# Nginx

## 1、正向代理：非常明确要访问的服务器地址；服务器只清楚请求来自哪个代理服务器，而不清楚来自哪个具体的客户端；正向代理模式屏蔽或者隐藏了真实客户端信息。

例子：VPN，我告诉代理服务器我需要访问的服务器，然后代理服务器访问我需要访问的服务器，且将需要访问的服务器的返回数据返回给我。

正向代理的用途**：**

（1）访问原来无法访问的资源，如google

（2） 可以做缓存，加速访问资源

（3）对客户端访问授权，上网进行认证

（4）代理可以记录用户访问记录（上网行为管理），对外隐藏用户信息

## 2、反向代理**：**请求的来源也就是客户端是明确的，但是请求具体由哪台服务器处理的并不明确了，nginx扮演的就是一个反向代理角色，隐藏了服务器的信息！

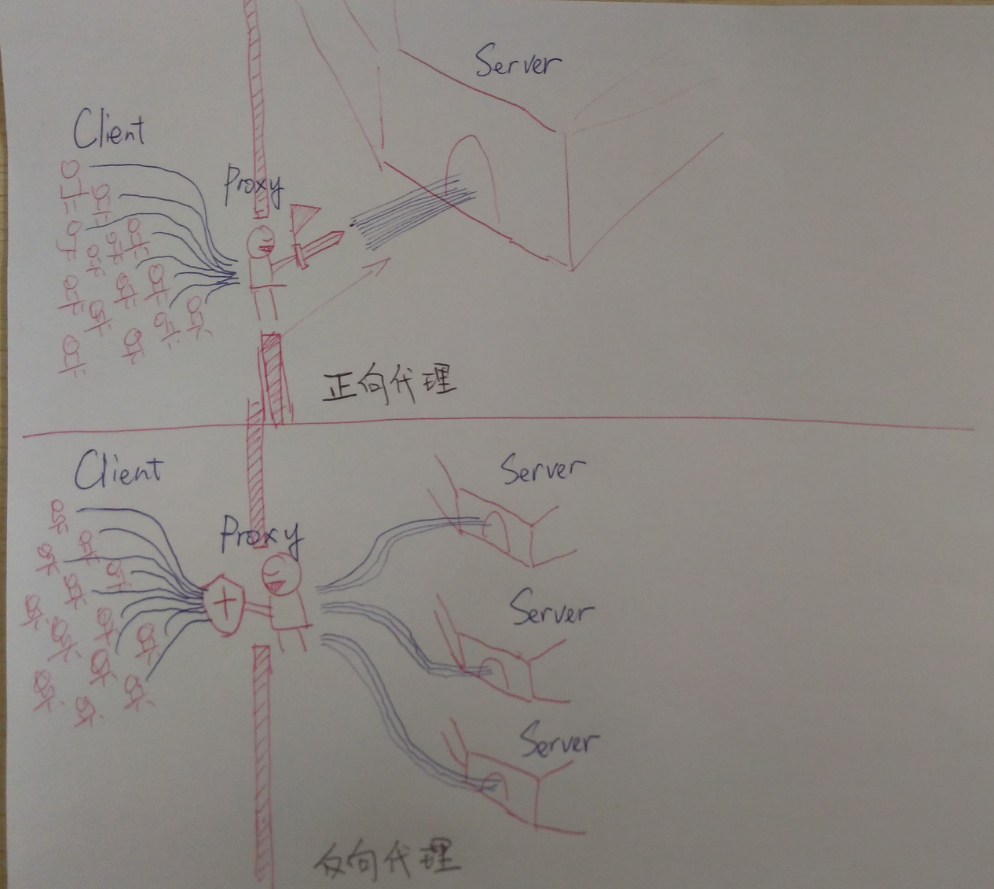
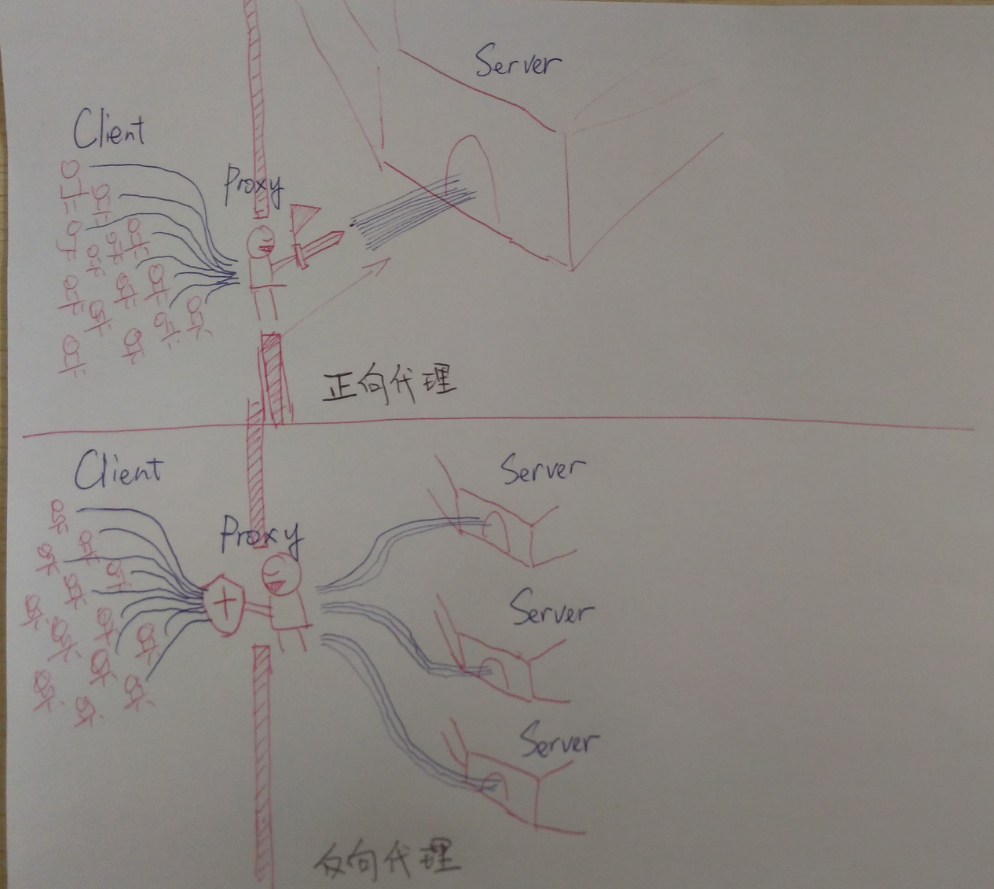
例子：访问淘宝，我的请求发送到代理服务器，代理服务器再将请求转发到真实的服务器。

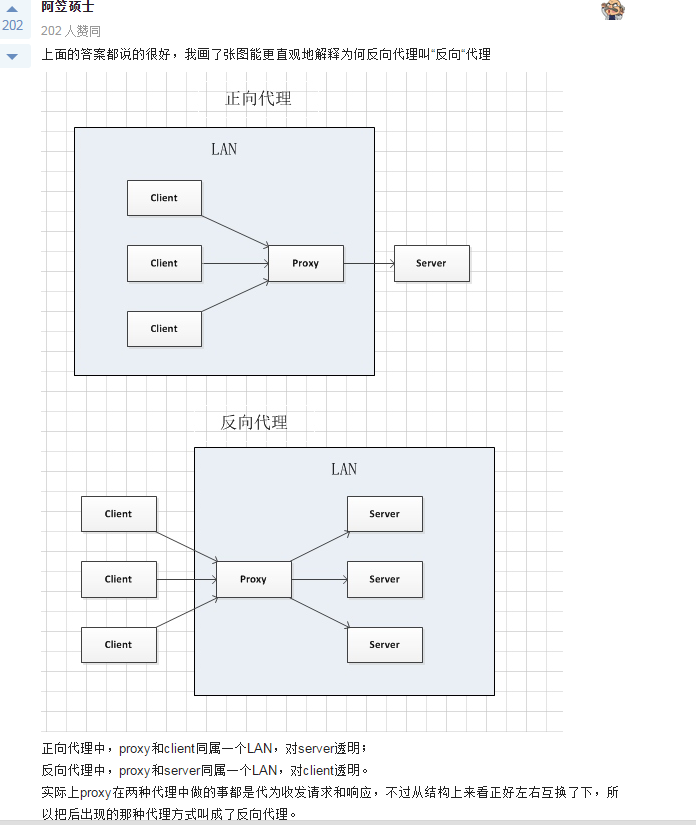
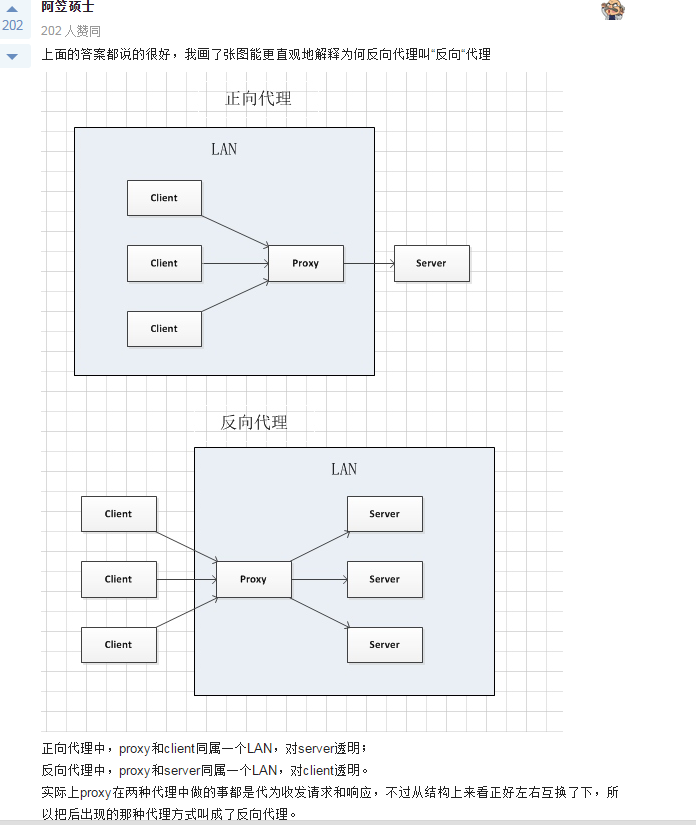
反向代理的作用：

（1）保证内网的安全，可以使用反向代理提供WAF功能，阻止web攻击；

（2）大型网站，通常将反向代理作为公网访问地址，Web服务器是内网；

（3）负载均衡，通过反向代理服务器来优化网站的负载；





## 3、负载均衡

（1）**轮询法**：将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

缺点：各个机器性能不一致，导致性能较差的服务器崩溃。

（2）**加权随机法**：与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

（2）**IP\_Hash**：根据获取客户端的IP地址，通过哈希函数计算得到一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

缺点:1.由于ip地址的原因.经过hash值计算.可能会导致负载不均；2.如果IP地址绑定的服务器宕机,那么很多的用户访问受限；3.安全性不好。

（3）**随机法**：通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。

（4）**加权轮询法**：不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

（5）**最小连接数法**：由于后端服务器的配置不尽相同，对于请求的处理有快有慢，最小连接数法根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负责合理地分流到每一台服务器。

ip\_hash：每个请求按照发起客户端的ip的hash结果进行匹配，这样的算法下一个固定ip地址的客户端总会访问到同一个后端服务器，这也在一定程度上解决了集群部署环境下session共享的问题。

（6）**fair**：智能调整调度算法，动态的根据后端服务器的请求处理到响应的时间进行均衡分配，响应时间短处理效率高的服务器分配到请求的概率高，响应时间长处理效率低的服务器分配到的请求少；结合了前两者的优点的一种调度算法。但是需要注意的是nginx默认不支持fair算法，如果要使用这种调度算法，请安装upstream\_fair模块

（7）**url\_hash**：按照访问的url的hash结果分配请求，每个请求的url会指向后端固定的某个服务器，可以在nginx作为静态服务器的情况下提高缓存效率。同样要注意nginx默认不支持这种调度算法，要使用的话需要安装nginx的hash软件包

（8）**backup**——其它所有的非backup机器down或者忙的时候，请求backup机器。所以这台机器压力会最轻。

（9）**down**——表示单前的server暂时不参与负载

## 4、备用机机制

说明:为了让nginx出现问题后,能够继续提供服务.采用备用机制.正常情况下备用不起作用.当主机遇忙时或者主机宕机时.这时备用机生效.

#负载均衡配置策略  轮询

   upstream jt {

      #ip\_hash;

      server  localhost:8091 weight=6;

      server  localhost:8092 weight=3 down;

      server  localhost:8093 weight=1 backup;

   }

## 5、Nginx中健康检测

1. 设定超时时间

说明:当nginx实现反向代理时,如果后台服务器没有及时的响应,那么必须等到超时间后.访?????????????.????????????????3????.????????????????????.?

proxy\_connect\_timeout       3;

proxy\_read\_timeout          3;

proxy\_send\_timeout          3;

1. 设定超时时间的优点

当访问后台的服务器.如果到了指定的超时时间,那么则自动访问下一台服务器.

缺点:

      因为配置权重或者轮询.定期都会连接故障机.导致用户查询效率低.

1. 检测检测实现

   说明:

      当在指定的周期内,如果发现后台服务器不能正常访问.那么在该周期内不会再将请求发往该故障机.直到下个周期再次检测.

配置:

      max\_fails=1    在指定的周期内,检测失败的次数

fail\_timeout=60s;   代表周期时间  单位秒

   upstream jt {

      #ip\_hash;

      server  localhost:8091  max\_fails=1 fail\_timeout=60s;

      server  localhost:8092  max\_fails=1 fail\_timeout=60s;

      server  localhost:8093  max\_fails=1 fail\_timeout=60s;

   }

## 6、Nginx和Tomcat的区别是什么？

虽然Tomcat也可以认为是HTTP服务器，内部集成了http服务器的相关功能，但通常它仍然会和Nginx配合在一起使用。nginx常用做静态内容服务和反向代理服务器，以及页面前端高并发服务器。严格的来说，Nginx 应该叫做「HTTP Server」；而 Tomcat 则是一个「Application Server」，或者更准确的来说，是一个「Servlet/JSP」应用的容器（Ruby/Python 等其他语言开发的应用也无法直接运行在 Tomcat 上），这种特性导致了它们在使用中有明确的分工职责。tomcat更多用来做做一个应用容器，让java web app跑在里面的东西。大部分适用于静态资源文件的访问（图片，文件）

## **7、异步非阻塞**：

（1）**阻塞**和非阻塞阻塞：应用程序在获取网络数据的时候，如果程序传输数据很慢，那么程序就一直等待着，直到数据传输完成。

**非阻塞**：应用程序直接可以获取已经准备就绪的数据, 无须等待。

BIO和NIO的本质区别就是阻塞和非阻塞。3.3

（2） **同步和异步**：同步和异步一般是面向操作系统与应用程序对I/O操作的层面上区别的。

同步：应用程序会直接参与IO读写操作，并且我们的应用程序会直接阻塞到某个方法上，直到数据准备完毕。

异步：所有的IO操作交个操作系统处理，与我们应用程序没有直接关系，我们程序不需要关系IO读写，当操作系统完成了 IO读写的时候，会给我们应用程序发出通知，数据传输完毕。3.4 在Java网络通信中同步和异步 阻塞和非阻塞

（3）同步异步说的是Server服务器端的执行方式。

阻塞和非阻塞说的就是接受数据的方式。

（4）BIO为同步阻塞模式，NIO为同步非阻塞。NIO并没有实现异步，在JDK1.7以后，升级了NIO库包，指出异步非阻塞通信模式NIO 2.0（AIO）

## 8、Nginx 为什么高效，参考已有的文章。

**（1）设计原理**

　　一个高性能服务器典型特点是处理速度块且占用资源少，尤其是当上万连接同时在线时候.若要做到处理速度快，并发模型设计尤其关键。

　　服务器并发量取决于两个因素：一是服务器连接的进程数量，二是每个进程可同时处理的并发请求数量,因而服务器并发模型由两部分构成，**服务的提供方式和链接处理机制，**由于这两种别具一格的方式使得Nginx在同类型的网页服务器中表现优秀。

**（2）连接方式**

　　一般web服务器连接方式分为三种:多进程方式、多线程方式和异步方式。多进程需要内存复制等额外开销、多线程不需要开销较小

　　异步则采用非阻塞的方式与每个客户通信，服务器用一个进程进行轮询，典型应用如Nginx的work process进程。其中效率最高是异步，最稳定是多进程、占用资源少是多线程

        Nginx基于事件模型（**异步方式**）提供服务，更**适合每秒连接数和请求数同时非线性增长的情况**。即使负载增加了，内存和CPU使用事件数量始终保持可预期。Nginx使用普通的硬件就能在一个服务器上处理数万的并发连接。这就是Nginx高性能的第一个原因。



**（3）连接处理机制**

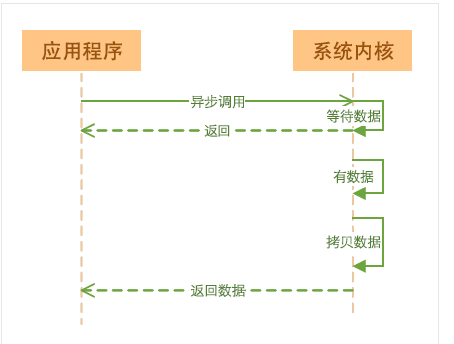
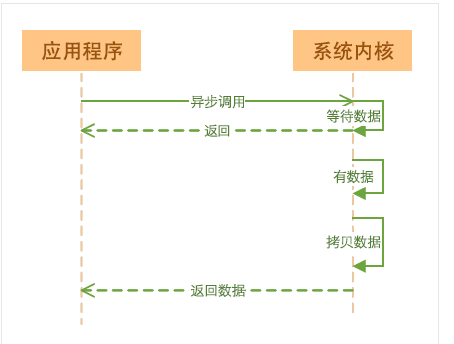
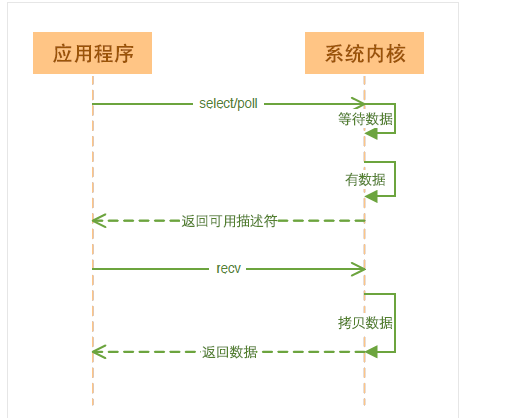
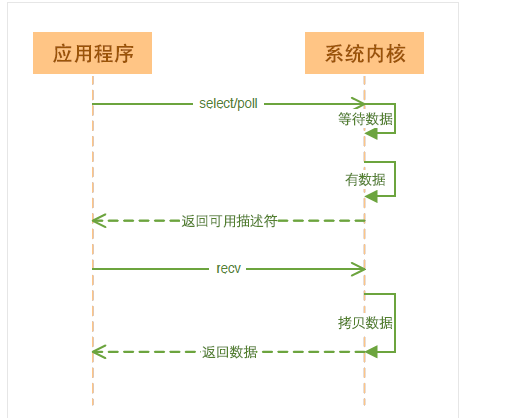
　　linux 连接模型有五种，同步io，同步费阻塞io，异步阻塞io（io复用），异步非阻塞io（信号驱动或者异步io）

　　以下是重点：

　　　① **io复用**，I/O复用模型会用到select或poll函数，这两个函数也会使进程阻塞，与阻塞I/O有所不同，他们也可以同时阻塞多个I/O操作，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。此处理机制redis中也用这也是redis高效的原因之一，redis是单进程单线程的。下图为io复用原理图

        在 io复用模型中，select和poll一般操作系统都会支持，但是每次等待都需要设置需要等待的套接字接口，并且内部实现不够高效，很那支付监听高并发的套接口集，不同的操作系统使用了不同的高级轮询技术来支持高性能的监听，一般这些方式都不是可移值的。而**Nginx针对不同的操作系统，定制了不同的I/O处理机制，且一般都会采用操作系统的高性能接口**，这是Nginx的高性能的第二个原因

　　　　②**异步IO**：当一个异步过程调用请求发出后，调用者不能立即得到结果。实际处理这个调用的部件在完成后，会通过状态、通知和回调来通知调用者的输入输出操作



**（4）Nginx所做的其他**

　　　　除了以上两大原理，内部有大量可见的细节优化，采用**多进程单线程**的工作方式，并且利用cpu和进程的亲缘性将进程和特定cpu绑定，避免了进程上下文切换的开销，从而减少了cpu占用。另外它实现了高效的**内存池**，将内存占用降到最低等。

# Redis

## 一、专业名词介绍

### 1、缓存穿透

条件：访问一个不存在的数据

说明：当访问一个不存在的数据时，因为缓存中没有这个key，导致缓存形同虚设。最终访问后台数据库。但是数据库中没有该数据所以返回null。

隐患：如果有人恶意频繁查询一个不存在的数据，可能会导致数据库负载高导致宕机。

总结：业务系统访问一个不存在的数据，称之为缓存穿透。

### 2、缓存击穿

条件：当缓存key失效/过期/未命中时，高并发访问该key

说明：如果给一个key设定了失效时间，当key失效时有一万的并发请求访问这个key，这时缓存失效，所有的请求都会访问后台数据库。称之为缓存击穿。

场景：微博热点消息访问量很大，如果该缓存失效则会直接访问后台数据库，导致数据库负载过高。

### 3、缓存雪崩

前提：高并发访问，缓存命中较低或者失效时

说明：假设缓存都设定了失效时间，在同一时间内缓存大量失效。如果这时用户高并发访问。缓存命中率过低。导致全部的用户访问都会访问后台真实的数据库。

场景：在高并发条件下。缓存动态更新时

### 4、脑裂

出现脑裂的可能原因：①主机 master 挂掉推选新的 master 时出现问题；②主机 master 网络连接异常，但是没有宕机，slave推选新的 master 导致脑裂

说明：由于推选机制同时选举出多台主机，导致程序不能正常的执行。该问题称之为脑裂。

解决办法：①机器的数量是奇数台；②配置的服务器的量一定要多。

### 5、 什么时候集群崩溃

说明：如果主节点在宕机时没有从节点则集群崩溃。

问题：3主6从 宕机几次集群崩溃？

特点：集群中如果主机宕机，那么从机可以继续提供服务，

当主机中没有从机时，则向其他主机借用多余的从机。继续提供服务。

如果主机宕机时没有从机可用，则集群崩溃。

答案：9台机器，宕机5-7次集群崩溃了。

6、Java程序连接异常

问题描述：

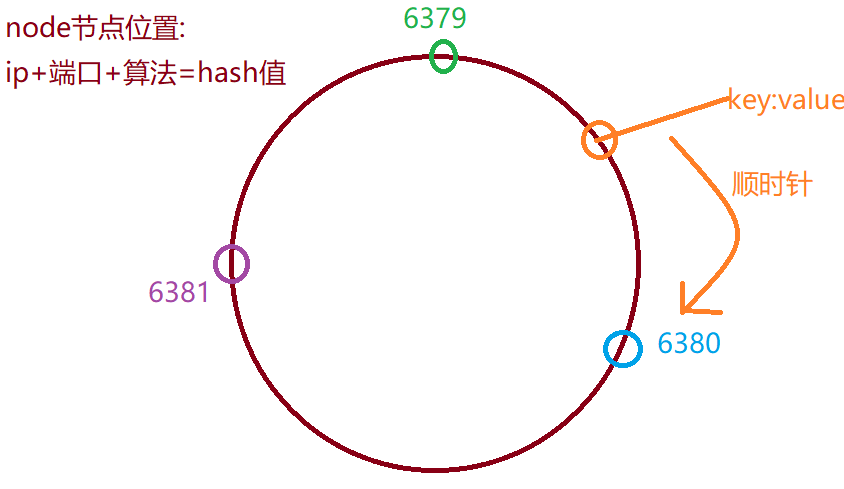
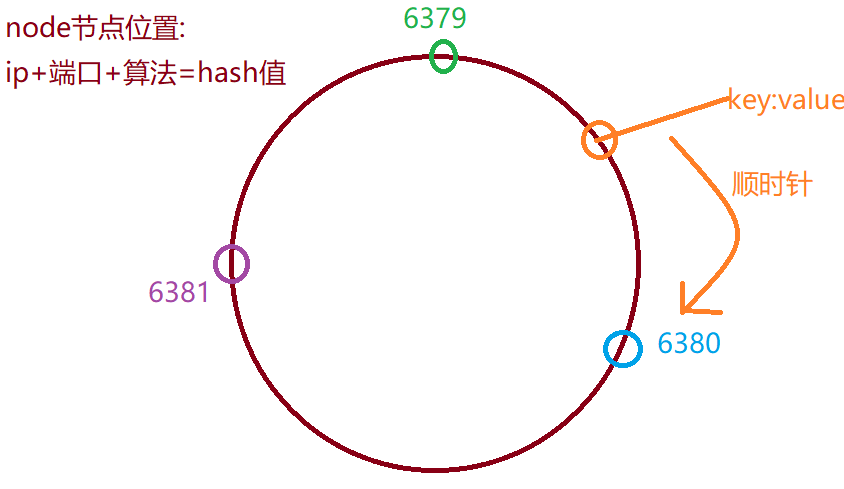
   如果java程序链接redis时，报错没有可用的集群节点时。检测虚拟机的IP地址和配置文件的地址是否匹配。

为什么：因为虚拟机的IP地址发生了变化。

## 二、一致性哈希算法

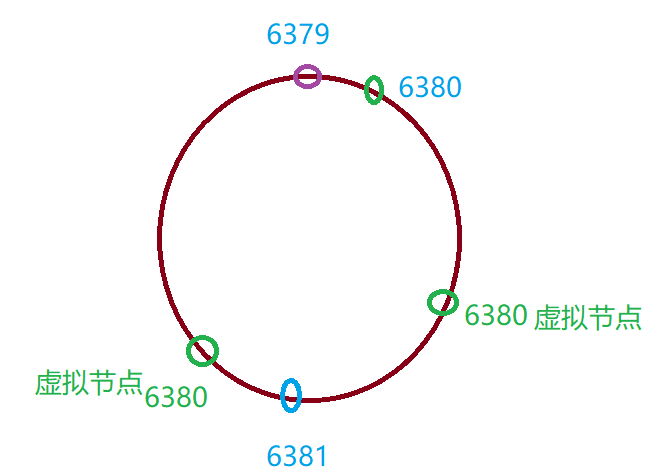
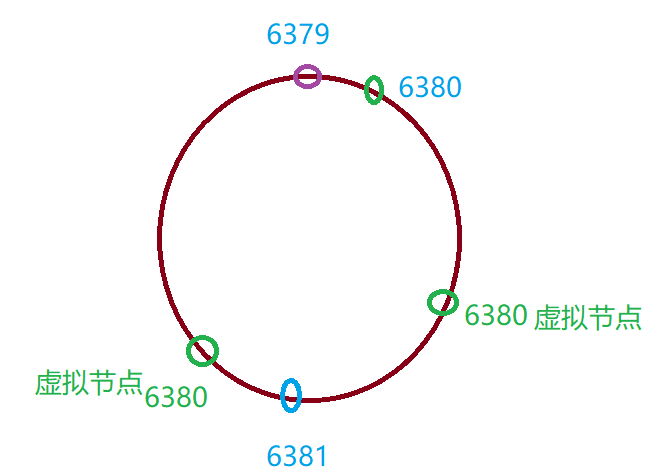
说明：redis分片时采用一致性哈希算法，实现数据动态的绑定。

术语：①节点(node) ：代表真实的redis服务器  ②key：用户保存/读取关键字



### 1、均衡性

问题：由于node节点是经过hash一致性算法计算的结果。那么可能会出现计算的节点所分配的内存不均。如果内存不均，会导致某些节点内存负载过高。特点说明：要求全部的数据，尽可能的均匀分配到不同的节点中，每个节点中保存的数据尽可能接近1/n实现步骤：   如果遇到节点负载不均时，会自动的启动虚拟节点，进行数据的平衡。



### 2、单调性

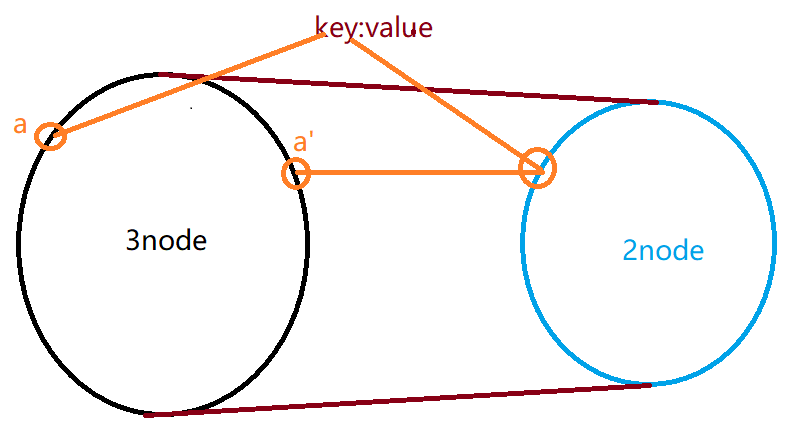
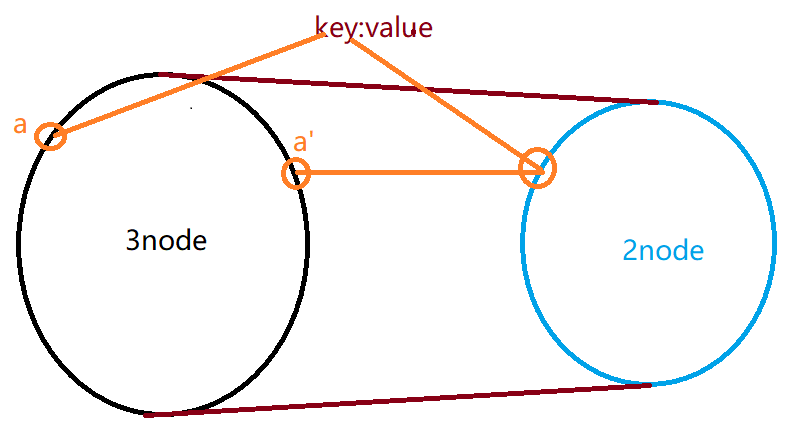
说明：如果节点的个数增加，原有的节点的挂载会自动的发生变化。将满足条件的数据自动的挂载到新的节点中。原则：尽可能保证原有的数据不变

总结：节点增加，数据动态发生迁移。

### 3、分散性

说明：由于分布式的项目部署，导致项目不能全部获取node节点。

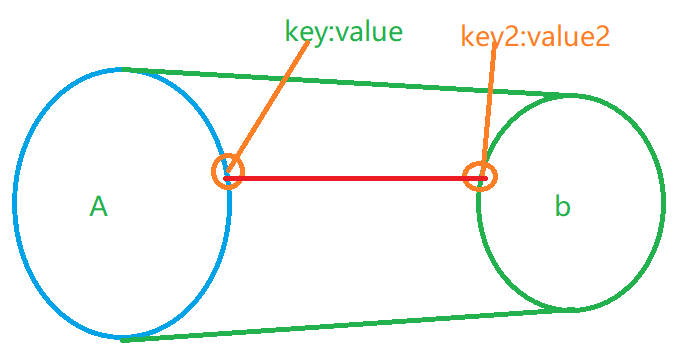
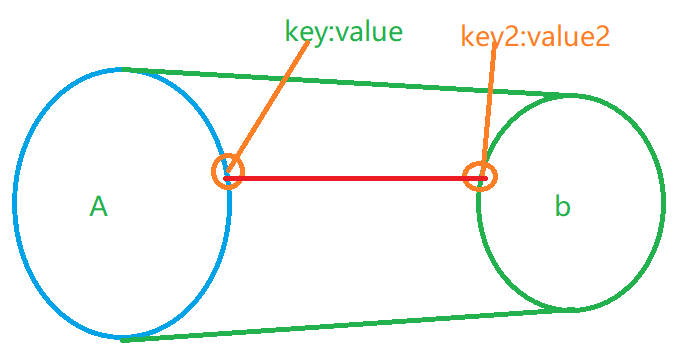
特点：一个key对应了多个位置。



### 4、负载

说明：负载是从另一个角度谈论分散性。

特点：一个位置，对应多个key



综上所述：

在分布式应用时，应该使用全部的node节点，避免出现这样的现象。

## 三、分片和哨兵说明

**分片后数据如何读取呢？**读取方式不变，一致性哈希算法保证了数据的多分片可读性。

### 1、哨兵主要功能

(1)集群监控，负责监控redis master 和slave进程是否正常工作。(2)消息通知，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员。(3)故障转移，如果master node挂掉了，会自动转移到slave node上。(4)配置中心，如果故障转移发生了，通知client客户端新的master地址。

### 2、哨兵核心知识

（1）哨兵至少需要3个实例，来保证自己的健壮性。

（2）哨兵+redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性

（3）对于哨兵+redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充分的测试和演练。

### 3、分片和哨兵优缺点

优点：①分片可以实现内存的动态的扩容 ②使用分片能够将数据分开保存，如果宕机/数据损坏只丢失1/n的数据 ③哨兵可以实现redis的高可用。

缺点：①如果分片中的一个节点宕机，那么整个分片将不能正常运行。② 分片中的数据因为一致性哈希的原因，可能会导致数据分布不是特别的平均。

③如果哨兵宕机，则整个服务陷入瘫痪。

### 4、redis哨兵主备切换的数据丢失问题

   两种丢失情况：

（1）异步复制导致的数据丢失：

因为master->slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，这些数据就丢失了。

（2）脑裂导致的数据丢失：

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着

这个时候，集群中就会出现两个master。

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master数据可能就会丢失。

因此master在恢复的时候，会被作为一个slave挂到新的master上，自己的数据会被清空，从新的master复制数据

5、解决异步复制和脑裂导致的数据丢失：

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了。

（1）减少异步复制的数据丢失

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失

上面的配置就确保了，如果跟任何一个slave丢了连接，在10秒后发现没有slave给自己ack，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

## 四、持久化策略说明

说明:因为redis中保存的数据都在内存中，当断电/宕机。缓存中的数据都会被清空。如果redis中没有配置持久化策略，安全性不够完善。

策略说明:

    1、RDB方式

该方式是redis默认选择的持久化策略

特点：持久化的效率更高，定期持久化可能会丢失数据

    2、AOF方式

该方式需要通过配置文件手动开启

特点：持久化效率低，每秒持久化/每次操作持久化，保证数据尽可能不丢失，文件过大导致恢复时操作时间过长。

  持久化步骤:

    1、当用户set操作时，redis中的数据会新增/更新

    2、这时根据用户选择的持久化的策略，自动的进行数据持久化操作，以下以RDB模式为例。

    3、定期会将redis中全部的数据通过xxx.RDB文件的方式保存。

    4、如果redis服务器宕机重启时，首先会加载持久化文件xxx.RDB。恢复内存中的数据。

    5、当用户使用redis时，这时redis内存中已经恢复了数据，为用户继续提供服务。

## 五、内存优化策略

1、需求说明

因为redis中的数据全部都写入内存中，当redis中的数据不能再次写入时这时redis服务会有问题，所以应该保证redis服务永远可写。如何实现redis数据内存优化呢？

LRU算法（Least Recently Used,最近最久未使用法）：内存管理的一种页面置换算法，思想：最近使用的页面数据会在未来一段时期内仍然被使用,已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。

2、内存优化手段:

volatile-lru ->  将设定超时时间的数据并且其中使用较少的数据进行删除

allkeys-lru ->   将redis中全部key进行LRU筛选,之后进行删除

volatile-random -> 设定了超时间的数据,随机删除

allkeys-random -> 全部的key随机删除

volatile-ttl ->  将已经设定了超时时间的数据,按照存活时间排序,将马上要过期的数据进行删除.

noeviction -> 不做任何操作,将报错信息返回给用户。

## 六、Redis 集群

说明：redis集群实质上将redis分片和redis哨兵的机制进行整合。redis集群中每个节点都可以与其他节点进行通讯，同时集群内部有心跳检测，如果节点发生宕机的现象，由所在集群的全部服务器负责推选，保证服务的正常运行；如果全部的从节点宕机，并且这时主节点宕机，那么整个集群才会奔溃。

## 七、Redis 高效原因

**1、纯内存访问**，Redis将所有数据放在内存中，内存的响应时间大约为100纳秒，这时Redis达到每秒万级别访问的重要基础；

       完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

**2、数据结构简单**，对数据操作也简单，Redis中的数据结构是专门进行设计的；

**3、采用单线程**，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

**4、使用多路I/O复用模型，且采用异步非阻塞IO**；Redis使用epoll作为I/O多路复用技术的实现，在加上Redis自身的事件处理模型将epoll中的链接、读写、关闭都转换为事件，不在网络I/O上浪费过多的时间；

**5、使用底层模型不同**，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

1、BIO：同步阻塞IO；NIO：同步非阻塞IO；AIO：异步非阻塞IO

阻塞I/O：调用阻塞 I/O 接口读写数据时，应用层线程或者进程挂起，一直等待系统准备好数据，之后继续执行。

非阻塞I/O：调用非阻塞 I/O 接口读写数据时，系统直接返回一个异常，如 EWOULDBLOCK，应用层线程或者进程继续执行，但是需要不断地去查询系统是否准备好数据，耗时，耗资源！

总结：阻塞、非阻塞 I/O 各有优缺点，阻塞占用资源少，但是面对高并发不友好，非阻塞适合高并发，但是占用资源多；

          阻塞和非阻塞 I/O **解决了应用层等待数据返回时的状态问题。**

2、同步I/O：应用层线程或者进程需要不断地查询系统内核数据是否准备好，如果没准备好，根据I/O阻塞和非阻塞的不同，决定是挂起还是继续执行。

异步I/O：应用层线程或者进程无需查询系统内核数据是否准备好，数据准备好系统会主动通知应用层，应用层在此期间也根据阻塞和非阻塞的不同决定是挂起还是继续执行。

总结：同步、非同步 I/O，非同步 I/O 显然更加高效，适合高并发的场景，

          同步和非同步 I/O **解决了系统内核如何返回数据给应用层的问题**。

3、I/O 多路复用：用一个线程来检查多个文件描述符（Socket）的就绪状态，比如调用select和poll函数，传入多个文件描述符（FileDescription，简称FD），如果有一个文件描述符（FileDescription）就绪，则返回，否则阻塞直到超时。就是派一个代表，同时监听多个文件描述符是否有数据到来。等着等着，如有有数据，就告诉某某你的数据来啦！赶紧来处理吧。

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以**让单个线程高效的处理多个连接请求**（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量

（1）select：缺点① 每次调用select()都需要把fd(文件描述符)从用户态拷贝到内核态，开销比较大；② 每次都需要在内核遍历传入的fd（文件描述符）；③ select支持文件数量比较小，默认是1024。

（2）poll：poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，支持的文件数量比较多，不仅仅是1024。

（3）epoll：select/poll只提供了一个函数，selct/poll函数，epoll提供三个函数：epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句 柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

（4）**epoll是如何解决 select 和 poll 的缺点的呢**？

    ①每次调用select()都需要把fd(文件描述符)从用户态拷贝到内核态，开销比较大；解决：在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

    ②每次都需要在内核遍历传入的fd（文件描述符）；解决：epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当数据就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule\_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

简而言之就是，在 epoll\_ctl（注册要监听的事件类型）时，为每一个fd指定一个回调函数，创建一个就绪链表，数据就绪时会调用回调函数，将fd加入就绪链表中，这样就可以只遍历准备好的fd了。

    ③epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048，和内存相关。

4、Redis 和 I/O 多路复用

（1）Redactor 设计模式：Redis 服务采用 Reactor 的方式来实现文件事件处理器（每一个网络连接其实都对应一个文件描述符（FD））；

        文件事件处理器使用 I/O 多路复用模块同时监听多个 FD，当 accept、read、write 和 close 文件事件产生时，文件事件处理器就会回调 FD 绑定的事件处理器。虽然整个文件事件处理器是在单线程上运行的，但是通过 I/O 多路复用模块的引入，实现了同时对多个 FD 读写的监控，提高了网络通信模型的性能，同时也可以保证整个 Redis 服务实现的简单。

（2）I/O多路复用模块

        I/O 多路复用模块封装了底层的 select、epoll、avport 以及 kqueue 这些 I/O 多路复用函数，为上层提供了相同的接口。

        子模块的选择：子模块的选择因为 Redis 需要在多个平台上运行，同时为了最大化执行的效率与性能，所以会**根据编译平台的不同选择不同的 I/O 多路复用函数作为子模块**，提供给上层统一的接口；在 Redis 中，我们通过宏定义的使用，合理的选择不同的子模块：

总结：Redis 对于 I/O 多路复用模块的设计非常简洁，通过宏保证了 I/O 多路复用模块在不同平台上都有着优异的性能，将不同的 I/O 多路复用函数封装成相同的 API 提供给上层使用。

整个模块使 Redis 能以单进程运行的同时服务成千上万个文件描述符，避免了由于多进程应用的引入导致代码实现复杂度的提升，减少了出错的可能性。

# Dubbo

配置

*# dubbo的配置  
# dubbo中的服务名称***spring.dubbo.application**=**item-web***# dubbo的通讯协议名称***spring.dubbo.protocol.name**=**dubbo***# zookeeper注册中心的地址***spring.dubbo.registry.address**=**192.168.1.104:2181***# zookeeper的通讯协议的名称***spring.dubbo.registry.protocol**=**zookeeper***# dubbo的服务的扫描路径***spring.dubbo.base-package**=**com.gmall***# 设置超时时间***spring.dubbo.consumer.timeout**=**600000***# 设置是否检查服务存在***spring.dubbo.consumer.check**=**false**

Bean 注入使用 @Refrence，Service使用 dubbo的@Service

注册中心挂了，还可以用，因为本地有缓存。

看笔记！！！

# Zookeeper

## [Zookeeper到底是干嘛的](https://www.cnblogs.com/ultranms/p/9585191.html)

在Zookeeper的官网上有这么一句话：ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

这大概描述了Zookeeper主要可以干哪些事情：配置管理，名字服务，提供分布式同步以及集群管理。那这些服务又到底是什么呢？我们为什么需要这样的服务？我们又为什么要使用Zookeeper来实现呢，使用Zookeeper有什么优势？接下来我会挨个介绍这些到底是什么，以及有哪些开源系统中使用了。

## 配置管理

在我们的应用中除了代码外，还有一些就是各种配置。比如数据库连接等。一般我们都是使用配置文件的方式，在代码中引入这些配置文件。但是当我们只有一种配置，只有一台服务器，并且不经常修改的时候，使用配置文件是一个很好的做法，但是如果我们配置非常多，有很多服务器都需要这个配置，而且还可能是动态的话使用配置文件就不是个好主意了。这个时候往往需要寻找一种集中管理配置的方法，我们在这个集中的地方修改了配置，所有对这个配置感兴趣的都可以获得变更。比如我们可以把配置放在数据库里，然后所有需要配置的服务都去这个数据库读取配置。但是，因为很多服务的正常运行都非常依赖这个配置，所以需要这个集中提供配置服务的服务具备很高的可靠性。一般我们可以用一个集群来提供这个配置服务，但是用集群提升可靠性，那如何保证配置在集群中的一致性呢？ 这个时候就需要使用一种实现了一致性协议的服务了。Zookeeper就是这种服务，它使用Zab这种一致性协议来提供一致性。现在有很多开源项目使用Zookeeper来维护配置，比如在HBase中，客户端就是连接一个Zookeeper，获得必要的HBase集群的配置信息，然后才可以进一步操作。还有在开源的消息队列Kafka中，也使用Zookeeper来维护broker的信息。在Alibaba开源的SOA框架Dubbo中也广泛的使用Zookeeper管理一些配置来实现服务治理。

## 名字服务

名字服务这个就很好理解了。比如为了通过网络访问一个系统，我们得知道对方的IP地址，但是IP地址对人非常不友好，这个时候我们就需要使用域名来访问。但是计算机是不能是别域名的。怎么办呢？如果我们每台机器里都备有一份域名到IP地址的映射，这个倒是能解决一部分问题，但是如果域名对应的IP发生变化了又该怎么办呢？于是我们有了DNS这个东西。我们只需要访问一个大家熟知的(known)的点，它就会告诉你这个域名对应的IP是什么。在我们的应用中也会存在很多这类问题，特别是在我们的服务特别多的时候，如果我们在本地保存服务的地址的时候将非常不方便，但是如果我们只需要访问一个大家都熟知的访问点，这里提供统一的入口，那么维护起来将方便得多了。

## 分布式锁

其实在第一篇文章中已经介绍了Zookeeper是一个分布式协调服务。这样我们就可以利用Zookeeper来协调多个分布式进程之间的活动。比如在一个分布式环境中，为了提高可靠性，我们的集群的每台服务器上都部署着同样的服务。但是，一件事情如果集群中的每个服务器都进行的话，那相互之间就要协调，编程起来将非常复杂。而如果我们只让一个服务进行操作，那又存在单点。通常还有一种做法就是使用分布式锁，在某个时刻只让一个服务去干活，当这台服务出问题的时候锁释放，立即fail over到另外的服务。这在很多分布式系统中都是这么做，这种设计有一个更好听的名字叫Leader Election(leader选举)。比如HBase的Master就是采用这种机制。但要注意的是分布式锁跟同一个进程的锁还是有区别的，所以使用的时候要比同一个进程里的锁更谨慎的使用。

## 集群管理

在分布式的集群中，经常会由于各种原因，比如硬件故障，软件故障，网络问题，有些节点会进进出出。有新的节点加入进来，也有老的节点退出集群。这个时候，集群中其他机器需要感知到这种变化，然后根据这种变化做出对应的决策。比如我们是一个分布式存储系统，有一个中央控制节点负责存储的分配，当有新的存储进来的时候我们要根据现在集群目前的状态来分配存储节点。这个时候我们就需要动态感知到集群目前的状态。还有，比如一个分布式的SOA架构中，服务是一个集群提供的，当消费者访问某个服务时，就需要采用某种机制发现现在有哪些节点可以提供该服务(这也称之为服务发现，比如Alibaba开源的SOA框架Dubbo就采用了Zookeeper作为服务发现的底层机制)。还有开源的Kafka队列就采用了Zookeeper作为Cosnumer的上下线管理。

自己的一些理解：

Leader可以接受client请求，也接收其他Server转发的写请求，负责更新系统状态。 Follower也可以接收client请求，如果是写请求将转发给Leader来更新系统状态，读请求则由Follower的内存数据库直接响应。

## 事务操作

在ZooKeeper中，能改变ZooKeeper服务器状态的操作称为事务操作。一般包括数据节点创建与删除、数据内容更新和客户端会话创建与失效等操作。对应每一个事务请求，ZooKeeper都会为其分配一个全局唯一的事务ID，用 ZXID 表示，通常是一个64位的数字。每一个 ZXID对应一次更新操作，从这些 ZXID 中可以间接地识别出 ZooKeeper 处理这些事务操作请求的

全局顺序。

Watcher(事件监听器)

ZooKeeper 中一个很重要的特性。ZooKeeper允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知到感兴趣的客户端上去。该机制是 ZooKeeper 实现分布式协调服务的重要特性。

ZooKeeper数据模型的结构整体上可以看作是一棵树，每个节点称做一个ZNode。每个ZNode都可以通过其路径唯一标识在每个ZNode上可存储少量数据(默认是1M, 可以通过配置修改, 通常不建议在ZNode上存储大量的数据)

ZooKeeper支持一种Watch操作，Client可以在某个ZNode上设置一个Watcher，来Watch该ZNode上的变化。如果该ZNode上有相应的变化，就会触发这个Watcher，把相应的事件通知给设置Watcher的Client。需要注意的是，ZooKeeper中的Watcher是一次性的，即触发一次就会被取消，如果想继续Watch的话，需要客户端重新设置Watcher

节点状态

每个集群中的节点都有一个状态 LOOKING, FOLLOWING, LEADING, OBSERVING。都属于这4种，每个节点启动的时候都是LOOKING状态，如果这个节点参与选举但最后不是leader，则状态是FOLLOWING，如果不参与选举则是OBSERVING，leader的状态是LEADING。

关于ZooKeeper集群服务器数

ZooKeeper 官方确实给出了关于奇数的建议，但绝大部分 ZooKeeper 用户对于这个建议认识有偏差。一个 ZooKeeper 集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信。基于这个特性，如果想搭建一个能够允许 N 台机器 down 掉的集群，那么就要部署一个由 2\*N+1 台服务器构成的 ZooKeeper 集群。因此，一个由 3 台机器构成的 ZooKeeper 集群，能够在挂掉 1 台机器后依然正常工作，而对于一个由 5 台服务器构成的 ZooKeeper 集群，能够对 2 台机器挂掉的情况进行容灾。注意，如果是一个由6台服务器构成的 ZooKeeper 集群，同样只能够挂掉 2 台机器，因为如果挂掉 3 台，剩下的机器就无法实现过半了。

## zookeeper有这样一个特性：

**【集群中只要有超过过半的机器是正常工作的，那么整个集群对外就是可用的】**

 也就是说如果有2个zookeeper，那么只要有1个死了zookeeper就不能用了，因为1没有过半，所以2个zookeeper的死亡容忍度为0；

同理，要是有3个zookeeper，一个死了，还剩下2个正常的，过半了，所以3个zookeeper的容忍度为1；

同理你多列举几个：2->0;3->1;4->1;5->2;6->2会发现一个规律，2n和2n-1的容忍度是一样的，

都是n-1，所以为了更加高效，何必增加那一个不必要的zookeeper呢。

## ZooKeeper可伸缩性

那么，ZooKeeper为什么要引入Observer这个角色呢？其实在ZooKeeper中引入Observer，主要是为了使ZooKeeper具有更好的可伸缩性。那么，何为可伸缩性？关于伸缩性，对于不同的人意味着不同的事情。 而在这里是说，如果我们的工作负载可以通过给系统分配更多的资源来分担，那么这个系统就是可伸缩的；一个不可伸缩的系统却无法通过增加资源来提升性能，甚至会在工作负载增加时，性能会急剧下降。

在Observer出现以前，ZooKeeper的伸缩性由Follower来实现，我们可以通过添加Follower节点的数量来保证ZooKeeper服务的读性能。但是随着Follower节点数量的增加，ZooKeeper服务的写性能受到了影响。

简单来说，Zab协议规定：来自Client的所有写请求，都要转发给ZK服务中唯一的Server**—**Leader，由Leader根据该请求发起一个Proposal（请求）。然后，其他的Server对该Proposal（请求）进行Vote（投票）。之后，Leader对Vote（投票）进行收集，当Vote数量过半时Leader会向所有的Server发送一个通知消息。最后，当Client所连接的Server收到该消息时，会把该操作更新到内存中并对Client的写请求做出回应。

　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　ZK 写请求工作流程图

从图中我们可以看出， ZooKeeper 服务器在上述协议中实际扮演了两个职能。它们一方面从客户端接受连接与操作请求，另一方面对操作结果进行投票。这两个职能在 ZooKeeper集群扩展的时候彼此制约。例如，当我们希望增加 ZK服务中Client数量的时候，那么我们就需要增加Server的数量，来支持这么多的客户端。然而，从Zab协议对写请求的处理过程中我们可以发现，增加服务器的数量，则增加了对协议中投票过程的压力。因为Leader节点必须等待集群中过半Server响应投票，于是节点的增加使得部分计算机运行较慢，从而拖慢整个投票过程的可能性也随之提高，写操作也会随之下降。这正是我们在实际操作中看到的问题——随着 ZooKeeper 集群变大，写操作的吞吐量会下降。

 所以，我们不得不得在增加Client数量的期望和我们希望保持较好吞吐性能的期望间进行权衡。要打破这一耦合关系，我们引入了不参与投票的服务器，称为Observer。 Observer可以接受客户端的连接，并将写请求转发给Leader节点。但是，Leader节点不会要求 Observer参加投票。相反，Observer不参与投票过程，仅仅在上述第3歩那样，和其他服务节点一起得到投票结果。

## Zookeeper工作流程-Leader

1 .恢复数据；

2 .维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；

3 .Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

PING 消息是指Learner的心跳信息；

REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；

ACK消息是 Follower的对提议的回复，超过半数的Follower通过，则commit该提议；

REVALIDATE消息是用来延长SESSION有效时间。

Zookeeper工作流程-Follower

Follower主要有四个功能：

1.向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；

2.接收Leader消息并进行处理；

3.接收Client的请求，如果为写请求，发送给Leader进行投票；

4.返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1 .PING消息： 心跳消息；

2 .PROPOSAL消息：Leader发起的提案，要求Follower投票；

3 .COMMIT消息：服务器端最新一次提案的信息；

4 .UPTODATE消息：表明同步完成；

5 .REVALIDATE消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；

6 .SYNC消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

## Zookeeper节点数据操作流程

注：

1.在Client向Follwer发出一个写的请求

2.Follwer把请求发送给Leader

3.Leader接收到以后开始发起投票并通知Follwer进行投票

4.Follwer把投票结果发送给Leader

5.Leader将结果汇总后如果需要写入，则开始写入同时把写入操作通知给Leader，然后commit;

6.Follwer把请求结果返回给Client

 Zookeeper设计目的

1.最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2.可靠性：具有简单、健壮、良好的性能，如果消息被到一台服务器接受，那么它将被所有的服务器接受。

3.实时性：Zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

4.等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5.原子性：更新只能成功或者失败，没有中间状态。

6.顺序性：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

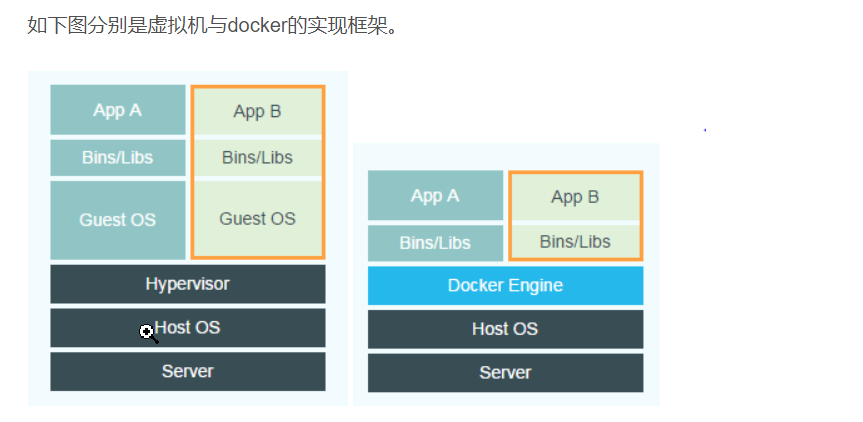
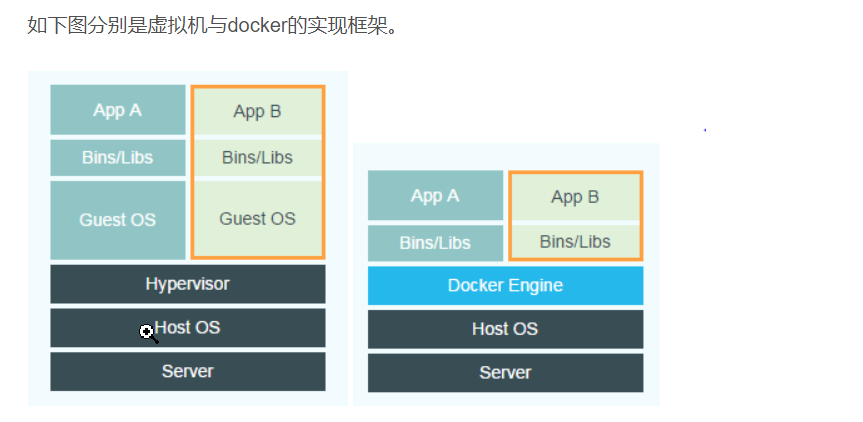
# Docker

## Docker简介

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口。

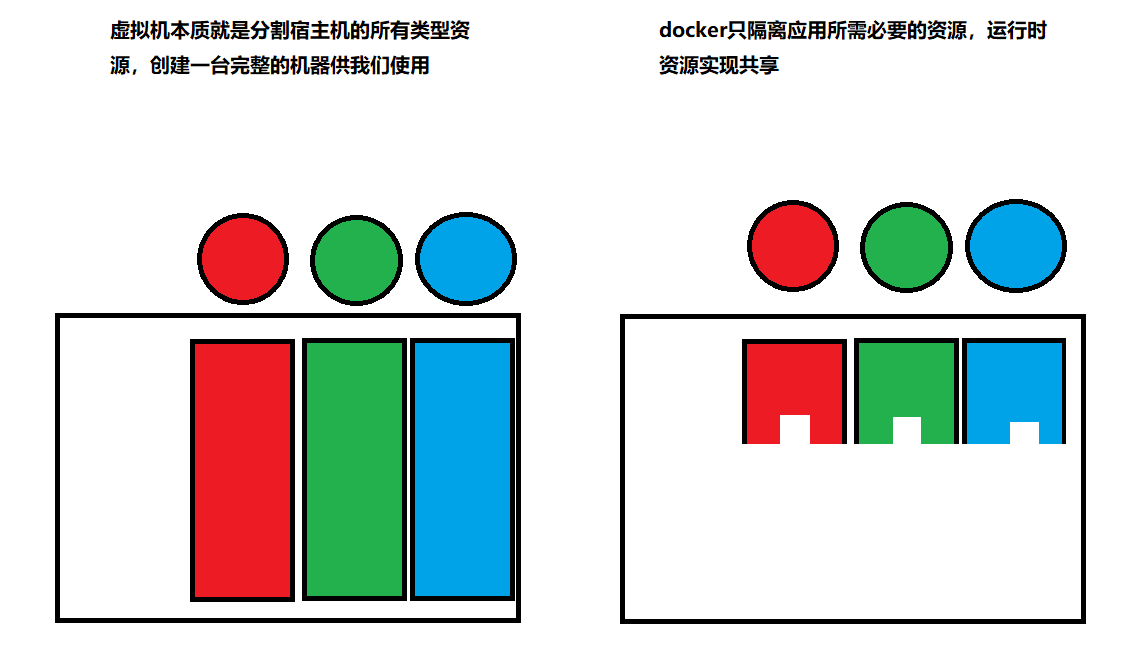
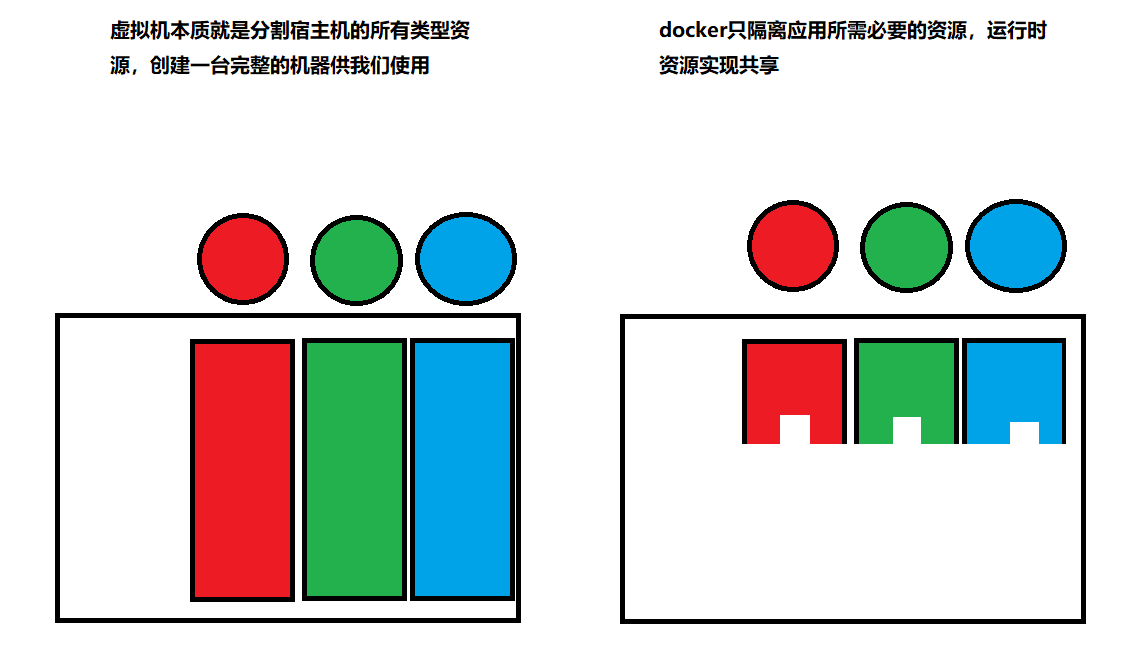
## Docker与虚拟机对比

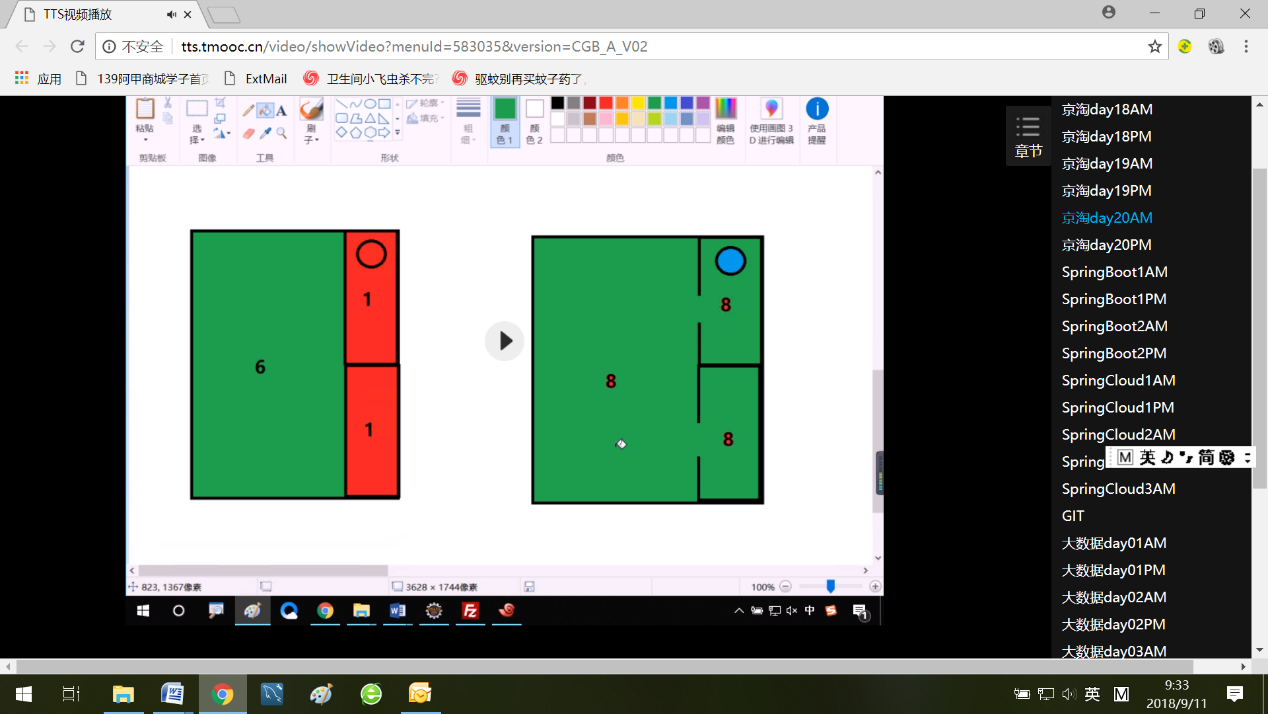
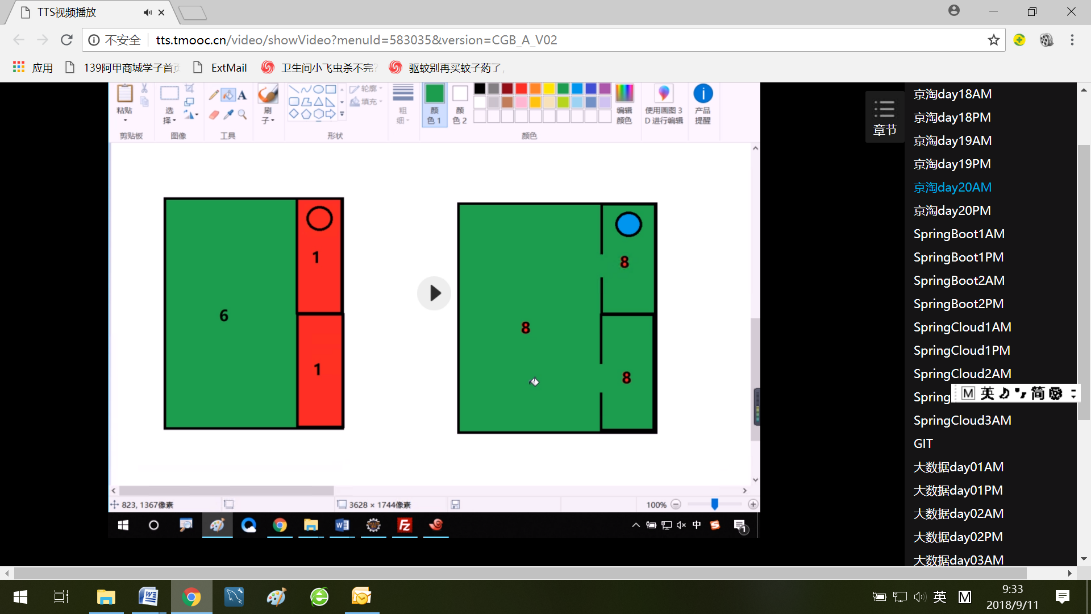
每个虚拟器需要安装一个操作系统

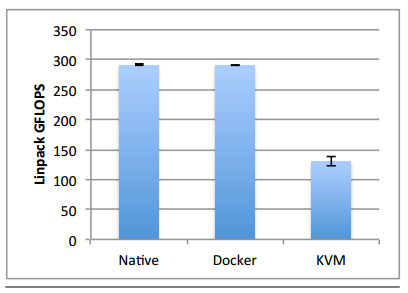
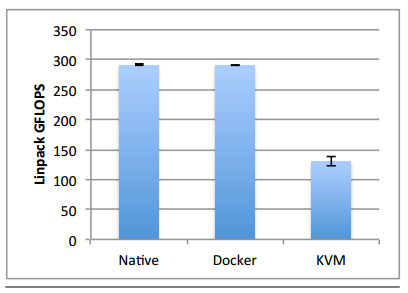


虚拟机：资源全部隔离，cpu、内存、磁盘相互独立，一台宿主机中可以安装的虚拟机数量是一定的，有限的，移植性不强（文件很大，拷贝传输困难）。

Docker：部分共享部分隔离，共享cpu，内存，磁盘甚至操作系统，极为轻量，可以有更多的个体对外提供服务。移植性强，共享仓库，移植时只需传输脚本文件。（类似maven，拿到pom文件就可以拿到所有的jar包）







由于以上提到的docker的特点，运行时资源共享宿主机，相当于在宿主机上边安装了一些应用，之后理想状态下享用宿主机的所有资源，所以性能与宿主机一样高，而虚拟机将资源完全划分出来并封闭不进行共享，所以他只能享用到自己分到的部分资源，也就是分多少用多少，并且还要被操作系统等资源占用很大一部分。

## Docker五大要素

1. 沙箱：隔离，将应用之间的必要资源隔离开，防止互相影响，应用之间互相独立。
2. 镜像：模板，centos+tomcat+mysql+redis,镜像可以拆分，传输用dockerfile
3. 容器：镜像的实例，镜像只读，容器可写，容器中可以保存应用产生的零时文件
4. 数据卷：挂载到容器上，用于保存必要数据。比如容器中数据库的数据，但是不建议用，因为重启容器需要挂载同步数据，效率较慢，解决方式：数据库拆分
5. 仓库：与maven仓库概念一致，应用都会保存在仓库中，创建容器是根据镜像的规定进行拉取，可以共享。

## docker命令

|  |  |
| --- | --- |
| **命令** | **用法** |
| yum -y install docker-ce | 下载最新版的docker |
| service docker start | 启动Docker服务 |
| service docker stop | 停止Docker服务 |
| service docker restart | 重新启动Docker服务 |
| docker version | 查看Docker的版本号 |
| docker pull 镜像地址:版本 | 从镜像仓库中下载 |
| docker save a2a69ca5184a > jt-centOS6.tar | 根据镜像id导出镜像 |
| docker save -o redis-3.2.8.tar redis:3.2.8 | 根据镜像名称导出镜像 |
| docker load -i docker-centos-6.5.tar | 指定jar包导入镜像文件 |
| docker rmi a2a69ca5184a | 根据Id号删除镜像文件 |
| docker rmi -f a2a69ca5184a | 强制删除镜像文件    删除镜像前需要先关闭容器 |
| docker images | 查询所有镜像文件 |
| docker inspect index.alauda.cn/tutum/centos:6.5 | 查看镜像文件细节信息 |
| docker tag 旧镜像名称和端口 redis-ali:0.0.1 | 修改镜像的名称 |
| docker build -t 镜像名称:版本号 | 根据dockerfile来创建镜像文件 |
| docker run -d --name 容器名 镜像名:版本号 | 根据镜像名称启动容器 |
| docker run -d --name 容器名(自定) 镜像id号 | 根据镜像id启动容器 |
| docker run -d -p 虚拟机端口:镜像端口 --name 容器名 镜像名:版本号 | 启动容器,并指定暴露端口 |
| docker ps | 查看活动的docker容器进程 |
| Docker ps -a/-all | 查看全部的容器 |
| docker exec -it 容器id bash | 进入指定的容器 |
| docker stop 容器Id号 | 停止指定容器 |
| docker start 容器Id号 | 启动创建好的容器 |
| docker stop $(docker ps -q) & docker rm $(docker ps -aq) | 关闭和删除所有的容器 |
| docker rm 容器Id | 删除指定的容器 |

## Dockerfile

Dockerfile是docker应用的核心，通过定义dockerfile来实现相同环境的自动部署，是docker实现快熟部署成百上千服务器的核心手段。

Dockerfile通过逐层layer叠加，使资源得到重复利用，同时变化无穷。

注意：FROM指定的镜像，本地如果有直接使用，无需网上下载。

### 关键字及含义

|  |  |  |
| --- | --- | --- |
| 序号 | 关键字 | 说明 |
|  | FROM | 指定基础镜像的来源 |
|  | MAINTAINER | 作者 |
|  | ADD | 复制文件，会自动解压 |
|  | WORKDIR | 设置当前工作目录 cd |
|  | VOLUME | 设置数据卷，挂载主机目录 |
|  | EXPOSE | 指定对外暴漏的端口 |
|  | RUN | 执行命令 sh |
|  | CMD | 执行命令 exec，一个Dockerfile只能一个 |
|  | COPY | 复制文件 |
|  | ENTRYPOINT | docker run时参数可以覆盖，指定参数值 |

# Git

## Git 的文件状态

 文件分为两种状态：

   （1）已跟踪的文件是指本来就被纳入版本控制管理的文件，在上次快照中有它们的记录，工作一段时间后，它们的状态可能是未更新，已修改或者已放入暂存区。

   （2）而所有其他文件都属于未跟踪文件。

untracked :未追踪，新增的文件；

unmodified:未修改，新增文件放入add（放入暂存区）后或修改的文件已提交；

modified:修改，修改后未提交

staged: 文件放入暂存区后

## 2、Git 的分支

本质上仅仅是个指向 commit 对象的可变指针，在每次提交的时候都会自动向前移动。

Git 中的分支实际上仅是一个包含所指**对象校验和**（40 个字符长度 SHA-1 字串）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。

这和大多数版本控制系统形成了鲜明对比，它们管理分支大多采取备份所有项目文件到特定目录的方式，所以根据项目文件数量和大小不同，可能花费的时间也会有相当大的差别，快则几秒，慢则数分钟。而 Git 的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换。同时，因为每次提交时都记录了祖先信息（译注：即 parent 对象），将来要合并分支时，寻找恰当的合并基础（译注：即共同祖先）的工作其实已经自然而然地摆在那里了，所以实现起来非常容易。Git 鼓励开发者频繁使用分支，正是因为有着这些特性作保障。

Git 会把工作目录的内容恢复为检出某分支时它所指向的那个提交对象的快照。它会自动添加、删除和修改文件以确保目录的内容和你当时提交时完全一样。

## 重点：

前提：假设现在 master 版本为 C2，HEAD 指向 master。

（1）现在在在master 分支创建 iss53 分支，此时iss53 的版本是 C2，我们切换到 iss53 分支，HEAD 指向 iss53 在 iss53 分支进行bug的修改，提交代码，这时iss53 的分支的版本更新到 C3，master版本还是C2。

（2）在 iss53 工作时，我们接到了一个紧急的任务，这时我们再新建一个分支 hotfix，切换到 hotfix 工作，此时 master 版本是 C2，hotfix 的版本是 C2，iss53的版本是C3；HEAD 指向 hotfix;

    注：切换到 hotfix 分支时，我们需要将修改状态 unmodified 和未追踪状态 untracked 的代码进行commit（不需要 push），再进行切换，这时 master 版本为 C2，hotfix 的版本是 C2，我们是看不到 iss53 中修改的代码的，只会显示 master 的 C2 版本代码。

（3）hotfix 在修改后push代码到 hotfix 分支，此时hotfix 版本为 C4，我们将代码合并到 master，git checkout master; git merge hotfix；现在master 和 hotfix 指向相同的对象。

注：合并时出现了“Fast forward”的提示。由于当前 master 分支所在的提交对象是要并入的 hotfix 分支的直接上游，Git 只需把 master 分支指针直接右移。换句话说，如果顺着一个分支走下去可以到达另一个分支的话，那么 Git 在合并两者时，只会简单地把指针右移，因为这种单线的历史分支不存在任何需要解决的分歧，所以这种合并过程可以称为快进（Fast forward）

（4）现在切换回 iss53 继续修改之前问题，提交 push，版本为 C5。

（5）修改完成后，push 到 iss53 中，在进行合并。

注：这次合并操作的底层实现，并不同于之前 hotfix 的并入方式。因为这次你的开发历史是从更早的地方开始分叉的。由于当前 master 分支所指向的提交对象（C4）并不是 iss53 分支的直接祖先，Git 不得不进行一些额外处理。就此例而言，**Git 会用两个分支的末端（C4 和 C5）(此处和 C3 无关，因为 C5 是包含 C3 的所有内容的)以及它们的共同祖先（C2）进行一次简单的三方合并计算。图 3-16 用红框标出了 Git 用于合并的三个提交对象。**

（6）合并完成后master 版本为 C6。

（7）合并之后的冲突解决：直接右击 Unmerged paths：下的路径，选择open，然后修改后再次add，commit，push。

看笔记！！！

# Mycat

## [分库分表的几种常见玩法及如何解决跨库查询等问题](https://www.cnblogs.com/cxxjohnson/p/9048518.html)

在谈论数据库架构和数据库优化的时候，我们经常会听到“分库分表”、“分片”、“Sharding”…这样的关键词。让人感到高兴的是，这些朋友所服务的公司业务量正在（或者即将面临）高速增长，技术方面也面临着一些挑战。让人感到担忧的是，他们系统真的就需要“分库分表”了吗？“分库分表”有那么容易实践吗？为此，笔者整理了分库分表中可能遇到的一些问题，并结合以往经验介绍了对应的解决思路和建议。

## 垂直分表

垂直分表在日常开发和设计中比较常见，通俗的说法叫做“大表拆小表”，拆分是基于关系型数据库中的“列”（字段）进行的。通常情况，某个表中的字段比较多，可以新建立一张“扩展表”，将不经常使用或者长度较大的字段拆分出去放到“扩展表”中，如下图所示：

### 小结

在字段很多的情况下，拆分开确实更便于开发和维护（笔者曾见过某个遗留系统中，一个大表中包含100多列的）。某种意义上也能避免“跨页”的问题（MySQL、MSSQL底层都是通过“数据页”来存储的，“跨页”问题可能会造成额外的性能开销，这里不展开，感兴趣的朋友可以自行查阅相关资料进行研究）。

拆分字段的操作建议在数据库设计阶段就做好。如果是在发展过程中拆分，则需要改写以前的查询语句，会额外带来一定的成本和风险，建议谨慎。

## 垂直分库

垂直分库在“微服务”盛行的今天已经非常普及了。基本的思路就是按照业务模块来划分出不同的数据库，而不是像早期一样将所有的数据表都放到同一个数据库中。如下图：

小结

系统层面的“服务化”拆分操作，能够解决业务系统层面的耦合和性能瓶颈，有利于系统的扩展维护。而数据库层面的拆分，道理也是相通的。与服务的“治理”和“降级”机制类似，我们也能对不同业务类型的数据进行“分级”管理、维护、监控、扩展等。

众所周知，数据库往往最容易成为应用系统的瓶颈，而数据库本身属于“有状态”的，相对于Web和应用服务器来讲，是比较难实现“横向扩展”的。数据库的连接资源比较宝贵且单机处理能力也有限，在高并发场景下，垂直分库一定程度上能够突破IO、连接数及单机硬件资源的瓶颈，是大型分布式系统中优化数据库架构的重要手段。

然后，很多人并没有从根本上搞清楚为什么要拆分，也没有掌握拆分的原则和技巧，只是一味的模仿大厂的做法。导致拆分后遇到很多问题（例如：跨库join，分布式事务等）。

## 水平分表

水平分表也称为横向分表，比较容易理解，就是将表中不同的数据行按照一定规律分布到不同的数据库表中（这些表保存在同一个数据库中），这样来降低单表数据量，优化查询性能。最常见的方式就是通过主键或者时间等字段进行Hash和取模后拆分。如下图所示：

### 小结

水平分表，能够降低单表的数据量，一定程度上可以缓解查询性能瓶颈。但本质上这些表还保存在同一个库中，所以库级别还是会有IO瓶颈。所以，一般不建议采用这种做法。

## 水平分库分表

水平分库分表与上面讲到的水平分表的思想相同，唯一不同的就是将这些拆分出来的表保存在不同的数据库中。这也是很多大型互联网公司所选择的做法。如下图：

某种意义上来讲，有些系统中使用的“冷热数据分离”（将一些使用较少的历史数据迁移到其他的数据库中。而在业务功能上，通常默认只提供热点数据的查询），也是类似的实践。在高并发和海量数据的场景下，分库分表能够有效缓解单机和单库的性能瓶颈和压力，突破IO、连接数、硬件资源的瓶颈。当然，投入的硬件成本也会更高。同时，这也会带来一些复杂的技术问题和挑战（例如：跨分片的复杂查询，跨分片事务等）

### 分库分表的难点

垂直分库带来的问题和解决思路：

### 跨库join的问题

在拆分之前，系统中很多列表和详情页所需的数据是可以通过sql join来完成的。而拆分后，数据库可能是分布式在不同实例和不同的主机上，join将变得非常麻烦。而且基于架构规范，性能，安全性等方面考虑，一般是禁止跨库join的。那该怎么办呢？首先要考虑下垂直分库的设计问题，如果可以调整，那就优先调整。如果无法调整的情况，下面笔者将结合以往的实际经验，总结几种常见的解决思路，并分析其适用场景。

### 跨库Join的几种解决思路

全局表

所谓全局表，就是有可能系统中所有模块都可能会依赖到的一些表。比较类似我们理解的“数据字典”。为了避免跨库join查询，我们可以将这类表在其他每个数据库中均保存一份。同时，这类数据通常也很少发生修改（甚至几乎不会），所以也不用太担心“一致性”问题。

字段冗余

这是一种典型的反范式设计，在互联网行业中比较常见，通常是为了性能来避免join查询。

举个电商业务中很简单的场景：

“订单表”中保存“卖家Id”的同时，将卖家的“Name”字段也冗余，这样查询订单详情的时候就不需要再去查询“卖家用户表”。

字段冗余能带来便利，是一种“空间换时间”的体现。但其适用场景也比较有限，比较适合依赖字段较少的情况。最复杂的还是数据一致性问题，这点很难保证，可以借助数据库中的触发器或者在业务代码层面去保证。当然，也需要结合实际业务场景来看一致性的要求。就像上面例子，如果卖家修改了Name之后，是否需要在订单信息中同步更新呢？

数据同步

定时A库中的tab\_a表和B库中tbl\_b有关联，可以定时将指定的表做同步。当然，同步本来会对数据库带来一定的影响，需要性能影响和数据时效性中取得一个平衡。这样来避免复杂的跨库查询。笔者曾经在项目中是通过ETL工具来实施的。

系统层组装

在系统层面，通过调用不同模块的组件或者服务，获取到数据并进行字段拼装。说起来很容易，但实践起来可真没有这么简单，尤其是数据库设计上存在问题但又无法轻易调整的时候。

具体情况通常会比较复杂。下面笔者结合以往实际经验，并通过伪代码方式来描述。

简单的列表查询的情况

伪代码很容易理解，先获取“我的提问列表”数据，然后再根据列表中的UserId去循环调用依赖的用户服务获取到用户的RealName，拼装结果并返回。

有经验的读者一眼就能看出上诉伪代码存在效率问题。循环调用服务，可能会有循环RPC，循环查询数据库…不推荐使用。再看看改进后的：

这种实现方式，看起来要优雅一点，其实就是把循环调用改成一次调用。当然，用户服务的数据库查询中很可能是In查询，效率方面比上一种方式更高。（坊间流传In查询会全表扫描，存在性能问题，传闻不可全信。其实查询优化器都是基本成本估算的，经过测试，在In语句中条件字段有索引的时候，条件较少的情况是会走索引的。这里不细展开说明，感兴趣的朋友请自行测试）。

### 小结

简单字段组装的情况下，我们只需要先获取“主表”数据，然后再根据关联关系，调用其他模块的组件或服务来获取依赖的其他字段（如例中依赖的用户信息），最后将数据进行组装。

通常，我们都会通过缓存来避免频繁RPC通信和数据库查询的开销。

列表查询带条件过滤的情况

在上述例子中，都是简单的字段组装，而不存在条件过滤。看拆分前的SQL：

这种连接查询并且还带条件过滤的情况，想在代码层面组装数据其实是非常复杂的（尤其是左表和右表都带条件过滤的情况会更复杂），不能像之前例子中那样简单的进行组装了。试想一下，如果像上面那样简单的进行组装，造成的结果就是返回的数据不完整，不准确。

有如下几种解决思路：

1. 查出所有的问答数据，然后调用用户服务进行拼装数据，再根据过滤字段state字段进行过滤，最后进行排序和分页并返回。

这种方式能够保证数据的准确性和完整性，但是性能影响非常大，不建议使用。

1. 查询出state字段符合/不符合的UserId，在查询问答数据的时候使用in/not in进行过滤，排序，分页等。过滤出有效的问答数据后，再调用用户服务获取数据进行组装。

这种方式明显更优雅点。笔者之前在某个项目的特殊场景中就是采用过这种方式实现。

跨库事务（分布式事务）的问题

按业务拆分数据库之后，不可避免的就是“分布式事务”的问题。以往在代码中通过spring注解简单配置就能实现事务的，现在则需要花很大的成本去保证一致性。这里不展开介绍，

感兴趣的读者可以自行参考《分布式事务一致性解决方案》，链接地址：

http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency

## 垂直分库总结和实践建议

本篇中主要描述了几种常见的拆分方式，并着重介绍了垂直分库带来的一些问题和解决思路。读者朋友可能还有些问题和疑惑。

1. 我们目前的数据库是否需要进行垂直分库？

根据系统架构和公司实际情况来，如果你们的系统还是个简单的单体应用，并且没有什么访问量和数据量，那就别着急折腾“垂直分库”了，否则没有任何收益，也很难有好结果。

切记，“过度设计”和“过早优化”是很多架构师和技术人员常犯的毛病。

2. 垂直拆分有没有原则或者技巧？

没有什么黄金法则和标准答案。一般是参考系统的业务模块拆分来进行数据库的拆分。比如“用户服务”，对应的可能就是“用户数据库”。但是也不一定严格一一对应。有些情况下，数据库拆分的粒度可能会比系统拆分的粒度更粗。笔者也确实见过有些系统中的某些表原本应该放A库中的，却放在了B库中。有些库和表原本是可以合并的，却单独保存着。还有些表，看起来放在A库中也OK，放在B库中也合理。

如何设计和权衡，这个就看实际情况和架构师/开发人员的水平了。

3. 上面举例的都太简单了，我们的后台报表系统中join的表都有n个了，

分库后该怎么查？

有很多朋友跟我提过类似的问题。其实互联网的业务系统中，本来就应该尽量避免join的，如果有多个join的，要么是设计不合理，要么是技术选型有误。请自行科普下OLAP和OLTP，报表类的系统在传统BI时代都是通过OLAP数据仓库去实现的（现在则更多是借助离线分析、流式计算等手段实现），而不该向上面描述的那样直接在业务库中执行大量join和统计。

# 消息队列

## 1.消息队列

arrayList,queue

MQ全称为Message Queue, 消息队列（MQ）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息来通信，而无需专用连接来链接它们。

消息队列,是一端写入数据到队列中,一端从队列中获取消息,并且通信通过RPC调用,可以传递java对象,方便操作

## 2.消息队列作用

(1) 使用场景

应用解耦:

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图

传统模式的缺点：

• 假如库存系统无法访问，则订单减库存将失败，从而导致订单失败

• 订单系统与库存系统耦合高

如何解决以上问题呢？引入应用消息队列后的方案，如下图

• 订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

• 库存系统：订阅下单的消息，获取下单信息，库存系统根据下单信息，进行库存操作

• 假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

## 流量削锋

流量削锋是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

• 用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面

• 秒杀业务根据消息队列中的请求信息，再做后续处理

• 通过消息队列缓解服务器压力.

## 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种: 1.串行的方式；2.并行方式

（1）串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端

（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间

假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

如何解决这个问题呢？

引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：

按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。

说明：用户点击注册，先将注册信息存入数据库，然后将用户的信息写入消息队列，然后返回给用户注册成功，发送邮件，发送短信从消息队列中取出用户注册信息进行后续操作，上述解决方案可用原因是，发送邮件和发送短信这两个步骤都是不需要立即执行反馈给用户的操作，因此可以借用消息队列进行信息的存储，避免了从数据库中的多次读写。

## 缺点

增加了复杂度

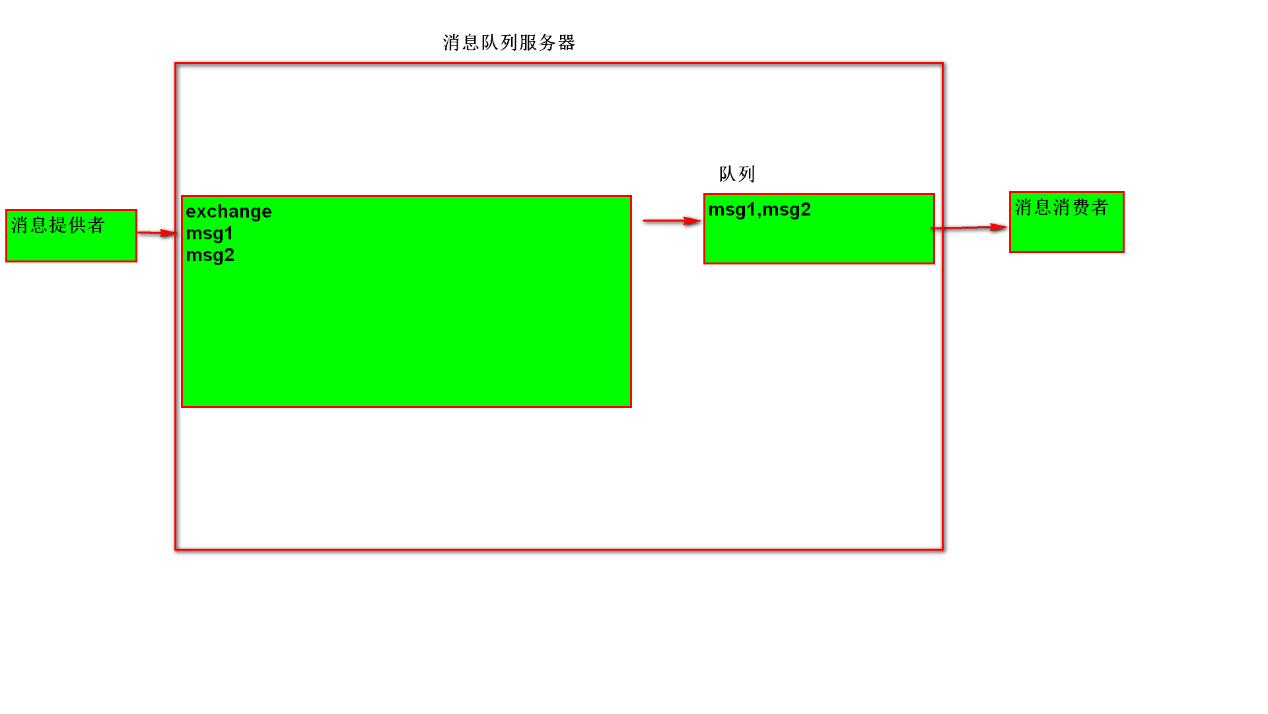
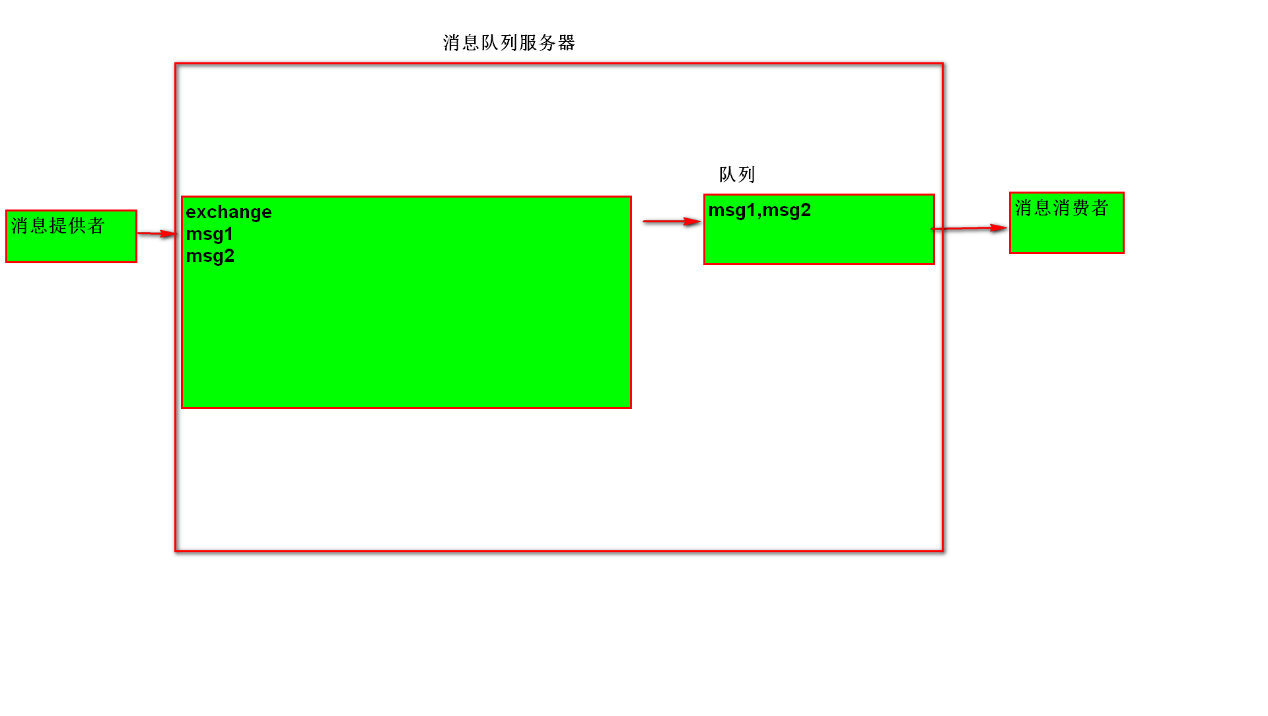
可靠性降低

## 消息队列模式

## 简单模式

生产者提供消息，消费者消费消息，1对1。

### 需求



C:\Users\Administrator\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\4208229D.tmp

P:消息的生产者

红色部分:代表队列

C:代表消费者  从队列中获取消息后执行.

工作原理:

      当客户端(生产者)将消息写入消息队列中时,消息队列中信息的数量加1.

      消费者**实时监听**消息队列,当消息队列中有消息时,则获取消息,之后执行业务逻辑.同时消息队列的数量减一

### 创建maven project

### 导入jar包

        <!-- 消息队列 -->

     <dependency>

        <groupId>com.rabbitmq</groupId>

        <artifactId>amqp-client</artifactId>

        <version>3.5.1</version>

     </dependency>

### 定义生产者

发送数据的过程与jdbc连接相似，共5步

public class Test\_1\_simple\_provider {

  @Test

  public void provider() throws Exception

  {

     //1,建立连接

     ConnectionFactory factory=new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection=factory.newConnection();

     //2,建立通道

     Channel channel=connection.createChannel();

     //3,定义队列

     //durable true 持久化,重启服务器后，数据还有

      //exclusive true,只能通过当前连接消费 false

     //autoDelete true 队列中消息处理完后，自动删除队列

     //arguments 参数

     channel.queueDeclare("orderQueue", true, false, false, null);

     //4,发送消息，routingKey必须与queue一致

     String msg="msg1";

     channel.basicPublish("", "orderQueue", null, msg.getBytes());

     //5,关闭

     channel.close();

     connection.close();

     System.out.println("发送数据成功");

  }

}

发消息，在后台overview,queue,查看队列，消息数。Default exchange

发送两个消息

Test\_1\_simple\_p2\_connection.java

//5,关闭

     //channel.close();

     //connection.close();

     System.out.println("发送数据成功");

     while(true){}

加while()查看channel,连接

去掉while，客户端就断开了，后台不显示channel,连接

### 定义消费者

接收数据的过程与jdbc查询相似，共5步

public class Test\_1\_simple\_consumer {

  @Test

  public void consumer() throws Exception

  {

     //1,建立连接

     ConnectionFactory factory=new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection=factory.newConnection();

     //2,建立通道

     Channel channel=connection.createChannel();

     //3,定义队列

     //durable true 持久化,重启服务器后，数据还有

      //exclusive true,只能通过当前连接消费 false

     //autoDelete true 队列中消息处理完后，自动删除队列

     //arguments 参数

     channel.queueDeclare("orderQueue", true, false, false, null);

     //4,创建消费者

     QueueingConsumer consumer=new QueueingConsumer(channel);

     //autoAck:自动回复消息

     channel.basicConsume("orderQueue", true, consumer);

     //5,取消息

     while(true)

     {

        Delivery delivery=consumer.nextDelivery();

        byte[] data=delivery.getBody();

        String mString=new String(data);

        System.out.println("消费者取到："+mString);

     }

  }

}

查看监控中消息被处理了

工作中，消息提供者和消息消费者运行在不同服务器上

Delivery的处理有点像resultSet和迭代器

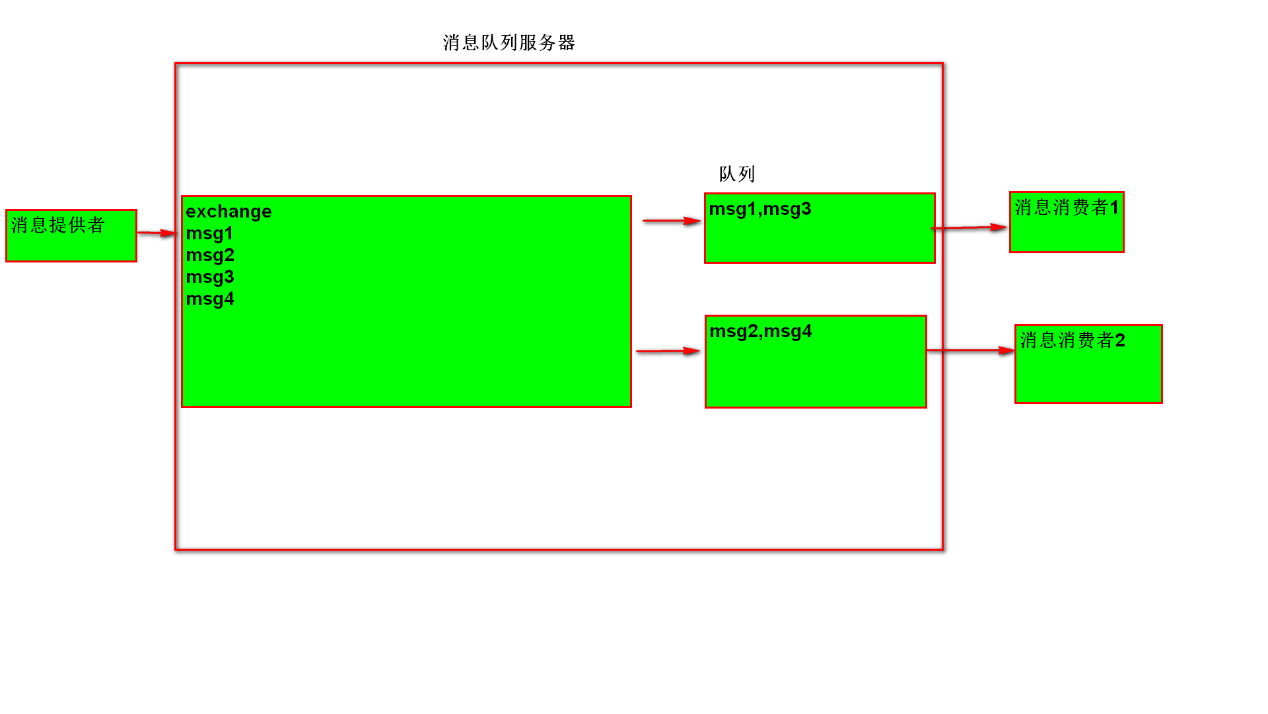
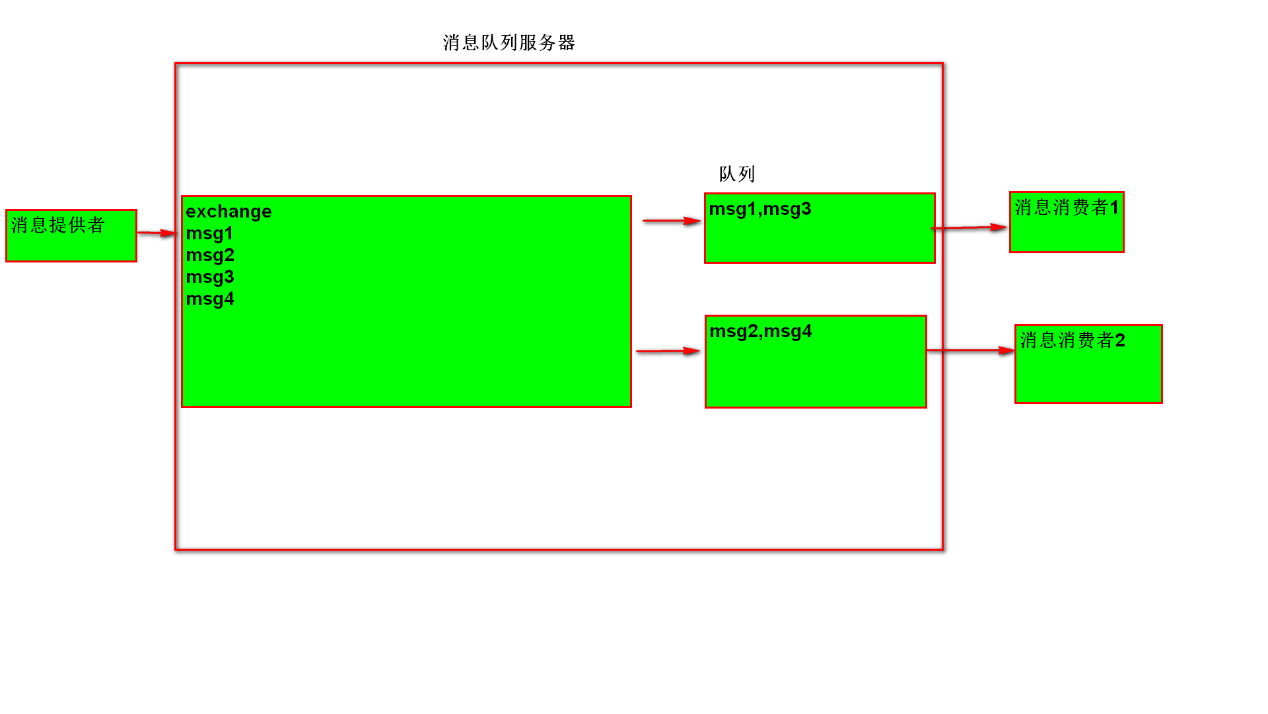
### 存在的问题

消费者处理态少，队列中消息积累很多

## 工作模式

生产者将消息放入消息队列，多个消费者随机进行消息的处理，每个消费者得到的消息都不一样，只是队列的一部分。

### 需求





说明:

      由一个生产者负责消息写入队列,但是如果有一个消费者负责消费,可能会造成消息的积压.所以准备多个消费者共同消费一个队列中的消息.

### 定义消费者1

拷贝简单模式的消费者代码，修改一下。

public class Test\_2\_work\_consumer1 {

  @Test

  public void consumer() throws Exception

  {

     //1,建立连接

     ConnectionFactory factory=new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection=factory.newConnection();

     //2,建立通道

     Channel channel=connection.createChannel();

     //3,定义队列

     //durable true 持久化,重启服务器后，数据还有

      //exclusive true,只能通过当前连接消费 false

     //autoDelete true 队列中消息处理完后，自动删除队列

     //arguments 参数

     channel.queueDeclare("orderQueue", true, false, false, null);

     //4,创建消费者

     QueueingConsumer consumer=new QueueingConsumer(channel);

     //autoAck:自动回复消息

     channel.basicConsume("orderQueue", true, consumer);

     //5,取消息

     System.out.println("消费者1启动");

     while(true)

     {

        Delivery delivery=consumer.nextDelivery();

        byte[] data=delivery.getBody();

        String mString=new String(data);

        System.out.println("消费者1取到："+mString);

     }

  }

}

拷贝消费者2

public class Test\_2\_work\_consumer2 {

  @Test

  public void consumer() throws Exception

  {

     //1,建立连接

     ConnectionFactory factory=new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection=factory.newConnection();

     //2,建立通道

     Channel channel=connection.createChannel();

     //3,定义队列

     //durable true 持久化,重启服务器后，数据还有

      //exclusive true,只能通过当前连接消费 false

     //autoDelete true 队列中消息处理完后，自动删除队列

     //arguments 参数

     channel.queueDeclare("orderQueue", true, false, false, null);

     //4,创建消费者

     QueueingConsumer consumer=new QueueingConsumer(channel);

     //autoAck:自动回复消息

     channel.basicConsume("orderQueue", true, consumer);

     //5,取消息

     System.out.println("消费者2启动");

     while(true)

     {

        Delivery delivery=consumer.nextDelivery();

        byte[] data=delivery.getBody();

        String mString=new String(data);

        System.out.println("消费者2取到："+mString);

     }

  }

}

### 定义生产者

拷贝简单模式代码加 while()  循环发送信息测试

public class Test\_2\_work\_provider {

  @Test

  public void provider() throws Exception {

     // 1,建立连接

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     // 2,建立通道

     Channel channel = connection.createChannel();

     // 3,定义队列

     // durable true 持久化,重启服务器后，数据还有

     // exclusive true,只能通过当前连接消费 false

     // autoDelete true 队列中消息处理完后，自动删除队列

     // arguments 参数

     channel.queueDeclare("orderQueue", true, false, false, null);

     // 4,发送消息

     for (int i = 3; i < 10; i++) {

        String msg = "msg" + i;

        channel.basicPublish("", "orderQueue", null, msg.getBytes());

     }

     // 5,关闭

     channel.close();

     connection.close();

     System.out.println("发送数据成功");

  }

}

运行三次，后台显示有有三个消息

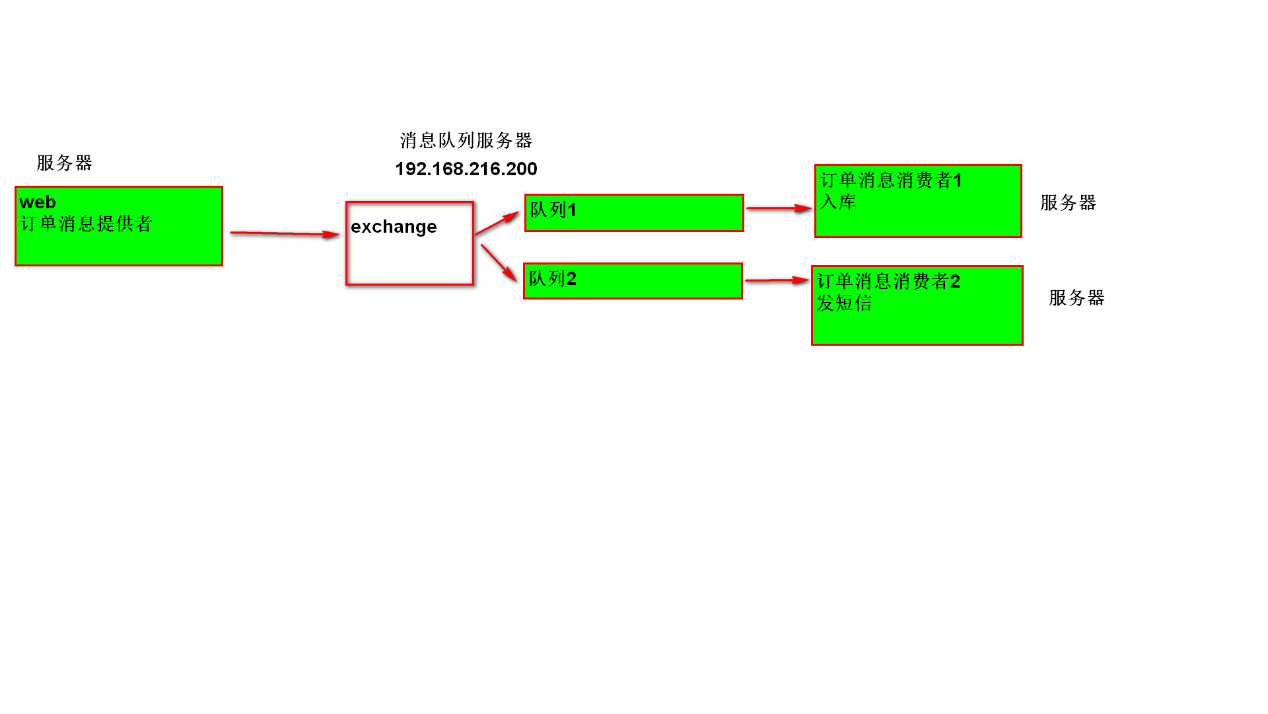
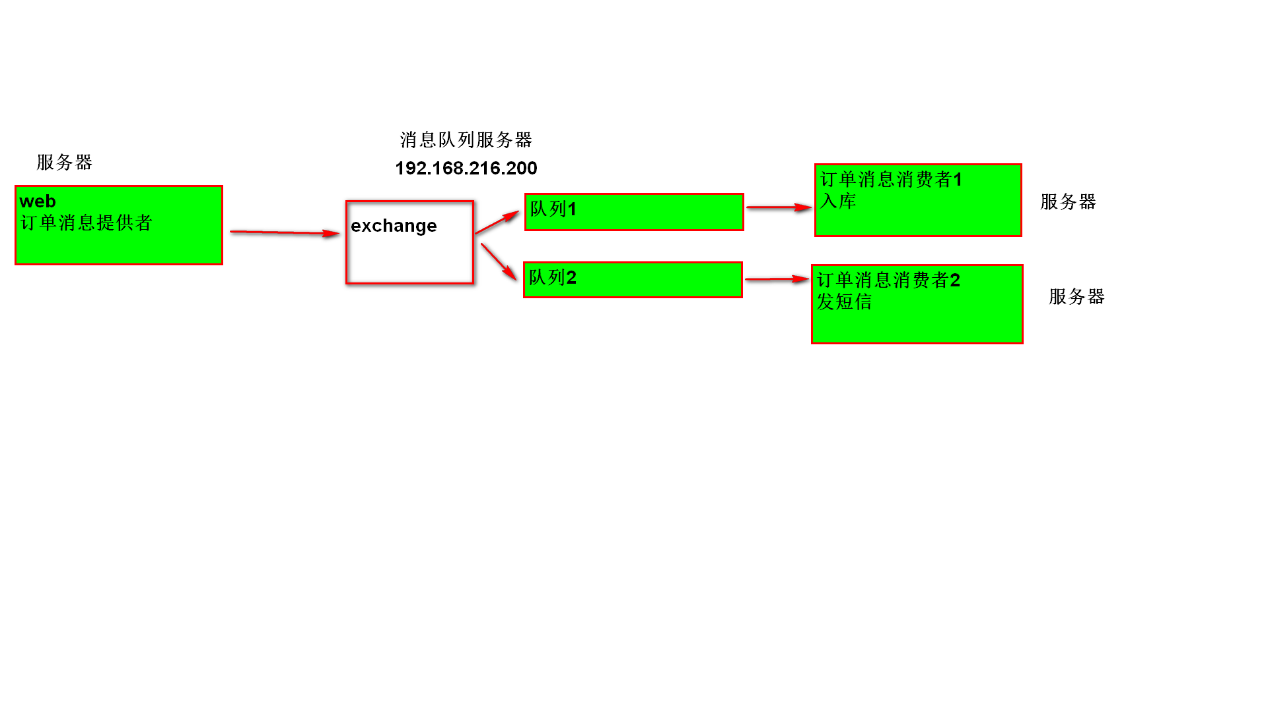
先启动消费者1,再启消费者2，消息全被第一个消费者处理了。

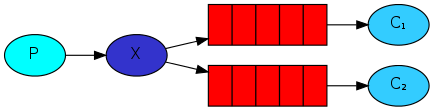
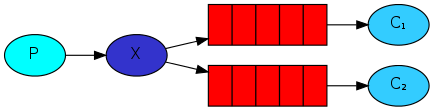
两个消费都启动后，再生产2个消息，分别被处理

## 发布订阅模式

### 需求

生产者将消息放进消息队列，所有消费者都会得到消息队列中的消息





X: 代表交换机  如果有消息则在发布订阅模式中,将消息发送到连接交换机的队列中

特点:如果生产者发送消息,那么订阅的全部消费者都会执行消息.

后期添加的消费者得不到以前的消息。需要先启动消费者

### 生产者

public class Test\_3\_publish\_p {

  // 定义生产者

  @Test

  public void provider() throws IOException {

     System.out.println("开始发布消息");

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     // 定义通道

     Channel channel = connection.createChannel();

     // 定义交换机名称

     String exchange\_name = "E1";

     // fanout是定义发布订阅模式  direct是 路由模式 topic是主题模式

     channel.exchangeDeclare(exchange\_name, "fanout");

     String msg = "order1" ;

     channel.basicPublish(exchange\_name, "", null, msg.getBytes());

     channel.close();

     connection.close();

  }

}

### 消费者1

public class Test\_3\_publish\_c1 {

  @Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E1";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "fanout");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者1");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"入库");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

先启生产者，再启消费者收不到前面的消息

后台查看交换机。有时看不到信息

一个消息被多个订阅者处理

### 消费者2

@Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E1";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "fanout");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者2");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"给用户发短信");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

### 案例

案例：微信公众号，订阅不同公众号，收到的消息不一样。

学生报名后，班主任要处理消息，项目经理也要处理消息

### 总结

订阅与工作模式区别是，订阅模式每个消费者得到的消息是一样的，工作模式每个消费者得到消息是不一样的。

解决高并发用工作模式

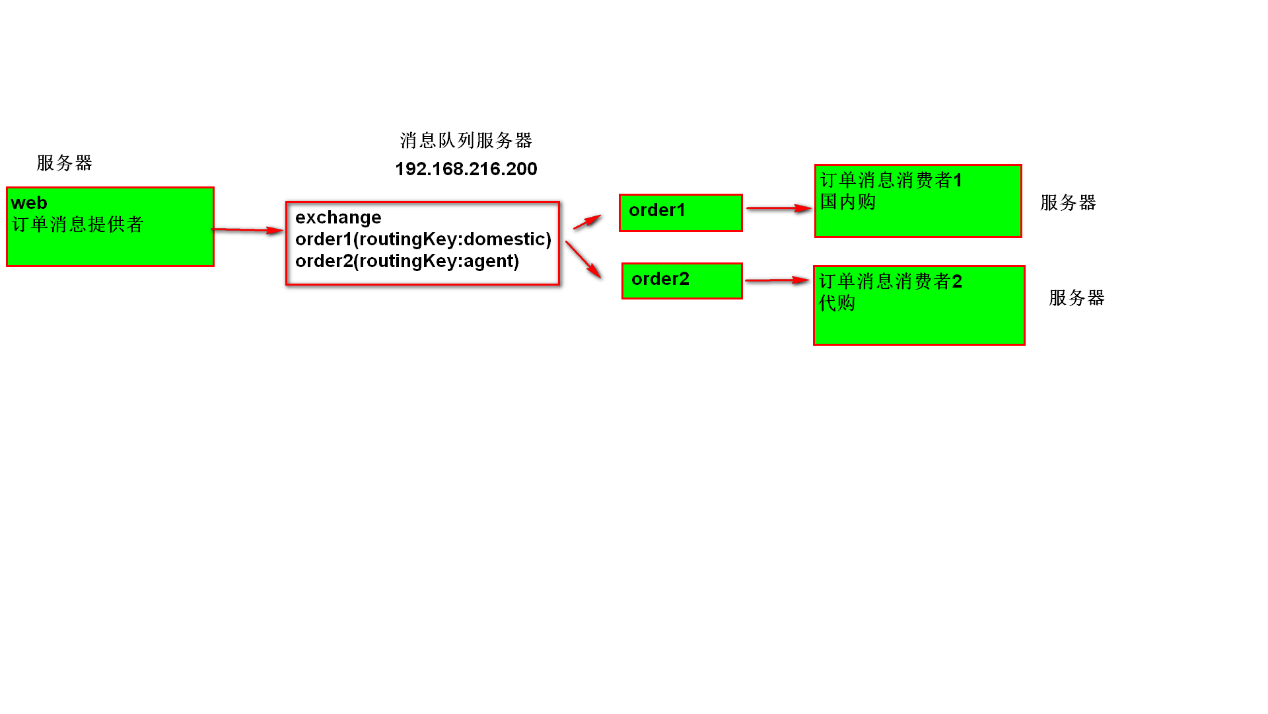
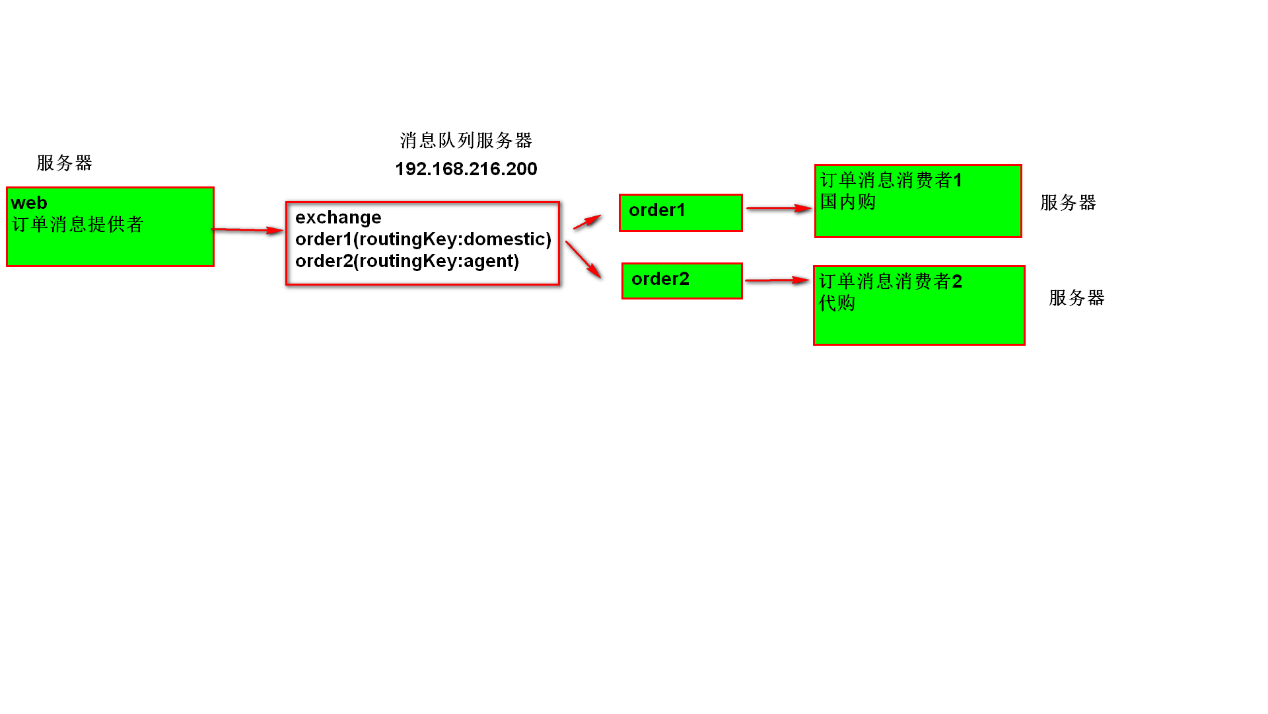
同一个消息，多个模块要处理用订阅

订阅像广播

## 路由模式

生产者将消息放进消息队列，消费者消费指定的消息，其他的不消费。

### 需求





说明:路由模式是发布订阅模式的升级,通过定义不同的路由key使得程序将消息发送到不同的队列中.

### 定义生产者

public class Test\_4\_routing\_provider {

  // 定义生产者

  @Test

  public void provider() throws IOException {

     System.out.println("开始发布消息");

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     // 定义通道

     Channel channel = connection.createChannel();

     // 定义交换机名称

     String exchange\_name = "E2";

     // fanout是定义发布订阅模式  direct是 路由模式 topic是主题模式

     channel.exchangeDeclare(exchange\_name, "direct");

     String msg = "order6" ;

     //channel.basicPublish(exchange\_name, "domestic", null, msg.getBytes());

     channel.basicPublish(exchange\_name, "agent", null, msg.getBytes());

     channel.close();

     connection.close();

  }

}

### 消费者1

public class Test\_4\_routing\_consumer1 {

  @Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E2";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "direct");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "domestic");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者1 国内库");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"入国内库");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

### 消费者2

拷贝消费1，修改routingkey

public class Test\_4\_routing\_consumer2 {

  @Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E2";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "direct");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "agent");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者2代购库");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"入代购库");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

先启动提供者发消息，再启动消费者，收不到以前的消息

启动消费者后，再用提供者发布消息，消费者能收到消息。

总结:消费者的队列只放有某个 routingkey的消息

相当于实现了select routingkey=’agent’功能.

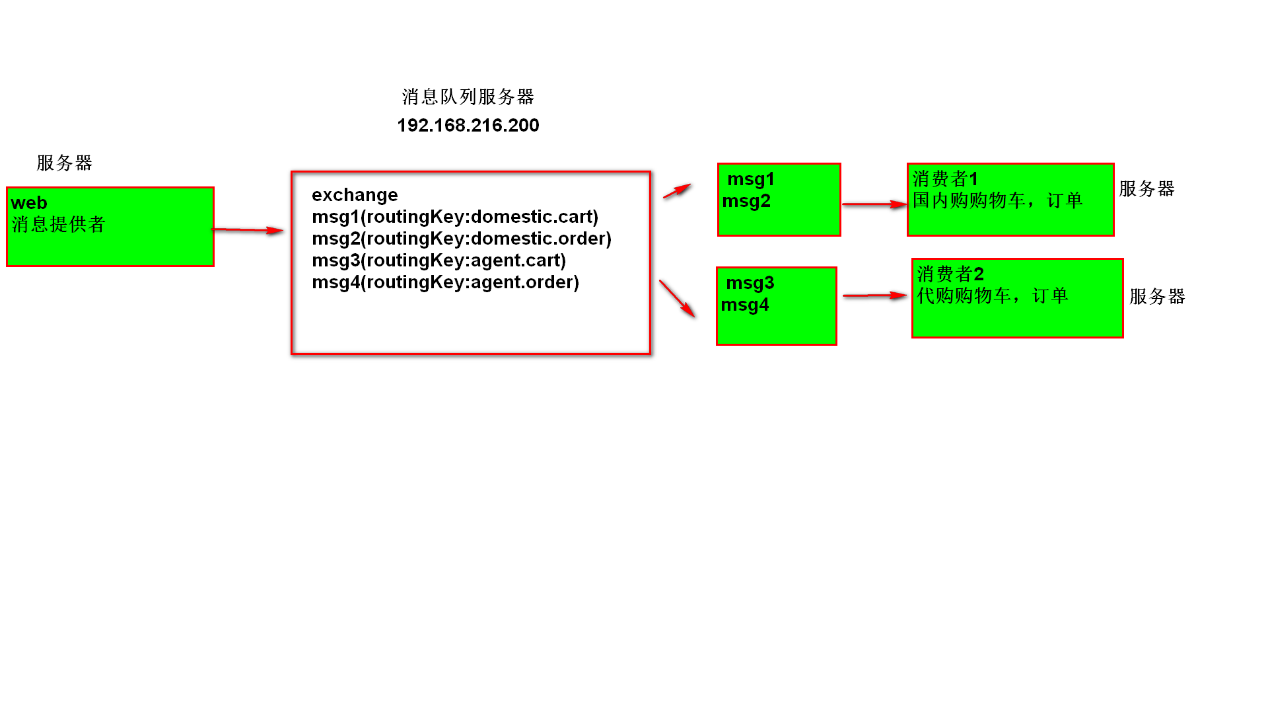
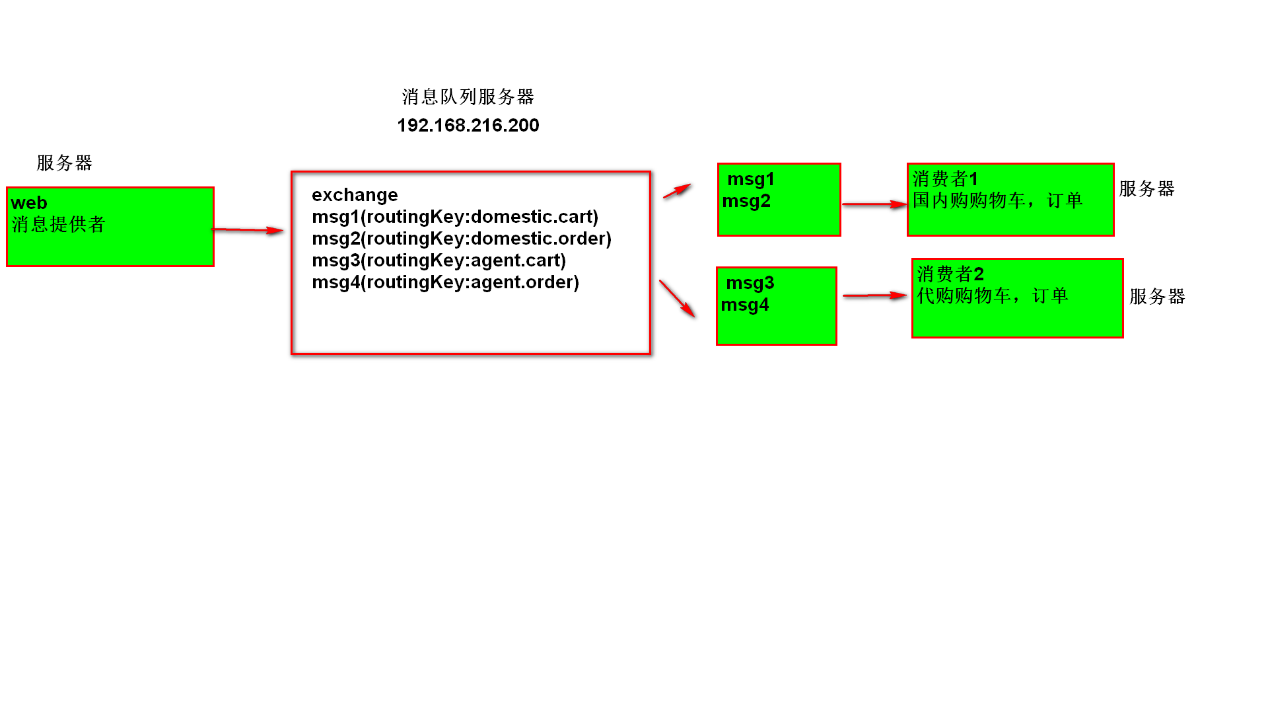
订阅模式：每个消费者收到的消息都一样

路由模式：每个消费者收到的消息不一样

## 主题模式

### 需求

将队列下的消息再进行细分





       说明:

              可以通过路由key将消息发送到一类相同的key中 使用通配符实现

       符号说明:

              #号:表示任意字符(任意个.)

              \*号:任意单个字符或者词组(单个.)

### 生产者

public class Test\_5\_topic\_provider {

  // 定义生产者

  @Test

  public void provider() throws IOException {

     System.out.println("开始发布消息");

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     // 定义通道

     Channel channel = connection.createChannel();

     // 定义交换机名称

     String exchange\_name = "E3";

     // fanout是定义发布订阅模式  direct是 路由模式 topic是主题模式

     channel.exchangeDeclare(exchange\_name, "topic");

     String msg = "msg10" ;

     //channel.basicPublish(exchange\_name, "domestic.cart", null, msg.getBytes());

     //channel.basicPublish(exchange\_name, "domestic.order", null, msg.getBytes());

     //channel.basicPublish(exchange\_name, "agent.cart", null, msg.getBytes());

     channel.basicPublish(exchange\_name, "agent.order", null, msg.getBytes());

     channel.close();

     connection.close();

  }

}

### 消费者1

public class Test\_5\_topic\_consumer1 {

  @Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E3";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "topic");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "domestic.\*");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者1 国内库");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"入国内库");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

### 消费者2

public class Test\_5\_topic\_consumer2 {

  @Test

  public void consumer1() throws Exception{

     ConnectionFactory factory = new ConnectionFactory();

     factory.setHost("192.168.216.202");

     factory.setPort(5672);

     factory.setUsername("jtadmin");

     factory.setPassword("jtadmin");

     factory.setVirtualHost("/jt");

     Connection connection = factory.newConnection();

     Channel channel = connection.createChannel();

     String exchange\_name = "E3";

     //定义交换机模式

     channel.exchangeDeclare(exchange\_name, "topic");

     String queue\_name = UUID.randomUUID().toString();

     System.out.println("队列名称"+queue\_name);

     //定义队列

     channel.queueDeclare(queue\_name, true, false, false, null);

     //将队列和交换机绑定   key:表示接收数据标识

     channel.queueBind(queue\_name, exchange\_name, "agent.\*");

     //定义消费数量

     channel.basicQos(1);

     //定义消费者

     QueueingConsumer consumer = new QueueingConsumer(channel);

     //将消费者和队列绑定,并且需要手动返回

     channel.basicConsume(queue\_name, false, consumer);

     System.out.println("消费者2代购库");

     while(true){

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

        String msg = new String(delivery.getBody());

        System.out.println(msg+"入代购库");

        //false表示一个一个返回

        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

     }

  }

}

# Elastic Search

# NAS-FastDFS

# NodeJS

# Vue.js

# Spring Cloud 改造