# gm\_create\_space，

会构造SQL语句，调用GMDB的接口去创建space。

# gm\_create\_table,

三个字段：KEY, VALUE, VERSION。会调用dmLoadFromJsonString创建一个schema。这个schema由jansson库从字符串创建，之后加载到gInstance.SchemaPoll.MemoryPool中去（存储时先创建一个MemoryContext, 这个结构可以将MemoryPool中的几个Page组织在一起。然后，从MemoryContext中分配内存给指定的对象。详见后面的MemoryPool原理）。并把其名称Hash之后插入到ObjectSchema对象指针数组的链表中去。（对于冲突的名称，即hashId相同的，用ObjectSchema组成链表来解决冲突。）

* Q: 这个schema放到哪里去了？

gInstance表示GMDB的一个实例(每个进程一个)，里面包含了instLock，包含所有的sessions和一个schemaPool。SchemaPool的存储由MemoryPool实现，SchemaPool包含一个ObjectSchema\*指针数组，所指的ObjectSchema对象就是从MemoryPool中分配， MemoryPool中的buffer由malloc根据大小来由tree工程的gaCreatePool<dmInitTreeInst< gm\_server\_init< gmdb\_init< gm\_startup函数来分配，这个函数实现了内核设计文档所描述的page式管理，一个MemoryPool包含Page大小，总Page数量，空闲Page信息，buffer指针等连续内存的Page通过PageHead实现链式管理。通过dmLoadFromJsonString，从字符串变成内存中的ObjectSchema对象(所以在程序中，schema用数据结构ObjectSchema表示/抽象)，被保存在SchemaPool.MemoryPool中。其通过一个SchemaMap进行查找。存储形式如下：

上图中，最左边表示ObjectSchema\*指针数组schemaNameMap，而所有浅蓝色的方框表示名称的hashId相同的ObjectSchema对象（ObjectSchema.tableName），所有带阴影的方框表示相同名称但是不同version的ObjectSchema对象。以上就是schema的组织方式。在外部只要通过ObjectSchema \* dmGetSchema(const gm\_char \* tableName, gm\_uint32 versionId)就可以精确获得schema对象。

schema字符串被存到SYSTREE表里面了。

ddInsertKVMeta，将插入如下元数据。

把表相关信息插入名为 SYSTREE 的表中，含四个字段 TABLE\_NAME，ORDER\_NO，SCHEMA\_TEXT，VERSION，分别是*字符串，整数，二进制，整数*类型。

* + Q: SYSTREE表是什么时候创建的？

调用函数从内到外分别是：ddConstructTreeCreate< ddInitSysTree< ddInitSchema< gm\_server\_init< gmdb\_init< gm\_startup

也就是说在启动时就已经创建了这个表了。

## 专题：MemoryPool原理

MemoryPool按照Page管理。整个MemoryPool的buffer可以看做一个Page数组，每个Page的索引号就是PageId。每个Page包含一个PageHead，PageHead包含了本Page中尚未分配的空闲地址的首地址allocPos，和一个next整数，表示下一个空闲Page的索引号。通过这样可以串联其所有的空闲的Page。使得在内存结构上连续的Pages在逻辑结构上是链式结构，在编程时，比较高效。在MemoryPool结构体中还有一个辅助结构体freePages，它包含first，last空闲Page索引，还有空闲Page数count。它是Page组成链表结构的关键所在。first表示空闲Page链表的首个Page节点，last则“指向”链中最后一个Page节点。

要从MemoryPool中分配内存，首先要创建一个MemoryContext，这个东西可以管理多个Page。Page也是链式的逻辑结构。在释放时也需要按照MemoryContext来释放。当MemoryContext中的内存不够时，会自动扩展。在有了MemoryContext之后，就可以从中分配内存给指定的对象了。每次分配指定大小的内存了。分配到的内存需要是连续的，不能夸Page，否则完整的对象分割裂。这样使得对象不能大于一个Page最大空闲尺寸（一个MemoryContext中的首个Page还因为要存储MemoryContext结构体而比其他Page小。）当一个Page无法提供所需容量时，直接在下一个Page分配，这使得当前Page中的空闲容量全部被浪费。

在以上描述的机制下，释放内存只能按照MemoryContext为单位进行，不能释放某个指针。

这样可以减少内存释放次数，间接减少内存泄漏的可能性。

## 专题：Schema的操作：ObjectSchema数据结构及其操作

struct ObjectSchema

{

gm\_char tableName[GMDB\_NAME\_BUFFER\_SIZE];

MemoryContext \* memory;

MemoryArea exprMemory; // memory for expression

SchemaNodeMap nodeMap;

gm\_uint32 tableId;

gm\_uint32 versionId;

LoadFrom loadFrom;

SchemaNode \* currentNode; // load schema

SchemaNode \* rootNode;

List \* nodeList; //存储所有的record类型的节点，并给其中的节点分配id。

gm\_uint32 indexCount;

SchemaIndex \* indexs;

ObjectSchema\* parentObjSchema; // for sub schema

ObjectSchema \* nextVersion; // for hash map

ObjectSchema \* nextTable;

};

其中一个关键函数：dmCreateSchemaFromJson (const gm\_char \* tblName, gm\_uint32 tableId, gm\_uint32 versionId, json\_t \* jsonRoot, ObjectSchema \*\* schema)

在调用该函数时，schema的json字符串已经通过jansson库的json\_loads加载到内存，返回了根指针jsonRoot。

该函数的主要工作是先初始化NodeList，NodeMap，然后根据json结构去填充NodeMap和NodeList结构。其中NodeList是List数据类型在分配内存时，根据给定的内存分配函数从给定的地方分配内存。我们这里被设置为从ObjectScheme.MemoryContext中分配内存，即从gInstance.SchemePool.MemoryPool中分配内存。与ObjectScheme存在于同一个MemoryContext的管理中。

看dmInitNodeMap(SchemaNodeMap \* map, ObjectSchema \* schema, gm\_uint32 id, gm\_uint32 size)：

struct SchemaNodeMap //unique hash index

{

gm\_uint32 id;

gm\_uint32 size;

SchemaNode \*\* items;

};

它首先从其绑定的MemoryContext中为ObjectSchema.SchemaNodeMap.items分配NODE\_MAP\_SIZE（实际值为128）个sizeof(SchemaNode \*)长度的数组。也就是说，分配了一个SchemaNode\*指针数组。其size指示数组长度。id的作用不明。**这个指针数组作为哈希桶**。

dmLoadFromJson(ObjectSchema \* schema, json\_t \* jsonRoot)

dmExtractJsonSchema(ObjectSchema \* schema, const json\_t \* jsonRoot)

dmExtractJsonNode(ObjectSchema \* schema, gm\_uint32 fieldId, const json\_t \* jsonNode)

上面三个函数前一个调用下一个，最下面这个包含了从json建立schema的逻辑，是一个递归函数。

**这里需要参看内存数据库设计文档，里面很详细地描述了schema在内存中的存储方式。还有图解。**

通过dmExtractJsonNode和dmExtractRecord(尤其是后者)函数，可以构建出以树状存储的Node树，子节点使用SchemeNode中的SchemaNodeMap-filedMap来存储，也同时存储在父节点的fileds数组中。而schema中的NodeMap存储了所有的record类型的节点。record类型的节点才能在fields中携带filed信息（和子节点信息）。所以通过NodeMap中的所有的节点能得到所有的filed。

另外，还有一个schema.NodeList，存储了所有的record节点。record类型的Node还通过fields成员存储了所有的非record子节点，即真实的存储数据的field。

struct SchemaNode

{

//For a field in record node

gm\_uint32 id; // 节点加入NodeList的次序（=List.itemCount）似乎只有record类型的节点才有？

gm\_type\_t type;// 节点类型，如record, string/uint8等

gm\_uint32 fieldSize;//字段本身的大小，对于str/bytes类型，如果非固定大小，则大小为一个offset变量的sizeof，offset的值才是真实大小。对于str/bytes类型，必须在json中显式指定其size，才能成为固定大小的字段。

Text name;

gm\_bool nullable;

gm\_bool isFixedSize;

gm\_char \* upreadStr;

gm\_uint32 fieldId; //id in parent record,是第几个field。

gm\_offset\_t offset; //offset in parent row以父节点为起点的offset，程序中被设为该节点添加前父节点的dataSize。

OmlExpr\* expr; // support function for default value

gm\_value\_t defaultVal; //default value，多种数据库类型。

gm\_bool isKey;

gm\_uint32 keyId; // index for the key field. Only valid when isKey = true.

以下三个成员是在分配新的节点时填充的。在分配时，schema.currentNode指向的是父节点，分配的新节点是子节点，所有这三个东西是在父节点的基础上进行操作的：

gm\_uint32 depth;//父节点的值+1

gm\_uint16 path[SCHEMA\_MAX\_TREE\_DEPTH];//先填0，然后从父节点复制depth个元素，填充第depth个元素为filedId，也即该节点在其兄弟节点中的相对位置，从0起。所以根节点的第一个子节点的path为[0]，第三个子节点为[2]。而第三个子节点的第2个孩子节点，其path为：[2,1]。这种事我以前也做过。要从path找到节点，只需要从根开始，一层层向下，第n层取第path[n]个子节点。

gm\_uint16 pathNode[SCHEMA\_MAX\_TREE\_DEPTH];//这个与path类似，但是里面的数据不是fieldId，而是节点的id。即节点在加入NodeList时的序号。对于根节点的首个子节点，其pathNode为[1], 若根有两个子节点，只有第一个是record，那么这第一个record节点的首个record节点的pathNode将会是[1,2]

分配完成后，currentNode指向子节点，在子树全部操作完成后，currentNode回到父节点，以处理下一个兄弟节点。

// for record node only

gm\_bool isArray;

gm\_uint32 dataSize; //record data size，所有子节点的filedSize之和，

gm\_uint32 fieldCount; //json中”fileds”数组元素个数

gm\_uint32 keyCount;//

gm\_uint16 bitmapOffset;

SchemaNode \*\* fields; //all fields, for record type，指针数组方式存放了所有的直接子节点。

SchemaNodeMap fieldMap;// Map的方式存放了所有的直接子节点。

ObjectSchema \* objSchema;

ObjectSchema \* subSchema; // for sub schema

SchemaNode \* parent;

SchemaNode \* hashNext[SCHEMA\_MAP\_COUNT]; // for hash map, 在fieldMap中解决冲突时，需要为相同hashId的子节点建立链表，用到该数据结构。

};

Q: subSchema是个什么概念？

# object内的操作：

gm\_int32 gm\_get\_node(gm\_object\_t \* object, const gm\_char \* name, gm\_path\_t\* path, gm\_node\_t \* node);

通过节点的名称获取节点结构。实际会先在对象关联的schema中找到对应的schema节点。这里path是输出。

它最终调用：这个接口： **dmGetNode**(Object \* **object**, SchemaNode \* **schema**, gm\_path\_t\* **path**, ObjectNode \* **node**, gm\_bool **fillPath**)

dmAllocObjectMem()如下：

static gm\_char \* dmAllocObjectMem(Object \* object, gm\_uint32 size, gm\_offset\_t \* offset)

dmExtendObject(object)

object的内存是从其object handler中分配而来的，默认固定有64KB（作为成员数组，不是动态分配的），如果不够，另外动态分配malloc。

主要是对于变长类型的字段需要相应的处理，定长的字段直接就可以存储进去。

这两个函数对于理解object的存储结构非常的关键。

**gm\_int32 gm\_create\_object(gm\_handler\_t handler, gm\_schema\_t schema, gm\_object\_t \* object);**

**gm\_int32 gm\_create\_child\_node(gm\_node\_t \* node, gm\_uint32 id, gm\_node\_t \* child);**

gm\_create\_object，外部需要传入有效的object对象的地址。然后，其最重要的两个属性：buffer/buffer\_size，被定位到handler的固定buff处（每个handler自带64KB的初始固定buffer）。然后，把buffer起始地址解释为ObjectHead结构体，

struct ObjectHead

{

gm\_uint32 size;

gm\_uint32 versionId;

gm\_uint32 tableId;

gm\_uint32 recycleSize;

NodeLink nodeHead;

gm\_offset\_t subObjInfo;

};

然后填充这个结构体，其中size表示已使用的量。ObjectHead中有个nodeHead，表示下一个节点（根节点）的offset和fieldCount。根节点会被分配其已知数据大小（dataSize）的内存（其实就是在object的buffer中分配）。然后调用dmInitNodeData，在后续为rootNode分配内存。这里展示了一种设计，通过本地变量管理内存。即rootNode其实是一个局部变量，它包含多种指针数据，通过它，dmInitNodeData能知道schema的root\_schema\_node在哪里，从而为各个字段分配固定大小的内存（非固定大小的字段分配2字节偏移量），初始化。这个分配动作是在object.buffer中进行的，从而使得这段内存就代表节点本身。其他的节点不是这样的，只有根节点能这样搞，因为根节点的头部信息可以很容易找到。

gm\_create\_child\_node，它需要给定父节点，给定要创建的子节点的index。然后从schema中找到对应的schemaNode，它必须为record类型的节点。子节点的首地址被解释为nodeHead，这是与根节点不同的地方，根节点的nodeHead信息存储在objectHead里面。可以说objectHead是一个特殊的被扩展了的nodeHead。在构建schema时，对于record类型的节点，初始的dataSize只有几个字节，在构建起子节点的过程中，其dataSize的值会加上子节点的fieldSize值，而子节点的offset就是加上它的fieldSize之前dataSize的值。这就把各个field依次落到了连续的内存上面，每个都有相对父节点的offset。对于record类型的节点，fieldSize只有NodeLink的大小，也就是说，如果子节点是一个record类型的节点，父节点只会记录它的一个指针。

这样，在create\_child\_node时，（只有record节点能在object中作为节点，也就是受create\_child\_node的node一定是record节点，其fieldSize一定是sizeof(NodeLink)）所以parent object node通过child object node的NodeLink中的offset和fieldCount，就知道了child object的数据了。

换句话说，父object节点在某个field上创建子节点，该field一定要对应record schema node，而在父节点的数据中，该field上存储的是NodeLink，待创建的子节点的NodeLink。

而子节点的link\_addr，就是指向了自己的NodeLink—父节点中的一个field。

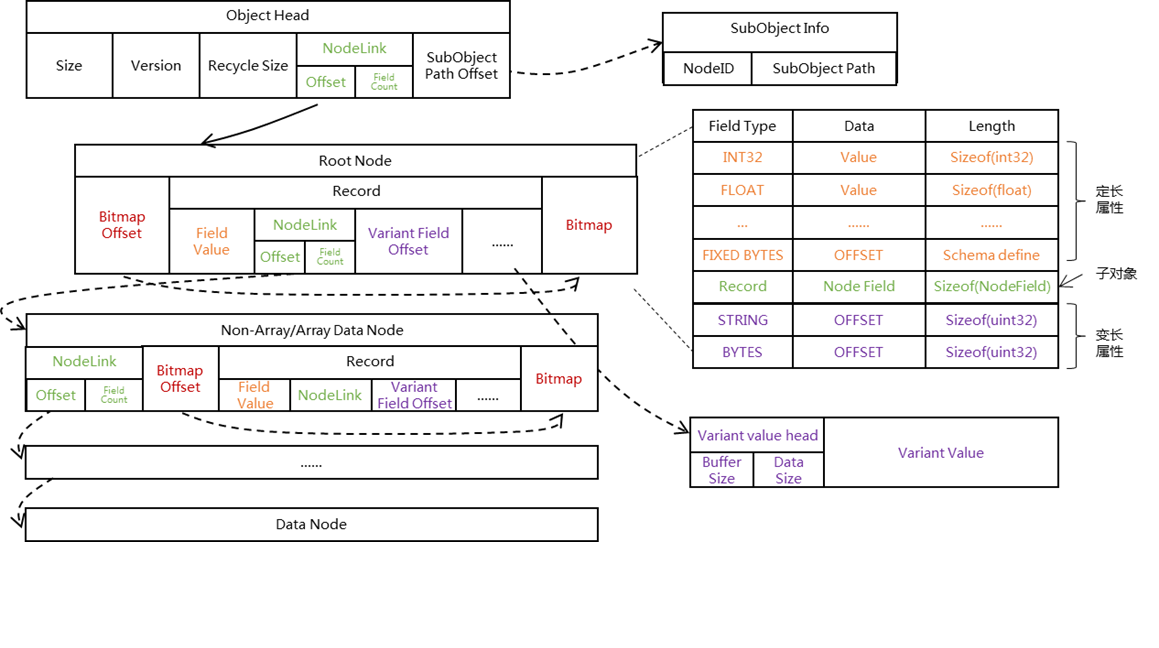
Q：一个Node的link\_addr是干嘛的？上面已答。

在为子节点分配其dataSize个内存之后，就要初始化这段内存，其中包括：NodeLink（仅数组节点），bitmapOffset, 各字段固定宽度，包括对于record子节点的NodeLink，bitmap字节。其中NodeLink要清0，因为系统靠检测其offset是否为0来判断是否重复创建子节点。

Q：NodeLink中的offset似乎没有使用？在源代码中看到，object head中的fieldCount可以不等于schema中的filedCount。例如在取值的时候，如果所取field id大于node的field count，则取默认值。可是什么情况下field count会是一个小余的情况呢？fieldCount是用于支持升级的，如果Schema有新加的列，在旧数据上记录下原来的fieldCount就能知道是不是有新加列，然后对这些列取默认值。

gm\_append\_into\_array，向一个array节点中，添加一个node节点。

下图是思南给我发的图，与我自己总结的基本相同，除了SubObjectInfo。这个SubObjectInfo是用于支持树的划分的，之前我们讲过，用户的动态数据和静态数据的问题，动态数据需要频繁更新，将它单独作为一棵树在存取时就只需要存取这部分而不用存取整棵大树。TAG\_SUBOBJECT\_USAGE: 如果一个对象的SubObjectInfo有数据，表示这个Object是属于某棵子树的。通过NodeId+objectSchema.NodeList链表就能找到与本Object对应的那个Record节点，这个节点才是本Object的根节点对应的Schema节点。



在进行object的整体操作（create/get/put等等）时，需要用到一个空闲的handler。一旦一个handler被绑定到一个object上，它就不能用于别的object操作。除非free handler或者destory object。

## 专题：Object内存回收。

当我们不断的插入数据，更新数据，删除数据之后，Object的buffer中会产生很多的空隙。我们会把这些空隙记录下来，并在put、update Object时，检查空隙比例有多少，如果达到了一定的比例，就重新组织这个object再进行put、update。在ObjectHead中有两个参数size, recycleSize，如果recycleSize/size>0.5，就会导致重新组织。

## Tip调试技巧

由于我们的object，node仅仅是一段内存，没有对应的结构体，而是需要依赖Schema来解释，使得程序调试时，我们无法直接检查object/node的值。但是我发现在VisualStudio的Watch窗口中可以调用函数，而我们有函数可以将object/node打印成文本。所以我就编写了一个函数，这个函数接受一个node参数，返回一段静态内存地址指针，我将node通过我们的函数打印成文本存放到我们的静态内存中，这样就可以看到node被翻译后的数据了。这个技巧对复杂数据的查看非常有用。而且VS中，只有我们按下刷新按钮，才会调用该函数，而不像变量一样实时刷新，这使得我们可以很好地控制调用时机，防止参数不对，运行环境不对等导致崩溃。

这样的函数最好不要改变程序的运行状况，例如不要修改变量等。

另外一点

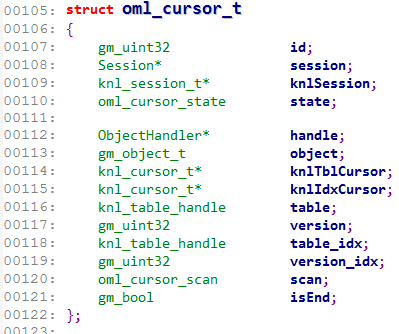
# 索引

索引的实现就是单独的一张索引表。首先会在SYSTREEINDEX系统表中创建一条记录，表示某些表的某些节点存在某些索引（KEY，FIELD\_1/2/3/4）。不用关心具体的。只需要知道存有元信息。然后在KV空间创建一张索引表，用KEY+FIELD的方式，创建之后，给FIELD创建联合索引，以后就能直接查找这些联合索引了。由于有索引，联合索引的查询会快些。

## select object的支持

select object需要到表中去查找，可能可以使用到索引。有执行计划来辅助。

这种语句的使用方法是先prepare，再open\_cursor，再fetch\_cursor，而不是execute\_oml



struct index\_plan

{

MemoryArea stack;

gm\_char stackBuffer[IDNEX\_PLAN\_STACK\_SIZE];

index\_scan\_range\_set rangeSet[MAX\_INDEX\_FIELD];

gm\_uint32 idxId[MAX\_INDEX]; //the index can be used

gm\_uint32 idxIdCnt; //inner variant for record the index id(idxId[MAX\_INDEX]) count

gm\_uint32 idxFieldCnt; //condition field count which is used index

SchemaIndex \* index; //the index chose at last

gm\_int32 order; //index order

oml\_cursor\_scan scan;

};

open\_cursor会创建cursor，创建plan,上面是plan的结构。对range\_set结构如下，

struct index\_scan\_range\_set

{

index\_set\_type setType;

gm\_uint32 setPos; //the last range position in list

cm\_list\_t ranges; //range list for one column

};

enum index\_set\_type

{

INDEX\_EMPTY\_SET = 0,

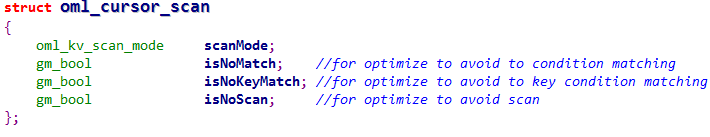
INDEX\_FULL\_SET = 1,

INDEX\_NORMAL\_SET = 2,

INDEX\_LIST\_SET = 3,

INDEX\_LIKE\_SET = 4

};



enum oml\_kv\_scan\_mode

{

OML\_KV\_SCAN\_INVALID = 0,

OML\_KV\_SCAN\_SEQ = 1,

OML\_KV\_SCAN\_INDEX = 2,

OML\_KV\_SCAN\_KEY = 3,

};

经过一些试验发现：

* + - 1. 如果数组节点时索引，那么每个数组元素都会在索引表中，每个符合条件的元素都会导致该对象被取出一次。例如我orders.f有两个元素1， 2，那么select object from table filter orders where f> 0将返回该对象两次。外部看起来会有些奇怪，疑惑其为什么被返回两次，其实是这样个原因。
      2. 对infinite的处理有些问题，例如int32\_inifinite用int32最大值表示，即2147483647。但是取边界时存在有些问题，即认为无限大的那一边应该是开区间，其实应该是闭区间。

### 采用索引接口新建后

# 启动与系统表结构

启动流程：如果不存在data/redo中的文件，那么就会创建系统表。据此我们可以看到系统创建了那些表，这些表有什么字段：

**表：SYSTREE**

**所在space：USER**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段 | **TABLE\_NAME** | **ORDER\_NO** | SCHEMA\_TEXT | **VERSION** |
| 类型 | VARCHAR(32) | INT | BINARY(?) | INT |
| 索引 | Y | Y | N | Y |

会为**SYSTREE**创建联合索引

**索引名：IDX\_SYSTREE\_TBLNAME**

**所在space：USER**

**表：SYSTREEINDEX**

**所在space：USER**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字段 | TABLE\_NAME | INDEX\_NAME | NODE\_NAME | FIELDS | UNIQUE |
| 类型 | VARCHAR(32) | VARCHAR(32) | VARCHAR(32) | VARCHAR(131) | INT |
| 主键 | Y | Y | / | / | / |

可以看到主要是两个元数据表。一个是SYSTREE，另一个是SYSTREEINDEX。前一个记录了数据库中有那些表。后一个记录数据库中有哪些索引。前者似乎没有指定主键。后一个的主键是联合主键，由table\_name和index\_name唯一指定。

**SYSTABLES表记录了数据库中有哪些表。**

启动时比较耗时的动作在于sql\_start\_lsnr()函数的调用.

共享内存的分析：首先会分配一大块共享内存，然后在其上创建view，以分配小部分共享内存，view都记录在一张表中。这样，view的指针有时候不需要释放，因为在整块大共享内存释放前，会扫描view表释放所有view。

# Node操作

gm\_get\_node

原型：

gm\_int32 gm\_get\_node(gm\_object\_t\* object, const gm\_char \* name, gm\_path\_t\* path, gm\_node\_t \* node)

它会调用以下函数：

gm\_int32 dmGetNode(gm\_object\_t \* object, const gm\_char \* name, gm\_path\_t\* path, gm\_node\_t \* node, gm\_bool fillPath = true);

注意这个函数最后一个参数默认为true，也就是说，会填充path，也就是说这个path是动态填充的，不是外面提供的。是个输出参数。它首先通过名称找到对应的SchemaNode（显然是唯一的，对应于Object中的Node，是record类型的schemaNode）

Q:在函数gm\_int32 dmGetRootNode(Object \* object, ObjectNode \* root, gm\_bool fillPath)中，为什么有以下逻辑：

if (NULL\_OFFSET != objHead->subObjInfo)

{

subObjInfo = (SubObjectInfo\*)(object->buffer + objHead->subObjInfo);

root->schema\_node = dmGetSchemaNode(objSchema, subObjInfo->nodeId);

}

else

{

root->schema\_node = objSchema->rootNode;

}

这个逻辑见本文档中TAG\_SUBOBJECT\_USAGE。目的是找到root\_object\_node真正对应的schema\_node(可能对应的是schema中某棵子树的根，而不是整个schema的根)

if (!schemaNode->isArray) then

object\_node.path.items[depth] = NULL

else:

object\_node.path.items[depth] = path\_node, 其中path\_node.items[]是所有的isKey的fields，保存了field中的value，及field对应的schemaNode。

另外如果有basePath，还会复制basePath的内容。可是array和子field有什么关系？is\_key又和它有什么关系？

Array类型的节点，必须要有1~5个key field，这个与index不同，index是动态创建，可以指定某个record节点及其下需要被index的属性field，不是记录在schema中。is\_key是用于指定哪些field用来区分array中各个object node的。比如products指定product\_id为is\_key, 那么所有products节点的product\_id都要求不同。那么一个array中任何两个节点的is\_key必须不同（也可以是联合的几个field作为key）。

而一个节点的path记录着：

* + - 1. 每个path由多个path\_node组成（用指针指向，指针数组）。每个path\_node对应的schemaNode也由path来记录。
      2. 对于每一层深度，有一个path\_node元素用于记录。
      3. 如果该层上的schemaNode is not array, 那么该path\_node元素指针为NULL。所以只有在该层为数组型时，才会实际创建path\_node。path\_node结构如下：

typedef struct

{

gm\_node\_key\_item\_t items[GMDB\_MAX\_NODE\_KEY\_NUM];

gm\_uint16 count;

}gm\_path\_node\_t;

其中gm\_node\_key\_item\_t结构如下。

typedef struct

{

void \* schema;

gm\_value\_t val;

}gm\_node\_key\_item\_t;

总结来说就是path由path\_node组成，path\_node由node\_key\_item组成，node\_key\_item记录着一个node中作为key的field的值。因此，要通过名称找到一个节点，除了需要名称外，还需要path，以便在通向目标节点的路上遇到array类型的节点时，能确定是array中哪个节点。对Object来说，每个node都是record node，都需要一个path。

gm\_int32 gm\_append\_into\_array(gm\_node\_t \* array, gm\_node\_t \* node)是创建一个新的节点并追加到array中.这个新节点的key都是默认值。这就要求，数组中所有的元素都需要不能是默认值，否则冲突。所以一个好习惯是在添加之后立即修改key fields的值。

ObjectNode结构体：

typedef struct

{

gm\_object\_t\* object;

gm\_char\* data;

gm\_void \* schema\_node;

NodeLink \* link\_addr;

gm\_char \* next;

gm\_char \* prior;

gm\_path\_t path;

}gm\_node\_t;

typedef gm\_nodt\_t ObjectNode;

这个结构体是作为一个object node的操纵杆、窗口、遥控器，是从Object中取节点时动态构建的，而不是存储在Object中。

其中的link\_addr成员我之前一直以为是指向节点存储区域，实际不是的。link\_addr实际是指向父节点的中对应该孩子节点的字段。即如果node.fields[3] is child, 那么get child node返回的ObjectNode的link\_addr就是指向node.field[3]的，对数组节点稍有不同，数组中后一个元素的信息由前一个元素记录而非父节点记录，只有首元素由父节点记录。节点的存储区域由data指定，节点的next/prior指向数组类节点的其他元素的data区。经过证实（node\_to\_json打印元素）发现append\_array实际上是在所选节点的前面插入一个元素。父节点中的record node 的 field是NodeLink结构，它作为头指针，指向数组首个元素，而各个元素头部有NL结构，指向下个数组元素，通过这样层层递进，构成一个单链表数组。

## path的使用

通过create\_path可以创建出一个指定深度的path。每个（其实只是填充一下path的path\_node指针数组，使其前depth个元素为NULL）

注意只有record类型的节点才会增加object node的深度。path的每个元素都对应一个深度。但是只有array类型的节点才需要用path去定位。所以path中的第n层上的元素en，只有在第n层上为array类型时，才需要通过add\_path创建一个path\_node。

通过add\_path,可以向path中添加指定深度添加path\_node，并向path\_node中添加key field的值。

当所有key field被添加后，就可以通过key fields的值的组合，唯一确定数组中的元素。

通过create + add path,就可以手动创建出一条确定的path，再加上节点对应的schema节点，就可以定位到指定的object节点了。

在得到一个具体的object节点之后，也可以通过get\_path获得已经填充好的节点的路径。而要先获得节点，可以通过get\_root\_node, get\_child\_node, get\_array\_next\_node得到。

通过key field的方式比起通过数组index的方式，在元素做了整体删除后，还有用。但是如果key field的值变了，那么path也就报废了，必须重新获取。

在获取子节点是，会从父节点拷贝path，再添加本层path。注意到path中的path\_node数组实际上是一个指针数组：

typedef struct

{

gm\_path\_node\_t \* items[GMDB\_MAX\_TREE\_DEPTH];

void \* schema[GMDB\_MAX\_TREE\_DEPTH];

gm\_uint32 depth;

gm\_bool isReadOnly; /\* true if the path is obtain from node\*/

}gm\_path\_t;

所以在拷贝是，其实也只是拷贝了指针而已，path\_node本身是被复用的。这样就使得每个array中的元素节点都对应着一个唯一的path\_node，由keys定位。

# oml与cursor操作

execut\_oml是在指定对象上面执行oml语句。

gm\_int32 omlPrepare(Statement \* stmt, ObjectSchema \* schema, const char \* oml)

Statement结构体如下：

struct Statement

{

gm\_uint32 id;

Session \* session;

OmlContextData \* omlContext;

ObjectSchema \* omlSchema;

gm\_char \* omlData;

MemoryArea omlArea;

OmlParam \* omlParams;

gm\_char omlBuffer[STMT\_OML\_BUFFER\_SIZE];

OmlExecutor \* executor;

gm\_bool isPrepared;

gm\_bool matchFound;

gm\_bool isPending; /\* object or delta is created/changed, but not submitted \*/

};

其中三个重要的尚未熟悉：OmlContextData，MemoryArea，OmlExecutor。现在分别分析这三个。

struct OmlContextData

{

gm\_uint32 size;

gm\_uint16 schemaNodeId;

gm\_uint16 stmtType;

gm\_bool isScan;

gm\_uint16 paramCount;

OmlOffset stmtContext;

OmlOffset filterMaxDepth;

OmlOffset filterMaxDepthNodeId;

OmlOffset filters[GMDB\_MAX\_TREE\_DEPTH];

OmlOffset keyFilter;

MemoryArea\* omlArea; 可能与Statement公用omlArea.

};

struct MemoryArea

{

gm\_char \* buffer;

gm\_uint32 bufferSize;

gm\_uint32 heapPos;

gm\_uint32 stackPos;

gm\_uint32 stackDepth;

gm\_uint32 stackOffsets[GA\_MAX\_STACK\_DEPTH];//栈中各元素指针。

};

从这个MemoryArea的成员来看，它似乎是分配一段内存，然后把这段内存的低段作为栈高段作为堆（或相反）。然后它记录了栈深度，栈中各元素offset和堆指针。经过验证发现，栈为高地址向低地址，堆为低地址向高地址。使用gaAllocMem从堆上分配内存，注意由于MemoryArea没有记录每次从堆中分配的内存的信息，所以无法释放和复用。好处是最后一次释放内存就可以了。

struct OmlExecutor

{

gm\_void \* session;

MemoryArea stack;

gm\_char stackBuffer[OML\_STACK\_SIZE];

Value \* keyVal;

gm\_char \* omlData;

SchemaNode \* omlSchemaNode;

Object \* omlObject;

ObjectNode \* objectNode; /\* current object node \*/

gm\_uint32 rownum;

gm\_bool isEof;

ObjectNode nodes[GMDB\_MAX\_TREE\_DEPTH];

};

这三个之中并没有更复杂的结构体了。

表达式在Parse时，前段将填充context中这部分信息：stmtType，isScan, schemaNodeId(对于select \*)。后面的都是filter。有key filter（select object from table\_name **where ...**）和独立Filter...。

重点在与Filter的存储方式。

在Statement.session（Session）中有omlArea，Statement中也有omlArea。OmlContextData中仅有\*omlArea指针。

## Lexer：词法解析（以下内容也包括语法解析）

根据以往的经验，词法解析类一般都是用状态机。设计和实现都很明确简洁。使用状态机加状态图是不是会好理解和设计一些呢？现在的设计方法是何种类型呢？有没有文档呢？

是一种“分层处理”的方法，也很好理解。

### struct OmlLexer

{

Text text;

gm\_uint32 stackDepth;

Text stackTexts[MAX\_LEXIAL\_STACK\_DEPTH];

gm\_bool isFuncMode;

};

这里必须先理解Text这个字符串处理结构。

struct Text

{

gm\_uint32 len;

gm\_char \* str;

};

这个小结构在某些情况下能极大地简化字符串操作，尤其是在**字符串划分**的时候有奇效。它在划分词时，它可以仅记录词的起始地址和长度，不用将词的内容复制出来，极大地提高了性能。且能嵌套处理。我在做一个划词的OJ题时也使用了类似的结构。这种节约内存的编程模式也应用在我们的ObjectNode里。仅通过记录内存的一段信息，来表达整个对象中的一个片段。

lexer在运作时，首先将oml语句转换为一个Text。再用omlLexerPush，把Text存入到stackTexts中来（存入的内容为---）。

#define CURRENT\_LEXER\_TEXT(lexer) (&(lexer)->stackTexts[lexer->stackDepth - 1])

这个宏用于获取栈顶Text元素的指针。栈顶在stackDepth-1处。Push时，向栈顶下一元素（stackDepth指向的元素）存入元素然后stackDepth++，栈顶元素永远在stackDepth-1处。用stackDepth==0表示栈空。

这个栈用于处理嵌套结构，例如括号。在处理括号时，会把括号内的Text入栈。

词法解析时，使用以下单元表示词

struct OmlWord

{

Text text;词的位置和长度信息

gm\_uint32 offset; 在整个oml语句中的偏移量

gm\_uint32 id; 精确的小类型，可能是不同的枚举类型，故这里用了gm\_uint32

OmlWordType type; 大类型，根据大类型调用不同的omlFetchXXX函数

gm\_bool isNaming; 是不是普通标识符（关键字保留字则isNaming=false）

};

omlHardParse将执行一次完整的从oml字符串开始的解析，先取首词，决定执行什么语句的解析（SELECT/INSERT/DELET/UPDATE）

取词函数omlLexerFetch，调用**omlFetchWord**来取词并过滤掉注释。

*当lexer栈空时，word type置为WORD\_TYPE\_EOF, 这个值得学习一下。*

取栈顶元素，注意这里不是复制出栈顶的Text结构体，而是通过一个Text \*text指针直接操作栈顶元素->Trim掉两端空白->word.text指向栈顶元素->word.offset表示栈顶元素到lexer.text起始元素的偏移量，也就是单词在整个oml语句中的偏移量。在处理之初，会复制lexer.text到栈顶。之后对这个栈顶元素进行一次遍历处理。每次通过omlFetchWord取出一段字符串作为“词”。

取词时调用了omlDiagWordType(text)这个函数对该词进行预判，预判会填充word type，其逻辑为测试text的首字符和次字符。

全部的Word类型由OmlWordType定义在文件gm\_lexer.h中。

* + - 1. 首字符为大小写字母，那么类型为WORD\_TYPE\_VARIANT，即变量。
      2. 首字符为数字0~9则为数字。WORD\_TYPE\_NUMBER
      3. 左括号则为WORD\_TYPE\_BRACKET
      4. 小数点`.`则看第二个字符，数字则WORD\_TYPE\_NUMBER，否则WORD\_TYPE\_SPEC\_CHAR
      5. 单引号是串
      6. \*+运算符
      7. /要看第二字符，若为\*则是块注释，否则运算符
      8. - 要看第二字符，若为 - 则是行注释，否则运算符
      9. >,<,=为比较操作符
      10. ?表示参数
      11. |要看第二字符，|则为运算符，否则错误

这里是一个预判逻辑，后面可能还是会改变的，例如遇到关键字的时候，会由WORD\_TYPE\_VARIANT变成WORD\_TYPE\_KEYWORD。

根据预判结果，调用不同的Fetch函数去获取词。取词时，才精确判断词的类型，存在word.id当中。

### omlFetchNum:

它真正开始处理栈顶的text。它会从text中取出首字符，把这个字符放入word中，并检测这个字符，分为4种情况：’.’，分隔符，数字以及其他。它使用dotExist来表示在数字中已经有了小数点，后续遇到小数点报错。如果是分隔符，那么退出循环。如果是数字就继续，其他就认为是错误。然后根据是否有小数点，将word.id设定为NUM\_TYPE\_REAL或NUM\_TYPE\_INT。所以，我们这里的数字不支持负号，也不支持科学计数法。但是小数点可以放在两端。这里数字本身不支持正负号，但是数字前面是可以加正负号的，会被作为单目操作符。（见表达式部分）。后面还会对得到的word进行一次cmTrimNumberText操作以去除两端多余的0.

### omlFetchCmp

对比较操作，使用omlFetchCmp函数，它会去检测</>/=后面是否还有其他操作符，然后根据后面的操作符组成<=,>=

### omlFetchCalc

计算包括了5种运算：

|  |  |
| --- | --- |
| || | 字符串连接 |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |

这个函数用到一个fetchSize的临时变量表示运算符长度。

### omlFetchComment

注释有两种：

|  |  |  |
| --- | --- | --- |
| -- | 行注释 | 以\n或串尾结束 |
| /\* | 块注释 | 以\*/结束 |

对第二种，当\*/不存在时，注释不完全，将报错。

### omlFetchSpecialChar

它同时用于处理WORD\_TYPE\_SPEC\_CHAR和WORD\_TYPE\_PARAM

直接读取一个字符。

|  |  |
| --- | --- |
| . | 点号 |
| ? | 参数占位符 |
| , | 逗号 |

所以在非注释非字符串中，像#，@，[，{，\都是非法符号。

### omlFetchString

它处理’abcdef’单引号字符串。但是用’’表示一个’，即a’’b->a’b。在处理时，从串中首个字符开始，碰到’后，要判断下个字符是不是也是’，如果是，则继续查找’，直到找到单个’为止。

所以在字符串中，一旦开启，后面就会一直找下一个不成双单引号，此时所有的字符都被当字符串吃掉，不会开启其他模式（如注释、括号，都没有用）

### omlFetchBracket

只处理括号与字符串。串外的括号才用于匹配。这个词将从这层括号开始到同层的右括号为止。里面可以有复杂的东西。

这里处理得有点简单粗暴，在括号中一定不能有注释，因会误把注释中的单引号和括号也用于划词。但是这样也简化了编程，暂时避免了递归。且适用大部分场景。

显然，对括号中的内容，还要二次处理。

### omlFetchVariant

这是最复杂的一个，因为它集合了真正的变量，单词类操作符和关键字等，在里面需要区分开来。

首先用omlFetchName，来获取一个标识符，这个标识符就是\w，没有要求不以数字开头。如果获取成功则word.isNaming置为True。

然后用omlMatchKeyWord去匹配标识符是不是关键字。这个函数首先检查word是不是保留字之一：

|  |  |
| --- | --- |
| false | RES\_WORD\_FALSE |
| key | RES\_WORD\_KEY |
| null | RES\_WORD\_NULL |
| rownum | RES\_WORD\_ROWNUM |
| sysdate | RES\_WORD\_SYSDATE |
| systimestamp | RES\_WORD\_SYSTIMESTAMP |
| sysutc | RES\_WORD\_SYSUTC |
| true | RES\_WORD\_TRUE |

如果不是保留字，则再看word是不是关键字：

|  |  |
| --- | --- |
| and | KEY\_WORD\_AND |
| by | KEY\_WORD\_BY |
| delete | KEY\_WORD\_DELETE |
| filter | KEY\_WORD\_FILTER |
| from | KEY\_WORD\_FROM |
| in | KEY\_WORD\_IN |
| insert | KEY\_WORD\_INSERT |
| into | KEY\_WORD\_INTO |
| is | KEY\_WORD\_IS |
| like | KEY\_WORD\_LIKE |
| not | KEY\_WORD\_NOT |
| null | KEY\_WORD\_NULL |
| object | KEY\_WORD\_OBJECT |
| or | KEY\_WORD\_OR |
| select | KEY\_WORD\_SELECT |
| set | KEY\_WORD\_SET |
| update | KEY\_WORD\_UPDATE |
| values | KEY\_WORD\_VALUES |
| where | KEY\_WORD\_WHERE |

根据word是保留字还是关键字，将word.type赋值为WORD\_TYPE\_RESERVED或WORD\_TYPE\_KEYWORD。这里就是设置的word.type而不是word.id。

关键字和保留字的isNaming属性都是false。所以通过测试isNaming可以确定是不是保留字和关键字。也知道isNaming的含义就是表示这是一个普通标识符。在某些地方，使用关键字作为名称不会报错，而是认为是一个普通标识符（字段名）。

lexer->isFuncMode，这个标志在初始化的时候是true，其作用是？

当isFuncMode时，调用omlLexerTryFetchBracket去获取参数列表。如果获取到了，就表示word.type为WORD\_TYPE\_FUNC，否则保留WORD\_TYPE\_VARIANT。

这种omlLexerTryFetchXXX的函数其实是整合了预判函数和omlFetchWord/omlFetchXXX函数的功能，只有在预判符合函数目的时才执行omlFetchWord/omlFetchXXX。

对于函数类型，会把函数名和参数列表全部存放在word中。

但是omlLexerTryFetch函数时用于判断下一个有效的词是不是期望的词。

### omlParseSelect

这个语句将给OmlContext变量赋值，

ctx->stmtType = OML\_TYPE\_SELECT

select不需要分配stmtContext。但是其他几个都是有的。

如果是select object则ctx->isScan为true。

否则要是\*

对select object from 判断表名

对select \* from 查找schema node.

继续

对select object from table 可以加 where，有where则调用omlParseKeyFilter。

没有where则看是不是filter，不是则报错。

然后处理各种filters。

### omlParseUpdate

ctx->stmtType = **OML\_TYPE\_UPDATE**;

ctx->stmtContext = gaAllocMem(ctx->omlArea, sizeof(**OmlUpdateContext**));

update需要分配一个OmlUpdateContext

struct OmlUpdateContext

{

List setPairs; 非key field

List setKeyPairs; 数组类节点需要有个key field的概念用以区分数组中各元素。

};

update首先要找到目标schemaNode。找不到则报错。但是分配的东西已经分配了。

update的语法结构为：

update node set field\_1=value\_1, field\_2=value\_2,field\_3=value\_3 filters...

为此使用了OmlUpdateContext来存储set后面这一些键值对。用函数omlParseUpdatePairs来解析这些键值对。每个键值对使用如下结构体存储：

struct OmlPair

{

gm\_uint16 fieldId;

gm\_char reserved[2];

OmlExpr \* setExpr;

};

在获取了fieldName，继而得到fieldId后，获取=号，然后获取表达式，需要一直获取到结束或逗号或filter字样。这里对表达式的处理也要特别关注：

struct OmlExpr

{

OmlContext \* owner;

SchemaNode \* schemaNode;

OmlExprNode \* root;

OmlExpr \* next; /\* for expression list, such as function args or in clause \*/

gm\_uint32 expect;

gm\_bool isNegative;

gm\_bool isGenerated;

OmlExprNodeChain chain;

};

struct OmlExprNodeChain

{

gm\_uint32 length;

OmlExprNode \* first;

OmlExprNode \* last;

};

struct OmlExprNode

{

OmlExpr \* owner;

OmlExpr \* args; /\* for function node \*/

OmlExprNodeType type;

gm\_bool isNegative;

Value value;

/\* for expression tree\*/

OmlExprNode \* left;

OmlExprNode \* right;

/\* for exprnode chain \*/

OmlExprNode \* next;

OmlExprNode \* prior;

};

可以看出，表达式用OmlExpr结构体表示，而OmlExprNode才是表达式的核心。它用到了两个数据结构：一个链表，一棵树。两种方式组织起OmlExprNode。

OmlExprNodeType表示节点类型：

|  |
| --- |
| EXPR\_NODE\_ADD |
| EXPR\_NODE\_SUB |
| EXPR\_NODE\_MUL |
| EXPR\_NODE\_DIV |
| EXPR\_NODE\_CAT |
| EXPR\_NODE\_CONST |
| EXPR\_NODE\_FUNC |
| EXPR\_NODE\_PARAM |
| EXPR\_NODE\_FIELD |
| EXPR\_NODE\_RESERVED |
| EXPR\_NODE\_UNKNOWN |

在ctx->omlArea也就是session.omlArea上进行内存分配。

首先用omlCreateExpr在ctx->omlArea上分配一个表达式结构体，然后用omlAddExprWord (OmlExpr \* expr, OmlLexer \* lexer, OmlWord \* word)读取表达式中后续的词。

这里的表达式的处理都是按照顺序处理，设定期待类型，与下个word对比。例如，field\_name=后面期待的是单目操作符和变量，即EXPR\_EXPECT\_UNARY\_OP | EXPR\_EXPECT\_VAR。

期待类型只有四种：

|  |
| --- |
| EXPR\_EXPECT\_NONE |
| EXPR\_EXPECT\_UNARY\_OP |
| EXPR\_EXPECT\_OPER |
| EXPR\_EXPECT\_VAR |

这里对期待变量需要特别的说明。期待变量的地方实际上可以包含以下类型的word：

#define EXPR\_VAR\_WORDS \

(\

WORD\_TYPE\_VARIANT | WORD\_TYPE\_FUNC | WORD\_TYPE\_STRING | WORD\_TYPE\_PARAM | WORD\_TYPE\_NUMBER \

| WORD\_TYPE\_RESERVED | WORD\_TYPE\_BRACKET | WORD\_TYPE\_KEYWORD \

)

即在期待变量的地方，可以出现变量、函数、字符串、参数等等.

这里可以画出一个状态图来：

表达式节点的类型由函数omlMatchExpected来确定。对于变量类，通过来确定omlGetExprNodeType来确定；对于操作符类，直接将word->id转化为EXPR\_NODE\_ADD等操作符类型。omlGetExprNodeType中有几点需要注意：

对于VARIANT类型的word，认为是EXPR\_NODE\_FIELD。

对于BRACKET类型的WORD，认为是EXPR\_NODE\_UNKNOWN。这里后面会处理。

对于KEYWORD类型的word，根据其isNaming属性，认为是FIELD或UNKNOWN。

对于word是括号表达式的情况，将会使用omlCreateExprFromText来创建括号内的表达式。

先讨论非括号的简单情况。显然我们要做的就是为这个词创建一个omlExprNode。使用的函数是omlAddExprNode。

对于操作符类节点，返回成功，无需多做处理。

对于函数，需要特殊处理。首先node.type显然是函数类型，记录的value是函数名，而args表达式记录着参数。对参数的处理也是调用omlCreateExprFromText，这个函数用于处理括号。

对于其他，要将word转化为值。调用cmWordToValue。字符串、数字、参数、保留字、字段分别处理。对参数，记录的是参数的索引（比如第一个参数就可以记录1）。对于保留字，就记录下word->id。对field，记录下field id。

这样，就记录下了一个node，及其对应type所需的属性。

创建Node之后，会将Node添加到表达式的chain中，即OmlExpr.OmlExprNodeChain中。OmlExprNode利用其next, prior成员可以构成链表，而OmlExprNodeChain记录着这个链表的长度、首尾指针信息。

括号的处理：

括号的处理就用到了lexer的栈。在omlCreateExprFromText这个处理括号表达式的函数中会把括号内的text入栈，处理完成后再出栈。处理的逻辑与omlAddExprWord类似，但是会将子表达式生成树（通过调用omlGenerateExpr，将链表生成语法树。然后把子表达式的根append到父表达式的omlExprNode链表中来。）值得注意的是，对于多子句，例如(a,b)这样的，a和b都是一个表达式，这个时候就会使用a的next来存放b（注意是omlExpr.next，不是omlExprNode.next）。这一般用在函数参数中，以及其他使用括号的地方，例如in子句。

在获取了所有的词，构成了表达式链之后，就会调用omlGenerateExpr来生成表达式树。它每次从节点链中取一个节点（omlExprNode）调用omlGenerateExprTree将节点添加到树中。

这个函数通过调用omlFormWithOpers来处理操作符。

它先处理乘除，再处理加减，最后处理连接符，所以连接符在这几个操作符中的优先级是最低的。

gm\_int32 omlFormWithOpers(OmlExpr \* expr, gm\_uint32 opers)

这个函数就是从链表到树的关键步骤之一。举个例子来理解它。例如有以下语句：

ret = omlFormWithOpers(expr, EXPR\_NODE\_MUL | EXPR\_NODE\_DIV);

那么在执行时，将遍历整个节点链，然后对每个类型为EXPR\_NODE\_MUL或者EXPR\_NODE\_DIV的节点执行树操作：

omlDownExprNode(expr, node);

这个函数执行的内容为：

prior节点变为左子节点，next节点变为右子节点。如下是示意图：对表达式a+b\*c-d会做如下处理：（先乘除号，再加减号，同优先级的按顺序处理）

a

+

b

\*

c

-

d

a

+

b

\*

c

-

d

a

+

b

\*

c

-

d

a

+

b

\*

c

-

d

整个执行过程如图所示。这样就完成了对简单表达式的语法树构建。而我们的oml中，对逻辑运算符和in, is null等作了特殊处理，与以上的处理过程分离（以上是四则运算，另外还有比较运算和条件运算，共三种运算构成filter）。高优先级的先被遍历一遍（\*/），低优先级的后被遍历（+-）

### Parsing Filter

对应函数：gm\_int32 omlParseFilters(OmlLexer \* lexer, OmlContext \* ctx)

filter的语法为：

filter node\_name where [not] expr\_xxx and/or [not]expr\_yyy

这里并不是很好定义，因为把表达式和in/is null/like等分离处理了。

在ctx中使用一个OmlFilter \*filters[]数组进行存储filter。我们的schema树的最大深度为16，而每层最多一个filter，所以这个数组有16个元素。根据filter子句中的node\_name找到schema node, 查看schema node的depth，将filter存储到filters[schema\_node.depth]上来。

Filter由三种运算组成：条件运算、比较运算、四则运算。条件运算的左右表达式是比较运算，而比较运算的左右表达式为四则运算。

以下是filter相关的关键数据结构。

struct OmlFilter

{

OmlContext \* owner;

gm\_bool isAlone; //is from make filter or not

SchemaNode \* schemaNode;

OmlCondNode \* root;

OmlCondNodeChain chain;

};

struct OmlCondNode

{

OmlCondNodeType type;

OmlCondNode \* left;

OmlCondNode \* right;

OmlCondNode \* next;

OmlCondNode \* prior;

OmlCompareNode \* cmpNode; /\* only used for node type is COND\_NODE\_COMPARE \*/

gm\_uint32 textPos;

gm\_bool processed;

};

struct OmlCompareNode

{

OmlCompareType type; /\* = <>, >, <, like, in, is, is not, not like, not in, \*/

OmlExpr \* left; /\* expr of left \*/

OmlExpr \* right; /\* expr of right \*/

gm\_uint32 textPos;

gm\_bool notPre; /\* not in, not exists, not like \*/

gm\_bool closed;

};

OmlCondNodeType只有以下几种，记住这点可以极大地加快源码理解。

|  |
| --- |
| COND\_NODE\_UNKNOWN |
| COND\_NODE\_TRUE |
| COND\_NODE\_FALSE |
| COND\_NODE\_COMPARE |
| COND\_NODE\_OR |
| COND\_NODE\_AND |

以上是filter相关的几个关键的数据结构。

进入一个filter处理后，会使用omlAddFilterWord来处理filter。这个函数同时用于处理filter和where key...(select object from table where key...)条件，所以在入口处做了相应的区分。

对于TRUE/FALSE会被单独当作一个OmlCompareNode节点，且这个特殊的节点没有左右子表达式。

整个filter的处理与表达式的处理非常相似。在遇到括号时，将括号中的内容使用omlCreateSubFilter来创建出一颗条件表达式树，然后将根节点添加到OmlFilter. OmlCondNodeChain中来。

omlAddCmpWord函数用于向filter链中添加OmlCompareNode，它代表了比较类型的子句，即包含如下运算符的子句（TRUE/FALSE是特例）：

=, <>, >, <, like, in, is, is not, not like, not in,

它们与四则运算被分开来处理。

对OmlCompareNode来说，它包含一个左四则表达式和一个右四则表达式，在表达式中已经不能嵌套其他的运算了（也确实没有意义，例如TRUE > FALSE，这一般没有意义，即四则运算表达式中不会再含有比较运算）所以一个filter的链的结构大致如此：

图后补。

语言描述filter的处理过程：

lexer一个词一个词地读取，然后喂给filter，filter优先处理and/or，为它们创建omlCondNode。由于这个优先级，喂给下层的词中就绝对不会包含and/or了。这种处理思想值得学习。

对于不是and/or的，会认为是omlCmpNode。如果是起始处或and/or后面，会创建一个omlCondNode来包含这个omlCmpNode，后续同一个表达式的词都会添加到这个omlCondNode.omlCmpNode中来。每个omlCmpNode一般来说都有左右两个子表达式omlExpr，以及一个比较操作符。同样，优先处理比较操作符，比较操作符不记录到omlExpr中，而是使用omlCmpNode.type来记录，例如>=就将omlCmpNode.type赋值CMP\_LARGE\_EQUAL。很多地方都是通过枚举变量记录表达式的部分词，而不是通过value。

注意not like, is not null中的not，在处理时，会将CMP\_LIKE变成CMP\_NOT\_LIKE。而is null中is也被当做操作符，只不过右子表达式为空，且会测试is后是不是null。

filter, eof也有很高的处理优先级，在遇到它们时，一个filter就处理完了。

在处理括号时，会创建子filter，并创建树，然后将根添加到omlCondNode链中来。最后调用omlGenerateFilter来生成树。

生成树前经过了‘判断’等预处理，使得链合法。

然后与处理表达式很类似，优先用omlFormWithLogicOper(filter, COND\_NODE\_AND);处理and。处理后所有的and节点都会成为一个小树根。

然后用omlFormWithLogicOper(filter, COND\_NODE\_OR);处理or的内容。最终形成一棵条件表达式树。

对于Word的类型，如果当前word是and、or那么就会调用omlAddLogicOperWord来添加一个OmlCondNode。而若非and, or 那么会调用omlAddCmpNode来添加一个比较节点。

阅读源码时，需要准确地区分omlAddExprWord和omlAddExprNode函数，前者会做提取word然后再调用后者向表达式的节点链中添加一个节点。

### omlParseInsert

类似

### omlParseDelete

类似

## 执行器

omlExecute函数负责执行一条已经prepared的oml语句。

执行器管理结构体：

struct OmlExecutor

{

gm\_void \* session;

MemoryArea stack;//自带stack 管理，这里stackBuffer是16K

gm\_char stackBuffer[OML\_STACK\_SIZE];

Value \* keyVal;

gm\_char \* omlData;

SchemaNode \* omlSchemaNode;//每一个非select object语句都有一个唯一的目标节点。这个元素就记录着那个节点。在context中也有记录（schemaId）。

Object \* omlObject;

ObjectNode \* objectNode; /\* current object node \*/

gm\_uint32 rownum;

gm\_bool isEof;

ObjectNode nodes[GMDB\_MAX\_TREE\_DEPTH];

};

它在handler中：

struct ObjectHandler

{ //其它成员省略

OmlExecutor omlExecutor;

};

即每个handler公用一个执行器。

Statement和OmlExecutor都是自带buffer，然后用自己的MemoryArea成员去管理这段buffer。

在执行器中，有如下语句：

executor->omlData = stmt->omlBuffer

我们之前所有的表达式OmlExpr, OmlExprNode, OmlCond, OmlCondNode, OmlCmpNode等等都是通过Statement.MemoryArea从Statement.omlBuffer里面申请出来的，且是在其堆上申请出来的。通过上面这条语句，就把executor->omlData指向了包含所有表达式数据的缓冲区。

经过一番准备，executor就得到了目标object，目标schemaNode，omlData等必要的运行信息。然后根据语句类型，选择不同的处理子函数，这里我们以omlExecuteSelect(executor);为例进行理解。

static gm\_int32 omlExecuteSelect(OmlExecutor \* executor)

{

return omlNestLoopPrepare(executor, 0);

}

可以看到仅调用了一个非常关键的函数。其原型如下：

static gm\_int32 omlNestLoopPrepare(OmlExecutor \* executor, gm\_uint32 depth)

ctx->filterMaxDepth表示最大filter节点深度。执行时，取其与omlSchemaNode->Depth中的大者作为终点深度endDepth。而用filterSchemaNode表示最大深度的schema节点。

而作为入参的depth则表示此次prepare的起始深度，通过封装，起始深度永远是0，即永远从根节点开始查找。对从depth到endDepth的每层的filter：

filter = ctxData->filters[depth];//ctxData->filters记录着动态分配的filter的offset。

filterData = (OmlFilterData \*)(executor->omlData + filter);//通过buffer+offset取到filter数据。

?filterData是在什么时候填充的?

答，在函数gm\_int32 omlSerializeFilter(MemoryArea \* area, OmlFilter \* filter, OmlOffset \* offset)中。这个动作是在prepare oml之后 serialize context的时候被调用的。暂时还不知道是什么作用。但是其中的内容透过filterData大约是知道的。

使用curNode记录当前节点，根据filterSchemaNode->path可以得到schema中从根到终点（最深节点）的路径，路径的每个元素记录着其在父节点中的field id。在任何一层，都能从父object节点定位到子object节点，也就是curNode->childNode。然后根据filter，从childNode数组中找到对应的那个元素。

这里可以深度优先，但是遍历所有节点。最终的结果是一个集合。

使用dmLookupArray函数来从数组节点中过滤出符合条件的那些元素。dmLookupArray会输入当前节点，每次调用会从当前节点开始查找，找到数组中下一个符合过滤条件的元素，并返回。

executor->nodes[depth]记录着第depth层数组节点的那个被找到的元素。

计算条件表达式使用的是omlMatchCondNode函数，这个函数的逻辑大概是先看是否是cmpNode如果是，则表示这是一个比交表达式，不包含and/or逻辑操作，直接返回omlMatchCmpNode的结果。否则递归调用omlMatchCondNode先求左表达式的值，如果不能依靠and/or的短路效应判断出结果的真假，则返回右表达式的值。以下图为例：



这个语法树对应于如下表达式

a > 2 and a < 21 or a is null

or/and将执行omlMatchCondNode，而<,>,is null将执行omlMatchCmpNode。

对于or，当左操作数and的结果为True时，右边就不用执行了。对于and，如果a <= 2那么也不用执行a < 21的测试了。

omlMatchCmpNode是各种比较计算的实现函数，其中有几个需要特别留意：like、not like。可以看做是简化版的正则引擎。处理时并未使用KMP算法防止回溯，不知道是不是不可行，还是性价比不高。想来是性价比不高，因为大多数情况下应用场景不是在很长串中查找，而是短小串的匹配，KMP需要不断的构建next数组，不划算。在数据库中如果一条信息有特殊的含义显然不能纯粹存放到字符串字段中，而应该建立特殊的属性字段以归类，这点在规划时就需要做好，而不是到了查询阶段再做。即使规划时没有做好，也应该在后续修正时创建这些字段并一次性填充完整，避免推迟到查询时。KMP最有优势的时候应该是重复较长较多的长模式串的匹配。这里还带有\_的匹配，使用KMP将徒增复杂性（且增加非常多）。

omlCalcExpr用于计算表达式的值（+-\*/||）。

executor->nodes[]中记录了查找过程中从根到目标节点的路径上的中间节点。通过fetchNode可以继续查找下一个节点。

dmLookupNext这个函数与dmLookupArray不太一样的地方在于它会从指定节点的下一个节点开始，指定节点不会被包含在搜索集中。

omlExecuteChildMatch函数的流程：

# 登录操作

在安全整改的时候要求密码不能明文硬编码在代码中。整改的方案有两个。

* + - 1. 之前的Tree引擎以客户端的方式连接GMDB数据库Server并使用SQL语句来操作数据库。连接时需要用户名和密码，被明文硬编码在代码中。因为我们的Tree引擎和GMDB的Server是在一个进程中，所以可以不用密码，直接通过API实现所有操作。但是直接使用API的方式绕过客户端，失去了客户端的带来的好处，比如详尽的使用文档，人性化的接口等。改动也很大。所以没有选择这个方案。
      2. 保留客户端模式，但是密码由外界传入，用户名使用固定的sys用户。

选择方案2后，刚开始决定外部传入的密码用AES加密保存到内存，用的时候再取出来。但是后面决定不需要这么麻烦，只需要明文存储就行了。但是增加一个需求，就是如果数据库不存在（首次startup）要创建数据库，并且以此时传入的密码作为初始密码。这个需求的实现需要了解数据库对密码的加密方式，因为存储在数据库中的密码是密文不是明文，我们在接口得到明文后，要先加密，然后存放到数据库中。

数据库的密文经过了两次加密。这里我犯了一个错误，以为只经过一次加密。犯错的原因是对源代码不熟悉。为了找到数据库的加密方式，我首先查看了加密流程，总结出以下加密流程：



基于这个流程，我只需要把en\_pw\_1存储到数据库中即可。

在测试的时候发现这样不对，客户端的en\_pw\_1与服务端从数据库读取出来的一样，但是en\_pw\_2与db\_en\_pw\_2不一样。显然，数据库服务端在加密算法中使用的是不同的算法。而我不该把en\_pw\_1直接存储到数据库中。为了搞清楚数据库到底如何存储密码，我找到了create user流程中对密码加密的函数：ddl\_encrypt\_password\_with\_salt，发现它调用了以下common模块的加密函数cm\_encrypt\_password\_str，还进一步调用了cm\_encrypt\_str，二次加密。后者是一个SHA256加密。相当于把en\_pw\_1再次加密了。且是不可逆的。

服务端得到客户端发过来的en\_pw\_2后，经过一番计算，获得一串数据，需要与数据库中存储的相同。调用的是这个函数：cm\_calc\_srv\_encrypt\_pwd(db\_uchar \* user\_pwd, db\_uchar \* key\_str, db\_uchar \* req\_pwd, db\_uchar \* out\_str)。它先用数据库存储的密码加key（字符串连接），进行SHA256加密，得到一个串。然后str2bin, 得到en\_key\_pwd\_bin,同时，客户端那边得到的密码也str2bin得到en\_pwd\_bin，然后两个bin异或，得到en\_tmp\_pwd\_bin，再bin2str得到en\_tmp\_pwd\_str。再对这个str进行sha256，得到一个数据，与客户端发来的对比。很难搞懂。先暂时不管它，有空再搞。总之时序图没有问题，但是存储在数据库中的数据和加密算法并非与客户端一样如此。

另外,在本地加载时,也应该验证密码,具体措施是在sql\_start\_server时，加载表

通过gm\_connect登录内核。

首先是连接方式的选择。我们的tree引擎使用的是tcp连接。当前是将登录密码写在了源代码中，连接时使用。不符合安全规范。现在希望在是同一进程时，可以不需要密码。服务器首先在进程的静态空间写随机数据，能拿到这个随机数据并登录的人，就认为是同进程的人。这里首先要了解下如下结构：

typedef struct tagclt\_conn

{

/\* BEGIN 2C11-587 liyi 193723 2013-1-22 ADD ONIP DataGrid V300R002C11B016 \*/

cm\_spinlock\_t conn\_mthrd\_debug\_lock;

/\* END 2C11-587 liyi 193723 2013-1-22 ADD ONIP DataGrid V300R002C11B016 \*/

cm\_spinlock\_t conn\_lock; /\* lock for communication \*/

clt\_conn\_mode\_e mode; /\* tcp / ipc / direct \*/

db\_void \* session; /\* the address of kernel session, for direct access \*/

db\_uint32 thread\_id; /\* current thread, for direct access \*/

clt\_conn\_status\_e status;

db\_bool is\_auto\_commit; /\* if auto commit the transaction \*/

cs\_pipe\_t pipe; /\* the connection pipe to server \*/

cs\_pack\_t pack; /\* for sending data to server \*/

db\_char key[SQL\_ENCRYPT\_KEY\_SIZE];

/\* BEGIN DTS2014090904009 hejinyang 00197555 2014-10-09 MODIFY V300R002C50SPC100 \*/

db\_char salt[SQL\_PASSWORD\_SALT\_BUF\_LEN];

db\_uint32 iteration;

/\* END DTS2014090904009 hejinyang 00197555 2014-10-09 MODIFY V300R002C50SPC100 \*/

/\* BEGIN DTS2012110100006 maqingli 180002 2012-11-05 GMDBV200R005C00B310 \*/

//db\_bool rev\_flag;

/\* END DTS2012110100006 maqingli 180002 2012-11-05 GMDBV200R005C00B310 \*/

db\_uint32 charset\_id;

cm\_list\_t stmts; /\* statements of this connection \*/

// db\_void \* stmts[SQL\_MAX\_STMTS];

/\* BEGIN AR-0000686441 yangshangde 176558 2014-1-23 ADD ONIP DataGrid V300R002C50 \*/

cm\_list\_t descriptors; /\* Descriptors of this connection \*/

/\* END AR-0000686441 yangshangde 176558 2014-1-23 ADD ONIP DataGrid V300R002C50 \*/

db\_int32 last\_error; /\* store the last error code \*/

db\_char error\_msg[CLT\_MAX\_MSG\_LEN]; /\* error message \*/

/\* BEGIN GMDBV200R005-304 maqingli 180002 2012-9-18 ADD FOR LOB DATA \*/

cs\_pack\_t \*lob\_recv\_pack; /\* for receving lob data from server \*/

/\* END GMDBV200R005-304 maqingli 180002 2012-9-18 ADD FOR LOB DATA \*/

db\_bool is\_tx\_open;

/\* BEGIN GMDBV200R005-301 liweichao00196241 2012-10-19 MODIFY GMDB V200R005C00B200 \*/

sql\_ssl\_mode\_e ssl\_mode;

db\_bool url\_ssl;

/\* END GMDBV200R005-301 liweichao00196241 2012-10-19 MODIFY GMDB V200R005C00B200 \*/

/\* BEGIN DTS2013011800713 pengwei 205084 2013-1-17 ADD GMDBV200R005C00SPC001B110\*/

db\_uint32 clt\_recv\_timeout; //the timeout of client recv

/\* END DTS2013011800713 pengwei 205084 2012-1-17 ADD GMDBV200R005C00SPC001B110\*/

// BEGIN C12-261 lvyao 00161577 2013-02-26 MODIFY ONIP DataGrid V300R002C12B011

db\_void \* group;

// END C12-261 lvyao 00161577 2013-02-26 MODIFY ONIP DataGrid V300R002C12B011

db\_char user\_name[DB\_MAX\_NAME\_LEN];

db\_char encrypt\_passwd[SHA\_BLOCK\_SIZE+1];

db\_bool is\_quick\_connect;

cs\_server\_desc\_t server\_info;

db\_uint16 session\_id;

db\_uint8 reserved[2];

}clt\_conn\_t;

如果要考虑其他的直接操作方式，首先要解决的是搞明白执行流程。

根据D:\USCDB\GMDB - story-all\V2R5C00\资料\GMDB V200R005C00 应用开发指南-C.chm可以看标准的流程是以插入数据为例：

1. 声明用于连接数据库的连接句柄、执行语句句柄。
2. db\_int32 ret;
3. db\_conn conn = NULL;

db\_stmt stmt = NULL;

1. 连接数据库。

ret = gm\_connect("sys", "Admin123", "127.0.0.1:8500", &conn);

1. 申请语句句柄。

ret = gm\_alloc\_stmt(conn, &stmt);

1. 准备语句。

ret = gm\_prepare(stmt, "insert into tb1(f0) values(1)");

1. 执行语句。

ret = gm\_execute(stmt);

执行成功后会返回SQL\_SUCCESS。执行失败后会返回错误码，可以通过**gmerror**工具查看错误码信息。

1. 提交或回滚事务。

提交事务：

ret = gm\_commit(conn);

回滚事务：

ret = gm\_rollback(conn);

说明：

GMDB C-API默认关闭自动提交。要打开自动提交，请执行：

ret = gm\_set\_auto\_commit(conn,DB\_TRUE);

关闭自动提交，请执行：

ret = gm\_set\_auto\_commit(conn,DB\_FALSE);

查看自动提交状态，请执行：

ret = gm\_get\_auto\_commit(conn, &is\_auto\_commit)；

1. 释放语句句柄。

ret = gm\_free\_stmt(&stmt);

1. 释放连接。

ret = gm\_disconnect(&conn);

其中连接数据库后，得到一个conn类型。而我们要做的就是消除这一步。其中在gm\_prepare和gm\_execute时可能会用到连接，而gm\_execute时应该会向server发送数据。如果能把这个发送过程改为直接调用就好了。

在执行中我们看到它会根据stmt->access\_mode来决定执行方式，一种是通过cs（tcp/ipc）方式，另一种是直接方式，直接方式应该就是我们需要的方式。但是也要得到一个连接句柄conn。连接方式有三种：

typedef enum tagsql\_access\_mode

{

SQL\_ACCESS\_SIMPLE\_DIRECT = 1,

SQL\_ACCESS\_DIRECT = 2,

SQL\_ACCESS\_CS = 3

}sql\_access\_mode\_e;

其中前两种方式是没有使用到通信通信机制的，也就不存在拦截危险。

但是这些方式都是基于从客户端去操作的思想。而我们现在既然客户端和服务端是同一进程，那就可以直接调用服务端接口。这部分接口没有文档，需要看看服务端是如何进行操作的。

哪些是服务端的，哪些是客户端的接口？

服务端从sql\_process\_request开始看，作为处理的起点。

客户端从gm\_prepare、gm\_execute等开始。

可以看到其中gm\_prepare最终调用clt\_call\_remote\_prepare来执行prepare动作。而服务端收到后调用sql\_prepare来执行动作。且若没有stmt则会为他分配一个（sql\_alloc\_stmt）。同理，gm\_execute也最终会调用clt\_call\_remote\_execute。

在内核中，pl表示存储过程的意思。而clt表示client的意思。

在cs方式下，执行是以CS传输方式操作的，即客户端将请求发给服务端，服务端回发响应。在direct方式下，执行是直接操作共享内存，能在同一机器上操作。

我们的层次结构大致是下图示：

KV

Client

内部kernel

SQL

这里看一下gm\_put\_object，它用的是KV层。

tree这一层gm\_attach\_server通过ddAllocSession来分配一个session。但是我们只使用ddl，所以不需要这个session.

以sql\_开头的是上面的SQL层，显然它用于执行sql语句。我们执行DDL也应该是通过它。从API来看，调用顺序应该是：

sql\_alloc\_conn

sql\_alloc\_stmt

sql\_prepare

sql\_execute

sql\_free\_stmt,sql\_free\_conn

搜索密码Admin123, 可以看到如下结果：

---- Admin123 Matches (25 in 10 files) ----

gmsql\_command.c (pkg\src\tools\gmsql): Description : parse the connect string "connect sys/Admin123 ipc|ip:port|direct " from terminal

gmsql\_command.c (pkg\src\tools\gmsql): /\* "connect sys/Admin123 ipc|ip:port|direct" only supports 3 options \*/

gmsql\_command.c (pkg\src\tools\gmsql): /\* parse the connect string: connect sys/Admin123 testdb@10.71.114.181:31500 \*/

gm\_ddl.cpp (pkg\src\treeserver): const gm\_char \* passWord = "Admin123";

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

sql\_interface.h (pkg\src\sql\include): ret = gm\_connect("sys", "Admin123", "ipc", &conn);

srclient.c (pkg\src\srep\tools\srclient): sys/Admin123@10.67.138.65:8500

srep\_cmpsrv.c (pkg\src\srep\kernel\src): //cmpsrv\_init\_conn\_info("sys", "Admin123");

tool\_data\_parse.c (pkg\src\tools\common\src): /\* to parse "sys/Admin123@ip:port" \*/

ut\_cm\_ssl.cpp (pkg\test\ut\common): ret = cs\_ssl\_write\_pwd\_file(pwd\_file\_path, key\_factor\_path, "Admin123", DB\_FALSE);

ut\_ddl\_user\_new.cpp (pkg\test\ut\sql): db\_char \* user\_name = "sys", \* pwd = "Admin123", \* out\_str = "ipc";

ut\_ddl\_user\_new.cpp (pkg\test\ut\sql): memcpy(out\_str, "Admin123", sizeof("Admin123"));

ut\_ddl\_user\_new.cpp (pkg\test\ut\sql): memcpy(conn.user\_info.pwd, "Admind1233", sizeof("Admin1233"));

ut\_ddl\_user\_new.cpp (pkg\test\ut\sql): memcpy(conn.user\_info.pwd, "Admind1233", sizeof("Admin1233"));

ut\_sql\_client.cpp (pkg\test\ut\sql): ret = clt\_set\_login\_user\_info(conn, &req, "sys", "Admin123");

ut\_sql\_client.cpp (pkg\test\ut\sql): ret = clt\_set\_login\_user\_info(conn, &req, "u01234567890123456789012345678", "Admin123");

ut\_sql\_client.cpp (pkg\test\ut\sql): ret = clt\_set\_login\_user\_info(conn, &req, "u012345678901234567890123456789", "Admin123");

ut\_sql\_client.cpp (pkg\test\ut\sql): ret = clt\_login(conn, "sys", "Admin123", CLT\_NORMAL\_LOGIN);

ut\_sql\_client.cpp (pkg\test\ut\sql): ret = clt\_connect("sys", "Admin123", "IPC", &conn);

ut\_tl\_gmsql\_pub.cpp (pkg\test\ut\tools): db\_char \*param[2] = {"gmsql", "sys/Admin123@127.0.0.1:8500"};

ut\_tl\_gmsql\_pub.cpp (pkg\test\ut\tools): EXPECT\_STREQ(multi\_sql, "connect sys/Admin123 127.0.0.1:8500;");

经过分析可以发现，仅有ddAttachSQL中的是危险的。所以只需要替换此处，也就是gm\_ddl.cpp中的实现方式即可。

从服务端流程来看（查看sql\_process\_request及其参数conn的来源），它首先会sql\_alloc\_conn。然后初始化一些数据。它包含user\_info，is\_login等元素，显然是用来保存登录信息的。在sql\_execute中有如下信息：

db\_int32 sql\_execute(sql\_stmt\_t \* stmt)

{

。。。

case SQL\_LANG\_TYPE\_DDL:

// BEGIN DTS2013040303547 wangnan 100089 2013-4-8 ADD ONIP DataGrid V300R002C12B012

cm\_spin\_lock(session->sid, &g\_ddl\_reference.lock);

/\* BEGIN DTS2014031708887 wangsinan 2014-3-19 ADD \*/

if (g\_ddl\_reference.srep\_runing || g\_ddl\_reference.schema\_sync\_running)

{

knl\_error(session, SQL\_ERR\_SREP\_RUNNING);

err\_code = SQL\_ERR\_SREP\_RUNNING;

cm\_spin\_unlock(&g\_ddl\_reference.lock);

break;

}

/\* END DTS2014031708887 wangsinan 2014-3-19 ADD \*/

g\_ddl\_reference.count++;

cm\_spin\_unlock(&g\_ddl\_reference.lock);

err\_code = ddl\_execute(stmt);

cm\_spin\_lock(session->sid, &g\_ddl\_reference.lock);

g\_ddl\_reference.count--;

cm\_spin\_unlock(&g\_ddl\_reference.lock);

// END DTS2013040303547 wangnan 100089 2013-4-8 ADD ONIP DataGrid V300R002C12B012

break;

。。。

}

可以看到并没有使用到stmt->conn的信息。这里不填user信息也行。一直向下几层也没有用户信息。sql\_prepare似乎也没有使用到conn的其他信息（实际上会检查用户权限，如果不添加用户并赋予权限，就不能正确地prepare）。

经过一番试验，DDL已经可以顺利使用内核的sql服务端（sql\_server）来处理。但是还要处理DML，用到fetch等。

执行本次修改后，修改的是treeserver/gm\_ddl.cpp文件，它不再使用客户端，因而，一些原来gm\_xxx即sql\_interface.c支持的函数将无法使用。

客户端所有可用方法：

|  |  |
| --- | --- |
| 函数名 | 是否有被使用到 |
| gm\_alloc\_desc\_handle |  |
| gm\_alloc\_group |  |
| gm\_alloc\_stmt |  |
| gm\_bind\_by\_name |  |
| gm\_bind\_by\_pos |  |
| gm\_bind\_column |  |
| gm\_bind\_row |  |
| gm\_bind\_struct\_array\_by\_name |  |
| gm\_bind\_struct\_array\_by\_pos |  |
| gm\_cancel\_operation |  |
| gm\_clean\_group |  |
| gm\_commit |  |
| gm\_connect |  |
| gm\_date2utc |  |
| gm\_decode\_date |  |
| gm\_desc\_obj |  |
| gm\_detach\_db |  |
| gm\_disconnect |  |
| gm\_encode\_date |  |
| gm\_execute |  |
| gm\_execute\_add |  |
| gm\_execute\_all |  |
| gm\_fetch |  |
| gm\_free\_desc\_handle |  |
| gm\_free\_group |  |
| gm\_free\_stmt |  |
| gm\_get\_affect\_rows |  |
| gm\_get\_attr |  |
| gm\_get\_auto\_commit |  |
| gm\_get\_batch\_result |  |
| gm\_get\_column |  |
| gm\_get\_column\_count |  |
| gm\_get\_ddl |  |
| gm\_get\_ddl\_by\_schema |  |
| gm\_get\_group\_affect\_rows |  |
| gm\_get\_last\_error |  |
| gm\_get\_output\_len |  |
| gm\_get\_output\_msg |  |
| gm\_get\_param\_by\_pos |  |
| gm\_get\_param\_count |  |
| gm\_get\_param\_info\_by\_pos |  |
| gm\_get\_param\_num |  |
| gm\_get\_stmt\_type |  |
| gm\_get\_tx\_status |  |
| gm\_is\_connected |  |
| gm\_numeric2str |  |
| gm\_prepare |  |
| gm\_quick\_connect |  |
| gm\_rollback |  |
| gm\_set\_auto\_commit |  |
| gm\_set\_column\_offset |  |
| gm\_set\_stmt\_attr |  |
| gm\_set\_update\_data |  |
| gm\_utc2date |  |

使用到的接口总结：

|  |  |  |
| --- | --- | --- |
| 函数名 | 备注 | 替换函数 |
| gm\_alloc\_stmt | ddAttachSQL | sql\_alloc\_stmt |
| **gm\_bind\_by\_pos** | 替换不明显，仅在ddInsertKVMeta中用到 | 用于将schema保存到SYSTREE表中，SYSTREE表并不包含lob数据，而是使用多条记录容纳一个schema，每条记录用order no表示序号。但是没有sql\_bind\_param这样的函数。gm\_bind\_by\_pos是个客户端函数，使用的数据结构都是客户端的，看起来需要自己写一个sql\_bind\_by\_pos函数，要同时参照客户端和服务端的实现方式。另一种做法是不要使用SQL语句来做DML操作，而是改用knl\_xx系列接口，此时要模仿gm\_put\_object， gm\_get\_object的操作。应该这种方式会比前者简单得多。 |
| gm\_commit |  | knl\_commit |
| gm\_connect | ddAttachSQL | sql\_alloc\_conn/ddl\_search\_user/sql\_privs\_load\_userprivs |
| gm\_disconnect | ddDetachSQL | sql\_free\_conn |
| gm\_execute |  | sql\_execute |
| **gm\_fetch** | 替换不明显，仅在ddGetKVMeta中使用到 | sql\_fetch或sql\_fetch\_rows。似乎gm\_fetch与gm\_get\_column是联动的，fetch之后才能get\_column。每次fetch，就把结果行放到&stmt->columns中，然后用get\_column+column\_id就可以获得column。 |
| gm\_free\_stmt | ddDetachSQL | sql\_free\_stmt |
| **gm\_get\_column** | 替换不明显，仅在ddGetKVMeta中使用到 |  |
| gm\_prepare |  | sql\_prepare |
| gm\_quick\_connect | 实际被注释掉 | None |

密码安全问题，数据库文件被盗后，黑客很容易得到非加密的信息，但是密码不会泄露，因为密码是以密文的方式存储在数据库中，得到密文也无济于事。

# 日志系统

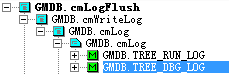
## 日志去向

### tree层的日志：

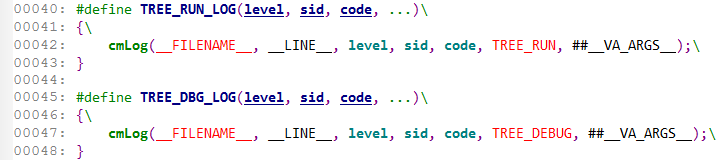
通过gm\_log.cpp/.h，在函数cmLogFlush中可以找到去向：

|  |  |
| --- | --- |
| 日志类型 | 日志去向 |
| DEBUG | dgmtree.log |
| RUN | rgmtree.log |

tree层日志调用关系图：



可以看到两个宏仅有的差别：



在使用时：以TREE\_RUN\_LOG为例：

通过两个组合：

enum treeLogLevel

{

INFO\_LEVEL = 0,

WARN\_LEVEL ,

ERR\_LEVEL ,

EMERGENCY\_LEVEL

};

enum treeLogType

{

TREE\_DEBUG = 0,

TREE\_RUN = 1

};

核心函数cmLog的解析：

*gm\_void* **cmLog**(const gm\_char \* ***fileName***, gm\_uint32 ***line***, treeLogLevel ***level***, gm\_uint32 ***sid***, gm\_int32 ***errCode***, treeLogType ***type***, const gm\_char \* ***fmt***, ...)

其8个输入参数（***fileName, line, level, sid, errCode, type, fmt,*** *args*）分别用在如下地方：

首先用多线程锁，这里用了自旋锁提升效率。然后调用如下两个函数：





这两个函数分别做了如下事情：

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 描述 |
| cmLogFormat() | level, sid, data, errCode, fmt, args | data为输出参数，构造形如如下Log： {level}:TREE {[session\_id]}{format\_args}{.|, error code: {error\_code}}。 |
| cmWriteLog() | fileName, line, data, type | data为输入参数, 将data写入到文件中，根据类型。logType是TREE\_DEBUG还是TREE\_RUN，有不同动作。两者都会分别写入各自的日志文件dgmtree.log/rgmtree.log，而TREE\_RUN类型还会添加[filename, line]到日志尾并写入到dgmtree.log！所以dgmtree.log是比较全面的。 |

通过cmLogFlush函数来写日志，输入参数为logType和日志行，它会主动添加时间信息到日志行前面，根据logType将日志输出到dgmtree.log/rgmtree.log。

当前没有办法选择开启日志级别。也不能对日志文件进行rotate，它会越来越大。到后期，将会有性能缺陷。

何时要输出日志：需要帮助定位问题的地方，可能要打日志。

不该输出日志：能返回很明确的错误码，且失败状况是预期的，可恢复的。

其他使用TREE\_RUN\_LOG宏的地方：

gm\_error.h中定义了很多宏，使用了TREE\_RUN\_LOG。

|  |  |  |
| --- | --- | --- |
| 宏原型 | 使用场景 | 备注 |
| TREE\_CM\_ERR(code, ...) | 参数配合错误码的描述字段 | 由于TREE\_RUN\_LOG的容错性，当错误码描述字段包含格式描述符却不提供参数时也不会崩溃。因为没有参数时，cmLogFormat将用复制替代sprintf。 |
| TREE\_CM\_ERR\_FMT(code, fmt, pos) | 只用于错误格式串仅含一个%s且只需要用fmt格式化一个参数pos时。 |  |
| TREE\_CM\_ERR\_TEXT(code, text) | 错误码格式串仅含一个%s时，text为Text解耦固体而非传统字符串时。 |  |

### 内核层：

内核层的日志比较复杂，待分析。  
首先看一些路径：

log\_path: *D:\Workspace\V500R008C50\USCDB\Source\Gmdb\pkg\test\ut\_home\log*

program\_name:*ut\_tree.exe*

|  |  |
| --- | --- |
| dbg\_log\_path | **log\_path**/debug/ |
| dbg\_file\_name | dut\_tree.exe |
| run\_log\_path | **log\_path**/run/ |
| run\_file\_name | rut\_tree.exe |
| full\_run\_file\_name | **log\_path**/run/rut\_tree.exe.log |
| opr\_log\_path | **log\_path**/operation/ |
| opr\_file\_name | out\_tree.exe |
| sec\_log\_path | **log\_path** /security/ |
| sec\_file\_name | sut\_tree.exe |
| login\_log\_path | **log\_path** /security/ |
| login\_file\_name | lgmserver |

内核层的log\_path由以下函数来确定：

ret = cm\_get\_path\_from\_mdb\_home("log", log\_path, sizeof(log\_path));

实际是GMDB\_HOME下的log文件夹。

关于日志还可以参考：D:\DocumentsUSCDB\GMDB - story-all\V2R5C00\资料\ GMDB V200R005C00 日志参考.chm

摘录比较重要的几点：

日志处理机制

日志处理的机制为直接记录本地日志。主线程写共享内存，由单独的线程或者进程来监控共享内存再写本地文件。

# 配置(db\_config.h/.c)

代码量：352 + 4078

接口类型数量：8

接口API数量：24

## 对外头文件：

基本功能描述：负责加载和解析server.ini配置文件，以及修改和写入该配置文件。

类型列表：

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 说明 |
| 1 | cm\_param\_id | 参数id列表，当前有85个 |
| 2 | cm\_param\_type\_e | 参数类型，如UINT32，STR，FILEPATH等 |
| 3 | cm\_param\_modifiable\_e | 修改后何时生效，RECREATE、REBOOT等 |
| 4 | cm\_param\_value\_u | 联合体，int32/uint32/str三者，str最大255+1 |
| 5 | cm\_param\_validate | 验证函数指针类型 |
| 6 | cm\_param\_data\_t | 数据，包含4，以及一个read\_from\_file标志 |
| 7 | cm\_param\_attr\_t | 属性/参数，对应server.ini的一个参数，g\_param\_attr是一个全局数组，包含所有参数的默认值。当前85个元素，与1对应 |
| 8 | cm\_param\_ctrl\_t | 参数句柄块，入口，定义了全局变量g\_param\_ctrl |

对外API接口列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 函数名 | 参数 | 功能说明 |
| 1 | cm\_load\_cfg | ini\_file\_name | 启动时从ini文件加载参数 |
| 2 | cm\_attach\_cfg | - | 启动时使用配置模块 |
| 3 | cm\_cfg\_initialized | - | 判断config是否已初始化 |
| 4 | cm\_get\_int32\_param | param\_id | 获取指定参数的int32值 |
| 5 | cm\_get\_uint32\_param | param\_id | 获取指定参数的uint32值 |
| 6 | cm\_get\_str\_param | param\_id | 获取指定参数的str值 |
| 7 | cm\_get\_param\_attr | param\_id | 获取指定参数的attr(cm\_param\_attr\_t) |
| 8 | cm\_get\_modifiable | modifiable | 获取modifiable对应的显示字符串 |
| 9 | cm\_fmt\_param | param\_id，str\_buf，buf\_size | 格式化指定参数值 |
| 10 | cm\_set\_param | sid，param\_name，param\_value | 更新srep配置，仅用于srep/redo |
| 11 | cm\_sr\_set\_param | param\_name，param\_value，old\_value | 根据参数名设置参数值 |
| 12 | cm\_check\_cfg\_port | - | 检查ini文件中的port参数 |
| 13 | cm\_refresh\_uint32\_param\_from\_file | param\_id | 从文件中读取指定参数值 |
| 14 | cm\_get\_int32\_param\_from\_file | param\_id，val | 从文件中读取指定id的int32值 |
| 15 | cm\_get\_uint32\_param\_from\_file | param\_id，val | 从文件中读取指定id的uint32值 |
| 16 | cm\_get\_str\_param\_from\_file | param\_id，param\_val，val\_buf\_len | 从文件中读取指定id的str值 |
| 17 | cm\_reflush\_cfg | ini\_file，param\_attr，in\_param\_cnt，report\_file\_path | 将配置信息写入到ini文件中去。 |
| 18 | cm\_multi\_ip\_validate | ip\_str | 验证以逗号分隔的多个ip地址的有效性 |
| 19 | cm\_parse\_multi\_ip | in\_str，split，max\_ip\_cnt，buf，count | 将以逗号分隔的多个ip地址划分为单独的IP地址的数组。 |
| 20 | cm\_check\_param\_values\_accordant | - | 检查参数一致性（内存和文件中） |
| 21 | cm\_init\_params | ini\_file\_name，param\_cnt，param\_attr，param\_data | 初始化全局参数控制块（句柄）并从ini文件中加载参数值 |
| 22 | cm\_get\_param\_id\_by\_name | param\_name，param\_id | 根据参数名查找参数id |
| 23 | cm\_get\_param\_value\_from\_file | param\_id，val，ini\_file，in\_param\_attr，in\_param\_attr | 从文件中获取指定参数的值 |
| 24 | get\_g\_param\_attr\_count | - | 获取gmserver参数数量，当前是85 |

获取指定参数的值都是从全局变量g\_param\_ctrl中去查找，即从全局控制块（句柄）去查找。全局控制块包含了所有数据的值，而数据的属性由g\_param\_attr存储。

## 实现文件db\_config.c

私有类型列表：

|  |  |  |
| --- | --- | --- |
| 1 | cm\_ini\_line\_type\_e | ini文件中一行的类型 |
| 2 | cm\_param\_def\_node\_data\_t | 参数定义节点，name:value |
| 3 | cm\_param\_node\_data\_u | 参数节点联合体，原文件该行+2 |
| 4 | cm\_param\_node\_t | 参数节点，包含双向链表节点，1和3 |
| 5 | cm\_section\_node\_t | 节节点，包含双向链表节点，节名和节内参数节点（也可能内嵌节节点，尚不清楚）列表 |

私有接口：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 函数名 | 参数 | 说明 |
| 1 | cm\_page\_size\_validate | value | PAGE\_SIZE参数的验证函数 |
| 2 | cm\_max\_sessions\_validate | value | 最大Session的验证函数 |
| 3 | cm\_free\_all\_params | param\_content | 删除参数列表 |
| 4 | cm\_add\_description\_to\_report\_file | description, file\_path | 向文件开头插入文本 |
| 5 | cm\_check\_number\_param | attr, value, convert\_value | 检查value是否符合参数设定 |
| 6 | cm\_check\_str\_param | attr, value | 属性和值 |
| 7 | cm\_check\_special\_str\_type | type, value | IP，FILE，FULLPATH，PATH类型参数的验证 |
| **8** | **cm\_check\_param** | **attr,value,converted\_value** | **综合以上各种验证函数，来验证一个参数的值** |
| 9 | cm\_copy\_param | attr,src\_value,dst\_value | 复制参数，主要区分参数是int还是str |
| 10 | cm\_set\_params\_defval | control | 默认值写入控制块 |
| 11 | cm\_check\_param\_attr | control | 检查所有控制块中的参数的值。以及参数属性的合理性，防止输入错误（很有必要！） |
| 12 | cm\_get\_param\_id | control, param\_name | 由参数name获取参数id |
| **13** | **cm\_decode\_line** | **buffer, line\_type, section, param\_name, param\_value** | **解析一行数据，在后面重点描述，除了buffer，全是输出参数** |
| 14 | cm\_special\_validate\_all\_params | control | 调用各参数的验证回调函数，当前只有PAGE\_SIZE / MAX\_SESSIONS有。 |
| 15 | cm\_get\_one\_line\_text | buf，line, line\_len | 从buf中复制一行出来 |
| 16 | cm\_find\_section\_node | param\_content,name | 从给定参数列表中按名字查找节 |
| 17 | cm\_decode\_ini\_add\_section | param\_content,section\_name | 向给定链尾部添加一个节节点 |
| 18 | cm\_decode\_ini\_add\_sub\_node | param\_content, section, line\_type, line\_text, param\_name, param\_val | 向尾部添加一个节子节点 |
| 19 | cm\_read\_ini\_file | ini\_file,buf,buf\_size | 将指定ini文件的内容读取到内存buf中。以’\0’标示结尾。 |
| **20** | **cm\_do\_decode** | **control,param\_content,ini\_file\_buf** | **解析ini文件。** |
| 21 | cm\_decode\_ini\_file | control,param\_content | 分配ini\_file\_buf并调用cm\_do\_decode |
| 22 | cm\_find\_def\_node | param\_content,section\_name,param\_name | 查找定义节点（就是键值对节点） |
| 23 | cm\_find\_def\_node\_in\_section | section,param\_name |  |
| 24 | cm\_add\_new\_param | param\_content,attr,prev\_attr | 添加一个新的参数到链表中 |
| 25 | cm\_put\_one\_line\_to\_buf |  | 这个函数要连续使用，在尾部添加一行数据 |
| 26 | cm\_do\_write | ini\_file, param\_content, ini\_file\_buf | 先写到tmp文件，再将原文件改名为bak文件，最后将tmp文件改名为配置文件。 |
| 27 | cm\_write\_ini\_file | ini\_file, param\_content | 包装cm\_do\_write，只是为他动态分配一个ini\_file\_buf，这样可以清晰地管理malloc。 |
| 28 | cm\_write\_params\_defval | control,param\_content\_report\_file\_path | 这个是管理工具使用的函数，将配置文件中缺失的选项填入默认值，并报告之 |
| 29 | cm\_add\_description\_to\_report\_file | description，file\_path | 与上联用的，给报告文件添加描述 |
| 30 | cm\_check\_all\_param\_exist | control | 顾名思义 |
| 31 | cm\_update\_check\_param | control, param\_content | 根据ini\_file两层双向链表param\_content更新control |
| 32 | cm\_free\_all\_params | param\_content | free双向链表 |
| 33 | cm\_set\_cfg\_initialization\_stat | ret | 设置g\_config\_initialized标志。 |
| 34 | cm\_cfg\_initialized |  | 读取上面标志 |
| 35 | cm\_load\_params | control | 从ini文件加载所有参数。 |
| 36 | cm\_write\_param | control,attr,param\_value | 更新参数值并写入文件，过程：**读取ini\_file，构建双层链表，从链表中查找指定参数节点并更新，将双层链表写入文件，释放链表**。 |
| 37 | cm\_init\_data\_path |  | 如果DATA\_PATH未填写则填入data，否则删除DATA\_PATH最后的路径分隔符。 |
| 38 | cm\_init\_redo\_path |  | 类上 |
| 39 | cm\_init\_params | ini\_file\_name,  param\_cnt,  param\_attr,  param\_data | 对外头文件中的对外接口，实际就是讲几个参数赋值给g\_param\_ctrl的各属性，然后调用cm\_load\_params读取参数值到g\_param\_ctrl.param\_data数组中。 |
| 40 | cm\_check\_redo\_data\_path |  | 检查redo/data/archive路径是否相同或相互有包含关系。 |
| 41 | cm\_load\_cfg | ini\_file\_name | 对外接口，详见后文分析 |
| 42 | cm\_attach\_cfg |  | 设置各全局变量关系 |

所有85个参数的默认值(及其他属性如上下限等)都存放在全局变量**g\_param\_attr**中：

static const cm\_param\_attr\_t **g\_param\_attr**[]

注意到其最后一列为validate function，其中只有PAGE\_SIZE/MAX\_SESSIONS这两个参数有验证函数，而验证函数分别是上面私有接口表的第1,2个函数。其他参数有这个能力，但是无需验证。

我们操作的每个参数的属性都由以下结构体定义，它是一个较关键的结构体。

typedef struct tagcm\_param\_attr

{

db\_uint32 id; /\* parameter ID \*/

const db\_char \*section; /\* parameter section \*/

const db\_char \*name; /\* parameter name \*/

const db\_char \*description; /\* parameter description \*/

cm\_param\_type\_e type;/\* parameter value type:int32/uint32/string/IP/file/path/fullpathname \*/

const db\_char \*defval; /\* parameter default value, set to "" if default value unavailable \*/

cm\_param\_modifiable\_e modifiable; /\* parameter modifiable flag: recreate DB/reboot server/reboot app \*/

db\_int64 min; /\* minimal value or string length of the parameter \*/

db\_int64 max; /\* maximal value or string length of the parameter \*/

cm\_param\_validate validate; /\* parameter value validation callback function \*/

}cm\_param\_attr\_t;

注意这个结构体中并没有参数的实际值，参数的实际值不存储在这里，而是用以下结构体来组织：

/\* paramter value union \*/

typedef union tagcm\_param\_value

{

db\_int32 int32;

db\_uint32 uint32;

db\_char str[CM\_PARAM\_MAX\_VAL\_LEN];

}cm\_param\_value\_u;

/\* parameter data \*/

typedef struct tagcm\_param\_data

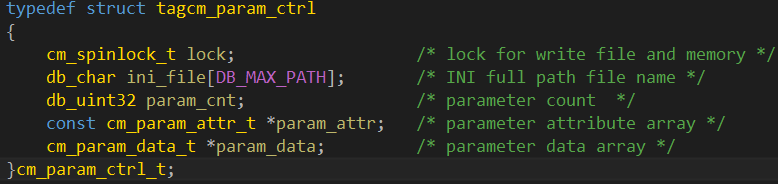
{

db\_bool read\_from\_file; /\* DB\_TRUE means parameter value has been initialized by INI file \*/

cm\_param\_value\_u value; /\* parameter value \*/

}cm\_param\_data\_t;

而最终的数据存储在以下结构体中：



*/\* judge whether the value is power of 2 or not \*/*

*#define CM\_IS\_VAL\_POWER\_OF\_2(val) (0 == (((val) - 1) & (val)))*

*上面妙用了一个数学原理：(x-1)&x能去掉x的二进制中最右边的1。*

由此也可以看到，参数组织的原理是把值和属性分开，类似于C语言中类型和变量的关系。

对IP地址的检查包含格式检查，还调用了一个inet\_addr/inet\_aton来将ip地址转换为32位数值，以判断ip地址有效性。

### 行解析

cm\_decode\_line函数，该函数首先trim掉前后的非可视字符，然后根据首字符判断行类型：

|  |  |
| --- | --- |
| 首字符 | 行类型 |
| ‘\0’ | 空行 |
| ‘#’ | 注释行 |
| 首字符’[’且尾字符’]’ | 节名，trim掉首位空白，得到节名 |
| 除此之外，都是键值对行，数据行 | 查找=号 |

这个函数使用inplace解析，即在buffer上写，返回的也是buffer上的指针。

### 文件解析

cm\_do\_decode函数执行了真正的解析工作，把整个ini文件解析成两层的双向链表。如下图示：



虽然ini文件比较自由，但是从cm\_add\_new\_param函数可以看出，其实际上包含一些规律：

每个参数在ini文件中由5行组成：

|  |  |
| --- | --- |
| 1 | 空白行 |
| 2 | 注释行 |
| 3 | 修改后起效时间 |
| 4 | 取值规范、范围描述 |
| 5 | 键值对 |

### 一个使用双向链表的好的实践：

typedef struct tagcm\_dlist\_head\_t

{

struct tagcm\_dlist\_head\_t \*next;

struct tagcm\_dlist\_head\_t \*prev;

db\_void \*self; /\* this member pointer to user defined structure. And it must be set to NULL, if it is head node. \*/

}cm\_dlist\_head\_t;

/\*\*

\* CM\_INIT\_DLIST\_HEAD - init head node of the double link list

\* $head: list head

\*/

#define CM\_INIT\_DLIST\_HEAD(head) \

do \

{\

(head)->next = (head); \

(head)->prev = (head); \

(head)->self = NULL; \

}/\*lint -save -e717\*//\*PL/MDE: zhangrixin 197021 717:do ... while(0)\*/ while (0) /\*lint -restore \*/

/\*\*

\* CM\_INIT\_DLIST\_NODE - init common node of the double link list

\* $node: pointer to common node

\* $member: member name in $node for cm\_dlist\_head\_t

\*/

#define CM\_INIT\_DLIST\_NODE(node, member) \

do \

{\

(node)->member.next = NULL; \

(node)->member.prev = NULL; \

((node)->member.self = (node)); \

}/\*lint -save -e717\*//\*PL/MDE: zhangrixin 197021 717:do ... while(0)\*/ while (0) /\*lint -restore \*/

以上定义了一个双向链表以及它的节点初始化宏（头节点初始化宏不用管）。

使用双向链表的数据结构定义如下：

typedef struct tagcm\_param\_node

{

cm\_dlist\_head\_t node; /\* dll node \*/

cm\_ini\_line\_type\_e type; /\* parameter node type \*/

cm\_param\_node\_data\_u data; /\* parameter node union \*/

}cm\_param\_node\_t;

其关键就在于目标**数据结构包含DLL节点！**DLL节点串联起来后，目标数据结构其实也就被串联起来了。因为DLL节点中包含void \*指针指向的是该节点所在的结构体对象。画成图如下：

node

node

node

node

head

CM\_INIT\_DLIST\_NODE两个参数node和member分别是

|  |  |
| --- | --- |
| node | 包含DLL节点的结构体对象 |
| member | DLL节点在node中的名称 |

这样的好处就是申请内存比较方便，只需要申请一次，就得到DLL节点和目标数据。另一个好处是用到通用的链表操作接口。只需要一套接口，不像C++ template那样。坏处依然是C语言处理通用数据类型绕不开的缺点：它绕过了类型检测系统导致容易出现难以调试的错误。

这个使用方式模拟了C++的继承，子类对象包含父类对象。

**一个改进是约定node需要作为节点的首元素，这样可以省略self指针，因为node的地址就是包含它对象的地址**。

### 写文件

cm\_do\_write函数实现了写文件的功能。

它遍历每个section，对任一个section，将它写入buffer，然后遍历该section的sub\_nodes也就是section下的每一行，将该行写入buffer。这样就还原了所有的行。之后就需要将buffer写入文件。

buffer写入文件需要保护，防止文件残缺。

* + - 1. 先将buffer写入一个临时文件tmp
      2. 将原配置文件改名为bak
      3. 将tmp改名为配置文件名
      4. 删除bak,tmp。

文件被破坏的风险是掉电、磁盘故障等，且时间正比。第一步写磁盘的故障发生时间长，故将它写入tmp，而不是立即覆盖原配置文件。第一步完成后，buffer已经在磁盘上。此时需要改掉原配置文件名或者删掉原配置文件。改名显然更保险快速，不易出错。所以第三步也改名。这样第2、3步很快。而第一步又不影响原配置文件，所以风险低。

### 一个确保分配的内存被删除的方法：

使用包裹函数wrapper。假设func\_1要申请一段内存，那么可以用func\_2包裹func\_1，并在func\_2中申请内存，这样释放内存的地方很好拿捏，可以降低内存泄露风险。

### 全局变量及关系

|  |  |  |
| --- | --- | --- |
| 全局变量名 | 类型、作用描述 | 关系描述 |
| g\_param\_attr | cm\_param\_attr\_t | 定义了所有参数的属性，被g\_param\_ctrl指向 |
| g\_param\_data | cm\_param\_data\_t | 参数数组指针 |
| g\_param\_ctrl | cm\_param\_ctrl\_t | 全局控制块，一般其会指向g\_param\_data |
| g\_config\_initialized | db\_bool | 是否初始化标志 |

### 加载流程

cm\_load\_cfg函数：

首先从共享内存中获取“其他”类共享内存起始地址，赋值给全局变量g\_param\_data。

然后调用cm\_init\_params函数，初始化全局控制块g\_param\_ctrl。调用原型如下：

cm\_init\_params(ini\_file\_name, ELEMENT\_COUNT(**g\_param\_attr**), **g\_param\_attr**, **g\_param\_data**);

该函数设置全局数据块g\_param\_ctrl，并调用cm\_load\_params函数执行加载。整体流程如下图示：



### 参数设置流程

cm\_set\_param用于参数设置。原型如下：

db\_int32 cm\_set\_param(db\_uint32 sid, const db\_char \*param\_name, const db\_char \*param\_value)

首先根据参数名，得到参数属性attr，看该参数是否可修改，如果不是PARAM\_MOD\_IMMEDIATE，则报错，也就是说该接口只能设置立即生效的参数。

接着使用cm\_fmt\_param，将参数现有值打印成串，与新值对比，若两者相同则成功退出。否则，用cm\_check\_param检查参数值,并调用参数属性中的validate回调函数检查参数值的合法性。然后调用cm\_write\_param写参数值。

这个cm\_write\_param前面我们已经分析过，它会**读取ini\_file，构建双层链表，从链表中查找指定参数节点并更新，将双层链表写入文件，释放链表。**

也就是说它会直接写文件。

接着将这个值复制到控制块的参数数组中使之生效。

最后有几个参数存放在全局变量中，在写了参数之后，这几个全局变量要重新从配置中读取。

|  |  |
| --- | --- |
| g\_core\_enable | CM\_PARAM\_CORE\_ENABLE |
| g\_sync\_write | CM\_PARAM\_SYNC\_WRITE |

# GMDB的编译流程

## 在Linux下

在linux下我们通过执行pkg/make目录下的gmdb\_make.sh文件来编译工程。它会自动调用make程序。

### gmdb\_make.sh文件解析：

首先COMP\_DIRS设定了要编译的目录

|  |  |
| --- | --- |
| common |  |
| kernel |  |
| rep |  |
| pldbg |  |
| pl |  |
| sql |  |
| srepknl |  |
| odbc |  |
| tree |  |
| treeserver |  |
| tools |  |

这些目录都在pkg/src目录下。

然后通过切换工作目录到gmdb\_make.sh文件所在目录，将GMDB\_HOME设置为它所在目录的上层路面，也就是包含pkg文件夹的目录。

接着export一些路径变量。

接着设定一些变量作为标志位。

接着是入参检测与执行，系统版本、编译器版本的获取。

可以接收以下参数：

|  |  |
| --- | --- |
| version |  |
| check |  |
| clean | 删除所有构建选项 |
| all |  |
| lcov |  |
| show |  |
| purify|purecov|quantify |  |
| crc |  |
| release |  |
| debug |  |
| llt |  |
| package |  |
| jdbc |  |
| \* | 其他的原样 |

各个选项的值连接起来放到MAKE\_OPTIONS值中。

其中有一个逻辑是如果没有指定是release/debug/clean，则默认为debug，并且把DEBUG字样写入到.compile\_type文件。

其中一个函数gmdb\_compile\_info\_update，用于更新db\_version.h中的版本信息。

**疑问：这样不会导致大量代码被重新编译吗？除非这个头文件不会被大量源码包含，只会被小部分源码包含，例如仅被一个叫db\_version.c的包含。这样就只有db\_version.c会被重新编译。在其他文件中要获取版本信息时，要调用db\_version.c中的函数或全局变量**。

fastore\_compile\_info\_update()用于更新db\_faversion.h中的日期信息，包括FA\_PRODUCE\_DATE

gmdb\_make()函数一次编译上面各个源代码目录下的代码。

其机制是将工作目录设置到目标文件夹下面，将所有的文件设为700的权限，然后调用make clean，接着make –j12 -w ${MAKE\_OPTIONS}

对于treeserver，在编译完成后需要创建publish目录并复制几个头文件到include下面：

|  |  |
| --- | --- |
| gm\_types.h |  |
| gm\_defs.h |  |
| gm\_common.h |  |
| gm\_error\_code.h |  |
| gmdb.h |  |
| gm\_tools.h |  |

每个目标编译失败后立即退出。

所有目标编译完成后，就会编译clouddb.so。**首先cd到GMDB\_HOME/make/so\_mk目录下，执行make clean，再执行make**。

在脚本的最后，执行jansson、gmdb、tree\_client的make。

总结：

首先建立各种变量，且读取命令行，然后编译各个target，编译时进入target目录并执行make clean+make。

make时都带有MAKE\_OPTIONS入参。一些公共的变量也已经在gmdb\_make.sh被export，尤其是GMDB\_HOME这个变量。有了这个变量，各个makefile就能以此为根去查找需要的文件。

### $GMDB\_HOME/make/common.in

所有的makefile都会包含GMDB\_HOME/make/common.in这个公共的makefile文件，这个文件的内容解析为：

|  |  |
| --- | --- |
| 公用程序或命令类： | AR打包库的程序  LINKLIB=ln -s  COMPLILER  RM=rm -fr |
| 标志类： |  |
| 选项类： | CFLAGS：默认就定义了DB\_KV  PTHREAD\_LIB=-lpthread  DL\_LIB=-ldl  MATH\_LIB=-lm  COMMON\_LIB=-lgmcommon  KERNEL\_LIB=-lgmkernel  SQLSERVER\_LIB=-lgmsqlserver  SQLCLIENT\_LIB=-lgmsqlclient  REP\_LIB=-lgmrep  SREP\_KNL\_LIB=-lgmsrepknl  PL\_LIB=-lgmpl  PL\_DBG\_LIB=-lgmpldbg  MD5\_LIB=-lgmmd5  ZLIB\_LIB=-lgmzlib  LZ4\_LIB=-lgmlz4  SSL\_LIB=  RT\_LIB=-lrt  TREE\_LIB=-lgmtree  TREESERVER\_LIB=-lgmtreeserver |
|  |  |
| 符号替换类： | HIDE=@  PRINT=echo  SO=so  SODLL=$SO  CO=-o  LCO=-o $@  AR\_LCO=-o $@  OBJFIX=o |
| 路径类 | PUBLISH\_DIR=${GMDB\_HOME}/publish  GMDB\_LIB=${GMDB\_HOME}/lib其实就是pkg/lib  ADD\_ONS\_LIB=../../add-ons/lib其实就是pkg/add-ons/lib  GMDB\_OBJ=../../obj 其实就是pkg/obj  LINKLAG=-L${GMDB\_LIB} -L${GMDB\_HOME}/add-ons/lib |

ifdef show

HIDE=

PRINT=echo

endif

如果传入参数show给gmdb\_make.sh，那么它会出现在MAKE\_OPTIONS中传递给make程序，进而在common.in中反映出来。然后将HIDE由@改为空，使得更多信息被打印出来。

ifdef debug

OFLAG=-O0

ifeq ($(OS\_NAME), Linux)

CFLAGS+=$(OFLAG) -DDB\_DEBUG\_VERSION -fstack-protector-all

CFLAGS\_HASH+=$(OFLAG) -DDB\_DEBUG\_VERSION -fstack-protector-all

else

CFLAGS+=$(OFLAG) -DDB\_DEBUG\_VERSION

CFLAGS\_HASH+=$(OFLAG) -DDB\_DEBUG\_VERSION

endif

endif

这里是说，如果debug选项被传入gmdb\_make.sh进而传入了make，那么就会将OFLAG这个优化选项设为-O0，即不优化。同时对Linux平台，会将优化选项加入到CFLAGS，还会定义DB\_DEBUG\_VERSION宏，打开这个编译器标志-fstack-protector-all。

[参考这里](https://www.ibm.com/developerworks/cn/linux/l-cn-gccstack/)可以得到关于GCC堆栈保护技术的实现。这个在我当年写的8051多任务调度程序中也有用到。由于硬件资源十分有限，每个任务的栈不能超过限定值，所以当时也是做了栈溢出的检测，这种检测必须在汇编级别，而调度程序必须使用汇编才能完成，所以也正契合。

ifdef crc

CFLAGS+=$(OFLAG) -DDB\_DEBUG\_VERSION -DDB\_CRC\_CHECK

CFLAGS\_HASH+=$(OFLAG) -DDB\_DEBUG\_VERSION -DDB\_CRC\_CHECK

endif

ifdef release

OFLAG=-O2

OFLAG\_HP=-O2

ifeq ($(OS\_NAME), HP-UX)

CFLAGS+=$(OFLAG\_HP)

else

CFLAGS+=$(OFLAG)

endif

ifeq ($(OS\_NAME), AIX)

CFLAGS\_HASH+=$(OFLAG) -qalias=noansi

else

CFLAGS\_HASH+=$(OFLAG)

endif

endif

ifdef purify

EXE\_COMPILER=purify

TOOLS\_COMPILER=purify

endif

ifdef purecov

EXE\_COMPILER=purecov

TOOLS\_COMPILER=purecov

endif

ifdef quantify

EXE\_COMPILER=quantify

TOOLS\_COMPILER=quantify

endif

以下是代码覆盖率选项：

ifdef lcov

CFLAGS+=-DBUILD\_LCOV -fprofile-arcs -ftest-coverage

CFLAGS\_HASH+=-DBUILD\_LCOV -fprofile-arcs -ftest-coverage

LINKLAG+=-lgcov

LIBFLAG+=-lgcov

endif

以下是llt工具（似乎是基于CUnit的一个测试框架）

ifdef llt

CFLAGS+=$(OFLAG) -DDB\_DEBUG\_VERSION

CFLAGS\_HASH+=$(OFLAG) -DDB\_DEBUG\_VERSION

LINKLAG+=-L${HUTAF\_HOME}

LINKLAG+=-lNCSCore\_suse10\_x86-64

EXE\_COMPILER=lltwrapper

TOOLS\_COMPILER=lltwrapper

COMPILER=lltwrapper

endif

INC路径：

|  |  |
| --- | --- |
| 1 | ${GMDB\_HOME}/add-ons/opensource/jansson/src |
| 2 | ${GMDB\_HOME}/add-ons/platform/uoa/include |
| 3 | ${GMDB\_HOME}/add-ons/platform/vpp/include/CRYPTO |
| 4 | ${GMDB\_HOME}/add-ons/platform/vpp/include/OSAL |
| 5 | ${GMDB\_HOME}/add-ons/platform/vpp/include/PSE |
| 6 | ${GMDB\_HOME}/add-ons/platform/vpp/include/SSL |
| 7 | ${GMDB\_HOME}/add-ons/platform/vpp/src |
| 8 | ${GMDB\_HOME}/src/base/rdma/common |
| 9 | ${GMDB\_HOME}/src/base/rdma/rdma\_transport |
| 10 | ${GMDB\_HOME}/src/common/include |
| 11 | ${GMDB\_HOME}/src/common/src |
| 12 | ${GMDB\_HOME}/src/kernel/include |
| 13 | ${GMDB\_HOME}/src/kernel/src |
| 14 | ${GMDB\_HOME}/src/pl/include |
| 15 | ${GMDB\_HOME}/src/pl/src |
| 16 | ${GMDB\_HOME}/src/rep/include |
| 17 | ${GMDB\_HOME}/src/rep/src |
| 18 | ${GMDB\_HOME}/src/sql/client |
| 19 | ${GMDB\_HOME}/src/sql/include |
| 20 | ${GMDB\_HOME}/src/sql/server |
| 21 | ${GMDB\_HOME}/src/sql/sqlcomm |
| 22 | ${GMDB\_HOME}/src/src |
| 23 | ${GMDB\_HOME}/src/srep/kernel/include |
| 24 | ${GMDB\_HOME}/src/srep/kernel/src |
| 25 | ${GMDB\_HOME}/src/srep/tools/srbar |
| 26 | ${GMDB\_HOME}/src/srep/tools/srclient |
| 27 | ${GMDB\_HOME}/src/srep/tools/srstat |
| 28 | ${GMDB\_HOME}/src/tools/common/include |
| 29 | ${GMDB\_HOME}/src/tree/include |
| 30 | ${GMDB\_HOME}/src/tree/src |
| 31 | ${GMDB\_HOME}/src/treeserver\* |

\*为定义TREE\_CLIENT\_LIB才包含

接着是平台相关选项。我们这里只看Linux：

OS\_DIR=linux

CFLAGS+=-g –fPIC –Wall –MMD –mtune=Nocona –fno-strict-aliasing -D\_GNU\_SOURCE -D\_LARGEFILE64\_SOURCE -DKNL\_PREFIX

其中-MMD是输出一个适用于make程序的依赖关系文件。

然后根据平台是SUSE还是Redhat，定义宏：

SUSE： -D\_SUSE\_11

RedHat: -D\_RHEL

LIBFLAG+=-shared $(CFLAGS) -L${GMDB\_LIB} -L${GMDB\_HOME}/add-ons/lib

工具链设定：

EXE\_COMPILER+=gcc

COMPILER+=gcc

TREE\_COMPILER+=g++

TOOLS\_COMPILER+=g++

LINK=${TOOLS\_COMPILER}

TOOLS\_LINK=${LINK}

### so\_mk/makefile

它定义了共享库的路径，链接出一个libclouddb\_tmp.so，再mv为libclouddb.so

### src/tree/makefile

首先依然是包含了make/common.in

INC+=-I${GMDB\_HOME}/add-ons/opensource/jansson-2.7/src

TREESRCS = $(wildcard src/\*.cpp)

TREE\_OBJS = $(patsubst src/%.cpp, ${GMDB\_OBJ}/tree/%.$(OBJFIX), $(TREESRCS))

其中GMDB\_OBJ就是pkg/obj/tree。编译输出.o时，由于有-MMD编译选项也顺便会输出.d文件。

**疑问：这个.d文件是如何生效的？这里我们并没有在编译.o时添加对.d文件的依赖。我们这里没有依赖关系，所以全部都先make clean再make。**

**实际上就是没有使用.d。包括之前内核的模块如common等也没有使用增量编译方式。每次都会make clean然后重新make。这样效率不高。**

### 调试src/treeserver/makefile

查看这个makefile，发现它还是比较简单的。依赖关系如下:

all:treeserver

treeserver:${PUBLISH\_LIB\_CLOUDDB}

${PUBLISH\_LIB\_CLOUDDB}:$(TREESERVER\_OBJS) $(STATIC\_LIB\_DEPENDS)

$(BUILD\_LIBRARY)

#$(HIDE) $(AR) $@ $^

#ranlib $@

$(TREESERVER\_OBJS):${GMDB\_OBJ}/treeserver/%.$(OBJFIX):%.cpp

@$(PRINT) "Compiling file "$<

$(HIDE) $(TREE\_COMPILER) $(CFLAGS) $(INC) $(CO)$@ -c $<

想要这个makefile工作，必须预定义一些全局变量，这里是：

GMDB\_HOME, PUBLISH\_LIB\_CLOUDDB

所以在调试前需要设置这两者。可以根据gmdb\_make.sh中设置：

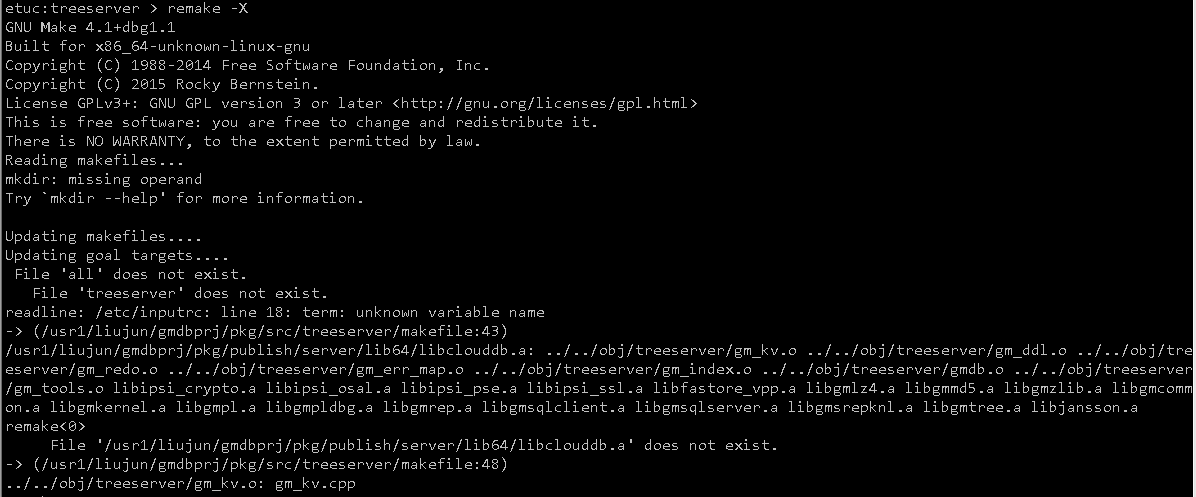
cd "$(dirname $0)"

export **GMDB\_HOME**="$PWD/../"

export PUBLISH\_CLOUDDB\_DIR=${GMDB\_HOME}/publish/server

export **PUBLISH\_LIB\_CLOUDDB**=${PUBLISH\_CLOUDDB\_DIR}/lib64/libclouddb.a

之后使用remake –X启动调试。在调试前，最好make clean，以便我们能查看完整的从0开始构建的流程。



使用p打印变量，使用s单步执行

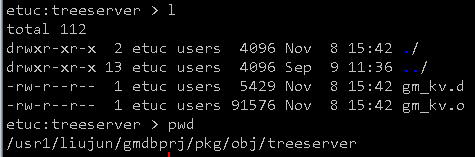


直接按回车重复上次命令，这里是s，单步

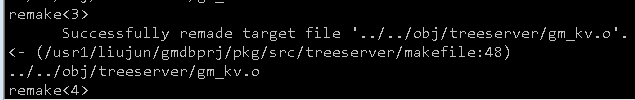


这里是对gm\_kv.o的编译流程

注意到虽然我们使用了-MMD，但是也用了-o，所以同时会生成gm\_kv.d和gm\_kv.o这两个文件：



继续单步。



然后是下一个文件gm\_ddl.cpp的编译了。出错时有如下信息：



### ut工程

ut工程中使用到了gtest，所以编译时也需要链接gtest库，ut工程有相对独立的工具链配置，放在ut/stub/common.in中。但依然包含全局的make/common.in。

它也有一个stub.cpp源文件，用于这个文件提供了打桩的功能，通过调用其提供的接口，可以替换动态库中函数为自定义函数，实现高层函数的单元测试。

~~这个函数在DTS中不会使用到。因为DTS不是单元测试，而是功能测试~~。

当前我们的makefile编译支持多平台，但从实际考虑可以简化，只支持Linux工程即可。

。参考DS的makefile。

### 使用makefile进行增量编译的方法及原理

源代码被修改后，依赖关系也可能被修改，手动写的依赖关系不靠谱。编译器可以自动生成依赖关系，例如GCC的-MM选项就可以输出依赖关系。

例如假设input.c包含input.h，则gcc –MM input.c输出如下：

input.o: input.c input.h

这是实现自动增量编译的关键之一。

在编译input.o时，我们需要得到这个依赖关系，并据此决定是否重新生成input.o，但是要生成依赖关系，又需要重新编译一次才行。这就形成一个悖论，似乎在一次编译过程中难以实现该功能。

实际的编译过程中，确实无法一次实现。以下讲述的方法，**实际上每次编译都是利用上一次生成的依赖关系**，乍一看这样似乎不对，其实是对的，后面讨论。

先写出一种实现的思路：

SRC:=input.c output.c main.c

OBJ:=$(patsubst %.c,%.o,$(SRC))

DEP:=$(patsubst %.c,%.d,$(SRC))

CC=gcc

CPPFLAGS=

all:main

main:$(OBJ)

$(CC) -o main $(OBJ)

%.o:%.c

$(CC) $(CPPFLAGS) -c $<

%.d:%.c

@set -e; rm -f $@;\

$(CC) -MM $(CPPFLAGS) $< > $@.$$$$;\

sed 's,\($\*\)\.o[ :]\*,\1.o $@ : ,g' < $@.$$$$ > $@;\

rm -f $@.$$$$;

-include $(DEP)

.PHONY:clean

clean:

-rm -rf $(OBJ) main $(DEP) \*.d.\*

首先定义了输入文件，是三个.c文件集合，以及同名.o/.d文件集合。

|  |  |
| --- | --- |
| SRC | input.c output.c main.c |
| OBJ | input.o output.o main.o |
| DEP | input.d output.d main.d |

然后定义了伪目标all，目标main。

接着是两个静态模式%.o:%.c和%.d:%.c。第一个模式定义了.o目标的生成命令。第二个定义了.d文件的生成命令。设两个相当于设定了两个规则，**这两个规则起作用的前提是，其他地方定义的.o/.d不带命令**。

接着是-include包含依赖文件.d。

makefile的执行顺序如下：

1、读入所有的Makefile。

2、读入被include的其它Makefile。

3、初始化文件中的变量。

4、推导隐晦规则，并分析所有规则。

5、为所有的目标文件创建依赖关系链。

6、根据依赖关系，决定哪些目标要重新生成。

7、执行生成命令。

注意，在执行任何目标下的命令之前，make程序会做很多事情，尤其是：

会先执行include命令，不管该命令在什么地方，即他不是从上往下执行的。针对我们上面的makefile，就是会先执行-include $(DEP)命令。

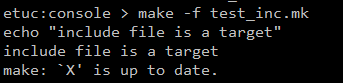
在首次运行时，.d文件不存在，include比如失败。**make会把include命令的参数也当做目标，去套用规则。这一点及其重要和关键并且非常隐晦，因为只有这样，才有可能套用我们定义的%d:%c规则去生成第一次的.d文件。**要验证这一点，可以把-include $(DEP)这行注释掉，再make看看会不会生成.d，经过测试不会生成.d。这是侧面验证，正面的验证：假设有如下makefile叫test\_inc.mk，内容如下

-include X

X:

echo "include file is a target"

执行make:

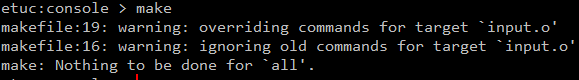
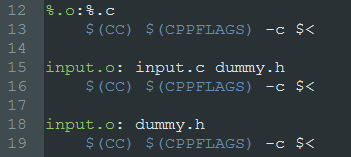


可以看到include实际上也隐含定义了一种依赖关系。

还有一点也非常关键，就是我们的.d文件的内容经过上面的makefile，会变成：

input.o input.d: input.c input.h

这条规则只定义了依赖关系，没有定义要执行的命令，这种规则会与其他同名规则合并。我们定义了模式规则%.o:%.c，这个%.o与input.o是会有重复的，make会合并依赖，并根据依赖关系决定是否执行命令。对同一个目标，如果多次定义中都有命令，那么最后的命令生效，例如：



首先第15、18行的显式规则会覆盖第12行的模式规则，然后18行的规则中的命令又替换了较前的15行规则中的命令。

思考：为什么可以复用上一次生成的.d文件作为本次编译的依赖？

这个问题分两种情况，一种是依赖关系不变，即没有多依赖任何一个文件也没有少依赖任何文件，这种情况显然上次的.d可以复用。

另一种情况是依赖关系本身改变了，比如，增加了一个头文件的依赖，那么上次的依赖关系还可以用吗？答案是可以的，因为我们必须要修改某个依赖链条中的文件，才能将新的头文件添加到依赖链中来。简而言之，要构建新的依赖链，就必须修改旧的依赖链中的某个文件。也就是说，上一次的依赖关系总能捕捉到任何修改，从而更新依赖关系以及重新编译目标。

但是上面的代码有一种情况无法正确处理，就是从依赖链中删除某个依赖的文件，并且从物理上删除该文件，那么使用上次的依赖关系将导致因该文件不存在而make失败。解决的办法就是使用GCC –MP命令，将被依赖的头文件声明为空目标。详见[此网站](http://www.evanjones.ca/makefile-dependencies.html)[Correct incremental builds with Makefiles]。里面解释了正确的增量编译的关键。

|  |  |
| --- | --- |
| -MF file | When used with -M or -MM, specifies a file to write the dependencies to. If no -MF switch is given the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options -MD or -MMD, -MF overrides the default dependency output file. |
| -MP | This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors make gives if you remove header files without updating the Makefile to match. This is typical output:  test.o: test.c test.h  test.h: |
| -MD | -MD is equivalent to -M -MF file, except that -E is not implied. The driver determines file based on whether an –o option is given. If it is, the driver uses its argument but with a suffix of .d, otherwise it takes the name of the input file, removes any directory components and suffix, and applies a .d suffix.  If -MD is used in conjunction with -E, any -o switch is understood to specify the dependency output file, but if used without -E, each -o is understood to specify a target object file.  Since -E is not implied, -MD can be used to generate a dependency output file as a side-effect of the compilation process. |
| -MMD | Like -MD except mention only user header files, not system header files. |

其实我们前面的实现稍微复杂，我们可以不需要%.o:%.c这个模式，而使用如下的简化的makefile:

SRC:=input.c output.c main.c

OBJ:=$(patsubst %.c,%.o,$(SRC))

DEP:=$(patsubst %.c,%.d,$(SRC))

CC=gcc

CPPFLAGS=

main:$(OBJ)

$(CC) -o main $(OBJ)

$(OBJ):%.o:%.c

$(CC) $(CPPFLAGS) -MMD -MP -o $@ -c $<

-include $(DEP)

.PHONY:clean

clean:

-rm -rf $(OBJ) main $(DEP)

在这个文件中，我们在生成.o文件的同时生成.d文件。第一次运行时，include没有，也就没有额外的依赖给.o，所有.o都只依赖与模式$(OBJ):%.o:%.c,所有的.o都会被生成，同时生成一个.d文件，供之后的编译使用。第二次开始，就可以包含上次生成.d文件从而使用增量编译了。

要求所有OBJ文件都要正确第被清理，所有.d都要被生成。如果用户手动删除了.d，没有删除.o文件，怎么办？这种情况下，就没法处理了，只要不修改.c，修改头文件，.o就不会更新，也就不会生成.d了。但是使用之前的办法，就会自动生成.d，但是.d虽然生成了，这一次的编译由于没有.d，所以也没有用。要确保.d被生成，可以使用添加模式%.d:%.c，并且使用简化的规则：

%.d:%.c

$(CC) $(CPPFLAGS) -MM -MP -MF $@ -c $<

也就是不要替换里面的内容，不需要增加.d的依赖。考虑到我们经常会在linux和windows之间倒换文件，可能误删除.d，所以这一个模式还是挺重要的。

如果想要使误删.d的下次编译仍然正确，只需添加.o对.d的依赖：

$(OBJ):%.o:%.c %.d

$(CC) $(CPPFLAGS) -MMD -MP -o $@ -c $\*.c

这样当.d不存在时，会重新编译.o，确保在无法确定依赖关系时，执行编译动作。而之前的模式下，如果.d被删除了，.o的依赖就只剩下.c，而这就很可能导致.c不被编译而产生错误。

最后总结一下：

**每次编译都是利用上一次生成的依赖关系**

**多依赖关系下，显式依赖覆盖隐式依赖，后面的命令覆盖前面的命令。**

**make会把include命令的参数也当做目标，去套用规则。**

其中第三点很不好理解，在我们新模板中，也已经不需要这一点了。

# gmsql工具源码阅读与功能开发

## 源码阅读

工程源代码列表：

|  |  |
| --- | --- |
| gmsql\_command.h/.c |  |
| gmsql\_history.h/.c | 历史命令记录相应 |
| gmsql\_pub.h/.c | 控制台打印输出相关函数 |
| gmsql\_main.c | 主函数 |
| UnitTests |  |
| tool\_param. h/.c |  |
| tool\_procedure. h/.c |  |
| tool\_public.h/.c |  |

文件之间的关系：

gmsql\_pub是最顶层，然后是history/command/main.

我们的gmsql工具在设计上隐含了我们是串行执行的前提。即单线程工具。所以像@命令这个从文件读取语句的命令会仅用一个全局变量表示当前要执行的文件即可。

实际上我们把许多的数据存放在一个全局结构体变量里面，随着程序执行，这个数据变量不断更新，由于是串行，所以这样做没有问题。

### main.c

main函数流程：

gmsql\_init()初始化

gmsql\_wrapper\_init\_hist\_cmd\_list

gmsql\_parse\_args

gmsql\_print\_version

进入处理循环：do/while,

gmsql\_receive\_command

gmsql\_wrapper\_save\_hist\_cmd

gmsql\_fetch\_one\_cmd

gmsql\_get\_command\_code

gmsql\_excute\_command

退出循环

gmsql\_wrapper\_uninit\_hist\_cmd\_list

gmsql\_uninit

### gmsql\_command.h/.c

gmsql\_cmd\_e定义了支持的命令类型。

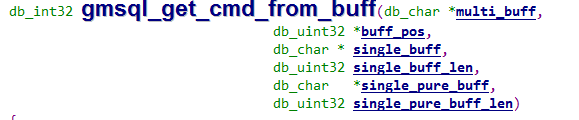
gmsql\_cmd\_word\_type\_e定义了词类型

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | gmsql\_receive\_command | buff, buff\_len | 接收命令存入buff。  linux下会设定终端模式，使得可以直接读取终端buffer从而处理上下左右键等特殊键信息。而读取完成后又会还原终端设置。在Windows下无需特殊处理，直接用fgets读入一行就行。 |
| 2 | gmsql\_get\_cmd\_from\_file | sgl\_command, sgl\_command\_len, single\_pure\_buf, single\_pure\_buflen | 从行缓存中读取一条命令。如果命令不完整，则再从文件中读取一行到行缓存中，继续执行读取一条命令，循环直到读取成功或失败。 |
| 3 | gmsql\_at\_cmd\_init | at\_file\_name | 打开文件，reset g\_gmsql\_local\_config.at\_cmd 结构 |
| 4 | gmsql\_is\_pl\_end | cmd\_start, division\_pos, end\_pos | PL/SQL 语句以一行单独的’/’字符结尾（可有空白）这个函数判断从cmd\_start到division\_pos处是否为单独的’/’，然后将结束为止写入end\_pos。 |
| 5 | gmsql\_get\_cmd\_from\_buff | multi\_buff, buff\_pos, single\_buff, single\_buff\_len, single\_pure\_buff, single\_pure\_buff\_len | 从可能包含多个SQL语句的buffer中读取一条SQL语句。如果SQL语句未完成，那么返回未完成。状态图：见后。 |
| 6 | gmsql\_is\_cmd\_at | cmd, file\_name | @前面不能有可显示字符 |
| 7 | gmsql\_fetch\_one\_cmd | multi\_buff, buff\_pos, single\_buff, single\_buff\_len, single\_pure\_buff, single\_pure\_buff\_len | 从multi\_buff中获取一条命令。有状态的，处理@filename时，会从文件中读取。 |
| 8 | gmsql\_get\_first\_verb | sql\_cmd, cmd\_verb, cmd\_verb\_len | 获取首个有效单词（可以包含各种字符，分隔符见代码，只有5种。）这个函数我们在升级时要用到。以便让它支持OML。 |
| 9 | gmsql\_get\_command\_code\_by\_verb | cmd\_verb, cmd\_code | 先构造“,cmd,”，再在g\_cmd\_info中查询对应的命令条目，得到条目的处理函数。每个条目包含一个执行函数和一个帮助函数。我们要支持OML也需要修改这个函数。 |
| 10 | gmsql\_get\_command\_code |  | 利用8,9 |
| 11 | gmsql\_local\_cmd\_version\_proc | sql | version命令的执行函数，打印版本信息。 |
| 12 | gmsql\_local\_cmd\_help\_proc | sql | 打印version命令的帮助信息。 |
| 13 | gmsql\_local\_cmd\_describe\_proc | sql | decribe命令处理函数。 |
| 14 | 一系列set函数以及一个gmsql\_local\_cmd\_set\_proc和一个gmsql\_local\_cmd\_show\_proc |  | 用于显示和设置一些选项，例如数字格式、日期格式，timer开关，显示方向等。 |
| 15 | gmsql\_spool系列  语法为spool filename flags |  | 设置录制文件，可以把操作过程录制下来 |
| 16 | gmsql\_prt\_all\_col\_cfg |  | 打印列配置 |
| 17 | gmsql\_parse\_conn\_sql |  | 这个函数给出了原来的代码解析命令的模式，我们可能需要遵照这种模式来添加OML支持。 |
| 18 | gmsql\_sql\_pre\_and\_exc | conn, stmt, sql | prepare&execute sql语句，调用gmsql\_do\_confirm\_ifneed来对危险命令做提示。如果开关打开会记录准备时间和执行时间。 |
| 19 | gmsql\_sql\_fetch\_row\_val\_horizontal | stmt, column\_format, column\_count, buff, buff\_len, result\_size | 显示一行的信息 |
| 20 | gmsql\_col\_val\_to\_disp\_str |  |  |
| 21 | gmsql\_sql\_result\_output | stmt, row\_count | 打印一行，对某些语句，打印有多少行，这有利于翻页时知道大约还有多少。 |
| 22 | gmsql\_sql\_select\_data\_horizontal | conn, stmt, row\_count | select语句查询出来的结果的输出。我们可能也要参考其逻辑来做。先打印标题，再打印数据，要根据输出的数据的配置来做。 |
| 23 | gmsql\_excute\_command | cmd\_code, sql, sql\_pure | 所有sql执行的入口函数，仅仅是调用回调函数而已。 |
| 24 | gmsql\_get\_input\_cmd | cmd\_buff, cmd\_buff\_len, is\_first\_line, line\_num | 在windows下面直接就可以处理。Linux下需要根据具体情况来处理。 |

gmsql\_get\_cmd\_from\_buff状态转换示意图（不是100%准确）



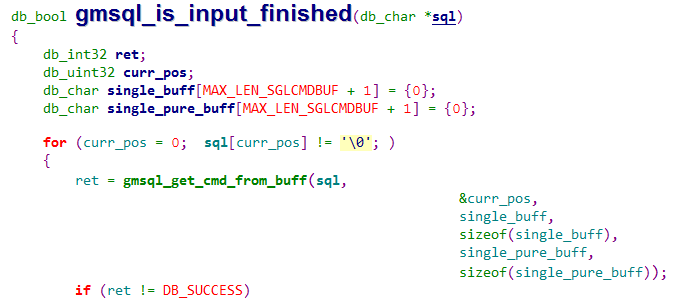
该函数的作用就是从一个可能包含不到一条、一条及多条命令的multi\_buff缓冲区中获取第一条完整的命令。再次查看其函数原型：



其中buff\_pos是一个输入输出参数，每次进入该函数后，从multi\_buff+\*buff\_pos开始处理，每处理一个字符都会将\*buff\_pos+1。这样，就可以多次调用该函数来获取命令。

在main函数中有gmsql\_receive\_command函数，它每次从命令行读取一行，然后调用gmsql\_is\_input\_finished来判断整个从第一行开始的串是否完成了。这样就使得整个输入被当做一条完整消息来判断，多行之间可以在一次过程中判断，使得状态得以保留，也就是说，第一行进入注释状态后，第二行依然在注释状态，因为第一行与第二行是一个整体。

而gmsql\_is\_input\_finished函数通过调用gmsql\_get\_cmd\_from\_buff来判断是否完成输入。它使用了一个for循环，



它判断所有语句都完整，才能顺利推出for循环并返回DB\_TRUE。这个逻辑下，最后一条语句不完整，就会返回未完成。

**Q:关于该函数的参数，single\_buff/single\_pure\_buff有何异同？**

**A:后者是不包含注释的语句。**

### gmsql\_pub.h/.c

与输出有关的，略。

### gmsql\_history.h/.c

关于这个文件可以参考：D:\DocumentsUSCDB\GMDB - story-all\V3R2C12\ONIP DataGrid V300R002C12 FaStore Story （gmsql支持history功能）.doc

类型：

|  |  |
| --- | --- |
| history\_cmd\_list\_node\_t | 包含命令字符串的链表节点 |
| history\_cmd\_list\_t | 历史命令管理，为支持上下翻，有cursor\_pos等元素 |

接口：

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | hist\_cmd\_list\_init | list, multi\_sql | 第二个参数是一个缓冲区，不使用动态分配而是用调用者的栈空间。这是避免使用动态分配的一种方法。这个链表结构在动态分配时是先分配了100个节点的连续内存结构，因为事先知道需要多少个元素，所以可以这样做。顺序存储的链式结构。 |
| 2 | hist\_cmd\_list\_uninit | list | 释放所有节点中的cmd以及链表本身 |
| 3 | hist\_cmd\_list\_reset\_attr | list |  |
| 4 | hist\_cmd\_list\_check\_arrow\_key | list, curr\_node, char\_ascii, cmd\_buf, cmd\_len |  |
| 5 | hist\_cmd\_list\_save\_text\_char | list, cmd\_buff, cmd\_buff\_len, char\_ascii, is\_first\_line, is\_enter\_key\_pressed |  |
| 6 | hist\_cmd\_list\_move\_last\_to\_first | list |  |
| 7 | hist\_cmd\_list\_alloc\_cmd\_mem | node, size |  |
| 8 | hist\_cmd\_list\_shield\_password | src\_cmd, err\_msg, inc\_word, dest\_cmd |  |
| 9 | hist\_cmd\_list\_add\_new\_cmd | list, new\_cmd |  |

历史命令不会从中间删除，所以这个链表从一开始就构造好，第一个元素指向第二个元素，第二个指向第三个以此类推。然后插入新命令时，放到最后一个元素，然后将最后一个元素移动到第一个元素。这样也自动实现了在历史命令达到最大时，删除最久远的命令的功能(LRU, least recently used)。

命令列表中元素的真实个数用list.node\_count来记录。

注意虽然用纯数组方式也完全能实现历史命令的功能，但是用链表的方式管理这个数组会简洁一些。注意链表中，越后面的元素越旧，所以处理up\_key时，实际上是取当前元素的next。

### 常用功能函数收集

内核带有一套自己的基础功能代码，例如处理字符串的代码等。开发新功能，必须要重用这些代码（注意其依赖方式不是直接源代码依赖而是通过库来依赖，例如下面几个文件都在common库中）。所以这里收集一下：

字符串处理类，参看文件：

|  |  |
| --- | --- |
| db\_text.h/.c |  |
| db\_utils.h/.h |  |
| db\_os\_adp.h/.c |  |
| db\_types.h |  |

重点看前两个。

db\_text.h/.c

|  |
| --- |
| cm\_getch |
| cm\_str\_ignore\_chars |
| cm\_str\_ignore\_single\_comment |
| cm\_str\_ignore\_multi\_comment |
| cm\_str\_starts\_with |
| cm\_str\_ends\_with |
| cm\_upper\_str |
| cm\_str\_comp |
| cm\_text\_comp |
| cm\_text\_equal |
| cm\_text\_rtrim |
| cm\_text\_ltrim |
| cm\_text\_trim |
| cm\_trim\_bracket |
| cm\_fetch\_text |
| cm\_is\_number |
| 各种数字处理、转换函数 |
| cm\_forward\_substr\_pos |
| cm\_backward\_substr\_pos |
| cm\_get\_substr\_pos |
| cm\_buf\_mgr\_init |
| 自己实现的内存管理 |

基本实现了一套对字符串的处理函数，Python的str的部分接口，C++的string的部分接口。

db\_utils.h/.c

|  |
| --- |
| cm\_get\_os\_type |
| cm\_get\_os\_name |
| cm\_get\_program\_fullname |
| cm\_init\_program\_name |
| cm\_get\_program\_name |
| cm\_get\_env |
| cm\_sleep/cm\_usleep/cm\_usleep\_ex |
| cm\_confirm |
| cm\_randomize/cm\_random |
| cm\_qsort\_partition |
| cm\_qsort\_recursive |
| cm\_quick\_sort |
| cm\_getuid |
| cm\_getgid |
| cm\_ipv4\_to\_str |
| cm\_ip\_valid |
| cm\_dns\_parse |
| cm\_path\_valid |
| cm\_right\_trim |
| cm\_left\_trim |
| cm\_trim\_invisible\_char |
| cm\_str\_to\_upper |
| cm\_str\_to\_lower |
| cm\_str\_to\_int64/uint64/int32/uint32/real |
| cm\_single\_instance\_lock |
| cm\_single\_instance\_unlock |
| cm\_crc\_64 |
| cm\_sigarray\_register |
| cm\_signle\_register |
| cm\_make\_background |
| cm\_malloc/cm\_free |
| cm\_memcpy\_s |
| cm\_generate\_bkfilename |
| cm\_new\_db\_id |
| cm\_compare\_db\_id |
| cm\_fmt\_db\_id |
| cm\_check\_digit |
| cm\_utc\_time\_to\_str |
| cm\_current\_user\_is\_owner |
| cm\_get\_pl\_text\_type |
| cm\_is\_interger |
| cm\_proc\_start\_time\_by\_ps |
| cm\_proc\_start\_time |
| cm\_get\_instance\_id |
| cm\_set\_instance\_id |
| cm\_get\_gmdb\_home |
| cm\_set\_gmdb\_home |

### 核心函数的实现

* main

其流程虽然简单，但还是有几点需要注意一下，几种在接收命令，执行命令的do-while循环中：

首先是否接收命令需要根据still\_have\_cmd标志来。当这个表示为真时，表示multi\_buff中尚有未执行完成的命令，此时不接受新命令，而是从multi\_buff中获取一条完整命令，交给gmsql\_execute\_command去执行。

而从命令行接收命令的函数gmsql\_receive\_command一定会等到输入的命令完整（通过gmsql\_is\_input\_finished检测）后才会返回。

* gmsql\_excute\_command

这个函数用于派发命令。根据命令id，从全局变量g\_cmd\_info中获取命令的回调函数，传入命令文本，执行代码。这个函数不需要我们修改。

main函数的逻辑保证了给gmsql\_execute\_command的参数一定是一条完整的语句。

* gmsql\_sql\_command\_excute\_proc：

这个函数用于执行sql语句，理解它对我们执行oml语句有帮助。主要考虑如何实现sql的解释以及sql结果的输出。

流程：

* + - 1. gmsql\_sql\_pre\_and\_exc，简略来说就是包含以下两步：
         1. gm\_prepare
         2. gm\_execute
      2. gm\_get\_stmt\_type，类型映射
      3. switch(sql\_cmd\_type)

|  |  |
| --- | --- |
| COMMAND TYPE | 执行代码 |
| explain/select | gmsql\_sql\_get\_select\_result |
| update/update\_accum/insert/delete/exec/call | gmsql\_display\_output\_msg, gmsql\_sql\_get\_other\_dml\_result |
| DDL/DCL/COMMIT/ROLLBACK | gmsql\_sql\_get\_dcl\_or\_ddl\_result |

* gmsql\_sql\_get\_select\_result:重点关注

根据g\_gmsql\_local\_config.is\_display\_vertical配置选择不同的执行函数：

gmsql\_sql\_select\_data\_vertical

gmsql\_sql\_select\_data\_horizontal

以gmsql\_sql\_select\_data\_horizontal为例：

调用gmsql\_get\_column\_info获取列信息。

获取col\_name\_len，并且获取设置的列宽。这里我们为了简化，可以使用固定列宽。

# knl层源码阅读

## 概述

kernel层是最关键的一层，它包括以下几个部分：数据、索引、事务、持久化、数据字典等。

heap:核心代码，包括页的组织等。

latch:锁

lob:跨页存储数据类型

page:被heap管理

recovery:异常关闭

restore->replay->redo

row

segment:数据库对象，含索引等

session:会话管理

srep:双或复制

systask:定时任务

sysview:视图

rep:主备

lredo:逻辑redo

先从page/heap/latch看起。

## page

参考内核设计文档V0.3，以及《ONIP DataGrid V300R002C10 设计规格（Storage Mgr）.docx》这个文档。先仔细看那两个文档。再结合源代码去调试理解。这两个文档包含很重要的信息，一定要仔细阅读。

另外还需要参看一个 内存管理.ppt 的文件描述了内存管理的方案。

视图表v$Table，可以查看系统相关的统计信息。在systables表中可以看到。

在Segment头部保存该segment中所有page的空闲和使用列表，听起来，类似MemoryPool和MemoryContext那种关系。space可以包含多个device。device包含extent，每个extent包含4个page。segment是逻辑结构，包含page，可以跨越device。extent基本不体现？

可以得知，一个page由头、尾和数据组成，就三部分。头中包含了该页的控制信息。在头中有一个page\_id，其高10位表示device id而低22位表示页id。page id全局唯一，故page id在device内部唯一。而page id的低22位就是其在device中的~~偏移量~~（不是偏移量，而是索引。page的大小在建库时就固定了不可修改，只需要存储索引就可以了）。