15-为什么LRU算法原理和代码实现不一样?

你好，我是蒋德钧。

从这节课开始，我们就进入了课程的第三个模块：缓存模块。在接下来的三节课当中，我会给你详细介绍LRU、LFU算法在Redis源码中的实现，以及Redis惰性删除对缓存的影响。

学习这部分内容，一方面可以让你掌握这些经典缓存算法在一个实际系统中该如何设计和实现；另一方面，你也可以学习到在计算机系统设计实现中的一个重要原则，也就是在进行系统设计开发的过程中，需要均衡算法复杂度和实现复杂度。另外，你还可以学习到缓存替换、惰性删除是如何释放Redis内存的。内存资源对Redis来说是非常宝贵的，所以掌握了这一点，你就可以有效减少Redis的内存使用问题了。

好，那么今天这节课呢，我们就先来学习下LRU算法在Redis中的实现。

LRU算法的基本原理

首先，我们需要理解LRU算法的基本原理。LRU算法就是指**最近最少使用**（Least Recently Used，LRU）算法，这是一个经典的缓存算法。

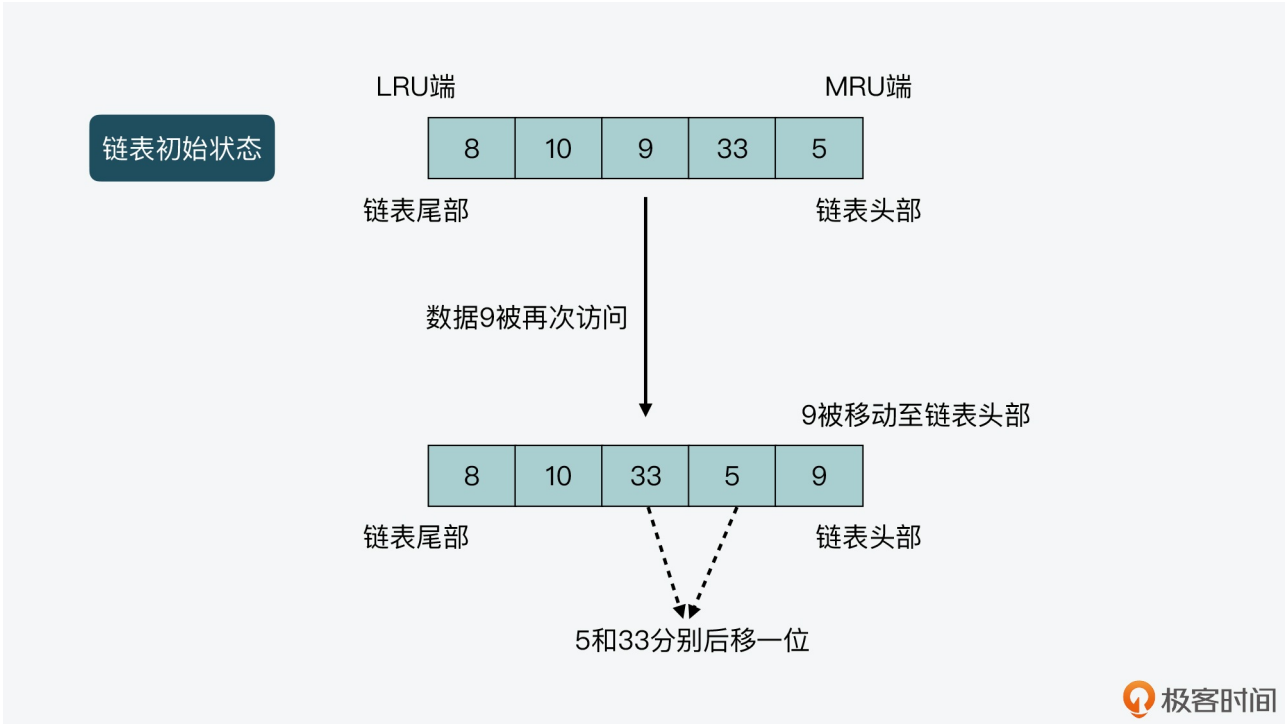
从基本原理上来说，LRU算法会使用一个链表来维护缓存中每一个数据的访问情况，并根据数据的实时访问，调整数据在链表中的位置，然后通过数据在链表中的位置，来表示数据是最近刚访问的，还是已经有一段时间没有访问了。

而具体来说，LRU算法会把链表的头部和尾部分别设置为MRU端和LRU端。其中，MRU是Most Recently Used的缩写，MRU端表示这里的数据是刚被访问的。而LRU端则表示，这里的数据是最近最少访问的数据。

我在第一季课程中曾介绍过LRU算法的执行过程，这里，我们来简要回顾下。LRU算法的执行，可以分成三种情况来掌握。

- **情况一**：当有新数据插入时，LRU算法会把该数据插入到链表头部，同时把原来链表头部的数据及其之后的数据，都向尾部移动一位。
- **情况二**：当有数据刚被访问了一次之后，LRU算法就会把该数据从它在链表中的当前位置，移动到链表头部。同时，把从链表头部到它当前位置的其他数据，都向尾部移动一位。
- **情况三**：当链表长度无法再容纳更多数据时，若再有新数据插入，LRU算法就会去除链表尾部的数据，这也相当于将数据从缓存中淘汰掉。

下图就展示了LRU算法执行过程的第二种情况，你可以看下。其中，链表长度为5，从链表头部到尾部保存的数据分别是5，33，9，10，8。假设数据9被访问了一次，那么9就会被移动到链表头部，同时，数据5和33都要向链表尾部移动一位。



所以你其实可以发现，如果要严格按照LRU算法的基本原理来实现的话，你需要在代码中实现如下内容：

- 要为Redis使用最大内存时，可容纳的所有数据维护一个链表；
- 每当有新数据插入或是现有数据被再次访问时，需要执行多次链表操作。

而假设Redis保存的数据比较多的话，那么，这两部分的代码实现，就既需要额外的内存空间来保存链表，还会在访问数据的过程中，让Redis受到数据移动和链表操作的开销影响，从而就会降低Redis访问性能。

所以说，无论是为了节省宝贵的内存空间，还是为了保持Redis高性能，Redis源码并没有严格按照LRU算法基本原理来实现它，而是**提供了一个近似LRU算法的实现**。

那么接下来，我们就来了解下这种近似LRU算法究竟是如何实现的。

Redis中近似LRU算法的实现

不过，在了解Redis对近似LRU算法的实现之前，我们需要先来看下，Redis的内存淘汰机制是如何启用近似LRU算法的，这可以帮助我们了解和近似LRU算法相关的配置项。

实际上，这和Redis配置文件redis.conf中的两个配置参数有关：

- **maxmemory**，该配置项设定了Redis server可以使用的最大内存容量，一旦server使用的实际内存量超出该阈值时，server就会根据maxmemory-policy配置项定义的策略，执行内存淘汰操作；
- **maxmemory-policy**，该配置项设定了Redis server的内存淘汰策略，主要包括近似LRU算法、LFU算法、按TTL值淘汰和随机淘汰等几种算法。

所以，一旦我们设定了maxmemory选项，并且将maxmemory-policy配置为allkeys-lru或是volatile-lru时，近似LRU算法就被启用了。这里，你需要注意的是，allkeys-lru和volatile-lru都会使用近似LRU算法来淘汰数据，它们的区别在于：采用allkeys-lru策略淘汰数据时，它是在所有的键值对中筛选将被淘汰的数据；而采用volatile-lru策略淘汰数据时，它是在设置了过期时间的键值对中筛选将被淘汰的数据。

好，了解了如何启用近似LRU算法后，我们就来具体学习下Redis是如何实现近似LRU算法的。这里，为了便于你理解，我把Redis对近似LRU算法的实现分成了三个部分。

- **全局LRU时钟值的计算**：这部分包括，Redis源码为了实现近似LRU算法的效果，是如何计算全局LRU时钟值的，以用来判断数据访问的时效性；
- **键值对LRU时钟值的初始化与更新**：这部分包括，Redis源码在哪些函数中对每个键值对对应的LRU时钟值，进行初始化与更新；
- **近似LRU算法的实际执行**：这部分包括，Redis源码具体如何执行近似LRU算法，也就是何时触发数据淘汰，以及实际淘汰的机制是怎么实现的。

那么下面，我们就先来看下全局LRU时钟值的计算。

全局LRU时钟值的计算

虽然Redis使用了近似LRU算法，但是，这个算法仍然**需要区分不同数据的访问时效性**，也就是说，Redis需要知道数据的最近一次访问时间。因此，Redis就设计了LRU时钟来记录数据每次访问的时间戳。

我们在第4讲中已经了解到，Redis在源码中对于每个键值对中的值，会使用一个redisObject结构体来保存指向值的指针。那么，redisObject结构体除了记录值的指针以外，它其实还会使用24 bits来保存LRU时钟信息，对应的是lru成员变量。所以这样一来，每个键值对都会把它最近一次被访问的时间戳，记录在lru变量当中。

redisOjbect结构体的定义是在server.h中，其中就包含了lru成员变量的定义，你可以看下。

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS;  //记录LRU信息，宏定义LRU_BITS是24 bits
    int refcount;
    void *ptr;
} robj;
```

那么，每个键值对的LRU时钟值具体是如何计算的呢？其实，Redis server使用了一个实例级别的全局LRU时钟，每个键值对的LRU时钟值会根据全局LRU时钟进行设置。

这个全局LRU时钟保存在了Redis全局变量server的成员变量**lruclock**中。当Redis server启动后，调用initServerConfig函数初始化各项参数时，就会对这个全局LRU时钟lruclock进行设置。具体来说，initServerConfig函数是调用getLRUClock函数，来设置lruclock的值，如下所示：

```
void initServerConfig(void) {
...
unsigned int lruclock = getLRUClock(); //调用getLRUClock函数计算全局LRU时钟值
atomicSet(server.lruclock,lruclock);//设置lruclock为刚计算的LRU时钟值
...
}
```

所以，**全局LRU时钟值就是通过getLRUClock函数计算得到的**。

getLRUClock函数是在evict.c文件中实现的，它会调用mstime函数（在server.c文件中）获得以毫秒为单位计算的UNIX时间戳，然后将这个UNIX时间戳除以宏定义LRU_CLOCK_RESOLUTION。宏定义LRU_CLOCK_RESOLUTION是在server.h文件中定义的，它表示的是以毫秒为单位的LRU时钟精度，也就是以毫秒为单位来表示的LRU时钟最小单位。

因为LRU_CLOCK_RESOLUTION的默认值是1000，所以，LRU时钟精度就是1000毫秒，也就是**1秒**。

这样一来，你需要注意的就是，**如果一个数据前后两次访问的时间间隔小于1秒，那么这两次访问的时间戳就是一样的**。因为LRU时钟的精度就是1秒，它无法区分间隔小于1秒的不同时间戳。

好了，了解了宏定义LRU_CLOCK_RESOLUTION的含义之后，我们再来看下getLRUClock函数中的计算。

首先，getLRUClock函数将获得的UNIX时间戳，除以LRU_CLOCK_RESOLUTION后，就得到了以LRU时钟精度来计算的UNIX时间戳，也就是当前的LRU时钟值。

紧接着，getLRUClock函数会把LRU时钟值和宏定义LRU_CLOCK_MAX做与运算，其中宏定义LRU_CLOCK_MAX表示的是LRU时钟能表示的最大值。

以下代码就展示了刚才介绍到的宏定义，以及getLRUClock函数的执行逻辑，你可以看下。

```
#define LRU_BITS 24
#define LRU_CLOCK_MAX ((1<<LRU_BITS)-1)  //LRU时钟的最大值
#define LRU_CLOCK_RESOLUTION 1000 //以毫秒为单位的LRU时钟精度

unsigned int getLRUClock(void) {
    return (mstime()/LRU_CLOCK_RESOLUTION) & LRU_CLOCK_MAX;
}
```

所以现在，你就知道了在默认情况下，全局LRU时钟值是以1秒为精度来计算的UNIX时间戳，并且它是在initServerConfig函数中进行了初始化。那么接下来，你可能还会困惑的问题是：**在Redis server的运行过程中，全局LRU时钟值是如何更新的呢？**

这就和Redis server在事件驱动框架中，定期运行的时间事件所对应的**serverCron函数**有关了。

serverCron函数作为时间事件的回调函数，本身会按照一定的频率周期性执行，其频率值是由Redis配置文件redis.conf中的**hz配置项**决定的。hz配置项的默认值是10，这表示serverCron函数会每100毫秒（1秒/10 = 100毫秒）运行一次。

这样，在serverCron函数中，全局LRU时钟值就会按照这个函数的执行频率，定期调用getLRUClock函数进行更新，如下所示：

```
int serverCron(struct aeEventLoop *eventLoop, long long id, void *clientData) {
...
unsigned long lruclock = getLRUClock(); //默认情况下，每100毫秒调用getLRUClock函数更新一次全局LRU时钟值
atomicSet(server.lruclock,lruclock); //设置lruclock变量
...
}
```

所以这样一来，每个键值对就可以从全局LRU时钟获取最新的访问时间戳了。

好，那么接下来，我们就来了解下，对于每个键值对来说，它对应的redisObject结构体中的lru变量，是在哪些函数中进行初始化和更新的。

键值对LRU时钟值的初始化与更新

首先，对于一个键值对来说，它的LRU时钟值最初是在这个键值对被创建的时候，进行初始化设置的，这个初始化操作是在**createObject函数**中调用的。createObject函数实现在object.c文件当中，当Redis要创建一个键值对时，就会调用这个函数。

而createObject函数除了会给redisObject结构体分配内存空间之外，它还会根据我刚才提到的maxmemory_policy配置项的值，来**初始化设置redisObject结构体中的lru变量**。

具体来说，就是如果maxmemory_policy配置为使用LFU策略，那么lru变量值会被初始化设置为LFU算法的计算值（关于LFU算法的代码实现，我会在下节课给你介绍）。而如果maxmemory_policy配置项没有使用LFU策略，那么，createObject函数就会调用LRU_CLOCK函数来设置lru变量的值，也就是键值对对应的LRU时钟值。

LRU_CLOCK函数是在evict.c文件中实现的，它的作用就是返回当前的全局LRU时钟值。因为一个键值对一旦被创建，也就相当于有了一次访问，所以它对应的LRU时钟值就表示了它的访问时间戳。

以下代码展示了这部分的执行逻辑，你可以看下。

```
robj *createObject(int type, void *ptr) {
    robj *o = zmalloc(sizeof(*o));
    ...
    //如果缓存替换策略是LFU，那么将lru变量设置为LFU的计数值
    if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
        o->lru = (LFUGetTimeInMinutes()<<8) | LFU_INIT_VAL;
    } else {
        o->lru = LRU_CLOCK();   //否则，调用LRU_CLOCK函数获取LRU时钟值
    }
    return o;
}
```

那么到这里，又出现了一个新的问题：**一个键值对的LRU时钟值又是在什么时候被再次更新的呢？**

其实，只要一个键值对被访问了，它的LRU时钟值就会被更新。而当一个键值对被访问时，访问操作最终都

会调用**lookupKey函数**。

lookupKey函数是在db.c文件中实现的，它会从全局哈希表中查找要访问的键值对。如果该键值对存在，那么lookupKey函数就会根据maxmemory_policy的配置值，来更新键值对的LRU时钟值，也就是它的访问时间戳。

而当maxmemory_policy没有配置为LFU策略时，lookupKey函数就会调用LRU_CLOCK函数，来获取当前的全局LRU时钟值，并将其赋值给键值对的redisObject结构体中的lru变量，如下所示：

```
robj *lookupKey(redisDb *db, robj *key, int flags) {
    dictEntry *de = dictFind(db->dict,key->ptr); //查找键值对
    if (de) {
        robj *val = dictGetVal(de); 获取键值对对应的redisObject结构体
        ...
        if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
                updateLFU(val);  //如果使用了LFU策略，更新LFU计数值
        } else {
                val->lru = LRU_CLOCK();  //否则，调用LRU_CLOCK函数获取全局LRU时钟值
        }
       ...
}}
```

这样一来，每个键值对一旦被访问，就能获得最新的访问时间戳了。不过现在，你可能要问了：这些访问时间戳最终是如何被用于近似LRU算法，来进行数据淘汰的呢？

接下来，我们就来学习下近似LRU算法的实际执行过程。

近似LRU算法的实际执行

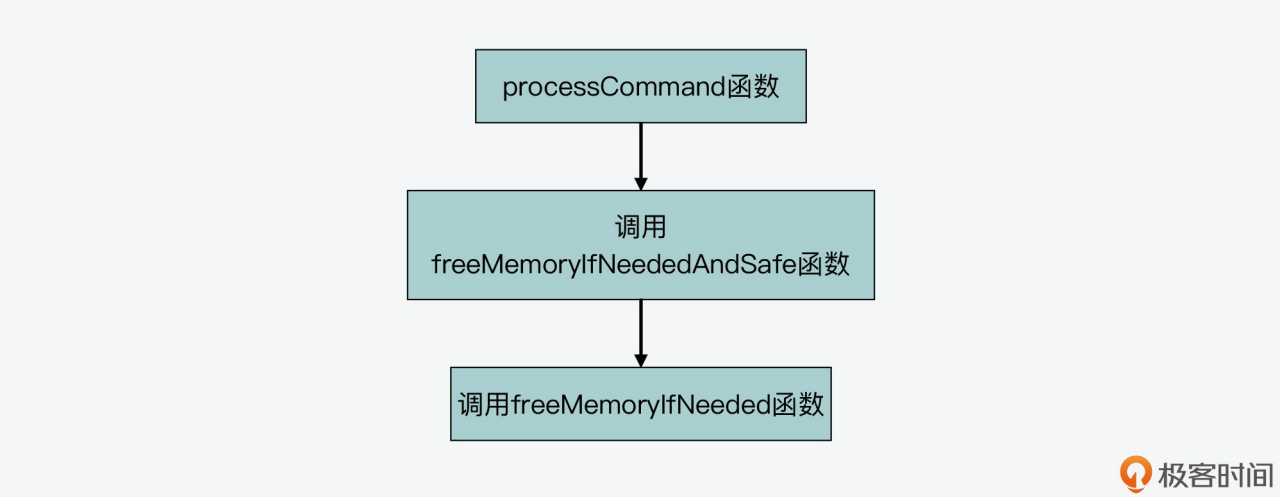
现在我们已经知道，Redis之所以实现近似LRU算法的目的，是为了减少内存资源和操作时间上的开销。那么在这里，我们其实可以从两个方面来了解近似LRU算法的执行过程，分别是：

- 何时触发算法执行？
- 算法具体如何执行？

何时触发算法执行？

首先，近似LRU算法的主要逻辑是在freeMemoryIfNeeded函数中实现的，而这个函数本身是在evict.c文件中实现。

freeMemoryIfNeeded函数是被freeMemoryIfNeededAndSafe函数（在evict.c文件中）调用，而freeMemoryIfNeededAndSafe函数又是被processCommand函数所调用的。你可以参考下面的图，展示了这三者的调用关系。



所以，我们看到processCommand函数，就应该知道这个函数是Redis处理每个命令时都会被调用的。我在第14讲中已经介绍过processCommand函数，你可以再去回顾下。

那么，processCommand函数在执行的时候，实际上会根据两个条件来判断是否调用freeMemoryIfNeededAndSafe函数。

- **条件一：设置了maxmemory配置项为非0值。**
- **条件二：Lua脚本没有在超时运行。**

如果这两个条件成立，那么processCommand函数就会调用freeMemoryIfNeededAndSafe函数，如下所示：

```
...
if (server.maxmemory && !server.lua_timedout) {
        int out_of_memory = freeMemoryIfNeededAndSafe() == C_ERR;
...
```

然后，freeMemoryIfNeededAndSafe函数还会再次根据两个条件，来判断是否调用freeMemoryIfNeeded函数。

- **条件一：Lua脚本在超时运行。**
- **条件二：Redis server正在加载数据。**

也就是说，只有在这两个条件都不成立的情况下，freeMemoryIfNeeded函数才会被调用。下面的代码展示了freeMemoryIfNeededAndSafe函数的执行逻辑，你可以看下。

```
int freeMemoryIfNeededAndSafe(void) {
    if (server.lua_timedout || server.loading) return C_OK;
    return freeMemoryIfNeeded();
}
```

这样，一旦freeMemoryIfNeeded函数被调用了，并且maxmemory-policy被设置为了allkeys-lru或volatile-lru，那么近似LRU算法就开始被触发执行了。接下来，我们就来看下近似LRU算法具体是如何执行的，也就是来了解freeMemoryIfNeeded函数的主要执行流程。

近似LRU算法具体如何执行？

近似LRU算法的执行可以分成三大步骤，分别是判断当前内存使用情况、更新待淘汰的候选键值对集合、选择被淘汰的键值对并删除。下面我们就依次来看下。

- **判断当前内存使用情况**

首先，freeMemoryIfNeeded函数会调用**getMaxmemoryState函数**，评估当前的内存使用情况。getMaxmemoryState函数是在evict.c文件中实现的，它会判断当前Redis server使用的内存容量是否超过了maxmemory配置的值。

如果当前内存使用量没有超过maxmemory，那么，getMaxmemoryState函数会返回C_OK，紧接着，freeMemoryIfNeeded函数也会直接返回了。

```
int freeMemoryIfNeeded(void) {
...
if (getMaxmemoryState(&mem_reported,NULL,&mem_tofree,NULL) == C_OK)
        return C_OK;
...
}
```

这里，**你需要注意的是**，getMaxmemoryState函数在评估当前内存使用情况的时候，如果发现已用内存超出了maxmemory，它就会计算需要释放的内存量。这个释放的内存大小等于已使用的内存量减去maxmemory。不过，已使用的内存量并不包括用于主从复制的复制缓冲区大小，这是getMaxmemoryState函数，通过调用freeMemoryGetNotCountedMemory函数来计算的。

我把getMaxmemoryState函数的基本执行逻辑代码放在这里，你可以看下。

```
int getMaxmemoryState(size_t *total, size_t *logical, size_t *tofree, float *level) {
...
mem_reported = zmalloc_used_memory(); //计算已使用的内存量
...
//将用于主从复制的复制缓冲区大小从已使用内存量中扣除
mem_used = mem_reported;
size_t overhead = freeMemoryGetNotCountedMemory();
mem_used = (mem_used > overhead) ? mem_used-overhead : 0;
...
//计算需要释放的内存量
mem_tofree = mem_used - server.maxmemory;
...
}
```

而如果当前server使用的内存量，的确已经超出maxmemory的上限了，那么freeMemoryIfNeeded函数

就会执行一个while循环，来淘汰数据释放内存。

其实，为了淘汰数据，Redis定义了一个数组EvictionPoolLRU，用来保存待淘汰的候选键值对。这个数组的元素类型是evictionPoolEntry结构体，该结构体保存了待淘汰键值对的空闲时间idle、对应的key等信息。以下代码展示了EvictionPoolLRU数组和evictionPoolEntry结构体，它们都是在evict.c文件中定义的。

```
static struct evictionPoolEntry *EvictionPoolLRU;

struct evictionPoolEntry {
    unsigned long long idle;    //待淘汰的键值对的空闲时间
    sds key;                    //待淘汰的键值对的key
    sds cached;                 //缓存的SDS对象
    int dbid;                   //待淘汰键值对的key所在的数据库ID
};
```

这样，Redis server在执行initSever函数进行初始化时，会调用evictionPoolAlloc函数（在evict.c文件中）为EvictionPoolLRU数组分配内存空间，该数组的大小由宏定义EVPOOL_SIZE（在evict.c文件中）决定，默认是16个元素，也就是可以保存16个待淘汰的候选键值对。

那么，freeMemoryIfNeeded函数在淘汰数据的循环流程中，就会更新这个待淘汰的候选键值对集合，也就是EvictionPoolLRU数组。下面我就来给你具体介绍一下。

- **更新待淘汰的候选键值对集合**

首先，freeMemoryIfNeeded函数会调用**evictionPoolPopulate函数**（在evict.c文件中），而evictionPoolPopulate函数会先调用dictGetSomeKeys函数（在dict.c文件中），从待采样的哈希表中随机获取一定数量的key。不过，这里还有两个地方你需要注意下。

第一点，dictGetSomeKeys函数采样的哈希表，是由maxmemory_policy配置项来决定的。如果maxmemory_policy配置的是allkeys_lru，那么待采样哈希表就是Redis server的全局哈希表，也就是在所有键值对中进行采样；否则，待采样哈希表就是保存着设置了过期时间的key的哈希表。

以下代码是freeMemoryIfNeeded函数中对evictionPoolPopulate函数的调用过程，你可以看下。

```
for (i = 0; i < server.dbnum; i++) {
       db = server.db+i;   //对Redis server上的每一个数据库都执行
       //根据淘汰策略，决定使用全局哈希表还是设置了过期时间的key的哈希表
       dict = (server.maxmemory_policy & MAXMEMORY_FLAG_ALLKEYS) ? db->dict : db->expires;
       if ((keys = dictSize(dict)) != 0) {
           //将选择的哈希表dict传入evictionPoolPopulate函数，同时将全局哈希表也传给evictionPoolPopulate函数
           evictionPoolPopulate(i, dict, db->dict, pool);
           total_keys += keys;
        }
 }
```

第二点，dictGetSomeKeys函数采样的key的数量，是由redis.conf中的配置项maxmemory-samples决定

的，该配置项的默认值是5。下面代码就展示了evictionPoolPopulate函数对dictGetSomeKeys函数的调用：

```
void evictionPoolPopulate(int dbid, dict *sampledict, dict *keydict, struct evictionPoolEntry *pool) {
    ...
    dictEntry *samples[server.maxmemory_samples];  //采样后的集合，大小为maxmemory_samples
    //将待采样的哈希表sampledict、采样后的集合samples、以及采样数量maxmemory_samples，作为参数传给dictGetSomeKeys
    count = dictGetSomeKeys(sampledict,samples,server.maxmemory_samples);
    ...
}
```

如此一来，dictGetSomeKeys函数就能返回采样的键值对集合了。然后，evictionPoolPopulate函数会根据实际采样到的键值对数量count，执行一个循环。

在这个循环流程中，evictionPoolPopulate函数会调用estimateObjectIdleTime函数，来计算在采样集合中的每一个键值对的空闲时间，如下所示：

```
for (j = 0; j < count; j++) {
...
if (server.maxmemory_policy & MAXMEMORY_FLAG_LRU) {
            idle = estimateObjectIdleTime(o);
}
...
```

紧接着，evictionPoolPopulate函数会遍历待淘汰的候选键值对集合，也就是EvictionPoolLRU数组。在遍历过程中，它会尝试把采样的每一个键值对插入EvictionPoolLRU数组，这主要取决于以下两个条件之一：

- 一是，它能在数组中找到一个尚未插入键值对的空位；
- 二是，它能在数组中找到一个空闲时间小于采样键值对空闲时间的键值对。

这两个条件有一个成立的话，evictionPoolPopulate函数就可以把采样键值对插入EvictionPoolLRU数组。等所有采样键值对都处理完后，evictionPoolPopulate函数就完成对待淘汰候选键值对集合的更新了。

接下来，freeMemoryIfNeeded函数，就可以开始选择最终被淘汰的键值对了。

- **选择被淘汰的键值对并删除**

因为evictionPoolPopulate函数已经更新了EvictionPoolLRU数组，而且这个数组里面的key，是按照空闲时间从小到大排好序了。所以，freeMemoryIfNeeded函数会遍历一次EvictionPoolLRU数组，从数组的最后一个key开始选择，如果选到的key不是空值，那么就把它作为最终淘汰的key。

这个过程的基本执行逻辑如下所示：

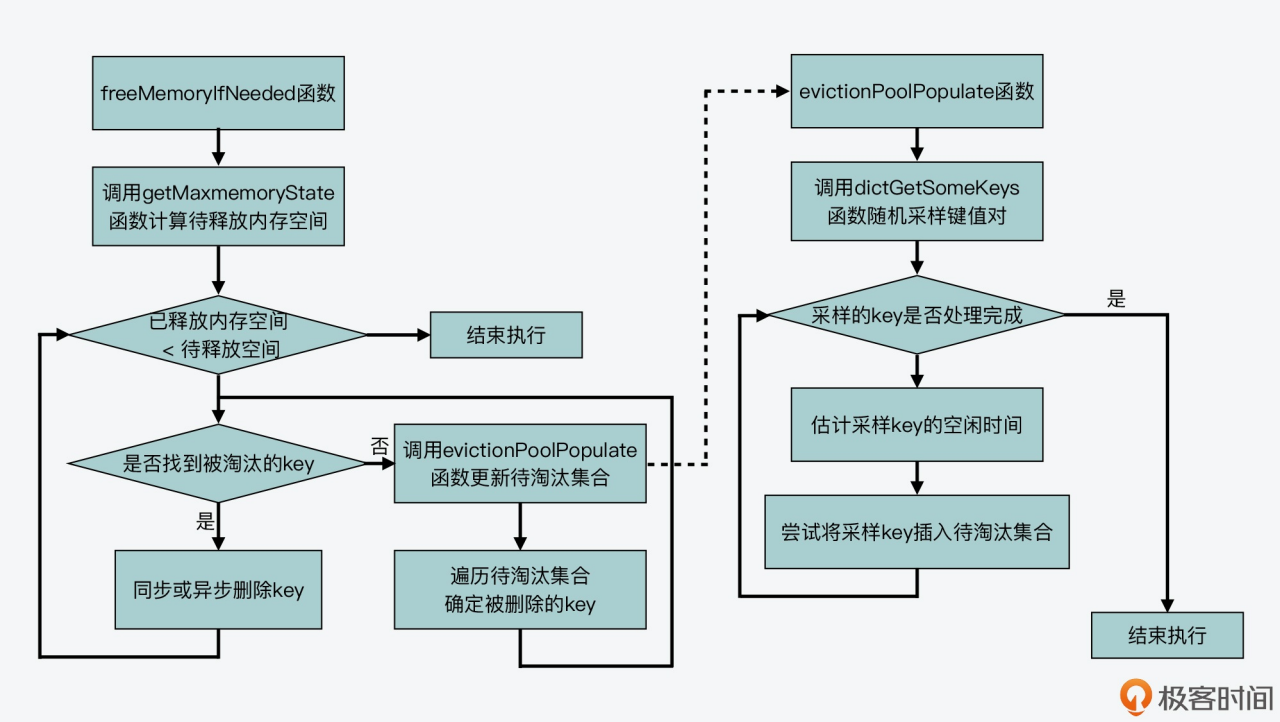
```
for (k = EVPOOL_SIZE-1; k >= 0; k--) { //从数组最后一个key开始查找
   if (pool[k].key == NULL) continue; //当前key为空值，则查找下一个key

   ... //从全局哈希表或是expire哈希表中，获取当前key对应的键值对；并将当前key从EvictionPoolLRU数组删除

    //如果当前key对应的键值对不为空，选择当前key为被淘汰的key
    if (de) {
      bestkey = dictGetKey(de);
      break;
     } else {} //否则，继续查找下个key
}
```

最后，一旦选到了被淘汰的key，freeMemoryIfNeeded函数就会根据Redis server的惰性删除配置，来执行同步删除或异步删除，如下所示：

```
if (bestkey) {
            db = server.db+bestdbid;
            robj *keyobj = createStringObject(bestkey,sdslen(bestkey));        //将删除key的信息传递给从库和AOF
            propagateExpire(db,keyobj,server.lazyfree_lazy_eviction);
            //如果配置了惰性删除，则进行异步删除
            if (server.lazyfree_lazy_eviction)
                dbAsyncDelete(db,keyobj);
            else  //否则进行同步删除
                dbSyncDelete(db,keyobj);
}
```

好了，到这里，freeMemoryIfNeeded函数就淘汰了一个key。而如果此时，释放的内存空间还不够，也就是说没有达到我前面介绍的待释放空间，那么freeMemoryIfNeeded函数还会**重复执行**前面所说的更新待淘汰候选键值对集合、选择最终淘汰key的过程，直到满足待释放空间的大小要求。

下图就展示了freeMemoryIfNeeded函数涉及的基本流程，你可以再来整体回顾下。



其实，从刚才介绍的内容中，你就可以看到，近似LRU算法并没有使用耗时耗空间的链表，而是使用了**固定大小的待淘汰数据集合**，每次随机选择一些key加入待淘汰数据集合中。最后，再按照待淘汰集合中key的空闲时间长度，删除空闲时间最长的key。这样一来，Redis就近似实现了LRU算法的效果了。

小结

好了，今天这节课就到这里，我们来总结下。

今天这节课我给你介绍了Redis中，是如何实现LRU算法来进行缓存数据替换的。其中，我们根据LRU算法的基本原理，可以发现如果严格按照原理来实现LRU算法，那么开发的系统就需要用额外的内存空间来保存LRU链表，而且系统运行时也会受到LRU链表操作的开销影响。

而对于Redis来说，内存资源和性能都很重要，所以Redis实现了近似LRU算法。而为了实现近似LRU算法，Redis首先是设置了**全局LRU时钟**，并在键值对创建时获取全局LRU时钟值作为访问时间戳，以及在每次访问时获取全局LRU时钟值，更新访问时间戳。

然后，当Redis每处理一个命令时，都会**调用freeMemoryIfNeeded函数**来判断是否需要释放内存。如果已使用内存超出了maxmemory，那么，近似LRU算法就会随机选择一些键值对，组成待淘汰候选集合，并根据它们的访问时间戳，选出最旧的数据，将其淘汰。

实际上，通过学习这节课的内容，你可以体会到一个算法的基本原理和算法的实际执行，在系统开发中会有一定的折中选择，主要就是因为我们需要综合考虑所开发的系统，在资源和性能方面的要求，以避免严格按照算法实现带来的资源和性能开销。因此，这一点就是你在进行计算机系统开发时，要秉承的一个原则。

每课一问

现在你已经知道，Redis源码中提供了getLRUClock函数来计算全局LRU时钟值，同时键值对的LRU时钟值是通过LRU_CLOCK函数来获取的，以下代码就展示了LRU_CLOCK函数的执行逻辑。这个函数包括两个分支，一个分支是直接从全局变量server的lruclock中获取全局时钟值，另一个是调用getLRUClock函数获取全局时钟值。

那么你知道，为什么键值对的LRU时钟值，不是直接通过调用getLRUClock函数来获取的呢？

```
unsigned int LRU_CLOCK(void) {
    unsigned int lruclock;
    if (1000/server.hz <= LRU_CLOCK_RESOLUTION) {
        atomicGet(server.lruclock,lruclock);
    } else {
        lruclock = getLRUClock();
    }
    return lruclock;
}
```

欢迎在留言区分享你的答案和思考过程，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

精选留言：

- Kaito 2021-08-28 01:40:28

  1、实现一个严格的 LRU 算法，需要额外的内存构建 LRU 链表，同时维护链表也存在性能开销，Redis 对于内存资源和性能要求极高，所以没有采用严格 LRU 算法，而是采用「近似」LRU 算法实现数据淘汰策略

  2、触发数据淘汰的时机，是每次处理「请求」时判断的。也就是说，执行一个命令之前，首先要判断实例内存是否达到 maxmemory，是的话则先执行数据淘汰，再执行具体的命令

  3、淘汰数据时，会「持续」判断 Redis 内存是否下降到了 maxmemory 以下，不满足的话会继续淘汰数据，直到内存下降到 maxmemory 之下才会停止

  4、可见，如果发生大量淘汰的情况，那么处理客户端请求就会发生「延迟」，影响性能

  5、Redis 计算实例内存时，不会把「主从复制」的缓冲区计算在内，也就是说不管一个实例后面挂了多少个从库，主库不会把主从复制所需的「缓冲区」内存，计算到实例内存中，即这部分内存增加，不会对数据淘汰产生影响

  6、但如果 Redis 内存已达到 maxmemory，要谨慎执行 MONITOR 命令，因为 Redis Server 会向执行 MONITOR 的 client 缓冲区填充数据，这会导致缓冲区内存增长，进而引发数据淘汰

  课后题：为什么键值对的 LRU 时钟值，不是直接通过调用 getLRUClock 函数来获取？

  本质上是为了性能。

  Redis 这种对性能要求极高的数据库，在系统调用上的优化也做到了极致。

  获取机器时钟本质上也是一个「系统调用」，对于 Redis 这种动不动每秒上万的 QPS，如果每次都触发一次系统调用，这么频繁的操作也是一笔不小的开销。

  所以，Redis 用一个定时任务（serverCron 函数），以固定频率触发系统调用获取机器时钟，然后把机器时钟挂到 server 的全局变量下，这相当于维护了一个「本地缓存」，当需要获取时钟时，直接从全局变量获取即可，节省了大量的系统调用开销。 [1赞]