**内存分配设计文档**

直到7.0版本， 在大规模的查询中，按引用传递数据时，PG有严重的内存泄漏问题. 之前的版本中，当查询结束时，没有规定内存的回收问题. 这个问题需要解决，特别是TOAST(它允许巨大的数据块在系统间传递)应用出现以来，更应如此.本文档描述了在7.1版本中应用的新的内存管理策略.

**检测**

我们(之前)已经在”MC”中实现了我们大部分的内存分配工作，它经常通过backend/utils/mmgr/aset.c的设置来实现. 我们需要做的就是创建更多的MC并定义适当的关于内存何时释放的规则.

一个MC上的基本操作为：

* 创建一个MC
* 在MC中分配一块内存(跟标准C库中的malloc()等价)
* 删除一个MC(包括释放所有的分配在那里的内存)
* 重置一个MC(释放所有分配给MC的内存，但是不释放MC对象自身)

给定先前已分配给一个MC的内存，你可以释放它或者对其再分配更大的或更小的内存(相当于标准库中的free()和realloc()函数). 这些操作返回内存给原始分配的内存块或者从原始分配的内存块中获取更多的内存.

在任何时候，都有一个由全局变量CurrentMemoryContext标识的”当前”MC.后台的宏palloc()隐式地为该MC分配内存. MemoryContextSwitchTo()操作选择用来选择一个新的当前MC(并返回之前的MC，所以该函数的调用者能在自己退出前归还之前的MC).

MC优于普通malloc/free的主要优点在于，整个MC的内存内容可以轻易地被释放，而不必在每个MC的内存块中来调用释放操作.这比在每个内存块中做free操作更快、更可靠. 我们已经在之前的版本的事务结束时使用了该技术：我们通过重置所有的活动MC来回收所有的内存. 我们需要的是额外的内存，它们可以在查询中的至关重要时刻重置和删除，比如在每次tuple之后.

**关于palloc和标准C库相比的一些注意事项**

Palloc以及其相关操作行为跟标准C库中的malloc及其相关操作相似，但是这里还是有一些需要慎重考虑的差别.以下是一些澄清这些差别的注意事项.

* 如果内存耗尽，palloc和repalloc通过elog(ERROR)来退出.它们并不会返回NULL， 并且检测其返回值没有必要，也没有用.
* palloc(0)是一个有效操作. 它不会返回一个NULL指针，但是一个没有任何字节的有效内存块会被使用.(该内存块可能会被重新分配得更大；它也可能被释放掉，而不会导致错误)(注意：这是8.0版本中的新行为；更早的版本禁用了palloc(0)，但允许该操作似乎更具有一致性.)类似地，repalloc允许将内存大小重置为0.
* pfree和repalloc不能接收一个NULL指针.这是有意而为的.

**pfree/repalloc不再依赖于CurrentMemoryContext**

在该建议下，pfree和repalloc()可以应用于任何内存块，而不论其是否属于CurrentMemoryContext---不管怎样，拥有内存块的MC将会被调用来处理这些操作(译注：MC被定义为一组函数指针的结构体，MC就是通过这些函数来处理这些操作的)这是对先前需求的修改，之前的编码需求是CurrentMemoryContext必须在pfree()和repalloc()调用之前，设置为MC已分配的那片内存块(译注：即如果要将一个MC设置为CurrentMemoryContext，必须要在MC未执行palloc()和repalloc()之前). 先前的编码需求显然是易于发生错误的，并且当我们做更多的MC切换时，问题更甚.所以我认为只为palloc使用CurrentMemoryContext是由必要的，在pfree/repalloc中我们可以避免这样，即通过在MC管理中设置限制.这在下文讨论.

我们甚至考虑了彻底去掉CurrentMemoryContext，以不需要显式地为目标MC的内存指定分配内存.但是我们想这会出现过多的标记—-我们必须在多个地方正确的传递一个MC来调用这些函数(译注：指MC结构中的那些函数指针).例如，copyObject函数可能需要传递进来一个MC，正如函数执行例程可能需要传递一个按引用传递的数据类型一样. 但是若函数临时在其内部分配空间，却又不将其返回给调用者，情况会怎样？我们当然不愿意每次调用”这是一个MC， 它是你想使用的临时内存”来扰乱系统. 所以，还有需要一个全局变量来制定一个合适的临时内存分配的MC.它就是CurrentMemoryContext.

**MC机制中的新加入内容**

如果我们想拥有更多的MC，我们就需要更多的机制来跟踪它们；否则，若发生错误，我们的MC将会有内存泄漏的危险.

我们可以通过创建”parent”和”child”的MC树来跟踪它们.当创建一个MC时，新的MC可以被指定为一个已存在的MC的子节点.一个MC可以有多个子节点，但只有一个父节点. 这种方式会形成一个森林(并不一定是一个单一的树，因为可能有多于一个的顶层MC).

接着我们假定，重置或删除一个特定的MC『也』会重置或删除其直接或间接的子节点. 这个特性使得我们可以管理很多的MC而不必担心它们之中的一些会泄漏掉；我们只需要跟踪一个顶层MC，我将在任务结束后将其删除，并且确保我们创建的任何短命MC都是该MC的后代. 由于这棵树可以有多个层次，在存储上，我们可以采取嵌套的存储生存期，比如按照每个事物，每个语句，每个扫描或每个元祖等. 存储生存期(只有部分重叠)可以通过在一个MC森林的不同的树中分配内存来解决(在下一节中有一些实例).

为方便起见，我们还允许这样的操作:”重置/删除给定MC的所有子节点，但是不重置或删除MC自身”.

**全局可见的MC**

这里有一些全局可见的MC，它们一般通过定义为全局变量来引用. 在任意时刻，系统中都可能包含很多附加的MC，但是所有的其它MC都应是这些附加MC的直接或间接孩子节点，以此来保证它们不会(甚至在发生错误时)泄漏掉.

TopMemoryContext --- 这实际上是MC树的顶层；每一个其它的MC都是它的一个直接或间接子节点.此处的内存分配在本质上跟”malloc”是一样的，因为这个MC永远不会(译注：相对来说)被重置或删除. 这是为一些永久生存的部件的考虑，或者为一些在适当时刻处理删除操作的控制模块的考虑. 一个例子就是fd.c中的『已打开文件表』,MC自身的MC节点管理也跟这相似. 除非必要，避免在这里分配一些东西，并且，尤其不要当CurrentMemoryContext指向该处时分配.

PostmasterContext --- 这是postmaster的常用工作MC. 一旦一个后台启动，它会删除PostmasterContext来释放其对postmaster的内存拷贝(如果这些内存不再使用).(任何从postmaster中传递到backend的内存都将传递到TopMemoryContext和ErrorContext中---顶层MC的剩余内存将会在后台启动时设置).

CacheMemoryContext --- permanent storage for relcache, catcache, and

related modules. This will never be reset or deleted, either, so it's

not truly necessary to distinguish it from TopMemoryContext. But it

seems worthwhile to maintain the distinction for debugging purposes.

(Note: CacheMemoryContext will have child-contexts with shorter lifespans.

For example, a child context is the best place to keep the subsidiary

storage associated with a relcache entry; that way we can free rule

parsetrees and so forth easily, without having to depend on constructing

a reliable version of freeObject().)

CacheMemoryContext --- relcache、catcache和相关模块的永久存储.这些将永不会重置或删除，所以区分它和TopMemoryContext并没有实际必要.但是掌握这些区别在debug过程中还是值得的.(注意：CacheMemoryContext将会有一些短命的子节点.例如，一个子节点是保存一个relcache入口的附加存储的最佳位置时)