pmemobj\_create(*const* *char* \**path*, *const* *char* \**layout*,

*size\_t* *poolsize*, *mode\_t* *mode*)

可以选择先创建文件，那么传入pool\_size=0即可，创建池时会先检查文件头位置是否有非零数据

也可以传入pool\_size由创池创建文件，LAYOUT NAME用来open时进行比较（按这个意思，多个文件应该是可以对应同一个LAYOUT NAME，反之则不行），但变量的创建和使用并没有依赖LAYOUT NAME，除了open

两种指针类型：TOID/PMEMoid

TOID(t)直接通过PMEMoid强转得到，分配空间得到地址，可以理解为指针

区别：PMEMoid无指针类型，TOID能够得到所指对象的类型

TOID(t)：

t \*\_type成员用来取\_\_typeof\_\_，并不需要考虑其值，整个union实际上都是PMEMoid的内容

使用typedef来做计数:

以pminvaders example的LAYOUT的创建举例

POBJ\_LAYOUT\_BEGIN(pminvaders);

POBJ\_LAYOUT\_ROOT(pminvaders, *struct* *game\_state*);

POBJ\_LAYOUT\_TOID(pminvaders, *struct* *player*);

POBJ\_LAYOUT\_TOID(pminvaders, *struct* *alien*);

POBJ\_LAYOUT\_TOID(pminvaders, *struct* *bullet*);

POBJ\_LAYOUT\_END(pminvaders);

同一个layout里的类型有递增的type\_num

#define POBJ\_LAYOUT\_BEGIN(*name*)\

typedef *uint8\_t* \_pobj\_layout\_##name##\_ref[\_\_COUNTER\_\_ + 1]

首先通过typedef定义一个类型，从而通过sizeof可以得到一个大小

POBJ\_LAYOUT\_ROOT

定义ROOT结构， type\_num为0，同样通过typedef来做计数

#define POBJ\_LAYOUT\_TOID(*name*, *t*)\

TOID\_DECLARE(t, (\_\_COUNTER\_\_ + 1 - \_POBJ\_LAYOUT\_REF(name)));

每一个新定义的结构都将分配一个递增的新type\_num

#define POBJ\_LAYOUT\_END(*name*)\

typedef *char* \_pobj\_layout\_##name##\_cnt[\_\_COUNTER\_\_ + 1 -\

\_POBJ\_LAYOUT\_REF(name)];

是可以不要root object的：array样例（但通常是有的）

root object和其他object是两种不同的object ：区别在哪？

root object可以每次重启时直接读取到（应该是由于直接存到了池结构内，即池文件的头部是有一些信息的），从而根据root object来找到其他数据（如在root里存了其他结构体，进而递归的找到所有内容）

root object : https://pmem.io/2015/06/13/accessing-pmem.html

POBJ\_ROOT 如果是第一次调用或者更大size的结构，将会创建root object，按文档所说如果前一次的地址空间足够，那么重新分配更大的空间，如果不够则会到其他地址分配空间；否则直接返回root object （root object在pool里有存）

所有的objects都存在一个internal collections里，不论是在LAYOUT中创建的对象结构还是其他地方手动创建的对象结构，可以通过迭代器操作，但是没有任何顺序

POBJ\_ALLOC/POBJ\_NEW 可以带参数o，那么新分配的object将从o获得地址，也可以传入NULL，那么没有获得返回地址，可以通过迭代器来获得具体object

btree（二叉树）/list（双向链表）/array

**map 典型样例**：

Layout里只创建了root object

type\_num：在layout创建的结构其type num递增，但也可以不在layout内创建结构，那么type\_num有什么作用？

tx内修改变量（包括空间释放）时需要 TX\_ADD\_FIELD/pmemobj\_tx\_add\_range\_direct，https://pmem.io/2015/06/15/transactions.html，意味着需要保证事务性的赋值需要snapshot， 而如一个局部中间变量的创建不需要snapshot

tx既有undo操作又有redo操作？

空间分配是用的redo，比如root object的alloc

btree\_map/skiplist\_map

object空间如何组织？

pop->lanes\_offset(struct lane\_layout)

struct ulog 由64B的metadata和capacity B的data组成

data由多个entry组成，entry由entry base（header）和entry data组成

entry base 64B = 3b标志位（表示类型）+ 61b地址

entry有两种，一种是8字节（以下？）值的and/or/set操作，用ulog\_entry\_val结构，另一种是n个字节的memset/memcpy操作，用ulog\_entry\_buf结构

ulog 初始化：

根据cacheline对齐更改起始地址与data的大小，要保证地址和结构总大小都是cacheline对齐的，同时固定地址范围在原始结构体大小内 只需要将第一个entry的header写为空即可，采用cacheline对齐写，8B写了64B 最后持久化

pop->heap\_offset(stuct heap\_layout)

（palloc\_init）按zone的最大大小分配zone空间，最后一个zone大小处于1个chunk到max个chunk的范围

zone0为第一个zone，由于除最后一个zone外每个zone都是max\_size，第n个zone就通过n\*max\_size取地址

pop->lanes\_desc（struct lane\_descriptor）runtime内存结构

（lane\_boot）初始化时会从persistent mem中读取并缓存layout的信息，以lane\_descriptor结构重新组织

（lane\_recover\_and\_section\_boot）调用ulog的recover,redo internal/external里每个entry的操作。struct ulog有一个next地址，entry如果是val类型，简单的对地址的内容进行and/or/set即可，如果是buf类型，memset操作将地址所在size大小set为data[0]，即data大小应该为1，memcpy操作在地址所在size大小cpy data的数据，即data是size大小。redo完后将internal/external的metadata置为0（capacity也变0了，初始化时分别为192B/640B ）

pop->heap(struct palloc\_heap) runtime内存结构

（pmalloc\_boot -> heap\_boot）填充heap

struct alloc\_class\_collection (多个alloc class的集合结构）

size\_t granularity （粒度，16

struct alloc\_class \*aclasses[255] (最多255个不同类型的alloc函数？）

size\_t last\_run\_max\_size (最后一个分配函数对应的chunk run的最大可分配空间)

uint8\_t \*class\_map\_by\_alloc\_size (MAX\_RUN\_SIZE / granularity + 1个槽，每 个槽是一个字节大小的id，对应aclasses[id]）

struct critnib \*class\_map\_by\_unit\_size

int fail\_on\_missing\_class （0,如果为1，那么当class\_id没有对应的alloc class时 返回NULL）

int autogenerate\_on\_missing\_class （1,如果为1，那么当class\_idx没有对应的 alloc class时，将选择一个浪费空间最少的alloc class，并赋给对应class\_idx的槽）

**alloc\_class\_collection\_new**

last\_run\_max\_size初始为 MAX\_RUN\_SIZE = 10 \* chunk\_size(256KB)

class\_map\_by\_alloc\_size初始为MAX\_RUN\_SIZE / granularity + 1=160K + 1个槽，每个槽的初始值为255

class\_map\_by\_unit\_size 初始化一棵critnib树

新建第1个alloc class：

unit\_size=chunk\_size=256K,type=CLASS\_HUGE,header\_type=HEADER\_COMPACT, flags 带有CHUNK\_FLAG\_FLEX\_BITMAP,id=0, run\_descriptor未初始化，存放在aclasses[0]

新建predefined class：

unit\_size=MIN\_UINT\_SIZE=128，type=CLASS\_RUN，header\_type=HEADER\_COMPACT,flags同上,rdsc.alignment=0,由于flags带有CHUNK\_FLAG\_FLEX\_BITMAP，rdsc.bitmap将是一个Flexible bitmaps：

（memblock\_run\_bitmap）首先content size=RUN\_CONTENT\_SIZE（除去chunk run header外，chunk run所剩空间，即size\_idx \* chunk\_size-chunk run header size，这里是默认的size\_idx=1，即单chunk size的chunk run），然后得到bitmap 的nbits=content\_size/unit\_size=2047，nvalues=nbits/RUN\_BITS\_PER\_VALUE=32，向上取整，其中RUN\_BITS\_PER\_VALUE=64，我的理解是将bit位分组，64个bit位一组，一共nvalues组，然后让nvalues和chunk run header对应的values数(2)加起来能够对齐CACHELINE\_SIZE对应的values数(8)（其实就是让bitmap后面的obj数据的起始地址能对齐cacheline），从而修改nvalues为38，size=nvalues\*8B（bit空间占的大小），nbits再改为（content\_size-size）/unit\_size=2045，去掉了bitmap的bit信息占用的空间，nvalues再减去（nvalues对应的bit位减去nbits后的bit位数换算成的values值，这里为(2432-2045) /64=6）得到最后的nvalues=32，value初始为空

rdsc.nallocs设置为nbits，rdsc.size\_idx=1（不知道有什么用），rdsc.unit\_size=unit\_size=MIN\_UINT\_SIZE,然后找到第一个空的槽，这里是1，因此存放在aclasses[1]，插入kv到class\_map\_by\_unit\_size（把map\_idx/size\_idx/flag组合成key，value为alloc class本身）

将 class\_map\_by\_alloc\_size 前128/16个槽都设置为1，即predefined class（猜测小于128的空间分配都采用128大小的空间分配）

然后开始创建一组多个不同size对应的alloc class（只填充到了alloc class数组里，没有填充到 class\_map\_by\_alloc\_size map里）：

分为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）

以128+64=192的alloc class创建为例：

（alloc\_class\_find\_or\_create）

required\_size\_bytes = 192 \* RUN\_MIN\_NALLOCS=192\*200，200表示每个chunk run预计能分配200次，就是说对于某个chunk run而言，按照可以分配200次192B的空间来预计容量

如果required\_size\_bytes> RUN\_DEFAULT\_SIZE,（即CHUNKSIZE减去CHUNK RUN里header和默认bitmap的size = 256K - 16 - 38 \* 8，这与alloc class里的bitmap的bit空间大小相符，也就是说bit空间是在pmem上，而bitmap的描述信息是在内存？），则required\_size\_bytes -=RUN\_DEFAULT\_SIZE,然后计算向上对齐chunksize后需要的chunksize个数n，required\_size\_idx=n+1，并且控制required\_size\_idx最大RUN\_SIZE\_IDX\_CAP=16（也就是说chunk run最大16个chunksize，size\_idx是用来表示chunk run占用了几个chunksize），后续只会改变192这个值而chunk run的总空间不会再改变

从后向前遍历alloc class集，如果alloc class为空或者是CLASS\_HUGE或者size\_idx小于要求则跳过，否则判断192是否整除alloc class的unit\_size以及整除结果是否小于RUN\_UNIT\_MAX\_ALLOC=8（就是说unit\_size是alloc class的每次分配大小，这次分配的大小192必须在8次内分配完，否则说明这个alloc class的unit\_size并不适合该大小的空间分配，实际上从RUN\_MIN\_NALLOCS就可以看出真正的分配次数是远不止8次的）

如果没有满足条件的alloc class则重新计算需要的分配大小n（这里是192），通过对n依次增加64，每次计算对于n的分配大小、size\_idx的总chunk run大小，所需要的bitmap空间，然后计算chunk run的剩余空间（也就是实际可分配空间）除以n（不断增长的n）得到的余数，也就是浪费的空间是否大于MAX\_RUN\_WASTED\_BYTES=1024，如果不大于那么n就取当前值，

再次遍历alloc class，以新的n值找是否有满足条件的alloc class，

如果还是没有，那么新建alloc class，参照 新建predefined class 的流程，其中unit\_size改为了新的n值，size\_idx为required\_size\_idx。

由于是初始化阶段，因此alloc class的id从2开始

当以上所有alloc class创建结束后，计算alloc class集里最后一个alloc class的理论最大可分配空间，unit\_size \* nallocs(如果小于8则取8）,并更新ac集的last\_run\_max\_size（如果大于MAX\_RUN\_SIZE则更新）

（heap\_boot -> head\_arenas\_init）

初始化arenas，nactive=0，vec初始化1024个arena指针（应该都是null）

nzones以max zone size计算得到heap最多有多少zones

（heap\_boot -> heap\_zone\_update\_if\_needed）

遍历每个zone（max size zone），计算该zone包含的chunk个数size\_idx（除了最后一个zone，都是MAX\_CHUNK=65535-7），将每个zone的第一个chunk header设置为{CHUNK\_TYPE\_FREE,0,size\_idx}（size\_idx表示这个zone只有一个chunk，因为在遍历zone里的chunk时每次是递增每个header里的size\_idx大小的），第size\_idx-1（最后一个）chunk header设置为{CHUNK\_TYPE\_FOOTER,0,size\_idx}，每个zone的header设置为{size\_idx,ZONE\_HEADER\_MAGIC}（由于zone header只有8字节的实际数据，56字节的保留位，因此也可以8字节原子直接刷到pmem？）

（还有一些heap相关的初始化，如recycler，太多了搞不清用来干嘛的）

（pmalloc\_boot -> heap\_buckets\_init）

从前向后遍历alloc class集，对每一个非NULL的alloc class c，如果是CLASS\_RUN类型的，新建对应的recycler（不知道有什么用），新建每一个arena（应该是对应每一个线程）里的bucket[c->id]（非NULL的，bucket也有255个，应该是每个alloc class都会有多个arena里的对应bucket来管理对应线程的空间？），其中每个bucket有一个container（这里是seglists结构体的super成员），一个memory\_block\_reserved结构体（里面包含一个memory\_block结构和nresv，好像类似于引用计数）

defalut\_bucket对应第一个alloc class（传入的container不一样，与rval结构相关）

heap初始化结束后处理undo log？

**从root obj和普通obj的alloc入手**

**分配过程：**

**先从pmem上的heap创建一个易失（内存）的memory bolck保留在内存的heap结构里**

**然后创建修改的redo log，包括chunk metadata和new obj地址**

**最后提交并执行redo log**

root obj：

obj\_alloc\_root

首先会进行redo日志的处理，然后调用 palloc\_operation 进行分配，class\_id/arena\_id都为0，由于class\_id=0,在 palloc\_reservation\_create 函数 内将使用 heap\_get\_best\_class 的alloc class，也就是会挑选一个最合适分配size的alloc class。

(heap\_get\_best\_class->alloc\_class\_by\_alloc\_size):

如果size大于last\_run\_max\_size，那么选择第一个也就是CLASS\_HUGE的alloc class，否则首先通过size和granularity=16转化为map的下标找到class\_id，判断class\_id是否是255（由于初始化阶段只初始化了128/16=8前9个槽，因此假如size>128，那么class\_id为255，此时需要进一步查找alloc class

(alloc\_class\_by\_alloc\_size->alloc\_class\_assign\_by\_size)

通过alloc\_class\_find\_min\_frag函数寻找碎片空间最少的alloc class，从后向前依次遍历alloc class，首先计算需要分配的real\_size=size+header\_size（似乎每一个分配的对象有一个头，根据分配函数确定leagcy或compact的，之前的初始化过程都是设置compact），然后计算对于该alloc class的unit\_size而言需要的分配次数units，如果超过RUN\_UNIT\_MAX\_ALLOC=8次则继续下一个alloc class，如果有每次分配的空间实际浪费大小为0的alloc class，那么不管最后的总空间浪费，返回即可，否则判断每次分配的空间浪费大小，首先计算units\*unit\_size-real\_size得到每次分配的空间实际浪费大小，然后计算总分配次数对units的取余数，也就是有多少次不能分配的次数得到整个chunk run不能用到的空间，平摊到总分配次数上，最后得出每次分配的平均空间浪费大小，直到找到平均空间浪费最小的alloc class。

得到alloc class后将其id添加到class\_map\_by\_alloc\_size对应的槽里。

然后计算一个size\_idx（与之前的不同，这里是表示需要多少个unit\_size才能完成一次分配）

然后取得一个arena里对应class id的bucket，arena\_id由于传入0，因此会选取一个arena（具体方式不清楚heap\_thread\_arena）

然后调用heap\_get\_bestfit\_block获取一个bestfit的memory block，首先在取得的bucket（不是第一个alloc class对应的，因此是seglist类型的）的container里进行获取，由于初始都是空的因此找不到，从而调用对于CLASS\_RUN类型的函数heap\_ensure\_run\_bucket\_filled来获取（或者说创建）。

（heap\_ensure\_run\_bucket\_filled）

没有active memory block，也无法recycler，因此获取default bucket（第一个alloc class对应的bucket，是ravl类型的），然后遍历zones\_exhausted(初始时为0，因此是zone0，同时自增1）里的chunk，实际只有第一个chunk，填充该chunk header到mem block得到一个zone0 chunk0 size\_idx=65535的memory block，然后插入到bucket的container里（这里是ravl类型的）

然后到defb里去找一个size\_idx为对应class的size\_idx（即分配函数对应的多少个chunk的chunk run，暂时用units表示）的memory block，同样调用heap\_get\_bestfit\_block函数，这次bucket是dfb，由于上一步在dfb的container里添加了一个最大范围的memory block，因此得到该memory block，并进行split（简单来说，从F....R -> F...R|F...R，前部分变成units个chunk)，得到分割后大小为size\_idx的memory block（暂时用m代替），然后进行memory block对应的chunk run的填充。首先将该run的run header的block\_size设置为该alloc class的unit\_size，rdsc（也就是alloc class里的run descriptor）里bitmap的values（即bit的空间）指向chunk run的content（即chunk run的内容由bitmap加实际obj数据组成）,并将有效的bit位都置为0，无效的置为1（因为之前有为了对齐浪费的空间），其中每个value（64B）从低位到高位依次代表0-63

然后更改整个chunk run对应的chunk header，比如最开始type是 F......FOOTER ，改为RUN RD RD RD，并且第一个chunk header的size\_idx为size\_idx，而之后的size\_idx依次为1 -- size\_idx-1。更新m的状态为RUN，并将rdsc的bitmap（是内存结构）指针再保存在m里面

然后对每一个有效的bit位，每64B（即一个value）如果全为0b，则对第一个0生成一个memory block，否则对该value的所有0都生成memory block run\_process\_bitmap\_value），将这些memory block插入到bucket的container里。这里的memory block的size\_idx是指代在一个value里的位置（64-0），block\_off指代在整个bit空间的位置（0-nbits）。也就是说假设每个value都为0，那么最后每一个value（64B）都有一个memory block，block\_off(0/64/...)将会全部插入到container（这里是seglists类型）的blocks[63]队列里，并将nonempty\_lists的最高位置为1，同时container的m将存该m，用来保存chunk\_id/zone\_id。即每个memory block都表示能分配64个unit\_size。

将bucket的active\_memory\_block设置到m（即表明了chunk run的地址和大小），并设置is\_active=1表示该bucket能用了？

RUN bucket完成filled以后再次从bucket里取memory block，这次找的size\_idx对应memory block的size\_idx（即此次分配需要多少个unit\_size，假设为3），假设取出一个（size\_idx=64，block\_off=0的m）那么将进行split，得到一个（size\_idx=61，block\_off=3的r），将r重新插回bucket的container，m修改size\_idx=3，从而得到m

然后调用alloc\_prep\_block，将用0填充m对应的空间，并填充m对应空间的header（每个memory block对应pmem上的空间是由头+数据组成的）

此时一个pobj\_action\_internal \*out构建完成，其m=最后得到的m，offset=m的数据部分对应heap的偏移地址，usable\_size=m的数据部分大小

最后执行action，完成bitmap的置位操作（通过redo log保证事务性？），并更新heap的stats计数

总结：HUGE的bucket用来给分配函数组织空间，一个分配函数对应一个由多个CHUNK组成的chunk run，而RUN的bucket来给实际obj组织空间，在chunk run内部是通过分配函数的unit\_size容纳每个obj的

普通obj：

obj\_alloc\_construct

分配过程和root完全一致，只是root会把offset和size保存到pool

问题：每次重启是重新恢复alloc class集吗/很多东西没有持久化，如bucket

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 chunks[] (256KB \* n 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