# Realm数据库使用

# Realm简介

1. 什么是Realm

  Realm 是一个手机数据库，是用来替代 SQlite 的解决方案，比 SQlite 更轻量级，速度更快，因为它有一套自己的数据库搜索引擎，并且还具有很多现代数据库的优点，支持 JSON，流式 API 调用，数据变更通知，自动数据同步，简单身份验证，访问控制，事件处理，最重要的是跨平台，目前已经支持 Java、Swift、Object - C、React - Native 等多种实现。

2. Realm优势

易用

  Ream 不是在SQLite基础上的ORM，它有自己的数据查询引擎。并且十分容易使用。

快速

  由于它是完全重新开始开发的数据库实现，所以它比任何的ORM速度都快很多，甚至比SLite速度都要快。

跨平台

  Realm 支持 iOS & OS X (Objective‑C & Swift) & Android。我们可以在这些平台上共享Realm数据库文件，并且上层逻辑可以不用任何改动的情况下实现移植。

高级

  Ream支持加密，格式化查询，易于移植，支持JSON，流式api，数据变更通知等高级特性

可视化

  Realm 还提供了一个轻量级的数据库查看工具，在Mac Appstore 可以下载“Realm Browser”这个工具，开发者可以查看数据库当中的内容，执行简单的插入和删除数据的操作。（windows上还不清楚）

3. 使用要求

（1）目前不支持Android以外的Java

（2）Android Studio >= 1.5.1（不支持Eclipse）

（3）JDK version >=7.

（4）支持API 9(Android 2.3)以及之后的版本

# 创建model

创建一个User类，需要继承RealmObject。支持public, protected和 private的类以及方法

package com.example.realmdemo.model;  
  
import io.realm.RealmObject;  
  
public class User extends RealmObject {  
 private String name;  
 private int age;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

  除了直接继承于RealmObject来声明 Realm 数据模型之外，还可以通过实现 RealmModel接口并添加 @RealmClass修饰符来声明。

package com.example.realmdemo.model;  
  
import io.realm.RealmModel;  
import io.realm.annotations.RealmClass;  
  
@RealmClass  
public class User1 implements RealmModel {  
  
 private String name;  
 private int sex;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getSex() {  
 return sex;  
 }  
  
 public void setSex(int sex) {  
 this.sex = sex;  
 }  
}

## 支持的属性

boolean

byte

short

int

long

float

double

String

Date

byte[]

RealmObject

RealmList<? extends RealmObject>

还支持

Boolean

Byte

Short

Integer

Long

Float

Double

提示：整数类型 short、int 和 long 都被映射到 Realm 内的相同类型（实际上为 long ）

## @PrimaryKey——表示该字段是主键

  使用过数据库的同学应该看出来了，PrimaryKey就是主键。使用@PrimaryKey来标注，字段类型必须是字符串（String）或整数（byte，short，int或long）以及它们的包装类型（Byte,Short, Integer, 或 Long）。不可以存在多个主键，使用字符串字段作为主键意味着字段被索引（注释@PrimaryKey隐式地设置注释@Index）。

@PrimaryKey

private String id;

## @Required——表示该字段非空

在某些情况下，有一些属性是不能为null的。使用@Required可用于用于强行要求其属性不能为空，只能用于Boolean, Byte, Short, Integer, Long, Float, Double, String, byte[] 和 Date。在其它类型属性上使用。

## @Required修饰会导致编译失败。

提示：基本数据类型不需要使用注解 @Required，因为他们本身就不可为空。

@Required

private String name;

## @Ignore——表示忽略该字段

被添加@Ignore标签后，存储数据时会忽略该字段。

@Ignore

private String name;

## @Index——添加搜索索引

  为字段添加搜索索引，这样会使得插入的速度变慢，数据量也变得更大。不过在查询速度将变得更快，建议只在优化读取性能的特定情况时添加索引。支持索引：String，byte，short，int，long，boolean和Date字段。

注意：如果你创建Model并运行过，然后修改了Model。那么就需要升级数据库，否则会抛异常。升级方式后面会提到

# Realm接入

在项目的build.gradle引入

buildscript **{** repositories **{** mavenCentral()  
 maven **{** url 'https://maven.aliyun.com/repository/google/'**}** maven **{** url 'https://maven.aliyun.com/repository/public/'**}** maven **{** url 'https://maven.aliyun.com/repository/gradle-plugin/'**}** google()  
 maven **{** url "https://jitpack.io" **}** maven **{** url 'https://mirrors.huaweicloud.com/repository/maven/' **}  
 }** dependencies **{** classpath "com.android.tools.build:gradle:7.0.2"  
 classpath "io.realm:realm-gradle-plugin:10.4.0"  
 **}  
}**

修改settings.gradle中的仓库

dependencyResolutionManagement **{** repositoriesMode.set(RepositoriesMode.PREFER\_SETTINGS)  
 repositories **{**  
 mavenCentral()  
 maven **{** url 'https://maven.aliyun.com/repository/google/'**}** maven **{** url 'https://maven.aliyun.com/repository/public/'**}** maven **{** url 'https://maven.aliyun.com/repository/gradle-plugin/'**}** google()  
 maven **{** url "https://jitpack.io" **}** maven **{** url 'https://mirrors.huaweicloud.com/repository/maven/' **}  
  
 }  
}**

注意修改下面这句话：

repositoriesMode.set(RepositoriesMode.PREFER\_SETTINGS)

在app的build.gradle文件引入插件

plugins **{** id 'com.android.application'  
 id 'realm-android'  
**}**

# 创建数据库

## 初始化Realm

重写Application类

public class MyApp extends Application {  
 @Override  
 public void onCreate() {  
 super.onCreate();

}  
}

在AndroidManifest.xml配置Application

<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.example.realmdemo">  
  
 <application  
 android:name=".MyApp"  
 ...

>  
 <activity

…

</activity>  
 </application>  
  
</manifest>

初始化数据库

Realm.*init*(context); // 初始化realm

## 创建Realm数据库

方式1

public Realm getRealm1() {  
 Realm mRealm = Realm.*getDefaultInstance*();  
 return mRealm;  
}

方式2

public Realm getRealm2() {  
 // 这时候会创建一个叫做 default.realm的Realm文件，一般来说，  
 // 这个文件位于/data/data/包名/files/。通过realm.getPath()来获得该Realm的绝对路径  
 RealmConfiguration config = new RealmConfiguration.Builder()  
 .name("myrealm.realm") //文件名  
 .schemaVersion(0) //版本号  
 .build();  
 Realm realm = Realm.*getInstance*(config);  
 return realm;  
}

方式3

public Realm getRealm3() {  
 //创建保存在内存中的realm，应用关闭后就清除了  
 RealmConfiguration myConfig = new RealmConfiguration.Builder()  
 .name("myrealm.realm")  
 .inMemory()  
 .build();  
 Realm realm = Realm.*getInstance*(myConfig);  
 return realm;  
}

方式4

public Realm getRealm4() {  
 //创建加密的realm  
 byte[] key = new byte[64];  
 new SecureRandom().nextBytes(key);  
 RealmConfiguration config = new RealmConfiguration.Builder()  
 .encryptionKey(key)  
 .build();  
 Realm realm = Realm.*getInstance*(config);  
 return realm;  
}

参数说明：

RealmConfiguration支持的方法：

Builder.name : 指定数据库的名称。如不指定默认名为default。

Builder.schemaVersion : 指定数据库的版本号。

Builder.encryptionKey : 指定数据库的密钥。

Builder.migration : 指定迁移操作的迁移类。

Builder.deleteRealmIfMigrationNeeded : 声明版本冲突时自动删除原数据库。

Builder.inMemory : 声明数据库只在内存中持久化。

build : 完成配置构建。

## 关闭数据库

mRealm.close();

# 数据库操作

## 创建model

创建一个User类，有2中方法实现：

1. 需要继承RealmObject。

2通过实现 RealmModel接口并添加 @RealmClass修饰符来声明

## 插入数据

写入操作需要在事务中进行，可以使用executeTransaction方法来开启事务。

### 使用executeTransaction方法插入数据

public void insert1() {  
 mRealm.executeTransaction(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 User user = realm.createObject(User.class);  
 user.setName("Gavin");  
 user.setAge(23);  
 }  
 });  
}

UI线程不允许事务操作，需要设置 allowWritesOnUiThread(true)才可以

### 使用beginTransaction和commitTransaction插入

public void insert2() {  
 mRealm.beginTransaction();//开启事务  
 User user = mRealm.createObject(User.class);  
 user.setName("Gavin");  
 user.setAge(30);  
 mRealm.commitTransaction();//提交事务  
}

这种方式反而不设置allowWritesOnUiThread() 也不会报错

### 使用executeTransactionAsync

该方法会开启一个子线程来执行事务，并且在执行完成后进行结果通知。

public void insert3() {  
 mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 User user = realm.createObject(User.class);  
 user.setName("Gavin111");  
 user.setAge(222);  
 }  
 });  
}

如果一个model含有主键，不能使用上面只有一个参数的createObject()方法插入数据，需要使用下面的方法创建一条记录：

public <E extends RealmModel> E createObject(Class<E> clazz, @Nullable Object primaryKeyValue)

### 使用copyToRealmOrUpdate插入数据

当Model中存在主键的时候，推荐使用copyToRealmOrUpdate方法插入数据。如果对象存在，就更新该对象；反之，它会创建一个新的对象

public void insert4() {  
 final User1 user = new User1();  
 user.setName("Jack");  
 user.setUid(1);  
 mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 realm.copyToRealmOrUpdate(user);  
 }  
 });  
}

### 使用copyToRealm插入数据

当Model没有主键，使用copyToRealm方法，否则将抛出异常。

public void insert5() {  
 final User user = new User();  
 user.setName("test1");  
 user.setAge(8);  
 mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 realm.copyToRealmOrUpdate(user);  
 }  
 });  
}

### 使用executeTransactionAsync监听

1. public void insert6() {  
    RealmAsyncTask transaction = mRealm.executeTransactionAsync(new Realm.Transaction() {  
    @Override  
    public void execute(Realm realm) {  
    User user = realm.createObject(User.class);  
    user.setName("Eric");  
    user.setAge(30);  
    }  
    }, new Realm.Transaction.OnSuccess() {  
    @Override  
    public void onSuccess() {  
    LogUtils.*log*("executeTransactionAsync onSuccess");  
    }  
    }, new Realm.Transaction.OnError() {  
    @Override  
    public void onError(Throwable error) {  
    LogUtils.*log*("executeTransactionAsync onError");  
    }  
    });  
   }

注意：如果当Acitivity或Fragment被销毁时，在OnSuccess或OnError中执行UI操作，将导致程序奔溃 。用RealmAsyncTask .cancel();可以取消事务在onStop中调用，避免crash

public void onStop () {  
 if (transaction != null && !transaction.isCancelled()) {  
 transaction.cancel();  
 }  
}

### Realm还是个很好的功能就是将Json字符串转化为对象

1. // 一个city model  
    public class City extends RealmObject {  
    private String city;  
    private int id;  
    // getters and setters left out ...  
    }  
   // 使用Json字符串插入数据  
   realm.executeTransaction(new Realm.Transaction() {  
    @Override  
    public void execute(Realm realm) {  
    realm.createObjectFromJson(City.class, "{ city: \"Copenhagen\", id: 1 }");  
    }  
    });  
   // 使用InputStream插入数据  
   realm.executeTransaction(new Realm.Transaction() {  
    @Override  
    public void execute(Realm realm) {  
    try {  
    InputStream is = new FileInputStream(new File("path\_to\_file"));  
    realm.createAllFromJson(City.class, is);  
    } catch (IOException e) {  
    throw new RuntimeException();  
    }  
    }  
    });

Realm 解析 JSON 时遵循如下规则：

使用包含空值（null）的 JSON 创建对象：

对于非必须（可为空值的属性），设置其值为 null；

对于必须（不可为空值的属性），抛出异常；

使用包含空值（null）的 JSON 更新对象：

对于非必须（可为空值的属性），设置其值为 null；

对于必须（不可为空值的属性），抛出异常；

使用不包含对应属性的 JSON： \* 该属性保持不变

## 查找

查找并不需在事务中操作，直接查询即可

### 普通查询 findAll

RealmResults<User> userList = mRealm.where(User.class).findAll();

### 异步查询 findAllAsync

RealmResults<User> userList = mRealm.where(User.class)  
 .equalTo("name", "Gavin")  
 .findAllAsync();

### 查询第一条数据 findFirst

User user = mRealm.where(User.class).findFirst();

### 根据条件查询 equalTo

## 数据库升级

当数据结构发生变化是，需要升级数据库。对于Realm来说，数据库升级就是迁移操作，把原来的数据库迁移到新结构的数据库。Realm的迁移通过实现RealmMigration接口，实现migrate()方法，在这个方法里面处理升级逻辑。当Realm检测到有新版本时，会回调migrate()方法一次。

这里需要考虑快版本迁移的逻辑，例如从版本1升级到版本2，和从版本1升级到版本5。处理逻辑就是添加各个旧版本的分支判断，处理完之后，旧版本+1，这样就会从当前的旧版本开始处理到最新版本。

### 示例1：添加普通字段

User表员原来的字段如下

private String name;  
private int sex;

现在需要添加一个uid字段，model代码如下：

private long uid;  
private String name;  
private int sex;

migrate()方法实现如下：

public class CustomMigration implements RealmMigration {  
 @Override  
 public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {  
 LogUtils.*log*("oldVersion: " + oldVersion + " newVersion: " + newVersion);  
 RealmSchema schema = realm.getSchema();  
 if (oldVersion == 0 && newVersion == 1) {  
 RealmObjectSchema personSchema = schema.get("User1");  
 personSchema  
 .addField("uid", long.class)  
 .transform(new RealmObjectSchema.Function() {  
 @Override  
 public void apply(DynamicRealmObject obj) {  
 obj.set("uid", "3"); // 为uid设置值  
 }  
 });  
 oldVersion++;  
 }  
 }  
}

上述代码在数据库有数据的情况下可以直接添加uid一列，并且都设置成3.

### 示例2：添加字段并设置成主键

添加一个Long类型的uid

注意问题：

如果在model添加的uid设置如下：

private long uid;

而在迁移的时候设置

personSchema  
 .addField("uid", Long.class)

升级的时候则会报错：

Property "xxx" has been made required.

原因是：

long.class : 是可nullable

Long.class : 是required

基本数据是不需要添加@required注解的，因为基本数据类型不能为空，而引用类型需要添加@required，上述的如果需要引用类型需要改成如下：

personSchema  
 .addField("uid", Long.class, FieldAttribute.*REQUIRED*)

因为model设置的是基本类型，不为空，而升级处设置的是引用类型，如果没有添加FieldAttribute.REQUIRED则可能为空，所以报 上面的错误。

现在model添加引用类型

private Long uid;

migrate()方法实现如下：

public class CustomMigration implements RealmMigration {  
 @Override  
 public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {  
 LogUtils.*log*("oldVersion: " + oldVersion + " newVersion: " + newVersion);  
 RealmSchema schema = realm.getSchema();  
 if (oldVersion == 0 && newVersion == 1) {  
 RealmObjectSchema personSchema = schema.get("User1");  
 personSchema  
 .addField("uid", Long.class)  
 .transform(new RealmObjectSchema.Function() {  
 @Override  
 public void apply(DynamicRealmObject obj) {  
 obj.set("uid", "5"); // 为uid设置值  
 }  
 })  
 /\*.addPrimaryKey("uid")\*/;  
 oldVersion++;  
 }  
 }  
}

上面的代码可以不添加FieldAttribute.REQUIRED标志，因为model使用引用类型，升级也使用引用类型，所以可能都为空。

上述代码在数据库有数据的情况下可以直接添加uid一列，并且都设置成5。

### 添加一张新表

添加一张新表User到数据库

public class User2 extends RealmObject {  
  
 @Required  
 private String name2;  
 private int age2;  
  
 public String getName2() {  
 return name2;  
 }  
  
 public void setName2(String name2) {  
 this.name2 = name2;  
 }  
  
 public int getAge2() {  
 return age2;  
 }  
  
 public void setAge2(int age2) {  
 this.age2 = age2;  
 }  
}

migate代码如下

public class CustomMigration implements RealmMigration {  
 @Override  
 public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {  
 。。。  
 if(oldVersion == 1) {  
 RealmObjectSchema user2 = schema.create("User2");  
 if(user2 != null) {  
 user2.addField("name2", String.class, FieldAttribute.*REQUIRED*);  
 user2.addField("age2", int.class);  
 }  
 oldVersion++ ;  
  
 }  
 }  
}

修改数据库版本号+1，

这样会新添加一张新表User2，可以用Realm Studio 查看。

注意问题，model使用的注解必须和migate()方法设置的属性一致，否则会报

Realm Property ‘XXX’' has been made optional

例如model name2属性并没有设置@Required

private String name2;

但是magire()方法设置了FieldAttribute.REQUIRED

.addField("uid", Long.class, FieldAttribute.*PRIMARY\_KEY*)

就会报上面的异常问题。

### 跨数版本升级

原来有2张数据库表User和User1，其中User的不需要改变

版本1中 User1添加一个字段uid，并且设置成主键

版本2中 添加一张新表User2.

migrate()方法代码如下：

public class CustomMigration implements RealmMigration {  
 @Override  
 public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {  
 LogUtils.*log*("oldVersion: " + oldVersion + " newVersion: " + newVersion);  
 RealmSchema schema = realm.getSchema();  
 if (oldVersion == 0) {  
 RealmObjectSchema personSchema = schema.get("User1");  
 personSchema  
 .addField("uid", Long.class, FieldAttribute.*PRIMARY\_KEY*)  
 .transform(new RealmObjectSchema.Function() {  
 @Override  
 public void apply(DynamicRealmObject obj) {  
 obj.set("uid", "5"); // 为uid设置值  
 }  
 })  
 /\*.addPrimaryKey("uid")\*/;  
 oldVersion++;  
 }  
 if(oldVersion == 1) {  
 RealmObjectSchema user2 = schema.create("User2");  
 if(user2 != null) {  
 user2.addField("name2", String.class, FieldAttribute.*REQUIRED*);  
 user2.addField("age2", int.class);  
 }  
 oldVersion++ ;  
 }  
 }  
}

### 版本冲突处理

数据库升级可能会遇到报错，如果遇到报错，处理最直接方式就是删除掉原来的数据库，重新创建新的。

1. 升级是删除原来的数据

设置RealmConfiguration deleteRealmIfMigrationNeeded（）方法

Realm.init(context);  
RealmConfiguration config = new RealmConfiguration  
 .Builder()  
 .deleteRealmIfMigrationNeeded()  
 .build();

这种处理会导致数据丢失，例如：

添加和移除一个属性

创建了一张新表

移除一张表

添加或移除@required 注解

添加或移除@primaryKey

添加或 移除@Index

等等

1. 捕获异常时，删除旧的数据库，重新创建新的数据库

RealmConfiguration realmConfiguration = new RealmConfiguration.Builder(this).build();  
  
try {  
 return Realm.getInstance(realmConfiguration);  
} catch (RealmMigrationNeededException e){  
 try {  
 Realm.deleteRealm(realmConfiguration);  
 //Realm file has been deleted.  
 return Realm.getInstance(realmConfiguration);  
 } catch (Exception ex){  
 throw ex;  
 //No Realm file to remove.  
 }  
}

参考文档：

<https://stackoverflow.com/questions/33940233/realm-migration-needed-exception-in-android-while-retrieving-values-from-real>

### 默认数据库版本

当数据库没有设置版本时，例如通过下面方式设置数据库时：

Realm.*init*(BasicConfig.*INSTANCE*.getAppContext());  
RealmConfiguration config = new RealmConfiguration.Builder()  
 .name("tanwanxingqiu.realm") //指定数据库的名称  
 .deleteRealmIfMigrationNeeded() //声明版本冲突时自动删除原数据库  
 .build();  
Realm.*setDefaultConfiguration*(config);

默认的版本号是0，可以通过以下方式获取：

mRealm.getConfiguration().getSchemaVersion()

### 删除所有数据库数据

当设置deleteRealmIfMigrationNeeded()时，并且数据版本增加了，自动删除之前所有表的内容

## 查看数据库

查看数据使用Realm Studio

下载地址：

<https://docs.realm.io/sync/realm-studio/>

# 问题

## 添加Realm库同步报错

An exception occurred applying plugin request [id: 'realm-android']

> Failed to apply plugin 'realm-android'.

> Build was configured to prefer settings repositories over project repositories but repository 'BintrayJCenter' was added by plugin 'realm-android'

这个项目是使用gradle 7.0.2创建的，仓库地址的设置已经移到了settings.gradle文件中，需要把下面的语句替换掉:

repositoriesMode.set(RepositoriesMode.FAIL\_ON\_PROJECT\_REPOS)

替换成

repositoriesMode.set(RepositoriesMode.PREFER\_SETTINGS)

参考文档：

https://github.com/realm/realm-java/issues/7374

## 新建的demo运行报错

Didn't find class "androidx.startup.R$string" on path

可能是Realm的版本太旧了，把版本从5.9.0升级到10.4.0就可以运行了

dependencies **{**  
 classpath "io.realm:realm-gradle-plugin:10.4.0"  
**}**

## 运行程序插入数据报错

Caused by: io.realm.exceptions.RealmException: Running transactions on the UI thread has been disabled. It can be enabled by setting 'RealmConfiguration.Builder.allowWritesOnUiThread(true)'.

是因为Realm已经禁止在UI线程进行事务处理，如果需要这样做，需要创建数据库时通过

RealmConfiguration.Builder设置下面的代码

allowWritesOnUiThread(true)

## 修改model添加主键，报错

java.lang.RuntimeException: Unable to create application com.example.realmdemo.MyApp: io.realm.exceptions.RealmMigrationNeededException: Migration is required due to the following errors:

- Property 'User1.uid' has been added.

- Primary Key for class 'User1' has been added.

应该是因为之前已经创建了表User1，这里面是没有uid这个字段的，后面添加了这个字段，并且使用@PrimaryKey设置成主键就报错了，原因是数据库没有升级，需要创建的User1在原来就已经存在，后面设置一下升级功能看看能不能解决，现在暂时先卸载app重装。

## 报错

Realm access from incorrect thread. Realm objects can only be accessed on the thread they were created.

意思就是Realm对象只能在它创建的线程上访问。

出现问题代码：

mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 //先查找后得到User对象  
 Test t = mRealm.where(Test.class).findFirst();  
 LogUtils.*log*("t: " + t.toString());  
 t.setAge(100);  
 }  
});

修改后， 把异步中的mRealm改成realm

mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 //先查找后得到User对象  
 Test t = realm.where(Test.class).findFirst();  
 LogUtils.*log*("t: " + t.toString());  
 t.setAge(100);  
 }  
});

## 报错：Cannot modify managed objects outside of a write transaction.

更新操作需要在事务中处理

## 从数据库中读取的bean，只能读不能写

Realm从数据中独取出来的任何对象，都只能读，不能写，更新操作只能在事务中处理。

## 跨线程 Realm 访问

Realm access from incorrect thread. Realm objects can only be accessed on the thread they were created.

这是一个初次使用 Realm 时常见的异常。请注意，RealmObject、RealmResults等相关对象都是与其线程中的 Realm 实例绑定的。因为两个线程中的 Realm 实例可能锁定了不同的 Realm 版本，这些对象也可能处于不同的数据版本，跨线程访问会引起数据的不一致性。所以，在另一个线程中访问同一个对象的时候，请在该线程中进行查询以获得这个对象绑定该线程 Realm 的实例。或者使用 Realm 提供的相应的异步查询接口，具体请参考相关文档。

下面是异常代码，获取Realm实例必须同一个，比如下面的execute(Realm realm)里面又使用外部的mRealm实例，是有问题的，正确的用法是把realm传递到getPrimaryKey方法中。

mRealm.executeTransactionAsync(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 Test test = realm.createObject(Test.class, getPrimaryKey());  
 test.setName("test111");  
 test.setAge(1);  
 }  
});

private long getPrimaryKey() {  
 long nextId = 0;  
 if(mRealm != null) {  
 Number maxId = mRealm.where(Test.class).max("uid");  
 nextId = (maxId == null) ? 1 : maxId.intValue() + 1;  
 }  
 return nextId;  
}

## 异步线程添加数据异常

启动了一个线程插入数据，

mRealm.executeTransaction(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 ZipBean zip = realm.createObject(ZipBean.class, 1);  
 zip.setZipAllSize(200);  
 zip.setZipName("测试");  
 zip.setZipDownloadSize(100);  
 zip.setZipVersion(1);  
 zip.setZipUrl("hello");  
 }  
});

但是还是报错：

Realm Running transactions on the UI thread has been disabled

原因是mRealm是在主线程中获取的，应该重新获取一下mRealm：

mRealm = Realm.*getDefaultInstance*();  
mRealm.executeTransaction(new Realm.Transaction() {  
 @Override  
 public void execute(Realm realm) {  
 ZipBean zip = realm.createObject(ZipBean.class, 1);  
 zip.setZipAllSize(200);  
 zip.setZipName("测试");  
 zip.setZipDownloadSize(100);  
 zip.setZipVersion(1);  
 zip.setZipUrl("hello");  
 }  
});

# 参考文档

https://realm.io/

<https://www.mongodb.com/docs/realm-sdks/java/latest/index.html>

式样

微软雅黑 Light 5号

段落 固定值 23磅

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5