静态代码分析

静态代码分析的组成部分

静态代码分析是一种方法用来测试代码和检测软件缺陷的方法，包括以下几个部分：

代码安全缺陷

代码发布

代码错误

代码标准违背

必须要注意到静态代码分析不能定义是否代码满足产品的要求。同时，这也不能确定是否代码可以运行。它只定义错误、易损性和源码中的缺点，保证代码质量。

静态代码分析是如何进行的

静态代码分析可以手动进行或者使用自动化工具做辅助。使用自动化工具更方便。

不需要人类，一个自动的静态代码分析工具可以在短时间内生成报告，在智能算法的帮助下，工具有效追踪报错，帮助开发者更快的修复。

pgsql 源码分析工具_SourceInsight

工具简介

工具网址：

https://www.sourceinsight.com/documentation/

实现的功能

01 - SourceInsight建立工程添加文件

菜单栏---》Project---》New Project---》New project name，命名并设置工程保存位置---》New project Settings，选择源代码目录---》Add and Remove Project Files，添加所有Add All---》Add to Project，勾选所有选项---》根据需要增删文件，点击Close。

02 - SourceInsight同步文件

同步文件可以自动找到源代码之间的依赖关系，例如：自动找到调用某个函数或变量的位置。

- 菜单栏---》Project---》Synchronize Files---》直接点击OK；
- 快捷键“Alt+Shift+S”；

03 - SourceInsigth的查找功能

- 菜单栏---》Search---》根据需要启动“Search Files”、“Search Project”、“Lookup References”等搜索框；
- 在工具栏点击蓝色大“R”图标，启动“Lookup References”搜索框；
- 快捷键“Ctrl+/”启动“Lookup References”搜索框；

04 - SourceInsigth的查看功能

将光标停留在关键字位置，将自动显示关键字的定义；双击显示内容，将跳转到该文件，从而可以继续查找。

05 - SourceInsight添加新类型文件

菜单栏---》Options---》Document Options---》Document Type---》选择所属具体类型---》右侧File filter，添加文件后缀---》点击Add Type。

06 - SourceInsight设置字体大小

菜单栏---》Options---》Document Options---》Screen Fonts，根据需要设置。

07 - SourceInsight设置快捷键

以设置全选快捷键为例：菜单栏---》Options---》Key Assignments，查询关键词找到select all---》Assign New Key，根据提示更改为“Ctrl+A”。

08 - SourceInsight设置背景色

菜单栏---》Options---》Preference---》Color---》Windows Background，设置背景色，例如护眼色“85,90,205” 。

09 - SourceInsight显示行号

打开代码文件---》菜单栏---》View---》Line Numbers。

10 - SourceInsight项目报告

获取当前项目的文件个数、代码行数等：菜单栏---》Project---》Project Report。

11 - SourceInsight重建项目

新同步代码依赖关系：菜单栏---》Project---》Rebuild Project。

12 - SourceInsight查看函数关系调用图

菜单栏---》View---》Panels---》勾选“Relation Window”， 会出现一个显示函数调用关系的窗口。
打开代码文件，在左侧的文件内容窗口中选择指定部分，即可看到函数调用关系。

13 - SourceInsight添加其他语言的语法高亮

通过使用语言插件SourceInsight可以添加其他语言的支持。
例如：Python的CLF插件（https://www.sourceinsight.com/pub/languages/Python.CLF）
详情请查看官网信息：https://www.sourceinsight.com/download/custom-languages/

14 - SourceInsight中文显示乱码

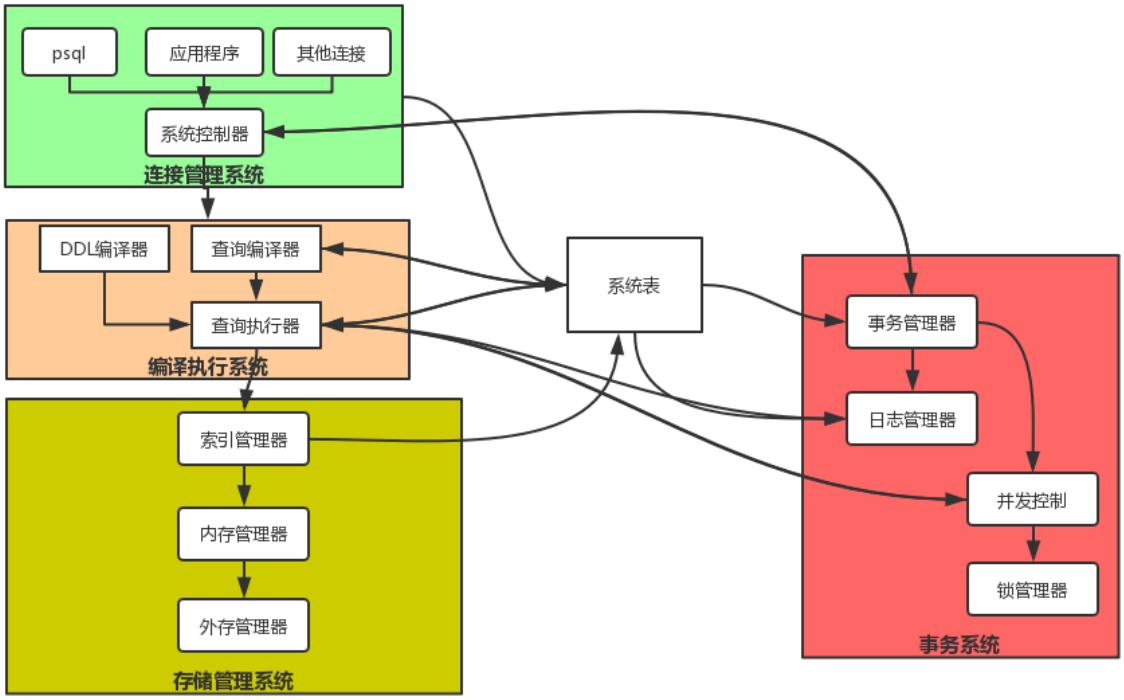
如果是SourceInsight3中文显示乱码，先关闭源代码文件，然后利用文本编辑器转换源代码文件编码为ANSI，再用SourceInsight3打开即可。
如果是SourceInsight4中文显示乱码：

- 针对单个文件中文乱码，菜单栏---》Reload As Encoding...---》选择对应的“Chinese Simplified”。
- 所有文件中文乱码，关闭所有打开的文件，菜单栏---》Options---》Preference---》Files---》Default encoding---》选择对应的“Chinese Simplified”---》再用SourceInsight4打开即可。

15 - SourceInsight恢复默认界面设置

方法一：菜单栏---》View，根据需要选择功能窗口；
方法二：关闭SourceInsight并删除Settings文件夹中的配置文件，然后重新打开SI即可。
查看Settings目录地址：菜单栏---》Options---》Preference---》Folder---》Settings Folder一栏。

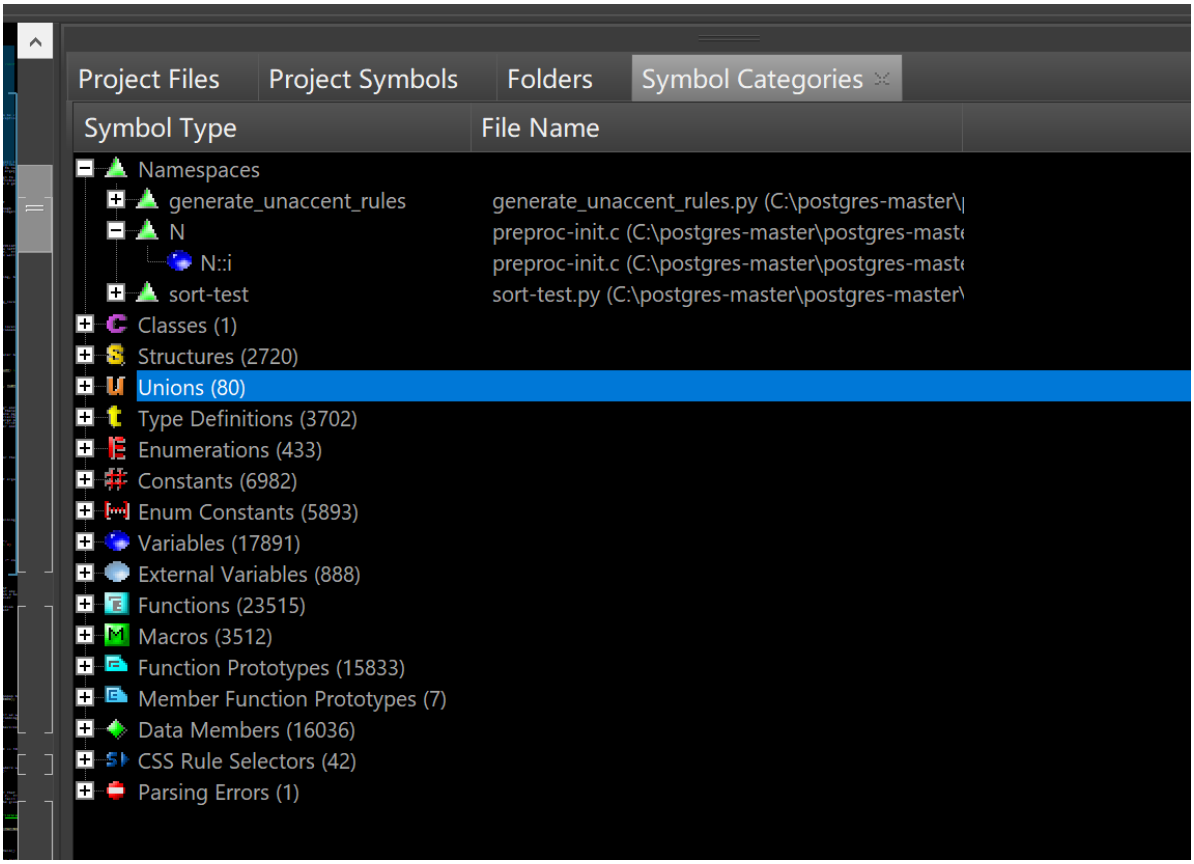
根据架构图分析pgsql 源码



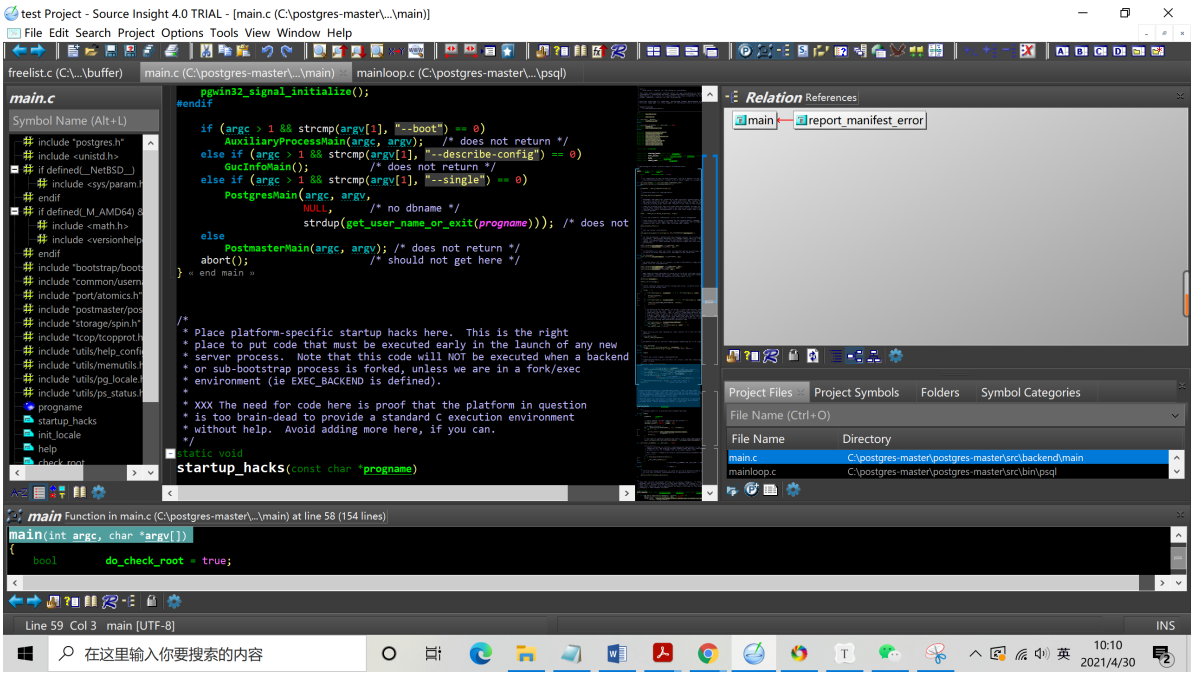
查看了下资料，Citus以插件的方式扩展到postgresql中。client,worker,buffer 的分工是典型的分布式架构。

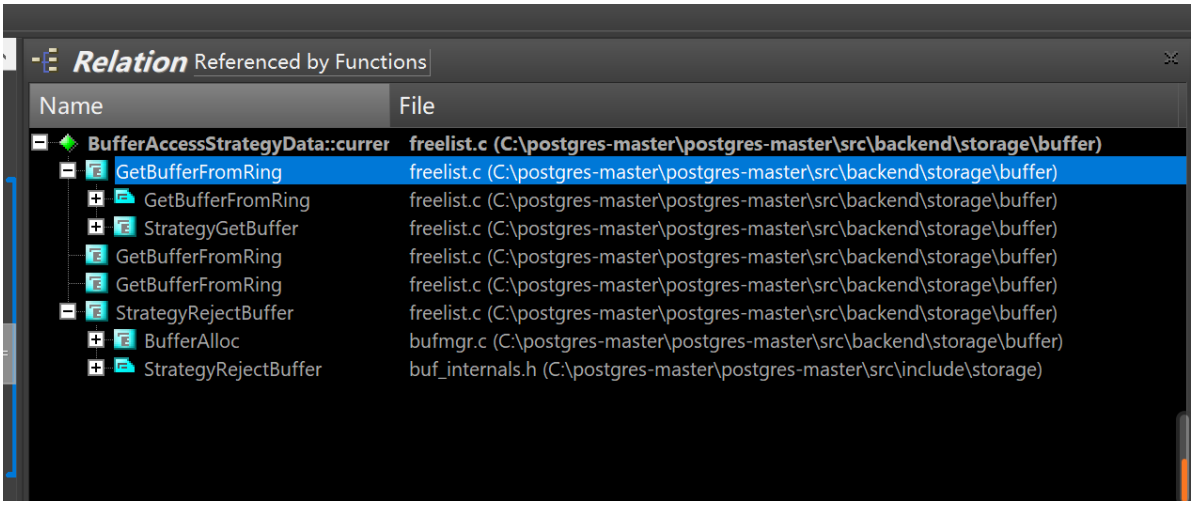
Postmaster:它主要负责在客户端第一次发送请求给服务器的时候建立一个服务器断进程。也就是上图中的Listener。postgre在完成建立服务器端进程之后，就不再对客户端和服务器端的通信进行干涉。
Listener：也就是每个客户端对应的服务器端进程，它的主要作用是和客户端进行通信，获取客户端的sql语句，并把查询结果返回给客户端。
Optimizer：查询优化器，主要功能是分析客户端提交的sql语句，给出所有的执行路径，并从中找出一个最优的方案，最后把这个执行方案交给执行器。
Buffer Manager:缓冲管理器，主要功能是对共享缓冲区和本地缓存区进行管理。

通过sourceInsight 查看pgsql 源码中的一些变量，函数和宏的定义



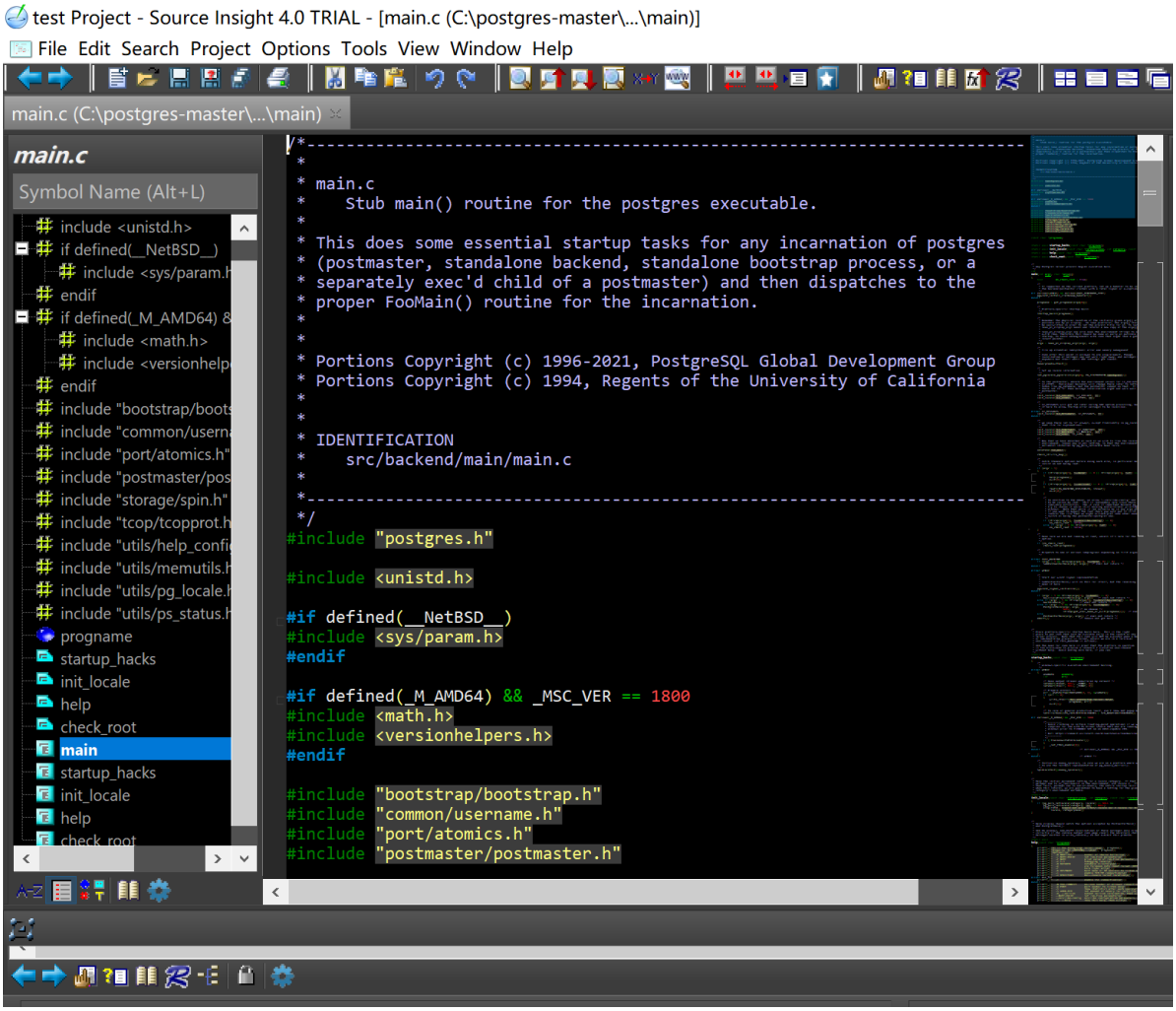
图形化功能查看函数之间的引用关系





main.c 静态分析

找到main.c 函数，开始分析：



修改一下代码，睡它30s。或者执行postgres可执行文件，set args 也OK。

```
56 /*
57  * Any Postgres server process begins execution here.
58  */
59 int
60 main(int argc, char *argv[])
61 {
62         bool            do_check_root = true;
63 sleep(30);
64         progname = get_progname(argv[0]);
```

启动数据库:

```
/usr/local/psql-9.5.3/bin/pg_ctl -D db2/ -l logfile  start -m fast
```

查看后台进程PID:

```
[postgres@localhost ~]$ ps -ef |grep postgres
root      57843  57805  0 10:58 pts/1    00:00:00 su - postgres
postgres  57844  57843  0 10:58 pts/1    00:00:00 -bash
postgres  57977      1  0 11:01 pts/1    00:00:00 /usr/local/psql-
9.5.3/bin/postgres -D db2
postgres  57981  57844  0 11:02 pts/1    00:00:00 ps -ef
postgres  57982  57844  0 11:02 pts/1    00:00:00 grep --color=auto postgres
```

进入调试模式,需要等30s:

```
cgdb -p 57977

(gdb) b main.c:64

Breakpoint 1 at 0x676229: file main.c, line 64.

(gdb) c

Continuing.

Breakpoint 1, main (argc=3, argv=0x7ffceddcbe88) at main.c:64

(gdb)
```

首先pg进入main函数，最先是获取到progname:

```
(gdb) p argv[0]

$1 = 0x7ffceddcd6d6 "/usr/local/psql-9.5.3/bin/postgres"
```

根据main函数的第一个参数进行字符串拆分计算出progname

```
(gdb) p progname
$15 = 0x14f9010 "postgres"
```

初始化内存(MemoryContextInit)是数据库启动的时候初始化的第一块内存，

```
typedef struct MemoryContextData
{
    NodeTag     type;           /* identifies exact kind of context */
    /* these two fields are placed here to minimize alignment wastage: */
    bool        isReset;        /* T = no space alloced since last reset */
    bool        allowInCritSection;     /* allow palloc in critical section */
    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) */
    MemoryContextCallback *reset_cbs;   /* list of reset/delete callbacks */
} MemoryContextData;
```

MemoryContextData是如何被初始化的:

```
 MemSet(node, 0, size);
    node->type = tag;
    node->methods = methods;
    node->parent = NULL;        /* for the moment */
    node->firstchild = NULL;
    node->nextchild = NULL;
    node->isReset = true;
    node->name = ((char *) node) + size;
    strcpy(node->name, name);

(gdb) p *node
$31 = {type = T_AllocSetContext, isReset = 1 '\001', allowInCritSection = 0
'\000', methods = 0xd1bbe0 <AllocSetMethods>, parent = 0x0, firstchild = 0x0,
nextchild = 0x0, name = 0x
14f9c70 "TopMemoryContext", reset_cbs = 0x0}
```

函数返回的是MemoryContext转成AllocSet.就是下面的结构体，MemoryContextData成了它的header。

这就是后面的NODE那个大enum，直接小转大。

```
typedef struct AllocSetContext
{
    MemoryContextData header;   /* Standard memory-context fields */
    /* Info about storage allocated in this context: */
    AllocBlock  blocks;         /* head of list of blocks in this set */
    AllocChunk  freelist[ALLOCSET_NUM_FREELISTS];       /* free chunk lists */
    /* Allocation parameters for this context: */
    Size        initBlockSize;  /* initial block size */
```

```
    Size        maxBlockSize;   /* maximum block size */
    Size        nextBlockSize;  /* next block size to allocate */
    Size        allocChunkLimit;    /* effective chunk size limit */
    AllocBlock  keeper;         /* if not NULL, keep this block over resets */
} AllocSetContext;

typedef AllocSetContext *AllocSet;
```

set内容,里面包含了数据库初始化的数据块大小，最大块，下一个块以及chunk的limit.

```
TopMemoryContext = AllocSetContextCreate((MemoryContext) NULL,
                                         "TopMemoryContext",
                                         0,
                                         8 * 1024,
                                         8 * 1024);
```

在初始化"TopMemoryContext"的时候，默认以及设定了最小块为0，初始化为8*1024，最大为*8*1024

```
(gdb) p    *set
$45 = {header = {type = T_AllocSetContext, isReset = 1 '\001', allowInCritSection
= 0 '\000', methods = 0xd1bbe0 <AllocSetMethods>, parent = 0x0, firstchild = 0x0,
nextchild = 0x0,
 name = 0x14f9c70 "TopMemoryContext", reset_cbs = 0x0}, blocks = 0x0, freelist =
{0x0 <repeats 11 times>}, initBlockSize = 8192, maxBlockSize = 8192,
nextBlockSize = 8192, allocChunkLimit = 1024, keeper = 0x0}
```

可以到初始化的数据块为8K，最大8K，allocChunkLimit为1024.

这样就把这个"TopMemoryContext"初始化完成了,然后把该内存上下文赋值给CurrentMemoryContext。

```
(gdb) p CurrentMemoryContext
$58 = (MemoryContext) 0x14f9bb0
(gdb) p    TopMemoryContext
$59 = (MemoryContext) 0x14f9bb0
(gdb)
```

现在初始化"TopMemoryContext"的第一个孩子，"ErrorContext"。

处理方式跟上面的区别，就是parent是"TopMemoryContext"，并且内存上下文是通过MemoryContext->methods->AllocSetAlloc这个函数指针来分配内存的

```
(gdb) p    *ErrorContext
$65 = {type = T_AllocSetContext, isReset = 1 '\001', allowInCritSection = 0
'\000', methods = 0xd1bbe0 <AllocSetMethods>, parent = 0x14f9bb0, firstchild =
0x0, nextchild = 0x0, nam
e = 0x14f9d80 "ErrorContext", reset_cbs = 0x0}
(gdb) p    *ErrorContext->parent
$66 = {type = T_AllocSetContext, isReset = 0 '\000', allowInCritSection = 0
'\000', methods = 0xd1bbe0 <AllocSetMethods>, parent = 0x0, firstchild =
0x14f9cc0, nextchild = 0x0, nam
e = 0x14f9c70 "TopMemoryContext", reset_cbs = 0x0}
(gdb) p *ErrorContext->parent->firstchild
$67 = {type = T_AllocSetContext, isReset = 1 '\001', allowInCritSection = 0
'\000', methods = 0xd1bbe0 <AllocSetMethods>, parent = 0x14f9bb0, firstchild =
0x0, nextchild = 0x0, nam
e = 0x14f9d80 "ErrorContext", reset_cbs = 0x0}
(gdb)
```

这样就把ErrorContext初始化完成了。PG中所有的内存上下文都挂载"TopMemoryContext"下面。

后面就是大量的环境变量设置了，以及root校验。最后调用函数"PostmasterMain(argc,argv)"。

动态代码分析

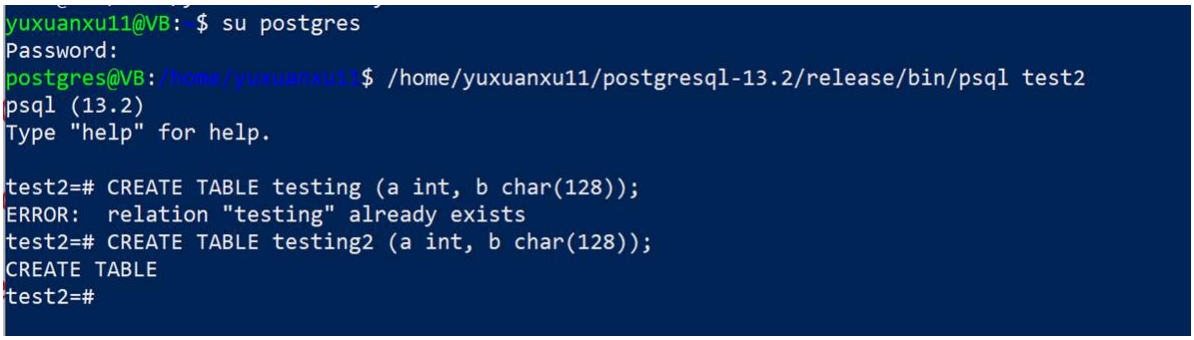
主要是gdb 追踪调试，以一次作业为例，（研究buffer manager)

(1) Tracing the Buffer Manager

As you have learned that buffer manager is an important component in PostgreSQL as it is responsible for reading and writing data to and from disk. Please use GDB to trace into buffer manager to observe the block data.

(2.1) On a psql client, create a 'testing' table with the following commands:

CREATE TABLE testing (a int, b char(128));



(2.2) Now attach your GDB to the backend process that handles your psql client requests.

(2.3) Set some break points on GDB, you can set breakpoints at some known functions, for example:

- ReadBufferExtended
- BufferAlloc

```
(gdb) b  ReadBufferExtended
Breakpoint 1 at 0x3a1400: file bufmgr.c, line 655.
(gdb) b BufferAlloc
Breakpoint 2 at 0x3a0ba0: file bufmgr.c, line 1025.
(gdb)
```

(2.4) Insert some data into the 'testing' table. For example:

INSERT INTO testing VAUES (1, 'this is the super secret of my life!');

```
test2=# INSERT INTO  testing2 VALUES(1,' this is the super secret of my life!');
INSERT 0 1
test2=#
```

Notes: do not copy this command from this document because the single quote (') on a word document is not recognized by PostgreSQL on Linux. You want to manually type the command out.

(2.5) Follow along in GDB (with 'n' and 's' commands) and observe where the program is going.

```
(gdb) n
660             RelationOpenSmgr(reln);
(gdb) s
667             if (RELATION_IS_OTHER_TEMP(reln))
(gdb)
```

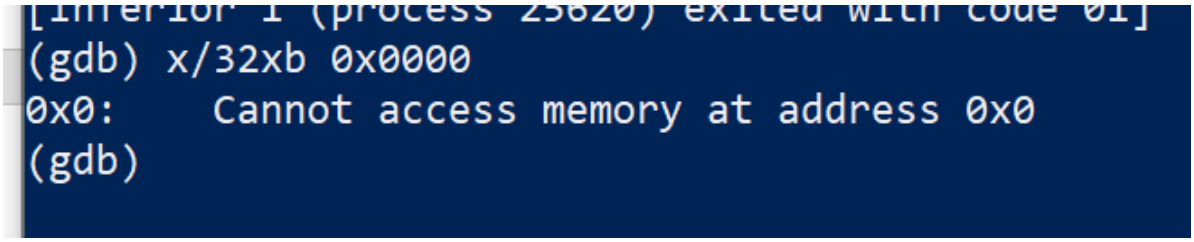
(2.6) Can you find the place where PostgreSQL adds that row value onto a page? (**Hint: It is somewhere in hio.c**)

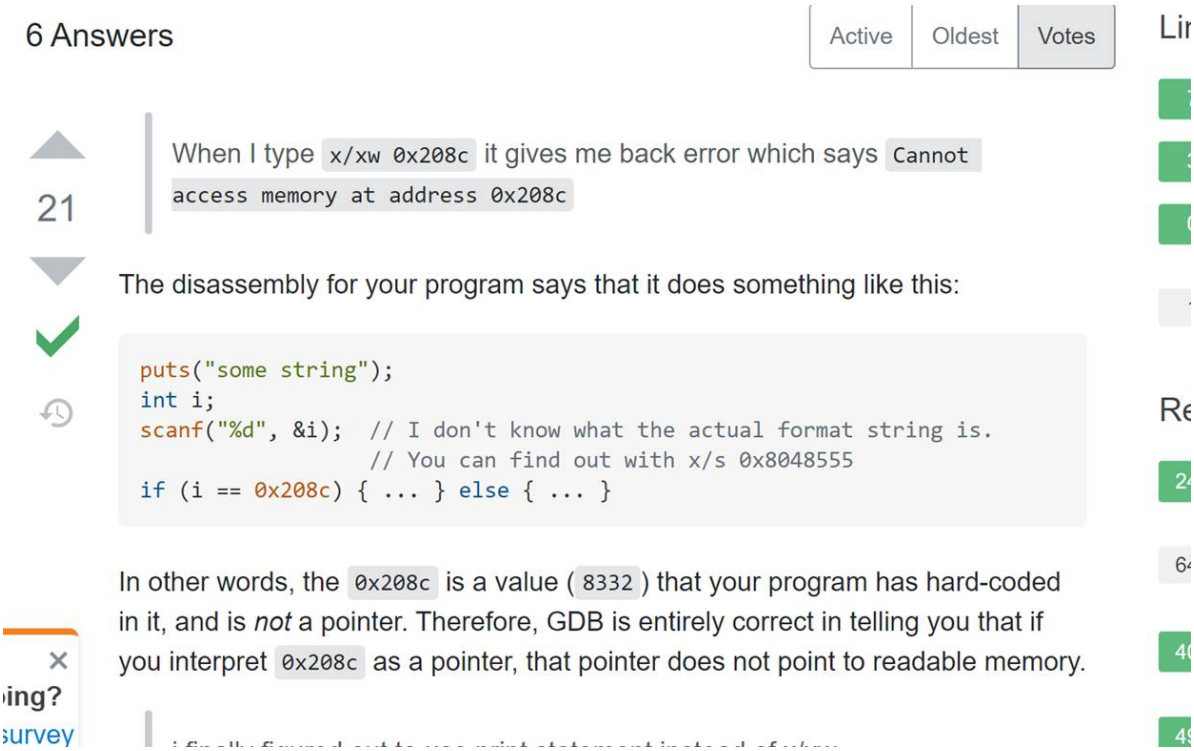
RelationPutHeapTuple()

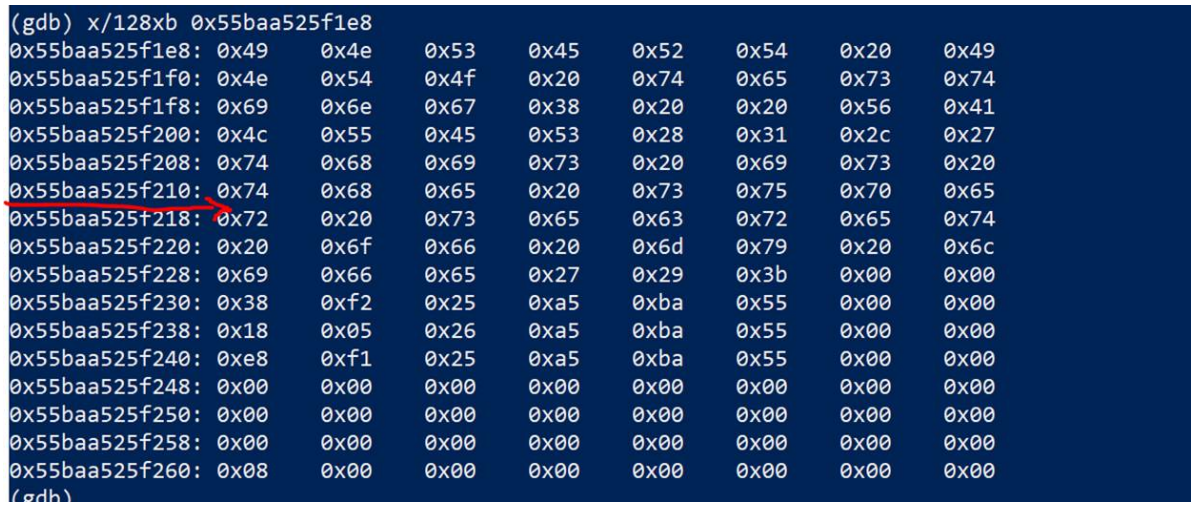
```
47       */
48      Assert(!token || HeapTupleHeaderIsSpeculative(tuple->t_data));
49
50      /*
51       * Do not allow tuples with invalid combinations of hint bits to be placed
52       * on a page.  This combination is detected as corruption by the
53       * contrib/amcheck logic, so if you disable this assertion, make
54       * corresponding changes there.
55       */
56      Assert(!((tuple->t_data->t_infomask & HEAP_XMAX_COMMITTED) &&
57               (tuple->t_data->t_infomask & HEAP_XMAX_IS_MULTI)));
58
59      /* Add the tuple to the page */
60      pageHeader = BufferGetPage(buffer);
61
62      offnum = PageAddItem(pageHeader, (Item) tuple->t_data,
63                           tuple->t_len, InvalidOffsetNumber, false, true);
64
65      if (offnum == InvalidOffsetNumber)
66          elog(PANIC, "failed to add tuple to page");
67
68      /* Update tuple->t_self to the actual position where it was stored */
69      ItemPointerSet(&(tuple->t_self), BufferGetBlockNumber(buffer), offnum);
70
71      /*
72       * Insert the correct position into CTID of the stored tuple, too (unless
73       * this is a speculative insertion, in which case the token is held in
74       * CTID field instead)
75       */
76      if (!token)
77      {
78          ItemId      itemId = PageGetItemId(pageHeader, offnum);
79          HeapTupleHeader item = (HeapTupleHeader) PageGetItem(pageHeader, itemId);
80
81          item->t_ctid = tuple->t_self;
82      }
83  }
```

(2.7) Once you have found the place where the program adds your row onto a page. Can you use the examine ('x') command to dump the contents of this page (8192 bytes in size)?

The stackflow first tells me it maybe a pointer if I see the error below







X is just gdb instruct

(2.8) Does it contain the string values you have just inserted? (ie. this is the super secret of my life!')? (Hint: You may need to consult an ASCII table to translate a character into a hexadecimal value. For example, the letter 'A' has hexadecimal value of 0x41. So what are the hex values for this string?)

```
(gdb) x/128xb 0x55baa525f1e8
0x55baa525f1e8: 0x49    0x4e    0x53    0x45    0x52    0x54    0x20    0x49
0x55baa525f1f0: 0x4e    0x54    0x4f    0x20    0x74    0x65    0x73    0x74
0x55baa525f1f8: 0x69    0x6e    0x67    0x38    0x20    0x20    0x56    0x41
0x55baa525f200: 0x4c    0x55    0x45    0x53    0x28    0x31    0x2c    0x27
0x55baa525f208: 0x74    0x68    0x69    0x73    0x20    0x69    0x73    0x20
0x55baa525f210: 0x74    0x68    0x65    0x20    0x73    0x75    0x70    0x65
0x55baa525f218: 0x72    0x20    0x73    0x65    0x63    0x72    0x65    0x74
0x55baa525f220: 0x20    0x6f    0x66    0x20    0x6d    0x79    0x20    0x6c
0x55baa525f228: 0x69    0x66    0x65    0x27    0x29    0x3b    0x00    0x00
0x55baa525f230: 0x38    0xf2    0x25    0xa5    0xba    0x55    0x00    0x00
0x55baa525f238: 0x18    0x05    0x26    0xa5    0xba    0x55    0x00    0x00
0x55baa525f240: 0xe8    0xf1    0x25    0xa5    0xba    0x55    0x00    0x00
0x55baa525f248: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55baa525f250: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55baa525f258: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55baa525f260: 0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

Based on the bt

#27 0x000055baa4574144 in exec_simple_query (query_string=0x55baa525f1e8 "INSERT INTO testing8 VALUES(1,'this is the super secret of my life');") at postgres.c:1155

(2) Observing the Data

(3.1) Find out which file is used to store your data. Remember the OID that is used as file name? What is yours?

```
test2=# select pg_relation_filepath('testing6');
 pg_relation_filepath
----------------------
 base/16384/16454
(1 row)
```

(3.2) Use the hexdump tool to see the content inside the data file

hexdump -C $FILE_NAME

```
postgres@VB:~/yuxuanxu/base/16384$ hexdump -C 16454
00000000  00 00 00 00 f0 b2 62 01  00 00 00 00 1c 00 60 1f  |......b.......`.|
00000010  00 20 04 20 00 00 00 00  60 9f 40 01 00 00 00 00  |. . ....`.@.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001f60  05 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001f70  01 00 02 00 02 08 18 00  01 00 00 00 10 02 00 00  |................|
00001f80  74 68 69 73 20 69 73 20  74 68 65 20 73 75 70 65  |this is the supe|
00001f90  72 20 73 65 63 72 65 74  20 6f 66 20 6d 79 20 6c  |r secret of my l|
00001fa0  69 66 65 20 20 20 20 20  20 20 20 20 20 20 20 20  |ife             |
00001fb0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                |
*
00002000
postgres@VB:~/yuxuanxu/base/16384$
```

(3.3) what do you see after the dump? Do you see the ‘this is the super secret of my life!’ that we have inserted in (2)?

Yeah

(3.4) If you do not see the value, what do you need to do to make PostgreSQL write out that data to the disk?

PostgreSQL works with data using a buffer cache. If needed data is not in the buffer cache, PostgreSQL reads it from the disk and puts into the buffer cache . If there is not enough space in this cache, then the least requested page is pushed out, evicted. If this page turns out to be in a “dirty” condition at the time of an eviction, then it must be written to disk in this exact moment.