lru

LRU 最近最少使用淘汰算法

简介

Redis中的LRU淘汰策略分析 Redis作为缓存使用时，一些场景下要考虑内存的空间消耗问题Redis会删除过期键以释放空间，过期键的删除策略有两种：

惰性删除：每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期，就返回该键。 定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。 另外Redis也可以开启LRU功能来自动淘汰一些键值对。

LRU算法 当需要从缓存中淘汰数据时，我们希望能淘汰那些将来不可能再被使用的数据，保留那些将来还会频繁访问的数据，但最大的问题是缓存并不能预言未来。一个解决方法就是通过LRU进行预测：最近被频繁访问的数据将来被访问的可能性也越大。缓存中的数据一般会有这样的访问分布：一部分数据拥有绝大部分的访问量。当访问模式很少改变时，可以记录每个数据的最后一次访问时间，拥有最少空闲时间的数据可以被认为将来最有可能被访问到。

举例如下的访问模式A每5s访问一次B每2s访问一次C与D每10s访问一次，|代表计算空闲时间的截止点：

```
~~~~~A~~~~~A~~~~~A~~~~A~~~~~A~~~~~A~~|
~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~|
~~~~~~~~~~C~~~~~~~~~C~~~~~~~~~C~~~~~~|
~~~~~D~~~~~~~~~~D~~~~~~~~~D~~~~~~~~~D|
```

可以看到LRU对于A B C工作的很好，完美预测了将来被访问到的概率B>A>C但对于D却预测了最少的空闲时间。

但是，总体来说LRU算法已经是一个性能足够好的算法了

LRU配置参数 Redis配置中和LRU有关的有三个：

```
maxmemory: 配置Redis存储数据时指定限制的内存大小，比如100m当缓存消耗的内存超过这个数值时, 将触发数据淘汰。该数据配置为0时，表示缓存的数据量没有限制, 即LRU功能不生效。64位的系统默认值为0，32位的系统默认内存限制为3GB
maxmemory_policy: 触发数据淘汰后的淘汰策略
maxmemory_samples: 随机采样的精度，也就是随即取出key的数目。该数值配置越大, 越接近于真实的LRU算法，但是数值越大，相应消耗也变高，对性能有一定影响，样本值默认为5。
```

淘汰策略 淘汰策略即maxmemory_policy的赋值有以下几种：

```
noeviction:如果缓存数据超过了maxmemory限定值，并且客户端正在执行的命令(大部分的写入指令，但DEL和几个指令例外)会导致内存分配，则向客户端返回错误响应
allkeys-lru: 对所有的键都采取LRU淘汰
volatile-lru: 仅对设置了过期时间的键采取LRU淘汰
allkeys-random: 随机回收所有的键
volatile-random: 随机回收设置过期时间的键
volatile-ttl: 仅淘汰设置了过期时间的键---淘汰生存时间TTL(Time To Live)更小的键
volatile-lru, volatile-random和volatile-ttl这三个淘汰策略使用的不是全量数据，有可能无法淘汰出足够的内存空间。在没有过期键或者没有设置超时属性的键的情况下，这三种策略和noeviction差不多。
```

一般的经验规则:

使用allkeys-lru策略：当预期请求符合一个幂次分布(二八法则等)，比如一部分的子集元素比其它其它元素被访问的更多时，可以选择这个策略。
使用allkeys-random，循环连续的访问所有的键时，或者预期请求分布平均（所有元素被访问的概率都差不多）
使用volatile-ttl，要采取这个策略，缓存对象的TTL值最好有差异
volatile-lru 和 volatile-random策略，当你想要使用单一的Redis实例来同时实现缓存淘汰和持久化一些经常使用的键集合时很有用。未设置过期时间的键进行持久化保存，设置了过期时间的键参与缓存淘汰。不过一般运行两个实例是解决这个问题的更好方法。

为键设置过期时间也是需要消耗内存的，所以使用allkeys-lru这种策略更加节省空间，因为这种策略下可以不为键设置过期时间。

近似LRU算法 我们知道，LRU算法需要一个双向链表来记录数据的最近被访问顺序，但是出于节省内存的考虑，Redis的LRU算法并非完整的实现，Redis并不会选择最久未被访问的键进行回收，相反它会尝试运行一个近似LRU的算法，通过对少量键进行取样，然后回收其中的最久未被访问的键。通过调整每次回收时的采样数量maxmemory-samples，可以实现调整算法的精度。

根据Redis作者的说法，每个Redis Object可以挤出24 bits的空间，但24 bits是不够存储两个指针的，而存储一个低位时间戳是足够的，Redis Object以秒为单位存储了对象新建或者更新时的unix time，也就是LRU clock，24 bits数据要溢出的话需要194天，而缓存的数据更新非常频繁，已经足够了。

Redis的键空间是放在一个哈希表中的，要从所有的键中选出一个最久未被访问的键，需要另外一个数据结构存储这些源信息，这显然不划算。最初，Redis只是随机的选3个key，然后从中淘汰，后来算法改进到了N个key的策略，默认是5个。

Redis3.0之后又改善了算法的性能，会提供一个待淘汰候选key的pool，里面默认有16个key，按照空闲时间排好序。更新时从Redis键空间随机选择N个key，分别计算它们的空闲时间idle，key只会在pool不满或者空闲时间大于pool里最小的时，才会进入pool，然后从pool中选择空闲时间最大的key淘汰掉。

真实LRU算法与近似LRU的算法可以通过下面的图像对比：

浅灰色带是已经被淘汰的对象，灰色带是没有被淘汰的对象，绿色带是新添加的对象。可以看出，maxmemory-samples值为5时Redis 3.0效果比Redis 2.8要好。使用10个采样大小的Redis 3.0的近似LRU算法已经非常接近理论的性能了。

数据访问模式非常接近幂次分布时，也就是大部分的访问集中于部分键时，LRU近似算法会处理得很好。

在模拟实验的过程中，我们发现如果使用幂次分布的访问模式，真实LRU算法和近似LRU算法几乎没有差别。

LRU源码分析 Redis中的键与值都是redisObject对象：

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                            * LFU data (least significant 8 bits frequency
                            * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

unsigned的低24 bits的lru记录了redisObj的LRU time

Redis命令访问缓存的数据时,均会调用函数lookupKey:

```
robj *lookupKey(redisDb *db, robj *key, int flags) {
    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 &&
            !(flags & LOOKUP_NOTOUCH))
        {
            if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
                updateLFU(val);
            } else {
                val->lru = LRU_CLOCK();
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

该函数在策略为LRU(非LFU)时会更新对象的lru值, 设置为LRU_CLOCK()值:

```
/* Return the LRU clock, based on the clock resolution. This is a time
 * in a reduced-bits format that can be used to set and check the
 * object->lru field of redisObject structures. */
unsigned int getLRUClock(void) {
    return (mstime()/LRU_CLOCK_RESOLUTION) & LRU_CLOCK_MAX;
}

/* This function is used to obtain the current LRU clock.
 * If the current resolution is lower than the frequency we refresh the
 * LRU clock (as it should be in production servers) we return the
 * precomputed value, otherwise we need to resort to a system call. */
unsigned int LRU_CLOCK(void) {
    unsigned int lruclock;
    if (1000/server.hz <= LRU_CLOCK_RESOLUTION) {
        atomicGet(server.lruclock,lruclock);
    } else {
        lruclock = getLRUClock();
    }
    return lruclock;
}
```

LRU_CLOCK()取决于LRU_CLOCK_RESOLUTION(默认值1000)，LRU_CLOCK_RESOLUTION代表了LRU算法的精度，即一个LRU的单位是多长，server.hz代表服务器刷新的频率，如果服务器的时间更新精度值比LRU的精度值要小，LRU_CLOCK()直接使用服务器的时间，减小开销。

Redis处理命令的入口是processCommand:

```
int processCommand(client *c) {

    /* Handle the maxmemory directive.
     *
     * Note that we do not want to reclaim memory if we are here re-entering
     * the event loop since there is a busy Lua script running in timeout
     * condition, to avoid mixing the propagation of scripts with the
     * propagation of DELs due to eviction. */
    if (server.maxmemory && !server.lua_timedout) {
        int out_of_memory = freeMemoryIfNeededAndSafe() == C_ERR;
        /* freeMemoryIfNeeded may flush slave output buffers. This may
result
         * into a slave, that may be the active client, to be freed. */
        if (server.current_client == NULL) return C_ERR;

        /* It was impossible to free enough memory, and the command the
client
         * is trying to execute is denied during OOM conditions or the
client
         * is in MULTI/EXEC context? Error. */
        if (out_of_memory &&
            (c->cmd->flags & CMD_DENYOOM ||
             (c->flags & CLIENT_MULTI && c->cmd->proc != execCommand))) {
            flagTransaction(c);
            addReply(c, shared.oomerr);
            return C_OK;
        }
    }
}
```

只列出了释放内存空间的部分，freeMemoryIfNeededAndSafe为释放内存的函数：

```
int freeMemoryIfNeeded(void) {
    /* By default replicas should ignore maxmemory
     * and just be masters exact copies. */
    if (server.masterhost && server.repl_slave_ignore_maxmemory) return
C_OK;

    size_t mem_reported, mem_tofree, mem_freed;
    mstime_t latency, eviction_latency;
    long long delta;
    int slaves = listLength(server.slaves);

    /* When clients are paused the dataset should be static not just from
the
```

```
     * POV of clients not being able to write, but also from the POV of
     * expires and evictions of keys not being performed. */
    if (clientsArePaused()) return C_OK;
    if (getMaxmemoryState(&mem_reported,NULL,&mem_tofree,NULL) == C_OK)
        return C_OK;

    mem_freed = 0;

    if (server.maxmemory_policy == MAXMEMORY_NO_EVICTION)
        goto cant_free; /* We need to free memory, but policy forbids. */

    latencyStartMonitor(latency);
    while (mem_freed < mem_tofree) {
        int j, k, i, keys_freed = 0;
        static unsigned int next_db = 0;
        sds bestkey = NULL;
        int bestdbid;
        redisDb *db;
        dict *dict;
        dictEntry *de;

        if (server.maxmemory_policy &
(MAXMEMORY_FLAG_LRU|MAXMEMORY_FLAG_LFU) ||
            server.maxmemory_policy == MAXMEMORY_VOLATILE_TTL)
        {
            struct evictionPoolEntry *pool = EvictionPoolLRU;

            while(bestkey == NULL) {
                unsigned long total_keys = 0, keys;

                /* We don't want to make local-db choices when expiring
keys,
                 * so to start populate the eviction pool sampling keys from
                 * every DB. */
                for (i = 0; i < server.dbnum; i++) {
                    db = server.db+i;
                    dict = (server.maxmemory_policy &
MAXMEMORY_FLAG_ALLKEYS) ?
                            db->dict : db->expires;
                    if ((keys = dictSize(dict)) != 0) {
                        evictionPoolPopulate(i, dict, db->dict, pool);
                        total_keys += keys;
                    }
                }
                if (!total_keys) break; /* No keys to evict. */

                /* Go backward from best to worst element to evict. */
                for (k = EVPOOL_SIZE-1; k >= 0; k--) {
                    if (pool[k].key == NULL) continue;
                    bestdbid = pool[k].dbid;
```

```
                    if (server.maxmemory_policy & MAXMEMORY_FLAG_ALLKEYS) {
                        de = dictFind(server.db[pool[k].dbid].dict,
                            pool[k].key);
                    } else {
                        de = dictFind(server.db[pool[k].dbid].expires,
                            pool[k].key);
                    }

                    /* Remove the entry from the pool. */
                    if (pool[k].key != pool[k].cached)
                        sdsfree(pool[k].key);
                    pool[k].key = NULL;
                    pool[k].idle = 0;

                    /* If the key exists, is our pick. Otherwise it is
                     * a ghost and we need to try the next element. */
                    if (de) {
                        bestkey = dictGetKey(de);
                        break;
                    } else {
                        /* Ghost... Iterate again. */
                    }
                }
            }
        }

        /* volatile-random and allkeys-random policy */
        else if (server.maxmemory_policy == MAXMEMORY_ALLKEYS_RANDOM ||
                 server.maxmemory_policy == MAXMEMORY_VOLATILE_RANDOM)
        {
            /* When evicting a random key, we try to evict a key for
             * each DB, so we use the static 'next_db' variable to
             * incrementally visit all DBs. */
            for (i = 0; i < server.dbnum; i++) {
                j = (++next_db) % server.dbnum;
                db = server.db+j;
                dict = (server.maxmemory_policy == MAXMEMORY_ALLKEYS_RANDOM)
?
                        db->dict : db->expires;
                if (dictSize(dict) != 0) {
                    de = dictGetRandomKey(dict);
                    bestkey = dictGetKey(de);
                    bestdbid = j;
                    break;
                }
            }
        }

        /* Finally remove the selected key. */
        if (bestkey) {
            db = server.db+bestdbid;
```

```
            robj *keyobj = createStringObject(bestkey,sdslen(bestkey));
            propagateExpire(db,keyobj,server.lazyfree_lazy_eviction);
            /* We compute the amount of memory freed by db*Delete() alone.
             * It is possible that actually the memory needed to propagate
             * the DEL in AOF and replication link is greater than the one
             * we are freeing removing the key, but we can't account for
             * that otherwise we would never exit the loop.
             *
             * AOF and Output buffer memory will be freed eventually so
             * we only care about memory used by the key space. */
            delta = (long long) zmalloc_used_memory();
            latencyStartMonitor(eviction_latency);
            if (server.lazyfree_lazy_eviction)
                dbAsyncDelete(db,keyobj);
            else
                dbSyncDelete(db,keyobj);
            latencyEndMonitor(eviction_latency);
            latencyAddSampleIfNeeded("eviction-del",eviction_latency);
            latencyRemoveNestedEvent(latency,eviction_latency);
            delta -= (long long) zmalloc_used_memory();
            mem_freed += delta;
            server.stat_evictedkeys++;
            notifyKeyspaceEvent(NOTIFY_EVICTED, "evicted",
                keyobj, db->id);
            decrRefCount(keyobj);
            keys_freed++;

            /* When the memory to free starts to be big enough, we may
             * start spending so much time here that is impossible to
             * deliver data to the slaves fast enough, so we force the
             * transmission here inside the loop. */
            if (slaves) flushSlavesOutputBuffers();

            /* Normally our stop condition is the ability to release
             * a fixed, pre-computed amount of memory. However when we
             * are deleting objects in another thread, it's better to
             * check, from time to time, if we already reached our target
             * memory, since the "mem_freed" amount is computed only
             * across the dbAsyncDelete() call, while the thread can
             * release the memory all the time. */
            if (server.lazyfree_lazy_eviction && !(keys_freed % 16)) {
                if (getMaxmemoryState(NULL,NULL,NULL,NULL) == C_OK) {
                    /* Let's satisfy our stop condition. */
                    mem_freed = mem_tofree;
                }
            }
        }

        if (!keys_freed) {
            latencyEndMonitor(latency);
            latencyAddSampleIfNeeded("eviction-cycle",latency);
```

```
            goto cant_free; /* nothing to free... */
        }
    }
    latencyEndMonitor(latency);
    latencyAddSampleIfNeeded("eviction-cycle",latency);
    return C_OK;

cant_free:
    /* We are here if we are not able to reclaim memory. There is only one
     * last thing we can try: check if the lazyfree thread has jobs in queue
     * and wait... */
    while(bioPendingJobsOfType(BIO_LAZY_FREE)) {
        if (((mem_reported - zmalloc_used_memory()) + mem_freed) >=
mem_tofree)
            break;
        usleep(1000);
    }
    return C_ERR;
}

/* This is a wrapper for freeMemoryIfNeeded() that only really calls the
 * function if right now there are the conditions to do so safely:
 *
 * - There must be no script in timeout condition.
 * - Nor we are loading data right now.
 *
 */
int freeMemoryIfNeededAndSafe(void) {
    if (server.lua_timedout || server.loading) return C_OK;
    return freeMemoryIfNeeded();
}
```

几种淘汰策略maxmemory_policy就是在这个函数里面实现的。

当采用LRU时，可以看到，从0号数据库开始(默认16个)，根据不同的策略，选择redisDb的dict(全部键)或者expires(有过期时间的键)，用来更新候选键池子pool□pool更新策略是evictionPoolPopulate:

```
void evictionPoolPopulate(int dbid, dict *sampledict, dict *keydict, struct
evictionPoolEntry *pool) {
    int j, k, count;
    dictEntry *samples[server.maxmemory_samples];

    count = dictGetSomeKeys(sampledict,samples,server.maxmemory_samples);
    for (j = 0; j < count; j++) {
        unsigned long long idle;
        sds key;
        robj *o;
        dictEntry *de;

        de = samples[j];
        key = dictGetKey(de);
```

```
        /* If the dictionary we are sampling from is not the main
         * dictionary (but the expires one) we need to lookup the key
         * again in the key dictionary to obtain the value object. */
        if (server.maxmemory_policy != MAXMEMORY_VOLATILE_TTL) {
            if (sampledict != keydict) de = dictFind(keydict, key);
            o = dictGetVal(de);
        }

        /* Calculate the idle time according to the policy. This is called
         * idle just because the code initially handled LRU, but is in fact
         * just a score where an higher score means better candidate. */
        if (server.maxmemory_policy & MAXMEMORY_FLAG_LRU) {
            idle = estimateObjectIdleTime(o);
        } else if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
            /* When we use an LRU policy, we sort the keys by idle time
             * so that we expire keys starting from greater idle time.
             * However when the policy is an LFU one, we have a frequency
             * estimation, and we want to evict keys with lower frequency
             * first. So inside the pool we put objects using the inverted
             * frequency subtracting the actual frequency to the maximum
             * frequency of 255. */
            idle = 255-LFUDecrAndReturn(o);
        } else if (server.maxmemory_policy == MAXMEMORY_VOLATILE_TTL) {
            /* In this case the sooner the expire the better. */
            idle = ULLONG_MAX - (long)dictGetVal(de);
        } else {
            serverPanic("Unknown eviction policy in
evictionPoolPopulate()");
        }

        /* Insert the element inside the pool.
         * First, find the first empty bucket or the first populated
         * bucket that has an idle time smaller than our idle time. */
        k = 0;
        while (k < EVPOOL_SIZE &&
               pool[k].key &&
               pool[k].idle < idle) k++;
        if (k == 0 && pool[EVPOOL_SIZE-1].key != NULL) {
            /* Can't insert if the element is < the worst element we have
             * and there are no empty buckets. */
            continue;
        } else if (k < EVPOOL_SIZE && pool[k].key == NULL) {
            /* Inserting into empty position. No setup needed before insert.
*/
        } else {
            /* Inserting in the middle. Now k points to the first element
             * greater than the element to insert.  */
            if (pool[EVPOOL_SIZE-1].key == NULL) {
                /* Free space on the right? Insert at k shifting
                 * all the elements from k to end to the right. */
```

```
                /* Save SDS before overwriting. */
                sds cached = pool[EVPOOL_SIZE-1].cached;
                memmove(pool+k+1,pool+k,
                    sizeof(pool[0])*(EVPOOL_SIZE-k-1));
                pool[k].cached = cached;
            } else {
                /* No free space on right? Insert at k-1 */
                k--;
                /* Shift all elements on the left of k (included) to the
                 * left, so we discard the element with smaller idle time.
*/
                sds cached = pool[0].cached; /* Save SDS before overwriting.
*/
                if (pool[0].key != pool[0].cached) sdsfree(pool[0].key);
                memmove(pool,pool+1,sizeof(pool[0])*k);
                pool[k].cached = cached;
            }
        }

        /* Try to reuse the cached SDS string allocated in the pool entry,
         * because allocating and deallocating this object is costly
         * (according to the profiler, not my fantasy. Remember:
         * premature optimizbla bla bla bla. */
        int klen = sdslen(key);
        if (klen > EVPOOL_CACHED_SDS_SIZE) {
            pool[k].key = sdsdup(key);
        } else {
            memcpy(pool[k].cached,key,klen+1);
            sdssetlen(pool[k].cached,klen);
            pool[k].key = pool[k].cached;
        }
        pool[k].idle = idle;
        pool[k].dbid = dbid;
    }
}
```

Redis随机选择maxmemory_samples数量的key，然后计算这些key的空闲时间idle time，当满足条件时(比pool中的某些键的空闲时间还大)就可以进pool，pool更新之后，就淘汰pool中空闲时间最大的键。

estimateObjectIdleTime用来计算Redis对象的空闲时间：

```
/* Given an object returns the min number of milliseconds the object was
never
 * requested, using an approximated LRU algorithm. */
unsigned long long estimateObjectIdleTime(robj *o) {
    unsigned long long lruclock = LRU_CLOCK();
    if (lruclock >= o->lru) {
        return (lruclock - o->lru) * LRU_CLOCK_RESOLUTION;
    } else {
        return (lruclock + (LRU_CLOCK_MAX - o->lru)) *
```

```
                    LRU_CLOCK_RESOLUTION;
    }
}
```

空闲时间基本就是就是对象的lru和全局的LRU_CLOCK()的差值乘以精度LRU_CLOCK_RESOLUTION，将秒转化为了毫秒。

参考链接

http://antirez.com/news/109

https://redis.io/topics/lru-cache

https://www.cnblogs.com/linxiyue/p/10945216.html