**一：Redis**

1. 他是一个开源免费的，遵循BSD协议，是一个高性能的key-value的非关系型数据库
2. redis采用多路复用机制：即多个网络socket复用一个io线程，实际是****单个线程通过记录跟踪每一个Sock(I/O流)的状态来同时管理多个I/O流****.

### 3）Redis特点：

Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

Redis不仅仅支持简单的key-value类型的数据，同时还提供String，list，set，zset，hash等数据结构的存储。

Redis支持数据的备份，即master-slave模式的数据备份。

性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis还支持 publish/subscribe, 通知, 设置key有效期等等特性。

### 4)Redis应用：

token生成、session共享、分布式锁、自增id、验证码等。

### 5)****比较重要的3个可执行文件：****

redis-server：Redis服务器程序

redis-cli：Redis客户端程序，它是一个命令行操作工具。也可以使用telnet根据其纯文本协议操作。

redis-benchmark：Redis性能测试工具，测试Redis在你的系统及配置下的读写性能。

## **redis的过期策略以及内存淘汰机制**

redis采用的是****定期删除+惰性删除策略****。

****为什么不用定时删除策略?****  
定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略.  
****定期删除+惰性删除是如何工作的呢?****  
定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。  
于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。  
****采用定期删除+惰性删除就没其他问题了么?****  
不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用****内存淘汰机制****。  
在redis.conf中有一行配置

# maxmemory-policy allkeys-lru

该配置就是配内存淘汰策略的(什么，你没配过？好好反省一下自己)  
1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。****应该没人用吧。****  
2）allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。****推荐使用。****  
3）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。****应该也没人用吧，你不删最少使用Key,去随机删。****  
4）volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。****这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐****  
5）volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。****依然不推荐****  
6）volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。****不推荐****  
ps：如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

## **redis和数据库双写一致性问题**

****回答****:首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

## **如何应对缓存穿透和缓存雪崩问题**

****缓存穿透：****即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

****解决方案****:  
(一)利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试  
(二)采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做****缓存预热****(项目启动前，先加载缓存)操作。  
(三)提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

****缓存雪崩****，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

****解决方案****:  
(一)给缓存的失效时间，加上一个随机值，避免集体失效。  
(二)使用互斥锁，但是该方案吞吐量明显下降了。  
(三)双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点

I 从缓存A读数据库，有则直接返回

II A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程。

III 更新线程同时更新缓存A和缓存B。

## **如何解决redis的并发竞争key问题**

****回答:****如下所示  
(1)如果对这个key操作，****不要求顺序****  
这种情况下，准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可，比较简单。  
(2)如果对这个key操作，****要求顺序****  
假设有一个key1,系统A需要将key1设置为valueA,系统B需要将key1设置为valueB,系统C需要将key1设置为valueC.  
期望按照key1的value值按照 valueA-->valueB-->valueC的顺序变化。这种时候我们在数据写入数据库的时候，需要保存一个时间戳。假设时间戳如下

系统A key 1 {valueA  3:00}

系统B key 1 {valueB  3:05}

系统C key 1 {valueC  3:10}

那么，假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。

Reids持久化

Redis持久化,就是将内存数据保存到硬盘，Redis 持久化存储分为 AOF 与 RDB 两种模式，默认开启rdb。

RDB 是在某个时间点将数据写入一个临时文件dump.rdb，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复，采用二进制文件形式进行存储。

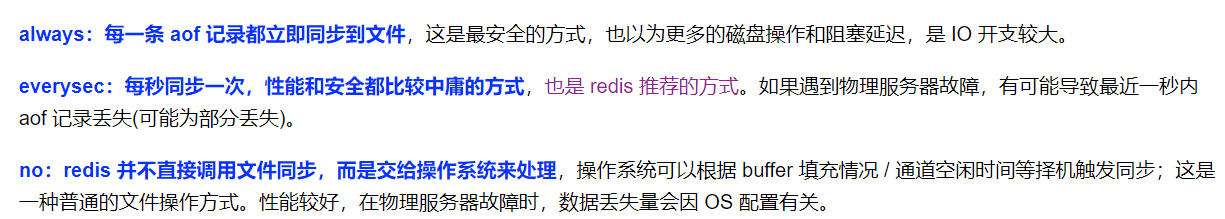
优点：****使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能****  
缺点：****RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候****

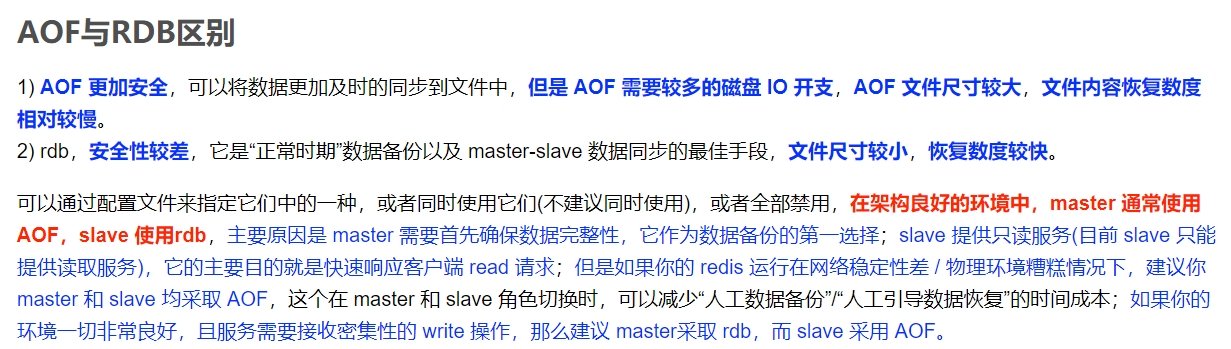
## **AOF持久化**

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在 append 操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当 server 需要数据恢复时，可以直接 replay 此日志文件，即可还原所有的操作过程。AOF 相对可靠，它和 mysql 中 bin.log、apache.log、zookeeper 中 txn-log 简直异曲同工。AOF 文件内容是字符串，非常容易阅读和解析。

优点：****可以保持更高的数据完整性****，如果设置追加 file 的时间是 1s，如果 redis 发生故障，最多会丢失 1s 的数据；且如果日志写入不完整支持 redis-check-aof 来进行日志修复；aof文件没被 rewrite 之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）。  
缺点：****AOF 文件比 RDB 文件大，且恢复速度慢****。



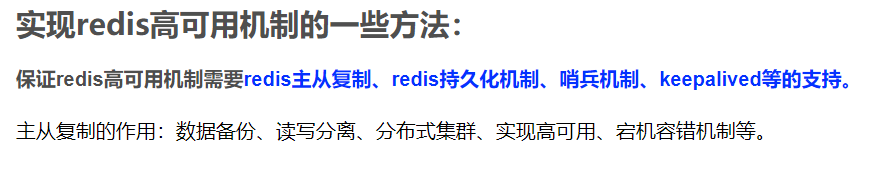


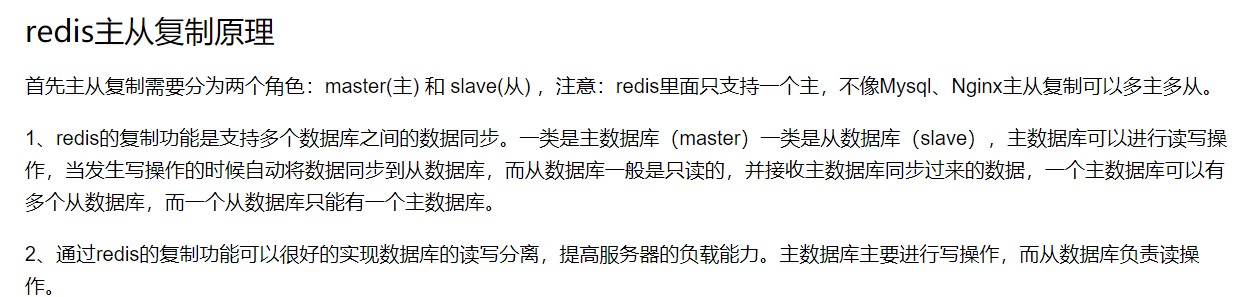


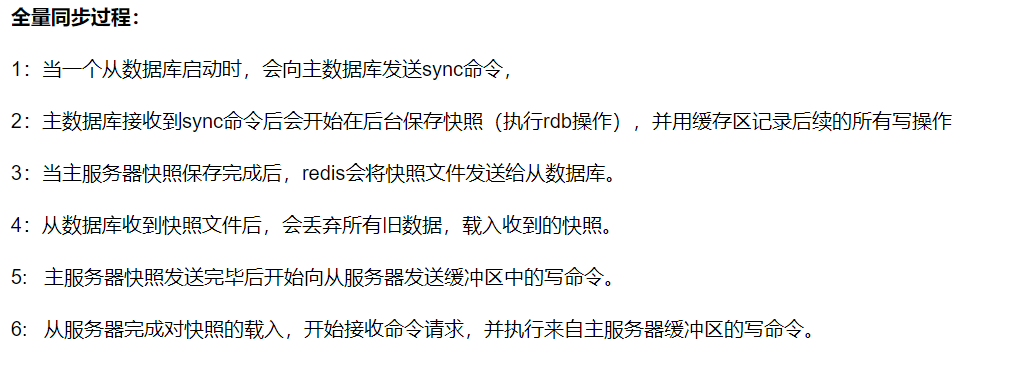
****redis宕机后，redis的值会失效吗？****

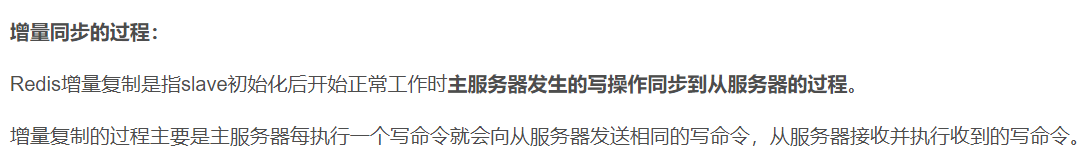
答：不会，redis默认开启rdb存储。注明：使用rdb持久化，只有在特定时间达到特定的修改数量，redis的值才会被持久化到dump.rdb中，但断开连接后，会自动更新【无则生成】dump.rdb，实现自动备份。需要注意的是，如果直接杀死进程或者直接关机/重启服务器，数据有可能会丢失，这种情况下不会自动备份dump.rdb。

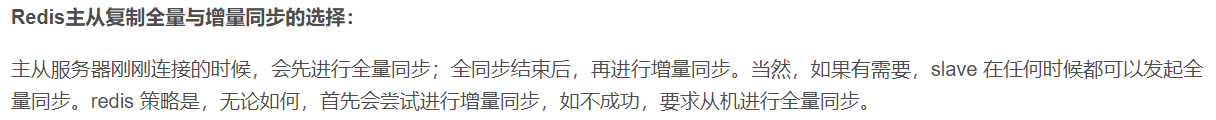
****Redis实现高可用的主从复制和哨兵机制****















哨兵原理

Sentinel 使用的算法核心是 Raft 算法，主要用途就是用于分布式系统，系统容错，以及Leader选举，每个Sentinel都需要定期的执行以下任务：

每个 Sentinel 会自动发现其他 Sentinel 和从服务器，它以每秒钟一次的频率向它所知的主服务器、从服务器以及其他 Sentinel 实例发送一个 PING 命令。

如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 那么这个实例会被 Sentinel 标记为主观下线。 有效回复可以是： +PONG 、 -LOADING 或者 -MASTERDOWN 。

如果一个主服务器被标记为主观下线， 那么正在监视这个主服务器的所有Sentinel要以每秒一次的频率确认主服务器的确进入了主观下线状态。

如果一个主服务器被标记为主观下线， 并且有足够数量的Sentinel（至少要达到配置文件指定的数量）在指定的时间范围内同意这一判断， 那么这个主服务器被标记为客观下线。

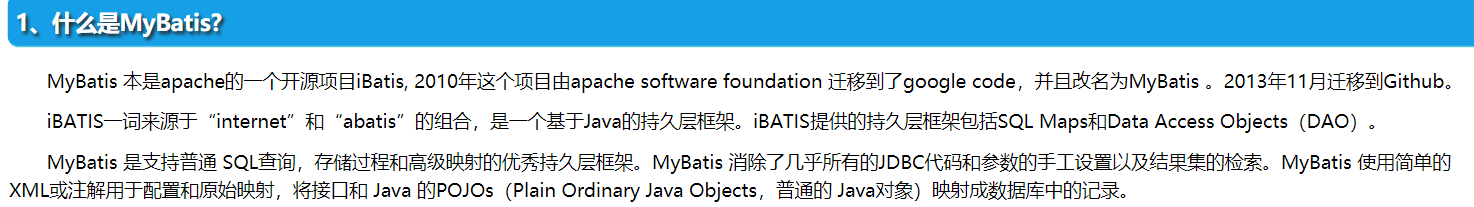
在一般情况下， 每个Sentinel会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令。 当一个主服务器被Sentinel标记为客观下线时，Sentinel向下线主服务器的所有从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。

当没有足够数量的Sentinel同意主服务器已经下线， 主服务器的客观下线状态就会被移除。 当主服务器重新向Sentinel的 PING 命令返回有效回复时， 主服务器的主管下线状态就会被移除。





二:MyBatis



主要组成:configuration,

Sqlsessionfactory sqlsession,mapperhandler,statementhandler,resultsethandler

主要使用映射接口，实体，对应的注解，以及对应的mapper文件来进行映射

三:RabbitMQ

https://blog.csdn.net/bestmy/article/details/84304964

# **一．AMQP协议**

## **1. AMQP协议简介**

AMQP，即Advanced Message Queuing Protocol,一个提供统一消息服务的应用层标准高级消息队列协议,是应用层协议的一个开放标准,为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。Erlang中的实现有 RabbitMQ等。

## **2. 功能范围**

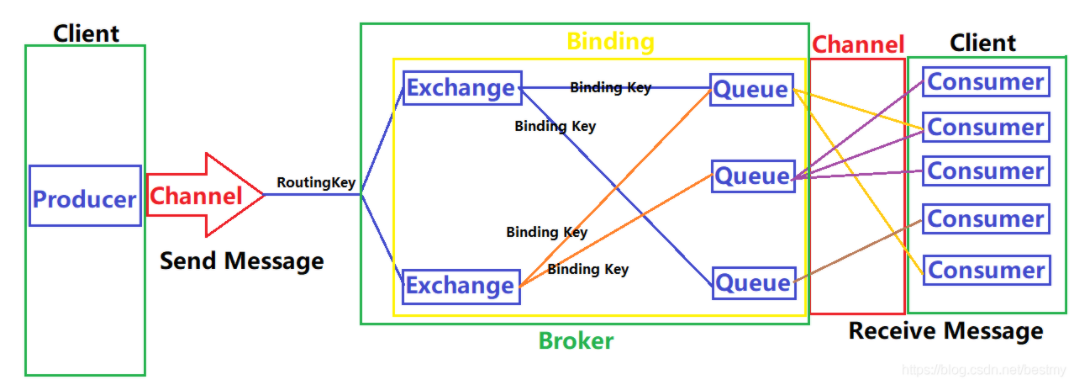
存储转发（多个消息发送者，单个消息接收者）。  
分布式事务（多个消息发送者，多个消息接收者）。  
发布订阅（多个消息发送者，多个消息接收者）。  
基于内容的路由（多个消息发送者，多个消息接收者）。  
文件传输队列（多个消息发送者，多个消息接收者）。  
点对点连接（单个消息发送者，单个消息接收者）。

## **3. 技术术语**

AMQP模型（AMQP Model）：一个由关键实体和语义表示的逻辑框架，遵从AMQP规范的服务器必须提供这些实体和语义。为了实现本规范中定义的语义，客户端可以发送命令来控制AMQP服务器。  
连接（Connection）：一个网络连接，比如TCP/IP套接字连接。  
会话（Session）：端点之间的命名对话。在一个会话上下文中，保证“恰好传递一次”。  
信道（Channel）：多路复用连接中的一条独立的双向数据流通道。为会话提供物理传输介质。  
客户端（Client）：AMQP连接或者会话的发起者。AMQP是非对称的，客户端生产和消费消息，服务器存储和路由这些消息。  
服务器（Server）：接受客户端连接，实现AMQP消息队列和路由功能的进程。也称为“消息代理”。  
端点（Peer）：AMQP对话的任意一方。一个AMQP连接包括两个端点（一个是客户端，一个是服务器）。  
搭档（Partner）：当描述两个端点之间的交互过程时，使用术语“搭档”来表示“另一个”端点的简记法。比如我们定义端点A和端点B，当它们进行通信时，端点B是端点A的搭档，端点A是端点B的搭档。  
片段集（Assembly）：段的有序集合，形成一个逻辑工作单元。  
段（Segment）：帧的有序集合，形成片段集中一个完整子单元。  
帧（Frame）：AMQP传输的一个原子单元。一个帧是一个段中的任意分片。  
控制（Control）：单向指令，AMQP规范假设这些指令的传输是不可靠的。  
命令（Command）：需要确认的指令，AMQP规范规定这些指令的传输是可靠的。  
异常（Exception）：在执行一个或者多个命令时可能发生的错误状态。  
类（Class）：一批用来描述某种特定功能的AMQP命令或者控制。  
消息头（Header）：描述消息数据属性的一种特殊段。  
消息体（Body）：包含应用程序数据的一种特殊段。消息体段对于服务器来说完全透明——服务器不能查看或者修改消息体。  
消息内容（Content）：包含在消息体段中的的消息数据。  
交换器（Exchange）：服务器中的实体，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。  
交换器类型（Exchange Type）：基于不同路由语义的交换器类。  
消息队列（Message Queue）：一个命名实体，用来保存消息直到发送给消费者。  
绑定器（Binding）：消息队列和交换器之间的关联。  
绑定器关键字（Binding Key）：绑定的名称。一些交换器类型可能使用这个名称作为定义绑定器路由行为的模式。  
路由关键字（Routing Key）：一个消息头，交换器可以用这个消息头决定如何路由某条消息。  
持久存储（Durable）：一种服务器资源，当服务器重启时，保存的消息数据不会丢失。  
临时存储（Transient）：一种服务器资源，当服务器重启时，保存的消息数据会丢失。  
持久化（Persistent）：服务器将消息保存在可靠磁盘存储中，当服务器重启时，消息不会丢失。  
非持久化（Non-Persistent）：服务器将消息保存在内存中，当服务器重启时，消息可能丢失。  
消费者（Consumer）：一个从消息队列中请求消息的客户端应用程序。  
生产者（Producer）：一个向交换器发布消息的客户端应用程序。  
虚拟主机（Virtual Host）：一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。客户端应用程序在登录到服务器之后，可以选择一个虚拟主机。

## **1. RabbitMQ基础概念**

通常我们谈到消息队列服务, 会有三个概念： 发消息者、消息队列、收消息者。RabbitMQ 在这个基本概念之上, 多做了一层抽象, 在发消息者和队列之间, 加入了交换器 (Exchange)。这样发消息者和消息队列就没有直接联系，转而变成发消息者把消息发给交换器，交换器根据调度策略再把消息转发给消息队列。  
消息生产者并没有直接将消息发送给消息队列，而是通过建立与Exchange的Channel，将消息发送给Exchange。Exchange根据路由规则，将消息转发给指定的消息队列。消息队列储存消息，等待消费者取出消息。消费者通过建立与消息队列相连的Channel,从消息队列中获取消息。



1. Channel（信道）：多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内的虚拟连接，复用TCP连接的通道。  
   2.Producer（消息的生产者）：向消息队列发布消息的客户端应用程序。  
   3.Consumer（消息的消费者）：从消息队列取得消息的客户端应用程序。  
   4.Message（消息）：消息由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（消息优先权）、delivery-mode（是否持久性存储）等。  
   5.Routing Key（路由键）：消息头的一个属性，用于标记消息的路由规则，决定了交换机的转发路径。最大长度255 字节。  
   6.Queue（消息队列）：存储消息的一种数据结构，用来保存消息，直到消息发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将消息取走。需要注意，当多个消费者订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理，每一条消息只能被一个订阅者接收。  
   7.Exchange（交换器|路由器）：提供Producer到Queue之间的匹配，接收生产者发送的消息并将这些消息按照路由规则转发到消息队列。交换器用于转发消息，它不会存储消息 ，如果没有 Queue绑定到 Exchange 的话，它会直接丢弃掉 Producer 发送过来的消息。交换器有四种消息调度策略（下面会介绍），分别是fanout, direct, topic, headers。  
   8.Binding（绑定）：用于建立Exchange和Queue之间的关联。一个绑定就是基于Binding Key将Exchange和Queue连接起来的路由规则，所以可以将交换器理解成一个由Binding构成的路由表。  
   6.Binding Key（绑定键）：Exchange与Queue的绑定关系，用于匹配Routing Key。最大长度255 字节。  
   7.Broker：RabbitMQ Server，服务器实体。

## **2. Exchange消息调度策略**

调度策略是指Exchange在收到生产者发送的消息后依据什么规则把消息转发到一个或多个队列中保存。调度策略与三个因素相关：Exchange Type（Exchange的类型），Binding Key（Exchange和Queue的绑定关系），消息的标记信息（Routing Key和headers）。  
Exchange根据消息的Routing Key和Exchange绑定Queue的Binding Key分配消息。生产者在将消息发送给Exchange的时候，一般会指定一个Routing Key，来指定这个消息的路由规则，而这个Routing Key需要与Exchange Type及Binding Key联合使用才能最终生效。  
在Exchange Type与Binding Key固定的情况下（一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定Routing Key来决定消息流向哪里。

1. Fanout(广播模式)

交换器会把所有发送到该交换器的消息路由到所有与该交换器绑定的消息队列中。订阅模式  
与Binding Key和Routing Key无关，交换器将接受到的消息分发给有绑定关系的所有消息队列队列（不论Binding Key和Routing Key是什么）。类似于子网广播，子网内的每台主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。

1. direct(精确匹配)

精确匹配：当消息的Routing Key与 Exchange和Queue 之间的Binding Key完全匹配，如果匹配成功，将消息分发到该Queue。只有当Routing Key和Binding Key完全匹配的时候，消息队列才可以获取消息。Direct是Exchange的默认模式。  
RabbitMQ默认提供了一个Exchange，名字是空字符串，类型是Direct，绑定到所有的Queue（每一个Queue和这个无名Exchange之间的Binding Key是Queue的名字）。所以，有时候我们感觉不需要交换器也可以发送和接收消息，但是实际上是使用了RabbitMQ默认提供的Exchange。

1. topic(通配符模式)\*，#

按照正则表达式模糊匹配：用消息的Routing Key与 Exchange和Queue 之间的Binding Key进行模糊匹配，如果匹配成功，将消息分发到该Queue。  
Routing Key是一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词）。Binding Key与Routing Key一样也是句点号“. ”分隔的字符串。Binding Key中可以存在两种特殊字符“ \* ”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）。

1. headers（键值匹配）

Headers不依赖于Routing Key与Binding Key的匹配规则来转发消息，交换器的路由规则是通过消息头的Headers属性来进行匹配转发的，类似HTTP请求的Headers。  
在绑定Queue与Exchange时指定一组键值对，键值对的Hash结构中要求携带一个键“x-match”，这个键的Value可以是any或all，代表消息携带的Hash是需要全部匹配(all)，还是仅匹配一个键(any)。  
当消息发送到Exchange时，交换器会取到该消息的headers，对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。Headers交换机的优势是匹配的规则不被限定为字符串(String)，而是Object类型。

RPC(remote procedure Call)做同步使用/./远程过程调用

