# 12.Bitmap

https://my.oschina.net/zbj1618/blog/2961367

## 12.1.Bitmap 内存占用的计算

占用的内存大小 = 像素总数量（图片宽x高）x 每个像素的字节大小

每个像素的字节大小与Bitmap的色彩模式有关

public static final Bitmap.Config ALPHA_8 //代表8位Alpha位图 每个像素占用1byte内存

public static final Bitmap.Config ARGB_4444 //代表16位ARGB位图 每个像素占用2byte内存

public static final Bitmap.Config ARGB_8888 //代表32位ARGB位图 每个像素占用4byte内存

public static final Bitmap.Config RGB_565 //代表8位RGB位图 每个像素占用2byte内存

假设这张图片是ARGB_8888的，那这张图片占的内存就是 width * height * 4个字节或者 width * height * inTargetDensity /inDensity * 4

**加载一张本地Res、Raw资源图片，得到的是图片的原始尺寸 * 缩放系数(inDensity)**

inTargetDensity 为当前屏幕像素密度(宽平方+高平方)/尺寸。

inDensity默认为图片所在文件夹对应的密度()

densityDpi 160 240 320 480 640

资源目录dpi mdpi hdpi xhdpi xxhdpi xxxhdpi；

像素密度 5.0英寸的手机的屏幕分辨率为1280x720，那么像素密度为192dpi

占用的内存 = width * height * targetDensity/inDensity * 一个像素所占的内存

**读取SD卡上的图,得到的是图片的原始尺寸**

占用的内存 = width * height * 一个像素所占的内存。

## 12.2.getByteCount() & getAllocationByteCount()的区别

如果被复用的Bitmap的内存比待分配内存的Bitmap大

getByteCount()获取到的是当前图片应当所占内存大小

getAllocationByteCount()获取到的是被复用Bitmap真实占用内存大小

在复用Bitmap的情况下，getAllocationByteCount()可能会比getByteCount()大。

## 12.3.Bitmap的压缩方式

质量压缩，不会对内存产生影响；

采样率压缩，比例压缩，会对内存产生影响；

**质量压缩**

不会减少图片的像素，它是在保持像素的前提下改变图片的位深及透明度等

图片的长，宽，像素都不变，那么bitmap所占内存大小是不会变的

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
src.compress(Bitmap.CompressFormat.JPEG, quality, baos);
byte[] bytes = baos.toByteArray();
Bitmap  mBitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.length);
```

**RGB_565法**

改变一个像素所占的内存

```
BitmapFactory.Options options2 = new BitmapFactory.Options();
            options2.inPreferredConfig = Bitmap.Config.RGB_565;
bm = BitmapFactory.decodeFile(Environment .getExternalStorageDirectory().getAbsolutePath()
 + "/DCIM/Camera/test.jpg", options2);
```

**采样率压缩**

设置inSampleSize的值(int类型)后，假如设为n，则宽和高都为原来的1/n，宽高都减少，内存降低

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 2;
bm = BitmapFactory.decodeFile(Environment.getExternalStorageDirectory().getAbsolutePath()+
"/DCIM/Camera/test.jpg", options);
```

**比例压缩**

根据图片的缩放比例进行等比大小的缩小尺寸，从而达到压缩的效果

压缩的图片文件尺寸变小，但是解码成bitmap后占得内存变小

Android中使用Matrix对图像进行缩放、旋转、平移、斜切等变换的。

```
Matrix matrix = new Matrix();
matrix.setScale(0.5f, 0.5f);
bm = Bitmap.createBitmap(bit, 0, 0, bit.getWidth(),bit.getHeight(), matrix, true);
```

## 12.4.LruCache & DiskLruCache原理

**LRU ( Least Recently Used )**

https://blog.csdn.net/u010983881/article/details/79050209

LruCache的核心思想就是维护一个缓存对象列表，从表尾访问数据，在表头删除数据。对象列表的排列方式是按照访问顺序实现，就是 当访问的数据项在链表中存在时，则将该数据项移动到表尾，否则在表尾新建一个数据项。当链表容量超过一定阈值，则移除表头的数据。

利用LinkedHashMap数组+双向链表的数据结构来实现的。其中双向链表的结构可以实现访问顺序和插入顺序，使得LinkedHashMap中的accessOrder设置为true则为访问顺序，为false，则为插入顺序。

写入缓存

1插入元素，并相应增加当前缓存的容量。

2调用trimToSize()开启一个死循环，不断的从表头删除元素，直到当前缓存的容量小于最大容量为止。

读取缓存

调用LinkedHashMap的get()方法，注意如果该元素存在，这个方法会将该元素移动到表尾.

**DiskLruCache**

https://juejin.cn/post/6844903556705681421#heading-6

https://nich.work/2017/DiskLruCache/

https://www.jianshu.com/p/400bda3e37ed

利用 LinkedHashMap实现算法LRU

DiskLruCache 有三个内部类，分别为 Entry、Snapshot 和 Editor。

Entry是 DiskLruCache 内 LinkedHashMap Value 的基本结构。

**Journal 文件**

DiskLruCache 通过在磁盘中创建并维护一个简单的Journal文件来记录各种缓存操作，。记录的类型有4种，分别为READ、REMOVE、CLEAN和DIRTY。

写入缓存的时候会向journal文件写入一条以DIRTY开头的数据表示正在进行写操作，当写入完毕时，分两种情况：1、写入成功，会向journal文件写入一条以CLEAN开头的文件，其中包括该文件的大小。 2、写入失败，会向journal文件写入一条以REMOVE开头的文件,表示删除了该条缓存。也就是说每次写入缓存总是写入两条操作记录。

读取的时候，会向journal文件写入一条以READ开头的文件,表示进行了读操作

删除的时候，会向journal文件写入一条以REMOVE开头的文件,表示删除了该条缓存

通过journal就记录了所有对缓存的操作。并且按照从上到下的读取顺序记录了对所有缓存的操作频繁度和时间顺序。这样当退出程序再次进来调用缓存时，就可以读取这个文件来知道哪些缓存用的比较频繁了。然后把这些操作记录读取到集合中，操作的时候就可以直接从集合中去对应的数据了。

在缓存记录之外，Journal 文件在初始化创建的时候还有一些固定的头部信息，包括了文件名、版本号和valueCount(决定每一个 key 能匹配的 Entry 数量)。

**读取**

Snapshot是Entry的快照(snapshot)。当调用 diskLruCache.get(key)时，便能获得一个Snapshot对象，该对象可用于获取或更新存于磁盘的缓存

```
String key = Util.hashKeyForDisk(Util.IMG_URL);
DiskLruCache.Snapshot snapshot = diskLruCache.get(key);
if (snapshot != null) {
  InputStream in = snapshot.getInputStream(0);
  return BitmapFactory.decodeStream(in);
}
```

1获取到缓存文件的输入流，等待被读取。

2向journal写入一行READ开头的记录，表示执行了一次读取操作。

3如果缓存总大小已经超过了设定的最大缓存大小或者操作次数超过了2000次，就开一个线程将集合中的数据删除到小于最大缓存大小为止并重新写journal文件。

4返回缓存文件快照，包含缓存文件大小，输入流等信息。

**写入**

```
DiskLruCache.Editor editor = diskLruCache.edit(key);
   if (editor != null) {
       OutputStream outputStream = editor.newOutputStream(0);
       if (downloadUrlToStream(Util.IMG_URL, outputStream)) {
           publishProgress("");
           //写入缓存
           editor.commit();
       } else {
            //写入失败
           editor.abort();
       }
   }
diskLruCache.flush();
```

1从集合中找到对应的实例（如果没有创建一个放到集合中），然后创建一个editor，将editor和entry关联起来。

2向journal中写入一行操作数据（DITTY 空格 和key拼接的文字），表示这个key当前正处于编辑状态。

newOutputStream

1向journal文件写入一行CLEAN开头的字符（包括key和文件的大小，文件大小可能存在多个 使用空格分开的）

2重新比较当前缓存和最大缓存的大小，如果超过最大缓存或者journal文件的操作大于2000条，就把集合中的缓存删除一部分，直到小于最大缓存，重新建立新的journal文件

## 12.5.如何设计一个图片加载库

https://juejin.cn/post/6844904099297624077#heading-0

1.对图片进行内存压缩；

2.高分辨率的图片放入对应文件夹；

3.缓存

4.及时回收

## 12.6.有一张非常大的图片,如何去加载这张大图片

## 12.7.如果把drawable-xxhdpi下的图片移动到drawable-xhdpi下，图片内存是如何变的。

验证 原图：1000宽X447高，位于drawable-xxhdpi（480dpi）文件包，设备Pixel-XL（560dpi）。使用默认Bitmap.Config=ARGB_8888,设置inSampleSize=2

```
    缩放比=主动设置×被动设置=1/2×(560/480)=0.5×1.166=0.5833
    一个像素所占的内存=ARGB_8888=32bit=4byte
    原始大小=1000×447

width * height *  *
    内存占用=(原始宽×缩放比)×(原始高×缩放比)×targetDensity/inDensity×一个像素所占的内存
=1000×0.5833×447×0.5833×4  =583×260×4  =606320byte  ≈0.578MB
```

## 12.8.如果在hdpi、xxhdpi下放置了图片，加载的优先级。如果是400*800*，*1080*1920，加载的优先级。

https://cloud.tencent.com/developer/article/1015960

优先会去更高密度的文件夹下找这张图片，我们当前的场景就是drawable-xxxhdpi文件夹，然后发现这里也没有android_logo这张图，接下来会尝试再找更高密度的文件夹，发现没有更高密度的了，这个时候会去drawable-nodpi文件夹找这张图，发现也没有，那么就会去更低密度的文件夹下面找，依次是drawable-xhdpi -> drawable-hdpi -> drawable-mdpi -> drawable-ldp

0dpi ~ 120dpi ldpi

120dpi ~ 160dpi mdpi

160dpi ~ 240dpi hdpi

240dpi ~ 320dpi xhdpi

320dpi ~ 480dpi xxhdpi

480dpi ~ 640dpi xxxhdpi

# 13.mvc&mvp&mvvm

## 1.MVC及其优缺点

原理

视图层(View)

一般采用XML文件进行界面的描述，这些XML可以理解为AndroidApp的View。

控制层(Controller)

Android的控制层的重任通常落在了众多的Activity的肩上。

模型层(Model)

我们针对业务模型，建立的数据结构和相关的类，就可以理解为AndroidApp的Model，Model是与View无关，而与业务相关的。对数据库的操作、对网络等的操作都应该在Model里面处理，当然对业务计算等操作也是必须放在的该层的。

缺点

随着界面及其逻辑的复杂度不断提升，Activity类的职责不断增加，以致变得庞大臃肿。

## 2.MVP及其优缺点

原理

MVP框架由3部分组成：View负责显示，Presenter负责逻辑处理，Model提供数据。

View: 显示数据, 并向Presenter报告用户行为。与用户进行交互(在Android中体现为Activity)。

Presenter: 逻辑处理，从Model拿数据，回显到UI层，响应用户的行为。

Model:负责存储、检索、操纵数据(有时也实现一个Model interface用来降低耦合)。

google todo-mvp加入契约类来统一管理view与presenter的所有的接口，这种方式使得view与presenter中有哪些功能，一目了然

优点

1.分离视图逻辑和业务逻辑，降低了耦合，修改视图而不影响模型，不需要改变Presenter的逻辑 模型与视图完全分离，我们可以修改视图而不影响模型；

2.视图逻辑和业务逻辑分别抽象到了View和Presenter的接口中，Activity只负责显示，代码变得更加简洁，提高代码的阅读性。

3.Presenter被抽象成接口，可以有多种具体的实现，所以方便进行单元测试。

Presenter是通过interface与View(Activity)进行交互的，这说明我们可以通过自定义类实现这个interface来模拟Activity的行为对Presenter进行单元测试，省去了大量的部署及测试的时间（不需要将应用部署到Android模拟器或真机上，然后通过模拟用 户操作进行测试）

缺点

1.那就是对 UI 的操作必须在 Activity 与 Fragment 的生命周期之内，更细致一点，最好在 onStart() 之后 onPause()之前，否则极其容易出现各种异常，内存泄漏。

2.Presenter与View之间的耦合度高，app中很多界面都使用了同一个Presenter 。一旦需要变更，那么视图需要变更了。

**MVP如何设计避免内存泄漏？**

Mvp模式在封装的时候会造成内存泄漏，因为presenter层，需要做网络请求，所以就需要考虑到网络请求的取消操作，如果不处理，activity销毁了，presenter层还在请求网络，就会造成内存泄漏。 **如何解决Mvp模式造成的内存泄漏？** 只要presenter层能感知activity生命周期的变化，在activity销毁的时候，取消网络请求，就能解决这个问题。 下面开始封装activity和presenter。

**定义IPresenter** 声明activity（Fragment）生命周期中的各个回调方法

```
<U extends IUI> void init(BaseActivity activity, U ui);

    /**
     * onUICreate:UI被创建的时候应该invoke这个method. <br/>
     * <p/>
     * 比如Activity的onCreate()、Fragment的onCreateView()的方法应该调用Presenter的这个方法
     *
     * @param savedInstanceState 保存了的状态
     */
    void onUICreate(Bundle savedInstanceState);

    /**
     * onUIStart:在UI被创建和被显示到屏幕之间应该回调这个方法. <br/>
```

```
     * <p/>
     * 比如Activity的onStart()方法应该调用Presenter的这个方法
     */
    void onUIStart();

    /**
     * onUIResume:在UI被显示到屏幕的时候应该回调这个方法. <br/>
     * <p/>
     * 比如Activity的onResume()方法应该调用Presenter的这个方法
     */
    void onUIResume();

    /**
     * onUIPause:在UI从屏幕上消失的时候应该回调这个方法. <br/>
     * <p/>
     * 比如Activity的onPause()方法应该调用Presenter的这个方法
     */
    void onUIPause();

    /**
     * onUIStop:在UI从屏幕完全隐藏应该回调这个方法. <br/>
     * <p/>
     * 比如Activity的onStop()方法应该调用Presenter的这个方法
     */
    void onUIStop();

    /**
     * onUIDestroy:当UI被Destory的时候应该回调这个方法. <br/>
     */
    void onUIDestroy();

    /**
     * onSaveInstanceState:保存数据. <br/>
     * <p/>
     * 一般是因为内存不足UI的状态被回收的时候调用
     *
     * @param outState 待保存的状态
     */
    void onSaveInstanceState(Bundle outState);

    /**
     * onRestoreInstanceState:当UI被恢复的时候被调用. <br/>
     *
     * @param savedInstanceState 保存了的状态
     */
    void onRestoreInstanceState(Bundle savedInstanceState);
```

**封装BaseActivity** 拥有一个 protected P mPresenter实例，类型写成泛型，protected修饰，子类实现 构造方法中，对mPresenter进行实例化：采用反射的形式（避免让子类进行实例化，繁琐）

```
public abstract class BaseMVPActivity<P extends IPresenter> extends BaseActivity {

  protected P mPresenter;
```

```
    public BaseMVPActivity(){
        this.mPresenter = createPresenter();
    }
  protected P createPresenter(){
        ParameterizedType type = (ParameterizedType)(getClass().getGenericSuperclass());
        if(type == null){
            return null;
        }
        Type[] typeArray = type.getActualTypeArguments();
        if(typeArray.length == 0){
            return null;
        }
        Class<P> clazz = (Class<P>) typeArray[0];
        try {
            return clazz.newInstance();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
        return null;
    }

 @Override
    protected void onDestroy() {
        mPresenter.onUIDestroy();
        super.onDestroy();
    }
```

**封装BasePresenter** 3.定义BasePresenter，页面销毁的回调里面。处理网络请求 定义一个集合，把每次网络请求装载到集合里面，在页面销毁的时候，取消所有的网络请求。防止内存泄漏。

```
public abstract class BasePresenter<U extends IUI> implements IPresenter {

 @Override
    public void onUIDestroy() {
        // 清空Call列表，并放弃Call请求
        clearAndCancelCallList();

    }
}
```

## 3.MVVM及其优缺点

View层包含布局,以及布局生命周期控制器(Activity/Fragment)

ViewModel与Presenter大致相同,都是负责处理数据和实现业务逻辑,但 ViewModel层不应该直接或者间接地持有View层的任何引用

model层是数据层

Model，模型层，即数据模型，用于获取和存储数据。

View，视图，即Activity/Fragment

ViewModel，视图模型，负责业务逻辑。

MVVM 的本质是 数据驱动，把解耦做的更彻底，viewModel不持有view 。

View 产生事件，使用 ViewModel进行逻辑处理后，通知Model更新数据，Model把更新的数据给ViewModel，ViewModel自动通知View更新界面，而不是主动调用View的方法

LiveData是具有生命周期的可观察的数据持有类。理解它需要注意这几个关键字，生命周期，可观察，数据持有类。

DataBinding用来实现View层与ViewModel数据的双向绑定，Data Binding 減輕原本 MVP 中 Presenter 要與 p 和 View 互動的職責

MVVM的优点

核心思想是观察者模式,它通过事件和转移View层数据持有权来实现View层与ViewModel层的解耦.

1.耦合度更低，复用性更强，没有内存泄漏

2.结合jetpack，写出更优雅的代码

缺点

ViewModel与View层的通信变得更加困难了,所以在一些极其简单的页面中请酌情使用,否则就会有一种脱裤子放屁的感觉,在使用MVP这个道理也依然适用.

---

# 4.MVC与MVP区别

View与Model并不直接交互，而是通过与Presenter交互来与Model间接交互。而在MVC中View可以与Model直接交互。MVP 隔离了MVC中的 M 与 V 的直接联系后，靠 Presenter 来中转，所以使用 MVP 时 P 是直接调用 View 的接口来实现对视图的操作的，

# 5.MVP如何管理Presenter的生命周期，何时取消网络请求

https://blog.csdn.net/mq2553299/article/details/78927617

https://www.jianshu.com/p/0d07fba84cb8

使用RxLifecycle，通过监听Activity、Fragment的生命周期，来自动断开subscription以防止内存泄漏。

RxLifecycle原理

**操作符**

1.takeUntil 发射来自原始Observable的数据，如果第二个Observable发射了一项数据或者发射了一个终止通知， 原始Observable会停止发射并终止。

2.CombineLatest 当两个Observables中的任何一个发射了数据时，使用一个函数结合每个Observable发射的最近数据项，并且基于这个函数的结果发射数据。

**流程**

1.BehaviorSubject 在ActivityActivity不同的生命周期，BehaviorSubject对象会发射对应的ActivityEvent，比如在onCreate()生命周期发射ActivityEvent.CREATE，在onStop()发射ActivityEvent.STOP。

BehaviorSubject，实际上也还是一个Observable

BehaviorSubject（释放订阅前最后一个数据和订阅后接收到的所有数据）

2.LifecycleTransformer 在自己的请求中bindActivity返回的是LifecycleTransformer

LifecycleTransformer中使用upstream.takeUntil(observable)，

upstream就是原始的Observable

observable是Activity中的BehaviorSubject

**指定生命周期断开**

原始的Observable通过takeUntil和BehaviorSubject绑定，当BehaviorSubject发出数据即Activity的生命周期走到和你指定销毁的事件一样时,BehaviorSubject才会把事件传递出去,原始的Observable就终止发射数据了

**不指定生命周期断开**

combineLatest() 当两个Observables中的任何一个发射了数据时，使用一个函数结合每个Observable发射的最近数据项，并且基于这个函数的结果发射数据。不指定生命周期时bindToLifecycle中通过combineLatest组合

比如说我们在onCreate()中执行了bindToLifecycle，那么lifecycle.take(1)指的就是ActivityEvent.CREATE，经过map(correspondingEvents)，这个map中传的函数就是 1中的ACTIVITY_LIFECYCLE

lifecycle.skip(1)就简单了，除去第一个保留剩下的，以ActivityEvent.Create为例，这里就剩下：ActivityEvent.START ActivityEvent.RESUME ActivityEvent.PAUSE ActivityEvent.STOP ActivityEvent.DESTROY

三个参数 意味着，lifecycle.take(1).map(correspondingEvents)的序列和 lifecycle.skip(1)进行combine，形成一个新的序列：

false,false,fasle,false,true

这意味着，当Activity走到onStart生命周期时，为false,这次订阅不会取消，直到onDestroy，为true，订阅取消。

最终还是和lifecycle（BehaviorSubject）发射的数据比较，如果两个一样说明Activity走到了该断开的生命周期了，upstream.takeUntil(observable)中的observable就要发通知告诉upstream（原始的Observable）该断开了。