本章对Redis服务器的数据库实现进行介绍，说明Redis数据库相关操作的实现，包括数据库中键值对的添加、删除、查看、更新等操作的实现；客户端切换数据库的实现；键超时相关功能的实现、键空间事件通知等。

以上这些功能，键空间事件通知是在src/notify.c中实现的，其他功能都是在src/db.c中实现的。

在redis.h中定义的redisServer数据结构，定义了redis服务器相关的所有属性，其中就包含了数据库的结构：

struct redisServer **{**

**...**

redisDb **\***db**;**

**...**

int dbnum**;**

**...**

**}**

其中，db是一个redisDb 结构类型的数组，该数组共有dbnum个redisDb结构，每一个redisDb结构就表示Redis中的一个数据库。dbnum的默认值为16，该值可以在配置文件中的databases配置选项进行配置。

**一：客户端切换数据库**

默认情况下，客户端使用0号数据库，客户端可以使用select命令，切换其要使用的数据库。在redis.h中定义的redisClient结构，就代表redis客户端，其定义如下

**typedef** struct redisClient **{**

**...**

redisDb **\***db**;**

**...**

**}** redisClient**;**

其中的db，就表示该客户端当前使用的数据库，该指针指向redisServer中数组db中的某个元素。因此，使用select切换数据库的实现很简单，仅仅改变redisClient中的db指向另一个数组成员即可，代码如下：

int selectDb**(**redisClient **\***c**,** int id**)** **{**

**if** **(**id **<** 0 **||** id **>=** server**.**dbnum**)**

**return** REDIS\_ERR**;**

c**->**db **=** **&**server**.**db**[**id**];**

**return** REDIS\_OK**;**

**}**

注意，到目前为止，Redis仍没有可以返回客户端当前使用的数据库的命令。在数次切换数据库之后，很可能会忘记自己当前正在使用的是哪个数据库。当出现这种情况时，为了避免对数据库进行误操作，在执行像FLUSHDB这样的危险命令之前，最好先执行一个SELECT命令，显式地切换到指定的数据库，然后才执行该命令。

**二：数据库中的键值对操作**

Redis是一个键值对数据库服务器，所有的键值对都是保存在字典(dict)这种数据结构中的。数据库redisDb定义在redis.h中，它的定义如下：

**typedef** struct redisDb **{**

dict **\***dict**;** */\* The keyspace for this DB \*/*

dict **\***expires**;** */\* Timeout of keys with a timeout set \*/*

**...**

int id**;** */\* Database ID \*/*

**...**

**}** redisDb**;**

其中，dict就是该数据库中，保存所有键值对的字典；id表示该数据库在redisServer中数组db中的索引。字典expires中，保存了所有设置了超时时间的键，其中的值记录了该键的超时时间。

注意，在底层实现中，dict中的键，都是以原始字符串的形式保存的，而非字符串对象。

1：查找键值对

在数据库中查找键值对，最终都是通过函数lookupKey实现的，它的代码如下：

robj **\***lookupKey**(**redisDb **\***db**,** robj **\***key**)** **{**

dictEntry **\***de **=** dictFind**(**db**->**dict**,**key**->**ptr**);**

**if** **(**de**)** **{**

robj **\***val **=** dictGetVal**(**de**);**

*/\* Update the access time for the ageing algorithm.*

*\* Don't do it if we have a saving child, as this will trigger*

*\* a copy on write madness. \*/*

**if** **(**server**.**rdb\_child\_pid **==** **-**1 **&&** server**.**aof\_child\_pid **==** **-**1**)**

val**->**lru **=** LRU\_CLOCK**();**

**return** val**;**

**}** **else** **{**

**return** **NULL;**

**}**

**}**

该函数在数据库db中寻找键key对应的值对象。找到则返回值的robj结构，否则返回NULL。

根据dictFind(db->dict, key->ptr)，说明在数据库db的dict中，键是以原始字符串的形式存储的；

找到值对象val后，如果当前没有进行RDB或者AOF过程，则更新该值对象的lru属性，也就是更新该键值对的访问时间。

2：设置键值对（添加或者更新）

向数据库中添加键值对，是通过函数dbAdd实现的，它的代码如下：

void dbAdd**(**redisDb **\***db**,** robj **\***key**,** robj **\***val**)** **{**

sds copy **=** sdsdup**(**key**->**ptr**);**

int retval **=** dictAdd**(**db**->**dict**,** copy**,** val**);**

redisAssertWithInfo**(NULL,**key**,**retval **==** REDIS\_OK**);**

**if** **(**val**->**type **==** REDIS\_LIST**)** signalListAsReady**(**db**,** key**);**

**if** **(**server**.**cluster\_enabled**)** slotToKeyAdd**(**key**);**

**}**

首先复制键对象key中的原始字符串为copy，然后利用dictAdd，将原始字符串copy，值对象val添加到db->dict中。

如果值对象是一个列表对象，则还调用signalListAsReady，使得在该列表键上调用brpop或blpop的客户端能停止阻塞，得到相应的值；如果Redis还开启了集群功能，则调用slotToKeyAdd保存该key在cluster中的槽位号。

注意，如果数据库字典db->dict中已经存在相同的key了，则添加失败，这种情况下程序直接报错退出，保证字典中不会有相同key的工作，是由调用者负责的；而且本函数也不增加键值对的引用计数，这也是由调用者负责的。

更新数据库中的某个键的值，是由函数dbOverwrite实现的，它的代码如下：

void dbOverwrite**(**redisDb **\***db**,** robj **\***key**,** robj **\***val**)** **{**

dictEntry **\***de **=** dictFind**(**db**->**dict**,**key**->**ptr**);**

redisAssertWithInfo**(NULL,**key**,**de **!=** **NULL);**

dictReplace**(**db**->**dict**,** key**->**ptr**,** val**);**

**}**

该函数主要是通过调用dictReplace实现的。

注意，如果数据库字典db->dict中找不到key，则程序直接报错退出，保证字典中key确实存在的工作，是由调用者负责的；而且本函数也不增加键值对的引用计数，这也是由调用者完成的。

在Redis数据库中，设置键值对的set类命令，是由函数setKey实现的，它的代码如下：

void setKey**(**redisDb **\***db**,** robj **\***key**,** robj **\***val**)** **{**

**if** **(**lookupKeyWrite**(**db**,**key**)** **==** **NULL)** **{**

dbAdd**(**db**,**key**,**val**);**

**}** **else** **{**

dbOverwrite**(**db**,**key**,**val**);**

**}**

incrRefCount**(**val**);**

removeExpire**(**db**,**key**);**

signalModifiedKey**(**db**,**key**);**

**}**

首先调用lookupKeyWrite，在db中寻找该key，找到了则调用dbOverwrite更新值，找不到则调用dbAdd添加键值对；

该函数还会做以下工作：调用incrRefCount，增加值对象的引用计数；调用removeExpire，将key从db->expires中删除，也就是清除键的生存时间，这是set命令的效果之一；调用signalModifiedKey，用于表明该key被更新了，以使watch该key的客户端感知到。

3：删除键值对

从数据库中删除键值对的操作，是由函数dbDelete实现的，它的代码如下：

int dbDelete**(**redisDb **\***db**,** robj **\***key**)** **{**

*/\* Deleting an entry from the expires dict will not free the sds of*

*\* the key, because it is shared with the main dictionary. \*/*

**if** **(**dictSize**(**db**->**expires**)** **>** 0**)** dictDelete**(**db**->**expires**,**key**->**ptr**);**

**if** **(**dictDelete**(**db**->**dict**,**key**->**ptr**)** **==** DICT\_OK**)** **{**

**if** **(**server**.**cluster\_enabled**)** slotToKeyDel**(**key**);**

**return** 1**;**

**}** **else** **{**

**return** 0**;**

**}**

**}**

首先，如果db->expires非空的话，尝试从db->expires中删除该key；然后，尝试从db->dict中删除该key，删除成功并且服务器开启了集群功能，则调用slotToKeyDel，从跳跃表server.cluster->slots\_to\_keys中删除该key的记录。

该函数删除成功返回1，否则返回0。

**三：键的生存时间功能**

通过expire类命令(expire, pexpire, expireat, pexpireat)，客户端可以为数据库中的某个键设置过期时间，当键的过期时间来临时，服务器就会自动从数据库中删除这个键。

可以通过ttl（或pttl）命令，查看某个键的剩余生存时间，也就是，返回距离这个键被服务器自动删除还有多长时间。

上文提到过，redisDb结构中的字典expires，保存了所有设置了超时时间的键。字典expires中的每个键值对，键与redisDb中的dict共享，值记录了该键的生存时间，生存时间都是以一个以毫秒表示的Unix时间戳进行保存的。

1：设置超时时间

在src/db.c中，setExpire函数实现了设置键key的超时时间为when的功能，代码如下：

void setExpire**(**redisDb **\***db**,** robj **\***key**,** long long when**)** **{**

dictEntry **\***kde**,** **\***de**;**

*/\* Reuse the sds from the main dict in the expire dict \*/*

kde **=** dictFind**(**db**->**dict**,**key**->**ptr**);**

redisAssertWithInfo**(NULL,**key**,**kde **!=** **NULL);**

de **=** dictReplaceRaw**(**db**->**expires**,**dictGetKey**(**kde**));**

dictSetSignedIntegerVal**(**de**,**when**);**

**}**

该函数首先从db->dict中查找key->ptr对应的dictEntry结构kde，如果找不到，则直接报错退出；

然后以key->ptr为键，以when为值，将该键值对添加到字典db->expires中。其中when表示超时时间的绝对值，也就是以毫秒表示的Unix时间戳。

注意，以下是初始化服务器的db->expires字典的代码：

**for** **(**j **=** 0**;** j **<** server**.**dbnum**;** j**++)** **{**

server**.**db**[**j**].**dict **=** dictCreate**(&**dbDictType**,NULL);**

server**.**db**[**j**].**expires **=** dictCreate**(&**keyptrDictType**,NULL);**

**...**

server**.**db**[**j**].**id **=** j**;**

**...**

**}**

初始化字典db->expires时使用的dictType是keyptrDictType，它的定义如下：

dictType keyptrDictType **=** **{**

dictSdsHash**,** */\* hash function \*/*

**NULL,** */\* key dup \*/*

**NULL,** */\* val dup \*/*

dictSdsKeyCompare**,** */\* key compare \*/*

**NULL,** */\* key destructor \*/*

**NULL** */\* val destructor \*/*

**};**

可见，除了哈希函数和键匹配函数之外，其他的函数均设置为NULL。

因此，在setExpire函数中，调用dictReplaceRaw将key->ptr添加到字典db->expires中时，字典项dictEntry中的key成员直接赋值为key->ptr，也就是说对于同一个键，db->expires中的dictEntry.key和db->dict中的dictEntry.key，它们的值是相同的，都指向同一个原始字符串。

调用dictSetSignedIntegerVal设置超时时间，是直接将when的值，赋值给dictEntry->v.s64，也就是直接保存在结构dictEntry中了。

2：删除键的超时时间

在src/db.c中，removeExpire函数实现了删除键的超时时间的功能，它的代码如下：

int removeExpire**(**redisDb **\***db**,** robj **\***key**)** **{**

*/\* An expire may only be removed if there is a corresponding entry in the*

*\* main dict. Otherwise, the key will never be freed. \*/*

redisAssertWithInfo**(NULL,**key**,**dictFind**(**db**->**dict**,**key**->**ptr**)** **!=** **NULL);**

**return** dictDelete**(**db**->**expires**,**key**->**ptr**)** **==** DICT\_OK**;**

**}**

该函数主要是通过调用dictDelete将key->ptr从db->expires中删除实现的。

因为，key->ptr是由db->dict和db->expires共享的。初始化字典db->expires的dictType是keyptrDictType，其中的”key destructor”和”val destructor”函数都是NULL，因此，在dictDelete中，并不会将key->ptr指向的内存空间释放，它仅仅是将对应的字典项dictEntry，从db->expires中删除释放而已。

3：删除超时的键

删除一个过期的键，一共有三种策略，它们分别是：

a、定时删除：在设置键的过期时间的同时，创建一个定时器，该定时器在键的过期时间来临时，立即执行对键的删除操作。

b、惰性删除：放任键过期不管，每次从键空间中访问键时，都检查该键是否过期，如果过期的话，则删除该键。

c、定期删除：每隔一段时间，程序就对数据库进行一次检查，删除其中的过期键。至 于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

定时删除策略对内存是最友好的：通过使用定时器，该策略可以保证过期键会尽快地被删除，并释放过期键所占用的内存。

另一方面，定时删除策略对CPU时间是最不友好的：在过期键比较多的情况下，需要创建大量的定时器，删除过期键这一行为可能会占用相当一部分CPU时间，可能会对服务器的响应时间和吞吐量造成影响。

惰性删除策略对CPU时间来说是最友好的：程序只会在访问键时才对键进行过期检查，这可以保证删除过期键的操作只会在非做不可的情况下进行，并且删除的目标仅限于当前处理的键，这个策略不会在删除其他无关的过期键上花费任何CPU时间。

另一方面，惰性删除策略的缺点是，它对内存是最不友好的：如果数据库中有非常多的过期键，而这些过期键又恰好没有被访问到的话，那么它们也许永远也不会被删除，这就类似于内存泄漏的情况了。

定期删除策略是前两种策略的一种折中：该策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对CPU时间的影响。另一方面，定期删除策略有效地减少了因为过期键而带来的内存浪费。

Redis服务器实际使用的是惰性删除和定期删除两种策略，通过配合使用这两种删除策略，服务器可以很好地在合理使用CPU时间和避免浪费内存空间之间取得平衡。

惰性删除时通过src/db.c中的expireIfNeeded实现的，每次访问数据库中的键值对时，都会调用该函数，判断该键值对是否已经过期，如果是，则在expireIfNeeded中将其删除，如果没有过期，则expireIfNeeded不采取任何动作。比如，获取键值对用于写操作的实现函数lookupKeyWrite：

robj **\***lookupKeyWrite**(**redisDb **\***db**,** robj **\***key**)** **{**

expireIfNeeded**(**db**,**key**);**

**return** lookupKey**(**db**,**key**);**

**}**

可见，在查找键之前，就会调用expireIfNeeded，对过期的键进行删除。

下面就是expireIfNeeded函数的代码：

int expireIfNeeded**(**redisDb **\***db**,** robj **\***key**)** **{**

mstime\_t when **=** getExpire**(**db**,**key**);**

mstime\_t now**;**

**if** **(**when **<** 0**)** **return** 0**;** */\* No expire for this key \*/*

*/\* Don't expire anything while loading. It will be done later. \*/*

**if** **(**server**.**loading**)** **return** 0**;**

*/\* If we are in the context of a Lua script, we claim that time is*

*\* blocked to when the Lua script started. This way a key can expire*

*\* only the first time it is accessed and not in the middle of the*

*\* script execution, making propagation to slaves / AOF consistent.*

*\* See issue #1525 on Github for more information. \*/*

now **=** server**.**lua\_caller **?** server**.**lua\_time\_start **:** mstime**();**

*/\* If we are running in the context of a slave, return ASAP:*

*\* the slave key expiration is controlled by the master that will*

*\* send us synthesized DEL operations for expired keys.*

*\**

*\* Still we try to return the right information to the caller,*

*\* that is, 0 if we think the key should be still valid, 1 if*

*\* we think the key is expired at this time. \*/*

**if** **(**server**.**masterhost **!=** **NULL)** **return** now **>** when**;**

*/\* Return when this key has not expired \*/*

**if** **(**now **<=** when**)** **return** 0**;**

*/\* Delete the key \*/*

server**.**stat\_expiredkeys**++;**

propagateExpire**(**db**,**key**);**

notifyKeyspaceEvent**(**REDIS\_NOTIFY\_EXPIRED**,**

"expired"**,**key**,**db**->**id**);**

**return** dbDelete**(**db**,**key**);**

**}**

首先调用getExpire从db中取得key的超时时间，如果该key未设置超时时间，则直接返回0；

如果服务器当前正在从内存中恢复数据，则直接返回0；

然后以毫秒形式，取得当前时间戳now；如果本进程当前运行在slave节点上，则立即返回该key是否超时，而不采取任何动作，因为删除动作是由master节点发起并发送给slave的，slave不主动删除key；

如果该key未超时，则直接返回0；如果该key确实已经超时了，则进行删除动作，首先server.stat\_expiredkeys++；然后调用propagateExpire，将超时删除动作发送给AOF文件和slave节点；然后调用notifyKeyspaceEvent，发起expired键空间通知；最后，调用dbDelete删除该key。

定期删除的实现后续补充......

4：AOF、RDB和复制功能对过期键的处理

a：RDB功能处理过期键的策略如下：

执行SAVE命令或BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新的RDB文件中；

如果服务器开启了RDB功能，那么启动Redis服务器时，将会载入RDB文件。如果服务器以主服务器模式运行，则在载人RDB文件时，程序会对文件中保存的键进行检查，过期键则会被忽略而不被载入；

如果服务器以从服务器模式运行，则在载人RDB文件时，文件中保存的所有键，不论是否过期，都会被载人到数据库中。不过，因为主从服务器在进行数据同步时，从服务器的数据库就会被清空，所以一般来讲，过期键对载人RDB文件的从服务器不会造成影响。

b：AOF功能处理过期键的策略如下：

当服务器以AOF持久化模式运行时，如果数据库中的某个键已经过期，但它还没有被惰性删除或定期删除，则AOF文件不会因为这个过期键而产生任何影响；

当过期键被惰性删除或定期删除之后，程序会向AOF文件追加一条DEL命令，显式地记录该键已被删除。这是通过propagateExpire函数实现的，而该函数会在expireIfNeeded中被调用。

在执行AOF重写的过程中，程序会对数据库中的键进行检查，已过期的键不会被保存到重写后的AOF文件中。

c：复制功能处理过期键的策略如下：

当服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制：

主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个DEL命令，告知从服务器删除这个过期键。这是通过propagateExpire函数实现的，而该函数会在expireIfNeeded中被调用。

从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将过期键删除，而是继续像处理未过期的键一样来处理过期键。从服务器只有在接到主服务器发来的DEL命令之后，才会删除过期键。

通过由主服务器来控制从服务器统一地删除过期键，可以保证主从服务器数据的一致性。

5：expire类命令的实现

expire类命令，最终都是通过函数expireGenericCommand实现的，代码如下：

void expireGenericCommand**(**redisClient **\***c**,** long long basetime**,** int unit**)** **{**

robj **\***key **=** c**->**argv**[**1**],** **\***param **=** c**->**argv**[**2**];**

long long when**;** */\* unix time in milliseconds when the key will expire. \*/*

**if** **(**getLongLongFromObjectOrReply**(**c**,** param**,** **&**when**,** **NULL)** **!=** REDIS\_OK**)**

**return;**

**if** **(**unit **==** UNIT\_SECONDS**)** when **\*=** 1000**;**

when **+=** basetime**;**

*/\* No key, return zero. \*/*

**if** **(**lookupKeyRead**(**c**->**db**,**key**)** **==** **NULL)** **{**

addReply**(**c**,**shared**.**czero**);**

**return;**

**}**

*/\* EXPIRE with negative TTL, or EXPIREAT with a timestamp into the past*

*\* should never be executed as a DEL when load the AOF or in the context*

*\* of a slave instance.*

*\**

*\* Instead we take the other branch of the IF statement setting an expire*

*\* (possibly in the past) and wait for an explicit DEL from the master. \*/*

**if** **(**when **<=** mstime**()** **&&** **!**server**.**loading **&&** **!**server**.**masterhost**)** **{**

robj **\***aux**;**

redisAssertWithInfo**(**c**,**key**,**dbDelete**(**c**->**db**,**key**));**

server**.**dirty**++;**

*/\* Replicate/AOF this as an explicit DEL. \*/*

aux **=** createStringObject**(**"DEL"**,**3**);**

rewriteClientCommandVector**(**c**,**2**,**aux**,**key**);**

decrRefCount**(**aux**);**

signalModifiedKey**(**c**->**db**,**key**);**

notifyKeyspaceEvent**(**REDIS\_NOTIFY\_GENERIC**,**"del"**,**key**,**c**->**db**->**id**);**

addReply**(**c**,** shared**.**cone**);**

**return;**

**}** **else** **{**

setExpire**(**c**->**db**,**key**,**when**);**

addReply**(**c**,**shared**.**cone**);**

signalModifiedKey**(**c**->**db**,**key**);**

notifyKeyspaceEvent**(**REDIS\_NOTIFY\_GENERIC**,**"expire"**,**key**,**c**->**db**->**id**);**

server**.**dirty**++;**

**return;**

**}**

**}**

basetime表示key超时时间的基准值，实际的超时时间根据basetime,以及客户端的命令参数得到：如果该函数由expireat或者pexpireat调用，则basetime为0，而具体的超时时间由客户端在命令参数中提供；如果该函数有expire或pexpire调用，则basetime为当前的Unix时间戳，客户端在命令参数中提供一个相对值。最终得到实际的超时时间when；unit表示命令参数的时间单位，如果为UNIT\_SECONDS，则需要将when转换为毫秒；

首先在c->db中寻找key，找不到则直接反馈给客户端失败信息，并返回；

如果超时时间比当前时间要早，并且当前进程运行在slave节点上，并且没有进行数据恢复，则首先直接在c->db中删除该key，然后server.dirty++，然后调用rewriteClientCommandVector？然后调用signalModifiedKey表明该key被更新了，以使watch该key的客户端感知到；然后调用notifyKeyspaceEvent，发布"del"键空间通知；最后反馈给客户端后返回

如果不满足上述条件，则首先调用setExpire在c->db中设置key的超时时间；然后反馈给客户端；然后调用signalModifiedKey表明该key被更新了，以使watch该key的客户端感知到；然后调用notifyKeyspaceEvent，发布"expire"键空间通知；最后server.dirty++并返回。

6：ttl类命令的实现

ttl类(ttl、pttl)命令，最终都是通过函数ttlGenericCommand实现的，代码如下：

void ttlGenericCommand**(**redisClient **\***c**,** int output\_ms**)** **{**

long long expire**,** ttl **=** **-**1**;**

*/\* If the key does not exist at all, return -2 \*/*

**if** **(**lookupKeyRead**(**c**->**db**,**c**->**argv**[**1**])** **==** **NULL)** **{**

addReplyLongLong**(**c**,-**2**);**

**return;**

**}**

*/\* The key exists. Return -1 if it has no expire, or the actual*

*\* TTL value otherwise. \*/*

expire **=** getExpire**(**c**->**db**,**c**->**argv**[**1**]);**

**if** **(**expire **!=** **-**1**)** **{**

ttl **=** expire**-**mstime**();**

**if** **(**ttl **<** 0**)** ttl **=** 0**;**

**}**

**if** **(**ttl **==** **-**1**)** **{**

addReplyLongLong**(**c**,-**1**);**

**}** **else** **{**

addReplyLongLong**(**c**,**output\_ms **?** ttl **:** **((**ttl**+**500**)/**1000**));**

**}**

**}**

首先调用lookupKeyRead，在c->db中查找key，如果找不到，则直接返回-2给客户端；

然后调用getExpire，取得该key的超时时间expire，如果该key设置了超时时间，则计算ttl；如果该key没有设置超时时间，则直接返回-1给客户端；

如果output\_ms为1，则直接返回ttl给客户端；否则，需要将ttl转换为秒后在返回给客户端。

**四：键空间通知的实现**

键空间通知功能，主要是在src/notify.c中的notifyKeyspaceEvent函数实现的。每次需要发送键空间通知，都会调用该函数，比如删除一个键的函数delCommand：

void delCommand**(**redisClient **\***c**)** **{**

int deleted **=** 0**,** j**;**

**for** **(**j **=** 1**;** j **<** c**->**argc**;** j**++)** **{**

expireIfNeeded**(**c**->**db**,**c**->**argv**[**j**]);**

**if** **(**dbDelete**(**c**->**db**,**c**->**argv**[**j**]))** **{**

signalModifiedKey**(**c**->**db**,**c**->**argv**[**j**]);**

notifyKeyspaceEvent**(**REDIS\_NOTIFY\_GENERIC**,**

"del"**,**c**->**argv**[**j**],**c**->**db**->**id**);**

server**.**dirty**++;**

deleted**++;**

**}**

**}**

addReplyLongLong**(**c**,**deleted**);**

**}**

可见，每次成功删除一个键之后，都会调用notifyKeyspaceEvent发送键空间通知。

notifyKeyspaceEvent的代码如下：

void notifyKeyspaceEvent**(**int type**,** char **\***event**,** robj **\***key**,** int dbid**)** **{**

sds chan**;**

robj **\***chanobj**,** **\***eventobj**;**

int len **=** **-**1**;**

char buf**[**24**];**

*/\* If notifications for this class of events are off, return ASAP. \*/*

**if** **(!(**server**.**notify\_keyspace\_events **&** type**))** **return;**

eventobj **=** createStringObject**(**event**,**strlen**(**event**));**

*/\* \_\_keyspace@<db>\_\_:<key> <event> notifications. \*/*

**if** **(**server**.**notify\_keyspace\_events **&** REDIS\_NOTIFY\_KEYSPACE**)** **{**

chan **=** sdsnewlen**(**"\_\_keyspace@"**,**11**);**

len **=** ll2string**(**buf**,sizeof(**buf**),**dbid**);**

chan **=** sdscatlen**(**chan**,** buf**,** len**);**

chan **=** sdscatlen**(**chan**,** "\_\_:"**,** 3**);**

chan **=** sdscatsds**(**chan**,** key**->**ptr**);**

chanobj **=** createObject**(**REDIS\_STRING**,** chan**);**

pubsubPublishMessage**(**chanobj**,** eventobj**);**

decrRefCount**(**chanobj**);**

**}**

*/\* \_\_keyevente@<db>\_\_:<event> <key> notifications. \*/*

**if** **(**server**.**notify\_keyspace\_events **&** REDIS\_NOTIFY\_KEYEVENT**)** **{**

chan **=** sdsnewlen**(**"\_\_keyevent@"**,**11**);**

**if** **(**len **==** **-**1**)** len **=** ll2string**(**buf**,sizeof(**buf**),**dbid**);**

chan **=** sdscatlen**(**chan**,** buf**,** len**);**

chan **=** sdscatlen**(**chan**,** "\_\_:"**,** 3**);**

chan **=** sdscatsds**(**chan**,** eventobj**->**ptr**);**

chanobj **=** createObject**(**REDIS\_STRING**,** chan**);**

pubsubPublishMessage**(**chanobj**,** key**);**

decrRefCount**(**chanobj**);**

**}**

decrRefCount**(**eventobj**);**

**}**

参数type为键空间通知的类型，比如REDIS\_NOTIFY\_STRING等；event是字符串形式的事件，比如"del"等；key是相应的键，dbid是发生事件的数据库id。

首先通过server.notify\_keyspace\_events & type的值，判断服务器是否开启了该type类型的事件，也就是notify-keyspace-events选项的值是否包含type；然后根据参数event创建事件的字符串对象eventobj；

如果服务器开启了keyspace类型的事件通知，则该类型的事件，消息就是事件字符串对象eventobj，频道chanobj则类似于："\_\_keyspace@dbid\_\_:key"这样的字符串对象，频道和消息确定后，通过pubsubPublishMessage，将消息发送给所有的频道订阅者；

如果服务器开启了keyevent类型的事件通知，则该类型的事件，消息就是key字符串对象，频道chanobj则类似于："\_\_keyevent@dbid\_\_:event"这样的字符串对象，频道和消息确定后，通过pubsubPublishMessage，将消息发送给所有的频道订阅者。

更多redis数据库相关的实现，可以参考：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/db.c