Redis

1. 为什么使用redis？

在项目中使用redis，主要是从两个角度去考虑:性能和并发。

当然，redis还具备可以做分布式锁等其他功能，但是如果只是为了分布式锁这些其他功能，完全还有其他中间件(如zookpeer等)代替，并不是非要使用redis。

* 1. 性能

我们在碰到需要执行耗时特别久，且结果不频繁变动的SQL，就特别适合将运行结果放入缓存。

* 1. 并发

在**大并发**的情况下，所有的请求直接访问数据库，数据库会出现连接异常。

这个时候，就需要使用redis做一个缓冲操作，让请求先访问到redis，而不是直接访问数据库;

1. 使用redis有什么缺点?
   1. 缓存和数据库双写一致性问题

一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。

答这个问题，先明白一个前提。就是如果对数据有强一致性要求，不能放缓存。

我们所做的一切，只能保证最终一致性。

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。

* 1. 缓存雪崩问题

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:

* 给缓存的失效时间，加上一个随机值，**避免集体失效**。
* 使用互斥锁,缓存失效后先获取锁再请求数据库，但是该方案吞吐量明显下降了。
* 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

1. 从缓存A读数据，有则直接返回；
2. 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；
3. 更新线程同时更新缓存A和缓存B；
   1. 缓存击穿问题

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。

* 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；
* 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。
* 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。
  1. 缓存的并发竞争问题

这个问题大致就是，同时有多个子系统去set一个key。

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（**但是没有依赖关系**）

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。

其他方法，比如利用队列，将set方法变成串行访问也可以。总之，灵活变通。

1. 单线程的redis为什么这么快

这个问题其实是对redis内部机制的一个考察。

很多人其实都不知道redis是单线程工作模型

* 1. 纯内存操作
  2. 单线程操作，避免了频繁的上下文切换
  3. 采用了非阻塞I/O多路复用机制
     1. 一个简单的例子

小曲在S城开了一家快递店，负责同城快送服务。小曲因为资金限制，雇佣了一批快递员，然后小曲发现资金不够了，只够买一辆车送快递；

方式一：

客户每送来一份快递，小曲就让一个快递员盯着，然后快递员开车去送快递。

慢慢的小曲就发现了这种经营方式存在下述问题：

1. 几十个快递员基本上时间都花在了抢车上了，大部分快递员都处在闲置状态，谁抢到了车，谁就能去送快递（**抢占式的线程调度**）；
2. 随着快递的增多，快递员也越来越多，小曲发现快递店里越来越挤，没办法雇佣新的快递员了(**线程数量**)；
3. 快递员之间的协调很花时间(**线程上线文切换**)；

方式二：

雇佣一个快递员。然后呢，客户送来的快递，小曲按送达地点标注好，然后依次放在一个地方。最后，那个快递员依次的去取快递，一次拿一个，然后开着车去送快递，送好了就回来拿下一个快递。

对比

上述两种经营方式对比，是不是明显觉得第二种，效率更高，更好呢。在上述比喻中:

每个快递员——————>每个线程

每个快递——————–>每个socket(I/O流)

快递的送达地点————–>socket的不同状态

客户送快递请求————–>来自客户端的请求

小曲的经营方式————–>服务端运行的代码

一辆车———————->CPU的核数

* 1. redis的线程模型

Redis客户端对服务端的每次调用都经历了**发送命令，执行命令，返回结果**三个过程。

其中执**行命令阶段**，由于Redis是单线程来处理命令的，所有每一条到达服务端的命令不会立刻执行，所有的命令都会进入**一个队列**中，然后逐个被执行。

并且多个客户端发送的命令的执行顺序是不确定的。但是**可以确定的是不会有两条命令被同时执行，不会产生并发问题**，这就是Redis的单线程基本模型

* + 1. redis的多路复用选择器

Redis 基于 Reactor 模式开发了自己的**网络事件处理器**：

这个处理器被称为文件事件处理器（file event handler）：

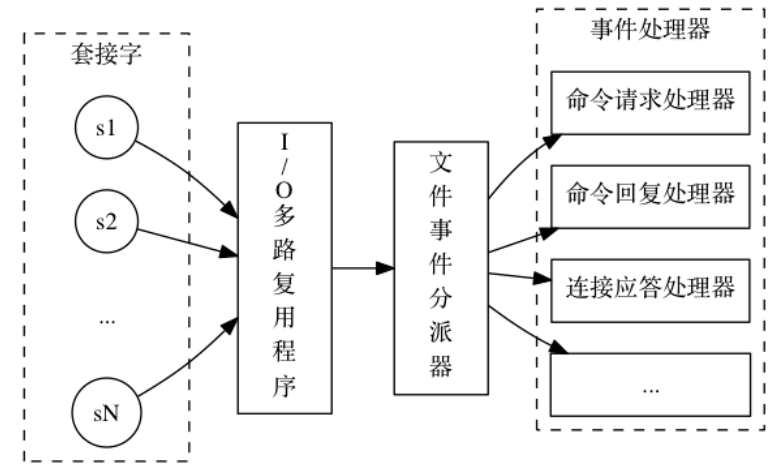
文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。

当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时**文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件**。

* + 1. 文件事件处理器的构成

文件事件处理器的四个组成部分：

* 套接字；
* I/O 多路复用程序；
* 文件事件分派器（dispatcher）；
* 以及事件处理器。



* 文件事件是对套接字操作的抽象， 每当一个套接字准备好执行连接应答（accept）、写入、读取、关闭等操作时， 就会产生一个文件事件。 因为一个服务器通常会连接多个套接字， 所以多个文件事件有可能会并发地出现。
* I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字；
* 尽管多个文件事件可能会并发地出现， 但 I/O 多路复用程序总是会将所有产生事件的套接字都入队到一个队列里面； 然后**文件事件分配器**通过这个队列， **以有序（sequentially）、同步（synchronously）、每次一个套接字**的方式读取： 当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O 多路复用程序才会继续向文件事件分派器传送下一个套接字。
  + 1. 单线程模型每秒万级别处理能力的原因
* 纯内存访问。数据存放在内存中，内存的响应时间大约是100纳秒，这是Redis每秒万亿级别访问的重要基础。
* 非阻塞I/O，Redis采用epoll做为I/O多路复用技术的实现，再加上Redis自身的事件处理模型将epoll中的连接，读写，关闭都转换为了时间，不在I/O上浪费过多的时间。
* 单线程避免了线程切换和竞态产生的消耗。
* Redis采用单线程模型，每条命令执行如果占用大量时间，会造成其他线程阻塞，对于Redis这种高性能服务是致命的，所以Redis是面向高速执行的数据库。

1. 数据类型
   1. String

* String是简单的 key-value 键值对，value 不仅可以是 String，也可以是数字。
* String在redis内部存储**默认就是一个字符串**，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。
  + 1. 应用场景

String是最常用的一种数据类型，普通的key/value存储都可以归为此类，这里就不所做解释了。

* + 1. 命令

MGET key1 [key2..] 得到所有的给定键的值

MSET key value [key value...] 设置多个键和多个值

* 1. List（列表）

Redis列表是简单的**字符串列表**，简单的说就是一个链表或者说是一个队列。

可以从头部或尾部向Redis列表添加元素。

列表的最大长度为2^32 - 1，也即每个列表支持超过40亿个元素。

**Redis list的实现为一个双向链表**，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销;

* + 1. 应用场景

Redis list的应用场景非常多，也是Redis最重要的数据结构之一;

如有的应用使用Redis的list类型实现一个简单的**轻量级消息队列**，生产者push，消费者pop/bpop。

利用LRANGE还可以很方便的实现分页的功能;

* + 1. 命令

LPOP key 获取并取出列表中的第一个元素；

LPUSH key value1 [value2] 在前面加上一个或多个值的列表

* 1. Hash（字典，哈希表）

Redis Hash对应**Value内部实际就是一个HashMap；**

实际这里会有2种不同实现：

这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap；

当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。

* + 1. 应用场景

假设有多个用户及对应的用户信息，可以用来存储以用户ID为key，将用户信息序列化为比如json格式做为value进行保存；

* + 1. 命令

HDEL key field[field...] 删除对象的一个或几个属性域，不存在的属性将被忽略；

HEXISTS key field 查看对象是否存在该属性域；

HGET key field 获取对象中该field属性域的值；

HSET **key** **field** **value** 设置对象**指定字段**的值；

HKEYS key 获取对象的所有属性字段；

HVALS key 获取对象的所有属性值；

* 1. Set（集合）

可以理解为一堆**值不重复的列表**，类似数学领域中的集合概念，且Redis也提供了针对集合的求交集、并集、差集等操作。

**set 的内部实现是一个 value永远为null的HashMap**，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。

* + 1. 应用场景

Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于**set是可以自动排重**的；

当需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择；

set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的；

在微博应用中，每个用户关注的人存在一个集合中，就很容易实现求两个人的共同好友功能，通过交集实现；

* + 1. 命令

**SADD** key member [member ...] 添加一个或者多个元素到集合(set)里；

**SCARD** key 获取集合里面的元素数量；

**SINTER** key [key ...] 获得两个集合的交集

* 1. Sorted Set（有序集合）

Redis有序集合类似Redis集合，不同的是增加了一个功能，即集合是有序的。一个有序集合的**每个成员带有分数**，用于进行排序。

Redis有序集合添加、删除和测试的时间复杂度均为O(1)(固定时间，无论里面包含的元素集合的数量)。列表的最大长度为2^32- 1元素(4294967295，超过40亿每个元素的集合)。

Redis sorted set的内部使用**HashMap和跳跃表(SkipList)**来保证数据的存储和有序：

HashMap里放的是**成员到score的映射；**

而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

* + 1. 使用场景

Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。

当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

又比如**用户的积分排行榜**需求就可以通过有序集合实现。还有上面介绍的使用List实现轻量级的消息队列，其实也可以通过Sorted Set实现有优先级或按权重的队列；

* + 1. 命令

ZADD key score1 member1 [score2 member2] 添加一个或多个成员到有序集合，或者如果它已经存在更新其分数（score）；

ZCARD key 得到的有序集合成员的数量；

1. redis的过期策略以及内存淘汰机制
   1. 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.

* 1. 定期删除+惰性删除的实现方式

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。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。

* 1. 内存淘汰机制

存在的原因：

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。

配置及策略：

在redis.conf中有一行配置

**# maxmemory-policy volatile-lru**

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，目前项目在用这种。

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。

* + 1. 原因

* + 1. 配置及策略

1. redis的主从

像MySQL一样，**redis是支持主从同步的**，而且也支持一主多从以及多级从结构。

主从结构:

一是为了纯粹的冗余备份

二是为了提升读性能，比如很消耗性能的SORT就可以由从服务器来承担。

**redis的主从同步是异步进行的**，这意味着主从同步不会影响主逻辑，也不会降低redis的处理性能。

* 1. 注意事项

主从架构中，可以考虑关闭主服务器的数据持久化功能，只让从服务器进行持久化，这样可以**提高主服务器的处理性能**。

在主从架构中，从服务器通常被设置为**只读模式**，这样可以避免从服务器的数据被误修改。但是从服务器仍然可以接受CONFIG等指令，所以还是不应该将从服务器直接暴露到不安全的网络环境中。如果必须如此，那可以考虑给重要指令进行重命名，来避免命令被外人误执行。

* 1. 主从同步原理

实现步骤：

1. 从服务器会向主服务器发出SYNC指令；
2. 当主服务器接到此命令后，就会调用**BGSAVE指令**来创建一个子进程专门进行数据持久化工作，也就是将主服务器的数据写入RDB文件中。在数据持久化期间，主服务器将执行的写指令都缓存在内存中。
3. 在BGSAVE指令执行完成后，主服务器会将持久化好的RDB文件发送给从服务器;
4. 从服务器接到此文件后会将其存储到磁盘上，然后再将其读取到内存中。
5. 4 完成后，主服务器会将这段时间缓存的写指令再以redis协议的格式发送给从服务器。

注意事项：

* 即使有多个从服务器同时发来SYNC指令，主服务器也只会执行一次BGSAVE，然后把持久化好的RDB文件发给多个下游；
* 在redis2.8版本之前，如果从服务器与主服务器因某些原因断开连接的话，都会进行一次主从之间的全量的数据同步；而在2.8版本之后，redis支持了效率更高的增量同步策略，这大大降低了连接断开的恢复成本。

增量通过原理：

* 主服务器会在内存中维护一个**缓冲区**，缓冲区中存储着将要发给从服务器的内容。
* 从服务器在与主服务器出现网络瞬断之后，从服务器会尝试再次与主服务器连接，一旦连接成功，从服务器就会把“**希望同步的主服务器ID**”和“**希望请求的数据的偏移位置**（replication offset）”发送出去。
* 主服务器接收到这样的同步请求后，**首先**会验证主服务器ID是否和自己的ID匹配;**其次**会检查“请求的偏移位置”是否存在于自己的缓冲区中；如果两者都满足的话，主服务器就会向从服务器发送增量内容。

增量同步功能，需要服务器端支持全新的PSYNC指令。这个指令，只有在redis-2.8之后才具有。

1. redis的事务

众所周知，事务是指“一个完整的动作，要么全部执行，要么什么也没有做”；

四个redis指令，即MULTI、EXEC、DISCARD、WATCH。这四个指令构成了redis事务处理的基础。

1.MULTI用来组装一个事务；

2.EXEC用来执行一个事务；

3.DISCARD用来取消一个事务；

**4.WATCH用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行；**

* 1. 实例

redis> MULTI //标记事务开始

OK

redis> INCR user\_id //多条命令按顺序入队

QUEUED

redis> INCR user\_id

QUEUED

redis> INCR user\_id

QUEUED

redis> PING

QUEUED

redis> EXEC //执行

**1) (integer) 1**

**2) (integer) 2**

**3) (integer) 3**

**4) PONG**

**我们看到了QUEUED的字样，这表示我们在用MULTI组装事务时，每一个命令都会进入到内存队列中缓存起来，**

**如果出现QUEUED则表示我们这个命令成功插入了缓存队列，在将来执行EXEC时，这些被QUEUED的命令都会被组装成一个事务来执行;**

* 1. 事务执行的两种错误

1.调用EXEC之前的错误：

“调用EXEC之前的错误”，有可能是由于**语法有误导致的**，也可能时由于内存不足导致的。只要出现某个命令无法成功写入缓冲队列的情况，redis都会进行记录，在客户端调用EXEC时，redis会拒绝执行这一事务。（这时2.6.5版本之后的策略。在2.6.5之前的版本中，redis会忽略那些入队失败的命令，只执行那些入队成功的命令）。

实例：

127.0.0.1:6379> multi

OK

**127.0.0.1:6379> haha //一个明显错误的指令**

(error) ERR unknown command 'haha'

127.0.0.1:6379> ping

QUEUED

127.0.0.1:6379> exec

**//redis无情的拒绝了事务的执行，原因是“之前出现了错误”**

(error) EXECABORT Transaction discarded because of previous errors.

2.调用EXEC之后的错误:

“调用EXEC之后的错误”，redis则采取了完全不同的策略，**即redis不会理睬这些错误，而是继续向下执行事务中的其他命令**。这是因为，对于应用层面的错误，并不是redis自身需要考虑和处理的问题，所以一个事务中如果某一条命令执行失败，并不会影响接下来的其他命令的执行。

实例：

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set age 23

QUEUED

**//age不是集合，所以如下是一条明显错误的指令**

127.0.0.1:6379> sadd age 15

QUEUED

127.0.0.1:6379> set age 29

QUEUED

127.0.0.1:6379> exec //执行事务时，redis不会理睬第2条指令执行错误

1) OK

2) (error) WRONGTYPE Operation against a key holding the wrong kind of value

3) OK

127.0.0.1:6379> get age

"29" //可以看出第3条指令被成功执行了

* 1. WATCH指令

这是一个很好用的指令，它可以帮我们实现类似于“**乐观锁**”的效果，即CAS（check and set）；

WATCH本身的作用是“**监视key是否被改动过**”，而且支持同时监视多个key，只要还没真正触发事务，WATCH都会尽职尽责的监视，一旦发现某个key被修改了，在执行EXEC时就会返回nil，表示事务无法触发。

实例：

127.0.0.1:6379> set age 23

OK

**127.0.0.1:6379> watch age //开始监视age**

OK

**127.0.0.1:6379> set age 24 //在EXEC之前，age的值被修改了**

OK

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set age 25

QUEUED

127.0.0.1:6379> get age

QUEUED

**127.0.0.1:6379> exec //触发EXEC**

(nil) //事务无法被执行

1. redis的持久化

redis提供了两种持久化的方式，**分别是RDB（Redis DataBase）和AOF（Append Only File）**。

* RDB：在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上；
* AOF：将redis执行过的**所有写指令记录下来**，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。
  1. redis持久化 – RDB

RDB方式，是将redis某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法。

* redis在进行数据持久化的过程中，会先将数据写入到一个临时文件中，待持久化过程都结束了，才会用这个临时文件替换上次持久化好的文件。正是这种特性，让我们可以随时来进行备份，因为快照文件总是完整可用的。
* 对于RDB方式，redis会单独创建（fork）一个子进程来进行持久化，而主进程是不会进行任何IO操作的，这样就确保了redis极高的性能。
* 如果需要进行大规模数据的恢复，且**对于数据恢复的完整性不是非常敏感**，那RDB方式要比AOF方式更加的高效。
* 虽然RDB有不少优点，但它的缺点也是不容忽视的。如果你对数据的完整性非常敏感，那么RDB方式就不太适合你，因为即使你每5分钟都持久化一次，当redis故障时，仍然会有近5分钟的数据丢失。所以，redis还提供了另一种持久化方式，那就是AOF。
  1. redis持久化 – AOF

AOF，英文是Append Only File，**即只允许追加不允许改写的文件**。

如果你直接执行BGREWRITEAOF命令，那么redis会生成一个全新的AOF文件，其中便包括了可以恢复现有数据的最少的命令集。

如果运气比较差，AOF文件出现了被写坏的情况，也不必过分担忧，redis并不会贸然加载这个有问题的AOF文件，而是报错退出。这时可以通过以下步骤来修复出错的文件：

1.备份被写坏的AOF文件

2.运行redis-check-aof –fix进行修复

3.用diff -u来看下两个文件的差异，确认问题点

4.重启redis，加载修复后的AOF文件

* + 1. 原理

原理：

AOF方式是将执行过的写指令记录下来，在数据恢复时按照从前到后的顺序再将指令都执行一遍，就这么简单。

* + 1. 启用配置

启用：

我们通过配置redis.conf中的appendonly yes就可以打开AOF功能。如果有写操作（如SET等），redis就会被追加到AOF文件的末尾。

* + 1. 持久化策略

默认的AOF持久化策略是**每秒钟fsync一次**（fsync是指把缓存中的写指令记录到磁盘中），因为在这种情况下，redis仍然可以保持很好的处理性能，即使redis故障，也只会丢失最近1秒钟的数据。

* + 1. 存在问题集解决方案

1、如果在追加日志时，恰好遇到磁盘空间满、inode满或断电等情况导致日志写入不完整:

redis提供了redis-check-aof工具，可以用来进行日志修复。

2、因为采用了追加方式，如果不做任何处理的话**，AOF文件会变得越来越大：**

redis提供了AOF文件**重写（rewrite）机制**，即当AOF文件的大小超过所设定的阈值时，redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集。

假如我们调用了100次INCR指令，在AOF文件中就要存储100条指令，但这明显是很低效的，完全可以把这100条指令合并成一条SET指令，这就是**重写机制的原理**。

3、在进行AOF重写时遇到断电、磁盘满等问题：

采用先写临时文件，全部完成后再替换的流程，所以都不会影响AOF文件的可用性。

* + 1. AOF重写

AOF重写的内部运行原理，我们有必要了解一下。

* 在重写即将开始之际，redis会创建（fork）一个“**重写子进程**”，这个子进程会首先读取现有的AOF文件，并将其包含的指令进行分析压缩并写入到一个临时文件中。
* 与此同时，主工作进程会将新接收到的写指令**一边累积到内存缓冲区中**，**一边继续写入到原有的AOF文件中**，这样做是保证原有的AOF文件的可用性，避免在重写过程中出现意外。
* 当“重写子进程”完成重写工作后，它会给父进程发一个信号，父进程收到信号后就会将内存中缓存的写指令追加到新AOF文件中。
* 当追加结束后，redis就会用新AOF文件来代替旧AOF文件，之后再有新的写指令，就都会追加到新的AOF文件中了。
  + 1. 优点

AOF方式的另一个好处，数据恢复：

我们通过一个“场景再现”来说明。某同学在操作redis时，不小心执行了FLUSHALL，导致redis内存中的数据全部被清空了，这是很悲剧的事情。不过这也不是世界末日，只要redis配置了AOF持久化方式，**且AOF文件还没有被重写（rewrite）：**

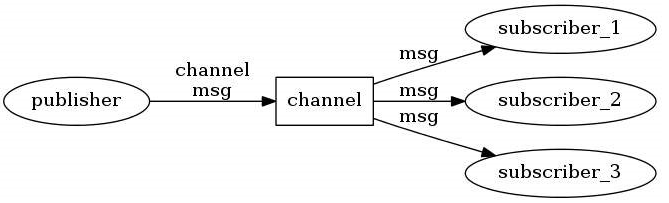
1. 我们可以用最快的速度暂停redis并编辑AOF文件；
2. 将最后一行的FLUSHALL命令删除；
3. 然后重启redis，就可以恢复redis的所有数据到FLUSHALL之前的状态了。

但是如果AOF文件已经被重写了，那就无法通过这种方法来恢复数据了。

* + 1. 缺点

1. 比如在同样数据规模的情况下，AOF文件要比RDB文件的体积大；
2. AOF方式的恢复速度也要慢于RDB方式。
3. redis的发布与订阅
   1. Redis发布订阅架构

Redis提供了发布订阅功能，可以用于消息的传输，Redis的发布订阅机制包括三个部分：**发布者**，**订阅者**和**Channel;**



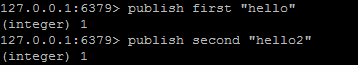
发布者和订阅者都是Redis客户端，Channel则为Redis服务器端，发布者将消息发送到某个的频道，订阅了这个频道的订阅者就能接收到这条消息。

Redis的这种发布订阅机制与基于主题的发布订阅类似，Channel相当于主题。

* 1. Redis发布订阅功能

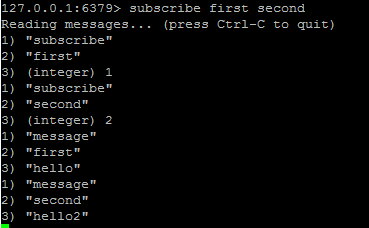
（1）发送消息

Redis采用**PUBLISH命令发送消息**，其返回值为接收到该消息的订阅者的数量。



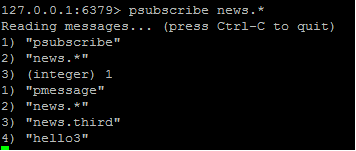
（2）订阅某个频道

Redis采用**SUBSCRIBE命令订阅某个频道**，其返回值包括客户端订阅的频道，目前已订阅的频道数量，以及接收到的消息，其中subscribe表示已经成功订阅了某个频道。



（3）模式匹配

模式匹配功能允许**客户端订阅符合某个模式的频道**，Redis采用**PSUBSCRIBE**订阅符合某个模式所有频道，用“”表示模式，“”可以被任意值代替。



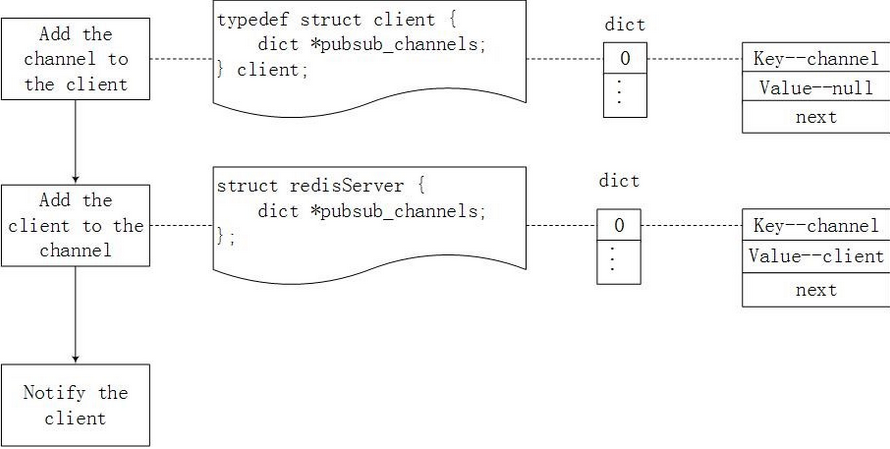
假设客户端同时订阅了某种模式和符合该模式的某个频道，那么发送给这个频道的消息将被客户端接收到两次，只不过这两条消息的类型不同，一个是message类型，一个是pmessage类型，但其内容相同。

（4）取消订阅

Redis采用**UNSUBSCRIBE和PUNSUBSCRIBE**命令取消订阅，其返回值与订阅类似。

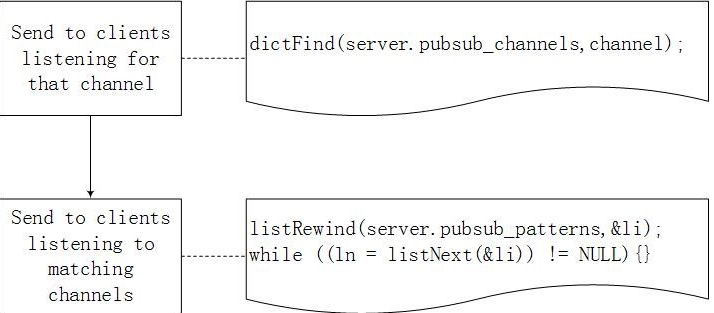
由于Redis的订阅操作是**阻塞式的**，因此一旦客户端订阅了某个频道或模式，就将会一直处于订阅状态直到退出。在SUBSCRIBE，PSUBSCRIBE，UNSUBSCRIBE和PUNSUBSCRIBE命令中，其返回值都包含了该客户端当前订阅的频道和模式的数量，当这个数量变为0时，该客户端会自动退出订阅状态。

* 1. Redis发布订阅实现
     1. SUBSCRIBE



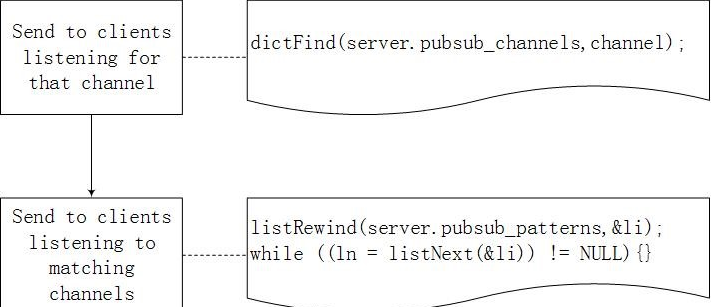
当客户端订阅某个频道时，Redis需要将该频道和该客户端绑定：

1. 首先，在客户端结构体**client**中，有一个属性为pubsub\_channels，该属性表明了该客户端订阅的所有频道，它是一个字典类型，通过哈希表实现，其中的每个元素都包含了一个键值对以及指向下一个元素的指针，每次订阅都要向其中插入一个结点，**键表示订阅的频道，值为空**。
2. 然后，在表示服务器端的结构体**redisServer**中，也有一个属性为pubsub\_channels，但此处它表示的是该服务器端中的所有频道以及订阅了这个频道的客户端，它也是一个字典类型，插入结点时，**键表示频道，值则是订阅了这个频道的所有客户端组成的链表**。最后Redis通知客户端其订阅成功。
   * 1. PSUBSCRIBE



当客户端订阅某个**模式**时，Redis同样需要将该模式和该客户端绑定。

1. 首先，在结构体client中，有一个属性为**pubsub\_patterns**，该属性表示该客户端订阅的所有模式，它是一个链表类型，每个结点包括了订阅的模式和指向下一个结点的指针，每次订阅某个模式时，都要向其中插入一个结点。
2. 然后，在结构体**redisServer**中，有一个属性也叫**pubsub\_patterns**，它表示了该服务器端中的所有模式和订阅了这些模式的客户端，它也是一个链表类型，插入结点时，每个结点都要包含订阅的模式，以及订阅这个模式的客户端，和指向下一个结点的指针
   * 1. PUBLISH



当客户端向某个频道发送消息时：

首先在结构体redisServer中的pubsub\_channels中找出键为该频道的结点，遍历该结点的值，即遍历订阅了该频道的所有客户端，将消息发送给这些客户端。

然后，遍历结构体redisServer中的pubsub\_patterns，找出包含该频道的模式的结点，将消息发送给订阅了该模式的客户端；

* 1. Redis发布订阅在Redis中的应用

Redis的发布订阅功能与Redis中的数据存储时无关的，它不会影响Redis的key space，即不会影响Redis中存储的数据，但通过发布订阅机制，Redis还提供了另一个功能，即Keyspace Notification，允许客户端通过订阅特定的频道，从而得知是否有改变Redis中的数据的事件。

例如，有一个客户端删除了Redis中键为mykey的数据，该操作会触发两条消息，mykey del和del mykey，前者属于频道keysapce，表示keyspace发生的变化，后者属于频道keyevent，表示执行的操作

* 1. Redis发布订阅与ActiveMQ的比较

（1）ActiveMQ支持多种消息协议，包括AMQP，MQTT，Stomp等，并且支持JMS规范，但Redis没有提供对这些协议的支持；

（2）ActiveMQ提供持久化功能，但Redis无法对消息持久化存储，一旦消息被发送，如果没有订阅者接收，那么消息就会丢失；

（3）ActiveMQ提供了消息传输保障，当客户端连接超时或事务回滚等情况发生时，消息会被重新发送给客户端，Redis没有提供消息传输保障。

总之，ActiveMQ所提供的功能远比Redis发布订阅要复杂，毕竟Redis不是专门做发布订阅的，但是如果系统中已经有了Redis，并且需要基本的发布订阅功能，就没有必要再安装ActiveMQ了，因为可能ActiveMQ提供的功能大部分都用不到，而Redis的发布订阅机制就能满足需求。

1. redis的分布式锁