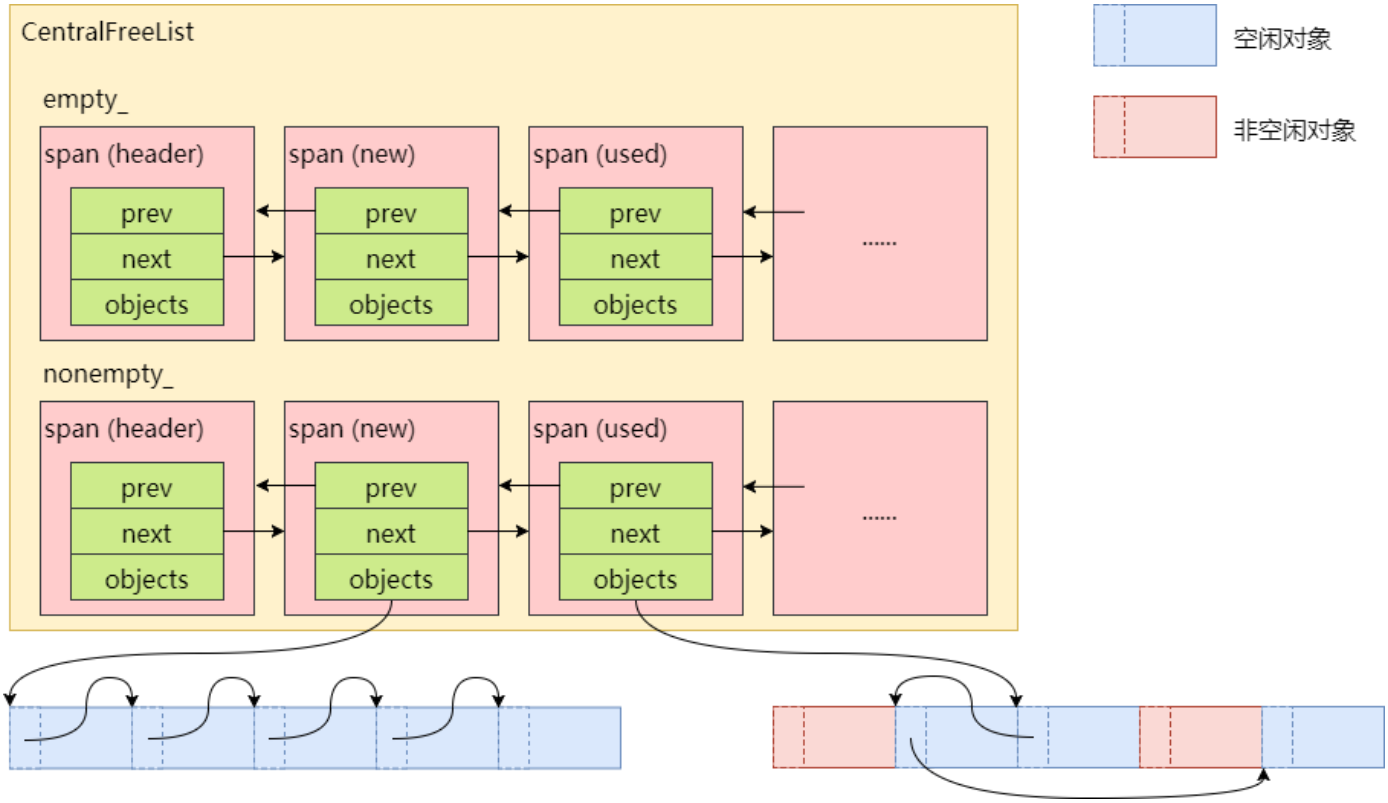
zhuanlan.zhihu.com /p/51433881

TCMalloc解密（三）



原文请移步我的博客：TCMalloc解密

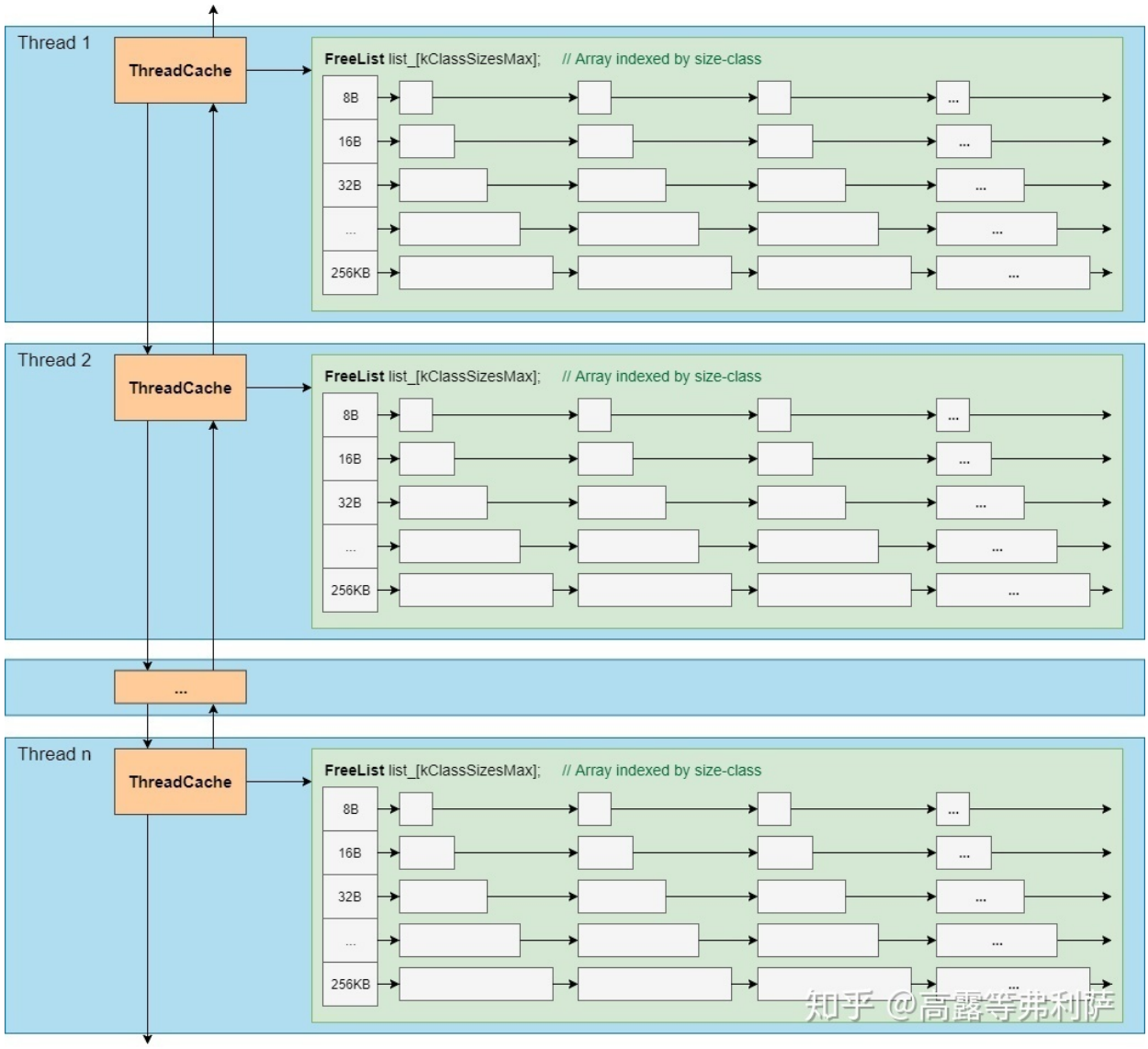
ThreadCache

TCMalloc分配小对象的速度是非常快的，这得益于其对每个线程都有一份单独的cache，即ThreadCache。

ThreadCache其实就是一组FreeList而已。对于每个size class，在ThreadCache中都有一个FreeList，缓存了一组空闲对象，应用程序申请256KB以内的小内存时，优先返回FreeList中的一个空闲对象。因为每个线程每个size class都有单独的FreeList，因此这个过程是**不需要加锁**的，速度非常快。

如果FreeList为空，TCMalloc才会从size class对应的CentralFreeList中获取一组空闲对象放入ThreadCache的FreeList中，并将其中一个对象返回。从CentralFreeList中获取空闲对象**需要加锁**的。

回顾下ThreadCache的结构：



各个线程的ThreadCache互相连接成为一个双向链表，主要目的是为了方便统计信息。

每线程Cache

那么每个线程一个ThreadCache是如何实现的呢？

这依赖于两种技术：Thread Local Storage（TLS)，和Thread Specific Data（TSD）。两者的功能基本是一样的，都是提供每线程存储。TLS用起来更方便，读取数据更快，但在线程销毁时TLS无法执行清理操作，而TSD可以，因此TCMalloc使用TSD为每个线程提供一个ThreadCache，如果TLS可用，则同时使用TLS保存一份拷贝以加速数据的访问。

TLS和TSD的具体细节可参考《The Linux Programming Interface》相关章节（31.3，31.4），本文不再展开讨论。

详细可参考源码中`ThreadCache::CreateCacheIfNecessary()`函数和`threadlocal_data_`变量相关代码。

何时创建ThreadCache

当某线程第一次申请分配内存时，TCMalloc为该线程创建其专属的ThreadCache（`ThreadCache::GetCache()` -> `ThreadCache::CreateCacheIfNecessary()`）。

何时销毁ThreadCache

在TCMalloc初始化TSD时，会调用Pthreads API中的`pthread_key_create()`创建ThreadCache对应的key，并且指定了销毁ThreadCache的函数`ThreadCache::DestroyThreadCache()`。因此，当一个线程销毁时，其对应的ThreadCache会由该函数销毁。

ThreadCache的大小

TCMalloc定义了一些变量来**建议**ThreadCache的大小。注意，是建议，而非强制。也就是说，实际的大小可能会超过这些值。

所有线程的ThreadCache的总大小限制（`overall_thread_cache_size_`）默认为32MB（`kDefaultOverallThreadCacheSize`），取值范围512KB~1GB，可以通过环境变量`TCMalloc_MAX_TOTAL_THREAD_CACHE_BYTES`或以下方式来进行调整：

```
MallocExtension::instance()-
>SetNumericProperty("TCMalloc.max_total_thread_cache_bytes", value);
```

每个线程的ThreadCache的大小限制默认为4MB（`kMaxThreadCacheSize`）。调整ThreadCache总大小时，会修改每个ThreadCache的大小限制到512KB~4MB之间的相应值。

慢启动算法：FreeList的长度控制

控制ThreadCache中各个FreeList中元素的数量是很重要的：

- 太小：不够用，需要经常去CentralCache获取空闲对象，带锁操作
- 太大：太多对象在空闲列表中闲置，浪费内存

不仅是内存分配，对于内存释放来说控制FreeList的长度也很重要：

- 太小：需要经常将空闲对象移至CentralCache，带锁操作
- 太大：太多对象在空闲列表中闲置，浪费内存

并且，有些线程的分配和释放是不对称的，比如生产者线程和消费者线程，这也是需要考虑的一个点。

类似TCP的拥塞控制算法，TCMalloc采用了慢启动（slow start）的方式来控制FreeList的长度，其效果如下：

- FreeList被使用的越频繁，最大长度就越大。
- 如果FreeList更多的用于释放而不是分配，则其最大长度将仅会增长到某一个点，以有效的将整个空闲对象链表一次性移动到CentralCache中。

分配内存时的慢启动代码如下（FetchFromCentralCache）：

```
const int batch_size = Static::sizemap()->num_objects_to_move(cl);

// Increase max length slowly up to batch_size.  After that,
// increase by batch_size in one shot so that the length is a
// multiple of batch_size.
if (list->max_length() < batch_size) {
  list->set_max_length(list->max_length() + 1);
} else {
  // Don't let the list get too long.  In 32 bit builds, the length
  // is represented by a 16 bit int, so we need to watch out for
  // integer overflow.
  int new_length = min<int>(list->max_length() + batch_size,
                            kMaxDynamicFreeListLength);
  // The list's max_length must always be a multiple of batch_size,
  // and kMaxDynamicFreeListLength is not necessarily a multiple
  // of batch_size.
  new_length -= new_length % batch_size;
  ASSERT(new_length % batch_size == 0);
  list->set_max_length(new_length);
}
```

max_length即为FreeList的最大长度，初始值为1。batch_size是size class一节提到的一次性移动空闲对象的数量，其值因size class而异。

可以看到，只要max_length没有超过batch_size，每当FreeList中没有元素需要从CentralCache获取空闲对象时（即FetchFromCentralCache），max_length就加1。

一旦max_length达到batch_size，接下来每次FetchFromCentralCache就会导致max_length增加batch_size。

但并不会无限制的增加，最大到kMaxDynamicFreeListLength（8192），以避免从FreeList向CentralCache移动对象时，因为对象过多而过长的占用锁。

再来看内存回收时的情况，每次释放小对象，都会检查FreeList的当前长度是否超过max_length：

```
if (PREDICT_FALSE(length > list->max_length())) {
  ListTooLong(list, cl);
  return;
}
```

如果超长，则执行以下逻辑：

```
void ThreadCache::ListTooLong(FreeList* list, uint32 cl) {
  size_ += list->object_size();
```

```
  const int batch_size = Static::sizemap()->num_objects_to_move(cl);
  ReleaseToCentralCache(list, cl, batch_size);

  // If the list is too long, we need to transfer some number of
  // objects to the central cache.  Ideally, we would transfer
  // num_objects_to_move, so the code below tries to make max_length
  // converge on num_objects_to_move.

  if (list->max_length() < batch_size) {
    // Slow start the max_length so we don't overreserve.
    list->set_max_length(list->max_length() + 1);
  } else if (list->max_length() > batch_size) {
    // If we consistently go over max_length, shrink max_length.  If we
don't
    // shrink it, some amount of memory will always stay in this freelist.
    list->set_length_overages(list->length_overages() + 1);
    if (list->length_overages() > kMaxOverages) {
      ASSERT(list->max_length() > batch_size);
      list->set_max_length(list->max_length() - batch_size);
      list->set_length_overages(0);
    }
  }

  if (PREDICT_FALSE(size_ > max_size_)) {
    Scavenge();
  }
}
```

与内存分配的情况类似，只要max_length还没有达到batch_size，每当FreeList的长度超过max_length，max_length的值就加1。

当max_length达到或超过batch_size后，并不会立即调整max_length，而是累计超过3次（`kMaxOverages`）后，才会将max_length减少batch_size。

垃圾回收

TODO：本节还没写完，请先参阅官方介绍Garbage Collection of Thread Caches一节。。

从ThreadCache中回收垃圾对象，将未使用的对象返回到CentralFreeList，可以控制缓存的大小。

不同线程对缓存大小的需求是不一样的，因此不能统一对待：有些线程需要大的缓存，有些线程需要小的缓存即可，甚至有些线程不需要缓存。

当一个ThreadCache大小超过其max_size_时，触发垃圾回收：

```
if (PREDICT_FALSE(size_ > max_size_)){
  Scavenge();
}
```

只有当应用程序释放内存时（`ThreadCache::Deallocate()`）才会触发垃圾回收，遍历ThreadCache中所有的FreeList，将FreeList中的一些对象移至对应的CentralFreeList中。

具体移动多少对象由低水位标记L（`lowater_`，每个FreeList一个）来决定。L记录自上次垃圾收集以来，FreeList的最小长度。

CentralCache

CentralCache是逻辑上的概念，其本质是CentralFreeListPadded类型（CentralFreeList的子类，用于64字节对齐）的数组，每个size class对应数组中的一个元素。

```
ATTRIBUTE_HIDDEN static CentralFreeListPadded
central_cache_[kClassSizesMax];
```

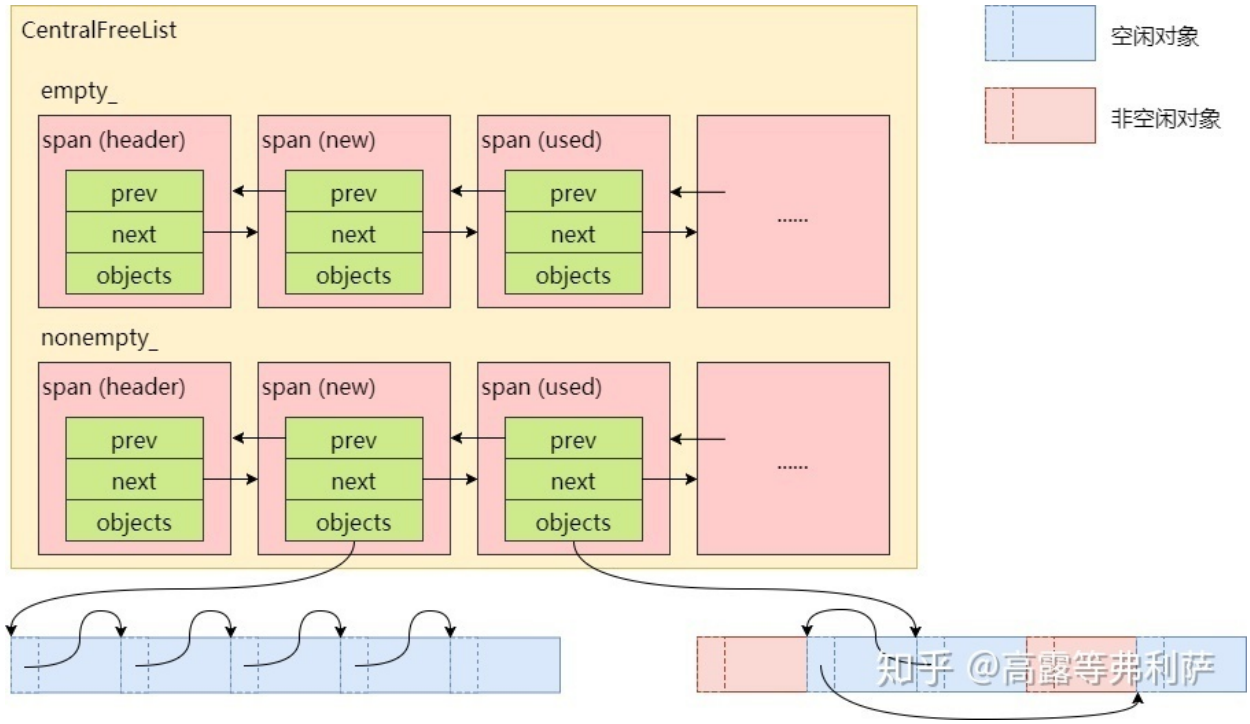
由于各线程公用一个CentralCache，因此，使用CentralCache时需要加锁。

以下讨论都是针对某一个size class的。

CentralFreeList中缓存了一系列小对象，供各线程的ThreadCache取用，各线程也会将多余的空闲小对象还给CentralFreeList，另外CentralFreeList还负责从PageHeap申请span以分割成小对象，以及将不再使用的span还给PageHeap。

管理span

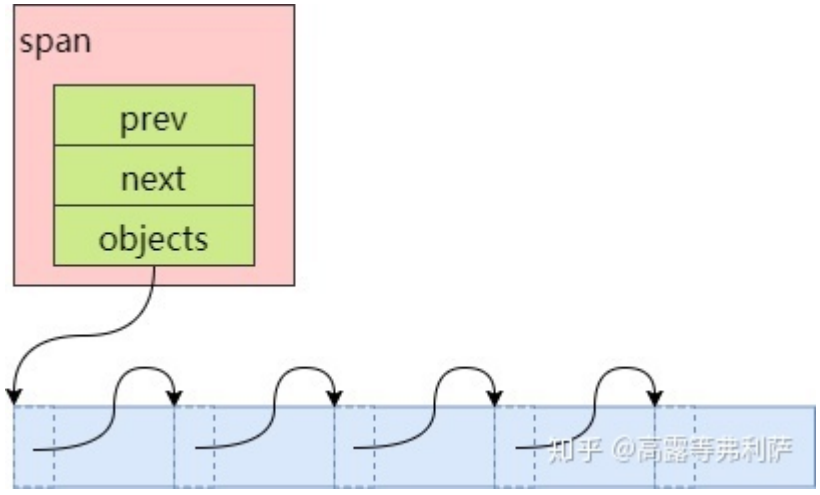
CentralFreeList真正管理的是span，而小对象是包含在span中的空闲对象链表中的。CentralFreeList的`empty_`链表保存了已经没有空闲对象可用的span，`nonempty_`链表保存了还有空闲对象可用的span：



CentralFreeList↔ PageHeap

从PageHeap获取span

当ThreadCache从CentralFreeList取用空闲对象（`RemoveRange`），但CentralFreeList的空闲对象数量不够时，CentralFreeList调用`Populate()`从PageHeap申请一个span拆分成若干小对象，首首连接记录在span的`objects`指针中，即每个小对象的起始位置处，记录了下一个小对象的地址。此时的span如下图：



可以看到，此时span包含的对象按顺序连接在一起。

新申请的span被放入CentralFreeList的`nonempty_`链表头部。

将span还给PageHeap

CentralFreeList维护span的成员变量`refcount`，用来记录ThreadCache从中获取了多少对象。

当ThreadCache将不再使用的对象归还给CentralCache以致refcount减为0，即span中所有对象都空闲时，则CentralCache将这个span还给PageHeap。截取`CentralFreeList::ReleaseToSpans()`部分代码如下：

```
span->refcount--;
if (span->refcount == 0) {
  Event(span, '#', 0);
  counter_ -= ((span->length<<kPageShift) /
               Static::sizemap()->ByteSizeForClass(span->sizeclass));
  tcmalloc::DLL_Remove(span);
  --num_spans_;

  // Release central list lock while operating on pageheap
  lock_.Unlock();
  {
    SpinLockHolder h(Static::pageheap_lock());
    Static::pageheap()->Delete(span);
  }
  lock_.Lock();
}
```

CentralFreeList↔ThreadCache

CentralFreeList和ThreadCache之间的对象移动是批量进行的：

```
// Insert the specified range into the central freelist.  N is the number
of
// elements in the range.  RemoveRange() is the opposite operation.
void InsertRange(void *start, void *end, int N);

// Returns the actual number of fetched elements and sets *start and *end.
int RemoveRange(void **start, void **end, int N);
```

start和end指定小对象链表的范围，N指定小对象的数量。批量移动小对象可以**均摊锁操作的开销**。

ThreadCache取用小对象

当ThreadCache中某个size class没有空闲对象可用时，需要从CentralFreeList获取N个对象，那么N的值是多少呢？从`ThreadCache::FetchFromCentralCache()`中可以找到答案：

```
const int batch_size = Static::sizemap()->num_objects_to_move(cl);
const int num_to_move = min<int>(list->max_length(), batch_size);
void *start, *end;
int fetch_count = Static::central_cache()[cl].RemoveRange(&start, &end,
num_to_move);
```

移动数量N为max_length和batch_size的最小值（两者的具体涵义参见ThreadCache慢启动一节）。

假设只考虑内存分配的情况，一开始移动1个，然后是2个、3个，以此类推，同时max_length每次也加1，直到达到batch_size后，每次移动batch_size个对象。

CentralFreeList和ThreadCache之间的对象移动有个优化措施，因为大部分情况都是每次移动batch_size个对象，为了减少链表操作，提升效率，CentralFreeList将移动的batch_size个对象的链表的首尾指针缓存在了TCEntry中。因此后续只要需要移动batch_size个对象，只需要操作链表的首尾即可。

```
// Here we reserve space for TCEntry cache slots.  Space is preallocated
// for the largest possible number of entries than any one size class may
// accumulate.  Not all size classes are allowed to accumulate
// kMaxNumTransferEntries, so there is some wasted space for those size
// classes.
TCEntry tc_slots_[kMaxNumTransferEntries];
```

ThreadCache归还小对象

当ThreadCache中的空闲对象过多时（`ThreadCache::ListTooLong()`），会将一部分空闲对象放回CentralFreeList（`ThreadCache::ReleaseToCentralCache()`）。如何判断空闲对象过多请参考ThreadCache慢启动一节。

线程销毁也会将其ThreadCache中所有的空闲对象都放回CentralFreeList。

如果ThreadCache缓存的内存大小超过其允许的最大值，会触发GC操作（`ThreadCache::Scavenge()`），在其中也会将部分小对象归还给CentralFreeList，具体请参考ThreadCache垃圾回收一节。

全文完。