**Class**

**概念**

**TLS:** 如果需要在一个线程内部的各个[函数调用](http://baike.baidu.com/view/2369016.htm" \t "_blank)都能访问、但其它线程不能访问的[变量](http://baike.baidu.com/view/296689.htm)（被称为static memory local to a thread 线程局部[静态变量](http://baike.baidu.com/view/675642.htm" \t "_blank)），就需要新的机制来实现。这就是TLS.

**GlobalVar**

defaultMemPool\_space 整数对齐，能装下memorypool

static MemoryPool \*defaultMemPool = defaultMemPool\_space

**Flow**

**内存池初始化：**

Initialization()

lock

initMemoryManager()

defaultMemPool->extMemPool.init()

initTLS();

tlsPointerKey.init()

TLS\_pointer\_key = TlsAlloc();//获得线程独占内存索引变量，

为后续线程独占内存创建获取做准备

loc.init() //大对象内存初始化，获取extpool

backend.init() //

usedAddrRange.init() //

coalescQ.init(&bkndSync) //

bkndSync.init(this) //

MemoryPool::initDefaultPool()

hugePages.init(hugePageSize); //大页初始化

unlock

**内存申请函数：**

internalPoolMalloc()

TLSData \*tls = memPool->getTLS(/\*create=\*/true); //获取线程独占内存

if (size >= minLargeObjectSize) //判断是否走大对象流程

return memPool->getFromLLOCache() //大对象内存申请流程

else

bin = tls->getAllocationBin(size); //从TLS中获取符合当前大

小的bin

for( mallocBlock = bin->getActiveBlock(); mallocBlock;

mallocBlock = bin->setPreviousBlockActive() ) //找一个不空的block

{

if( FreeObject \*result = mallocBlock->allocate() )//从当前block里申请

内存

return result;

}

**//如果没有不空的block，该bin里所有block都是空**

mallocBlock = bin->getPublicFreeListBlock(); //获取Freelist里面的block

if (mallocBlock){ //可以成功获取

if (mallocBlock->emptyEnoughToUse()) { //这个block足够使用

bin->moveBlockToBinFront(mallocBlock) //移动这个block到bin

的前端

FreeObject \*result = mallocBlock->allocateFromFreeList()

//从block的freelist中申请内存

**//Freelist里如果也没有block可用的话**

mallocBlock = memPool->orphanedBlocks.get(tls, size);//从孤立块中开辟一块内存交给bin

while (mallocBlock) {

bin->pushTLSBin(mallocBlock); //把这块内存添加到bin中

bin->setActiveBlock(mallocBlock); // 设置成激活块

if( FreeObject \*result = mallocBlock->allocate() )

return result; //从新块中申请内存，返回

}

**//如果孤立块中也没有内存可用的话**

mallocBlock = memPool->getEmptyBlock(size); //获取空的未初始化的一块内存

bin->pushTLSBin(mallocBlock);

bin->setActiveBlock(mallocBlock);

if( FreeObject \*result = mallocBlock->allocate() ) //内存申请部分与之前相同/

**//空块都申请不出来的话。。。**

return *NULL*; //内存不足或者什么原因了

**大对象申请流程：**

MemoryPool::getFromLLOCache(TLSData\* tls, *size\_t* size, *size\_t* alignment)

*size\_t* headersSize = sizeof(LargeMemoryBlock)+sizeof(LargeObjectHdr);//头大小

*size\_t* allocationSize = LargeObjectCache::alignToBin(size+headersSize+alignment);

//块对其大小

LargeMemoryBlock \*lmb = *NULL*;

lmb = tls->lloc.get(allocationSize); //获取大内存块

lmb = extMemPool.mallocLargeObject(this, allocationSize);//如果获取不到就申请一块

//大对象对齐算法，增加size

输出最后申请的空间（具体算法后续补充）

**空内存块申请并初始化流程：**

memPool->getEmptyBlock(size);

TLSData\* tls = extMemPool.tlsPointerKey.getThreadMallocTLS();//通过Key获取TLSData

FreeBlockPool::ResOfGet resOfGet = tls->freeSlabBlocks.getBlock() //从slabblock中获取blocks

Block \*result = resOfGet.block; //得到block

//没找到block的话

if (!result) {

result = static\_cast<Block\*>(extMemPool.backend.getSlabBlock(num));//从后台获取slabblock

// resources were allocated, register blocks注册block

result->initEmptyBlock(tls, size); //初始化block

**init FreeObject list 申请freeObject**

nextPrivatizable ， 设置block头 bin 中block数组中的首地址

**Struct**

MemoryPool:线程池主体，对外接口类，主要包括：ExtMemoryPool(外部线程池),TLSData(线程独占内存数据)。

ExtMemoryPool:外部线程池，主要包括三部分：Backend，TLSKey，LargeObjectCache。

TLSKey：线程独占内存索引信息，方便从线程中获取和创建独占内存。

Backend：

LargeObjectCache：大对象缓存，包括largeCache(大对象内存)，HugeCache(巨大对象内存)，这两个对象解构一致，都是LargeObjectCacheImpl。

LargeObjectCacheImpl:LargeObjectCache组成实体。负责管理一个CacheBin数组。提供put largememoryBlock方法，通过block大小计算出这个block所在bin的索引，然后put到对应bin中。

CacheBin：相同大小的双向链表，为什么使用2-link list。这种链表允许我们减少内存消耗，在解锁的时候可以做更少的操作，管理了大量的LargeMemoryBlock。

LargeMemoryBlock：大对象内存块