数据处理及持久化

String

创建

```
String s1 = "test";                 // 直接创建
String s2 = s1;                     // 相同引用
String s3 = new String("test");     // String 对象创建
```

String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了

连接字符串

- s1.concat(s2);
- "Hello " + " world" + "!"

格式化字符串

```
String fs;
fs = String.format("浮点型" + "%f, 整型" + " %d, 字符串" + "%s", floatVar, intVar,
stringVar);
```

常用方法

序号	方法
1	char charAt(int index) **返回**指定索引处的 **char** 值。
2	int compareTo(String anotherString) 按字典顺序比较两个字符串。
3	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。
4	boolean endsWith(String suffix) 此字符串是否以指定的后缀结束。
5	boolean equals(Object anObject) 将此字符串与指定的对象比较。
6	boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。
7	int indexOf(int ch) 返回指定**字符**在此字符串中第一次出现处的索引。
8	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定**字符**处的索引，从指定的索引开始搜索。
9	int indexOf(String str) 返回指定子**字符串**在此字符串中第一次出现处的索引。
10	int indexOf(String str, int fromIndex) 返回指定子**字符串**在此字符串中第一次出现处的索引，从指定的索引开始。
11	int length() 返回此字符串的长度。
12	boolean matches(String regex) 此字符串是否匹配给定的正则表达式。
13	String replace(char oldChar, char newChar)替换字符串。
14	String[] split(String regex) 根据给定正则表达式的匹配拆分此字符串。
15	String[] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串。
16	boolean startsWith(String prefix) 此字符串是否以指定的前缀开始。
17	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
18	String substring(int beginIndex) 返回一个新的字符串，它是此字符串的一个子字符串。
19	String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。
20	char[] toCharArray() 将此字符串转换为一个新的字符数组。
21	String toLowerCase() 所有字符都转换为小写。
22	String toUpperCase() 所有字符都转换为大写。
23	String trim() 返回字符串的副本，忽略前导空白和尾部空白。
24	contains(CharSequence chars) 判断是否包含指定的字符系列。

Bitmap

`Bitmap`是位图，是由像素点组成的。Bitmap可以当作`ImageView`的背景
`imageView.setImageBitmap(bitmap);`

存储格式

`Bitmap`有四种存储方式

- ALPHA_8：只存储透明度，不存储色值，1个像素点占1个字节
- ARGB_4444：ARGB各用4位存储，1个像素点16位占2个字节，画质不行
- ARGB_8888：ARGB各用8位存储，1个像素点32位占4个字节
- RGB_565：只存储色值，不存储透明度，默认不透明，一个像素点占用16位2个字节。

一般情况下用`ARGB_8888`格式存储`Bitmap`

它有多大?

以我的手机为例，分辨率2310 x 1080，采用ARGB_8888格式存储。大小 = 2310*1080*4B = 9.52MB。

创建

BitmapFactory.Options

它可以设置Bitmap的采样率，通过改变图片的宽度高度和缩放比例等，减少图片像素数

https://www.jianshu.com/p/34117921a6a1?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation

BitmapFactory

```
// 从资源文件中通过id加载bitmap
// Resources res:资源文件，可以context.getResources()获得
// id:资源文件的id
public static Bitmap decodeResource(Resources res, int id)
public static Bitmap decodeResource(Resources res, int id, Options opt)
```

```
// 传入文件路径加载，比如加载sd卡中的文件
// pathName:文件的全路径名
public static Bitmap decodeFile(String pathName);
public static Bitmap decodeFile(String pathName, Options opt);
```

```
// 从byte数组中加载
// offset:对应data数组的起始下标
// length:截取的data数组的长度
public static Bitmap decodeByteArray(byte[] data, int offset, int length);
public static Bitmap decodeByteArray(byte[] data, int offset, int length,
Options opt);
```

```
// 从输入流中加载图片
// InputStream is:输入流
public static Bitmap decodeStream(InputStream is);
```

Bitmap部分静态方法

```
// width和height是长和宽单位px，config是存储格式
static Bitmap createBitmap(int width , int height Bitmap.Config config)
// 根据一幅图像创建一份副本
static Bitmap createBitmap(Bitmap bm)
// 截取一幅bitmap，起点是（x，y）,width和height分别对应宽高
static Bitmap createBitmap(Bitmap bm,int x,int y,int width,int height)
```

注意

- 加载图像可以使用BitmapFactory和Bitmap.create方法
- 可以通过Options实现缩放图片，获取图片信息，配置缩放比例等功能
- 如果需要裁剪或者缩放图片，只能使用create系列函数

三级缓存

什么是三级缓存

三级缓存：从服务器获取的数据除了在本地之外，在内存中也有一份。当请求数据时，可以先**从内存中获取**数据，如果内存中没有数据，或者内存中的数据是不正确的，再从**本地**（例如数据库）**获取数据**。如果本地数据库的数据也是不正确的，再从服务器获取数据。取回的数据一份存内存中，一份存在本地中。

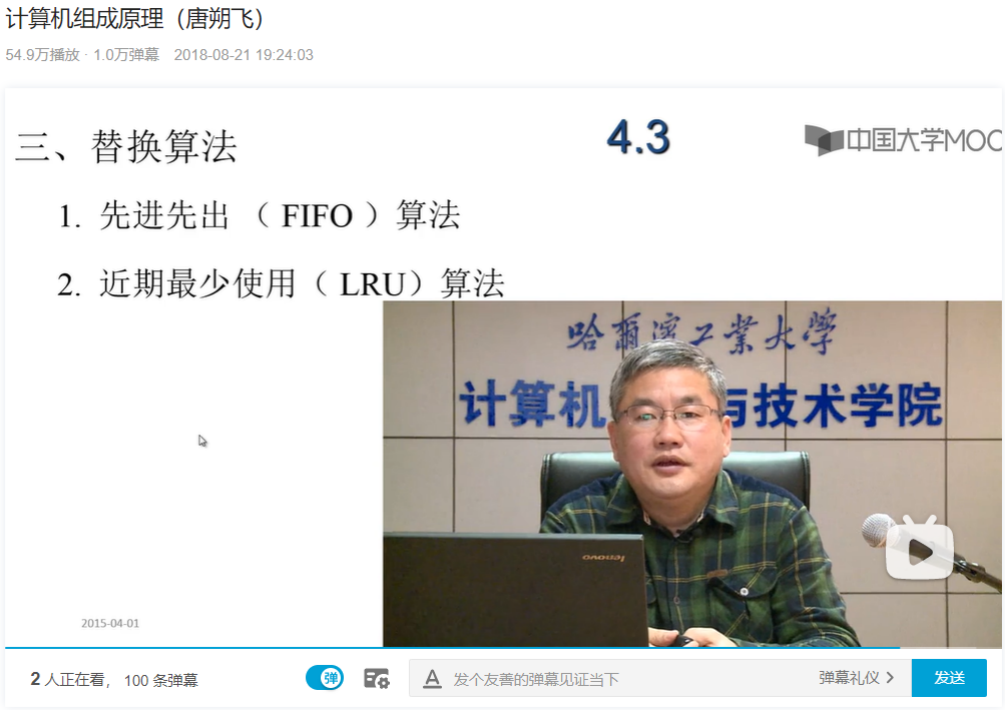
如果去掉内存的缓存，就是二级缓存。

为什么使用三级缓存?

现在需要网络获取一张图片，如果每次都从网络获取图片，会消耗很多流量，加载速度也会降低。

内存缓存：LruCache

Lru是Least Recently Used的缩写，最近最少使用算法，进行**内存缓存**。




```
public LruCache(int maxSize) {
    if (maxSize <= 0) {
        throw new IllegalArgumentException("maxSize <= 0");
    }
    this.maxSize = maxSize;
    this.map = new LinkedHashMap<K, V>(0, 0.75f, true);
}
```

LruCache构造函数内new了一个LinkedHashMap，LinkedHashMap是一个双向链表的数据结构，适用于实现Lru算法。LruCache的构造函数中传入参数为可缓存的最大数量并赋值给maxSize。

LruCache 利用 LinkedHashMap 的一个特性（`accessOrder=true` 基于访问顺序）再加上对 LinkedHashMap 的数据操作上锁实现的缓存策略。

首先设置了内部 LinkedHashMap 构造参数 `accessOrder=true`， 实现了数据排序按照访问顺序。LruCache类在调用`get(K key)`方法时，都会调用`LinkedHashMap.get(Object key)`。

如设置了`accessOrder=true`后，调用`LinkedHashMap.get(Object key)`都会通过LinkedHashMap的afterNodeAccess()方法将数据移到队尾。

由于最新访问的数据在尾部，在 put 和 trimToSize 的方法执行下，如果发生数据移除，会优先移除掉头部数据

sizeOf

```
/**
     * Returns the size of the entry for {@code key} and {@code value} in
     * user-defined units.  The default implementation returns 1 so that size
     * is the number of entries and max size is the maximum number of entries.
     */
protected int sizeOf(K key, V value) {
    return 1;
}
```

默认返回1，缓存图片时可以重写（缓存Bitmap需要设置总大小，sizeOf返回每张Bitmap大小）。只有`maxSize`和`sizeOf`返回值是同一个单位制缓存的判断才有意义。

```
protected int sizeOf(String key, Bitmap value) {
    return value.getByteCount();    //计算每张图片的大小
}
```

LruCache.put(K key, V value)

```
public final V put(K key, V value) {
        if (key == null || value == null) {
            throw new NullPointerException("key == null || value == null");
        }

        V previous;
        synchronized (this) {
            putCount++;
             //safeSizeOf(key, value)。
             //这个方法返回的是1，也就是将缓存的个数加1.
            // 当缓存的是图片的时候，这个size应该表示图片占用的内存的大小，所以应该重写里面调用的sizeOf(key, value)方法
            size += safeSizeOf(key, value);
            //向map中加入缓存对象,若缓存中已存在，返回已有的值，否则执行插入新的数据，并返回null
            previous = map.put(key, value);
            //如果已有缓存对象，则缓存大小恢复到之前
            if (previous != null) {
                size -= safeSizeOf(key, previous);
            }
        }
        if (previous != null) {
            entryRemoved(false, key, previous, value);
        }

        trimToSize(maxSize);
        return previous;
    }
```

开始的时候把值放入LinkedHashMap，不管超不超过你设定的缓存容量。

根据 safeSizeOf方法计算 此次添加数据的容量是多少，并且加到size里。

方法执行到最后时，通过trimToSize()方法 来判断size 是否大于maxSize，如果满了就要删除近期最少使用的数据。

LruCache.trimToSize(int maxSize)

```
public void trimToSize(int maxSize) {
        while (true) {
            K key;
            V value;
            synchronized (this) {
              //如果map为空并且缓存size不等于0或者缓存size小于0，抛出异常
                if (size < 0 || (map.isEmpty() && size != 0)) {
```

```
                    throw new IllegalStateException(getClass().getName()
                            + ".sizeOf() is reporting inconsistent results!");
                }
    //如果缓存大小size小于最大缓存，不需要再删除缓存对象，跳出循环
                if (size <= maxSize) {
                    break;
                }
    //在缓存队列中查找最近最少使用的元素，若不存在，直接退出循环，若存在则直接在map中删除。
                Map.Entry<K, V> toEvict = map.eldest();
                if (toEvict == null) {
                    break;
                }

                key = toEvict.getKey();
                value = toEvict.getValue();
                map.remove(key);
                size -= safeSizeOf(key, value);
                //回收次数+1
                evictionCount++;
            }

            entryRemoved(true, key, value, null);
        }
    }
```

有个while(true)，循环删除LinkedHashMap中队首的元素，即近期最少访问的，直到缓存大小小于最大值。

LruCache.get(K key)

```
public final V get(K key) {
        if (key == null) {
            throw new NullPointerException("key == null");
        }

        V mapValue;
        synchronized (this) {
              //从LinkedHashMap中获取数据。
            mapValue = map.get(key);
            if (mapValue != null) {
                hitCount++;
                return mapValue;
            }
            missCount++;
        }
    /*
     * 正常情况走不到下面
     * 因为默认的 create(K key) 逻辑为null
     * 走到这里的话说明实现了自定义的create(K key) 逻辑，比如返回了一个不为空的默认值
     * 如果通过key从缓存集合中获取不到缓存数据，就尝试使用creat(key)方法创造一个新数据。
     * create(key)默认返回的也是null，需要的时候可以重写这个方法。
     */

        V createdValue = create(key);
        if (createdValue == null) {
            return null;
        }
```

```
        //如果重写了create(key)方法，创建了新的数据，就讲新数据放入缓存中。
        synchronized (this) {
            createCount++;
            mapValue = map.put(key, createdValue);

            if (mapValue != null) {
                // There was a conflict so undo that last put
                map.put(key, mapValue);
            } else {
                size += safeSizeOf(key, createdValue);
            }
        }

        if (mapValue != null) {
            entryRemoved(false, key, createdValue, mapValue);
            return mapValue;
        } else {
            trimToSize(maxSize);
            return createdValue;
        }
    }
```

调用LruCache的get()方法获取集合中的缓存对象时，就代表访问了一次该元素，将会更新队列，保持整个队列是按照访问顺序排序,这个更新过程就是在LinkedHashMap中的get()方法中完成的。

总结

> LruCache中维护了一个集合LinkedHashMap，该LinkedHashMap是以访问顺序排序的。
>
> 调用put()方法，会添加元素，并调用trimToSize()判断缓存是否已满，如果满了就删除LinkedHashMap队首元素，即近期最少访问的元素。
>
> 调用get()方法时，会调用LinkedHashMap的get()方法获得对应集合元素，同时会更新该元素到队尾。

硬盘缓存：DiskLruCache

LruCache只是管理了内存中图片的存储与释放，其实Google还提供了一套硬盘缓存的解决方案：DiskLruCache(非Google官方编写，但获得官方认证)。

DiskLruCache是不能new出实例的，我们需要调用它的open()方法。

```
public static DiskLruCache open(File directory, int appVersion, int valueCount,
long maxSize)
```

第一个参数指定的是数据的缓存地址，第二个参数指定当前应用程序的版本号，第三个参数指定同一个key可以对应多少个缓存文件，基本都是传1，第四个参数指定最多可以缓存多少字节的数据。

注意：每当版本号改变，缓存路径下存储的所有数据都**会被清除掉**，因为DiskLruCache认为当应用程序有版本更新的时候，所有的数据都应该从网上重新获取。

写入

下面我们就可以使用DiskLruCache来进行写入了，写入的操作是借助DiskLruCache.Editor这个类完成的。这个类也是不能new的，需要调用DiskLruCache的edit()方法来获取实例

```
public Editor edit(String key) throws IOException
```

key将会成为缓存文件的文件名，并且必须要和图片的URL是一一对应的，可以将图片的URL进行MD5编码作为文件名。

可以调用它的newOutputStream()方法来创建一个输出流

在写入操作执行完之后，我们还需要调用一下commit()方法进行提交才能使写入生效，调用abort()方法的话则表示放弃此次写入。

```
DiskLruCache.Editor editor = mDiskLruCache.edit(key);
if (editor != null) {
    OutputStream outputStream = editor.newOutputStream(0);
    if (downloadUrlToStream(url, outputStream)) {
        editor.commit();
    } else {
        editor.abort();
    }
}
mDiskLruCache.flush();
```

flush()

这个方法用于将内存中的操作记录同步到日志文件（也就是journal文件）中。DiskLruCache能正常工作的前提就是依赖于journal文件。其实并不是每次写入缓存都要调用一次flush()方法的，会额外增加同步journal文件的时间。比较标准的做法是在Activity的onPause()方法中去调用一次flush()方法。

读取

使用get()方法实现

```
public synchronized Snapshot get(String key) throws IOException
```

key即是我们上面计算的MD5码，这里获取到的是DiskLruCache.Snapshot对象，调用它的getInputStream()方法可以得到缓存文件的输入流。

```
DiskLruCache.Snapshot snapShot = mDiskLruCache.get(key);
if (snapShot != null) {
    InputStream is = snapShot.getInputStream(0);
    Bitmap bitmap = BitmapFactory.decodeStream(is);
    mImage.setImageBitmap(bitmap);
}
```

移除

```
public synchronized boolean remove(String key) throws IOException
```

key即是我们上面计算的MD5码

这个方法我们并不应该经常去调用它。因为不需要担心缓存的数据过多，DiskLruCache会根据我们在调用open()方法时设定的缓存最大值来自动删除多余的缓存。只有你确定某个key对应的缓存内容已经过期，需要从网络获取最新数据的时候才应该调用remove()方法来移除缓存。

size()

这个方法会返回当前缓存路径下所有缓存数据的总字节数，以byte为单位，例如我要“清理缓存”按钮后面显示缓存大小，就可以通过调用这个方法计算出来。

close()

这个方法用于将DiskLruCache关闭，是和open()方法对应的一个方法。关闭之后就不能再调用DiskLruCache中任何操作缓存数据的方法，通常只应该在Activity的onDestroy()方法中去调用。

delete()

这个方法用于将所有的缓存数据全部删除，比如“清理缓存”按钮的逻辑就是调用DiskLruCache的delete()方法就可以了。

Room

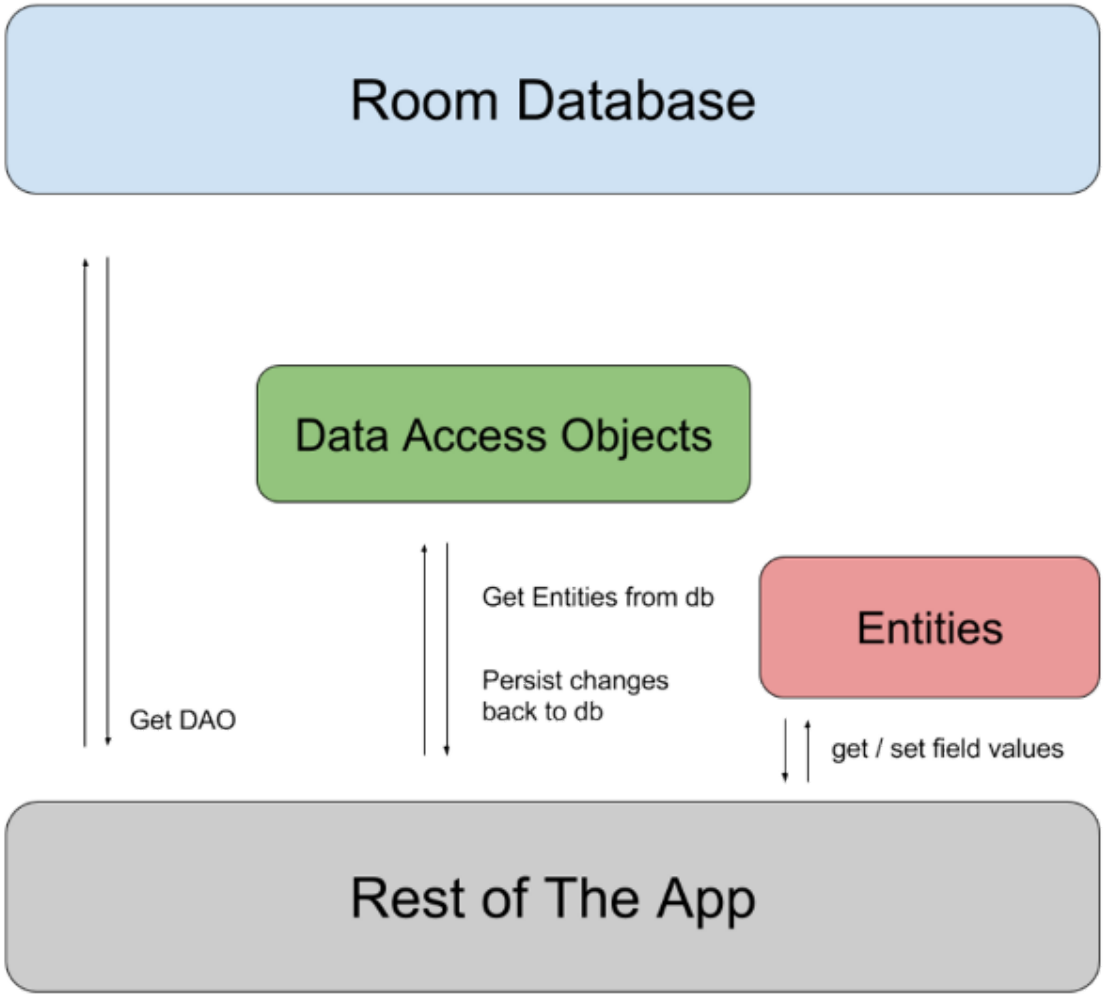
Room 持久性库在 SQLite 的基础上提供了一个抽象层，让用户能够在充分利用 SQLite 的强大功能的同时，获享更强健的数据库访问机制。

简单来说就是可以让我们更加方便的使用数据库。

官方文档： https://developer.android.google.cn/training/data-storage/room?hl=zh_cn

Room有**3个主要组件**

- Database： 数据库
- Entity : 代表数据库一个表结构
- Dao : 包含访问数据库的方法




```
abstract class AppDatabase : RoomDatabase() {

    abstract fun searchHistoryDao(): SearchHistoryDao
    abstract fun animeDownloadDao(): AnimeDownloadDao
    abstract fun favoriteAnimeDao(): FavoriteAnimeDao
    abstract fun historyDao(): HistoryDao

    companion object {
        private var instance: AppDatabase? = null

        private val migration1To2 : Migration = object : Migration(1, 2) {
            override fun migrate(database: SupportSQLiteDatabase) {
                database.execSQL( sql: "ALTER TABLE animeDownloadList ADD fileName TEX
            }
        }

        private val migration2To3 : Migration = object : Migration(2, 3) {
            override fun migrate(database: SupportSQLiteDatabase) {
                database.execSQL( sql: "CREATE TABLE favoriteAnimeList(type TEXT NOT N
                database.execSQL( sql: "CREATE TABLE historyList(type TEXT NOT NULL, a
            }
        }

        fun getInstance(context: Context): AppDatabase {
```

```
@Entity(tableName = "historyList")
class HistoryBean(       //下面的url都是partUrl
    @ColumnInfo(name = "type")
    override var type: String,
    @ColumnInfo(name = "actionUrl")
    override var actionUrl: String,
    @PrimaryKey
    @ColumnInfo(name = "animeUrl")
    var animeUrl: String,
```

```
@Dao
interface HistoryDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertHistory(historyBean: HistoryBean)

    //按照时间戳顺序，从大到小。最后搜索的元组在最上方（下标0）显示
    @Query(value = "SELECT * FROM historyList ORDER BY time DESC")
    fun getHistoryList(): MutableList<HistoryBean>

    @Query(value = "SELECT * FROM historyList WHERE animeUrl = :animeUrl")
    fun getHistory(animeUrl: String): FavoriteAnimeBean?

    @Update(onConflict = OnConflictStrategy.REPLACE)
    fun updateHistory(favoriteAnimeBean: FavoriteAnimeBean)
```

添加依赖

```
dependencies {
  implementation "androidx.room:room-runtime:2.2.6"
  annotationProcessor "androidx.room:room-compiler:2.2.6"
}
```

在编译时，Room 会将数据库的架构信息导出为 JSON 文件

如果要导出，则在Gradle.build添加

```
android {
    defaultConfig {
        javaCompileOptions {
            annotationProcessorOptions {
                arguments += ["room.schemaLocation":
"$projectDir/schemas".toString()]
            }
        }
    }
}
```

如果不导出，则在数据库注解中添加exportSchema = false

@Database(entities = {entity.class}, version = 1, **exportSchema = false**)

Entity实体类

当一个类用@Entity注解并且被@Database注解中的entities属性所引用，Room就会在数据库中为那个entity创建一张表。

```
// 存储历史记录的表，表名叫historyTable，每一个变量对应一列
@Entity(tableName = "historyTable")
public class HistoryEntity {
    // 主键（主码），通过主键可以唯一找到一行数据，在数据库中主键不能重复，不可为空
    @PrimaryKey
    @NonNull
    public String name;

    // 明确指名列名是timeStamp
    @ColumnInfo(name = "timeStamp")
    public Long timeStamp;

    // 告诉数据库不储存这个
    @Ignore
    public String ignoreString;
}
```

Dao（Data Access Object）

定义了访问数据库的方法， Dao里面所有的操作都是依赖方法来实现的。

```
@Dao
public interface HistoryDao {
    // 执行SQL语句，查询所有历史记录，返回可以是数组也可以是List
    @Query("SELECT * FROM historyTable")
    List<HistoryEntity> getAllHistory();

    // 执行SQL语句，查询给定时间戳的历史记录
    @Query("SELECT * FROM historyTable WHERE timeStamp = :timeStamp")
    List<HistoryEntity> getHistoryByTimeStamp(long timeStamp);

    // 插入一条历史记录，onConflict指的是在插入时发生冲突该怎么办
    // public @interface OnConflictStrategy {
    //     // 替换旧的数据
    //     int REPLACE = 1;
    //     // 回滚事务，相当于啥也没做，被废弃，用ABORT替代
    //     int ROLLBACK = 2;
    //     // 回滚事务
    //     int ABORT = 3;
    //     // 使事务失败(Does not work as expected. The transaction is rolled
back.)，被废弃，用ABORT替代
    //     int FAIL = 4;
    //     // 忽略冲突
    //     int IGNORE = 5;
    // }
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertHistory(HistoryEntity entity);

    // 删除一条历史记录，会根据参数里面的PrimaryKey做删除操作
```

```
    @Delete
    void deleteHistory(HistoryEntity entity);
    // 删除也可以通过Query实现，如下（更新同理）
    @Query("DELETE FROM historyTable WHERE name = :name")
    void deleteHistory2(String name);
    // 删除所有数据
    @Query("DELETE FROM historyTable")
    void deleteAllHistory();

    // 更新一条历史记录的数据，会根据参数里面的PrimaryKey做更新操作
    @Update
    void updateHistory(HistoryEntity entity);
}
```

@Update和@Insert可以设置onConflict来表明冲突的时候的解决办法。

TypeConverters

如果有一些我们自己定义的比较复杂的类，我们想把它存到数据库中，但是数据库不知道该怎么存放，因此我们就需要TypeConverters来转换一下，转换成数据库认识的数据类型。

@TypeConverters注解

```
public class EnumConverter {
    // int转枚举
    @TypeConverter
    public EnumDemo intToEnum(int status) {
        return EnumDemo.values()[status];
    }

    // 枚举转int
    @TypeConverter
    public int enumToInt(EnumDemo enumDemo) {
        return enumDemo.ordinal();
    }
}
```

Database

@Database注解可以用来创建数据库的持有者。注解定义了实体列表

在运行时，可以通过调用**Room.databaseBuilder()**或者**Room.inMemoryDatabaseBuilder()**获取实例。因为每次创建Database实例都会产生**比较大的开销**，所以应该将Database设计成**单例**的，或者直接放在Application中创建。

两种方式获取Database对象的区别:

- Room.databaseBuilder()：生成Database对象，并且创建一个存在**文件**系统中的数据库。
- Room.inMemoryDatabaseBuilder()：顾名思义，生成Database对象并且创建一个存在**内存中**的数据库。当应用退出的时候(应用进程关闭)数据库也消失。

```
// 指名数据库的实体有哪些，数据库的版本是多少
@Database(entities = {HistoryEntity.class}, version = 1)
// 指名类型转换器TypeConverter有哪些
@TypeConverters({EnumConverter.class})
```

```
public abstract class AppDataBase extends RoomDatabase {
    private static AppDataBase instance;

    // 获取historyDao的方法
    public abstract HistoryDao historyDao();

    // 数据库升级1->2
    private static final Migration migration1To2 = new Migration(1, 2) {
        @Override
        public void migrate(SupportSQLiteDatabase database) {
            // 通过执行SQL语句来改变数据库的“型”
            database.execSQL("ALTER TABLE historyTable ADD url TEXT");
        }
    };

    public static AppDataBase getInstance(Context context) {
        if (instance == null) {
            synchronized (AppDataBase.class) {
                if (instance == null) {
                    instance =
Room.databaseBuilder(context.getApplicationContext(),
                            AppDataBase.class, "app.db")
                            .addMigrations(migration1To2)
                            .build();
                }
            }
        }
        return instance;
    }
}
```

当我们的数据库的**“型”改变后**，例如某张表的结构发生了改变，多了或少了一列、某一列的数据类型发生改变，多了一张表等等，都要**增加version版本号**，并且增加**相应的Migration函数**，以便对旧的数据库进行升级（如果不升级不转换，可能会导致崩溃）

内存泄漏

内存泄漏是指该被垃圾回收的，由于有另外一个对象仍然在引用它，**导致无法回收**，造成内存泄漏，过多的内存泄漏会导致OOM。

例如：一个长期在后台运行的线程持有 Activity 的引用，这个时候 Activity 执行了 onDestroy 方法，那么这个 Activity 就是从根节点可到达并且无用的对象， 这个 Activity 对象就是泄漏的对象，给这个对象分配的内存将**无法被回收**。

常见内存泄漏

Handler 引起的内存泄漏

主线程的Looper对象不断从消息队列中取出消息，然后再交给Handler处理。如果在Activity中的Handler不是静态的，那么Handler肯定是持有Activity的引用。而每个Message对象是持有Handler的引用的（Message对象的target属性持有Handler引用），从而导致Message间接引用到了Activity。如果在Activity destroy之后，消息队列中还有Message对象，Activity是不会被回收的。**解决方法**：静态Handler，如果要持有引用，要写成弱引用，在Activity被释放的时候要清空Message，取消Handler的Runnable

单例模式引起的内存泄漏

单例的静态特性使得它的**生命周期同应用的生命周期一样长**，如果一个对象已经没有用处了，但是**单例还持有它的引用**，那么在整个应用程序的生命周期它都不能正常被回收，从而导致内存泄漏。

资源对象没有关闭引起的内存泄漏

当我们打开资源时，一般都会使用缓存。比如读写文件资源、打开数据库资源、使用Bitmap资源等等。当我们不再使用时，应该关闭它们，使得缓存内存区域及时回收。

注册/反注册未成对使用引起的内存泄漏

我们经常会在Activity的onCreate中注册广播接受器、EventBus等，如果忘记成对的使用反注册，可能会引起内存泄漏。开发过程中应该养成良好的相关，在onCreate或onResume中注册，要记得相应的在onDestroy或onPause中反注册。

非静态匿名内部类引起的内存泄漏

非静态内部类、匿名内部类 都会持有外部类的一个引用，如果有一个静态变量引用了非静态内部类或者匿名内部类，导致非静态内部类或者匿名内部类的生命周期比外部类（Activity）长，就会导致外部类在该被回收的时候，无法被回收掉，引起内存泄漏。

静态实例持有引用引起的内存泄漏

一旦静态变量初始化后，它所持有的引用只有等到进程结束才会释放。**建议**：对于生命周期比Activity长的对象，要避免直接引用Activity的context，可以考虑使用ApplicationContext

检测内存泄漏

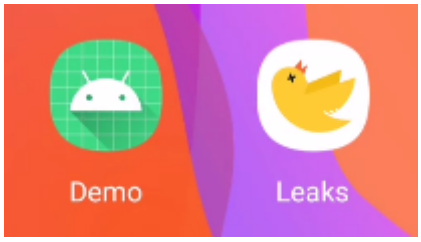
LeakCanary 检测Android的内存泄漏 https://github.com/square/leakcanary

添加依赖

```
dependencies {
  // debugImplementation because LeakCanary should only run in debug builds.
  debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.6'
}
```

debugImplementation只在debug模式下有效，在release下不会生效。

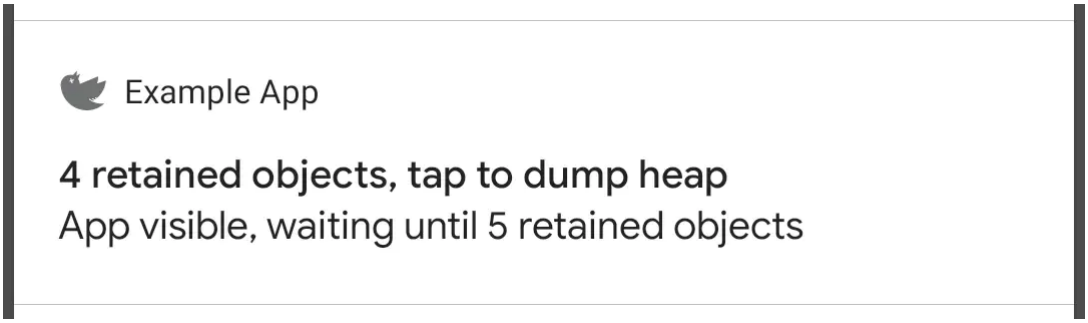
然后运行debug下app，可以看到桌面上不仅有你的app图标，还有一个Leaks



LeakCanary使用

LeakCanary的基础是一个叫做ObjectWatcher Android的library。它hook了Android的生命周期，当activity和fragment 被销毁并且应该被垃圾回收时候自动检测。这些被销毁的对象被传递给ObjectWatcher， ObjectWatcher持有这些被销毁对象的弱引用（weak references）。如果弱引用在等待5秒钟并运行垃圾收集器后仍未被清除，那么被观察的对象就被认为存在泄漏的隐患。

在使用app一段时间后，手机通知栏上面出现了这样的提示



意思是已经发现了4个保留的对象，点击通知可以触发堆转储（dump heap）。在app可见的时候，会一直等到5个保留的对象才会触发堆转储。这里要补充的一点是：当应用可见的时候默认的阈值是5，应用不可见的时候阈值是1。

> **堆转储**是JVM内存中某一时刻所有对象的快照。它们对于解决内存泄漏问题和优化Java应用程序中的内存使用非常有用。

点了上面的通知之后开始堆转储，生成.hprof文件，LeakCanary将java heap的信息存到该文件中。同时在应用程序中也会出现一个提示。

例如我把activity给一个静态变量

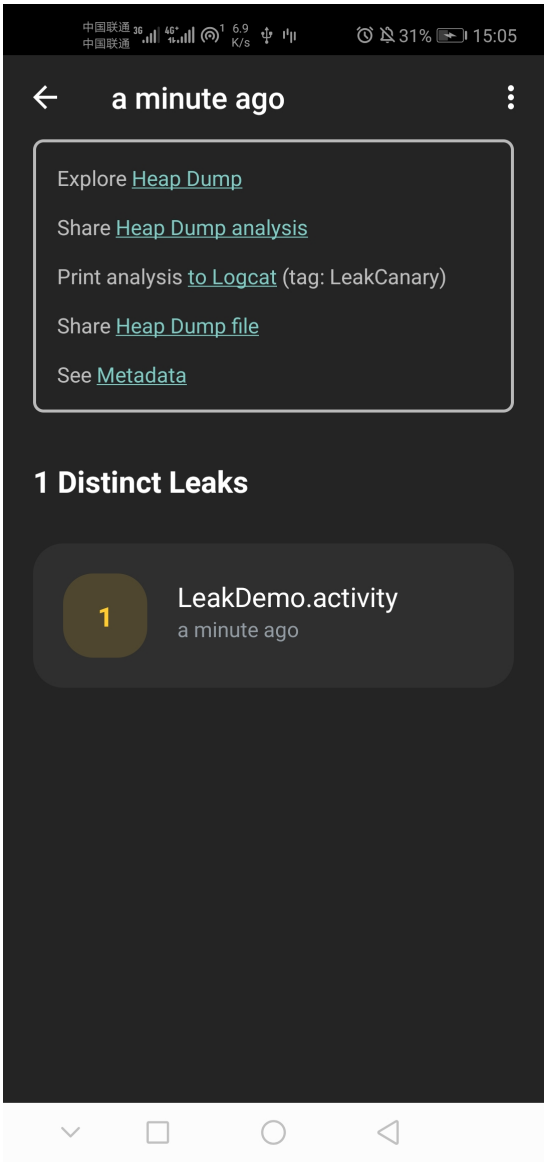
```
public class LeakDemo {
    public static Activity activity;
}
```

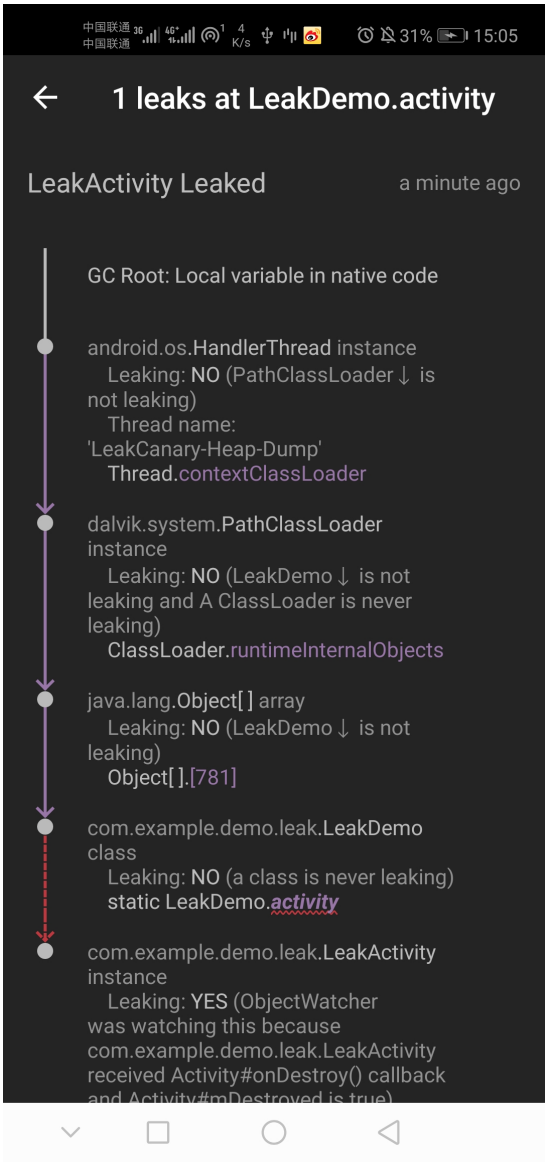
```
public class LeakActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_leak);

        // 造成内存泄漏
        LeakDemo.activity = this;
    }
}
```

如下图，显然我们可以在Leaks里面看到内存泄漏的地方，然后就可以着手进行修复了。


a minute ago
Explore Heap Dump
Share Heap Dump analysis
Print analysis to Logcat (tag: LeakCanary)
Share Heap Dump file
See Metadata
1 Distinct Leaks
1
LeakDemo.activity
a minute ago


1 leaks at LeakDemo.activity
LeakActivity Leaked
a minute ago
GC Root: Local variable in native code
android.os.HandlerThread instance
Leaking: NO (PathClassLoader ↓ is not leaking)
Thread name: 'LeakCanary-Heap-Dump'
Thread.contextClassLoader
dalvik.system.PathClassLoader instance
Leaking: NO (LeakDemo ↓ is not leaking and A ClassLoader is never leaking)
ClassLoader.runtimeInternalObjects
java.lang.Object[ ] array
Leaking: NO (LeakDemo ↓ is not leaking)
Object[ ].[781]
com.example.demo.leak.LeakDemo class
Leaking: NO (a class is never leaking)
static LeakDemo.activity
com.example.demo.leak.LeakActivity instance
Leaking: YES (ObjectWatcher was watching this because com.example.demo.leak.LeakActivity received Activity#onDestroy() callback