

# PostgreSQL 源码分析之内存上下文

## 前言

PostgreSQL 是我们项目采用的数据库，从来没读过数据库的代码，尽管也曾写过一些应用层的代码，希望今年能粗读一遍 PostgreSQL 的源码，提高自己对数据库的理解。

本系列以 PostgreSQL 的最新版本 9.2.3 源码为例，学习 PostgreSQL。

PostgreSQL 从 7.1 开始，引入了内存上下文（MemoryContext）机制，片汤话我不多说，简单的理解，内存上下文提供了一种管理内存的机制。我们通过图表和代码分析来理解 PostgreSQL 的内存上下文。



- 创建了 TopMemoryContext 的第一个子节点 ErrorContext。

创建内存上下文的工作是由 AllocSetContextCreate 函数完成的。这个很有意思。MemoryContext 明明是上面提到的第一种数据结构，他的创建函数偏偏是 AllocSetContextCreate，这个函数顾名思义也知道是创建第二个数据结构的。这其实很好理解，看上面绘制的图片 可以看出，MemoryContextData 不过是核心数据结构 AllocSetContext 的第一个成员变量（更严格的说是它的一个指针类型的成员 变量指向这个 MemoryContextData）。

AllocSetContextCreate 这个函数其实是分成两部分的

- MemoryContextCreate , 创建 MemoryContext
- 创建 AllocSetContext 剩余的部分，主要是确定 initBlockSize,nextBlockSize,maxBlockSize 和 allocChunkLimit 的大小。

对于 MemoryContextCreate 这个函数，TopMemoryContext 是没有 parent 的，所以他的 parent 指针是 NULL；另 外一个需注意的点是 method，在 TopMemoryContext 创建 methods 指针指向了一个结构体，这个结构体内是一系列分配释放相关的函数，都画在了上图的右上角。因为 TopMemoryContext 是根，所以他的分配 需要用 malloc，其他的 MemoryContext 创建的时候，就不需要调用系统函数 malloc 了，直接用 method 函数指针系列里面 AllocSetAlloc 函数分配就行了。见如下代码

```

1.  if (TopMemoryContext != NULL)
2.  {
3.      /* Normal case: allocate the node in TopMemoryContext */
4.      node = (MemoryContext) MemoryContextAlloc(TopMemoryContext,
5.          needed);
6.  }
7.  else
8.  {
9.      /* Special case for startup: use good ol' malloc */
10.     node = (MemoryContext) malloc(needed);
11.     Assert(node != NULL);
12. }
```

确定 `initBlockSize`, `nextBlockSize` 等的代码比较简单, 我就不赘述了。

`allocChunkLimit` 这个参数的含义是在这个内存上下文, 大内存块的门限值。比如如果 `maxBlockSize=8K`, 那么系统认为 1KB 是比较大的内存块, 低于 1KB 的这个门限值的, 都认为是小块内存。在分配策略和释放策略上, 是不同的。我们认为大内存的分配不频繁, 所以我们采用直接 `malloc` 的方法, 如果释放的话, 就真的调用 `free`, 将内存返还给系统。但是小内存块则不同, 我们认为小内存块的分配是频繁的, 而且频繁的 `malloc/free` 会造成内存碎片, 所以当用户调用 `AllocSetFree` 的时候, 我们并不真正的返还给系统, 而是挂在可用 `chunk` 列表中。等待下一次的分配。

OK, 我们已经透露了一些分配和释放的原则, 那么, 我们就看下分配和释放部分的代码吧。

## 2 内存上下文中的内存操作

在 PostgreSQL 中, 内存的分配, 重分配, 释放都是在内存上下文中进行的, 不再直接调用系统提供的 `malloc/realloc/free`。PostgreSQL 提供了一个系列的函数, 来管理内存

1. `/*`
2. `* This is the virtual function table for AllocSet contexts.`
3. `*/`
4. `static MemoryContextMethods AllocSetMethods = {`
5.  `AllocSetAlloc,`
6.  `AllocSetFree,`
7.  `AllocSetRealloc,`
8.  `AllocSetInit,`
9.  `AllocSetReset,`
10.  `AllocSetDelete,`
11.  `AllocSetGetChunkSpace,`

```

12. AllocSetIsEmpty,

13. AllocSetStats

14.#ifdef MEMORY_CONTEXT_CHECKING

15. ,AllocSetCheck

16.#endif

17.};

```

下面我们重点介绍 Alloc Free Realloc 这几个函数。

前面我们提到过，在 AllocSetContext 这个结构体中有一个很重要的成员变量：  
allocChunkLimit，如下所示：

```

1. typedef struct AllocSetContext
2. {
3.     MemoryContextData header; /* Standard memory-context fields */
4.     /* Info about storage allocated in this context: */
5.     AllocBlock blocks; /* head of list of blocks in this set */
6.     AllocChunk freelist[ALLOCSET_NUM_FREELISTS]; /* free chunk lists */
7.     /* Allocation parameters for this context: */
8.     Size initBlockSize; /* initial block size */
9.     Size maxBlockSize; /* maximum block size */
10.    Size nextBlockSize; /* next block size to allocate */
11.    Size allocChunkLimit; /* effective chunk size limit */
12.    AllocBlock keeper; /* if not NULL, keep this block over resets */
13.} AllocSetContext;
14.
15. typedef AllocSetContext *AllocSet;

```

这个 allocChunkLimit 的是内容上下文中一个很重要的参数，这个参数的含义上面也曾提及到，含义是大小 chunk 的门限值。

如果 PostgreSQL 需要在内存上下文分配大于 allocChunkLimit 的内存区域，那么内存上下文认为这是分配较大的内存，采用 malloc 的方法，同时将分配出来的 block

链入内存上下文的 block 链表中。如果用户释放该内存区域（实际上是 chunk），那么内存上下文会真正的 free，返还给操作系统。

如果 PostgreSQL 需要在内存上下文分配小于 allocChunkLimit 的内存区域，那么行为是不同，往根本上讲，这些小块内存当用户选择释放的时候，并不真正的调用 free，而是将小块内存作为 free chunk，根据大小链接在 freelist。freelist 的概念和伙伴内存系统有些类似，有 11 条链表，每条链表的 chunk 大小是不同的。分别是 8/16/32/64/128/256/512/1024/2048/4096/8192。当进程调用 AllocSetFree 去释放这些小块内存的时候，内存上下文会将这些内存块放到 freelist 对应的链表中，以待下一次分配。这么做的好处是防止小块内存的不停 malloc/free 造成大量的碎片产生。

这么看起来 allocChunkLimit 这个值很重要，那么这个值是怎么算出来的呢。首先需要说 allocChunkLimit，不同的内存上下文，其大小可能是不同的。它的值大小是在 AllocSetContextCreate 函数里面计算出来的。

```
1.  #define ALLOC_MINBITS 3 /* smallest chunk size is 8 bytes */
2.  #define ALLOCSET_NUM_FREELISTS 11
3.  #define ALLOC_CHUNK_LIMIT (1 << (ALLOCSET_NUM_FREELISTS-
    1+ALLOC_MINBITS))
4.  /* Size of largest chunk that we use a fixed size for */
5.  #define ALLOC_CHUNK_FRACTION 4
6.  /* We allow chunks to be at most 1/4 of maxBlockSize (less overhead) */
7.
1.  context->allocChunkLimit = ALLOC_CHUNK_LIMIT;
2.  while ((Size) (context->allocChunkLimit + ALLOC_CHUNKHDRSZ) >
3.      (Size) ((maxBlockSize - ALLOC_BLOCKHDRSZ) / ALLOC_CHUNK_FRACTION))
4.      context->allocChunkLimit >>= 1;
```

ALLOC\_CHUNK\_LIMIT 的值为 8K，也就是说内存上下文的 allocChunkLimit 最大就是 8K，但是实际的 context->allocChunkLimit，还需要根据 maxBlockSize 来计算。下面这个 while 的含义是一个最大的 block 的应该不小于 4 倍的 allocChunkLimit。以 TopMemoryContext 为例，maxBlockSize = 8K，那么 allocChunkLimit 应该是小于 2K，所以最终计算的结果是 allocChunkLimit = 1K。TopMemoryContext 作为根内存上下文，从这里分配的内存多是用来存储子内存上下文，而子内存上下文对应的数据结

构非常的小，不会超过 1K，所以 allocChunkLimit=1K 是合理的。而 PostmasterContext 的 maxBlockSize = 8M，所以 PostmasterContext 的 allocChunkLimit=8K。

讲完了 allocChunkLimit 这个参数，nextBlockSize 也很重要。block 和 chunk 是这个内存上下文的比较重要的概念。这个概念 简单理解就是大公司管理网线（因为内存有申请和释放，网线不用之后，还可以归还回去）。操作系统是个全公司总仓库，它的有点是货源充足（仓库里有大量的内存空间可用），缺点是提货不方便，你可以想想，几万人要 1 米 2 米的网线都要去千里之外的全公司总仓库，我们有多烦，不光我们烦躁，网线管理员也很烦躁，因为短则 1 米，长则上千米网线频繁的切割，会造成仓库的混乱。对应操作系统来讲，就是小块内存的频繁申请和释放，会造成内存碎片，仓库空间虽大，但是横七竖八的小网线弄得在也分配不了长网线了。那么怎么办呢。很简单，在每个分公司成立分仓库。分仓库就是内存上下文。分仓库负责申请一段很长的网线，然后给公司员工用。员工用完了网线，再还给分仓库，就不用归还 到全公司总仓库了，直接归还分仓库，分仓库会按照网线长短放在 11 个地方，存放网线，下次员工来取了，直接向对应的房间（对应的 freelist）去取。有时候员工可能会取比较长的网线，比如这个员工要 10000 米的网线，分仓库去总仓库去取（malloc），然后员工用，员工归还的时候（AllocSetFree），分仓库真的将这 10000 米网线归还给总仓库（free）。

block 就是分仓库批发过来的很长的网线，既然是批发，就要有规则，不可能今天去总仓库取 1 米，明天去取 3 米，公司总仓库烦都烦死了。maxBlockSize 是分仓库一次最多取的长度，nextBlockSize 记录的是下一次我应该去总仓库取多少米。以 Postmastercontext 为例，刚初始化的时候，nextBlockSize=8K，maxBlockSize=8M。这个分仓库刚开始的时候，他取的是 8K，因为员工用完了还会归还，所以，一旦发生货源不足的话，下一次进货，应该是 nextBlockSize×2。请看分仓库去总仓库申请长网线的代码：

```
1.  if (block == NULL)
2.  {
3.      Size      required_size;
4.
5.      /*
6.       * The first such block has size initBlockSize, and we double the
7.       * space in each succeeding block, but not more than maxBlockSize.
```

```

8.      */
9.      blksize = set->nextBlockSize;
10.     set->nextBlockSize <= 1;    //下一次去总仓库取网线，要多取 1 倍
11.     if (set->nextBlockSize > set->maxBlockSize)
12.         set->nextBlockSize = set->maxBlockSize; //取网线最多不能超过 maxBlockSize
13.
14.     /*
15.      * If initBlockSize is less than ALLOC_CHUNK_LIMIT, we could need more
16.      * space... but try to keep it a power of 2.
17.      */
18.     required_size = chunk_size + ALLOC_BLOCKHDRSZ + ALLOC_CHUNKHDRSZ;
19.     while (blksize < required_size)
20.         blksize <= 1;           //如果当前要网线的员工要的太多，
21.                                   //超过了本次应该的取的长度，则 double
22.
23.     block = (AllocBlock) malloc(blksize);
24.     .....
25.
26.     block->aset = set;
27.     block->freeptr = ((char *) block) + ALLOC_BLOCKHDRSZ;
28.     block->endptr = ((char *) block) + blksize;
29.     ....
30. }

```

有了很长的网线，就能满足当前员工的需求了。但是去总仓库申请来的很长的网线是不是立刻就截断成 1 米 2 米 4 米 8 米这种长度呢？答案是否定的。

我们来看下员工申请网线的情况。员工申请 14 米的网线 1 根，那么仓库管理员首先干的事情是看下有没有 16 米的网线。对应 freelist 的某个 chunk。如果有的话，皆大欢喜，员工拿了网线走人。对应的代码如下：

```

1.  fidx = AllocSetFreeIndex(size);
2.  chunk = set->freelist[fidx];
3.  if (chunk != NULL)
4.  {

```



```

5.     Assert(chunk->size >= size);
6.
7.     set->freelist[fixd] = (AllocChunk) chunk->aset;//长度 16 的网线，是链接在一起的。
8.
9.     chunk->aset = (void *) set;
10.
11.#ifdef MEMORY_CONTEXT_CHECKING
12.     chunk->requested_size = size;
13.     /* set mark to catch clobber of "unused" space */
14.     if (size < chunk->size)
15.         ((char *) AllocChunkGetPointer(chunk))[size] = 0x7E;
16.#endif
17.#ifdef RANDOMIZE_ALLOCATED_MEMORY
18.     /* fill the allocated space with junk */
19.     randomize_mem((char *) AllocChunkGetPointer(chunk), size);
20.#endif
21.
22.     AllocAllocInfo(set, chunk);
23.     return AllocChunkGetPointer(chunk);
24. }

```

很不幸，没有 16 米的网线，则去看下上次从总仓库那回来的网线还有多长。

1 剩余的网线超过 16 米，则可以在这个剩余的网线上截取。

2 如果不够长的话，比如从总仓库带回的网线已经只剩下 13 米了，此时分仓库管理员会将 13 米的网线截成 1 米 4 米 8 米，放入 Freelist 中，共员工来申请使用。同时去总仓库再次申请，当然不是申请 16 米，好不容易去一次总仓库，不可能只申请 16 米，而是申请 nextBlocksize, 如前所述。

剩余网线长度不够长，被分仓库管理员截断成规整的长度代码如下：

```

1. if ((block = set->blocks) != NULL)
2. {
3.     Size    availspace = block->endptr - block->freeptr;
4.
5.     if (availspace < (chunk_size + ALLOC_CHUNKHDRSZ))//剩余长度不够长

```

```

6.      {
7.      /*
8.      * The existing active (top) block does not have enough room for
9.      * the requested allocation, but it might still have a useful
10.     * amount of space in it. Once we push it down in the block list,
11.     * we'll never try to allocate more space from it. So, before we
12.     * do that, carve up its free space into chunks that we can put on
13.     * the set's freelists.
14.     *
15.     * Because we can only get here when there's less than
16.     * ALLOC_CHUNK_LIMIT left in the block, this loop cannot iterate
17.     * more than ALLOCSET_NUM_FREELISTS-1 times.
18.     */
19.     while (availspace >= ((1 << ALLOC_MINBITS) + ALLOC_CHUNKHDRSZ))
20.     {
21.         Size      availchunk = availspace - ALLOC_CHUNKHDRSZ;
22.         int        a_fidx = AllocSetFreeIndex(availchunk);
23.
24.         /*
25.         * In most cases, we'll get back the index of the next larger
26.         * freelist than the one we need to put this chunk on. The
27.         * exception is when availchunk is exactly a power of 2.
28.         */
29.         if (availchunk != ((Size) 1 << (a_fidx + ALLOC_MINBITS)))
30.         {
31.             a_fidx--;
32.             Assert(a_fidx >= 0);
33.             availchunk = ((Size) 1 << (a_fidx + ALLOC_MINBITS));
34.         }
35.
36.         chunk = (AllocChunk) (block->freeptr);

```

```

37.
38.         block->freeptr += (availchunk + ALLOC_CHUNKHDRSZ);
39.         availspace -= (availchunk + ALLOC_CHUNKHDRSZ);
40.
41.         chunk->size = availchunk;
42. #ifdef MEMORY_CONTEXT_CHECKING
43.         chunk->requested_size = 0;    /* mark it free */
44. #endif
45.         chunk->aset = (void *) set->freelist[a_fidx];
46.         set->freelist[a_fidx] = chunk;
47.     }
48.
49.     /* Mark that we need to create a new block */
50.     block = NULL; //block = NULL，需要分仓库去总仓库申请一卷长网线回来。
51. }
52. }

```

`freelist` 上面可供分配的 `chunk` 是从哪里来的呢？上面的代码是一个途径即剩余长度不能满足员工本次需求的时候，分仓库管理员会将剩余的网线截断 成 8 米 4 米这种和 `freelist` 匹配的长度，放入对应的 `freelist` 中。另外一个途径是员工归还，即 `AllocSetFree`。

`AllocSetFree` 和 `AllocSetAlloc` 一样，也是分情况的。如果超过 `allocChunkLimit`，表明员工要归还长网线，那么分仓库 会将长网线亲自归还到总仓库（`free`）。如果员工归还的网线是 16 米的网线 1 根，直接放到 16 米对应的 `freelist` 中去。

```

1. else
2. {
3.     /* Normal case, put the chunk into appropriate freelist */
4.     int      fidx = AllocSetFreeIndex(chunk->size);
5.
6.     chunk->aset = (void *) set->freelist[fidx];
7.
8. #ifdef CLOBBER_FREED_MEMORY

```

```
9.      /* Wipe freed memory for debugging purposes */
10.      memset(pointer, 0x7F, chunk->size);
11.#endif
12.
13.#ifdef MEMORY_CONTEXT_CHECKING
14.      /* Reset requested_size to 0 in chunks that are on freelist */
15.      chunk->requested_size = 0;
16.#endif
17.      set->freelist[fidx] = chunk;
18. }
```

AllocSetRealloc 部分的代码也很好理解，只要用这个网线申请理论去理解，这个内存上下文其实是比较简单的。

## 参考文献：

1 PostgreSQL 数据库内核分析