rev 04.07.02

PostgreSQL 解析資料

～ メモリ管理 ～

(株) NTT データ
ビジネス開発事業本部 システム方式技術 BU
井久保 寛明

---

1. はじめに

本資料では、PostgreSQL のメモリ管理モジュールを紹介する。大規模なプログラムでは、malloc() などで動的なメモリ割り当てを行っているうちに、メモリの開放忘れなどでメモリリークが発生する場合がよくある。少量のメモリリークでも、長時間運用するサーバの場合、それらが積み重なって大変なことになる。PostgreSQL のメモリマネージャでは、メモリコンテキストと呼ばれる単位で利用目的別にメモリを管理する仕組みと、メモリコンテキストを階層化して管理する仕組みを提供している。これらの機構によって、メモリリークの可能性を大きく減らしている。
なお、この資料では、共有メモリの管理やバッファ管理の情報は含んでいない。

1.1. 対象バージョン

本資料は、PostgreSQL7.3.4 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。

1.2. 対象モジュール

メモリマネージャ(utils/mmgr)を対象としている。

2. メモリマネージャ（utils/mmgr）

メモリマネージャは、バックエンド内でのメモリ管理を受け持つモジュールである。PostgreSQLでは、メモリコンテキスト（アロケーションセット[1]）と呼ばれる単位で、特定の目的に対するメモリ割り当てを集中して管理する方法をとっている。メモリマネージャの提供するメモリアロケーションを利用するためには、palloc() を用いる。palloc()は、現在指定されているメモリコンテキストからメモリの割り当てを行う。
通常の malloc/free を使っている場合、メモリを開放するためにはポインタを手繰って全てのノードのメモリを順番に開放する必要がある。メモリマネージャでは、メモリコンテキスト単位でまとめてメモリを管理しているので、不要になったメモリコンテキストを指定することで、一気に関連するメモリを開放できる。このため、メモリリークの発生する可能性が大幅に減っている。つまり、メモリコンテキストとは、メモリを開放するときにまとめて開放したいメモリの集まりと考えればよい。

---

[1] メモリコンテキストとアロケーションセットという用語が出てくるが、結局は同じものを指す。

Copyright © 2004 NTTDATA

PostgreSQL では、さらの階層型のメモリ管理をすることで、さらにメモリリークを起こさないようにしている。

メモリコンテキストを利用したメモリ割り当ては次のようになる。

- 目的に合わせてメモリコンテキストを生成する
- カレントメモリコンテキスト(後述)を生成したメモリコンテキストに変更する
- メモリ割り当てを行う場合、malloc()の代わりに palloc() を呼び出す
- 部分的にメモリを開放する場合は、pfree()を呼び出す
- 利用目的の作業が終了したら、カレントメモリコンテキストを別のメモリコンテキストに変更し、メモリコンテキストを削除する。

このようにしてメモリコンテキストを開放すると、このメモリコンテキストとして palloc()で確保した全メモリが開放される。

メモリマネージャは、utils/mmgr ディレクトリにあり、次の３つのファイルから構成される。

aset.c	１つのメモリコンテキスト内でのメモリ割り当ての実装。ブロックでまとめてメモリを確保し、必要なサイズに分割してメモリを提供する機能と、断片化されたメモリをサイズごとに管理する機能が実装されている。
mctx.c	メモリコンテキストの実装のうち、階層型のメモリコンテキスト管理の実装。
portalmem.c	ポータルメモリ[2]の管理を行う。

2.1. アロケーションセット （aset.c）

アロケーションセット(個々のメモリコンテキスト)を実装しているモジュールである。アロケーションセットとは、メモリコンテキストごとに、使用するメモリをまとめて管理する機構である。実際には、メモリをブロック単位で確保し、それを分割して提供したり、開放されたメモリをサイズごとに管理したりするような機能が実装されている。

2.1.1. アロケーションセットのデータ構造

アロケーションセット、つまり、１つのメモリコンテキストの構造は、次のようになっている。

[2] カーソルを使用するクエリを管理するための仕組み。

Copyright © 2004 NTTDATA

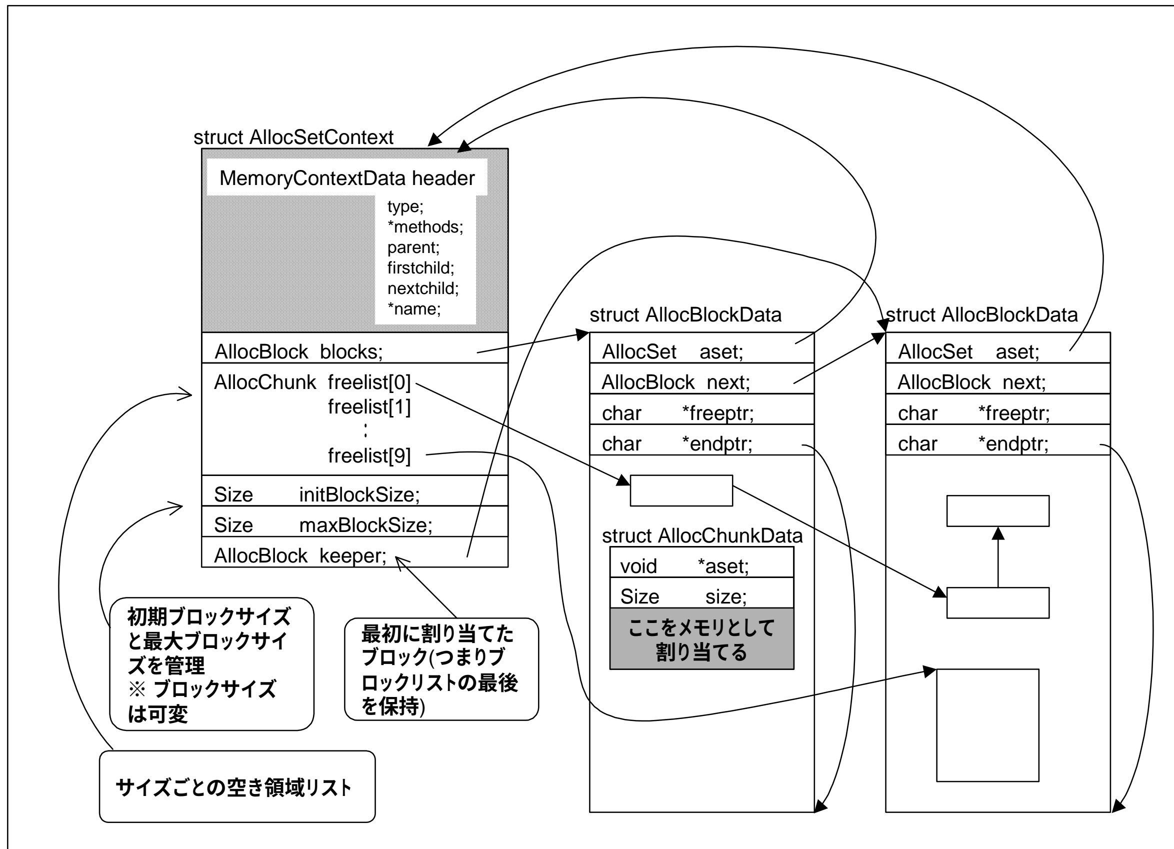

図 2.1 アロケーションセットのデータ構造

AllocSetContext 構造体が、アロケーションセット１つ、つまり、あるメモリコンテキスト１つを管理するデータ構造である。１つのメモリコンテキストで管理するメモリは、全てこの AllocSetContext 構造体からポインタが張られているブッロク内に割り当てられる。

header は、メモリコンテキストのツリー構造を維持するための部分で、mcxt.c の実装で使用される。
blocks には、このアロケーションセットに割り当てるためのメモリ領域を、一定量まとめて確保したブロック(AllocBlockData)のリストがつながれる。
freelist[] は、開放されるなどして[3]未使用になったメモリ領域を、サイズごとにメモリチャンク(AllocChunkData)のリストとして管理する。管理方法は後述する。
アロケーションセットに割り当てられるブロックは、可変サイズであり、メモリコンテキストごとにブロックサイズの最小値と最大値が指定できる。これらの値が、**initBlockSize** と **maxBlockSize** に保存されている。
keeper は、blocks につながれているブロックリストの最後を示すために使用される。

アロケーションセットで割り当てるメモリとして、メモリブロック AllocBlockData が確保してある。実際に palloc() でメモリを割り当てるときは、メモリブロック AllocBlockData の中にメモリチャンク AllocChunkData を確保する。そして、その中の **size** の後ろの領域に要求サイズのメモリを割り当てる。

[3] 開放される場合以外に、先頭のメモリブロックの空きが足りない場合に新しいブロックを追加すると、元の残りの領域がフリーリストにつながれる。

Copyright © 2004 NTTDATA

各ブロック(AllocBlockData)や割り当てられたメモリチャンク(AllocChunckData)は、自分の所属するアロケーションセットへのポインタを持っている。これによって、pfree() や repalloc() では、メモリコンテキストの指定がいらないようになっている。

2.1.2. ブロックのサイズの増え方

アロケーションセットに割り当てられるブロックは、可変サイズである。最小サイズは 1024Byte とプログラム中で決められている。1024 以上の値になら、パラメータで指定が可能である。また、メモリコンテキストごとにブロックサイズの最大値が指定できる。

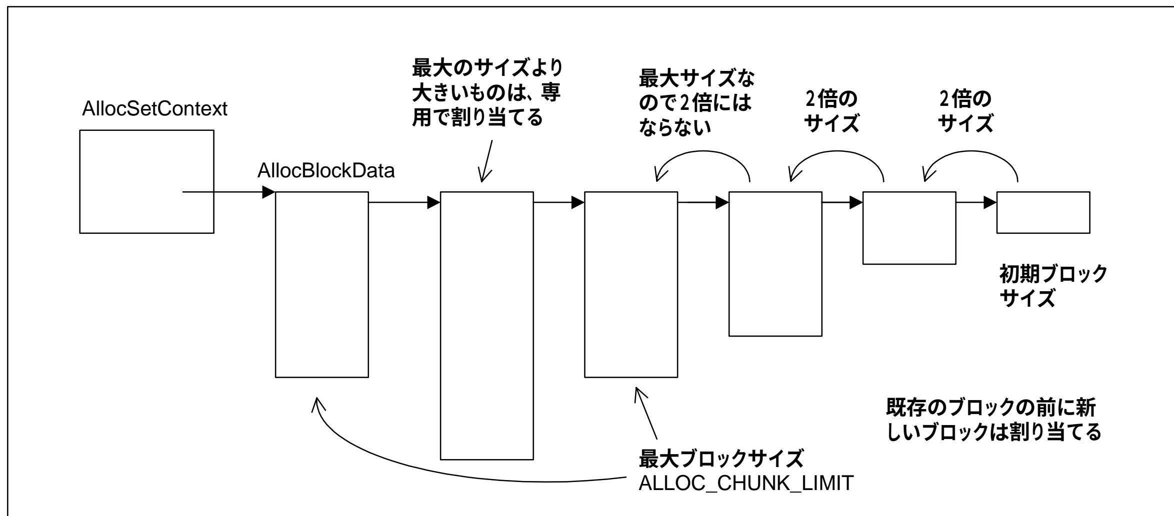

図 2.2 ブロックの割り当て

１つ目のブロックのサイズは、まず、初期ブロックサイズで割り当てられる。パラメータで指定されている初期ブロックサイズが 1024 に満たない場合、プログラム中で強制的に 1024 が採用される。
２番目以降のブロックが割り当てられる際には、前のブロックの２倍のサイズで割り当てる。あとは、最大サイズになるまで、サイズを２倍にしながら割り当てていき、最大サイズを超えたら、最大サイズで割り当てる。
「AllocSetAlloc() によるメモリ割り当て」で説明するが、最初からブロックサイズより大きいサイズでメモリ割り当ての要求が来た場合は、それ専用でブロックを割り当てる（図 2.2 のリストの２番目の AllocBlockData）。

2.1.3. 空きメモリの管理

アロケーションセット中では、メモリのサイズごとに、別々のフリーリストで管理している。最小のメモリのサイズは、次のように 4bit = 16Byte で定義されている。

```
#define ALLOC_MINBITS       4   /* smallest chunk size is 16 bytes */
```

1 ～ 16 バイト、17 ～ 32 バイト、33 ～ 64 バイトと２倍ずつ上限を増やしながら空きメモリを管理する。これは、AllocSetFreeIndex() を使って freelist[ALLOCSET_NUM_FREELISTS] のどのエントリにつながれているか計算される。エントリ数は、次のように定義されているので 10 個で、最大 8192 バイトのメモリまで管理する。

Copyright © 2004 NTTDATA

```
#define ALLOCSET_NUM_FREELISTS  10
```

各インデックス番号の扱うサイズの領域は次のようになっている。

idx	size （単位 Byte）
0	1 ～ 16
1	17 ～ 32
2	33 ～ 64
:	
7	1025 ～ 2048
8	2049 ～ 4096
9 (最大)	4097 ～ 8192

2.1.4. アロケーションセットのインタフェース

アロケーションセットとして定義されている関数は次のようになっている。

AllocSetFreeIndex()	アロケーションセットのフリーリストのインデックスを計算する
AllocSetContextCreate()	新しいメモリコンテキストを生成する
AllocSetAlloc()	メモリコンテキストの中からメモリを割り当ててポインタを返す
AllocSetFree()	メモリの開放 free() 相当の処理
AllocSetRealloc()	realloc() 相当の処理
AllocSetInit()	MemoryContextCreate() から呼ばれるのだが、MemoryContextCreate() で初期化が終わっているのでここでは何もしない
AllocSetReset()	先頭の１ブロック以外を残し、リストにつながれているブロックを開放する。残しておく最初の１ブロックは初期化を行う。
AllocSetDelete()	アロケーションセットに割り当てられている全てのブロックを開放する。AllocSetData の領域だけ残る。
AllocSetGetChunkSpace()	引数で与えられたチャンクの使用済みのサイズを計算する
AllocSetStats()	メモリの消費状況を表示する

１つ目の AllocSetFreeIndex()は、前述のフリーリストのエントリ(インデックス番号)を計算するために内部で使用する関数である。２つ目の AllocSetContextCreate()は、初期化のための関数で、メモリコンテキストを作る際に直接呼び出される。３番目以降は、次節で紹介する mcxt.c から呼び出されて使用するアロケーションセットを操作するための関数である。

2.1.5. AllocSetAlloc() によるメモリ割り当て

AllocSetAlloc() とは、palloc() を実際に実装している部分である。実際は、palloc()はマクロとして定

Copyright © 2004 NTTDATA

義してあり、カレントコンテキスト(後述)を引数に渡して、AllocSetAlloc()を呼び出す。AllocSetAlloc()によるメモリ割り当てを理解すると、アロケーションセット（aset.c）のデータ構造が理解しやすいと思うので、この関数の機能を説明する。説明の都合、実際の関数の処理順序とは大きく異なるので、ソースコードと比較する場合は、注意して頂きたい。

1. メモリ割り当ての要求が、フリーリストで管理できるサイズ(ALLOC_CHUNK_LIMIT = 8192)以内のサイズなら、まず、フリーリストのインデックス番号を AllocSetFreeIndex(size) で計算する。次に、該当するインデックス番号のフリーリストを検索する。フリーリストから空きメモリが見つかった場合は、それをフリーリストから外して、そのメモリを返す。
2. 先頭のブロック（AllocSetContext の blocks ポインタが指しているブロック）の空きスペースを調べる。要求サイズ分の空きがあれば、先頭からチャンクを作成し、メモリチャンク内のメモリのポインタを返す。
3. 既にブロックに空きスペースが足りない場合、まず、先頭ブロックの空き領域をチャンクに切り分けて、フリーリストにつなぐ。フリーリストにつなぐときは、空き領域中で最大のチャンクを切り出し、フリーリストにつなぎ、さらに残りで最大のチャンクを切り出してフリーリストにつなぐという作業を繰り返す。続いて、先頭のブロックに空きスペースが足りない状態だったので、新しいブロックを割り当てる。ブロックのサイズは、１つ前のブロックの２倍を割り当てる。メモリコンテキスト初期化時に設定された最大サイズ（maxBlockSize）を超える場合は、その最大サイズにする。この時点のサイズがALLOC_CHUNK_LIMIT以下の場合、要求サイズに満たない場合があるので微調整してブロックサイズを増やす[4]。
   ここでブロックサイズ分のメモリを malloc()で割り当てる。
   ブロックの作成に成功したら、先頭からメモリチャンクを作成し、メモリチャンク内のメモリのポインタを返す。
4. もし、フリーリストで管理できるサイズ(ALLOC_CHUNK_LIMIT)よりも大きいサイズが要求された場合、専用の AllocBlockData 領域を割り当てる。AllocBlockData の中には、AllocChunkData が１つだけ入っている状態になる。ブロックが１つもない場合は、ブロックリストの先頭につなぐが、１つでもブロックがある場合、２番目に入れる。AllocSetReset()でリセットをかけると、先頭のブロックだけを残してリストにつながれているブロックを開放するが、２番目に入れるのは、最大サイズを超えたものが残らないようにするための配慮である。

解析した際のコメント：

- フリーリストからメモリが見つからなかった場合、１つ上のメモリを使うなどの最適化は行っていない。

2.2. 階層型のメモリコンテキスト管理（mctx.c）

mctx.c は、階層型のメモリコンテキスト管理を実装しているモジュールである。加えて、メモリマネージャとして外部に公開しているインタフェースも含んでいる。これらのインタフェースのメモリ操作やメモリ管理の部分は aset.c で実装されていて、メモリコンテキストに定義している関数のポインタ経由で呼び出される。

[4] ALLOC_CHUNK_LIMIT 分のサイズを要求した場合、ヘッダ分のサイズがブロックに入らないため、ブロックを少し大きくする。

Copyright © 2004 NTTDATA

メモリコンテキストに対する基本的な操作は、次のようなものである。

+ メモリコンテキストの作成 [AllocSetContextCreate() を使用する]
+ メモリコンテキスト内部でのメモリ割り当て [malloc()相当 : palloc() などを使用する]
+ メモリコンテキストのリセット [メモリコンテキスト中の全メモリの free。メモリコンテキストは開放しない。]
+ メモリコンテキストの削除 [メモリコンテキスト中の全メモリの free+メモリコンテキストの削除。]

個別のインタフェースは、「メモリコンテキストのインタフェース」で説明する。

2.2.1. 階層型のメモリコンテキスト管理

多くのメモリコンテキストを扱うようになると、メモリコンテキスト自体の管理が複雑になってくる。PostgreSQL では、これを解決するために、ツリー構造による階層型のメモリコンテキスト管理を行っている。これによって、あるノード以下のサブコンテキストのメモリをまとめて開放するということができる。

次の図は、PostgreSQL のメモリコンテキストのツリーである。図のように TopMemoryContext をルートとした１つのツリーになっている。

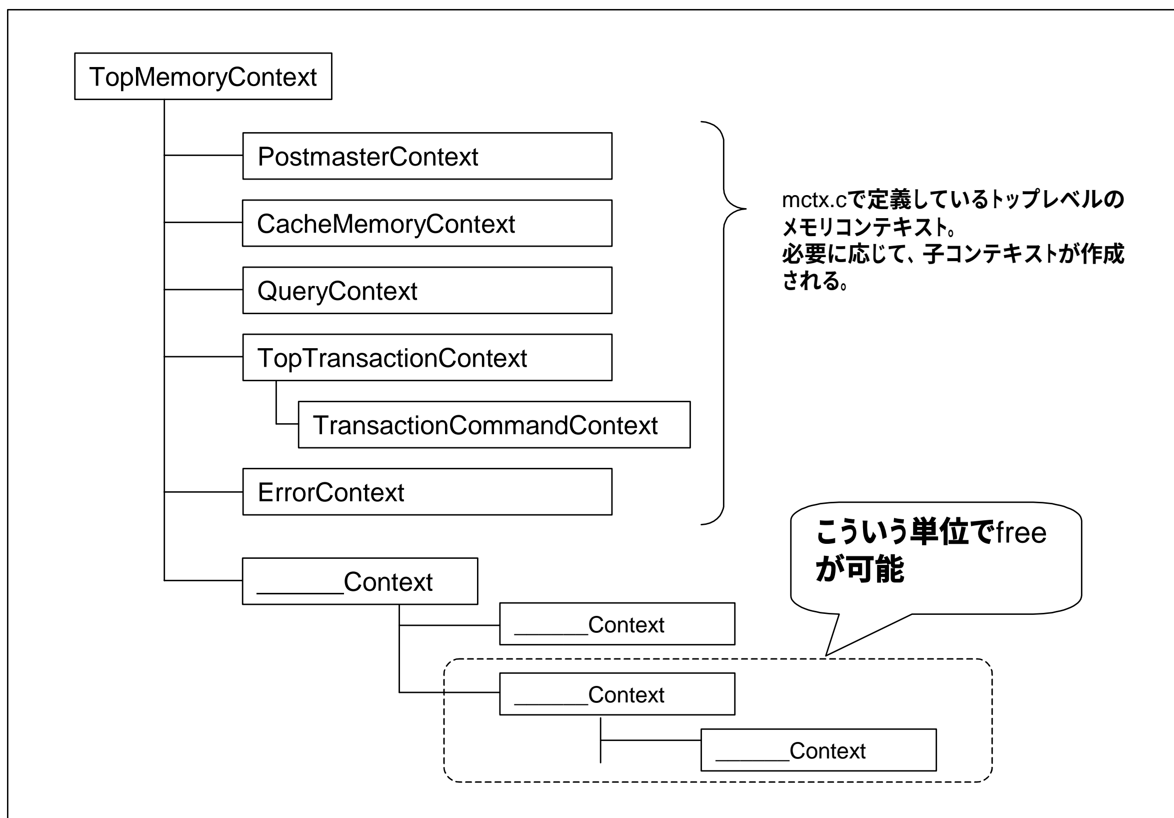

図 2.3 メモリコンテキストのツリー

2.2.2. トップレベルのメモリコンテキスト

PostgreSQL には、いくつかのトップレベルコンテキストと呼ばれるものがある。一番上のメモリコンテキストは TopMemoryContext １つである。この TopMemoryContext のすぐ下に作られている、

Copyright © 2004 NTTDATA

目的ごとのメモリコンテキストが、トップレベルメモリコンテキストと呼ばれている。これらのトップレベルメモリコンテキストは、グローバル変数で直接参照できるようになっている。
PostgreSQL のソースコード中に新しいモジュールを追加する際に、新しいメモリコンテキストを追加してもいいことになっている。しかし、エラーイベントなどの際のメモリリークを防ぐため、それらは全て、いずれかのトップレベルメモリコンテキストのツリーの配下に作らなければならない。

mctx.c 内部で管理している、トップレベルの主なメモリコンテキストには次のようなものがある。

TopMemoryContext	全メモリコンテキストのトップである。他のトップレベルのメモリコンテキストは、この下に作られる。他のトップレベルのメモリコンテキストの管理領域（ヘッダ用のメモリなど）を保持するために使われる。
PostmasterContext	postmaster の処理に使用されるメモリコンテキスト。 postmaster から fork されたバックエンドプロセスでは、このメモリコンテキストを削除する。
CacheMemoryContext	relcache, catcache などに使用されるメモリコンテキスト。
QueryContext	クエリの文字列やパーズツリーを保存するために使用される。 PostgresMain()のループが１回まわるごとに初期化される。
TopTransactionContext	トランザクションの終了まで使用するものを全て入れておくためのメモリコンテキスト。
TransactionCommandContext[5]	実際は TopTransactionContext の子なので、厳密にはトップレベルコンテキストではないが、グローバル変数で呼び出せるのでここに分類してある。クエリのプランニングや実行に必要なメモリのためのメモリコンテキストで、１文(ステートメント)ごとに初期化される。
ErrorContext	エラーリカバリのために使用されるメモリコンテキスト。

この他にも、その他のモジュールで定義されているトップレベルメモリコンテキストもある。(Ex. PortalMemory[6]：後述)

全てのメモリコンテキストの中で、TopMemoryContext だけが、親の指定が NULL で作成される。

```
TopMemoryContext = AllocSetContextCreate((MemoryContext) NULL, "TopMemoryContext",
                                          8 * 1024, 8 * 1024, 8 * 1024);
```

親の指定が NULL なので、TopMemoryContext の管理領域は、malloc() によって確保される。
その他のメモリコンテキストは、次のように TomMemoryContext を親にして作られる。

[5] 7.4 以降、TransactionCommandContextは、トップレベルコンテキストから外れている。
[6] 7.4 以降、PortalMemory は、PortalContextとしてこのモジュール内で定義されている。

Copyright © 2004 NTTDATA

```
ErrorContext = AllocSetContextCreate(TopMemoryContext, "ErrorContext",
                                     8 * 1024, 8 * 1024, 8 * 1024);
```

これらのメモリコンテキストの管理領域は、次のように TopMemoryContext の中に作成される。

```
node = (MemoryContext) MemoryContextAlloc(TopMemoryContext, needed);
```

ここで使っている MemoryContextAlloc()とは、メモリコンテキストを明示的に指定してメモリを確保する関数である。

2.2.3. カレントメモリコンテキストと palloc() / pfree() / repalloc() インタフェース

毎回メモリコンテキストを指定してメモリアロケーションを行うのは、コードを書く上で、非常に面倒である。これを解決するために、カレントメモリコンテキスト(CurrentMemoryContext) と呼ばれるグローバル変数が用いられる。palloc() を実行すると常にカレントメモリコンテキストからメモリが割り当てられるようになっている。従って、現在のメモリコンテキストが目的のメモリコンテキストでない場合は、palloc() を呼ぶ前に、MemoryContextSwitchTo() を使ってメモリコンテキストの切り替えを行う必要がある。
pfree() や repalloc() では、いちいちメモリコンテキストを切り替えるのが面倒になるので、それぞれのメモリがどのメモリコンテキストか(正しくはどのアロケーションセットか)を覚えていて自動的にそのメモリコンテキストで実行される。従って、pfree() や repalloc() を使うときには、メモリコンテキストを意識する必要がない。

2.2.4. メモリコンテキストのインタフェース

メモリコンテキストを操作するためのインタフェースは、次のようなものである。
メモリコンテキストの作成だけは、aset.c で定義されている AllocSetContextCreate() を使用する。

MemoryContextInit()	メモリコンテキストモジュールの初期化
MemoryContextReset()	コンテキストツリー内で割り当てた全てのメモリを開放する
MemoryContextResetChildren()	子コンテキストをリセットする
MemoryContextDelete()	メモリコンテキストの削除（子コネテキストも全て削除する）
MemoryContextDeleteChildren()	子コンテキストを全て削除する
MemoryContextResetAndDeleteChildren()	子コンテキストを全て削除し、自分はリセットを行う
MemoryContextStats()	名前を付けられたコンテキストの統計情報の表示
MemoryContextContains()	第２引数で与えられたメモリチャンクが、第１引数で与えられたメモリコンテキストに入っているかどうか調べる
MemoryContextCreate()	メモリコンテキストの管理領域 AllocSetContext 構造体を作成する。AllocSetContext 構造体の全体のメモリを確保し、MemoryContextData 構造体の部分の初期化を行う。

Copyright © 2004 NTTDATA

	外部からメモリコンテキストを作成するためには、AllocSetContextCreate() を呼び出す必要がある。
MemoryContextAlloc()	メモリコンテキストを指定して、メモリのアロケートを行う
pfree()	メモリを開放する
repalloc()	割り当てたメモリのサイズ変更を行う
MemoryContextSwitchTo()	カレントのメモリコンテキストを切り替える
MemoryContextStrdup()	メモリコンテキスを指定した、文字列のデュプリケート

ここで定義されている関数は、主にメモリコンテキストのツリー構造を操作するコードが書かれており、メモリコンテキスト内のメモリ操作に関しては、aset.c に定義されている関数を呼び出す。

ちなみに、palloc() は、palloc.h に次のように定義されており、カレントコンテキストからメモリを割り当てるようになっている。

```
#define palloc(sz)  MemoryContextAlloc(CurrentMemoryContext, (sz))
```

2.2.5. メモリコンテキストのデータ構造

メモリコンテキストのデータ構造は次のようになっている。

```
typedef struct MemoryContextData
{
    NodeTag     type;           /* identifies exact kind of context */
    MemoryContextMethods *methods;      /* virtual function table */
    MemoryContext parent;       /* NULL if no parent (toplevel context) */
    MemoryContext firstchild;   /* head of linked list of children */
    MemoryContext nextchild;    /* next child of same parent */
    char       *name;           /* context name (just for debugging) */
} MemoryContextData;
```

先に説明したアロケーションセットのヘッダ部分として使用され、アロケーションセットを階層構造で結びつける役割を果たす。
メモリコンテキストの種類を識別するための **type** の型(**NodeTag**)は node.h に次のように定義されている。

```
typedef enum NodeTag
{
    T_Invalid = 0,

        :

    /*
     * TAGS FOR MEMORY NODES (memnodes.h)
     */
    T_MemoryContext = 400,
    T_AllocSetContext,
```

Copyright © 2004 NTTDATA

```
            :

} NodeTag;
```

メモリコンテキストの呼び出すメソッドを、メモリコンテキストごとに指定できるように**methods** に関数ポインタ群を保存している[7]。methodsの型である**MemoryContextMethods** は、aset.c で次のように定義されている。

```
typedef struct MemoryContextMethods
{
    void      *(*alloc) (MemoryContext context, Size size);
    /* call this free_p in case someone #define's free() */
    void       (*free_p) (MemoryContext context, void *pointer);
    void      *(*realloc) (MemoryContext context, void *pointer, Size size);
    void       (*init) (MemoryContext context);
    void       (*reset) (MemoryContext context);
    void       (*delete) (MemoryContext context);
    Size       (*get_chunk_space) (MemoryContext context, void *pointer);
    void       (*stats) (MemoryContext context);
#ifdef MEMORY_CONTEXT_CHECKING
    void       (*check) (MemoryContext context);
#endif
} MemoryContextMethods;
```

このメソッド群は、aset.c で次のように定義されている。

```
static MemoryContextMethods AllocSetMethods = {
    AllocSetAlloc,
    AllocSetFree,
    AllocSetRealloc,
    AllocSetInit,
    AllocSetReset,
    AllocSetDelete,
    AllocSetGetChunkSpace,
    AllocSetStats
};
```

さらに、MemoryContextData 構造体の **parent**, **firstchild**, **nextchild** でツリー構造を形成する。また、**name** にメモリコンテキストの名前を保存している。

2.3. ポータルメモリ （utils/mmgr/potalmem.c）

ポータルメモリとは、カーソルを使うクエリを管理するための構造である。バックエンドプロセスは、declare cursor 文を見つけたら、PortalData の構造体を割り当てる。そして、クエリプランを立て、ポータルメモリにクエリプランを格納する。この時点では、クエリは全く実行しない。
その後、バックエンドが 「 fetch 1 from FOO 」というようなフェッチ文を見つけたら、ポータルメモリの管理テーブル（ハッシュテーブル）から "FOO" という名前を見つけ、格納しておいたクエリプランを取り出す。そして、実行結果の取り出し数 count を 1 としてエグゼキュータを呼び出す。エ

[7] 現状のPostgreSQLでは、aset.cで定義されている関数しか呼び出さないため、メモリコンテキストごとに呼び出す関数が異なることはない。

Copyright © 2004 NTTDATA

グゼキュータは、クエリを実行し、１タプルの結果を返す。
問題は、close が実行されるまで、そのクエリの状態を保持しなければならないことである。これは、注意してメモリ管理を行わなければならない。
ポータルメモリのモジュールでは、ポータルメモリ用のメモリコンテキスト("PortalMemory") を作成して使用する。"PortalMemory" は、トップレベルのメモリコンテキストの１つである。

2.3.1. ポータルメモリのインタフェース

ポータルメモリのインタフェースは、次のものがある。用意されているのは、ポータルメモリ管理モジュールの初期化とポータルメモリの生成／削除／設定などである。

EnablePortalManager()	backend startup 時に、ポータルメモリ管理モジュールを初期化する
GetPortalByName()	引数 name に与えられた名前で、ポータルメモリを取り出す
PortalSetQuery()	クエリプランをポータルメモリに格納する
CreatePortal()	新しいポータルメモリを生成する
PortalDrop()	ポータルメモリの削除
AtEOXact_portals()	カレントトランザクションで作成された全てのポータルメモリを削除する

EnablePortalManager() で、ポータルメモリ用のメモリコンテキストとハッシュの領域が準備される。ポータルメモリを生成するときは、ポータルメモリ用のメモリコンテキストが使用される。

2.3.2. ポータルメモリを実現するために使われるマクロ

前述のインタフェースを実現するために、次の３つのマクロが使われている。これらは、ハッシュテーブルに、ポータルメモリを登録／削除したり、ポータルメモリを検索したりするために使用される。

PortalHashTableLookup(NAME, PORTAL)	NAME をキーにして、ハッシュから PORTAL を検索する
PortalHashTableInsert(PORTAL)	ハッシュテーブルに PORTAL を入れる
PortalHashTableDelete(PORTAL)	ハッシュテーブルから PORTAL を削除する

2.3.3. データ構造

ポータルメモリデータの構造体は次のように定義されている。

```
typedef struct PortalData
{
    char         *name;          /* Portal's name */
    MemoryContext heap;          /* subsidiary memory */
    QueryDesc    *queryDesc;     /* Info about query associated with portal */
    TupleDesc     attinfo;
    EState       *state;         /* Execution state of query */
    bool          atStart;       /* T => fetch backwards is not allowed */
    bool          atEnd;         /* T => fetch forwards is not allowed */
```

Copyright © 2004 NTTDATA

```
    void        (*cleanup) (Portal);    /* Cleanup routine (optional) */
} PortalData;
```

１つのポータルメモリで使用するメモリは、**heap** に新しいメモリコンテキストを生成して、そのポータルメモリ専用のメモリコンテキストからメモリを割り当てる。クエリの情報は、**queryDesc** に保存されている。クエリの実行状態は、**state** に保持されている。フェッチを許す方向を示すために、**atStart** と **atEnd** という２つのフラグが用意されている。

ポータルメモリのデータ構造とは、PortalData 構造体のデータをハッシュテーブルで管理しているだけである。

<EOF>

Copyright © 2004 NTTDATA