源码 9: 如履薄冰 —— 懒惰删除的巨大牺牲

前面我们讲了 Redis 懒惰删除的特性，它是使用异步线程对已删除的节点进行内存回收。但是还不够深入，所以本节我们要对异步线程逻辑处理的细节进行分析，看看 Antirez 是如何实现异步线程处理的。

异步线程在 Redis 内部有一个特别的名称，它就是BIO，全称是Background IO，意思是在背后默默干活的 IO 线程。不过内存回收本身并不是什么 IO 操作，只是 CPU 的计算消耗可能会比较大而已。

懒惰删除的最初实现不是异步线程

Antirez 实现懒惰删除时，它并不是一开始就想到了异步线程。最初的尝试是在主线程里，使用类似于字典渐进式搬迁那样来实现渐进式删除回收。比如对于一个非常大的字典来说，懒惰删除是采用类似于 scan 操作的方法，通过遍历第一维数组来逐步删除回收第二维链表的内容，等到所有链表都回收完了，再一次性回收第一维数组。这样也可以达到删除大对象时不阻塞主线程的效果。

但是说起来容易做起来却很难。渐进式回收需要仔细控制回收频率，它不能回收的太猛，这会导致 CPU 资源占用过多，也不能回收的蜗牛慢，因为内存回收不及时可能导致内存持续增长。

Antirez 需要采用合适的自适应算法来控制回收频率。他首先想到的是检测内存增长的趋势是增长 (+1) 还是下降 (-1) 来渐进式调整回收频率系数，这样的自适应算法实现也很简单。但是测试后发现在服

务繁忙的时候，QPS 会下降到正常情况下 65% 的水平，这点非常致命。

所以 Antirez 才使用了如今使用的方案——异步线程。异步线程这套方案就简单多了，释放内存不用为每种数据结构适配一套渐进式释放策略，也不用搞个自适应算法来仔细控制回收频率。将对象从全局字典中摘掉，然后往队列里一扔，主线程就干别的去了。异步线程从队列里取出对象来，直接走正常的同步释放逻辑就可以了。

不过使用异步线程也是有代价的，主线程和异步线程之间在内存回收器 (jemalloc) 的使用上存在竞争。这点竞争消耗是可以忽略不计的，因为 Redis 的主线程在内存的分配与回收上花的时间相对整体运算时间而言是极少的。

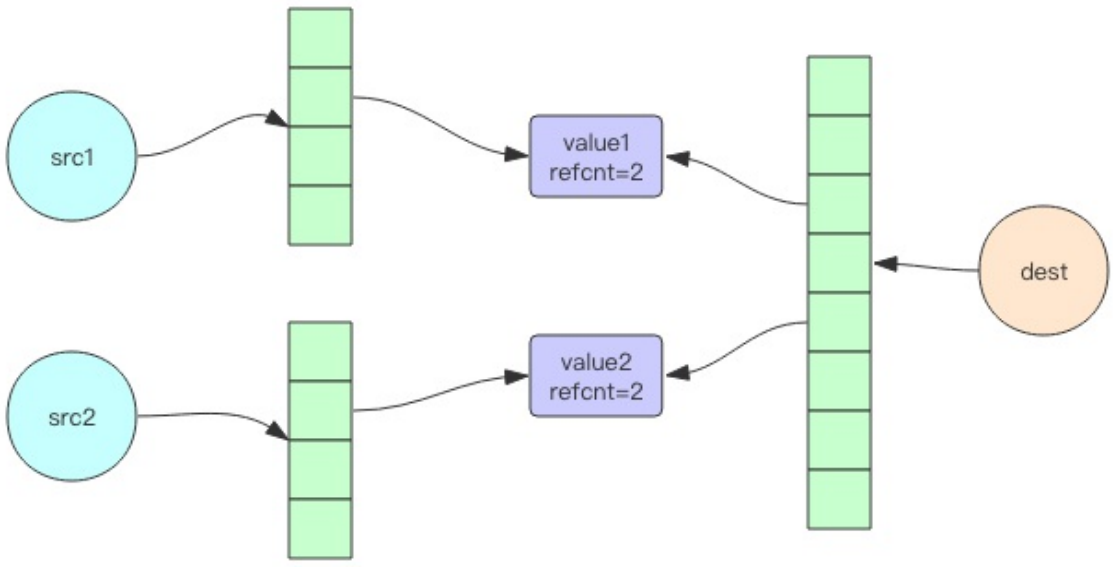
异步线程方案其实也相当复杂

刚才上面说异步线程方案很简单，为什么这里又说它很复杂呢？因为有一点我们之前没有想到。这点非常可怕，严重阻碍了异步线程方案的改造，那就是 Redis 的内部对象有共享机制。

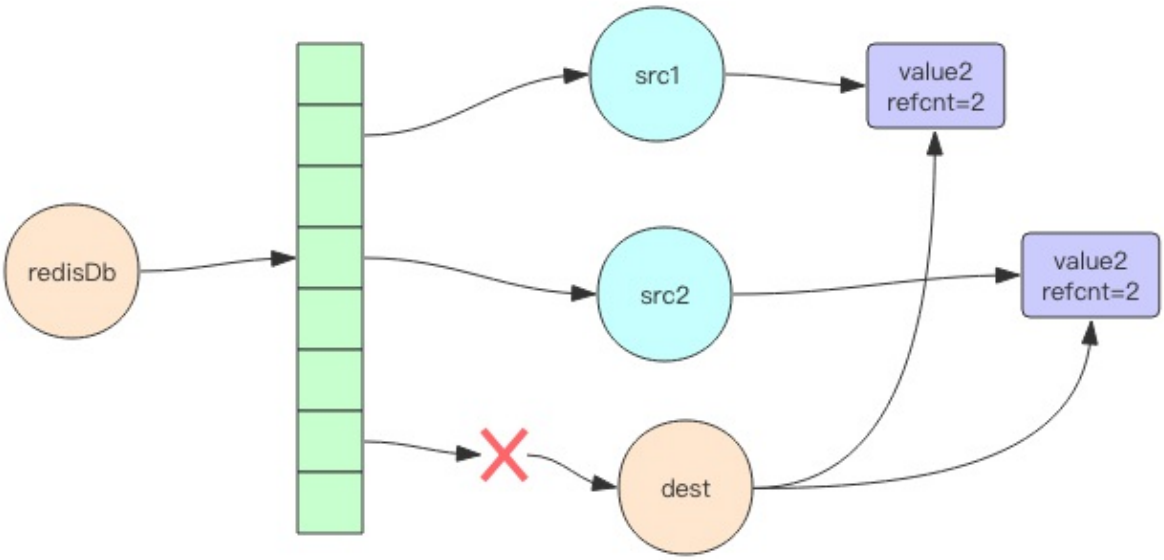
比如集合的并集操作 sunionstore 用来将多个集合合并成一个新集合

```
> sadd src1 value1 value2 value3
(integer) 3
> sadd src2 value3 value4 value5
(integer) 3
> sunionstore dest src1 src2
(integer) 5
> smembers dest
1) "value2"
2) "value3"
3) "value1"
4) "value4"
5) "value5"
```

我们看到新的集合包含了旧集合的所有元素。但是这里有一个我们没看到的 trick。那就是底层的字符串对象被共享了。



为什么对象共享是懒惰删除的巨大障碍呢？因为懒惰删除相当于彻底砍掉某个树枝，将它扔到异步删除队列里去。注意这里必须是彻底删除，而不能藕断丝连。如果底层对象是共享的，那就做不到彻底删除。



所以 Antirez 为了支持懒惰删除，将对象共享机制彻底抛弃，它将这种对象结构称为「share-nothing」，也就是无共享设计。但是甩掉对象共享谈何容易！这种对象共享机制散落在源代码的各个角落，牵一发而动全身，改起来犹如在布满地雷的道路上小心翼翼地行走。

不过 Antirez 还是决心改了，他将这种改动描述为「绝望而疯狂」，可见改动之大之深之险，前后花了好几周的时间才改完。不过效果也是很明显的，对象的删除操作再也不会导致主线程卡顿了。

异步删除的实现

主线程需要将删除任务传递给异步线程，它是通过一个普通的双向链表来传递的。因为链表需要支持多线程并发操作，所以它需要有锁来保护。

执行懒惰删除时，Redis 将删除操作的相关参数封装成一个`bio_job`结构，然后追加到链表尾部。异步线程通过遍历链表摘取 job 元素来挨个执行异步任务。

```
struct bio_job {
    time_t time;  // 时间字段暂时没有使用，应该是预留的
    void *arg1, *arg2, *arg3;
};
```

我们注意到这个 job 结构有三个参数，为什么删除对象需要三个参数呢？我们继续看代码：

```
    /* What we free changes depending on what
arguments are set:
     * arg1 -> free the object at pointer.
     * arg2 & arg3 -> free two dictionaries (a
Redis DB).
     * only arg3 -> free the skiplist. */
    if (job->arg1)
        // 释放一个普通对象，string/set/zset/hash 等
等，用于普通对象的异步删除
        lazyfreeFreeObjectFromBioThread(job-
>arg1);
    else if (job->arg2 && job->arg3)
        // 释放全局 redisDb 对象的 dict 字典和
expires 字典，用于 flushdb
        lazyfreeFreeDatabaseFromBioThread(job-
>arg2,job->arg3);
    else if (job->arg3)
        // 释放 Cluster 的 slots_to_keys 对象，参见
源码篇的「基数树」小节
        lazyfreeFreeSlotsMapFromBioThread(job-
>arg3);
```

可以看到通过组合这三个参数可以实现不同结构的释放逻辑。接下来我们继续追踪普通对象的异步删除lazyfreeFreeObjectFromBioThread是如何进行的，请仔细

阅读代码注释。

```
void lazyfreeFreeObjectFromBioThread(robj *o) {
    decrRefCount(o); // 降低对象的引用计数，如果为零，
就释放
    atomicDecr(lazyfree_objects,1); //
lazyfree_objects 为待释放对象的数量，用于统计
}

// 减少引用计数
void decrRefCount(robj *o) {
    if (o->refcount == 1) {
        // 该释放对象了
        switch(o->type) {
        case OBJ_STRING: freeStringObject(o);
break;
        case OBJ_LIST: freeListObject(o); break;
        case OBJ_SET: freeSetObject(o); break;
        case OBJ_ZSET: freeZsetObject(o); break;
        case OBJ_HASH: freeHashObject(o); break;
// 释放 hash 对象，继续追踪
        case OBJ_MODULE: freeModuleObject(o);
break;
        case OBJ_STREAM: freeStreamObject(o);
break;
        default: serverPanic("Unknown object
type"); break;
        }
        zfree(o);
    } else {
        if (o->refcount <= 0)
serverPanic("decrRefCount against refcount <=
0");
```

```
        if (o->refcount != OBJ_SHARED_REFCOUNT)
o->refcount--; // 引用计数减 1
    }
}

// 释放 hash 对象
void freeHashObject(robj *o) {
    switch (o->encoding) {
    case OBJ_ENCODING_HT:
        // 释放字典，我们继续追踪
        dictRelease((dict*) o->ptr);
        break;
    case OBJ_ENCODING_ZIPLIST:
        // 如果是压缩列表可以直接释放
        // 因为压缩列表是一整块字节数组
        zfree(o->ptr);
        break;
    default:
        serverPanic("Unknown hash encoding
type");
        break;
    }
}

// 释放字典，如果字典正在迁移中，ht[0] 和 ht[1] 分别存储
旧字典和新字典
void dictRelease(dict *d)
{
    _dictClear(d,&d->ht[0],NULL); // 继续追踪
    _dictClear(d,&d->ht[1],NULL);
    zfree(d);
}
```

```
// 这里要释放 hashtable 了
// 需要遍历第一维数组，然后继续遍历第二维链表，双重循环
int _dictClear(dict *d, dictht *ht,
void(callback)(void *)) {
    unsigned long i;

    /* Free all the elements */
    for (i = 0; i < ht->size && ht->used > 0;
i++) {
        dictEntry *he, *nextHe;

        if (callback && (i & 65535) == 0)
callback(d->privdata);

        if ((he = ht->table[i]) == NULL)
continue;
        while(he) {
            nextHe = he->next;
            dictFreeKey(d, he); // 先释放 key
            dictFreeVal(d, he); // 再释放 value
            zfree(he); // 最后释放 entry
            ht->used--;
            he = nextHe;
        }
    }
    /* Free the table and the allocated cache
structure */
    zfree(ht->table); // 可以回收第一维数组了
    /* Re-initialize the table */
    _dictReset(ht);
    return DICT_OK; /* never fails */
}
```

这些代码散落在多个不同的文件，我将它们凑到了一块便于读者阅读。从代码中我们可以看到释放一个对象要深度调用一系列函数，每种对象都有它独特的内存回收逻辑。

队列安全

前面提到任务队列是一个不安全的双向链表，需要使用锁来保护它。当主线程将任务追加到队列之前它需要加锁，追加完毕后，再释放锁，还需要唤醒异步线程，如果它在休眠的话。

```
void bioCreateBackgroundJob(int type, void *arg1,
void *arg2, void *arg3) {
    struct bio_job *job = zmalloc(sizeof(*job));

    job->time = time(NULL);
    job->arg1 = arg1;
    job->arg2 = arg2;
    job->arg3 = arg3;
    pthread_mutex_lock(&bio_mutex[type]); // 加锁
    listAddNodeTail(bio_jobs[type],job); // 追加任
务
    bio_pending[type]++; // 计数
    pthread_cond_signal(&bio_newjob_cond[type]);
// 唤醒异步线程
    pthread_mutex_unlock(&bio_mutex[type]); // 释
放锁
}
```

异步线程需要对任务队列进行轮训处理，依次从链表表头摘取元素逐个处理。摘取元素的时候也需要加锁，摘出来之后再解锁。如果一个元素都没有，它需要等待，直到主线程来唤醒它继续工作。

```
// 异步线程执行逻辑
```

```
void *bioProcessBackgroundJobs(void *arg) {
...
    pthread_mutex_lock(&bio_mutex[type]); // 先加锁
    ...
    // 循环处理
    while(1) {
        listNode *ln;

        /* The loop always starts with the lock hold. */
        if (listLength(bio_jobs[type]) == 0) {
            // 对列空，那就睡觉吧

pthread_cond_wait(&bio_newjob_cond[type],&bio_mutex[type]);
            continue;
        }
        /* Pop the job from the queue. */
        ln = listFirst(bio_jobs[type]); // 获取队列头元素
        job = ln->value;
        /* It is now possible to unlock the background system as we know have
         * a stand alone job structure to process.*/
        pthread_mutex_unlock(&bio_mutex[type]); // 释放锁

        // 这里是处理过程，为了省纸，就略去了
        ...

        // 释放任务对象
```

```
        zfree(job);

        ...

        // 再次加锁继续处理下一个元素
        pthread_mutex_lock(&bio_mutex[type]);
        // 因为任务已经处理完了，可以放心从链表中删除节点了
        listDelNode(bio_jobs[type],ln);
        bio_pending[type]--; // 计数减 1
    }
```

研究完这些加锁解锁的代码后，我开始有点当心主线程的性能。我们都知道加锁解锁是一个相对比较耗时的操作，尤其是悲观锁最为耗时。如果删除很频繁，主线程岂不是要频繁加锁解锁。所以这里肯定还有优化空间，Java 的 ConcurrentLinkQueue 就没有使用这样粗粒度的悲观锁，它优先使用 cas 来控制并发。

思考

1. Redis 还有其它地方用到了对象共享机制么?
2. Java 的 ConcurrentLinkQueue 具体是如何实现的?