池空间组织：

struct pmemobjpool：

memory pool header (struct pool\_hdr hdr)

persistent part/descriptor（2KB）

struct lane\_layout (需要cacheline对齐 per 3KB）

struct ULOG(LANE\_REDO\_INTERNAL\_SIZE) internal (64B+192B)

struct ULOG(LANE\_REDO\_EXTERNAL\_SIZE) external (64B+640B)

struct ULOG(LANE\_UNDO\_SIZE) undo (64B+2048B)

struct lane\_layout

...

struct lane\_layout ( \* 1024)

struct heap\_layout （地址pagesize对齐，pool后续全部空间）

struct heap\_header header(1KB)

struct zone zone0（除了最后一个zone，都为MAX\_SIZE，由pool所剩空间决定）

struct zone\_header (64B)

struct chunk\_header (8B \* （65535-7） 8的倍数，表示chunk的最大数量）

struct chunk chunk (256KB \* （65535-7） 可变长度）

struct chunk\_run (chunk\_run = ? chunk, 16B hdr + content)

16B hdr

? \*8B bitmap（根据计算得到bitmap占用空间）

256KB-hdr-bitmap

n \* chunk

...

struct chunk chunk

...

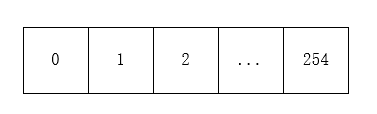
struct zone zoneN

其中chunk run是chunk的实际组织形式，一个chunk run可能包括1或最多10个chunk，由 hdr+bitmap+data组成

如何完成分配：

pop create时创建一个alloc class collection

alloc class数组，最多可以有255个alloc class



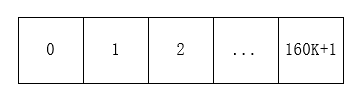
每个alloc class对应一个unit size，即该alloc class的分配单元是多大，对于一个obj对象的分配，最多用8个单元，同时每个alloc class根据一个总分配次数的预计值计算一个总空间，从而维护一个chunk的倍数值，用来表示该alloc class将来使用时对应的chunk run的大小，并且维护一个bitmap的信息（不包括bit空间本身），该bitmap将来在chunk run内保存实际bit空间，每个bit位代表一个unit size

alloc class分为CLASS\_HUGE和CLASS\_RUN两类，初始时ac[0]为CLASS\_HUGE，ac[1]指向unit\_size=128的alloc class

并在128，1024，2048，4096，8192，16384，32768，131072，393216（384K） 8个区间

每个区间（l+64,r]内依次0.05倍的步长增加size大小（并且需要64位对齐），去尝试创建新的alloc class并依次添加到数组内

class\_map\_by\_alloc\_size，以粒度16来取index，每个slot对应一个alloc class数组的下标，初始值为FF



初始化时前128/16=8个slot都存1

每个alloc class对应1个bucket，bucket用来管理alloc class的具体分配，bucket用一个container来保存memory block结构，memory block结构可以理解为用来指代chunk run上的一片空间，分为HUGE和RUN两类，第一类memory block用来表示多个chunk，用于给alloc class分配对应的chunk run，第二类memory block用来表示chunk run上的多个unit，用于给obj分配空间

两类memory block对应的bucket container分别为二叉树和队列

以root分配为例：

root或其他obj走的是相同的流程

首先获取一个bestfit的alloc class：如果class\_map\_by\_alloc\_size里找不到，则反向遍历alloc class数组找到碎片空间最少的alloc class，并将该ac插入到map的对应位置

然后尝试从alloc class对应的bucket里取出一个memory block，表示在pmem上root对应的空间

由于初始状态下bucket都为空，因此首先会在ac[0]对应的bucket container里插入指代zone0全部空间的memory block，然后根据alloc class需要的chunk run大小，从memory block里取出一个对应的memory block（这里的memory block就是第一类memory block，用于给alloc class分配需要的chunk run）；当alloc class有了对应的空间之后，将整个空间同样以memory block的形式组织（每个memory block对应64个unit\_size，和bitmap以u64的形式保存在内存相匹配），并插入到alloc class对应的bucket container里

最后从container里取出一个对应大小的memory block，比如分配3个unit\_size大小，那么将取出队头的memory block进行拆分，剩下的memory block重新插回container