正文需要有：

1. 分析MiniOB的Create Table实现原理，要画出Create Table实现的过程，并进行相应的描述。

2. 设计Drop Table和Alter Table的实现方案，画出设计方案图。

3. Drop Table 、Alter Table的实现代码。

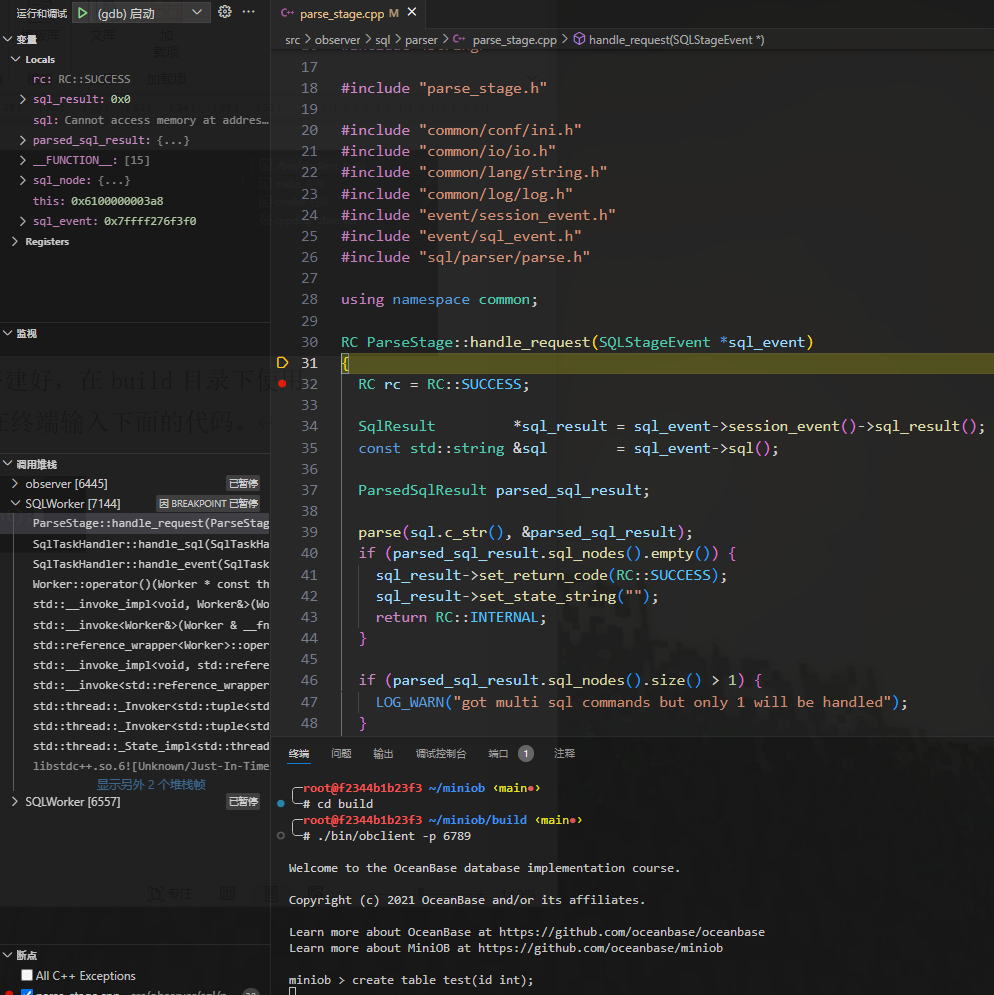
4. Drop Table 、Alter Table功能测试。

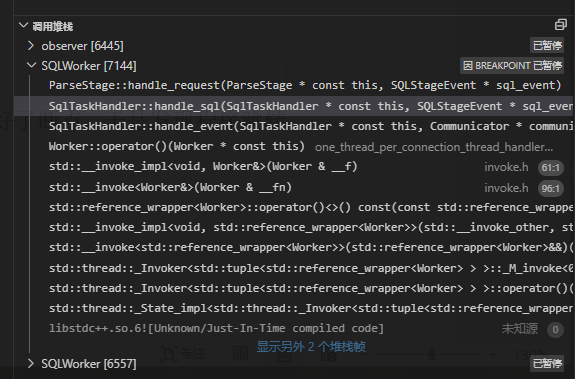
5. 总结

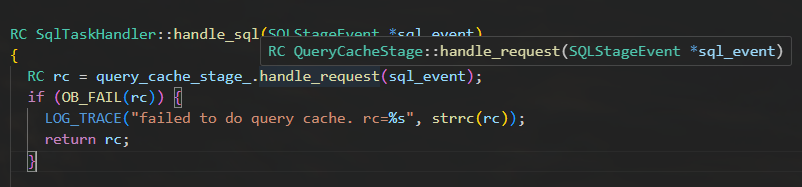
正文

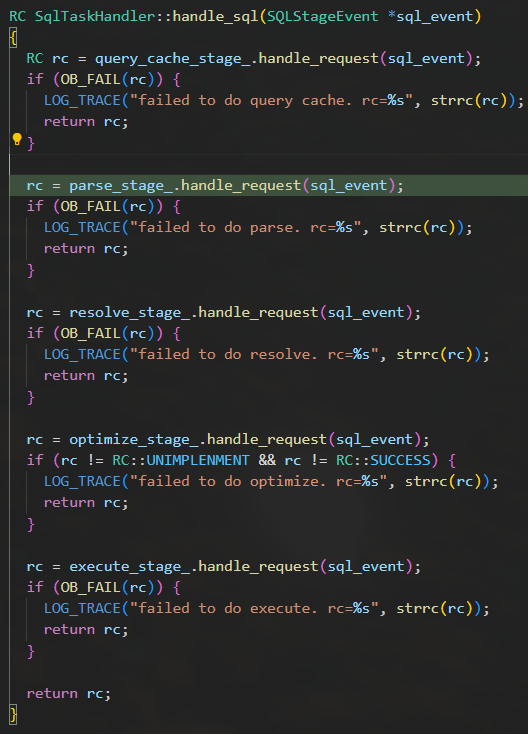
1. **MiniOB的Create Table实现原理及过程**

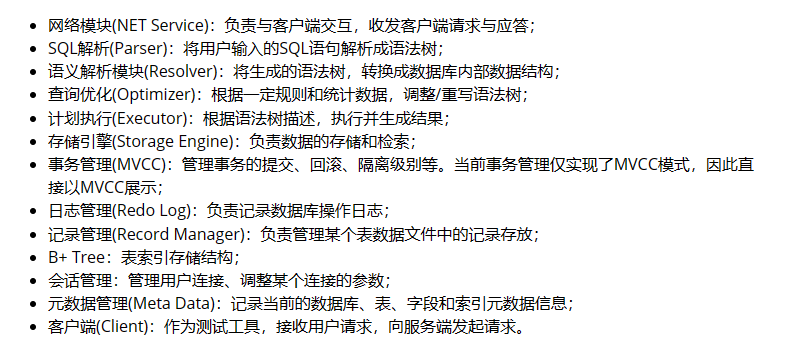
在正式开始调试前，我们先把基础的环境搭建好，在build目录下使用命令开启一个client并监听6789端口。接着在终端输入下面的代码。

结果如下。

因为我们提前在parse\_stage.cpp文件中打好了断点，于是发现程序跳转到了里面。

看到调用堆栈这里，根据名称推测create table的操作应该跟SqlTaskHandler有关，于是点击进去查看。

这里看到一个查询缓存的方法，但是点进去查看函数体基本为空。

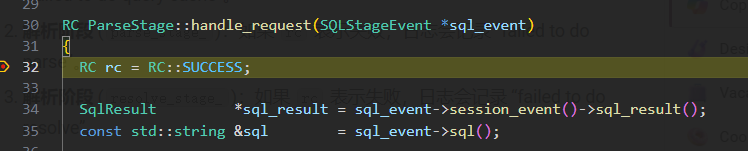
从上面的代码中可以看出，在执行create table时，数据库系统先要对输入的sql语句进行处理。同时，根据官网文档可知，从输入指令到执行分为网络模块、SQL解析、语义解析、查询优化、计划执行等等阶段，接下来按照阶段执行顺序依次解析。

网络模块不涉及实现原理，因此直接进入SQL解析模块。

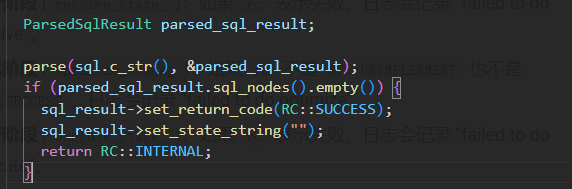


在每个阶段，都会调用名为handle\_request的函数，这个函数在解析SQL语句的过程中十分重要，应当着重分析。在SQL解析阶段的handle\_request中，包含纠错功能，如果有问题则会打印对应的错误内容并且直接返回。

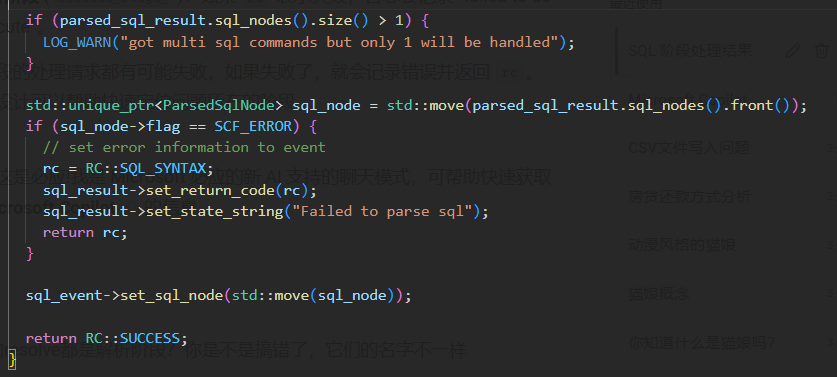
同时我们可以看见，在一整个流程中，sql\_event这个参数是每次执行都传入的，说明它很重要。

对源码进行查看后发现这个event类内部定义了很多变量以及函数，与整个sql语句的解析、处理等息息相关。

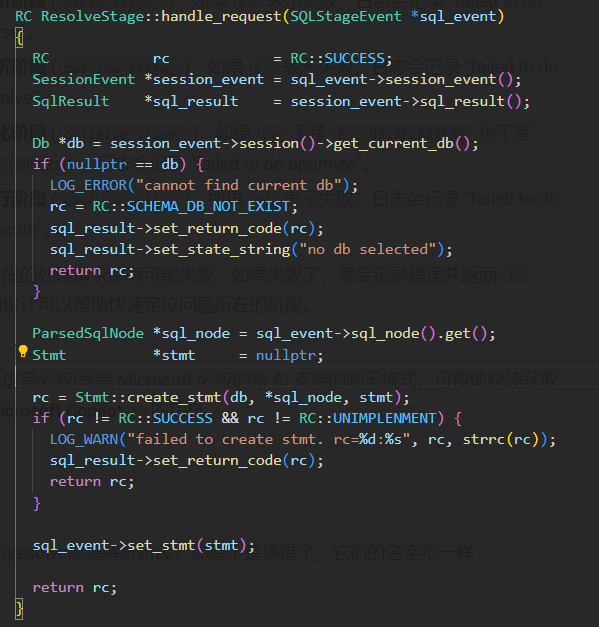
可以发现数据库在进入函数后，先把rc设置成了success状态，再从sql\_event变量中提取出sql\_result和sql。

然后将sql和sql\_result传给parse。这里的sql\_result即代表语法解析后的数据。

接下来程序把sql的结果写入到 parsed\_sql\_result 内，并做判断，如果是空的，就设置返回码为成功、设置状态码为空后直接返回。

之后便是把parsed\_sql\_result，即解析后的数据定义为一个sql\_node的智能指针，再将其存回sql\_event方便后续阶段使用。

Sql解析阶段的handle\_request分析完毕，紧接着是语义解析模块(resolver)的handle\_request。



在resolve阶段的handle\_request接受SQL解析阶段生成的sql\_event，并获取它的session\_event和sql\_result。

在这一阶段获取的session\_event，是沟通DB和我们目前所在进程的桥梁，因此在获取session\_event后，创建了一个DB对象，获取session\_event中存储的current\_db。

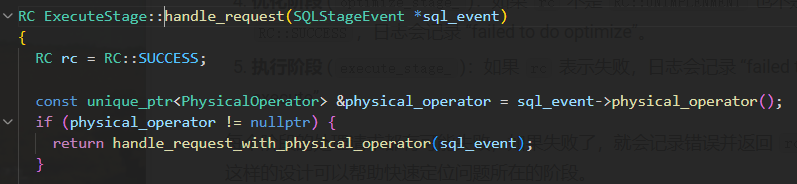
之后便是一些错误检查。

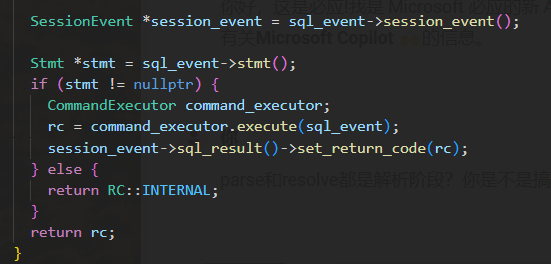
与SQL解析阶段不同的是，resolve阶段在handle\_request内进行了状态设置，根据刚才获取的db对象，以及之在sql解析阶段定义的sql\_node，对rc进行了状态设置，并将状态保存回sql\_event中。

语义解析模块(Resolver)阶段结束，接下来进入查询优化(Optimizer)阶段。

根据官网给出的定义以及实际的代码可见。查询优化(Optimizer)：根据一定规则和统计数据，调整/重写语法树；这一阶段是对刚才几个部分生成的语法树进行优化，因此与分析create\_table的实现原理关系不大，故不作分析。

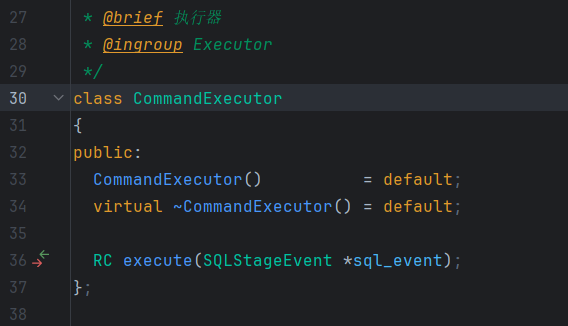


之后进入计划执行(Executor)阶段：根据语法树描述，执行并生成结果；

这一阶段是create\_table实现的重点阶段，如果我们要了解create\_table实现的底层原理，应当着重分析这一部分。

可以看到进入函数后，程序先执行了一个判断，判断物理操作数是否为空，如果不是的话，程序就使用带有物理操作数的函数去处理sql语句。即先判断上一阶段是否进行了物理优化，如果进行了物理优化就按照物理优化的方法继续执行，如果没有的话就按没有物理优化的方法继续执行。

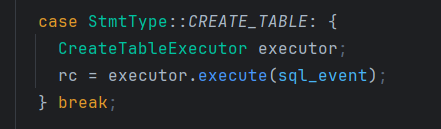
之后取出sql\_event中的session\_event和stmt，如果stmt非空，就创建一个在CommandExecutor类中的command\_executor。紧接着我们分析CommandExecutor这个类，在CommandExecutor这个类中定义了一个execute的函数，接收sql\_event作为参数，可见他就是这次create\_table的实际执行者。

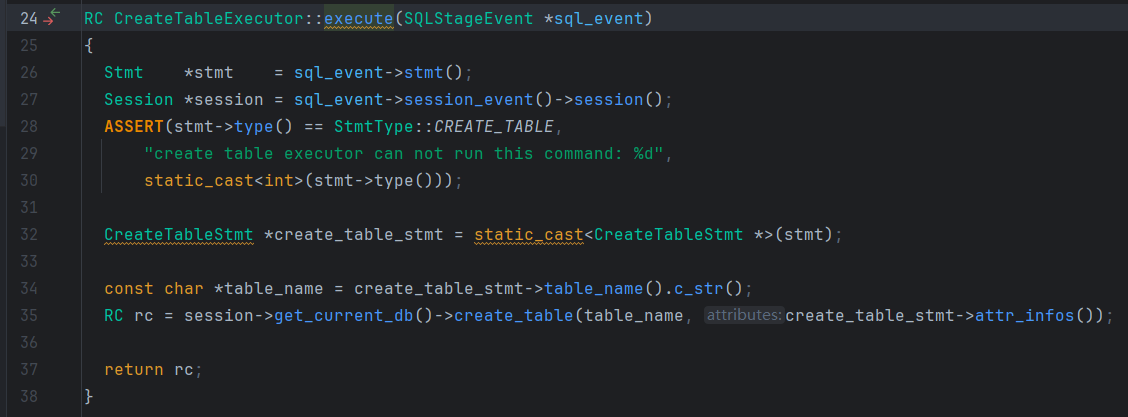


然后我们进入这个execute函数中，查看其具体执行效果。

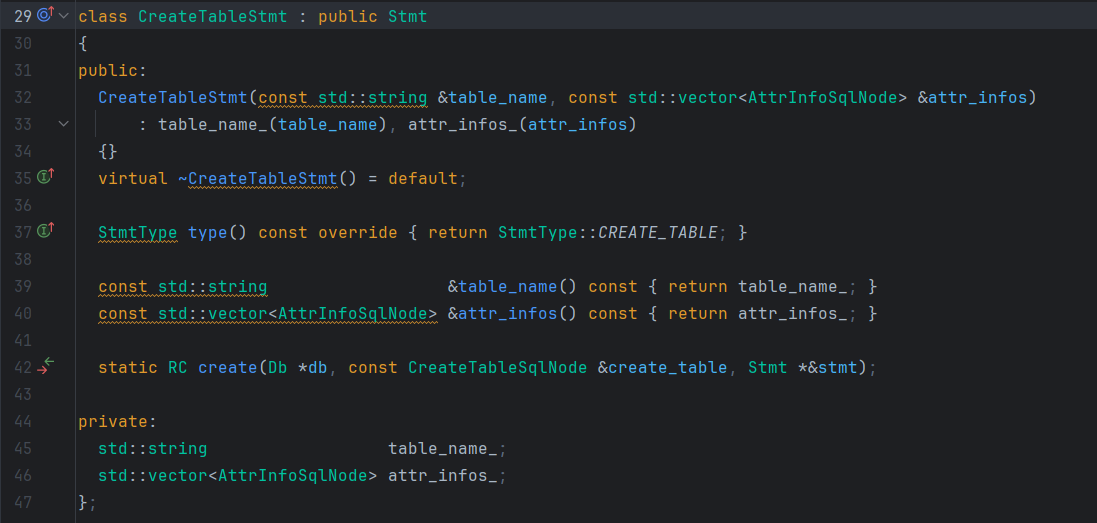


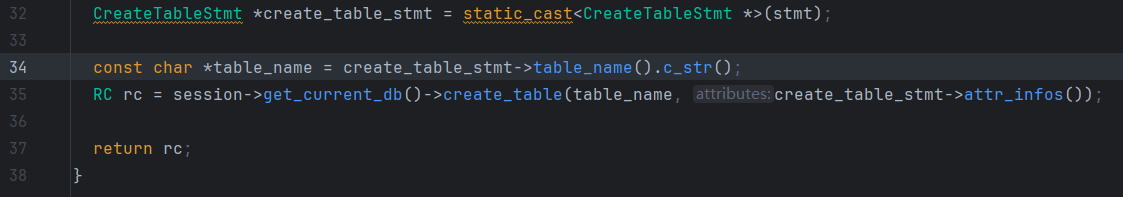
这个execute函数会根据sql\_event中保存的stmt来进行对应的操作，比如CREATE\_INDEX、CREATE\_TABLE、DESC\_TABLE等等。

这其中就有我们本次要分析的重点，create\_table。

当stmt为CREATE\_TABLE时，会创建一个CreateTableExecutor，之后接收sql\_event并对其使用CreateTableExecutor的execute方法，我们进一步跳转会发现，真正创建表的函数在这里。我们来到了create\_table的实际执行部分。

在实际执行create\_table时，依旧是先取出sql\_event中的stmt和session，这个session和我们上文提到的一样，是沟通db和我们现在进程的一个类，方便后续执行取得当前db等操作。之后判断stmt是否是CREATE\_TABLE这个类型，如果不是的话就报错。

接着是对stmt进行一个强制转换，将其转换为create\_table\_stmt，即子类的一个属性，我们紧接着跳转到CreateTableStmt这个子类中。

可以发现，在create\_table\_stmt中，可以定义table\_name和attr\_info，即表名以及表的一些属性，这些都是我们建表过程中要用到的重要属性。

对stmt进行强制转换为create\_table\_stmt后，便可以调用子类的函数，取出create\_table\_stmt中的表名table\_name以及属性信息attr\_info，进行建表。

因为session是沟通进程和实际db之间的桥梁，因此要从session中取出当前的db，然后使用当前db的一个create\_table方法建表，传入的参数正是刚刚从create\_table\_stmt中获取的table\_name以及attr\_info。

我们转入db类的这个create\_table函数

（这里我们发现，在2024年4月29号，miniob官方仓库对create\_table函数的实现以及参数进行了更新，因此目前报告中以最新一次更新为准。）

在create\_table中，参数获取了表名和属性。

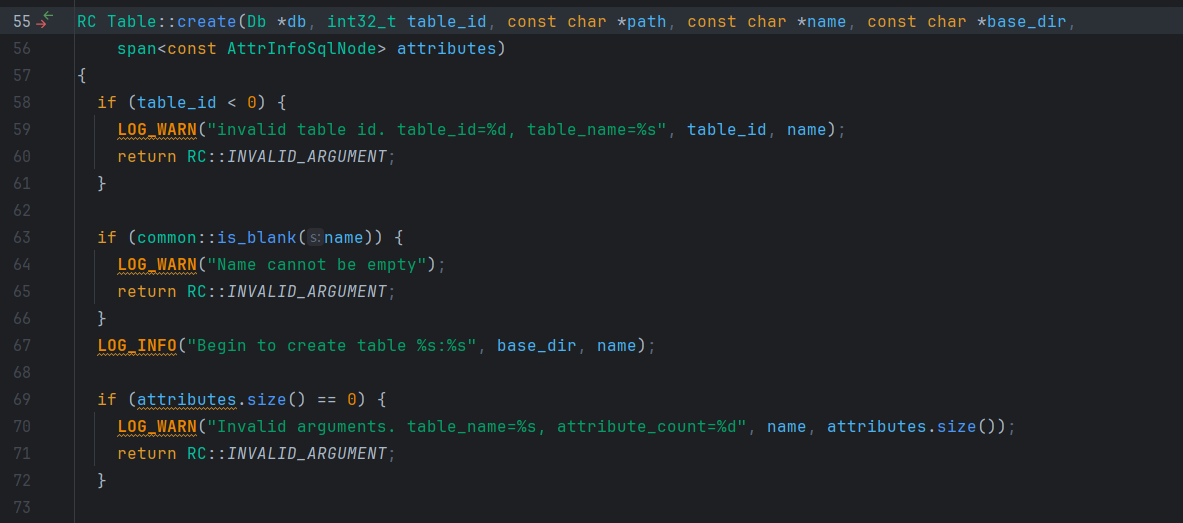
首先程序会对数据库已经打开的表进行一个检查，如果有同名的表的话就会报错警告。这个表已经被打开了。

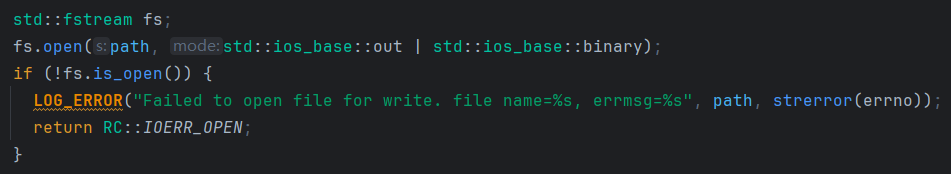
 rc = table->create(table\_id, table\_file\_path.c\_str(), table\_name, path\_.c\_str(), attribute\_count, attributes);

之后便是用数据库的路径和表名创建了一个元数据存储的路径。

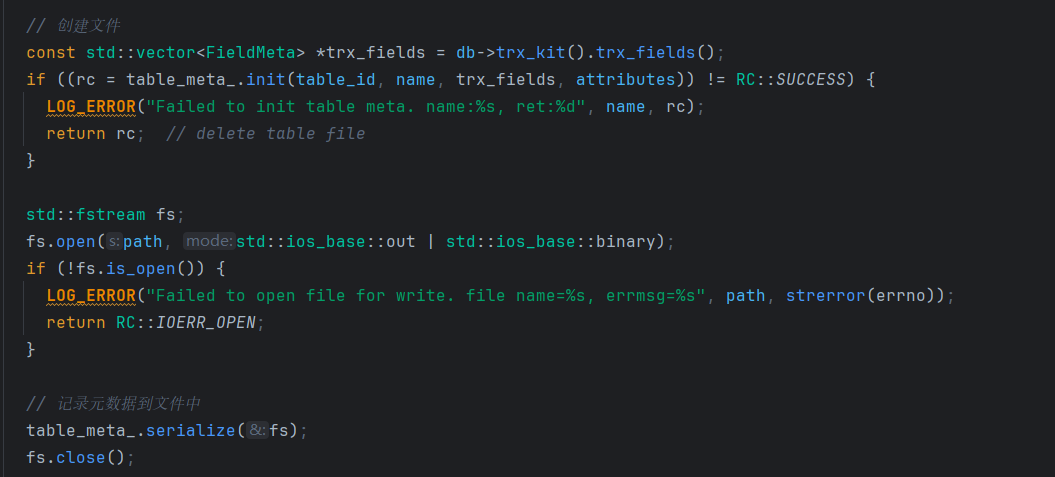
然后开一个新表。

并设置表的id是自增的，每次创建新表id+1。

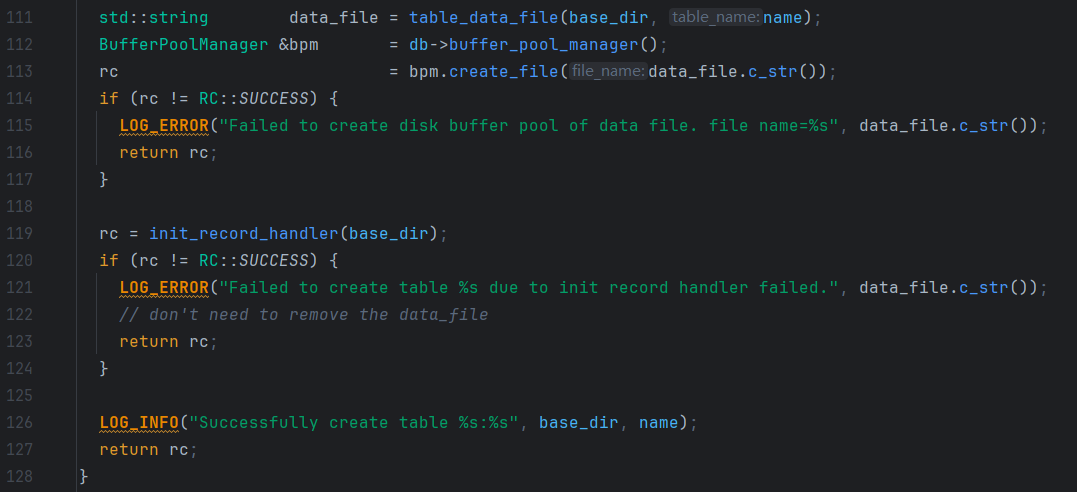
然后调用table的一个create函数，这个函数参数的意思稍微有点模糊，经过查询可知table\_file\_path是表的元数据的位置，而Path则是数据库记录会存放在哪里，二者是不一样的。

在create这个函数中，先对一些东西进行检查，比如表的id是否小于0，表名是否为空白，属性的数量是否为0。

检查过后打开一个fstream，然后open打开，然后做一些检查，如果没开起来或者文件已存在就报错，如果成功开启或者成功创建，就记录元数据到文件中。

这就完成了表的创建。

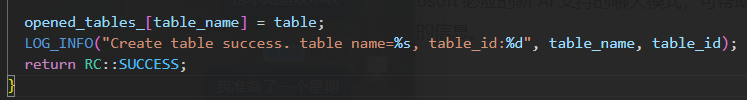
在create这个函数中还有一些语句，虽然表的创建已经完成但还是对其做个解释。

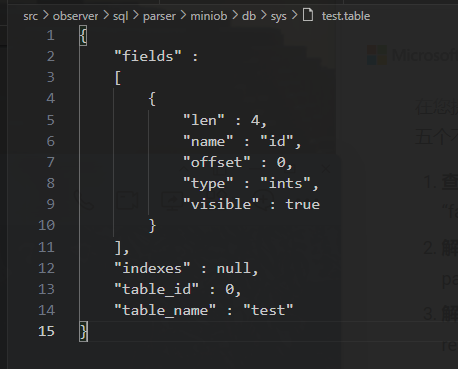
这一部分是对刚刚数据库中的地址和表名进行一个拼接，并将其作为存放地址，之后调用db中的buffer\_pool\_manager函数创建一个bufferpool。

之后初始化一个record\_handler，并将base\_dir作为参数传入。

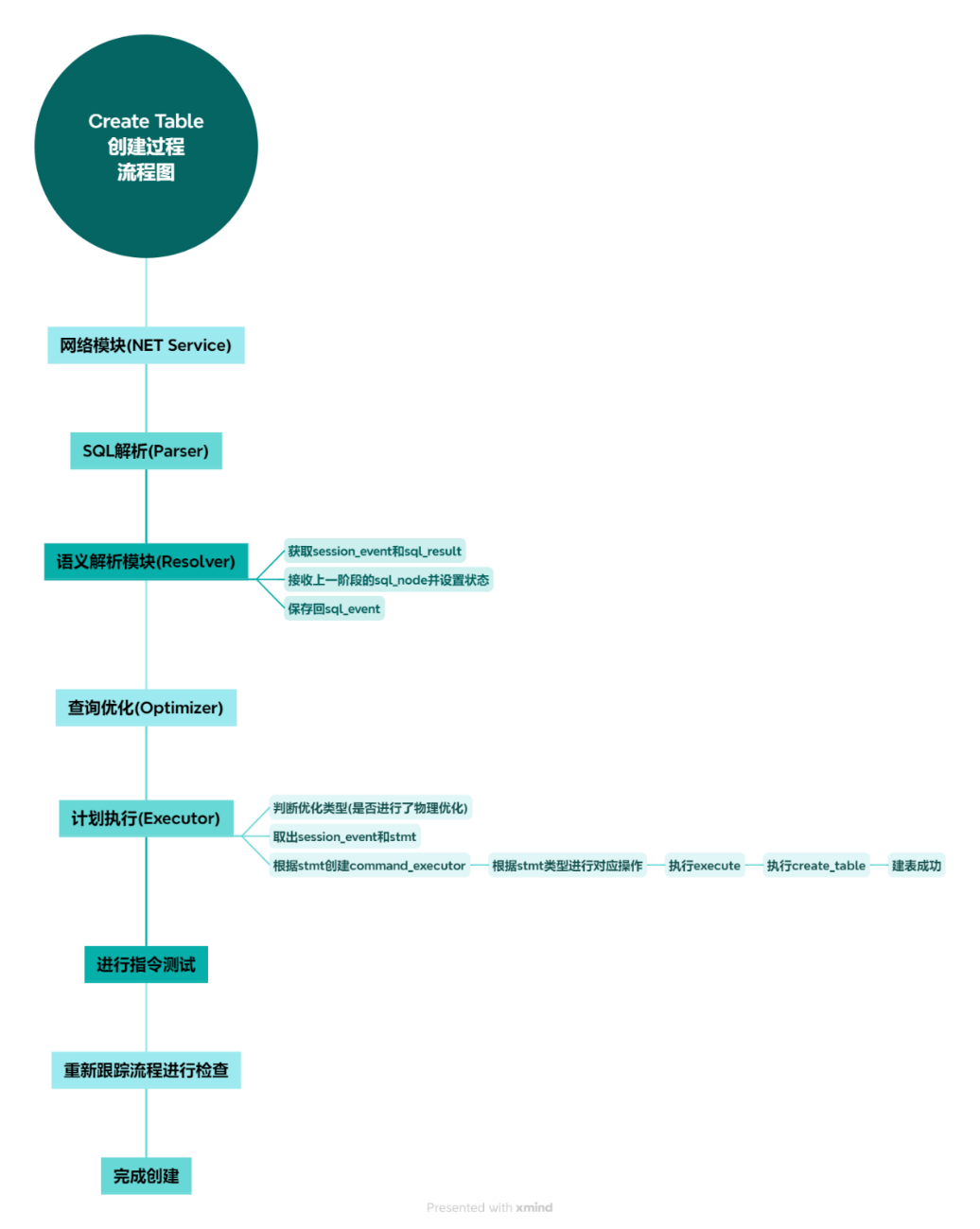
init\_record\_handler基本上就是对路径进行一些定义和保存，即流程图中提到的记录管理(Record Manager)：负责管理某个表数据文件中的记录存放；

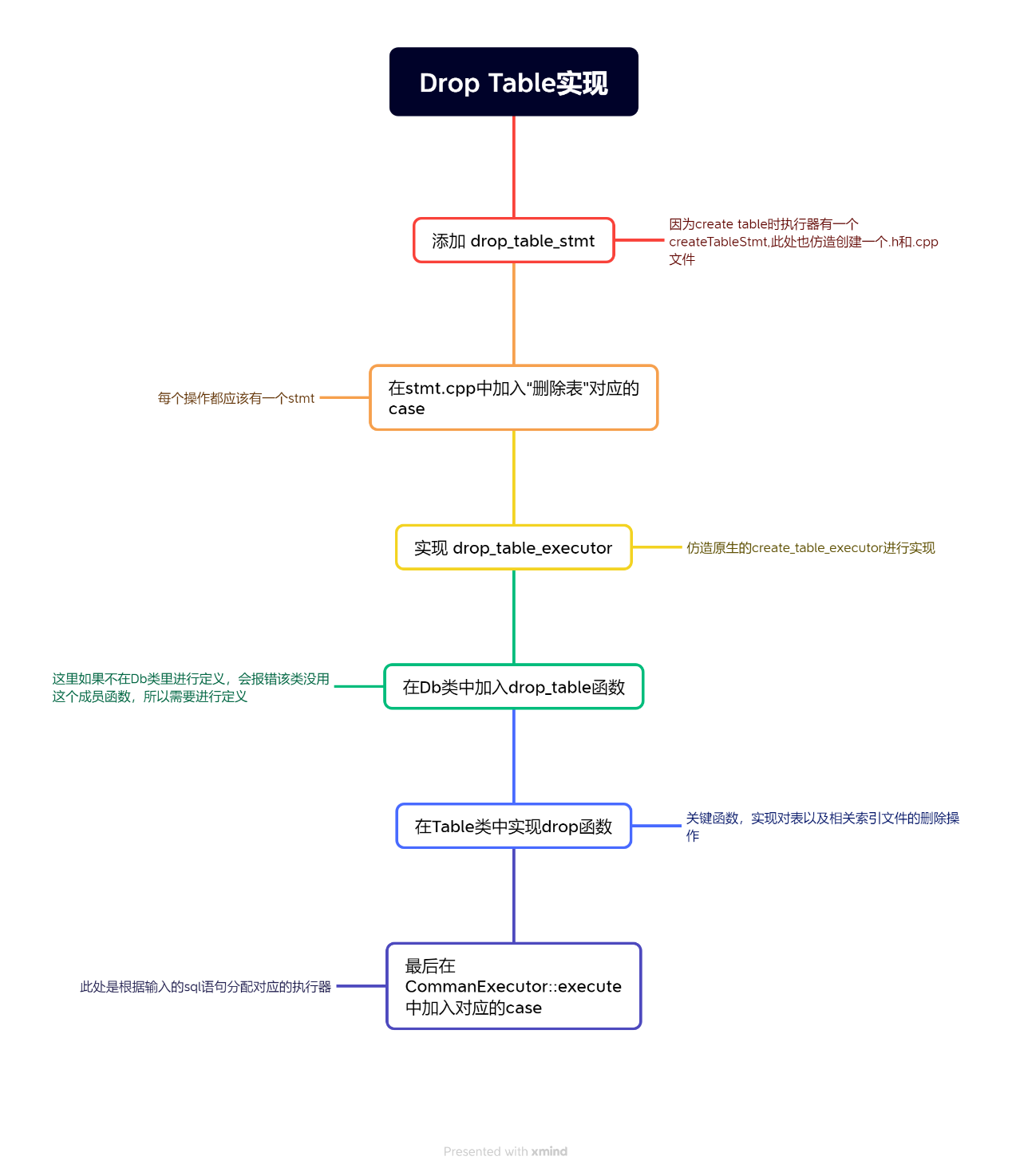
至此，表的创建底层原理部分流程结束。

接着返回外围的函数继续执行。

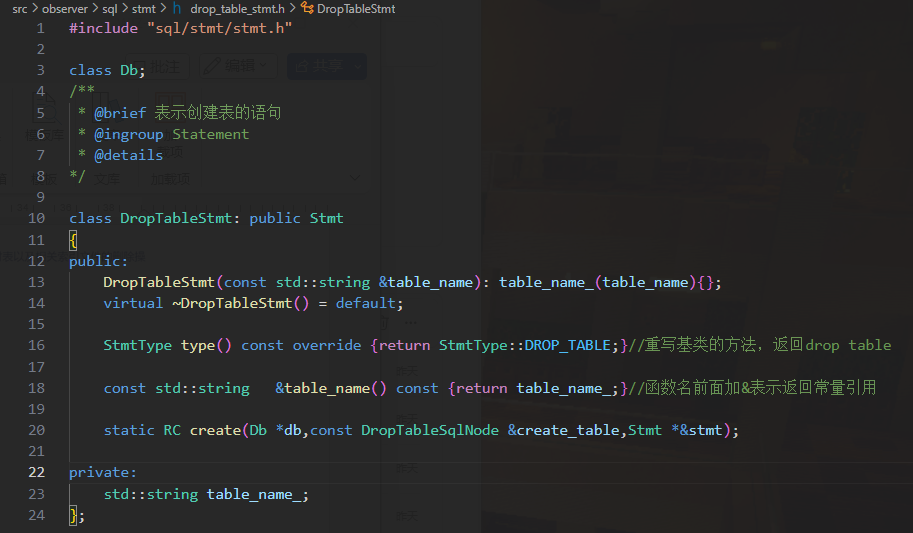
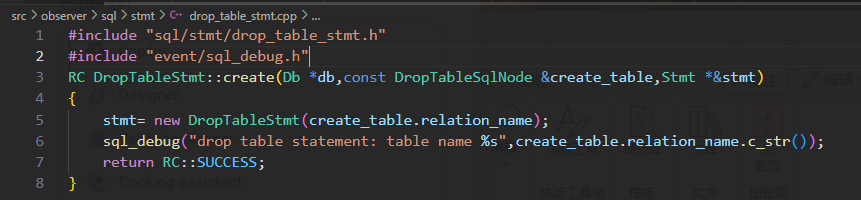
我们输入指令进行调试，发现表成功创建，跟踪流程就是上文提到的整个过程，整个create\_table的流程正式结束。

在此附上Create Table创建过程的流程图。

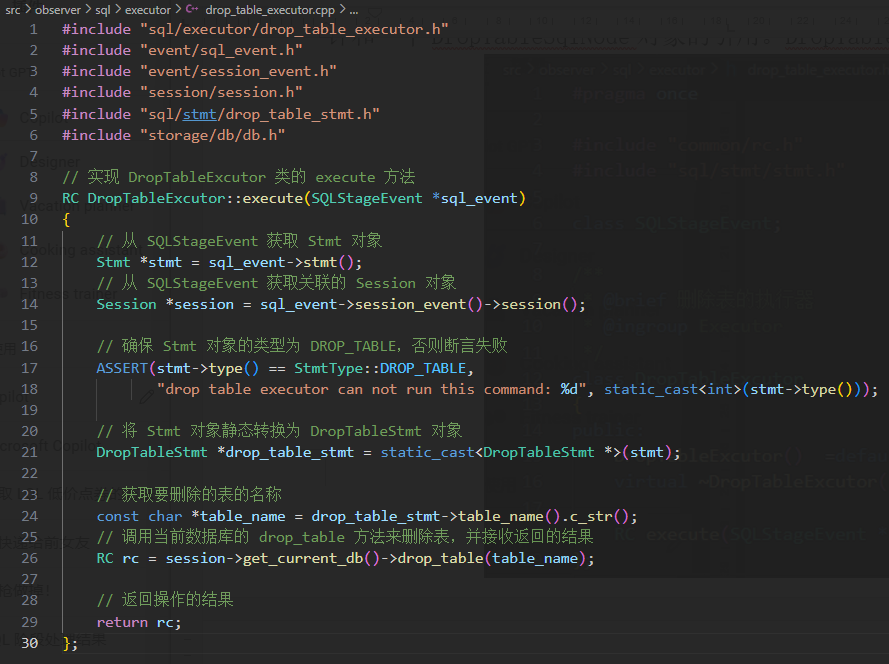


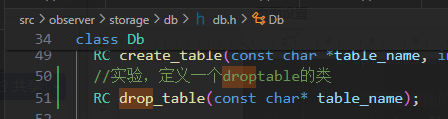
1. Drop Table的实现
2. 2. 设计Drop Table和Alter Table的实现方案，画出设计方案图。
3. 3. Drop Table 、Alter Table的实现代码。
4. 4. Drop Table 、Alter Table功能测试。
5. 5. 总结

根据create table的对应操作流程，我们做了上图所示的 drop table 操作流程。下面依次给出我们的具体实现代码。

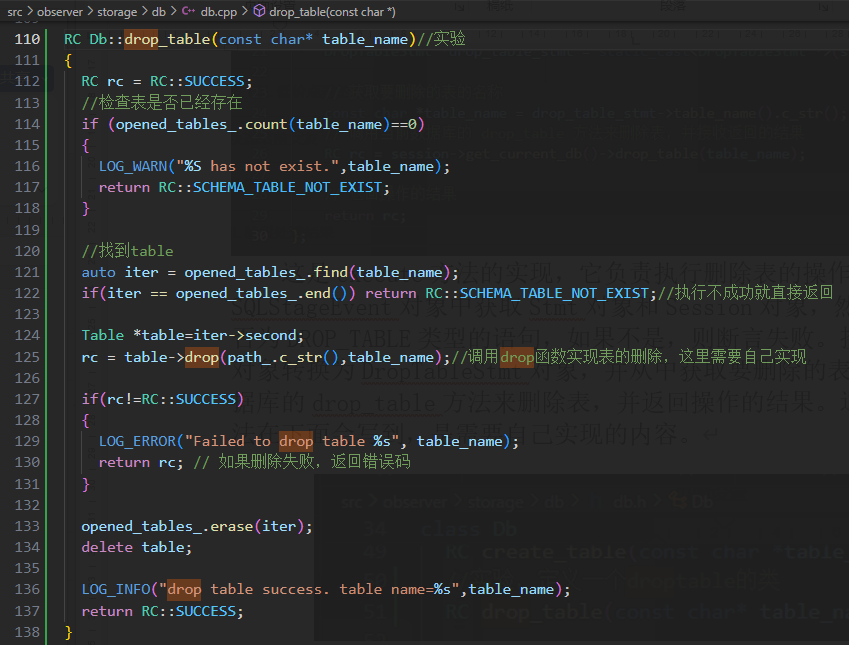
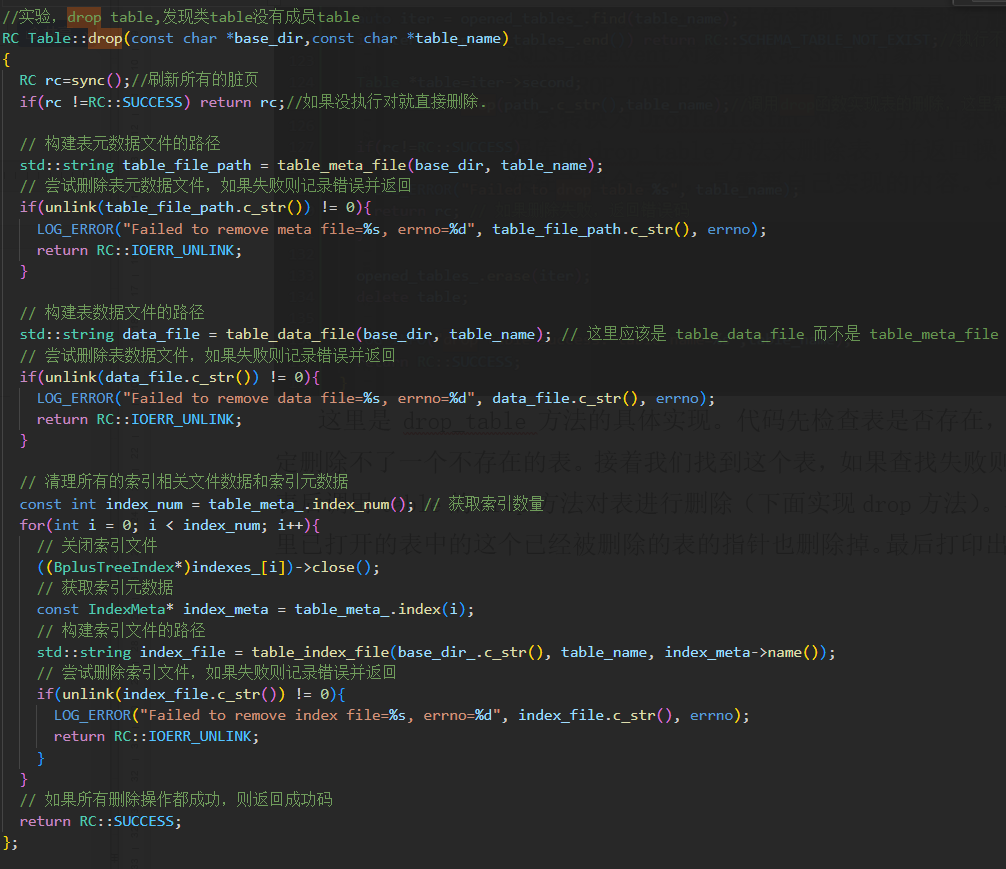
这里定义了一个DropTableStmt类，它继承自Stmt基类，表示一个删除表的SQL语句。它包含一个构造函数，用于接收表名，并将其存储在私有成员变量中。类中还有一个方法type，用于返回语句的类型（在这里是DROP\_TABLE），以及一个返回表名的方法。此外，还有一个静态方法create，用于创建一个DropTableStmt对象。

这是create方法的实现。它的作用是创建一个新的DropTableStmt对象，并将其地址赋值给传入的stmt指针。它接受两个参数，一个指向Db类对象的指针和一个DropTableSqlNode对象的引用。DropTableSqlNode对象里有表名。

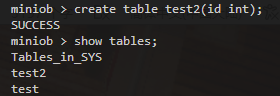
这里是drop table执行器的类定义，主要是定义drop table需要用到的execute函数，以供后续实现和调用。

这是execute方法的实现，它负责执行删除表的操作。首先从SQLStageEvent对象中获取Stmt对象和Session对象，然后验证Stmt对象是否为DROP\_TABLE类型的语句，如果不是，则断言失败。接下来，代码将Stmt对象转换为DropTableStmt对象，并从中获取要删除的表名。最后，它调用数据库的drop\_table方法来删除表，并返回操作的结果。这里的drop\_table方法在下面会写到，是需要自己实现的内容。

首先需要在Db.h头文件中进行定义。下面进行具体的实现。

上面是drop\_table方法的具体实现。代码先检查表是否存在，因为我们肯定删除不了一个不存在的表。接着我们找到这个表，如果查找失败则返回。找到表后调用table的drop方法对表进行删除（下面实现drop方法）。再把buffer里已打开的表中的这个已经被删除的表的指针也删除掉。最后打印出对应的信息。

最后在table.cpp内实现drop函数。方法负责删除表及其相关文件。首先刷新脏页以确保数据一致性，然后尝试删除表的元数据和数据文件。如果文件删除失败，会记录错误并返回相应错误码。接下来，方法会遍历所有索引，关闭并删除索引文件。如果所有文件都成功删除，方法返回成功码，表示表及其相关文件已被成功删除。这个过程确保了数据库的完整性和一致性在删除操作中得到维护。

到这里为止，我们实现了drop table功能。下面进行测试，看看是否能正常使用。

我们首先创建两个测试用表。



我们尝试删除一个不存在的表，发现失败了。

删除之前创建的表，显示删除成功，并且再次展现table的时候被删除的表也消失了，测试成功！到此，drop table的功能已成功实现。

3. Alter Table实现

根据我们小组在网上查询到的SQL语句，AlterTable的功能是用于在已有的表中添加、修改或删除列。

添加表中的列的语句是：

ALTER TABLE table\_name

ADD column\_name datatype

删除表中的列的语句是

ALTER TABLE table\_name

DROP COLUMN column\_name

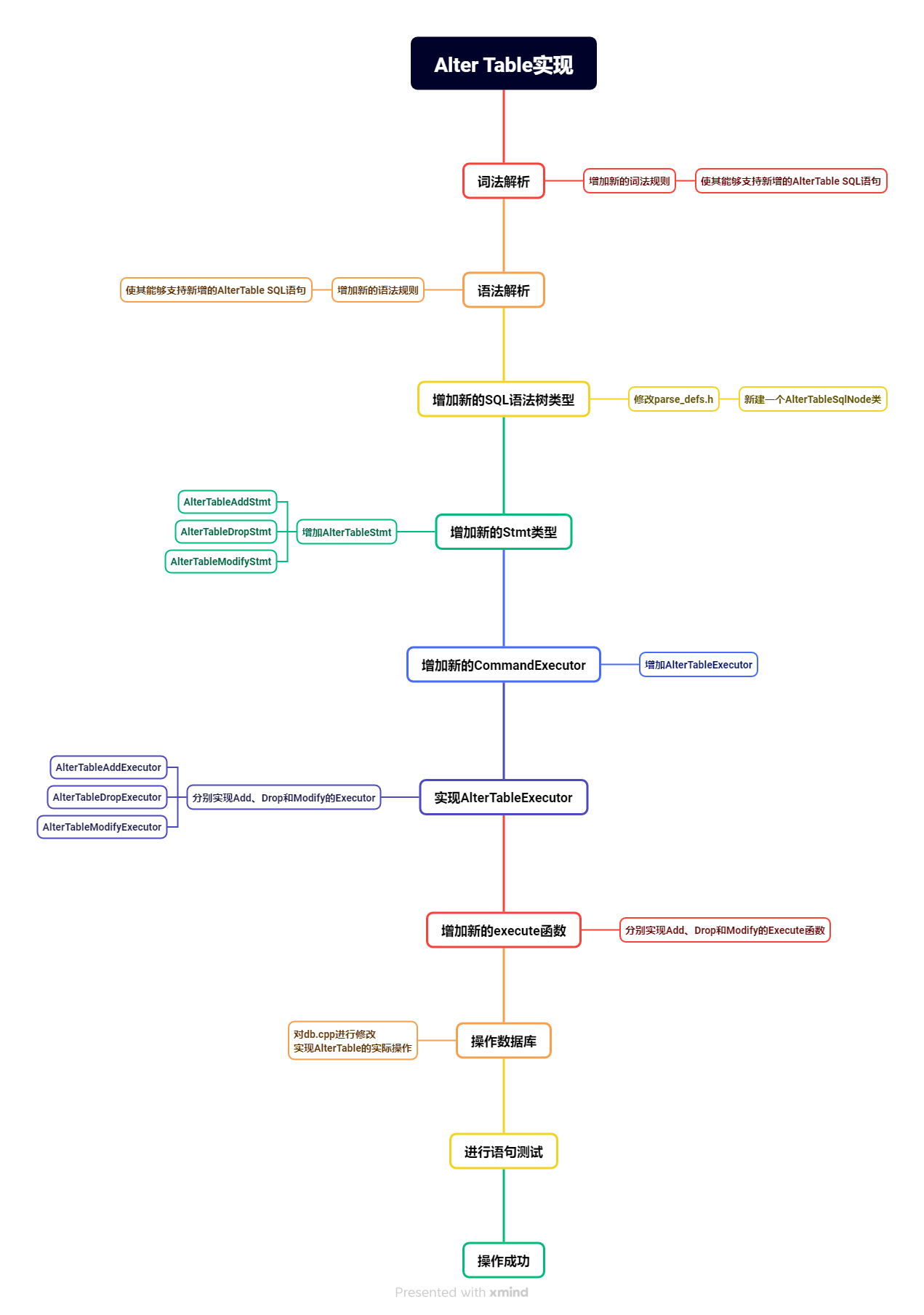
修改表中的列的属性的语句是

ALTER TABLE table\_name

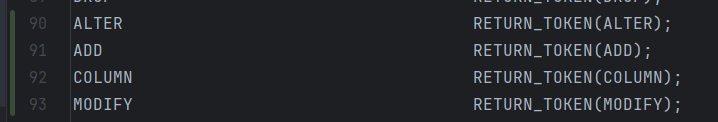
ALTER COLUMN column\_name datatype

根据这些语句，我们可以进行AlterTable的实现，完成实验要求。

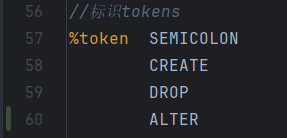
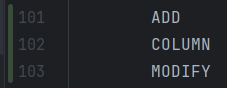
首先是AlterTable实现的流程图：



首先，要能够读取我们输入的SQL语句，必须进行词法解析和语法解析。因此我们需要编写对应的文件。

 我们打开src/observer/sql/parser/lex\_sql.l 这个文件是进行词法解析的编写文件，我们要在其中加入我们的与AlterTable有关操作的TOKEN，方便后续语法解析使用。

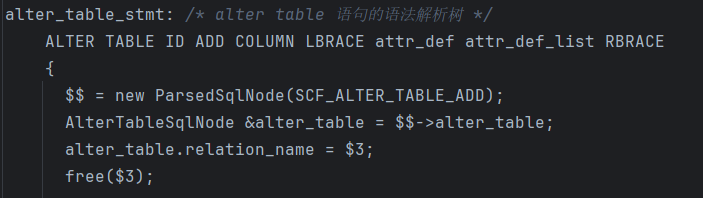
添加完毕后，进入src/observer/sql/parser/yacc\_sql.y 这个文件是进行语法解析的编写文件，我们也要在其中加入合适的代码，让其能够识别我们的AlterTable有关的SQL语句。

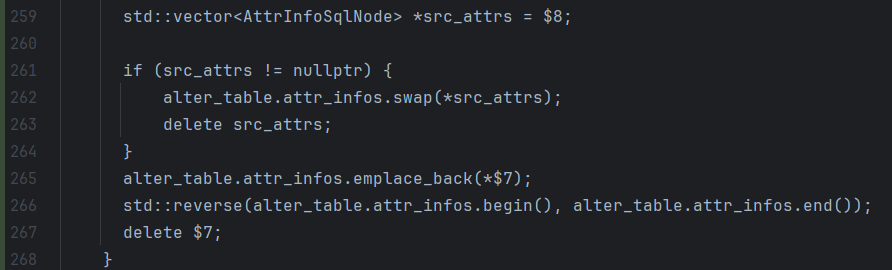
 因为刚刚在lex\_sql.l文件中加了一些TOKEN，所以我们要在yacc\_sql.y文件中标识出来。

 将其标识出来后，对Type进行编辑，添加alter\_table\_stmt。

同时，在command\_wrapper中也添加alter\_table\_stmt。

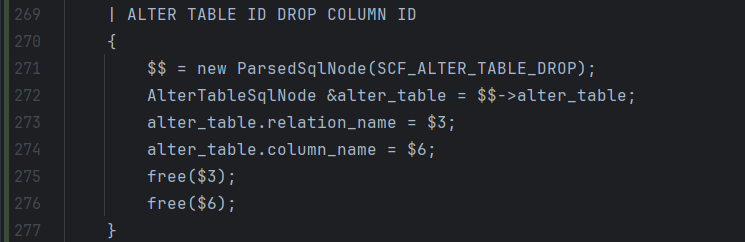
之后，对alter\_table\_stmt进行定义，即写出alter table 语句的语法解析树。

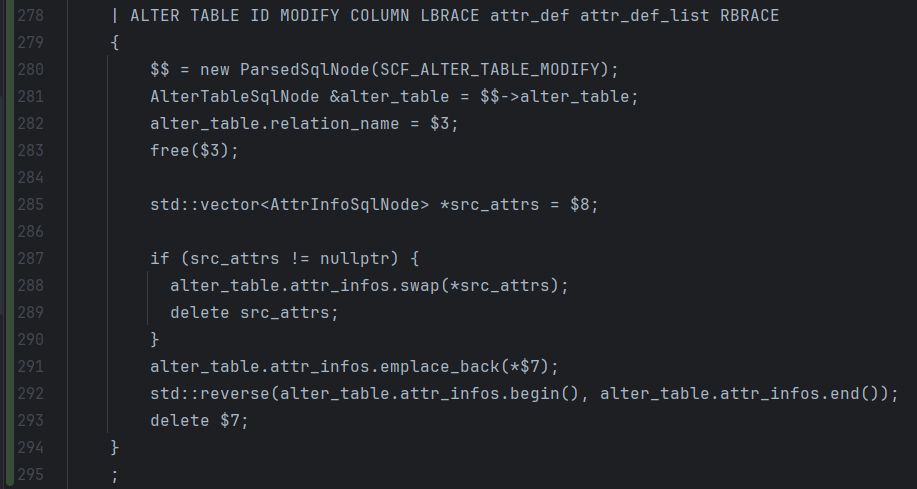




这里用到的attr\_def和attr\_def\_list是在本就有的create\_table\_stmt中写好的一部分，这里直接拿来调用。

也要编写Drop和Modify的部分。





编写完毕后，词法解析和语法解析的文件编写部分就结束了。

然后我们在src/observer/sql/parser/目录下输入该指令  
 ./gen\_parser.sh

就会对这两个文件进行编译，编译出对应的.cpp文件。

至此，词法解析和语法解析部分完成。

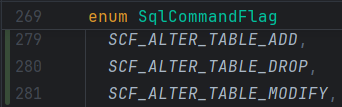
接下来，增加新的SQL语法树类型。

我们进入src/observer/sql/parser/parse\_defs.h文件中，会发现其中定义了很多的SQLParser，其中也有我们之前用到的CreateTable和DropTable的Parser，我们模仿一下CreateTable的实现。



写出了一个AltertableSqlNode的结构体，其中包含了表名，列名和属性。这三个值都是我们后续进行AlterTable时要用到的。

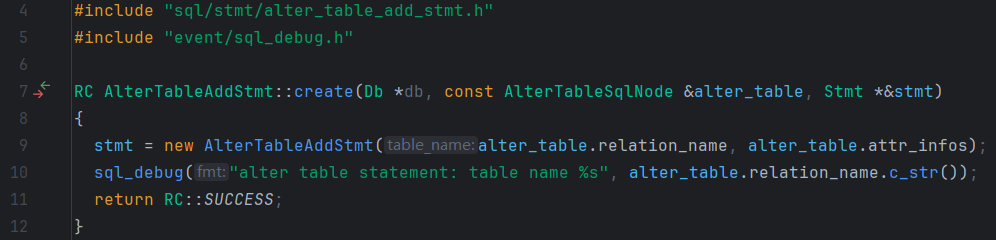
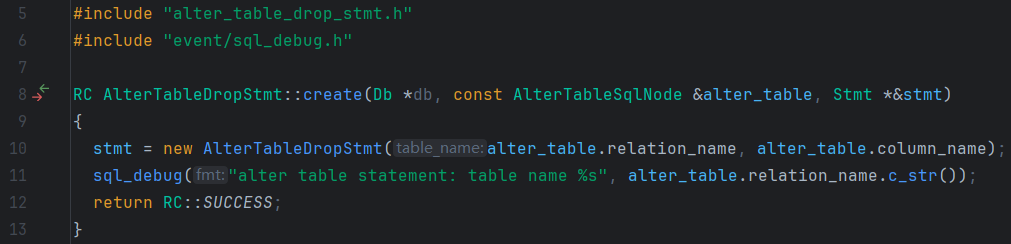
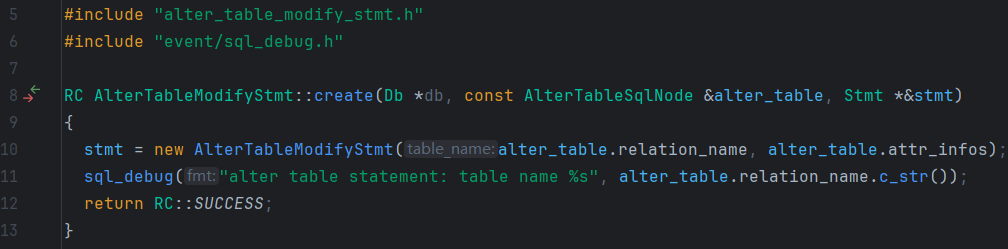
之后，在SqlCommandFlag枚举中加入对应的AlterTable指令，方便后续使用。

   
  
  
  
  
 因为上边写了一个结构体，所以需要在ParsedSqlNode类中添加一个AlterTableSqlNode。

  
  
  
 新的SQL语法树类型添加完毕。

紧接着，我们要增加新的stmt类型，进入src/observer/sql/stmt目录下。

模仿着create\_table\_stmt的代码，可以写出alter\_table\_add\_stmt, alter\_table\_drop\_stmt, alter\_table\_modify\_stmt的代码。

   
  
  
  
  
  
  
  
  
  
  
  
  
  
 三个stmt比较类似，只是针对不同的输入进行了一些细微的调整。

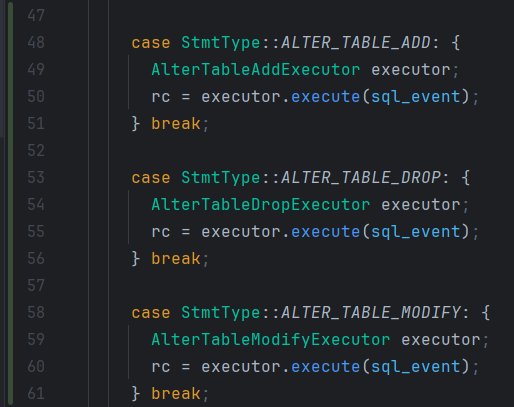
   
  
  
  
  
  
  
  
  
  
  
  
  
 上图是alter\_table\_add\_stmt.h，也是模仿着create\_table\_stmt.h进行编写的，只对一些参数和变量类型进行了修改。

alter\_table\_drop\_stmt.h和alter\_table\_modify\_stmt.h同理。只有一些变量需要调整。

Stmt类型增加完成后，我们需要增加新的CommandExecutor，这个CommandExecutor是一个执行器，是执行命令的主要部分。

根据我们刚刚定义的StmtType，我们可以写出三个对应的CommandExecutor

分别执行我们输入的三条不同的AlterTable语句



这其中又包含了三个具体的Executor需要我们去实现。

来到src/observer/sql/executor目录下，创建三个不同的executor，分别是alter\_table\_add\_executor, alter\_table\_drop\_executor, alter\_table\_modify\_executor。每个执行器都编写.cpp和.h文件，分别根据不同的输入执行不同的功能。

   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
 三张图片从上到下依次是add、drop、modify的executor的具体实现。

三个executor比较类似，只有参数和Stmt不同。

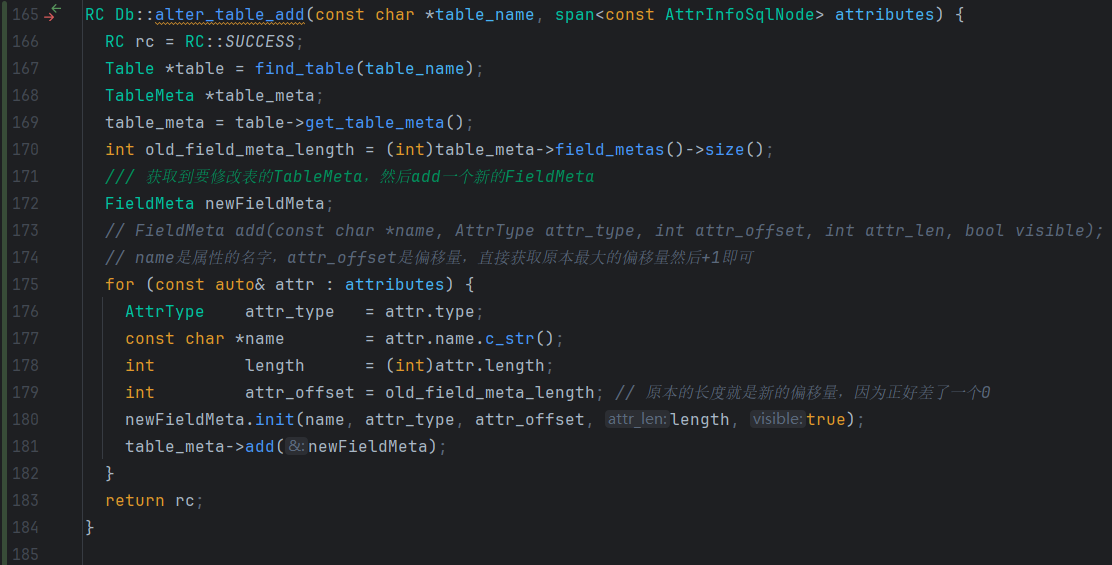
 该图是alter\_table\_add\_executor的头文件，drop和modify的都十分类似，因此不在此过多展示。

在每个executor中，都调用了db的相关函数实际上来执行我想要执行的操作，数据库本身提供的create\_table指令也是这样实现的。

因此，我们要继续在db.cpp中编写我们AlterTable相关的函数，以便我们的功能能够顺利执行。

进入src/observer/storage/db/db.cpp文件中，我们会发现这里有很多熟悉的指令，比如说create\_table, find\_table之类的，db.cpp就是编写实际执行命令的函数的地方。

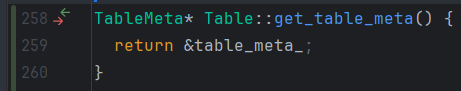
我们要在这里编写AlterTable相关的函数。



在我写的alter\_table\_add函数中，我接收了表名和属性作为参数，因为要对表的列进行添加，属性中包含属性名、属性类型和属性长度，所以我们可以直接对元数据进行操作，直接添加。

经过一番研究我们可以发现，表的元数据是存储在TableMeta中的，属性（即字段）的元数据是存储在FieldMeta中的，既然我们要添加字段，那就得获取到表名对应的TableMeta，直接往TableMeta中加入根据我们新输入的属性创建的FieldMeta。  
 每一个表都是一个TableMeta对象，每一个字段都是一个FieldMeta对象。

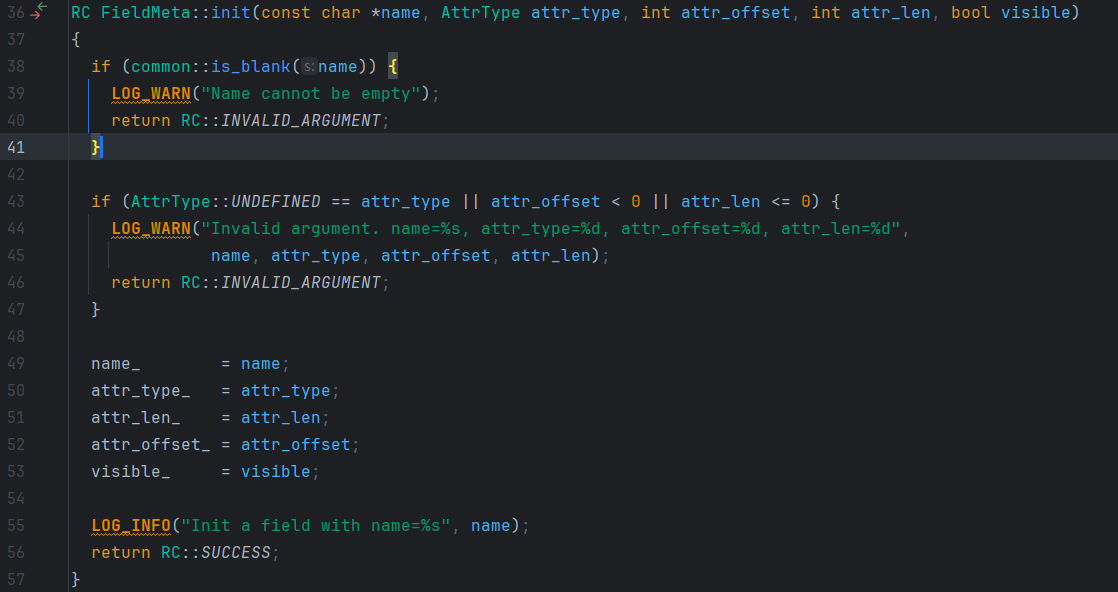
因此，我们先使用miniob自带的函数find\_table，根据我们传入的表名找到对应的表，然后对其引用，得到表名对应的TableMeta，然后对这个TableMeta进行后续操作，因为不能直接获取Table对象的TableMeta，因此我写了一个get函数来获取TableMeta。



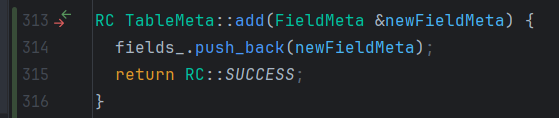
函数也比较简单，只是获取Table对象的TableMeta再将其传出。该函数在src/observer/storage/table/table.cpp中进行实现

然后new一个新的FieldMeta，用来根据新输入的属性对FieldMeta对象进行操作。

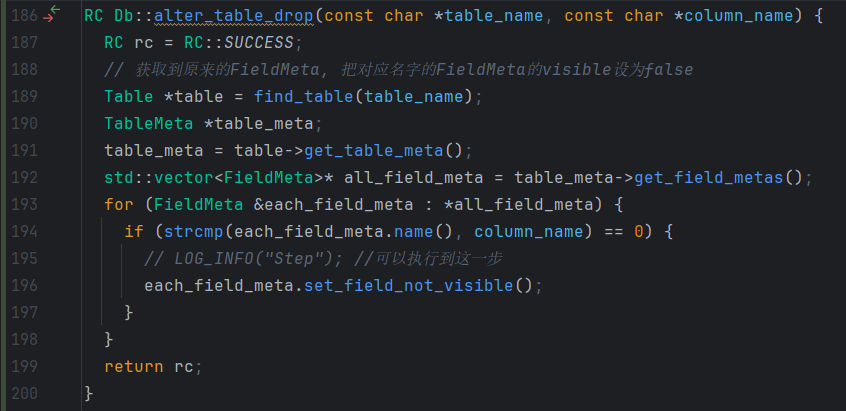
这里我们调用src/observer/storage/field/field\_meta.cpp中对FieldMeta对象进行初始化的init函数，把我们从attributes中获取出的属性名、属性类型、长度都作为参数传进init中。

 init后，我们创建的FieldMeta就包含了我们传入的attributes中的所有内容，这之后我们把newFieldMeta加入TableMeta中即可，这时候的TableMeta就包含了我们新建的FieldMeta，字段添加成功。

这里我对TableMeta实现了一个add函数，方便添加FieldMeta对象。

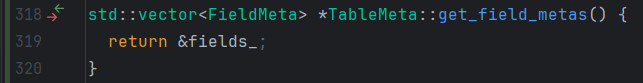
   
  
  
  
 比较简单，只是对TableMeta的fields\_添加了一个FieldMeta对象。在src/observer/storage/table/table\_meta.cpp中实现。

之后便是在db.cpp中添加AlterTable drop有关的功能

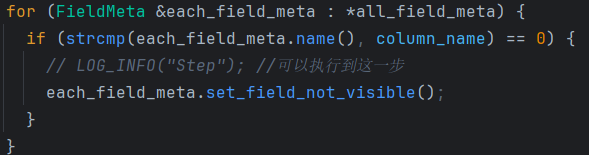
   
  
  
  
  
  
  
  
  
  
 drop和add的实现差别不大，用for循环遍历FieldMeta对象的name，和传入的列名进行比较，如果相同就把该列设置为不可见。

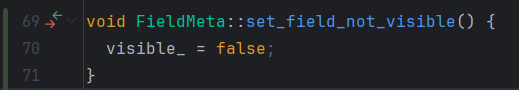
不直接删除该列的原因有两点：1. 当用户操作的表是一张超级大表时，直接删除会占用很长的时间，用户体验极差。2. 对表的列进行删除时，需要考虑偏移量和列是否会对其他表、其他列造成影响，考虑的因素较多，会对后续用户的操作产生影响。

和刚刚add的操作一样，先获取TableMeta，然后取出其中所有的FieldMeta



还是因为不能直接获取field\_，因此写了一个get函数，在src/observer/storage/table/table\_meta.cpp中实现。

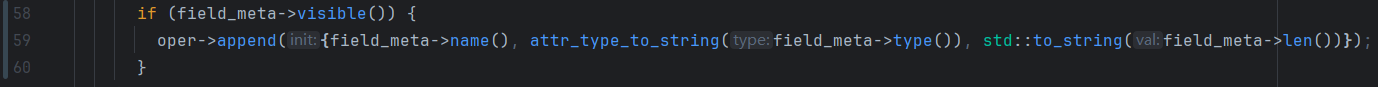
   
  
  
  
  
 for循环遍历比较name，相同就执行set\_field\_not\_visible，把相同的FieldMeta对象的visible属性设置为false。



在src/observer/storage/field/field\_meta.cpp中实现。

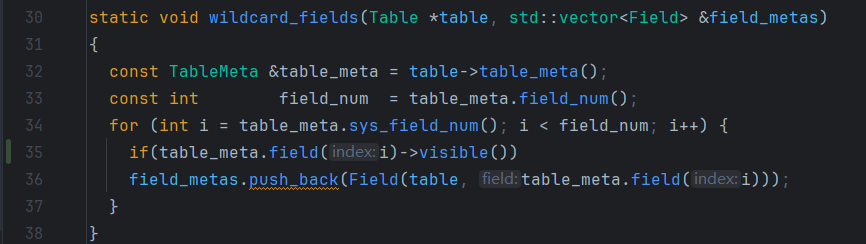
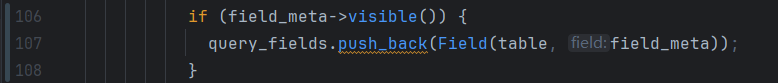
visible设置为false之后，还不能保证用户不可见，因此要对能够查询表的两个指令desc和select进行修改。

在src/observer/sql/executor/desc\_table\_executor.cpp中对desctable进行修改。



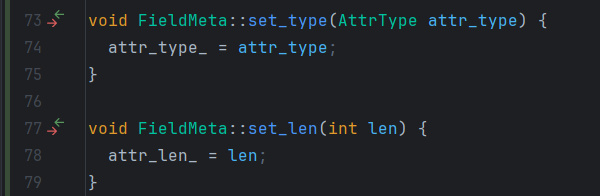
如果field\_meta可见，才将该field\_meta放入展示列表中，保证了不可见的对象不会被用户查询到或是看到。

同时，还要在src/observer/sql/stmt/select\_stmt.cpp对select进行修改。

   
  
  
  
  
  
   
  
 修改其中包含显示逻辑的语句，把visible作为条件进行筛选，以此可以确保如果FieldMeta是不可见的，用户不能查询到该FieldMeta。

Add与Drop实现完毕，接下来是Modify。

   
  
  
  
  
  
  
  
  
  
  
  
 Modify与Add还有Drop异曲同工，有了前两个的基础，Modify的实现就简单了很多。和刚刚的流程一样，先获取TableMeta，然后获取FieldMeta，根据for循环遍历，找到所有FieldMeta对象中，与传入属性的name相同的FieldMeta，然后编写两个set函数，修改其属性类型和长度即可。

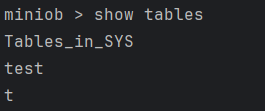


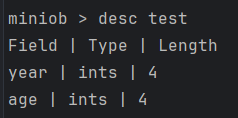
两个set函数均在src/observer/storage/field/field\_meta.cpp中实现。

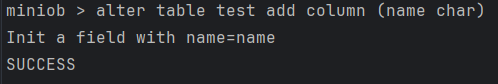
至此，AlterTable的功能已全部实现，代码均已展示。

接下来对实现的功能进行测试。

首先，我们对alter\_table\_add进行测试。

  
  
  
  
 为方便演示，我已创建好表test，我们可以对test进行查看

  
  
  
  
  
 可以看到test中有两个Field，同时也显示了它们的类型和长度。

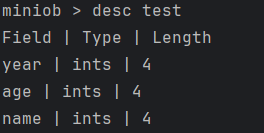
  
  
  
 当我们使用altertable add时，会提示我们SUCCESS

再次查看test表

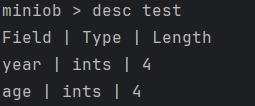
  
  
  
  
  
 会发现已经添加了char类型的name字段，add功能正常。

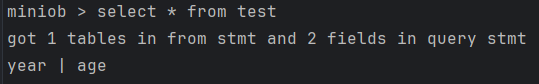
方便起见，我们先测试Modify



   
  
  
  
  
 先执行Modify，然后对test进行查看，发现name字段的type已经变为int，modify功能正常。

最后，测试Drop功能。

   
   
   
  
  
  
 先drop column后再查看该表，发现只有两个Field。

  
  
  
  
 使用select查询也只能看到2个fields，drop功能正常。

至此，已经完成了Alter Table的所有功能，实现代码均已展示，相关功能均已进行说明以及图片展示。