# 一、Main.c文件

在SQLite的源代码中，main.c 文件是一个非常重要的组成部分，它主要负责实现SQLite的命令行工具sqlite3。这个命令行工具允许用户通过终端或命令提示符与SQLite数据库进行交互。main.c文件中的代码实现了读取用户输入的SQL语句、执行这些语句，并显示查询结果的功能。

## sqlite3Close（）函数1228

作用：关闭所有数据库连接

流程：

1.检查数据库指针是否为空

if( !db ){

return SQLITE\_OK;

2.检查数据库是否处于可接受的状态

if( !sqlite3SafetyCheckSickOrOk(db) ){

return SQLITE\_MISUSE\_BKPT;

}

3.进入互斥锁

sqlite3\_mutex\_enter(db->mutex);

4.检查是否启用了跟踪标志，若启用，调用相应的跟踪回调函数：

if( db->mTrace & SQLITE\_TRACE\_CLOSE ){

db->trace.xV2(SQLITE\_TRACE\_CLOSE, db->pTraceArg, db, 0);

}

5.断开所有虚拟表连接：

disconnectAllVtab(db);

6.回滚所有未完成的虚拟表事务：

sqlite3VtabRollback(db);

7.如果没有强制关闭并且连接正忙，则报告忙碌并退出：

if( !forceZombie && connectionIsBusy(db) ){

sqlite3ErrorWithMsg(db, SQLITE\_BUSY, "unable to close due to unfinalized "

"statements or unfinished backups");

sqlite3\_mutex\_leave(db->mutex);

return SQLITE\_BUSY;

}

8.若启用，则记录sql日志

#ifdef SQLITE\_ENABLE\_SQLLOG

if( sqlite3GlobalConfig.xSqllog ){

/\* Closing the handle. Fourth parameter is passed the value 2. \*/

sqlite3GlobalConfig.xSqllog(sqlite3GlobalConfig.pSqllogArg, db, 0, 2);

}

#endif

9.清理客户端数据链表

while( db->pDbData ){

DbClientData \*p = db->pDbData;

db->pDbData = p->pNext;

assert( p->pData!=0 );

if( p->xDestructor ) p->xDestructor(p->pData);

sqlite3\_free(p);

}

10.将数据库设置为僵尸状态：

db->eOpenState = SQLITE\_STATE\_ZOMBIE;

11.解除互斥锁，并执行清理僵尸的工作：

sqlite3LeaveMutexAndCloseZombie(db);

return SQLITE\_OK;

## sqlite3\_txn\_state（）函数1290

作用：返回指定数据库或所有附加数据库的最大事务状态

流程：

1.安全检查

ifdef SQLITE\_ENABLE\_API\_ARMOR

if( !sqlite3SafetyCheckOk(db) ){

(void)SQLITE\_MISUSE\_BKPT;

return -1;

}

#endif

2.获取互斥锁

sqlite3\_mutex\_enter(db->mutex);

3.查询事务状态：

if( zSchema ){

nDb = iDb = sqlite3FindDbName(db, zSchema);

if( iDb<0 ) nDb--;

}else{

iDb = 0;

nDb = db->nDb-1;

}

for(; iDb<=nDb; iDb++){

Btree \*pBt = db->aDb[iDb].pBt;

int x = pBt!=0 ? sqlite3BtreeTxnState(pBt) : SQLITE\_TXN\_NONE;

if( x>iTxn ) iTxn = x;

}

若zSchema不为null并可查询到，则直接查询该指定数据库的事务状态。

若zSchema不为null但未查询到，则跳出循环且返回-1.

若zSchema为null，则进入循环查询所有附加数据库的最大事务状态。

4.释放互斥锁

sqlite3\_mutex\_leave(db->mutex);

return iTxn;

## sqlite3\_close（）与sqlite3\_close\_v2（）函数1325

作用：两个关闭SQLite数据库连接的公共接口函数：实际上是调用同一个内部函数sqlite3\_Close，但传递不同的参数来控制关闭行为。

## sqlite3\_LeaveMutexAndCloseZombie（）函数1337

作用：函数为数据库关闭连接的过程，尤其是处理“僵尸”连接的情况。函数通过确保所有未提交的事务回滚，释放与数据库连接相关的各种资源（包括保存点、模式、虚拟表、函数和排序规则等），最终完全关闭并释放连接的内存。

流程：

1.检查数据库连接状态

if( db->eOpenState!=SQLITE\_STATE\_ZOMBIE || connectionIsBusy(db) ){

sqlite3\_mutex\_leave(db->mutex);

return;

}

若数据库不处于zombie状态或者连接正忙，则释放互斥锁并返回。

2.回滚未提交的事务

sqlite3RollbackAll(db, SQLITE\_OK);

3.释放保存点

sqlite3CloseSavepoints(db);

4.关闭所有数据库连接

for(j=0; j<db->nDb; j++){

struct Db \*pDb = &db->aDb[j];

if( pDb->pBt ){

sqlite3BtreeClose(pDb->pBt);

pDb->pBt = 0;

if( j!=1 ){

pDb->pSchema = 0;

}

}

}

遍历所有数据库，关闭btree结构并将非temp数据库的模式结构（pSchema）指针置为 NULL

5.清除临时模式

if( db->aDb[1].pSchema ){

sqlite3SchemaClear(db->aDb[1].pSchema);

}

特别处理temp数据库的模式清理

6.释放虚拟表相关的锁

sqlite3VtabUnlockList(db);

7.释放数据库数组

sqlite3CollapseDatabaseArray(db);

assert( db->nDb<=2 );

assert( db->aDb==db->aDbStatic );

释放数据库数组中的任何额外资源，这里清除所有非主数据库的资源

断言数据库连接最多包含两个数据库

确保数据库连接数组指向的是静态的默认数组

8.清理自定义函数

for(i=sqliteHashFirst(&db->aFunc); i; i=sqliteHashNext(i)){

FuncDef \*pNext, \*p;

p = sqliteHashData(i);

do{

functionDestroy(db, p); 清理每个函数的内部数据结构。

pNext = p->pNext;

sqlite3DbFree(db, p); 释放与自定义函数相关的内存。

p = pNext;

}while( p );

}

sqlite3HashClear(&db->aFunc); 清空自定义函数哈希表

9.清理排序规则

for(i=sqliteHashFirst(&db->aCollSeq); i; i=sqliteHashNext(i)){

CollSeq \*pColl = (CollSeq \*)sqliteHashData(i);

/\* Invoke any destructors registered for collation sequence user data. \*/

for(j=0; j<3; j++){

if( pColl[j].xDel ){

pColl[j].xDel(pColl[j].pUser);

}

}

sqlite3DbFree(db, pColl); 释放与 pColl 相关的内存

}

sqlite3HashClear(&db->aCollSeq);

sqliteHashData(i) 返回哈希表中当前条目的数据，在这里是一个 CollSeq 结构体。pColl 指向该 CollSeq，用于后续的清理操作。

CollSeq 对象可能包含多种比较模式。在 SQLite 中，字符串比较可以分为不同的形式，例如按二进制（binary）、按文本（text）、或者按某些自定义的比较规则进行比较。因此每个 CollSeq 结构体可能包含三种不同的比较函数（pColl[0], pColl[1], pColl[2]）。

pColl[j].xDel：这是一个函数指针，用来释放与该比较规则相关的资源（如果有的话）。只有当这个函数指针存在时，才会调用它来释放 pColl[j].pUser 中的用户数据。

释放与 pColl 相关的内存

10.清理虚拟表模块

#ifndef SQLITE\_OMIT\_VIRTUALTABLE

for(i=sqliteHashFirst(&db->aModule); i; i=sqliteHashNext(i)){

Module \*pMod = (Module \*)sqliteHashData(i);

sqlite3VtabEponymousTableClear(db, pMod);

sqlite3VtabModuleUnref(db, pMod);

}

sqlite3HashClear(&db->aModule);

#endif  
11.清理错误消息和扩展模块

sqlite3Error(db, SQLITE\_OK); 重置数据库的错误状态并释放错误信息

sqlite3ValueFree(db->pErr); 释放错误值对象

sqlite3CloseExtensions(db); 关闭与数据库连接相关的所有扩展模块

12. 释放用户认证用户名和密码占用的内存

#if SQLITE\_USER\_AUTHENTICATION

sqlite3\_free(db->auth.zAuthUser);

sqlite3\_free(db->auth.zAuthPW);

#endif

13. 设置数据库连接状态为错误

db->eOpenState = SQLITE\_STATE\_ERROR;

14. 释放临时数据库模式（Schema）

sqlite3DbFree(db, db->aDb[1].pSchema);

释放与临时数据库模式相关的内存

15. 调用自动清理函数

if( db->xAutovacDestr ){

db->xAutovacDestr(db->pAutovacPagesArg);

}

16.释放互斥锁，设置数据库连接状态为已关闭

sqlite3\_mutex\_leave(db->mutex); 离开当前数据库连接的互斥锁

db->eOpenState = SQLITE\_STATE\_CLOSED;

sqlite3\_mutex\_free(db->mutex); 清理所有与互斥锁相关的资源。

17. 释放 Lookaside 缓存的内存

assert( sqlite3LookasideUsed(db,0)==0 );

if( db->lookaside.bMalloced ){

sqlite3\_free(db->lookaside.pStart);

}

18.释放数据库连接的对象

sqlite3\_free(db);

## sqlite3RollbackAll函数1460

作用：该函数用于对所有数据库文件进行回滚操作。

**1.确保数据库的互斥锁被持有**

assert( sqlite3\_mutex\_held(db->mutex) );

**2.开始良性分配**

sqlite3BeginBenignMalloc();

开启一种特殊的内存分配模式，允许在执行期间分配临时内存，而不影响数据库的内存跟踪。这对于临时计算是安全的。

**3．获取所有B-tree互斥锁**

sqlite3BtreeEnterAll(db);

**4．检查模式变换**

schemaChange = (db->mDbFlags & DBFLAG\_SchemaChange)!=0 && db->init.busy==0;

如果上面的位与操作结果不为零，说明模式变化的标志被设置了。这表示数据库的模式（如表结构）发生了变化。

== 0 表示当前没有其他操作（如连接、准备语句等）在进行，这意味着可以安全地处理模式变化。

整体来说，这行代码的作用是判断当前是否存在数据库模式的变化，并且在没有其他正在进行的初始化操作的情况下，确保可以安全地处理这种变化。这对于数据库的一致性和完整性是至关重要的，因为模式的改变可能影响到现有的数据结构和访问方式。

**5.遍历所有数据库**

for(i=0; i<db->nDb; i++){

Btree \*p = db->aDb[i].pBt;

if( p ){

if( sqlite3BtreeTxnState(p)==SQLITE\_TXN\_WRITE ){

inTrans = 1; 检查当前 B-tree 的事务状态。如果是写事务，则将 inTrans 设置为 1，表示存在活动事务

}

sqlite3BtreeRollback(p, tripCode, !schemaChange); 进行实际的回滚操作。传入 tripCode 以及一个布尔值，指示是否有模式变化。

}

}

**6.回滚虚拟表**

sqlite3VtabRollback(db);

**7.结束良性内存分配**

sqlite3EndBenignMalloc();

**8. 处理模式变化**

if( schemaChange ){

sqlite3ExpirePreparedStatements(db, 0); 失效所有准备好的语句，以确保没有引用被更改的结构。

sqlite3ResetAllSchemasOfConnection(db); 重置连接的所有模式，确保数据库处于一致状态

}

**9.释放B-tree锁**

sqlite3BtreeLeaveAll(db);

**10.清除延迟约束**

db->nDeferredCons = 0;

db->nDeferredImmCons = 0;

db->flags &= ~(u64)(SQLITE\_DeferFKs|SQLITE\_CorruptRdOnly);

将延迟约束计数器重置为 0，并清除相关的标志位，确保数据库的状态干净。

**11.调用回滚回调**

if( db->xRollbackCallback && (inTrans || !db->autoCommit) ){

db->xRollbackCallback(db->pRollbackArg);

}

}

如果设置了回滚回调，并且当前有活动的事务或未处于自动提交状态，则调用该回调。

## sqlite3ErrName函数1509

**作用：**函数的主要作用是将 SQLite 的错误代码转换为对应的错误名称字符串

流程：

## sqlite3ErrStr函数 1622

作用：用于将SQLite返回的错误码转换成人类可读的错误消息字符串

## sqliteDefaultBusyCallback函数 1692

作用：它是SQLite数据库连接在遇到忙状态时的默认回调函数。当一个进程试图访问被另一个进程锁定的数据库表时，就会触发这个回调函数。此函数的主要目的是让当前进程等待一段时间，然后重试操作，直到成功或超时。

## sqlite3InvokeBusyHandler函数 1744

作用：它是 SQLite 数据库库的一部分，用于处理数据库访问时遇到的“忙”状态。在多线程或多进程环境中使用 SQLite 时，可能会出现一个连接尝试读取或写入被另一个连接锁定的数据的情况。这时，SQLite 将调用忙碌处理器（busy handler）来决定如何处理这种情况。

int sqlite3InvokeBusyHandler(BusyHandler \*p){

int rc;

if( p->xBusyHandler==0 || p->nBusy<0 ) return 0;

rc = p->xBusyHandler(p->pBusyArg, p->nBusy);

if( rc==0 ){

p->nBusy = -1;

}else{

p->nBusy++;

}

return rc;

}

## createFunctionApi函数 2000

作用：用于创建或修改用户定义的 SQL 函数

流程：

1.初始化返回码

int rc = SQLITE\_ERROR;

FuncDestructor \*pArg = 0;

2.安全检查

#ifdef SQLITE\_ENABLE\_API\_ARMOR

if( !sqlite3SafetyCheckOk(db) ){

return SQLITE\_MISUSE\_BKPT;

}

#endif

3.加锁

sqlite3\_mutex\_enter(db->mutex);

4.处理用户数据销毁回调

if( xDestroy ){

pArg = (FuncDestructor \*)sqlite3Malloc(sizeof(FuncDestructor));

if( !pArg ){

sqlite3OomFault(db);

xDestroy(p);

goto out;

}

pArg->nRef = 0;

pArg->xDestroy = xDestroy;

pArg->pUserData = p;

}

5.创建函数

rc = sqlite3CreateFunc(db, zFunc, nArg, enc, p,

xSFunc, xStep, xFinal, xValue, xInverse, pArg

);

6.清理未使用的用户数据

if( pArg && pArg->nRef==0 ){

assert( rc!=SQLITE\_OK || (xStep==0 && xFinal==0) );

xDestroy(p);

sqlite3\_free(pArg);

}

如果 pArg 不为 NULL 且引用计数 nRef 为 0，说明 sqlite3CreateFunc 没有成功创建或修改函数

7.解锁并返回

out:

rc = sqlite3ApiExit(db, rc);

sqlite3\_mutex\_leave(db->mutex);

return rc;

## sqlite3InvalidFunction函数 2131

作用：该函数总是返回一个错误消息，指出该函数在不正确的上下文中被使用。

## sqlite3\_trace函数2190

用于注册一个跟踪（trace）回调函数。这个回调函数会在每个 SQL 语句开始执行时被调用，从而允许你记录或调试这些 SQL 语句

## sqlite3WalDefaultHook函数2404

展示了如何实现一个写前日志（Write-Ahead Logging, WAL）的默认钩子（hook）。这个钩子函数会在每次写入 WAL 日志时被调用，并且可以用来触发一些自定义的行为。

流程：

**1.检查是否启用了WAL**

){

#ifdef SQLITE\_OMIT\_WAL

return SQLITE\_OK;

#else

**2.安全检查**

#ifdef SQLITE\_ENABLE\_API\_ARMOR

if( !sqlite3SafetyCheckOk(db) ) return SQLITE\_MISUSE\_BKPT;

#endif

**3.检查检查点模式的有效性**

assert( SQLITE\_CHECKPOINT\_PASSIVE==0 );

assert( SQLITE\_CHECKPOINT\_FULL==1 );

assert( SQLITE\_CHECKPOINT\_RESTART==2 );

assert( SQLITE\_CHECKPOINT\_TRUNCATE==3 );

if( eMode<SQLITE\_CHECKPOINT\_PASSIVE || eMode>SQLITE\_CHECKPOINT\_TRUNCATE ){

/\* EVIDENCE-OF: R-03996-12088 The M parameter must be a valid checkpoint

\*\* mode: \*/

return SQLITE\_MISUSE\_BKPT;

**4．查找数据库**

if( zDb && zDb[0] ){

iDb = sqlite3FindDbName(db, zDb);

}else{

iDb = SQLITE\_MAX\_DB; /\* This means process all schemas \*/

}

if( iDb<0 ){

rc = SQLITE\_ERROR;

sqlite3ErrorWithMsg(db, SQLITE\_ERROR, "unknown database: %s", zDb);

}else{

db->busyHandler.nBusy = 0;

rc = sqlite3Checkpoint(db, iDb, eMode, pnLog, pnCkpt);

sqlite3Error(db, rc);

}

**5.清理并释放互斥锁**

rc = sqlite3ApiExit(db, rc);

/\* If there are no active statements, clear the interrupt flag at this

\*\* point. \*/

if( db->nVdbeActive==0 ){

AtomicStore(&db->u1.isInterrupted, 0);

}

sqlite3\_mutex\_leave(db->mutex);

return rc;

#endif

}

## createCollation函数2754

于在数据库中注册一个新的排序函数。此外，还包含了一些关于 SQLite 限制值的硬性上限定义和检查。

## sqlite3\_limit函数2894

用于更改 SQLite 数据库连接的某些限制值，并返回旧的限制值。这个函数允许你设置数据库的各种运行时限制，例如 SQL 语句的最大长度、列的最大数量等.

流程：

**1.安全检查**

fdef SQLITE\_ENABLE\_API\_ARMOR

if( !sqlite3SafetyCheckOk(db) ){

(void)SQLITE\_MISUSE\_BKPT;

return -1;

}

#endif

**2．确保每个限制标识符的硬性上限与编译时定义的常量一致**

**3.** **检查 limitId 的有效性**

if( limitId<0 || limitId>=SQLITE\_N\_LIMIT ){

return -1;

}

4. **获取旧的限制值**

oldLimit = db->aLimit[limitId];

5. **设置新的限制值**

if( newLimit>=0 ){ /\* IMP: R-52476-28732 \*/

if( newLimit>aHardLimit[limitId] ){

newLimit = aHardLimit[limitId]; /\* IMP: R-51463-25634 \*/

}else if( newLimit<1 && limitId==SQLITE\_LIMIT\_LENGTH ){

newLimit = 1;

}

db->aLimit[limitId] = newLimit;

}

## sqlite3ParseUri函数 2966

用于解析一个给定的 URI，并从中提取文件名、VFS（虚拟文件系统）和其他参数。

## openDatabase函数 3221

作用：打开或创建数据库连接

流程：

**1.api防护**

ifdef SQLITE\_ENABLE\_API\_ARMOR

if( ppDb==0 ) return SQLITE\_MISUSE\_BKPT;

#endif

检查 ppDb 是否为空。ppDb 是一个指向 SQLite 数据库连接的指针的指针，如果为空，返回 SQLITE\_MISUSE\_BKPT，表示 API 被错误地调用

**2.数据库初始化**

\*ppDb = 0;

#ifndef SQLITE\_OMIT\_AUTOINIT

rc = sqlite3\_initialize();

if( rc ) return rc;

**3.线性安全性检查**

if( sqlite3GlobalConfig.bCoreMutex==0 ){ 检查核心互斥锁是否启用

isThreadsafe = 0;

}else if( flags & SQLITE\_OPEN\_NOMUTEX ){

isThreadsafe = 0;

}else if( flags & SQLITE\_OPEN\_FULLMUTEX ){

isThreadsafe = 1;

}else{

isThreadsafe = sqlite3GlobalConfig.bFullMutex;

}

**4. 共享缓存标志处理**

if( flags & SQLITE\_OPEN\_PRIVATECACHE ){

flags &= ~SQLITE\_OPEN\_SHAREDCACHE;

}else if( sqlite3GlobalConfig.sharedCacheEnabled ){

flags |= SQLITE\_OPEN\_SHAREDCACHE;

}

如果请求使用私有缓存，则禁用共享缓存；否则，如果全局配置启用了共享缓存，则启用共享缓存标志。

**5.无效标志处理**

flags &= ~( SQLITE\_OPEN\_DELETEONCLOSE |

SQLITE\_OPEN\_EXCLUSIVE |

SQLITE\_OPEN\_MAIN\_DB |

SQLITE\_OPEN\_TEMP\_DB |

SQLITE\_OPEN\_TRANSIENT\_DB |

SQLITE\_OPEN\_MAIN\_JOURNAL |

SQLITE\_OPEN\_TEMP\_JOURNAL |

SQLITE\_OPEN\_SUBJOURNAL |

SQLITE\_OPEN\_SUPER\_JOURNAL |

SQLITE\_OPEN\_NOMUTEX |

SQLITE\_OPEN\_FULLMUTEX |

SQLITE\_OPEN\_WAL

);

**6. 数据库结构内存分配**

db = sqlite3MallocZero( sizeof(sqlite3) );

if( db==0 ) goto opendb\_out;

分配 sqlite3 结构的内存并将其初始化为零。如果分配失败，跳转到 opendb\_out 进行清理

**7.互斥锁分配**

if( isThreadsafe

#ifdef SQLITE\_ENABLE\_MULTITHREADED\_CHECKS

|| sqlite3GlobalConfig.bCoreMutex

#endif

){

db->mutex = sqlite3MutexAlloc(SQLITE\_MUTEX\_RECURSIVE);

if( db->mutex==0 ){

sqlite3\_free(db);

db = 0;

goto opendb\_out;

}

if( isThreadsafe==0 ){

sqlite3MutexWarnOnContention(db->mutex);

}

}

如果需要线程安全，分配一个递归互斥锁。如果互斥锁分配失败，释放数据库结构的内存并跳转到清理部分。若不是线程安全，调用 sqlite3MutexWarnOnContention 函数以警告潜在的竞争条件。

**8. 初始化数据库实例**

sqlite3\_mutex\_enter(db->mutex);

db->errMask = (flags & SQLITE\_OPEN\_EXRESCODE)!=0 ? 0xffffffff : 0xff;

db->nDb = 2;

db->eOpenState = SQLITE\_STATE\_BUSY;

db->aDb = db->aDbStatic;

db->lookaside.bDisable = 1;

db->lookaside.sz = 0;

进入互斥锁，设置错误掩码、数据库数量、打开状态和其他属性。初始化 lookaside 结构用于快速内存分配

**9.复制限制**

assert( sizeof(db->aLimit)==sizeof(aHardLimit) );

memcpy(db->aLimit, aHardLimit, sizeof(db->aLimit));

确保限制数组的大小与硬限制相同，并将硬限制复制到数据库的限制数组中

**10.默认值设置**

db->aLimit[SQLITE\_LIMIT\_WORKER\_THREADS] = SQLITE\_DEFAULT\_WORKER\_THREADS;

db->autoCommit = 1;

db->nextAutovac = -1;

db->szMmap = sqlite3GlobalConfig.szMmap;

db->nextPagesize = 0;

db->init.azInit = sqlite3StdType; /\* Any array of string ptrs will do \*/

设置一些默认值，包括工作线程的限制、自动提交状态、下一个自动清理时间、内存映射和页面大小。

**11. 排序内存映射设置**

#ifdef SQLITE\_ENABLE\_SORTER\_MMAP

db->nMaxSorterMmap = 0x7FFFFFFF;

#endif

**12.设置数据库标志**

db->flags |= SQLITE\_ShortColNames

| SQLITE\_EnableTrigger

| SQLITE\_EnableView

| SQLITE\_CacheSpill

#if !defined(SQLITE\_TRUSTED\_SCHEMA) || SQLITE\_TRUSTED\_SCHEMA+0!=0

| SQLITE\_TrustedSchema

#endif

#if !defined(SQLITE\_DQS)

# define SQLITE\_DQS 3

#endif

#if (SQLITE\_DQS&1)==1

| SQLITE\_DqsDML

#endif

#if (SQLITE\_DQS&2)==2

| SQLITE\_DqsDDL

#endif

#if !defined(SQLITE\_DEFAULT\_AUTOMATIC\_INDEX) || SQLITE\_DEFAULT\_AUTOMATIC\_INDEX

| SQLITE\_AutoIndex

#endif

#if SQLITE\_DEFAULT\_CKPTFULLFSYNC

| SQLITE\_CkptFullFSync

#endif

#if SQLITE\_DEFAULT\_FILE\_FORMAT<4

| SQLITE\_LegacyFileFmt

#endif

#ifdef SQLITE\_ENABLE\_LOAD\_EXTENSION

| SQLITE\_LoadExtension

#endif

#if SQLITE\_DEFAULT\_RECURSIVE\_TRIGGERS

| SQLITE\_RecTriggers

#endif

#if defined(SQLITE\_DEFAULT\_FOREIGN\_KEYS) && SQLITE\_DEFAULT\_FOREIGN\_KEYS

| SQLITE\_ForeignKeys

#endif

#if defined(SQLITE\_REVERSE\_UNORDERED\_SELECTS)

| SQLITE\_ReverseOrder

#endif

#if defined(SQLITE\_ENABLE\_OVERSIZE\_CELL\_CHECK)

| SQLITE\_CellSizeCk

#endif

#if defined(SQLITE\_ENABLE\_FTS3\_TOKENIZER)

| SQLITE\_Fts3Tokenizer

#endif

#if defined(SQLITE\_ENABLE\_QPSG)

| SQLITE\_EnableQPSG

#endif

#if defined(SQLITE\_DEFAULT\_DEFENSIVE)

| SQLITE\_Defensive

#endif

#if defined(SQLITE\_DEFAULT\_LEGACY\_ALTER\_TABLE)

| SQLITE\_LegacyAlter

#endif

#if defined(SQLITE\_ENABLE\_STMT\_SCANSTATUS)

| SQLITE\_StmtScanStatus

#endif

;

sqlite3HashInit(&db->aCollSeq);

#ifndef SQLITE\_OMIT\_VIRTUALTABLE

sqlite3HashInit(&db->aModule);

#endif

根据编译时选项和默认设置启用相应的数据库标志

**13.添加默认排序序列**

createCollation(db, sqlite3StrBINARY, SQLITE\_UTF8, 0, binCollFunc, 0);

createCollation(db, sqlite3StrBINARY, SQLITE\_UTF16BE, 0, binCollFunc, 0);

createCollation(db, sqlite3StrBINARY, SQLITE\_UTF16LE, 0, binCollFunc, 0);

createCollation(db, "NOCASE", SQLITE\_UTF8, 0, nocaseCollatingFunc, 0);

createCollation(db, "RTRIM", SQLITE\_UTF8, 0, rtrimCollFunc, 0);

添加内置的排序规则，包括 BINARY、NOCASE 和 RTRIM 等。

**14. 检查内存分配是否失败**

if( db->mallocFailed ){

goto opendb\_out;

}

如果在创建排序规则时发生内存分配失败，跳转到清理部分。

**15. 处理特殊文件名（Unix）**

#if SQLITE\_OS\_UNIX && defined(SQLITE\_OS\_KV\_OPTIONAL)

/\* Process magic filenames ":localStorage:" and ":sessionStorage:" \*/

if( zFilename && zFilename[0]==':' ){

if( strcmp(zFilename, ":localStorage:")==0 ){

zFilename = "file:local?vfs=kvvfs";

flags |= SQLITE\_OPEN\_URI;

}else if( strcmp(zFilename, ":sessionStorage:")==0 ){

zFilename = "file:session?vfs=kvvfs";

flags |= SQLITE\_OPEN\_URI;

}

}

#endif

**16. 解析 URI**

db->openFlags = flags;

assert( SQLITE\_OPEN\_READONLY == 0x01 );

assert( SQLITE\_OPEN\_READWRITE == 0x02 );

assert( SQLITE\_OPEN\_CREATE == 0x04 );

testcase( (1<<(flags&7))==0x02 ); /\* READONLY \*/

testcase( (1<<(flags&7))==0x04 ); /\* READWRITE \*/

testcase( (1<<(flags&7))==0x40 ); /\* READWRITE | CREATE \*/

if( ((1<<(flags&7)) & 0x46)==0 ){

rc = SQLITE\_MISUSE\_BKPT; /\* IMP: R-18321-05872 \*/

}else{

if( zFilename==0 ) zFilename = ":memory:";

rc = sqlite3ParseUri(zVfs, zFilename, &flags, &db->pVfs, &zOpen, &zErrMsg);

}

设置打开标志并进行一些有效性测试。如果标志的组合不合理，返回 SQLITE\_MISUSE\_BKPT。否则，解析 URI，获取有效的文件名。

**17.错误处理**

if( rc!=SQLITE\_OK ){

if( rc==SQLITE\_NOMEM ) sqlite3OomFault(db);

sqlite3ErrorWithMsg(db, rc, zErrMsg ? "%s" : 0, zErrMsg);

sqlite3\_free(zErrMsg);

goto opendb\_out;

}如果 URI 解析返回错误，进行相应处理并清理内存。

**18. 打开后端数据库驱动**

rc = sqlite3BtreeOpen(db->pVfs, zOpen, db, &db->aDb[0].pBt, 0,

flags | SQLITE\_OPEN\_MAIN\_DB);

if( rc!=SQLITE\_OK ){

if( rc==SQLITE\_IOERR\_NOMEM ){

rc = SQLITE\_NOMEM\_BKPT;

}

sqlite3Error(db, rc);

goto opendb\_out;

}

调用 sqlite3BtreeOpen 打开后端数据库。如果出错，进行错误处理。

**19. 进入和离开 Btree**

sqlite3BtreeEnter(db->aDb[0].pBt);

db->aDb[0].pSchema = sqlite3SchemaGet(db, db->aDb[0].pBt);

if( !db->mallocFailed ){

sqlite3SetTextEncoding(db, SCHEMA\_ENC(db));

}

sqlite3BtreeLeave(db->aDb[0].pBt);

进入 Btree，获取模式并设置文本编码，然后离开 Btree

**20. 设置数据库名称和安全级别**

db->aDb[0].zDbSName = "main";

db->aDb[0].safety\_level = SQLITE\_DEFAULT\_SYNCHRONOUS+1;

db->aDb[1].zDbSName = "temp";

db->aDb[1].safety\_level = PAGER\_SYNCHRONOUS\_OFF;

**21. 注册内置函数**

sqlite3Error(db, SQLITE\_OK);

sqlite3RegisterPerConnectionBuiltinFunctions(db);

rc = sqlite3\_errcode(db);

注册所有内置函数，确保数据库准备好接受 SQL 命令。

**22. 加载编译的扩展**

for(i=0; rc==SQLITE\_OK && i<ArraySize(sqlite3BuiltinExtensions); i++){

rc = sqlite3BuiltinExtensions[i](db);

}

加载所有已编译的扩展，直到遇到错误。

**23. 加载自动扩展**

if( rc==SQLITE\_OK ){

sqlite3AutoLoadExtensions(db);

rc = sqlite3\_errcode(db);

if( rc!=SQLITE\_OK ){

goto opendb\_out;

}

}

如果没有错误，加载自动扩展，并检查错误。

**24．设置默认锁定模式**

#ifdef SQLITE\_DEFAULT\_LOCKING\_MODE

db->dfltLockMode = SQLITE\_DEFAULT\_LOCKING\_MODE;

sqlite3PagerLockingMode(sqlite3BtreePager(db->aDb[0].pBt),

SQLITE\_DEFAULT\_LOCKING\_MODE);

#endif

根据编译时配置设置默认的锁定模式

**25. 进行错误处理**

if( rc ) sqlite3Error(db, rc);

如果有错误发生，记录错误信息

**26. 启用 lookaside 内存分配**

setupLookaside(db, 0, sqlite3GlobalConfig.szLookaside,

sqlite3GlobalConfig.nLookaside);

**27. 自动检查点设置**

sqlite3\_wal\_autocheckpoint(db, SQLITE\_DEFAULT\_WAL\_AUTOCHECKPOINT);

**28. 清理和返回**

opendb\_out:

if( db ){

assert( db->mutex!=0 || isThreadsafe==0

|| sqlite3GlobalConfig.bFullMutex==0 );

sqlite3\_mutex\_leave(db->mutex);

}

rc = sqlite3\_errcode(db);

assert( db!=0 || (rc&0xff)==SQLITE\_NOMEM );

if( (rc&0xff)==SQLITE\_NOMEM ){

sqlite3\_close(db);

db = 0;

}else if( rc!=SQLITE\_OK ){

db->eOpenState = SQLITE\_STATE\_SICK;

}

\*ppDb = db;

清理阶段，确保释放锁并设置返回的数据库句柄。如果分配失败，关闭数据库并清空指针。

**29．SQL 日志记录**

ifdef SQLITE\_ENABLE\_SQLLOG

if( sqlite3GlobalConfig.xSqllog ){

/\* Opening a db handle. Fourth parameter is passed 0. \*/

void \*pArg = sqlite3GlobalConfig.pSqllogArg;

sqlite3GlobalConfig.xSqllog(pArg, db, zFilename, 0);

}

#endif

如果启用了 SQL 日志记录，记录打开数据库的事件

**30.释放内存并返回**

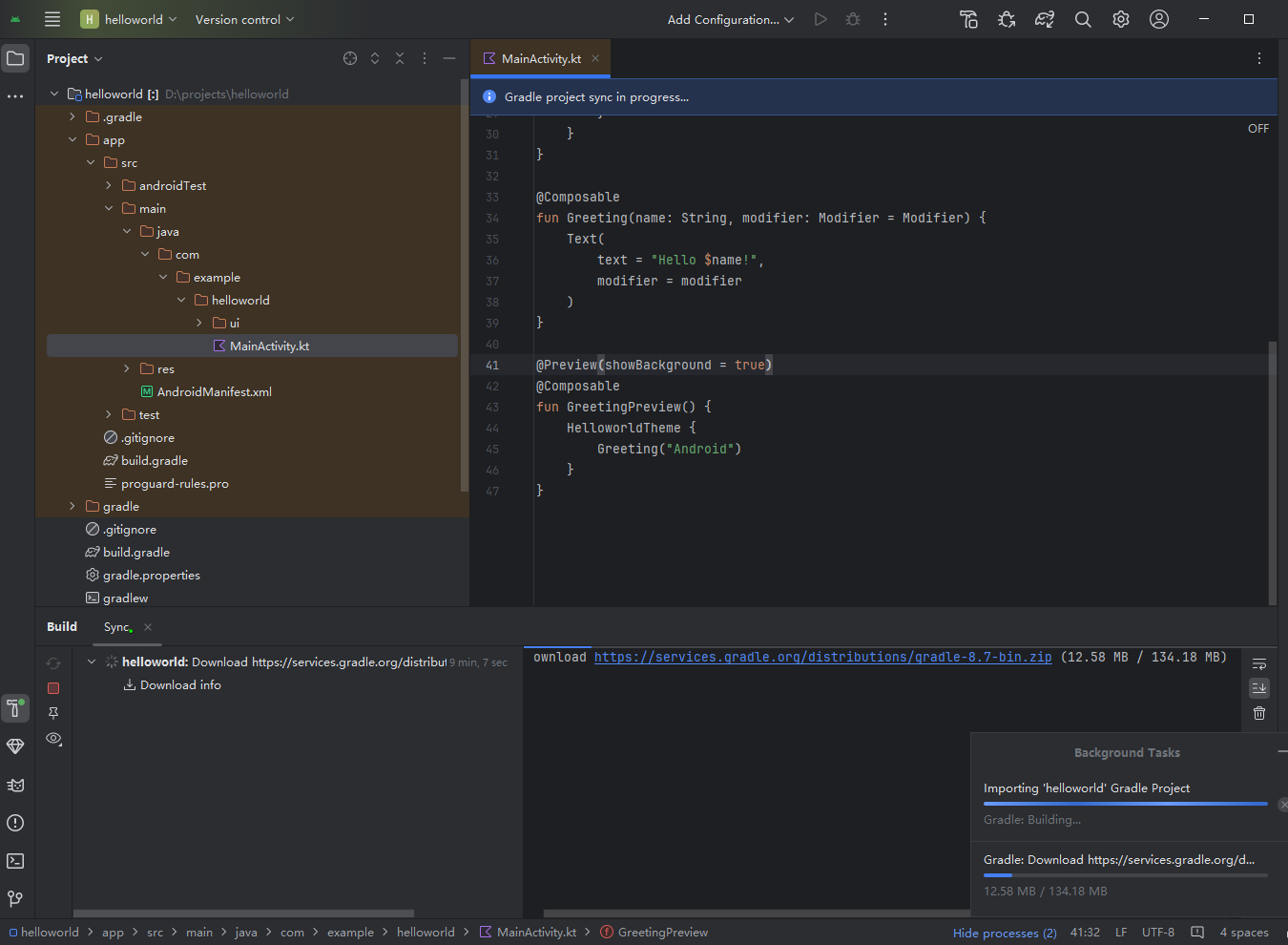
sqlite3\_free\_filename(zOpen);

return rc;

最后，释放内存并返回函数的返回码

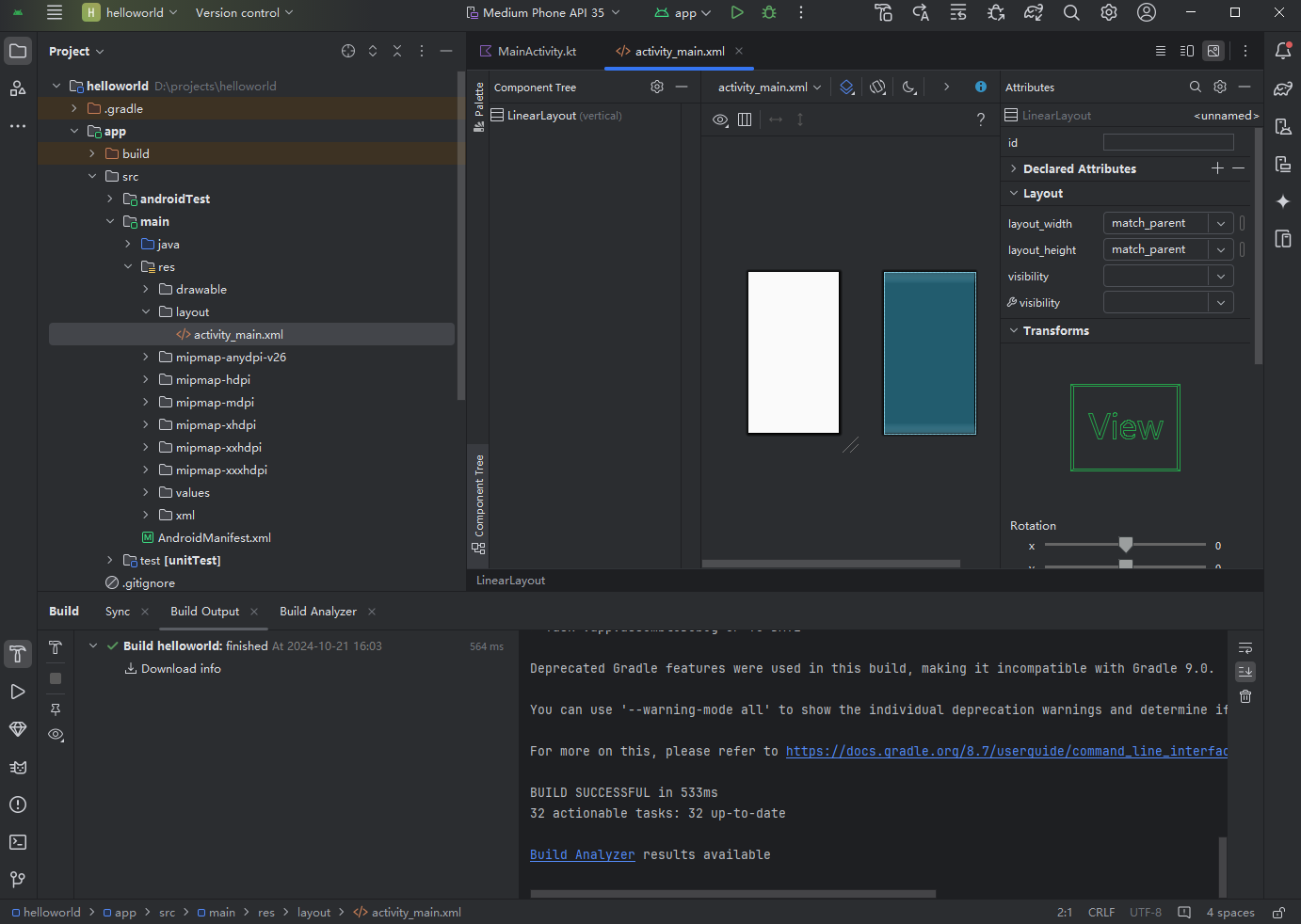
# 二、Andriod studio 运行hello world app

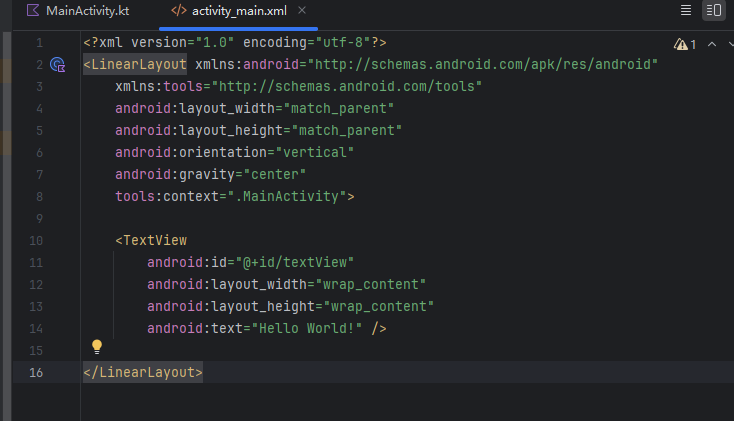
搭建完项目之后，等待building

A

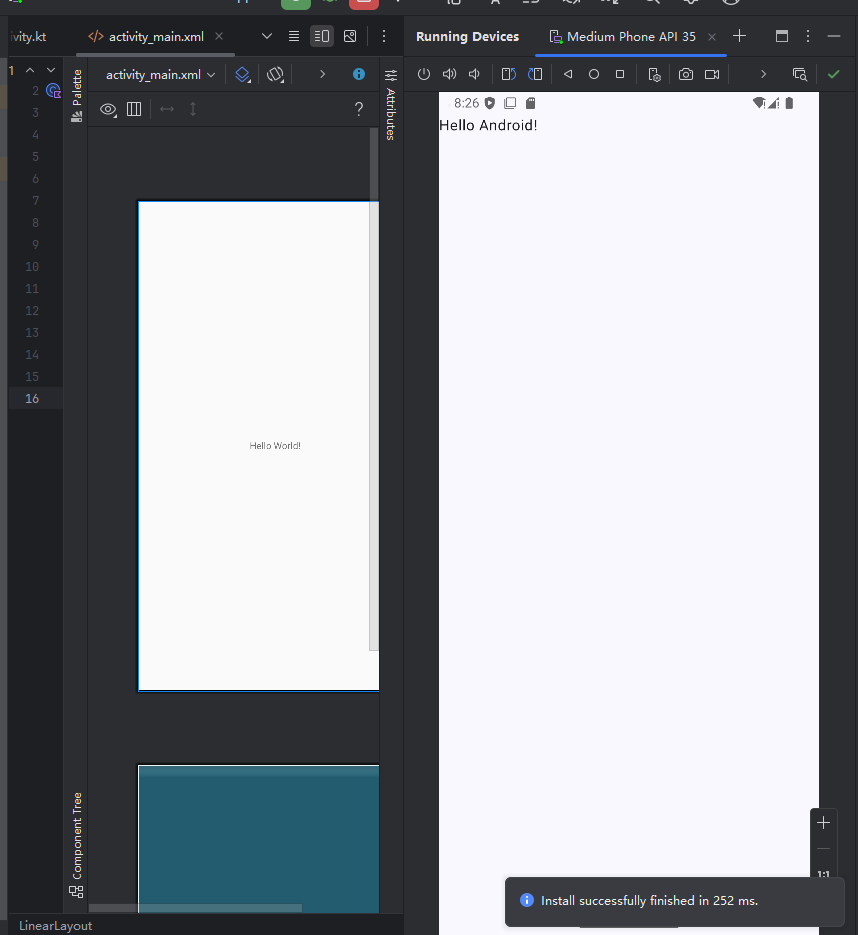
有关博客的内容为在layout文件中修改testview组件来达到hello world的结果

但我搭建完后在res下没有发现layout，尝试手动创建layout以及activity\_main.xml文件，添加如下代码：

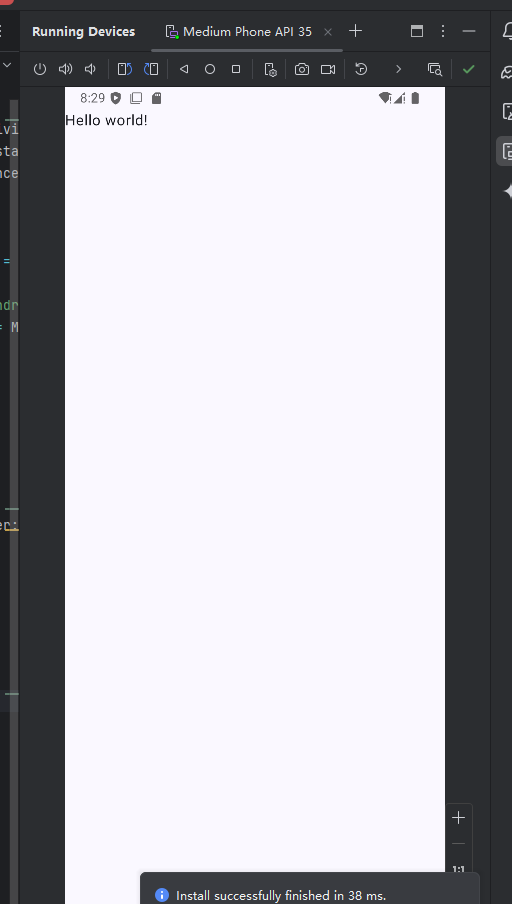
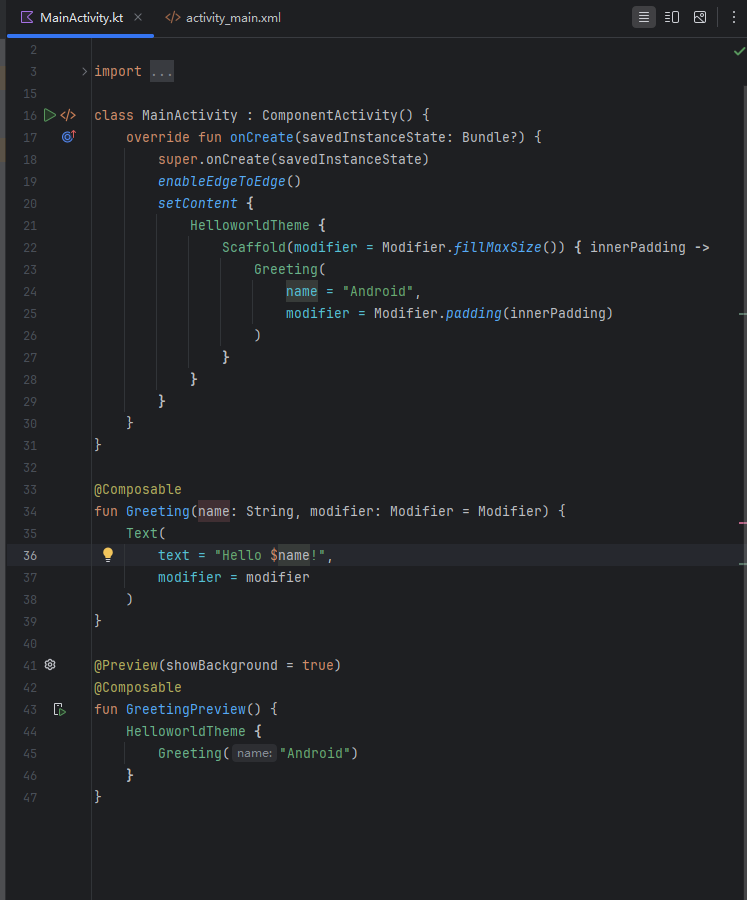




运行结果如下，发现并没有成功运行hello world，而是hello android

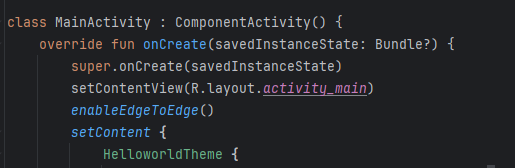


可能是因为运行时没有调用这个布局文件，且在mainactivity.kt文件中，发现了有hello android相关的代码，我修改代码为world，再次运行结果为hello world！。

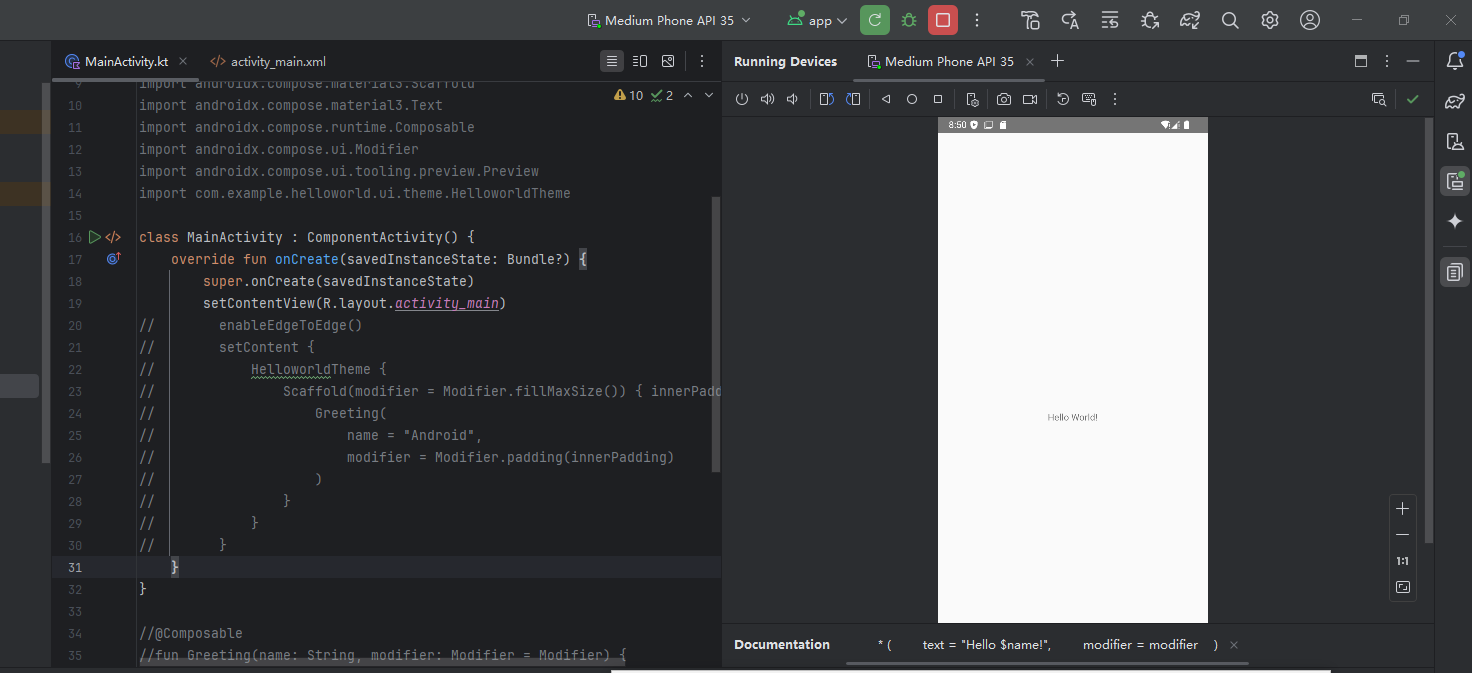


但与相关博客中修改activity\_main.xml中内容来达成hello world不符，于是尝试修改mainactivity.kt文件中的代码来调用activity\_main.xml文件

添加setContentView（R.layout.activity\_main）如下所示：



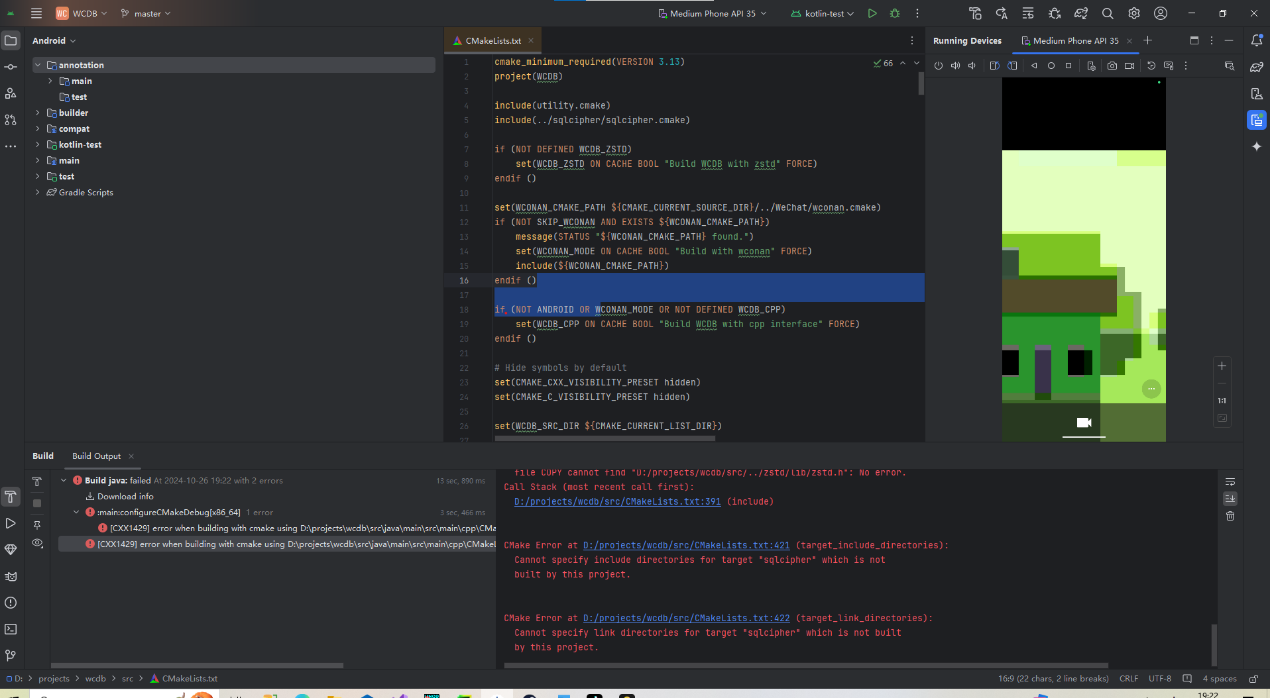
运行之后结果仍然不调用该布局文件，后结合gpt回答发现是因为文件中混合了传统的 XML 布局和 Jetpack Compose。setContentView(R.layout.activity\_main) 和 setContent 不能同时使用，如果要使用传统的 XML 布局，需要移除 setContent 相关的代码。于是我尝试注释掉setContent相关代码，可得到正确的运行结果：



总结：新版本的andriod studio中empty activity默认使用jetpack Compose布局，不会生成layout文件，与相关博客的旧版本内容不符。

# 三、Andriod studio 运行wcdb kotlin-test模块

尝试很多次后项目完成搭建，运行kotlin-test后，结果如下：



原因：拉取wcdb源码时未运行git submodule update --init sqlcipher zstd

重新拉取源码后再次debug kotlin-test，结果如下：

