# 无锁队列的详细设计

发布于 2014年11月06日

这篇文章相当详细地概述了我设计的高效无锁队列，支持多个并发生产者和使用者（MPMC队列）。我的C++11 实现可以在 GitHub 上找到。队列的更高级别概述（包括基准结果）可以在介绍我的实现的初始博客文章中找到。

一个关键的见解是，元素从队列中出来的总顺序是无关紧要的，只要元素进入给定线程的顺序与它们在另一个线程上的顺序匹配。这意味着队列可以安全地实现为一组独立的队列，每个生产者一个；编写单生产者、多消费者锁无锁队列比编写多生产者、多消费者队列要容易得多，而且可以更高效地实现。通过让使用者根据需要从不同的 SPMC 队列中撤出，可以将 SPMC 队列推广到 MPMC 队列（这也可以通过一些巧妙的操作高效完成）。在典型情况下，使用启发式技术来加快取消队列，尽可能多地将消费者与生产者配对（这大大减少了系统中的总体争用）。

除了高级 SPMC 队列集设计之外，队列的另一个关键部分是核心队列算法本身，这是全新的（我自己设计的），不像我听说过任何其他算法。它使用原子计数器来跟踪有多少元素可用，一旦一个或多个元素被声称（通过增加相应的元素消耗计数器并检查该增量是否有效），原子

可以安全地递增索引，以获得要引用的元素的实际 ID。然后，问题减少到将整数 ID 映射到单个元素，而不必担心引用相同对象的其他线程（每个 ID 只给一个线程）。详情如下！

# 系统概述

队列由一系列单生产者、多消费者 （SPMC） 队列组成。每个生产者有一个 SPMC 队列;使用者使用启发式方法确定从下一步开始使用这些队列中哪一个。队列是无锁的（虽然不太无等待）。它设计为健壮、速度非常快（特别是在 x86 上），并允许批量进行分水和取消队列，而额外的开销（与单个项目相比）非常小。

每个生产者都需要一些线程本地数据，线程本地数据也可以选择用于加速使用者。此线程本地数据可以与用户分配的令牌关联，或者为了简化接口，如果用户未为生产者提供令牌，则使用无锁哈希表（键到当前线程 ID）查找线程本地生成者队列：为每个显式分配的生产者令牌创建一个 SPMC 队列，并为不提供令牌而生成项的每个线程创建另一个隐式队列。由于令牌包含相当于线程特定数据的内容，因此绝不应同时从多个线程使用令牌（尽管将令牌的所有权转移到另一个线程是可以的;特别是，这允许在线程池任务内使用令牌，即使运行该任务的线程部分更改）。

所有生产者队列将自己链接在一起，并放入一个无锁链接列表中。当显式生产者不再向其添加元素（即其令牌被销毁）时，它将被标记为未关联到任何生产者，但它保留在列表中及其内存中

未释放;下一个新生产者重用旧生产者的内存（无锁生产者列表是仅添加这种方式）。隐式生产者永远不会被销毁（直到高级队列本身），因为无法知道给定线程是否使用数据结构完成。请注意，最坏情况下的取消排队速度取决于有多少生产者队列，即使它们都是空的。

显式生产者队列与隐式生产者队列的生存期存在根本区别：显式队列具有与令牌的生存期绑定的有限生产生存期，而隐式队列具有无边界生存期，并且存在于高级队列本身的持续时间内。因此，使用两种略有不同的 SPMC 算法来最大限度地提高速度和内存使用率。通常，显式生产者队列设计为稍微快一点，占用内存稍快，而隐式生产者队列设计为稍慢，但将更多内存回收回高级队列的全局池。为了获得最佳速度，请始终使用显式令牌（除非您觉得它太不方便）。

只有在销毁高级队列时（尽管存在多个重新使用机制），才释放分配的任何内存。
内存分配可以预先完成，如果内存不足（而不是分配更多），则操作将失败。各种默认大小参数（以及队列使用的内存分配函数）可根据需要由 使用项覆盖。

# 完整 API（伪代码）

* 必要时分配更多内存（项目）：布尔队列

(prod_token，项目) ： 布尔

enqueue_bulk (item_first, 计数) : 布尔enqueue_bulk (prod_token, item_first, 计数) : 布尔

* 如果内存不足，无法try_enqueue (项目) : bool try_enqueue (prod_token, 项目) : bool try_enqueue_bulk (item_first, 计数) : bool try_enqueue_bulk (prod_token, item_first, 计数) : 布尔

# 尝试从队列中取消排队（从不分配）try_dequeue (项目=) : bool try_dequeue (cons_token, 项目 &) : 布尔 try_dequeue_bulk (item_first, 最大) : size_t try_dequeue_bulk (cons_token, item_first, 最大值) : size_t

# 如果您碰巧知道您希望从哪个生产商取消队列 try_dequeue_from_producer (prod_token, 项目&) : 布尔 try_dequeue_bulk_from_producer (prod_token, item_first, 最大值) : size_t

# 元素总数的不一定准确计数 size_approx() : size_t

# 生产者队列 （SPMC） 设计

## 跨隐式和显式版本的共享设计

生产者队列由块组成（显式和隐式生产者队列使用相同的块对象来允许更好的内存共享）。最初，它开始时没有块。每个块可以容纳固定数量的元素（所有块具有相同的容量，即 2 的功率）。此外，块包含一个标志，指示已填充槽是否已完全使用（由显式版本用于确定块何时为空），以及完全取消槽数的元素数的原子计数器（由隐式版本用于确定块何时为空）。

出于无锁操作的目的，生产者队列可视为抽象的无限数组。尾部索引指示生产者要填充的下一个可用插槽;它还加倍作为曾经被重新批批的元素数（队列计数）的计数。尾部指数仅由生产者编写，并且始终在增加（除非它溢出并环绕，这仍被视为"增加"， 就我们的目的而言）。由于只有一个线程正在更新所涉及的变量，因此生成项是微不足道的。头索引指示接下来可以使用哪些元素。头索引由使用者以原子方式递增，可能同时增加。为了防止头部指数到达感知的尾部指数，使用了一个额外的原子计数器：去队列计数。取消队列计数是乐观的，即当消费者推测认为有什么东西可以取消时，它增加了。如果递增后取消队列计数的值小于队列计数（尾数），则保证

至少有一个元素要取消队列（即使考虑到并发），并且可以安全地增加头索引，因为知道之后它将小于尾部索引。另一方面，如果递增后，取消队列计数超过（或等于）尾部，则取消队列操作将失败，并且取消队列计数在逻辑上递减（以保持其最终与队列计数一致）：这可以通过直接递减取消队列计数来完成，但相反（增加并行性并保持涉及的所有变量都单调地增加），则取消队列计数将改为递减。为了获取取消队列计数的逻辑值，我们从取消队列计数变量中减去取消队列过度提交值。

使用时，一旦确定如上文所述的有效索引，它仍然需要映射到块和偏移到该块;某种索引数据结构用于此目的（这取决于它是隐式队列还是显式队列）。
最后，元素可以移出，并更新某种状态，以便最终知道块何时完全使用。下面在涉及隐式和显式特定细节的各个章节中提供了这些机制的完整说明。

如前所述，尾部和头部指数最终将溢出。这是预料之中和考虑的。因此，索引xcount 被视为在最大整数值大小的圆上存在（类似于 360 度的圆，其中 359 在 1 之前）。为了检查一个索引计数，例如 a，是否先于另一个索引计数，例如 b，（即逻辑小于），我们必须通过圆上的顺时针弧确定 a 是否更接近 b。以下用于循环小于使用的算法 （32 位版本）： [b] 变为 a - b = （1u = 31u） 。b 变为 a - b - 1ull = （1ull = 31ull）。请注意，循环减法"只是工作"与正常的无符号整数

（假设两个补充）。注意确保尾部索引不会超过头索引（这将损坏队列）。请注意，尽管如此，在技术上仍有一个竞赛条件，即消费者（或生产者）看到的索引值太陈旧，几乎在其当前值后面是整个圆的价值（或更多），导致队列的内部状态损坏。但是，实际上这不是问题，因为需要一段时间才能通过 2^31 值（对于 32 位索引类型），到那时，其他内核将看到更新的东西。事实上，许多无锁算法都基于相关的标记指针习惯用法，其中前 16 位用于重复递增的标记，而后 16 位用于指针值;这依赖于类似的假设，即一个内核不能增加标记超过2~15倍，而其他内核不知道。但是，队列的默认索引类型是 64 位宽（即使 16 位似乎足够，即使理论上也应该阻止任何潜在的冲突）。

内存分配失败也得到正确处理，永远不会损坏队列（它只是报告为失败）。但是，假定元素本身永远不会在队列操作时引发异常。

## 块池

使用两个不同的块池：首先，有预分配块的初始数组。一旦使用，此池将永远保持空。这通过检查（确保该索引在范围内）将其无等待实现简化为单个提取和添加原子指令（获取自由块的下一个索引）。其次，有一个无锁（虽然不是无等待）全局免费列表（"全局"表示全局到高级队列），这些已使用块已准备好重新使用，实现为无锁单独链接列表：头指针最初指向

无 （空）。要向自由列表添加块，块的下一个指针设置为头部指针，然后将头指针更新为使用比较和交换 （CAS） 指向块，条件是头未更改;如果有，则重复此过程（这是一个经典的无锁 CAS 循环设计模式）。若要从自由列表中删除块，将使用类似的算法：读取头块的下一个指针，然后将头设置为下一个指针（使用 CAS），条件是头在此期间未更改。为了避免 ABA 问题，每个块都有一个引用计数，在使用 CAS 删除块之前递增该计数，并在之后递增;如果尝试在其引用计数高于 0 时重新向自由列表添加块，则设置了指示该块应位于自由列表中的标记，并且完成持有最后一个引用的下一个线程将检查此标志并将该块添加到该列表（这适用于我们不关心顺序）。我已经在另一篇博文中详细描述了这个无锁列表的确切设计和实现。当生产者队列需要一个新的块时，它首先检查初始块池，然后检查全局自由列表，只有当它找不到一个可用块时，它才在堆上分配一个新的块（如果不允许内存分配，则失败）。

## 显式生产者队列

显式生产者队列作为循环单独链接的块列表实现。它在快速路径上是无等待的，但当需要从块池（或分配的新块）获取块时，它仅无锁定;只有当其块的内部缓存都已满（或没有，这在开头就是这种情况时才发生）。

将块添加到显式生产者队列的循环链接列表中后，它永远不会被删除。尾部块指针由创建者维护，该指针指向当前要插入的元素的块;当尾部块已满时，下一个块为

以确定是否为空。如果是，尾块指针将更新为指向该块;如果是，则更新尾块指针以指向该块。如果没有，则将重新征用新块，并立即在当前尾块之后插入链接列表中，然后更新该尾块以指向此新块。

当元素从块中解奎后，将设置每个元素标志，以指示插槽已满。（实际上，所有标志都从设置开始，并且仅在插槽变为空时关闭。生产者通过检查所有这些标志来检查块是否为空。如果块大小较小，则速度足够快;如果块大小较小，则速度足够快。否则，对于较大的块，而不是标志系统，每次使用者完成元素时，都会增加块级原子计数。当此计数等于块的大小，或者所有标志都关闭时，该块为空，可以安全地重新使用。

要在恒定时间对块进行索引（即，在元素的全局索引中从取消队列算法中快速找到元素中的块），使用循环缓冲区（连续数组）。此索引由生产者维护;消费者阅读它， 但从来没有写信给它。数组的前面是最近写入的块（尾部块）。在数组的后面是最后一个可能包含元素的块。将此索引（从高级角度）视为已使用块的历史记录的单一长功能区是很有帮助的。每当生产者在另一个块上启动时，前面都会递增（可能新分配或从其圆形块列表中重新使用）。每当重新使用循环列表中的块时，后部都会递增（因为块仅在空时才重新使用，在这种情况下，增加后部始终是安全的）。保留使用的插槽数（这避免了在圆形缓冲区中对备用元件的需要，并简化了实现），而不是显式存储后部。如果索引中没有足够的空间来添加新项，则分配一个新的索引数组，其大小是上一个数组的两倍（显然，这仅在以下

内存分配是允许的 - 如果没有，则整个 enqueue 操作将正常失败）。由于使用者可能仍在使用旧索引，因此不会释放它，而只是链接到新索引（这形成了一个索引块链，当高级队列被销毁时，可以正确释放）。当生产者队列的排队计数递增时，它将释放对索引的所有写入;当使用者执行获取（它已经需要取消队列算法时），那么从该点到使用者看到哪个索引上，都将包含对使用者感兴趣的块的引用。由于块的大小相同，并且功率为 2，因此我们可以使用移位和掩蔽来确定目标块与索引中任何其他块（以及目标块中的偏移）偏移量，前提是我们知道索引中给定块的基本索引。因此，索引不仅包含块指针，还包含每个块的相应基础索引。在索引中选择作为参考点（计算偏移量）的块在使用时不得被编制者覆盖 -- 使用索引的（感知）前面作为参考点可以保证这一点，因为（知道块索引至少与我们正在查找的取消索引前面的队列计数一样最新），索引的前面必须位于或前面目标块，目标块永远不会在索引中覆盖，直到它（和之前的所有块）为空，并且它不能为空，直到取消队列操作本身完成。索引大小为 2 的功率，允许更快地环绕前极变量。

显式生产者队列要求在排队时传递用户分配的"生产者令牌"。此令牌仅包含指向生产者队列对象的指针。创建令牌时，将创建相应的生产者队列;在创建令牌时，将创建相应的生产者队列。当令牌被销毁时，生产者队列可能仍然包含未使用的元素，因此队列本身的寿命超过令牌。事实上，一旦分配，生产者队列永远不会被销毁（直到

级别队列被销毁），但在下次创建生产者令牌时会重新使用它（而不是导致新生产者队列的堆分配）。

## 隐式生产者队列

隐式生产者队列实现为一组未链接的块。它无锁定，但不是无等待，因为主自由块自由列表的实现是无锁的，并且块不断从该池获取并插入回该池（调整块索引大小也不是恒定时间，并且需要内存分配）。实际的队列和取消队列操作在单个块内仍无等待。

维护当前块指针;这是当前正在被 enqueu 的块。当一个块填满时，一个新的块被征用，旧的块（从生产者的角度来看）被遗忘。在将元素添加到块之前，块将插入块索引（允许使用者查找生产者已经忘记的块）。当使用完块中的最后一个元素时，该块会从块索引中逻辑删除。

隐式生产者队列永远不会被重新使用 -- 一旦创建，它在整个高级队列的生存期内。因此，为了减少内存消耗，而不是占用它曾经使用的所有块（如显式生产者），而是将已花的块返回到全局自由列表。为此，一旦使用者完成对项目进行取消队列，每个块中的原子取消队列计数将增加;当计数器达到块的大小时，看到此值的使用者知道它只是取消最后一项的奎点，并将块放在全局自由列表中。

隐式生产者队列使用循环缓冲区来实现其块索引，允许以恒定时间搜索块

鉴于其基本索引。每个索引条目由表示块基本索引的键值对和指向相应块本身的指针组成。由于块始终按顺序插入，因此索引中每个块的基本索引保证在相邻条目之间只增加一个块大小。这意味着，通过查看最后插入的基础索引、计算到所需的基本索引的偏移量以及查找该偏移量的索引条目，可以很容易地找到索引中已知位于索引中的任何块。特别注意确保算法在块索引环绕时仍然有效（尽管假定在任何给定时刻，索引中不会出现（或看到相同的基本索引存在）两次）。

当使用块时，将从索引中删除它（为将来的块插入提供空间）；由于另一个使用者可能仍在使用索引中的该条目（以计算偏移量），因此索引条目不会完全删除，但块指针设置为 null，而是指示生产者该槽可以重新使用;对于仍在使用它计算偏移量的任何使用者，块基数保持不变。由于生产者仅在前面的所有插槽都是免费的后才重新使用插槽，并且当使用者查找索引中的块时，索引中必须至少有一个非自由槽（对应于它查找的块），并且使用者用于查找块的块索引条目至少与最近为该块的块一样，在生产者重新使用插槽和使用该插槽查找块的消费者之间永远不会出现竞争条件。

当使用者希望对项目进行排队，并且块索引中没有空间时，它（如果允许）分配另一个块索引（链接到旧块索引，以便最终在队列被析时释放其内存），该索引从此成为主索引。新索引是旧索引的副本，但大索引的两倍;复制所有索引条目允许使用者

只需查看一个索引，就可以找到它们要查找的块（在块内不断取消队列）。由于使用者在构造新索引时可能正在将索引条目标记为空（通过将块指针设置为 null）进行标记，因此索引条目本身不会复制，而是指向它们。这可确保使用者对旧索引的任何更改也会正确影响当前索引。

# 隐式生产者队列哈希

无锁哈希表用于将线程 ID 映射到隐式生产者;当没有为各种队列方法提供显式生产者令牌时，使用此选项。它基于 Jeff Preshing 的无锁定哈希算法，并进行了一些调整：键的大小与依赖于平台的数字线程 ID 类型相同;值是指针;当哈希变得太小（使用原子计数器跟踪元素的数量）时，将分配一个新的哈希并链接到旧哈希，并且旧哈希中的元素在读取时被懒洋洋地传输。由于元素数的原子计数可用，并且元素永远不会被删除，因此希望在哈希表中插入元素的线程必须尝试调整大小或等待另一个线程完成调整大小（调整大小使用锁进行保护以防止杂散分配）。为了在争用下加快调整大小（即最大限度地减少线程等待另一个线程完成分配的自旋等待量），可以在旧哈希表中插入项，其阈值明显大于触发调整大小的阈值（例如，负载因子为 0.5 将导致开始调整大小，同时可以在旧线程中插入元素，加载系数高达 0.75， 说）。

# 链接的制作人名单

如前所述，将保留所有生产者的单独链接 （LIFO） 列表。此列表使用尾指针实现，并采用每个生产者的侵入性下一个（真正"前一个"）指针。尾部最初指向 null;创建新生产者时，它会通过先读取尾部，然后使用设置其旁边的尾部，然后使用 CAS 操作将尾部（如果尚未更改）设置为新生产者（必要时循环）来将自己添加到列表中。生产者永远不会从列表中删除，但可以标记为非活动。

当消费者想要取消排队项目时，它只需在查找包含物料的 SPMC 队列的生产者列表中走一步（因为生产者的数量是不受限制的，这在一定程度上使得高级队列只是无锁而不是无等待）。

# 去奎元化启发式

使用者可以将令牌传递给各种取消队列方法。此令牌的目的是加快选择适当的内部生产者队列，从中尝试取消排队。使用一个简单的方案，通过该方案，每个显式使用者都分配一个自动递增偏移，表示应取消排队的生产者队列的索引。以这种方式，消费者尽可能公平地分布在生产者之间;然而，并非所有生产者都有相同数量的可用元素，而且有些消费者的消费速度可能高于其他;为了解决这个问题，第一个从同一内部生产者队列中连续消耗 256 个物料的使用者将增加全局偏移量，导致所有使用者在其下一个取消排队操作时旋转，并开始从下一个生产者使用。（请注意，这意味着旋转速率由最快的使用者决定。如果使用者的指定队列上没有可用的元素，它将移动到具有可用元素的下一个元素。这种简单的启发式技术是有效的，能够将消费者与具有近乎完美扩展的生产者配对，从而实现令人印象深刻的取消队列速度。

# 关于线性性的注意

如果数据结构的所有操作似乎都按顺序（线性）顺序执行，即使在并发条件下，数据结构都是线性的（本文有一个良好的定义）。虽然这是一个有用的属性，因为它使并发算法明显正确，更容易推理，它是一个非常强大的一致性模型。我在这里介绍的队列是无法线性的，因为要进行线性计算会导致性能降低得多;然而，我相信它仍然是相当有用的。我的队列具有以下一致性模型：任何给定线程上的 Enqueue 操作（显然）在该线程上是线性的，但其他线程上不是线性的（这应该并不重要，因为即使对于完全可线性的队列，元素的最终顺序也是不确定的，因为它取决于线程之间的种族）。请注意，即使 enqueue 操作不能跨线程线性化，它们仍然是原子的 —— 只有完全重新表示的元素才能取消重新奎点。如果所有生产者队列在检查时都为空，允许取消排队操作失败。这意味着取消排队操作也是非线性的，因为队列作为一个整体在失败的取消排队操作期间的任何一个点不一定为空。（即使是单个生产者队列的空量检查在技术上也是非线性的，因为 enqueue 操作可以完成，但内存效果尚未传播到去排队线程中——同样，这应该并不重要，因为它依赖于非确定性的种族条件。

这种非线性在实际中意味着，如果仍有其他生产者正在排队（无论是否在其他线程正在取消排队），取消排队操作可能会在队列完全为空之前失败。请注意，即使具有完全可线性的队列，此竞争条件也存在。如果队列已稳定（即所有排队操作已完成，并且其内存效果已变得可见任何潜在

使用者线程），那么只要队列不为空，取消排队操作就永远不会失败。同样，如果给定元素集对所有取消排队线程可见，则这些线程上的取消排队操作永远不会失败，直到至少使用该组元素（但之后可能会失败，即使队列不是完全为空）。

# 结论

所以，你有它！比你想知道我的设计通用无锁队列我已经使用C++11实现了这个设计，但我确信它可以移植到其他语言。如果有人用另一种语言实现这个设计，我很想听听！

# 用于其他设备的快速通用无锁定C++

发布于 2014年11月06日

所以我被无锁的虫子咬了！完成后
我的单生产者，单消费者无锁队列，我决定设计和实现一个更通用的多生产者，多消费者队列，看看它是如何堆叠在现有的无锁C++队列。经过一年的业余时间开发和测试，终于到了公开发布的时候了。
TL;DR：你可以从 GitHub C++ 11 实现（或跳转到基准）。

我认为，队列的工作方式很有趣，所以这就是这篇博文的用。顺便说一下， 姐妹博客文章中提供了一个更详细和完整的 （但也更干燥） 的描述。

## 共享数据：哦，困境

乍一看，通用无锁队列似乎很容易实现。事实并非如此。问题的根源是，相同的变量必然需要与多个线程共享。例如，采用一种常见的基于链接列表的方法：至少需要共享列表的头部和尾部，因为使用者都需要能够读取和更新头，并且生产者都需要能够更新尾部。

到目前为止，这听起来还不算太坏，但当线程需要更新多个变量以保持队列处于一致状态时，会出现真正的问题—— 仅确保单个变量的原子性，而复合变量（结构）的原子性几乎肯定会导致某种锁（在大多数平台上，取决于变量的大小）。例如，如果

使用者从队列中读取最后一项， 只更新了头？尾巴不应该仍然指向它，因为对象将很快被释放！但是使用者可能会作系统中断并挂起几毫秒，然后再更新尾部，在此期间，尾部可能会被另一个线程更新，然后第一个线程将尾值设置为 null 为时已晚。

共享数据的这一根本问题的解决方案是无锁编程的症结所在。通常，最好的方法就是设想一种不需要更新多个变量以保持一致性的算法，或者一个增量更新仍然使数据结构保持一致状态的算法。可以使用各种技巧，例如，一旦分配了内存，就永远不会释放内存（这有助于从不是最新的线程读取）、在指针的最后两个位中存储额外状态（这适用于 4 字节对齐指针）和引用计数指针。但这样的技巧只走这么远;真正的努力是开发算法本身。

# 我的队列

线程争夺相同数据的线程类型就越好。因此，不使用线性化所有操作的单个数据结构，而是使用一组子队列，即每个生产者线程一个。这意味着不同的线程可以完全并行地对项进行分水，彼此独立。

当然，这也使得取消排队稍微复杂一些：现在我们必须在取消排队时检查每个子队列中的项目。有趣的是，事实证明，从子队列中拔出元素的顺序真的无关紧要。取消排队时，给定生产者线程的所有元素仍必须按彼此相对的相同顺序看到（因为子队列保留该顺序），尽管其他子队列的元素可能交错。交错元素正常

因为即使在传统的单队列模型中，元素从不同的生产者线程输入的顺序也是非确定性的（因为不同的生产者之间有一个竞争条件）。[编辑：只有当生产者是独立的，这不一定是这样。请参阅评论。此方法的唯一缺点是，如果队列为空，则必须检查每个子队列才能确定这一点（而且，在检查一个子队列时，以前为空的子队列可能变为非空队列，但在实践中这不会导致问题）。但是，在非空情况下，总体争用性要小得多，因为子队列可以与使用者"配对"。这会将数据共享缩短到近乎最佳的水平（其中每个使用者都只与一个生产者匹配），而不会失去处理一般案例的能力。此配对使用启发式方法完成，该启发式方法考虑到生产者成功从该子队列中成功退出的最后一个子队列（本质上，它为使用者提供了亲和力）。当然，为了进行这种配对，在调用取消队列之间必须保持某种状态 ——这是使用用户负责分配的特定于消费者的"令牌"完成的。
请注意，令牌是完全可选的 —— 队列只是还原到搜索每个子队列中搜索一个没有一个元素的元素，这是正确的，只是在涉及许多线程时稍慢一些。

所以，这就是高级设计。每个子队列中使用的核心算法如何？嗯，我基于数组模型，而不是基于节点链接列表（这意味着不断分配和释放或重新使用元素，并且通常依赖于在激烈争用下速度较慢的比较和交换循环）。我没有链接单个元素，而是有一个包含多个元素的"块"。队列的逻辑头和尾指数使用原子递增整数表示。
在这些逻辑索引和块之间，有一个方案，用于将每个索引映射到该块中的块和子索引。

排队操作只是增加尾部（请记住，每个子队列只有一个生产者线程）。如果发现头部小于尾部，则取消排队操作会增加头部，然后检查其是否意外地将头增量到尾部（这可能发生在争用下 - 每个子队列有多个使用者线程）。如果它做了过度增量的头，修正计数器递增（使队列最终一致），如果没有，它继续前进，并递增另一个整数，为它提供实际的最终逻辑索引。此最终索引的增量始终在实际队列中生成有效索引，而不管其他线程正在执行或已执行哪些操作;这之所以有效，是因为最终索引仅在保证至少有一个元素要取消队列时才递增（当第一个索引递增时已选中）。

所以，你有它。enquee 操作使用单个原子增量完成，取消队列使用快速路径中的两个原子增量，以及额外的一个。（当然，这是打折扣的所有块分配重新使用计数块映射 goop，这虽然重要，不是很有趣 - 在任何情况下，这些成本大部分摊销在整个块的元件价值。这个设计真正有趣的部分是，它允许极其高效的批量操作 - 在原子指令方面（这往往是一个瓶颈），在块中对 X 项进行队列的开销与对单个项目进行队列（用于取消队列的 ditto）的开销完全相同，前提是它们位于同一个块中。这就是真正的业绩提升的来向在 : -)

# 我听说有密码

由于我认为在一个用户中相当缺乏高质量的无锁定队列C++，我用我的这个设计写了一个。（虽然还有其他，特别是升压和英特尔的TBB，

mine 具有更多功能，例如对元素类型没有限制，并且启动速度更快。你可以在 GitHub 找到它。它全部包含在单个标头中，可在简化的 BSD 许可证下使用。只需把它丢在你的项目中，享受吧！

# 基准，耶！

因此，创建数据结构的有趣部分是编写综合基准，并查看您的基准测试与其他现有基准相比的速度。为了进行比较，我使用了 Boost 1.55 无锁队列，英特尔的 TBB 4.3 concurrent_queue，另一个基于链接列表的无锁队列（一个天真的设计供参考），一个使用 std: mutex 的基于锁的队列，以及一个普通 std: queue（用于仅从一个线程访问的正常数据结构）。请注意，下面的图表仅显示结果的子集，并省略了天真的无锁和单
线程 std: : 队列实现。

下面是结果！详细的原始数据遵循漂亮的图表（请注意，由于绝对吞吐量的巨大差异，我不得不使用对数比例）。

如您所见，我的队列通常至少与下一个最快的实现一样好，而且速度通常要快得多（尤其是批量操作，它们位于他们自己的联盟中！显示与其他队列吞吐量显著减少的唯一基准是"从空队列取消排队"，这是实现最糟糕的情况取消排队方案，因为它必须检查所有内部队列；不过，平均而言，它比成功的取消队列操作要快。这一点的影响也在某些其他基准中可以看到，例如单生产者多消费类基准，其中大多数取消队列操作失败（因为生产者跟不上需求），以及相对

因此， 我的队列吞吐量与其他队列相比略有下降。另请注意，锁定基队列在我的 Windows 上网本上非常慢;我认为这是一个实施质量的问题与MinGW std: : mutex，它不使用Windows的高效CRITICAL_SECTION似乎遭受可怕的结果。

我认为，从这些基准中可以收集的最重要的事情是，随着线程数量的增加（而且经常会下降），系统总吞吐量最多只能保持不变。这意味着，即使每个基准的队列最快，每个线程完成的工作量在添加每个线程时会单独下降，而由四个线程和十六个线程完成的总工作量大致相同。这是最好的情况。在此争用和吞吐量级别上没有线性缩放;这个故事的寓意是远离需要密集共享数据的设计， 如果你关心性能。当然，实际应用程序往往不是纯粹在队列中移动东西，因此具有低得多的争用，并且确实有可能在线程添加时向上扩展（至少一点点）。

总的来说，我对结果很满意。享受排队！