zhuanlan.zhihu.com /p/51432385

TCMalloc解密（一）

原文请移步我的博客：TCMalloc解密

写在前面

本文首先简单介绍TCMalloc及其使用方法，然后解释TCMalloc替代系统的内存分配函数的原理，然后从宏观上讨论其内存分配的策略，在此之后再深入讨论实现细节。

有几点需要说明：

- 本文只讨论gperftools中TCMalloc部分的代码，对应版本gperftools-2.7。
- 本文是根据TCMalloc源码以及简短的官方介绍作出的个人理解，难免有纰漏之处，且难以覆盖TCMalloc的方方面面，不足之处还请各位看官留言指正。
- 除非特别说明，以下讨论均以32位系统、TCMalloc默认page大小（8KB）为基础，不同架构或不同page大小，有些相关数值可能不一样，但基本原理是相似的。
- 为了控制篇幅，我会尽量少贴大段代码，只给出关键代码的位置或函数名，各位看官可自行参阅TCMalloc相关代码。

TCMalloc是什么

TCMalloc全称Thread-Caching Malloc，即线程缓存的malloc，实现了高效的多线程内存管理，用于替代系统的内存分配相关的函数（malloc、free，new，new[]等）。

TCMalloc是gperftools的一部分，除TCMalloc外，gperftools还包括heap-checker、heap-profiler和cpu-profiler。本文只讨论gperftools的TCMalloc部分。

git仓库：https://github.com/gperftools/gperftools.git

官方介绍：https://gperftools.github.io/gperftools/TCMalloc.html（里面有些内容已经过时了）

如何使用TCMalloc

安装

以下是比较常规的安装步骤，详细可参考gperftools中的INSTALL。

1. 从git仓库clone版本的gperftools的安装依赖autoconf、automake、libtool，以Debian为例：

```
bash # apt install autoconf automake libtool
```

1. 生成configure等一系列文件

```
bash $ ./autogen.sh
```

1. 生成Makefile

```
bash $ ./configure
```

1. 编译

```
bash $ make
```

1. 安装

```
bash # make install
```

默认安装在`/usr/local/`下的相关路径（bin、lib、share），可在configure时以`--prefix=PATH`指定其他路径。

64位Linux系统需要注意

在64位Linux环境下，gperftools使用glibc内置的stack-unwinder可能会引发死锁，因此官方推荐在配置和安装gperftools之前，先安装libunwind-0.99-beta，最好就用这个版本，版本太新或太旧都可能会有问题。

即便使用libunwind，在64位系统上还是会有问题，但只影响heap-checker、heap-profiler和cpu-profiler，TCMalloc不受影响，因此不再赘述，感兴趣的读者可参阅gperftools的INSTALL。

如果不希望安装libunwind，也可以用gperftools内置的stack unwinder，但需要应用程序、TCMalloc库、系统库（比如libc）在编译时开启帧指针（frame pointer）选项。

在x86-64下，编译时开启帧指针选项并不是默认行为。因此需要指定`-fno-omit-frame-pointer`编译所有应用程序，然后在configure时通过`--enable-frame-pointers`选项使用内置的gperftools stack unwinder。

使用

以动态库的方式

安装之后，通过`-ltcmalloc`或`-ltcmalloc_minimal`将TCMalloc链接到应用程序即可。

```
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    malloc(1);
}
$ g++ -O0 -g -ltcmalloc test.cc && gdb a.out
(gdb) b test.cc:5
Breakpoint 1 at 0x7af: file test.cc, line 5.
```

```
(gdb) r
Starting program: /home/wanglong/test/https://wallenwang.com/tcmalloc/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffddd8) at test.cc:5
5           malloc(1);
(gdb) s
tc_malloc (size=1) at src/tcmalloc.cc:1892
1892      return malloc_fast_path<tcmalloc::malloc_oom>(size);
(gdb)
```

通过gdb断点可以看到对`malloc()`的调用已经替换为TCMalloc的实现。

以静态库的方式

gperftools的README中说静态库应该使用`libtcmalloc_and_profiler.a`库而不是`libprofiler.a`和libtcmalloc.a，但简单测试后者也是OK的，而且在实际项目中也是用的后者，不知道是不是文档太过老旧了。

```
$ g++ -O0 -g -pthread test.cc /usr/local/lib/libtcmalloc_and_profiler.a
```

如果使用了libunwind，需要指定`-Wl,--eh-frame-hdr`选项，以确保libunwind可以找到编译器生成的信息来进行栈回溯。

eh-frame（exception handling frame）参考资料：

TCMalloc是如何生效的

为什么指定`-ltcmalloc`或者与`libtcmalloc_and_profiler.a`连接之后，对malloc、free、new、delete等的调用就由默认的libc中的函数调用变为TCMalloc中相应的函数调用了呢？答案在`libc_override.h`中，下面只讨论常见的两种情况：使用了glibc，或者使用了GCC编译器。其余情况可自行查看相应的libc_override头文件。

使用glibc（但没有使用GCC编译器）

在glibc中，内存分配相关的函数都是弱符号（weak symbol），因此TCMalloc只需要定义自己的函数将其覆盖即可，以malloc和free为例：

libc_override_redefine.h

```
extern "C" {
  void* malloc(size_t s)                          { return tc_malloc(s);
}
  void  free(void* p)                             { tc_free(p);
```

```
}
}  // extern "C"
```

可以看到，TCMalloc将`malloc()`和`free()`分别定义为对`tc_malloc()`和`tc_free()`的调用，并在`tc_malloc()`和`tc_free()`中实现具体的内存分配和回收逻辑。

new和delete也类似：

```
void* operator new(size_t size)                    { return tc_new(size);
}
void operator delete(void* p) CPP_NOTHROW          { tc_delete(p);
}
```

使用GCC编译器

如果使用了GCC编译器，则使用其支持的函数属性：alias。

libc_override_gcc_and_weak.h：

```
#define ALIAS(tc_fn)    __attribute__ ((alias (#tc_fn), used))

extern "C" {
  void* malloc(size_t size) __THROW                ALIAS(tc_malloc);
  void free(void* ptr) __THROW                     ALIAS(tc_free);
}   // extern "C"
```

将宏展开，`__attribute__ ((alias ("tc_malloc"), used))`表明tc_malloc是malloc的别名。

具体可参阅GCC相关文档：

> **alias ("target")** The alias attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance, `void __f () { /* Do something. */; } void f () __attribute__ ((weak, alias ("__f")));` defines `f` to be a weak alias for `__f`. In C++, the mangled name for the target must be used. It is an error if `__f` is not defined in the same translation unit. Not all target machines support this attribute.
> **used** This attribute, attached to a function, means that code must be emitted for the function even if it appears that the function is not referenced. This is useful, for example, when the function is referenced only in inline assembly. When applied to a member function of a C++ class template, the attribute also means that the function will be instantiated if the class itself is instantiated.

TCMalloc的初始化

何时初始化

TCMalloc定义了一个类`TCMallocGuard`，并在文件tcmalloc.cc中定义了该类型的静态变量`module_enter_exit_hook`，在其构造函数中执行TCMalloc的初始化逻辑，以确保TCMalloc在`main()`函数之前完成初始化，防止在初始化之前就有多个线程。

http://tcmalloc.cc：

```
static TCMallocGuard module_enter_exit_hook;
```

如果需要确保TCMalloc在某些全局构造函数运行之前就初始化完成，则需要在文件顶部创建一个静态TCMallocGuard实例。

如何初始化

TCMallocGuard的构造函数实现：

```
static int tcmallocguard_refcount = 0;  // no lock needed: runs before
main()
TCMallocGuard::TCMallocGuard() {
  if (tcmallocguard_refcount++ == 0) {
    ReplaceSystemAlloc();    // defined in libc_override_*.h
    tc_free(tc_malloc(1));
    ThreadCache::InitTSD();
    tc_free(tc_malloc(1));
    // Either we, or debugallocation.cc, or valgrind will control memory
    // management.  We register our extension if we're the winner.
#ifdef TCMALLOC_USING_DEBUGALLOCATION
    // Let debugallocation register its extension.
#else
    if (RunningOnValgrind()) {
      // Let Valgrind uses its own malloc (so don't register our
extension).
    } else {
      MallocExtension::Register(new TCMallocImplementation);
    }
#endif
  }
}
```

可以看到，TCMalloc初始化的方式是调用`tc_malloc()`申请一字节内存并随后调用`tc_free()`将其释放。至于为什么在InitTSD前后各申请释放一次，不太清楚，猜测是为了测试在TSD（Thread Specific Data，详见后文）初始化之前也能正常工作。

初始化内容

那么TCMalloc在初始化时都执行了哪些操作呢？这里先简单列一下，后续讨论TCMalloc的实现细节时再逐一详细讨论：

- 初始化SizeMap（Size Class）
- 初始化各种Allocator
- 初始化CentralCache
- 创建PageHeap

总之一句话，创建TCMalloc自身的一些元数据，比如划分小对象的大小等等。

TCMalloc的内存分配算法概览

TCMalloc的官方介绍中将内存分配称为*Object Allocation*，本文也沿用这种叫法，并将object翻译为**对象**，可以将其理解为具有一定大小的内存。

按照所分配内存的大小，TCMalloc将内存分配分为三类：

- 小对象分配，(0, 256KB]
- 中对象分配，(256KB, 1MB]
- 大对象分配，(1MB, +∞)

简要介绍几个概念，Page，Span，PageHeap：

与操作系统管理内存的方式类似，TCMalloc将整个虚拟内存空间划分为n个同等大小的**Page**，每个page默认8KB。又将连续的n个page称为一个**Span**。

TCMalloc定义了**PageHeap**类来处理向OS申请内存相关的操作，并提供了一层缓存。可以认为，PageHeap就是整个可供应用程序动态分配的内存的抽象。

PageHeap以span为单位向系统申请内存，申请到的span可能只有一个page，也可能包含n个page。可能会被划分为一系列的小对象，供小对象分配使用，也可能当做一整块当做中对象或大对象分配。

小对象分配

Size Class

对于256KB以内的小对象分配，TCMalloc按大小划分了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），称为**Size Class**，每个size class都对应一个大小，比如8字节，16字节，32字节。应用程序申请内存时，TCMalloc会首先将所申请的内存大小向上取整到size class的大小，比如1~8字节之间的内存申请都会分配8字节，9~16字节之间都会分配16字节，以此类推。因此这里会产生内部碎片。TCMalloc将这里的内部碎片控制在12.5%以内，具体的做法在讨论size class的实现细节时再详细分析。

ThreadCache

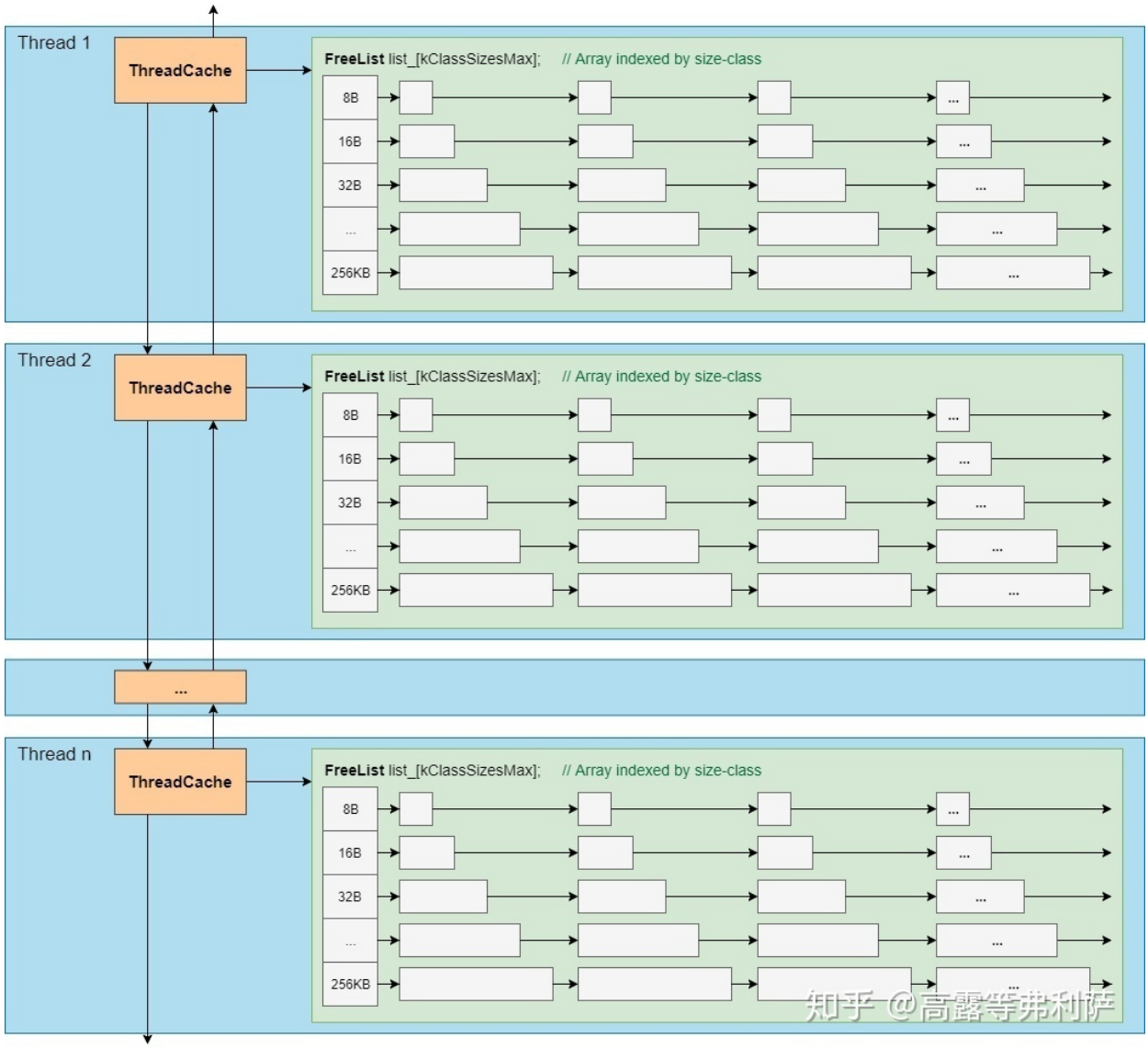
对于每个线程，TCMalloc都为其保存了一份单独的缓存，称之为**ThreadCache**，这也是TCMalloc名字的由来（Thread-Caching Malloc）。每个ThreadCache中对于每个size class都有一个单独的**FreeList**，缓存

了n个还未被应用程序使用的空闲对象。

小对象的分配直接从ThreadCache的FreeList中返回一个空闲对象，相应的，小对象的回收也是将其重新放回ThreadCache中对应的FreeList中。

由于每线程一个ThreadCache，因此从ThreadCache中取用或回收内存是**不需要加锁**的，速度很快。

为了方便统计数据，各线程的ThreadCache连接成一个双向链表。ThreadCache的结构示大致如下：



CentralCache

那么ThreadCache中的空闲对象从何而来呢？答案是**CentralCache**——一个所有线程公用的缓存。

与ThreadCache类似，CentralCache中对于每个size class也都有一个单独的链表来缓存空闲对象，称之为**CentralFreeList**，供各线程的ThreadCache从中取用空闲对象。

由于是所有线程公用的，因此从CentralCache中取用或回收对象，是**需要加锁**的。为了平摊锁操作的开销，ThreadCache一般从CentralCache中一次性取用或回收多个空闲对象。

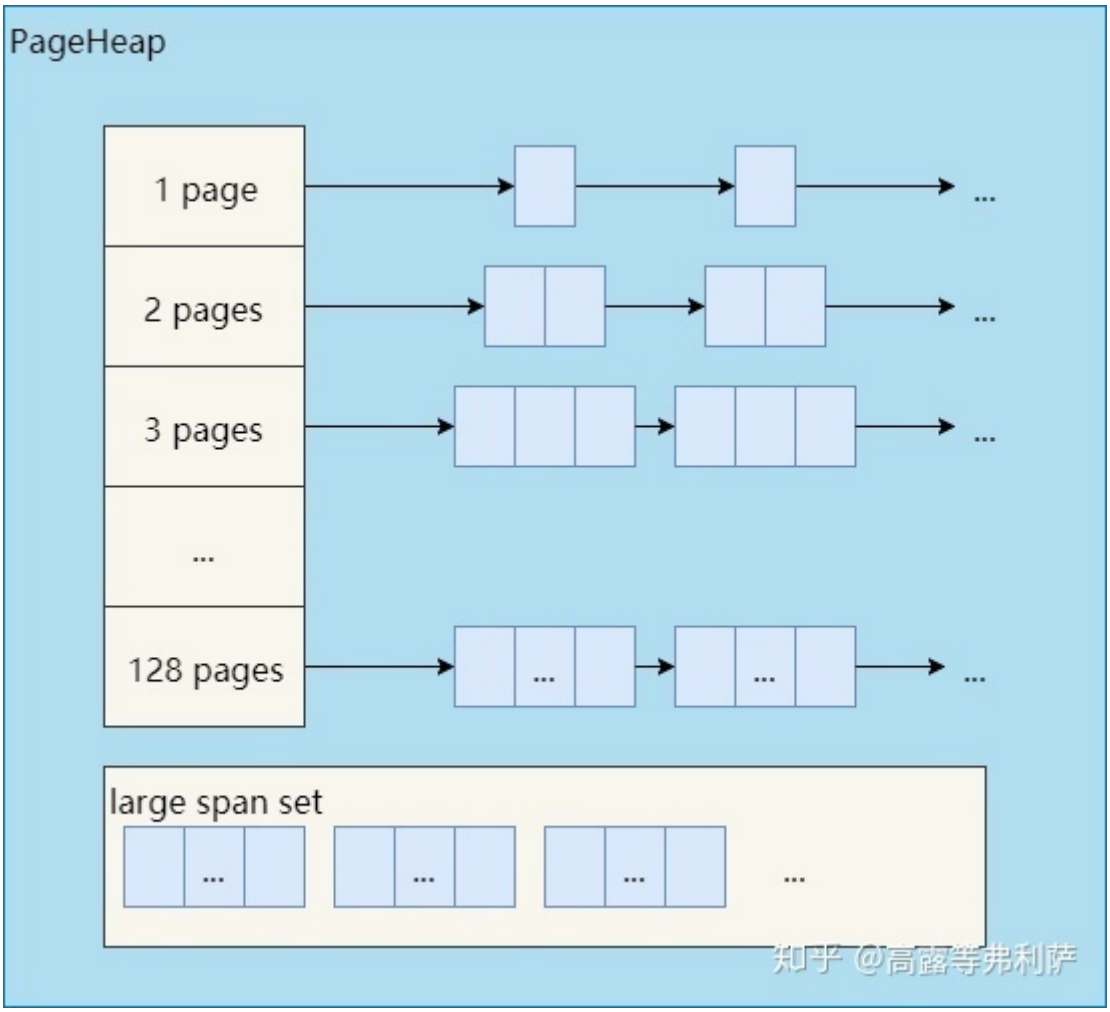
CentralCache在TCMalloc中并不是一个类，只是一个逻辑上的概念，其本质是**CentralFreeList**类型的数组。后文会详细讨论CentralCache的内部结构，现在暂且认为CentralCache的**简化结构**如下：

PageHeap

CentralCache中的空闲对象又是从何而来呢？答案是之前提到的**PageHeap**——TCMalloc对可动态分配的内存的抽象。

当CentralCache中的空闲对象不够用时，CentralCache会向PageHeap申请一块内存（可能来自PageHeap的缓存，也可能向系统申请新的内存），并将其拆分成一系列空闲对象，添加到对应size class的CentralFreeList中。

PageHeap内部根据内存块（span）的大小采取了两种不同的缓存策略。128个page以内的span，每个大小都用一个链表来缓存，超过128个page的span，存储于一个有序set（std::set）。讨论TCMalloc的实现细节时再具体分析，现在可以认为PageHeap的**简化结构**如下：



内存回收

上面说的都是内存分配，内存回收的情况是怎样的？

应用程序调用free()或delete一个小对象时，仅仅是将其插入到ThreadCache中其size class对应的FreeList中而已，不需要加锁，因此速度也是非常快的。

只有当满足一定的条件时，ThreadCache中的空闲对象才会重新放回CentralCache中，以供其他线程取用。同样的，当满足一定条件时，CentralCache中的空闲对象也会还给PageHeap，PageHeap再还给系统。

内存在这些组件之间的移动会在后文详细讨论，现在先忽略这些细节。

小结

总结一下，小对象分配流程大致如下：

- 将要分配的内存大小映射到对应的size class。
- 查看ThreadCache中该size class对应的FreeList。
- 如果FreeList非空，则移除FreeList的第一个空闲对象并将其返回，分配结束。
- 如果FreeList是空的：

- 从CentralCache中size class对应的CentralFreeList获取一堆空闲对象。
  - 如果CentralFreeList也是空的，则：
  - 向PageHeap申请一个span。
  - 拆分成size class对应大小的空闲对象，放入CentralFreeList中。

- 将这堆对象放置到ThreadCache中size class对应的FreeList中（第一个对象除外）。
- 返回从CentralCache获取的第一个对象，分配结束。

中对象分配

超过256KB但不超过1MB（128个page）的内存分配被认为是中对象分配，采取了与小对象不同的分配策略。

首先，TCMalloc会将应用程序所要申请的内存大小向上取整到**整数个page**（因此，这里会产生1B~8KB的内部碎片）。之后的操作表面上非常简单，向PageHeap申请一个指定page数量的span并返回其起始地址即可：

```
Span* span = Static::pageheap()->New(num_pages);
result = (PREDICT_FALSE(span == NULL) ? NULL : SpanToMallocResult(span));
return result;
```

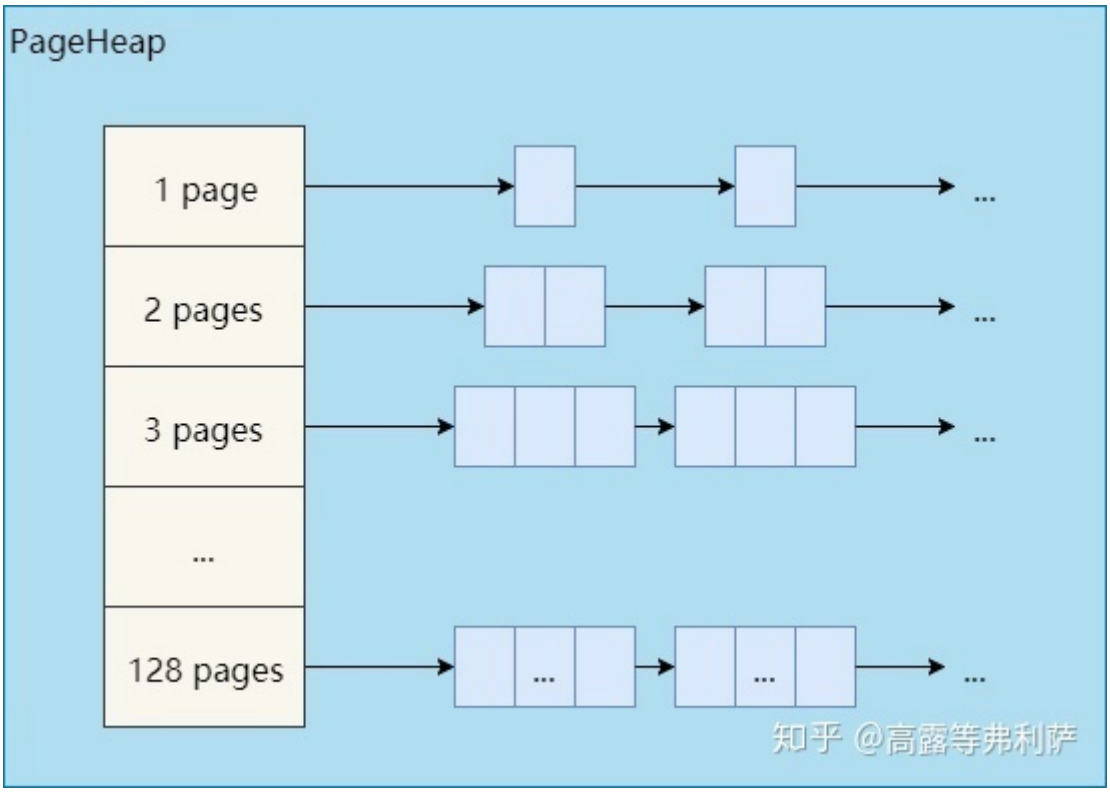
问题在于，PageHeap是如何管理这些span的？即`PageHeap::New()`是如何实现的。

前文说到，PageHeap提供了一层缓存，因此`PageHeap::New()`并非每次都向系统申请内存，也可能直接从缓存中分配。

对128个page以内的span和超过128个page的span，PageHeap采取的缓存策略不一样。为了描述方便，以下将128个page以内的span称为小span，大于128个page的span称为大span。

先来看小span是如何管理的，大span的管理放在大对象分配一节介绍。

PageHeap中有128个小span的链表，分别对应1~128个page的span：



假设要分配一块内存，其大小经过向上取整之后对应k个page，因此需要从PageHeap取一个大小为k

个page的span，过程如下：

- 从k个page的span链表开始，到128个page的span链表，按顺序找到第一个非空链表。
- 取出这个非空链表中的一个span，假设有n个page，将这个span拆分成两个span：
- 一个span大小为k个page，作为分配结果返回。
- 另一个span大小为n - k个page，重新插入到n - k个page的span链表中。
- 如果找不到非空链表，则将这次分配看做是大对象分配，分配过程详见下文。

大对象分配

超过1MB（128个page）的内存分配被认为是大对象分配，与中对象分配类似，也是先将所要分配的内存大小向上取整到整数个page，假设是k个page，然后向PageHeap申请一个k个page大小的span。

对于中对象的分配，如果上述的span链表无法满足，也会被当做是大对象来处理。也就是说，TCMalloc在源码层面其实并没有区分中对象和大对象，只是对于不同大小的span的缓存方式不一样罢了。

大对象分配用到的span都是超过128个page的span，其缓存方式不是链表，而是一个按span大小排序的**有序set**（std::set），以便按大小进行搜索。

假设要分配一块超过1MB的内存，其大小经过向上取整之后对应k个page（k>128），或者是要分配一块1MB以内的内存，但无法由中对象分配逻辑来满足，此时k <= 128。不管哪种情况，都是要从PageHeap的span set中取一个大小为k个page的span，其过程如下：

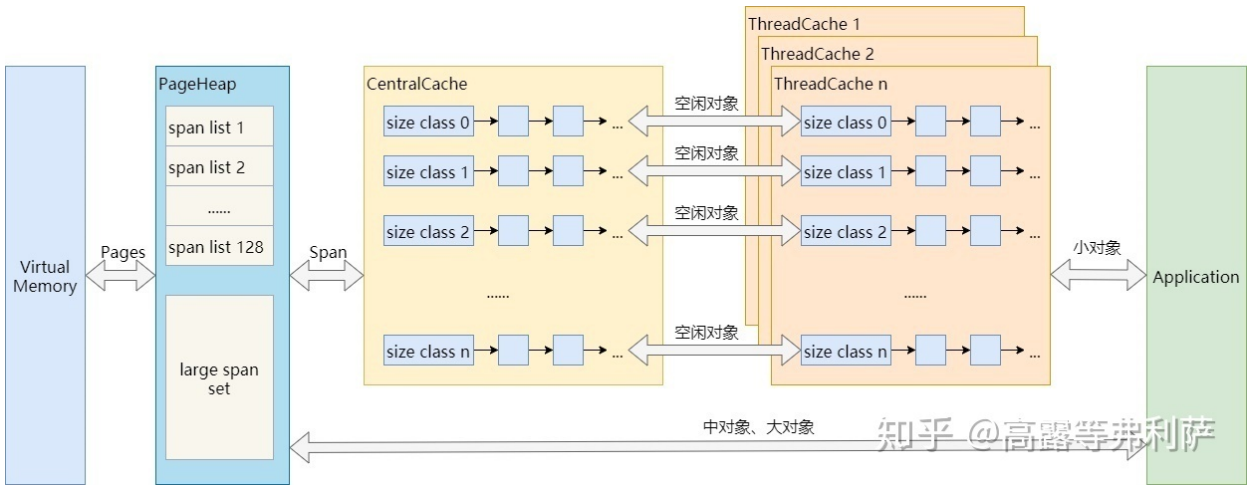
- 搜索set，找到不小于k个page的最小的span（**best-fit**），假设该span有n个page。
- 将这个span拆分为两个span：

- 一个span大小为k个page，作为结果返回。
- 另一个span大小为n - k个page，如果n - k > 128，则将其插入到大span的set中，否则，将其插入到对应的小span链表中。
- 如果找不到合适的span，则使用sbrk或mmap向系统申请新的内存以生成新的span，并重新执行中对象或大对象的分配算法。

小结

以上讨论忽略了很多实现上的细节，比如PageHeap对span的管理还区分了normal状态的span和returned状态的span，接下来会详细分析这些细节。

在此之前，画张图概括下TCMalloc的管理内存的策略：



可以看到，不超过256KB的小对象分配，在应用程序和内存之间其实有三层缓存：PageHeap、CentralCache、ThreadCache。而中对象和大对象分配，则只有PageHeap一层缓存。