Redis第二次课 Redis的原理分析

过期时间设置

在Redis中提供了Expire命令设置一个键的过期时间，到期以后Redis会自动删除它。这个在我们实际使用过程中用得非常多。

EXPIRE命令的使用方法为

EXPIRE key seconds

其中seconds 参数表示键的过期时间，单位为秒。

EXPIRE 返回值为1表示设置成功，0表示设置失败或者键不存在

如果向知道一个键还有多久时间被删除，可以使用TTL命令

TTL key

当键不存在时，TTL命令会返回-2

而对于没有给指定键设置过期时间的，通过TTL命令会返回-1

如果向取消键的过期时间设置（使该键恢复成为永久的），可以使用PERSIST命令，如果该命令执行成功或者成功清除了过期时间，则返回1 。 否则返回0（键不存在或者本身就是永久的）

EXPIRE命令的seconds命令必须是整数，所以最小单位是1秒，如果向要更精确的控制键的过期时间可以使用PEXPIRE命令，当然实际过程中用秒的单位就够了。 PEXPIRE命令的单位是毫秒。即PEXPIRE key 1000与EXPIRE key 1相等；对应的PTTL以毫秒单位获取键的剩余有效时间

还有一个针对字符串独有的过期时间设置方式

setex(String key,int seconds,String value)

过期删除的原理

Redis 中的主键失效是如何实现的，即失效的主键是如何删除的？实际上，Redis 删除失效主键的方法主要有两种：

消极方法（passive way）

在主键被访问时如果发现它已经失效，那么就删除它

积极方法（active way）

周期性地从设置了失效时间的主键中选择一部分失效的主键删除

对于那些从未被查询的key，即便它们已经过期，被动方式也无法清除。因此Redis会周期性地随机测试一些key，已过期的key将会被删掉。Redis每秒会进行10次操作，具体的流程：

\1. 随机测试 20 个带有timeout信息的key；

\2. 删除其中已经过期的key；

\3. 如果超过25%的key被删除，则重复执行步骤1；

这是一个简单的概率算法（trivial probabilistic algorithm），基于假设我们随机抽取的key代表了全部的key空间。

Redis发布订阅

Redis提供了发布订阅功能，可以用于消息的传输，Redis提供了一组命令可以让开发者实现“发布/订阅”模式(publish/subscribe) . 该模式同样可以实现进程间的消息传递，它的实现原理是

发布/订阅模式包含两种角色，分别是发布者和订阅者。订阅者可以订阅一个或多个频道，而发布者可以向指定的频道发送消息，所有订阅此频道的订阅者都会收到该消息

发布者发布消息的命令是PUBLISH， 用法是

PUBLISH channel message

比如向channel.1发一条消息:hello

PUBLISH channel.1 “hello”

这样就实现了消息的发送，该命令的返回值表示接收到这条消息的订阅者数量。因为在执行这条命令的时候还没有订阅者订阅该频道，所以返回为0. 另外值得注意的是消息发送出去不会持久化，如果发送之前没有订阅者，那么后续再有订阅者订阅该频道，之前的消息就收不到了

订阅者订阅消息的命令是

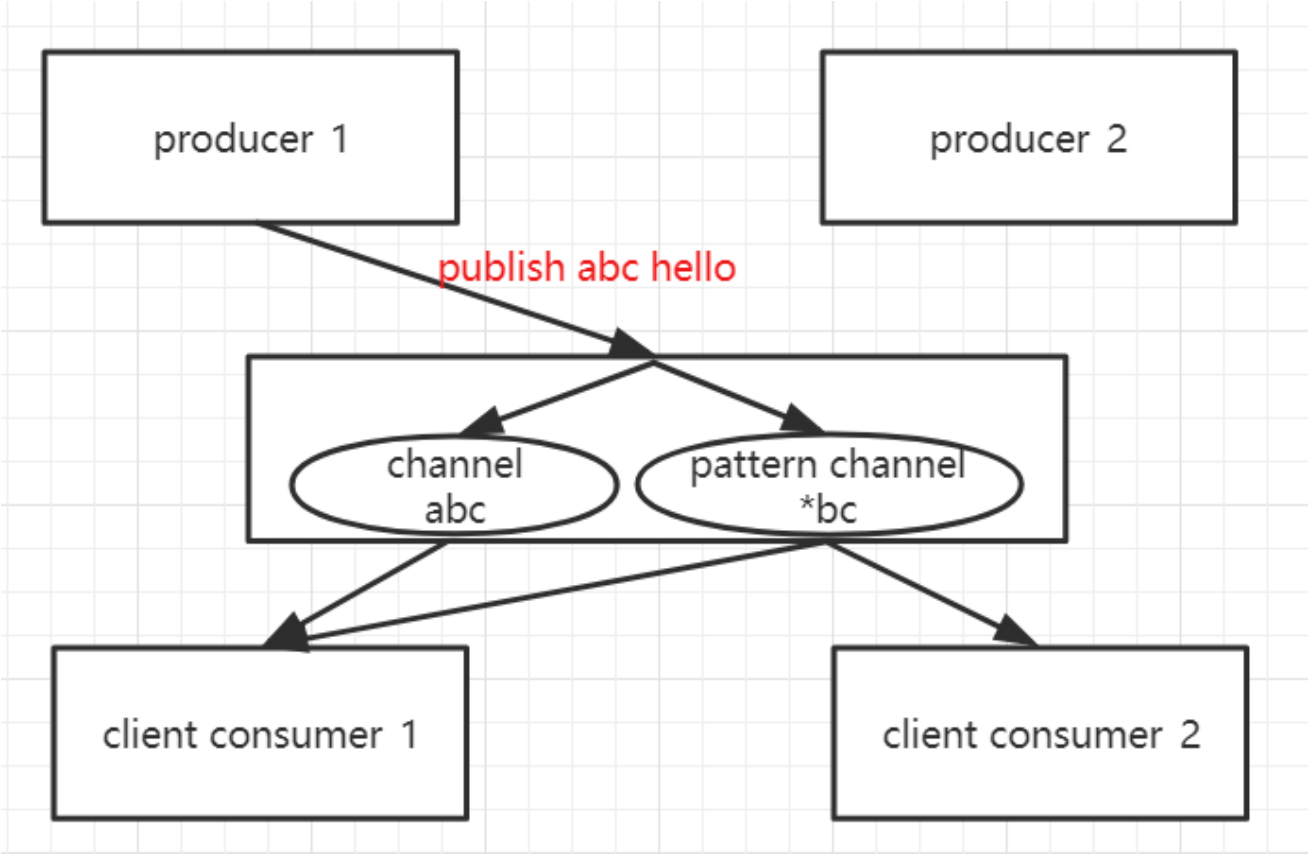
SUBSCRIBE channel [channel ...]

该命令同时可以订阅多个频道，比如订阅channel.1的频道。 SUBSCRIBE channel.1

执行SUBSCRIBE命令后客户端会进入订阅状态

结构图

channel分两类，一个是普通channel、另一个是pattern channel（规则匹配）， producer1发布了一条消息【publish abc hello】,redis server发给abc这个普通channel上的所有订阅者，同时abc也匹配上了pattern channel的名字，所以这条消息也会同时发送给pattern channel *bc上的所有订阅者



Redis的数据是如何持久化的?

Redis支持两种方式的持久化，一种是RDB方式、另一种是AOF（append-only-file）方式。前者会根据指定的规则“定时”将内存中的数据存储在硬盘上，而后者在每次执行命令后将命令本身记录下来。两种持久化方式可以单独使用其中一种，也可以将这两种方式结合使用

RDB方式

当符合一定条件时，Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，等到持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能。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。RDB的缺点是最后一次持久化后的数据可能丢失

--fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

Redis会在以下几种情况下对数据进行快照

\1. 根据配置规则进行自动快照

\2. 用户执行SAVE或者GBSAVE命令

\3. 执行FLUSHALL命令

\4. 执行复制(replication)时

根据配置规则进行自动快照

Redis允许用户自定义快照条件，当符合快照条件时，Redis会自动执行快照操作。快照的条件可以由用户在配置文件中配置。配置格式如下

save

第一个参数是时间窗口，第二个是键的个数，也就是说，在第一个时间参数配置范围内被更改的键的个数大于后面的changes时，即符合快照条件。redis默认配置了三个规则

save 900 1

save 300 10

save 60 10000

每条快照规则占一行，每条规则之间是“或”的关系。 在900秒（15分）内有一个以上的键被更改则进行快照。

用户执行SAVE或BGSAVE命令

除了让Redis自动进行快照以外，当我们对服务进行重启或者服务器迁移我们需要人工去干预备份。redis提供了两条命令来完成这个任务

\1. save命令

当执行save命令时，Redis同步做快照操作，在快照执行过程中会阻塞所有来自客户端的请求。当redis内存中的数据较多时，通过该命令将导致Redis较长时间的不响应。所以不建议在生产环境上使用这个命令，而是推荐使用bgsave命令

\2. bgsave命令

bgsave命令可以在后台异步地进行快照操作，快照的同时服务器还可以继续响应来自客户端的请求。执行BGSAVE后，Redis会立即返回ok表示开始执行快照操作。

通过LASTSAVE命令可以获取最近一次成功执行快照的时间； （自动快照采用的是异步快照操作）

执行FLUSHALL命令

该命令在前面讲过，会清除redis在内存中的所有数据。执行该命令后，只要redis中配置的快照规则不为空，也就是save 的规则存在。redis就会执行一次快照操作。不管规则是什么样的都会执行。如果没有定义快照规则，就不会执行快照操作

执行复制时

该操作主要是在主从模式下，redis会在复制初始化时进行自动快照。这个会在后面讲到；

这里只需要了解当执行复制操作时，及时没有定义自动快照规则，并且没有手动执行过快照操作，它仍然会生成RDB快照文件

AOF方式

当使用Redis存储非临时数据时，一般需要打开AOF持久化来降低进程终止导致的数据丢失。AOF可以将Redis执行的每一条写命令追加到硬盘文件中，这一过程会降低Redis的性能，但大部分情况下这个影响是能够接受的，另外使用较快的硬盘可以提高AOF的性能

开启AOF

默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数启用，在redis.conf中找到 appendonly yes

开启AOF持久化后每执行一条会更改Redis中的数据的命令后，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是apendonly.aof. 可以在redis.conf中的属性 appendfilename appendonlyh.aof修改

AOF的实现

AOF文件以纯文本的形式记录Redis执行的写命令例如开启AOF持久化的情况下执行如下4条命令

set foo 1

set foo 2

set foo 3

get

redis 会将前3条命令写入AOF文件中，通过vim的方式可以看到aof文件中的内容

	我们会发现AOF文件的内容正是Redis发送的原始通信协议的内容，从内容中我们发现Redis只记录了3条命令。然后这时有一个问题是前面2条命令其实是冗余的，因为这两条的执行结果都会被第三条命令覆盖。随着执行的命令越来越多，AOF文件的大小也会越来越大，其实内存中实际的数据可能没有多少，那这样就会造成磁盘空间以及redis数据还原的过程比较长的问题。因此我们希望Redis可以自动优化AOF文件，就上面这个例子来说，前面两条是可以被删除的。 而实际上Redis也考虑到了，可以配置一个条件，每当达到一定条件时Redis就会自动重写AOF文件，这个条件的配置问 auto-aof-rewrite-percentage 100 auto-aof-rewrite-min-size 64mb

auto-aof-rewrite-percentage 表示的是当目前的AOF文件大小超过上一次重写时的AOF文件大小的百分之多少时会再次进行重写，如果之前没有重写过，则以启动时AOF文件大小为依据

auto-aof-rewrite-min-size 表示限制了允许重写的最小AOF文件大小，通常在AOF文件很小的情况下即使其中有很多冗余的命令我们也并不太关心。

另外，还可以通过BGREWRITEAOF 命令手动执行AOF，执行完以后冗余的命令已经被删除了

在启动时，Redis会逐个执行AOF文件中的命令来将硬盘中的数据载入到内存中，载入的速度相对于RDB会慢一些

AOF的重写原理

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。

重写的流程是这样，主进程会fork一个子进程出来进行AOF重写，这个重写过程并不是基于原有的aof文件来做的，而是有点类似于快照的方式，全量遍历内存中的数据，然后逐个序列到aof文件中。在fork子进程这个过程中，服务端仍然可以对外提供服务，那这个时候重写的aof文件的数据和redis内存数据不一致了怎么办？不用担心，这个过程中，主进程的数据更新操作，会缓存到aof_rewrite_buf中，也就是单独开辟一块缓存来存储重写期间收到的命令，当子进程重写完以后再把缓存中的数据追加到新的aof文件。

当所有的数据全部追加到新的aof文件中后，把新的aof文件重命名为，此后所有的操作都会被写入新的aof文件。

如果在rewrite过程中出现故障，不会影响原来aof文件的正常工作，只有当rewrite完成后才会切换文件。因此这个rewrite过程是比较可靠的

Redis内存回收策略

Redis中提供了多种内存回收策略，当内存容量不足时，为了保证程序的运行，这时就不得不淘汰内存中的一些对象，释放这些对象占用的空间，那么选择淘汰哪些对象呢?

其中，默认的策略为noeviction策略，当内存使用达到阈值的时候，所有引起申请内存的命令会报错

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

适合的场景：如果我们的应用对缓存的访问都是相对热点数据，那么可以选择这个策略

allkeys-random：随机移除某个key。

适合的场景：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-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰。

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰。

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

适合场景：这种策略使得我们可以向Redis提示哪些key更适合被淘汰，我们可以自己控制

总结

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算法淘汰内存数据，但是实际上被淘汰的键并不一定是真正的最少使用的数据，这里涉及到一个权衡的问题，如果需要在所有的数据中搜索最符合条件的数据，那么一定会增加系统的开销，Redis是单线程的，所以耗时的操作会谨慎一些。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，也就是放弃了从所有数据中搜索解改为采样空间搜索最优解。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。

Redis是单进程单线程？性能为什么这么快

Redis采用了一种非常简单的做法，单线程来处理来自所有客户端的并发请求，Redis把任务封闭在一个线程中从而避免了线程安全问题；redis为什么是单线程?

官方的解释是，CPU并不是Redis的瓶颈所在，Redis的瓶颈主要在机器的内存和网络的带宽。那么Redis能不能处理高并发请求呢？当然是可以的，至于怎么实现的，我们来具体了解一下。【注意并发不等于并行，并发性I/O流，意味着能够让一个计算单元来处理来自多个客户端的流请求。并行性，意味着服务器能够同时执行几个事情，具有多个计算单元】

多路复用

Redis 是跑在单线程中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对其它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

了解多路复用之前，先简单了解下几种I/O模型

（1）同步阻塞IO（Blocking IO）：即传统的IO模型。

（2）同步非阻塞IO（Non-blocking IO）：默认创建的socket都是阻塞的，非阻塞IO要求socket被设置为NONBLOCK。

（3）IO多路复用（IO Multiplexing）：即经典的Reactor设计模式，也称为异步阻塞IO，Java中的Selector和Linux中的epoll都是这种模型。

（4）异步IO（Asynchronous IO）：即经典的Proactor设计模式，也称为异步非阻塞IO。

同步和异步、阻塞和非阻塞，到底是什么意思，感觉原理都差不多，我来简单解释一下

同步和异步，指的是用户线程和内核的交互方式

阻塞和非阻塞，指用户线程调用内核IO操作的方式是阻塞还是非阻塞

就像在Java中使用多线程做异步处理的概念，通过多线程去执行一个流程，主线程可以不用等待。而阻塞和非阻塞我们可以理解为假如在同步流程或者异步流程中做IO操作，如果缓冲区数据还没准备好，IO的这个过程会阻塞，这个在之前讲TCP协议的时候有讲过.

在Redis中使用Lua脚本

我们在使用redis的时候，会面临一些问题，比如

原子性问题

前面我们讲过，redis虽然是单一线程的，当时仍然会存在线程安全问题，当然，这个线程安全问题不是来源安于Redis服务器内部。而是Redis作为数据服务器，是提供给多个客户端使用的。多个客户端的操作就相当于同一个进程下的多个线程，如果多个客户端之间没有做好数据的同步策略，就会产生数据不一致的问题。举个简单的例子

多个客户端的命令之间没有做请求同步，导致实际执行顺序可能会不一致，最终的结果也就无法满足原子性了。

效率问题

redis本身的吞吐量是非常高的，因为它首先是基于内存的数据库。在实际使用过程中，有一个非常重要的因素影响redis的吞吐量，那就是网络。我们在使用redis实现某些特定功能的时候，很可能需要多个命令或者多个数据类型的交互才能完成，那么这种多次网络请求对性能影响比较大。当然redis也做了一些优化，比如提供了pipeline管道操作，但是它有一定的局限性，就是执行的多个命令和响应之间是不存在相互依赖关系的。所以我们需要一种机制能够编写一些具有业务逻辑的命令，减少网络请求

Lua

Redis中内嵌了对Lua环境的支持，允许开发者使用Lua语言编写脚本传到Redis中执行，Redis客户端可以使用Lua脚本，直接在服务端原子的执行多个Redis命令。

使用脚本的好处：

\1. 减少网络开销，在Lua脚本中可以把多个命令放在同一个脚本中运行

\2. 原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件

\3. 复用性，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

Lua是一个高效的轻量级脚本语言(javascript、shell、sql、python、ruby...)，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能;

Redis与Lua

先初步的认识一下在redis中如何结合lua来完成一些简单的操作

在Lua脚本中调用Redis命令

在Lua脚本中调用Redis命令，可以使用redis.call函数调用。比如我们调用string类型的命令

redis.call('set','hello','world')

local value=redis.call('get','hello')

redis.call 函数的返回值就是redis命令的执行结果。前面我们介绍过redis的5中类型的数据返回的值的类型也都不一样。redis.call函数会将这5种类型的返回值转化对应的Lua的数据类型

从Lua脚本中获得返回值

在很多情况下我们都需要脚本可以有返回值，毕竟这个脚本也是一个我们所编写的命令集，我们可以像调用其他redis内置命令一样调用我们自己写的脚本，所以同样redis会自动将脚本返回值的Lua数据类型转化为Redis的返回值类型。 在脚本中可以使用return 语句将值返回给redis客户端，通过return语句来执行，如果没有执行return，默认返回为nil。

EVAL命令的格式是

[EVAL][脚本内容] [key参数的数量][key ...] [arg ...]

可以通过key和arg这两个参数向脚本中传递数据，他们的值可以在脚本中分别使用**KEYS**和**ARGV** 这两个类型的全局变量访问。比如我们通过脚本实现一个set命令，通过在redis客户端中调用，那么执行的语句是：

lua脚本的内容为： return redis.call('set',KEYS[1],ARGV[1]) //KEYS和ARGV必须大写

eval "return redis.call('set',KEYS[1],ARGV[1])" **1** lua1 hello

注意：EVAL命令是根据 key参数的数量-也就是上面例子中的**1**来将后面所有参数分别存入脚本中KEYS和ARGV两个表类型的全局变量。当脚本不需要任何参数时也不能省略这个参数。如果没有参数则为0

```
eval "return redis.call('get','lua1')" 0
```

EVALSHA命令

考虑到我们通过eval执行lua脚本，脚本比较长的情况下，每次调用脚本都需要把整个脚本传给redis，比较占用带宽。为了解决这个问题，redis提供了EVALSHA命令允许开发者通过脚本内容的SHA1摘要来执行脚本。该命令的用法和EVAL一样，只不过是将脚本内容替换成脚本内容的SHA1摘要

\1. Redis在执行EVAL命令时会计算脚本的SHA1摘要并记录在脚本缓存中

\2. 执行EVALSHA命令时Redis会根据提供的摘要从脚本缓存中查找对应的脚本内容，如果找到了就执行脚本，否则返回“NOSCRIPT No matching script,Please use EVAL”

通过以下案例来演示EVALSHA命令的效果

script load "return redis.call('get','lua1')" 将脚本加入缓存并生成sha1命令

evalsha "a5a402e90df3eaeca2ff03d56d99982e05cf6574" **0**

我们在调用eval命令之前，先执行evalsha命令，如果提示脚本不存在，则再调用eval命令