tcmalloc源码分析

导语

目前工作中主要使用golang开发，想学习一下golang的内存分配实现原理，在阅读内存分配相关代码的时候，发现会涉及垃圾回收、协程调度、系统调用、plan6汇编代码，增加了学习的难度，因为golang的内存分配参考了tcmalloc，而tcmalloc使用c++实现，不会涉及到协程调度、垃圾回收等方面的知识，能够更加清楚地看到内存分配的实现原理，所以决定从tcmalloc入手学习。

基础回顾

1、内存映射

tcmalloc直接使用mmap向操作系统申请内存，tcmalloc使用下面的函数完成内存的分配与释放。

```
/* system-alloc.cc第604行,mmap完成地址的分配
 * hint是内存的分配起始地址,size是分配内存的大小,这里分配的都是一段虚拟地址空间,并没有真正的分配物理内存
 * PROT_NONE属性表示页不可访问
 * MAP_PRIVATE是建立一个写入时的临时拷贝,MAP_ANONYMOUS表示匿名映射,映射区不与任何文件关联
 */
void* result = mmap(hint, size, PROT_NONE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

/* system-alloc.cc第193行
 * mmap返回的地址是虚拟地址,还没有分配真正的物理内存,这片地址不可读不可写
 * result_ptr地址开始的actual_size字节大小,是上面mmap返回的虚拟地址空间中的一部分
 * mprotect的作用就是让字段内存地址可读可写,但是此时仍然没有分物理空间,在真正使用这片内存时,
 * 会出发缺页中断,操作系统才分配物理内存,然后将物理内存地址和虚拟内存地址关联上
 */
mprotect(result_ptr, actual_size, PROT_READ | PROT_WRITE);

/* system-alloc.cc第428行
 * madvise函数通知内核,可以将start地址开始的长度length的内存资源进行回收
 */
ret = madvise(start, length, MADV_DONTNEED);
```

2、c++的new/delete operator、operator new/delete、placement new

它们之间的区别，简单来说就是new operator/delete是操作符，operator new/delete是函数，placement new是在指定的内存上构造对象。

```
class User {
 public:
  User(std::string name) : name_(name) { }
  std::string Name() { return name_;}
 private:
```

```
6.   std::string name_;
7. };
8. User* user = new User("rtx");
9. delete user;
```

当我们执行上述代码的new User("rtx")和delete user时,new和delete就是new/delete operator。new operator的执行分为三步:

（1）调用operator new函数分配内存

（2）在分配的内存上调用类的构造函数

（3）返回分配的内存地址

上面的第一步分配内存，默认是使用c++标准的::opearator new函数，所以如果我们想要接替c++标准的内存分配函数，就需要我们自己重载operator new函数，operator new的重载可以在类中实现，也可以在全局实现，当我们执行new object的时候，编译器首先检查类中是否已经重载，已经重载了就使用类中的重载版本，如果类中没有重载，就检查有没有全局的重载版本，有就使用它，如果类中和全局都没有重载new，编译器将使用c++的标准版本。

下面的代码在类的内部重载operator new。

```
1. class User {
2.  public:
3.     User(std::string name) : name_(name) { }
4.     std::string Name() { return name_;}
5.     // 重载类的new
6.     void* operator new(size_t size) {
7.         std::cout << "User operator new is called." << std::endl;
8.         // 调用全局的operator new (c++标准的new)
9.         return ::operator new(size);
10.    }
11.    // 重载类的delete
12.    void operator delete(void *ptr) {
13.        std::cout << "User operator delete is called." << std::endl;
14.        // 调用全局operator delete (c++标准的delete)
15.        ::operator delete(ptr);
16.    }
17. private:
18.    std::string name_;
19. };
20. int main() {
21.    User *user = new User("rtx");
22.    delete user;
23. }
24. // 运行上面代码输出
25. // User operator new is called.
26. // User operator delete is called.
```

我们可以在类User的operator new函数里面完成自己想要的操作，包括自己控制如何分配内存。

下面的代码在全局重载operator new/delete。

```
1. int global = 100;
```

```
// 重载全局operator new
void* operator new(size_t size) {
    std::cout << "Global operator new is called." << std::endl;
    return &global;
}
// 重载全局operator delete
void operator delete(void *ptr) {
    std::cout << "Global operator delete is called." << std::endl;
}

class Integer {
 public:
    Integer(int val) : val_(val) {}
    int Value() { return val_; }
 private:
    int val_;
};
int main() {
    Integer *value = new Integer(200);
    std::cout << value→Value() << std::endl; // 输出200,覆盖了初始值100
    delete value;
}
// 执行上面的代码输出
// Global operator new is called.
// 200
// Global operator delete is called.
```

我们在全局重载了operator new函数,这样所有对象的内存分配都将由我们自己写的operator new函数完成。tcmalloc就是在全局重载了operator new/delete函数，从而实现自己管理内存的分配与释放。

最后是placement new，下面的代码说明了placement new的用法。

```
class Integer {
 public:
    Integer(int val) : val_(val) {}
    int Value() { return val_; }
 private:
    int val_;
};
int main() {
    int stackInt = 100;
    // placement new,在堆栈上构造Integer对象
    Integer *value = new (&stackInt) Integer(200);
    std::cout << value→Value() << std::endl;
}
// 执行上面的代码输出
// 200
```

placement new，它主要是在指定的内存上构造对象,tcmalloc在很多地方使用了placement new，比如。

```
// system-alloc.cc第339 340行
```

```
// tcmalloc的地址region管理器
region_manager = new (&region_manager_space) RegionManager();
// tcmalloc region对象工厂类
region_factory = new (&mmap_space) MmapRegionFactory();
```

3、TLS (Thread Local Storage)

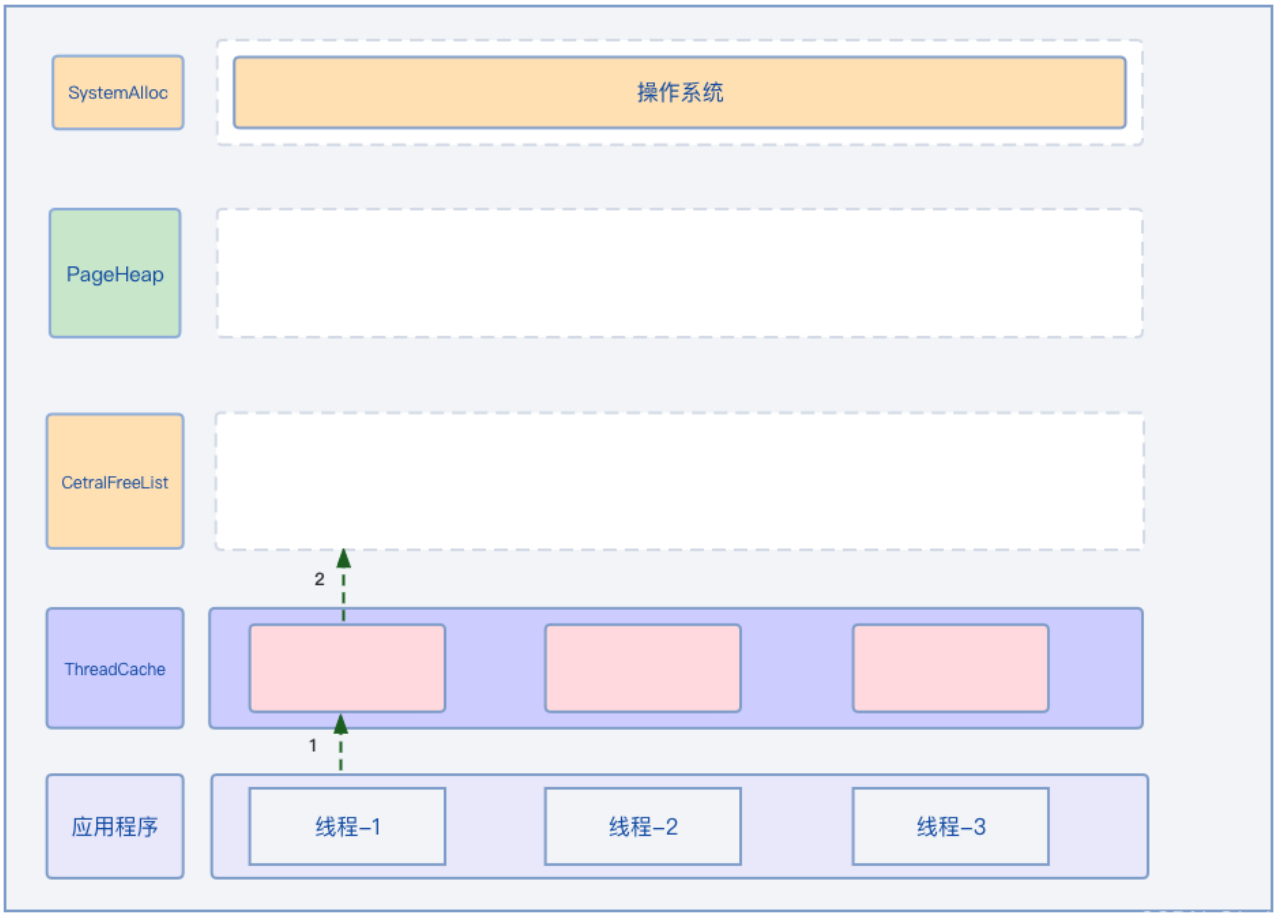
TLS（线程局部存储），我们只需要在进程中定义一个pthread_key_t，所有的线程都可以都通过pthread_setspecific和pthread_getspecific设置自己的局部变量。

```
#include <iostream>
#include <string>

#define err_handle(x) if(x != 0) { return -1;}
#define NUM_THREADS    3
pthread_key_t   tlsKey = 0;

// 销毁每个线程局部数据
void globalDestructor(void *value) {
    std::cout << "In global destructor" << std::endl;
    free(value);
    pthread_setspecific(tlsKey, NULL);
}

void showGlobal() {
    int *global  = (int*)pthread_getspecific(tlsKey);
    std::cout << "Thread local data: " << *global << std::endl;
}
// 线程执行函数
void *threadfunc(void *parm) {
    // 构造每个线程的局部变量
    void* myThreadData = malloc(sizeof(int));
    std::memcpy(myThreadData, parm, sizeof(int));
    // 每个线程使用同一个key,set/get线程特有的value
    pthread_setspecific(tlsKey, myThreadData);
    showGlobal();
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t thread[NUM_THREADS];
    int thread_para[NUM_THREADS] = {1, 2, 3};
    // tlsKey创建与销毁函数
    int rc = pthread_key_create(&tlsKey, globalDestructor);
    err_handle(rc)
    // 线程key现在可以被所有线程使用
    for (int i=0; i<NUM_THREADS; i++) {
        rc = pthread_create(&thread[i], NULL, threadfunc, &thread_para[i]);
        err_handle(rc)
    }
    // 线程退出的时候,会调用thread_key_t关联的globalDestructor函数
    for (int i=0; i<NUM_THREADS; i++) {
        rc = pthread_join(thread[i], NULL);
        err_handle(rc)
    }
    return pthread_key_delete(tlsKey);
}
```

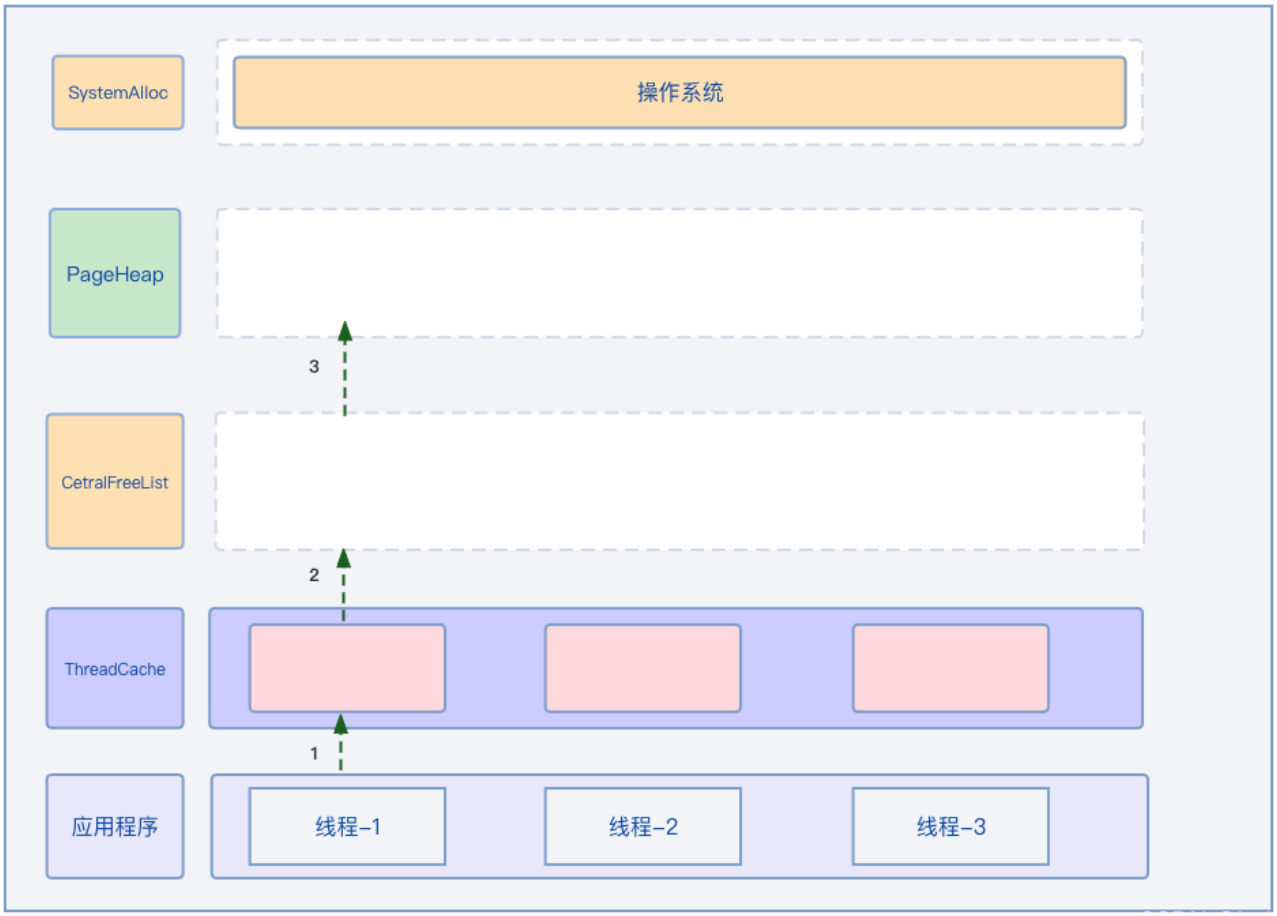
tcmalloc在类ThreadCache中定义了一个静态变量pthread_key_t类型的heap_key_，每个线程设置的局部变量是TheadCache类型的指针，而ThreadCache是线程内存分配的入口类，每个线程访问自己ThreadCache对象分配内存。

tcmalloc内存分配流程概述

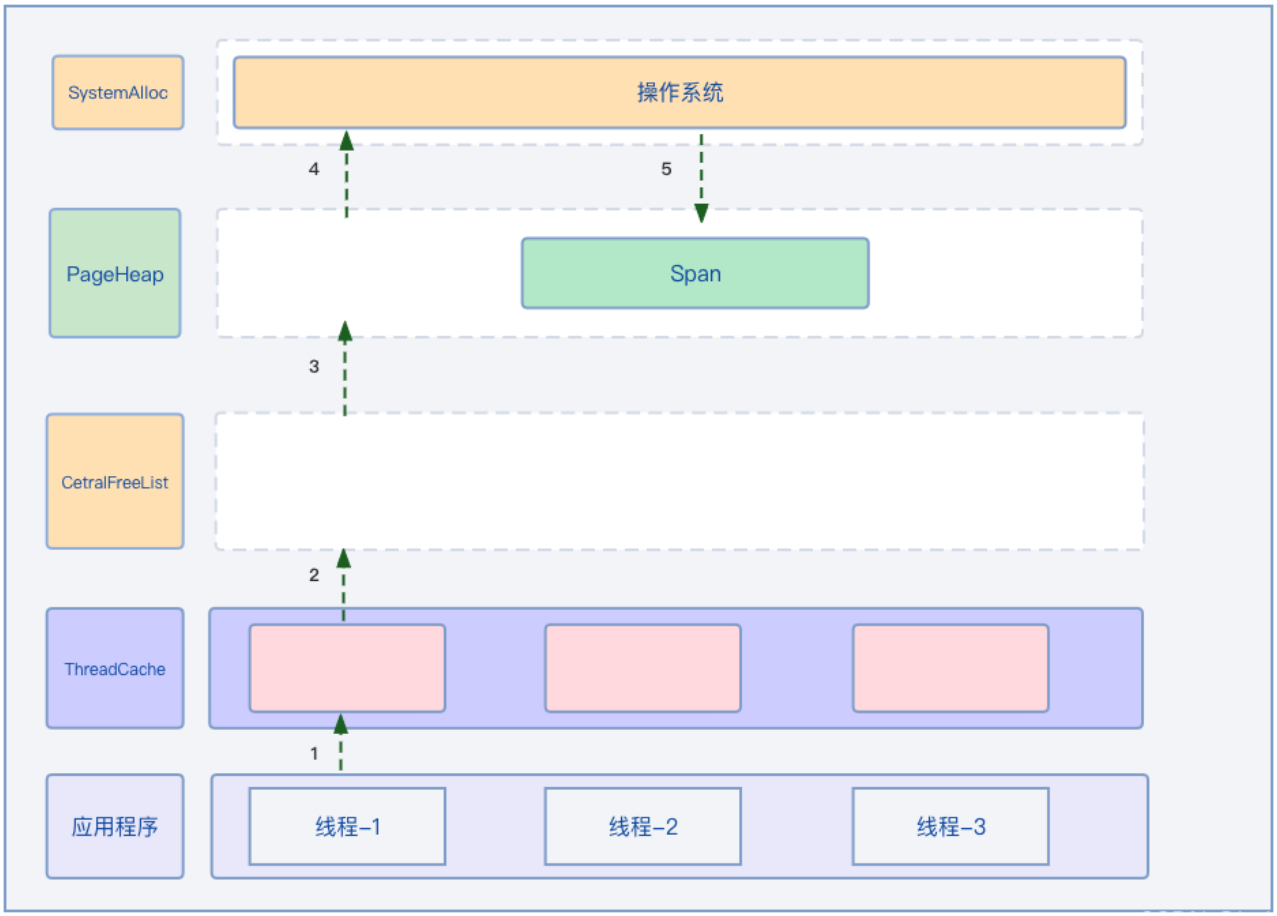
步骤1 如上图所示，假设c++应用程序启动了三个线程，通过TLS，每个线程都会有一个ThreadCache类。假设有一个类A，sizeof(A) = 32字节，现在线程1执行A *p = new A()，线程将通过ThreadCache提供的方法申请内存。



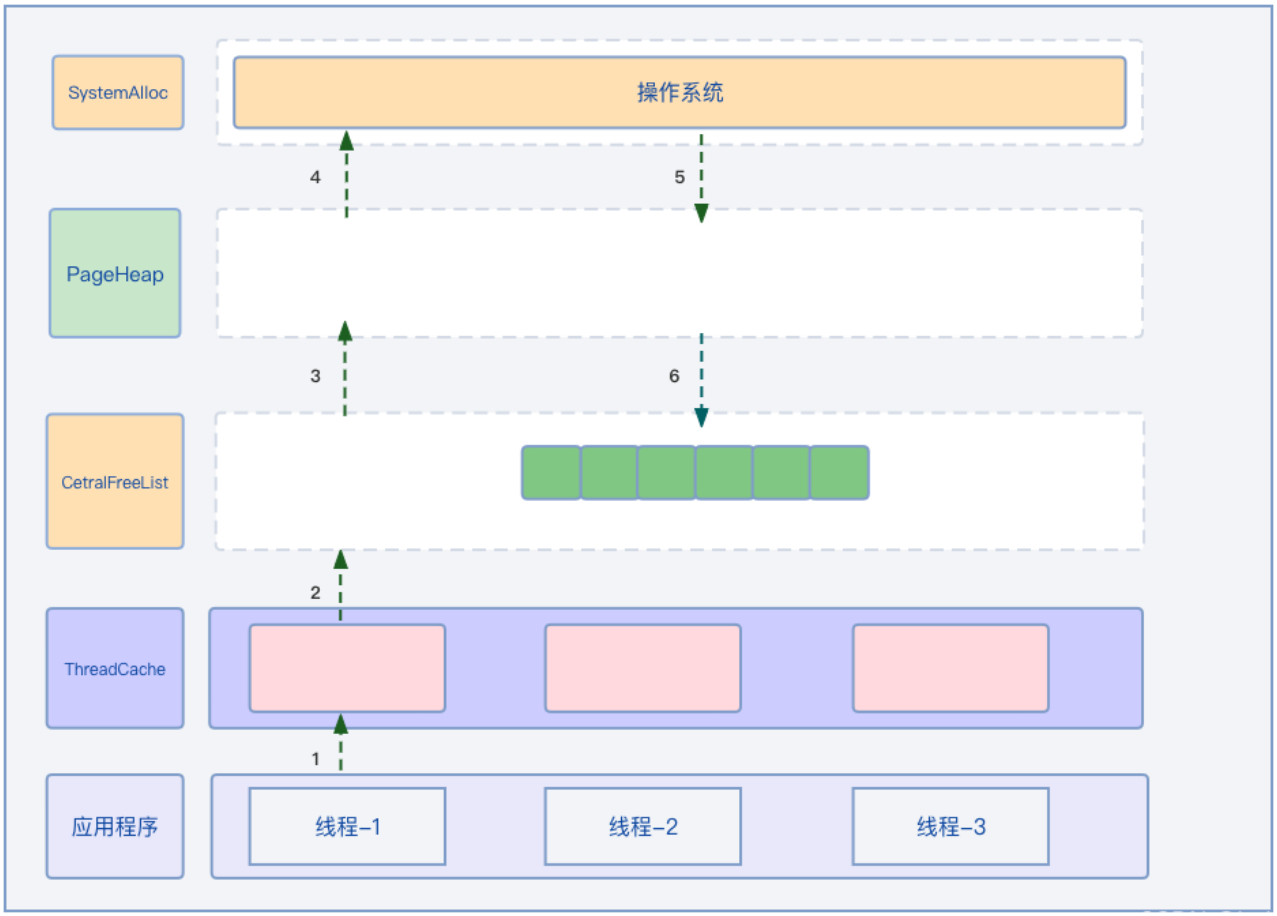
步骤2 ThreadCache初始化完成之后，还没有持有任何内存，所以ThreadCache此时将调用CentralFreeList的方法申请内存。



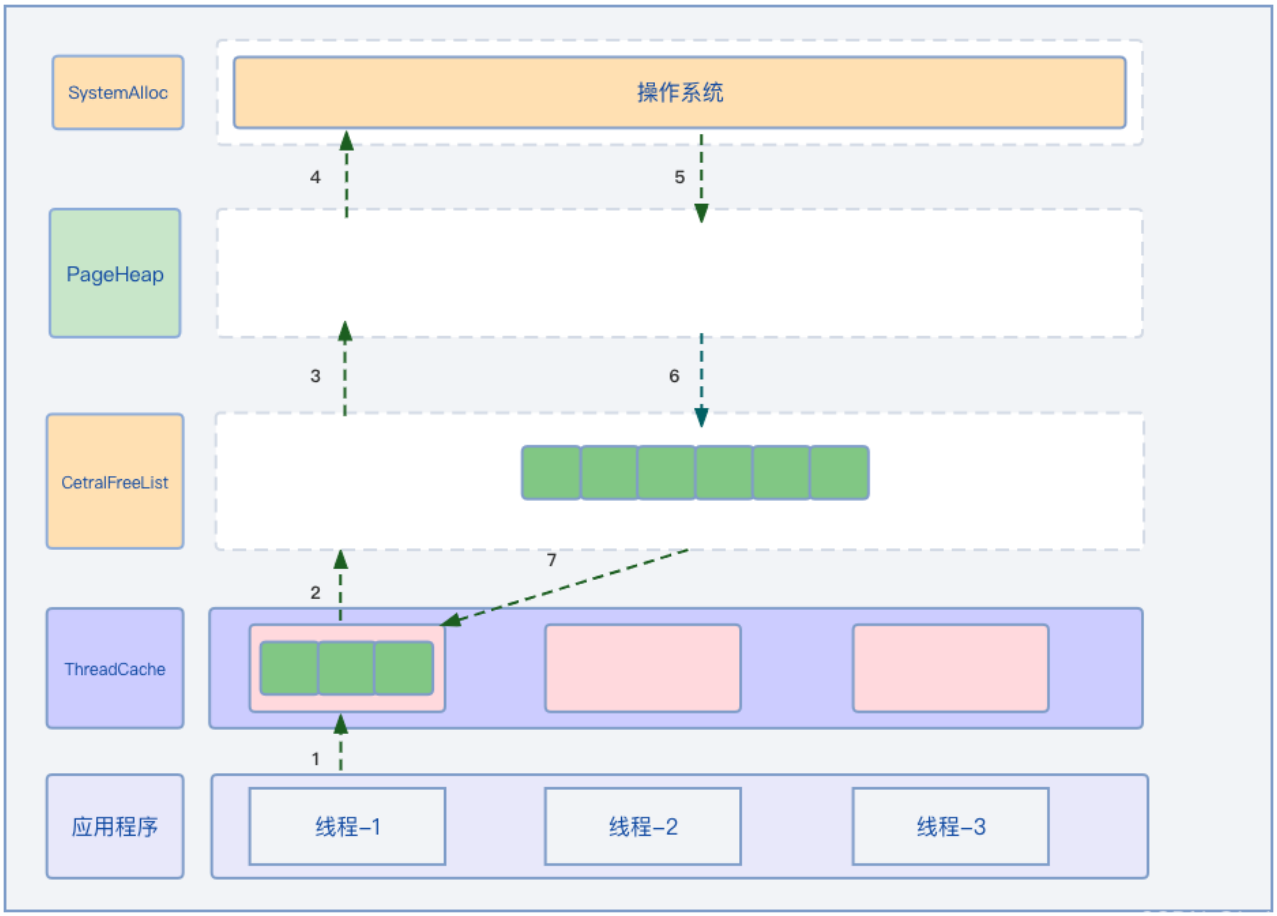
步骤3 CentralFreeList初始化完成之后，同样没有持有任何内存，所以CentralFreeList将调用PageHeap提供的方法申请内存。



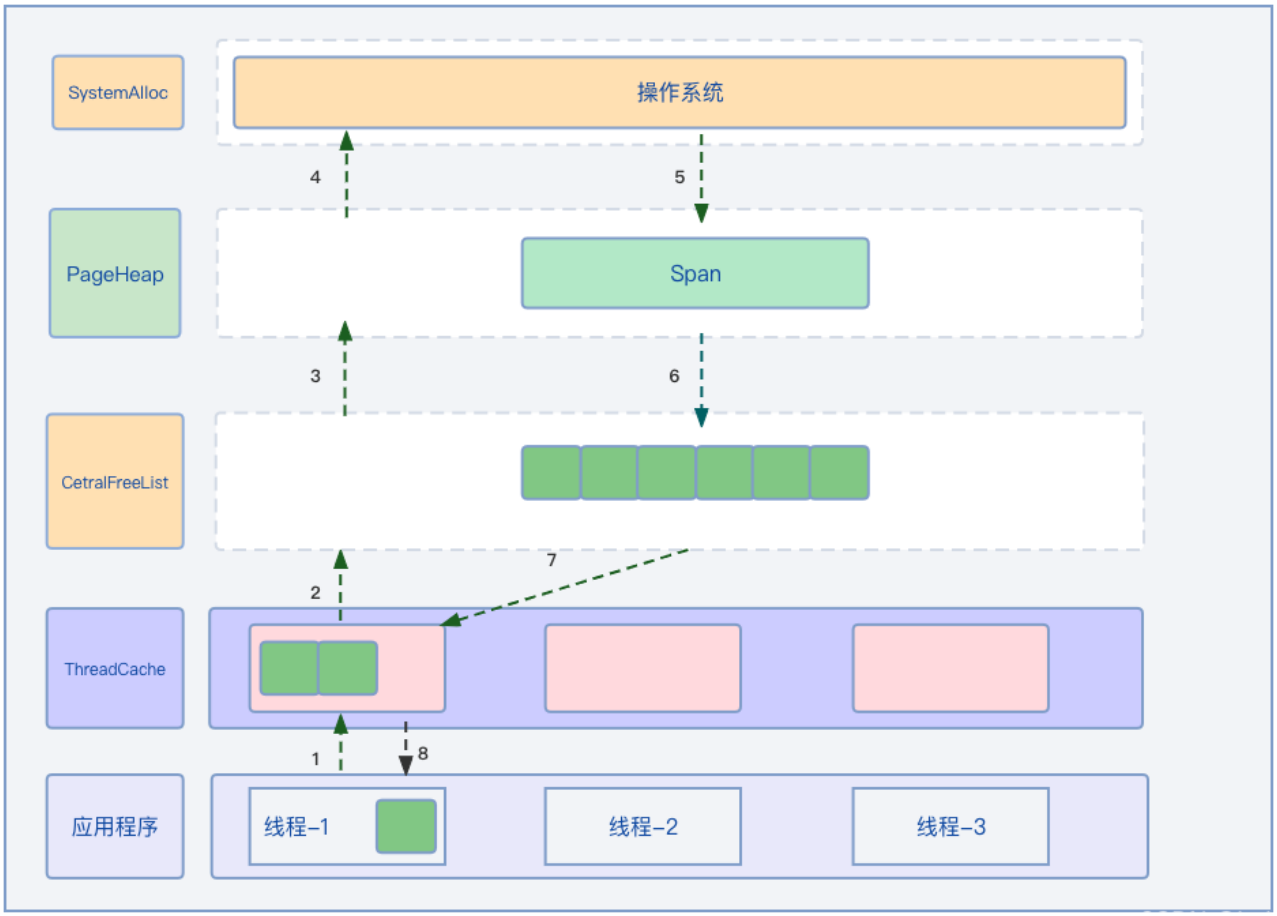
步骤4-5 PageHeap初始化完成之后，同样没有持有任何内存。此时将调用SystemAlloc方法，该方法将向操作系统申请内存，返回的内存空间称为Span，因为步骤1中的类A占用32字节，这里tcmalloc返回的Span是一段大小为8K（默认页大小）的连续内存空间，PageHeap会持有这个Span，然后返回给下层的CentralFreeList。



步骤6 CentralFreeList从PageHeap获得Span之后，会在CentralFreeList中将Span划分为8KB / 32B = 256个对象，每个对象32字节，CentralFreeList通过单向链表管理这些对象。



步骤7 ThreadCache向CentralFreeList申请内存的时候，每次会申请多个对象(类A，多个32字节的对象)，具体多少个，tcmalloc会根据使用的情况动态调整，目的是减少锁竞争。我们的图中假设是3个，这三个对象在ThreadCache中通过双向链表管理。



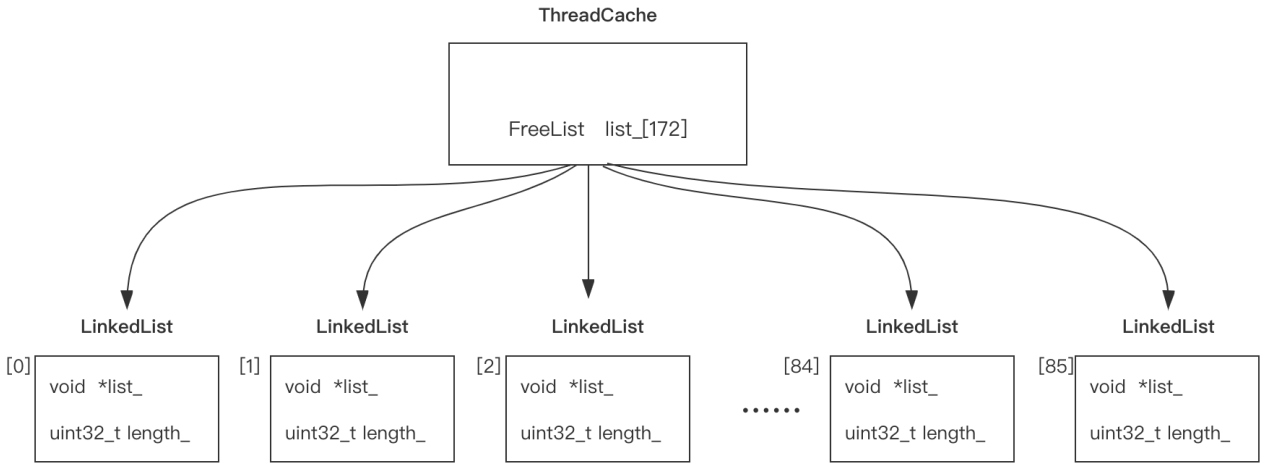
步骤8 ThreadCache向应用程序返回一个32字节的内存地址，用于类A的构造。这就完成了从申请内存到获得内存的整个流程。

从图中可以看出，每个线程会有一个ThreadCache类，全局只有一个CentralFreeLis和PageHeap类，所以应用程序访问CentralFreeLis和PageHeap都需要加锁，但访问ThreadCache不需要加锁，这也是为什么ThreadCache每次向CentralFreeList申请内存对象的时候，会一次申请多个，这样下次线程在申请内存对象的时候，直接从ThreadCache中就可以获得，不需要访问CentralFreeList和PageHeap，减少了锁竞争。

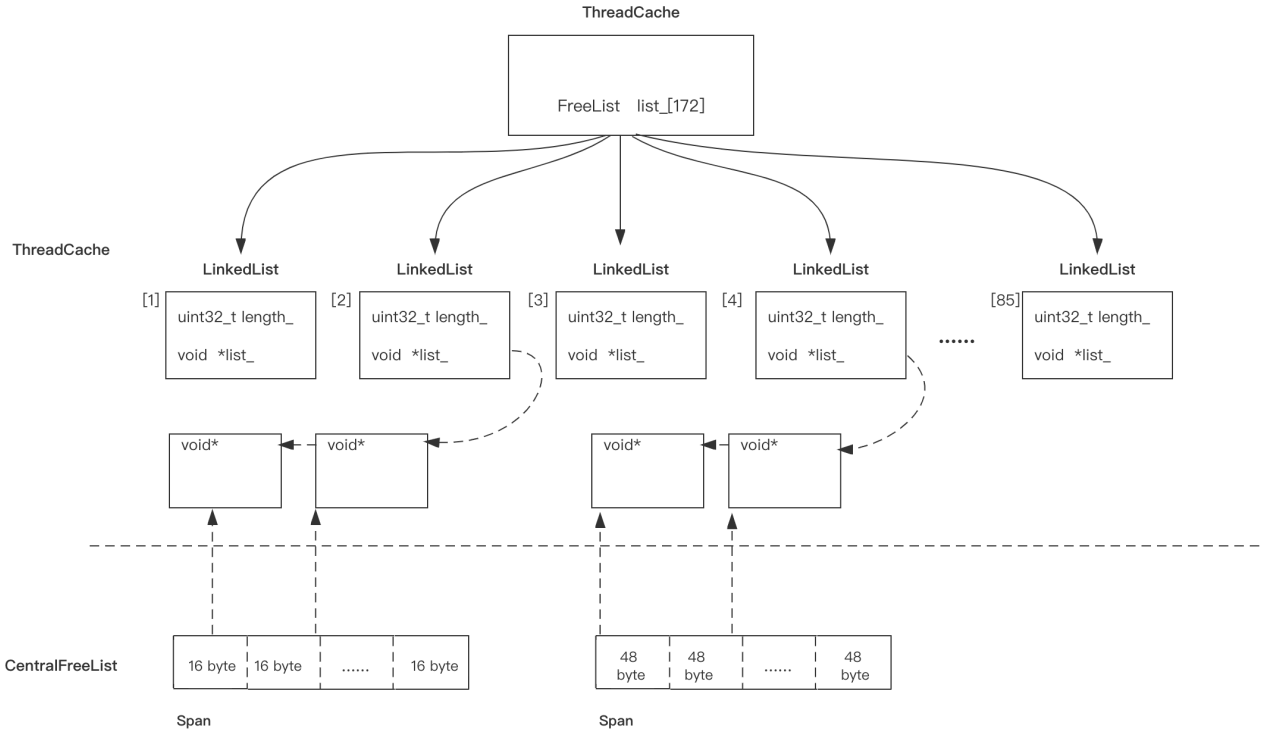
tcmalloc数据结构

1. ThreadCache

每个线程会通过pthread_key_t持有一个线程局部变量(TLS)，变量类型是ThreadCache类型的指针。C++应用程序与tcmalloc交互的第一个数据结构便是ThreadCache。



ThreadCache管理内存的主要成员是FreeList数据结构。上面提到tcmalloc有86种小对象，ThreadCache区分这些对象，将每种大小的对象通过FreeList单向链表管理。链表的实现是由LinkedList类完成，FreeList类继承LinkedList类型，图中直接用LinkedList表示。上图的FreeList list_数组大小为172，为了简单我们只关注前86个成员，初始化后的list_就是上图的样子。现在假设我们定义了一个类A，sizeof(A) = 16,然后我们在代码中执行A *ptr = new A();此时就会调用ThreadCache的Allocate方法，该方法首先通过SizeMap的GetSizeClass获得内存大小为16的数组下标，该下标为2，将2带入list_数组，发现此时LinkedList是空的，没有可分配的内存，这个时候向CentralFreeList申请内存。



当list_[2]的LinkedList为空的时候，会向CentralFreeList申请内存，CentralFreeList实际管理的是Span，Span会对应一种对象大小，Span对应的空间会被平均分配为对象大，假设对象大小为16字节，对应的Span（1页）会被划分为8K / 16B个对象，每个对象大小为16字节。类A占用16字节空间，当LinkedList为空的时候，会从CentralFreeList管理的Span中分配多个16字节的

对象，为什么一次分配多个(分配多少个tcmalloc使用的动态增加算法)呢？因为CentralFreeList全局只有一个，每个线程访问的时候需要加锁，一次分配多个，是为了减少锁开销。假设这次分配了三个8byte对象，返回一个给应用程序，剩下两个放入LinkedList链表，下次就可以直接从LinkedList中分配，而不用访问CentralFreeList。当应该程序用完内存后，调用delete ptr的时候，也是将其放回到LinkedList的链表当中。如果我们有一个类B，sizeof(B) = 48，那么应用程序执行B *ptr = new B()后，ThreadCache的数据结构可能如上图所示。

2. RegionManager

```
1  RegionManager* region_manager = nullptr;
2  void InitSystemAllocatorIfNecessary() {
3    if(region_factory) return;
4
5    preferred_alignment = std::max(pagesize, kMinSystemAlloc);
6    region_manager = new (&region_manager_space) RegionManager();
7    region_factory = new (&mmap_space) MmapRegionFactory();
8  }
```

1 tcmalloc用RegionManager类来管理mmap返回的一段虚拟地址空间，在system-alloc.cc中定义了一个全局的region_manager初始化为空的指针。

2-7 函数InitSystemAllocatorIfNecessary()只有在region_manager为空的情况下执行一次，region_factory初始化为MmapRegionFactory，用于构造一个MmapRegion对象。

```
1  std::pair<void*, size_t> RegionManager::Allocate(size_t size, size_t alignment,......) {
2    AddressRegion*& region = *[&]() {
3      switch (tag) {
4        case MemoryTag::kNormal:
5          return &normal_region_[0];
6      }
7    }();
8    if (region) {
9      std::pair<void*, size_t> result = region→Alloc(size, alignment);
10     if (result.first) return result;
11   }
12   void* ptr = MmapAligned(kMinMmapAlloc, kMinMmapAlloc, tag);
13   region = region_factory→Create(ptr, kMinMmapAlloc, region_type);
14   return region→Alloc(size, alignment);
15 }
```

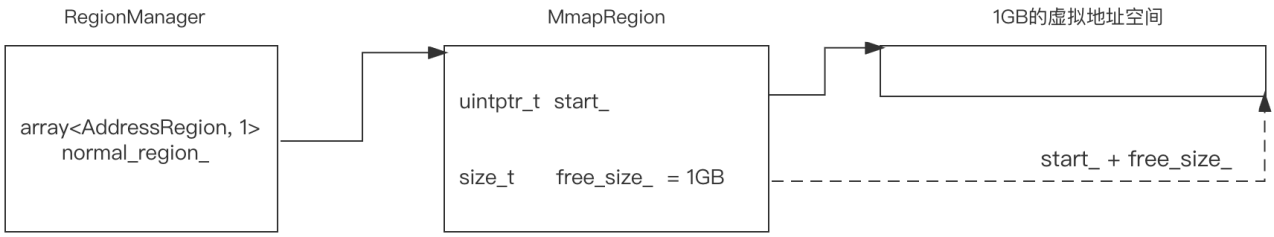
1 tcmalloc外部类(CentralFreeList类)通过调用RegionManager的Allocate方法申请一段alignment对齐，大小为size的内存。

2 normal_region_[0]初始为空地址，所以第一次调用Allocate方法的时候，不会执行8-11行的逻辑。

8-11 当region不为空时，调用MmapRegion的Alloc分配内存，当MmapRegion有足够地址空间时，返回地址。没有足够的内存空间时会返回nullptr，跳过第10行，执行12行MmapAligned向操作系统申请地址空间。

12　kMinMmapAlloc的定义在common.h的310行(64机器)，大小为1GB，MmapAligned调用mmap返回一段虚拟地址空间，该函数保证如果没有失败，返回的地址ptr 1GB对齐,大小为1GB。返回的地址空间是虚拟地址，此时还没有真正的分配内存。

13　region_factory的Create方法通过mmap返回的地址和大小构造一个MmapReion类，构造后的MmapRegion和RegionManager关系如下图。



14　使用MmapRegion的Alloc方法分配size（传入的参数）大小的内存空间，返回分配的地址。

```
 1. 1  std::pair<void*, size_t> MmapRegion::Alloc(size_t request_size, size_t alignment) {
 2. 2    size_t size = RoundUp(request_size, kMinSystemAlloc);
 3. 3    alignment = std::max(alignment, preferred_alignment);
 4. 4    uintptr_t end = start_ + free_size_;
 5. 5    uintptr_t result = end - size;
 6. 6    // 忽略这里的对齐操作代码
 7. 7    size_t actual_size = end - result;
 8. 8    void* result_ptr = reinterpret_cast<void*>(result);
 9. 9    if (mprotect(result_ptr, actual_size, PROT_READ | PROT_WRITE) != 0) {
10. 10   }
11. 11   free_size_ -= actual_size;
12. 12   return {result_ptr, actual_size};
13. 13  }
```

1　上面提到，MmapRegion管理的是一段虚拟地址空间，这段虚拟地址空间不可读不可写。

2　kMinSystemAlloc大小为2MB，RoundUp对请求分配的内存大小request_size向上取整为2MB的整数倍。

4-8　调整MmapRegion的start_、free_size_成员，result_ptr指针指向分配的起始地址。

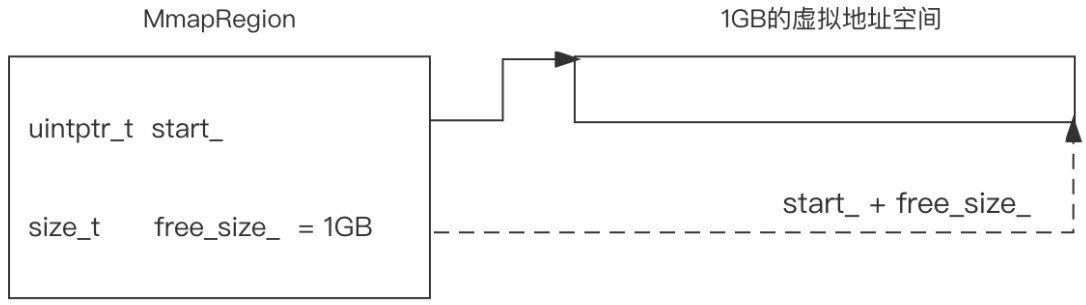
9　调用mprotect函数，更改虚拟地址空间的属性为可读可写，可以认为此时已经分配了物理内存，因为在使用这段地址空间的时候，操作系统会产生缺页中断，完成物理内存的分配，这个过程对我们来说是透明的。

11　更新MmapRegion的free_size_变量。

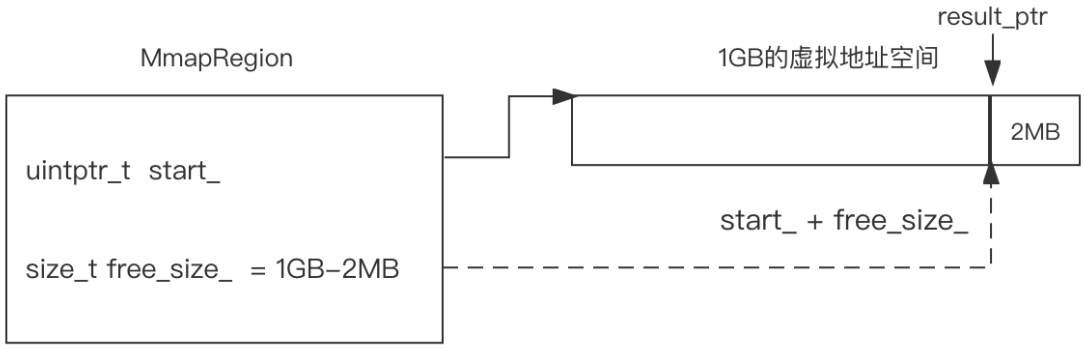
12　返回分配的起始地址和内存实际大小。

假设我们分配2MB的内存空间，则分配前后，MmapRegion的成员变量如下。

分配内存前

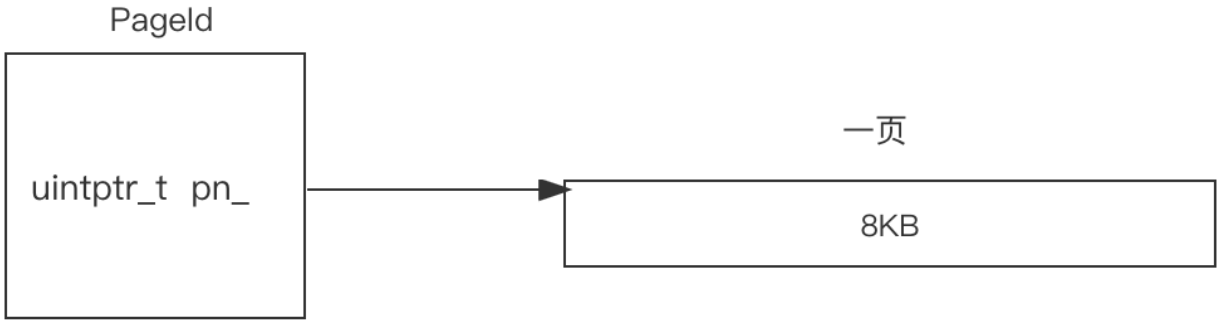


分配内存后



3、PageId

tcmalloc中的一页大小默认情况下为8KB，使用PageId类唯一的表示这一页，假设我们分配了1页的地址空间，会有下面的数据结构。



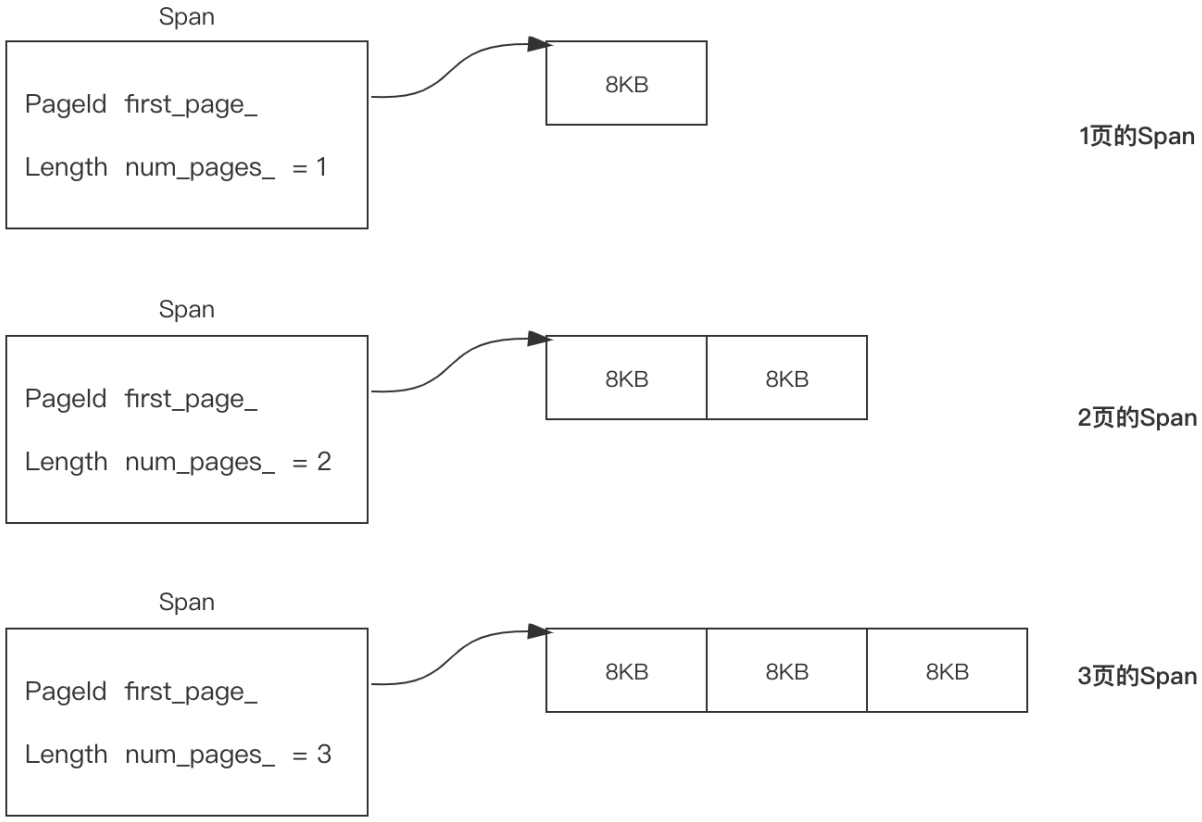
因为page大小固定为8K，所有PageId没有表示内存大小的字段。Page的起始地址都是8K对齐，所以pn_成员实际保存的是起始地址右移13位(2^13=8K)后的结果，PageId类提供了start_uintptr方法返回页的实际地址。

```
1. // common.h 150行
2. inline constexpr size_t kPageShift = 13;
3. // pages.h 120行
```

```
4. uintptr_t start_uintptr() const { return pn_ << kPageShift; }
```

4、Span

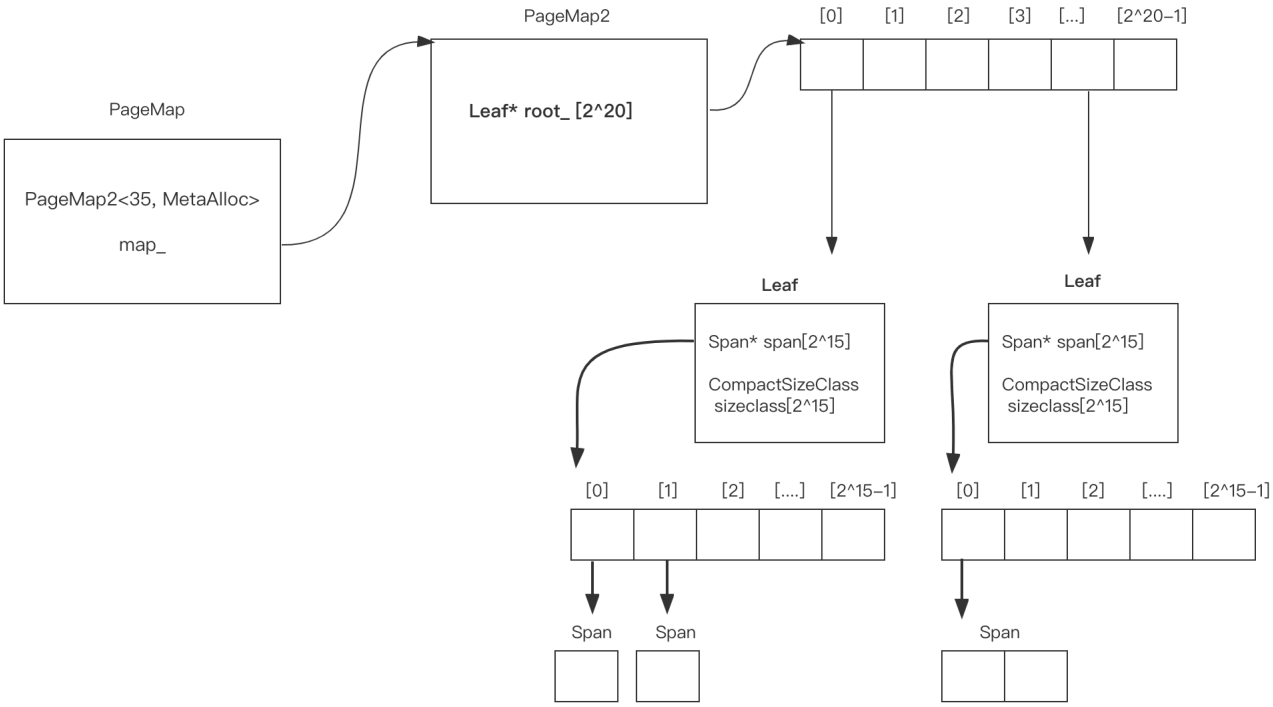
tcmalloc将连续的n页(1 ≤ n ≤ 128，为了简单只考虑最大128页的情况)叫做Span，Span是一段连续的地址空间。所以一共有128种Span,1页组成的Span，2页组成的Span，一直到128页组成的Span，Span的数据结构如下。



Span的first_page_表示第一页的PageId，num_pages_表示组成Span的页数。

5、PageMap

为了(1)快速定位页所属的Span(通过PageId找到Span)，(2)指针对应的内存大小(类似c语言的free ptr场景，不仅要知道释放内存的地址ptr，还要知道释放内存的大小)。tcmalloc使用radix tree数据结构，tcmalloc支持二层和三层的radix tree，通过编译选项指定，我们使用默认的二层radix tree为例说明。



PageMap的成员map_是一个PageMap2模板类型，模板参数Bits在64机器上为35，为什么是35呢？在x86_64架构下的虚拟地址只使用了低48位，高16位(64-48=16)没有使用。PageMap2管理的是Span的地址，Span对应的页地址都是8KB字节对齐，地址的最低13位不需要，所以为了通过PageId找到Span只需要35(48-13)bits。PageMap2将35位的地址分为两部分，高20位作为radix tree第一层的节点索引，低15位作为radix tree的叶子节点。

假设我们现在有一个1页的Span，该Span对应的对象大小都是32字节，我们有一个类A，sizeof(A) = 32，我们现在执行A *ptr = new A()，返回的起始为0x000010002020，我们现在指针delete ptr(0x000010002020)，如何通过PageMap2找到ptr对应的内存大小呢？

```
 1. 1  static constexpr int kLeafBits = 15;
 2. 2  static constexpr int kLeafLength = 1 << kLeafBits;
 3. 3  typedef unintptr_t Number
 4.    // pagemap.h 107行
 5. 4  CompactSizeClass ABSL_ATTRIBUTE_ALWAYS_INLINE
 6. 5    sizeclass(Number k) const ABSL_NO_THREAD_SAFETY_ANALYSIS {
 7. 6     const Number i1 = k >> kLeafBits;
 8. 7     const Number i2 = k & (kLeafLength - 1);
 9. 8     ASSERT((k >> BITS) == 0);
10. 9     ASSERT(root_[i1] != nullptr);
11. 10    return root_[i1]->sizeclass[i2];
12. 11  }
```

5 sizeclass的参数Number的类型是uintptr_t类型，调用sizeclass之前，会先将地址0x000010002020右移13为（8KB对齐），所以传入的参数k = 0x000000008001 = 0x000010002020 >> 13。

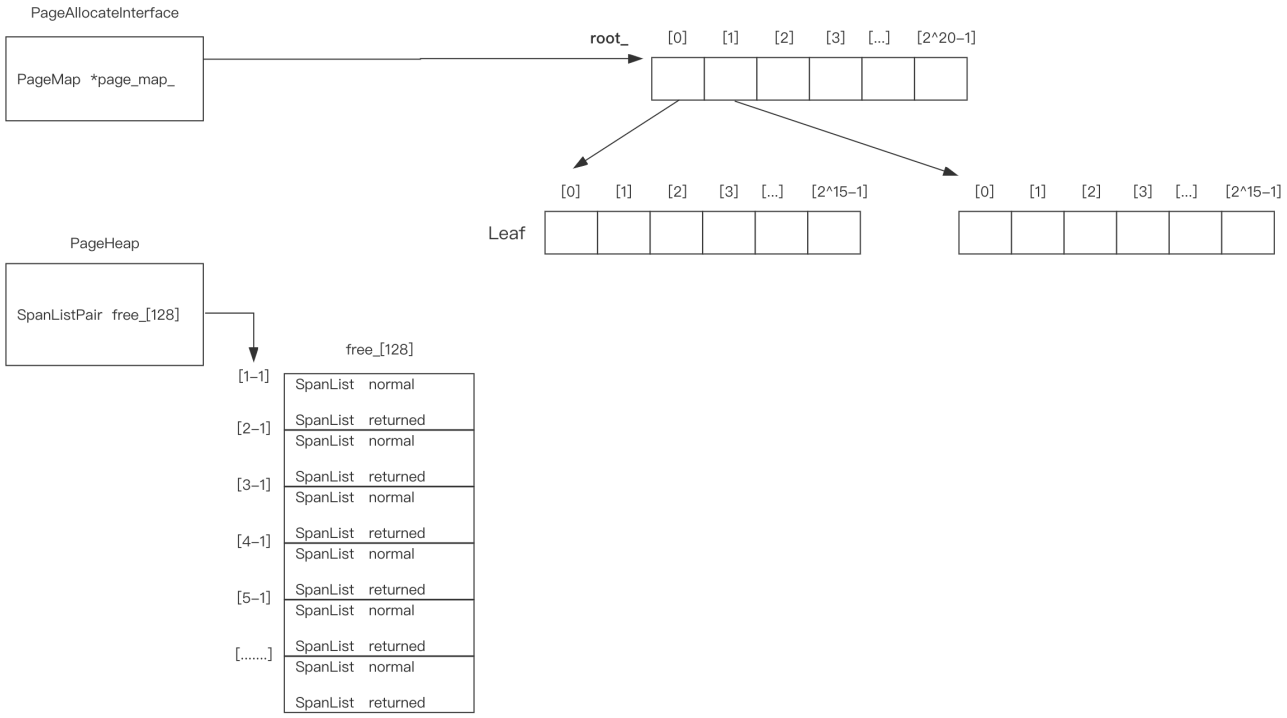
6 i1 = 1 = 0x000000000001 = k >> 15 = 0x000000008001 >> 15。

7 i2 = 1 = 0x000000008001 & 0x7FFF。

10 将i1作为PageMap2的root_数据下标可以得到对应的叶子节点Leaf的指针，Leaf结构定义的sizeclass数组保存的正是Span对应的对象大小，所以将i2作为Leaf结构的sizeclass数据下标就可以得到对象的大小。这样通过PageMap2的sizeclass方法，我们就找到了C++应用程序delete ptr时，ptr对应的内存大小

6、PageHeap

PageHeap通过间接调用region_manager的Alloc方法获得Span。外部方法(StaticForwarder类)通过调用PageHeap提供的AllocateSpan方法获得Span，上面提到Span中一共有128种Span，PageHeap通过大小为128的数组来管理这些Span。从region_manager返回的Span放在SpanListPair的returned双向链表，应用程序释放的Span放在SpanListPair的normal双向链表。PageHeap初始化完成之后的数据结构如下(radix tree数据结构中的数组指针实际是在用到时分配，这里为了方便说明)。



下面我们分析Span分配和归还的4个过程，来理解PageHeap对Span的管理，包括Span的合并和拆分。

(一)、申请2页的Page。

```
Span* PageHeap::AllocateSpan(Length n, bool* from_returned) {

  Span* result = SearchFreeAndLargeLists(n, from_returned);
  if (result != nullptr) return result;
  ......
  if (!GrowHeap(n)) {}
  result = SearchFreeAndLargeLists(n, from_returned);

  return result;
}
```

1 PageHeap通过AllocateSpan方法返回Span，参数n是Span的页数，from_returned标识返回的Span是否来自SpanListPair的returned链表。

3 Span的分配会首先尝试从SpanListPair的normal和returned链表中获得，PageHeap初始化完成之后的链表都是空的，所以不会返回的result = nullptr。

6 执行到这里，说明没有空闲的Span或者没有满足要求页数的Span，此时调用GrowHeap向region_manager申请Span。

```
 1. 1 bool PageHeap::GrowHeap(Length n) {
 2. 2
 3. 3   void* ptr = SystemAlloc(n.in_bytes(), &actual_size, kPageSize, tag_);
 4. 4
 5. 5   if (pagemap_->Ensure(p - Length(1), n + Length(2))) {
 6. 6
 7. 7     Span* span = Span::New(p, n);
 8. 8     RecordSpan(span);
 9. 9     span->set_location(Span::ON_RETURNED_FREELIST);
10. 10    MergeIntoFreeList(span);
11. 11    ASSERT(Check());
12. 12    return true;
13. 13  }
14. 14  ......
15. 15}
```

3 system-alloc.cc中的SystemAlloc方法会调用region_manager的Alloc方法，参数n.in_bytes()是n页Page的字节数，返回结果ptr是分配的内存地址。

5 保证radix tree有足够的内存空间，这个if只要系统可用内存足够，都为true。

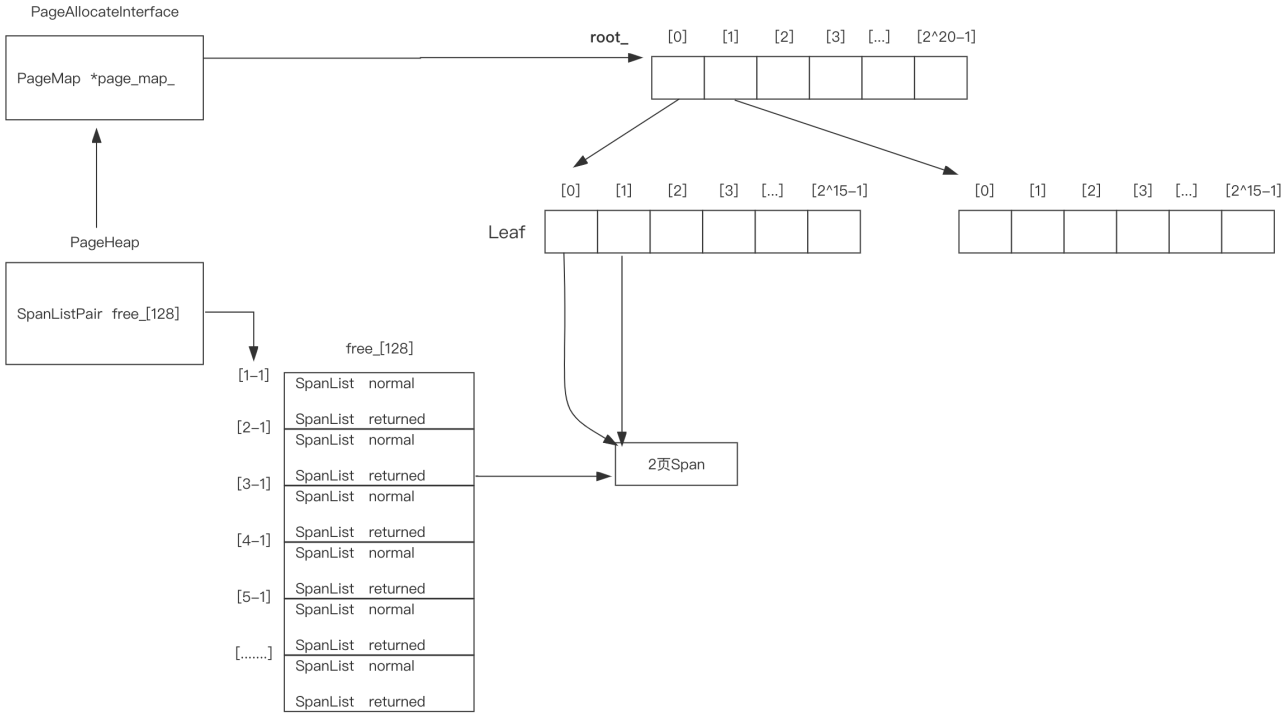
7 用返回的内存地址和页数构造Span对象。

8 RecordSpan方法将记录从PageId到Span的映射，使用的数据结构就是前面介绍的2层radix tree。

9 从region_manager返回的Span位于SpanListPair的returned链表中。

10 将分配的Span放入SpanListPair的returned链表

分配完成之后的数据结构如下。



（二）、 申请3页的Page。

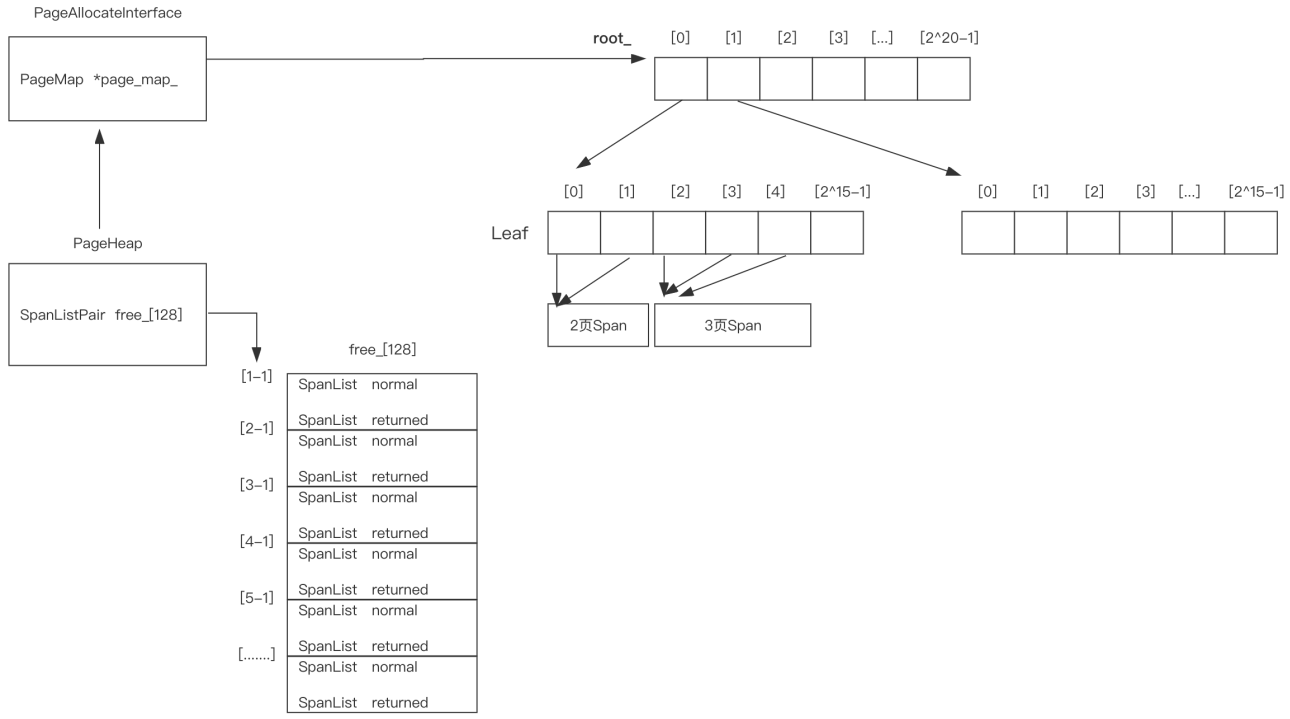
同申请两页的Span一样，申请三页的Span地址也会首先放在returned链表中。

```
1. 1 Span* PageHeap::AllocateSpan(Length n, bool* from_returned) {
2. 2   Span* result = SearchFreeAndLargeLists(n, from_returned);
3. 3   if (result != nullptr) return result;
4. 4    ......
5. 5  }
```

1 外部方法调用AllocateSpan方法申请Span
2 此时2页和3页的SpanListPair的returned链表中存在Span，所以直接从链表中分配。

```
 1. 1 Span* PageHeap::SearchFreeAndLargeLists(Length n, bool* from_returned) {
 2. 2
 3. 3   for (Length s = n; s < kMaxPages; ++s) {
 4. 4     SpanList* ll = &free_[s.raw_num()].normal;
 5. 5     if (!ll->empty()) {
 6. 6       ......
 7. 7     }
 8. 8     ll = &free_[s.raw_num()].returned;
 9. 9     if (!ll->empty()) {
10. 10
11. 11       *from_returned = true;
12. 12       return Carve(ll->first(), n);
13. 13     }
14. 14   }
15. 15
16. 16}
```

3-13　分配Span的时候，会遍历PageHeap的SpanListPair free[128]数组，优先在normal链表查找，此时normal链表为空，从returned中查找。找到之后返回给外部，同时将Span从normal或者returned中移除，但是仍然会保留在radix tree中。所以，外部方法申请2 page的Span和3 page的Span之后的数据结构如下。



（三）、释放2页的Page，然后释放3页的Span。

```
1. 1 void PageHeap::Delete(Span* span) {
2. 2   ......
3. 3   span→set_location(Span::ON_NORMAL_FREELIST);
4. 4   MergeIntoFreeList(span);  // Coalesces if possible
5. 5   ASSERT(Check());
6. 6}
```

1　释放Span的操作由PageHeap的Delete方法完成。

4　MergeIntoFreeList记录空闲的Span。

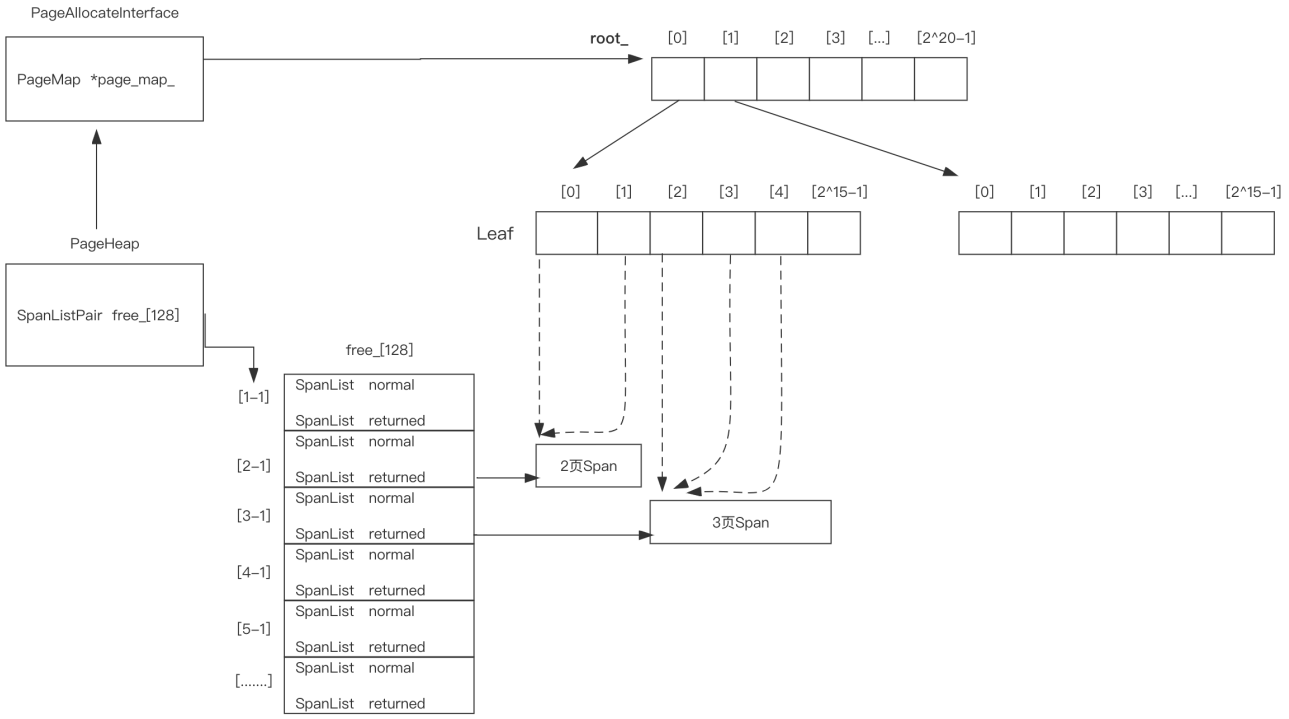
```
 1. 1 void PageHeap::MergeIntoFreeList(Span* span) {
 2. 2   ......
 3. 3   const PageId p = span→first_page();
 4. 4   const Length n = span→num_pages();
 5. 5   Span* prev = pagemap_→GetDescriptor(p - Length(1));
 6. 6   if (prev ≠ nullptr && prev→location() == span→location()) {
 7. 7     // Merge preceding span into this span
 8. 8     ......
 9. 9     RemoveFromFreeList(prev);
10. 10    Span::Delete(prev);
11. 11    span→set_first_page(span→first_page() - len);
12. 12    span→set_num_pages(span→num_pages() + len);
13. 13    pagemap_→Set(span→first_page(), span);
14. 14  }
15. 15  ......
```

```
16. 16  PrependToFreeList(span);
17. 17}
```

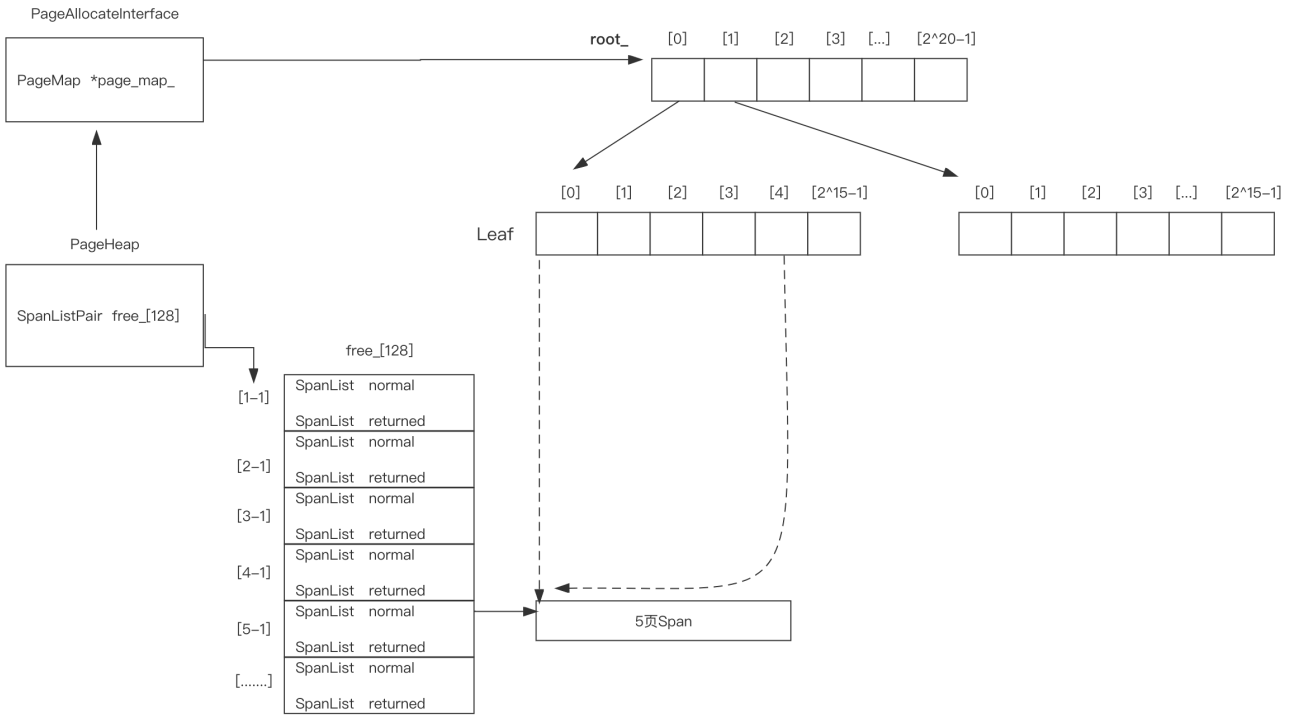
1-4 计算Span的PageId和页数。

5 通过radix找到当前Span第一页的前一页的Span地址。当我们释放一个2页的Span时，prev = nullptr。此时直接执行16将Span放入SpanListpair的normal链表中。

6-14 当我们释放3页的Page的时候，如果2页的Span和3页的Span不是连续的页(连续的内存地址)，prev = nullptr;释放完之后的数据结构如下。



如果释放的两个Span是连续的内存地址，tcmalloc为了减少内存碎片和分配效率，将这2个Span合并为一个5页的Span，然后放在位于free[5-1]的normal双向链表中，并且调整radix tree的指针，将Leaf的[0]和[4]指向合并后的Span的地址，将[1][2][3]位置的指针置为空。合并之后的数据结构如下图。



(四)、申请1页的Page。

```
 1. 1 Span* PageHeap::SearchFreeAndLargeLists(Length n, bool* from_returned) {
 2. 2   ......
 3. 3   for (Length s = n; s < kMaxPages; ++s) {
 4. 4     SpanList* ll = &free_[s.raw_num()].normal;
 5. 5     if (!ll->empty()) {
 6. 6       ASSERT(ll->first()->location() == Span::ON_NORMAL_FREELIST);
 7. 7       *from_returned = false;
 8. 8       return Carve(ll->first(), n);
 9. 9     }
10. 10    ......
11. 11  }
12. 12  ......
13. 13}
```

3-10 外部方法通过PageHeap申请Span的时候，如果没有完全匹配的Span(申请的Span和某个normal链表的Span页数不一样)，会尝试去更大的Span中分配。比如现在分配一个1页的Span的，1页的Span的free[1-1]的normal和returned链表都为空，此时会从2页的Span到128页的Span中遍历，直到找到第一个不为空的normal和returned链表，在我们的例子的中，找到了一个5页的Span，此时会将这个Span从free_[5-1]对应的norma数组中删除，然后划分为1页的Span和4页Span，1页的Span的返回给申请者，4页的Span的放入free[4-1]_的normal链表中。还会调整radix tree叶子节点Leaf的节点指针。划分Span的功能由Carve函数完成。

```
 1. 1 Span* PageHeap::Carve(Span* span, Length n) {
 2. 2
 3. 3   RemoveFromFreeList(span);
 4. 4   span->set_location(Span::IN_USE);
 5. 5   const Length extra = span->num_pages() - n;
 6. 6   if (extra > Length(0)) {
 7. 7     Span* leftover = nullptr;
 8. 8
 9. 9     if (pagemap_->GetDescriptor(span->first_page() - Length(1)) == nullptr &&
```

```
10. 10        pagemap_->GetDescriptor(span->last_page() + Length(1)) != nullptr) {
11. 11      ......
12. 12    } else {
13. 13      leftover = Span::New(span->first_page() + n, extra);
14. 14    }
15. 15    leftover->set_location(old_location);
16. 16    RecordSpan(leftover);
17. 17    PrependToFreeList(leftover);  // Skip coalescing - no candidates possible
18. 18    leftover->set_freelist_added_time(span->freelist_added_time());
19. 19    span->set_num_pages(n);
20. 20    pagemap_->Set(span->last_page(), span);
21. 21  }
22. 22  ASSERT(Check());
23. 23  return span;
24. 24}
```

3 首先将5页Span从free_[5-1]的SpanListPair链表中删除。

5 extra = 5 - 1 = 4，5页的Span,分配一个1页的Span,剩下4页。

9-10 在我们的例子中为false。

13 构造一个4页的Span。

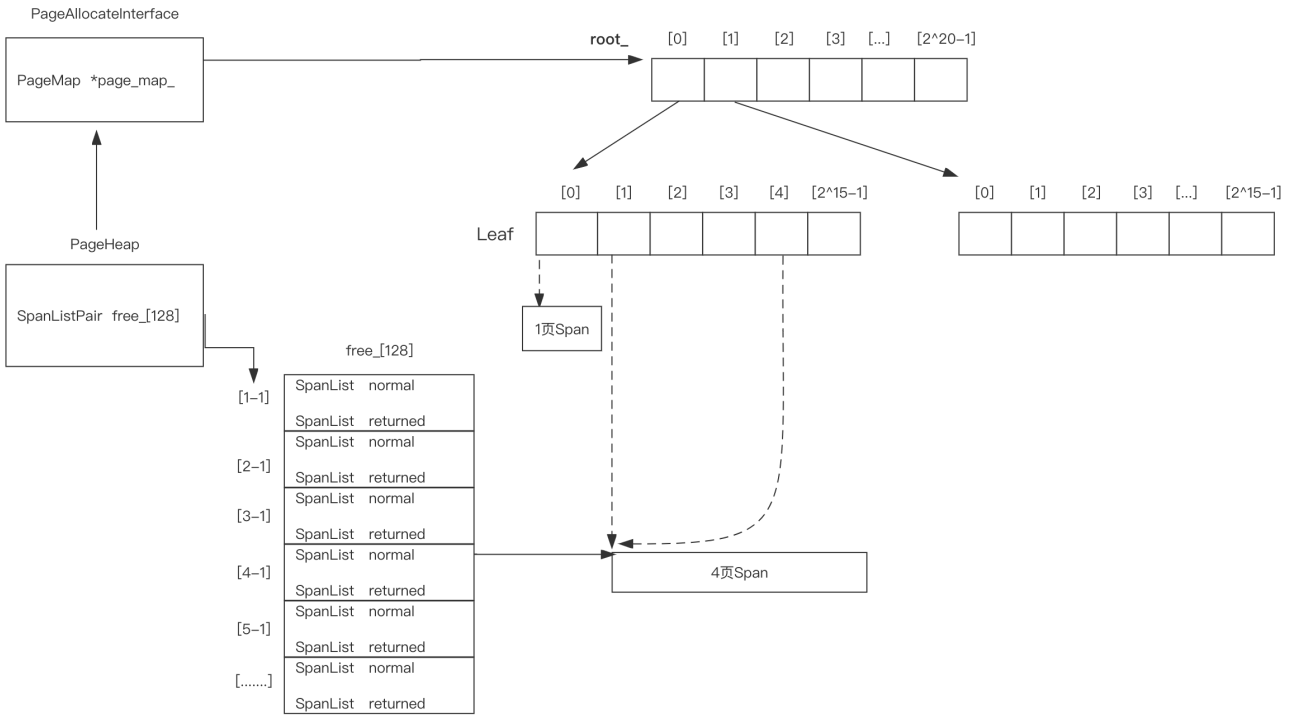
16 将4页的Span记录在radix tree中。

17 将4页的Span记录在SpanListPair的normal链表中。

19-20 将分配的1页Span记录在radix tree中。

24 返回1页的Span。

当分配1页的Span完成之后，数据结构如下图所示。



7、SizeMap

tcmalloc为了避免内存碎片，将小对象按照内存大小分为85种，8字节的对象，16字节的对象，一直到262144字节(256KB)的对象，所有小对象的大小定义在size_classes.cc文件中。大于256KB的内存不属于小对象，tcmalloc会单独处理。

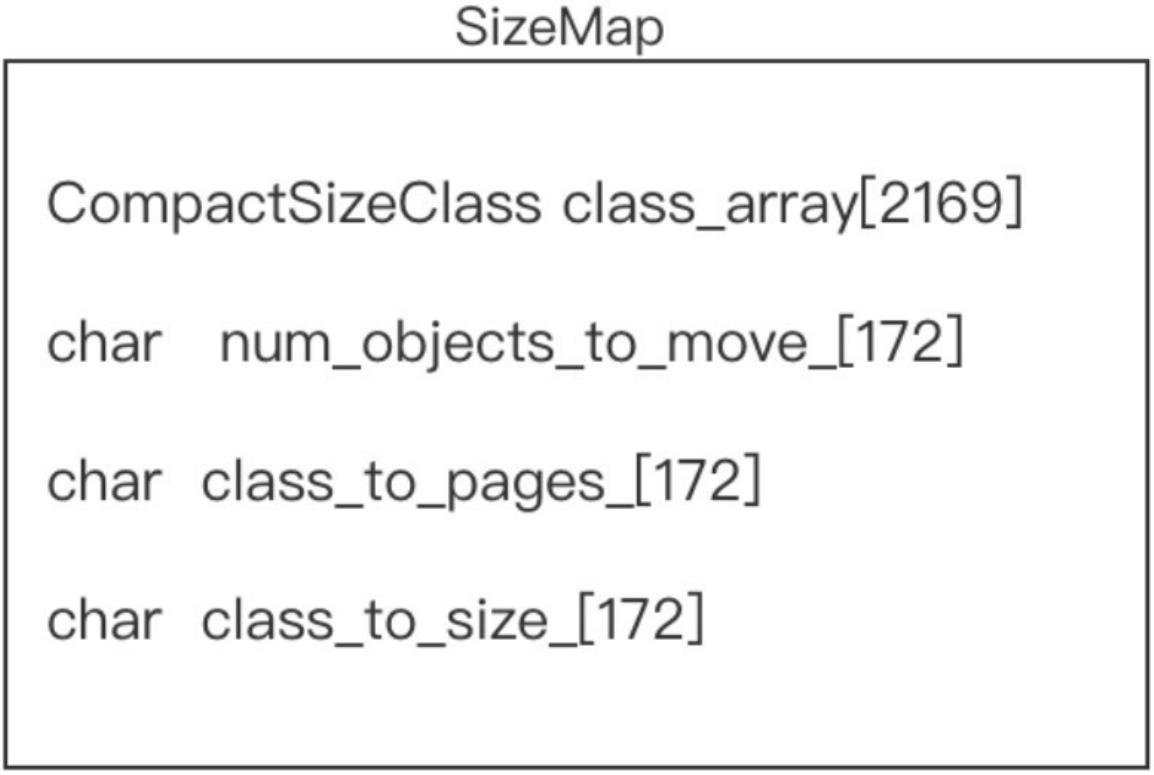
```
 1. const SizeClassInfo SizeMap::kSizeClasses[SizeMap::kSizeClassesCount] = {
 2.     // <bytes>, <pages>, <batch size>    <fixed>
 3.     {        0,       0,           0},  // +Inf%
 4.     {        8,       1,          32},  // 0.59%
 5.     {       16,       1,          32},  // 0.59%
 6.     {       32,       1,          32},  // 0.59%
 7.     {       48,       1,          32},  // 0.98%
 8.     {       64,       1,          32},  // 0.59%
 9.     {       80,       1,          32},  // 0.98%
10.     {       96,       1,          32},  // 0.98%
11.
12.     {   237568,      29,           2},  // 0.02%
13.     {   262144,      32,           2},  // 0.02%
14. };
```

tcmalloc在内存分配的时候都是以这些对象大小为基准，c++应用申请的内存大小和tcmalloc实际分配的内存大小对应关系，可以用下面的表说明。

C++应用程序要求分配的内存大小	tcmalloc实际分配的内存大小
1 字节 2 字节 3 字节 4 字节 5 字节 6 字节 7 字节 8 字节	8 字节
9 – 16 byte	16 byte
17 – 32 byte	32 byte
33 – 48 byte	48 byte
49 – 64 byte	64 byte
65 – 80	80 byte
81 – 96	96
.....	.....
221185 – 237568 byte	237568 byte
237569 – 262144 byte	262144 byte

tmalloc将C++应用申请分配的内存大小对齐到对象的大小。比如应用申请分配9-16字节之间某个大小的内存，tcmalloc实际都分配16字节的对象大小。

SizeMap的数据结构如下。



小对象的85种对象大小保存在SizeMap的class_to_size_成员中，如何通过C++应用申请的内存大小找到实际应该分配的对象大小呢？tcmalloc的处理分为两步：

（1）调用SizeMap的GetSizeClass方法，参数size为C++应用申请分配的内存大小，size_class为返回值。

```
1. template <typename Policy>
2.   inline bool ABSL_ATTRIBUTE_ALWAYS_INLINE GetSizeClass(Policy policy,
3.                                                        size_t size,
4.                                                        uint32_t* size_class)
```

（2）将（1）中返回的size_class作为SizeMap成员class_to_size_数组的下标，就可以得到tcmalloc实际分配的内存大小。

比如我们现在A *ptr = new A(); A占用内存24字节，此时会申请24字节的内存空间，tcmalloc首先调用SizeMap的GetSizeClass方法返回的size_class等于3。将3作为数组class_to_size_的下标，class_to_size_[3]等于32，所以应用申请24字节的内存，tcmalloc实际分配32字节的内存空间。