DBFlow
官方教程

yumenokanata

Published
with GitBook

目錄

DBFlow

> 偶然发现的DBFlow ORM库，它的很多特性我很喜欢。
>
> 刚开始接触的大家可能会发现它的接口和ActiveAndroid很相似，使用一个全局单例进行操作而没有专门的Dao，但DBFlow采用的是模板代码生成的方式进行数据操作，相比于使用反射的ActiveAndroid，在性能有着绝对的优势。
> 而相比于同样使用模板代码生成的GreenDao，它不需要单独建立一个纯java的程序，并通过代码描述表结构。它采用了更加清晰简洁的注解加注解处理器(AnnotationProcessing)的技术生成表结构。使用更简洁，但又有着不相上下的高性能。
>
> DBFlow还有着许多有趣、简单而又强大的功能，希望你能喜欢上它。
>
> 目前DBFlow的中文的介绍和教程几乎没有，所以决定汉化此教程。但本人英语水平有限，如有勘误，欢迎大家指正。

DBFlow是一个基于AnnotationProcessing(注解处理器)的强大、健壮同时又简单的ORM框架。

此框架设计为了速度、性能和可用性。消除了大量死板的数据库代码，取而代之的是强大而又简介的API接口。

DBFlow使数据库的操作变得简洁而稳定，让您能够更专注地编写令人惊讶的APP。

为什么使用DBFlow

DBFlow的设计吸取了其他很多ORM框架中好的特征，并将之做得更好。它很灵活，让你能更专注于App中真正需要关注的地方。不要让一个ORM库限制了你的思维，而是让代码在你的App中工作得更好。

- 扩展性： ORM所需的数据类只需要实现 `Model` 接口即可，而不需要必须继承一个类，同时为了方便，我们还是推荐继承 `BaseModel` ，这是 `Model` 接口的一个标准实现。这样你既可以通过继承一个来自其他包的非 `Model` 类来生成你的数据库表，也可以通过继承一个 `Model` 类并通过添加 `@Column` 注解

的属性向表中自由添加列。这一切都是为了方便你的使用。

- 速度：DBFlow基于AnnotationProcessing(注解处理器），通过编译期代码生成，运行时对性能是零损耗的。通过模板来为你维护生成的代码。通过缓存和尽可能地重用对象，我们得到了比原生SQLite更快的速度。同时我们还支持懒加载（lazy-loading），比如对于 `@ForeignKey` 和 `@OneToMany` ，这使得我们有着更高效得查询效率。
- **SQLite查询流(SQLite Query Flow)**： DBFlow的查询语法尽可能地和SQL语句相似，使您能更快上手。 `select(name, screenSize).from(Android.class).where(name.is("Nexus 5x")).and(version.is(6.0)).querySingle()`
- 开源： 整个DBFlow库都是开源的，而且也非常欢迎大家来为这个库贡献自己的力量。
- **Robust**： 我们支持 `Trigger` , `ModelView` , `Index` , `Migration` ,所有的数据库操作都在同一个线程（线程安全），还有其他特性。
- 多数据库、多表单： 我们无缝支持多数据库文件，database modules using DBFlow in other dependencies, simultaneously.
- 基于**SQLite**： SQLite是世界上使用最广泛的数据库引擎，基于SQLite的DBFlow使你不需要被限制在某些平台上。

更新日志

跟踪我们的进度、查看每次的版本的更新内容： releases tab

for older changes, from other xx.xx versions, check it out here

---

引入到你的工程

我们需要先倒入 apt plugin 库到你的classpath，以启用AnnotationProcessing（注解处理器）：

```
buildscript {
    repositories {
      // required for this library, don't use mavenCentral()
        jcenter()
    }
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

然后添加我们的托管仓库网址.

```
allProjects {
  repositories {
    maven { url "https://jitpack.io" }
  }
}
```

最后即可添加我们的库到你项目级别的build.gradle文件中了:

```
  apply plugin: 'com.neenbedankt.android-apt'

  def dbflow_version = "3.0.0-beta2"
  // or dbflow_version = "develop-SNAPSHOT" for grabbing latest dep

  dependencies {
    apt 'com.github.Raizlabs.DBFlow:dbflow-processor:${dbflow_versi
    compile "com.github.Raizlabs.DBFlow:dbflow-core:${dbflow_versio
    compile "com.github.Raizlabs.DBFlow:dbflow:${dbflow_version}"

    // sql-cipher database encyrption (optional)
    compile "com.github.Raizlabs.DBFlow:dbflow-sqlcipher:${dbflow_v
  }
```

You can also specify a commit hash instead of `develop-SNAPSHOT` to grab a specific commit.

开始

本章我们将学习如何建立一个数据库、表单和 `Model` 之间的联系。

蚂蚁女皇：我们的教程将以蚂蚁王国为例，我们将建立 蚂蚁领地 `Colony` 、蚂蚁女王 `Queen` 、蚂蚁 `Ant` 之间的关系表。

它们的关系如下：

```
Colony (1..1) -> Queen (1...many)-> Ants
```

初始化DBFlow

初始化DBFlow的操作，包括打开数据库、进行数据迁移和创建的操作都需要在自定义的 `Application` 中：

```
public class ExampleApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        FlowManager.init(this);
    }

}
```

当然，自定义的Application需要在 `AndroidManifest.xml` 文件中声明：

```
<application
  android:name="{packageName}.ExampleApplication"
  ...>
</application>
```

创建我们的数据库

在DBFlow中，通过Table类中声明对数据库类的连接来产生交互，并生成相应的代码。（In DBFlow, databases are placeholder objects that generate interactions from which tables "connect" themselves to.）

我们需要定义一个蚂蚁领地 `Colony` 的数据库来保存我们的小蚂蚁们：

```
@Database(name = ColonyDatabase.NAME, version = ColonyDatabase.VERS
public class ColonyDatabase {

  public static final String NAME = "Colonies";

  public static final int VERSION = 1;
}
```

作为一种推荐的写法，我们定义了 `NAME` 和 `VERSION` 两个公共常量，以方便我们定义的其他DBFlow组件能够在之后引用到这两个属性（如果需要的话）。

Note: 如果你希望使用 SQLCipher 对数据库进行加密的话，可以阅读 数据库加密 章节。

创建表并建立关系

现在我们有了一个地方可以创建我们的蚂蚁领地 `Colony` 了，但还需要建立 `Model` 来表示我们希望底层数据被如何保存。

蚂蚁女王的表（Table）

现在我们将会从上至下地建立我们的蚂蚁领地 `Colony`，目前我们每个领地里只有一个女王。我们将会通过ORM（object-relational mapping）的方式建立数据库，这意味着这些Model中的每一项属性都将被映射为数据表中的一列。

在DBFlow中，每个与数据库连接的ORM对象类都必须继承 `Model` 接口。这是为了确保这些对象都能有相同的一些方法。通常为了方便我们继承 `BaseModel` 类，这个类也是 `Model` 的标准实现。

一个正确的数据表类需要以下几项：

1. 对类添加 `@Table` 注解

2. 声明所连接的数据库类，这里是 `ColonyDatabase` 。
3. 定义至少一个主键。
4. 这个类和这个类中数据库相关列的修饰符必须是包内私有或者 `public` 。
5. 这样生成的 `_Adapter` 类能够访问到它。

NOTE: 列（Column）属性可以是 `private` ，但这样就必须指定公有 `public` 的getter和setter方法。

```
@Table(database = ColonyDatabase.class)
public class Queen extends BaseModel {

  @PrimaryKey(autoincrement = true)
  long id;

  @Column
  String name;

}
```

现在我们有了一个蚂蚁女王 `Queen` ，还需要为我们的女王建立一个领地 `Colony` 。

蚂蚁领地的表（Table）

```
@ModelContainer
@Table(database = ColonyDatabase.class)
public class Colony extends BaseModel {

  @PrimaryKey(autoincrement = true)
  long id;

  @Column
  String name;

}
```

我们有了一个女王 `Queen` 和一个领地 `Colony`，我们希望给他们建立一个一对一的关系。 我们希望当一个数据被移除，比如如果有一场大火烧毁了一个蚂蚁领地 `Colony`，那么这个领地的女王 `Queen` 也活不长了，所以我们希望能“杀掉”与这个领地 `Colony` 相关联的女王 `Queen` 的数据。

1-1关系

为了建立关系，我们将会定义一个外键：

```
@ModelContainer
@Table(database = ColonyDatabase.class)
public class Queen extends BaseModel {

  //...previous code here

  @Column
  @ForeignKey(saveForeignKeyModel = false)
  Colony colony;

}
```

定义了外键的 `Model` 将会在载入数据库时通过查询自动将外键引用的值保存在这里。对于性能有要求的话可显式设置 `saveForeignKeyModel=false` 使 `Queen` 对象保存时不会保存对应的 `Colony` 对象。

在3.0版本中，我们不再需要显式设置 `@ForeignKeyReference` 来规定列属性了，DBFlow会自动分析和生成。

对于被用于 `ForeignKeyContainer` 属性的 `Model` 类， `@ModelContainer` 注解是必须的。因为 `ForeignKeyContainer` 的实现需要生成一些必须的额外代码。

蚂蚁表（Table）与一对多关系

我们有了领地 `Colony` 和女王 `Queen`，现在我们可以添加一些蚂蚁进去了：

```
@Table(database = ColonyDatabase.class)
public class Ant extends BaseModel {

  @PrimaryKey(autoincrement = true)
  long id;

  @Column
  String type;

  @Column
  boolean isMale;

  @ForeignKey(saveForeignKeyModel = false)
  ForeignKeyContainer<Queen> queenForeignKeyContainer;

  /**
  * Example of setting the model for the queen.
  */
  public void associateQueen(Queen queen) {
    queenForeignKeyContainer = FlowManager.getContainerAdapter(Quee
  }
}
```

里面有 `type` 和 `isMale` 属性，用于表示这只蚂蚁的基本信息，比如是“工蚁”或者“其他”，是“公”还是“母”。

对于我们的几千只蚂蚁，我们使用了 `ForeignKeyContainer` 来保存外键。这是一种为性能优化的懒加载（lazy-load），只有在调用 `ForeignKeyContainer` 的 `toModel()` 方法时才会对数据库执行实际的查询。

为了使用 `ForeignKeyContainer`，我们需要给 `Queen` 添加 `@ModelContainer` 注解以生成相应的额外代码。

接下来我们可以通过懒加载添加一对多的关系了：

```
@ModelContainer
@Table(database = ColonyDatabase.class)
public class Queen extends BaseModel {
  //...

  // needs to be accessible for DELETE
  List<Ant> ants;

  @OneToMany(methods = {OneToMany.Method.SAVE, OneToMany.Method.DEL
  public List<Ant> getMyAnts() {
    if (ants == null || ants.isEmpty()) {
            ants = SQLite.select()
                    .from(Ant.class)
                    .where(Ant_Table.queenForeignKeyContainer_id.eq
                    .queryList();
    }
    return ants;
  }
}
```

如果你希望由你来进行关系的懒加载，只需要将 `OneToMany.Method.DELETE` 和 `SAVE` 替换为 `ALL` 。

如果你希望无论 `Queen` 的数据如何变化它们都不会保存，只需要设置 `DELETE` 和 `LOAD` 。

表和数据库属性

创建你的数据库

在DBFlow中，创建数据库是非常简单的，只需要定义一个 `@Database` 类：

```
@Database(name = AppDatabase.NAME, version = AppDatabase.VERSION)
public class AppDatabase {

  public static final String NAME = "AppDatabase";

  public static final int VERSION = 1;
}
```

*P.S.*你可以定义多个 `@Database` ，但请保证他们没有重名。

预包装的数据库

如果你需要在创建时载入一个预包装好的数据库，只需要将数据库文件 `.db` 放在 `src/main/assets/{databaseName}.db` ，这样在DBFlow创建数据库的时候就会自动将这个数据库拷贝进来使用。但请注意，由于预包装的数据库文件已经被打包进了APK文件，所以我们无法在拷贝后将其删除，这会增加APK的大小（等于预包装数据库文件的大小）。

配置属性

全局冲突处理：通过定义 `insertConflict()` 和 `updateConflict()` ，任何没有被明确定义的 `@Table` 都将使用与之对应的 `@Database` 中的表。
(Global Conflict Handling: By specifying `insertConflict()` and `updateConflict()` here, any `@Table` that does not explicitly define either itself will use the corresponding one from the associated `@Database` .)

Kotlin：3.0开始，我们提供了对Kotlin的良好支持。

以前我们需要定义一个 `generatedClassSeparator()` 为之工作：

```
@Database(generatedClassSeparator = "_")
```

数据库完整性检查：如果设置了 `consistencyChecksEnabled()` 方法，我们将会在数据库打开时运行 `PRAGMA quick_check(1)` 进行检查。如果检查失败，我们将尝试拷贝预定义的数据库文件覆盖当前数据库。

简单的数据库备份： `backupEnabled()` 开启数据库备份后，我们便可以简单的备份数据库了：

```
FlowManager.getDatabaseForTable(table).backupDB()
```

NOTE：请注意，这会首先生成一个临时的数据库，以防止备份失败。

开启外键常量**(Constrants)**：使用 `foreignKeysSupported()=true` 开启强制外键（the database enforce foreign keys）。如果设为false，我们仍然能够定义 `@ForeignKey`，但它们的关系将不是强制的。

自定义**OpenHelper**类：通过继承 `FlowSQLiteOpenHelper` 我们可以定义自己的Helper类。注意，在自定义Helper类中需要实现 `FlowSQLiteOpenHelper` 的构造函数：

```
public FlowSQLiteOpenHelper(BaseDatabaseDefinition flowManager, Dat
```

自定义的Helper类需要在 `@Database` 注解中通过 `sqlHelperClass()` 参数声明使用。

Model & Creation

所有的标准Table类都必须带上 `@Table` 注解并实现 `Model` 接口。为了方便，我们提供了 `Model` 接口的标准实现 `BaseModel`。

Table数据类中支持的数据类型：

1. 所有Java标准的数据类型

(`boolean` 、 `byte` 、 `short` 、 `int` 、 `long` 、 `float` 、 `double` 等)及相应的包装类，以及 `String` ，当然我们还默认提供了对 `java.util.Date` 、 `java.sql.Date` 与 `Calendar` 的支持。
2. 其他不被支持的类型可以通过自定义 `TypeConverter` 来提供支持。范型类型的对象不被支持，容器类型如List、Map等同样也是不推荐的。如果你一定要使用容器，你也只能在无法获得范型参数情况下使用。
3. 支持复合主键。
4. 通过 `@ForeignKey` 可以嵌套其他的 `Model` ，即一对一的关系。
5. 任何被作为 `@ForeignKey` 使用的 `Model` 都必须是 `ModelContainer` 。
6. 支持多个 `@ForeignKey`
7. 通过在列属性中显式声明 `@Column(typeConverter = SomeTypeConverter.class)` 来提供自定义的 `TypeConverter` 。

表**Model**的规则：

1. 所有的 `Model` 都必须有一个公共无参构造。在查询时我们将会使用到这个构造函数。
2. 对于一个继承自另一个 `Model` 的子类，library将会收集全部被 `@Column` 注解标记的属性（包括父类中的）。
3. 默认情况下为了方便，我们使用属性变量名作为数据表的列名，当然你也可以通过 `@Column` 中的参数来自定义列名。
4. 为了使 `_Adapter` 类可以访问，属性必须是 `public` 或者包私有的。*NOTE*:Package private fields need not be in the same package as DBFlow will generate the necessary access methods to get to them.
5. 属性如果被定义为 `private` ，就必须提供对应的getter和setter方法，命名规则为：对属性 `{name}` ，必须有 `get{Name}()` 和 `set{Name}(columnType)` 方法。
6. 所有的Model类都必须是可访问的。内部类在3.0.0-beta1+后被支持。

Model示例

这里是一个 `Model` 类的示例，其中包含一个主键（Table中必须有至少一个主键）和一个其他属性。

```
@Table(database = AppDatabase.class)
public class TestModel extends BaseModel {

    // All tables must have a least one primary key
    @PrimaryKey
    String name;

    // By default the column name is the field name
    @Column
    int randomNumber;

}
```

高级Table特性

为特定列指定自定义类型转换器

从3.0版本开始，你可以为特定列指定自定义的类型转换器 `TypeConverter` ：

```
@Column(typeConverter = SomeTypeConverter.class)
SomeObject someObject;
```

这个自定义转换器将会覆盖默认的转换器和访问方法（除了私有属性，对于私有属性，自定义转换器将会拦截访问器方法）。

设置所有属性都作为列

我们可以通过设置 `@Table(allFields = true)` 将所有类中的属性都设置为列。当打开这个设置后，DBFlow会将所有public/package private、non-final、non-static属性都作为 `@Column` 。当然，这时你仍然需要显式地设置至少一个 `@PrimaryKey` 主键。

私有属性列

正如上面所说，你需要将作为 `@Column` 的属性都设为 `public` 或package private，而如果一定要使用私有属性则需要提供相应的getter与setter方法：

```
@Table(database = TestDatabase.class)
public class PrivateModelTest extends BaseModel {

    @PrimaryKey
    private String name;


    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

访问器方法的命名方式按照java标准的命名方式，所以 `boolean` 类型可以使用“is”：

```
@Table(database = TestDatabase.class, useIsForPrivateBooleans = tru
public class PrivateModelTest extends BaseModel {

    @PrimaryKey
    private String name;

    @Column
    private boolean selected;

    public boolean isSelected() {
      return selected;
    }

    public void setSelected(boolean selected) {
      this.selected = selected;
    }

    //... etc
}
```

UNIQUE 约束

在SQLite中我们可以定义UNIQUE约束，即对于列或者列集合其在这个表中一定是唯一的。

```
UNIQUE('name', 'number') ON CONFLICT FAIL, UNIQUE('name', 'address'
```

与SQL语句相似，DBFlow中我们是这样定义UNIQUE约束的：

```
@Table(database = AppDatabase.class,
  uniqueColumnGroups = {@UniqueGroup(groupNumber = 1, uniqueConflic
                        @UniqueGroup(groupNumber = 2, uniqueConflic
public class UniqueModel extends BaseModel {

  @PrimaryKey
  @Unique(unique = false, uniqueGroups = {1,2})
  String name;

  @Column
  @Unique(unique = false, uniqueGroups = 1)
  String number;

  @Column
  @Unique(unique = false, uniqueGroups = 2)
  String address;

}
```

多数据库Module的处理

DBFlow会生成一个 `GeneratedDatabaseHolder` 类，里面描述了所有的数据库类、数据表和其他与库进行交互的代码。

这个类的名称是固定的，但这也意味着可能会出现这样的情况：当我们导入其他库或者子工程时，如果它也是使用DBFlow管理数据库的话，生成的这个类就会冲突。以前的版本中，DBFlow在遇到这种情况时就会报错。

为了处理这种情况，我们需要对生产器进行一点小小的设置。幸运的是，这种处理非常简单。

如果我们需要导入一个产生冲突的Module，首先需要配置一下它的 `build.gradle` 文件，定义一下 `apt` 的参数：

```
apt {
    arguments {
        targetModuleName 'Test'
    }
}
```

通过设置 `targetModuleName` 参数，生成器将会把 `GeneratedDatabaseHolder` 类重命名为 `GeneratedDatabaseHolderTest`，这样类名就不会产生冲突了。

而在被导入的app中我们首先需要用通常的方法初始化我们的数据库类 `GeneratedDatabaseHolder` ：

```
public class ExampleApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        FlowManager.init(this);
    }
}
```

最后，由于默认方法只会载入默认命名的Holder类，所以我们还需要手动载入那个被重命名的Holder类 `GeneratedDatabaseHolderTest` ：

```
FlowManager.initModule(GeneratedDatabaseHolderTest.class);
```

这个方法被执行多次也不会有异常的影响，因为其内部是通过Map来进行管理的，以保证只会被载入一次。

SQL操作的封装类

在Android中对数据库的操作很多时候还是得用SQL语句，手写SQL语句的操作是繁琐、难看而又容易出错的。为了使对数据库的操作更加简单好用，我们对数据库操作进行了封装，同时尽量保证了与原生SQL语法的相似，使得更容易上手。

在第一节中，我们将演示封装类是如何大大简化你的代码的。

一个例子

比如我们想要从 `Ant` 中选出所有为“worker”的雌蚂蚁。这个操作的SQL语句很简单：

```
SELECT * FROM Ant where type = 'worker' AND isMale = 0;
```

而如果要将数据库里面的数据转为我们可用的数据，在Android中我们需要写这些代码：

```
String[] args = new String[2];
args[0] = "worker";
args[1] = "0";
Cursor cursor = db.rawQuery("SELECT * FROM Ant where type = ? AND i
final List<Ant> ants = new ArrayList<Ant>();
Ant ant;

if (cursor.moveToFirst()) {
  do {
    // get each column and then set it on each
    ant = new Ant();
    ant.setId(cursor.getLong(cursor.getColumnIndex("id")));
    ant.setType(cursor.getString(cursor.getColumnIndex("type")));
    ant.setIsMale(cursor.getInt(cursor.getColumnIndex("isMale") ==
    ant.setQueenId(cursor.getLong(cursor.getColumnIndex("queen_id")
    ants.add(ant);
  }
  while (cursor.moveToNext());
}
```

这本是一个简单的查询，但为什么我们还需要写这么多无用的代码？

想想在这些情况：

1. 添加或删除列
2. 为了在操作其他表、进行其他的查询或者支持其他的数据类型，我们还不得不写下更多的废代码。

我们希望这些操作的代码能变得短小，但同时又能方便维护和富于表现力(一下就能看懂做了哪些操作)。DBFlow正是为你提供了这些：

```
// 同步执行操作
List<Ant> devices = SQLite.select().from(Ant.class)
  .where(Ant_Table.type.eq("worker"))
  .and(Ant_Table.isMale.eq(false)).queryList();

// 异步执行操作
  SQLite.select()
  .from(DeviceObject.class)
  .where(Ant_Table.type.eq("worker"))
  .and(Ant_Table.isMale.eq(false))
  .async().queryList(transactionListener);
```

DBFlow支持许多的查询操作：

1. SELECT
2. UPDATE
3. INSERT
4. DETELE
5. JOIN

SELECT操作和事务

Select用于从数据库中查找数据。我们还可以使用 `TransactionManager` 来进行事务（对于大量操作更推荐使用事务）

```
// Query a List
SQLite.select().from(SomeTable.class).queryList();
SQLite.select().from(SomeTable.class).where(conditions).queryList()

// Query Single Model
SQLite.select().from(SomeTable.class).querySingle();
SQLite.select().from(SomeTable.class).where(conditions).querySingle

// Query a Table List and Cursor List
SQLite.select().from(SomeTable.class).where(conditions).queryTableL
SQLite.select().from(SomeTable.class).where(conditions).queryCursor

// Query into a ModelContainer!
SQLite.select().from(SomeTable.class).where(conditions).queryModelC

// SELECT methods
SQLite.select().distinct().from(table).queryList();
SQLite.select().from(table).queryList();
SQLite.select(Method.avg(SomeTable_Table.salary))
  .from(SomeTable.class).queryList();
SQLite.select(Method.max(SomeTable_Table.salary))
  .from(SomeTable.class).queryList();

// Transact a query on the DBTransactionQueue
TransactionManager.getInstance().addTransaction(
  new SelectListTransaction<>(new Select().from(SomeTable.class).wh
  new TransactionListenerAdapter<List<SomeTable>>() {
    @Override
    public void onResultReceived(List<SomeTable> someObjectList) {
      // retrieved here
});

// Selects Count of Rows for the SELECT statment
long count = SQLite.selectCountOf()
  .where(conditions).count();
```

Oeder By

```
// true为'ASC'正序, false为'DESC'反序
SQLite.select()
  .from(table)
  .where()
  .orderBy(Customer_Table.customer_id, true)
  .queryList();
```

Group By

```
SQLite.select()
  .from(table)
  .groupBy(Customer_Table.customer_id, Customer_Table.customer_name
  .queryList();
```

Having

```
SQLite.select()
  .from(table)
  .groupBy(Customer_Table.customer_id, Customer_Table.customer_name
  .having(Customer_Table.customer_id.greaterThan(2))
  .queryList();
```

LIMIT + OFFSET (分页)

```
SQLite.select()
  .from(table)
  .limit(3)
  .offset(2)
  .queryList();
```

UPDATE语句

有两种方式更新数据：

1. 使用 `SQLite.update()` 方法或者 `Update` 类
2. 使用 `TransactionManager` 运行事务（尽管是异步的，但是线程安全的）

在这一节，我们将展示批量更新数据。

在之前的例子中，我们想要将所有的“worker”雄蚂蚁变为“other”，因为这些蚂蚁们变懒了都不想工作了（笑）。

SQL语句描述为：

```
UPDATE Ant SET type = 'other' WHERE male = 1 AND type = 'worker';
```

DBFlow中这样进行这个操作：

```
// Native SQL wrapper
Where<Ant> update = SQLite.update(Ant.class)
  .set(Ant_Table.type.eq("other"))
  .where(Ant_Table.type.is("worker"))
    .and(Ant_Table.isMale.is(true));

//译者注：queryClose()方法在3.0.0beta2中并没有找到，可能是一个过时的方法。
update.queryClose();

// TransactionManager (more methods similar to this one)
TransactionManager
        .getInstance()
        .addTransaction(
                new QueryTransaction(
                        DBTransactionInfo.create(BaseTransaction.PR
                update);
```

DELETE语句

```
// Delete a whole table
Delete.table(MyTable.class, conditions);

// Delete multiple instantly
Delete.tables(MyTable1.class, MyTable2.class);

// Delete using query
SQLite.delete(MyTable.class)
  .where(DeviceObject_Table.carrier.is("T-MOBILE"))
    .and(DeviceObject_Table.device.is("Samsung-Galaxy-S5"))
  .query();
```

JOIN语句

JOIN example

`JOIN` 语句对于多对多关系是一个很好的操作符。

比如我们有一个 `Customer` 表和一个 `Reservations` 表：

```
SELECT FROM `Customer` AS `C` INNER JOIN `Reservations` AS `R` ON
```

```
// 由于查询结果可能没有合适的Table类可以容纳，所以可以通过`@QueryModel`注解
List<CustomTable> customers = new Select()
  .from(Customer.class).as("C")
  .join(Reservations.class, JoinType.INNER).as("R")
  .on(Customer_Table.customerId
      .withTable(new NameAlias("C"))
    .eq(Reservations_Table.customerId.withTable("R"))
    .queryCustomList(CustomTable.class);
```

对于 `IProperty.withTable()` 方法，我们既可以像上面那样使用 `NameAlias` 别名，也可以直接使用表示关系，例如上面的例子可简化为：

```
SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
      ON COMPANY.ID = DEPARTMENT.EMP_ID
```

DBFlow：

```
SQLite.select(Company_Table.EMP_ID, Company_Table.DEPT)
  .from(Company.class)
  .leftOuterJoin(Department.class)
  .on(Company_Table.ID.withTable().eq(Department_Table.EMP_ID.withT
  .queryList();
```

Property与查询条件

DBFlow中会自动生成与你的 `Model` 类相对应的 `_Table` 类。其中是将 `Model` 类中的属性都转为了 `Property` 。 `Property` 是一种类型安全的处理方式，方便我们使用 `条件` 。

在SQL中 `Condition` 条件语句类似于这样：

```
name = 'Test'

name = `SomeTable`.`Test`

name LIKE '%Test%'

name != 'Test'

salary BETWEEN 15000 AND 40000

name IN('Test','Test2','TestN')

((`name`='Test' AND `rank`=6) OR (`name`='Bob' AND `rank`=8))
```

如何描述条件

我们会自动生成条件语句所需要的 `Property` 。

例如现在我们有一张简单的数据表：

```
@Table(database = TestDatabase.class, name = "TestModel32")
public class TestModel3 {

    @Column
    @PrimaryKey
    public String name;

    @Column
    String type;
}
```

这时DBFlow会生成相应的 `TestModel3_Table` 类：

```
public final class TestModel3_Table {
  public static final Property<String> name = new Property<String>(

  public static final Property<String> type = new Property<String>(

  public static BaseProperty getProperty(String columnName) {
    columnName = QueryBuilder.quoteIfNeeded(columnName);
    switch (columnName)  {
      case "`name`":  {
        return name;
      }
      case "`type`":  {
        return type;
      }
      default:  {
        throw new IllegalArgumentException("Invalid column name pas
      }
    }
  }
}
```

这样我们就可以安全、方便地使用 `Property` 来描述我们的条件了：

```
TestModel3_Table.name.is("Test"); // name = 'Test'
TestModel3_Table.name.withTable().is("Test"); // TestModel3.name =
TestModel3_Table.name.like("%Test%")
```

对于 `Property`，我们提供了以下操作符的支持：

1. `is()`, `eq()` -> =
2. `isNot()`, `notEq()` -> !=
3. `isNull()` -> IS NULL / `isNotNull()` IS NOT NULL
4. `like()`, `glob()`
5. `greaterThan()`, `greaterThanOrEqual()`, `lessThan()`, `lessThanOrEqual()`
6. `between()` -> BETWEEN
7. `in()`, `notIn()`

组合条件ConditionGroup

`ConditionGroup` 是 `ConditionQueryBuilder` 的继任者，但它相比于 `QueryBuilder` 是有缺陷的。

`ConditionGroup` 可以作为 `Condition` 使用，也可以用于组合其他一些 `Condition`。

The `ConditionGroup` is the successor to the `ConditionQueryBuilder`. It was flawed in that it conformed to `QueryBuilder`, yet contained `Condition`, and required a type-parameter that referenced the table it belonged in.

`ConditionGroup` are arbitrary collections of `Condition` that can combine into one SQlite statement *or* be used as `Condition` within another `ConditionGroup`.

Any Sqlite wrapper class that takes in multiple `Condition` use this class to construct the query for it.

```
SQLite.select()
  .from(MyTable.class)
  .where(MyTable_Table.someColumn.is("SomeValue"))
  .and(MyTable_Table.anotherColumn.is("ThisValue"));

  // 或者
  SQLite.select()
    .from(MyTable.class)
    .where(ConditionGroup.clause()
      .and(MyTable_Table.someColumn.is("SomeValue")
      .or(MyTable_Table.anotherColumn.is("ThisValue"));
```

事务管理器

`TransactionManager` 事务管理器是用于管理批量数据库操作的类。对于恢复在大量数据更新、保存或删除时的数据非常有用。它同样被包装为了本地方法，使你能够非常简单地使用事务。这个类不仅提供了同步执行的方法，也支持异步执行，防止UI线程被大量数据库操作堵塞。

`TransactionManager` 主要使用 `DBTransactionQueue` 。这个队列(Queue)基于Volley框架的 `VolleyRequestQueue` ， `VolleyRequestQueue` 使用的是 `PriorityBlockingQueue` 。这种队列采用优先级的方式对数据库的各个事务进行排序，主要有这些优先级：

1. **UI**: 保证事务能够被立刻执行以显式到UI中。
2. **HIGH**: 保留的任务，将影响用户的互动。比如在未来显示数据。
3. **NORMAL**: `Transaction` 的默认优先级，很适合在app不需要立刻得到结果的情况下。
4. **LOW**: 低优先级，适合一些不必要的事务。

`DBTransactionInfo` ：用于保存 `DBTransactionQueue` 如何处理 `BaseTransaction` 的一些信息，包含名称和优先级。名称是为了在Debug的时候能更容易区分各个事务。

优先级在内部是一个整数值，所以你可以自定义事务的优先级、或者是自定义优先级的大小。

在高级的使用中，你可以使用 `TableTransactionManager` 或者继承 `TransactionManager` 来自定义自己的 `DBTransactionQueue` ，这样你就可以使用自定义的事务管理类更灵活地管理事务了。

批量操作

将大量数据保存到数据库而不阻塞主线程，在以前我们是这样做的：

```
new Thread() {
  @Override
  public void run(){
      database.beginTransaction();
      try {
          for(ModelClass model: models){
            // save model here
          }
          database.setTransactionSuccessful();
      } finally {
          database.endTransaction();
      }
  }
}.start();
```

而现在我们只需要使用更为简单的方式即可完成：

```
TransactionManager.getInstance().addTransaction(new SaveModelTransa
```

Model的批量处理

`ProcessModelInfo` ：描述保存了事务相关的信息，比如包括 `DBTransactionInfo` 、表、一些Model和 `TransactionListener` 事务回调。

NOTE：这个类可以通过 `TransactionManager` 中的方法很简单的创建。

对于大量的操作更好的方式是通过 `DBTransactionQueue` 来执行，这样能有效优化由多线程同步或主线程操作带来的阻塞。

大量 `save()` 操作，首选的方法是 `DBBatchSaveQueue` ，它将会运行一个 `DBTransaction` 并一次性填满（默认情况下是50个Models，但也可以设置）。而对于少量的保存操作，更好的方式是定期保存。

示例

```
    ProcessModelInfo<SomeModel> processModelInfo = ProcessModelInfo<S



    TransactionManager.getInstance().saveOnSaveQueue(models);

    // 或者添加到队列
    TransactionManager.getInstance().addTransaction(new SaveModelTran

    // Updating only updates on the ``DBTransactionQueue``
    TransactionManager.getInstance().addTransaction(new UpdateModelLi

    TransactionManager.getInstance().addTransaction(new DeleteModelLi
```

`TransactionManager` 中还有各种各样的方法对 `DBTransactionQueue` 进行排序和操作。

接收结果

`SelectListTransaction` 和 `SelectSingleModelTransaction` 将会在 `DBTransactionQueue` 中产生一个SELECT操作。当它们的操作完成将会在UI线程回调 `TransactionListener` 函数。一个普通的SELECT操作则会在当前线程完成，对于简单的查询操作这样也没问题，但更好的方式是通过 `DBTransactionQueue` 事务来完成，以减少对主线程的阻塞。

```
    // 获取表中的所有数据
    // 你也可以加入条件语句进行更为复杂的查询
    TransactionManager.getInstance().addTransaction(new SelectListTra
       @Override
      public void onResultReceived(List<TestModel> testModels) {
          // 这里会在UI线程被执行
      }
    }, TestModel.class, condition1, condition2,..);
```

自定义事务类

DBFlow也支持你很简单地创建自定义的事务类，通过以下方法即可将它们添加到 `TransactionManager` 中执行：

```
TransactionManager.getInstance().addTransaction(myTransaction);
```

这里是一些你可以创建的自定义事务：

- 继承 `BaseTransaction` 类，你可以在 `onExecute()` 方法中定义你要完成的事情：

```
BaseTransaction<TestModel1> testModel1BaseTransaction = new Bas
  @Override
  public TestModel1 onExecute() {
      // do something and return an object
      return testModel;
  }
};
```

- `BaseResultTransaction` 类则加入了一个简单的 `TransactionListener` 回调函数，让你可以监听到事务的更新状态：

```
BaseResultTransaction<TestModel1> baseResultTransaction = new B
  @Override
  public TestModel1 onExecute() {
      return testmodel;
  }
};
```

- `ProcessModelTransaction` 让你能够定义如何处理 `DBTransactionQueue` 中的每一个model：

```
public class CustomProcessModelTransaction<ModelClass extends M

  public CustomProcessModelTransaction(ProcessModelInfo<ModelCl
      super(modelInfo);
  }

  @Override
  public void processModel(ModelClass model) {
     // process model class here!
  }
}
```

- `QueryTransaction` 让你能够使用 `Queriable` 操作检索数据库。

```
// any Where, From, Insert, Set, and StringQuery all are valid
TransactionManager.getInstance().addTransaction(new QueryTransa
SQLite.delete().from(MyTable.class).where(MyTable_Table.name.is
```

类型转换器

类型转换器使你可以在 `Model` 中使用默认未被支持的类型。你可以自定义属性值如何被写入到数据库，也可以自定义如何被从数据库中载入。

NOTE： `TypeConverter` 只能支持普通的 `@Column` ，而不能有 `@PrimaryKey` 或 `@ForeignKey` 等非限定性的数据。

一个类型转换器对所有数据库都是可用的。

如果我们指定的类型为一个 `Model` ，则在作为 `Column.FOREIGN_KEY` 时，可能会有一些不确定的行为。

作为示例，这里我们创建了一个 `LocationConverter` ，用于将Locations类型转换为String：

```
// 第一个范型代表被保存到数据库的数据类型
// 第二个范型表示Model中属性的类型
@com.raizlabs.android.dbflow.annotation.TypeConverter
public class LocationConverter extends TypeConverter<String,Locat

  @Override
  public String getDBValue(Location model) {
      return model == null ? null : String.valueOf(model.getLatit
  }

  @Override
  public Location getModelValue(String data) {
      String[] values = data.split(",");
      if(values.length < 2) {
          return null;
      } else {
          Location location = new Location("");
          location.setLatitude(Double.parseDouble(values[0]));
          location.setLongitude(Double.parseDouble(values[1]));
          return location;
      }
  }
}
```

通过自定义 `LocationConverter` 转换器，现在我们可以在table中使用 `Location` 类型的属性了：

```
@Table(...)
public class SomeTable extends BaseModel {


  @Column
  Location location;

}
```

为列指定类型转换器

在3.0之后，列的类型转换器需要被显式指定：

```
@Table(...)
public class SomeTable extends BaseModel {


  @Column(typeConverter = SomeTypeConverter.class)
  Location location;

}
```

NOTE: `enum` 枚举类型默认并不被支持。如果你需要使用，必须要自定义类型转换器。

强大的模型缓存机制

在DBFlow库中，模型缓存被设计得非常简单，同时又具有高度可扩展性、可访问性和可用性。

`ModelCache` 是一个接口，可被用于SQLite检索、 `FlowQueryList` 、 `FlowCursorList` 或者任何需要缓存的地方。

在数据表中开启缓存

通过在 `@Table` 注解中设置 `cachingEnabled = true` 我们就可以很简单地开启缓存。在复合主键 `@PrimaryKey` 的表中，我们还需要定义一个 `@MultiCacheField` 对象（之后会进行讲解）

当我们检索数据库，模型数据对象将被缓存下来，DBFlow会高效地管理这些缓存。

这些是被支持的常见的缓存管理方式：

1. `ModelLruCache` -> 使用LruCache算法，会按照既定的规则添加和释放缓存。
2. `SimpleMapCache` -> 会被缓存到一个预定义大小的 `Map` (默认为 `HashMap`)
3. `SparseArrayBasedCache` -> 底层是一个int->object的 `SparseArray` 稀疏数组。它用于缓存原生数字类型及其包装类(比如Integer、Double、Long等)、或者返回数字类型的 `MulitCacheConverter` 转换器。

NOTE： 如果你执行了 `SELECT` 操作，它将会在生成全部副本之前缓存部分数据，在这里推荐载入全部数据，因为缓存能解决大部分的性能问题。
（if you run a `SELECT` with columns specified, it may cache partial `Model` classes if they're loaded before the full counterparts. It is highly recommended to just load the full models in this case, since the caching mechanism will make up most efficiency problems.）

默认情况下，缓存使用的是 `SimpleMapCache` ，你可以在 `@Table` 注解中通过指定 `cacheSize()` 参数来设定默认的缓存大小。注意，当我们特别指定了自定义缓存后，这个参数将变得无效。

使用一个自定义缓存，只需要在我们的 `Model` 类中添加 `@ModelCacheField` 属性：

```
@ModelCacheField
public static ModelCache<CacheableModel3, ?> modelCache = new Simpl
```

`@ModelCacheField` 属性必须是 `public static`

在3.0版本之后，BDFlow对于从缓存中载入的 `@ForeignKey` 会进行更加智能的处理。

缓存如何工作

1. 每个 `@Tabel` / `Model` 类都有自己的缓存，表与表直接、子类与父类直接并不会共享缓存。
    i. 它们会拦截对数据库的检索操作，并将其中的数据缓存下来(后面会详解)
    ii. 使用任何的 `Insert` 、 `Update` 或 `Delete` 方法时，其中的对象都会被直接缓存而不会更新（考虑到性能），。如果你想要强制刷新缓存则可以使用 `FlowManager.getModelAdapter(MyTable.class).getModelCache().clear()` 方法使缓存失效。
    iii. 直接修改从缓存中得到的对象会导致缓存本身被直接修改，由于在使用 `save()` 、 `insert()` 、 `update()` 等方法之后对象数据才会被写入数据库，所以这可能会导致数据的不一致。

当我们执行一个检索操作，DBFlow将会：

1. 运行检索语句，并从数据库得到一个 `Cursor` 的结果
2. 从 `Cursor` 中取出主键的值。
3. 如果这个键值在缓存中，则：
    i. 刷新关系数据（如 `@ForeignKey` ）。*TIP*：将关系所对应的表启动缓存能使这个操作变得更快。
    ii. 返回缓存的对象
4. 如果这个键值不存在，则说明此对象没有被缓存，这时我们会从数据库继续载入此对象的其他数据并将之缓存下来（直到该缓存失效）

复合主键的缓存处理

在3.0版本开始，DBFlow支持对复合主键的缓存。这需要 `Model` 类显式定义至少一个 `@MultiCacheField` 属性：

```
@Table(database = TestDatabase.class, cachingEnabled = true)
public class MultipleCacheableModel extends BaseModel {

    @MultiCacheField
    public static IMultiKeyCacheConverter<String> multiKeyCacheMode

        @Override
        @NonNull
        public String getCachingKey(@NonNull Object[] values) { //
            return "(" + values[0] + "," + values[1] + ")";
        }
    };

    @PrimaryKey
    double latitude;

    @PrimaryKey
    double longitude;

    @ForeignKey(references = {@ForeignKeyReference(columnName = "as
            columnType = String.class, foreignKeyColumnName = "name
    TestModel1 associatedModel;

}
```

`MultiCacheField` 里的返回值可以是任意类型，只要与你所定义 `ModelCache` 的Key类型一致即可。

FlowCursorList + FlowQueryList

`FlowCursorList` 与 `FlowQueryList` 使用独立的 `ModelCache` 缓存，并不与 `@Tabel` / `Model` 类的缓存通用。可以通过重写方法自定义它们的缓存方式：

```
@Override
protected ModelCache<? extends BaseCacheableModel, ?> getBackingCac
        return new MyCustomCache<>();
}
```

自定义缓存

当然，你甚至可以完全自定义自己的缓存算法。

比如，这是内置的 `LruCache` 缓存算法的代码：

```
public class ModelLruCache<ModelClass extends Model> extends ModelC

    public ModelLruCache(int size) {
        super(new LruCache<Long, ModelClass>(size));
    }

    @Override
    public void addModel(Object id, ModelClass model) {
        if(id instanceof Number) {
            synchronized (getCache()) {
                Number number = ((Number) id);
                getCache().put(number.longValue(), model);
            }
        } else {
            throw new IllegalArgumentException("A ModelLruCache mus
                                               "a Number to convert
        }
    }

    @Override
    public ModelClass removeModel(Object id) {
        ModelClass model;
        if(id instanceof Number) {
            synchronized (getCache()) {
                model = getCache().remove(((Number) id).longValue()
            }
        }  else {
```

```
                throw new IllegalArgumentException("A ModelLruCache use
                                                   "a Number to convert
            }
            return model;
        }

        @Override
        public void clear() {
            synchronized (getCache()) {
                getCache().evictAll();
            }
        }

        @Override
        public void setCacheSize(int size) {
            getCache().resize(size);
        }

        @Override
        public ModelClass get(Object id) {
            if(id instanceof Number) {
                return getCache().get(((Number) id).longValue());
            } else {
                throw new IllegalArgumentException("A ModelLruCache mus
                                                   "a Number to convert
            }
        }
    }
```

生成Content Provider

BDFlow甚至可以很简单地生成 `Content Provider` 。

开始

此功能主要使用了schematic库。

ContentProvider描述类

为了生成一个 `ContentProvider` ，我们首先需要定义一个描述类：

```
@ContentProvider(authority = TestContentProvider.AUTHORITY,
        database = TestDatabase.class,
        baseContentUri = TestContentProvider.BASE_CONTENT_URI)
public class TestContentProvider {

    public static final String AUTHORITY = "com.raizlabs.android.db

    public static final String BASE_CONTENT_URI = "content://";

}
```

当然你也可以在任何类上通过注解将其定义为ContentProvider的描述类。推荐的一个写法就是将 `@Database` 描述类同时作为 `@ContentProvider` 的描述类。它们同时注解于同一个类上是完全没问题的：

```
@ContentProvider(authority = TestDatabase.AUTHORITY,
        database = TestDatabase.class,
        baseContentUri = TestDatabase.BASE_CONTENT_URI)
@Database(name = TestDatabase.NAME, version = TestDatabase.VERSION)
public class TestDatabase {

    public static final String NAME = "TestDatabase";

    public static final int VERSION = "1";

    public static final String AUTHORITY = "com.raizlabs.android.db

    public static final String BASE_CONTENT_URI = "content://";

}
```

添加到Manifest中

ContentProvider将会被生成为 `类名$Provider` ，在使用ContentProvider的其他App或本App中我们需要在 `AndroidManifest.xml` 文件中声明：

```
<provider
    android:authorities="com.raizlabs.android.dbflow.test.provider"
    android:exported="true|false"
    android:name=".provider.TestContentProvider$Provider"/>
```

`android:exported` ：设定改服务是否可以跨进程(App)使用，为true时其他应用才可以使用此服务。

注意，你必须有至少一个 **`@TableEndpoint`** 以确保它能通过编译检测

对数据添加Endpoints

有两种方式定义 `@TableEndpoint` ：

1. 在 `@ContentProvider` 描述类中创建一个内部类。

2. 在 `@Table` 类中指定content provider的类名

`@TableEndpoint` ：将 `ContentProvider` 中的检索、删除和更新的操作映射到本地数据库。

一些建议：

1. 如果使用 `@ContentProvider` 内部类作为 `@TableEndpoint` ，其类名最好与被引用的表同名。
2. 创建一个 `public static final String ENDPOINT = "{tableName}"` 属性，使之可以在其他地方被引用。
3. 创建一个 `buildUri()` 方法(如下)，方便创建URI

示例：

```
@TableEndpoint(ContentProviderModel.ENDPOINT)
public static class ContentProviderModel {

    public static final String ENDPOINT = "ContentProviderModel";

    private static Uri buildUri(String... paths) {
        Uri.Builder builder = Uri.parse(BASE_CONTENT_URI + AUTHORIT
        for (String path : paths) {
            builder.appendPath(path);
        }
        return builder.build();
    }

    @ContentUri(path = ContentProviderModel.ENDPOINT,
            type = ContentUri.ContentType.VND_MULTIPLE + ENDPOINT)
    public static Uri CONTENT_URI = buildUri(ENDPOINT);

}
```

或者对已有的数据表进行设置：

```
@TableEndpoint(name = ContentProviderModel.NAME, contentProviderNar
@Table(database = ContentDatabase.class, tableName = ContentProvide
public class ContentProviderModel extends BaseProviderModel<Content
```

```
    public static final String NAME = "ContentProviderModel";

    @ContentUri(path = NAME, type = ContentUri.ContentType.VND_MULT
    public static final Uri CONTENT_URI = ContentUtils.buildUri(Cor

    @Column
    @PrimaryKey(autoincrement = true)
    long id;

    @Column
    String notes;

    @Column
    String title;

    @Override
    public Uri getDeleteUri() {
        return TestContentProvider.ContentProviderModel.CONTENT_URI
    }

    @Override
    public Uri getInsertUri() {
        return TestContentProvider.ContentProviderModel.CONTENT_URI
    }

    @Override
    public Uri getUpdateUri() {
        return TestContentProvider.ContentProviderModel.CONTENT_URI
    }

    @Override
    public Uri getQueryUri() {
        return TestContentProvider.ContentProviderModel.CONTENT_URI
    }
}
```

之后的章节会更详细地介绍 `@ContentUri` 的使用方法。

更方便地创建ContentProvider

这里还有两种更方便的方式将你的 `Model` 改为支持ContentProvider，只需要继承下面两个类即可：

1. `BaseProviderModel` ：重写了 `Model` 中的所有方法使之可以用于 `ContentProvider`
2. `BaseSyncableProviderModel` ：与上面相同，但它将会把数据的更新与本地的数据更新及时同步。

使用Content Provider

你可以通过 `ContentUtils` 方便地使用 `ContentProvider` ：

```
ContentProviderModel contentProviderModel = ...; // some instance

int count = ContentUtils.update(getContentResolver(), ContentProvid

Uri uri = ContentUtils.insert(getContentResolver(), ContentProvider

int count = ContentUtils.delete(getContentResolver(), someContentUr
```

推荐：通过继承 `BaseSyncableProviderModel` 来使本地数据库保持一致，除非使用 `BaseProviderModel` 就足够了。

```
MyModel model = new MyModel();
model.id = 5;
model.load(); // queries the content provider

model.someProp = "Hello"
model.update(false); // runs an update on the CP

model.insert(false); // inserts the data into the CP
```

高级用法

通知方法

你可以通过定义 `@Notify` 方法监听 `ContentProvider` 中指定的操作，并通过返回 `Uri[]` 来通知相应的 `ContentResolver` 。

支持监听：

1. Update
2. Insert
3. Delete

Example:

```
@Notify(method = Notify.Method.UPDATE,
paths = {}) // specify paths that will call this method when specif
public static Uri[] onUpdate(Context context, Uri uri) {

  return new Uri[] {
    // return custom uris here
  };
}
```

高级ContentUri

Path Segments

通过 `#` 符号，你可以定义高级 `path` 匹配：

path:

```
path = "Friends/#/#"
```

segments:

```
segments = {@PathSegment(segment = 1, column = "id"),
    @PathSegment(segment = 2, column = "name")}
```

组合起来：

```
@ContentUri(type = ContentType.VND_MULTIPLE,
path = "Friends/#/#",
segments = {@PathSegment(segment = 1, column = "id"),
    @PathSegment(segment = 2, column = "name")})
public static Uri withIdAndName(int id, String name) {
  return buildUri(id, name);
}
```

DBFlow使你的数据迁移变得异常简单

当你升级数据库 `@Database` 版本，只需要添加一个 `Migration` 来配置升级操作。

默认的 `Migration` 操作会自动载入 `/assets/migrations/{DatabaseName}/{versionName.sql}` 文件进行数据库升级。

如果你想要在第一次创建时对数据库进行初始化操作，只需要使用版本号 **0**！

NOTE：如果你使用任意一个Migration子类比如 `AlterTableMigration`、`UpdateTableMigration` 或 `IndexMigration`，你都应该只重写 `onPreMigrate()` 方法，并且在其中调用 `super.onPreMigrate()`，里面需要进行一些必要的实例化操作。

NOTE：所有的 `Migration` 都应该只有一个公共无参构造函数。

Migration类

作为一个基本的类，`BaseMigration` 是一个非常简单的类，让你定义自己的迁移操作：

```
@Migration(version = 2, database = AppDatabase.class)
public class Migration1 extends BaseMigration {

    @Override
    public void migrate(DatabaseWrapper database) {

    }
}
```

添加列

这里举一个简单的例子，我们需要向已有的数据表中添加一个列。

这里是原来数据表的类：

```
@Table
public class TestModel extends BaseModel {

    @Column
    @PrimaryKey
    String name;

    @Column
    int randomNumber;
}
```

现在我们想要向这个数据表里面添加一列，这里有两种方式：

- 写SQL语句 `ALTER TABLE TestModel ADD COLUMN timestamp INTEGER;` 到文件 `{dbVersion.sql}` ，并放到assets的指定目录中，DBFlow将会自动执行。
- 使用 `Migration` 类：

```
@Migration(version = 2, database = AppDatabase.class)
public class Migration1 extends AlterTableMigration<TestModel>

  @Override
  public void onPreMigrate() {
    // Simple ALTER TABLE migration wraps the statements into a
    addColumn(Long.class, "timestamp");
  }
}
```

升级列

```
@Migration(version = 2, database = AppDatabase.class)
public class Migration1 extends UpdateTableMigration<TestModel> {

    @Override
    public void onPreMigrate() {
      // UPDATE TestModel SET deviceType = "phablet" WHERE screenSi
      set(TestModel_Table.deviceType.is("phablet"))
        .where(TestModel_Table.screenSize.greaterThan(5.7), TestMod
    }
}
```

Model Containers

模型容器(`Model Containers`)是实际 `Model` 类的一种镜像。对于从一个Map、JSON字符串或其他数据转换得到的 `Model` 对象，模型容器会在性能上对其进行优化：保存数据时并不会保存 `Model` 对象而是直接保存其来源数据本身，因为去掉了转换这一层，效率会很高。

但请注意它的一些限制：

1. 它们必须引用在相同数据库中已定义的数据表 `Model` 。
2. 你无法对它们使用 `SELECT` 操作，因为它们并不是一个已存在的表，只是提供了一些方便的方法使它们看起来像一个 `Model` 。
3. 请确保被引用的表是正确的，因为它本身无法辨别表是否正确。
4. 对于JSON/Map/Object，它并不是一个全功能的语法分析器，它只是一个简单、好用的数据容器。

有趣的特性：

1. 使用 `ForeignKeyContainer` 能让外键拥有懒加载 **lazy-loading**的特性
2. 通过继承或实现 `ModelContainer` ，你可以定义你自己的模型容器。
3. 列可以指定 `@ContainerKey` 来让 `ModelContainer` 拥有与列名不同的Key名

示例

```
JSONModel<TestObject> jsonModel = new JSONModel<>(json, TestObject

// constructs an insert or update query based on the JSON contents
jsonModel.save(false);

// deletes a model based on the JSON contents
jsonModel.delete(false);
```

支持的类型

`MapModelContainer`：将 `Map` 转换为 `Model` 一样使用 `JSONModel`：基于 `JSONObject` `JSONArrayModel`：一个 `JSONModel` 的容器
`ForeignKeyContainer`：使外键对象具有懒加载特性。当当前对象被查询时并不会触发对指向对象的查询，我们可以选择任意时刻通过 `toModel()` 方法发起实际检索。

Observable Models

`FlowContentObserver` 可用于监听 `Model` 的数据变化。它是 `ContentObserver` 类的一个包装以添加对Uri事件的监听。你可以创建一个观察者observer然后向里面添加一些 `OnModelStateChangeListener` 以响应model的数据变化事件。

NOTE：需要Android API16+

如何使用

创建一个 `FlowContentObserver` ：

```
FlowContentObserver observer = new FlowContentObserver();

// registers for callbacks from the specified table
observer.registerForContentChanges(context, TestNotifiableModel.cla

// call this to release itself from content changes
observer.unregisterForContentChanges(context);
```

创建Table类：

```
@Table(database = SomeDatabase.class)
public class TestNotifiableModel extends BaseModel {

    @PrimaryKey
    String name;

}
```

创建一个回调：

```
FlowContentObserver.OnModelStateChangeListener modelChangeListener
            @Override
            public void onModelStateChanged(Class<? extends Model>

            }
        };
```

将回调添加到观察者里：

```
observer.addModelChangeListener(modelChangeListener);

TestNotifiableModel testModel = new TestNotifiableModel();
testModel.setName("myName");

// will notify our observer automatically, no configuration needed
testModel.insert();
testModel.update();
testModel.save();
testModel.delete();

// when done with listener
observer.removeModelChangeListener(modelChangeListener);
```

打包事务通知

通过 `beginTransaction()...endTransactionAndNotify()` 方法，我们能将所有包含其中的通知事件打包在最后统一发送：

```
flowContentObserver.beginTransaction();

someModel.save();
// More modifications on a table for what the Flow Content Observer

// collects all unique URI and calls onChange here
flowContentObserver.endTransactionAndNotify();
```

检索结果的List包装

在BDFlow库中提供了 `FlowCursorList` 和 `FlowQueryList` 两个容器类，可用于容纳检索结果。它们提供与List相同的方法使你能简单地将其用于 `BaseAdapter` 或其他 `List<? extends Model>` 可用的地方。

FlowCursorList

`FlowCursorList` 是一个只读的用于缓存的容器，它包含的方法很类似与 `BaseAdapter` 比如 `getCount()` 和 `getItem(position)` 方法（但并没有继承BaseAdapter和List）

对于我们需要在屏幕上显示大量数据但实际只需要载入显示部分的数据时，这个容器非常有用。它虽然像是 `CursorAdapter` 但并不继承任何Adapter类，但提供了与数据库数据检索、转换的方便的API接口。

这里是定义它的一些方式：

- 使用任意的查询语句初始化它，像是 `From` 、 `StringQuery` 和 `Where`

```
// 第一个参数指定是否开启缓存
new FlowCursorList<>(true, SQLite.select().from(TestModel.class
.where(TestModel1_Table.name.like("pasta%")));

// 开启缓存并指定缓存大小（当缓存不可用时忽略）
new FlowCursorList<>(1000, SQLite.select().from(TestModel.class
.where(TestModel1_Table.name.like("pasta%")));
```

- 对特定类和列条件进行检索

```
new FlowCursorList<>(true, TestModel.class, TestModel1_Table.na

new FlowCursorList<>(1000, TestModel.class, TestModel1_Table.na
```

示例

```
private class TestModelAdapter extends BaseAdapter {
  private FlowCursorList<TestModel1> mFlowCursorList;

  public TestModelAdapter() {

  // retrieve and cache rows that have a name like pasta%
    mFlowCursorList = new FlowCursorList<>(true, TestModel.class,
          TestModel1_Table.name.like("pasta%"));
  }

  @Override
  public int getCount() {
    return mFlowCursorList.getCount();
  }

  @Override
  public TestModel1 getItem(int position) {
    return mFlowCursorList.getItem(position);
  }

  @Override
  public long getItemId(int position) {
    return position;
  }

  @Override
  public View getView(int position, View convertView, ViewGroup p
    // Implement here
  }
}
```

自定义缓存

可以通过重写 `getBackingCache()` 方法来替换默认的 `LruCache` 缓存方式：

```
FlowCursorList<TestModel> list = new FlowCursorList<TestModel>(true

    @Override
    protected ModelCache<TestModel, ?> getBackingCache() {
      return new SparseArrayBasedCache();
    }

}
```

FlowQueryList

`FlowQueryList` ：

1. 实现了java的 `List` 接口
2. 所有对表的修改都是实时的
3. 默认情况下所有的修改都是即时的
4. 如果你不希望这些修改操作在主线程中进行，请设置 `flowTableList.setTransact(true)`
5. Internally its backed by a `FlowCursorList` to include all of it's existing functionality.

好的实践

- 不要在循环中进行大量的单一操作，因为每个操作都会调用 `refresh()` 方法刷新数据库。推荐的做法是使用事务包裹这些操作：

```
FlowQueryList<TableModel> flowQueryList = ...;

// DON'T
for (int i = 0; i < 50; i++) {
    TableModel object = anotherList.get(i);
    flowQueryList.remove(object);
}

// better
flowQueryList.beginTransaction();
for (int i = 0; i < 50; i++) {
    TableModel object = anotherList.get(i);
    flowQueryList.remove(object);
}
flowQueryList.endTransactionAndNotify();

// DO
flowQueryList.removeAll(anotherList);
```

- 在其他地方对数据进行了修改后， `FlowQueryList` 内的数据会不一致，有两种方式刷新数据：

  1. 调用 `refresh()` 方法手动刷新数据
  2. 开启自动刷新：

     ```
     flowQueryList.enableSelfRefreshes(context);
     ```

- 如果你进行了大量的更改同时对这些事件都注册了监听，好的方式是使用 `endTransactionAndNotify()` 方法将通知在最后统一发送。

```
flowQueryList.beginTransaction();

// perform model changes!!

// calls any listeners associated with it (including the lister
// refreshes the list here (if registered)
flowQueryList.endTransactionAndNotify();
```

示例

```
FlowQueryList<TestModel1> flowQueryList = new FlowQueryList<TestM

flowQueryList.beginTransaction();

// Deletes from the table and returns the item
TestModel1 model1 = flowQueryList.remove(0);

// Saves the item back into the DB, updates if it already exists
flowQueryList.add(model1);

// Updates the item in the DB
flowQueryList.set(model1);

// Clears the whole table
flowQueryList.clear();

flowQueryList.endTransactionAndNotify();
```

触发器、引索和其他

本节包含一些SQLite的高级用法。这些功能是非常有用的，可用于提高数据库和应用的效能。

触发器

`Trigger` 是在数据库的某些操作之前或之后自动触发执行的一些动作。比如我们想要记录 `Friend` 表中所有被更新的数据：

```
CompletedTrigger<Friend> trigger = Trigger.create("NameTrigger")
                                    .after().update(Friend.class, F
                                    .begin(
                                        SQLite.insert(FriendLog.cla
                                          .columnValues(FriendLog_T
                                          FriendLog_Table.newName.e

// starts a trigger
trigger.enable();

// stops a trigger
trigger.disable();
```

引索

`Index` 引索是表中特殊的列，可使检索变得非常快速。但建立引索的时间与数据库大小成正比，如果检索性能更重要，这种建立引索是值得的。

引索通过 `@Table` 中的 `indexGroups()` 参数声明。它们有点像 `UniqueGroup` ：

1. 指定一个 `@IndexGroup`
2. 对指定的属性添加 `@Index` 注解
3. 编译工程，一个 `IndexProperty` get方法将会被生成

你可以非常简单的开启和关闭引索。

你可以添加任意多个 `@Table` 中的属性作为 `@IndexGroup` ：

```
@Table(database = TestDatabase.class,
       indexGroups = {
               @IndexGroup(number = 1, name = "firstIndex"),
               @IndexGroup(number = 2, name = "secondIndex"),
               @IndexGroup(number = 3, name = "thirdIndex")
       })
public class IndexModel2 extends BaseModel {

   @Index(indexGroups = {1, 2, 3})
   @PrimaryKey
   @Column
   int id;

   @Index(indexGroups = 1)
   @Column
   String first_name;

   @Index(indexGroups = 2)
   @Column
   String last_name;

   @Index(indexGroups = {1, 3})
   @Column
   Date created_date;

   @Index(indexGroups = {2, 3})
   @Column
   boolean isPro;
}
```

现在我们可以利用生成的 `IndexProperty` 加速我们的检索了：

```
SQLite.select()
  .from(IndexModel2.class)
  .indexedBy(IndexModel2_Table.firstIndex)
  .where(...); // do a query here.
```