# GreenDao简单使用

　　greenDAO是Android的对象/关系映射（ORM）工具。它为关系数据库SQLite提供了面向对象的接口。像greenDAO这样的ORM工具可以为您完成许多重复性任务，并为您的数据提供简单的界面。

# 使用GreenDao的优点

1 只需要定义数据模型，GreenDao框架将创建数据实例和DAO（数据访问对象)，能够节省部分代码

2使用GreenDao大多数实体可以以每秒几千个实体的速率进行插入，更新和加载

3.GreenDao支持加密数据库来保护敏感数据

4.微小的依赖库，GreenDao的关键依赖库大小不超过100kb

5.如果需要，实体可以被激活。而活动实体可以透明的解析关系，并且有更新/删除/刷新方法，以便访问持久性功能

6.GreenDao允许你将协议缓冲区对象直接保存到数据库中，如果你通过protobuf通话到你的服务器则不需要另一个映射。常规实体的所有持久性操作都可以用于protobuf对象。

7.自动生成代码，我们需要关注实体类以及Dao，因为GreenDao已经帮我们生成了。

8.开源

# GreenDao对外提供的核心类

1 DaoMaster

　　保存数据库对象 SQLiteDatabase 并管理特定模式的Dao类。它具有静态方法来创建表或将他们删除。其内部类OpenHelper和DevOpenHelper时SQLite数据库中创建模式的SQLiteOpenHelper实现

2 DaoSession

　　管理特定模式的所有可用Dao对象，可以使用其中一个getter方法获取，DaoSession还为实体提供了一些通用的持久性方法如插入、加载、更新、刷新、删除。最后Daosession对象也跟踪一个身份范围

3 Dao层

　　数据访问对象Dao持续存在并查询实体。对于每个实体，GreenDao生成一个Dao，它比DaoSesssion有更多的持久化方法，例如：count，loadAll，insertInTx

4. 实体

　　持久对象，通常实体时使用标准java属性如POJO或JavaBean来表示数据库的对象

# 关于注解的解释

## 实体类中详细注解说明

@Entity：表明这个实体类会在数据库中生成一个与之相对应的表，其中可配置项：

nameInDb：可以自定义表名，表明该实体对应数据库中的那张表，默认为实体类名；

indexes：定义索引，这里可跨越多个列；

createInDb：如果是有多个实体都关联这个表，可以把多余的实体里面设置为false避免重复创建（默认是true）；

schema：一个项目中有多个schema时，表明要让这个dao属于哪个schema；

active：是否应该生成更新/删除/刷新方法。如果Entity定义了 @ToOne 或 @ToMany关系，那么独立于该值是有效的。意为是否支持实体类之间update，refresh，delete等操作。

@Id：对应数据表中的主键，是一条数据的唯一标识。如果实体没有声明主键，默认创建Long类型主键"\_id"自增。使用Long类型主键时可以通过@Id(autoincrement = true)设置为自增。

@Property(nameInDb = "USER\_NAME" )：可以自定义字段名，注意外键不能使用该属性。表明这个属性对应数据表中的 USER\_NAME 字段。

@NotNull：该属性值不能为空。

@Transient：该属性不会被存入数据库中。

@Unique：表明该属性在数据库中只能有唯一值。

@Index：创建一个索引。通过name设置索引别名，也可以通过unique给索引添加约束。

@Convert：指定一个PropertyConverter用于支持自定义类型(没用过)。

@ToOne：定义自己与一个实体对象的关系。

@ToMany：定义自己与多个实体对象的关系(可不与@ToOne联合使用)。@ToMany的属性referencedJoinProperty，类似于外键约束。

@JoinProperty：对于更复杂的关系，可以使用这个注解标明目标属性的源属性，起关联作用。

@JoinEntity:如果你在做多对多的关系，有其他的表或实体参与，可以给目标属性添加这个额外的注解。

@OrderBy：指定{@ToMany}关系的相关集合的排序，(propertyA, propertyB)默认为按主键ASC排序。

@Generated：这个是build后greendao自动生成的，这个注解理解为防止重复，每一块代码生成后会加个hash作为标记。

## 生成实体类

实体类建完毕后，通过点击AndroidStudio中的Make Project（小锤子的图标），便发现GreenDao为我们的User实体类生成了对应的Getter、Setter方法以及俩个构造函数，同时在我们配置的com.nianlun.greendao.gen包下生成了三个对应类文件DaoMaster、DaoSession和UserDao，之后所有相关的数据库操作都依靠这三个文件了：

DaoMaster：使用greenDAO的切入点。DaoMaster保存数据库对象（SQLiteDatabase）并管理特定模式的DAO类（而不是对象）。 它具有静态方法来创建表或将它们删除。 其内部类OpenHelper和DevOpenHelper是在SQLite数据库中创建模式的SQLiteOpenHelper实现。一个DaoMaster就代表着一个数据库的连接；

DaoSession：管理特定模式的所有可用DAO对象，您可以使用其中一个getter方法获取。 DaoSession还为实体提供了一些通用的持久性方法，如插入，加载，更新，刷新和删除。 DaoSession可以让我们使用一些Entity的基本操作和获取Dao操作类，DaoSession可以创建多个，每一个都是属于同一个数据库连接的；

XxDAO：数据访问对象（DAO）持续存在并查询实体。 对于每个实体，GreenDAO生成一个DAO。 它比DaoSession有更多的持久化方法，例如：count，loadAll和insertInTx等

# 下面进入GreenDao的使用

## 配置环境，添加依赖

在工程目录下build.gradle下dependencies添加插件

复制代码

buildscript **{** repositories **{**

….

**}** dependencies **{** classpath "com.android.tools.build:gradle:4.0.1"  
 classpath 'org.greenrobot:greendao-gradle-plugin:3.3.0' // 添加GreenDao插件  
 // NOTE: Do not place your application dependencies here; they belong  
 // in the individual module build.gradle files  
 **}  
}**

在app的build.gradle文件下进行如下配置

apply plugin: 'com.android.application'  
apply plugin: 'org.greenrobot.greendao' // greendao  
  
android **{** greendao **{** schemaVersion 4 // 指定数据库schema版本号，迁移等操作会用到;  
 daoPackage 'com.example.greendaodemo.dao' // 生成的DAO，DaoMaster和DaoSession的包名，默认是entity所在的包；  
 targetGenDir 'src/main/java' // 生成数据库文件的目录;  
 **}**

**}**

dependencies **{**

….

implementation 'org.greenrobot:greendao:3.2.2' // add library  
**}**

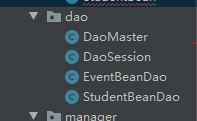
## 新建实体类

用@Entity注解，实体类中的属性即为数据库中对应的字段，最后build项目机会生成相应的代码

@Entity

public class StudentBean {  
 private long id;  
 private String name;  
 private int age;  
 private String gender;  
}

实体类建完毕后，通过点击AndroidStudio中的Make Project（小锤子的图标），便发现GreenDao为我们的User实体类生成了对应的Getter、Setter方法以及俩个构造函数，同时在我们配置的com.nianlun.greendao.gen包下生成了三个对应类文件DaoMaster、DaoSession和UserDao



## GreenDao使用

### 编写DaoManager

用于创建数据库、创建数据库表、包含增删改查的操作

public class DaoManager {  
  
 private Context mContext;  
  
 //创建数据库的名字  
 private static final String *DB\_NAME* = "MyGreenDaoDb.db";  
  
 //多线程中要被共享的使用volatile关键字修饰 GreenDao管理类  
 private volatile static DaoManager *mInstance*;  
  
 //它里边实际上是保存数据库的对象  
 private static DaoMaster *mDaoMaster*;  
  
 //创建数据库的工具  
 private static DaoMaster.DevOpenHelper *mHelper*;  
  
 //管理gen里生成的所有的Dao对象里边带有基本的增删改查的方法  
 private static DaoSession *mDaoSession*;  
  
  
 private DaoManager() {  
 mContext = MyApplication.*app*();  
 }  
  
 */\*\*  
 \* 单例模式获得操作数据库对象  
 \*  
 \** ***@return*** *\*/* public static DaoManager getInstance() {  
 if (*mInstance* == null) {  
 synchronized (DaoManager.class) {  
 if (*mInstance* == null) {  
 *mInstance* = new DaoManager();  
 }  
 }  
 }  
 return *mInstance*;  
 }  
  
 */\*\*  
 \* 判断是否有存在数据库，如果没有则创建  
 \*  
 \** ***@return*** *\*/* public DaoMaster getDaoMaster() {  
 if (*mDaoMaster* == null) {  
 *mHelper* = new MyDevOpenHelper(mContext, *DB\_NAME*);  
 *mDaoMaster* = new DaoMaster(*mHelper*.getWritableDatabase());  
 }  
 return *mDaoMaster*;  
 }  
  
 */\*\*  
 \* 完成对数据库的添加、删除、修改、查询操作，  
 \*  
 \** ***@return*** *\*/* public DaoSession getDaoSession() {  
 if (*mDaoSession* == null) {  
 if (*mDaoMaster* == null) {  
 *mDaoMaster* = getDaoMaster();  
 }  
 *mDaoSession* = *mDaoMaster*.newSession();  
 }  
 return *mDaoSession*;  
 }  
  
 */\*\*  
 \* 关闭所有的操作，数据库开启后，使用完毕要关闭  
 \*/* public void closeConnection() {  
 closeHelper();  
 closeDaoSession();  
 }  
  
 public void closeHelper() {  
 if (*mHelper* != null) {  
 *mHelper*.close();  
 *mHelper* = null;  
 }  
 }  
  
 public void closeDaoSession() {  
 if (*mDaoSession* != null) {  
 *mDaoSession*.clear();  
 *mDaoSession* = null;  
 }  
 }  
}

### 编写XxxTable类，用于完成对数据表的操作

public class StudentBeanTable {  
  
 private static volatile StudentBeanTable *table* = null;  
 private StudentBeanDao studentDao;  
  
 public static StudentBeanTable getInstance() {  
 if(*table* == null) {  
 synchronized (StudentBeanTable.class) {  
 if (*table* == null) {  
 *table* = new StudentBeanTable();  
 }  
 }  
 }  
 return *table*;  
 }  
  
 public StudentBeanTable() {  
 studentDao = DaoManager.*getInstance*().getDaoSession().getStudentBeanDao();  
 }  
  
 public List<StudentBean> loadAll() {  
 return studentDao.loadAll();  
 }  
  
 public void deleteAll() {  
 studentDao.deleteAll();  
 }  
  
 public void insert(StudentBean student) {  
 studentDao.insert(student);  
 }  
  
 public void insertOrReplace(StudentBean student) {  
 studentDao.insertOrReplace(student);  
 }  
  
 public void save(StudentBean student) {  
 studentDao.save(student);  
 }  
  
 public void delete(StudentBean student) {  
 studentDao.delete(student);  
 }  
  
 public void update(StudentBean student) {  
 studentDao.update(student);  
 }  
  
 public List<StudentBean> queryRaw(String where, String... selectionArg) {  
 return studentDao.queryRaw(where, selectionArg);  
 }  
  
 public QueryBuilder queryBuilder() {  
 return studentDao.queryBuilder();  
 }  
  
 public void insertOrReplaceInTx(List<StudentBean> students) {  
 studentDao.insertOrReplaceInTx(students);  
 }  
   
 public StudentBean query() {  
 StudentBean students = studentDao.loadByRowId(0L);  
 return students;  
 }  
  
}

### 编写XxxUtils类，封装所有操作方法

public class StudentUtils {  
  
 */\*\*  
 \* 查找全部数据  
 \** ***@return*** *\*/* public static List loadAll() {  
 List<StudentBean> students = StudentBeanTable.*getInstance*().loadAll();  
 return students;  
 }  
  
 */\*\*  
 \* 删除全部数据  
 \** ***@return*** *\*/* public static void deleteAll() {  
 StudentBeanTable.*getInstance*().deleteAll();  
 }  
  
  
 */\*\*  
 \* 插入一条数据  
 \** ***@return*** *\*/* public static void insert(StudentBean bean) {  
 if(bean != null) {  
 StudentBeanTable.*getInstance*().insert(bean);  
 }  
 }  
  
 */\*\*  
 \* insertOrReplace 数据存在则替换，数据不存在则插入  
 \** ***@param*** *bean  
 \*/* public static void insertOrReplace(StudentBean bean) {  
 if(bean != null) {  
 StudentBeanTable.*getInstance*().insertOrReplace(bean);  
 }  
 }  
  
 */\*\*  
 \* save 类似于insertOrReplace，区别在于save会判断传入对象的key，有key的对象执行更新，无key的执行插入。当对象有key但并不在数据库时会执行失败.适用于保存本地列表。  
 \** ***@param*** *bean  
 \*/* public static void save(StudentBean bean) {  
 if(bean != null) {  
 StudentBeanTable.*getInstance*().save(bean);  
 }  
 }  
  
 */\*\*  
 \* 删除一个数据  
 \** ***@param*** *bean  
 \*/* public static void delete(StudentBean bean) {  
 if(bean != null) {  
 StudentBeanTable.*getInstance*().delete(bean);  
 }  
 }  
  
 */\*\*  
 \* 修改一个数据  
 \** ***@param*** *bean  
 \*/* public static void update(StudentBean bean) {  
 if(bean != null) {  
 StudentBeanTable.*getInstance*().update(bean);  
 }  
 }  
  
 */\*\*  
 \* 查询数据  
 \** ***@param*** *where  
 \** ***@param*** *\*/* public static List<StudentBean> queryRaw(String where, String... selectionArg) {  
 return StudentBeanTable.*getInstance*().queryRaw(where, selectionArg);  
 }  
  
 */\*\*  
 \* 查询数据  
 \*/* public static QueryBuilder queryBuilder() {  
 return StudentBeanTable.*getInstance*().queryBuilder();  
 }  
  
 */\*\*  
 \* 插入一组数据  
 \** ***@return*** *\*/* public static void insertOrReplaceInTx(List<StudentBean> students) {  
 if(students != null) {  
 StudentBeanTable.*getInstance*().insertOrReplaceInTx(students);  
 }  
 }  
}

### 调用XxxUtils类的方法，实现数据库操作

public void onTest1(View v){  
 List list = StudentUtils.*loadAll*();  
 if(list != null) {  
 if(list.size() == 0)  
 Utils.*log*("数据为空");  
 for (int i = 0; i < list.size(); i ++) {  
 Utils.*log*("index " + i + ": " + list.get(i).toString());  
 }  
 }  
}

# 数据库操作

1.插入

insert(User entity)：插入一条记录, 当指定主键在表中存在时会发生异常，插入的id在数据已经存在会报错：

android.database.sqlite.SQLiteConstraintException: UNIQUE constraint

failed: STUDENT\_BEAN.\_id (code 1555 SQLITE\_CONSTRAINT\_PRIMARYKEY)

insertOrReplace(User entity)：当指定主键在表中存在时会覆盖数据,有该数据时则更新，推荐同步数据库时使用该方法

save(User entity):save 类似于insertOrReplace，区别在于save会判断传入对象的key，有key的对象执行更新，无key的执行插入。当对象有key但并不在数据库时会执行失败.适用于保存本地列表。

//save的源码

public void save(T entity) {

if (hasKey(entity)) {  
 update(entity);  
 } else {  
 insert(entity);  
 }  
}

public void inserOrReplace(StudentBean student){

daoSession.insertOrReplace(student);  
}

StudentBean student = new StudentBean();

student.setId(1);  
student.setName("李四");  
student.setAge(12);  
student.setGender("男");  
//insertData(student);  
inserOrReplace(student);

其他一些插入方法

insertInTx(T... entities)：使用事务在数据库中插入给定的实体。

insertInTx(Iterable<T> entities)：使用事务操作，将给定的实体集合插入数据库。  
insertInTx(Iterable<T> entities, boolean setPrimaryKey)：使用事务操作，将给定的实体集合插入数据库，并设置是否设定主键 。  
insertOrReplaceInTx(T... entities)：使用事务操作，将给定的实体插入数据库，若此实体类存在，则覆盖  
insertOrReplaceInTx(Iterable<T> entities)：使用事务操作，将给定的实体插入数据库，若此实体类存在，则覆盖 。  
insertOrReplaceInTx(Iterable<T> entities, boolean setPrimaryKey)：使用事务操作，将给定的实体插入数据库，若此实体类存在，则覆盖，并设置是否设定主键 。  
insertWithoutSettingPk(T entity)：将给定的实体插入数据库,但不设定主键。  
// 新增数据插入相关API  
save(T entity)：将给定的实体插入数据库  
saveInTx(Iterable<T> entities)：将给定的实体集合插入数据库  
saveInTx(T... entities)：使用事务操作，将给定的实体插入数据库

2.查询

List<StudentBean> students = studentDao.loadAll();

StudentBean students2 = studentDao.load(1L);  
StudentBean students3 = studentDao.loadByRowId(0L);

条件查询

//查询全部

List<User> list = mUserDao.queryBuilder().list();  
 //查询 name等于xyh8的数据  
 List<User> list= mUserDao.queryBuilder().where(UserDao.Properties.Name.eq("xyh8")).list();  
 //查询 name不等于xyh8的数据  
 List<User> list= mUserDao.queryBuilder().where(UserDao.Properties.Name.notEq("xyh8")).list();  
 //like 模糊查询  
//查询 name以xyh3开头的数据  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Name.like("xyh3%")).list();  
 //between 区间查询 年龄在20到30之间  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Age.between(20,30)).list();  
 //gt: greater than 半开区间查询，年龄大于18  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Age.gt(18)).list();  
 //ge: greater equal 半封闭区间查询，年龄大于或者等于18  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Age.ge(18)).list();  
 //lt: less than 半开区间查询，年龄小于18  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Age.lt(18)).list();  
 //le: less equal 半封闭区间查询，年龄小于或者等于18  
 List<User> list = mUserDao.queryBuilder().where(UserDao.Properties.Age.le(18)).list();  
//排序  
 //名字以xyh8开头，年龄升序排序  
 List<User> list = mUserDao.queryBuilder()  
 .where(UserDao.Properties.Name.like("xyh8%"))  
 .orderAsc(UserDao.Properties.Age)  
 .list();  
  
 //名字以xyh8开头，年龄降序排序  
 List<User> list = mUserDao.queryBuilder()  
 .where(UserDao.Properties.Name.like("xyh8%"))  
 .orderDesc(UserDao.Properties.Age)  
 .list();

3 更新

update(T entity) ：更新给定的实体

updateInTx(Iterable<T> entities) ：使用事务操作，更新给定的实体  
updateInTx(T... entities)：使用事务操作，更新给定的实体  
studentDao.update(student);  
studentDao.updateInTx(student);

4 删除

//删除全部

mUserDao.deleteAll();  
delete(T entity)：从数据库中删除给定的实体  
deleteByKey(K key)：从数据库中删除给定Key所对应的实体  
deleteInTx(T... entities)：使用事务操作删除数据库中给定的实体  
deleteInTx(<T> entities)：使用事务操作删除数据库中给定实体集合中的实体  
deleteByKeyInTx(K... keys)：使用事务操作删除数据库中给定的所有key所对应的实体  
deleteByKeyInTx(Iterable<K> keys)：使用事务操作删除数据库中给定的所有key所对应的实体   
public void delete(StudentBean student)  
{  
 studentDao.delete(student);  
}

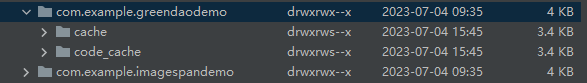
5 封装

public class DaoManager {

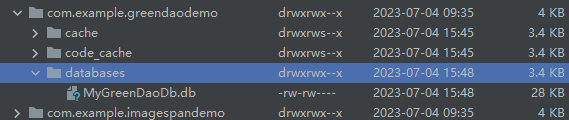
private Context mContext;  
  
 //创建数据库的名字  
 private static final String *DB\_NAME* = "MyGreenDaoDb.db";  
  
 //多线程中要被共享的使用volatile关键字修饰 GreenDao管理类  
 private volatile static DaoManager *mInstance*;  
  
 //它里边实际上是保存数据库的对象  
 private static DaoMaster *mDaoMaster*;  
  
 //创建数据库的工具  
 private static DaoMaster.DevOpenHelper *mHelper*;  
  
 //管理gen里生成的所有的Dao对象里边带有基本的增删改查的方法  
 private static DaoSession *mDaoSession*;  
  
  
 private DaoManager() {  
 mContext = MyApplication.*app*();  
 }  
  
 */\*\*  
 \* 单例模式获得操作数据库对象  
 \*  
 \** ***@return*** *\*/* public static DaoManager getInstance() {  
 if (*mInstance* == null) {  
 synchronized (DaoManager.class) {  
 if (*mInstance* == null) {  
 *mInstance* = new DaoManager();  
 }  
 }  
 }  
 return *mInstance*;  
 }  
  
 */\*\*  
 \* 判断是否有存在数据库，如果没有则创建  
 \*  
 \** ***@return*** *\*/* public DaoMaster getDaoMaster() {  
 if (*mDaoMaster* == null) {  
 *mHelper* = new MyDevOpenHelper(mContext, *DB\_NAME*);  
 *mDaoMaster* = new DaoMaster(*mHelper*.getWritableDatabase());  
 }  
 return *mDaoMaster*;  
 }  
  
 */\*\*  
 \* 完成对数据库的添加、删除、修改、查询操作，  
 \*  
 \** ***@return*** *\*/* public DaoSession getDaoSession() {  
 if (*mDaoSession* == null) {  
 if (*mDaoMaster* == null) {  
 *mDaoMaster* = getDaoMaster();  
 }  
 *mDaoSession* = *mDaoMaster*.newSession();  
 }  
 return *mDaoSession*;  
 }  
  
 */\*\*  
 \* 关闭所有的操作，数据库开启后，使用完毕要关闭  
 \*/* public void closeConnection() {  
 closeHelper();  
 closeDaoSession();  
 }  
  
 public void closeHelper() {  
 if (*mHelper* != null) {  
 *mHelper*.close();  
 *mHelper* = null;  
 }  
 }  
  
 public void closeDaoSession() {  
 if (*mDaoSession* != null) {  
 *mDaoSession*.clear();  
 *mDaoSession* = null;  
 }  
 }  
}

# 数据库存储路径

在没有使用数据库前，私有目录是如下这样的：



使用数据库后：



数据库的存储路径是在私有目录的databases目录下的。

数据库名称实在创建DaoMaster.DevOpenHelper时指定的。

public DaoMaster getDaoMaster() {  
 if (*mDaoMaster* == null) {  
 *mHelper* = new MyDevOpenHelper(mContext, *DB\_NAME*);  
 *mDaoMaster* = new DaoMaster(*mHelper*.getWritableDatabase());  
 }  
 return *mDaoMaster*;  
}

# 数据库升级

## 介绍

比如需要在实体类加一个字段 或者 改变字段属性等 就需要版本更新来保存以前的数据了；

1.思路：创建临时表–>删除原表–>创建新表–>复制临时表数据到新表并删除临时表；这样数据库表的更新就完成了

2.由于升级数据库需要在DevOpenHelper类的onUpgrade()方法里面继续，因此我们需要自定义一个类继承DevOpenHelper重写onUpgrade()方法

3 修改在项目根目录build.gradle文件中配置的数据库版本号(新版本号一定要比老版本大)

greendao {

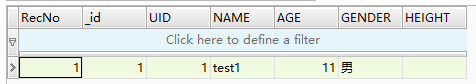
schemaVersion 6  
 daoPackage 'com.example.greendaodemo1'  
 targetGenDir 'src/main/java'  
}

## 实践

1. 假设现在已经创建了一张表

@Entity  
public class StudentBean {  
  
 @Id  
 Long \_id;  
 private long uid;  
 private String name;  
 private int age;  
 private String gender;  
 private String height;

}



1. 往该表添加一个字段

private String test;

点击Build->Make Project之后会自动生成构造方法和get，set方法

@Generated(hash = 292478697)  
public StudentBean(Long \_id, long uid, String name, int age, String gender,  
 String height, String test) {  
 this.\_id = \_id;  
 this.uid = uid;  
 this.name = name;  
 this.age = age;  
 this.gender = gender;  
 this.height = height;  
 this.test = test;  
}

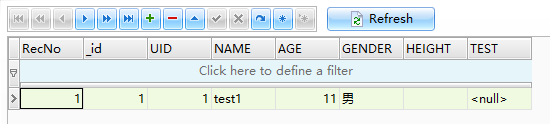
public String getTest() {  
 return this.test;  
}  
  
public void setTest(String test) {  
 this.test = test;  
}

1. 修改数据库版本号为5，原来是4

greendao **{** schemaVersion 5 // 指定数据库schema版本号，迁移等操作会用到;  
 daoPackage 'com.example.greendaodemo.dao' // 生成的DAO，DaoMaster和DaoSession的包名，默认是entity所在的包；  
 targetGenDir 'src/main/java' // 生成数据库文件的目录;  
**}**

1. 运行程序

可以看到之前的数据还在，而且添加了一个test字段



# 多表关联

## 一对一关联

假如有两个表，A表和B表，A表的一条记录只对应于B表的一条记录，反之亦然，就是一对一关联。比如学生表和身份证表，一条学生记录只对应于一条身份证记录，一条身份证记录只对应一条学生记录，学生表和身份证表就是一对一关联。

@ToOne：表示实体类A和实体类B是一对一关联

joinProperty：该参数表示两个实体类的连接属性

### 单向关联

A表中持有B表的引用，而B表没有持有A表的引用。新建Card实体类：

@Entity  
public class Card1 {  
  
 @Id(autoincrement = true)  
 private Long id;  
  
 @Unique  
 @NotNull  
 private String cardCode;

}

User类更改如下：

@Entity(  
 nameInDb = "USERS",  
 indexes = {  
 @Index(value = "name DESC")  
 }  
)  
public class User1 {  
  
 @Id(autoincrement = true)  
 private Long id;  
  
 private Long cardId; // 新增的，外键  
  
 //设置一对一关联，连接属性是cardId  
 @ToOne(joinProperty ="cardId") // 注意该参数的值  
 private Card1 card; // 新增的  
  
 @Index(name="usercode\_index",unique = true)  
 private String usercode;  
  
 @Property(nameInDb = "userName")  
 @NotNull  
 private String name;  
  
 private String userAddress;  
  
 @Transient  
 private int tempUserSign;

}

更改数据库版本号，sync now后，make project。在MainActivity中添加以下方法，然后调用即可：

// 单向关联插入  
private void insertOneToOne(){  
 // 先生成一条Card记录  
 Card1 card1 = new Card1();  
 card1.setCardCode("434377777");  
 cardDao.insert(card1);  
  
 User1 user1 = new User1();  
 user1.setName("孙悟空");  
 user1.setUserAddress("花果山水帘洞");  
 user1.setUsercode("001");  
 user1.setCard(card1);  
 userDao.insert(user1);  
}  
  
// 单向关联查询  
private void queryOneToOne(){  
 User1 user = userDao.queryBuilder().where(User1Dao.Properties.*Name*.eq("孙悟空")).build().unique();  
 Card1 card = user.getCard();  
 if(user!=null && card!=null){  
 Log.*d*("TAG", "一对一添加记录，查询后的结果是：\n" + "姓名：" + user.getName()  
 + "\n身份证号" + card.getCardCode() + "\n"  
 + "Card表的id主键值为：" + card.getId()+ "\n"  
 + "User表的外键cardId的值为：" + user.getCardId());  
 }  
}

### 双向关联

Card类更改如下，然后更改数据库版本号和make project，User类保持不变：

@Entity  
public class Card2 {  
 @Id(autoincrement = true)  
 private Long id;  
  
 @Unique  
 @NotNull  
 private String cardCode;  
  
 private Long userId; // 新增的，外键  
 //设置一对一关联  
 @ToOne(joinProperty = "userId") // 注意该参数的值  
 private User2 user; // 新增的

}

双向关联的插入稍微不同，但查询的思想时一样的。在MainActivity中添加如下方法：

private void insertCardOneTOne2(){  
 User2 user1 = new User2();  
 user1.setName("孙悟空");  
 user1.setUserAddress("花果山水帘洞");  
 user1.setUsercode("001");  
  
 Card2 card1 = new Card2();  
 card1.setCardCode("434377777");  
  
 /\* 注意以下代码的顺序 \*/  
 userDao2.insert(user1);  
 card1.setUser(user1);  
  
 cardDao2.insert(card1);  
 //补上之前没有设置的user1的外键值  
 user1.setCard(card1);  
 //更新user1对象  
 userDao2.update(user1);  
  
}  
  
private void queryCardOneToOne2(){  
 Card2 card1 = cardDao2.queryBuilder().where(Card2Dao.Properties.*CardCode*.eq("434377777")).build().unique();  
 User2 user1 = card1.getUser();  
 if(user1 != null && card1 != null){  
 Log.*d*("TAG", "姓名："+user1.getName()+"\n"  
 +"Card表的id主键值："+card1.getId()+"\n"  
 +"User表的外键cardId的值为："+user1.getCardId());  
 }  
}

来个小结，先来看单向关联，可以通过一个User对象得到一个Card对象与之对应，反之不能。但是，双向关联可以做到，这就是它们之间最主要的区别。

## 一对多关联

两个表，A表和B表，A表的一条记录可对应B表的多条记录，而B表的一条记录只能对应A表的一条记录，则B表对A表就是多对一关联，而A表对B表就是一对多关联。举个例子，班级表的一个班级可对应于学生表的多个学生，而一个学生只能属于一个班级，学生表对班级表就是多对一关联，班级表对学生表就是一对多关联。因此，一对多和多对一是相互关系。

@ToOne：代表“一”

joinProperty

@ToMany：代表“多”

referencedJoinProperty

示例的场景是，一个用户可以购买多个商品。新建Orders类，代码如下：

@Entity  
public class Orders {  
  
 @Id(autoincrement = true)  
 private Long id;  
  
 private String goodsName;  
  
 private Long userId;//外键，对于User类的主键  
  
 @ToOne(joinProperty = "userId") // 注意该参数的值  
 private User3 user;

}

User类的更改如下：

@Entity(  
 nameInDb = "USERS3",  
 indexes = {  
 @Index(value = "name DESC")  
 }  
)  
public class User3 {  
 @Id(autoincrement = true)  
 private Long id;  
  
 private Long cardId;  
  
 //设置一对一关联，连接属性是cardId  
 @ToOne(joinProperty ="cardId") //注意参数的值  
 private Card1 card;  
  
 //设置一对多关联，连接属性是Orders类的外键userId  
 @ToMany(referencedJoinProperty = "userId") // 注意参数的值  
 private List<Orders> orders;  
  
 @Property(nameInDb = "userName")  
 @NotNull  
 private String name;  
  
 private String userAddress;  
  
 @Transient  
 private int tempUserSign;

}

更改数据库版本号，然后make project，接着在MainActivity中添加如下代码，然后直接调用即可：

private void insertOneToMany3(){  
 List<Orders> orderList = new ArrayList<Orders>();  
 // 这些数据的来源请参考上一章所讲的内容，因为在上一章中有方法为测试提供数据源  
 User3 user1 = new User3();  
 user1.setName("孙悟空");  
 user1.setUserAddress("花果山水帘洞");  
 userDao3.insert(user1);  
  
 User3 user2 = new User3();  
 user2.setName("猪八戒");  
 user2.setUserAddress("花果山水帘洞");  
 userDao3.insert(user2);  
  
 Orders order1 = new Orders();  
 order1.setGoodsName("金箍棒");  
 order1.setUser(user1); //设置外键值时，要用setUser()方法，以确保外键值不会出错  
  
 Orders order2 = new Orders();  
 order2.setGoodsName("黄金甲");  
 order2.setUser(user1);  
  
 Orders order3 = new Orders();  
 order3.setGoodsName("紫金冠");  
 order3.setUser(user1);  
  
 Orders order4 = new Orders();  
 order4.setGoodsName("紫金冠");  
 order4.setUser(user2);  
  
 orderList.add(order1);  
 orderList.add(order2);  
 orderList.add(order3);  
 orderList.add(order4);  
  
 ordersDao.insertInTx(orderList);  
}  
  
private void queryToManyUserToOrder3() {  
 List<Orders> ordersList;  
 User3 user1 = userDao3.queryBuilder().where(User3Dao.Properties.*Name*.eq("孙悟空")).build().unique();  
  
 //直接通过User对象的getOrders()方法获得此用户的所有订单  
 ordersList = user1.getOrders();  
 Log.*d*("TAG", user1.getName() + "的订单内容为：");  
  
 int i = 0;  
 if (ordersList != null) {  
 for (Orders order : ordersList) {  
 i = i + 1;  
 Log.*d*("TAG", "第" + i + "条订单的结果：" + ",id:" + order.getId()  
 + ",商品名：" + order.getGoodsName()  
 + ",用户名：" + user1.getName());  
 }  
 }  
}

## 多对多关联

两个表，A表和B表，A表的一条记录可对应于B表的多条记录，而B表的一条记录也对应A表的多条记录，则A表和B表之间就是多对多关联。比如作者表和书表，作者表的一个作者可以写多本书，而一本书也可以是多个作者，作者表和书表之间就是多对多关联。

@ToMany：设置多对多

@JoinEntity：设置连接中间类

entity：中间类，需要创建

sourceProperty：源属性，就是本类，而参数的值就是中间类中代表该类的属性

targetProperty：目标属性，就是要关联的类，而参数的值就是中间类中代表该类的属性

示例的场景是，一个老师可以教多门课程，一个课程可以也可以由多个老师任教。

新建中间类JoinTeacherWithCourse，代码如下：

@Entity  
public class JoinTeacherWithCourse {  
 @Id(autoincrement = true)  
 private Long id;  
  
 // 代表老师的id  
 private Long tId;  
  
 // 代表课程的id  
 private Long cId;  
}

新建Teacher类和Course类，代码如下：

@Entity  
public class Course {  
 @Id(autoincrement = true)  
 private Long id;  
  
 private String name;  
  
 //多对多  
 @ToMany  
 // 连接两个多对多实体类的中间类  
 @JoinEntity(  
 // 中间类类名  
 entity = JoinTeacherWithCourse.class,  
 // 源属性，中间类的外键，对应Course类的主键  
 sourceProperty = "cId",  
 // 目标属性，中间类的外键，对应Teacher类的主键  
 targetProperty = "tId"  
 )  
 private List<Teacher> teachers;  
}

更改数据库版本号，然后make project，接着在MainActivity中添加如下代码：

@Entity  
public class Teacher {  
 @Id(autoincrement = true)  
 private Long id;  
  
 private String name;  
  
 // 多对多  
 @ToMany  
 // 连接两个多对多实体类的中间类  
 @JoinEntity(  
 // 中间类类名  
 entity = JoinTeacherWithCourse.class,  
 // 源属性，中间类的外键，对应Teacher类的主键  
 sourceProperty = "tId",  
 // 目标属性，中间类的外键，对应Course类的主键  
 targetProperty = "cId"  
 )  
 private List<Course> courses;  
}  
  
// 多对多插入  
private void insertManyToMany() {  
 List<Course> courses = new ArrayList<>();  
 Course course1 = new Course();  
 course1.setName("英语");  
  
 Course course2 = new Course();  
 course2.setName("语文");  
  
 Course course3 = new Course();  
 course3.setName("数学");  
  
 courses.add(course1);  
 courses.add(course2);  
 courses.add(course3);  
 courseDao.insertInTx(courses);  
  
 List<Teacher> teacherList = new ArrayList<>();  
 Teacher teacher1 = new Teacher();  
 teacher1.setName("孙悟空");  
  
 Teacher teacher2 = new Teacher();  
 teacher2.setName("猪八戒");  
  
 Teacher teacher3 = new Teacher();  
 teacher3.setName("沙和尚");  
  
 teacherList.add(teacher1);  
 teacherList.add(teacher2);  
 teacherList.add(teacher3);  
 teacherDao.insertInTx(teacherList);  
  
 List<JoinTeacherWithCourse> teacherWithCourses = new ArrayList<>();  
 // 悟空教英语  
 JoinTeacherWithCourse teacherWithCourse1 = new JoinTeacherWithCourse();  
 teacherWithCourse1.setTId(teacher1.getId());  
 teacherWithCourse1.setCId(course1.getId());  
  
 // 悟空叫语文  
 JoinTeacherWithCourse teacherWithCourse2 = new JoinTeacherWithCourse();  
 teacherWithCourse2.setTId(teacher1.getId());  
 teacherWithCourse2.setCId(course2.getId());  
  
 // 悟空叫数学  
 JoinTeacherWithCourse teacherWithCourse3 = new JoinTeacherWithCourse();  
 teacherWithCourse3.setTId(teacher1.getId());  
 teacherWithCourse3.setCId(course3.getId());  
  
 // 沙和尚教语文  
 JoinTeacherWithCourse teacherWithCourse4 = new JoinTeacherWithCourse();  
 teacherWithCourse4.setTId(teacher2.getId());  
 teacherWithCourse4.setCId(course2.getId());  
  
 teacherWithCourses.add(teacherWithCourse1);  
 teacherWithCourses.add(teacherWithCourse2);  
 teacherWithCourses.add(teacherWithCourse3);  
 teacherWithCourses.add(teacherWithCourse4);  
 teacherWithCourseDao.insertInTx(teacherWithCourses);  
}  
  
// 多对多查询,通过”教师“找到课程  
private void queryManyToManyT() {  
 Teacher teacher = teacherDao.queryBuilder().where(TeacherDao.Properties.Name.eq("孙悟空"))  
 .build().unique();  
 List<Course> courses = teacher.getCourses();  
  
 if (courses != null) {  
 Log.*d*("TAG", "孙悟空所教的课程：");  
 for (Course course : courses) {  
 Log.*d*("TAG", "课程名：" + course.getName());  
 }  
 }  
}  
  
// 多对多查询,通过”课程“找到课程  
private void queryManyToManyC() {  
 Course course = courseDao.queryBuilder().where(CourseDao.Properties.Name.eq("语文"))  
 .build().unique();  
 List<Teacher> teachers = course.getTeachers();  
  
 if (teachers != null) {  
 Log.*d*("TAG", "教语文的老师有：");  
 for (Teacher teacher : teachers) {  
 Log.*d*("TAG", "教师名：" + teacher.getName());  
 }  
 }  
}

注意：要先插入Teacher和Course的记录，然后才能插入JoinTeacherWithCourse的记录

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5