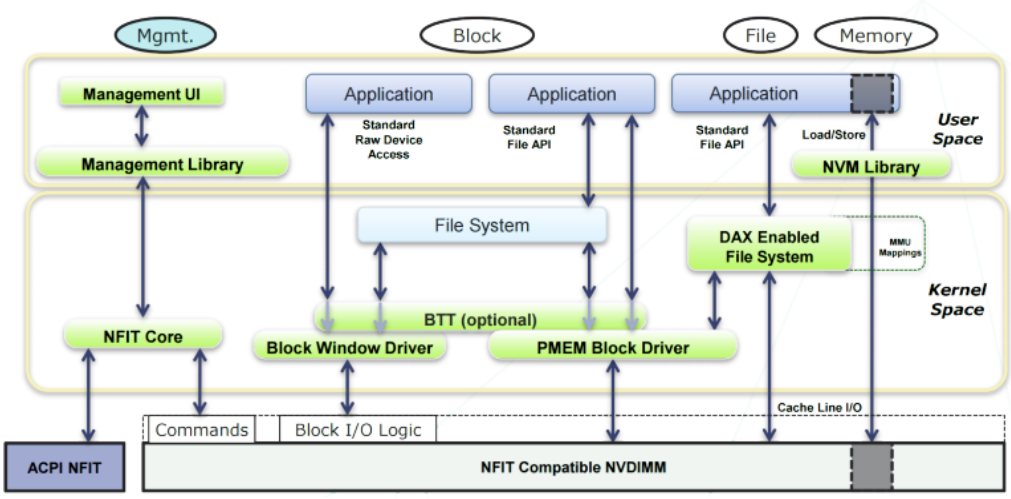
PMDK(NVML)事务实现机制源码分析

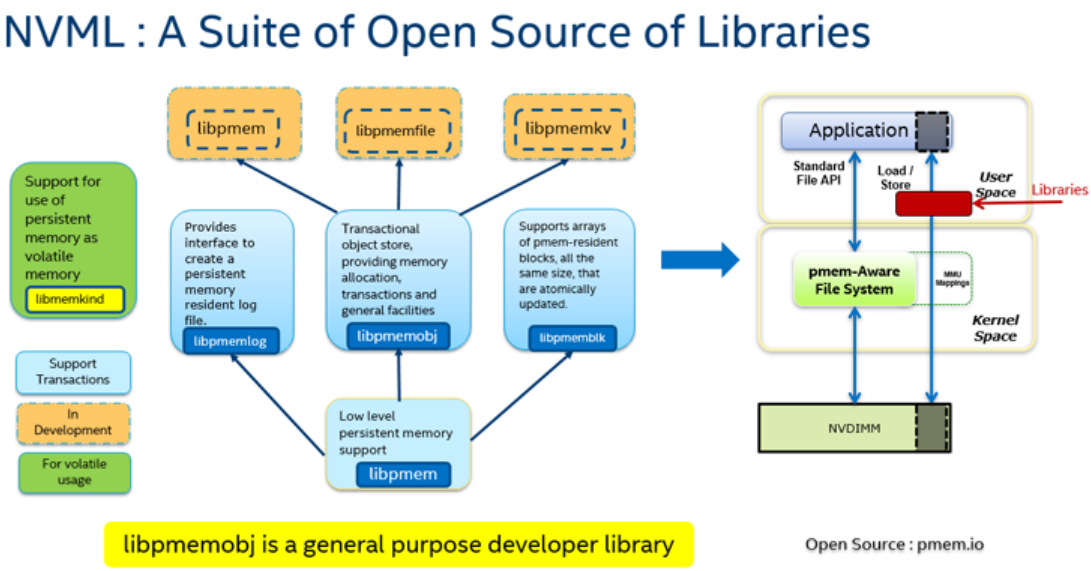
原创　黑客画家　分布式数据库　2018/03/05 10:51　阅读数 1.3W

一、PMDK介绍

pmdk，全称Persistent Memory Development Kit，它是一套用于使用具有DAX（Direct Access）访问特性存储的开发工具库。如下如所示，NVM（NON-VOLATILE MEMORY）存储可以使用使能DAX功能的文件系统直接暴露在用户空间，用户态程序可以使用标准的文件系统API来操作NVM，同样也可以使用mmap将其直接映射到用户空间，无论使用哪种方式，对NVM的操作都会直接转换为对NVM的load和store，中间是没有page cache的（这就是支持DAX模式的文件系统和普通文件系统之间的一点区别）。在使用文件系统时，数据的完整性一般都由文件系统来保证，而NVM作为一种非易失性存储，在使用mmap方式来读写时，如何保证数据的完整性和一致性就显得尤为重要。通常可以有很多种方式可以做到这一点（也就是后文即将讨论的），比如靠上层应用程序自己的策略来保证，也可以使用第三方库来做，而pmdk（更具体点来说是pmdk中的 libpmemobj）就是用来完成这项工作的。



下图是pmdk的各个组件之间的依赖关系，其中libpmem提供底层的内存持久化、刷新接口，基于它，pmdk上层还提供了很多其他组件，比如本文重点介绍的libpmemobj就提供了存储的事务特性，其他组件的介绍将在以后的文章中逐步体现。



二、基本概念介绍

1、内存池(Memory pools)

如前文所述，NVM作为一种快速、可字节寻址、持久型的存储，在被以DAX暴露在用户态以后，可以在其上创建很多被mmap的文件，这些被mmap的文件就称之为内存池（Memory pools），当然，有了pmdk之后，内存池的创建就不需要我们自己手动mmap了，可以使用pmemobj_create来完成，接口定义：

```
#define pmemobj_create pmemobj_createW
PMEMobjpool *pmemobj_createW(const wchar_t *path, const wchar_t *layout,
        size_t poolsize, mode_t mode);
```

其中path是要创建的文件的路径（也就是NVM被挂在的文件系统路径），layout可以理解为布局，它作为该pool的唯一标志，poolsize是要创建的pool的大小，mode为文件的读写权限。

如果要打开一个已经存在的pool文件，那么可以使用pmemobj_open，接口定义：

```
#define pmemobj_open pmemobj_openU
PMEMobjpool *pmemobj_openU(const char *path, const char *layout);
```

pmdk同样提供了pmemobj_check接口用来检测pool的完整性，接口定义如下：

```
#define pmemobj_check pmemobj_checkU
int pmemobj_checkU(const char *path, const char *layout);
```

pmemobj_create和pmemobj_open都会返回一个PMEMobjpool指针，它就是创建或打开的pool的句柄（定义在pmdk/src/libpmemobj/obj.h），下面列出其主要成员：

```
struct pmemobjpool {
        struct pool_hdr hdr;    /* memory pool header */

        /* 以下成员会保存在NVM中 (2kB) 进行持久化*/
        char layout[PMEMOBJ_MAX_LAYOUT];//#define PMEMOBJ_MAX_LAYOUT ((size_t)1024)
        uint64_t lanes_offset;
        uint64_t nlanes;
        uint64_t heap_offset;
        uint64_t unused3;

    /* 内存池描述符中持久化的部分占用的大小 (2kB) */
    // #define OBJ_DSC_P_SIZE           2048
    /* 内存池持久化部分未使用的部分大小 */
    // #define OBJ_DSC_P_UNUSED (OBJ_DSC_P_SIZE - PMEMOBJ_MAX_LAYOUT - 40)
        unsigned char unused[OBJ_DSC_P_UNUSED]; /* must be zero */
        uint64_t checksum;      /* 对以上成员的校验和 */

        uint64_t root_offset; // 根对象在pool中的偏移，一个pool中只能最多有一个根对象

        /* unique runID for this program run - persistent but not checksummed */
        uint64_t run_id;

        uint64_t root_size;// 根对象的大小

        /*
         * These flags can be set from a conversion tool and are set only for
         * the first recovery of the pool.
         */
        uint64_t conversion_flags;

        uint64_t heap_size;

        struct stats_persistent stats_persistent;

        char pmem_reserved[496]; /* must be zeroed */

        /* 以下成员为运行时状态，不会持久化到NVM中 */
        void *addr;             /* mapped后的地址 */
        int is_pmem;            /* 如果存储介质是PMEM就为true */
        int rdonly;             /* 如果内存池以只读方式打开就为true */
        struct palloc_heap heap;
        struct lane_descriptor lanes_desc;
        uint64_t uuid_lo;
        int is_dev_dax;         /* 如果device dax模式就为true */

        struct ctl *ctl;
        struct stats *stats;
        struct ringbuf *tx_postcommit_tasks;

        struct pool_set *set;           /* 内存池集合 */
```

```
        struct pmemobjpool *replica;    /* next replica */
        struct redo_ctx *redo;// redo log，事务相关

        /* 每个副本都需要的共用函数: pmem 或 non-pmem */
        persist_local_fn persist_local; /* 内存持久化函数 */
        flush_local_fn flush_local;     /* 刷新缓冲区到内存 */
        drain_local_fn drain_local;     /* 排空缓冲区 */
        memcpy_local_fn memcpy_persist_local; /* 持久型的memcpy函数 */
        memset_local_fn memset_persist_local; /* 持久型的memset函数 */

        /* 主副本: with or without data replication */
        struct pmem_ops p_ops;

        PMEMmutex rootlock;     /* root object lock */
        int is_master_replica;
        int has_remote_replicas;

        /* 远程副本,用于RDMA，不是本文讨论的重点 */
        void *rpp;      /* RPMEMpool opaque handle if it is a remote replica */
        uintptr_t remote_base;  /* beginning of the pool's descriptor */
        char *node_addr;        /* address of a remote node */
        char *pool_desc;        /* descriptor of a poolset */

        persist_remote_fn persist_remote; /* 远端持久化函数，用于RDMA */

        struct tx_parameters *tx_params;
};
```

下面来重点讨论下pmemobjpool中提到的几个持久化函数，由于pmdk运行在模拟NVM的环境上运行，因此对于是否是pmem分别会有两种不同的缓存刷新策略，如下：

```
    if (rep->is_pmem) {//如果是持久型内存
                rep->persist_local = pmem_persist;
                rep->flush_local = pmem_flush;
                rep->drain_local = pmem_drain;
                rep->memcpy_persist_local = pmem_memcpy_persist;
                rep->memset_persist_local = pmem_memset_persist;
        } else {// 否则就是普通内存，易失性内存
                rep->persist_local = (persist_local_fn)pmem_msync;
                rep->flush_local = (flush_local_fn)pmem_msync;
                rep->drain_local = obj_drain_empty;
                rep->memcpy_persist_local = obj_nopmem_memcpy_persist;
                rep->memset_persist_local = obj_nopmem_memset_persist;
        }
```

可以看到，针对非pmem时，persist_local被指向了pmem_msync，最终就是调用msync，这是个通用的函数（使用mmap映射的内存区域都可以只用msync进行刷新），但是这在NVM中会损失一些性能。针对NVM的特性，当pmdk处于NVM编程环境中时，persist_local就指向pmem_persist了，它的实现如下：

```
/*
 * pmem_persist -- make any cached changes to a range of pmem persistent
 */
void
pmem_persist(const void *addr, size_t len)
{
        LOG(15, "addr %p len %zu", addr, len);
   // 将缓冲区中的内容刷新到NVM中
        pmem_flush(addr, len);
   // 等待排空缓冲区，本质是一个存储内存屏障
        pmem_drain();
}
```

进一步看看pmem_flush的实现：

```
/*
 * pmem_flush() calls through Func_flush to do the work.  Although
 * initialized to flush_clflush(), once the existence of the clflushopt
 * feature is confirmed by pmem_init() at library initialization time,
 * Func_flush is set to flush_clflushopt().  That's the most common case
 * on modern hardware that supports persistent memory.
 */
static void (*Func_flush)(const void *, size_t) = flush_clflush;

/*
 * pmem_flush -- flush processor cache for the given range
 */
void
pmem_flush(const void *addr, size_t len)
{
        LOG(15, "addr %p len %zu", addr, len);
        Func_flush(addr, len);
}
```

Func_flush是一个函数指针，它会根据当前cpu架构的型号、环境变量的设置等因素指向不同的处理函数，下午在讨论，下面再看pmem_drain函数：

```
/*
 * pmem_drain() calls through Func_predrain_fence to do the fence.  Although
 * initialized to predrain_fence_empty(), once the existence of the CLWB or
 * CLFLUSHOPT feature is confirmed by pmem_init() at library initialization
 * time, Func_predrain_fence is set to predrain_fence_sfence().  That's the
 * most common case on modern hardware that supports persistent memory.
 */
static void (*Func_predrain_fence)(void) = predrain_fence_empty;

/*
 * pmem_drain -- wait for any PM stores to drain from HW buffers
 */
```

```
void
pmem_drain(void)
{
        Func_predrain_fence();
}
```

Func_predrain_fence同样也是一个函数指针，下面就看看Func_flush和Func_predrain_fence是怎么设置的，在pmem_get_cpuinfo函数中（该函数在pmem_init中被调用）：

```
/*
 * pmem_get_cpuinfo -- configure libpmem based on CPUID
 * 根据cpu架构来配置libpmem
 */
static void
pmem_get_cpuinfo(void)
{
        LOG(3, NULL);
    // 当前cpu是否支持clflush指令
        if (is_cpu_clflush_present()) {
                Func_is_pmem = is_pmem_detect;
                LOG(3, "clflush supported");
        }
    // 当前cpu是否支持clflushopt指令
        if (is_cpu_clflushopt_present()) {
                LOG(3, "clflushopt supported");
        // 还可以手动设置环境变量来改变缓存刷新方式
                char *e = os_getenv("PMEM_NO_CLFLUSHOPT");
                if (e && strcmp(e, "1") == 0)
                        LOG(3, "PMEM_NO_CLFLUSHOPT forced no clflushopt");
                else {
                        Func_flush = flush_clflushopt;
                        Func_predrain_fence = predrain_fence_sfence;
                }
        }
    // 当前cpu是否支持clwb指令
        if (is_cpu_clwb_present()) {
                LOG(3, "clwb supported");
        // 还可以手动设置环境变量来改变缓存刷新方式
                char *e = os_getenv("PMEM_NO_CLWB");
                if (e && strcmp(e, "1") == 0)
                        LOG(3, "PMEM_NO_CLWB forced no clwb");
                else {
                        Func_flush = flush_clwb;
                        Func_predrain_fence = predrain_fence_sfence;
                }
        }
}
```

针对 Func_flush，分别可能指向flush_clflushopt和flush_clwb，其定义为：

```
/*
 * flush_clflushopt -- (internal) flush the CPU cache, using clflushopt
 */
static void
flush_clflushopt(const void *addr, size_t len)
{
        LOG(15, "addr %p len %zu", addr, len);

        uintptr_t uptr;

        /*
         * Loop through cache-line-size (typically 64B) aligned chunks
         * covering the given range. 每次刷新一个缓存行
         */
        for (uptr = (uintptr_t)addr & ~(FLUSH_ALIGN - 1);
                uptr < (uintptr_t)addr + len; uptr += FLUSH_ALIGN) {
                _mm_clflushopt((char *)uptr);
        }
}
```

同样flush_clwb函数定义为：

```
/*
 * flush_clwb -- (internal) flush the CPU cache, using clwb
 */
static void
flush_clwb(const void *addr, size_t len)
{
        LOG(15, "addr %p len %zu", addr, len);

        uintptr_t uptr;

        /*
         * Loop through cache-line-size (typically 64B) aligned chunks
         * covering the given range.
         */
        for (uptr = (uintptr_t)addr & ~(FLUSH_ALIGN - 1);
                uptr < (uintptr_t)addr + len; uptr += FLUSH_ALIGN) {
                _mm_clwb((char *)uptr);
        }
}
```

其中_mm_clflushopt和_mm_clwb定义为两个宏：

```
/*
 * The x86 memory instructions are new enough that the compiler
 * intrinsic functions are not always available.  The intrinsic
 * functions are defined here in terms of asm statements for now.
 * 内联汇编指令
 */
```

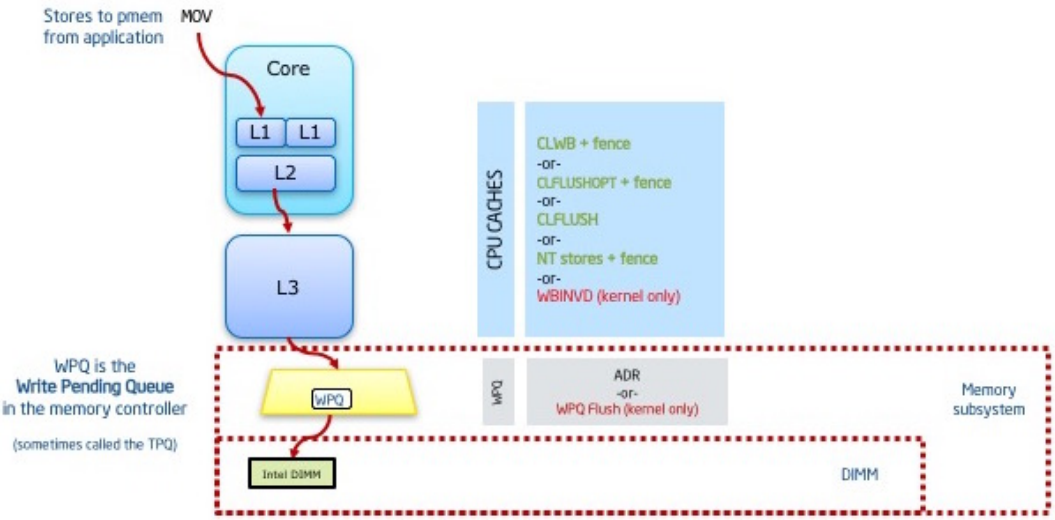
```
#define _mm_clflushopt(addr)\
        asm volatile(".byte 0x66; clflush %0" : "+m" (*(volatile char *)addr));
#define _mm_clwb(addr)\
        asm volatile(".byte 0x66; xsaveopt %0" : "+m" (*(volatile char *)addr));
```

Func_predrain_fence函数指针在两种情况下都是指向predrain_fence_sfence，定义如下：

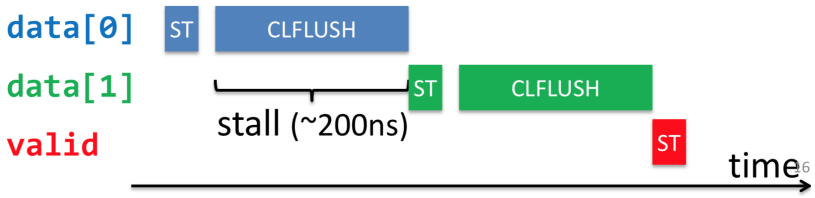
```
/*
 * predrain_fence_sfence -- (internal) issue the pre-drain fence instruction
 */
static void
predrain_fence_sfence(void)
{
        LOG(15, NULL);
        _mm_sfence();   /* 就是一个内存屏障，为了保证CLWB指令或CLFLUSHOPT指令完成*/
}
```

clflushopt和clwb指令是Intel为了支持NVM特地加入的两条优化指令，他们都是用来将CPU多级缓存刷新到NVM中，下面先看看应用程序在向NVM中刷新一条数据时的过程。

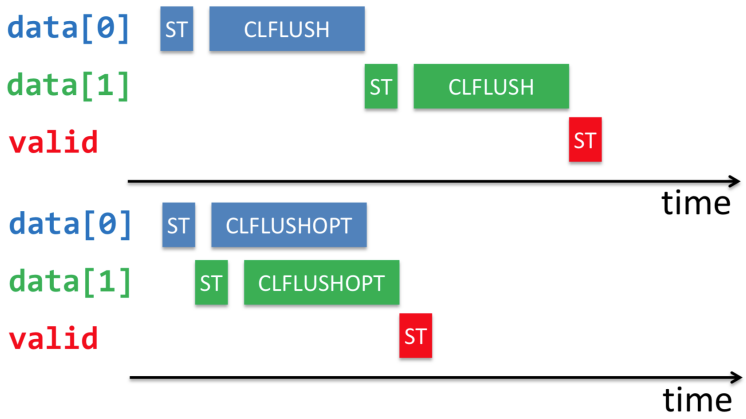
首先，数据开始的时候被存储在cpu的多级缓存中，在执行CLFLUSH/CLFLUSHOPT/CLWB缓存刷新指令的时候，缓存中的数据会被刷新到内存控制器的写队列里面WPQ（也就是没有最终写到介质上），因此，理论上如果此时系统掉电，那么将会出现数据丢失的现象。但是在ADR（异步内存刷新）的保证下，即使掉电，写队列里面的数据也会在超级电容的作用下（电容里面存有足够的电量）安全的写到介质上。



最开始，intel只支持CLFLUSH缓存指令，CLFLUSH的特点是顺序化、串行化的刷新缓存，其缺点是会导致cpu的流水线出现较大的stall时间，导致性能较差：



后来针对NVM加入了CLFLUSHOPT优化指令，他们之间的区别如下图所示：



可见，CLFLUSHOPT就是相当于无序版本的CLFLUSH，新能自然会高很多，如下：

Instruction	Meaning
CLFLUSH addr	Cache Line Flush: Available for a long time
CLFLUSHOPT addr	Optimized Cache Line Flush: New to allow concurrency
CLWB addr	Cache Line Write Back: Leave value in cache for performance of next access

由于CLFLUSHOPT不在保证顺序顺序，因此对于上面的代码，需要在为valid置1之前加一个内存屏障，保证之前的CLFLUSHOPT操作全部完成（前文所说的pmem_drain就是完成这个功能）。

CLWB和CLFLUSHOPT完成的功能类似，唯一不同的是，CLWB在把缓存中的数据刷新之后，并不会失效它， 因此后续的读还是可以读到缓存中的数据，因此性能会好一些。下面总结一下他们之间的区别：

和操作系统中的其它资源一样，pool作为一种资源，在使用完成之后也需要进行关闭操作，该操作使用pmemobj_close函数完成：

```
/*
 * pmemobj_close -- close a transactional memory pool
```

```
 */
void
pmemobj_close(PMEMobjpool *pop)
```

2、持久指针(Persistent pointers)

当文件被映射之后，我们如何去访问它呢？对于传统的指针而言，它只是一个在虚拟地址空间的相对（0地址）偏移量。但是由于这里的pool可能会有很多个，因此持久化的指针没办法只使用一个简单的偏移量来表示，在pmdk中，持久化指针占用两个64 bit，定义如下：

```
typedef struct pmemoid {
        uint64_t pool_uuid_lo;// 每一个pool都有一个uuid唯一标识自己
        uint64_t off;// 对象在该pool中的偏移量
} PMEMoid;
```

因此，一旦获得了这个持久指针，就可以非常简单的将其转为我们可以使用的直接指针（direct pointer）：

(void *)((uint64_t)pool + oid.off)，其实这就是pmemobj_direct函数的原理，其接口定义如下：

```
/*
 * pmemobj_direct -- returns the direct pointer of an object
 */
void *
pmemobj_direct(PMEMoid oid)
{
        return pmemobj_direct_inline(oid);
}

/*
 * Returns the direct pointer of an object.
 */
static inline void *
pmemobj_direct_inline(PMEMoid oid)
{
        if (oid.off == 0 || oid.pool_uuid_lo == 0)
                return NULL;
    // _pobj_cached_pool是线程局部存储的obj缓存
        struct _pobj_pcache *cache = &_pobj_cached_pool;

    // _pobj_cache_invalidate会在每次pmemobj_close被调用的时候加1
    // 也就说，如果在本次转换之前，有pool被close了，那么就让缓存失效，需要重新查找
    // 否则，即使缓存有效，但是uuid和缓存的没有匹配，那么也要重新查找
        if (_pobj_cache_invalidate != cache->invalidate ||
                        cache->uuid_lo != oid.pool_uuid_lo) {
       // 更新缓存技术
                cache->invalidate = _pobj_cache_invalidate;
```

```
        // 根据oid的uuid去cuckoo hash table中查找
                if (!(cache->pop = pmemobj_pool_by_oid(oid))) {
                        cache->uuid_lo = 0;
                        return NULL;
                }
        // 更新到cache中
                cache->uuid_lo = oid.pool_uuid_lo;
         }
    // 计算直接指针
         return (void *)((uintptr_t)cache->pop + oid.off);
}
```

3、根对象(The root object)

现在先假设一个场景：

> 1、分配一个持久型的内存
>
> 2、向里面写一个字符串
>
> 3、应用程序退出

那么现在问题来了，当我程序重新打开的时候，我怎么找到我刚才写的那个字符串在哪里？首先， 它肯定在我刚才写的那个pool里，所以你可以尝试遍历那个pool进行匹配查找，或者，你可以在写入的时候随机选择一个地址来写，然后将这个地址记下来以备后用。但是我们都知道，在传统的编程模型中，你是不能随便操作一个地址的（因为这可能是一个非法地址），同样，在该场景中，随机的写一个地址可能会导致不确定的写覆盖情况，因此pmdk提供的方式就是，使用一个根对象（The root object）来解决（类比于我们在常规编程中，需要先定义一个变量，然后使用这个变量来找到它占用的那块内存）。你可以将你自己定义的数据结构依附在这个根对象上，要分配一个根对象可以直接使用pmemobj_root接口，类似如：

```
PMEMoid root = pmemobj_root(pop, sizeof (struct my_root));
```

其中，pop是我们使用前文所述方式创建的PMEMobjpool指针，my_root是我们自己定义的数据结构，返回值就是持久指针。其实这个可以类比c++中的placement new操作，在指定位置分配一个对象。需要注意的是，pmemobj_root分配的对象内存已经自动被初始化为0，如果你想重新调整对象的大小，只需要改变size参数就好了，如果使用新的size无法完成就地分配，那么一个新的对象将被创建，因此它在pool中的偏移也将被改变，所以，千万不要在使用过程中把持久指针保存在任何地方，因为它是可变的。

三、如何安全的存储数据

经过上文的讨论，我们已经成功的拿到我们定义的数据结构的根对象地址， 那么接下就是向里面存储数据了，比如：

```
void set_name(const char *my_name) {
   memcpy(root->name, my_name, strlen(my_name));
}
```

为了便于讨论，我们假设根对象my_root的定义如下：

```
struct my_root {
        char name[MAX_BUF_LEN];
};
```

如果是在易失性存储上，上面的代码不会有什么问题。但是如果在持久型存储上，上面的代码结果是不确定的。因为程序会crash，机器也可能掉电关机，这些导致程序非正常退出的因素都可能导致上述的数据写入出现不一致的状态。还是以上面的代码为例，假设程序在数据拷贝之前（memcpy执行之前）crash掉了，我们可以想到的方案是，因为根对象默认都是被清0的，因此程序重启之后只要我们识别到根对象内存依然为0就不会有问题。但是假设程序数据拷贝到一半的时候crash掉了有改怎么办呢？假设my_name为“Brianna”，实际存储了的可能为“Brian”，这是有效的但是结果也并不是我们想要的。最后一种情况，假设memcpy完成了程序crash掉就没问题了吗？答案是否定的，由于cpu chache刷新的顺序的不确定性，可能会导致“anna”先被刷新，而位于另一个cache line上的“Bri”却还没来得及被刷新。

那么我们又该如何做呢？继续看下文分解。

1、事务实现方式--应用程序自己保证

现在我们调整一下我们定义的数据结构，加入一个长度字段，如下：

```
struct my_root {
        size_t length;
        char name[MAX_BUF_LEN];
};
```

然后将存储函数实现为：

```
void set_name(const char *my_name) {
   root->length = strlen(my_name);
```

```
    pmemobj_persist(&root->length, sizeof (root->length));
    pmemobj_memcpy_persist(root->name, my_name, root->length);
}
```

从上面的代码可以看出，我们在根对象中新增了长度字段，这样我们在拷贝真正的数据之前，会先将长度字段持久化到NVM中，这是使用pmemobj_persist接口完成的，真正的后续数据拷贝是使用pmemobj_memcpy_persist完成的。在pmdk中，所有以_persist结尾的接口都会确保它们所操作的内存都会从cpu cache中刷新到NVM中。所以，在上面的set_name函数中，一旦程序执行到第5行，我们就可以完全确认root->length里面已经存储了长度信息（并且已经完整的持久化了），因此，在我们对内存进行读之前，可以用长度字段对后面的数据进行完整性校验。其实，pmdk为我们提供了更加方便的事务相关的方法来达到这种效果，具体的可以在后文中看到。

需要注意的是，在当前的硬件架构中，只有8字节的内存可以被以原子方式的写（意思是要么8字节全部写成功，要么写失败）。该结论来自于IA64官方手册，部分截图如下：

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

所以，下面的代码是正确的：

```
root->u64var = 123;
pmemobj_persist(&root->u64var, 8);
```

但是下面这种就是错误的（有可能在出现异常的时候只写了前8个字节）：

```
root->u64var = 123;
root->u32var = 321;
pmemobj_persist(&root->u64var, 12);
```

最后列出完整的例子，首先是write端：

```
#include <stdio.h>
#include <string.h>
#include <libpmemobj.h>

#define LAYOUT_NAME "intro_1"
#define MAX_BUF_LEN 10
```

```
struct my_root {
        size_t len;
        char buf[MAX_BUF_LEN];
};

int
main(int argc, char *argv[])
{
        if (argc != 2) {
                printf("usage: %s file-name\n", argv[0]);
                return 1;
        }
    // 创建内存池
        PMEMobjpool *pop = pmemobj_create(argv[1], LAYOUT_NAME,
                                PMEMOBJ_MIN_POOL, 0666);

        if (pop == NULL) {
                perror("pmemobj_create");
                return 1;
        }
    // 创建根对象
        PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
    // 获取根对象的直接指针
        struct my_root *rootp = pmemobj_direct(root);

        char buf[MAX_BUF_LEN] = {0};
        if (scanf("%9s", buf) == EOF) {
                fprintf(stderr, "EOF\n");
                return 1;
        }

        rootp->len = strlen(buf);
    // 持久化长度
        pmemobj_persist(pop, &rootp->len, sizeof(rootp->len));
    // 持久化内容
        pmemobj_memcpy_persist(pop, rootp->buf, buf, rootp->len);
    // 关闭pool
        pmemobj_close(pop);
        return 0;
}
```

read端（read端比较简单，就不加注释了）：

```
#include <stdio.h>
#include <string.h>
#include <libpmemobj.h>

#define LAYOUT_NAME "intro_1"
#define MAX_BUF_LEN 10

struct my_root {
        size_t len;
```

```
        char buf[MAX_BUF_LEN];
};

int main(int argc, char *argv[])
{
        if (argc != 2) {
                printf("usage: %s file-name\n", argv[0]);
                return 1;
        }

        PMEMobjpool *pop = pmemobj_open(argv[1], LAYOUT_NAME);
        if (pop == NULL) {
                perror("pmemobj_open");
                return 1;
        }

        PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
        struct my_root *rootp = pmemobj_direct(root);

        if (rootp->len == strlen(rootp->buf))
                printf("%s\n", rootp->buf);

        pmemobj_close(pop);

        return 0;
}
```

2、事务实现方式--PMDK事务接口

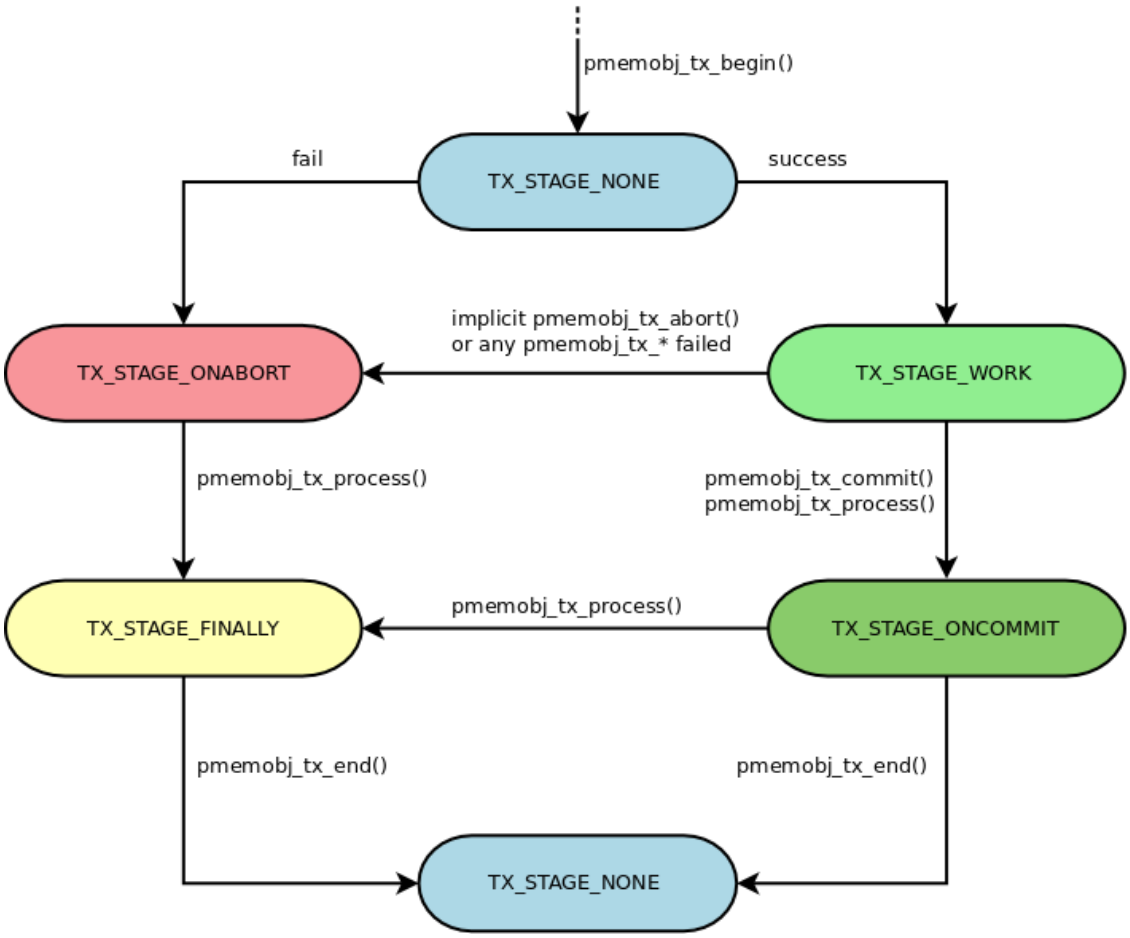
简介

终于快要到我们的重点了。前文中，我们用自己的方式（比如数据前加一个长度字段）实现了写的事务性，但是pmdk为我们提供了更为简单的实现方式，这就是事务（transactions）。在pmdk中，事务操作都是使用以pmemobj_tx_前缀开始的接口完成的，一个事务是分阶段执行的，各阶段具体定义在pobj_tx_stage枚举中：

```
/*
 * Transactions
 *
 * Stages are changed only by the pmemobj_tx_* functions, each transition
 * to the TX_STAGE_ONABORT is followed by a longjmp to the jmp_buf provided in
 * the pmemobj_tx_begin function.
 */
enum pobj_tx_stage {
        TX_STAGE_NONE,          /* no transaction in this thread */
        TX_STAGE_WORK,          /* transaction in progress */
        TX_STAGE_ONCOMMIT,/* successfully committed */
        TX_STAGE_ONABORT,       /* tx_begin failed or transaction aborted */
        TX_STAGE_FINALLY,       /* always called */
```

```
        MAX_TX_STAGE
};
```

上述各个事务阶段按照一定的逻辑进行组合、运转，最后性能一个完成的事务生命周期，如下图所示：



上图清晰的展示了事务接口的使用方法和所处的阶段，其中，pmemobj_tx_process接口可以在你不知道事务当前所处阶段的情况下驱动事务继续执行到下一个阶段。虽然上述接口的使用已经很简单了，但是为了进一步简化事务的使用，pmdk为我们进一步的封装了一下，联想于java框架spring中的事务，也是除了提供API接口之外，又单独使用注解提供了一套声明式事务编程模型，而在pmdk中，它使用宏定义做的，比如上述的事务声明周期可以这样来表示：

```
/* TX_STAGE_NONE */

TX_BEGIN(pop) {
        /* TX_STAGE_WORK */
} TX_ONCOMMIT {
        /* TX_STAGE_ONCOMMIT */
} TX_ONABORT {
        /* TX_STAGE_ONABORT */
} TX_FINALLY {
```

```
        /* TX_STAGE_FINALLY */
} TX_END

/* TX_STAGE_NONE */
```

是不是看上去非常的简介，上面的代码中，除了TX_BEGIN和TX_END是必须的之外，其他的阶段都是可选的。不仅如此，这种写法也是支持嵌套事务的（理论上嵌套事务层数不受限制），一旦嵌套的事务执行失败，那么整个事务就执行失败。

下面举个例子：

```
void do_work() {
        struct my_task *task = malloc(sizeof *task);
        if (task == NULL) return;

        TX_BEGIN(pop) {
                /* important work */
                pmemobj_tx_abort(-1);
        } TX_END

        free(task);
}

...
TX_BEGIN(pop)
        do_work();
TX_END
```

上面的代码片段中，一个外层事务（根事务）包含了一个嵌套事务（子事务），这段代码看上去是很简单、没什么问题的。但是，这段代码是存在内存泄露的，原因是，pmdk的事务是基于setjmp和longjmp来做的（在事务开始的时候setjmp，在事务abort的时候longjmp），因此你没办法保证在嵌套事务中位于TX_END后面的代码一定会被执行。比如上述的代码中，free操作就不会被执行，因为一旦执行到pmemobj_tx_abort，事务就会longjmp到嵌套事务的开始处。因此，正确的写法应该如下：

```
void do_work() {
    // 注意这里的volatile
        volatile struct my_task *task = NULL;

        TX_BEGIN(pop) {
                task = malloc(sizeof *task);
                if (task == NULL) pmemobj_tx_abort(ENOMEM);

                /* important work */
                pmemobj_tx_abort(-1);
        } TX_FINALLY {
                free(task);
```

```
        } TX_END
}
```

熟悉java异常处理的都应该对上面的代码很熟悉，它就相当于try{}cache{}finally{}，同时，还要注意上面的volatile关键字，因为一个局部变量，在setjmp之后值很可能会被改变的（比如在TX_STAGE_WORK阶段），如果不加volatile修饰，那么在longjmp之后这个变量的值很可能不被观察到，因此，凡是会被用在TX_STAGE_ONABORT/TX_STAGE_FINALLY阶段的局部变量，建议都要加上volatile修饰。

• 事务操作

pmdk将事务分为三种基本操作：分配（allocation），释放（free），设置（set）,本文就先介绍一下set的事务操作，它就是用来安全可靠的将一段内存设置为某一个值。先大概介绍一下其原理：pmdk会先将要操作的内存的一个快照（snapshot）保存在undo log中，之后应用程序可以随意修改原来的那块内存，过程中一旦出现任何失败，所有的操作就被回滚。

内存的设置操作一共由两个接口实现：pmemobj_tx_add_range 和 pmemobj_tx_add_range_direct，根据上面的原理介绍，这两个函数中的任何一个被调用的时候，一个新的对象将会被创建，然后会将原内存块的内容拷贝过来，除非用于事务回滚，否则原来的对象将被丢弃。并且需要注意的是，pmdk会假设你要分配的内存都是用来写的，因此在事务提交时它默认会自动被持久化，所以你不需要再自己手动调用pmemobj_persist接口。

```
/*
 * pmemobj_tx_add_range -- adds persistent memory range into the transaction
 */
int
pmemobj_tx_add_range(PMEMoid oid, uint64_t hoff, size_t size)
/*
 * pmemobj_tx_add_range_direct -- adds persistent memory range into the
 *                                transaction
 */
int
pmemobj_tx_add_range_direct(const void *ptr, size_t size)
```

pmemobj_tx_add_range接口的第一个参数是持久指针PMEMoid，第二个参数是要分配的内存在PMEMoid指向的持久内存的偏移（也就是在根对象内的偏移），第三个参数代表要添加的内存块的大小，看下面的例子会更清晰一点：

```
struct vector {
        int x;
        int y;
        int z;
```

```
}
// 创建根对象
PMEMoid root = pmemobj_root(pop, sizeof (struct vector));
// 获得其直接指针
struct vector *vectorp = pmemobj_direct(root);
TX_BEGIN(pop) {
    // 为根对象的x成员添加事务内存块
        pmemobj_tx_add_range(root, offsetof(struct vector, x), sizeof(int));
    // 之后可以放心操作x
        vectorp->x = 5;
    // 为根对象的y成员添加事务内存块
        pmemobj_tx_add_range(root, offsetof(struct vector, y), sizeof(int));
    // 之后可以放心操作y
        vectorp->y = 10;
    // 为根对象的z成员添加事务内存块
        pmemobj_tx_add_range(root, offsetof(struct vector, z), sizeof(int));
    // 之后可以放心操作z
        vectorp->z = 15;
} TX_END
```

上面的代码中，在事务中通过三次调用pmemobj_tx_add_range在undo log中添加了三个日志项，看上去上面的操作显得太繁琐了，特别是在根对象比较复杂的时候，因此，pmdk允许我们将根对象作为一个整体，只需要调用一次pmemobj_tx_add_range在undo log中添加一个日志项就可以了，因此代码可以修改为：

```
struct vector *vectorp = pmemobj_direct(root);
TX_BEGIN(pop) {
    // 将根对象作为一个整体，只需要在undo log中添加一个日志项
        pmemobj_tx_add_range(root, 0, sizeof (struct vector));
    // 之后可以放心的操作根对象
        vectorp->x = 5;
        vectorp->y = 10;
        vectorp->z = 15;
} TX_END
```

再来说一下pmemobj_tx_add_range_direct接口，它完成和pmemobj_tx_add_range一样的事情，唯一的区别就是，pmemobj_tx_add_range_direct直接使用成员的直接指针，而不是持久指针，如下：

```
struct vector *vectorp = pmemobj_direct(root);
int *to_modify = &vectorp->x;
TX_BEGIN(pop) {
        pmemobj_tx_add_range_direct(to_modify, sizeof (int));
        *to_modify = 5;
} TX_END
```

经过上面的讨论，我们似乎看到，仅仅使用TX_BEGIN和TX_END就可以完成事务的操作，那么是不是就不需要TX_ONCOMMIT（当事务被提交的时候执行）和TX_ONABORT（放事务失败的时候被执行）了呢？ 其实，如果只有单个事务（没有嵌套事务），这种想法就是对的，但是一旦出现了嵌套事务，那么问题就来了，先看以下代码：

```
#define MAX_HASHMAP 1000
// TOID将在后文的类型安全章节讨论，此处可以先忽略，它只是声明了一个类型
// 需要注意的是这里的hashmap是放在易失内存中的
TOID(struct hash_entry) hashmap[MAX_HASHMAP]; /* volatile hashmap */

void hash_set(int key, int value) {
        TOID(struct hash_entry) nentry;

        TX_BEGIN(pop) {
       // 在NVM中分配hash_entry，此处的TX_NEW可以先忽略
                nentry = TX_NEW(struct hash_entry);
                D_RW(nentry)->key = key;
                D_RW(nentry)->value = value;
        } TX_ONCOMMIT {
       // 事务成功时，会把key也加入内存中的hashmap
                size_t hash = hash_func(key);
                if (TOID_IS_NULL(hashmap[hash]))
                        hashmap[hash] = nentry;
                else
                        /* ... */
        } TX_END
}

TX_BEGIN(pop) {
        hash_set(5, 10);
        pmemobj_tx_abort(-1);
} TX_END
```

上面的代码中，最外层事务中pmemobj_tx_abort被执行时，由于嵌套事务的TX_ONCOMMIT已经被执行了，因此这会导致嵌套事务中的hash_entry被回滚，但是内存中的hashmap没办法自动回滚，因此保存了一个无效的持久指针。因此，这里可以在最外层事务中加上一个TX_ONABORT，并提供一个hash_revert_previous函数来解决上面的问题。

TX_ONABORT和TX_ONCOMMIT还可以用来打印结果日志和设置返回值，这都是非常有用和方便的：

```
int do_work() {
        int ret;
        TX_BEGIN(pop) {
        } TX_ONABORT {
                LOG_ERR("work transaction failed");
                ret = 1;
        } TX_ONCOMMIT {
```

```
                LOG("work transaction successful");
                ret = 0;
        } TX_END

        return ret;
}
```

最后，我们来使用本节介绍的事务接口来完成前文的那个实例，这里，我们不再需要在根对象中额外添加一个长度字段了，是不是更为简单：

```
#include <stdio.h>
#include <string.h>
#include <libpmemobj.h>

#define LAYOUT_NAME "intro_2"
#define MAX_BUF_LEN 10
struct my_root {
        char buf[MAX_BUF_LEN];
};

int
main(int argc, char *argv[])
{
        if (argc != 2) {
                printf("usage: %s file-name\n", argv[0]);
                return 1;
        }
    // 创建内存pool
        PMEMobjpool *pop = pmemobj_create(argv[1], LAYOUT_NAME,
                                PMEMOBJ_MIN_POOL, 0666);

        if (pop == NULL) {
                perror("pmemobj_create");
                return 1;
        }
    // 创建根对象
        PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
        struct my_root *rootp = pmemobj_direct(root);

        char buf[MAX_BUF_LEN] = {0};
        if (scanf("%9s", buf) == EOF) {
                fprintf(stderr, "EOF\n");
                return 1;
        }
    // 开始事务操作
        TX_BEGIN(pop) {
           // 向事务中增加一块持久内存
                pmemobj_tx_add_range(root, 0, sizeof(struct my_root));
           // 然后可以放心的操作原来的内存块
                memcpy(rootp->buf, buf, strlen(buf));
        } TX_END
```

```
        pmemobj_close(pop);

        return 0;
}
```

3、类型安全

在前面的章节中，我们多次使用了持久指针（PMEMoid），它本质就是一个简单的C结构体，没有其他的类型信息，这在开发大型程序的时候是非常容易出错的，并且使用起来也比较 蹩脚。因此，本节将介绍类型安全的指针类型，它好比于C++11中的shared_ptr。

为了便于说明，我们再把PMEMoid贴出来一下：

```
typedef struct pmemoid {
        uint64_t pool_uuid_lo;
        uint64_t off;
} PMEMoid;
```

PMEMoid仅仅包含一个uuid（唯一标志一个内存pool）和偏移（在pool内的偏移），直接操作PMEMoid相当于我们直接操作void *指针，都知道，这样做是很容易出错的，比如：

```
PMEMoid car = pmemobj_tx_alloc(pop, sizeof (struct car), TYPE_CAR);
PMEMoid pen = pmemobj_tx_alloc(pop, sizeof (struct pen), TYPE_PEN);
...
car = pen;
```

上面的代码在编译时是没问题的，但是这可能不是你的实际意图。为了避免这种操作，pmdk提供了一些列的宏来避免在指向不同类型的PMEMoid之间直接赋值，并且会产生编译时错误。

还有另一个问题，如果我们想从非特定类型的指针里面获取结构体或者联合体中的成员，那么我们首先需要先把它强转为我们期望的类型，如下：

```
PMEMoid car;
...
struct car *carp = pmemobj_direct(car);
carp->velocity = 0;
```

这会导致代码中的所有对象都需要两个变大方式：PMEMoid和类型化指针。

匿名联合体（Anonymous unions）

一个包含对象实际类型和PMEMoid持久指针的匿名联合体，可以用来在类型转换和赋值操作时做安全检查。

```
#define OID_TYPE(type)\
union {\
        type *_type;\
        PMEMoid oid;\
}
```

当使用了上面的宏之后，下面的代码将会差生编译时错误：

```
// 由于是匿名的，可以直接用来定义变量而不需要实现声明
OID_TYPE(struct car) car;
OID_TYPE(struct pen) pen;
...
OID_ASSIGN_TYPED(car, pen);
```

同样，如果想把PMEMoid转换为类型化指针，可以直接使用DIRECT_RW（以读写方式）和DIRECT_RO（以只读方式），示例如下：

```
OID_TYPE(struct car) car1;
OID_TYPE(struct car) car2;
...
DIRECT_RW(car1)->velocity = DIRECT_RO(car2)->velocity * 2;
```

DIRECT_RW和DIRECT_RO宏定义如下：

```
#define DIRECT_RW(o) ((typeof(*(o)._type)*)pmemobj_direct((o).oid)))
#define DIRECT_RO(o) ((const typeof (*(o)._type)*)pmemobj_direct((o).oid))
```

与后文即将提到的命名联合体相比，匿名联合体不需要进行声明操作，OID_TYPE宏可以随时被用在任何类型上，这是非常简单和实用的。对类型化的指针之间进行赋值必须实用特定的宏来完成，如果将两个匿名两合体包含的type成员类型不匹配，那么将产生如下编译错误：

> error: incompatible types when assigning to type ‘union <anonymous>’ from type ‘union <anonymous>’

要实现这种效果，应该是实用宏OID_ASSIGN_TYPED()，其定义如下：

```
#define OID_ASSIGN_TYPED(lhs, rhs)\
__builtin_choose_expr(\
        __builtin_types_compatible_p(\
                typeof((lhs)._type),\
                typeof((rhs)._type)),\
        (void) ((lhs).oid = (rhs).oid),\
        (lhs._type = rhs._type))
```

它直接使用gcc内建的__builtin_types_compatible_p来完成类型的匹配检测（https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html），如果类型匹配那么真正的复制操作就会发生（复制PMEMoid），否则就执行假的复制操作（复制_type），以便获取编译时异常信息：

```
OID_TYPE(struct car) car;
OID_TYPE(struct pen) pen;

OID_ASSIGN_TYPED(car, pen);
```

会产生如下信息：

> error: assignment from incompatible pointer type [-Werror] (lhs._type = rhs._type)) ^
>
> note: in expansion of macro ‘OID_ASSIGN_TYPED’ OID_ASSIGN_TYPED(car, pen);

类型化的持久性指针不能作为函数的参数，否则将产生编译错误，示例如下：

```
void stop(OID_TYPE(struct car) car)
{
        D_RW(car)->velocity = 0;
}
...
OID_TYPE(struct car) car;
...
stop(car);
```

将会得到以下的错误信息：

> error: incompatible type for argument 1 of ‘stop’ stop(car); ^
>
> note: expected ‘union <anonymous>’ but argument is of type ‘union <anonymous>’
> stop(OID_TYPE(struct car) car)

在每一次对象分配时，libpmemeobj都需要一个类型编号。在使用匿名联合体时，如果要为每一种类型关联一个唯一的类型编号，需要单独的定义或者枚举，然后在匿名联合体中嵌入类型编号，这就要求每次使用OID_TYPE时都要传递类型编号。

命名联合体（Named unions）

命名联合体是第二种类型安全的解决手段，它的基本思想和前文所述的匿名联合体一致，唯一不同的是，每一个从NVM中分配的对象类型（内存）都有一个相关联的命名联合体，它可以持有PMEMoid信息和类型信息，命名联合体的声明可以使用如下的宏来完成，如下：

```
#define TOID(type)\
union _toid_##type##_toid

#define TOID_DECLARE(type)\
TOID(type)\
{\
        PMEMoid oid;\
        type *_type;\
}
```

其中，TOID_DECLARE宏用来声明一个命名联合体，该联合体被用作类型化的持久指针（typed persistent pointer），而TOID宏被迎来声明一个该类型的变量，示例如下：

```
// 命名联合体需要事先进行类型声明
TOID_DECLARE(struct car);
...

// 用声明过的类型进行变量的定义
TOID(struct car) car1;
TOID(struct car) car2;
...
D_RW(car1)->velocity = 2 * D_RO(car2)->velocity;
```

通过将所需的类型名称与_toid_前缀和_toid后缀连接起来，可以获得一个命名联合体的名称。_toid_前缀需要处理两个符号比如struct name, union name和enum name，在这种情况下，一个宏将被扩展成两个符号，在只有一个符号会被使用的情况下，其中第一个符号声明为一个空的宏（否则从语法层面就是错误的），比如我们使用TOID(struct car)，进行宏扩展如下：

```
union _toid_struct car_toid
```

其中_toid_struct被声明为一个空的宏，类似的_toid_union、_toid_enum也是：

```
#define _toid_struct
#define _toid_union
#define _toid_enum
```

在声明只有一个符号的类型时，前缀和后缀就都会被使用了，比如我们要声明一个size_t类型，TOID宏会被扩展如下：

```
union _toid_size_t_toid
```

因此使用这种机制，可以声明两种符号类型。

在使用命名联合体之后，在类型之前赋值操作也就不需要像前文所说的使用匿名联合体时必须使用特定的宏：

```
// 类型声明
TOID_DECLARE(struct car);
TOID_DECLARE(struct pen);
...

// 变量定义
TOID(struct car) car1;
TOID(struct car) car2;
...

// 可以直接互相赋值
car1 = car2;
```

上面的代码在编译的时候没有任何问题，但是下面的代码将导致编译错误：

```
OID_DECLARE(struct car);
TOID_DECLARE(struct pen);
...

TOID(struct car) car;
TOID(struct pen) pen;
...
car = pen;
```

> error: incompatible types when assigning to type ‘union car_toid’ from type ‘union pen_toid’
> car = pen; ^

在使用命名联合体解决方案之后，我们终于可以将其用于函数的参数声明了，如下：

```
TOID_DECLARE(struct car);

void stop(TOID(struct car) car)
{
        D_RW(car)->velocity = 0;
}
..

TOID(struct car) car;

stop(car);
```

如果将一个不同的类型传递给一个类型化的持久指针，将会导致一个编译错误：

```
TOID_DECLARE(struct car);
TOID_DECLARE(struct pen);

void stop(TOID(struct car) car)
{
        D_RW(car)->velocity = 0;
}
..

TOID(struct pen) pen;

stop(pen);
```

> error: incompatible type for argument 1 of ‘stop’ stop(pen);

因为一个命名联合体在用之前必须先声明，因此类型编号（前文提到过）可以在声明的时候进行赋值。类型值可以在编译时被赋值，修改一下TOID_DECLARE并将其嵌进去：

```
#define TOID_DECLARE(type, type_num)\
typedef uint8_t _toid_##type##_toid_id[(type_num)];\  // 用于根据类型type计算类型编号
TOID(type)\
{\
        PMEMoid oid;\   // 持久指针
        type *_type;\   // 表示的实际类型
        _toid_##type##_toid_id *_id;\  // 用于根据对象计算类型编号
}
```

这个类型编号可以使用sizeof关键字从一个对象或者一个类型中获取：

```
// 获取特定类型的类型编号
#define TOID_TYPE_ID(type) (sizeof (_toid_##type##_toid_id))
```

```
// 获取特定对象的类型编号
#define TOID_TYPE_ID_OF(obj) (sizeof (*(obj)._id))
```

因此，声明一个类型化的持久指针就编程下面这样了：

```
TOID_DECLARE(struct car, 1);
TOID_DECLARE(struct pen, 2);
```

你可能会使用宏或者枚举来定义类型编号：

```
enum {
        TYPE_CAR,
        TYPE_PEN
};

TOID_DECLARE(struct car, TYPE_CAR);
TOID_DECLARE(struct pen, TYPE_PEN);
```

上面的方式需要在声明类型的时候显示的给类型编号赋值。由于从pmemeobj中分配的类型在编译时是已知的，所以可以通过声明一个pool的布局（layout）来声明所有类型，而不需要显示的分配类型编号：

```
/*
 * Declaration of layout
 */
POBJ_LAYOUT_BEGIN(my_layout)
// 对TOID_DECLARE的包装
POBJ_LAYOUT_TOID(my_layout, struct car)
// 对TOID_DECLARE的包装
POBJ_LAYOUT_TOID(my_layout, struct pen)
POBJ_LAYOUT_END(my_layout)
```

使用上述声明方式，所有在POBJ_LAYOUT_BEGIN和POBJ_LAYOUT_END之间声明的类型都将被隐式的赋予连续的类型编号。其原理如下：

```
/*
 * Declaration of typed OID inside layout declaration
 */
#define POBJ_LAYOUT_TOID(name, t)\
TOID_DECLARE(t, (__COUNTER__ + 1 - _POBJ_LAYOUT_REF(name)));

#define _POBJ_LAYOUT_REF(name) (sizeof(_pobj_layout_##name##_ref))
```

原理就是使用编译计数器__COUNTER__。

布局声明（Layout declaration）

在我们前面的所有示例中，我们可以看到，在NVM编程模型中，始终有一个特殊的概念存在，那就是布局（layout），我们需要清晰的定义我们的布局，并且最好将其定义在一个单独的文件中。同时，为了做到运行时类型安全，pmdk提供了更多的宏来包装了布局的定义，比如：

```
// string_store表示当前声明的这个布局的名字
POBJ_LAYOUT_BEGIN(string_store);
// 注意，一个pool布局中，最多只会存在一个根对象，但是可以存在任意多个其他非根对象
POBJ_LAYOUT_ROOT(string_store, struct my_root);
POBJ_LAYOUT_END(string_store);

#define MAX_BUF_LEN 10
struct my_root {
        char buf[MAX_BUF_LEN];
};
```

其中，我们来看一下POBJ_LAYOUT_BEGIN、POBJ_LAYOUT_ROOT和POBJ_LAYOUT_END宏做了什么事情：

```
/* 根对象的类型编号一定为0 */
#define POBJ_ROOT_TYPE_NUM 0

/*
 * Begin of layout declaration
 */
#define POBJ_LAYOUT_BEGIN(name)\
typedef uint8_t _pobj_layout_##name##_ref[__COUNTER__ + 1]

/*
 * Declaration of typed OID of an object
 * 声明一个普通对象，一个pool布局中可以多个
 */
#define TOID_DECLARE(t, i) _TOID_DECLARE(t, i)

/*
 * Declaration of typed OID of a root object
 * 声明一个根对象，一个pool布局中最多一个
 */
#define TOID_DECLARE_ROOT(t) _TOID_DECLARE(t, POBJ_ROOT_TYPE_NUM)

/*
 * End of layout declaration
 */
#define POBJ_LAYOUT_END(name)\
typedef char _pobj_layout_##name##_cnt[__COUNTER__ + 1 -\
_POBJ_LAYOUT_REF(name)];
```

因此我们尝试把上面的代码用宏展开，结果如下：

```
// 布局的起始编号
typedef uint8_t _pobj_layout_string_store_ref[__COUNTER__ + 1]

typedef uint8_t  my_root_toid_type_num[1];// 用于根据类型求类型编号
union  my_root_toid
{
        PMEMoid oid;// 持久指针
        struct my_root *_type;// 实际类型
    my_root_toid_type_num *_type_num;// 类型编码，对其求sizeof就得到类型编码的值，根对象的类型编码
}
// 布局的结束编号，可以算出布局中声明了多少个对象
typedef char _pobj_layout_string_store_cnt[__COUNTER__ + 1 - (sizeof(_pobj_layout_string_store_re
```

简单列举下layout同时包含多个对象的情况：

```
POBJ_LAYOUT_BEGIN(slab_allocator);
POBJ_LAYOUT_ROOT(slab_allocator, struct root);// 最多只能有一个根对象被其他对象依附
POBJ_LAYOUT_TOID(slab_allocator, struct bar);// 依附在根对象上的其他对象
POBJ_LAYOUT_TOID(slab_allocator, struct foo);// 依附在根对象上的其他对象
POBJ_LAYOUT_END(slab_allocator);

struct foo {
        char data[100];
};

struct bar {
        char data[500];
};

struct root {
        TOID(struct foo) foop;
        TOID(struct bar) barp;
};
```

将上面的代码用宏展开之后，得到：

```
typedef uint8_t _pobj_layout_slab_allocator_ref[__COUNTER__ + 1]
// 根对象
typedef uint8_t root_toid_type_num[1];\
union  root_toid\
{\
        PMEMoid oid;\
        struct root *_type;\
        root_toid_type_num *_type_num;\
}
// 之所以减去sizeof(_pobj_layout_slab_allocator_ref)是为了包装每一个单独的布局中编号都是从头开始，
typedef uint8_t foo_toid_type_num[(__COUNTER__ + 1 - (sizeof(_pobj_layout_slab_allocator_ref))) +
```

```
union foo_toid\
{\
        PMEMoid oid;\
        t *_type;\
        foo_toid_type_num *_type_num;\
}

typedef uint8_t bar_toid_type_num[(__COUNTER__ + 1 - (sizeof(_pobj_layout_slab_allocator_ref))) +
union bar_toid\
{\
        PMEMoid oid;\
        t *_type;\
        bar_toid_type_num *_type_num;\
}

typedef char _pobj_layout_slab_allocator_cnt[__COUNTER__ + 1 -(sizeof(_pobj_layout_slab_allocator
```

PMEMoid和TOID之间的转换操作

pmdk一共提供了两种类型安全的宏：一种是作用在PMEMoid上，以OID_为前缀的，另一种是作用在类型化的以TOID_为前缀的宏。另外，所有以pmemobj_为前缀的函数都只接受PMEMoid类型的指针参数。通常情况下还是比较推荐用宏，但有些时候我们也需要PMEMoid，此时可以这么做：

```
TOID(struct foo) data;
pmemobj_direct(data.oid);
```

所有没有以OID_或TOID_为前缀的宏通常都是使用类型化的指针，并将他们作为结果返回，比如

```
#define POBJ_ROOT(pop, t) (\
(TOID(t))pmemobj_root((pop), sizeof(t)))
```

运行时类型安全

前文所述，在每一个布局中，每一个类型都会内在分配一个唯一的类型编号，它主要用于后期的类型验证。比如，你的程序可能会这样更改你的布局：

```
// 第一个版本的布局
struct my_root_v1 {
    // 定义变量
        TOID(struct foo) data;
}
// 第二个版本的布局
struct my_root_v2 {
    // 定义变量
```

```
        TOID(struct bar) data;
}
```

为了检测是使用的布局版本和已经存在的布局是否相同，那么可以用以下的方法：

```
if (TOID_VALID(D_RO(root)->data)) {
        /* can use the data ptr safely */
} else {
        /* declared type doesn't match the object */
}
```

TOID_VALID宏就是用类型编号来完成的：

```
/*
 * Validates whether type number stored in typed OID is the same
 * as type number stored in object's metadata
 */
#define TOID_VALID(o) (TOID_TYPE_NUM_OF(o) == pmemobj_type_num((o).oid))
/*
 * Type number of object read from typed OID
 */
#define TOID_TYPE_NUM_OF(o) (sizeof(*(o)._type_num) - 1)
```

pmemobj_type_num会找到对象的类型编号：

```
/*
 * pmemobj_type_num -- returns type number of object
 */
uint64_t
pmemobj_type_num(PMEMoid oid)
{
    // 先根据uuid找到对应的pool
        PMEMobjpool *pop = pmemobj_pool_by_oid(oid);
        return (palloc_extra(&pop->heap, oid.off));
}
```

当然你也可以直接使用内嵌的类型编号，比如：

```
PMEMoid data;
TOID(struct foo) foo;
TOID(struct bar) bar;
// 原理都是通过类型编号类检测是否是同一个类型定义的变量
if (OID_INSTANCEOF(data, struct foo)) {
        TOID_ASSIGN(foo, data);
} else if (OID_INSTANCEOF(data, struct bar)) {
        TOID_ASSIGN(bar, data);
} else {
```

```
        /* error */
}
```

OID_INSTANCEOF宏的实现如下：

```
/*
 * Checks whether the object is of a given type
 */
#define OID_INSTANCEOF(o, t) (TOID_TYPE_NUM(t) == pmemobj_type_num(o))
/*
 * Type number of specified type
 */
#define TOID_TYPE_NUM(t) (sizeof(_toid_##t##_toid_type_num) - 1)

#define TOID_ASSIGN(o, value) ((o).oid = value, (o))
```

示例

```
#include <stdio.h>
#include <string.h>
#include <libpmemobj.h>

#define MAX_BUF_LEN 10
// 使用类型安全的方式声明布局
POBJ_LAYOUT_BEGIN(string_store);
POBJ_LAYOUT_ROOT(string_store, struct my_root);
POBJ_LAYOUT_END(string_store);

struct my_root {
        char buf[MAX_BUF_LEN];
};
int
main(int argc, char *argv[])
{
        if (argc != 2) {
                printf("usage: %s file-name\n", argv[0]);
                return 1;
        }

        PMEMobjpool *pop = pmemobj_create(argv[1],
                        POBJ_LAYOUT_NAME(string_store), PMEMOBJ_MIN_POOL, 0666);

        if (pop == NULL) {
                perror("pmemobj_create");
                return 1;
        }

        char buf[MAX_BUF_LEN] = {0};
        int num = scanf("%9s", buf);
```

```
        if (num == EOF) {
                fprintf(stderr, "EOF\n");
                return 1;
        }
    // 获取根对象
        TOID(struct my_root) root = POBJ_ROOT(pop, struct my_root);

        TX_BEGIN(pop) {
                TX_MEMCPY(D_RW(root)->buf, buf, strlen(buf));
        } TX_END

        pmemobj_close(pop);

        return 0;
}
```

TX_MEMCPY仅仅是对前文我们提到的函数的一个封装而已：

```
static inline void *
TX_MEMCPY(void *dest, const void *src, size_t num)
{
        pmemobj_tx_add_range_direct(dest, num);
        return memcpy(dest, src, num);
}
```

4、支持事务的动态类型分配

pmemobj库在NVM上实现了临时内存分配器（scratch memory allocator），它主要有两类API：事务型的和非事务型的。

事务性分配

让我们先以一个代码开始（在常规的易失性内存）：

```
struct rectangle {
        int a;
        int b;
};

int perimeter_calc(const struct rectangle *rect) {
        return rect->a * rect->b;
}

...
struct rectangle *rect = malloc(sizeof *rect);
if (rect == NULL) return;
rect->a = 5;
```

```
rect->b = 10;
int p = perimeter_calc(rect);
/* busy work */
free(rect):
```

根据前面所有章节的讨论，我们很容易将其修改到NVM上，除了malloc和free我们还没有介绍：

```
/* struct rectangle doesn't change */

strucy my_root {
        TOID(struct rectangle) rect;
};

POBJ_LAYOUT_BEGIN(rect_calc);
// 声明根对象类型
POBJ_LAYOUT_ROOT(rect_calc, struct my_root);
// 声明内嵌对象类型
POBJ_LAYOUT_TOID(rect_calc, struct rectangle);
POBJ_LAYOUT_END(rect_calc);
```

将perimeter_calc函数修改为使用持久指针：

```
int perimeter_calc(const TOID(struct rectangle) rect) {
        return D_RO(rect)->a * D_RO(rect)->b;
}
```

参数中使用const意味着在函数内部你不想使用D_RW，否则就会编译出错。

rectangle对象的分配和初始化必须在事务中完成，因为这是在NVM中，因此我们必须找到一种方式在应用程序重启的时候还能找到你刚才分配的对象地址，是的，这里我们还是使用根对象（因为一个根对象在一个pool中是唯一的），在根对象中我们内嵌一个rect变量：

```
TOID(struct my_root) root = POBJ_ROOT(pop);
TX_BEGIN(pop) {
        TX_ADD(root); /* 只是对pmemobj_tx_add_range的包装 */
        TOID(struct rectangle) rect = TX_NEW(struct rectangle);/* 对pmemobj_tx_alloc的包装 */
        D_RW(rect)->x = 5;
        D_RW(rect)->y = 10;
    // 将分配并初始化好的对象赋值给根对象，以便后面可以找到它
        D_RW(root)->rect = rect;
} TX_END

int p = perimeter_calc(D_RO(root)->rect);
/* busy work */
```

上面的代码中唯一新出现的就是TX_NEW宏，它简单的分配一块大小为sizeof(T)的内存，并返回一个TOID(t)，因此后期你只能将正确的类型赋值给它。如果你想自己定义内存块的带下，比如你在分配数组，此时你可以使用TX_ALLOC函数（或者使用Z前缀的版本，它会默认将内存初始化为0），同样需要注意的是，这些分配的内存在事务提交的时候都是自动持久化的，因此你不需要自己手动执行persist。看上去TX_ADD也是一个新出现的宏，但是它只是对pmemobj_add_range的简单封装。

```
#define TX_ADD(o)\
pmemobj_tx_add_range((o).oid, 0, sizeof(*(o)._type))
```

TX_NEW实现如下：

```
#define TX_NEW(t)\
((TOID(t))pmemobj_tx_alloc(sizeof(t), TOID_TYPE_NUM(t)))
```

在我们使用完释放这块内存时，我们可以这样做：

```
TX_BEGIN(pop) {
        TX_ADD(root);// 添加到undo log
        TX_FREE(D_RW(root)->rect);// 释放内存，对pmemobj_tx_free的包装
        D_RW(root)->rect = TOID_NULL(struct rectangle);// 根对象中置空
} TX_END
```

释放操作同样需要在一个事务中，因为它也是一个两阶段操作，同时记得要把根事务中持有的指针置为NULL，这有两种方式：

```
// 方式一
D_RW(root)->rect.oid = OID_NULL;
// 方式二
TOID_ASSIGN(D_RW(root)->rect, OID_NULL);
```

非事务下的内存原子型分配

上一节介绍的事务型的内存分配方法，它用起来比较方便，但是它需要维护一个undo log因此增大了很多开销。一个更高效的内存分配提供一个非事务的、原子性的分配方式。这就是本节要讨论的。

相比事务型，非事务型（指的是不需要将其包裹在一个外层事务中）内存分配的接口看上去不是那么正常。首先，这些函数要么分配要么释放内存指针，对目标指针的修改都是原子性的，所以操作全都是有效的（指针要么指向一个合法的初始化过的内存，要么就是OID_NULL）。这些函数或者

宏会强制你将分配的对象初始化为一个确定的状态，比如使用POBJ_ZNEW或POBJ_ZALLOC将其初始化为0，或者自己提供一个构造函数（POBJ_NEW和POBJ_ALLOC）。

因此可以将之前的例子改写成：

```
// 作为构造函数
int rect_construct(PMEMobjpool *pop, void *ptr, void *arg) {
        struct rectangle *rect = ptr;
        rect->x = 5;
        rect->y = 10;
        pmemobj_persist(pop, rect, sizeof *rect);

        return 0;
}
// 注意此处的内存操作是原子的，传入的构造函数用于内存初始化
POBJ_NEW(pop, &D_RW(root)->rect, struct rectangle, rect_construct, NULL);
int p = perimeter_calc(D_RO(root)->rect);
/* busy work */

// 这里的释放也是原子的
POBJ_FREE(&D_RW(root)->rect);
```

上面用到的所有原子宏定义如下：

```
#define POBJ_NEW(pop, o, t, constr, arg)\
pmemobj_alloc((pop), (PMEMoid *)(o), sizeof(t), TOID_TYPE_NUM(t),\
        (constr), (arg))

#define POBJ_ALLOC(pop, o, t, size, constr, arg)\
pmemobj_alloc((pop), (PMEMoid *)(o), (size), TOID_TYPE_NUM(t),\
        (constr), (arg))

#define POBJ_ZNEW(pop, o, t)\
pmemobj_zalloc((pop), (PMEMoid *)(o), sizeof(t), TOID_TYPE_NUM(t))

#define POBJ_ZALLOC(pop, o, t, size)\
pmemobj_zalloc((pop), (PMEMoid *)(o), (size), TOID_TYPE_NUM(t))

#define POBJ_REALLOC(pop, o, t, size)\
pmemobj_realloc((pop), (PMEMoid *)(o), (size), TOID_TYPE_NUM(t))

#define POBJ_ZREALLOC(pop, o, t, size)\
pmemobj_zrealloc((pop), (PMEMoid *)(o), (size), TOID_TYPE_NUM(t))

#define POBJ_FREE(o)\
pmemobj_free((PMEMoid *)(o))
```

上面的这种用法看起来和之前的接口有些不同，但是用不用这种方式实现内存的事务操作完全取决于你。在 考虑性能的前提下，如果你会分配大量的对象，那么这种方式可以具有更好的新能。它

还允许你使用_persist函数对内存进行更细粒度的控制（再大型对象中尤其有用，因为你可以在构造函数中只刷新真正需要刷新的部分）。

构造函数还支持返回非零值取消正在进行的分配操作，这在构造函数中还依赖于其他的不同受限资源的时候非常有用，比如，它允许你在构造函数中进行易失性内存的分配，而且如果分配失败那么整个操作可以回滚。目标指针是可选的，它也可以是一个栈上的变量，其实更正确的使用方式是使用内部集合（Internal collections），在pmdk中，所有已经存在的对象都被存储在一个集合中，你可以通过POBJ_FIRST和POBJ_NEXT接口来访问和遍历这些对象。有了该集合，你将永远不会丢失对象的引用（操作NVM的内存泄露），也可以将其作为一个无序列表来遍历对象，而不用再讲对象绑定在一个根对象上。比如：

```
TOID(struct rectangle) rect = POBJ_FIRST(pop, struct rectangle);
```

如果你拥有多个struct rectangle对象，那么你可以这样获取他们：

```
rect = POBJ_NEXT(rect, struct rectangle);
```

但是你不需要自己手动迭代，pmdk为我们提供了宏定义：

```
TOID(struct rectangle) iter;
POBJ_FOREACH_TYPE(pop, iter) {
        int p = perimeter_calc(D_RO(iter));
        printf("Perimeter of rectangle = %d", p);
}
```

上面将会根据迭代器的类型(struct rectangle)迭代rectangle对象。POBJ_FOREACH_TYPE宏定义如下：

```
#define POBJ_FIRST(pop, t) ((TOID(t))POBJ_FIRST_TYPE_NUM(pop, TOID_TYPE_NUM(t)))

#define POBJ_NEXT(o) ((__typeof__(o))POBJ_NEXT_TYPE_NUM((o).oid))

/*
 * Iterates through every object of the specified type.
 */
#define POBJ_FOREACH_TYPE(pop, var)\
POBJ_FOREACH(pop, (var).oid)\
if (pmemobj_type_num((var).oid) == TOID_TYPE_NUM_OF(var))

/*
 * Iterates through every existing allocated object.
 */
#define POBJ_FOREACH(pop, varoid)\
```

```
for (_pobj_debug_notice("POBJ_FOREACH", __FILE__, __LINE__),\
        varoid = pmemobj_first(pop);\
                (varoid).off != 0; varoid = pmemobj_next(varoid))
```

如果你只想迭代所有类型的对象，那么可以使用宏POBJ_FOREACH来完成：

```
/*
 * Iterates through every existing allocated object.
 */
#define POBJ_FOREACH(pop, varoid)\
for (_pobj_debug_notice("POBJ_FOREACH", __FILE__, __LINE__),\
        varoid = pmemobj_first(pop);\
                (varoid).off != 0; varoid = pmemobj_next(varoid))
```

如果想释放对象怎么做呢？pmdk提供了两个相应的释放宏：

```
/*
 * Safe variant of POBJ_FOREACH in which pmemobj_free on varoid is allowed
 */
#define POBJ_FOREACH_SAFE(pop, varoid, nvaroid)\
for (_pobj_debug_notice("POBJ_FOREACH_SAFE", __FILE__, __LINE__),\
        varoid = pmemobj_first(pop);\
                (varoid).off != 0 && (nvaroid = pmemobj_next(varoid), 1);\
                varoid = nvaroid)
/*
 * Safe variant of POBJ_FOREACH_TYPE in which pmemobj_free on var
 * is allowed.
 */
#define POBJ_FOREACH_SAFE_TYPE(pop, var, nvar)\
POBJ_FOREACH_SAFE(pop, (var).oid, (nvar).oid)\
if (pmemobj_type_num((var).oid) == TOID_TYPE_NUM_OF(var))
```

关于原子性接口的使用例子可以参考官方代码：
https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/pminvaders

四、线程

pmemobj库提供的所有函数都是线程安全的，但有两个列外，一个是pool的管理函数（open，close等），另一个就是pmemobj_root（在不同的线程中使用不同的size）。至于宏，通常只有FOREACH宏不是线程安全的，原因显而易见。

1、同步

如果你想在驻留在NVM中的结构中放一把锁，pmdk同样提供了一个类型pthread接口来达到此目的。但是你不用去初始化这些所或者验证它的状态，当应用程序crash的时候它们将自动解锁，比如：

```
struct foo {
        pthread_mutex_t lock;
        int bar;
};

int fetch_and_add(TOID(struct foo) foo, int val) {
        pthread_mutex_lock(&D_RW(foo)->lock);

        int ret = D_RO(foo)->bar;
        D_RW(foo)->bar += val;

        pthread_mutex_unlock(&D_RW(foo)->lock);

        return ret;
}
```

如果在fetch_and_add函数的任何地方发生crash，pthread_mutex_t结构将包含一个无效的值，当程序再次尝试访问它时很可能会产生segfault。解决方法是为每个单独的锁调用pthread_mutex_init函数，这里给出一个可能的方式：

```
struct foo {
        PMEMmutex lock;
        int bar;
};

int fetch_and_add(TOID(struct foo) foo, int val) {
        pmemobj_mutex_lock(pop, &D_RW(foo)->lock);

        int ret = D_RO(foo)->bar;
        D_RW(foo)->bar += val;

        pmemobj_mutex_unlock(pop, &D_RW(foo)->lock);

        return ret;
}
```

PMEMmutex结构定义如下：

```
typedef struct pmemmutex {
        char data[64];
} PMEMmutex;
```

五、事务原理

还记得前文介绍的事务使用方式，再次列出：

```
/* TX_STAGE_NONE */

TX_BEGIN(pop) {
        /* TX_STAGE_WORK */
} TX_ONCOMMIT {
        /* TX_STAGE_ONCOMMIT */
} TX_ONABORT {
        /* TX_STAGE_ONABORT */
} TX_FINALLY {
        /* TX_STAGE_FINALLY */
} TX_END

/* TX_STAGE_NONE */
```

其实， 只是一些宏定义的包装而已，为了探究其时间本质，我们先将宏全部进行展开。

先看TX_BEGIN实现，在tx.h中：

```
#define TX_BEGIN(pop) _POBJ_TX_BEGIN(pop, TX_PARAM_NONE)

#define _POBJ_TX_BEGIN(pop, ...)\
{\
        jmp_buf _tx_env;\
        int _stage;\
        int _pobj_errno;\
        if (setjmp(_tx_env)) {\
                errno = pmemobj_tx_errno();\
        } else {\
                _pobj_errno = pmemobj_tx_begin(pop, _tx_env, __VA_ARGS__,\
                                TX_PARAM_NONE);\
                if (_pobj_errno)\
                        errno = _pobj_errno;\
        }\
        while ((_stage = pmemobj_tx_stage()) != TX_STAGE_NONE) {\
                switch (_stage) {\
                        case TX_STAGE_WORK:
```

TX_ONCOMMIT实现：

```
#define TX_ONCOMMIT\
                                pmemobj_tx_process();\
                                break;\
                        case TX_STAGE_ONCOMMIT:
```

TX_ONABORT实现：

```
#define TX_ONABORT\
                                pmemobj_tx_process();\
                                break;\
                        case TX_STAGE_ONABORT:
```

TX_FINALLYS实现：

```
#define TX_FINALLY\
                                pmemobj_tx_process();\
                                break;\
                        case TX_STAGE_FINALLY:
```

TX_END实现：

```
#define TX_END\
                                pmemobj_tx_process();\
                                break;\
                        default:\
                                TX_ONABORT_CHECK;\
                                pmemobj_tx_process();\
                                break;\
                }\
        }\
        _pobj_errno = pmemobj_tx_end();\
        if (_pobj_errno)\
                errno = _pobj_errno;\
}
```

综上，将前文的事务代码进行宏展开之后，如下：

```
{
  jmp_buf _tx_env;// 存储当前运行上下文
  int _stage;// 事务当前所处于的阶段
  int _pobj_errno;
  if (setjmp(_tx_env)) {// 保存当前上下文
      errno = pmemobj_tx_errno();
  } else {
     // 开始事务，__VA_ARGS__将可变参数直接传给pmemobj_tx_begin，TX_PARAM_NONE表示参数结束
     _pobj_errno = pmemobj_tx_begin(pop, _tx_env, __VA_ARGS__,TX_PARAM_NONE);
    if (_pobj_errno)
      errno = _pobj_errno;
  }
  // 开始事务生命周期
  while ((_stage = pmemobj_tx_stage()) != TX_STAGE_NONE) {
    switch (_stage) {// 判断当前处于事务的哪个阶段
```

```
      case TX_STAGE_WORK:
        /* TX_STAGE_WORK */
        pmemobj_tx_process();// 驱动事务到下一个阶段
      break;
      case TX_STAGE_ONCOMMIT:
        /* TX_STAGE_ONCOMMIT */
        pmemobj_tx_process();// 驱动事务到下一个阶段
      break;
      case TX_STAGE_ONABORT:
        /* TX_STAGE_ONABORT */
        pmemobj_tx_process();// 驱动事务到下一个阶段
      break;
      case TX_STAGE_FINALLY:
        /* TX_STAGE_FINALLY */
        pmemobj_tx_process();// 驱动事务到下一个阶段
      break;
      default:
        if (_stage == TX_STAGE_ONABORT)
           abort();
        pmemobj_tx_process();// 驱动事务到下一个阶段
      break;
    }
  }
  // 事务结束，清理事务资源
  _pobj_errno = pmemobj_tx_end();
  if (_pobj_errno)
     errno = _pobj_errno;
}
```

首先看事务上下文结构：

```
struct tx {
        PMEMobjpool *pop;// 事务上下文对应的pool描述符指针
        enum pobj_tx_stage stage;// 事务当前阶段
        int last_errnum;
        struct lane_section *section;// 会存在持久内存中
        struct txd {
             struct tx_data *slh_first;   // 头结点
    } tx_locks; // tx_lock_data链表
        struct txd {
             struct tx_data *slh_first;   // 头结点
    } tx_entries;// tx_data链表，里面包含jmp_buf

    //typedef void (*pmemobj_tx_callback)(PMEMobjpool *pop, enum pobj_tx_stage stage,void *);
        pmemobj_tx_callback stage_callback;// 每个阶段对应的毁掉函数
        void *stage_callback_arg;// 传给回调函数的参数
};
```

重点关注tx_data：

```
struct tx_data {
        struct {
             struct tx_data *sle_next;// 下一个元素
    } tx_entry;
        jmp_buf env;// 运行上下文
};
```

事务开始代码：

```
/*
 * pmemobj_tx_begin -- initializes new transaction
 */
int
pmemobj_tx_begin(PMEMobjpool *pop, jmp_buf env, ...)
{
        LOG(3, NULL);

        int err = 0;
        struct tx *tx = get_tx();// 获取线程局部存储的事务上下文

        struct lane_tx_runtime *lane = NULL;
        if (tx->stage == TX_STAGE_WORK) {// 说明这是一个嵌套事务
                ASSERTne(tx->section, NULL);
                if (tx->pop != pop) {// 校验，嵌套的事务必须同时服务于同一个pool
                        ERR("nested transaction for different pool");
                        return obj_tx_abort_err(EINVAL);
                }

        } else if (tx->stage == TX_STAGE_NONE) {// 说明这是一个根事务

                unsigned idx = lane_hold(pop, &tx->section,
                        LANE_SECTION_TRANSACTION);

                lane = tx->section->runtime;
                VALGRIND_ANNOTATE_NEW_MEMORY(lane, sizeof(*lane));

                SLIST_INIT(&tx->tx_entries);
                SLIST_INIT(&tx->tx_locks);

                lane->ranges = ctree_new();
                lane->cache_offset = 0;
                lane->lane_idx = idx;

                lane->actvcnt = 0;
                lane->actvundo = 0;

                struct lane_tx_layout *layout =
                        (struct lane_tx_layout *)tx->section->layout;
        // 重建redo
                if (tx_rebuild_undo_runtime(pop, layout, &lane->undo) != 0) {
                        tx->stage = TX_STAGE_ONABORT;
                        err = errno;
```

```
                return err;
        }

        tx->pop = pop;
    } else {
        FATAL("Invalid stage %d to begin new transaction", tx->stage);
    }

    struct tx_data *txd = Malloc(sizeof(*txd));
    if (txd == NULL) {
        err = errno;
        ERR("!Malloc");
        goto err_abort;
    }

    tx->last_errnum = 0;
    if (env != NULL)
        memcpy(txd->env, env, sizeof(jmp_buf));
    else
        memset(txd->env, 0, sizeof(jmp_buf));

    SLIST_INSERT_HEAD(&tx->tx_entries, txd, tx_entry);

    tx->stage = TX_STAGE_WORK;

    /* handle locks */
    va_list argp;
    va_start(argp, env);
    enum pobj_tx_param param_type;
// 解析传给pmemobj_tx_begin接收的不定参数，直到TX_PARAM_NONE
    while ((param_type = va_arg(argp, enum pobj_tx_param)) != TX_PARAM_NONE) {
       // 如果参数类型为设置回调函数
  if (param_type == TX_PARAM_CB) {
       // 解析回调函数
                pmemobj_tx_callback cb = va_arg(argp, pmemobj_tx_callback);
       // 解析回调函数的参数
                void *arg = va_arg(argp, void *);

                if (tx->stage_callback &&
                        (tx->stage_callback != cb ||
                        tx->stage_callback_arg != arg)) {
                    FATAL("transaction callback is already set, "
                        "old %p new %p old_arg %p new_arg %p",
                        tx->stage_callback, cb,
                        tx->stage_callback_arg, arg);
                }
       // 设置回调函数和参数
                tx->stage_callback = cb;
                tx->stage_callback_arg = arg;
        } else {
                err = add_to_tx_and_lock(tx, param_type,va_arg(argp, void *));
                if (err) {
                    va_end(argp);
                    goto err_abort;
                }
```

```
                }
        }
        va_end(argp);
        return 0;
err_abort:
        if (tx->stage == TX_STAGE_WORK)
                obj_tx_abort(err, 0);
        else
                tx->stage = TX_STAGE_ONABORT;
        return err;
}
```

tx_rebuild_undo_runtime用于重建undo日志：

```
/*
 * tx_rebuild_undo_runtime -- (internal) reinitializes runtime state of vectors
 */
static int
tx_rebuild_undo_runtime(PMEMobjpool *pop, struct lane_tx_layout *layout,
        struct tx_undo_runtime *tx_rt)
{
        LOG(7, NULL);

        int i;
        for (i = UNDO_ALLOC; i < MAX_UNDO_TYPES; ++i) {
                if (tx_rt->ctx[i] == NULL)
                        tx_rt->ctx[i] = pvector_new(pop, &layout->undo_log[i]);
                else
                        pvector_reinit(tx_rt->ctx[i]);

                if (tx_rt->ctx[i] == NULL)
                        goto error_init;
        }

        return 0;

error_init:
        for (--i; i >= 0; --i)
                pvector_delete(tx_rt->ctx[i]);

        return -1;
}
```

pmemobj_tx_process用于驱动事务的生命周期到下一个阶段，实现如下：

```
/*
 * pmemobj_tx_process -- processes current transaction stage
 */
void
pmemobj_tx_process(void)
{
```

```
        LOG(5, NULL);
        struct tx *tx = get_tx();

        ASSERT_IN_TX(tx);

        switch (tx->stage) {
        case TX_STAGE_NONE:
                break;
        case TX_STAGE_WORK:
                pmemobj_tx_commit();// 提交事务
                break;
        case TX_STAGE_ONABORT:
        case TX_STAGE_ONCOMMIT:
                tx->stage = TX_STAGE_FINALLY;
                obj_tx_callback(tx);
                break;
        case TX_STAGE_FINALLY:
                tx->stage = TX_STAGE_NONE;
                break;
        case MAX_TX_STAGE:
                ASSERT(0);
        }
}
```

pmemobj_tx_abort是事务的关键，事务中的任何失败都应该调用pmemobj_tx_abort以让事务进行回滚操作：

```
/*
 * obj_tx_abort -- aborts current transaction
 */
static void
obj_tx_abort(int errnum, int user)
{
        struct tx *tx = get_tx();
    // 必须在是在一个事务中调用
        ASSERT_IN_TX(tx);
    // 事务当前必须处于WORK阶段
        ASSERT_TX_STAGE_WORK(tx);

        if (errnum == 0)
                errnum = ECANCELED;
    // 更新事务阶段为TX_STAGE_ONABORT
        tx->stage = TX_STAGE_ONABORT;
        struct lane_tx_runtime *lane = tx->section->runtime;
    // 取出tx_entries链表头部节点
        struct tx_data *txd = SLIST_FIRST(&tx->tx_entries);//#define    SLIST_FIRST(head)
    // #define  SLIST_NEXT(elm, field)  ((elm)->field.sle_next)
        if (SLIST_NEXT(txd, tx_entry) == NULL) {
                /* 如果tx_entries链表只剩一个节点，说明这是最外层事务了 */
                struct lane_tx_layout *layout = (struct lane_tx_layout *)tx->section->layout;

                /* 处理 undo log */
```

```
                tx_abort(tx->pop, lane, layout, 0 /* abort */);
        // 释放lane
                lane_release(tx->pop);
                tx->section = NULL;
        }

        tx->last_errnum = errnum;
        errno = errnum;
        if (user)
                ERR("!explicit transaction abort");

        /* ONABORT */
        obj_tx_callback(tx);// 调用相应的回调（如果设置了的话）
    // 检查给第的地址是不是都是0
    // 此处就是校验jmp_buf的有效性
        if (!util_is_zeroed(txd->env, sizeof(jmp_buf)))
                longjmp(txd->env, errnum);// 跳转到setjmp，并将错误码传给setjmp
}

/*
 * pmemobj_tx_abort -- aborts current transaction
 *
 * Note: this function should not be called from inside of pmemobj.
 */
void
pmemobj_tx_abort(int errnum)
{
        obj_tx_abort(errnum, 1);
}
```

tx_abort主要用于abort事务中所有已经分配了的对象：

```
/*
 * tx_abort -- (internal) abort all allocated objects
 */
static void
tx_abort(PMEMobjpool *pop, struct lane_tx_runtime *lane,
                struct lane_tx_layout *layout, int recovery)
{
        struct tx_undo_runtime *tx_rt;
        struct tx_undo_runtime new_rt = { .ctx = {NULL, } };
        if (recovery) {
                if (tx_rebuild_undo_runtime(pop, layout, &new_rt) != 0)
                        FATAL("!Cannot rebuild runtime undo log state");

                tx_rt = &new_rt;
        } else {
                tx_rt = &lane->undo;
        }
    // 还记得前文曾经说过，pmdk提供了三种内存事务操作，分别是设置、分配、释放，这里就是针对这三种类
        tx_abort_set(pop, tx_rt, recovery);
        tx_abort_alloc(pop, tx_rt, lane);
```

```
        tx_abort_free(pop, tx_rt);

        if (recovery) {
                tx_destroy_undo_runtime(tx_rt);
        } else {
                tx_cancel_reservations(pop, lane);
                ASSERTne(lane, NULL);
                ctree_delete(lane->ranges);
                lane->ranges = NULL;
        }
}
```

pmemobj_tx_end用于在事务结束时清理、释放一些资源：

```
/*
 * pmemobj_tx_end -- ends current transaction
 */
int
pmemobj_tx_end(void)
{
        struct tx *tx = get_tx();

        if (tx->stage == TX_STAGE_WORK)
                FATAL("pmemobj_tx_end called without pmemobj_tx_commit");

        if (tx->pop == NULL)
                FATAL("pmemobj_tx_end called without pmemobj_tx_begin");

        if (tx->stage_callback &&
                        (tx->stage == TX_STAGE_ONCOMMIT ||
                        tx->stage == TX_STAGE_ONABORT)) {
                tx->stage = TX_STAGE_FINALLY;
                obj_tx_callback(tx);
        }

        struct tx_data *txd = SLIST_FIRST(&tx->tx_entries);
        SLIST_REMOVE_HEAD(&tx->tx_entries, tx_entry);

        Free(txd);

        if (SLIST_EMPTY(&tx->tx_entries)) {
                // 已经是最外层事务了
                release_and_free_tx_locks(tx);
                tx->pop = NULL;
                tx->stage = TX_STAGE_NONE;

                if (tx->stage_callback) {
                        pmemobj_tx_callback cb = tx->stage_callback;
                        void *arg = tx->stage_callback_arg;

                        tx->stage_callback = NULL;
                        tx->stage_callback_arg = NULL;
```

```
                        cb(tx->pop, TX_STAGE_NONE, arg);
                }
        } else {
       // 是嵌套事务，还有外层事务未完成
                /* resume the next transaction */
                tx->stage = TX_STAGE_WORK;

                /* abort called within inner transaction, waterfall the error */
                if (tx->last_errnum)
                        obj_tx_abort(tx->last_errnum, 0);
        }

        return tx->last_errnum;
}
```

obj_tx_callback用于回调设置的回调函数：

```
/*
 * obj_tx_callback -- (internal) executes callback associated with current stage
 */
static void
obj_tx_callback(struct tx *tx)
{
        if (!tx->stage_callback)
                return;

        struct tx_data *txd = SLIST_FIRST(&tx->tx_entries);

        /* 先判断是否是最外层事务，只有最外层事务才会回调 */
        if (SLIST_NEXT(txd, tx_entry) == NULL)
                tx->stage_callback(tx->pop, tx->stage, tx->stage_callback_arg);
}
```

pmemobj_tx_commit实现为一个两阶段提交过程：

```
/*
 * pmemobj_tx_commit -- commits current transaction
 */
void
pmemobj_tx_commit(void)
{
        struct tx *tx = get_tx();

        ASSERT_IN_TX(tx);
        ASSERT_TX_STAGE_WORK(tx);

        /* WORK */
        obj_tx_callback(tx);

        struct lane_tx_runtime *lane =
```

```
                (struct lane_tx_runtime *)tx->section->runtime;
        struct tx_data *txd = SLIST_FIRST(&tx->tx_entries);

        if (SLIST_NEXT(txd, tx_entry) == NULL) {
                /* 已经是最外层事务了 */

                struct lane_tx_layout *layout =
                        (struct lane_tx_layout *)tx->section->layout;
                PMEMobjpool *pop = tx->pop;

                /* pre-commit 阶段 */
                tx_pre_commit(pop, tx, lane);
       // 内存屏障
                pmemops_drain(&pop->p_ops);

                /* 设置事务状态为committed，记住这里的修改是原子的 */
                tx_set_state(pop, layout, TX_STATE_COMMITTED);

                if (pop->tx_postcommit_tasks != NULL &&
                        ringbuf_tryenqueue(pop->tx_postcommit_tasks,
                                tx->section) == 0) {
                        lane_detach(pop);
                } else {
                        tx_post_commit_cleanup(pop, tx->section, 0);
                }

                tx->section = NULL;
        }

        tx->stage = TX_STAGE_ONCOMMIT;

        /* ONCOMMIT */
        obj_tx_callback(tx);
}
```

预提交tx_pre_commit实现如下：

```
/*
 * tx_pre_commit -- (internal) do pre-commit operations
 */
static void
tx_pre_commit(PMEMobjpool *pop, struct tx *tx, struct lane_tx_runtime *lane)
{
        ASSERTne(tx->section->runtime, NULL);

        tx_fulfill_reservations(tx);

        /* Flush all regions and destroy the whole tree. */
        ctree_delete_cb(lane->ranges, tx_flush_range, pop);
        lane->ranges = NULL;
}
```

```
/*
 * tx_fulfill_reservations -- fulfills all volatile state
 *      allocation reservations
 */
static void
tx_fulfill_reservations(struct tx *tx)
{
        struct lane_tx_runtime *lane =
                (struct lane_tx_runtime *)tx->section->runtime;

        if (lane->actvcnt == 0)
                return;

        PMEMobjpool *pop = tx->pop;
    // 生成redo log
        struct redo_log *redo = pmalloc_redo_hold(pop);

        struct operation_context ctx;
        operation_init(&ctx, pop, pop->redo, redo);

        palloc_publish(&pop->heap, lane->alloc_actv, lane->actvcnt, &ctx);
        lane->actvcnt = 0;
        lane->actvundo = 0;

        pmalloc_redo_release(pop);
}
```

预提交完成之后，修改事务状态执行最终提交：

```
/*
 * tx_set_state -- (internal) set transaction state
 */
static inline void
tx_set_state(PMEMobjpool *pop, struct lane_tx_layout *layout, uint64_t state)
{
        layout->state = state;// 8字节原子更新
        pmemops_persist(&pop->p_ops, &layout->state, sizeof(layout->state));
}
```

光有外层事务的报过还不足以完成事务的控制，也要用事务内部的操作支持事务的特性，比如，即使在事务包裹中，你直接使用mmap去分配NVM内存，那么这块内存并不受事务管理。因此，根据前文的描述，要在事务中申请一个内存块需要使用特殊的函数，比如pmemobj_tx_add_range_direct，下面我们看一下它的实现：

```
/*
 * pmemobj_tx_add_range_direct -- adds persistent memory range into the
```

```
 *                                       transaction
 */
int
pmemobj_tx_add_range_direct(const void *ptr, size_t size)
{
        struct tx *tx = get_tx();
    // 当前必须处于事务中
        ASSERT_IN_TX(tx);
    // 当前必须处于WORK阶段
        ASSERT_TX_STAGE_WORK(tx);

        PMEMobjpool *pop = tx->pop;

        if (!OBJ_PTR_FROM_POOL(pop, ptr)) {
                ERR("object outside of pool");
                return obj_tx_abort_err(EINVAL);
        }
    // 为内存块申请设置参数
        struct tx_add_range_args args = {
                .pop = pop,
                .offset = (uint64_t)((char *)ptr - (char *)pop),
                .size = size,
                .flags = 0,
        };
    // 通用的内存块申请函数，内部一旦失败就会调用obj_tx_abort_err终止事务
        return pmemobj_tx_add_common(tx, &args);
}
```

六、redis实践

先来看redis里面的根对象和布局是怎么定义的：

```
// 根对象
struct redis_pmem_root {
        uint64_t num_dict_entries;// 字典项计数
        TOID(struct key_val_pair_PM) pe_first;// 头结点
};

typedef struct key_val_pair_PM {
    PMEMoid key_oid;// key的持久指针
    PMEMoid val_oid;// value的持久指针
    TOID(struct key_val_pair_PM) pmem_list_next;
    TOID(struct key_val_pair_PM) pmem_list_prev;
} key_val_pair_PM;

int pmemReconstruct(void);
void pmemKVpairSet(void *key, void *val);
PMEMoid pmemAddToPmemList(void *key, void *val);
void pmemRemoveFromPmemList(PMEMoid kv_PM_oid);
```

根据NVM中持久化的数据重建redis数据库

```
int
pmemReconstruct(void)
{
    // 定义根对象
    TOID(struct redis_pmem_root) root;
    // 定义key_val_pair_PM
    TOID(struct key_val_pair_PM) kv_PM_oid;
    struct key_val_pair_PM *kv_PM;
    dict *d;
    void *key;
    void *val;
    void *pmem_base_addr;// pool基地址

    root = server.pm_rootoid;// pool_uuid
    pmem_base_addr = (void *)server.pm_pool->addr;// 获取pool的基地址
    d = server.db[0].dict;// 获取db0的字典指针
    dictExpand(d, D_RO(root)->num_dict_entries);// 以num_dict_entries大小重建dict
    // 遍历根对象中的kv链表
    for (kv_PM_oid = D_RO(root)->pe_first; TOID_IS_NULL(kv_PM_oid) == 0; kv_PM_oid = D_RO(kv_PM_
                // 找到kv对的直接地址
       kv_PM = (key_val_pair_PM *)(kv_PM_oid.oid.off + (uint64_t)pmem_base_addr);
       // 找到key对应的直接地址
                key = (void *)(kv_PM->key_oid.off + (uint64_t)pmem_base_addr);
       // 找到value对应的直接地址
                val = (void *)(kv_PM->val_oid.off + (uint64_t)pmem_base_addr);
       // 重建数据库
        (void)dictAddReconstructedPM(d, key, val);
    }
    return C_OK;
}
```

dictAddReconstructedPM实现如下：

```
dictEntry *dictAddReconstructedPM(dict *d, void *key, void *val)
{
    int index;
    dictEntry *entry;
    robj *val_robj;
    dictht *ht;

    if (dictIsRehashing(d)) _dictRehashStep(d);

    /* Get the index of the new element, or -1 if
     * the element already exists. */
    if ((index = _dictKeyIndex(d, (const void *)key)) == -1)
        return NULL;

    /* Allocate the memory and store the new entry.
```

```
     * Insert the element in top, with the assumption that in a database
     * system it is more likely that recently added entries are accessed
     * more frequently. */
    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    entry = zmalloc(sizeof(*entry));
    val_robj = createObjectPM(OBJ_STRING, val);// 此处感觉和标准redis没区别

    entry->next = ht->table[index];
    ht->table[index] = entry;
    ht->used++;

    dictSetKey(d, entry, key);
    dictSetVal(d, entry, val_robj);

    return entry;
}
```

下面我们来看一下一条set命令是怎么持久化的，set命令最终会交由setGenericCommand实现：

```
void setGenericCommand(client *c, int flags, robj *key, robj *val, robj *expire, int unit, robj *
  long long milliseconds = 0; /* initialized to avoid any harmness warning */
#ifdef USE_NVML
    robj* newVal = 0;
#endif
    if (expire) {
        if (getLongLongFromObjectOrReply(c, expire, &milliseconds, NULL) != C_OK)
            return;
        if (milliseconds <= 0) {
            addReplyErrorFormat(c,"invalid expire time in %s",c->cmd->name);
            return;
        }
        if (unit == UNIT_SECONDS) milliseconds *= 1000;
    }

    if ((flags & OBJ_SET_NX && lookupKeyWrite(c->db,key) != NULL) ||
        (flags & OBJ_SET_XX && lookupKeyWrite(c->db,key) == NULL))
    {
        addReply(c, abort_reply ? abort_reply : shared.nullbulk);
        return;
    }
#ifdef USE_NVML
    if (server.persistent) {
        int error = 0;
        // 开始事务
        /* Copy value from RAM to PM - create RedisObject and sds(value) */
        TX_BEGIN(server.pm_pool) {
            //  Duplicate a string object. New object is created in PM
            //  The resulting object always has refcount set to 1.
            newVal = dupStringObjectPM(val);// newval是存在NVM中了
            /* Set key in PM - create DictEntry and sds(key) linked to RedisObject with value
             * Don't increment value "ref counter" as in normal process. */
            setKeyPM(c->db,key,newVal);
        } TX_ONABORT {
```

```
            error = 1;
        } TX_END

        if (error) {
            addReplyError(c, "setting key in PM failed!");
            return;
        }
    } else {
        setKey(c->db,key,val);
    }
#else
    setKey(c->db,key,val);
#endif
    server.dirty++;
    if (expire) setExpire(c->db,key,mstime()+milliseconds);
    notifyKeyspaceEvent(NOTIFY_STRING,"set",key,c->db->id);
    if (expire) notifyKeyspaceEvent(NOTIFY_GENERIC,
        "expire",key,c->db->id);
    addReply(c, ok_reply ? ok_reply : shared.ok);
}
```

dupStringObjectPM

```
/* Duplicate a string object. New object is created in PM
 * The resulting object always has refcount set to 1. */
robj *dupStringObjectPM(robj *o) {
    robj *d;

    serverAssert(o->type == OBJ_STRING);

    switch(o->encoding) {
    case OBJ_ENCODING_RAW:
    case OBJ_ENCODING_EMBSTR:
        // 创建一个EMBSTR类型的SDS
        return createRawStringObjectPM(o->ptr,sdslen(o->ptr));
    case OBJ_ENCODING_INT:
        // 创建一个INT编码类型的SDS
        d = createObjectPM(OBJ_STRING, NULL);
        d->encoding = OBJ_ENCODING_INT;
        d->ptr = o->ptr;
        return d;
    default:
        serverPanic("Wrong encoding.");
        break;
    }
}
```

createObjectPM实现如下：

```
robj *createObjectPM(int type, void *ptr) {
    robj *o = zmalloc(sizeof(*o));

    o->type = type;
    o->encoding = OBJ_ENCODING_RAW;
    o->ptr = ptr;
    o->refcount = 1;

    /* Set the LRU to the current lruclock (minutes resolution). */
    o->lru = LRU_CLOCK();
    return (robj *)o;
}
```

createRawStringObjectPM实现如下：

```
/* Create a string object with encoding OBJ_ENCODING_RAW, that is a plain
 * string object where o->ptr points to a proper sds string.
 * Located in PM */
robj *createRawStringObjectPM(const char *ptr, size_t len) {
    return createObjectPM(OBJ_STRING,sdsnewlenPM(ptr,len));
}
```

sdsnewlenPM用于在NVM中分配一个SDS结构内存：

```
sds sdsnewlenPM(const void *init, size_t initlen) {
    void *sh;
    sds s;
    char type = sdsReqType(initlen);// 根据字符串长度来确定SDS类型
    PMEMoid oid;
    /* Empty strings are usually created in order to append. Use type 8
     * since type 5 is not good at this. */
    if (type == SDS_TYPE_5 && initlen == 0) type = SDS_TYPE_8;
    int hdrlen = sdsHdrSize(type);// 根绝SDS类型获取头部长度
    unsigned char *fp; /* flags pointer. */

    hdrlen += sizeof(PMEMoid);// 在header后面多申请一个持久指针

    // allocates a new zeroed object
    // PMEMoid pmemobj_tx_zalloc(size_t size, uint64_t type_num)
    oid = pmemobj_tx_zalloc((hdrlen+initlen+1),PM_TYPE_SDS);
    // 获得直接指针
    sh = pmemobj_direct(oid);

    if (!init)
        memset(sh, 0, hdrlen+initlen+1);
    if (sh == NULL) return NULL;
    s = (char*)sh+hdrlen;
    fp = ((unsigned char*)s)-1;
    switch(type) {
        case SDS_TYPE_5: {
```

```
            *fp = type | (initlen << SDS_TYPE_BITS);
            break;
        }
        case SDS_TYPE_8: {
            SDS_HDR_VAR(8,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_16: {
            SDS_HDR_VAR(16,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_32: {
            SDS_HDR_VAR(32,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_64: {
            SDS_HDR_VAR(64,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
    }
    if (initlen && init)
        memcpy(s, init, initlen);
    s[initlen] = '\0';
    return s;
}

// 从SDS中取出携带的持久指针
PMEMoid *sdsPMEMoidBackReference(sds s)
{
    void *p;
    p = (u_char *)s - sdsHdrSize(s[-1]) - sizeof(PMEMoid);
    return (PMEMoid *)p;
}
```

对象分配好了之后，setKeyPM将其设置到数据库中：

```
/* High level Set operation. Used for PM */
void setKeyPM(redisDb *db, robj *key, robj *val) {
    if (lookupKeyWrite(db,key) == NULL) {
        dbAddPM(db,key,val);// 源数据库中不存在，就添加
    } else {
```

```
        dbOverwritePM(db,key,val);// 否则就覆盖
    }
    /* TODO: incrRefCount(val); */
    removeExpire(db,key);
    signalModifiedKey(db,key);
}
```

dbAddPM如下：

```
/*
 * Add the key to the DB using libpmemobj transactions.
 */
void dbAddPM(redisDb *db, robj *key, robj *val) {
    PMEMoid kv_PM;
    PMEMoid *kv_pm_reference;// 存储key的持久指针
    // 在NVM中分配key对应的SDS
    sds copy = sdsdupPM(key->ptr, (void **) &kv_pm_reference);
    int retval = dictAddPM(db->dict, copy, val);// 添加到dict
    // 添加到根对象的链表中
    kv_PM = pmemAddToPmemList((void *)copy, (void *)(val->ptr));
    *kv_pm_reference = kv_PM;

    serverAssertWithInfo(NULL,key,retval == C_OK);
    if (val->type == OBJ_LIST) signalListAsReady(db, key);
    if (server.cluster_enabled) slotToKeyAdd(key);
 }
```

sdsdupPM实现如下：

```
/* Duplicate an sds string. */
sds sdsdupPM(const sds s, void **oid_reference) {
    sds new_sds;
    // 在NVM中分配一个新的SDS，里面包含它的持久指针（在header之后）
    new_sds = sdsnewlenPM(s, sdslen(s));
    // 获取SDS里面的持久指针
    *oid_reference = (void *)sdsPMEMoidBackReference(new_sds);
    return new_sds;
}
```

pmemAddToPmemList 用于向链表的根节点中添加一个节点：

```
PMEMoid
pmemAddToPmemList(void *key, void *val)
{
    PMEMoid key_oid;
    PMEMoid val_oid;
    PMEMoid kv_PM;
    struct key_val_pair_PM *kv_PM_p;// key_val_pair_PM的直接指针
    TOID(struct key_val_pair_PM) typed_kv_PM;// 定义key_val_pair_PM
```

```
    struct redis_pmem_root *root;// 根对象

    // 设置key持久指针
    key_oid.pool_uuid_lo = server.pool_uuid_lo;
    key_oid.off = (uint64_t)key - (uint64_t)server.pm_pool->addr;// 偏移
    // 设置value指针
    val_oid.pool_uuid_lo = server.pool_uuid_lo;
    val_oid.off = (uint64_t)val - (uint64_t)server.pm_pool->addr;
    // 分配链表节点
    kv_PM = pmemobj_tx_zalloc(sizeof(struct key_val_pair_PM), pm_type_key_val_pair_PM);
    kv_PM_p = (struct key_val_pair_PM *)pmemobj_direct(kv_PM);
    kv_PM_p->key_oid = key_oid;
    kv_PM_p->val_oid = val_oid;
    typed_kv_PM.oid = kv_PM;

    root = pmemobj_direct(server.pm_rootoid.oid);// 根对象

    kv_PM_p->pmem_list_next = root->pe_first;
    if(!TOID_IS_NULL(root->pe_first)) {
        D_RW(root->pe_first)->pmem_list_prev = typed_kv_PM;
    }
    // 加入到链表中
    root->pe_first = typed_kv_PM;
    // 计数加1
    root->num_dict_entries++;

    return kv_PM;
}
```

对于key已经存在的请，dbOverwritePM将被调用：

```
void dbOverwritePM(redisDb *db, robj *key, robj *val) {
    dictEntry *de = dictFind(db->dict,key->ptr);

    serverAssertWithInfo(NULL,key,de != NULL);
    dictReplacePM(db->dict, key->ptr, val);
}
```

dictReplacePM实现如下：

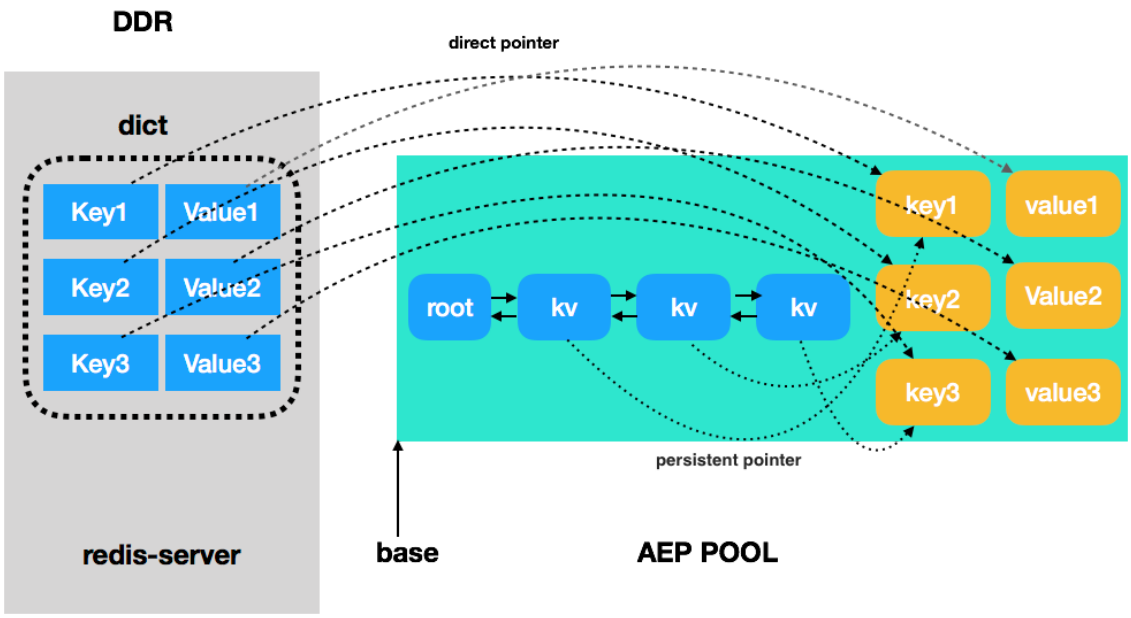
```
int dictReplacePM(dict *d, void *key, void *val)
{
    dictEntry *entry, auxentry;

    /* Try to add the element. If the key
     * does not exists dictAdd will suceed. */
    if (dictAddPM(d, key, val) == DICT_OK)
        return 1;
    /* It already exists, get the entry */
    entry = dictFind(d, key);
    /* Set the new value and free the old one. Note that it is important
```

```
     * to do that in this order, as the value may just be exactly the same
     * as the previous one. In this context, think to reference counting,
     * you want to increment (set), and then decrement (free), and not the
     * reverse. */
    auxentry = *entry;
    dictSetVal(d, entry, val);// 添加到dict
    pmemKVpairSet(entry->key, ((robj *)val)->ptr);// 替换链表中的元素
    dictFreeVal(d, &auxentry);
    return 0;
}
```

pmemKVpairSet就是替换链表中已经存在的元素：

```
void pmemKVpairSet(void *key, void *val)
{
    PMEMoid *kv_PM_oid;
    PMEMoid val_oid;
    struct key_val_pair_PM *kv_PM_p;
    // 得到key对应的持久指针
    kv_PM_oid = sdsPMEMoidBackReference((sds)key);
    kv_PM_p = (struct key_val_pair_PM *)pmemobj_direct(*kv_PM_oid);
    //
    val_oid.pool_uuid_lo = server.pool_uuid_lo;
    val_oid.off = (uint64_t)val - (uint64_t)server.pm_pool->addr;
    // 更新value
    kv_PM_p->val_oid = val_oid;
    return;
}
```



PMDK NVM 事务 源码分析