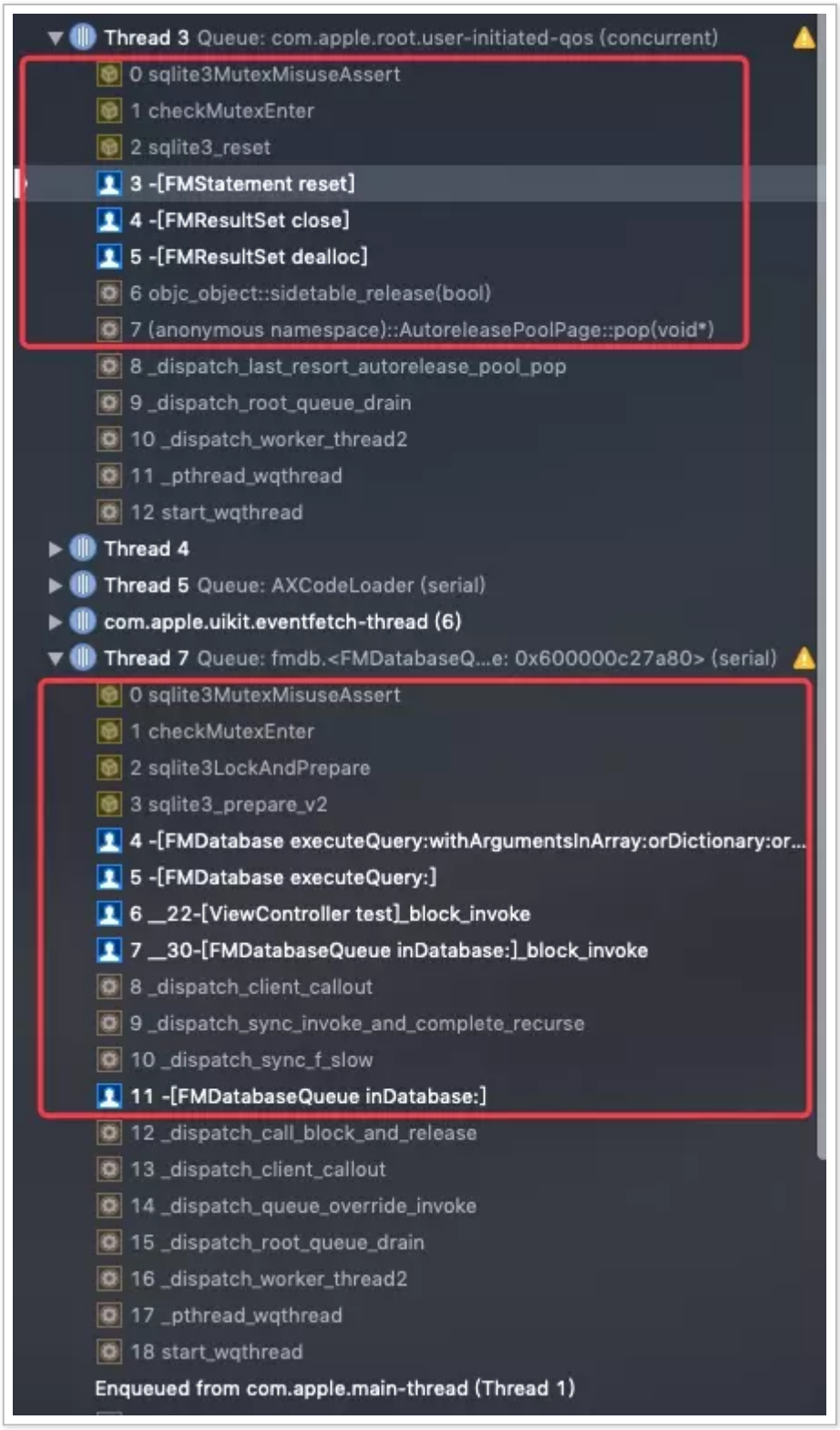
从 FMDB 线程安全问题说起

> 本文讨论的 FMDB 版本为 `2.7.5` ，测试环境是 `Xcode 10.1 & iOS 12.1` 。

一、问题记录

最近在分析崩溃日志的时候发现一个 FMDB 的 crash 频繁出现，crash 堆栈如下：



在控制台能看到报错：

```
[logging] BUG IN CLIENT OF sqlite3.dylib: illegal multi-threaded access to dat
Warning: there is at least one open result set around after performing [FMData
```

复制代码

从日志中能大概猜到，这是多线程访问数据库导致的 crash。FMDB 提供了 `FMDatabaseQueue` 在多线程环境下操作数据库，它内部维护了一个串行队列

来保证线程安全。我检查了所有操作数据库的代码，都是在 `FMDatabaseQueue` 队列里执行的，为啥还是会报多线程问题（一脸懵逼🤔）？

在网上找了一圈，发现 github 上有人遇到了同样的问题， Issue 724 和 Issue 711，Stack Overflow 上有相关的 讨论 。

项目里业务太复杂，很难排查问题，于是写了一个简化版的 Demo 来复现问题：

```
    NSString *dbPath = [docPath stringByAppendingPathComponent:@"test.sqlite"]
    _queue = [FMDatabaseQueue databaseQueueWithPath:dbPath];

    // 构建测试数据，新建一个表test，inert一些数据
    [_queue inDatabase:^(FMDatabase * _Nonnull db) {
        [db executeUpdate:@"create table if not exists test (a text, b text, c
        for (int i = 0; i < 10000; i++) {
            [db executeUpdate:@"insert into test (a, b, c, d, e, f, g, h, i) v
        }
    }];

    // 多线程查询数据库
    for (int i = 0; i < 10; i++) {
        dispatch_async(dispatch_get_global_queue(0, 0), ^{
            [_queue inDatabase:^(FMDatabase * _Nonnull db) {
                FMResultSet *result = [db executeQuery:@"select * from test wh
                // 这里要用if，改成while就没问题了
                if ([result next]) {
                }
                // 这里不调用close
//                [result close];
            }];
        });
    }
```

复制代码

问题完美复现，接下来就可以排查问题了，有两个问题亟待解决：

- iOS 系统自带的 SQLite 究竟是不是线程安全的?
- 为什么使用了线程安全队列 `FMDatabaseQueue` ， 还是出现了线程安全问题?

二、SQLite 线程安全

我们先来看第一个问题，iOS 系统自带的 SQLite 究竟是不是线程安全的?

Google 了一下，发现了关于 SQLite 的 官方文档 – Using SQLite In Multi–Threaded Applications 。文档写的很清晰，有时间最好认真读读，这里简单总结一下。

SQLite 有 3 种线程模式：

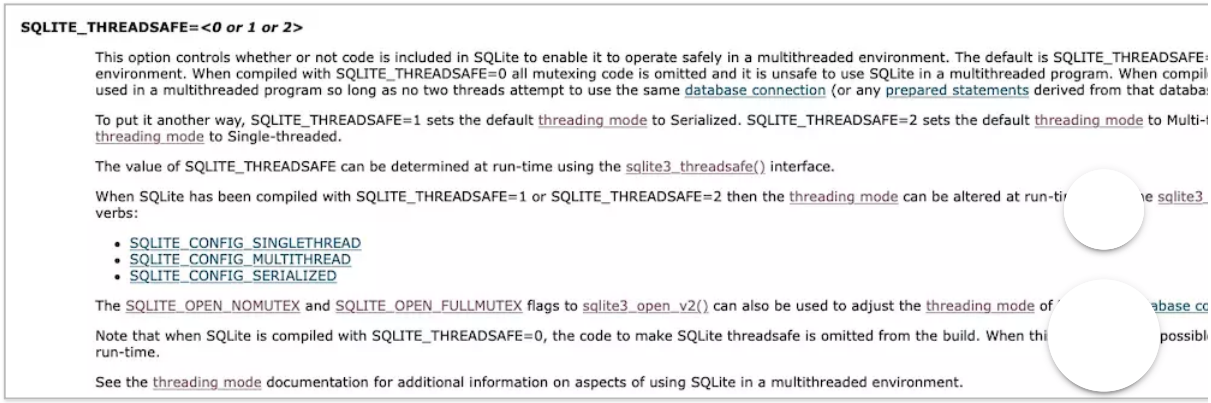
- Single–thread，单线程模式，编译时所有互斥锁代码会被删除掉，多线程环境下不安全。
- Multi–thread，在大部分情况下多线程环境安全，比如同一个数据库，开多个线程，每个线程都开一个连接同时访问这个库，这种情况是安全的。但是也有不安全情况：多个线程同时使用同一个数据库连接（或从该连接派生的任何预准备语句）
- Serialized，完全线程安全。

有 3 个时间点可以配置 threading mode，编译时（compile–time）、初始化时（start–time）、运行时（run–time）。配置生效规则是 run–time 覆盖 start–time 覆盖 compile–time，有一些特殊情况：

- 编译时设置 `Single-thread` ，用户就不能再开启多线程模式，因为线程安全代码被优化了。
- 如果编译时设置的多线程模式，在运行时不能降级为单线程模式，只能在 `Multi-thread` 和 `Serialized` 间切换。

threading mode 编译选项

SQLite threading mode 编译选项的 官方文档



SQLITE_THREADSAFE=*<0 or 1 or 2>*

This option controls whether or not code is included in SQLite to enable it to operate safely in a multithreaded environment. The default is SQLITE_THREADSAFE= environment. When compiled with SQLITE_THREADSAFE=0 all mutexing code is omitted and it is unsafe to use SQLite in a multithreaded program. When compil used in a multithreaded program so long as no two threads attempt to use the same database connection (or any prepared statements derived from that databa

To put it another way, SQLITE_THREADSAFE=1 sets the default threading mode to Serialized. SQLITE_THREADSAFE=2 sets the default threading mode to Multi- threading mode to Single-threaded.

The value of SQLITE_THREADSAFE can be determined at run-time using the sqlite3_threadsafe() interface.

When SQLite has been compiled with SQLITE_THREADSAFE=1 or SQLITE_THREADSAFE=2 then the threading mode can be altered at run-ti e sqlite3_ verbs:

- SQLITE_CONFIG_SINGLETHREAD
- SQLITE_CONFIG_MULTITHREAD
- SQLITE_CONFIG_SERIALIZED

The SQLITE_OPEN_NOMUTEX and SQLITE_OPEN_FULLMUTEX flags to sqlite3_open_v2() can also be used to adjust the threading mode of abase co

Note that when SQLite is compiled with SQLITE_THREADSAFE=0, the code to make SQLite threadsafe is omitted from the build. When thi possibl run-time.

See the threading mode documentation for additional information on aspects of using SQLite in a multithreaded environment.

编译时，通过配置项 `SQLITE_THREADSAFE` 可以配置 SQLite 在多线程环境下是否安全。有三个可选项：

- 0，对应 Single-thread，编译时所有互斥锁代码会被删除掉，SQLite 在多线程环境下不安全。
- 1，对应 Serialized，在多线程环境下安全，如果不手动指定，这是默认选项。
- 2，对应 Multi-thread，在大部分情况下多线程环境安全，不安全情况：有两个线程同时尝试使用相同数据库连接（或从该数据库连接派生的任何预处理语句 Prepared Statements）

除了编译时可以指定 threading mode，还可以通过函数 `sqlite3_config()`（start-time）改变全局的 threading mode 或者通过 `sqlite3_open_v2()`（run-time）改变某个数据库连接的 threading mode。

但是如果编译时配置了 `SQLITE_THREADSAFE = 0`，编译时所有线程安全代码都被优化掉了，就不能再切换到多线程模式了。

有了前面的知识，我们就可以分析问题一了。调用函数 `sqlite3_threadsafe()` 可以获取编译时的配置项，我们可以用这个函数获取系统自带的 SQLite 在编译时的配置，结论是 2（Multi-thread）。

也就是说，系统自带的 SQLite 在不做任何配置的情况下不是完全线程安全的。当然可以手动将模式切换到 `Serialized` 就可以实现完全线程安全了。

```
// 方案一：全局设置模式
sqlite3_config(SQLITE_CONFIG_SERIALIZED);

// 方案二：设置 connecting 模式，调用 sqlite3_open_v2 时 flag 加上 SQLITE_OPEN_FULI
sqlite3_open_v2(path, &db, SQLITE_OPEN_CREATE | SQLITE_OPEN_READWRITE | SQLITE
```

复制代码

经过测试，通过上面两种方案改造之后，Demo 中的 crash 问题完美解决。但是我认为这不是最优的解决方案，苹果为啥不直接将编译选项设置为 `Serialized`，这篇文章就永远不会出现了😂，劳民伤财让大家折腾半天，去手动设置模式。我认为性能是一个重要因素，`Multi-thread` 性能优于 `Serialized`，用户只要保证一个连接不在多线程同时访问就没问题了，其实能满足大部分需求。

比如 FMDB 的 `FMDatabaseQueue` 就是为了解决该问题。

三、FMDatabaseQueue 其实并不安全

FMDB 的官方文档写到：

> FMDatabaseQueue will run the blocks on a serialized queue (hence the name of the class). So if you call FMDatabaseQueue's methods from multiple threads at the same time, they will be executed in the order they are received. This way queries and updates won't step on each other's toes, and every one is happy.

在多线程使用 `FMDatabaseQueue` 的确很安全，通过 GCD 的串行队列来保证所有读写操作都是串行执行的。它的核心代码如下：

```
_queue = dispatch_queue_create([[NSString stringWithFormat:@"fmdb.%@", self] U

- (void)inDatabase:(__attribute__((noescape)) void (^)(FMDatabase *db))block {
    // ...省略部分代码

    dispatch_sync(_queue, ^() {
        FMDatabase *db = [self database];
        block(db);
    });

    // ...省略部分代码
}
```
复制代码

但是分析第一节 Demo 的 crash 堆栈，可以看到崩溃发生在线程 3 的函数 `[FMResultSet reset]`，函数定义如下：

```
- (void)reset {
    if (_statement) {
        // 释放预处理语句（Reset A Prepared Statement Object）
        sqlite3_reset(_statement);
    }
    _inUse = NO;
}
```
复制代码

这个函数的调用栈如下：

```
- [FMStatement reset]
- [FMResultSet close]
- [FMResultSet dealloc]
```
复制代码

顺着调用堆栈，我们来看看 `FMResultSet` 的 `dealloc` 和 `close` 方法：

```
- (void)dealloc {
    [self close];
    FMDBRelease(_query);
    _query = nil;
    FMDBRelease(_columnNameToIndexMap);
    _columnNameToIndexMap = nil;
}

- (void)close {
    [_statement reset];
    FMDBRelease(_statement);
    _statement = nil;
    [_parentDB resultSetDidClose:self];
    [self setParentDB:nil];
}
```
复制代码

这里可以得出结论，在 `FMResultSet dealloc` 时会调用 `close` 方法，来关闭预处理语句。再回到第一节的 crash 堆栈，不难发现线程 7 在用同一个数据库连接读数据库，结合官方文档中的一段话，我们就可以得出结论了。

> When compiled with SQLITE_THREADSAFE=2, SQLite can be used in a multithreaded program so long as no two threads attempt to use the same database connection (or any prepared statements derived from that database connection) at the same time.

使用 `FMDatabaseQueue` 还是发生了多线程使用同一个数据库连接、预处理语句的情况，于是就崩溃了。

解决方案

问题找到了，接下来聊聊怎么避免问题。

FMDB 的正确打开方式

如果用 `while` 循环遍历 `FMResultSet` 就不存在该问题，因为 `[FMResultSet next]` 遍历到最后会调用 `[FMResultSet close]` 。

```
[_queue inDatabase:^(FMDatabase * _Nonnull db) {
    FMResultSet *result = [db executeQuery:@"select * from test where a = '1'"
    // 安全
    while ([result next]) {
    }

    // 安全
    if ([result next]) {
    }
    [result close];
}];
```

复制代码

如果一定要用 `if ([result next])` ，手动加上 `[FMResultSet close]` 也没有问题。

写在最后

写在最后

我遇到这个问题，是被官方文档的一句话误导了。

> Typically, there's no need to -close an FMResultSet yourself, since that happens when either the result set is deallocated, or the parent database is closed.

于是我提了一个 Pull requests ，我提出了两种解决方案：

- 修改文档，在文档中强调，用户需要手动调用 close。
- 在 `[FMDatabaseQueue inDatabase:]` 函数的最后，调用 `[FMDatabase closeOpenResultSets]` 帮助调用者关闭所有 FMResultSet。

FMDB 的作者 `ccgus` 采用了第一种方案，在最新的一次 commit 修改了文档，加上了相关说明。

> Typically, there's no need to -close an FMResultSet yourself, since that happens when either the result set is exhausted. However, if you only pull out a single request or any other number of requests which don't exhaust the result set, you will need to call the -close method on the FMResultSet.

参考

- Using SQLite In Multi-Threaded Applications
- sqlite3.dylib: illegal multi-threaded access to database connection
- FMDB
- SQLite 编译选项官方文档

全文完