tcmalloc

Table of Contents

1 写在前面

- 内存管理内幕 http://www.ibm.com/developerworks/cn/linux/l-memory/
- hoard内存分配器 http://www.hoard.org/
- dlmalloc内存分配器 http://gee.cs.oswego.edu/dl/html/malloc.html
- ptmalloc2内存分配器 http://www.malloc.de/en/index.html
- jemalloc内存分配器 http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

- jemalloc地址 http://www.canonware.com/download/jemalloc/
- tcmalloc内存分配器 http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html
- tcmalloc地址 http://google-perftools.googlecode.com/files/google-perftools-1.8.3.tar.gz

如果你对使用tcmalloc有什么问题的话，请给我发邮件。我会尽量给你答复，对于常见的问题也会整理到FAQ上面。

jemalloc论文上面谈到了很多关于内存分配器方面的基本概念与问题。 性能指标主要体现在分配时间以及平均和高峰内存使用大小上面。但是两个指标很难单独测量，所以现在比较权威的测量方式还是使用benchmark然后看看运行时间以及内存使用量。内存碎片分为内部碎片和外部碎片。 内部碎片通常都是因为分配的话会进行round之后而没有使用的部分，而外部碎片指已经回收但是因为地址不连续等原因没有办法被应用程序使用的部分。 jemalloc提高CPU Cache命中率有两个途径：

- 首先尽可能地让内存使用更小。working set可以放在cache-line里面全部存放效率就会比较好。
- 另外就是应该让连续开辟的对象放在一起。jemalloc是假设在一个线程内部调用两次的malloc的话，那么通常是在一起访问的。

第一个假设非常合理，但是第二个假设不一定是合理的。jemalloc首先确保第一个前提，然后尽可能地保证第二个条件。 另外cache命中率的一个问题就是false-cache-line.简单的说就是两个线程开辟的对象(A,B)可能连在一起，可以一起载入cache-line.线程1对于A的修改会造成线程2下一次读取B的时候，需要重新从内存载入，因为对于A的修改会使得所在的cache-line失效。解决这个问题的办法就是有多个allocation arena.不同的线程尽可能在不同的arena下面开辟。

锁冲突是造成传统malloc在多线程情况下表现差的主要原因。解决的方法和解决false-cache-line是一样的，都是开辟多个allocation arena然后让不同的线程尽可能地在不同的arena分配。

ptmalloc2解决为了解决锁冲突这个问题，也采用了arena-per-thread的方法。但是ptmalloc2内部依然存在一个大的问题，就是各个arena之间是没有办法迁移的。 如果一个线程一开始开辟很大但是之后释放了，那个这块内存是没有办法被其他线程所使用的。

同样tcmalloc会为每一个线程分配一个arena,这样每一个线程分配时候都不需要进行加锁。但是tcmalloc解决了ptmalloc2的内存迁移问题。 tcmalloc如果发现thread cache内部占用率高但是使用率低的话，那么会将部分内存存放在中心部分。基本上jemalloc原理上和tcmalloc相似。 每个线程有一个arena池，但是线程按照round-robin方式在每一个arena上面取。代价是需要加锁，但是假设冲突应该不严重。

后面我们着重针对tcmalloc进行分析。tcmalloc文档写的足够好了，看完一遍基本上就知道内部原理了。所以这里我也只是自己总结一下，然后用自己的理解写出来。里面尽量附上代码分析:) tcmalloc代码写得相当得好，虽然很多地方没有看懂(而且我猜想有很多地方已经过时了但是没有删去，所以对于代码阅读有一定的影响).基本上阅读完tcmalloc阅读和编写代码能力会提高很多。 看下面分析之前，还是强烈建议先阅读一次文档。

2 函数入口

tcmalloc.cc

tcmallo.cc里面定义了函数入口.对于安排在section(google_malloc)不知道有作用。当然下面有很多相关的函数比如tc_memalign，但是这些并不影响阅读主线。 我们只需要关心两个函数tc_malloc以及tc_free即可。

```
#include <malloc.h>
#define __THROW
# define ATTRIBUTE_SECTION(name) __attribute__ ((section (#name)))
extern "C" {
  void* tc_malloc(size_t size) __THROW
      ATTRIBUTE_SECTION(google_malloc);
  void tc_free(void* ptr) __THROW
      ATTRIBUTE_SECTION(google_malloc);
}
```

然后在libc_override_gcc_and_weak.h和libc_override_glibc.h进行了函数替换

```
#define ALIAS(tc_fn)   __attribute__ ((alias (#tc_fn)))
extern "C" {
  void* __libc_malloc(size_t size)               ALIAS(tc_malloc);
  void __libc_free(void* ptr)                    ALIAS(tc_free);
} // extern "C"
extern "C" {
  void* malloc(size_t size) __THROW              ALIAS(tc_malloc);
  void free(void* ptr) __THROW                   ALIAS(tc_free);
} // extern "C"
```

同时在里面还覆盖了malloc_hook和free_hook这两个函数，不允许用户自己进行hook. 我猜想tcmalloc应该是自己提供了malloc_hook和free_hook的定义.

```
extern "C" {
static void* glibc_override_malloc(size_t size, const void *caller) {
  return tc_malloc(size);
}
static void glibc_override_free(void *ptr, const void *caller) {
  tc_free(ptr);
}
void* (* MALLOC_HOOK_MAYBE_VOLATILE __malloc_hook)(size_t, const void*)
    = &glibc_override_malloc;
void (* MALLOC_HOOK_MAYBE_VOLATILE __free_hook)(void*, const void*)
```

```
    = &glibc_override_free;
} // extern "C"
```

仔细阅读tcmalloc.cc接口面还发现了下面这些接口非常有意思

```
// 返回当前malloc信息,在malloc.h里面有定义
struct mallinfo tc_mallinfo(void) __THROW ATTRIBUTE_SECTION(google_malloc);
// 这个指针实际可用内存大小
size_t tc_malloc_size(void* p) __THROW ATTRIBUTE_SECTION(google_malloc);
// 打印当前malloc状态
void tc_malloc_stats(void) __THROW  ATTRIBUTE_SECTION(google_malloc);
// 修改malloc参数,在malloc.h里面有修改选项
int tc_mallopt(int cmd, int value) __THROW ATTRIBUTE_SECTION(google_malloc);
```

可以结合当前的ptmalloc2(glibc.2.3.4)来看看这些接口的行为.了解这些行为主要是对于内存分配器如果出问题的话，那么至少有方法可以了解内部情况.

3 全局内存

system-alloc.h

```
extern void* TCMalloc_SystemAlloc(size_t bytes, size_t *actual_bytes,
                                  size_t alignment = 0);
extern void TCMalloc_SystemRelease(void* start, size_t length);
```

基本可以认为Release部分没有任何操作。对于SystemAlloc底层实现非常巧妙.首先tcmalloc定义了SysAllocator这个接口，然后底层有两个实现：

- SbrkSysAllocator.使用sbrk来分配内存
- MmapSysAllocator.使用mmap来分配内存

SysAllocator需要实现一个接口void* Alloc(size_t size, size_t *actual_size, size_t alignment);因为全局只是需要一个这样的对象， 所以这个对象可以静态分配即可.然后定义了一个DefaultSysAllocator允许设置Children.

```
static char sbrk_space[sizeof(SbrkSysAllocator)];
static char mmap_space[sizeof(MmapSysAllocator)];
static char default_space[sizeof(DefaultSysAllocator)];
```

在初始化InitSystemAllocators的时候将sbrk_space以及mmap_space作为default_space的两个children.

```
MmapSysAllocator *mmap = new (mmap_space) MmapSysAllocator();
SbrkSysAllocator *sbrk = new (sbrk_space) SbrkSysAllocator();
DefaultSysAllocator *sdef = new (default_space) DefaultSysAllocator();
if (kDebugMode && sizeof(void*) > 4) {
  sdef->SetChildAllocator(mmap, 0, mmap_name);
  sdef->SetChildAllocator(sbrk, 1, sbrk_name);
} else {
  sdef->SetChildAllocator(sbrk, 0, sbrk_name);
  sdef->SetChildAllocator(mmap, 1, mmap_name);
}
```

实际操作时候都是先sbrk尝试先，然后使用mmap.DefaultAllocator按照children顺序尝试分配，也就意味着首先使用sbrk如果不成功尝试mmap

```
void* DefaultSysAllocator::Alloc(size_t size, size_t *actual_size,
                                 size_t alignment) {
  for (int i = 0; i < kMaxAllocators; i++) {
    if (!failed_[i] && allocs_[i] != NULL) {
      void* result = allocs_[i]->Alloc(size, actual_size, alignment);
      if (result != NULL) {
        return result;
      }
      TCMalloc_MESSAGE(__FILE__, __LINE__, "%s failed.\n", names_[i]);
      failed_[i] = true;
    }
  }
  // After both failed, reset "failed_" to false so that a single failed
  // allocation won't make the allocator never work again.
  for (int i = 0; i < kMaxAllocators; i++) {
    failed_[i] = false;
  }
  return NULL;
}
```

可以说系统里面所有使用的内存都是从这个地方分配的，包括thread_cache,page_allocator以及管理对象。 此外还需要注意的是，因为会有多线程调用这个东西，所以在SystemAlloc之前的话会调用自选锁进行锁定。SpinLockHolder lock_holder(&spinlock);

4 管理对象

- tcmalloc_guard.h
- static_vars.h
- page_heap_allocator.h

- common.h
- central_freelist.h
- page_heap.h
- page_map.h
- packed-cache-inl.h
- thread_cache.h

4.1 TCMallocGuard

tcmalloc_guard.h

TCMallocGuard主要是为了确保在tc_malloc之前所有静态变量都已经完成了初始化。首先全局存在一个static TCMallocGuard module_enter_exit_hook; 这个变量来确保静态初始化，但是同时为了防止重复初始化还加了引用计数进行判断

```
static int tcmallocguard_refcount = 0;  // no lock needed: runs before main()
TCMallocGuard::TCMallocGuard() {
  if (tcmallocguard_refcount++ == 0) {
    ReplaceSystemAlloc();    // defined in libc_override_*.h // 这个对于Linux来说没有任何操作
    tc_free(tc_malloc(1)); // 这个地方个人觉得没有必要，可能只是为了看看是否可以再InitTSD之前run起来
    ThreadCache::InitTSD(); // 初始化一下tc的线程局部变量
    tc_free(tc_malloc(1));
    if (RunningOnValgrind()) { // 从代码上看可能是从环境变量里面获取的。
      // Let Valgrind uses its own malloc (so don't register our extension).
    } else {
      MallocExtension::Register(new TCMallocImplementation);
    }
  }
}
```

对于释放来说的话也非常简单，可以根据环境变量来选择是否打印统计信息

```
TCMallocGuard::~TCMallocGuard() {
  if (--tcmallocguard_refcount == 0) {
    const char* env = getenv("MALLOCSTATS");
    if (env != NULL) {
      int level = atoi(env);
      if (level < 1) level = 1;
      PrintStats(level);
    }
  }
}
```

4.2 PageHeapAllocator

page_heap_allocator.h

如果管理对象预先知道了大小那么可以静态分配使用in-placement new方式完成，但是如果管理对象是动态分配的话，那么如何管理这些对象的分配呢？ 答案非常简单使用sample_alloc.所以sample_alloc就是这个分配器知道了每次分配对象的大小，回收缓存起来挂在free_list上面，分配首先从free_list尝试分配， 如果free_list为空的话，那么久会调用全局内存分配。

page_heap_allocator.h里面实现了一个sample_alloc叫做PageHeapAllocator.原理来说非常简单，这里就不赘述了。需要注意的是每一个节点肯定都是>sizeof(void*)的， 所以每个节点不用分配额外的next指针空间，这个是一个基本上所以写过内存分配器程序员公开的技巧了。另外需要关注的是每次向全局内存空间要的大小是多少

```
static const int kAllocIncrement = 128 << 10; // 128K
```

里面还维护了一个inuse()接口表示当前有多少个object正在被使用。

另外为了更好的统计管理对象使用的内存，在common.cc里面记录了元信息分配的内存大小

```
static uint64_t metadata_system_bytes_ = 0;
void* MetaDataAlloc(size_t bytes) {
  void* result = TCMalloc_SystemAlloc(bytes, NULL);
  if (result != NULL) {
    metadata_system_bytes_ += bytes;
  }
  return result;
}
uint64_t metadata_system_bytes() { return metadata_system_bytes_; }
```

只要所有的元信息都从MetaDataAlloc这里分配即可。

4.3 SizeMap

common.h

SizeMap定义了slab大小，大小到slab编号的映射，一种slab每次分配的多少个pages，一种slab的话在tc和central cache中每次移动多少个对象。 具体定义可以阅读common.h.里面的算法个人觉得还是比较复杂的没有仔细研究。slab的一共有

```
#if defined(TCMALLOC_LARGE_PAGES)
static const size_t kPageShift  = 15;
static const size_t kNumClasses = 78;
#else
static const size_t kPageShift  = 13;
static const size_t kNumClasses = 86;
#endif
```

对于我们如果使用大页面的话，32K的话那么有77种slab,否则只有85种。注意这里slab的编号从1开始计算。

tcmalloc提供了一个Dump方法可以查看最终这些数值。我们需要和源代码联合编译才有可能看到

```
#include <cstdio>
#include <src/internal_logging.h>
#include <src/static_vars.h>

char buf[1024*1024];
int main() {
  // initialize tcmalloc
  void* p=malloc(10);
  free(p);
  tcmalloc::SizeMap* sizemap=tcmalloc::Static::sizemap();
  // print aux info
  for(int i=1;i<kNumClasses;i++){
    printf("SC %d [%d]\n",i,sizemap->num_objects_to_move(i));
  }
  // print stats.
  TCMalloc_Printer printer(buf,sizeof(buf));
  sizemap->Dump(&printer);
  printf("%s\n",buf);
  return 0;
}
```

查看结果是

```
SC 1 [32]
SC 2 [32]
SC 3 [32]
SC 4 [32]
SC 5 [32]
SC 6 [32]
SC 7 [32]
SC 8 [32]
SC 9 [32]
...

SC   1 [        1 ..        8 ] from     8192 ; 88% maxwaste
SC   2 [        9 ..       16 ] from     8192 ; 44% maxwaste
SC   3 [       17 ..       32 ] from     8192 ; 47% maxwaste
SC   4 [       33 ..       48 ] from     8192 ; 32% maxwaste
SC   5 [       49 ..       64 ] from     8192 ; 23% maxwaste
SC   6 [       65 ..       80 ] from     8192 ; 19% maxwaste
SC   7 [       81 ..       96 ] from     8192 ; 16% maxwaste
....
```

这个意思就很清楚，对于slab1的对象来说的话，每次会将32个对象在tc(thread cache)和cc(central cache)之间调动。 如果是1–8字节的话那么按照8字节分配，如果分配pages的话那分配8192字节。最大浪费率是88%(8–1)/8. 对于81–96字节的话，那么最大浪费率就是(96–81)/96–16%. (注意这里打印分配pages的话已经<< kPageShift,如果kPageShift=12的话，8192字节那么相当于2pages)

4.4 Central Cache

central_freelist.h

4.4.1 Data Structure

首先在static里面定义的central_cache是一个数组大小为kNumClasses，相当于和每一个thread cache里面的slab对应。 数组每个元素是CentralFreeListPadded,在central_freelist.h里面定义的。阅读CentralFreeListPadded这个结构，就会发现， 实际上这个功能是在CentralFreeList里面的，为了能够进行align进行了padded,还是非常巧妙的

```
template<int kFreeListSizeMod64>
class CentralFreeListPaddedTo : public CentralFreeList {
 private:
  char pad_[64 - kFreeListSizeMod64];
};

template<>
class CentralFreeListPaddedTo<0> : public CentralFreeList {
};

class CentralFreeListPadded : public CentralFreeListPaddedTo<
  sizeof(CentralFreeList) % 64> {
};
```

所以后续的话我们只需要关注CentralFreeList即可。

数据结构基本上还是很好理解的:).

```
class CentralFreeList {
 private:
  // TransferCache is used to cache transfers of
  // sizemap.num_objects_to_move(size_class) back and forth between
  // thread caches and the central cache for a given size class.
  struct TCEntry {
    void *head;  // Head of chain of objects.
    void *tail;  // Tail of chain of objects.
  };
  // A central cache freelist can have anywhere from 0 to kMaxNumTransferEntries
  // slots to put link list chains into.
#ifdef TCMALLOC_SMALL_BUT_SLOW
  // For the small memory model, the transfer cache is not used.
  static const int kMaxNumTransferEntries = 0;
#else
  // Starting point for the the maximum number of entries in the transfer cache.
  // This actual maximum for a given size class may be lower than this
  // maximum value.
  static const int kMaxNumTransferEntries = 64;
#endif
  // This lock protects all the data members.  cached_entries and cache_size_
  // may be looked at without holding the lock.
  SpinLock lock_;

  // We keep linked lists of empty and non-empty spans.
  size_t   size_class_;     // My size class
  Span     empty_;          // Dummy header for list of empty spans
  Span     nonempty_;       // Dummy header for list of non-empty spans
  size_t   num_spans_;      // Number of spans in empty_ plus nonempty_
  size_t   counter_;        // Number of free objects in cache entry

  // 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];

  // Number of currently used cached entries in tc_slots_.  This variable is
  // updated under a lock but can be read without one.
  int32_t used_slots_;  // 当前使用的tc entries.
  // The current number of slots for this size class.  This is an
  // adaptive value that is increased if there is lots of traffic
  // on a given size class.
  int32_t cache_size_; // 当前允许的最大的tc entries.
  // Maximum size of the cache for a given size class.
  int32_t max_cache_size_; // 最大允许多少个tc entries.
}
```

CentralFreeList的接口非常少

- void Init(size_t cl); // 初始化,cl表示自己是第几个class
- void InsertRange(void *start, void *end, int N); // 回收部分objects.
- int RemoveRange(void **start, void **end, int N); // 分配部分objects.
- length // 在cache里面存在多少个free objects(不包含transfer cache)
- tc_length // transfer cache里面包含多少free objects.
- OverheadBytes // 因为内部碎片造成的额外开销

因为cc是被全局操作的，所以这些接口在实际操作的时候内部都会首先尝试加上自选锁。很明显cc里面使用了free list链表结构管理这些free object. 之前说过ptmalloc2会有这么一个问题，就是如果局部线程分配过多的话没有机制将内存返回给主区域。而tcmalloc解决了这个问题。 对于每一个slab的tc返回的对象个数都是固定的，如果cc可以将这个返回的部分特殊处理的话，那么下次tc还需要这个部分的话， 那么就可以很快地进行分配，否则需要遍历如果freelist不够的话那么还需要从pageheap里面进行切片。而这个部分就叫做transfer cache.:) 了解了这些之后就可以看各个接口实现了。

4.4.2 Init

init主要是计算了tc(transfer cache)的max_cache_size以及cache_size,然后初始化了字段。 我们这里暂时不关注empty以及nonempty这两个字段的数据结构

```
void CentralFreeList::Init(size_t cl) {
  size_class_ = cl;
  tcmalloc::DLL_Init(&empty_);
  tcmalloc::DLL_Init(&nonempty_);
  num_spans_ = 0;
  counter_ = 0;

  max_cache_size_ = kMaxNumTransferEntries;
#ifdef TCMALLOC_SMALL_BUT_SLOW
  // Disable the transfer cache for the small footprint case.
  cache_size_ = 0;
#else
  cache_size_ = 16;
#endif
  if (cl > 0) {
    int32_t bytes = Static::sizemap()->ByteSizeForClass(cl);
    int32_t objs_to_move = Static::sizemap()->num_objects_to_move(cl);
    max_cache_size_ = (min)(max_cache_size_,
```

```
                          (max)(1, (1024 * 1024) / (bytes * objs_to_move)));
    cache_size_ = (min)(cache_size_, max_cache_size_);
  }
  used_slots_ = 0;
  ASSERT(cache_size_ <= max_cache_size_);
}
```

4.4.3 InsertRange

这个接口就是为了回收[start,end]并且长度为N objects的内存链。首先注意它加了自选锁确保了线程安全。 然后有一个逻辑就是判断是否可以进入tc,如果不允许进入tc的话那么挂到链上去。

```
void CentralFreeList::InsertRange(void *start, void *end, int N) {
  SpinLockHolder h(&lock_);
  if (N == Static::sizemap()->num_objects_to_move(size_class_) &&
    MakeCacheSpace()) { // 这里没有看懂MakeCacheSpace里面一个逻辑，我自己觉得是无关紧要的。
    // 因为看上去像是收缩其他的slab cc(EvictRandomSizeClass).
    // 这里我们可以简单地认为，它就是在计算tc_slots里面是否有slot可以分配.
    int slot = used_slots_++;
    ASSERT(slot >=0);
    ASSERT(slot < max_cache_size_);
    TCEntry *entry = &tc_slots_[slot]; // 如果分配成功的话，那么直接挂载.
    entry->head = start;
    entry->tail = end;
    return;
  }
  ReleaseListToSpans(start); // 如果不允许挂到tc的话，那么就需要单独处理.
}
```

回收到tc这个逻辑非常简单，然后看看ReleaseListToSpans这个逻辑。大致逻辑就是遍历start知道end, 然后对于每一个object调用ReleaseToSpans单独进行处理。

```
void CentralFreeList::ReleaseToSpans(void* object) {
  Span* span = MapObjectToSpan(object); // 将object映射到span
  ASSERT(span != NULL);
  ASSERT(span->refcount > 0);

  // If span is empty, move it to non-empty list
  if (span->objects == NULL) { // 如果span上面没有任何free objects的话.
    tcmalloc::DLL_Remove(span); // 那么将span从原来挂载链表删除(empty).
    tcmalloc::DLL_Prepend(&nonempty_, span); // 挂载到这个cc的nonempty链表上.
    Event(span, 'N', 0);
  }

  counter_++; // 当前free objects增加了
  span->refcount--; // 这个span的ref count减少了.
  // span refcount表示里面有多少个objects分配出去了.
  if (span->refcount == 0) { // 如果==0的话，那么说明这个span可以回收了.
    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); // 将span回收pageheap里面去，这个地方可能会进行内存合并
    }
    lock_.Lock();
  } else {
    // 否则就将这个object挂在span链上.
    *(reinterpret_cast<void**>(object)) = span->objects;
    span->objects = object;
  }
}
```

这里有一个最重要的问题就是MapObjectToSpan,object是如何映射到span的。这里我们首先可以大致说一下， 就是tcmalloc因为是按照page来分配的，所以如果知道地址的话，那么其实就知道于第几个页。而span可以管理多个页， 这样的话就可以知道这个页是哪个span来管理的了。具体代码的话会在span管理部分说明。

4.4.4 RemoveRange

这个接口就是为了尝试分配N个objects对象，然后将首地址尾地址给start和end.同样内部逻辑会判断是否可以从tc 中直接取出，如果可以取出的话那么分配就非常快。注意函数开始也尝试加锁了。

```
int CentralFreeList::RemoveRange(void **start, void **end, int N) {
  ASSERT(N > 0);
  lock_.Lock();
  if (N == Static::sizemap()->num_objects_to_move(size_class_) &&
      used_slots_ > 0) { // 如果可以直接从tc里面分配.
    int slot = --used_slots_;
    ASSERT(slot >= 0);
    TCEntry *entry = &tc_slots_[slot];
```

```
    *start = entry->head;
    *end = entry->tail;
    lock_.Unlock();
    return N;
  }

  int result = 0;
  void* head = NULL;
  void* tail = NULL;
  // TODO: Prefetch multiple TCEntries?
  tail = FetchFromSpansSafe(); // 逻辑是首先放在尾部,然后不断地在头部拼接.
  if (tail != NULL) {
    SLL_SetNext(tail, NULL);
    head = tail;
    result = 1;
    while (result < N) {
      void *t = FetchFromSpans();
      if (!t) break;
      SLL_Push(&head, t);
      result++;
    }
  }
  lock_.Unlock();
  *start = head;
  *end = tail;
  return result;
}
```

其中FetchFromSpanSafe逻辑也比较简单，就是

```
void* CentralFreeList::FetchFromSpansSafe() {
  void *t = FetchFromSpans();
  if (!t) {
    Populate(); // 尝试迁移
    t = FetchFromSpans();
  }
  return t;
}
```

首先我们要看懂FetchFromSpans()逻辑，才能够清楚什么情况下面需要调用Populate

```
void* CentralFreeList::FetchFromSpans() {
  if (tcmalloc::DLL_IsEmpty(&nonempty_)) return NULL; // 如果span里面都空了的.
  Span* span = nonempty_.next;

  ASSERT(span->objects != NULL);
  span->refcount++;
  void* result = span->objects; // 否则就会从span里面分配object.
  span->objects = *(reinterpret_cast<void**>(result));
  if (span->objects == NULL) {
    // Move to empty list
    tcmalloc::DLL_Remove(span);
    tcmalloc::DLL_Prepend(&empty_, span);
    Event(span, 'E', 0);
  }
  counter_--;
  return result;
}
```

4.4.5 Populate

基本上了解了调用Populate的时机，是如果cc里面nonempty里面没有span的话。代码有点长. 这里为了减少阻塞的部分，首先进行解锁然后让全局进行分配。只是针对局部操作没有任何问题。 最后加入nonempty的部分的话这个部分需要加锁。非常巧妙。

```
void CentralFreeList::Populate() {
  // Release central list lock while operating on pageheap
  lock_.Unlock();  // 首先需要计算出我们需要多少个pages
  const size_t npages = Static::sizemap()->class_to_pages(size_class_);

  Span* span;
  {
    SpinLockHolder h(Static::pageheap_lock());
    span = Static::pageheap()->New(npages); // 分配到pages得到span.
    if (span) Static::pageheap()->RegisterSizeClass(span, size_class_);
  }
  if (span == NULL) {
    MESSAGE("tcmalloc: allocation failed", npages << kPageShift);
    lock_.Lock();
    return;
  }
  ASSERT(span->length == npages);
  for (int i = 0; i < npages; i++) { // 将span和size_class之间关联起来
    // 应该是为了后面查找方便，但是现在还不知道有什么用途。但是不影响阅读.
    Static::pageheap()->CacheSizeClass(span->start + i, size_class_);
  }

  // 对这个span里面的所有objects组织成链表形式
  // Split the block into pieces and add to the free-list
  // TODO: coloring of objects to avoid cache conflicts?
  void** tail = &span->objects;
```

```
  char* ptr = reinterpret_cast<char*>(span->start << kPageShift);
  char* limit = ptr + (npages << kPageShift);
  const size_t size = Static::sizemap()->ByteSizeForClass(size_class_);
  int num = 0;
  while (ptr + size <= limit) {
    *tail = ptr;
    tail = reinterpret_cast<void**>(ptr);
    ptr += size;
    num++;
  }
  ASSERT(ptr <= limit);
  *tail = NULL;
  span->refcount = 0; // No sub-object in use yet

  // 将这个span加入nonempty链表的话需要加锁。
  // Add span to list of non-empty spans
  lock_.Lock();
  tcmalloc::DLL_Prepend(&nonempty_, span);
  ++num_spans_;
  counter_ += num;
}
```

4.5 PageHeap

page_heap.h

4.5.1 Data Structure

PageHeap是在page_heap.h里面定义的，主要是用来分配page的。对于PageHeap结构还是比较复杂的.阅读tcmalloc文档也会发现， 管理page的方法和cc是一样的，也是按照page大小做成数组。每个数组的结构是这样的

```
// We segregate spans of a given size into two circular linked
// lists: one for normal spans, and one for spans whose memory
// has been returned to the system.
struct SpanList {
  Span        normal;
  Span        returned; // 其实对于这个部分没有必要区分的，因为代码里面大部分都是挂在normal这个链上的。
};

// List of free spans of length >= kMaxPages
SpanList large_; // 对于>=kMaxPages的页面单独维护一个free list.

// Array mapping from span length to a doubly linked list of free spans
SpanList free_[kMaxPages]; // 针对每个页面大小做的free list.
```

span的状态只有三种，一种是IN_USE表示正在被使用，一种表示ON_NORMAL_FREELIST表示放在了normal freelist上面。 另外一种是ON_RETURNED_FREELIST表示放在returned freelist上面。这里简单地说明一下normal freelist与returned freelist差别。 normal freelist是普通的回收进行缓存起来，而returned freelist表示已经完全unmmap回到系统内存部分了。不过因为实际并没有交回给系统内存， 所以这两个仅仅是概念上面的差别.

另外在PageHeap里面还定义了如何通过PageID查找到Span这个结构，使用了两种方式，一种是Cache,另外一种是radix tree(32位是另外一个结构). 这个会在下面分析

```
// Selector class -- general selector uses 3-level map
template <int BITS> class MapSelector {
 public:
  typedef TCMalloc_PageMap3<BITS-kPageShift> Type;
  typedef PackedCache<BITS-kPageShift, uint64_t> CacheType;
};

  // Pick the appropriate map and cache types based on pointer size
  typedef MapSelector<kAddressBits>::Type PageMap;
  typedef MapSelector<kAddressBits>::CacheType PageMapCache;
  PageMap pagemap_;
  mutable PageMapCache pagemap_cache_;
```

其中kAddressBits的定义在common.h

```
#if defined __x86_64__
// All current and planned x86_64 processors only look at the lower 48 bits
// in virtual to physical address translation.  The top 16 are thus unused.
// TODO(rus): Under what operating systems can we increase it safely to 17?
// This lets us use smaller page maps.  On first allocation, a 36-bit page map
// uses only 96 KB instead of the 4.5 MB used by a 52-bit page map.
static const int kAddressBits = (sizeof(void*) < 8 ? (8 * sizeof(void*)) : 48); // __x86_64__就是64位
#else
static const int kAddressBits = 8 * sizeof(void*);
#endif
```

对于PageHeap比较重要的接口包括下面这些：

- Span* New(Length n); // 分配n个pages并且返回Span对象
- void Delete(Span* span); // 删除Span对象管理的内存
- void RegisterSizeClass(Span* span, size_t sc); // 注册这个span对象管理的slab大小多少(0表示不是用于分配小内存)

- Span* Split(Span* span, Length n); // 将当前的span切分，一个管理n个页面的span,一个是剩余的。
- inline Span* GetDescriptor(PageID p) const //根据PageID得到管理这个Page的Span对象
- void Dump(TCMalloc_Printer* out); // Dump出PageHeap信息
- bool GetNextRange(PageID start, base::MallocRange* r); // 如果page heap管理了>=start的span,那么返回这个信息
- Length ReleaseAtLeastNPages(Length num_pages); // 尝试至少释放num_pages个页面
- size_t GetSizeClassIfCached(PageID p) // 在cache中返回这个page id对应的slab class
- void CacheSizeClass(PageID p, size_t cl) // 在cache中存放page id对应的slab class.

这里有一个点可能有疑问，就是为什么span需要上面标记slab class.原因非常简单，就是如果用户在释放内存的时候，根据ptr查找到对应的span. 然后肯定想知道这个ptr到底应该如何归还，本身带有多少内存。此外还需要注意的是，对于page来说的话，一共管理了(kMaxPages)种页面大小。 tcmalloc代码里面kMaxPages==1 << (20- kPageShift) 相同于有256种页面。但是最后一种页面大小的话可以超过255 pages,这样才可以用于分配大内存。

4.5.2 New

New的逻辑非常简单，首先会尝试在free list里面查找，如果没有的话在lage free list里面查找，不行的话尝试要更多的内存，然后重试。需要注意的是，因为这个是一个全局的操作，所以前面都会加上自选锁 SpinLockHolder h(Static::pageheap_lock());

```
Span* PageHeap::New(Length n) {
  ASSERT(Check());
  ASSERT(n > 0);

  Span* result = SearchFreeAndLargeLists(n);  // free list然后在large里面查找
  if (result != NULL)
    return result;

  // Grow the heap and try again.
  if (!GrowHeap(n)) { // 不行的话尝试分配更多内存
    ASSERT(Check());
    return NULL;
  }
  return SearchFreeAndLargeLists(n); // 然后重新尝试分配
}
```

SearchFreeAndLargeLists相对来说还是比较简单的，但是里面Carve这个需要单独来看

```
Span* PageHeap::SearchFreeAndLargeLists(Length n) {
  ASSERT(Check());
  ASSERT(n > 0);

  // Find first size >= n that has a non-empty list
  for (Length s = n; s < kMaxPages; s++) { // 遍历所有的Pages看看是否有合适的。
    Span* ll = &free_[s].normal;
    // If we're lucky, ll is non-empty, meaning it has a suitable span.
    if (!DLL_IsEmpty(ll)) {
      ASSERT(ll->next->location == Span::ON_NORMAL_FREELIST);
      return Carve(ll->next, n); // 如果有合适的话，那么可能需要切割一下,从里面切割出n pages出来
    }
    // Alternatively, maybe there's a usable returned span.
    ll = &free_[s].returned;
    if (!DLL_IsEmpty(ll)) {
      ASSERT(ll->next->location == Span::ON_RETURNED_FREELIST);
      return Carve(ll->next, n);
    }
  }
  // No luck in free lists, our last chance is in a larger class.
  return AllocLarge(n);  // May be NULL // 如果没有分配成功的话那么从AllocLarge里面分配
}
```

对于AllocLarge部分的话非常简单，就是使用最佳匹配算法。完了之后调用Carve同样进行切割。这里就不贴出代码详细分析。

4.5.3 Carve

我们看看Carve代码，然后在里面的话会稍微粗略地提到pagemap管理span对象的细节

```
Span* PageHeap::Carve(Span* span, Length n) {
  ASSERT(n > 0);
  ASSERT(span->location != Span::IN_USE);
  const int old_location = span->location;
  RemoveFromFreeList(span); // 从freelist里面删除，同时记录信息也会更改。
  span->location = Span::IN_USE; // 修改一下location.
  Event(span, 'A', n);

  const int extra = span->length - n;
  ASSERT(extra >= 0);
  if (extra > 0) {
    Span* leftover = NewSpan(span->start + n, extra); // 创建一个新的span对象
    leftover->location = old_location; // 这个新的对象里面存放到是原来location.
    Event(leftover, 'S', extra);
    RecordSpan(leftover); // 将剩余的span记录下来并且插入到free list里面.
    PrependToFreeList(leftover);  // Skip coalescing - no candidates possible
    span->length = n;
```

```
    pagemap_.set(span->start + n - 1, span); // 同时标记span管理的范围.
  }
  ASSERT(Check());
  return span;
}
```

逻辑可以说非常简单，但是如果之前看过文档的话需要知道这里面pagemap为什么需要set. 非常简单，如果span管理的是[p..q]的范围的话，那么在pagemap里面只需要记录(p,span),(q,span). 这样如果有一个span回收的话，那么在pagemap里面查找p-1和q+1的span,然后尝试合并。非常精巧。 所以在RecordSpan里面很明显就是需要设置前后的边界

```
void RecordSpan(Span* span) {
  pagemap_.set(span->start, span); // 这时span开始
  if (span->length > 1) {
    pagemap_.set(span->start + span->length - 1, span); // 设置span结束
  }
}
```

4.5.4 GrowHeap

GrowHeap就是需要尝试从系统中拿出更多的内存出来然后好做切分，满足本次allocate n pages的请求。 GrowHeap里面有一些策略

```
// 这个就是相当于允许分配的最大Pages
static const Length kMaxValidPages = (~static_cast<Length>(0)) >> kPageShift;
static const int kMinSystemAlloc = kMaxPages; // 调用GrowHeap最小的页数

bool PageHeap::GrowHeap(Length n) {
  ASSERT(kMaxPages >= kMinSystemAlloc);
  if (n > kMaxValidPages) return false;
  Length ask = (n>kMinSystemAlloc) ? n : static_cast<Length>(kMinSystemAlloc); // 会判断是否超过，如果没有超过的话，
  // 那么按照kMinSystemAlloc分配
  size_t actual_size;
  void* ptr = TCMalloc_SystemAlloc(ask << kPageShift, &actual_size, kPageSize);
  if (ptr == NULL) {
    if (n < ask) {
      // Try growing just "n" pages
      ask = n;
      ptr = TCMalloc_SystemAlloc(ask << kPageShift, &actual_size, kPageSize); // 如果ask分配不了，那么尝试分配n
    }
    if (ptr == NULL) return false;
  }
  ask = actual_size >> kPageShift;
  RecordGrowth(ask << kPageShift);

  uint64_t old_system_bytes = stats_.system_bytes;
  stats_.system_bytes += (ask << kPageShift);
  const PageID p = reinterpret_cast<uintptr_t>(ptr) >> kPageShift;
  ASSERT(p > 0);

  // If we have already a lot of pages allocated, just pre allocate a bunch of
  // memory for the page map. This prevents fragmentation by pagemap metadata
  // when a program keeps allocating and freeing large blocks.

  //  static const size_t kPageMapBigAllocationThreshold = 128 << 20;(128MB)
  // 这个地方判断，这次分配是不是已经越过了一个threshold
  // 如果越过的话，那么意味着pagemap里面可能需要分配更多的内存
  // 但是对于64位来说的话，里面没有任何逻辑.
  if (old_system_bytes < kPageMapBigAllocationThreshold
      && stats_.system_bytes >= kPageMapBigAllocationThreshold) {
    pagemap_.PreallocateMoreMemory();
  }

  // Make sure pagemap_ has entries for all of the new pages.
  // Plus ensure one before and one after so coalescing code
  // does not need bounds-checking.
  if (pagemap_.Ensure(p-1, ask+2)) {   // 因为需要插入新的span,所以必须确保这个pagemap确实存在.
    // Pretend the new area is allocated and then Delete() it to cause
    // any necessary coalescing to occur.
    Span* span = NewSpan(p, ask);
    RecordSpan(span);
    Delete(span); // 将这个Span返回给large_里等待下次分配
    ASSERT(Check());
    return true;
  } else {
    // We could not allocate memory within "pagemap_"
    // TODO: Once we can return memory to the system, return the new span
    return false;
  }
}
```

4.5.5 Delete

Delete逻辑非常简单

```
void PageHeap::Delete(Span* span) {
  ASSERT(Check());
  ASSERT(span->location == Span::IN_USE);
  ASSERT(span->length > 0);
  ASSERT(GetDescriptor(span->start) == span);
```

```
  ASSERT(GetDescriptor(span->start + span->length - 1) == span);
  const Length n = span->length;
  span->sizeclass = 0;
  span->sample = 0;
  span->location = Span::ON_NORMAL_FREELIST;
  Event(span, 'D', span->length);
  MergeIntoFreeList(span);  // Coalesces if possible // 会尝试进行合并
  IncrementalScavenge(n); // 增量收集. 后面会仔细看这个函数的定义
  ASSERT(Check());
}
```

里面有两个函数我们需要仔细关心MergeIntoFreeList以及IncrementalScavenge.首先看看MergeIntoFreeList

```
void PageHeap::MergeIntoFreeList(Span* span) {
  ASSERT(span->location != Span::IN_USE);
  const PageID p = span->start;
  const Length n = span->length;
  // 首先尝试合并p-1 pages这个span
  Span* prev = GetDescriptor(p-1);
  if (prev != NULL && prev->location == span->location) {
    // Merge preceding span into this span
    ASSERT(prev->start + prev->length == p);
    const Length len = prev->length;
    RemoveFromFreeList(prev);
    DeleteSpan(prev);
    span->start -= len;
    span->length += len;
    pagemap_.set(span->start, span);
    Event(span, 'L', len);
  }
 // 然后尝试合并p+n pages这个span.
  Span* next = GetDescriptor(p+n);
  if (next != NULL && next->location == span->location) {
    // Merge next span into this span
    ASSERT(next->start == p+n);
    const Length len = next->length;
    RemoveFromFreeList(next);
    DeleteSpan(next);
    span->length += len;
    pagemap_.set(span->start + span->length - 1, span);
    Event(span, 'R', len);
  }
  // 合并完成之后就会放入free list里面去
  PrependToFreeList(span);
}
```

4.5.6 IncrementalScavenge

IncrementalScavenge这个意思就是增量回收，大致内容就是说将一部分的页面交回给系统内存。虽然在tcmalloc里面实现并没有完全交回给系统内存， 而只是简单地挂在了_returned_free_list上面，但是里面的策略还是值得看看的。这里所谓的scavenge_counter_意思就是如果归还了多少内存之后， 那么我们就会尝试进行一次完全交回给系统内存.

```
void PageHeap::IncrementalScavenge(Length n) {
  // Fast path; not yet time to release memory
  scavenge_counter_ -= n;
  if (scavenge_counter_ >= 0) return;  // Not yet time to scavenge

  // 默认值的话是1.0,这个可以有环境变量设置.
  // 如果回收率很低的哈，那么相当于不会归还给系统内存
  const double rate = FLAGS_tcmalloc_release_rate;
  if (rate <= 1e-6) {
    // Tiny release rate means that releasing is disabled.
    //   static const int kDefaultReleaseDelay = 1 << 18;
    scavenge_counter_ = kDefaultReleaseDelay;
    return;
  }

  // 尝试至归还一个页面.
  // 具体这个函数实现在后面会提到.
  Length released_pages = ReleaseAtLeastNPages(1);

  // 如果实际上没有归还的话，那么下次需要等待这么多次之后尝试归还.
  if (released_pages == 0) {
    // Nothing to scavenge, delay for a while.
    scavenge_counter_ = kDefaultReleaseDelay;
  } else { // 否则会按照一定的策略设定次数然后尝试归还
    // Compute how long to wait until we return memory.
    // FLAGS_tcmalloc_release_rate==1 means wait for 1000 pages
    // after releasing one page.
    const double mult = 1000.0 / rate;
    double wait = mult * static_cast<double>(released_pages);
    if (wait > kMaxReleaseDelay) {
      // Avoid overflow and bound to reasonable range.
       // static const int kMaxReleaseDelay = 1 << 20;
      wait = kMaxReleaseDelay;
    }
    scavenge_counter_ = static_cast<int64_t>(wait);
  }
}
```

4.5.7 ReleaseAtLeastNPages

这个函数的语义就是至少尝试释放n pages.实现方式非常简单，每次都从一种pages里面取出一个东西并且进行释放，直到全部释放为止。算是一种round-robin的方式吧，我猜想这样释放的方式对于后面分配的性能影响比较小，每一种大小都释放一些。

```
Length PageHeap::ReleaseAtLeastNPages(Length num_pages) {
  Length released_pages = 0;
  Length prev_released_pages = -1;

  // Round robin through the lists of free spans, releasing the last
  // span in each list.  Stop after releasing at least num_pages.
  while (released_pages < num_pages) {
    if (released_pages == prev_released_pages) { // 如果自上次依赖没有多余释放的话
      // Last iteration of while loop made no progress.
      break;
    }
    prev_released_pages = released_pages;

    for (int i = 0; i < kMaxPages+1 && released_pages < num_pages;
         i++, release_index_++) { // 每个大小类型都会尝试释放一个.
      if (release_index_ > kMaxPages) release_index_ = 0;
      SpanList* slist = (release_index_ == kMaxPages) ?
          &large_ : &free_[release_index_];
      if (!DLL_IsEmpty(&slist->normal)) {
        Length released_len = ReleaseLastNormalSpan(slist);
        released_pages += released_len;
      }
    }
  }
  return released_pages;
}
```

然后我们看看ReleaseLastNormalSpan这个过程，非常简单

```
Length PageHeap::ReleaseLastNormalSpan(SpanList* slist) {
  Span* s = slist->normal.prev;
  ASSERT(s->location == Span::ON_NORMAL_FREELIST);
  RemoveFromFreeList(s); // 从当前链中释放掉.
  const Length n = s->length;
  // 实际上这个部分并没有释放哦.
  TCMalloc_SystemRelease(reinterpret_cast<void*>(s->start << kPageShift),
                         static_cast<size_t>(s->length << kPageShift));
  s->location = Span::ON_RETURNED_FREELIST; // 标记为returned状态
   // 丢回return free list时候会尝试合并.
  MergeIntoFreeList(s);  // Coalesces if possible.
  return n;
}
```

4.5.8 Split

Split过程和Carve过程是非常相似的，只不过Split针对的是IN_USE状态的这种span. 代码阅读到这里暂时还不知道这个Split什么时候调用:(.what a shame.

4.5.9 GetNextRange

得到page id >=start的span的具体内容。首先看看MallocRange的内容

```
struct MallocRange {
  // 这个malloc范围是什么类型
  enum Type {
    INUSE,                // Application is using this range
    FREE,                 // Range is currently free
    UNMAPPED,             // Backing physical memory has been returned to the OS
    UNKNOWN,
    // More enum values may be added in the future
  };
  // 地址，长度，类型
  uintptr_t address;    // Address of range
  size_t length;        // Byte length of range
  Type type;            // Type of this range
  // =0 !INUSE,如果=1表示这个被当做page使用
  // 如果[0,1]之间的话，表明被做成了小对象分配
  double fraction;      // Fraction of range that is being used (0 if !INUSE)
};
```

然后来看看这个过程

```
bool PageHeap::GetNextRange(PageID start, base::MallocRange* r) {
  Span* span = reinterpret_cast<Span*>(pagemap_.Next(start));
  if (span == NULL) {
    return false;
  }
  r->address = span->start << kPageShift;
  r->length = span->length << kPageShift;
  r->fraction = 0;
  switch (span->location) {
```

```
    case Span::IN_USE:
      r->type = base::MallocRange::INUSE;
      r->fraction = 1;
      if (span->sizeclass > 0) {
        // Only some of the objects in this span may be in use.
        const size_t osize = Static::sizemap()->class_to_size(span->sizeclass); // 首先知道这个class每个object size多少
       // refcount表示已经使用了多少个objects.,这样就可以得到使用率
        r->fraction = (1.0 * osize * span->refcount) / r->length;
      }
      break;
    case Span::ON_NORMAL_FREELIST:
      r->type = base::MallocRange::FREE;
      break;
    case Span::ON_RETURNED_FREELIST:
      r->type = base::MallocRange::UNMAPPED;
      break;
    default:
      r->type = base::MallocRange::UNKNOWN;
      break;
  }
  return true;
}
```

4.6 TCMalloc_PageMap3

page_map.h

之前pageheap里面可以看到有这么一个要求，就是从一个page ID映射到span这么一个过程。在64位下面的话逻辑地址空间有1 << 64，如果按照4K per page计算的话，那么最多会存在1<<52个page.如果使用数组存储的话那么是会存在问题的。所以这里使用了radix tree来进行映射。 对于64位的话使用了3-level radix tree.每段分别是(18,18,16)

```
// How many bits should we consume at each interior level
static const int INTERIOR_BITS = (BITS + 2) / 3; // Round-up
static const int INTERIOR_LENGTH = 1 << INTERIOR_BITS;

// How many bits should we consume at leaf level
static const int LEAF_BITS = BITS - 2*INTERIOR_BITS;
static const int LEAF_LENGTH = 1 << LEAF_BITS;
```

对于一个地址映射称为每一个level的number index的函数可以参看get这个方法

```
void* get(Number k) const {
  const Number i1 = k >> (LEAF_BITS + INTERIOR_BITS);
  const Number i2 = (k >> LEAF_BITS) & (INTERIOR_LENGTH-1);
  const Number i3 = k & (LEAF_LENGTH-1);
  if ((k >> BITS) > 0 ||
      root_->ptrs[i1] == NULL || root_->ptrs[i1]->ptrs[i2] == NULL) {
    return NULL;
  }
  return reinterpret_cast<Leaf*>(root_->ptrs[i1]->ptrs[i2])->values[i3];
}
```

初次之外，这个pagemap还有两个比较重要的接口

- bool Ensure(Number start, size_t n)

因为get,set接口的话都是假设每一层对应的array都是存在的，所以基本上在调用之前的话都必须确保这个array存在。 而Ensure就是做这件事情的，确保[start,start+n-1]这些PageId对应的每一层array都存在。

- void* Next(Number k) const

Next接口就纯粹想知道>=k的这些PageId首先映射到的span对象是什么，实现起来非常巧妙可以仔细阅读一下

```
void* Next(Number k) const {
  while (k < (Number(1) << BITS)) {
    const Number i1 = k >> (LEAF_BITS + INTERIOR_BITS);
    const Number i2 = (k >> LEAF_BITS) & (INTERIOR_LENGTH-1);
    if (root_->ptrs[i1] == NULL) { // 如果这层为空的话，那么直接跳到下一层
      // Advance to next top-level entry
      k = (i1 + 1) << (LEAF_BITS + INTERIOR_BITS);
    } else {
      Leaf* leaf = reinterpret_cast<Leaf*>(root_->ptrs[i1]->ptrs[i2]);
      if (leaf != NULL) {
        for (Number i3 = (k & (LEAF_LENGTH-1)); i3 < LEAF_LENGTH; i3++) { // 遍历这一层(第三层)看看是否存在.
          if (leaf->values[i3] != NULL) {
            return leaf->values[i3];
          }
        }
      }
      // Advance to next interior entry
      k = ((k >> LEAF_BITS) + 1) << LEAF_BITS; // 如果第二层为空的话，那么同样进入下一层.
    }
  }
  return NULL;
}
```

4.7 PackedCache

packed-cache-inl.h

PackedCache是一种非常精巧的数据结构。它的作用主要是想知道对于一个pageId所管理的span而言的话，对应的sizeclass是什么。 在pageheap里面是这样定义的 typedef PackedCache<BITS-kPageShift, uint64_t> CacheType; 我们还是看看这个结构是什么样的

```
template <int kKeybits, typename T>
class PackedCache {
 public:
  typedef uintptr_t K;
  typedef size_t V;
#ifdef TCMALLOC_SMALL_BUT_SLOW
  // Decrease the size map cache if running in the small memory mode.
  static const int kHashbits = 12;
#else
  static const int kHashbits = 16;
#endif
  // array_ is the cache.  Its elements are volatile because any
  // thread can write any array element at any time.
  volatile T array_[1 << kHashbits];
};
```

首先它还是一个KV结构，只不过K+V大小可以放在sizeof(T)字节里面。回顾一下对于64位而言，PageId 52位，而sizeclass只有85中，完全可以存放在sizeof(uint64_t)里面。 将K放在高字节，而V放在低字节，组成一个<sizeof(uint64_t)大小的值存放在array_里面。此外还需要注意一个问题就是，这个有可能被多线程访问， 但是如果我们将这个内容设置称为volatile的话，那么是不需要加锁就可以完成的。

4.8 Thread Cache

thread_cache.h

4.8.1 Data Structure

Thread Cache就是每一个线程里面管理小对象分配的cache.tcmalloc应该是假设局部线程里面通常分配的都是小对象，这样可以减少锁竞争。 而如果是分配大对象的话，那么会直接从page heap里面进行分配。如果本地小对象不够的话，那么会尝试从central cache里面要。Thread Cache比较重要的接口有下面这些：

- void Init(pthread_t tid); // 初始化
- void Cleanup();
- void* Allocate(size_t size, size_t cl); // 从class里面分配size大小
- void Deallocate(void* ptr, size_t size_class); // 将ptr放回class对应slab里面
- void Scavenge(); // 回收内存到central cache.就是文档里面说的GC
- bool SampleAllocation(size_t k); // 是否认为这次分配的k字节需要进行采样.

还有一些静态方法也非常值得关注

- InitModule // 初始化模块
- InitTSD // 初始化thread storage data.
- GetThreadHeap // thread cache.
- GetCache // tc
- GetCacheIfPresent // tc
- CreateCacheIfNecessary // 如果tc不存在就创建
- BecomeIdle // 标记这个thread已经idle，所以可以释放这个tc了

涉及到的静态变量有下面这些

```
namespace tcmalloc {

static bool phinited = false;

volatile size_t ThreadCache::per_thread_cache_size_ = kMaxThreadCacheSize; // 每个tc的大小 (4 << 20,4MB)
size_t ThreadCache::overall_thread_cache_size_ = kDefaultOverallThreadCacheSize;// 所有tc大小 (8 * kMaxThreadCacheSize = 32MB)
ssize_t ThreadCache::unclaimed_cache_space_ = kDefaultOverallThreadCacheSize;  // 管理对象所持有的tc大小(相当于总tc里面还有多少可用).
// (= overall_thread_cache_size_ - sum(tc.max_size))
PageHeapAllocator<ThreadCache> threadcache_allocator; // tc sample alloc.
ThreadCache* ThreadCache::thread_heaps_ = NULL; // tc链.
int ThreadCache::thread_heap_count_ = 0; // 多少个tc
ThreadCache* ThreadCache::next_memory_steal_ = NULL; // 下一次steal的tc.
bool ThreadCache::tsd_inited_ = false; // 是否已经初始化了线程局部数据
pthread_key_t ThreadCache::heap_key_; // 如果使用pthread线程局部数据解决办法

}
```

4.8.2 InitModule

```
void ThreadCache::InitModule() {
  SpinLockHolder h(Static::pageheap_lock());  // 全局自选锁
```

```
  if (!phinited) {
    Static::InitStaticVars(); // 初始化一些静态数据
    threadcache_allocator.Init(); // PageHeapAllocator<ThreadCache>,sample_alloc初始化
    phinited = 1;
  }
}
```

4.8.3 InitTSD

```
void ThreadCache::InitTSD() {
  ASSERT(!tsd_inited_); // 这个变量标记是否已经初始化了线程局部变量，如果没有的话那么是没有任何tc的.
  perftools_pthread_key_create(&heap_key_, DestroyThreadCache); // 这个就是设置好线程局部变量
  // 因为每一个线程都会有一个线程局部变量thread cache.
  tsd_inited_ = true;
}
```

然后我们看看DestroyThreadCache.很容易想到其实这个方法就是销毁掉线程的tc

```
void ThreadCache::DestroyThreadCache(void* ptr) {
  // Note that "ptr" cannot be NULL since pthread promises not
  // to invoke the destructor on NULL values, but for safety,
  // we check anyway.
  if (ptr == NULL) return;
  DeleteCache(reinterpret_cast<ThreadCache*>(ptr));
}
```

我们可能会很想看看这个调用InitTSD的时机是什么？这个是放在一个全局静态变量里面一起调用的。之前已经提到了TCMallocGuard

4.8.4 GetCache

关于GetCache我们也可以一起看看GetThreadHeap,GetCacheIfPresent,CreateCacheIfNecessary

```
inline ThreadCache* ThreadCache::GetCache() {
  ThreadCache* ptr = NULL;
  if (!tsd_inited_) {
    InitModule(); // 初始化模块
  } else {
    ptr = GetThreadHeap(); // 直接查看是否存在
  }
  if (ptr == NULL) ptr = CreateCacheIfNecessary(); // 如果不存在的话那么就创建
  return ptr;
}
```

GetThreadHeap非常简单直接从线程局部变量里面取出即可

```
inline ThreadCache* ThreadCache::GetThreadHeap() {
  return reinterpret_cast<ThreadCache *>(
      perftools_pthread_getspecific(heap_key_));
}
inline ThreadCache* ThreadCache::GetCacheIfPresent() {
  if (!tsd_inited_) return NULL;
  return GetThreadHeap();
}
```

4.8.5 CreateCacheIfNecessary

然后看看CreateCacheIfNecessary这个实现,看看是如何创建tc的

```
ThreadCache* ThreadCache::CreateCacheIfNecessary() {
  // Initialize per-thread data if necessary
  ThreadCache* heap = NULL;
  {
    SpinLockHolder h(Static::pageheap_lock());
    const pthread_t me = pthread_self();
    // 查找里面是否已经存在,每个线程都创建一个ThreadCache.
    // 并且这个是按照链组织起来的。
    for (ThreadCache* h = thread_heaps_; h != NULL; h = h->next_) {
      if (h->tid_ == me) {
        heap = h;
        break;
      }
    }
    if (heap == NULL) heap = NewHeap(me);
  }
  if (!heap->in_setspecific_ && tsd_inited_) {
    heap->in_setspecific_ = true; // 避免setspecific里面还调用
    perftools_pthread_setspecific(heap_key_, heap);
    heap->in_setspecific_ = false;
  }
  return heap;
}
```

4.8.6 NewHeap

NewHeap是产生一个新的tc调用Init.将这个tc插入到队列里面.注意这里NewHeap已经加了锁了。

```
ThreadCache* ThreadCache::NewHeap(pthread_t tid) {
  // Create the heap and add it to the linked list
  ThreadCache *heap = threadcache_allocator.New();
  heap->Init(tid); // 调用Init
  heap->next_ = thread_heaps_; // 组织成为一个双向链表
  heap->prev_ = NULL;
  if (thread_heaps_ != NULL) {
    thread_heaps_->prev_ = heap;
  } else {
    // This is the only thread heap at the momment.
    ASSERT(next_memory_steal_ == NULL);
    next_memory_steal_ = heap; // 如果这个是第一个元素的话，那么设置next_memory_steal.
  }
  thread_heaps_ = heap;
  thread_heap_count_++; // tc数量.
  return heap;
}
```

4.8.7 BecomeIdle

BecomeIdle触发条件现在还不是很清楚，但是作用是认为这个tc没有必要了可以删除。不过在大部分使用应该不会有这个调用吧。

```
void ThreadCache::BecomeIdle() {
  if (!tsd_inited_) return;               // No caches yet
  ThreadCache* heap = GetThreadHeap();
  if (heap == NULL) return;             // No thread cache to remove
  if (heap->in_setspecific_) return;    // Do not disturb the active caller

  heap->in_setspecific_ = true; // 防止递归调用
  perftools_pthread_setspecific(heap_key_, NULL);
  heap->in_setspecific_ = false;
  if (GetThreadHeap() == heap) { // 应该是不会调用这个部分逻辑的.
    // Somehow heap got reinstated by a recursive call to malloc
    // from pthread_setspecific.  We give up in this case.
    return;
  }
  // 然后将这个heap释放掉
  // We can now get rid of the heap
  DeleteCache(heap);
}
```

这里我想到一个问题，就是如果不断地启动线程然后关闭线程，如果tid是不允许复用的话那么会导致thread_cache不断地开辟。 如果使用gettid的话那么可能会有这个情况，而如果用pthread_self的话可能就不会有了(至少从程序上看可以复用)

```
#include <cstdio>
#include <pthread.h>

char buf[1024*1024];
void* foo(void* arg){
  return NULL;
}
int main() {
  pthread_t tid;
  for(int i=0;i<10;i++){
    pthread_create(&tid,NULL,foo,NULL);
    pthread_join(tid,NULL);
    printf("%zu\n",static_cast<size_t>(tid));
    pthread_create(&tid,NULL,foo,NULL);
    pthread_join(tid,NULL);
    printf("%zu\n",static_cast<size_t>(tid));
  }
  return 0;
}
```

从程序运行结果来看的话都是一样的tid.

4.8.8 Init

注意这里Init已经在外围的NewHeap加锁了。这个地方进行初始化。设置一下最大分配多少空间以及初始化每一个slab

```
void ThreadCache::Init(pthread_t tid) {
  size_ = 0;

  max_size_ = 0;
  IncreaseCacheLimitLocked(); // 这个地方在计算到底可以分配多少max size.
  if (max_size_ == 0) {
    // There isn't enough memory to go around.  Just give the minimum to
    // this thread.
    // static const size_t kMaxSize    = 256 * 1024;(256K)
    // static const size_t kMinThreadCacheSize = kMaxSize * 2;(512K)
    max_size_ = kMinThreadCacheSize; // 512K.

    // Take unclaimed_cache_space_ negative.
    unclaimed_cache_space_ -= kMinThreadCacheSize; // 那么相当于tc持有空闲空间也对应减少
    ASSERT(unclaimed_cache_space_ < 0);
  }

  next_ = NULL;
```

```
  prev_ = NULL;
  tid_  = tid;
  in_setspecific_ = false;
  for (size_t cl = 0; cl < kNumClasses; ++cl) {
    list_[cl].Init(); // 初始化每个slab
  }

  uint32_t sampler_seed;
  memcpy(&sampler_seed, &tid, sizeof(sampler_seed));
  sampler_.Init(sampler_seed); // 初始化sampler
}
```

这里我们有两个问题没有搞懂，一个是slab到底结构是怎么样的，一个就是IncreaseCacheLimitLocked里面是如何计算max_size_的。

4.8.9 ThreadCache::FreeList

freelist就是对应的slab.本质上数据结构就是一个单向链表，毕竟这个分配对于顺序没有任何要求。

```
class FreeList {
 private:
  void*    list_;       // Linked list of nodes

  // On 64-bit hardware, manipulating 16-bit values may be slightly slow.
  uint32_t length_;      // Current length. // 当前长度多少
  uint32_t lowater_;     // Low water mark for list length. // 长度最少时候达到了多少
  uint32_t max_length_;  // Dynamic max list length based on usage. // 认为的最大长度多少
  // Tracks the number of times a deallocation has caused
  // length_ > max_length_.  After the kMaxOverages'th time, max_length_
  // shrinks and length_overages_ is reset to zero.
  uint32_t length_overages_; // 超过最大长度的次数
};
```

所有的这些参数其实都是为了进行方便做一些策略。

4.8.10 IncreaseCacheLimitLocked

之前说到这个函数是在计算这个tc里面最多可以分配多少内存，那么看看这个函数的实现.调用这个函数的时候必然都是已经加了自旋锁的。

```
void ThreadCache::IncreaseCacheLimitLocked() {
  if (unclaimed_cache_space_ > 0) { // 如果tc里面还有空闲的内容的话，那么获取64KB过来
    // static const size_t kStealAmount = 1 << 16;(64KB)
    // Possibly make unclaimed_cache_space_ negative.
    unclaimed_cache_space_ -= kStealAmount;
    max_size_ += kStealAmount;
    return;
  }
  // 如果发现依然不够的话，那么会从每一个以后的tc里面获取偷取部分出来.
  // 这个链是按照next_memory_steal_取出来的，如果==NULL那么从头开始。
  // 但是很快会发现这个max_size其实并不是一成不变的.
  // Don't hold pageheap_lock too long.  Try to steal from 10 other
  // threads before giving up.  The i < 10 condition also prevents an
  // infinite loop in case none of the existing thread heaps are
  // suitable places to steal from.
  for (int i = 0; i < 10;
       ++i, next_memory_steal_ = next_memory_steal_->next_) {
    // Reached the end of the linked list.  Start at the beginning.
    if (next_memory_steal_ == NULL) {
      ASSERT(thread_heaps_ != NULL);
      next_memory_steal_ = thread_heaps_;
    }
    if (next_memory_steal_ == this ||
        next_memory_steal_->max_size_ <= kMinThreadCacheSize) {
      continue;
    }
    next_memory_steal_->max_size_ -= kStealAmount;
    max_size_ += kStealAmount;

    next_memory_steal_ = next_memory_steal_->next_;
    return;
  }
}
```

总之tc的max_size分配策略的话就是根据当前所有tc剩余的空间，如果没有空间的话那么尝试从其他的tc里面获取。应该是想限制一开始每个tc的最大大小。 但是需要注意的是，这个tc最大大小并不是一成不变的，可能会随着时间变化而增加。

4.8.11 DeleteCache

DeleteCache作用就是删除一个tc.大致逻辑非常简单，首先将自己持有的内存归还给central cache,然后将自己从tc的链中删除即可。

```
void ThreadCache::DeleteCache(ThreadCache* heap) {
  // Remove all memory from heap
  heap->Cleanup(); // 稍后我们查看Cleanup实现。

  // Remove from linked list
```

```
  SpinLockHolder h(Static::pageheap_lock());
  if (heap->next_ != NULL) heap->next_->prev_ = heap->prev_;
  if (heap->prev_ != NULL) heap->prev_->next_ = heap->next_;
  if (thread_heaps_ == heap) thread_heaps_ = heap->next_;
  thread_heap_count_--;

  if (next_memory_steal_ == heap) next_memory_steal_ = heap->next_;
  if (next_memory_steal_ == NULL) next_memory_steal_ = thread_heaps_;
  unclaimed_cache_space_ += heap->max_size_;

  threadcache_allocator.Delete(heap);
}
```

将自己删除之后需要重新计算thread_heaps以及next_memory_steal这两个变量。

4.8.12 Cleanup

Cleanup是在DeleteCache，会在BecomeIdle里面可以调用，也会在销毁线程局部变量里面调用。作用就是将自己持有的内存归还给系统

```
void ThreadCache::Cleanup() {
  // Put unused memory back into central cache
  for (int cl = 0; cl < kNumClasses; ++cl) {
    if (list_[cl].length() > 0) {
      ReleaseToCentralCache(&list_[cl], cl, list_[cl].length());
    }
  }
}
```

遍历所有的slab并且将上面挂在的free list归还给central cache.这个在ReleaseToCentralCache里面调用

4.8.13 ReleaseToCentralCache

```
void ThreadCache::ReleaseToCentralCache(FreeList* src, size_t cl, int N) {
  ASSERT(src == &list_[cl]);
  if (N > src->length()) N = src->length(); // 这个地方感觉不是很有必要.不过其他地方的话可能这两个参数不同
  size_t delta_bytes = N * Static::sizemap()->ByteSizeForClass(cl); // 了解有多少个对象占用内存大小释放.

  // We return prepackaged chains of the correct size to the central cache.
  // TODO: Use the same format internally in the thread caches?
  int batch_size = Static::sizemap()->num_objects_to_move(cl);
  while (N > batch_size) { // 每次归还batch_size个内容，这样central cache可以放在transfer cache里面
    void *tail, *head;
    src->PopRange(batch_size, &head, &tail);
    Static::central_cache()[cl].InsertRange(head, tail, batch_size);
    N -= batch_size;
  }
  void *tail, *head;
  src->PopRange(N, &head, &tail);
  Static::central_cache()[cl].InsertRange(head, tail, N);
  size_ -= delta_bytes;
}
```

PopRange这个语义非常简单，但是我们稍微看看这个的实现，

```
void PopRange(int N, void **start, void **end) {
  SLL_PopRange(&list_, N, start, end);
  ASSERT(length_ >= N);
  length_ -= N;
  if (length_ < lowater_) lowater_ = length_;
}
```

问题就在于，这里设置了lowater mark.如果当前的长度小于最低水位的话，那么需要更新最低水位。

4.8.14 Allocate

Allocate就是从对应的slab里面分配出一个object.注意在Init时候的话每个tc里面是没有任何内容的，size_=0.FreeList也是空的。

```
inline void* ThreadCache::Allocate(size_t size, size_t cl) {
  ASSERT(size <= kMaxSize);
  ASSERT(size == Static::sizemap()->ByteSizeForClass(cl));

  FreeList* list = &list_[cl];
  if (list->empty()) {
    return FetchFromCentralCache(cl, size); // 如果list里面为空的话，那么尝试从cc的cl里面分配size出来.
  }
  size_ -= size; // 如果存在的话那么就直接-size并且弹出一个元素
  return list->Pop();
}
```

4.8.15 FetchFromCentralCache

这个部分的逻辑是从cc里面取出一系列的slab对象出来。里面有很多策略，非常精巧

```
void* ThreadCache::FetchFromCentralCache(size_t cl, size_t byte_size) {
  FreeList* list = &list_[cl];
```

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

  // 看看每次允许的分配的个数是多少
  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);

  ASSERT((start == NULL) == (fetch_count == 0));
  // 取出来并且设置一下当前维护的空闲大小是多少
  if (--fetch_count >= 0) {
    size_ += byte_size * fetch_count;
    list->PushRange(fetch_count, SLL_Next(start), end);
  }
  // 这里需要增长max_length.如果<batch_size的话那么+1
  // 如果>=batch_size的话，那么会设置成为某个上线
  // static const int kMaxDynamicFreeListLength = 8192;
  if (list->max_length() < batch_size) {
    list->set_max_length(list->max_length() + 1);
  } else {
    int new_length = min<int>(list->max_length() + batch_size,
                              kMaxDynamicFreeListLength);
    // 这里也非常好理解，按照batch_size来分配的话，可以直接从tc里面得到
    // 使用这个作为max_kength的话通常意味着分配速度会更快.
    new_length -= new_length % batch_size;
    ASSERT(new_length % batch_size == 0);
    list->set_max_length(new_length);
  }
  return start;
}
```

4.8.16 Deallocate

释放内存部分非常简单，但是同样里面有很多策略。并且里面涉及到了tc的GC问题

```
inline void ThreadCache::Deallocate(void* ptr, size_t cl) {
  FreeList* list = &list_[cl];
  size_ += Static::sizemap()->ByteSizeForClass(cl); // 释放了这个内存所以空闲大小增大
  ssize_t size_headroom = max_size_ - size_ - 1;  // 在size上面的话还有多少空闲.

  list->Push(ptr); // 归还
  ssize_t list_headroom =
      static_cast<ssize_t>(list->max_length()) - list->length(); // 在长度上还有多少空闲

  // There are two relatively uncommon things that require further work.
  // In the common case we're done, and in that case we need a single branch
  // because of the bitwise-or trick that follows.
  if ((list_headroom | size_headroom) < 0) { // 这个部分应该是有任意一个<0的话，那么就应该进入。优化手段吧.
    if (list_headroom < 0) { // 如果当前长度>max_length的话，那么需要重新设置max_length.
      ListTooLong(list, cl);
    }
    // 条件相当 if(size_headroom < 0)
    // 因为ListTooLog会尝试修改size_所以这里重新判断..:(tricky:(.
    if (size_ >= max_size_) Scavenge(); // 如果当前size>max_size的话，那么需要进行GC.
  }
}
```

然后我们这里看看这两个触发动作时如何执行的。

4.8.17 ListTooLong

到这个地方必须思考一个问题，就是什么时候max_length会发生变化以及如何变化的(触发这些变化的意义是什么). 我们可以看到Allocate里面如果从cc里面取在不断地增加max_length(存在上限).问题是我们不能够让这个部分缓存太多的内容， 所以我们必须在一段时间内缩小max_length，一旦length>max_length的话就会触发ListTooLong. 而ListTooLong里面的操作就是将max_length尝试缩小并且将一部分object归还给cc.

```
void ThreadCache::ListTooLong(FreeList* list, size_t cl) {
  const int batch_size = Static::sizemap()->num_objects_to_move(cl);
  ReleaseToCentralCache(list, cl, batch_size); // 首先尝试将batch_size的内容归还到tc里面取

  // 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); // 记录下overage的次数
    if (list->length_overages() > kMaxOverages) { // > kMaxOverages的话那么需要对max_length进行缩减.
      ASSERT(list->max_length() > batch_size);
      list->set_max_length(list->max_length() - batch_size); // 缩减batch_size.
```

```
      list->set_length_overages(0);
    }
  }
}
```

ListTooLong是第一个确保在tc里面不会持有太多内存的机制.虽然对这里的整个过程算是比较了解，但是没有从大体上想清楚这个是如何设计的:(

4.8.18 Scavenge

同样Scavenge是第二个确保在tc里不会持有太多内存的机制。同样虽然对这个过程比较了解但是也没有从大体生了解这个策略是如何设计出来的。

```
// Release idle memory to the central cache
void ThreadCache::Scavenge() {
  // If the low-water mark for the free list is L, it means we would
  // not have had to allocate anything from the central cache even if
  // we had reduced the free list size by L.  We aim to get closer to
  // that situation by dropping L/2 nodes from the free list.  This
  // may not release much memory, but if so we will call scavenge again
  // pretty soon and the low-water marks will be high on that call.
  //int64 start = CycleClock::Now();
  for (int cl = 0; cl < kNumClasses; cl++) {
    FreeList* list = &list_[cl];
    const int lowmark = list->lowwatermark(); // 上一次最短的free list length是多少.如果free list length越长
    // 意味着在大多数时候有很多空闲内存是没有使用，所以可以将其归还.
    if (lowmark > 0) {
      const int drop = (lowmark > 1) ? lowmark/2 : 1; // 将最最短的部分的1/2归还给cc.
      ReleaseToCentralCache(list, cl, drop);

      // Shrink the max length if it isn't used.  Only shrink down to
      // batch_size -- if the thread was active enough to get the max_length
      // above batch_size, it will likely be that active again.  If
      // max_length shinks below batch_size, the thread will have to
      // go through the slow-start behavior again.  The slow-start is useful
      // mainly for threads that stay relatively idle for their entire
      // lifetime.
      const int batch_size = Static::sizemap()->num_objects_to_move(cl);
      if (list->max_length() > batch_size) { // 调整max_length.
        list->set_max_length(
            max<int>(list->max_length() - batch_size, batch_size));
      }
    }
    list->clear_lowwatermark();
  }

  IncreaseCacheLimit(); // 触发这个Scavenge本身的原因就是因为size_>max_size_所以有必要提高max_size_.
}
```

5 用户对象

tcmalloc.h

5.1 函数入口

我们还是以最初的函数入门进行分析，我们只是关注tc_malloc与tc_free.

```
extern "C" PERFTOOLS_DLL_DECL void* tc_malloc(size_t size) __THROW {
  void* result = do_malloc_or_cpp_alloc(size);
  MallocHook::InvokeNewHook(result, size);
  return result;
}

extern "C" PERFTOOLS_DLL_DECL void tc_free(void* ptr) __THROW {
  MallocHook::InvokeDeleteHook(ptr);
  do_free(ptr);
}
```

可以看到两个函数调用之前都有hook存在。hook是在malloc_hook_inl.h以及malloc_hook.cc里面定义的，通过一个HookList来进行管理。 调用Invoke时候就是遍历里面的内容，这个后续可以仔细分析。do_malloc_or_cpp_alloc里面可以看到，因为tc_new_mode==0所以实际调用的就是do_malloc这个函数。 我们首先关注malloc的过程，对于malloc过程了解清楚之后，那么free过程就非常直接了。

5.2 分配逻辑

我们先看看do_malloc这个过程

```
inline void* do_malloc(size_t size) {
  void* ret = NULL;

  // The following call forces module initialization
  ThreadCache* heap = ThreadCache::GetCache(); // 首先得到thread_cache
  if (size <= kMaxSize) { // kMaxSize = 256K
    size_t cl = Static::sizemap()->SizeClass(size);
    size = Static::sizemap()->class_to_size(cl);
```

```
     // 尝试进行采样分配.
     // 这里我们暂时忽略采样部分的逻辑
    if ((FLAGS_tcmalloc_sample_parameter > 0) && heap->SampleAllocation(size)) {
      ret = DoSampledAllocation(size);
    } else {
      // The common case, and also the simplest.  This just pops the
      // size-appropriate freelist, after replenishing it if it's empty.
      ret = CheckedMallocResult(heap->Allocate(size, cl)); // 这个部分的就是直接在tc上面调用Allocate进行分配
    }
  } else {
    ret = do_malloc_pages(heap, size); // 如果分配对象过大的话
  }
  if (ret == NULL) errno = ENOMEM;
  return ret;
}
```

对于小对象分配逻辑已经清楚了，接着看看大对象分配调用do_malloc_pages这个部分

```
inline void* do_malloc_pages(ThreadCache* heap, size_t size) {
  void* result;
  bool report_large;

  Length num_pages = tcmalloc::pages(size); // 转换需要分配多少个pages.
  size = num_pages << kPageShift;

  if ((FLAGS_tcmalloc_sample_parameter > 0) && heap->SampleAllocation(size)) { // 同样我们暂时忽略采样部分
    result = DoSampledAllocation(size);

    SpinLockHolder h(Static::pageheap_lock());
    report_large = should_report_large(num_pages);
  } else {
    SpinLockHolder h(Static::pageheap_lock());
    Span* span = Static::pageheap()->New(num_pages);
    result = (span == NULL ? NULL : SpanToMallocResult(span)); // 这个部分就是检查一下span是否OK,已经将span的slab(0)cache住.
    report_large = should_report_large(num_pages);  // 判断这个pages是否开辟过大
  }

  if (report_large) {
    ReportLargeAlloc(num_pages, result); // 如果开辟过大的话那么可以选择进行report.
  }
  return result;
}
```

然后稍微看看should_report_large是如何判断的以及如何report

```
// 通过获取环境变量即可得到
const int64 kDefaultLargeAllocReportThreshold = static_cast<int64>(1) << 30; // 默认是1GB
DEFINE_int64(tcmalloc_large_alloc_report_threshold,
             EnvToInt64("TCMALLOC_LARGE_ALLOC_REPORT_THRESHOLD",
                        kDefaultLargeAllocReportThreshold),
             "Allocations larger than this value cause a stack "
             "trace to be dumped to stderr.  The threshold for "
             "dumping stack traces is increased by a factor of 1.125 "
             "every time we print a message so that the threshold "
             "automatically goes up by a factor of ~1000 every 60 "
             "messages.  This bounds the amount of extra logging "
             "generated by this flag.  Default value of this flag "
             "is very large and therefore you should see no extra "
             "logging unless the flag is overridden.  Set to 0 to "
             "disable reporting entirely.");

// 这个large_alloc_threshold肯定要比kPageSize要打
static int64_t large_alloc_threshold =
  (kPageSize > FLAGS_tcmalloc_large_alloc_report_threshold
   ? kPageSize : FLAGS_tcmalloc_large_alloc_report_threshold);

inline bool should_report_large(Length num_pages) {
  const int64 threshold = large_alloc_threshold;
  if (threshold > 0 && num_pages >= (threshold >> kPageShift)) { // 如果超过large_alloc_threshold的话
    // Increase the threshold by 1/8 every time we generate a report.
    // We cap the threshold at 8GiB to avoid overflow problems.
    // 那么这次的threshold可能需要进行调整
    large_alloc_threshold = (threshold + threshold/8 < 8ll<<30 // 8GB
                             ? threshold + threshold/8 : 8ll<<30);
    return true;
  }
  return false;
}
```

然后看看如果进行report的.代码上看基本上就是打印出这个函数调用堆栈到stderr上面，使用的buffer空间1000B.

```
static void ReportLargeAlloc(Length num_pages, void* result) {
  StackTrace stack;
  stack.depth = GetStackTrace(stack.stack, tcmalloc::kMaxStackDepth, 1);

  static const int N = 1000;
  char buffer[N];
  TCMalloc_Printer printer(buffer, N);
  printer.printf("tcmalloc: large alloc %"PRIu64" bytes == %p @ ",
                 static_cast<uint64>(num_pages) << kPageShift,
                 result);
```

```
  for (int i = 0; i < stack.depth; i++) {
    printer.printf(" %p", stack.stack[i]);
  }
  printer.printf("\n");
  write(STDERR_FILENO, buffer, strlen(buffer));
}
```

5.3 释放逻辑

相对分配来说，释放逻辑要稍微简单一些.

```
inline void do_free_with_callback(void* ptr, void (*invalid_free_fn)(void*)) {
  if (ptr == NULL) return;
  ASSERT(Static::pageheap() != NULL);  // Should not call free() before malloc()
  const PageID p = reinterpret_cast<uintptr_t>(ptr) >> kPageShift;
  Span* span = NULL;
  size_t cl = Static::pageheap()->GetSizeClassIfCached(p); // 首先查看cache里面是否有class的信息

  if (cl == 0) { // 如果没有class的信息的话，那么需要去pagemap里面查询到span.
    span = Static::pageheap()->GetDescriptor(p);
    if (!span) { // 如果查询不到span的话那么认为这个指针式错误的
      // span can be NULL because the pointer passed in is invalid
      // (not something returned by malloc or friends), or because the
      // pointer was allocated with some other allocator besides
      // tcmalloc.  The latter can happen if tcmalloc is linked in via
      // a dynamic library, but is not listed last on the link line.
      // In that case, libraries after it on the link line will
      // allocate with libc malloc, but free with tcmalloc's free.
      (*invalid_free_fn)(ptr);  // Decide how to handle the bad free request
      return;
    }
    // 然后取出slab class并且cache住.
    cl = span->sizeclass;
    Static::pageheap()->CacheSizeClass(p, cl);
  }
  if (cl != 0) { // 如果是小对象释放的话
    ASSERT(!Static::pageheap()->GetDescriptor(p)->sample);
    ThreadCache* heap = GetCacheIfPresent(); // 那么获得到当前线程的tc
    if (heap != NULL) {
      heap->Deallocate(ptr, cl); // 然后回收到这个tc里面
    } else { // 不知道这个情况什么时候出现，如果出现的话，那么就放到cc里面,非常直接.
      // Delete directly into central cache
      tcmalloc::SLL_SetNext(ptr, NULL);
      Static::central_cache()[cl].InsertRange(ptr, ptr, 1);
    }
  } else {
    SpinLockHolder h(Static::pageheap_lock());
    ASSERT(reinterpret_cast<uintptr_t>(ptr) % kPageSize == 0);
    ASSERT(span != NULL && span->start == p);
    if (span->sample) { // 暂时不理会这个sample逻辑
      StackTrace* st = reinterpret_cast<StackTrace*>(span->objects);
      tcmalloc::DLL_Remove(span);
      Static::stacktrace_allocator()->Delete(st);
      span->objects = NULL;
    }
    // 如果是大对象的话那么直接由pageheap释放.
    Static::pageheap()->Delete(span);
  }
}

// The default "do_free" that uses the default callback.
inline void do_free(void* ptr) {
  return do_free_with_callback(ptr, &InvalidFree); // 默认情况就是打印一个log并且直接crash掉.
}
```

6 Discussion

6.1 tcmalloc中的 MmapSysAllocator::Alloc 疑问(nwlzee)

Question

您好，我看到这函数有点不了解。
在MmapSysAllocator::Alloc 中:

```
// 。。。
  if ((ptr & (alignment - 1)) != 0) {
    adjust = alignment - (ptr & (alignment - 1));
  }

  // Return the unused memory to the system
  if (adjust > 0) {
    munmap(reinterpret_cast<void*>(ptr), adjust);
  }
  if (adjust < extra) {
    munmap(reinterpret_cast<void*>(ptr + adjust + size), extra - adjust);
  }
```

```
  ptr += adjust;
  return reinterpret_cast<void*>(ptr);
```

我从man 手册知道munmap 是以page 单位大小释放的内存的，
当 munmap(reinterpret_cast<void*>(ptr), adjust); 释放adjust所
包含的页了，则返回 ptr += adjust (可能指向刚才释放的页中某一地址)，这地址ptr不是无效了?

Answer

看看MmapSysAllocator::Alloc这个函数吧，假设这里的alignment==page_size的情况的话，

1. extra = alignment – pagesize; 所以extra==0
2. ptr肯定和page_size对齐，因此adjust==0

所以你可以看到其实两个分支都没有走到的。

其实在实际使用的时候alignment通常也是page_size的倍数。如果alignment==k*page_size的话，你会发现

1. extra也是page_size倍数
2. adjust也是page_size倍数

因此在munmap的时候不会存在跨越page_size边界这样的问题的。

最后你看看tcmalloc是怎么使用MmapSysAllocator对象的。相信你也看得非常仔细，tcmalloc是 DefaultSysAllocator创建两个对象

1. SbrkSysAllocator
2. MmapSysAllocator

你看看DefaultSysAllocator调用情况，

```
[dirlt@umeng-ubuntu-pc] > grep "TCMalloc_SystemAlloc" *
common.cc:  void* result = TCMalloc_SystemAlloc(bytes, NULL);
page_heap.cc:#include "system-alloc.h"      // for TCMalloc_SystemAlloc, etc
page_heap.cc:  void* ptr = TCMalloc_SystemAlloc(ask << kPageShift, &actual_size, kPageSize);
page_heap.cc:      ptr = TCMalloc_SystemAlloc(ask << kPageShift, &actual_size, kPageSize);
system-alloc.cc:  // This doesn't overflow because TCMalloc_SystemAlloc has already
system-alloc.cc:    // NOTE: not a devmem_failure - we'd like TCMalloc_SystemAlloc to
system-alloc.cc:void* TCMalloc_SystemAlloc(size_t size, size_t *actual_size,
system-alloc.h:extern void* TCMalloc_SystemAlloc(size_t bytes, size_t *actual_bytes,
```

你会看到实际上调用TCMalloc_SystemAlloc时候，alignment都是==kPageSize的。因此实际tcmalloc 使用时候不会出现这个问题。