健壮的 C++：对象池

格雷格·乌塔斯

2020 年 9 月 7 日 GPL3

评价我： 4.82/5 (16 票)

从内存泄漏中恢复

当系统长时间运行时，使用共享堆最终会因为逐渐碎片化和内存泄漏而导致崩溃。通过从固定大小的块池中分配对象，系统可以限制碎片并使用后台垃圾收集器来恢复泄漏的块。本文介绍了一个提供这些功能的 ObjectPool 类。

下载源 - 8.2 MB

从存储库下载

介绍

一个需要持续可用的系统必须从内存泄漏中恢复，这样它就不需要为了“日常维护”而定期关闭。本文介绍对象池如何帮助满足此要求。在这里，对象池也不会指的是永远不会被销毁共享对象池。相反，它指的是从固定大小的块池而不是堆分配内存的对象。

背景

在 C++ 中，从堆分配的对象，默认实现`operator new`，必须显式删除以将其内存返回到堆。不发生删除时会导致内存泄漏，通常是因为指向对象的指针丢失。C++11 中智能指针的添加降低了这种风险，但它仍然存在（`unique_ptr`例如，如果 a被践踏）。

另一个风险是内存碎片。随着不同大小的内存块从堆中分配并返回到堆中，空洞就会产生。在很长一段时间内，这会减少有效可用内存的数量，除非堆管理器花时间合并相邻的空闲区域并实施最佳匹配策略。

对象池的使用允许系统从内存泄漏中恢复并避免不断升级的碎片。正如我们将看到的，它还支持其他一些有用的功能。

使用代码

与Robust C++: Safety Net 一样，我们将查看的代码取自Robust Services Core (RSC)。但这一次，代码更易于复制到现有项目中并进行修改以满足其需求。如果这是您第一次阅读有关 RSC 某个方面的文章，请花几分钟时间阅读本前言。

课程概览

与许多技术不同，通常可以将对象池引入大型遗留系统，而无需进行大量的重新设计。原因是分配和释放对象的内存是由运算符`new`和封装的`delete`。通过调整类层次结构，这些使用默认堆的运算符的常用版本可以轻松地替换为使用对象池的版本。

对象池

ObjectPool是对象池的基类，它的每个子类都是一个实现一个池的单例。除了ObjectPool带有适当参数的invoke构造函数之外，这些子类几乎没有其他作用。每个子类在创建时都会被添加到一个ObjectPoolRegistry跟踪所有系统对象池的对象中。每个池中的块在系统初始化期间分配，以便系统在启动并运行后可以专注于其工作。

汇集

Pooled是内存来自池的对象的基类。delete当一个对象被删除时，它会覆盖操作符以将一个块返回到它的池中。它还定义了池用于管理每个块的一些数据成员。

对象池审计

ObjectPoolAudit是一个线程，它定期唤醒以调用一个函数，该函数查找孤立块并将其返回到池中。应用程序仍然期望delete对象，因此审计存在以修复可能逐渐导致系统内存不足的内存泄漏。审计使用典型的标记和清除策略，但可以称为后台垃圾收集器，而不是前台垃圾收集器。它的运行频率低于常规垃圾收集器，而不会在执行其工作时将系统冻结很长一段时间。

演练

本文中的代码已被编辑以删除会分散中心概念的内容。这些东西在某些应用程序中很重要，而在其他应用程序中则不那么重要。如果您查看代码的完整版本，您会遇到它们，因此已删除代码中提供了删除内容的摘要。

创建对象池

的每个单例子类ObjectPool调用其基类构造函数来创建其池：

C++ 复制代码

```
ObjectPool::ObjectPool(ObjectPoolId pid, size_t size, size_t segs) :
   blockSize_(0),
   segIncr_(0),
   segSize_(0),
   currSegments_(0),
   targSegments_(segs),
   corruptQHead_(false)
{
   //  The block size must account for the header above each Pooled object.
   //
   blockSize_ = BlockHeaderSize + Memory::Align(size);
   segIncr_ = blockSize_ >> BYTES_PER_WORD_LOG2;
   segSize_ = segIncr_ * ObjectsPerSegment;
   for(auto i = 0; i < MaxSegments; ++i) blocks_[i] = nullptr;

   //  Initialize the pool's free queue of blocks.
   //
   freeq_.Init(Pooled::LinkDiff());

   //  Set the pool's identifier and add it to the registry of object pools.
   //
   pid_.SetId(pid);
   Singleton< ObjectPoolRegistry >::Instance()->BindPool(*this);
}
```

Memory::Align将每个块与底层平台的字长（32 或 64 位）对齐。与堆类似，对象池需要一些数据来管理其块。它BlockHeader在每个块的开头放置一个：

C++ 复制代码

```
//  The header for a Pooled (a block in the pool).  Data in the header
//  survives when an object is deleted.
```

```
//
struct BlockHeader
{
   ObjectPoolId pid : 8;       // the pool to which the block belongs
   PooledObjectSeqNo seq : 8;  // the block's incarnation number
};

const size_t BlockHeader::Size = Memory::Align(sizeof(BlockHeader));

//  This struct describes the top of an object block for a class that
//  derives from Pooled.
//
struct ObjectBlock
{
   BlockHeader header;  // block management information
   Pooled obj;          // the actual location of the object
};

constexpr size_t BlockHeaderSize = sizeof(ObjectBlock) - sizeof(Pooled);
```

作为参考，以下Pooled是与本文相关的成员：

C++ 缩小▲ 复制代码

```
//  A pooled object is allocated from an ObjectPool created during system
//  initialization rather than from the heap.
//
class Pooled : public Object
{
   friend class ObjectPool;
public:
   //  Virtual to allow subclassing.
   //
   virtual ~Pooled() = default;

   //  Returns the offset to link_.
   //
   static ptrdiff_t LinkDiff();

   //  Overridden to claim blocks that this object owns.
   //
   void ClaimBlocks() override;

   //  Clears the object's orphaned_ field so that the object pool audit
   //  will not reclaim it.  May be overridden, but the base class version
   //  must be invoked.
   //
   void Claim() override;

   //  Overridden to return a block to its object pool.
   //
   static void operator delete(void* addr);

   //  Deleted to prohibit array allocation.
   //
   static void* operator new[](size_t size) = delete;
protected:
   //  Protected because this class is virtual.
   //
   Pooled();
private:
   //  Link for queueing the object.
   //
   Q1Link link_;

   //  True if allocated for an object; false if on free queue.
   //
   bool assigned_;
```

```
   //  Zero for a block that is in use.  Incremented each time through the
   //  audit; if it reaches a threshold, the block is deemed to be orphaned
   //  and is recovered.
   //
   uint8_t orphaned_;

   //  Used by audits to avoid invoking functions on a corrupt block.  The
   //  audit sets this flag before it invokes any function on the object.
   //  If the object's function traps, the flag is still set when the audit
   //  resumes execution, so it knows that the block is corrupt and simply
   //  recovers it instead of invoking its function again.  If the function
   //  returns successfully, the audit immediately clears the flag.
   //
   bool corrupt_;

   //  Used to avoid double logging.
   //
   bool logged_;
};
```

构造函数`freeq_`为尚未分配的块初始化了一个队列 ( )。RSC 提供了两个队列模板，`Q1Way`以及`Q2Way`. 它们的实现与 STL 队列的不同之处在于它们在系统初始化后从不分配内存。相反，其对象将被排队的类`ptrdiff_t`向数据成员提供偏移量，该数据成员用作到队列中下一项的链接。该偏移量是`freeq_.Init()`.

一个关键的设计决策是池的数量。推荐的方法是为从同一主要框架类派生的所有类使用公共池。`NodeBase`例如，RSC为用于线程间消息传递的缓冲区定义了一个对象池。请注意，池块的大小必须大于例如，`sizeof(MsgBuffer)`以便子类有空间添加自己的数据。

对从同一框架类派生的所有类使用公共池可以显着简化池大小的工程。当系统达到最大值时，每个池必须有足够的块来处理峰值负载时间。如果每个子类都有自己的池，那么每个池都需要足够的块来处理该子类恰好特别受欢迎的时间。让子类共享一个池可以消除这种波动，减少所需的块总数。

增加对象池的大小

无论池是在系统初始化期间创建其初始块池，还是在服务时分配更多块，代码都是相同的：

C++ 缩小▲ 复制代码

```
bool ObjectPool::AllocBlocks()
{
   auto pid = Pid();

   while(currSegments_ < targSegments_)
   {
      //  Allocate memory for the next group of blocks.
      //
      auto size = sizeof(uword) * segSize_;
      blocks_[currSegments_] = (uword*) Memory::Alloc(size, false);
      if(blocks_[currSegments_] == nullptr) return false;
      ++currSegments_;
      totalCount_ = currSegments_ * ObjectsPerSegment;

      //  Initialize each block and add it to the free queue.
      //
      auto seg = blocks_[currSegments_ - 1];

      for(size_t j = 0; j < segSize_; j += segIncr_)
      {
         auto b = (ObjectBlock*) &seg[j];
         b->header.pid = pid;
         b->header.seq = 0;
         b->obj.link_.next = nullptr;
         b->obj.assigned_ = false;
         b->obj.orphaned_ = OrphanThreshold;
         EnqBlock(&b->obj);
      }
   }
```

```
   return true;
}
```

请注意，块被分配在每个 1K 块的段中，这样单个块由 寻址`blocks_[i][j]`。这使得在系统运行时分配更多块变得容易：只需添加另一个段。由于稍后将出现的其他原因，段中的块是连续的这一事实是有用的。

创建池对象

让我们把注意力转向创建一个驻留在 an`ObjectPool`块中的对象。这是以通常的方式完成的，但调用`new`最终会调用使用池的框架类中的实现：

C++ 复制代码

```
void* MsgBuffer::operator new(size_t size)
{
   return Singleton< MsgBufferPool >::Instance()->DeqBlock(size);
}
```

也许您在`Pooled`所有池对象的基类中注意到了这一点：

C++ 复制代码

```
static void* operator new[](size_t size) = delete;
```

这禁止分配池对象数组，这必须通过找到一组相邻的空闲块并将每个块从空闲队列中碰巧位于的任何位置删除来实现。经过。

每个基类的实现调用的代码`operator new`如下所示：

C++ 复制代码

```
Pooled* ObjectPool::DeqBlock(size_t size)
{
   //  Check that the object will fit into the block.
   //
   auto maxsize = blockSize_ - BlockHeader::Size;

   if(size > maxsize)
   {
      throw AllocationException(size);
   }

   auto item = freeq_.Deq();

   if(item == nullptr)
   {
      throw AllocationException(size);
   }

   --availCount_;
   return item;
}
```

当一个对象不适合其池的块时，最简单的解决方案是增加块的大小。少量增加应该是可以容忍的，但是如果一个类的对象比使用池的其他对象大得多，则可能会浪费太多内存。在这种情况下，违规类必须使用PIMPL 习惯用法将其某些数据移动到`private`对象中。默认情况下，这个对象将从堆中分配。然而，也可以为此目的提供辅助数据块。它们也使用对象池并有各种大小，例如小型、中型和大型。它们实施起来并不困难，因此 RSC 最终会将它们包括在内。

删除池对象

当`delete`在池对象上调用时，它最终会找到它的方式：

C++ 复制代码

```
void Pooled::operator delete(void* addr)
{
   auto obj = (Pooled*) addr;
   auto pid = ObjectPool::ObjPid(obj);
   auto pool = Singleton< ObjectPoolRegistry >::Instance()->Pool(pid);
   if(pool != nullptr) pool->EnqBlock(obj, true);
}
```

在这里，从 中`ObjectPool::ObjPid`获取对象池的标识符`BlockHeader.pid`，该标识符出现在较早且位于 的正上方`obj`。这允许操作员将块返回到正确的池：

C++ 缩小▲ 复制代码

```
void ObjectPool::EnqBlock(Pooled* obj)
{
   if(obj == nullptr) return;

   //  If a block is already on the free queue or another queue, putting it
   //  on the free queue creates a mess.
   //
   if(!obj->assigned_)
   {
      if(obj->orphaned_ == 0) return;  // already on free queue
   }
   else if(obj->link_.next != nullptr) return;  // on some other queue

   //  Trample over some or all of the object.
   //
   auto nullify = ObjectPoolRegistry::NullifyObjectData();
   obj->Nullify(nullify ? blockSize_ - BlockHeader::Size : 0);

   //  Return the block to the pool.
   //
   auto block = ObjToBlock(obj);

   if(block->header.seq == MaxSeqNo)
      block->header.seq = 1;
   else
      ++block->header.seq;

   obj->link_.next = nullptr;
   obj->assigned_ = false;
   obj->orphaned_ = 0;
   obj->corrupt_ = false;
   obj->logged_ = false;

   if(!freeq_.Enq(*obj)) return;
   ++availCount_;
}
```

请注意，我们在将对象返回到池中之前践踏了该对象。践踏的数量由读入的配置参数决定`nullify`。这允许`0xfd`在测试期间用字节之类的东西填充整个块。在已发布的软件中，只需踩踏物体的顶部即可节省时间。这类似于许多堆的“调试”版本。这个想法是，通过践踏对象，更可能检测到陈旧的访问（对已删除的数据）。即使我们只是践踏了对象的顶部，我们`vptr`也会破坏它的，如果稍后有人试图调用它的`virtual`函数之一，这将导致异常。

审计池

早些时候，我们注意到它`ObjectPoolAudit`使用标记和清除策略来查找孤立块并将它们返回到它们的池中。它的代码很简单，因为它`ObjectPoolRegistry`实际上拥有所有的对象池。因此，线程只是定期唤醒并告诉注册表审核池：

C++ 复制代码

```
void ObjectPoolAudit::Enter()
{
   while(true)
```

```
   {
      Pause(interval_);
      Singleton< ObjectPoolRegistry >::Instance()->AuditPools();
   }
}
```

审计分为三个不同的阶段。phase_正在审计的池的当前和标识符（pid_）是ObjectPoolAudit其自身的成员，但由以下代码使用：

C++ 缩小▲ 复制代码

```
void ObjectPoolRegistry::AuditPools() const
{
   auto thread = Singleton< ObjectPoolAudit >::Instance();

   //  This code is stateful.  When it is reentered after an exception, it
   //  resumes execution at the phase and pool where the exception occurred.
   //
   while(true)
   {
      switch(thread->phase_)
      {
      case ObjectPoolAudit::CheckingFreeq:
         //
         //  Audit each pool's free queue.
         //
         while(thread->pid_ <= ObjectPool::MaxId)
         {
            auto pool = pools_.At(thread->pid_);

            if(pool != nullptr)
            {
               pool->AuditFreeq();
               ThisThread::Pause();
            }

            ++thread->pid_;
         }

         thread->phase_ = ObjectPoolAudit::ClaimingBlocks;
         thread->pid_ = NIL_ID;
         //  [[fallthrough]]

      case ObjectPoolAudit::ClaimingBlocks:
         //
         //  Claim in-use blocks in each pool.  Each ClaimBlocks function
         //  finds its blocks in an application-specific way.  The blocks
         //  must be claimed after *all* blocks, in *all* pools, have been
         //  marked, because some ClaimBlocks functions claim blocks from
         //  multiple pools.
         //
         while(thread->pid_ <= ObjectPool::MaxId)
         {
            auto pool = pools_.At(thread->pid_);

            if(pool != nullptr)
            {
               pool->ClaimBlocks();
               ThisThread::Pause();
            }

            ++thread->pid_;
         }

         thread->phase_ = ObjectPoolAudit::RecoveringBlocks;
         thread->pid_ = NIL_ID;
         //  [[fallthrough]]

      case ObjectPoolAudit::RecoveringBlocks:
```

```
         //
         //  For each object pool, recover any block that is still marked.
         //  Such a block is an orphan that is neither on the free queue
         //  nor in use by an application.
         //
         while(thread->pid_ <= ObjectPool::MaxId)
         {
            auto pool = pools_.At(thread->pid_);

            if(pool != nullptr)
            {
               pool->RecoverBlocks();
               ThisThread::Pause();
            }

            ++thread->pid_;
         }

         thread->phase_ = ObjectPoolAudit::CheckingFreeq;
         thread->pid_ = NIL_ID;
         return;
      }
   }
}
```

审计可用块的队列

在第一阶段，AuditPools调用每个池的AuditFreeq函数。该函数首先将池中的所有块标记为孤立块，然后声明已经在空闲队列中的块。

损坏的队列很可能会导致连续的异常。因此，AuditFreeq需要检测空闲队列是否损坏，如果损坏，则修复它：

C++ 缩小▲ 复制代码

```
void ObjectPool::AuditFreeq()
{
   size_t count = 0;

   //  Increment all orphan counts.
   //
   for(size_t i = 0; i < currSegments_; ++i)
   {
      auto seg = blocks_[i];

      for(size_t j = 0; j < segSize_; j += segIncr_)
      {
         auto b = (ObjectBlock*) &seg[j];
         ++b->obj.orphaned_;
      }
   }

   //  Audit the free queue unless it is empty.  The audit checks that
   //  the links are sane and that the count of free blocks is correct.
   //
   auto diff = Pooled::LinkDiff();
   auto item = freeq_.tail_.next;

   if(item != nullptr)
   {
      //  Audit the queue header (when PREV == nullptr), then the queue.
      //  The queue header references the tail element, so the tail is
      //  the first block whose link is audited (when CURR == freeq_).
      //  The next block to be audited (when PREV == freeq_) is the head
      //  element, which follows the tail.  The entire queue has been
      //  traversed when CURR == freeq_ (the tail) for the second time.
      //
      //  Before a link (CURR) is followed, the item (queue header or
```

```
      //  block) that provided the link is marked as corrupt.  If the
      //  link is bad, a trap should occur at curr->orphaned_ = 0.
      //  Thus, if we get past that point in the code, the link should
      //  be sane, and so its owner's "corrupt" flag is cleared before
      //  continuing down the queue.
      //
      //  If a trap occurs, this code is reentered.  It starts traversing
      //  the queue again.  Eventually it reaches an item whose corrupt_
      //  flag *is already set*, at which point the queue gets truncated.
      //
      Pooled* prev = nullptr;
      auto badLink = false;

      while(count <= totalCount_)
      {
         auto curr = (Pooled*) getptr1(item, diff);

         if(prev == nullptr)
         {
            if(corruptQHead_)
               badLink = true;
            else
               corruptQHead_ = true;
         }
         else
         {
            if(prev->corrupt_)
               badLink = true;
            else
               prev->corrupt_ = true;
         }

         //  CURR has not been claimed, so it should still be marked as
         //  orphaned (a value in the range 1 to OrphanThreshold).  If it
         //  isn't, PREV's link must be corrupt.  PREV might be pointing
         //  back into the middle of the queue, or it might be a random
         //  but legal address.
         //
         badLink = badLink ||
            ((curr->orphaned_ == 0) || (curr->orphaned_ > OrphanThreshold));

         //  If a bad link was detected, truncate the queue.
         //
         if(badLink)
         {
            if(prev == nullptr)
            {
               corruptQHead_ = false;
               freeq_.Init(Pooled::LinkDiff());
               availCount_ = 0;
            }
            else
            {
               prev->corrupt_ = false;
               prev->link_.next = freeq_.tail_.next;  // tail now after PREV
               availCount_ = count;
            }

            return;
         }

         curr->orphaned_ = 0;
         ++count;

         if(prev == nullptr)
            corruptQHead_ = false;
         else
            prev->corrupt_ = false;
```

```
            prev = curr;
            item = item->next;

            if(freeq_.tail_.next == item) break;  // reached tail again
         }
      }

      availCount_ = count;
   }
```

声明使用中的块

在第二阶段，AuditPools调用每个池的ClaimBlocks函数。正是此功能必须声明使用中的块，以便审计不会恢复它们。因此，池必须能够从池中找到可以拥有对象的所有对象，以便每个所有者都可以声明其对象。

声明对象的过程是系统对象模型的级联过程。早些时候，我们注意到NodeBase有一个用于线程间消息传递的缓冲区池。以下代码片段说明了如何声明使用中的缓冲区。

线程拥有用于线程间消息传递的缓冲区，因此缓冲区的池ClaimBlocks通过告诉每个线程声明其对象来实现。为此，它通过ThreadRegistry跟踪系统的所有线程的 。这将启动级联：

C++ 复制代码

```
void MsgBufferPool::ClaimBlocks()
{
   Singleton< ThreadRegistry >::Instance()->ClaimBlocks();
}
```

C++ 复制代码

```
void ThreadRegistry::ClaimBlocks()
{
   //  Have all threads mark themselves and their objects as being in use.
   //
   for(auto t = threads_.First(); t != nullptr; threads_.Next(t))
   {
      t->ClaimBlocks();
   }
}
```

ClaimBlocks对线程的调用很快就会达到以下内容，其中该线程声明对它排队的任何消息缓冲区：

C++ 复制代码

```
void Thread::Claim()
{
   for(auto m = msgq_.First(); m != nullptr; msgq_.Next(m))
   {
      m->Claim();
   }
}

void Pooled::Claim()
{
   orphaned_ = 0;  // finally!
}
```

恢复孤立块

在第三个也是最后一个阶段，AuditPools调用每个池的RecoverBlocks函数，恢复孤儿。为了防止不可预见的竞争条件，一个块必须在一个以上的审计周期内保持孤立状态，然后才能被回收。这就是常量 的用途OrphanThreshold，其值为2。

C++ 缩小▲ 复制代码

```
void ObjectPool::RecoverBlocks()
{
   auto pid = Pid();

   //  Run through all of the blocks, recovering orphans.
   //
   for(size_t i = 0; i < currSegments_; ++i)
   {
      auto seg = blocks_[i];

      for(size_t j = 0; j < segSize_; j += segIncr_)
      {
         auto b = (ObjectBlock*) &seg[j];
         auto p = &b->obj;

         if(p->orphaned_ >= OrphanThreshold)
         {
            //  Generate a log if the block is in use (don't bother with
            //  free queue orphans) and it hasn't been logged yet (which
            //  can happen if we reenter this code after a trap).
            //
            ++count;

            if(p->assigned_ && !p->logged_)
            {
               auto log = Log::Create(ObjPoolLogGroup, ObjPoolBlockRecovered);

               if(log != nullptr)
               {
                  *log << Log::Tab << "pool=" << int(pid) << CRLF;
                  p->logged_ = true;
                  p->Display(*log, Log::Tab, VerboseOpt);
                  Log::Submit(log);
               }
            }

            //  When an in-use orphan is found, we mark it corrupt and clean
            //  it up.  If it is so corrupt that it causes an exception during
            //  cleanup, this code is reentered and encounters the block again.
            //  It will then already be marked as corrupt, in which case it
            //  will simply be returned to the free queue.
            //
            if(p->assigned_ && !p->corrupt_)
            {
               p->corrupt_ = true;
               p->Cleanup();
            }

            b->header.pid = pid;
            p->link_.next = nullptr;
            EnqBlock(p);
         }
      }
   }
}
```

兴趣点

既然我们已经看到了如何实现一个对象池来修复损坏的空闲队列并恢复泄漏的对象，那么应该提到对象池的一些其他功能。

迭代池化对象

`ObjectPool`为此提供迭代函数。在以下代码片段中，`FrameworkClass`是子类驻留在`pool`'s 块中的类：

C++ 复制代码

```
PooledObjectId id;

for(auto obj = pool->FirstUsed(id); obj != nullptr; obj = pool->NextUsed(id))
{
   auto item = static_cast< FrameworkClass* >(obj);
   //  ...
}
```

验证指向池对象的指针

`ObjectPool`提供一个函数，该函数接受一个指向池对象的指针并返回池中对象的标识符。该标识符是一个介于 1 和池中块数之间的整数。如果指针无效，函数返回`NIL_ID`：

C++ 缩小▲ 复制代码

```
PooledObjectId ObjectPool::ObjBid(const Pooled* obj, bool inUseOnly) const
{
   if(obj == nullptr) return NIL_ID;
   if(inUseOnly && !obj->assigned_) return NIL_ID;

   //  Find BLOCK, which houses OBJ and is the address that we'll look for.
   //  Search through each segment of blocks.  If BLOCK is within MAXDIFF
   //  distance of the first block in a segment, it should belong to that
   //  segment, as long as it actually references a block boundary.
   //
   auto block = (const_ptr_t) ObjToBlock(obj);
   auto maxdiff = (ptrdiff_t) (blockSize_ * (ObjectsPerSegment - 1));

   for(size_t i = 0; i < currSegments_; ++i)
   {
      auto b0 = (const_ptr_t) &blocks_[i][0];

      if(block >= b0)
      {
         ptrdiff_t diff = block - b0;

         if(diff <= maxdiff)
         {
            if(diff % blockSize_ == 0)
            {
               auto j = diff / blockSize_;
               return (i << ObjectsPerSegmentLog2) + j + 1;
            }
         }
      }
   }

   return NIL_ID;
}
```

区分块的化身

`BlockHeader`在**创建对象池中介绍**。`EnqBlock`增加了它的`seq`成员，它作为一个化身数字。这在分布式系统中很有用。假设一个池对象从另一个处理器接收消息，并且它给那个处理器它的`this`指针。该处理器在要传递给对象的每条消息中都包含该指针。然后可能会发生以下情况：

1. 该对象被删除，其块被快速分配给一个新对象。
2. 在另一个处理器获知删除之前，它会向对象发送一条消息。
3. 消息到达并被传递到新对象。

`this`对象应该提供它的`PooledObjectId`（由上述函数返回的）和它的化身 *number*，而不是作为它的消息地址提供。这样，很容易检测到消息是陈旧的并丢弃它。请注意，如果对象未合并，则需要一些其他机制来检测陈旧消息。

一个对象`PooledObjectId`从`ObjectPool::ObjBid`上面获得它。反向映射由 提供 `ObjectPool::BidToObj`，并`ObjectPool::ObjSeq`允许对象获取其化身编号。

更改对象池的大小

每个池的大小由配置参数设置，该参数的值在系统初始化时从文件中读取。这允许自定义每个池的大小，这在服务器中很重要。

当系统运行时，只需使用 CLI 命令`>cfgparms set <parm> <size>`更改其配置参数的值即可增加池的大小。

由于池的使用中块可能是从其 1K 块的任何段中分配的，因此不支持在系统运行时减小池的大小。如果配置参数设置为较低的值，则直到下一次重新分配池的块的重新启动时才会生效。（重新启动是一种比重新启动更不严重的重新初始化：请参阅Robust C++: Initialization and Restarts。）

当可用块的数量低于阈值时，池会发出警报。警报的严重性（次要、主要或关键）由可用块的数量（小于池中总块的1/8th、1/ 16th或 1/32nd）确定。该警报用作警告池的大小可能设计不足。

尽管系统可以在中等数量的分配失败中幸存下来，但不太可能在大量分配失败中幸存下来。因此，当池发出严重警报时，它会自动将其大小增加一个段（1K 块）。

删除的代码

该软件的完整版本包括出于本文目的而被删除的方面[1]：

- **配置参数**，在**更改池的大小中**提到。
- **日志**。该代码在恢复孤立块时生成了日志，但还有许多其他情况也会导致日志。
- **统计**。每个池都提供统计信息，例如从池中分配块的次数以及池中剩余可用块数的低水位线。
- **警报**，在**更改池的大小中**提到。
- **内存类型**。RSC 根据内存是否将被写保护以及它将继续存在的重启级别来定义内存类型。当它调用`ObjectPool`的构造函数时，子类指定用于池块的内存类型。
- **跟踪工具**。大多数函数`Debug::ft`在它们的第一条语句中调用。它支持函数跟踪工具和 Robust C++: Safety Net 中提到的其他东西。还有一个跟踪工具可以记录正在分配、声明和释放的对象块。

笔记

[1] 其中大部分确实是面向方面编程中的方面。

历史

- 2020 年 9 月 7 日：添加关于修改池大小的部分
- 3RD 九月2019：初始版本

执照

本文以及任何相关的源代码和文件均根据GNU 通用公共许可证 (GPLv3)获得许可

分享

关于作者

格雷格·乌塔斯
建筑师
加拿大
手表该会员

着有强大的服务核心（GitHub上）和强大的通信软件（Wiley出版社，2005）。曾任核心网络服务器首席软件架构师，负责处理AT&T 无线网络中的呼叫。

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

第一 页上一页 下一页

从其他池创建的池
Philippe Verdy 13-Jun-20 8:22

回复：从其他池创建的池
Greg Utas 13-Jun-20 20:07

回复：从其他池创建的池
Philippe Verdy 18-Mar-21 16:32

回复：从其他池创建的池
Greg Utas 18-Mar-21 21:36

这真的是个好主意吗?
Member 13301679 12-Jun-20 7:43

回复：这真的是个好主意吗?
Greg Utas 12-Jun-20 8:20

我的4票
SeattleC++ 31-Dec-19 3:21

回复：我投了 4 票
Greg Utas 31-Dec-19 5:28

块头和控制数据保护字节
Blake Miller 7-Sep-19 1:39

回复：块头和控制数据保护字节

Greg Utas 7-Sep-19 5:38

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。