aliyuncs.com/google_containers
12
13 images=(kube-proxy:${KUBE_VERSION}
14 kube-scheduler:${KUBE_VERSION}
15 kube-controller-manager:${KUBE_VERSION}
16 kube-apiserver:${KUBE_VERSION}
17 pause:${KUBE_PAUSE_VERSION}
18 etcd:${ETCD_VERSION}
19 coredns:${CORE_DNS_VERSION})
20
21 for imageName in ${images[@]} ; do
22     docker pull $ALIYUN_URL/$imageName
23     docker tag $ALIYUN_URL/$imageName $GCR_URL/$imageName
24     docker rmi $ALIYUN_URL/$imageName
25 done
```

- 运行脚本和查看镜像

```
1 # 运行脚本
2 sh ./kubeadm.sh
3
4 # 查看镜像
5 docker images
```

- 将这些镜像推送到自己的阿里云仓库【可选，根据自己实际的情况】

```
1 # 登录自己的阿里云仓库
2 docker login --username=xxx registry.cn-
hangzhou.aliyuncs.com
```

```
1#!/bin/bash
2
3 set -e
4
5 KUBE_VERSION=v1.14.0
6 KUBE_PAUSE_VERSION=3.1
7 ETCD_VERSION=3.3.10
8 CORE_DNS_VERSION=1.3.1
9
10 GCR_URL=k8s.gcr.io
11 ALIYUN_URL=xxx
12
```

```
13 images=(kube-proxy:${KUBE_VERSION}
14 kube-scheduler:${KUBE_VERSION}
15 kube-controller-manager:${KUBE_VERSION}
16 kube-apiserver:${KUBE_VERSION}
17 pause:${KUBE_PAUSE_VERSION}
18 etcd:${ETCD_VERSION}
19 coredns:${CORE_DNS_VERSION})
20
21 for imageName in ${images[@]} ; do
22     docker tag $GCR_URL/$imageName $ALIYUN_URL/$imageName
23     docker push $ALIYUN_URL/$imageName
24     docker rmi $ALIYUN_URL/$imageName
25 done
```

运行脚本 sh ./kubeadm-push-aliyun.sh

## 1.9 kube init初始化master

(1)kube init流程

```
1 01-进行一系列检查，以确定这台机器可以部署kubernetes
2
3 02-生成kubernetes对外提供服务所需要的各種证书可对应目录
4 /etc/kubernetes/pki/*
5
6 03-为其他组件生成访问kube-ApiServer所需的配置文件
7     ls /etc/kubernetes/
8         admin.conf  controller-manager.conf  kubelet.conf
9         scheduler.conf
10
11 04-为 Master组件生成Pod配置文件。
12     ls /etc/kubernetes/manifests/*.yaml
13         kube-apiserver.yaml
14         kube-controller-manager.yaml
15         kube-scheduler.yaml
16
17 05-生成etcd的Pod YAML文件。
18     ls /etc/kubernetes/manifests/*.yaml
19         kube-apiserver.yaml
20         kube-controller-manager.yaml
21         kube-scheduler.yaml
22         etcd.yaml
```

```
23 06-一旦这些 YAML 文件出现在被 kubelet 监视  
的/etc/kubernetes/manifests/目录下，kubelet就会自动创建这些  
yaml文件定义的pod，即master组件的容器。master容器启动后，  
kubeadm会通过检查localhost: 6443/healthz这个master组件的健康  
状态检查URL，等待master组件完全运行起来  
24  
25 07-为集群生成一个bootstrap token  
26  
27 08-将ca.crt等 Master节点的重要信息，通过ConfigMap的方式保存在  
etcd中，工后续部署node节点使用  
28  
29 09-最后一步是安装默认插件，kubernetes默认kube-proxy和DNS两个插  
件是必须安装的
```

## (2)初始化master节点

官网：<https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/>

**注意：此操作是在主节点上进行**

```
1 # 本地有镜像  
2 kubeadm init --kubernetes-version=1.14.0 --apiserver-  
advertise-address=192.168.8.51 --pod-network-  
cidr=10.244.0.0/16  
3 【若要重新初始化集群状态：kubeadm reset，然后再进行上述操作】
```

## 记得保存好最后kubeadm join的信息

### (3)根据日志提示

```
1 mkdir -p $HOME/.kube  
2 sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
3 sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

## 此时kubectl cluster-info查看一下是否成功

### (4)查看pod验证一下

等待一会儿，同时可以发现像etc，controller，scheduler等组件都以pod的方式安装成功了

**注意：coredns没有启动，需要安装网络插件**

```
1 | kubectl get pods -n kube-system
```

(5)健康检查

```
1 | curl -k https://localhost:6443/healthz
```

## 1.10 部署calico网络插件

选择网络插件: <https://kubernetes.io/docs/concepts/cluster-administration/addons/>

calico网络插件: <https://docs.projectcalico.org/v3.9/getting-started/kubernetes/>

calico, 同样在master节点上操作

```
1 | # 在k8s中安装calico
2 | kubectl apply -f
   https://docs.projectcalico.org/v3.9/manifests/calico.yaml
3 |
4 | # 确认一下calico是否安装成功
5 | kubectl get pods --all-namespaces -w
```

## 1.11 kube join

记得保存初始化master节点的最后打印信息【注意这边大家要自己的，下面我的只是一个参考】

```
1 | kubeadm join 192.168.0.51:6443 --token
   yu1ak0.2dcecvmpozsy81oh \
2   --discovery-token-ca-cert-hash
   sha256:5c4a69b3bb05b81b675db5559b0e4d7972f1d0a61195f217161
   522f464c307b0
```

(1)在woker01和worker02上执行上述命令

(2)在master节点上检查集群信息

```
1 kubectl get nodes
2
3 NAME STATUS ROLES AGE VERSION
4 master-kubeadm-k8s Ready master 19m v1.14.0
5 worker01-kubeadm-k8s Ready <none> 3m6s v1.14.0
6 worker02-kubeadm-k8s Ready <none> 2m41s v1.14.0
```

## 1.12 再次体验Pod

(1) 定义pod.yml文件，比如pod\_nginx\_rs.yaml

```
1 cat > pod_nginx_rs.yaml <<EOF
2 apiVersion: apps/v1
3 kind: ReplicaSet
4 metadata:
5   name: nginx
6   labels:
7     tier: frontend
8 spec:
9   replicas: 3
10  selector:
11    matchLabels:
12      tier: frontend
13  template:
14    metadata:
15      name: nginx
16      labels:
17        tier: frontend
18  spec:
19    containers:
20      - name: nginx
21        image: nginx
22        ports:
23          - containerPort: 80
24 EOF
```

(2) 根据pod\_nginx\_rs.yaml文件创建pod

```
1 kubectl apply -f pod_nginx_rs.yaml
```

(3) 查看pod

```
1 | kubectl get pods  
2 | kubectl get pods -o wide  
3 | kubectl describe pod nginx
```

| (4)感受通过rs将pod扩容

```
1 | kubectl scale rs nginx --replicas=5  
2 | kubectl get pods -o wide
```

| (5)删除pod

```
1 | kubectl delete -f pod_nginx_rs.yaml
```

## 02 Basic

### 2.1 yaml文件

#### 2.1.1 简介

YAML (IPA: /'jæməl/) 是一个可读性高的语言，参考了XML、C、Python等。

理解：Yet Another Markup Language

后缀：可以是.yml或者是.yaml，更加推荐.yaml，其实用任意后缀都可以，只是阅读性不强

#### 2.1.2 基础

- 区分大小写
- 缩进表示层级关系，相同层级的元素左对齐
- 缩进只能使用空格，不能使用TAB
- "#"表示当前行的注释
- 是JSON文件的超级，两个可以转换
- ---表示分隔符，可以在一个文件中定义多个结构
- 使用key: value，其中":"和value之间要有一个英文空格

#### 2.1.3 Maps

##### 2.1.3.1 简单

```
1 | apiVersion: v1  
2 | kind: Pod
```

---表示分隔符，可选。要定义多个结构一定要分隔

apiVersion表示key，v1表示value，英文":"后面要有一个空格

kind表示key，Pod表示value

也可以这样写apiVersion: "v1"

转换为JSON格式

```
1 {
2   "apiVersion": "v1",
3   "kind": "Pod"
4 }
```

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   labels:
6     app: nginx
```

### 2.1.2.2 复杂

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   labels:
6     app: nginx
```

metadata表示key，下面的内容表示value，该value中包含两个直接的key：name和labels

name表示key，nginx-deployment表示value

labels表示key，下面的表示value，这个值又是一个map

app表示key，nginx表示value

相同层级的记得使用空间缩进，左对齐

转换为JSON格式

```

1  {
2   "apiVersion": "apps/v1",
3   "kind": "Deployment",
4   "metadata": {
5     "name": "nginx-deployment",
6     "labels": {
7       "app": "nginx"
8     }
9   }
10 }

```

## 2.1.4 Lists

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: myapp-pod
5   labels:
6     app: myapp
7 spec:
8   containers:
9     - name: myapp-container01
10    image: busybox:1.28
11    - name: myapp-container02
12    image: busybox:1.28

```

containers表示key，下面的表示value，其中value是一个数组

数组中有两个元素，每个元素里面包含name和image

image表示key，myapp-container表示value

转换成JSON格式

```

1  {
2   "apiVersion": "v1",
3   "kind": "Pod",
4   "metadata": {
5     "name": "myapp",
6     "labels": {
7       "app": "myapp"
8     }
9   },

```

```
10 "spec": {  
11   "containers": [{  
12     "name": "myapp-container01",  
13     "image": "busybox:1.28",  
14   },  
15   {  
16     "name": "myapp-container02",  
17     "image": "busybox:1.28",  
18   }]  
19 }  
20 }
```

## 2.1.5 找个k8s的yaml文件

官网：<https://kubernetes.io/docs/reference/>

```
1 # yaml格式对于Pod的定义:  
2 apiVersion: v1          #必写，版本号，比如v1  
3 kind: Pod               #必写，类型，比如Pod  
4 metadata:                #必写，元数据  
5   name: nginx            #必写，表示pod名称  
6   namespace: default     #表示pod名称属于的命名空间  
7   labels:  
8     app: nginx           #自定义标签名字  
9 spec:  
10   containers:           #必写，pod中容器的详细定义  
11     - name: nginx        #必写，容器列表  
12       image: nginx       #必写，容器名称  
13       ports:  
14         - containerPort: 80    #必写，容器的镜像名称  
15   #表示容器的端口
```

## 2.2 Container

官网：<https://kubernetes.io/docs/concepts/containers/>

### 2.2.1 Docker世界中

可以通过docker run运行一个容器

或者定义一个yml文件，本机使用docker-compose，多机通过docker swarm 创建

### 2.2.2 K8S世界中

同样以一个yaml文件维护， container运行在pod中

## 2.3 Pod

官网：<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>

### 2.3.1 What is Pod

- 1 A Pod is the basic execution unit of a Kubernetes application
- 2 A Pod encapsulates an application's container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run

### 2.3.2 Pod初体验

(1) 创建一个pod的yaml文件，名称为nginx\_pod.yaml

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx-pod
5   labels:
6     app: nginx
7 spec:
8   containers:
9     - name: nginx-container
10    image: nginx
11    ports:
12      - containerPort: 80
```

(2) 根据该nginx\_pod.yaml文件创建pod

```
1 | kubectl apply -f nginx_pod.yaml
```

(3) 查看pod

01 kubectl get pods

| 1 | NAME      | READY | STATUS  | RESTARTS | AGE |
|---|-----------|-------|---------|----------|-----|
| 2 | nginx-pod | 1/1   | Running | 0        | 29s |

02 kubectl get pods -o wide

| 1 | NAME           | READY | STATUS  | RESTARTS | AGE |
|---|----------------|-------|---------|----------|-----|
| 2 | IP             |       | NODE    |          |     |
| 2 | nginx-pod      | 1/1   | Running | 0        | 40m |
|   | 192.168.80.194 |       | w2      |          |     |

03 kubectl describe pod nginx-pod

```
1 Name:          nginx-pod
2 Namespace:     default
3 Priority:      0
4 PriorityClassName: <none>
5 Node:          w2/192.168.0.62
6 Start Time:   Sun, 06 Oct 2019 20:45:35 +0000
7 Labels:        app=nginx
8 Annotations:   cni.projectcalico.org/podIP:
9                 192.168.80.194/32
9                 kubectl.kubernetes.io/last-applied-
10                configuration:
10
11               {"apiVersion": "v1", "kind": "Pod", "metadata": {
12                 "annotations": {}, "labels": {"app": "nginx"}, "name": "nginx-
13                 pod", "namespace": "default"}, "spec": {"c...
14 Status:        Running
15 IP:            192.168.80.194
16 Containers:
17   nginx-container:
18     Container ID: docker://eb2fd0b2906f53e9892e22a6fd791c9ac68fb8e5efce3bbf
19                 94ec12bae96e1984
20     Image:         nginx
21     Image ID:     docker-pullable:/
```

(4)可以发现该pod运行在worker02节点上

于是来到worker02节点， docker ps一下

|   | CONTAINER ID | IMAGE | COMMAND                  | CREATED       |
|---|--------------|-------|--------------------------|---------------|
|   | STATUS       | PORTS | NAMES                    |               |
| 2 | eb2fd0b2906f | nginx | "nginx -g 'daemon off;'" | 6 minutes ago |

不妨进入该容器试试[可以发现只有在worker02上有该容器，因为pod运行在worker02上]:

```
docker exec -it k8s_nginx-container_nginx-pod_default_3ee0706d-e87a-11e9-a904-5254008afee6_0 bash
```

```
1 | root@nginx-pod:/#
```

#### (5)访问nginx容器

```
1 | curl 192.168.80.194      OK, 并且在任何一个集群中的Node上访问都成功
```

#### (6)删除Pod

```
1 | kubectl delete -f nginx_pod.yaml
2 | kubectl get pods
```

### 2.3.3 Storage and Networking

官网：<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#networking>

- Networking

```
1 | Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports.
```

官网：<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#storage>

- Storage

1 A Pod can specify a set of shared storage volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data.

## 性能优化

### JVM

字节码文件,

(1) 装载

a-先找到类文件所在的位置----: 磁盘 全路径 ---->类装载器ClassLoader.find(String name)---->寻找类

b-类文件的信息交给JVM --->类文件字节码流静态存储结构---->JVM里面的某一块区域【方法区】

c-类文件所对应的对象Class---->JVM ---->堆

【解决了】

class文件是用的

保证被加载的类的

b-准备

要为类的静态变量分配内存空间，并将其的值初始化默认值

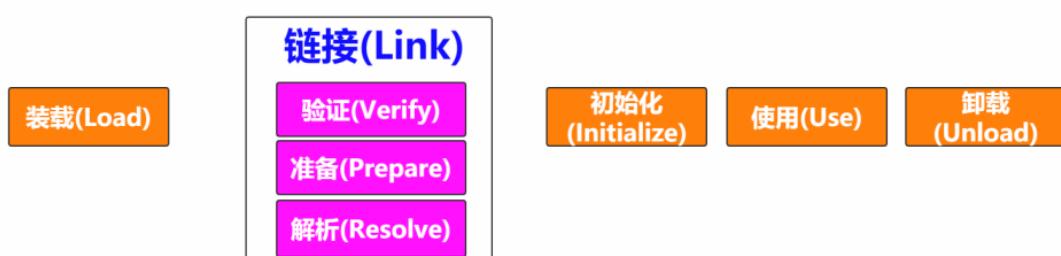
static int a=10; 0

c-解析

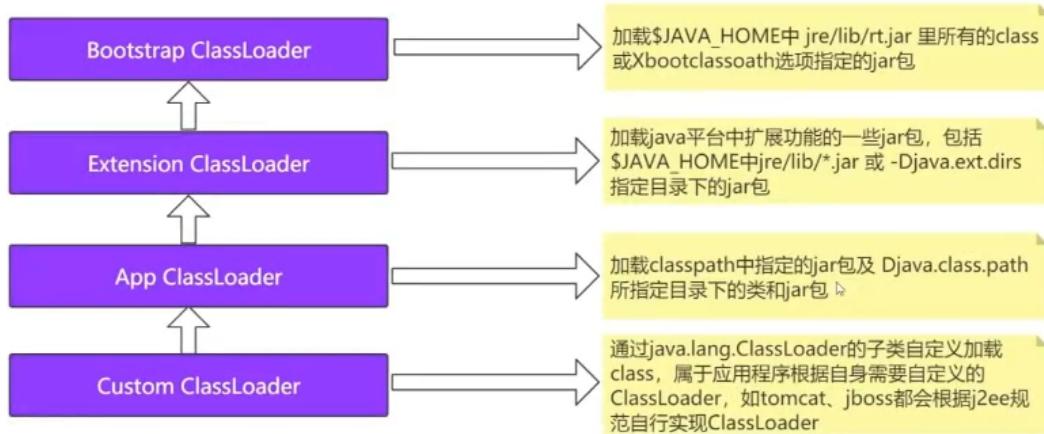
将类中的符号引用转换为直接引用：符号引用，直接引用

地址：String str[aecl flag]=地址是什么 对应内存中的某一个真实的地址指向了

类加载过程



类加载器



## 详聊运行时数据区



## Method Area(方法区)

方法区是各个线程共享的内存区域，在虚拟机启动时创建。用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

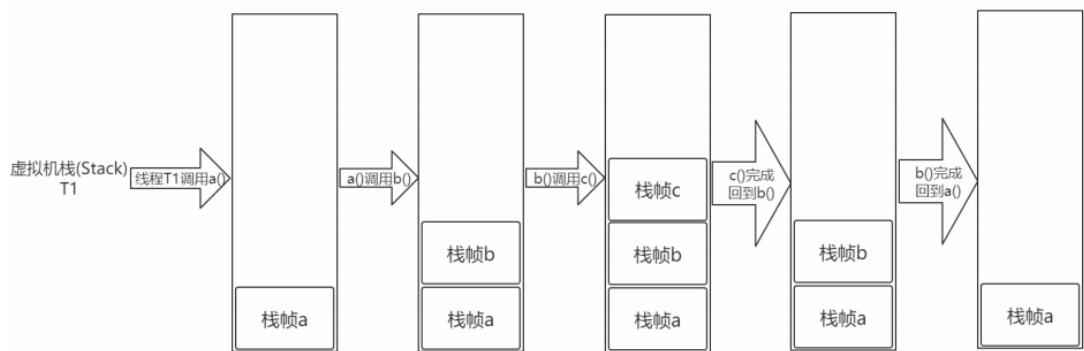
### Heap(堆)

Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。Java对象实例以及数组都在堆上分配。

### Java Virtual Machine Stacks(虚拟机栈)

虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。

## 画图理解栈和栈帧



## 帧栈内容

局部变量,操作数栈,动态链接,方法返回地址



## The pc Register(程序计数器)

程序计数器占用的内存空间很小，由于Java虚拟机的多线程是通过线程轮流切换，并分配处理器执行时间的方式来实现的，在任意时刻，一个处理器只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程需要有一个独立的程序计数器(线程私有)。如果线程正在执行Java方法，

则计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，则这个计数器为空

## Native Method Stacks(本地方法栈)

如果当前线程执行的方法是Native类型的，这些方法就会在本地方法栈中执行。

# Garbage Collect(垃圾回收)

## 1.1 如何确定一个对象是垃圾？

要想进行垃圾回收，得先知道什么样的对象是垃圾。

### 1.1.1 引用计数法

对于某个对象而言，只要应用程序中持有该对象的引用，就说明该对象不是垃圾，如果一个对象没有任何指针对其引用，它就是垃圾。弊端：如果AB相互持有引用，导致永远不能被回收。

### 1.1.2 可达性分析

通过GC Root的对象，开始向下寻找，看某个对象是否可能作为GC Root：类加载器、Thread、虚拟机栈的本地变量表、static成员、常量引用、本地方法栈的变量等。

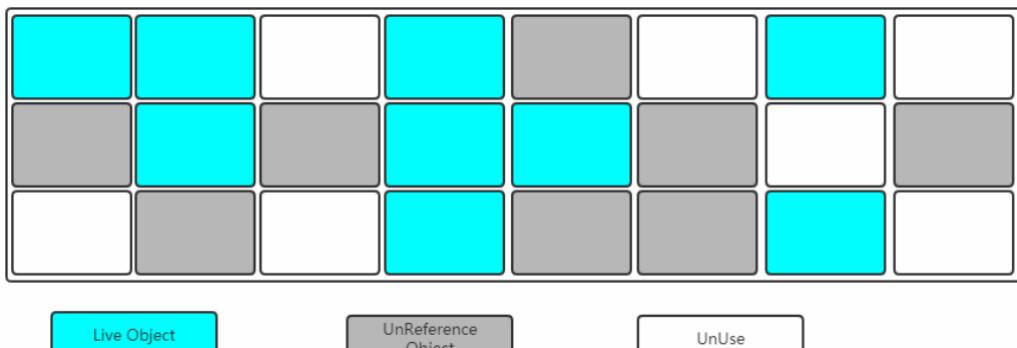
## 1.2 垃圾收集算法

已经能够确定一个对象为垃圾之后，接下来要考虑的就是回收，怎么回收呢？得要有对应的算法，下面聊聊常见的垃圾回收算法。

### 1.2.1 标记清除(Mark-Sweep)

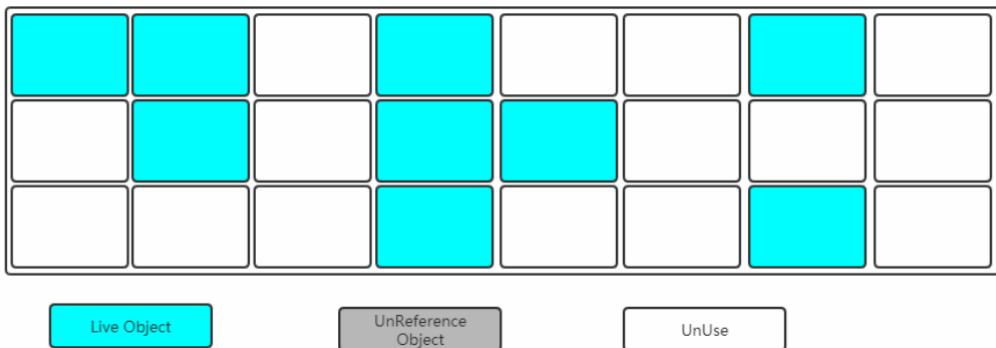
标记

找出内存中需要回收的对象，并且把它们标记出来此时堆中所有的对象都会被扫描一遍，从而才能确定需要回收的对象，比较耗时



## 清除

清除掉被标记需要回收的对象，释放出对应的内存空间



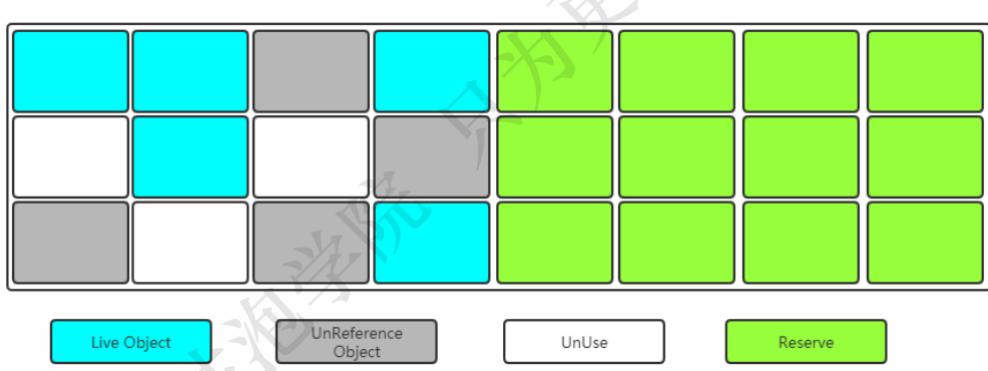
## 缺点:

标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

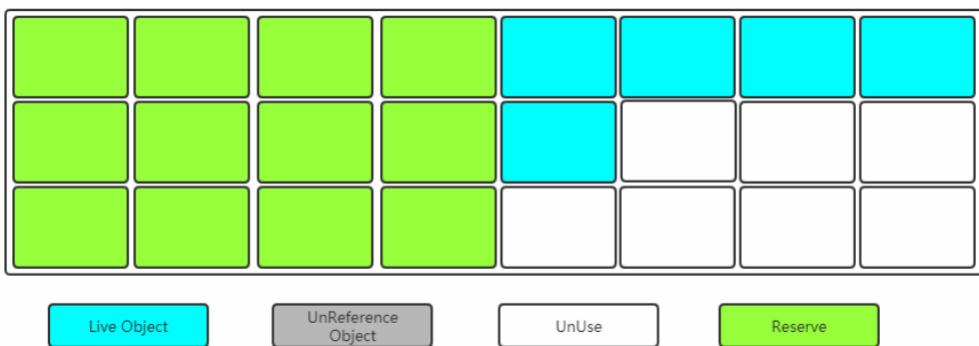
(1)耗时，效率不高 (2)内存不连续

### 1.2.2 复制算法(Copying)

将内存划分为两块相等的区域，每次只使用其中一块，如下图所示：



当其中一块内存使用完了，就将还存活的对象复制到另外一块上面，然后把已经使用过的内存空间一次清除掉。

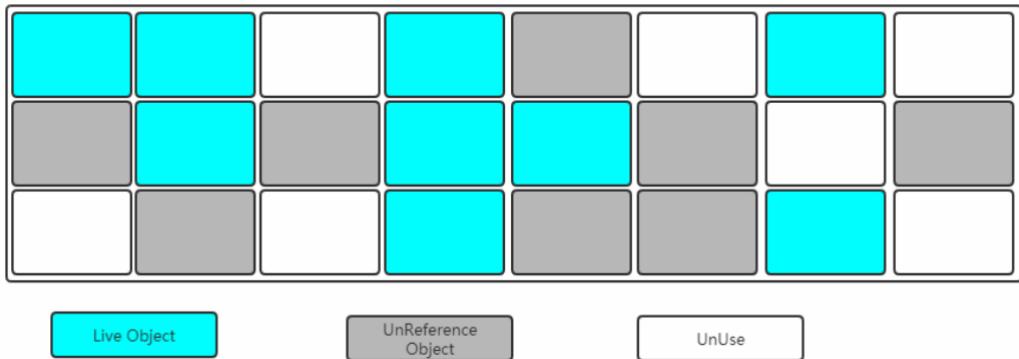


缺点：空间浪费

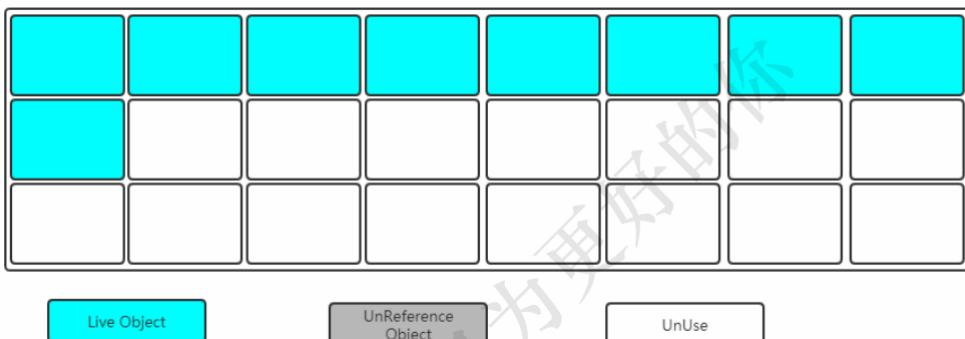
优点:空间连续

### 1.2.3 标记整理(Mark-Compact)

标记过程仍然与"标记-清除"算法一样，但是后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存



让所有存活的对象都向一端移动，清理掉边界意外的内存

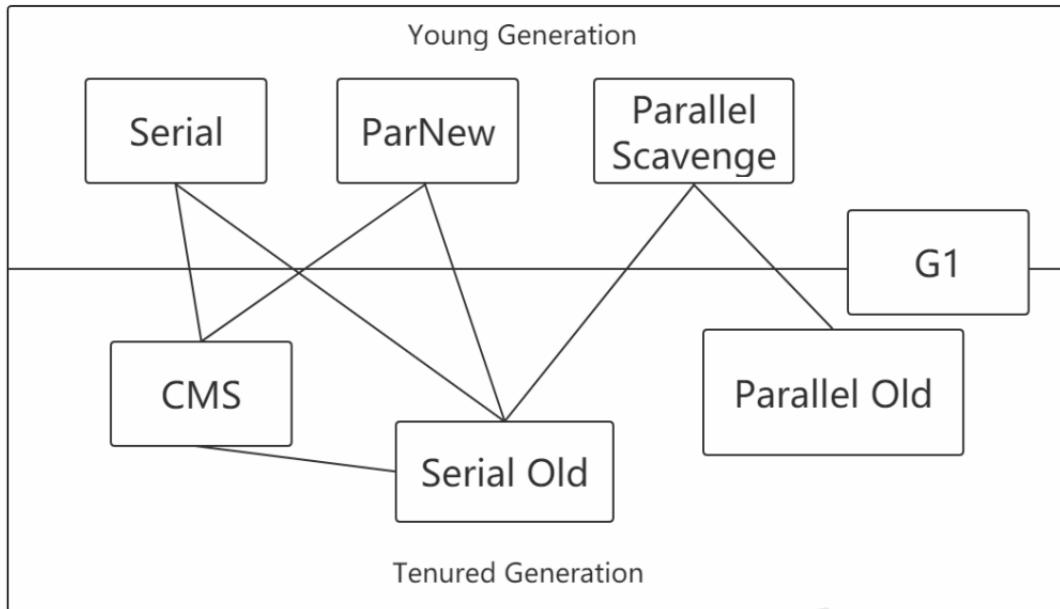


### 1.3 分代收集算法

既然上面介绍了3中垃圾收集算法，那么在堆内存中到底用哪一个呢？

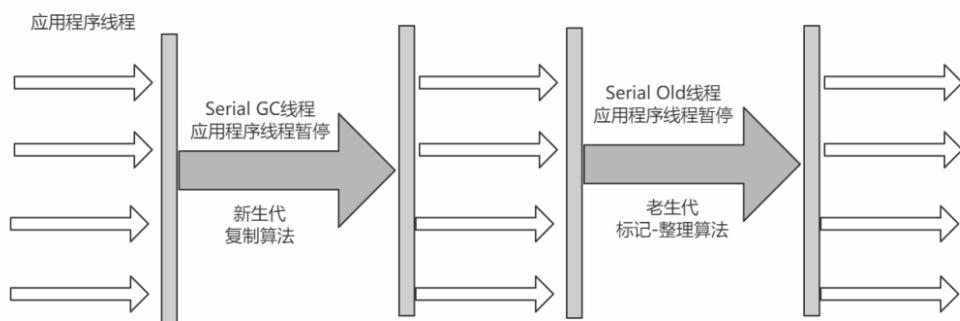
Young区：**复制算法**(对象在被分配之后，可能生命周期比较短，Young区复制效率比较高)

Old区：**标记清除或标记整理**(Old区对象存活时间比较长，复制来复制去没必要，不如做个标记再清理)

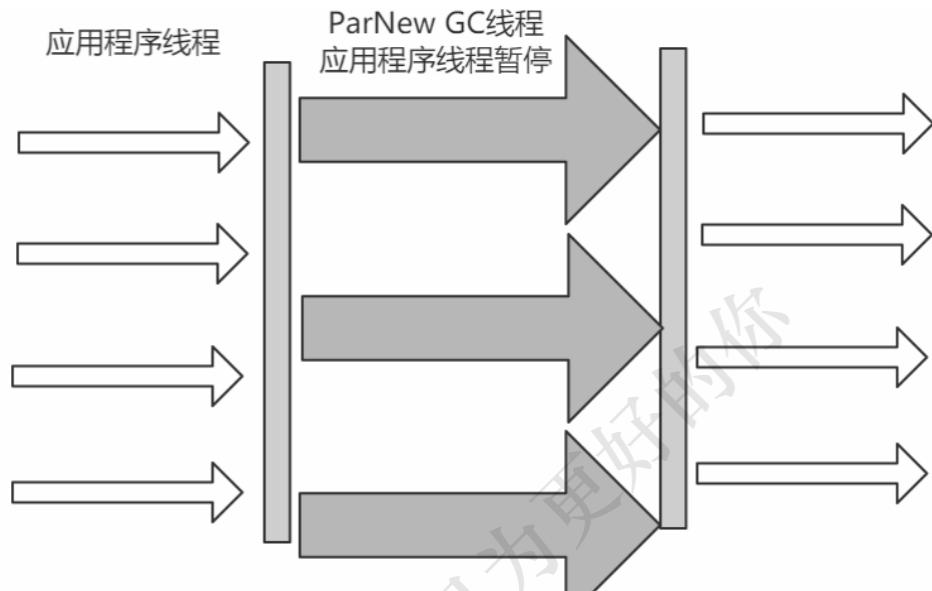


**Serial**配对收集器,用户线程和垃圾收集线程相互切换,收集器是单线程

### 老年代标记-整理



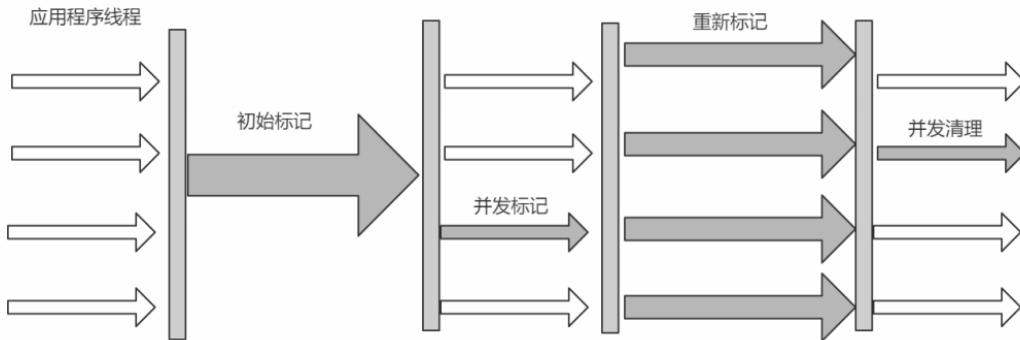
**ParNew** 收集器 垃圾收集器顺应时代,变成多线程,其他和serial一致



**Parallel** 收集器,垃圾收集器顺应时代,变成多线程,其他和serial一致,和ParNew相比更注重用户线程吞吐量

CMS收集器(并发收集器),可以让用户线程和垃圾收集线程并发,比较关注停顿时间(降低了吞吐量),老年代标记-清除算法

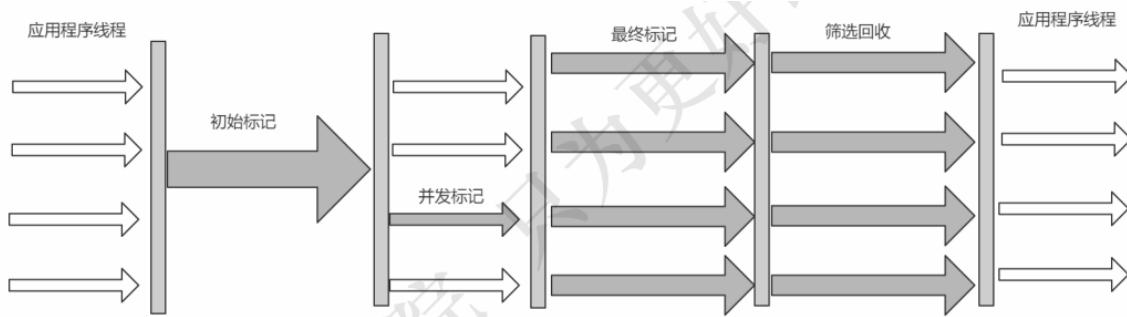
初始标记,并发标记,重复标记,并发清理



G1,元空间,还是分代的,但是都是使用基本的元空间,设定固定大小的储存空间

目标:在延迟可控的情况下,尽可能提升吞吐量

JDK 7开始使用, JDK 8非常成熟, JDK 9默认的垃圾收集器, 适用于新老生代,标记-整理



设置赛选回收的时间,满足停顿目标

- (1) 50%以上的堆被存活对象占用
- (2) 对象分配和晋升的速度变化非常大
- (3) 垃圾回收时间比较长

```
1 (1) 串行 -XX: +UseSerialGC -XX: +UseSerialOldGC
2 (2) 并行(吞吐量优先): -XX: +UseParallelGC -XX:
+UseParallelOldGC
3 (3) 并发收集器(响应时间优先) -XX: +UseConcMarkSweepGC -XX:
+UseG1GC哈
```

使用选择



### (1) GC收集器：停顿时间和吞吐量

停顿时间【set pause time】：垃圾收集器进行垃圾回收Client执行相应的时间--->很好的体验--->和用户交互比较多的场景，web程序

吞吐量：运行用户代码时间/(运行用户代码时间+垃圾收集时间)： 用户代码执行占用 CPU资源的时间比较大，跑任务，运算的任务

Java：1995年 并没有意识到这个语言将来会多更多的web开发，没有意识到要进行更多的web交互场景，停顿时间要比较小的场景，停顿时间 Server-Client 停止交互 ---->JDK往上 web开发 低停顿

极致：干脆没有停顿时间

停顿时间是要追求的：---->JDK11

**停顿时间小：**CMS 20ms、G1[set pause time,not strict 15ms] 使用于web应用 ---->**并发类的收集器**

**吞吐量优先：**Parallel Scanvent+Parallel Old ----->**并行类的收集器**

**串行收集：**Serial 和 Serial Old ----->内存比较小，嵌入式的设备

## JVM参数

标准参数:不会随JDK版本变化而变化,java -version/-help

-X,非标准参数,jdk版本变动,-Xint

-XX参数,使用最多

a-boolean 类型

-xx[+/-]name 启动或停止

b非BOOlean

-XX:name=value

-XX:MaxHeapSize

其他参数

-Xms1000等价于-XX:InitialHeapSize=1000

-Xmx1000等价于-XX:MaxHeapSize=1000

-Xss100等价于-XX:ThreadStackSize=100

常用参数

| 参数                                                                                                | 含义                 | 说明                                                                   |
|---------------------------------------------------------------------------------------------------|--------------------|----------------------------------------------------------------------|
| -XX:CICompilerCount=3                                                                             | 最大并行编译数            | 如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加VM崩溃的可能                             |
| -XX:InitialHeapSize=100M                                                                          | 初始化堆大小             | 简写-Xms100M                                                           |
| -XX:MaxHeapSize=100M                                                                              | 最大堆大小              | 简写-Xmx100M                                                           |
| -XX:NewSize=20M                                                                                   | 设置年轻代的大小           |                                                                      |
| -XX:MaxNewSize=50M                                                                                | 年轻代最大大小            |                                                                      |
| -XX:OldSize=50M                                                                                   | 设置老年代大小            |                                                                      |
| -XX:MetaspaceSize=50M                                                                             | 设置方法区大小            |                                                                      |
| -XX:MaxMetaspaceSize=50M                                                                          | 方法区最大大小            |                                                                      |
| -XX:UseParallelGC                                                                                 | 使用UseParallelGC    | 新生代，吞吐量优先                                                            |
| -XX:UseParallelOldGC                                                                              | 使用UseParallelOldGC | 老年代，吞吐量优先                                                            |
| -XX:UseConcMarkSweepGC                                                                            | 使用CMS              | 老年代，停顿时间优先                                                           |
| -XX:+UseG1GC                                                                                      | 使用G1GC             | 新生代，老年代，停顿时间优先                                                       |
| -XX:NewRatio                                                                                      | 新老生代的比例            | 比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5                        |
| -XX:SurvivorRatio                                                                                 | 两个S区和Eden区的比例      | 比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8，也就是说一个S占整个新生代的1/10         |
| -XX:+HeapDumpOnOutOfMemoryError                                                                   | 启动堆内存溢出打印          | 当JVM堆内存发生溢出时，也就是说OOM，自动生成dump文件                                      |
| -XX:HeapDumpPath=heap.hprof                                                                       | 指定堆内存溢出打印目录        | 表示在当前目录生成一个heap.hprof文件                                              |
| -XX:+PrintGCDetails-XX:+PrintGCTimeStamps-XX:+PrintGCDateStampsXloggc:\$CATALINA_HOME/logs/gc.log | 打印出GC日志            | 可以使用不同的垃圾收集器，对比查看GC情况                                                |
| -Xss128k                                                                                          | 设置每个线程的堆栈大小        | 经验值是3000~5000最佳                                                      |
| -XX:MaxTenuringThreshold=6                                                                        | 提升年老代的最大临界值        | 默认值为15                                                               |
| -XX:InitiatingHeapOccupancyPercent                                                                | 启动并发GC周期时堆内存使用占比   | G1之类的垃圾收集器用它来触发并发GC周期，基于整个堆的使用率，而不仅是某一代内存使用比，值为0则表示“一直执行GC循环”，默认值为45 |
| -XX:G1HeapWastePercent                                                                            | 允许浪费堆空间的占比         | 默认实10%，如果并发标记可回收的空间小于10%，则不会触发MixedGC。                               |

| 参数                                    | 含义                                                                 | 说明                                                                        |
|---------------------------------------|--------------------------------------------------------------------|---------------------------------------------------------------------------|
| -XX:MaxGCPauseMillis=200ms            | G1最大停顿时间                                                           | 暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数调优是一个持续的过程，逐步调整到最佳状态。 |
| -XX:ConcGCThreads=n                   | 并发垃圾收集器使用的线程数量                                                     | 默认值随JVM运行的平台不同而不同                                                         |
| -XX:G1MixedGCLiveThresholdPercent=65  | 混合垃圾回收周期中要包括的旧区域设置占用率阈值                                            | 默认占用率为65%                                                                 |
| -XX:G1MixedGCCountTarget=8            | 设置标记周期完成后，对存活数据上限为 G1MixedGCLiveThresholdPercent 的旧区域执行混合垃圾回收的目标次数 | 默认8次混合垃圾回收，混合回收的目标是要控制在此目标次数以内                                            |
| -XX:G1OldCSetRegionThresholdPercent=1 | 描述Mixed GC时， Old Region被加入到CSet中                                   | 默认情况下， G1只把10%的Old Region加入到CSet中                                         |

## 常用命令

**jps** 查看java进程

**jinfo** (1)实时查看和调整JVM配置参数

(2)jinfo -flflag name PID 查看某个java进程的name属性的值

(3)修改参数只有被标记为**manageable**的flflags可以被实时修改,jinfo -flflag [+ | -] PID,jinfo -flflag PID

jstat:查看虚拟机性能统计信息

(3)查看垃圾收集信息 jstat -gc PID 1000 10

**jstack** jstack PID 查看线程堆栈信息

jmap 生成快照

dump出堆内存相关信息

jmap -dump:format=b,fifile=heap.hprof PID

**要是在发生堆内存溢出的时候，能自动dump出该文件就好了一般在开发中，JVM参数可以加上下面两句，这样内存溢出时，会自动dump出该文件**

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof关于dump下来的文件

手动jmap -dump:format=b,file=heap.hprof 448082

一般dump下来的文件可以结合工具来分析

常用工具

JConsole工具是JDK自带的可视化监控工具。查看java应用程序的运行概况、监控堆信息、永久区使用情况、类加载情况；

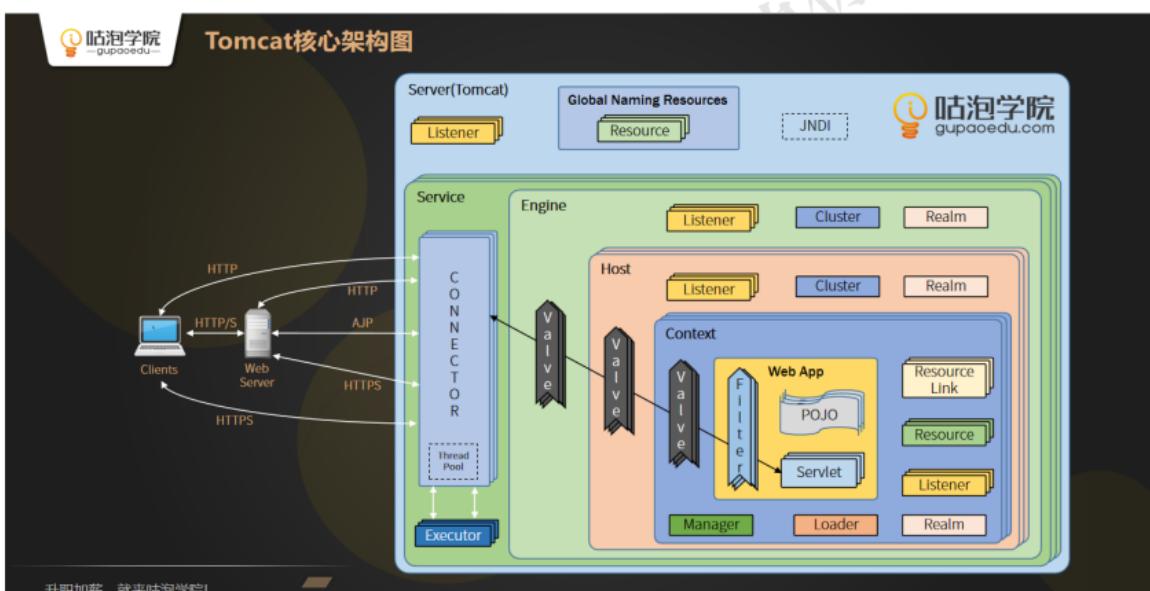
**Arthas**是Alibaba开源的Java诊断工具，采用命令行交互模式，是排查jvm相关问题的利器。

**MAT** 分析gc日志文件

## Tomcat

tomcat 原理,就是封装severlet,先建立list集合,然后根据connect.socket绑定端口,根据指定协议转换为requestserverlet和responseserverlet对象操作,实际就是输入流和输出流转换,

### tomcat架构图



Resource:配置文件,realm存放用户登录账号密码,manager生成管理sessionID

Connector: 主要负责处理Socket连接，以及Request与Response的转化

每个context对应一个war

### tomcat三种部署方式

直接将 web 项目文件（一般是复制生成的war包）复制到tomcat的webapps目录中。

## /修改Server.xml文件

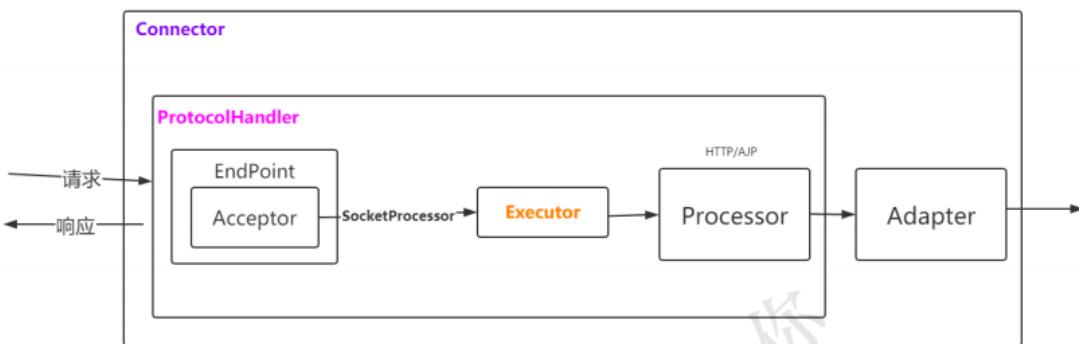
```
<Context path="/jfinal_demo" docBase="F:workjfinal_demoWebRoot"  
reloadable ="true" debug="0" privileged="true"
```

- path：是访问时的根地址，表示访问的路径，可以自定义，如上述例子中，访问该应用程序地址如下：[http://localhost:8080/jfinal\\_demo](http://localhost:8080/jfinal_demo)；
- docbase：表示应用程序的路径，docBase可以使用绝对路径，也可以使用相对路径，相对路径相对于webapps；
- reloadable：表示可以在运行时在classes与lib文件夹下自动加载类包。这个属性在开发阶段通常都设为true，方便开发；在发布阶段应该设置为false，提高应用程序的访问速度；

tomcat的Engine和Host都有默认的名称，所以默认的存储路径为：

tomcat/conf/Catalina/localhost/\*.xml (xml文件可以随意命名)

对Tomcat的侵入性最小，并且如果想取消部署，直接删除xml配置文件即可。



## EndPoint

监听通信端口，是对传输层的抽象，用来实现 TCP/IP 协议的。对应的抽象类为 AbstractEndPoint，有很多实现类，比如NioEndPoint, JIoEndPoint等。其中有两个组件，一个是Acceptor，另外一个是SocketProcessor。 Acceptor用于监听Socket连接请求， SocketProcessor用于处理接收到的Socket请求。

## Processor

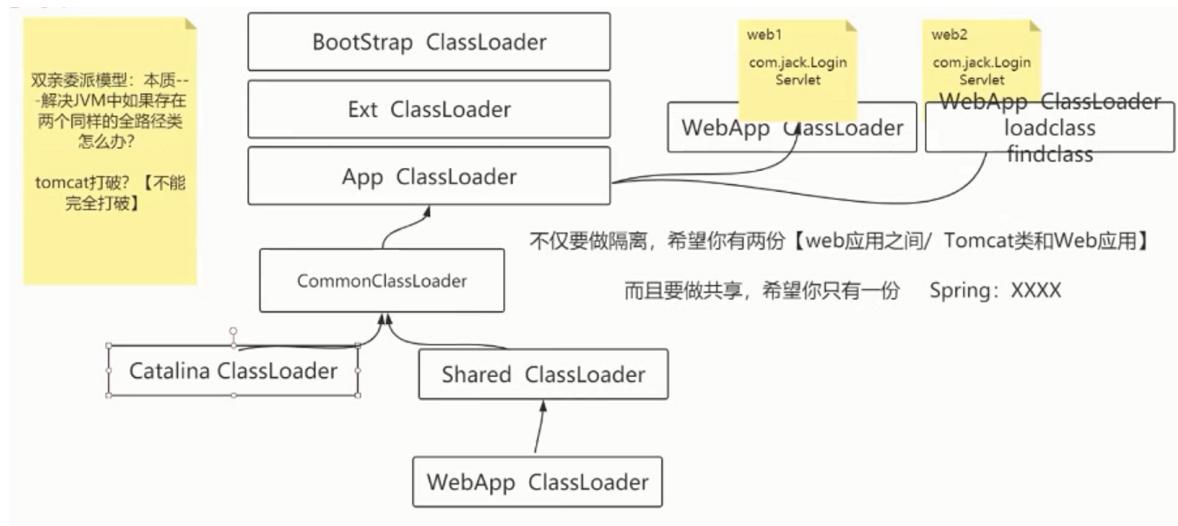
Processor是用于实现HTTP协议的，也就是说Processor是针对应用层协议的抽象。 Processor接受来自EndPoint的Socket，然后解析成Tomcat Request和Tomcat Response对象，最后通过Adapter提交给容器。 对应的抽象类为 AbstractProcessor，有很多实现类，比如AjpProcessor、Http11Processor等。

## Adapter

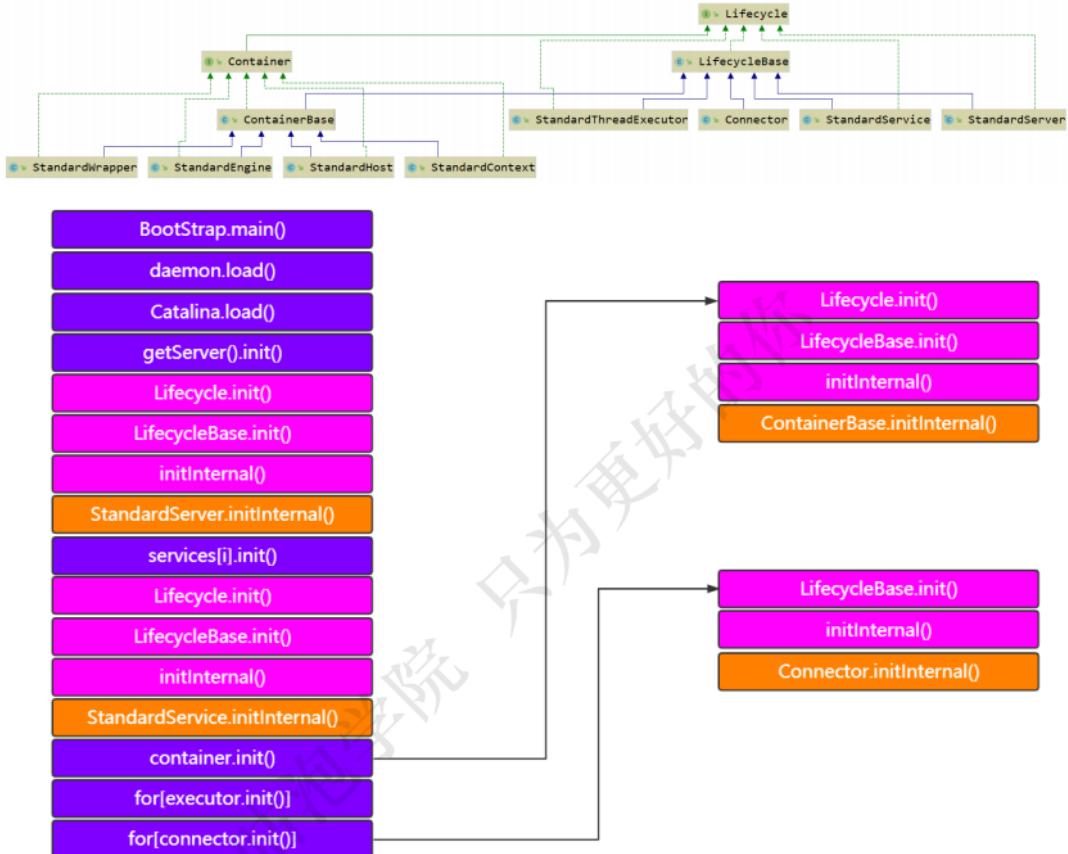
ProtocolHandler接口负责解析请求并生成 Tomcat Request 类。需要把这个 Request 对象转换成 ServletRequest。Tomcat 引入CoyoteAdapter，这是适配器模式的经典运用，连接器调用 CoyoteAdapter 的 service 方法，传入的是 Tomcat Request 对象，CoyoteAdapter 负责将 Tomcat Request 转成 ServletRequest，再调用容器的 service 方法

Container：包括Engine、Host、Context和Wrapper，主要负责内部的处理以及Servlet的管理

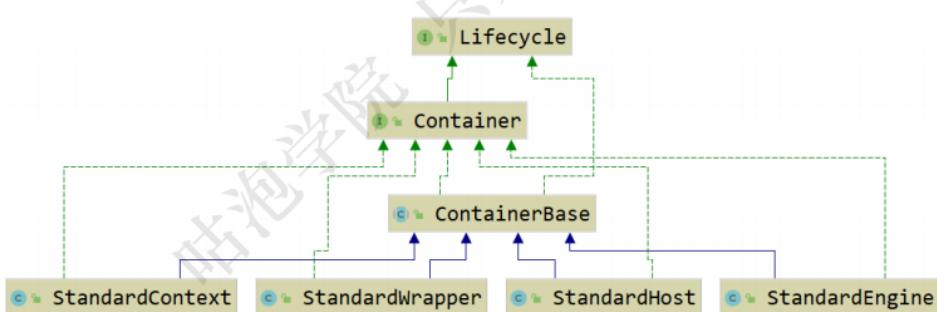
## tomcat自定义类加载器



## tomcat启动流程



ContainerBase的类关系图



## tomcat性能优化

Tomcat性能指标：吞吐量、响应时间、错误数、线程池、CPU、内存等。使用jmeter进行压测，然后观察相关指标

- 1 01 查看tomcat进程
- 2 pid ps -ef | grep tomcat
- 3 02 查看进程的信息
- 4 cat /proc/pid/status
- 5 03 查看进程的cpu和内存
- 6 top -p pid
- 7 工具查询
- 8 jconsole、jvisualvm、arthas、psi-probe等

# git工具基本使用

## 上传到git

1.在远程创建仓库

2.创建本地仓库并配置 .gitignore 上传忽略文件

```
1 git init // 初始化git仓库
2 git add . // 提交所有的文件
3 git commit -m xxxx // 暂存区的代码提交到远程
    仓库 xxx写提交说明
4 git remote add origin xxx // 连接远程仓库 xxx为远程
    仓库git地址
5 git push -u origin master // 第一次本地仓库推送到远
    程仓库
6 12345
```

## 查询操作记录

```
1 git status // 查看当前项目的状态
2 git log // 查看之前提交的记录
3 git log --author='wankcn' // 查看某人提交的代码 若无
    显示 表明未提交代码或没author这个人
4 123
```

## 配置用户名和邮箱

git config --global 是指全局配置

```
1 git config --global user.name 'xxx' // 配置用户名
2 git config --global user.name 'xx@xx.com' // 配置邮箱
3 git config --global --list // 检查当前配
    置的用户名和邮箱是否成功
4 123
```

## 修改或新增项目中的文件

```
1 git add xxx xxx // 本地文件添加到暂存区 多  
    个文件之间用空格隔开  
2 git commit -m xxx // 将修改的文件添加到远程仓库  
    库  
3 12
```

## 删除文件

### 1. 手动删除文件

在文件夹中删除文件以后执行以下命令

```
1 git add . // 先把删除后剩余的所有文  
    件提交到暂存区  
2 git commit -m xxx // 提交到远程仓库并注释  
3 12
```

### 2. 命令行删除文件

```
1 git rm xxx // 删除xxx文件  
2 git add . // 剩余文件加到暂存区  
3 git commit -m xxx // 提交到远程仓库并注释  
4 123
```

## 文件重命名

### 1. 手动重命名文件

假设工程里有一个文件 `111.txt` 被改成了 `222.txt`，执行下面的命令

```
1 git add 222.txt  
2 git rm 111.txt  
3 git commit -m xxx // 提交到远程仓库并注释  
4 123
```

### 2. 命令行重命名文件

假设把文件 `333.txt` 被改成了 `444.txt`

```
1 git mv 333.txt 444.txt // 先改动之前的名字 后改动  
    之后的名字  
2 git commit -m xxx // 提交到远程仓库并注释  
3 12
```

# 移动文件

## 1. 使用重命名来移动文件

将文件 111.txt 移动到文件夹 files 中

```
1 | git mv 111.txt files          // 移动的文件 移动的位置  
2 | git commit -m xxx            // 提交到远程仓库并注释  
3 | 12
```

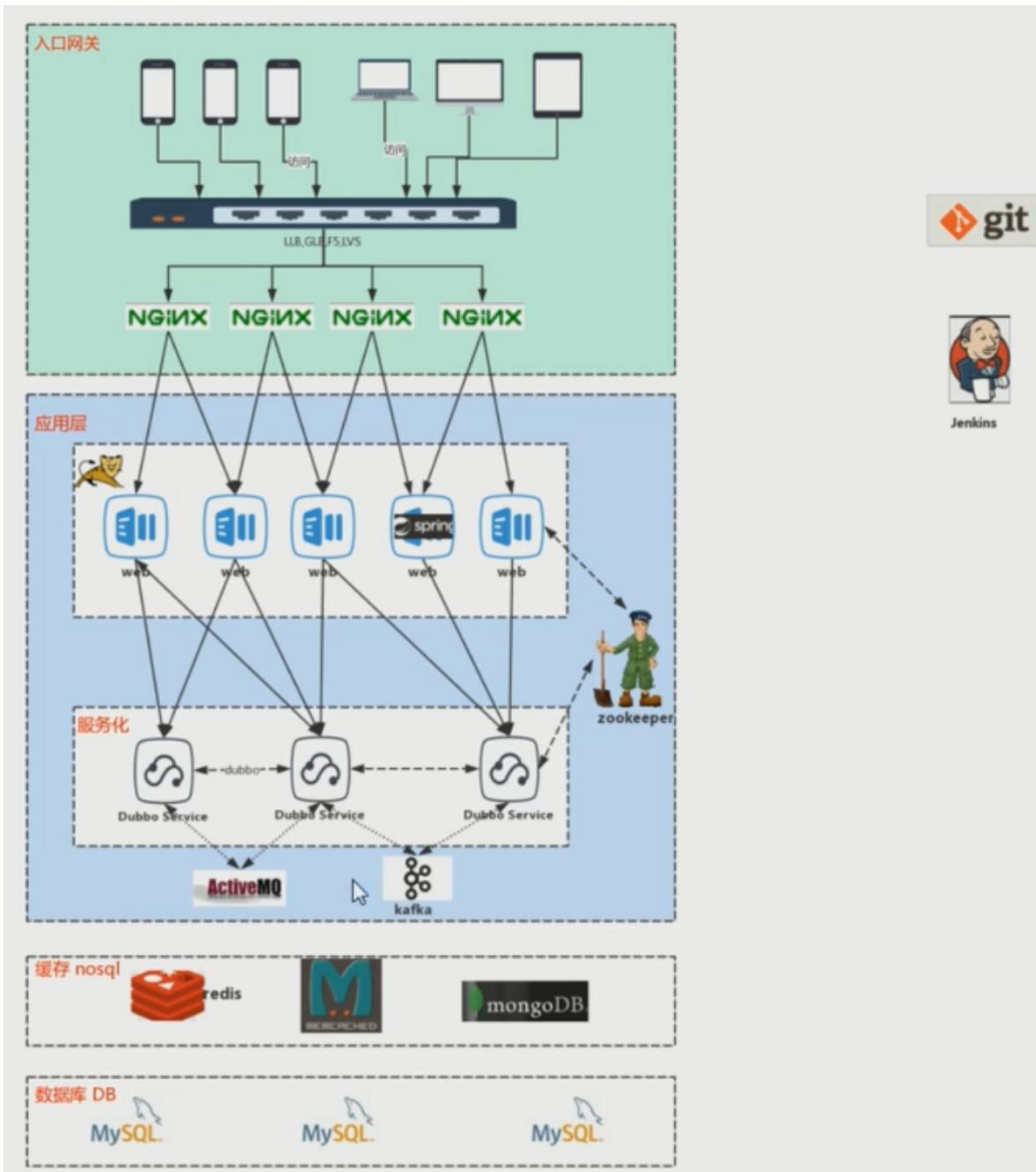
## 2. 移动文件并重命名

将文件 111.txt 移动到文件夹 files 中并重命名为 222.txt

```
1 | git mv 111.txt files/222.txt      // 移动的文件 移动的位置/  
   | 重命名  
2 | git commit -m xxx                // 提交到远程仓库并注释
```

# MongoDB

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写



## memory 内存引擎

NoSQL最大的特点：

- 1、默认支持分布式（内置分布式解决方案）
- 2、高性能，高可用性和可伸缩性

在NoSQL界，

MongoDB是一个最像关系型数据库的非关系型数据库

## MongoDB应用场景

### 适用范围

- 1) 网站实时数据: 例如: 日志、Timeline、用户行为 (代替方案: 用日志)
- 2) 数据缓存: 缓存的数据, 它一定是临时的 (关系型数据有一份已经持久化)

3) 大尺寸、低价值数据存储: 搜索引擎的图片文件、视频文件(结构化),  
一份存磁盘、一份存Mongo

4) 高伸缩性场景: 机器可以任意的增减

5) 对象或JSON数据存储: 完全可以选择用Redis

### 不适用范围

1) 高度事务性系统: 例如: 金融系统的核心数据高机密的用户数据 (只能选择传统关系型数据库)

2) 传统的商业智能应用: 结构化查询要求非常高, 经常做关联查询统计 (如果都是单表查询, 用Java程序来实Map,List (id\_az\_a)) MongoDB 4.0 支持事务操作 (分布式事务的一种解决方案)

学习, 不要老是找乱七八糟的博客看不懂, 给大家补软肋 (新东方英语)

mongo 客户端

mongos 路由器

mongod 数据存储

zookeeper 分布式协调 (路由)

把整个集群下面的所有已经注册机器的信息, 人手持一份 (区块链)

leader选取 (leader不存配置信息, 监控)

套路差不多

mongos, 协调所有的mongod

统一管理配置中心

Master

副本集, 认为就是一台存储数据的机器 (mongo) ,

一个副本集数据一定是一个完整的整体

Image镜像

高可用: 把一个服务部署多份 (一台坏了, 另一台可以顶上)

M-S

M-A-S

数据热点:

数据的平均存储问题 (传一个视频，切成四份分别存到四张表里面去)

每张表存储的极限 (遇到瓶颈)

类似于Map (拆分) Reduce (归并) (Hadoop里面)

怎么切，那是策略

P2P? 原来要1个小时，只需要1分钟

分表、分区?

微观的维度 (你看不到的一个维度)

chunk (块) --> shard(片) --> Replica Set(副本) --> Data(数据)

宏观的维度 (你能看到的)

Field (字段) --> Document(文档) --> Collection(集合) --> DataBase (数据库)

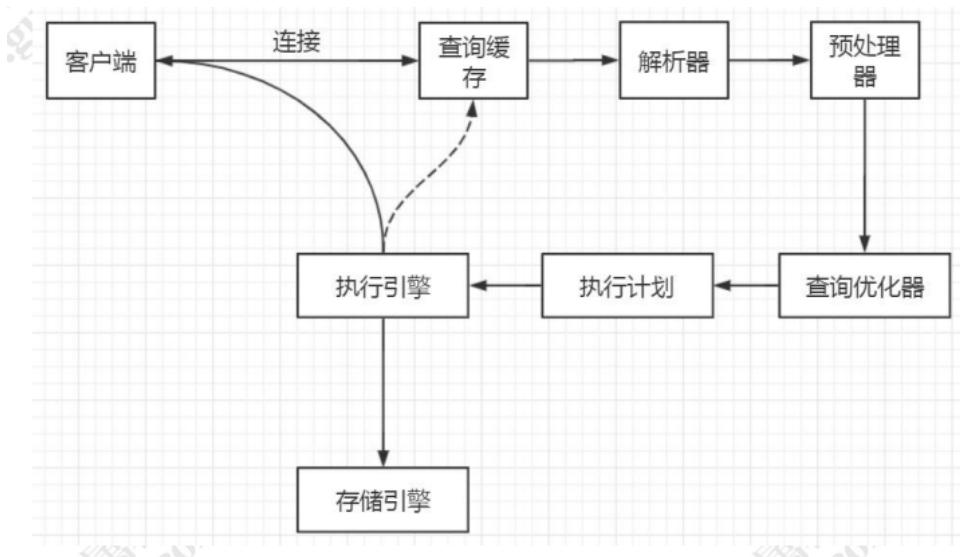
官网java API学习

<https://mongodb.github.io/mongo-java-driver/4.2/driver/getting-started/installation/>

## mysql

---

## 一条sql查询过程



## 连接

mysql支持长连接(默认8h)和短连接,半双工(类似对讲机)  
TCP/IP 协议,mysql -h192.168.8.211 -uroot -p123456  
Unix Socket (/var/lib/mysql/mysql.sock) show global status like  
'Thread%';查看连接数  
在 5.7 版本中默认是 151 个, 最大可以设置成 16384 (2^14) show 的参数说明:

- 1、级别: 会话 session 级别 (默认) ; 全局 global 级别
- 2、动态修改: set, 重启后失效; 永久生效, 修改配置文件/etc/my.cnf

更多参数<https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html>

## 缓存

缓存默认关闭,原因:sql必须严格一致,数据发生更新都会清掉缓存.  
一般解决办法,交给ORM框架用缓存,独立服务redis缓存

## 解析预处理

判断语法是否有误,构架解析树判断字段是否存在

## 查询优化器

是根据解析树生成不同的执行计划 (Execution Plan)择一种最优的执行计划,优化器无法每次最优,建议sql质量  
show status like 'Last\_query\_cost';查看优化器

## 查看执行计划

EXPLAIN select name from user where id=1;

## 存储引擎

sql无法拆分,max=4m,所以要有limit分页要求,

如果对数据**一致性要求比较高**, 需要事务支持, 可以选择 InnoDB。 支持读写并发, 写不阻塞读 (MVCC) 。

如果数据**查询多更新少**, 对查询性能要求比较高, 可以选择 MyISAM。 表锁,无事务

如果需要一个用于查询的临时表, 可以选择 Memory。

如果所有的存储引擎都不能满足你的需求, 并且技术能力足够, 可以根据官网内部

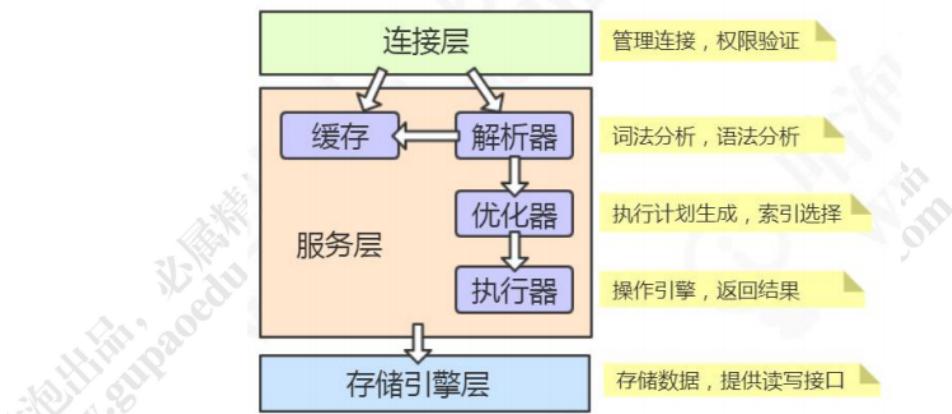
手册用 C 语言开发一个存储引擎：<https://dev.mysql.com/doc/internals/en/custom-engine.html>

## 执行引擎

sql查询执行过程,最后执行引擎调用存储引擎api返回出去,不同功能的存储引擎实现的 API 是相同的。

sql查询执行过程,最后执行引擎调用存储引擎api返回出去

## mysql架构图



## 连接层

MySQL 服务器 3306 端口，管理所有的连接，验证客户端的身份和权限，这些功能就在连接层完成。

## 服务层

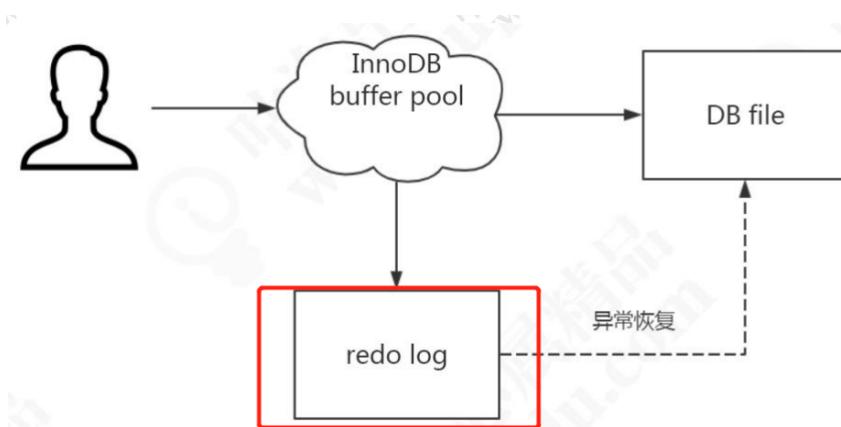
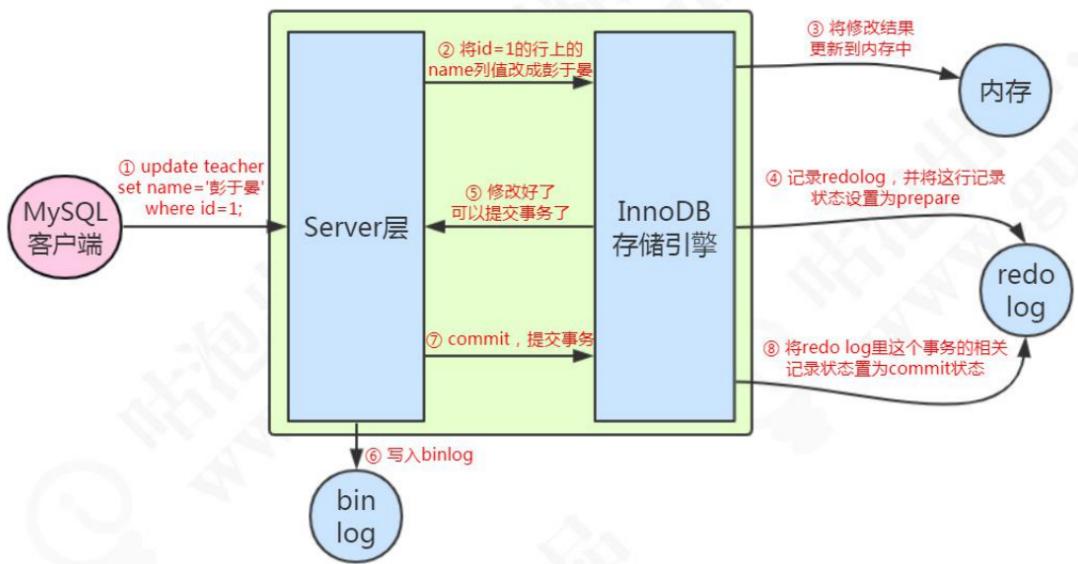
连接层会把 SQL 语句交给服务层，这里面又包含一系列的流程：

比如查询缓存的判断、根据 SQL 调用相应的接口，对我们的 SQL 语句进行词法和语法的解析（比如关键字怎么识别，别名怎么识别，语法有没有错误等等）。然后就是优化器，MySQL 底层会根据一定的规则对我们的 SQL 语句进行优化，最后再交给执行器去执行

## 存储引擎

存储引擎就是我们的数据真正存放的地方，在 MySQL 里面支持不同的存储引擎。再往下就是内存或者磁盘

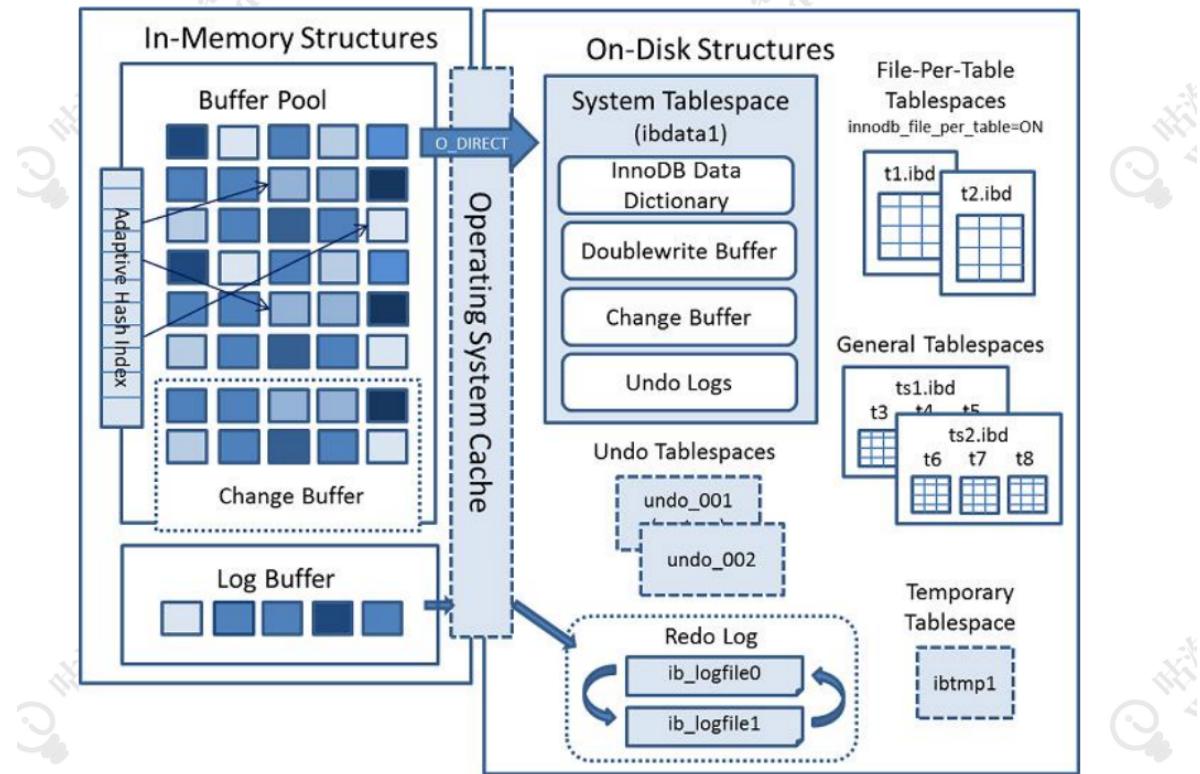
## 一条更新 SQL 是如何执行的



## 缓冲池 Buffer Pool

InnoDB 的数据都是放在磁盘上的，最小的逻辑单位叫做页（索引页和数据页）。使用了一种缓冲池的技术，把磁盘读到的页放到一块内存区域里面。这个内存区域就叫 Buffer Pool。

## InnoDB 内存结构和磁盘结构



Buffer Pool 主要分为 3 个部分： Buffer Pool、 Change Buffer、 Adaptive Hash Index，另外还有一个 (redo) log buffer

Buffer Pool 缓存的是页面信息，包括数据页、索引页。默认大小是 128M (134217728 字节)，可以调整, LRU策略管理缓冲池

**Change Buffer 写缓冲**, Change Buffer 占 Buffer Pool 的比例，默认 25%。

**Adaptive Hash Index**: 自适应hash索引, 在查找非聚集索引的时候，要遍历非聚集索引B+树，然后找到相应主键，再根据主键去聚集索引树查找

当发现某一棵非聚集索引树中的某页已经成为热点页了，就可以对其建立哈希表，下次访问就可以直接通过哈希表查找 (热点,缓存差不多)

innodb\_adaptive\_hash\_index 是否开启, 哈希索引只能用来搜索等值的查询，不可以进行范围查找

**(redo) log buffer**, 先写日志再持久化, 保证事务的永久性(用于恢复数据), 对应于 /var/lib/mysql/ 目录下的 ib\_logfile0 和 ib\_logfile1，每个 48M

顺序io提升吞吐量,

特点

- 1、 redo log 是 InnoDB 存储引擎实现的，并不是所有存储引擎都有
- 2、 不是记录数据页更新之后的状态，而是记录这个页做了什么改动，属于物理日志
- 3、 redo log 的大小是固定的，前面的内容会被覆盖

**内存满了怎么处理** LRU算法, 最近最少使用算法, 内存缓存数据满了后用LRU算法, redo log为有序io

缓存防止数据丢失,redo log 成功后处于准备状态,bin log成功有在提交,符合分布式数据一致性的二阶段提交策略

## 磁盘结构

InnoDB 系统表空间包含 InnoDB **数据字典和双写缓冲区**, **Change Buffer** 和 **Undo Logs**, 如果没有指定 file-per-table, 也包含**用户创建的表和索引数据**

双写缓冲 (InnoDB 的一大特性):InnoDB 页大小一般为 16K, 操作系统页大小为 4K, InnoDB 的页写入到磁盘时, 一个页需要分 4 次写, 存有副本可以恢复, 和 redo log ,类似一部分内存,一部分磁盘,io顺序写,这个在relog 之前操作,实现了数据页的可靠性

undo log (撤销日志或回滚日志) 记录了事务发生之前的数据状态。如果修改数据时出现异常, 可以用 undo log 来实现回滚操作 (保持原子性) 。

## Binlog

binlog 以事件的形式记录了所有的 DDL 和 DML 语句, 可以用来做主从复制和数据恢复

# MySQL索引

作用:加速检索,类似书的目录

MySQL 的存储结构分为 5 级: 表空间、段、簇、页、行

索引类型:普通索引,唯一索引,全文索引(针对字段大量文本内容取缔like语法)

## mysql索引存储模型推演

**二分查找-->二叉查找树-->平衡二叉树 (AVL Tree) (左旋、右旋) -->多路平衡查找树 (B Tree) -->B+树 (加强版多路平衡查找树)**

**二分查找折半查找**, 每一次, 我们都把候选数据缩小了一半。

**二叉查找树**:左子树所有的节点都小于父节点, 右子树所有的节点都大于父节点。投影到平面以后, 就是一个有序的线性表

**AVL平衡二叉树**:左右子树深度差绝对值不能超过 1,解决二叉树最坏的斜链  
用树的结构来存储索引的时候, **访问一个节点就要跟磁盘之间发生一次**  
IO,InnoDB 操作磁盘的最小的单位是一页16k,浪费空间,每个节点数据过少,io增多,影响性能,解决办法**多路平衡查找树 (B Tree)**

## 多路平衡查找树 (B Tree) :产生三字叶节点,存储更多,深度变小

**B+树**:加强版多路平衡树,叶子节点存取数据带指针,形成链表;根节点不存数据,存更多关键字;数据存储更多,树深度大大降低减少io

InnoDB 中 B+ 树深度一般为 1-3 层, 它就能满足千万级的数据存储。

- 1)它是 B Tree 的变种, B Tree 能解决的问题, 它都能解决。B Tree 解决的两大问题是什么? (每个节点存储更多关键字; 路数更多)
- 2)扫库、扫表能力更强 (如果我们要对表进行全表扫描, 只需要遍历叶子节点就可以了, 不需要遍历整棵 B+Tree 拿到所有的数据)
- 3)B+Tree 的磁盘读写能力相对于 B Tree 来说更强 (根节点和枝节点不保存数据区, 所以一个节点可以保存更多的关键字, 一次磁盘加载的关键字更多)
- 4)排序能力更强 (叶子节点上有下一个数据区的指针, 数据形成了链表)
- 5)效率更加稳定 (叶子节点拿到数据, 所以 IO 次数是稳定的)

**不用红黑树原因:**1、只有两路; 2、不够平衡;红黑树一般只放在内存里面用。  
例如 Java 的 TreeMap

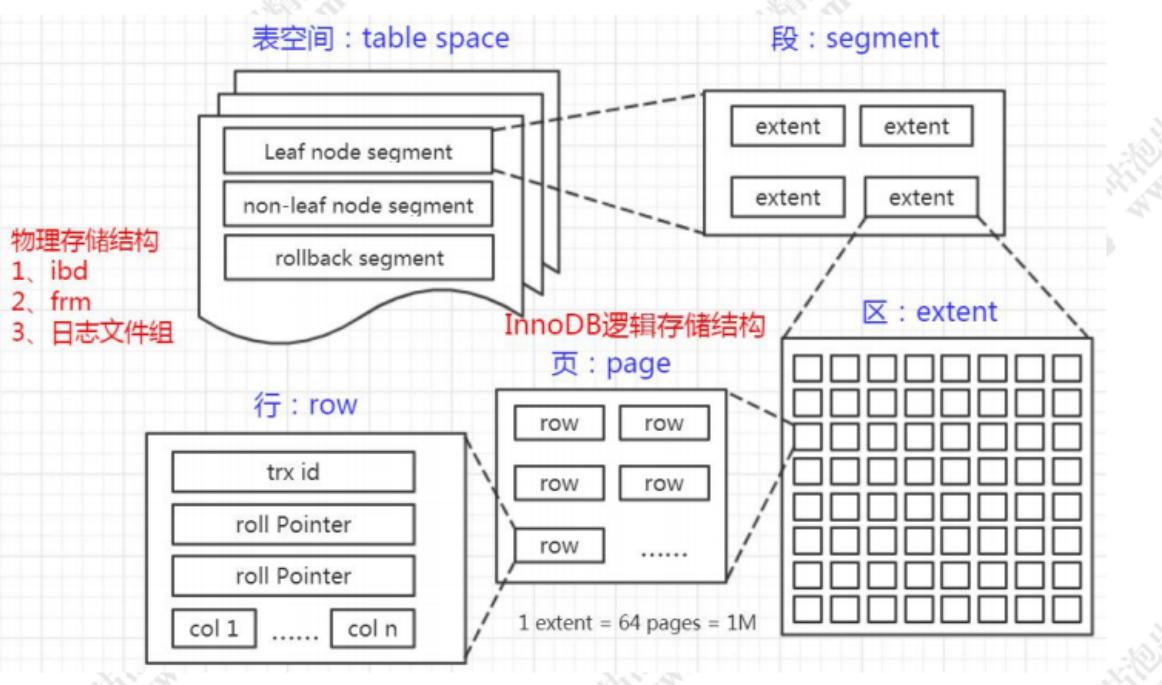
索引存储内容:字段键值,物理地址,AVL(平衡二叉树)动态快速排序

```
1 ALTER TABLE table_name ADD UNIQUE(索引类型) indexName(索引  
民) ON (column(length))(字段名))  
2 //组合索引  
3 ALTER TABLE `table` ADD INDEX name_city_age  
(name,city,age);
```

B+Tree存储索引

## InnoDB 逻辑存储结构

MySQL 的存储结构分为 5 级: 表空间、段、簇、页、行。



**表空间**: 系统表空间、独占表空间、通用表空间、临时表空间、Undo 表空间

**段**: 数据段、索引段、回滚段,一个 ibd 文件 (独立表空间文件) 里面会由很多个段组成。

**簇 Extent** : 最小扩展单位, 1m, 64个连续的页, 磁盘存储的物理最小单元

**页**: 页面在物理上和逻辑上都是连续的

每张 InnoDB 的表有两个文件 (.frm 和.ibd) ,

MyISAM 的表有三个文件 (.frm、.MYD(数据)、.MYI(索引))

.frm 是 MySQL 里面表结构定义的文件, 不管你建表的时候选用任何一个存储引擎都会生成

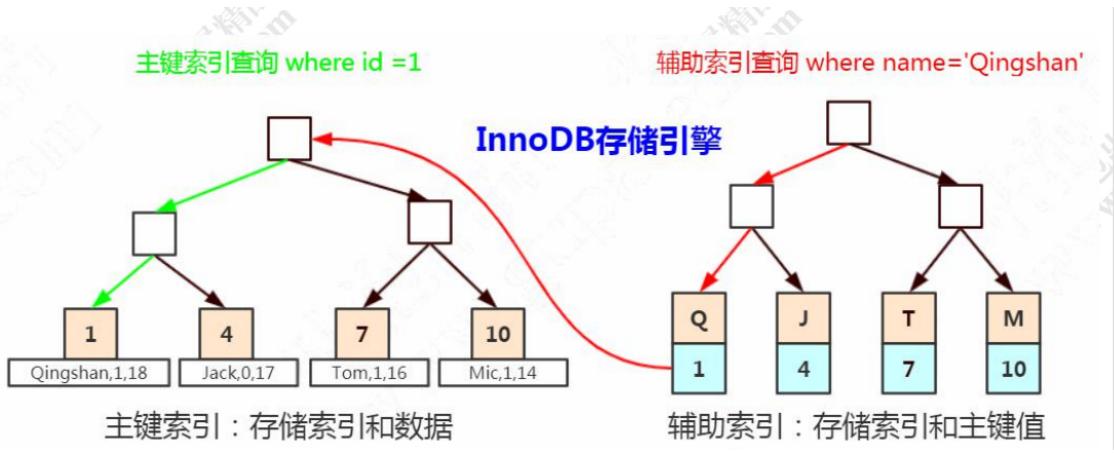
InnoDB InnoDB 以**主键为索引**来组织数据的存储的, 索引和数据都在.ibd 文件里面

**聚集索引**: 索引键值的逻辑顺序跟表数据行的物理存储顺序是一致的

InnoDB 中, 选择主键作为聚集索引; 没有主键择第一个不包含有 NULL 值的唯一索引作为主键索引; 没有唯一内置 6 字节长的ROWID 作为隐藏的聚集索引, 它会随着行记录的写入而主键递增。

**非聚集索引**: 索引和数据分开的, 没有 MyISAM 使用, 叶子节点没有完整的数据, 查询结果要回标, 性能低

先非聚集索引可以查到记录对应的主键值, 再使用主键的值通过聚集索引查找到需要的数据



## 索引使用原则

**列的离散 (sàn) 度:**同一列不同值越多越好,方便优化走索引,减少表范围扫描

**联合索引最左匹配:**多字段联合查询,优先使用左边的索引

## 什么时候用不到索引?

索引列上使用函数 (replace\SUBSTR\CONCAT\sum count avg) 、表达式、计算 (+ - \* /)

explain SELECT \* FROM t2 where id+1 = 4;

字符串不加引号, 出现隐式转换

like 条件中前面带%

NOT LIKE 不能

## 索引建立原则

在用于 where 判断 order 排序和 join 的 (on) 字段上创建索引

索引的个数不要过多

3、区分度低的字段, 例如性别, 不要建索引。

——离散度太低, 导致扫描行数过多。

4、频繁更新的值, 不要作为主键或者索引。

——页分裂

5、组合索引把散列性高 (区分度高) 的值放在前面。

## --最左原则,优化器选择

6、创建复合索引，而不是修改单列索引。

7、过长的字段，怎么建立索引？

建立前缀索引,来减少索引的长度

添加个 crc 字段，存储身份证号的时候 存入身份证号的 crc 信息(业务层计算)。

8、为什么不建议用无序的值（例如身份证、UUID）作为索引？

无序字段作为索引会导致page的分裂与合并

UUID等数据占用字节数太多，而自增的int类型只需要占用4个字节，所有索引最终都会在叶子节点存储主键的id，会导致占用更多的磁盘空间以及降低io性能

## Mysqli事物

事务的四大特性：ACID,原子性不可分割,一致性操作必须能一起错误就回滚,隔离性,事物之间透明,并发处理不干扰,持久性,无论怎么操作改变后都是完全写入数据库,利用页缓存 redo.log double双写

事物并发三大问题：

脏读:两次查询中间出现其他事物的修改但未提交

不可重复读:两次查询中间出现其他事物的修改删除相同数据,已提交

幻读:两次查询中间出现其他事物的增加操作提交,多了数据

| 隔离级别<br>并发度 | 事务隔离级别                    | 脏读  | 不可重复读 | 幻读           |
|-------------|---------------------------|-----|-------|--------------|
|             | 未提交读 ( Read Uncommitted ) | 可能  | 可能    | 可能           |
|             | 已提交读 ( Read Committed )   | 不可能 | 可能    | 可能           |
|             | 可重复读 ( Repeatable Read )  | 不可能 | 不可能   | 对 InnoDB 不可能 |
|             | 串行化 ( Serializable )      | 不可能 | 不可能   | 不可能          |

InnoDB 默认使用 RR 作为事务隔离级别的原因，既保证了数据的一致性，又支持较高的并发度

解决读一致性的办法

Lock Based Concurrency Control (LBCC) :锁定要操作的数据，不允许其他的事务修改(性能降低).

多版本的并发控制 Multi Version Concurrency Control (MVCC) 在修改数据的时候给它建立一个备份或者叫快照，后面再来读取

InnoDB 为每行记录都实现了两个隐藏字段：DB\_TRX\_ID 创建版本号,DB\_ROLL\_PTR 删除版本号.查询结果都一致

InnoDB 中， MVCC 和锁是协同使用的，**这两种方案并不是互斥的**

## 锁方案

**共享锁**(读锁):select ..... lock in share mode; 的方式手工加上一把**读锁**

**排它锁**(写锁):一个事务获取了一行数据的排它锁,也称写锁，其他的事务就不能再获取这一行数据的共享锁和排它锁

SELECT \* FROM student where id=1 **FOR UPDATE;**

数据库自动加锁

意向共享锁(标识):有其他的事务给其中的某些数据行加 上了共享锁

意向排他锁(标识):有其他的事务给其中的某些数据行加 上了排他锁

标识用作表硕

行锁的原理:只有通过索引条件来检索数据，才能使用行 级锁，否则，直接使用表级锁。通过唯一索引给数据行加锁，主键索引也会被锁住

查询没有使用索引，会进行全表扫描，然后把每一个隐藏的聚集索引都锁住了

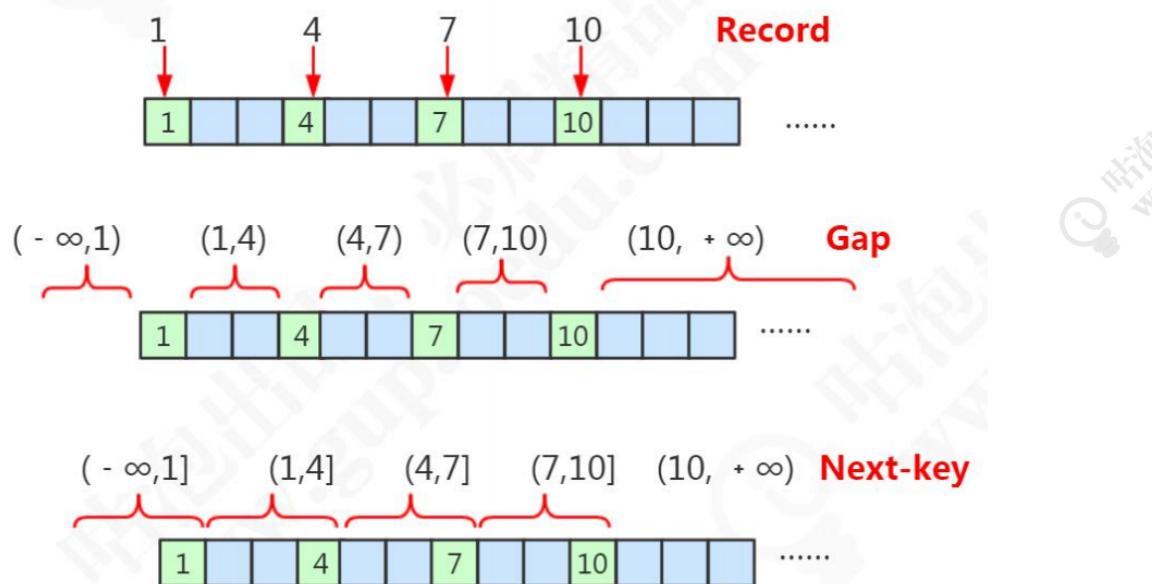
另外三种锁:算法实现的.**记录锁**(等值查询),间隙锁(范围查询),临界锁(MySQL 里面默认的行锁算法，相当于记录锁加上间隙锁)

唯一性索引，等值查询匹配到一条记录的时候，退化成记录锁。

没有匹配到任何记录的时候，退化成间隙锁。

临键锁，锁住最后一个 key 的下一个左开右闭的区间,解决幻读

主键索引不是整形，是字符 用 ASCII 码来排序



## 隔离级别的实现:

**Read Uncommitted**: RU 隔离级别：不加锁

**Read Committed** : RC 隔离级别下，普通的 select 都是快照读，使用 MVCC 实现。加锁的 select 都使用记录锁，因为没有间隙锁;RC 会出现幻读的问题,用到了间隙锁

## 查看锁信息

概要:show status like 'innodb\_row\_lock\_%'

具体:select \* from information\_schema.INNODB\_TRX; --

当前运行的所有事务，还有具体的语句

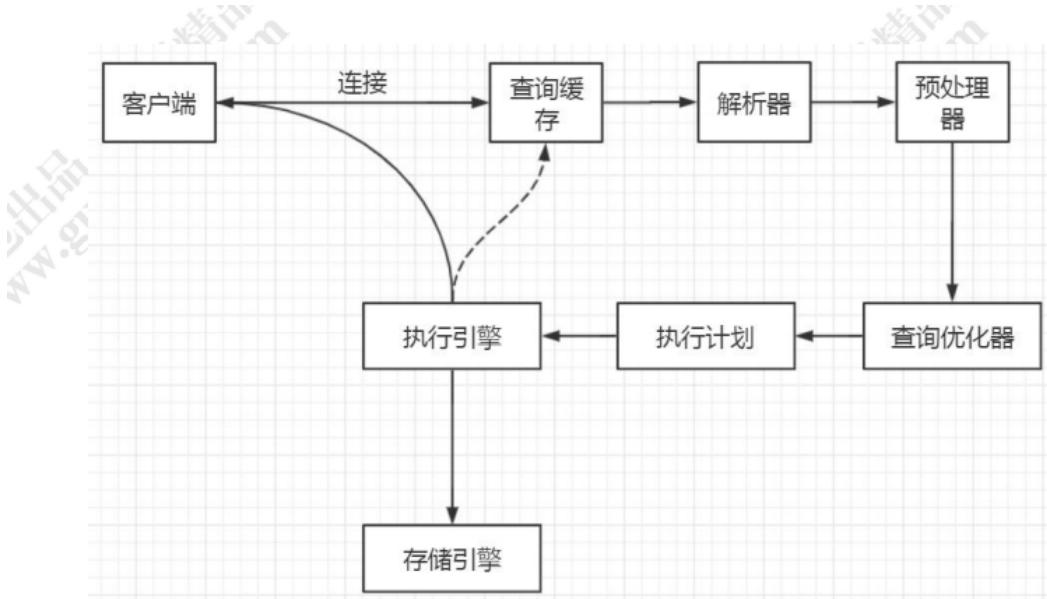
## 避免环形等待出现死锁

- 1、在程序中，操作多张表时，尽量以相同的顺序来访问（避免形成等待环路）；
- 2、批量操作单张表数据的时候，先对数据进行排序（避免形成等待环路）；
- 3、申请足够级别的锁，如果要操作数据，就申请排它锁；
- 4、尽量使用索引访问数据，避免没有 where 条件的操作，避免锁表；
- 5、如果可以，大事务化成小事务；
- 6、使用等值查询而不是范围查询查询数据，命中记录，避免间隙锁对并发的影响

查看锁状态:

```
show status like 'innodb_row_lock_%';
```

## 性能优化



思路:优化每个过程

### 连接——配置优化

池化技术;阿里的 Druid、Spring Boot 2.x 版本默认的连接池 Hikari、老牌的 DBCP 和 C3P0,客户端

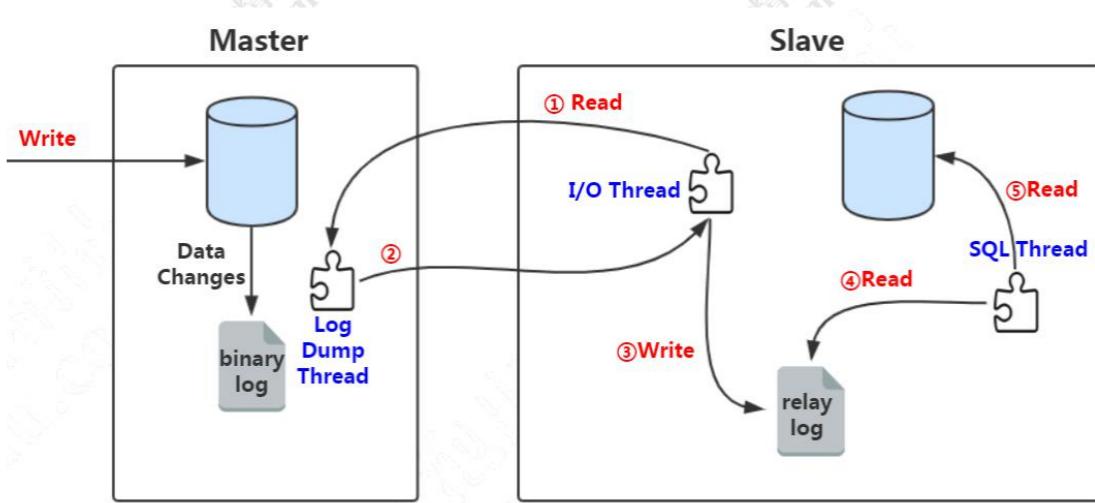
### 缓存——架构优化

缓存



集群方案 主从复制,

MySQL 5.7 引入了 Group Replication 功能, 可以在一组 MySQL 服务器之间实现自动主机选举, 形成一主多从结构。经过高级配置后, 可以实现多主多从结构。



master 获取 binlog，并且解析 binlog 写入中继日志，这个线程叫做 I/O 线程。

Master 节点上有一个 log dump 线程，是用来发送 binlog 给 slave 的。  
从库的 SQL 线程，是用来读取 relay log，把数据写入到数据库的。

**半同步复制:**至少一个slave写完binlog返回给客户端,利用插件/usr/lib64/mysql/plugin/

```

1 -- 主库执行
2 INSTALL PLUGIN rpl_semi_sync_master SONAME
  'semisync_master.so';
3 set global rpl_semi_sync_master_enabled=1;
4 show variables like '%semi_sync%';
5 -- 从库执行
6 INSTALL PLUGIN rpl_semi_sync_slave SONAME
  'semisync_slave.so';
7 set global rpl_semi_sync_slave_enabled=1;
8 show global variables like '%semi%';

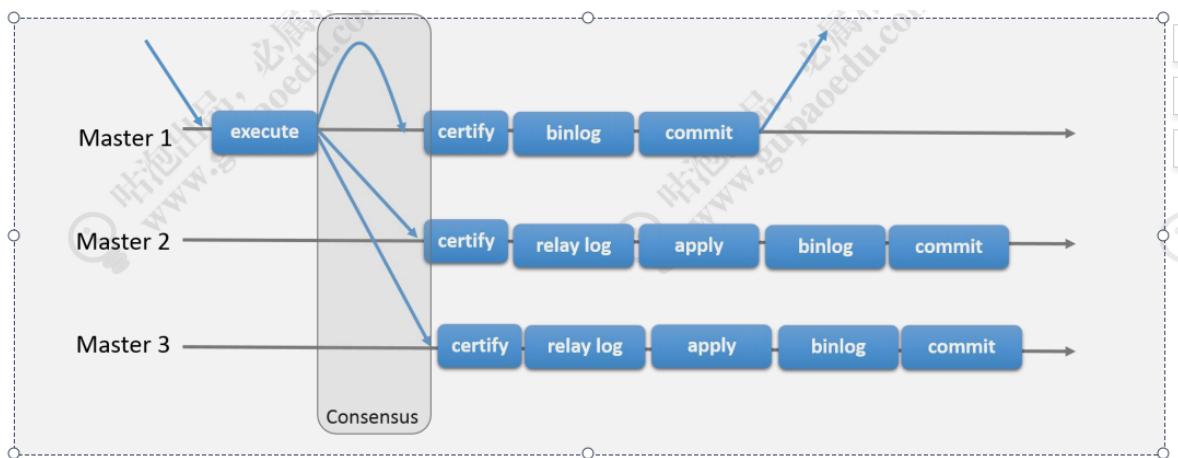
```

异步主从复制:基于全局id GTID

分库分表

高可用方案

传统的 HAProxy + keepalived 的方案，基于主从复制。



**解析器**, 词法和语法分析, 主要保证语句的正确性, 语句不出错就沒问题。

## 优化器——SQL 语句分析与优化

慢查询日志 slow query log ,show variables like 'slow\_query%'(默认关闭), 日志分析mysqlslow 的工具, 在 MySQL 的 bin 目录下。

SHOW PROFILE 是谷歌高级架构师 Jeremy Cole 贡献给 MySQL 社区的, 可以查看 SQL 语句执行的时候使用的资源, 比如 CPU、IO 的消耗情况

## EXPLAIN 执行计划

**SQL语句优化** SQL 语句比较复杂, 有多个关联和子查询的时候, 就要分析 SQL 语句有没有改写的方法

**索引优化:**参考索引建立规则和使用规则

## 存储引擎

查询插入操作多的业务表, 用 MyISAM。临时数据用 Memeroy。常规的并发大更新多的表用 InnoDB。

## 建表之类

非空字段尽量定义成 NOT NULL, 提供默认值, 或者使用特殊值、空串代替 null

## 不要用外键、触发器、视图

不要用数据库存储图片 (比如 base64 编码) 或者大文件; 数据库只需要存储 URI (相对路径)

**表拆分** 将不常用的字段拆分出去, 避免列数过多和数据量过大



其他优化:业务层面的优化也不能忽视

应用层面同样有很多其他的方案来优化，达到尽量减轻数据库的压力的目的，比如限流，或者引入 MQ 削峰，等等等等

## mycat

<http://dl.mycat.io/>

解决多案分库分表带来的问题 带来的问题

**跨库关联查询 分布式事务 排序、翻页、函数计算问题 全局主键避重问题**

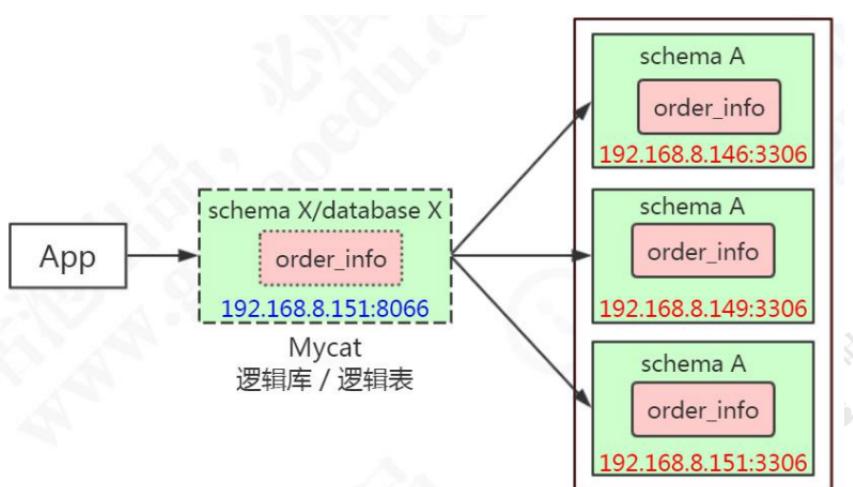
**多数据源/读写数据源的解决方案**

DAO——Mapper (ORM) ——JDBC——代理——数据库服务

## Mycat 的关键特性

- 1、可以当做一个 MySQL 数据库来使用
- 2、支持 MySQL 之外的数据库，通过 JDBC 实现
- 3、解决，**多表 join、分布式事务、全局序列号、翻页排序**
- 4、支持 ZK 配置，带监控 mycat-web
- 5、2.0 正在开发中

术语：  
主机 / 实例  
物理数据库  
物理表  
分片 (切分)  
分片节点  
分片键  
分片算法  
逻辑表  
逻辑数据库



## mycat 配置

主要的配置文件 server.xml、schema.xml、rule.xml 和具体的分片配置文件 system 标签 例如字符集、线程数、心跳、分布式事务开关等等

user 标签：配置登录用户和权限。schema.xml 包括逻辑库、表、分片规则、分片节点和数据源，可以定义多个 schema。这里面有三个主要的标签

(table、dataNode、dataHost) :

```
<schema name="catmall" checkSQLschema="false" sqlMaxLimit="100">
    <!-- 范围分片 -->
    <table name="customer" primaryKey="id" dataNode="dn1,dn2,dn3" rule="rang-long-cust" />
    <!-- 取模分片 -->
    <table name="order_info" dataNode="dn1,dn2,dn3" rule="mod-long-order" >
        <!-- ER 表 -->
        <childTable name="order_detail" primaryKey="id" joinKey="order_id" parentKey="order_id"/>
    </table>
    <!-- 全局表 -->
    <table name="student" primaryKey="sid" type="global" dataNode="dn1,dn2,dn3" />
</schema>
```

zk配置 <https://www.cnblogs.com/leeSmall/p/9551038.html>

explain 可以用来看路由结果

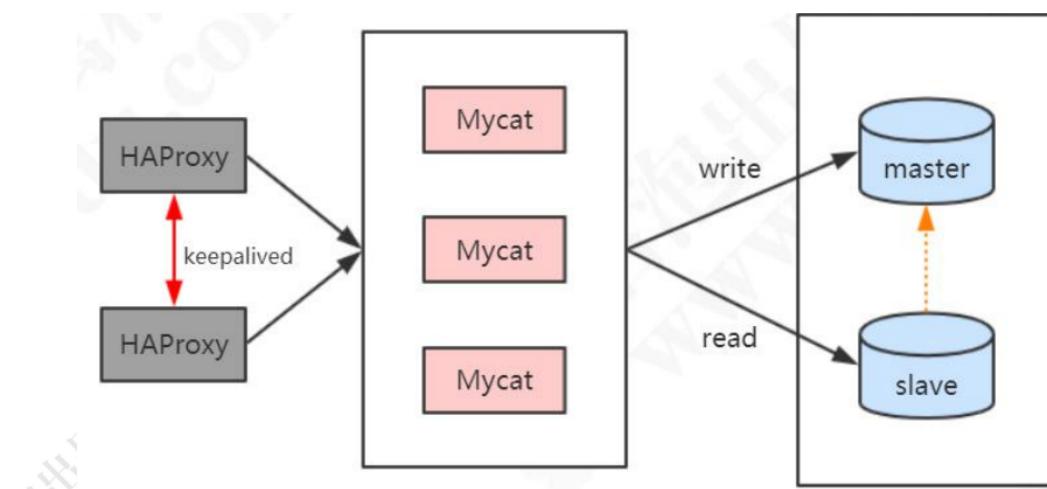
Mycat 全局序列实现方式主要有 4 种：本地文件方式、数据库方式、本地时间戳算法、ZK。也可以自定义业务序列

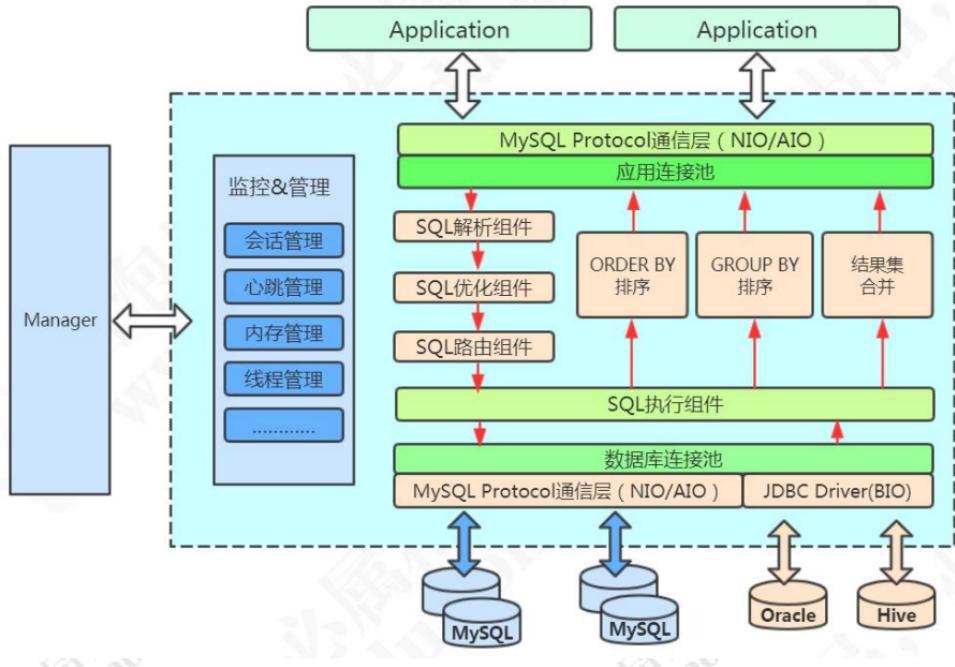
命令行监控 mycatweb 监控 log4j 的 level 配置要改成 debug

## Mycat 高可用

目前 Mycat 没有实现对多 Mycat 集群的支持，可以暂时使用 HAProxy 来做负载

思路：HAProxy 对 Mycat 进行负载。Keepalived 实现 VIP





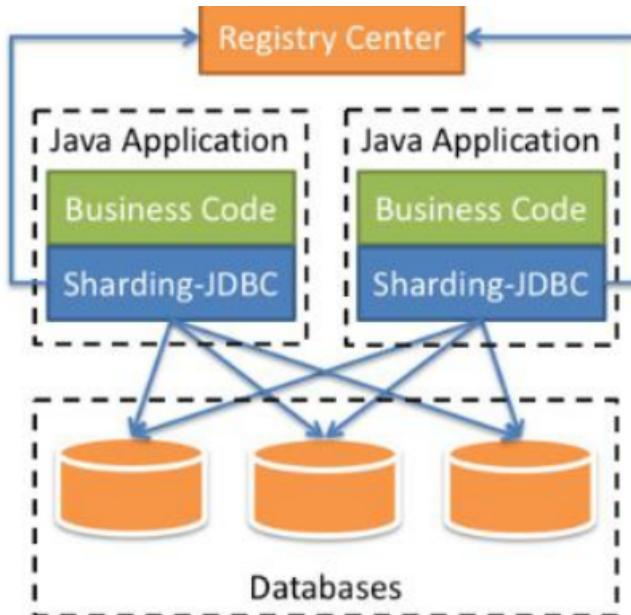
# Sharding-JDBC

## 概念

定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据 库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架,无需独立部署.

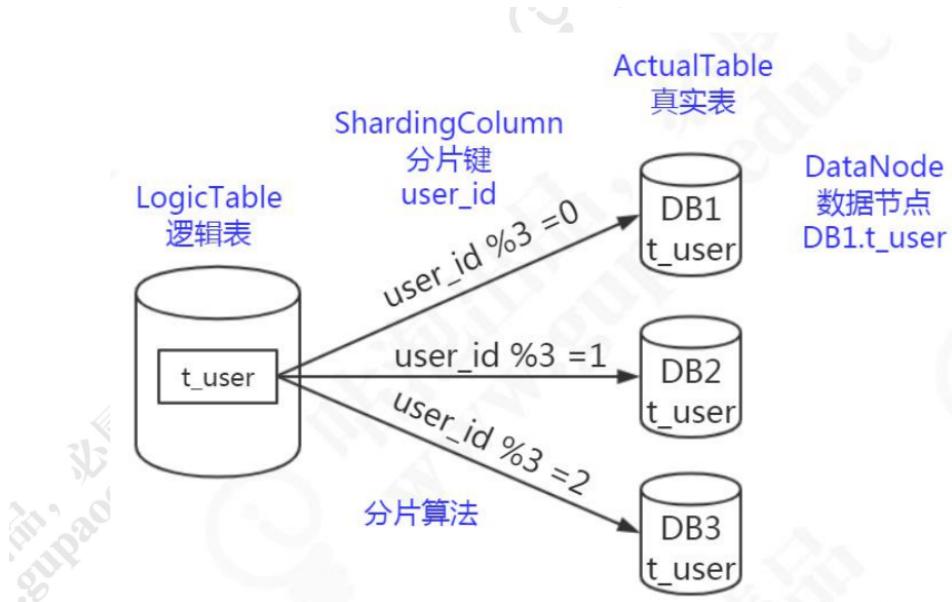
## 架构

工作在客户端,SQL 解析， 路由， 执行， 结果处理

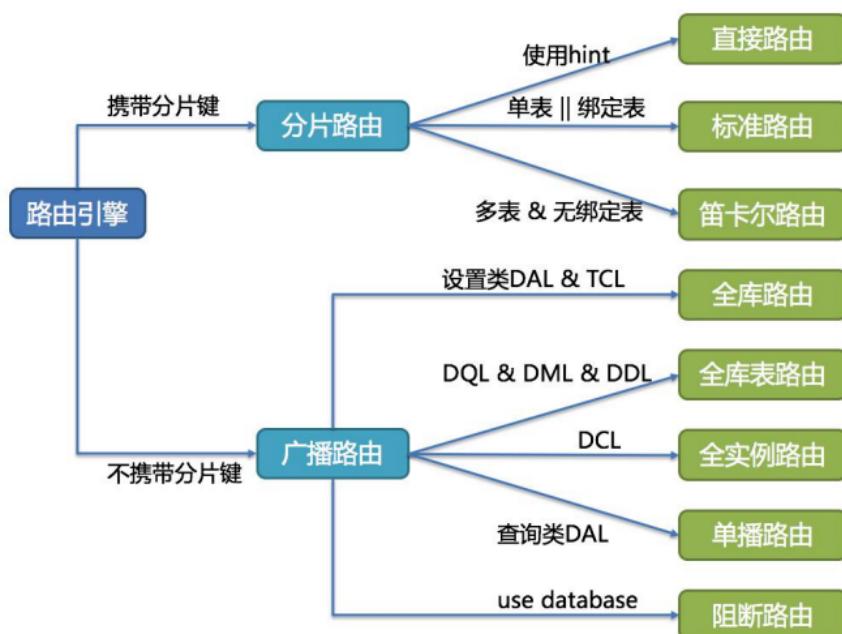
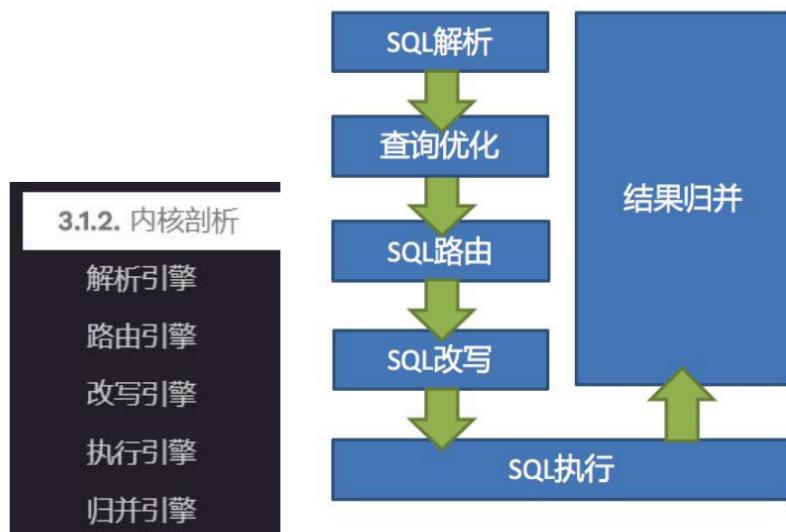


## 功能

解决:跨库关联查询、分布式事务、排序翻页计算、全局主键



sql流程



# Quartz

特点：用来执行定时任务

精确到毫秒级别的调度

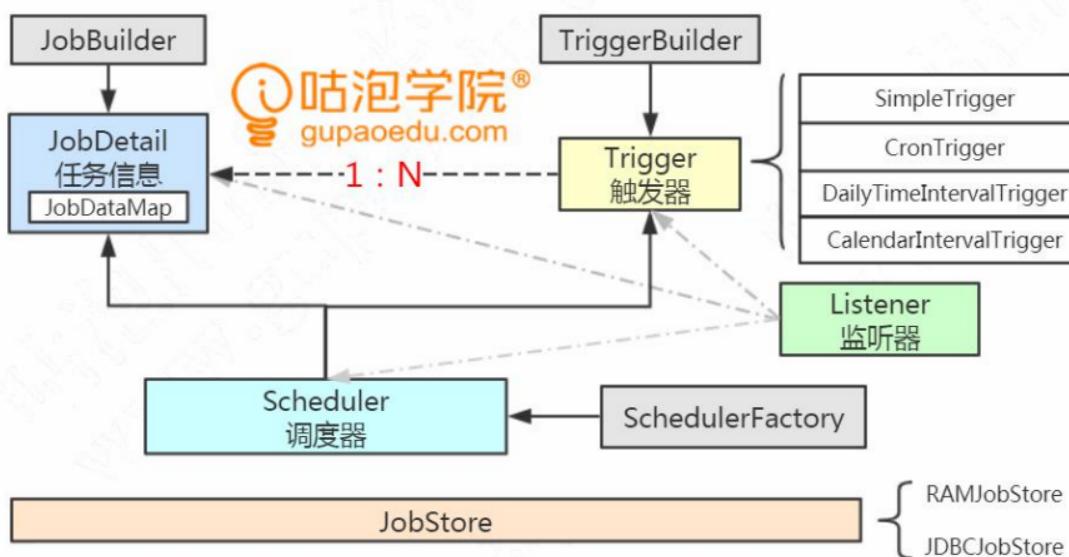
可以独立运行，也可以集成到容器中

支持事务 (JobStoreCMT )

支持集群

支持持久化

咕泡出品，必属精品 www.gupaoedu.com



\*生产推荐数据库储存job定义及运行状态

springboot启动时需要初始化查询数据库,就要加入扫描规则,实现  
CommandLineRunner

```
*/  
@Component  
public class InitStartSchedule implements CommandLineRunner {  
    private final Logger logger = LoggerFactory.getLogger(this.getClass());  
  
    @Autowired
```

```

1 <dependency>
2   <groupId>org.quartz-scheduler</groupId>
3   <artifactId>quartz</artifactId>
4   <version>2.3.0</version>
5 </dependency>
6 #配置
7 org.quartz.jobStore.class:org.quartz.impl.jdbcjobstore.Jo
bStoreTX
8 org.quartz.jobStore.driverDelegateClass:org.quartz.impl.j
dbcjobstore.StdJDBCDelegate
9 # 使用 quartz.properties, 不使用默认配置
10 org.quartz.jobStore.useProperties:true
11 #数据库中 quartz 表的表名前缀
12 org.quartz.jobStore.tablePrefix:QRTZ_
org.quartz.jobStore.dataSource:myDS
12 #配置数据源
org.quartz.dataSource.myDS.driver:com.mysql.jdbc.Driver
org.quartz.dataSource.myDS.URL:jdbc:mysql://localhost:330
6/gupao?useUnicode=true&characterEncoding=utf8
org.quartz.dataSource.myDS.user:root
org.quartz.dataSource.myDS.password:123456
org.quartz.dataSource.myDS.validationQuery=select 0 from
dua

```

## Cron 表达式

位置	时间域		特殊值
1	秒	0-59	, * /
2	分钟	0-59	, - * /
3	小时	0-23	, - * /
4	日期	1-31	, - * ? / L W C
5	月份	1-12	, - * /
6	星期	1-7	, - * ? / L W C
7	年份 (可选)	1-31	, - * /

星号(\*)：可用在所有字段中，表示对应时间域的每一个时刻，例如，在分钟字段时，表示“每分钟”；

问号(?)：该字符只在日期和星期字段中使用，它通常指定为“无意义的值”，相当于点位符；

减号(-)：表达一个范围，如在小时字段中使用“10-12”，则表示从 10 到 12 点，即 10,11,12；

逗号(,)：表达一个列表值，如在星期字段中使用“MON,WED,FRI”，则表示星期一，星期三和星期五；

斜杠(/)：x/y 表达一个等步长序列，x 为起始值，y 为增量步长值。如在分钟字段中使用 0/15，则表示为 0,15,30 和 45 秒，而 5/15 在分钟字段中表示 5,20,35,50，你也可以使用\*/y，它等同于 0/y；

L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L 在日期字段中，表示这个月份的最后一天，如一月的 31 号，非闰年二月的 28 号；如果 L 用在星期中，则表示星期六，等同于 7。但是，如果 L 出现在星期字段里，而且在前面有一个数值 X，则表示“这个月的最后 X 天”，例如，6L 表示该月的最后星期五；

W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如 15W 表示离该月 15 号最近的工作日，如果该月 15 号是星期六，则匹配 14 号星期五；如果 15 日是星期日，则匹配 16 号星期一；如果 15 号是星期二，那结果就是 15 号星期二。但必须注意关联的匹配日期不能够跨月，如你指定 1W，如果 1 号是星期六，结果匹配的是 3 号星期一，而非上个月最后的那天。W 字符串只能指定单一日期，而不能指定日期范围；

LW 组合：在日期字段可以组合使用 LW，它的意思是当月的最后一个工作日；

井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如 6#3 表示当月的第三个星期五(6 表示星期五，#3 表示当前的第三个)，而 4#5 表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；

C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期。例如 5C 在日期字段中就相当于日历 5 日以后的第一天。1C 在星期字段中相当于星期日后的第一天。

Quartz 集成到 Spring 中，用到 SchedulerFactoryBean，其实现了 InitializingBean 方法，在唯一的方法 afterPropertiesSet() 在 Bean 的属性初始化后调用。调度器用 AdaptableJobFactory 对 Job 对象进行实例化。所以，如果我们可以把这个 JobFactory 指定为我们自定义的工厂的话，就可以在 Job 实例化完成之后，把 Job 纳入到 Spring 容器中管理

**Quartz 集群部署** 是随机的负载均衡算法，不能指定节点执行

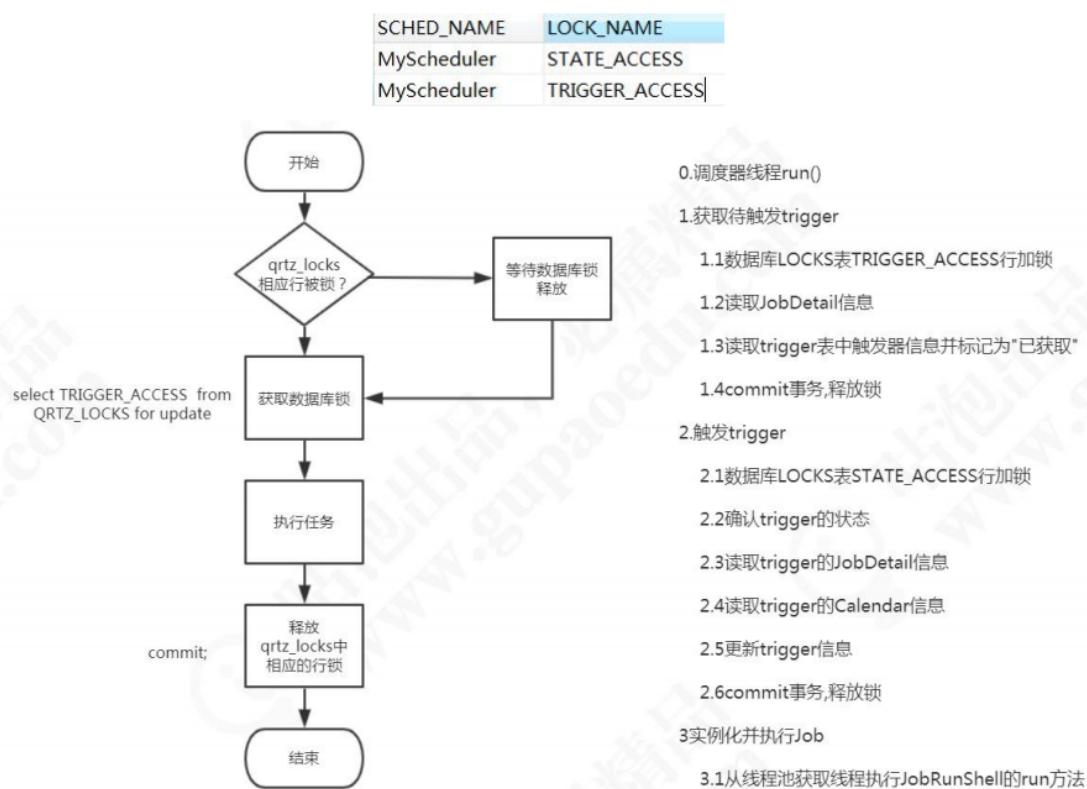
## 线程模型总结

SimpleThreadPool：包工头，管理所有 WorkerThread

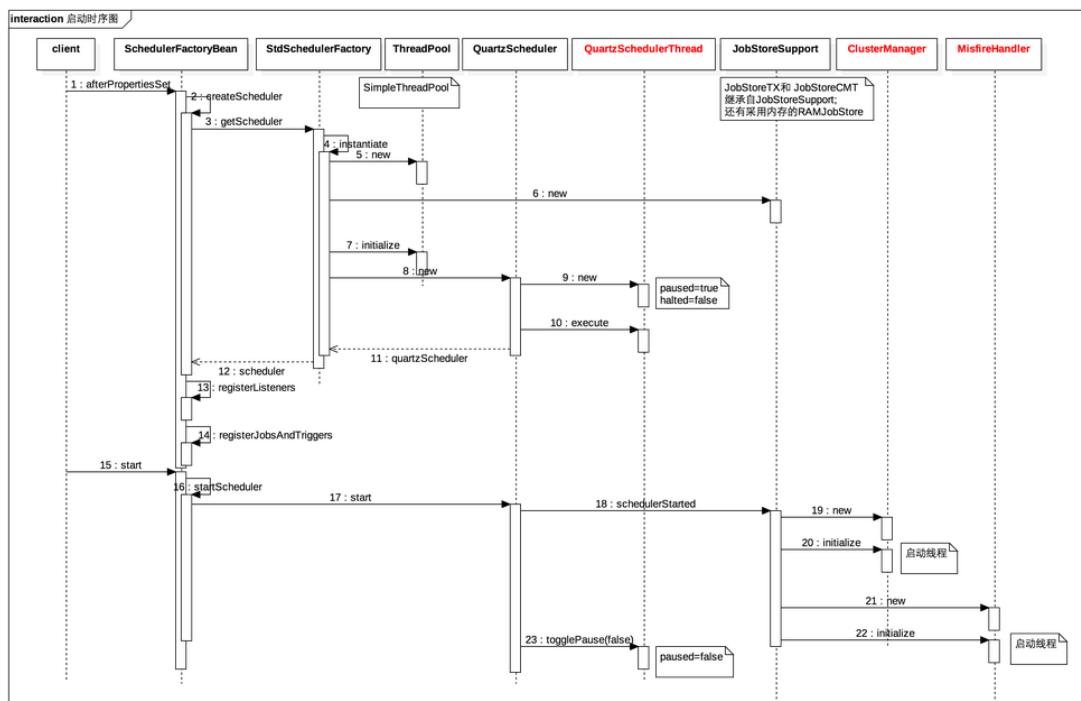
WorkerThread：工人，把 Job 包装成 JobRunShell，执行

QuartzSchedulerThread：项目经理，获取即将触发的 Trigger，从包工头出拿到 worker，执行 Trigger 绑定的任务

## 自动枷锁



整个启动流程如下图：



## Elastic-Job

封装升级了quartz,提供了运维平台

Quartz 的不足：

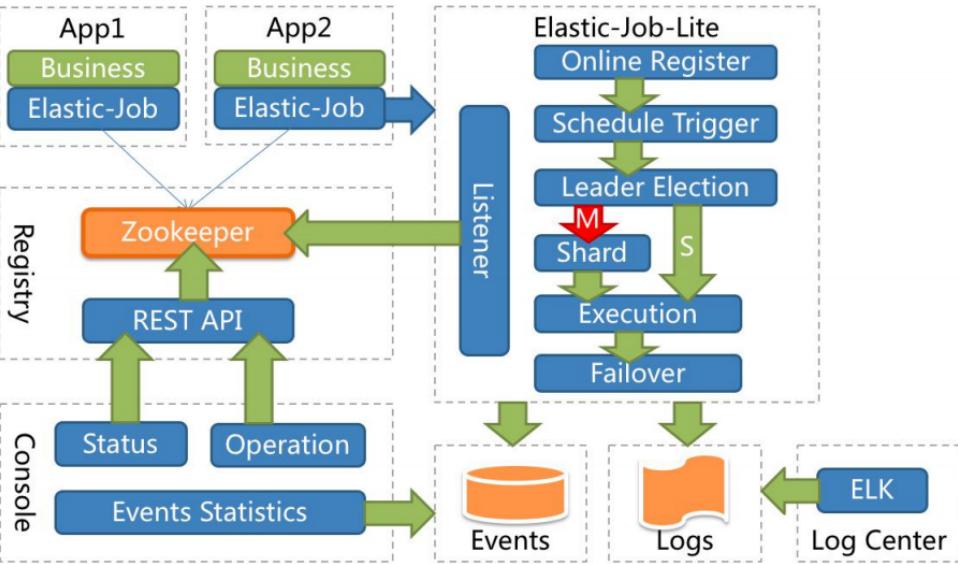
- 1、作业只能通过 DB 抢占随机负载，无法协调
- 2、任务不能分片——单个任务数据太多了跑不完，消耗线程，负载不均
- 3、作业日志可视化监控、统计

集群的每个节点都是对等的，

节点之间通过注册中心进行分布式协调。E-Job 存在主节点的概念，但是主节点没有调度的功能，而是用于处理一些集中式任务，如分片，清理运行时信息等

利用zookeeper做分布式注册中心,无中心存在,leader配置一些重要的数据用到了选举,

分片,是一个任务分成几个线程并发处理提升速度



## 基本使用

```
<dependency>
    <groupId>com.dangdang</groupId>
    <artifactId>elastic-job-lite-core</artifactId>
    <version>2.1.5</version>
</dependency>
```

SimpleJob: 简单实现，未经任何封装的类型。需实现 SimpleJob 接口

```
public class MyElasticJob implements SimpleJob {
    public void execute(ShardingContext context) {
        System.out.println(String.format("Item: %s | Time: %s | Thread: %s ",
            context.getShardingItem(), new SimpleDateFormat("HH:mm:ss").format(new Date()),
            Thread.currentThread().getId()));
    }
}
```

DataFlowJob: Dataflow 类型用于处理数据流，必须实现 fetchData()和 processData()的方法，一个用来获取数据，一个用来处理获取到的数据。  
ejob-standalone MyDataFlowJob.java

```
public class MyDataFlowJob implements DataflowJob<String> {
    @Override
    public List<String> fetchData(ShardingContext shardingContext) {
        // 获取到了数据
        return Arrays.asList("qingshan", "jack", "seven");
    }

    @Override
    public void processData(ShardingContext shardingContext, List<String> data) {
        data.forEach(x-> System.out.println("开始处理数据: "+x));
    }
}
```

Script: Script 类型作业意为脚本类型作业，支持 shell, python, perl 等所有类型脚本

```

public class SimpleJobTest {

    public static void main(String[] args) {
        // ZK 注册中心
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(new
ZookeeperConfiguration("localhost:2181", "elastic-job-demo"));
        regCenter.init();

        // 定义作业核心配置
        JobCoreConfiguration simpleCoreConfig = JobCoreConfiguration.newBuilder("MyElasticJob", "0/2 * * * * ?",
1).build();
        // 定义 SIMPLE 类型配置
        SimpleJobConfiguration simpleJobConfig = new SimpleJobConfiguration(simpleCoreConfig,
MyElasticJob.class.getCanonicalName());

        // 定义 Lite 作业根配置
        LiteJobConfiguration simpleJobRootConfig = LiteJobConfiguration.newBuilder(simpleJobConfig).build();

        // 构建 Job

        new JobScheduler(regCenter, simpleJobRootConfig).init();
    }
}

```

## ZK 注册中心数据结构

一个任务一个二级节点。这里面有些节点是临时节点，只有任务运行的时候才能看到。注意：修改了任务重新运行任务不生效，是因为 ZK 的信息不会更新，除非把 overwrite 修改成 true

### config 节点

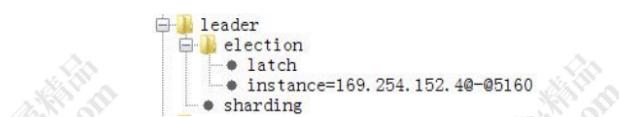
JSON 格式存储。

存储任务的配置信息，包含执行类，cron 表达式，分片算法类，分片数量，分片参数等等 config 节点的数据是通过 ConfigService 持久化到 zookeeper 中去的。默认状态下，如果你修改了 Job 的配置比如 cron 表达式、分片数量等是不会更新到 zookeeper 上去的，除非你在 Lite 级别的配置把参数 overwrite 修改成 true

### instances 节点

同一个 Job 下的 elastic-job 的部署实例。一台机器上可以启动多个 Job 实例，也就是 Jar 包。instances 的命名是 IP+@-@+PID。只有在运行的时候能看到。

### leader 节点



election：主节点选举

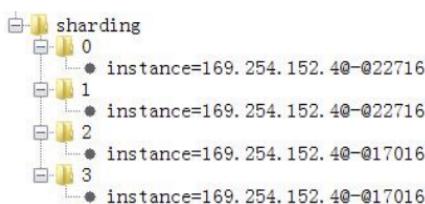
sharding: 分片

failover: 失效转移

election 下面的 instance 节点显示了当前主节点的实例 ID: jobInstanceId。 election 下面的 latch 节点也是一个永久节点用于选举时候的实现分布式锁。

sharding 节点下面有一个临时节点，necessary，是否需要重新分片的标记。如果分片总数变化，或任务实例节点上下线或启用/禁用，以及主节点选举，都会触发设置重分片标记，主节点会进行分片计算

### sharding 节点



子节点名	是否临时节点	描述
instance	否	执行该分片项的作业运行实例主键
running	是	分片项正在运行的状态 仅配置 monitorExecution 时有效
failover	是	如果该分片项被失效转移分配给其他作业服务器，则此节点值记录执行此分片的作业服务器 IP
misfire	否	是否开启错过任务重新执行
disabled	否	是否禁用此分片项

## 运维平台

git 下载源码 <https://github.com/elasticjob/elastic-job-lite>

### 添加 ZK 注册中心

第一步，添加注册中心，输入 ZK 地址和命名空间，并连接。运维平台和 elastic-job-lite 并无直接关系，是通过读取作业注册中心数据展现作业状态，或更新注册中心数据修改全局配置。控制台只能控制作业本身是否运行，但不能控制作业进程的启动，因为控制台和作业本身服务器是完全分离的，控制台并不能控制作业服务器。可以对作业进行操作



### 分片策略

策略类	描述	具体规则
AverageAllocationJobShardingStrategy	基于平均分配算法的分片策略，也是默认的分片策略。	<p>如果分片不能整除，则不能整除的多余分片将依次追加到序号小的服务器。如：</p> <ul style="list-style-type: none"> <li>如果有 3 台服务器，分成 9 片，则每台服务器分到的分片是：1=[0,1,2], 2=[3,4,5], 3=[6,7,8]</li> <li>如果有 3 台服务器，分成 8 片，则每台服务器分到的分片是：1=[0,1,6], 2=[2,3,7], 3=[4,5]</li> <li>如果有 3 台服务器，分成 10 片，则每台服务器分到的分片是：1=[0,1,2,9], 2=[3,4,5], 3=[6,7,8]</li> </ul>
OdevitySortByNameJobShardingStrategy	根据作业名的哈希值奇偶数决定 IP 升降序算法的分片策略。	<p>根据作业名的哈希值奇偶数决定 IP 升降序算法的分片策略。</p> <ul style="list-style-type: none"> <li>作业名的哈希值为奇数则 IP 升序。</li> <li>作业名的哈希值为偶数则 IP 降序。</li> </ul> <p>用于不同的作业平均分配负载至不同的服务器。</p>
RotateServerByNameJobShardingStrategy	根据作业名的哈希值对服务器列表进行轮转的分片策略。	
自定义分片策略		实现 JobShardingStrategy 接口并实现 sharding 方法，接口方法参数为作业服务器 IP 列表和分片策略选项，分片策略选项包括作业名称，分片总数以及分片序列号和个性化参数对照表，可以根据需求定制化自己的分片策略。

## 分片方案

- 1、对业务主键进行取模，获取余数等于分片项的数据
- 2、在表中增加一个字段，根据分片数生成一个 mod 值。
- 3、如果从业务层面，可以用 ShardingParamter 进行分片

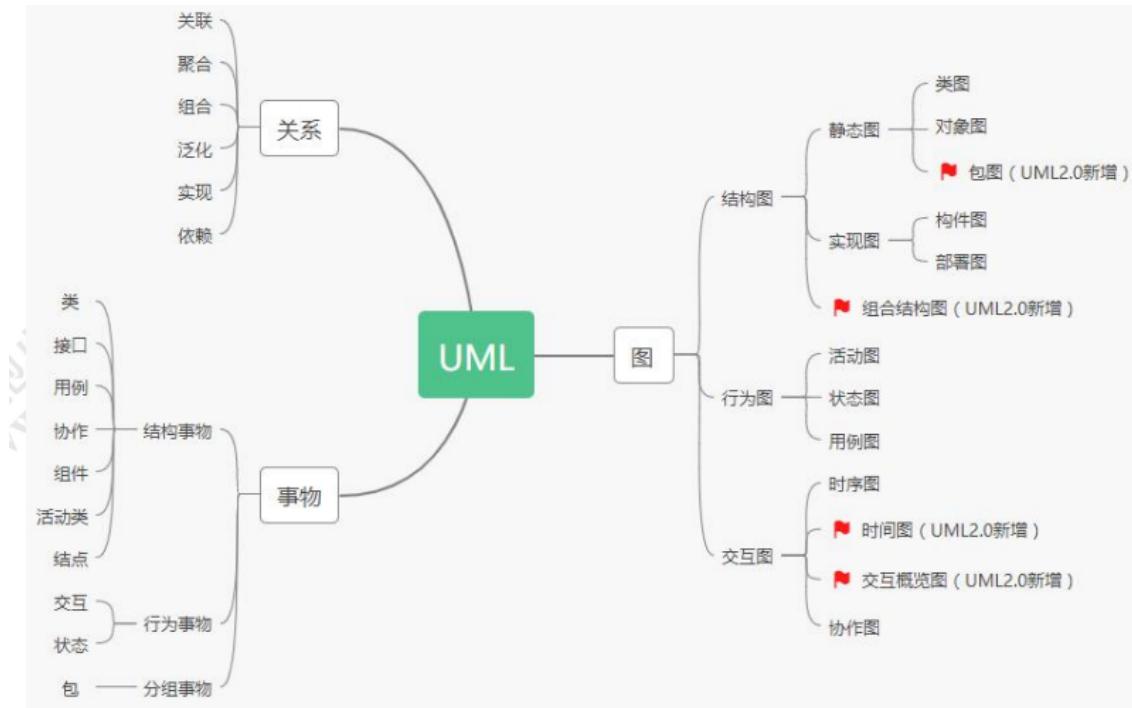
工程 : elasticjob-spring-boot-starter

需求 (一个 starter 应该有什么样子) :

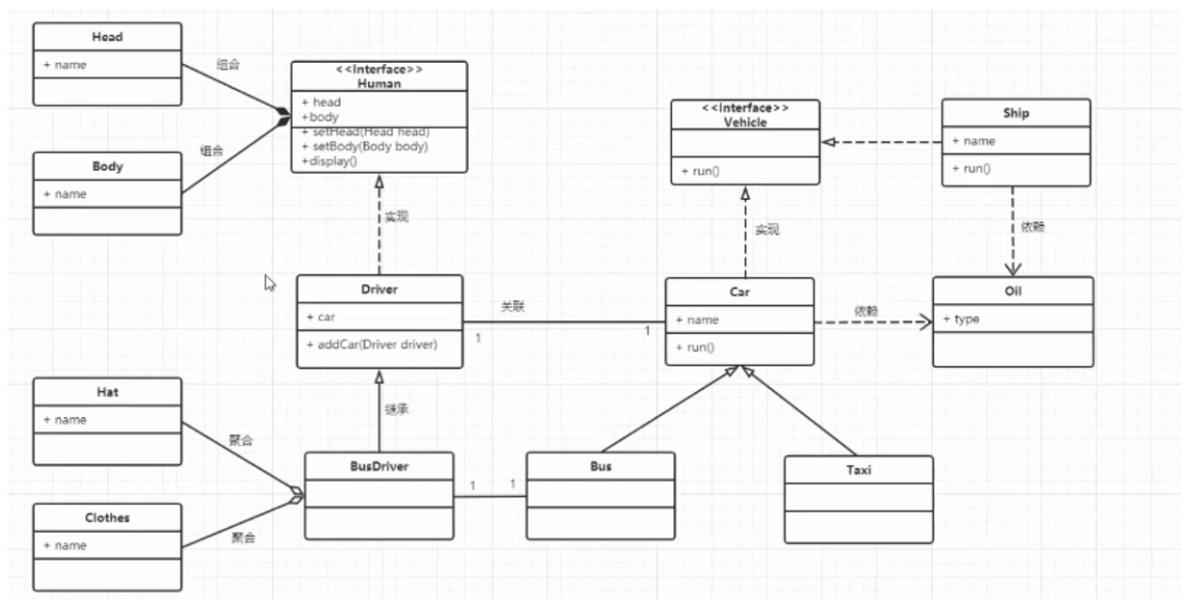
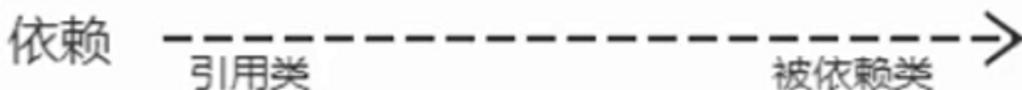
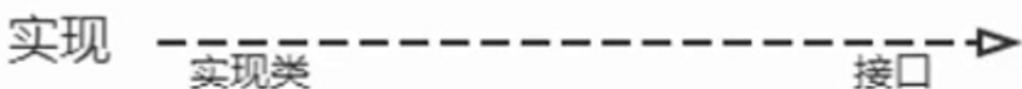
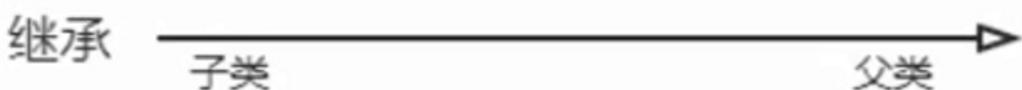
需求	实现	作用
可以在启动类上使用@Enable 功能开启 E-Job 任务调度	注解@EnableElasticJob	在自动配置类上用@ConditionalOnBean 决定是否自动配置
可以在 properties 或 yml 中识别配置内容	配置类 RegCenterProperties.java	支持在 properties 文件中使用 elasticjob.regCenter 前缀，配置注册中心参数
在类上加上注解，直接创建任务	注解 @JobScheduled	配置任务参数，包括定分片项、分片参数等等
不用创建 ZK 注册中心	自动配置类 RegCentreAutoConfiguration.java	注入从 RegCenterProperties.java 读取到的参数，自动创 ZookeeperConfiguration
不用创建三级 (Core、Type、Lite) 配置	自动配置类 JobAutoConfiguration.java	读取注解的参数，创建 JobCoreConfiguration 、 JobTypeConfiguration 、 LiteJobConfiguration 在注册中心创建之后再创建
Spring Boot 启动时自动配置	创建 Resource/META-INF/spring.factories	指定两个自动配置类

# UML

应用在架构设计阶段,以图表的形式具象化



## 关联关系



箭头方向：从子类指向父类。

记忆技巧：1、定义子类是需要通过 extends 关键字指定父类；

2、子类一定是知道父类定义的，但父类并不知道子类的定义；

3、只有知道对方信息时才能指向对方；

4、所以箭头的方向是从子类指向父类。

继承实现：用线条连接两个类。

记忆技巧：1、空心三角箭头表示继承或实现

关联依赖：用线条连接两个类。

记忆技巧：1、虚线表示依赖关系：临时用一下，若即若离，虚无缥缈，若有若无；表示一种使用关系，一个类需要借助另一类来实现功能；**一般是一个类将另一个类作为参数使用，或作为返回值。**

2、实线表示关联关系：关系稳定，实打实的关系，铁哥们；表示一个类对象和另一个类对象有关联；**通常是一个类中有另一个类对象作为属性。**

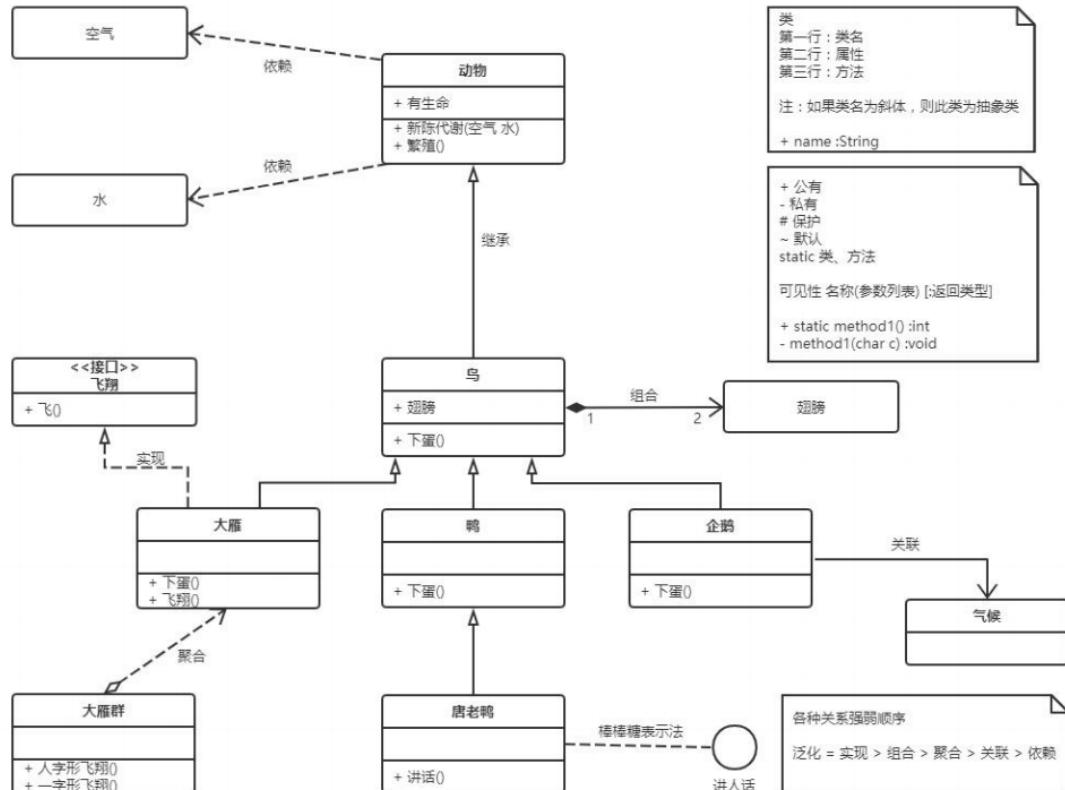
组合聚合：用菱形表示。

记忆技巧：1、菱形就是像是一个盛东西的器皿（比如盘子）；

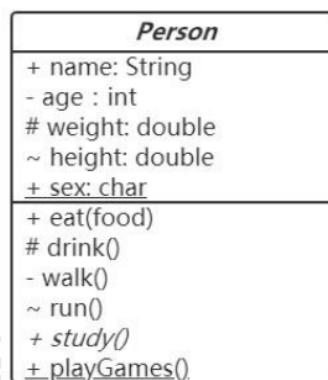
2、聚合：空心菱形，代表空器皿里可以放很多相同的东西，聚集在一起（箭头

3、组合：实心菱形，代表器皿里已经有实体结构的存在，生死与共；整体与局部的关系，和聚合关系对比，关系更加强烈；**两者具有相同的生命周期，contains-a 的关系；强关系，积极的词：强-满。**

注意：UML 类关系图中，没有实心箭头。

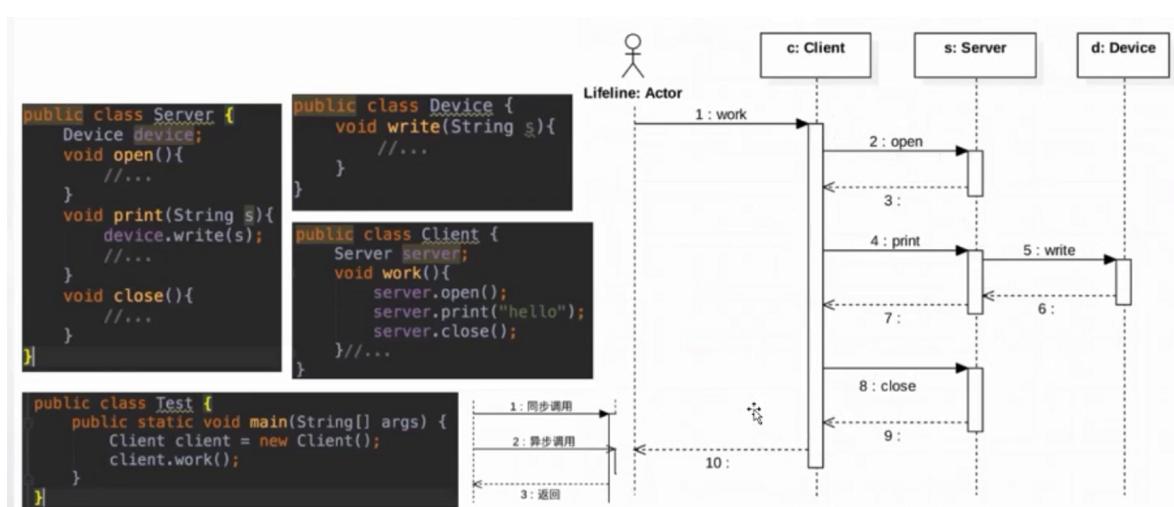


+ 表示 public  
 - 表示 private  
 # 表示 protected  
 ~ 表示 default，可省略不写。  
 字段和方法返回值的数据类型非必须。  
 抽象类或抽象方法用斜体表示。  
 静态类或静态方法加下划线。  
 如果是接口在类名上方加<<Interface>>。



## 时序图

纵向虚线表示时间,水平表示各种对象



## 时序图画法及实践

时序图的绘制步骤可简单总结如下：

- 1、划清边界，识别交互的语境；
- 2、将所要绘制的交互场景中的角色以及对象梳理出来；
- 3、从触发整个交互的某个消息开始，在生命线之间从上到下依次画出所有消息，并注明每个消息的特性（如参数等）。

资料地址问w3c

[https://www.w3cschool.cn/zookeeper/zookeeper\\_installation.html](https://www.w3cschool.cn/zookeeper/zookeeper_installation.html)

常用中间件事务实现机制

<https://blog.csdn.net/huangshulang1234/article/details/78747018/>

[https://blog.csdn.net/qq\\_33589510/article/details/106078434](https://blog.csdn.net/qq_33589510/article/details/106078434)

线程池技术，

<https://blog.csdn.net/jek123456/article/details/90601351>

## 微服务Spring Cloud与Kubernetes比较

---

<https://www.jdon.com/48605>

oauth2

<https://blog.csdn.net/kefengwang/article/details/81213025>