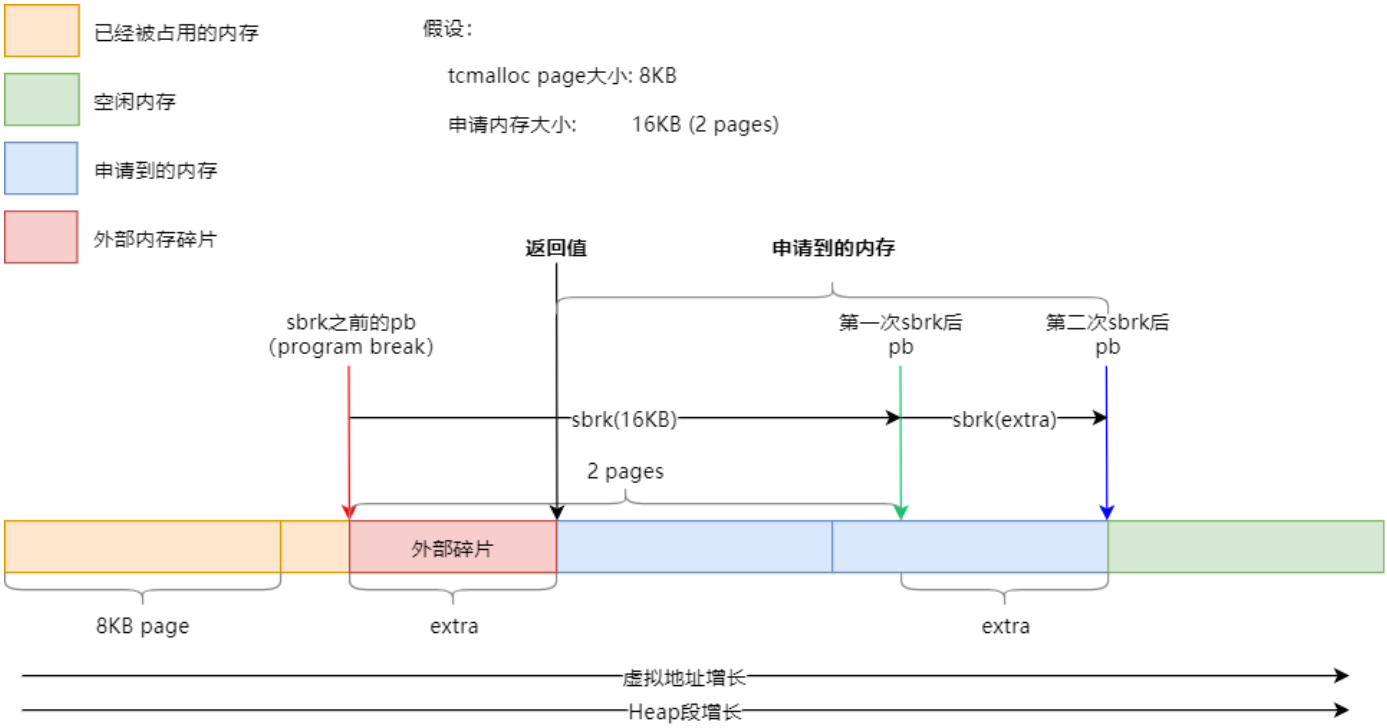
zhuanlan.zhihu.com /p/51433411

TCMalloc解密（二）



原文请移步我的博客：TCMalloc解密

TCMalloc的实现细节

算法概览一节涉及到了很多概念，比如Page，Span，Size Class，PageHeap，ThreadCache等，但只是粗略的一提，本节将详细讨论这些概念所涉及的实现细节。

Page

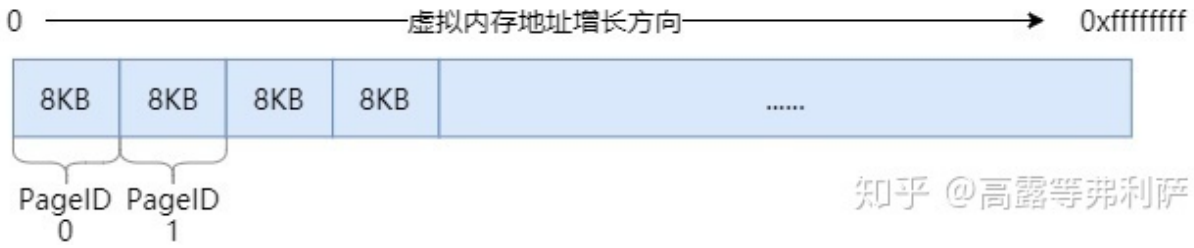
Page是TCMalloc管理内存的基本单位（这里的page要区分于操作系统管理虚拟内存的page），page的默认大小为8KB，可在configure时通过选项调整为32KB或64KB。

```
./configure <other flags> --with-tcmalloc-pagesize=32 (or 64)
```

page越大，TCMalloc的速度相对越快，但其占用的内存也会越高。简单说，就是空间换时间的道理。默认的page大小通过减少内存碎片来最小化内存使用，但跟踪这些page会花费更多的时间。使用更大的page则会带来更多的内存碎片，但速度上会有所提升。官方文档给出的数据是在某些google应用上有3%~5%的速度提升。

PageID

TCMalloc并非只将堆内存看做是一个个的page，而是将整个虚拟内存空间都看做是page的集合。从内存地址0x0开始，每个page对应一个递增的PageID，如下图（以32位系统为例）：



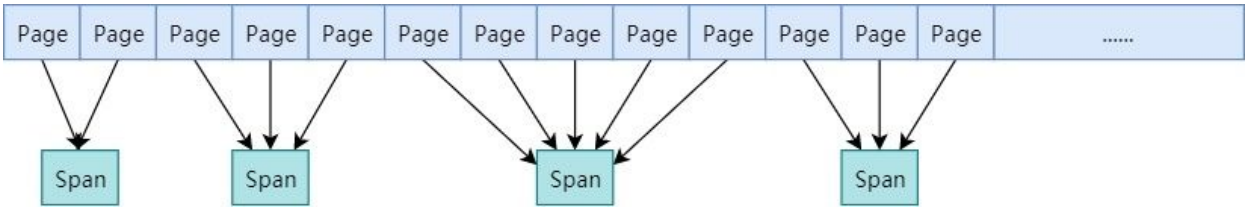
对于任意内存地址`ptr`，都可通过简单的移位操作来计算其所在page的PageID：

```
static const size_t kPageShift  = 13; // page大小：1 << 13 = 8KB
const PageID p = reinterpret_cast<uintptr_t>(ptr) >> kPageShift;
```

即，`ptr`所属page的PageID为`ptr / page_size`。

Span

一个或多个连续的Page组成一个Span（a contiguous run of pages）。**TCMalloc以Span为单位向系统申请内存。**



如图，第一个span包含2个page，第二个和第四个span包含3个page，第三个span包含5个page。

一个span记录了起始page的PageID（`start`），以及所包含page的数量（`length`）。

一个span要么被拆分成多个相同size class的小对象用于小对象分配，要么作为一个整体用于中对象或大对象分配。当作用作小对象分配时，span的`sizeclass`成员变量记录了其对应的size class。

span中还包含两个Span类型的指针（`prev`, `next`），用于将多个span以链表的形式存储。

span的三种状态

一个span处于以下三种状态中的一种：

- IN_USE
- ON_NORMAL_FREELIST
- ON_RETURNED_FREELIST

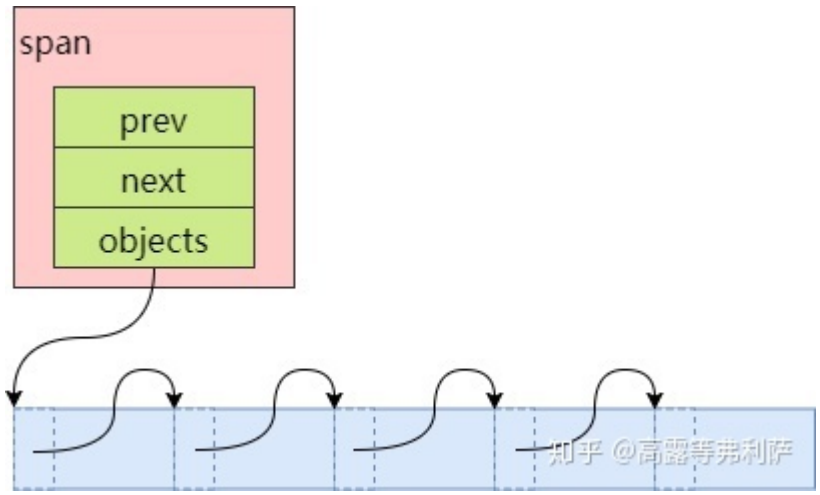
`IN_USE`比较好理解，正在使用中的意思，要么被拆分成小对象分配给CentralCache或者ThreadCache了，要么已经分配给应用程序了。因为span是由PageHeap来管理的，因此即使只是分配给了CentralCache，还没有被应用程序所申请，在PageHeap看来，也是IN_USE了。

`ON_NORMAL_FREELIST`和`ON_RETURNED_FREELIST`都可以认为是空闲状态，区别在于，ON_RETURNED_FREELIST是指span对应的内存已经被PageHeap释放给系统了（在Linux中，对于MAP_PRIVATE|MAP_ANONYMOUS的内存使用madvise来实现）。需要注意的是，即使归还给系统，其虚拟内存地址依然是可访问的，只是对这些内存的修改丢失了而已，在下一次访问时会导致page fault以用0来重新初始化。

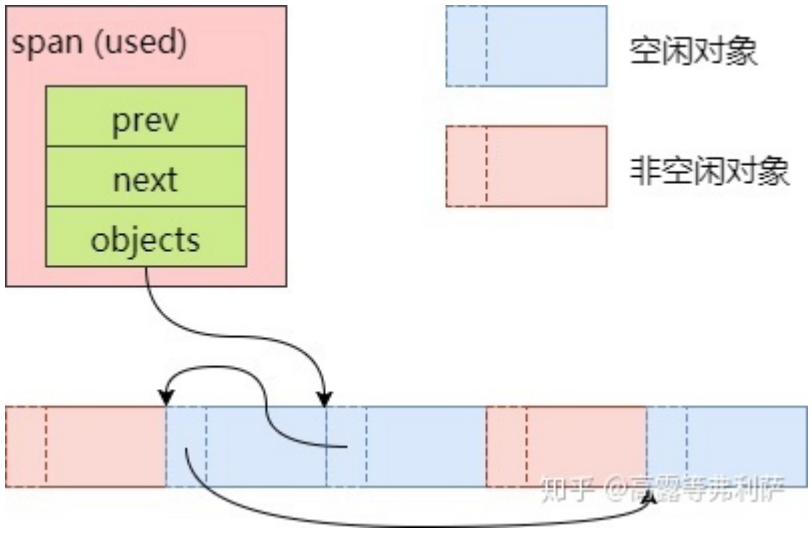
空闲对象链表

被拆分成多个小对象的span还包含了一个记录空闲对象的链表`objects`，由CentralFreeList来维护。

对于新创建的span，将其对应的内存按size class的大小均分成若干小对象，在每一个小对象的起始位置处存储下一个小对象的地址，首首相连：



但当span中的小对象经过一系列申请和回收之后，其顺序就不确定了：



可以看到，有些小对象已经不再空闲对象链表objects中了，链表中的元素顺序也已经被打乱。

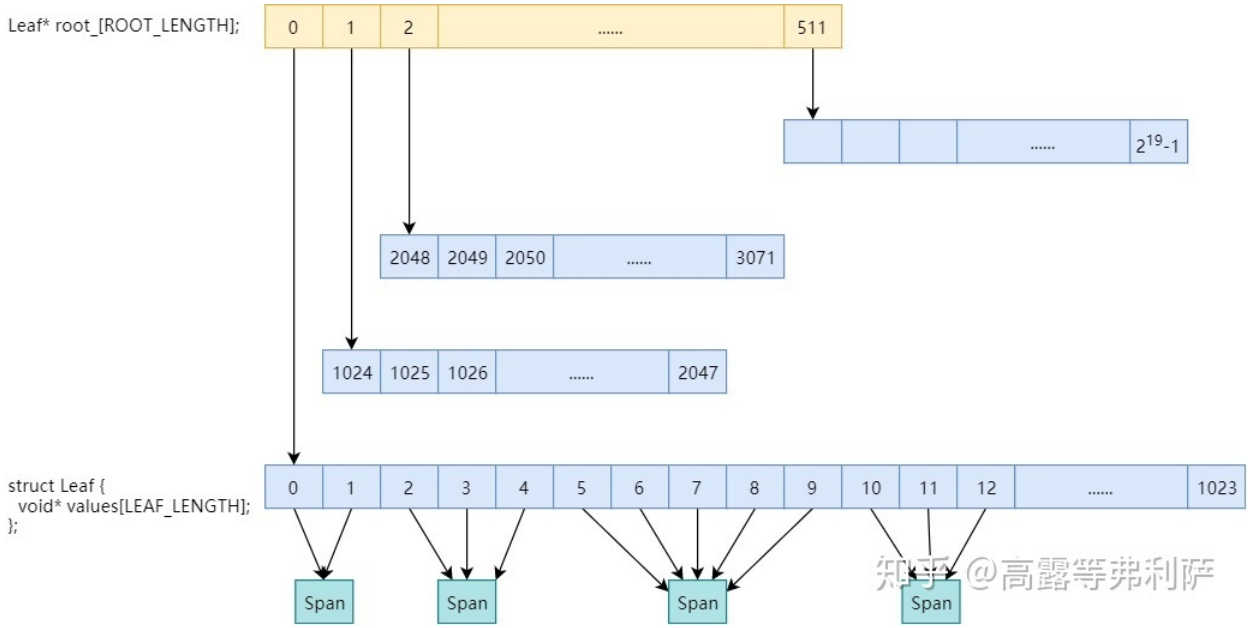
空闲对象链表中的元素乱序没什么影响，因为只有当一个span的所有小对象都被释放之后，CentralFreeList才会将其还给PageHeap。

PageMap

PageMap之前没有提到过，它主要用于解决这么一个问题：**给定一个page，如何确定这个page属于哪个span？**

即，PageMap缓存了PageID到Span的对应关系。

32位系统、x86-64、arm64使用两级PageMap，以32位系统为例：



在root_数组中包含512个指向Leaf的指针，每个Leaf又是1024个void*的数组，数组索引为PageID，数组元素为page所属Span的指针。一共2^{19}个数组元素，对应32位系统的2^{19}个page。

使用两级map可以减少TCMalloc元数据的内存占用，因为初始只会给第一层（即root_数组）分配内存（2KB），第二层只有在实际用到时才会实际分配内存。而如果初始就给2^{19}个page都分配内存的话，则会占用$2^{19} * 4 bytes = 2MB$的内存。

Size Class

TCMalloc将每个小对象的大小（1B~256KB）分为85个类别（官方介绍中说是88个左右，但我个人实际测试是85个，不包括0字节大小），称之为Size Class，每个size class一个编号，从0开始递增（实际编号为0的size class是对应0字节，是没有实际意义的）。

举个例子，896字节对应编号为30的size class，下一个size class 31大小为1024字节，那么897字节到1024字节之间所有的分配都会向上舍入到1024字节。

`SizeMap::Init()`实现了对size class的划分，规则如下：

划分跨度

- 16字节以内，每8字节划分一个size class。
- 满足这种情况的size class只有两个：8字节、16字节。
- 16~128字节，每16字节划分一个size class。
- 满足这种情况的size class有7个：32, 48, 64, 80, 96, 112, 128字节。
- 128B~256KB，按照每次步进`(size / 8)`字节的长度划分，并且步长需要向下对齐到2的整数次幂，比如：
- 144字节：128 + 128 / 8 = 128 + 16 = 144
- 160字节：144 + 144 / 8 = 144 + 18 = 144 + 16 = 160
- 176字节：160 + 160 / 8 = 160 + 20 = 160 + 16 = 176
- 以此类推

一次移动多个空闲对象

ThreadCache会从CentralCache中获取空闲对象，也会将超出限制的空闲对象放回CentralCache。ThreadCache和CentralCache之间的对象移动是**批量**进行的，即一次移动多个空闲对象。CentralCache由于是所有线程公用，因此对其进行操作时需要加锁，一次移动多个对象可以**均摊锁操作的开销**，提升效率。

那么一次批量移动多少呢？每次移动64KB大小的内存，即因size class而异，但至少2个，至多32个（可通过环境变量`TCMALLOC_TRANSFER_NUM_OBJ`调整）。

移动数量的计算也是在size class初始化的过程中计算得出的。

一次申请多个page

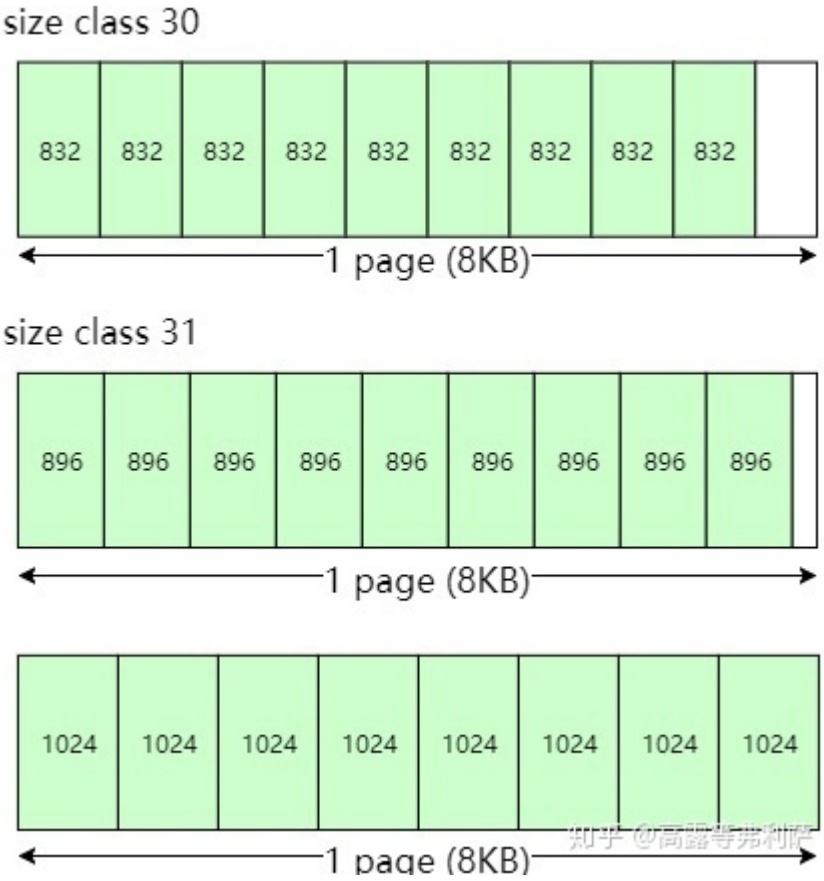
对于每个size class，TCMalloc向系统申请内存时一次性申请n个page（一个span），然后均分成多个小对象进行缓存，以此来**均摊系统调用的开销**。

不同的size class对应的page数量是不同的，如何决定n的大小呢？从1个page开始递增，一直到均分成若干小对象后所剩的空间小于span总大小的1/8为止，因此，浪费的内存被控制在12.5%以内。这是TCMalloc**减少内部碎片**的一种措施。

另外，所分配的page数量还需满足一次移动多个空闲对象的数量要求（源码中的注释是这样说的，不过实际代码是满足1/4即可，原因不明）。

合并操作

在上述规则之上，还有一个合并操作：TCMalloc会将**相同page数量，相同对象数量**的相邻的size class合并为一个size class。比如：



第30个size class的对象大小是832字节，page数量为1个，因此包含8192 / 832 = 9个小对象。

第31个size class对应的page数量（1个）和对象数量（9个）跟第30个size class完全一样，因此第30个size class和第31个size class合并，所以第30个size class对应的对象大小为896字节。

下一个size class对应的对象大小为1024字节，page数量为1个，因此对象数量是8个，跟第30个size class的对象数量不一样，无法合并。

最终，第30个size class对应的对象大小为896字节。

记录映射关系

由以上划分规则可以看到，一个size class对应：

- 一个对象大小
- 一个申请page的数量
- 一个批量移动对象的数量

TCMalloc将size class与这些信息的映射关系分别记录在三个以size class的编号为索引的数组中（`class_to_size_`，`num_objects_to_move_`，`class_to_pages_`）。

还有一项非常重要的映射关系：小对象大小到size class编号的映射。TCMalloc将其记录在一个一维数组`class_array_`中。

256KB以内都是小对象，而size class的编号用一个字节就可以表示，因此存储小对象大小对应的size class编号需要256K个unsigned char，即256KB的内存。但由于size class之间是有间隔的（1024字节以内间隔至少8字节，1024以上间隔至少128字节），因此可以通过简单的计算对class_array_的索引进行压缩，以减少内存占用。

给定大小s，其对应的class_array_索引计算方式如下：

```
// s <= 1024
static inline size_t SmallSizeClass(size_t s) {
  return (static_cast<uint32_t>(s) + 7) >> 3;
}

// s > 1024
static inline size_t LargeSizeClass(size_t s) {
  return (static_cast<uint32_t>(s) + 127 + (120 << 7)) >> 7;
}
```

当s = 256KB时，计算结果即为`class_array_`的最大索引2169，因此数组的大小为2170字节。

计算任意内存地址对应的对象大小

当应用程序调用`free()`或`delete`释放内存时，需要有一种方式获取所要释放的内存地址对应的内存大小。结合前文所述的各种映射关系，在TCMalloc中按照以下顺序计算任意内存地址对应的对象大小：

- 计算给定地址计所在的PageID（ptr >> 13）
- 从PageMap中查询该page所在的span
- span中记录了size class编号
- 根据size class编号从`class_to_size_`数组中查询对应的对象大小

这样做的好处是：**不需要在内存块的头部记录内存大小，减少内存的浪费**。

小结

size class的实现中有很多省空间省时间的做法：

- 省空间
- 控制划分跨度的最大值（8KB），减少内部碎片
- 控制一次申请page的数量，减少内部碎片
- 通过计算和一系列元数据记录内存地址到内存大小的映射关系，避免在实际分配的内存块中记录内存大小，减少内存浪费
- 两级PageMap或三级PageMap
- 压缩class_array_
- 省时间
- 一次申请多个page
- 一次移动多个空闲对象

PageHeap

前面介绍的都是TCMalloc如何对内存进行划分，接下来看TCMalloc如何管理如此划分后的内存，这是PageHeap的主要职责。

TCMalloc源码中对PageHeap的注释：

```
// -------------------------------------------------------------------------
-
// Page-level allocator
//  * Eager coalescing
//
// Heap for page-level allocation.  We allow allocating and freeing a
// contiguous runs of pages (called a "span").
// -------------------------------------------------------------------------
-
```

空闲Span管理器

如前所述，128page以内的span称为小span，128page以上的span称为大span。PageHeap对于这两种span采取了不同的管理策略。小span用链表，而且每个大小的span都用一个单独的链表来管理。大span用std::set。

前文没有提到的是，从另一个维度来看，PageHeap是分开管理`ON_NORMAL_FREELIST`和`ON_RETURNED_FREELIST`状态的span的。因此，每个小span对应两个链表，所有大span对应两个set。

```
// 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 {
```

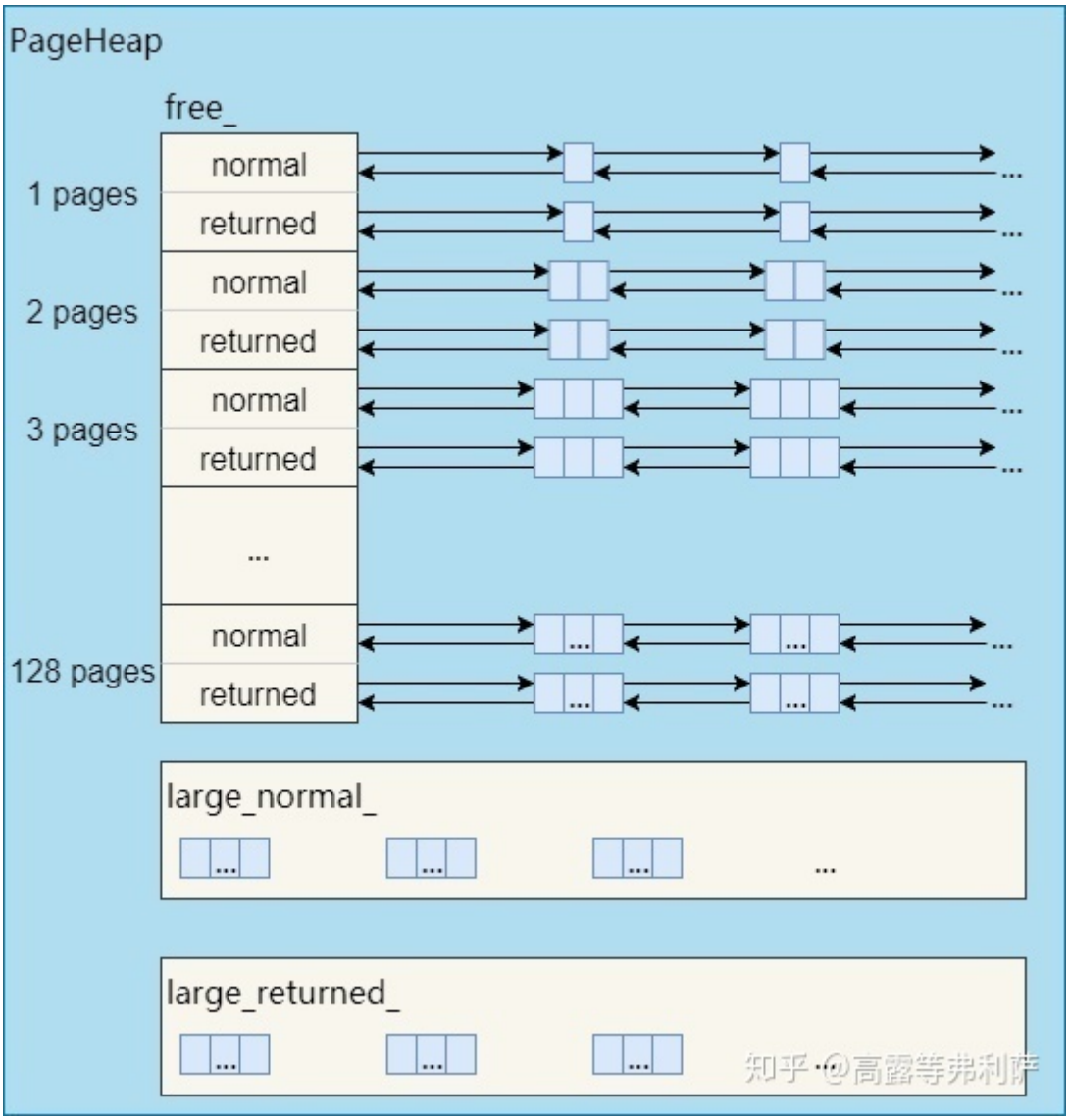
```
  Span        normal;
  Span        returned;
};

// Array mapping from span length to a doubly linked list of free spans
//
// NOTE: index 'i' stores spans of length 'i + 1'.
SpanList free_[kMaxPages];    // 128

typedef std::set<SpanPtrWithLength, SpanBestFitLess,
STLPageHeapAllocator<SpanPtrWithLength, void> > SpanSet;

// Sets of spans with length > kMaxPages.
//
// Rather than using a linked list, we use sets here for efficient
// best-fit search.
SpanSet large_normal_;
SpanSet large_returned_;
```

因此，实际的PageHeap是这样的：



Heap段的使用限制

可以通过`FLAGS_tcmalloc_heap_limit_mb`对进程heap段的内存使用量进行限制，默认值0表示不做限制。如果开启了限制并且对heap段内存的使用量接近这个限制时，TCMalloc会更积极的将空闲内存释放给系统，进而会引发更多的软分页错误（minor page fault）。

为了简化讨论，**后文均假设没有对heap段的内存使用做任何限制**。

创建Span

```
// Allocate a run of "n" pages.  Returns zero if out of memory.
// Caller should not pass "n == 0" -- instead, n should have
// been rounded up already.
Span* New(Length n);
```

创建span的过程其实就是分配中对象和大对象的过程，假设要创建k个page大小的span（以下简称大小为k的span），过程如下：

搜索空闲span

```
Span* SearchFreeAndLargeLists(Length n);
```

1. 搜索空闲span链表，按照以下顺序，找出第一个不小于k的span：
2. 从大小为k的span的链表开始依次搜索
3. 对于某个大小的span，先搜normal链表，再搜returned链表
4. 如果span链表中没找到合适的span，则搜索存储大span的set：
5. 从大小为k的span开始搜索
6. 同样先搜normal再搜returned
7. 优先使用长度最小并且起始地址最小的span（**best-fit**）
8. 如果通过以上两步找到了一个大小为m的span，则将其拆成两个span：
9. 大小为m - k的span重新根据大小和状态放回链表或set中
10. 大小为k的span作为结果返回，创建span结束
11. 如果没搜到合适的span，则继续后面的步骤：向系统申请内存。

小插曲：释放所有空闲内存

```
ReleaseAtLeastNPages(static_cast<Length>(0x7fffffff));
```

当没有可用的空闲span时，需要向系统申请新的内存，但在此之前，还有一次避免向系统申请新内存的机会：释放所有空闲内存。向系统申请的内存每达到128MB，且空闲内存超过从系统申请的总内存的1/4，就需要将所有的空闲内存释放。

因为TCMalloc将normal和returned的内存分开管理，而这两种内存不会合并在一起。因此，可能有一段连续的空闲内存符合要求（k个page大小），但因为它既有normal的部分，又有returned的部分，因此前面的搜索规则搜不到它。而释放所有内存可以将normal的内存也变为returned的，然后就可以合并了（合并规则详细后文合并span）。

之所以控制在每128MB一次的频率，是为了避免高频小量申请内存的程序遭受太多的minor page fault。

释放完毕后，会按照前面的搜索规则再次尝试搜索空闲span，如果还搜不到，才继续后续的步骤。

向系统申请内存

```
bool GrowHeap(Length n);
```

找不到合适的空闲span，就只能向系统申请新的内存了。

TCMalloc以`sbrk()`和`mmap()`两种方式向系统申请内存，所申请内存的大小和位置均按page对齐，优先使用`sbrk()`，申请失败则会尝试使用`mmap()`（64位系统Debug模式优先使用mmap，原因详见`InitSystemAllocators()`注释）。

TCMalloc向系统申请应用程序所使用的内存时，**每次至少尝试申请1MB**（`kMinSystemAlloc`），申请TCMalloc自身元数据所使用的内存时，每次至少申请8MB（`kMetadataAllocChunkSize`）。这样做有

两点好处：

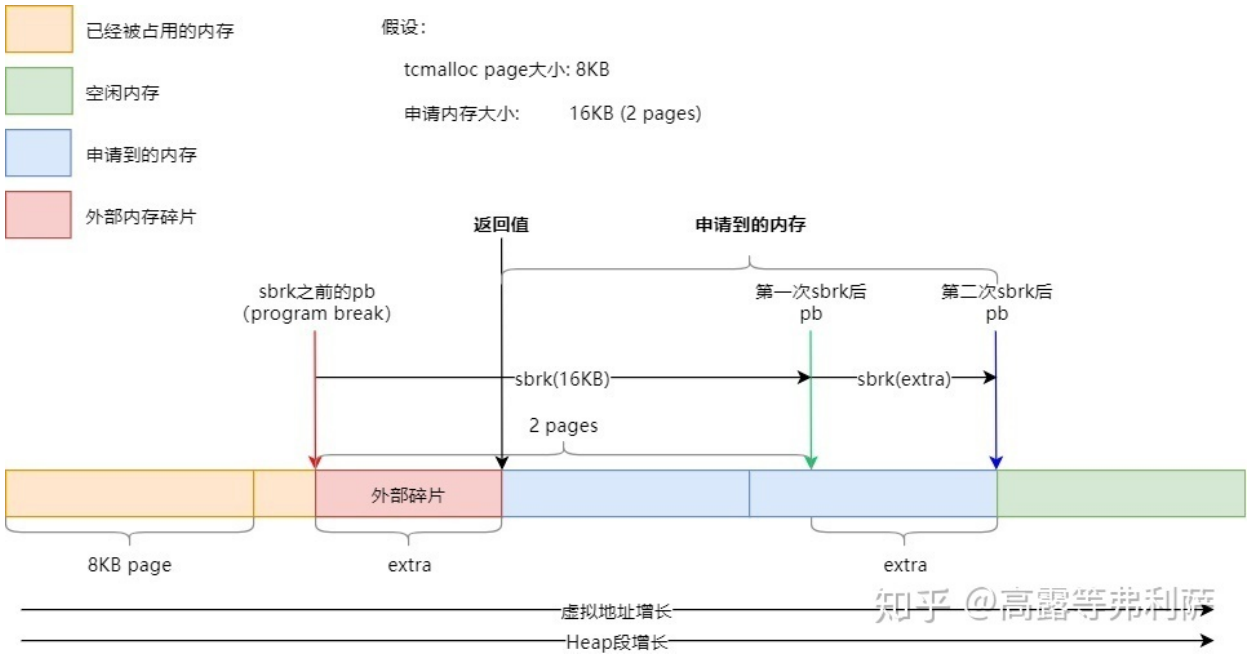
- 减少外部内存碎片（减少所申请内存与TCMalloc元数据所占内存的交替）
- 均摊系统调用的开销，提升性能

另外，当**从系统申请的总内存超过128MB时就为PageMap一次性申请一大块内存**，保证可以存储所有page和span的对应关系。这一举措可以减少TCMalloc的元数据将内存分块而导致的外部碎片。从源码中可以发现，仅在32位系统下才会这样做，可能是因为64位系统内存的理论上限太大，不太现实。

```
// bool PageHeap::GrowHeap(Length n);
if (old_system_bytes < kPageMapBigAllocationThreshold
    && stats_.system_bytes >= kPageMapBigAllocationThreshold) {
  pagemap_.PreallocateMoreMemory();
}
```

sbrk

先来看如何使用`sbrk()`从Heap段申请内存，下图展示了`SbrkSysAllocator::Alloc()`的执行流程，为了说明外部碎片的产生，并覆盖到`SbrkSysAllocator::Alloc()`的大部分流程，假设page大小为8KB，所申请的内存大小为16KB：

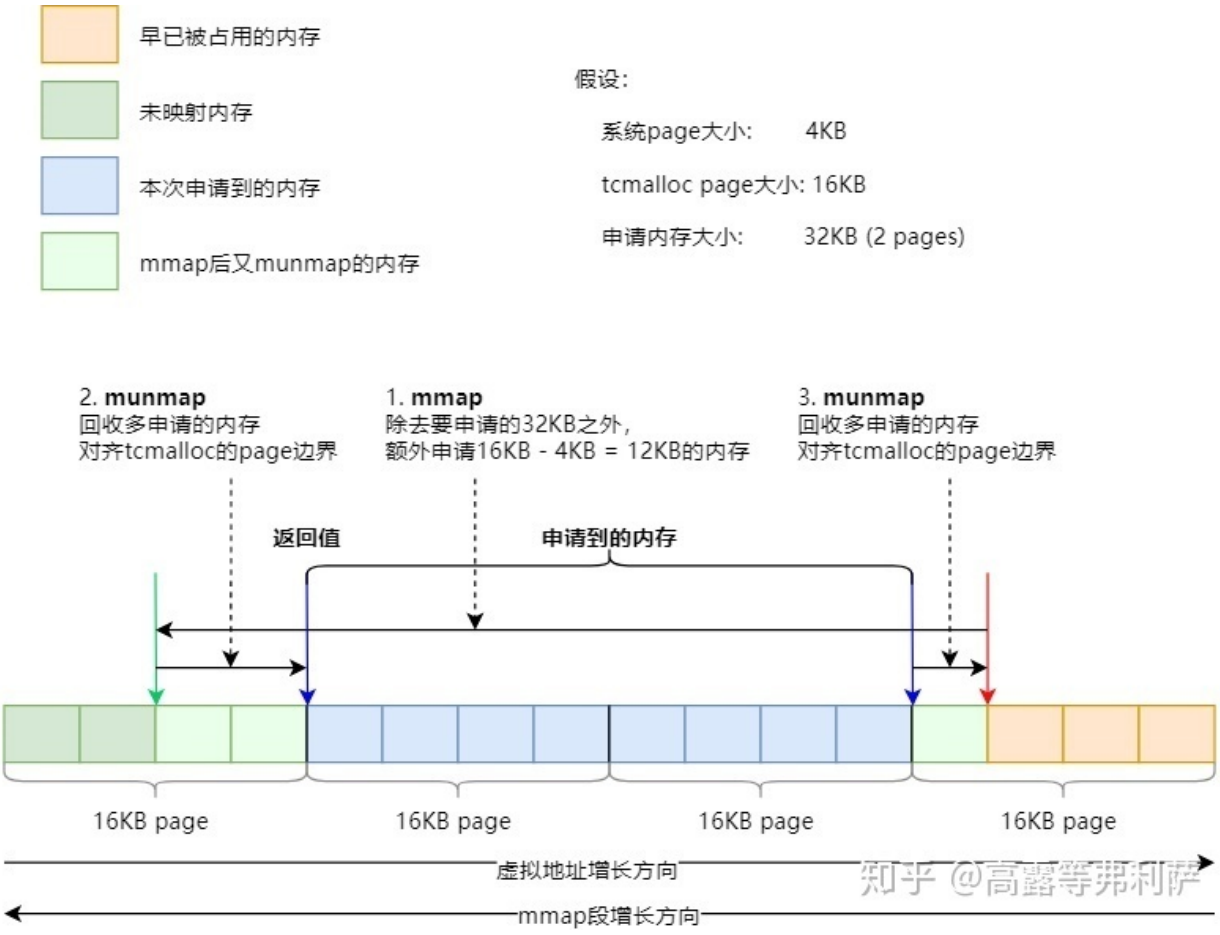


1. 假设在申请内存之前，pb（program break，可以认为是堆内存的上边界）指向红色箭头所示位置，即没有在page的边界处。
2. 第一次sbrk申请16KB内存，因此pb移至绿色箭头所示位置。
3. 由于需要对申请的内存按page对齐，因此需要第二次sbrk，pb指向蓝色箭头所示位置，page的边界处。
4. 最终，返回的内存地址为黑色箭头所示位置，黑色和蓝色箭头之间刚好16KB。

可以看出，红色箭头和黑色箭头之间的内存就无法被使用了，产生了**外部碎片**。

mmap

如果使用sbrk申请内存失败，TCMalloc会尝试使用mmap来分配。同样，为了覆盖`MmapSysAllocator::Alloc()`的大部分情况，下图假设系统的page为4KB，TCMalloc的page为16KB，申请的内存大小为32KB：



1. 假设在申请内存之前，mmap段的边界位于红色箭头所示位置。
2. 第一次mmap，会在32KB的基础上，多申请(对齐大小 - 系统page大小) = 16 -4 = 12KB的内存。此时mmap的边界位于绿色箭头所示位置。
3. 然后通过两次munmap将所申请内存的两侧边界分别对齐到TCMalloc的page边界。
4. 最终申请到的内存为两个蓝色箭头之间的部分，返回左侧蓝色箭头所指示的内存地址。

如果申请内存成功，则创建一个新的span并立即删除，可将其放入空闲span的链表或set中，然后继续后面的步骤。

最后的搜索

最后，重新搜索一次空闲span，如果还找不到合适的空闲span，那就认为是创建失败了。

至此，创建span的操作结束。

删除Span

```
// Delete the span "[p, p+n-1]".
// REQUIRES: span was returned by earlier call to New() and
//           has not yet been deleted.
void Delete(Span* span);
```

当span所拆分成的小对象全部被应用程序释放变为空闲对象，或者作为中对象或大对象使用的span被应用程序释放时，需要将span删除。不过并不是真正的删除，而是放到空闲span的链表或set中。

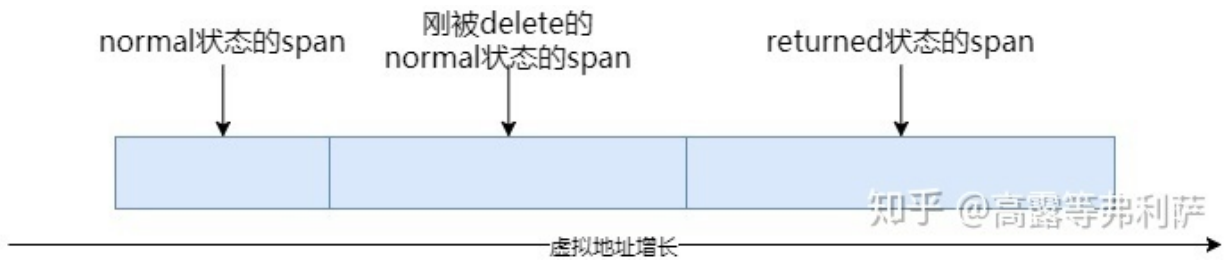
删除的操作非常简单，但可能会触发合并span的操作，以及释放内存到系统的操作。

合并Span

当span被delete时，会尝试向前向后合并一个span。

合并规则如下：

- 只有在虚拟内存空间上连续的span才可以被合并。
- 只有同是normal状态的span或同是returned状态的span才可以被合并。



上图中，被删除的span的前一个span是normal状态，因此可以与之合并，而后一个span是returned状态，无法与之合并。合并操作之后如下图：

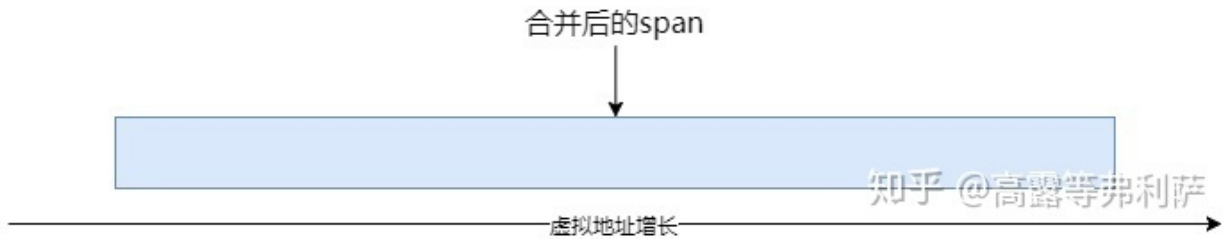
还有一个值得注意的开关：`aggressive_decommit_`，开启后TCMalloc会积极的释放内存给系统，默认是关闭状态，可通过以下方式更改：

```
MallocExtension::instance()->SetNumericProperty("tcmalloc.aggressive_memory_decommit", value)
```

当开启了`aggressive_decommit_`后，删除normal状态的span时会尝试将其释放给系统，释放成功则状态变为returned。

在合并时，如果被删除的span此时是returned状态，则会将与其相邻的normal状态的span也释放给系统，然后再尝试合并。

因此，上面的例子中，被删除的span和其前一个span都会被更改为returned状态，合并之后如下，即三个span被合并成为一个span：



释放span

```
Length ReleaseAtLeastNPages(Length num_pages);
```

在delete一个span时还会以一定的频率触发释放span的内存给系统的操作（`ReleaseAtLeastNPages()`）。释放的频率可以通过环境变量`TCMALLOC_RELEASE_RATE`来修改。**默认值为1，表示每删除1000个page就尝试释放至少1个page**，2表示每删除500个page尝试释放至少1个page，依次类推，10表示每删除100个page尝试释放至少1个page。0表示永远不释放，值越大表示释放的越快，合理的取值区间为[0, 10]。

释放规则：

- 从小到大循环，按顺序释放空闲span，直到释放的page数量满足需求。
- 多次调用会从上一次循环结束的位置继续循环，而不会重新从头（1 page）开始。
- 释放的过程中能合并span就合并span
- 可能释放少于num_pages，没那么多free的span了。
- 可能释放多于num_pages，还差一点就够num_pages了，但刚好碰到一个很大的span。

释放方式：

如果系统支持，通过`madvise(MADV_DONTNEED)`释放span对应的内存。因此，即使释放成功，对应的虚拟内存地址空间仍然可访问，但内存的内容会丢失，在下次访问时会导致minor page fault以用0来重新初

始化。