Slab 共享内存使用及实现原理

在生产环境中，Nginx 一般采用的是一个 master 进程，多个 worker 进程的模型，这样设计有一个好处，就是服务更加健壮，但是另一方面，如果一个请求分布在不同的进程上，当进程间需要互相配合才能完成请求的处理时，进程间通信的困难就凸显出来，虽然已经有许多进程间通信的方法，但是那些都只适合简单语意的场景，如果进程间需要交互复杂对象，如树、图等，则需要新的进程间通信方式，那就是今天要讲述的 slab 共享内存。

在讲述如何使用 slab 共享内存之前，先介绍几个和 slab 相关密切的结构体：

1、 ngx_cycle_t 中的 shared_memory 成员

```
struct ngx_cycle_s {
    ……
    ngx_list_t    shared_memory;
    ……
};
```

我们知道，在 Nginx 中，无论是 master 还是 worker 进程，ngx_cycle_t 结构体对于一个进程来说是唯一的。在读取或者初始化配置文件时，如果某个 Nginx 模块需要使用共享内存，则通过调用 ngx_shared_memory_add 函数向上面的全局共享内存链表 shared_memory 中添加一个共享内存的节点信息（并未真正申请内存并做初始化），并返回这个节点供模块后续操作。

2、ngx_shm_zone_t 结构体

这个结构体就是 ngx_shared_memory_add 函数返回的节点地址，其定义如下：

```
struct ngx_shm_zone_s {
    void    *data;    //  作为 init 方法的参数，用于传递数据
    ngx_shm_t    shm;     //描述共享内存的结构体
    ngx_shm_zone_init_pt    init; //在真正创建好 slab 共享内存池后调用这个方法
    void    *tag;     //对应于 ngx_shared_memory_add 的 tag 参数
    ngx_uint_t    noreuse;    /* unsigned    noreuse:1; */
};
```

这个结构体中各个成员的意义和用法在注释中已经说明，其中需要强调的是 tag 成员，这个成员是用来防止两个毫不相关的 Nginx 模块定义的共享内存恰好具有同样的名字，从而导致数据管理的混乱。通常，tag 参数均会传入本 Nginx 模块的结构体的地址。

3、ngx_slab_pool_t 结构体

这个结构体就是 slab 共享内存池的管理结构体，用于描述这块共享内存池的信息，其定义如下：

```
typedef struct {
    ngx_shmtx_sh_t       lock;
    size_t               min_size;    //一页中最小内存块(chunk)大小
    size_t               min_shift;    //一页中最小内存块对应的偏移
```

```
    ngx_slab_page_t    *pages;        //slab 内存池中所有页的描述
    ngx_slab_page_t    *last;         //指向最后一个可用页
    ngx_slab_page_t     free;         //内存池中空闲页组成链表头部
    u_char              *start;       //实际页起始地址
    u_char              *end;         //实际页结束地址
    ngx_shmtx_t          mutex;       //slab 内存池互斥锁
    u_char              *log_ctx;
    u_char               zero;
    unsigned             log_nomem:1;
    void                *data;
    void                *addr;        //指向内存池起始地址
} ngx_slab_pool_t;
```

内存池管理结构中成员的具体作用在下面将会介绍的 slab 共享内存的内存布局时会有体现。

4、ngx_slab_page_t 结构体

slab 共享内存有一个很重要的思想就是分页机制，这和操作系统的分页是类似的。例如将一块 slab 共享内存以 4KB 作为一页的大小分成许多页，然后每个页就用 ngx_slab_page_t 结构体进行管理，其定义如下：

```
struct ngx_slab_page_s {
    uintptr_t    slab;   //多用途，描述页相关信息，bitmap 和内存块大小
    ngx_slab_page_t   *next;   //指向双向链表中的下一个页
    uintptr_t    prev;   //指向双向链表前一个页，低 2 位用于存放内存块类型
};
```

这个结构体中的 slab 和 prev 都是有多用途的，它们的意义如下：

	ngx_slab_page_t		
	Slab	prev	next
小块内存 NGX_SLAB_SMALL	表示该页中存放的等长内存块的大小对应的偏移量	指向双向链表的前一个元素，低 2 位为 11，以 NGX_SLAB_SMALL 表示小块内存	指向双向链表中的下一个元素（链表中页等分的内存块大小相等）
中等内存 NGX_SLAB_EXACT	作为 bitmap 表示该页中内存块的使用情况，bitmap 中某个位为 1 表明对应内存块已使用	指向双向链表的前一个元素，低 2 为 10，NGX_SLAB_EXACT 表示中等大小内存	同上
大块内存 NGX_SLAB_BIG	高 16 位作为 bitmap 表示该页中内存块使用情况，低 4 位用来表示内存块大小对应的位移	指向双向链表的前一个元素，低 2 位为 01，NGX_SLAB_BIG 表示大块内存	同上

超大块内存 NGX_SLAB_PAGE	超大块内存会使用一页或者多页，这些页都在一起使用，对于这批页面中的第一页，slab 前三位会被设置为 NGX_SLAB_PAGE_START, 其余为表示紧随其后的相邻的同批页面书；其余页的 slab 会被设置为 SLAB_PAGE_BUSY	指向双向链表的前一个元素，低 2 位为 00，NGX_SLAB_PAGE 表示大块内存	同上

上面就是 slab 涉及到的几个结构体及其相关成员的意义和用法，如果一个 Nginx 模块需要使用要共享内存，则一般遵循下面的使用步骤：

1、 在读取和初始化配置文件时，调用 ngx_shared_memory_add 向全局共享内存链表 ngx_cycle_t->shared_memory 中添加一个共享内存节点，此时并没有真正分配内存及初始化，告诉 Nginx 内核模块需要使用共享内存，等 Nginx 内核读取完配置文件，初始化服务的时候才会去申请共享内存，并做初始化，此部分详见 ngx_init_cycle。这个函数一般是由模块调用的。

2、 Nginx 内核读取完配置文件后，初始化服务的时候会在 ngx_init_cycle 中调用 ngx_shm_alloc 和 ngx_slab_init 遍历全局 slab 共享内存链表 shared_memory，依次申请和初始化每一块 slab 共享内存。ngx_shm_alloc 函数用于申请共享内存，其具体通过系统调用 mmap 实现；ngx_slab_init 函数用于初始化 ngx_shm_alloc 申请的用作 slab 的共享内存。

3、 在模块向 Nginx 内核申请了共享内存节点以及 Nginx 内核完成共享内存申请和初始化之后，模块就可以根据自身功能需要去申请和释放共享内存了，其涉及的主要函数如下：

1） 申请函数：ngx_slab_alloc、ngx_slab_alloc_locked 和 ngx_slab_alloc_pages

2） 释放函数：ngx_slab_free、ngx_slab_free_locked 和 ngx_slab_free_pages

介绍完模块使用 slab 共享内存的一般步骤之后，来介绍下 slab 共享内存的实现原理，这里只是对内存布局，申请和释放过程简单做了总结，想要完全知道实现原理，还是得去看源码。

首先来看下 slab 共享内存的内存布局。slab 内存布局如下图所示，需要补充说明的是图中的结构体成员为了画图的方便对其定义顺序进行了必要的调整，并且有部分成员没有列举。

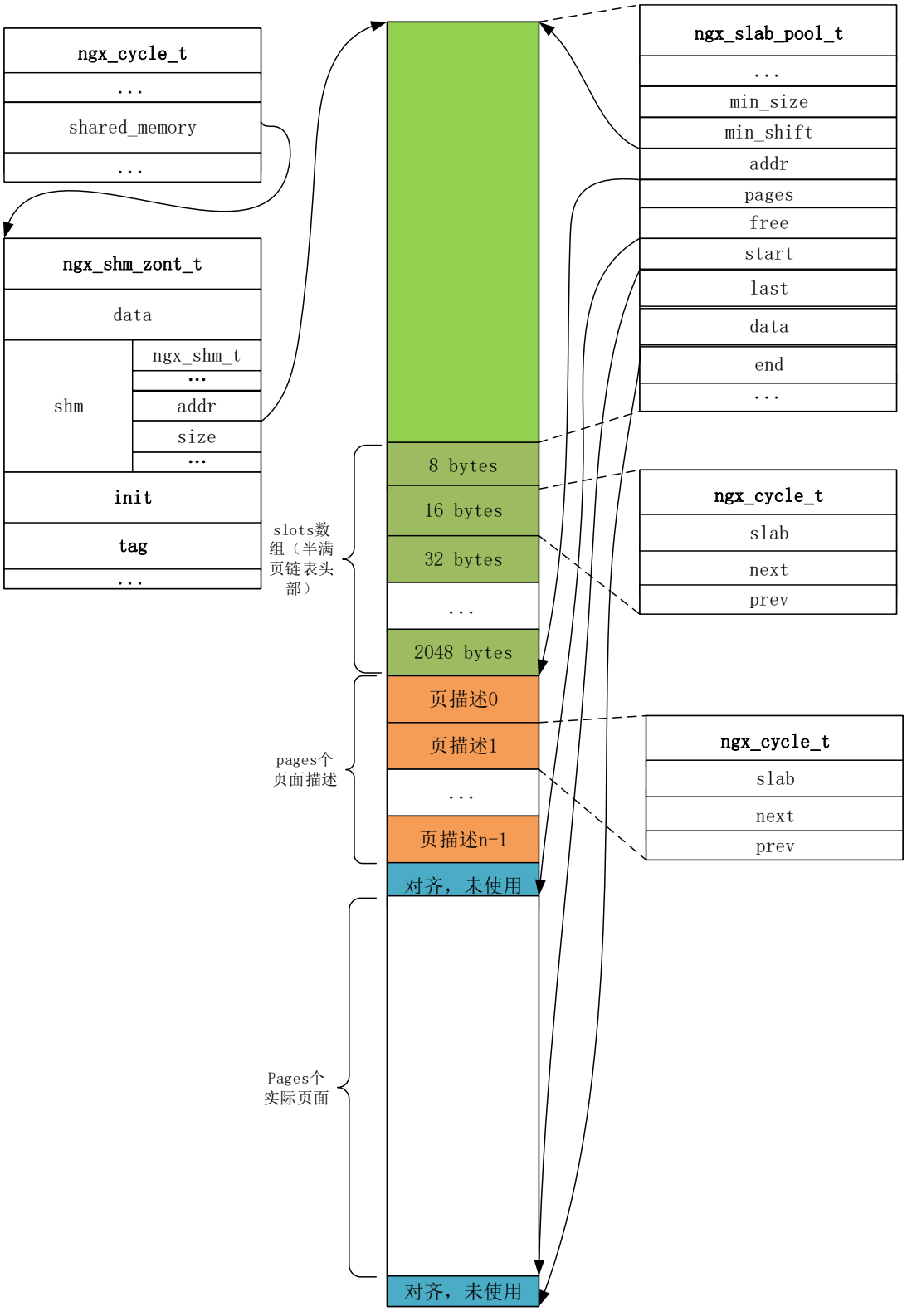

图 1 slab 内存布局

slab 内存布局体现在 ngx_slab_init 函数对 slab 共享内存的初始化中，下面对内存布局进行简单的介绍：

1、 ngx_slab_pool_t。这个是整块 slab 共享内存池的管理结构，其中描述的就是

这块 slab 共享内存池的信息，如实际用于分配的页的起始地址、页描述数组首地址等。

2、 slots 数组。在 slab 共享内存池中有三种状态的页，分别是空闲页、半满页和全满页。ngx_slab_pool_t->free 便指向的就是空闲页组成的链表，半满页就是指其中的内存块未完全被分配完的页，这些页会以其中内存块的大小对应的位移作为下标存放自 slots 数组对应的元素中，并且互相之间也是以链表连接；然后对于所有内存块都已经使用完的全满页，其会脱离半满页链表。综上，slots 数组中每个元素存放的是对应页中内存块大小的半满页链表。

3、 pages 数组。pages 数组是 slab 内存池中用于管理实际用于分配的页的管理结构组成的数组。

4、 因为 slab 内存池中是通过地址对齐的方式将每个用于实际分配的页和其对应的管理结构关联起来的，因此每一页的首地址都必须是以 4k 大小对齐的。图中部分便是由于地址对齐而浪费的内存空间。

5、 实际用于分配的页。其是通过地址对齐方式同对应的页管理结构关联，即 pages 数组中的元素。

6、 同第 4 部分中一样也是浪费的内存空间。

再看下 slab 共享内存的申请。slab 共享内存的申请涉及到的接口函数有以下几个：ngx_slab_alloc、ngx_slab_alloc_locked 和 ngx_slab_alloc_pages。我们以申请一块大小为 13 bytes 的内存块为例。

1、 初始状态下所有页都是空闲页，需要注意的是空闲页之间并不是通过链表连接的，其布局如下：

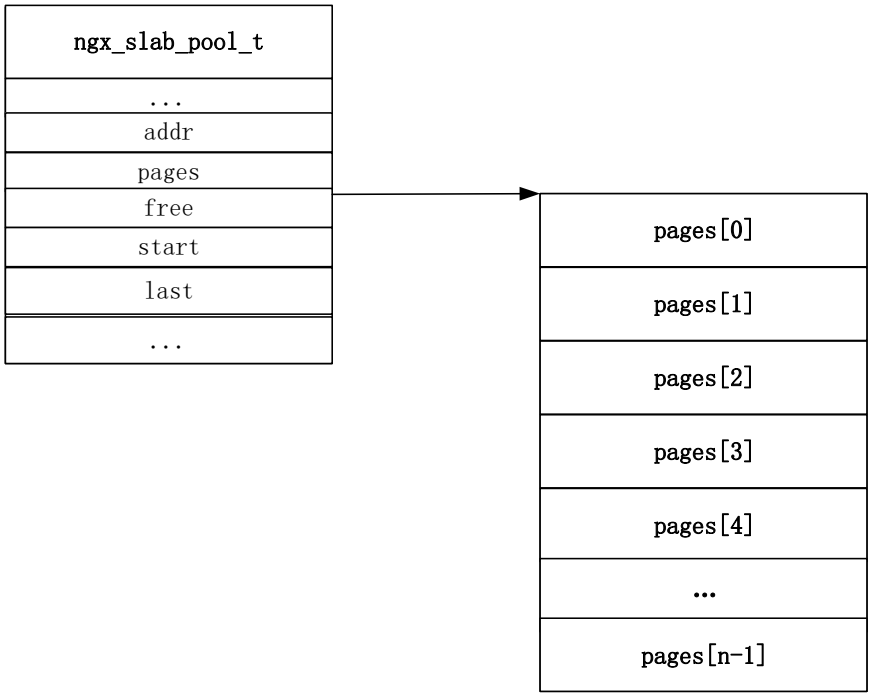

图 2 初始状态

2、 当前 Nginx 内核支持以下几种规格内存块大小的页：8 bytes、16 bytes、32 bytes、64 bytes、128 bytes、…、2048 bytes，这说明 slots 数组中有 9 个元素。如果我们要申请 13 bytes 大小的内存块，由于其介于[8, 16]bytes 之间，因此要申请其中存放的内存块是 16 bytes 的页，此时内存池中布局如下：

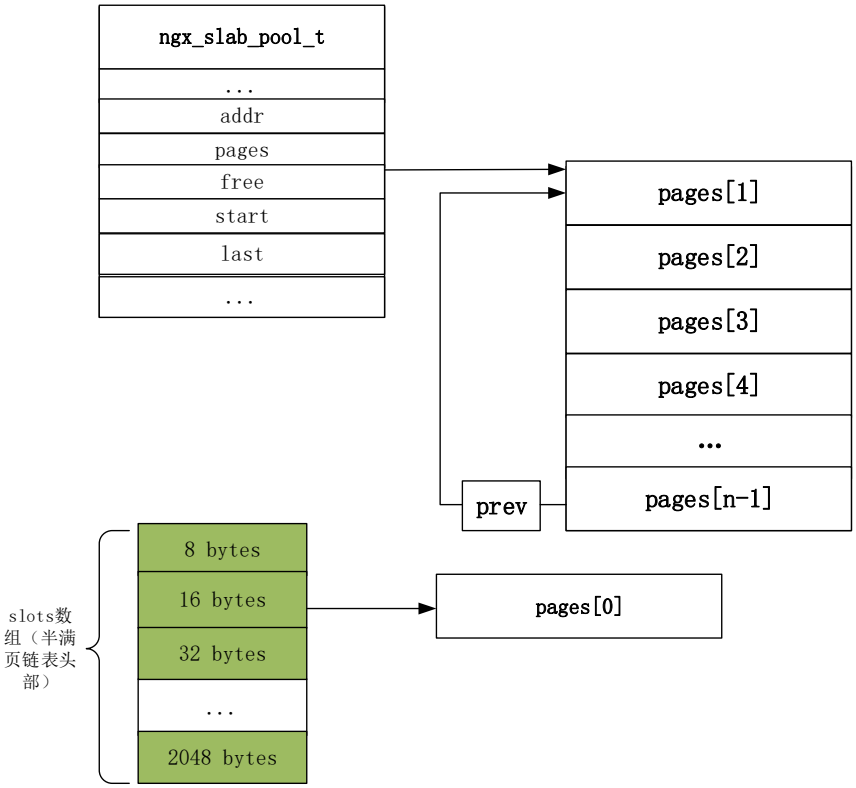

图 3 申请一页状态

3、 我们的目的是为了申请一块大小为 13 bytes 的内存块，因为之前对应的 slots[1]中是空链表，所以会从实际页面中申请一个页，并将其在 free 空闲链表中对应的页管理结构挂在 slots[1]中，并且该页中的内存块大小为 16 bytes。那此时页管理结构中每个成员的值如下：

成员	意义描述	值
uintptr_t slab	表示该页中内存块大小	0x000000004
ngx_slab_page_t *next	表示同处 slots[1]的下一个链表元素，如果没有，为 NULL	NULL
uintptr_t prev	低两位为 11，表示小块内存，其余位为 slots[1]地址	&slots[1] | NGX_SLAB_SMALL

4、 因为 13 bytes 的内存块在 slab 中属于是 NGX_SLAB_SMALL 类型的内存，其在一页中可以划分的内存块数量大于 8 * sizeof(uintptr_t)个，因此一个 slab 的位数不足以表示其中内存块的使用情况，此时其页管理结构中的

ngx_slab_page_t->slab 用于表示内存块大小对应的偏移，那么它的 bitmap 用什么来表示呢？答案是用页中开头的内存块依据需要作为 bitmap 来使用。还是以 13 bytes 为例(以 32 位为例)，计算如下：虽然我们要申请的是 13 bytes，但是 slab 分配的内存会是 16 bytes。

内存块为 16 bytes 的页中内存块数量 = 1 << (ngx_pagesize_shift - 4) = 256 个

所需 bitmap 个数 = 1 << (ngx_pagesize_shift - 4) / (8 * sizeof(uintprt_t)) = 8 个

所有 bitmap 占用内存块个数 = 8 * sizeof(uintptr_t) / 16 = 2 个

此时该页内存布局和 bitmap 值如下：

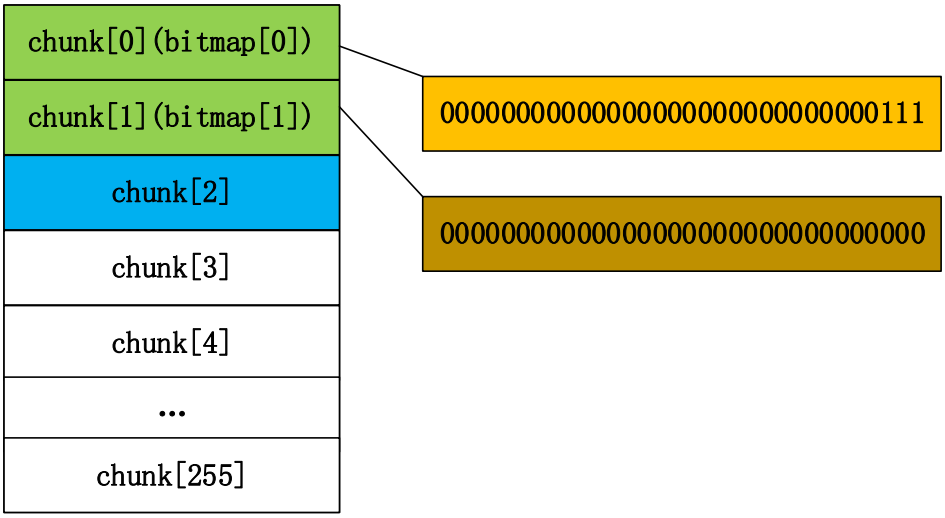

图 4 页布局和 bitmap 值

bitmap[0]的第三位为 111，表示低两位对应的内存块用于 bitmap，第三位对应的是此次申请的内存块，以上三块均在使用中，故均为 1。

最后讲述下 slab 共享内存的释放，释放的具体实现较为复杂，其中包含了全满页释放内存块后入半满页链表，半满页释放内存块后如果其中内存块均为使用则需要如空闲链表，在最开始处说过所有未使用的空闲页之间不是以链表相连接的，但是如果是重入空闲链表，则重入的页和剩余未使用的页之间使用链表相连接的。

还是以刚刚申请的内存块为例。如果使用共享内存的模块释放了申请的共享内存，那么由于此时其中的所有内存块均未使用（用于 bitmap 的两个内存块除外）需要将该页重新加入空闲页组成的链表中，此时 slab 内存池状态如下：

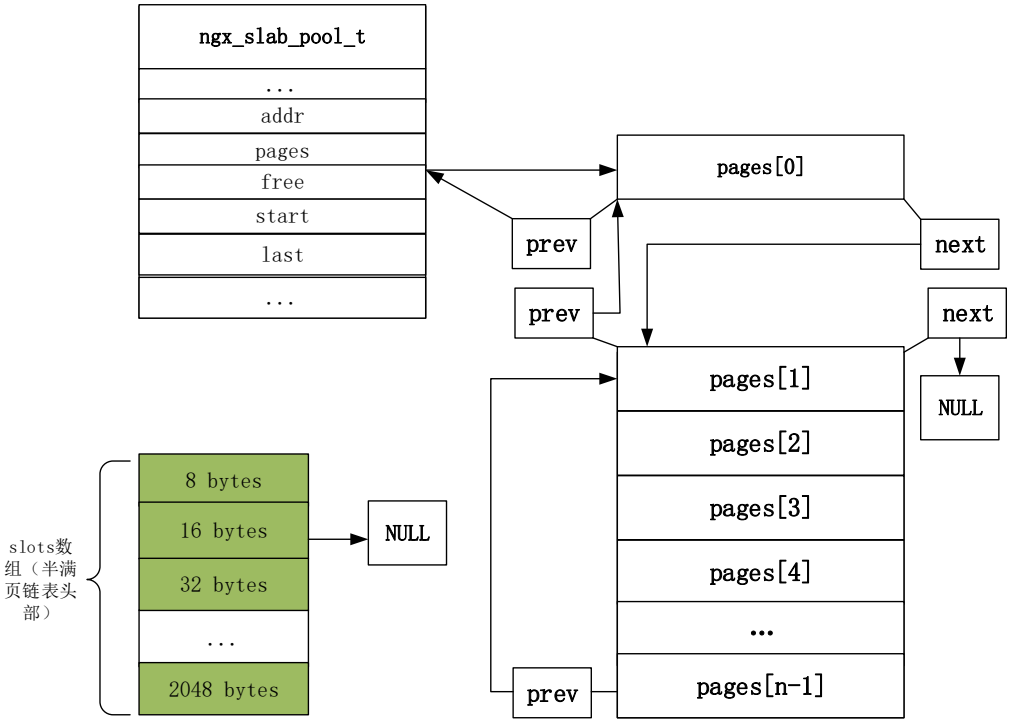

图 5 释放状态图

到这里，slab 共享内存池的使用及实现原理就介绍完了，对于 slab 共享内存池的实现原理，在 Nginx 源码中的实现比这里介绍的更为复杂，这里只是对一种一种情况进行的简要分析，如果要更好的理解实现原理，还是需要参考相关资料并从源码入手才能真正掌握。