WCDB源码学习

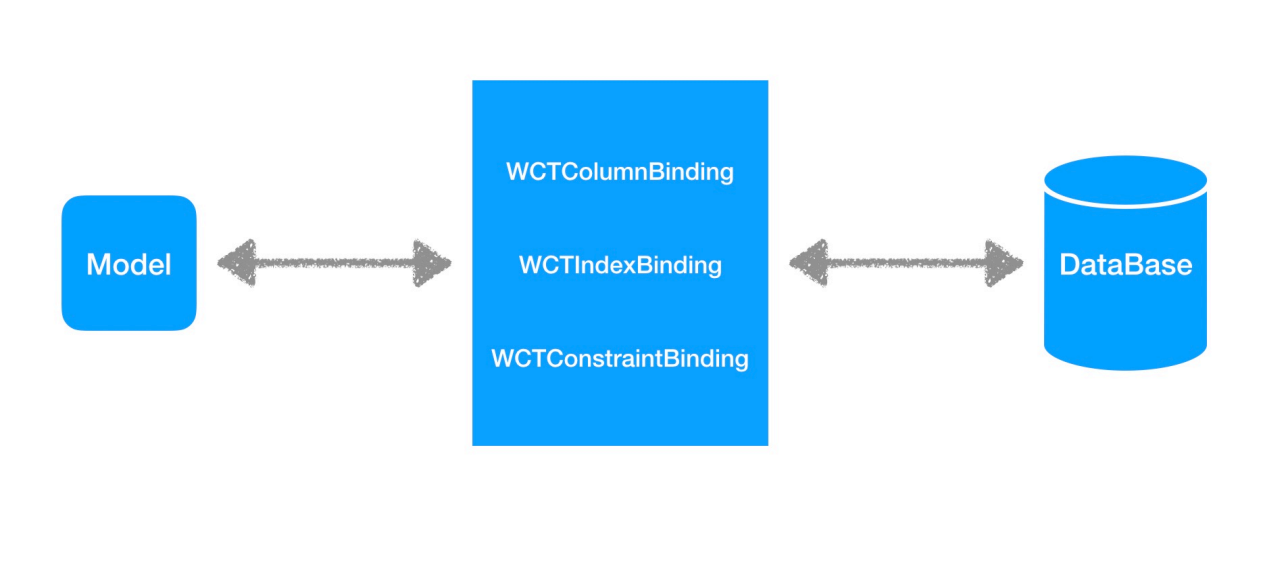
WCDB是微信团队开源的一款基于FMDB优化的Sqlite数据库组件，致力于提供一个易用、高效、完整的移动端储存方案（兼容MacOS）。牛牛行情模块涉及了大量数据的监听和交互，提高数据库的性能，能够更好的改善用户体验。

WCDB优势：

- 易用：相对于FMDB冗长繁杂的胶水代码，WCDB提供了便捷的对象关系映射（ORM，Object Relational Mapping）、CRUD接口，开发者可以便捷的定义表和索引，同时还提供了WINQ，方便开发者操作SQL。
- 高效：多线程高并发，并行执行读与读、读与写，优化串行执行写与写的操作。
- 完整：损坏修复、反注入、统计分析等。

（一）对象关系映射（ORM, Object Relational Mapping）

ORM是将一个对象的类与表和索引关联起来，同时将类的属性映射到数据库表的字段中。通过WCDB可以省去手写拼装类中属性与表字段关联的代码。



- **WCTColumnBinding：定义了类属性与字段之间的联系，支持自定义字段名和默认值**
- WCTIndexBinding：定义了数据库的索引属性，支持定义索引的排序方式
- WCTConstraintBinding：包括字段主键约束和表多主键约束

ORM的简单使用

建表

- 使用表路径初始化 `WCTDatabase` 对象

```
WCTDatabase *dataBase = [[WCTDatabase alloc]
initWithPath:@"/Users/.../stock.db"];
BOOL result = [dataBase createTableAndIndexesOfName:@"stock"
withClass:TZStock.class];
```

2. 通过宏创建属性

- `WCDB_PROPERTY()` 声明属性

```
@interface TZStock : NSObject<WCTTableCoding>

@property(retain) NSString *name;
@property(retain) NSDate *date;
@property(assign) int code;

WCDB_PROPERTY(code);
WCDB_PROPERTY(name);
WCDB_PROPERTY(date);

@end
```

- `WCDB_IMPLEMENTATION` 实现协议 `WCDB_SYNTHESIZE` 合成相关属性

```
@implementation TZStock

WCDB_IMPLEMENTATION(TZStock);

WCDB_SYNTHESIZE(TZStock, code);
WCDB_SYNTHESIZE(TZStock, name);
WCDB_SYNTHESIZE(TZStock, date);

@end
```

3. 数据库操作

```
WCTDatabase *dataBase = [[WCTDatabase alloc]
initWithPath:@"/Users/zoutan/Desktop/stock.db"];
```

```
        BOOL result = [dataBase createTableAndIndexesOfName:@"stock"
withClass:TZStock.class];
        if (result) {
            NSLog(@"建表成功");

            NSArray *stocks = @[@{@"name": @"FLH", @"code": @666},
@{@"name": @"ND", @"code": @777}, @{@"name": @"QT", @"code": @000}];
            [stocks enumerateObjectsUsingBlock:^(NSDictionary *obj,
NSUInteger idx, BOOL * _Nonnull stop) {
                TZStock *stock = [[TZStock alloc] init];
                stock.name = obj[@"name"];
                stock.code = ((NSNumber *)obj[@"code"]).intValue;
                stock.date = [NSDate dateWithTimeIntervalSinceNow:111];
                [dataBase insertObject:stock into:@"stock"];
            }];


            [dataBase deleteObjectsFromTable:@"stock" where:TZStock.code ==
777];

            TZStock *editStock = [[TZStock alloc] init];
            editStock.code = 888;
            [dataBase updateRowsInTable:@"stock" onProperty:TZStock.code
withObject:editStock limit:nil];

            NSArray <TZStock *> *searchStocks = [dataBase
getObjectsOfClass:TZStock.class fromTable:@"stock"
orderBy:TZStock.code.order()];
            NSLog(@"finish");
```

通过第一和第二步，就将类与表关联起来了，同时将属性映射到对应的表字段中。

另附ORM详细使用教程: https://github.com/Tencent/wcdb/wiki/ORM使用教程

ORM实现分析

建表

通过 `tableName` 和 `cls` 建表，且 `cls` 需要遵从 `WCTTableCoding` 协议。

```
- (BOOL)createTableAndIndexesOfName:(NSString *)tableName withClass:
(Class<WCTTableCoding>)cls;
```

绑定属性与表字段

1. 关联表的对象类需要遵从 `WCTTableCoding` 协议，如下所示：

```
@interface TZStock : NSObject<WCTTableCoding>
...
@end

@implementation TZStock
WCDB_IMPLEMENTATION(TZStock);
...
@end
```

且需要实现 `require` 方法， `WCTTableCoding` 协议：

```
@protocol WCTTableCoding
@required
+ (const WCTBinding *)objectRelationalMappingForWCDB;
+ (const WCTPropertyList &)AllProperties;
+ (const WCTAnyProperty &)AnyProperty;
+ (WCTPropertyNamed)PropertyNamed; //className.PropertyNamed(propertyName)
@optional
@property(nonatomic, assign) long long lastInsertedRowID;
@property(nonatomic, assign) BOOL isAutoIncrement;
@end
```

其中， `+ (const WCTBinding *)objectRelationalMappingForWCDB` 是实现ORM的关键，其实现定义在 `WCDB_IMPLEMENTATION()` 宏中，将宏展开：

```
#define __WCDB_BINDING(className) _s_##className##_binding

static WCTBinding __WCDB_BINDING(className)(className.class);
static WCTPropertyList __WCDB_PROPERTIES(className);
+ (const WCTBinding *) objectRelationalMappingForWCDB {
  if (self.class != className.class) {
    WCDB::Error::Abort("Inheritance is not supported for ORM");
  }
  return &__WCDB_BINDING(className);
}
```

宏 `#define __WCDB_BINDING(TZStock) _s_##className##_binding` 可以将TZStock替换至##className##，生成 `_s_TZStock_binding` 。替换后的代码：

```
static WCTBinding _s_TZStock_binding(TZStock.class);
static WCTPropertyList _s_TZStock_properties;
+ (const WCTBinding *) objectRelationalMappingForWCDB {
   if (self.class != TZStock.class) {
     WCDB::Error::Abort("Inheritance is not supported for ORM");
   }
   return &_s_TZStock_binding; //已生成绑定关系
}
```

`objectRelationalMappingForWCDB` 主要做了两件事：

- 判断宏 `WCDB_IMPLEMENTATION(TZStock)` 传进来的类是否是当前类
  - 保证当前类与映射类一致
  - 继承类不能直接继承ORM
- 返回 `_s_TZStock_binding` 静态变量
  - _s_TZStock_binding(TZStock.class) 是一个静态函数
  - `_s_TZStock_binding` 与 `_s_TZStock_binding(TZStock.class)` 地址一致

output：

```
p _s_TZStock_binding
(WCTBinding) $1 = {
  m_cls = TZStock
  m_columnBindingMap = size=3 {
    [0] = {
      first = "code"
      second = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100f002e0 strong=4 weak=1 {
        __ptr_ = 0x0000000100f002e0
      }
    }
    [1] = {
      first = "date"
      second = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000102805e20 strong=4 weak=1 {
        __ptr_ = 0x0000000102805e20
      }
    }
    [2] = {
      first = "name"
      second = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100f16b20 strong=4 weak=1 {
```

```
          __ptr_ = 0x0000000100f16b20
        }
      }
    }
    m_columnBindingList = size=3 {
      [0] = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100f002e0 strong=4 weak=1 {
        __ptr_ = 0x0000000100f002e0
      }
      [1] = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100f16b20 strong=4 weak=1 {
        __ptr_ = 0x0000000100f16b20
      }
      [2] = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000102805e20 strong=4 weak=1 {
        __ptr_ = 0x0000000102805e20
      }
    }
    m_indexBindingMap = nullptr {
      __ptr_ = 0x0000000000000000
    }
    m_constraintBindingMap = nullptr {
      __ptr_ = 0x0000000000000000
    }
    m_constraintBindingList = nullptr {
      __ptr_ = 0x0000000000000000
    }
    m_virtualTableArgumentList = nullptr {
      __ptr_ = 0x0000000000000000
    }
    m_virtualTableModuleName = ""
  }
```

o((⊙﹏⊙))o

接下来我们再来看一下这个绑定的过程。

2. `WCTBinding` 对象包含类与表绑定信息，但此时该对象内绑定关系以及形成，它是通过字段宏 `WCDB_SYNTHESIZE(className, propertyName)` 进行创建与绑定。

```
#define WCDB_SYNTHESIZE(className, propertyName)
__WCDB_SYNTHESIZE_IMP(className, propertyName,
WCDB_STRINGIFY(propertyName))

#define __WCDB_SYNTHESIZE_IMP(className, propertyName, columnName) +(const
WCTProperty &) propertyName {...}

#define _WCDB_STRINGIFY(str) #str
#define WCDB_STRINGIFY(str) _WCDB_STRINGIFY(str)
#define UNUSED_UNIQUE_ID CONCAT(_unused, __COUNTER__)
```

以 `WCDB_SYNTHESIZE(TZStock, code)` 为例，将宏展开后：

```
+ (const WCTProperty &) code {

    static const WCTProperty s_property("code", TZStock.class,
_s_TZStock_binding.addColumnBinding<decltype([TZStock new].code)>("code",
"code"));
    return s_property;
}
static const auto _unused0 = [](WCTPropertyList &propertyList) {
      propertyList.push_back(TZStock.code);
      return nullptr;

}(_s_TZStock_properties);
```

- `columnName` 是通过宏 `WCDB_STRINGIFY` 将属性名 `code` 转为字符串 `"code"`
- `decltype` 操作符用于声明返回类型，此处是 `code` 属性类型即 `int` ，同时传入表达式 `<[TZSTock new].code>`
- `UNUSED_UNIQUE_ID` 宏将 `_unused` 与 `__COUNTER__` 拼接，其中 `__COUNTER__` 在编译时每扫描到一次 `__COUNTER__` 时，他的替换值都会加1，以保证当前文件中每个静态变量的唯一性

  ```
  #define UNUSED_UNIQUE_ID CONCAT(_unused, __COUNTER__)
  ```

绑定分析：

- 首先，会先创建静态变量 `_unused0` ，调用 Lambda 表达式匿名函数，将 `TZStock` 的类属性 `code` 添加到 `_s_TZStock_properties` 中：

  ```
  propertyList.push_back(TZStock.code);
  ```

output:

```
p _s_TZStock_binding // 在访问code类方法前的绑定关系还是空的
(WCTBinding) $0 = {
  m_cls = TZStock
  m_columnBindingMap = size=0 {}
  m_columnBindingList = size=0 {}
  m_indexBindingMap = nullptr {
    __ptr_ = 0x0000000000000000
  }
  m_constraintBindingMap = nullptr {
    __ptr_ = 0x0000000000000000
  }
  m_constraintBindingList = nullptr {
    __ptr_ = 0x0000000000000000
  }
  m_virtualTableArgumentList = nullptr {
    __ptr_ = 0x0000000000000000
  }
  m_virtualTableModuleName = ""
}
```

- 类方法 `+ (const WCTProperty &) code`，会调用 `addColumnBinding`，将生成的绑定关系（`columnBinding`）存到 `_s_TZStock_binding` 中，同时返回绑定关系（`columnBinding`）给 `_s_TZStock_properties`

```
+ (const WCTProperty &) code {

    static const WCTProperty s_property("code", TZStock.class,
_s_TZStock_binding.addColumnBinding<decltype([TZStock new].code)>
("code", "code"));
    return s_property;

}
```

定义并返回静态变量 `s_property`，同时通过 `addColumnBinding` 方法将字段绑定关系添加到 `_s_TZStock_binding`

```
template <typename T>
    std::shared_ptr<WCTColumnBinding>
    addColumnBinding(const std::string &propertyName,
                     const std::string &columnName)
    {
```

```
        std::shared_ptr<WCTColumnBinding> columnBinding(new
WCTColumnBinding(
            m_cls, propertyName, columnName, (T *) nullptr));
        _addColumnBinding(columnName, columnBinding);
        return columnBinding;
    }

void WCTBinding::_addColumnBinding(const std::string &columnName, const
std::shared_ptr<WCTColumnBinding> &columnBinding)
{
    m_columnBindingList.push_back(columnBinding);
    m_columnBindingMap.insert({columnName, columnBinding});
}
```

- 将生成的类属性 `+ (const WCTProperty &) code` 返回值 `code` 添加到属性列表 `_s_TZStock_properties` 中
- `addColumnBinding` 方法分别将 `columnName` 绑定到 `_s_TZStock_binding` 中的 `m_columnBindingMap` 和 `m_columnBindingList` 中：

output：

```
p _s_TZStock_binding
(WCTBinding) $2 = {
  m_cls = TZStock
  m_columnBindingMap = size=1 {
    [0] = {
      first = "code"
      second = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100e68060 strong=3 weak=1 {
        __ptr_ = 0x0000000100e68060
      }
    }
  }
  m_columnBindingList = size=1 {
    [0] = std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100e68060 strong=3 weak=1 {
      __ptr_ = 0x0000000100e68060
    }
  }
  ...
}

p _s_TZStock_properties
(WCTPropertyList) $3 = {
```

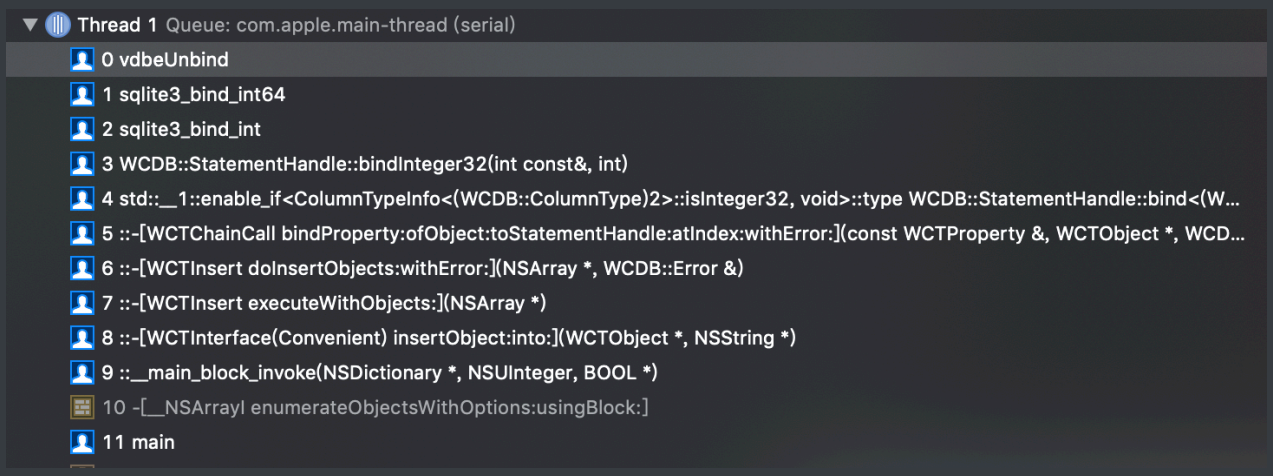
```
    std::__1::list<const WCTProperty, std::__1::allocator<const
  WCTProperty> > = size=1 {
      [0] = {
        WCDB::Column = {
          WCDB::Describable = (m_description = "code")
        }
        WCTPropertyBase = {
          m_cls = TZStock
          m_columnBinding =
  std::__1::shared_ptr<WCTColumnBinding>::element_type @
  0x0000000100e68060 strong=4 weak=1 {
            __ptr_ = 0x0000000100e68060
          }
        }
      }
    }
  }
```

此时我们的映射绑定关系就生成好了，主要保存在 `_s_TZStock_properties` 和 `_s_TZStock_binding` 中。

CRUD

以插入为例：

调用栈：



插入时调用，主要遍历要插入的对象及对象属性，依次插入：

```
- (BOOL)doInsertObjects:(NSArray<WCTObject *> *)objects withError:
(WCDB::Error &)error
{
  ...
```

```
    for (WCTObject *object in objects) { //遍历插入的对象
        index = 1; // 标记属性
        for (const WCTProperty &property : _propertyList) { // 遍历model属性
            const std::shared_ptr<WCTColumnBinding> &columnBinding =
property.getColumnBinding();
            if (!_replace && columnBinding->isPrimary() && columnBinding-
>isAutoIncrement() && object.isAutoIncrement) { // 主键相关
                statementHandle->bind<(WCDB::ColumnType) WCTColumnTypeNil>
(index);
            } else { // 执行插入
                if (![self bindProperty:property
                                ofObject:object
                        toStatementHandle:statementHandle
                                  atIndex:index
                                withError:error]) {
                    return NO;
                }
            }
            ++index;
        }
    ...
    return error.isOK();
}
```

判断插入类型:

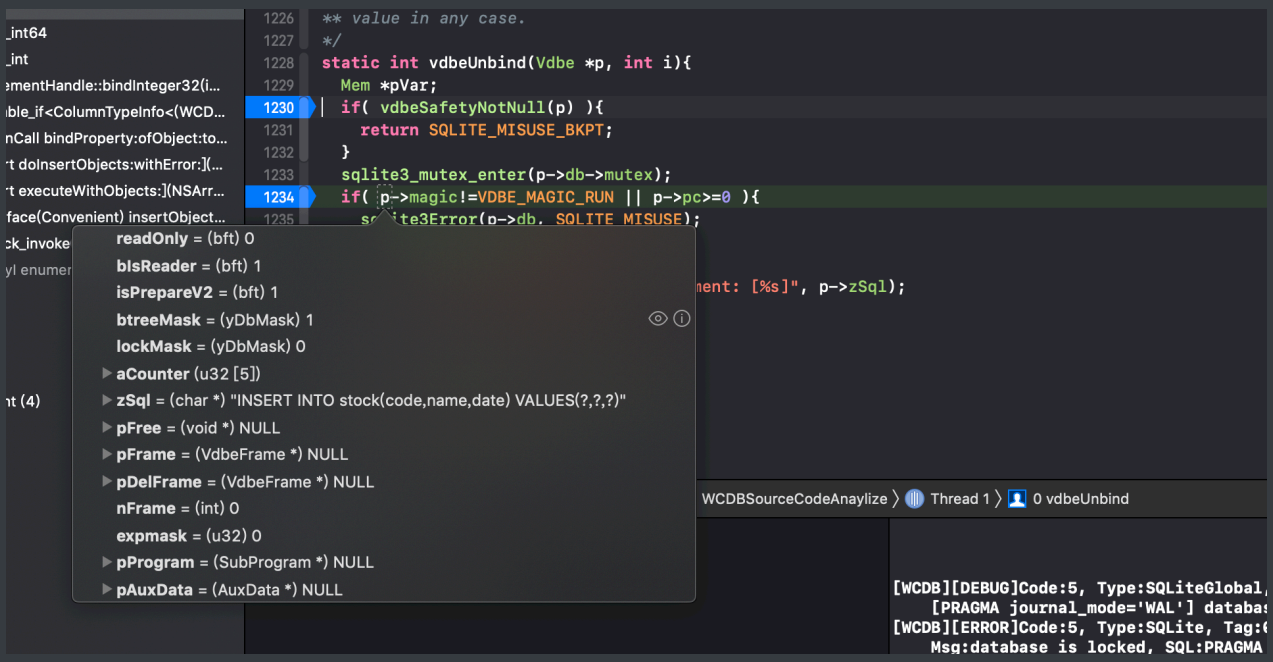
```
...
  const std::shared_ptr<WCTColumnBinding> &columnBinding =
property.getColumnBinding();

const std::shared_ptr<WCTBaseAccessor> &accessor = columnBinding->accessor;

case WCTAccessorObjC: {
    WCTObjCAccessor *objcAccessor = (WCTObjCAccessor *) accessor.get();
    switch (accessor->getColumnType()) { // accessor包含了属性绑定信息
        case WCTColumnTypeInteger32: {
            NSNumber *number = objcAccessor->getObject(object);
            statementHandle->bind<(WCDB::ColumnType)
WCTColumnTypeInteger32>(number.intValue, index);
            break;
        }
        case WCTColumnTypeInteger64: {
            NSNumber *number = objcAccessor->getObject(object);
```

```
            statementHandle->bind<(WCDB::ColumnType)
    WCTColumnTypeInteger64>(number.longLongValue, index);
            break;
        }
        case WCTColumnTypeDouble: {
            NSNumber *number = objcAccessor->getObject(object);
            statementHandle->bind<(WCDB::ColumnType) WCTColumnTypeDouble>
    (number.doubleValue, index);
            break;
        }
        case WCTColumnTypeString: {
            NSString *string = objcAccessor->getObject(object);//取出属性值
            statementHandle->bind<(WCDB::ColumnType) WCTColumnTypeString>
    (string.UTF8String, index); // 插入
            break;
        }
        case WCTColumnTypeBinary: {
            NSData *data = objcAccessor->getObject(object);
            statementHandle->bind<(WCDB::ColumnType) WCTColumnTypeBinary>
    (data.bytes, (int) data.length, index);
            break;
        }
        default:
     ...
```

执行插入SQL：



（二）WINQ

微信移动端数据库组件 WCDB 系列：WINQ原理篇（三）

学习中遇到的问题：

1. 为什么官方给的实例声明的属性不定义为 `nonatomic`，默认下都是原子性，且使用 `retain` 来修饰

```
@property(retain) NSString *name;
@property(retain) NSDate *date;
@property(assign) int code;
```

   似乎使用 `nonatomic` 以及 `copy`、 `strong` 也没什么问题，且这些属性是类本身的，不应该与WCDB耦合。因此个人认为这边可以按ARC的关键字修饰。

2. 绑定关系中的 `weak` `strong` 指的是什么

```
 [0] = {
      WCDB::Column = {
        WCDB::Describable = (m_description = "code")
      }
      WCTPropertyBase = {
        m_cls = TZStock
        m_columnBinding =
std::__1::shared_ptr<WCTColumnBinding>::element_type @
0x0000000100e68060 strong=4 weak=1 {
          __ptr_ = 0x0000000100e68060
        }
      }
```

   o((⊙﹏⊙))o，还不知道

参考目录：

https://github.com/Tencent/wcdb/wiki/ORM使用教程

https://github.com/Tencent/wcdb/wiki

另外，在学习WCDB源码中，学习到了一些宏的高级用法，忽然记起来以前看过的一篇讲述宏的各种黑魔法的博客，也一并分享给大家：

ReactiveCocoa 中奇妙无比的“宏”魔法