# Redis高级特性

## 事务

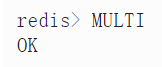
Redis 通过 [MULTI](http://redis.readthedocs.org/en/latest/transaction/multi.html#multi) 、 [DISCARD](http://redis.readthedocs.org/en/latest/transaction/discard.html#discard) 、 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 和 [WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 四个命令来实现事务功能。

事务提供了一种“将多个命令打包， 然后一次性、按顺序地执行”的机制， 并且事务在执行的期间不会主动中断 —— 服务器在执行完事务中的所有命令之后， 才会继续处理其他客户端的其他命令。

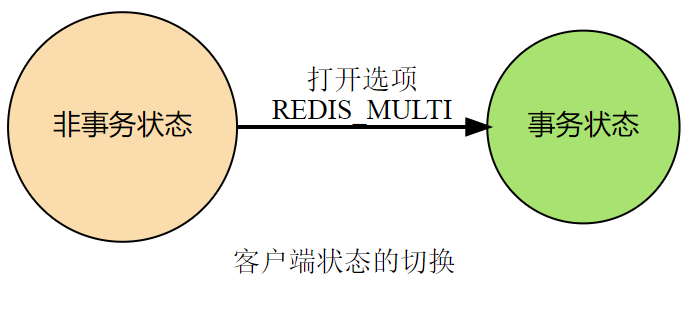
事务以multi开始，将多个命令入队，由exec执行。

### 事务执行过程

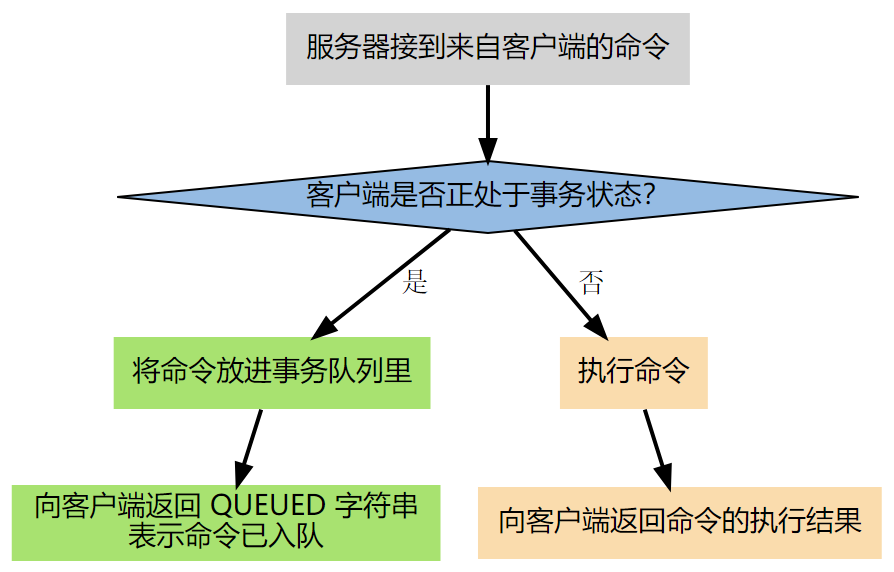
#### 开始事务



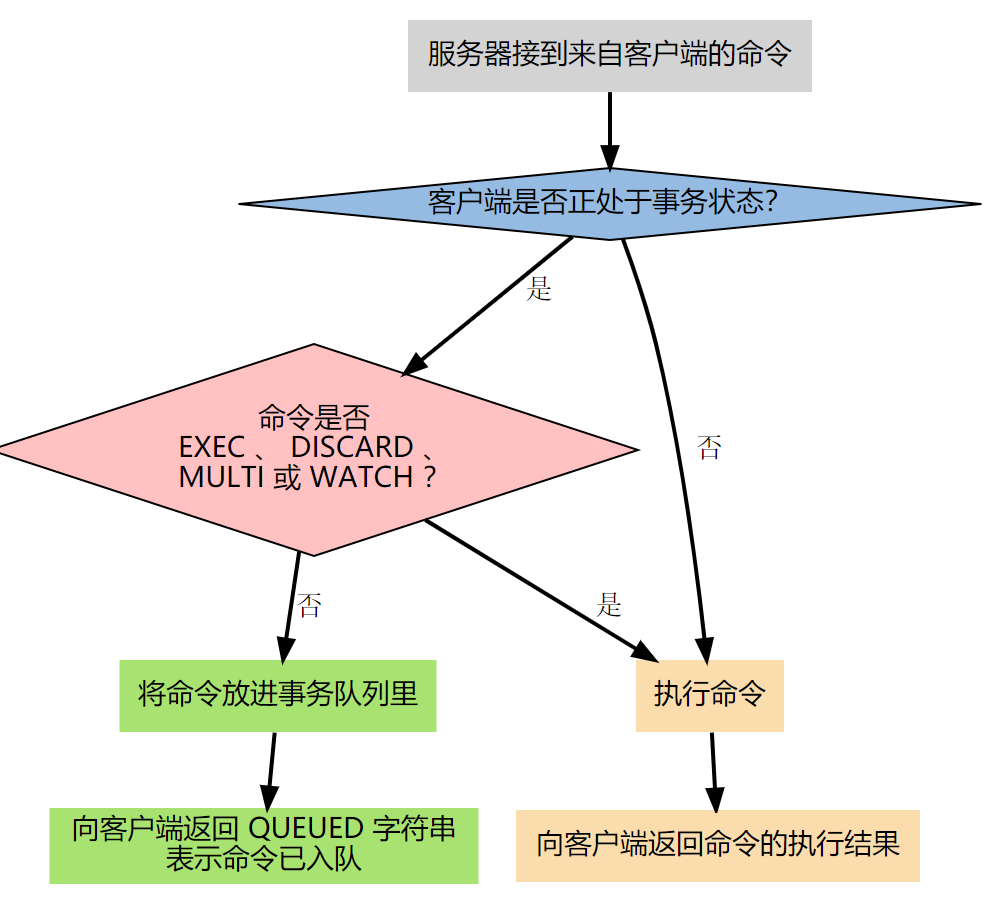
这个命令唯一做的就是， 将客户端的 REDIS\_MULTI 选项打开， 让客户端从非事务状态切换到事务状态。



#### 命令入队



#### 执行事务



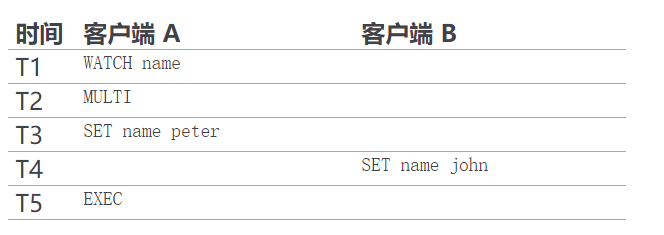
服务器根据客户端所保存的事务队列， 以先进先出（FIFO）的方式执行事务队列中的命令： 最先入队的命令最先执行， 而最后入队的命令最后执行。执行事务中的命令所得的结果会以 FIFO 的顺序保存到一个回复队列中。当事务队列里的所有命令被执行完之后， [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令会将回复队列作为自己的执行结果返回给客户端， 客户端从事务状态返回到非事务状态， 至此， 事务执行完毕。

[DISCARD](http://redis.readthedocs.org/en/latest/transaction/discard.html#discard) 命令用于取消一个事务， 它清空客户端的整个事务队列， 然后将客户端从事务状态调整回非事务状态， 最后返回字符串 OK 给客户端， 说明事务已被取消。

Redis 的事务是不可嵌套的， 当客户端已经处于事务状态， 而客户端又再向服务器发送 [MULTI](http://redis.readthedocs.org/en/latest/transaction/multi.html#multi) 时， 服务器只是简单地向客户端发送一个错误， 然后继续等待其他命令的入队。 [MULTI](http://redis.readthedocs.org/en/latest/transaction/multi.html#multi) 命令的发送不会造成整个事务失败， 也不会修改事务队列中已有的数据。

[WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 只能在客户端进入事务状态之前执行， 在事务状态下发送 [WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 命令会引发一个错误， 但它不会造成整个事务失败， 也不会修改事务队列中已有的数据（和前面处理 [MULTI](http://redis.readthedocs.org/en/latest/transaction/multi.html#multi) 的情况一样）。

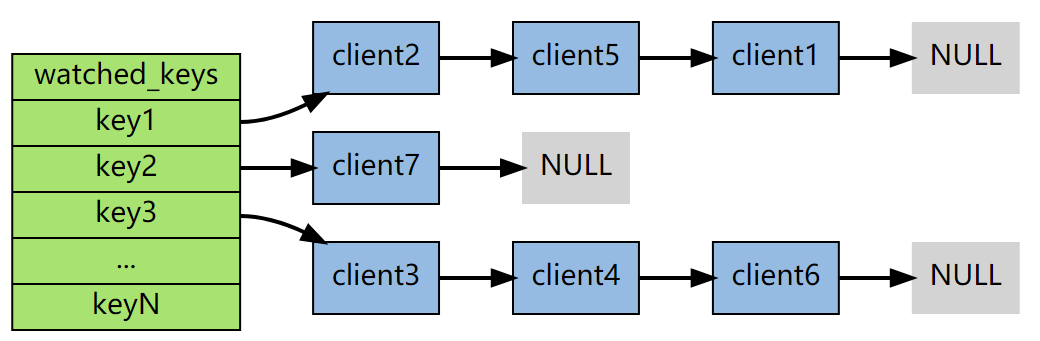
### 带 WATCH 的事务

[WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 命令用于在事务开始之前监视任意数量的键： 当调用 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令执行事务时， 如果任意一个被监视的键已经被其他客户端修改了， 那么整个事务不再执行， 直接返回失败。

在时间 T4 ，客户端 B 修改了 name 键的值， 当客户端 A 在 T5 执行 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 时，Redis 会发现 name 这个被监视的键已经被修改， 因此客户端 A 的事务不会被执行，而是直接返回失败。

#### WATCH 命令的实现

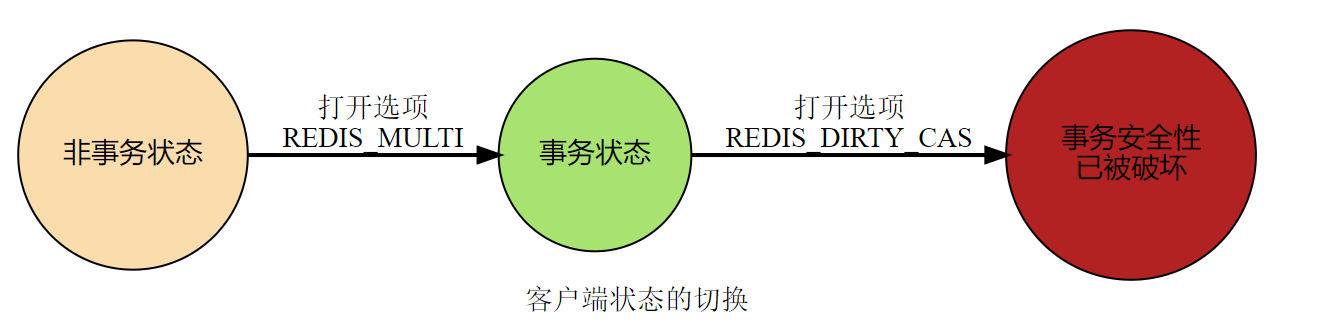
在每个代表数据库的 redis.h/redisDb 结构类型中， 都保存了一个 watched\_keys 字典， 字典的键是这个数据库被监视的键， 而字典的值则是一个链表， 链表中保存了所有监视这个键的客户端。



[WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 命令的作用， 就是将当前客户端和要监视的键在 watched\_keys 中进行关联。通过 watched\_keys 字典， 如果程序想检查某个键是否被监视， 那么它只要检查字典中是否存在这个键即可； 如果程序要获取监视某个键的所有客户端， 那么只要取出键的值（一个链表）， 然后对链表进行遍历即可。

#### WATCH 的触发

在任何对数据库键空间（key space）进行修改的命令成功执行之后 （比如 [FLUSHDB](http://redis.readthedocs.org/en/latest/server/flushdb.html#flushdb) 、 [SET](http://redis.readthedocs.org/en/latest/string/set.html#set) 、 [DEL](http://redis.readthedocs.org/en/latest/key/del.html#del) 、 [LPUSH](http://redis.readthedocs.org/en/latest/list/lpush.html#lpush) 、 [SADD](http://redis.readthedocs.org/en/latest/set/sadd.html#sadd) 、 [ZREM](http://redis.readthedocs.org/en/latest/sorted_set/zrem.html#zrem) ，诸如此类）， multi.c/touchWatchedKey 函数都会被调用 —— 它检查数据库的 watched\_keys 字典， 看是否有客户端在监视已经被命令修改的键， 如果有的话， 程序将所有监视这个/这些被修改键的客户端的 REDIS\_DIRTY\_CAS 选项打开：



当客户端发送 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令、触发事务执行时， 服务器会对客户端的状态进行检查：

* 如果客户端的 REDIS\_DIRTY\_CAS 选项已经被打开，那么说明被客户端监视的键至少有一个已经被修改了，事务的安全性已经被破坏。服务器会放弃执行这个事务，直接向客户端返回空回复，表示事务执行失败。
* 如果 REDIS\_DIRTY\_CAS 选项没有被打开，那么说明所有监视键都安全，服务器正式执行事务。

最后，当一个客户端结束它的事务时，无论事务是成功执行，还是失败， watched\_keys 字典中和这个客户端相关的资料都会被清除。

### 事务的 ACID 性质

#### 原子性（Atomicity）

单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。

如果一个事务队列中的所有命令都被成功地执行，那么称这个事务执行成功。

另一方面，如果 Redis 服务器进程在执行事务的过程中被停止 —— 比如接到 KILL 信号、宿主机器停机，等等，那么事务执行失败。当事务失败时，Redis 也不会进行任何的重试或者回滚动作。

#### 持久性（Durability）

因为事务不过是用队列包裹起了一组 Redis 命令，并没有提供任何额外的持久性功能，所以事务的持久性由 Redis 所使用的持久化模式决定：

* 在单纯的内存模式下，事务肯定是不持久的。
* 在 RDB 模式下，服务器可能在事务执行之后、RDB 文件更新之前的这段时间失败，所以 RDB 模式下的 Redis 事务也是不持久的。
* 在 AOF 的“总是 SYNC ”模式下，事务的每条命令在执行成功之后，都会立即调用 fsync 或 fdatasync 将事务数据写入到 AOF 文件。但是，这种保存是由后台线程进行的，主线程不会阻塞直到保存成功，所以从命令执行成功到数据保存到硬盘之间，还是有一段非常小的间隔，所以这种模式下的事务也是不持久的。

其他 AOF 模式也和“总是 SYNC ”模式类似，所以它们都是不持久的。

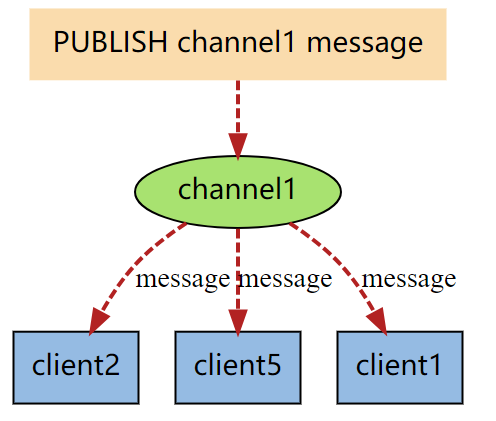
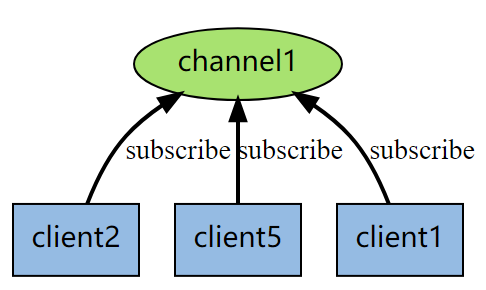
Redis 的事务保证了 ACID 中的一致性（C）和隔离性（I），但并不保证原子性（A）和持久性（D）

## 订阅与发布

Redis 通过 [PUBLISH](http://redis.readthedocs.org/en/latest/pub_sub/publish.html#publish) 、 [SUBSCRIBE](http://redis.readthedocs.org/en/latest/pub_sub/subscribe.html#subscribe) 等命令实现了订阅与发布模式， 这个功能提供两种信息机制， 分别是订阅/发布到频道和订阅/发布到模式。

### 频道的订阅与信息发送

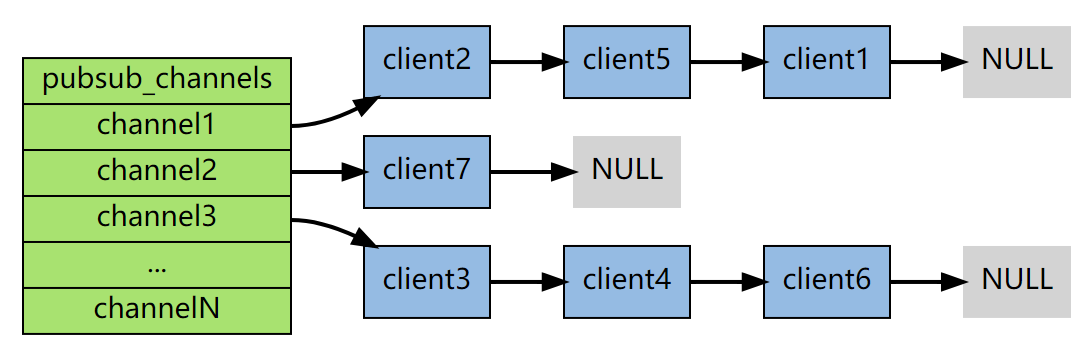
Redis 的 [SUBSCRIBE](http://redis.readthedocs.org/en/latest/pub_sub/subscribe.html#subscribe) 命令可以让客户端订阅任意数量的频道， 每当有新信息发送到被订阅的频道时， 信息就会被发送给所有订阅指定频道的客户端。



#### 订阅频道

每个 Redis 服务器进程都维持着一个表示服务器状态的 redis.h/redisServer 结构， 结构的 pubsub\_channels 属性是一个字典， 这个字典就用于保存订阅频道的信息：

其中，字典的键为正在被订阅的频道， 而字典的值则是一个链表， 链表中保存了所有订阅这个频道的客户端。



当客户端调用 [SUBSCRIBE](http://redis.readthedocs.org/en/latest/pub_sub/subscribe.html#subscribe) 命令时， 程序就将客户端和要订阅的频道在 pubsub\_channels 字典中关联起来。通过 pubsub\_channels 字典， 程序只要检查某个频道是否为字典的键， 就可以知道该频道是否正在被客户端订阅； 只要取出某个键的值， 就可以得到所有订阅该频道的客户端的信息。

#### 发送**信息到频道**

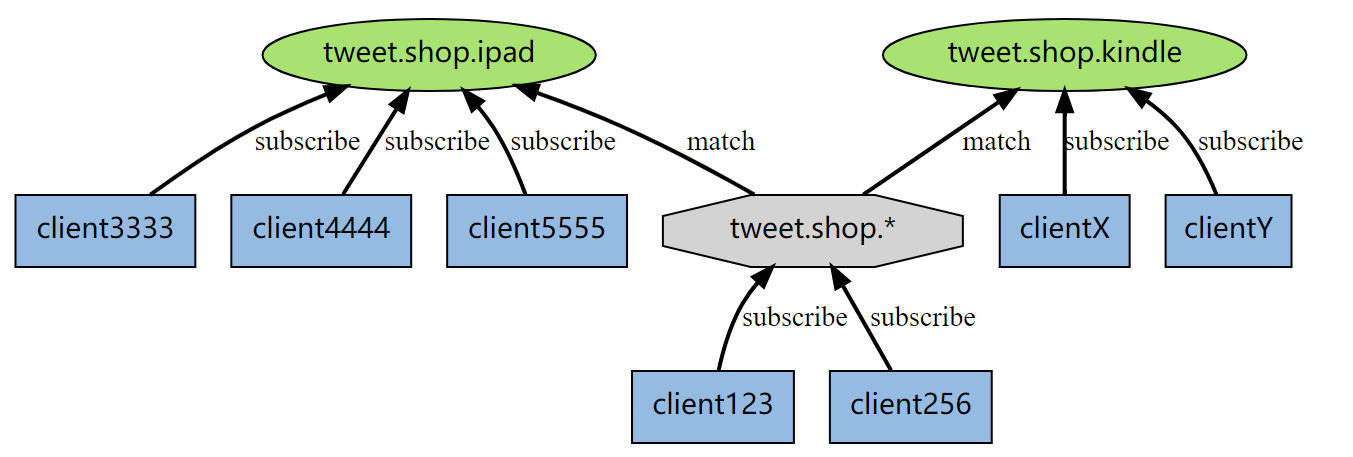
了解了 pubsub\_channels 字典的结构之后， 解释 [PUBLISH](http://redis.readthedocs.org/en/latest/pub_sub/publish.html#publish) 命令的实现就非常简单了： 当调用 PUBLISH channel message 命令， 程序首先根据 channel 定位到字典的键， 然后将信息发送给字典值链表中的所有客户端。

#### 退订频道

使用 [UNSUBSCRIBE](http://redis.readthedocs.org/en/latest/pub_sub/unsubscribe.html#unsubscribe) 命令可以退订指定的频道， 这个命令执行的是订阅的反操作： 它从 pubsub\_channels 字典的给定频道（键）中， 删除关于当前客户端的信息， 这样被退订频道的信息就不会再发送给这个客户端

### 模式的订阅与信息发送

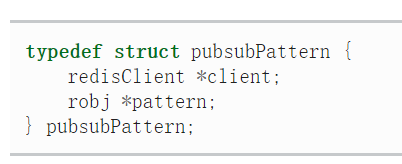
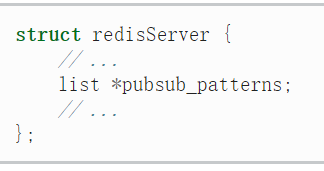
当使用 [PUBLISH](http://redis.readthedocs.org/en/latest/pub_sub/publish.html#publish) 命令发送信息到某个频道时， 不仅所有订阅该频道的客户端会收到信息， 如果有某个/某些模式和这个频道匹配的话， 那么所有订阅这个/这些频道的客户端也同样会收到信息。



当有信息发送到 tweet.shop.kindle 频道时， 信息除了发送给 clientX 和 clientY 之外， 还会发送给订阅 tweet.shop.\* 模式的 client123 和 client256 ；另一方面， 如果接收到信息的是频道 tweet.shop.ipad ， 那么 client123 和 client256 同样会收到信息。

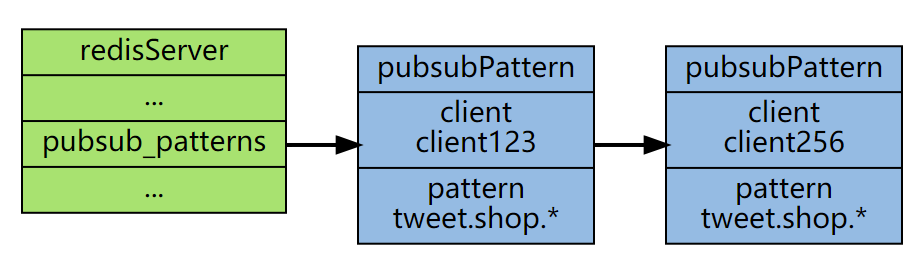
#### 订阅模式

redisServer.pubsub\_patterns 属性是一个链表，链表中保存着所有和模式相关的信息，链表中的每个节点都包含一个 redis.h/pubsubPattern 结构：



client 属性保存着订阅模式的客户端，而 pattern 属性则保存着被订阅的模式。

每当调用 PSUBSCRIBE 命令订阅一个模式时， 程序就创建一个包含客户端信息和被订阅模式的 pubsubPattern 结构， 并将该结构添加到 redisServer.pubsub\_patterns 链表中。



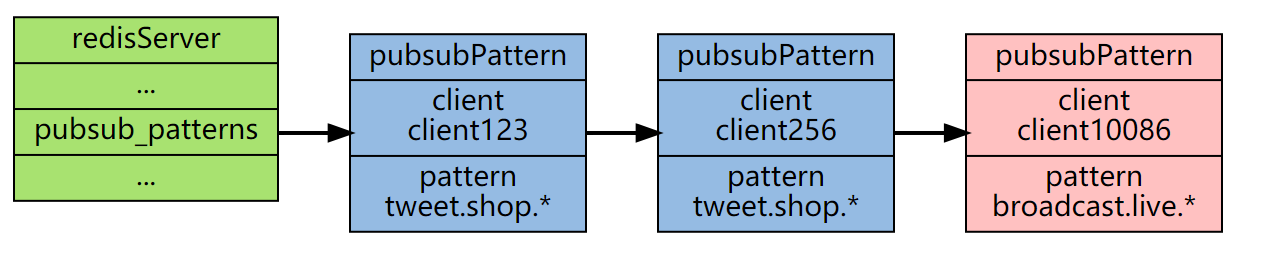
通过遍历整个 pubsub\_patterns 链表，程序可以检查所有正在被订阅的模式，以及订阅这些模式的客户端。

#### 发送信息到模式

完整描述 [PUBLISH](http://redis.readthedocs.org/en/latest/pub_sub/publish.html#publish) 功能的伪代码定于如下：



举个例子，如果 Redis 服务器的 pubsub\_patterns 状态如下：



那么当某个客户端发送信息 "Amazon Kindle, $69." 到 tweet.shop.kindle 频道时， 除了所有订阅了 tweet.shop.kindle 频道的客户端会收到信息之外， 客户端 client123 和 client256 也同样会收到信息， 因为这两个客户端订阅的 tweet.shop.\* 模式和 tweet.shop.kindle 频道匹配。

#### 退订模式

使用 [PUNSUBSCRIBE](http://redis.readthedocs.org/en/latest/pub_sub/punsubscribe.html#punsubscribe) 命令可以退订指定的模式， 这个命令执行的是订阅模式的反操作： 程序会删除 redisServer.pubsub\_patterns 链表中， 所有和被退订模式相关联的 pubsubPattern 结构， 这样客户端就不会再收到和模式相匹配的频道发来的信息。

## 持久化

### RDB

在 Redis 运行时， RDB 程序将当前内存中的数据库快照保存到磁盘文件中， 在 Redis 重启动时， RDB 程序可以通过载入 RDB 文件来还原数据库的状态。

RDB 功能最核心的是 rdbSave 和 rdbLoad 两个函数， 前者用于生成 RDB 文件到磁盘， 而后者则用于将 RDB 文件中的数据重新载入到内存中

#### 保存

rdbSave 函数负责将内存中的数据库数据以 RDB 格式保存到磁盘中， 如果 RDB 文件已存在， 那么新的 RDB 文件将替换已有的 RDB 文件。

[SAVE](http://redis.readthedocs.org/en/latest/server/save.html#save) 和 [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 两个命令都会调用 rdbSave 函数，但它们调用的方式各有不同：

* [SAVE](http://redis.readthedocs.org/en/latest/server/save.html#save) 直接调用 rdbSave ，阻塞 Redis 主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端的任何请求。
* [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 则 fork 出一个子进程，子进程负责调用 rdbSave ，并在保存完成之后向主进程发送信号，通知保存已完成。因为 rdbSave 在子进程被调用，所以 Redis 服务器在 [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 执行期间仍然可以继续处理客户端的请求。

Save执行时， 新的 [SAVE](http://redis.readthedocs.org/en/latest/server/save.html#save) 、 [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 或 [BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 调用都不会产生任何作用。另外， 因为 AOF 写入由后台线程完成， 而 [BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 则由子进程完成， 所以在 [SAVE](http://redis.readthedocs.org/en/latest/server/save.html#save) 执行的过程中， AOF 写入和 [BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 可以同时进行。

Bgsave执行时，调用新 [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 命令的客户端会收到一个出错信息，[BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 和 [BGSAVE](http://redis.readthedocs.org/en/latest/server/bgsave.html#bgsave) 不能同时执行。

#### 载入

当 Redis 服务器启动时， rdbLoad 函数就会被执行， 它读取 RDB 文件， 并将文件中的数据库数据载入到内存中。

在载入期间， 服务器每载入 1000 个键就处理一次所有已到达的请求， 只有 PUBLISH 、 SUBSCRIBE 、 PSUBSCRIBE 、 UNSUBSCRIBE 、 PUNSUBSCRIBE 五个命令的请求会被正确地处理， 其他命令一律返回错误。 等到载入完成之后， 服务器才会开始正常处理所有命令。

另外， 因为 AOF 文件的保存频率通常要高于 RDB 文件保存的频率， 所以一般来说， AOF 文件中的数据会比 RDB 文件中的数据要新。

因此， 如果服务器在启动时， 打开了 AOF 功能， 那么程序优先使用 AOF 文件来还原数据。 只有在 AOF 功能未打开的情况下， Redis 才会使用 RDB 文件来还原数据。

### AOF

AOF 则以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件，以此达到记录数据库状态的目的。同步命令到 AOF 文件的整个过程可以分为三个阶段

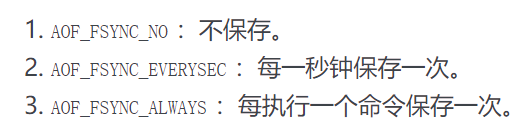
1. 命令传播：Redis 将执行完的命令、命令的参数、命令的参数个数等信息发送到 AOF 程序中。
2. 缓存追加：AOF 程序根据接收到的命令数据，将命令转换为网络通讯协议的格式，然后将协议内容追加到服务器的 AOF 缓存中。
3. 文件写入和保存：AOF 缓存中的内容被写入到 AOF 文件末尾，如果设定的 AOF 保存条件被满足的话， fsync 函数或者 fdatasync 函数会被调用，将写入的内容真正地保存到磁盘中。

#### AOF 保存模式

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件。

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

Redis 目前支持三种 AOF 保存模式，它们分别是：



**不保存**

在这种模式下， 每次调用 flushAppendOnlyFile 函数， WRITE 都会被执行， 但 SAVE 会被略过。

在这种模式下， SAVE 只会在以下任意一种情况中被执行：

* Redis 被关闭
* AOF 功能被关闭
* 系统的写缓存被刷新（可能是缓存已经被写满，或者定期保存操作被执行）

这三种情况下的 SAVE 操作都会引起 Redis 主进程阻塞。

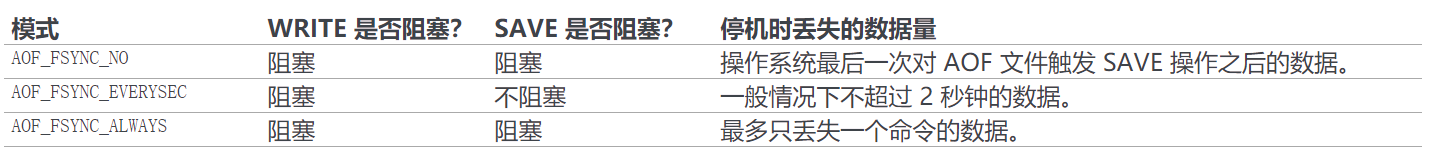
**每一秒钟保存一次**

在这种模式中， SAVE 原则上每隔一秒钟就会执行一次， 因为 SAVE 操作是由后台子线程调用的， 所以它不会引起服务器主进程阻塞。

**每执行一个命令保存一次**

在这种模式下，每次执行完一个命令之后， WRITE 和 SAVE 都会被执行。另外，因为 SAVE 是由 Redis 主进程执行的，所以在 SAVE 执行期间，主进程会被阻塞，不能接受命令请求

三种模式的性能对比：



#### AOF 文件的读取和数据还原

根据 AOF 文件里的协议， 重新执行一遍里面指示的所有命令， 就可以还原 Redis 的数据库状态了。

Redis 读取 AOF 文件并还原数据库的详细步骤如下：

1. 创建一个不带网络连接的伪客户端（fake client）。
2. 读取 AOF 所保存的文本，并根据内容还原出命令、命令的参数以及命令的个数。
3. 根据命令、命令的参数和命令的个数，使用伪客户端执行该命令。
4. 执行 2 和 3 ，直到 AOF 文件中的所有命令执行完毕。

完成第 4 步之后， AOF 文件所保存的数据库就会被完整地还原出来。

#### AOF 重写

AOF 文件通过同步 Redis 服务器所执行的命令， 从而实现了数据库状态的记录， 但是， 这种同步方式会造成一个问题： 随着运行时间的流逝， AOF 文件会变得越来越大。为了解决以上的问题， Redis 需要对 AOF 文件进行重写（rewrite）： 创建一个新的 AOF 文件来代替原有的 AOF 文件， 新 AOF 文件和原有 AOF 文件保存的数据库状态完全一样， 但新 AOF 文件的体积小于等于原有 AOF 文件的体积。

**AOF 重写的实现**

所谓的“重写”其实是一个有歧义的词语， 实际上， AOF 重写并不需要对原有的 AOF 文件进行任何写入和读取， 它针对的是数据库中键的当前值。根据键的类型， 使用适当的写入命令来重现键的当前值， 这就是 AOF 重写的实现原理。

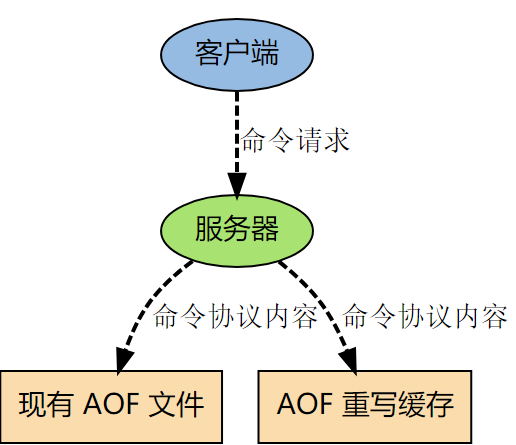
**AOF 后台重写**

执行重写创建新aof文件的过程会导致进程阻塞， 作为一种辅佐性的维护手段， Redis 不希望 AOF 重写造成服务器无法处理请求， 所以 Redis 决定将 AOF 重写程序放到（后台）子进程里执行， 这样处理的最大好处是：

1. 子进程进行 AOF 重写期间，主进程可以继续处理命令请求。
2. 子进程带有主进程的数据副本，使用子进程而不是线程，可以在避免锁的情况下，保证数据的安全性。

不过， 使用子进程也有一个问题需要解决： 因为子进程在进行 AOF 重写期间， 主进程还需要继续处理命令， 而新的命令可能对现有的数据进行修改， 这会让当前数据库的数据和重写后的 AOF 文件中的数据不一致。

为了解决这个问题， Redis 增加了一个 AOF 重写缓存， 这个缓存在 fork 出子进程之后开始启用， Redis 主进程在接到新的写命令之后， 除了会将这个写命令的协议内容追加到现有的 AOF 文件之外， 还会追加到这个缓存中：



当子进程完成 AOF 重写之后， 它会向父进程发送一个完成信号， 父进程在接到完成信号之后， 会调用一个信号处理函数， 并完成以下工作：

1. 将 AOF 重写缓存中的内容全部写入到新 AOF 文件中。
2. 对新的 AOF 文件进行改名，覆盖原有的 AOF 文件。

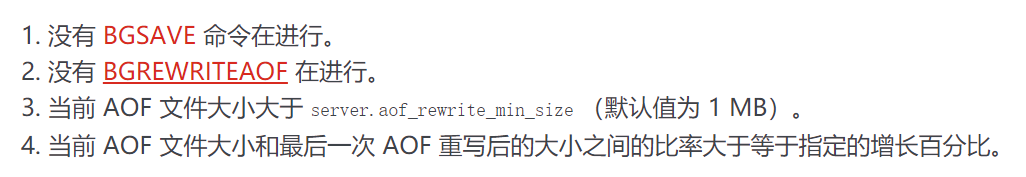
当步骤 1 执行完毕之后， 现有 AOF 文件、新 AOF 文件和数据库三者的状态就完全一致了；当步骤 2 执行完毕之后， 程序就完成了新旧两个 AOF 文件的交替。

这个信号处理函数执行完毕之后， 主进程就可以继续像往常一样接受命令请求了。 在整个 AOF 后台重写过程中， 只有最后的写入缓存和改名操作会造成主进程阻塞， 在其他时候， AOF 后台重写都不会对主进程造成阻塞， 这将 AOF 重写对性能造成的影响降到了最低。

以上就是 AOF 后台重写， 也即是 [BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 命令的工作原理。

AOF 重写可以由用户通过调用 [BGREWRITEAOF](http://redis.readthedocs.org/en/latest/server/bgrewriteaof.html#bgrewriteaof) 手动触发；

每次当 serverCron 函数执行时， 它都会检查以下条件是否全部满足， 如果是的话， 就会触发自动的 AOF 重写：



## 内存淘汰机制

### 使用

我们可以通过配置redis.conf中的maxmemory这个值来开启内存淘汰功能，我们可以通过这个值来设置内存淘汰算法

内存淘汰的过程是：

1. 客户端发起了需要申请更多内存的命令（如set）。
2. Redis检查内存使用情况，如果已使用的内存大于maxmemory则开始根据用户配置的不同淘汰策略（就是配置的maxmemory这个值）来淘汰内存（key），从而换取一定的内存。
3. 如果上面都没问题，则这个命令执行成功。

maxmemory为0的时候表示我们对Redis的内存使用没有限制。

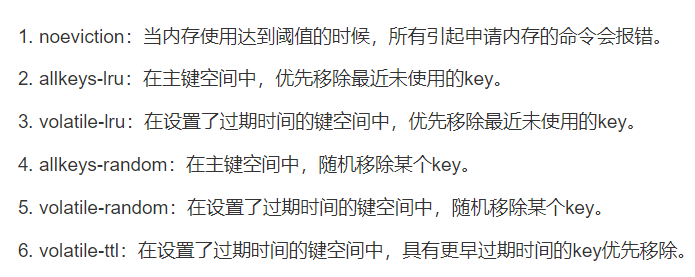
### 内存淘汰策略

Redis 为我们提供了多种内存淘汰策略，我们可以根据不同的应用场景应用不同的策略。

这里补充一下主键空间和设置了过期时间的键空间，举个例子，假设我们有一批键存储在Redis中，则有那么一个哈希表用于存储这批键及其值，如果这批键中有一部分设置了过期时间，那么这批键还会被存储到另外一个哈希表中，这个哈希表中的值对应的是键被设置的过期时间。设置了过期时间的键空间为主键空间的子集。我们来看如何设置key的过期时间，通过以下的指令



Redis提供了下面几种淘汰策略供用户选择，其中默认的策略为noeviction策略：



* allkeys-lru：如果我们的应用对缓存的访问符合幂律分布（也就是存在相对**热点数据**），或者我们不太清楚我们应用的缓存访问分布状况，我们可以选择allkeys-lru策略。
* allkeys-random：如果我们的应用对于缓存key的访问概率相等，则可以使用这个策略。
* volatile-ttl：这种策略使得我们可以向Redis提示哪些key更适合被eviction。

### 非精准的LRU

上面提到的LRU（Least Recently Used）策略，实际上Redis实现的LRU并不是可靠的LRU，也就是名义上我们使用LRU算法淘汰键，但是实际上被淘汰的键并不一定是真正的最久没用的，这里涉及到一个权衡的问题，如果需要在**全部键空间内搜索最优解**，则必然会增加系统的开销，Redis是单线程的，也就是同一个实例在每一个时刻只能服务于一个客户端，所以耗时的操作一定要谨慎。为了在一定成本内实现相对的LRU，早期的Redis版本是基于采样的LRU，也就是放弃全部键空间内搜索解改为**采样空间搜索最优解**。自从Redis3.0版本之后，Redis作者对于基于采样的LRU进行了一些优化，目的是在一定的成本内让结果更靠近真实的LRU。