**MariaDB 源码调试**

# 编译mariadb 源码

[root@jg-72 source]# pwd

/data/source

[root@jg-72 source]# ls

mariadb-10.1.11.tar.gz

源码压缩包解压缩

tar -zxvf mariadb-10.1.11.tar.gz

进入到BUILD子目录，它已经提供了一些一键编译的脚本

cd mariadb-10.1.11/BUILD

选择执行 compile-amd64-debug-all 脚本，因为我们要编译X86\_64平台上带DEBUG调试信息的mysql server。

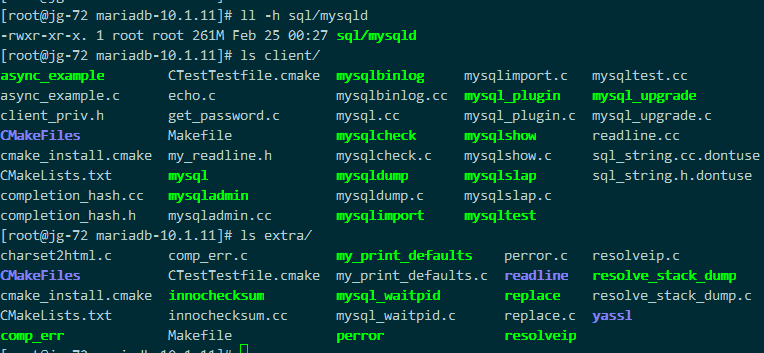
[root@jg-72 BUILD]# ./compile-amd64-debug-all

You must run this script from the MySQL top-level directory

cd mariadb-10.1.11

BUILD/compile-amd64-debug-all

静静等待编译结束，大概几分钟



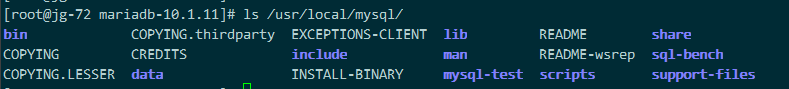
可以看到编译生成的mysqld 文件大小 261 M，而实际生产环境中使用的不带调试信息的mysqld文件，只有约89M。

client 和 extra目录下的可执行文件是编译生成的mysql自带的工具集，这里我们只关注mysqld。

# 使用编译的mysqld启动mysql实例

最简单的方法，在已安装好的mariadb的安装目录下，把mysqld用编译出来的版本替换掉即可。

但是我们不想破坏已经安装好的用于生产的MariaDB，所以这里我们在它的源码目录下构建属于它自己的mysql 基础目录（mysql basedir）



实际上就是参照安装的mysql的目录结构组织一下文件即可

把编译生成的可执行文件都copy到创建的bin目录下

(下面所有操作都是在MariaDB的源代码根目录下)

[root@jg-72 mariadb-10.1.11]# mkdir bin

cp sql/mysqld bin/

cp scripts/mysqld\_safe bin/

cp sql/{add\_errmsg,gen\_lex\_hash,gen\_lex\_token,mysql\_tzinfo\_to\_sql} bin/

cp client/{async\_example,mysqlbinlog,mysql,mysqladmin,mysqlcheck,mysqldump,mysql\_plugin,mysqlslap,mysql\_upgrade,mysqltest,mysqlshow,mysqlimport} bin/

cp extra/{comp\_err,my\_print\_defaults,mysql\_waitpid,perror,replace,resolveip,resolve\_stack\_dump} bin/

cp -r sql/share/ .

cp scripts/\*.sql share/

创建一个 my.cnf 文件， 其中basedir 设置为MariaDB源码的根目录

[mysqld]

user=mysql

port = 3310

**basedir = /data/source/mariadb-10.1.11**

socket = /data/source/3310/mysql.sock

datadir = /data/source/3310/data

log-error = /data/source/3310/mysqld.err

pid-file = /data/source/3310/mysqld.pid

character-set-server=utf8

… …

然后，初始化MySQL系统数据库，可以看到，它使用我们编译的mysqld来做初始化

[root@jg-72mariadb-10.1.11]# scripts/mysql\_install\_db --defaults-file=/data/source/3310/my.cnf --user=mysql

Installing MariaDB/MySQL system tables in '/data/source/3310/data' ...

2016-02-25 0:58:35 139708455159584 [Note] **/data/source/mariadb-10.1.11/bin/mysqld (mysqld 10.1.11-MariaDB-debug**) starting as process 21624 ...

OK

Filling help tables...

2016-02-25 0:59:00 140501225490208 [Note] /data/source/mariadb-10.1.11/bin/mysqld (mysqld 10.1.11-MariaDB-debug) starting as process 21664 ...

OK

Creating OpenGIS required SP-s...

2016-02-25 0:59:07 140226519934752 [Note] /data/source/mariadb-10.1.11/bin/mysqld (mysqld 10.1.11-MariaDB-debug) starting as process 21703 ...

OK

… …

启动实例：

可以看到是使用我们编译的mysqld启动的

[root@jg-72mariadb-10.1.11]# bin/mysqld\_safe --defaults-file=/data/source/3310/my.cnf --user=mysql &

[1] 21808

[root@jg-72 mariadb-10.1.11]# ps -ef|grep mysql |grep 3310

root 21808 19360 0 01:04 pts/5 00:00:00 /bin/sh bin/mysqld\_safe --defaults-file=/data/source/3310/my.cnf --user=mysql

mysql **22096** 21808 28 01:04 pts/5 00:00:03 **/data/source/mariadb-10.1.11/bin/mysqld** --defaults-file=/data/source/3310/my.cnf --basedir=/data/source/mariadb-10.1.11 --datadir=/data/source/3310/data --plugin-dir=/usr/local/mysql/lib/plugin --user=mysql --log-error=/data/source/3310/mysqld.err --pid-file=/data/source/3310/mysqld.pid --socket=/data/source/3310/mysql.sock --port=3310

# 使用gdb调试mysql进程

gdb 连接到要调试的进程的命令很简单，上面知道这个进程的id是22096

命令: gdb – 22096

Loaded symbols for /usr/lib64/libltdl.so.7

Reading symbols from /lib64/libfreebl3.so...(no debugging symbols found)...done.

Loaded symbols for /lib64/libfreebl3.so

Reading symbols from /lib64/libnss\_files.so.2...(no debugging symbols found)...done.

Loaded symbols for /lib64/libnss\_files.so.2

0x0000003893adf1b3 in poll () from /lib64/libc.so.6

Missing separate debuginfos, use: debuginfo-install bzip2-libs-1.0.5-7.el6\_0.x86\_64 glibc-2.12-1.149.el6.x86\_64 libaio-0.3.107-10.el6.x86\_64 libgcc-4.4.7-11.el6.x86\_64 libstdc++-4.4.7-11.el6.x86\_64 libtool-ltdl-2.2.6-15.5.el6.x86\_64 libxml2-2.7.6-14.el6\_5.2.x86\_64 nss-softokn-freebl-3.14.3-17.el6.x86\_64 snappy-1.1.0-1.el6.x86\_64 unixODBC-2.2.14-14.el6.x86\_64 xz-libs-4.999.9-0.5.beta.20091007git.el6.x86\_64 zlib-1.2.3-29.el6.x86\_64

(gdb)

最后会看到如上的输出，此时这个mysql进程已经被gdb挂起，用客户端连接是没有响应的。

设置函数断点，这样当mysql进程执行到这个函数的时候，就会被gdb捕获到并且停在函数的入口处。

(gdb) b dict\_index\_too\_big\_for\_tree

Breakpoint 1 at 0xdd291b: file /data/source/mariadb-10.1.11/storage/xtradb/dict/dict0dict.cc, line 2390.

(gdb)

输入c 命令，让程序正常运行

(gdb) c

Continuing.

此时在另一个终端使用mysql客户端进行连接

[root@jg-72 mariadb-10.1.11]# ./bin/mysql -uroot -h168.168.207.72 -P3310

Welcome to the MariaDB monitor. Commands end with ; or \g.

Your MariaDB connection id is 2

Server version: 10.1.11-MariaDB-debug Source distribution

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>

我们可以正常的执行一些SQL

MariaDB [(none)]> use test;

Database changed

MariaDB [test]> show variables like 'char%';

+--------------------------+----------------------------------------------+

| Variable\_name | Value |

+--------------------------+----------------------------------------------+

| character\_set\_client | utf8 |

| character\_set\_connection | utf8 |

| character\_set\_database | utf8 |

| character\_set\_filesystem | binary |

| character\_set\_results | utf8 |

| character\_set\_server | utf8 |

| character\_set\_system | utf8 |

| character\_sets\_dir | /data/source/mariadb-10.1.11/share/charsets/ |

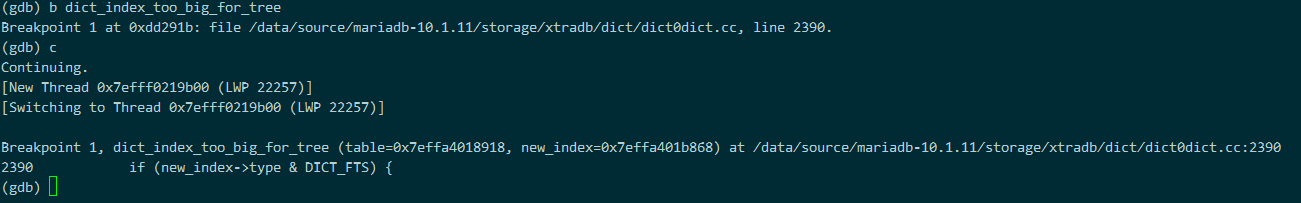
+--------------------------+----------------------------------------------+

8 rows in set (0.00 sec)

尝试创建一个表，发现hang住了，

MariaDB [test]> create table testtable ( c1 varchar(100), c2 varchar(100), c3 varchar(100));

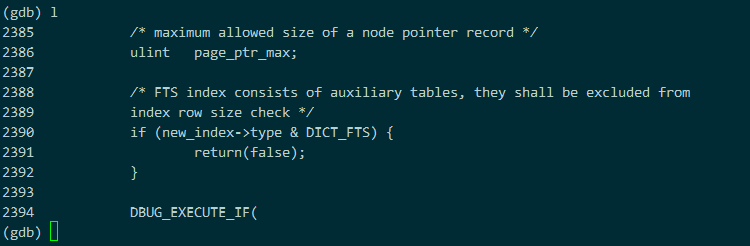
切换到gdb所在的终端，



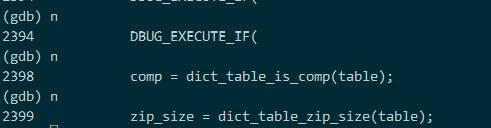
可以看到，gdb在设置断点的函数的入口处停了一下，等待我们单步调试

L 命令可以看接下来10行的代码，后面可以跟数字

l 100就是看100行代码， l -10 就是看上面10行的代码

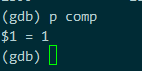


单步调试的命令是 n，也就是一行行执行代码（next）



想看某个变量的值，使用 p 命令（print）

在执行到某函数调用处，比如上图的2398行，也可以使用 s (step)命令，这个时候会进入被调用的函数dict\_table\_is\_comp()内部，继续单步执行

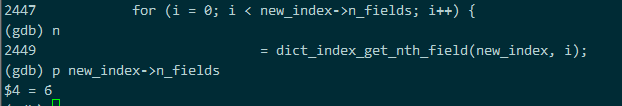


comp = 1这说明，当前创建的表的格式是 compact的。

![C:\Users\006975\AppData\Roaming\Tencent\Users\2837796568\QQ\WinTemp\RichOle\Q`O)(F)1TZ6}J$0](R6A2TN.png](data:image/png;base64,)

我们这个表有3个nullable的列，每条记录的长度限制，page\_rec\_max 是 8126，是通过

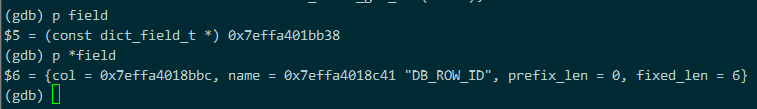
2425行的函数计算出来的。



接下来进入计算每一个字段占用空间的计算。这里我们看到，

这个表总共有 6 个 fields，而不是我们定义的三个，这是因为，mysql隐含的会添加三个字段

* Records in the clustered index contain fields for all user-defined columns. In addition, there is a 6-byte transaction ID field and a 7-byte roll pointer field.
* If no primary key was defined for a table, each clustered index record also contains a 6-byte row ID field.

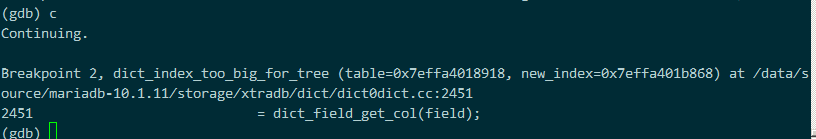


我们用 p 看第一个field的信息，确实是 ROW\_ID，长度确实是6 byte，和 文档互相印证。

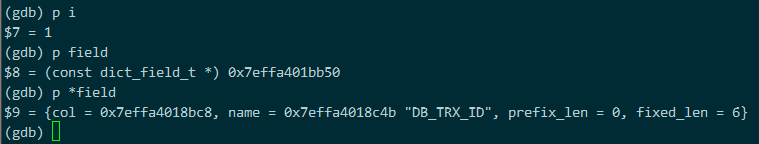
对于隐含列的处理，我们可能不关心，所以想直接跳过接下来的代码，那可以设置一个行断点，

C:\Users\006975\AppData\Roaming\Tencent\Users\2837796568\QQ\WinTemp\RichOle\(_3@J7Y6{V]_G~RQ4@Z{(WY.png

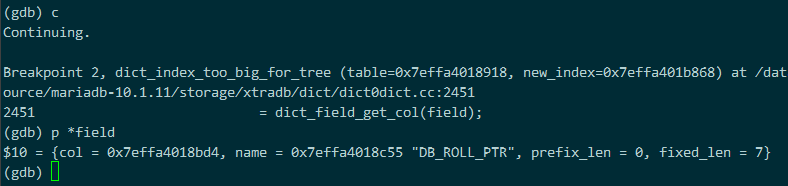
b 命令不加参数就是在当前行设置断点， b <line num> 就是在当前文件的指定行设置行断点， b <file name>:<line num>就是在指定文件的指定行设置行断点



接下来 continue，我们发现它运行到 2451行后再次停下，说明循环进入了第二次迭代(i已经变成1)



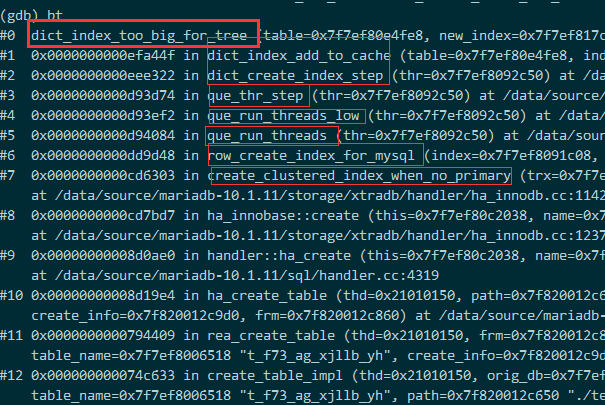
可以看到，第二个字段也是默认添加的 TRX\_ID 列，长度是6个字节，

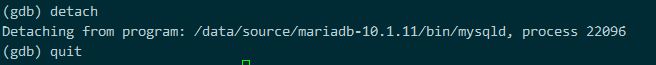


第三个字段是默认添加的ROLL\_PTR列，长度为7字节

可以预期，接下来就是我们建表的真实的字段信息。

如果想看当前程序的函数调用栈，可以使用 bt 或者where 命令





Detach命令使gdb释放对mysql进程的连接，quit命令退出gdb

# 如何从MySQL的一条报错信息定位到源代码

以下面这个问题举例：

297个字段 varchar(73)字段的表

create table t\_f73\_ag\_xjllb\_yh

(

jydm VARCHAR(73),

rq VARCHAR(73),

cb VARCHAR(73),

khdkjdkjjse VARCHAR(73),

xyhjkjzje VARCHAR(73),

… …

xjdjwdqmye\_yoy VARCHAR(73),

jxjdjwdqcye\_yoy VARCHAR(73),

xjjxjdjwjzje1\_yoy VARCHAR(73)

);

报如下错误

ERROR 1118 (42000): Row size too large (> 8126). Changing some columns to TEXT or BLOB or using ROW\_FORMAT=DYNAMIC or ROW\_FORMAT=COMPRESSED may help. In current row format, BLOB prefix of 768 bytes is stored inline.

在源码根目录搜索消息的部分内容

[root@jg-72 mariadb-10.1.11]# find \* |xargs grep 'Row size too large'

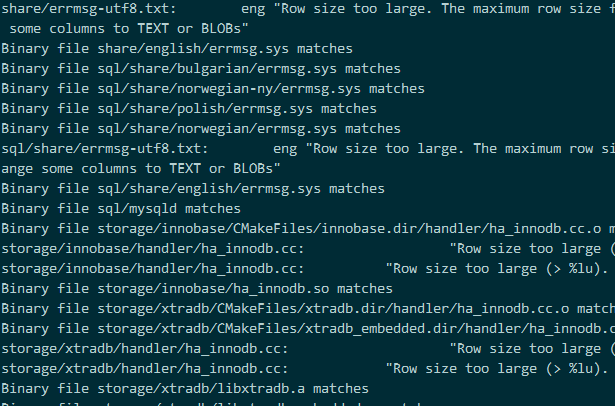
输出很多，但是我们应该重点关注源代码和文本文件，所以

share/errmsg-utf8.txt sql/share/errmsg-utf8.txt

storage/innobase/handler/ha\_innodb.cc

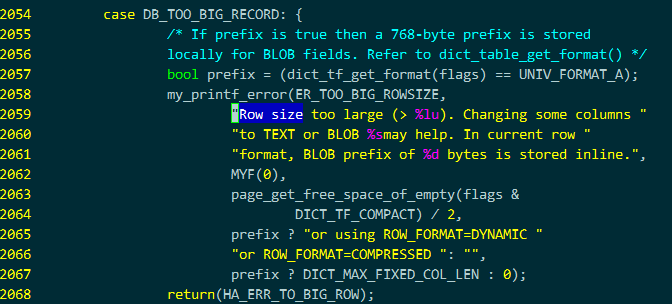
storage/xtradb/handler/ha\_innodb.cc

这几个文件需要重点关注



因为MariaDB的INNODB存储引擎实际上是xtraDB，所以先看

storage/xtradb/handler/ha\_innodb.cc



发现两个关键的宏定义 ER\_TOO\_BIG\_ROWSIZE 和DB\_TOO\_BIG\_RECORD

从errmsg-utf8.txt文件的内容ER\_TOO\_BIG\_ROWSIZE 42000 和错误消息头 1118 (42000)，可以推测出ER\_TOO\_BIG\_ROWSIZE 应该就是错误消息号 1118，但是errmsg-utf.txt中的1118错误的内容，不包含 (> xxx)，进一步发现，包含这个错误内容的只有上面的 ha\_innodb.cc的2058行， 函数是 convert\_error\_code\_to\_mysql()

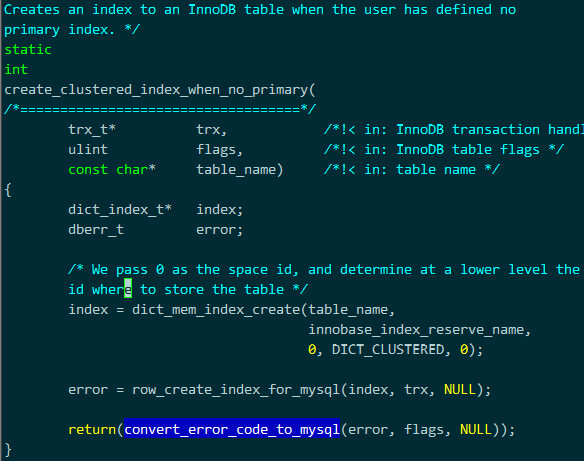
继续寻找这个函数调用的地方

find \* |xargs grep 'convert\_error\_code\_to\_mysql'

发现，全部都在ha\_innodb.cc 文件中（大概有几十处调用）

通过遍历这个文件调用这个函数的地方，可以看到很多地方，error号都是通过

row\_create\_index\_for\_mysql()函数的返回值得到，我们更是找到一个函数



这里，首先需要知道的一个概念

从文档中可以知道，InnoDB的表是一种索引组织表，也就是说，它的表实际上就是索引（clustered index），他的索引也就是表，**它的clustered index实际上包含了所有用户定义的字段。**

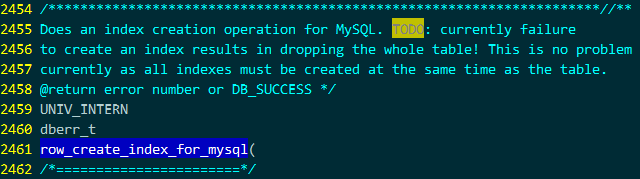
另外，可以在表的选定的字段上创建二级索引，如果表中定义了主键，那么二级索引将隐含的包括主键这一字段。

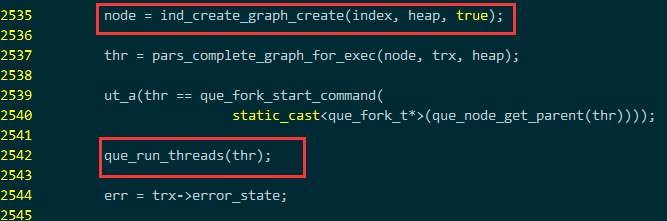
The data in each InnoDB table is divided into [pages](http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_page). The pages that make up each table are arranged in a tree data structure called a [B-tree index](http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_b_tree). Table data and [secondary indexes](http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_secondary_index) both use this type of structure. The B-tree index that represents an entire table is known as the [clustered index](http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_clustered_index), which is organized according to the [primary key](http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_primary_key) columns. **The nodes of the index data structure contain the values of all the columns in that row (for the clustered index)** or the index columns and the primary key columns (for secondary indexes).

所以，基本可以断定，这个函数create\_clustered\_index\_when\_no\_primary()实际上就是我们创建一个不包含主键的InnoDB表主要要调用的函数。而它的主要工作，在

row\_create\_index\_for\_mysql()中完成。

同样的方式，在storage/innobase/row/row0mysql.cc中找到 row\_create\_index\_for\_mysql() 的定义

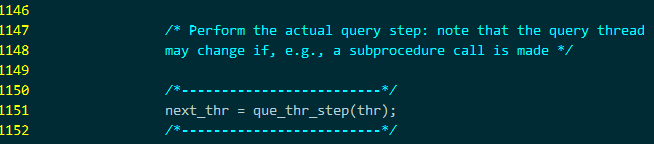




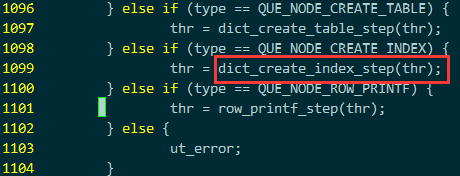
通过大概浏览代码知道这个函数会调用que\_run\_threads()去做实际的工作。

找到它的定义处 storage/xtradb/que/que0que.cc 1183，发现它实际调用的是

que\_run\_threads\_low(), 这个也定义在相同的文件里，进入这个函数



在同一个文件里，我们也找到que\_thr\_step()函数的定义，实际上可以推测出，它里面会根据不同的语句调用不同的入口函数



很幸运，它用了老土的 if else条件判断，而不是函数指针，所以可以知道调用了

dict\_create\_index\_step()

这里也可能是dict\_create\_table\_step()呀？ 一方面，我们是从create\_index一路跟下来的，

另一方面，不放心的话，搜一下 QUE\_NODE\_CREATE\_INDEX这个type，

发现是在storage/xtradb/dict/dict0crea.cc ind\_create\_graph\_create()函数中赋值的，而这个函数，被 row\_create\_index\_for\_mysql()在调用que\_run\_threads()之前调用了，所以，可以确定，接下来执行的函数是dict\_create\_index\_step()

在storage/innobase/dict/dict0crea.cc 中找到这个函数的定义

这个时候发现这个函数真TM长，分了好几个阶段，

它调用的函数有

dict\_build\_index\_def\_step()

dict\_build\_field\_def\_step()

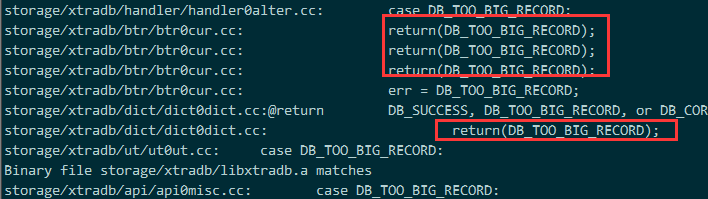
dict\_index\_add\_to\_cache()

dict\_index\_get\_if\_in\_cache\_low()

dict\_create\_index\_tree\_step()

**一个个函数跟进去相当于尝试好几条岔路，一般这种情况，每个函数都跟进去会掉进无限的调用陷阱里，显然不是个好办法。**

**于是回到最开始，想办法自底向上确定函数调用栈，之前是查找ER\_TOO\_BIG\_ROWSIZE从上往下找，现在查找 DB\_TOO\_BIG\_RECORD 这个关键字**

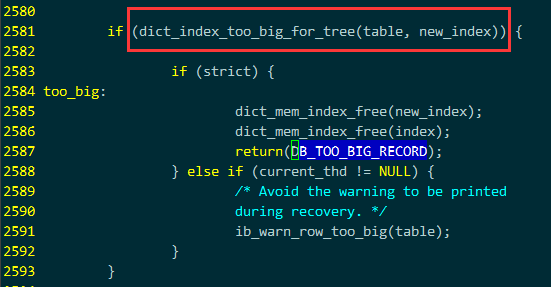


重点关注其中返回 DB\_TOO\_BIG\_RECORD的函数

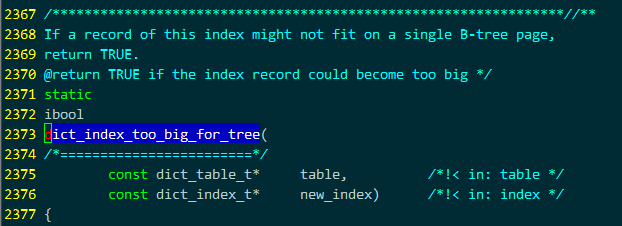
在storage/xtradb/btr/btr0cur.cc 文件中，几处返回DB\_TOO\_BIG\_RECORD的函数从名字看不是update就是insert，显然不是我们的create table

于是在storage/xtradb/dict/dict0dict.cc 中，发现返回DB\_TOO\_BIG\_RECORD的函数

**恰好是dict\_index\_add\_to\_cache(), 是被dict\_create\_index\_step()调用的函数之一。**



到此，基本可以确定，**最后通过函数dict\_index\_too\_big\_for\_tree()来检查创建一个innodb表是否会超出innodb表的一些长度限制。**



通过前面的gdb调试中的bt命令，也可以印证这一点，可以看到函数调用的堆栈

Create innodb table without primary key …

                                               ->create\_clustered\_index\_when\_no\_primary()

                                                        ->row\_create\_index\_for\_mysql()

                                                                 ->que\_run\_threads()

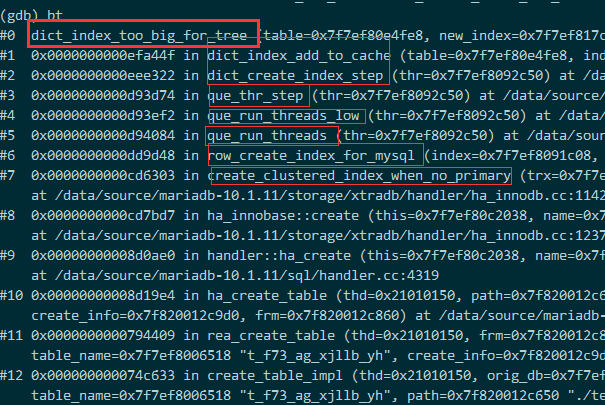
                                                                           ->que\_run\_threads\_low()

                                                                                    ->que\_thr\_step()

                                                                                             ->dict\_create\_index\_step()

                                                                                                       ->dict\_index\_add\_to\_cache()

                                                                                                                ->dict\_index\_too\_big\_for\_tree()



**Tip:**

当然，这个查找过程可以不通过命令行，也可以使用一些工具，比如sourceinsight把整个mariadb的源代码导入，然后在sourceinsight里执行类似的搜索过程。Source Insight的安装使用这里就不说了。