

Android

架构

- MVP
- MVVM
- lifecycle
- viewModel

ViewModel的生命周期只有在绑定的Activity或Fragment销毁的时候才会清空数据，所以在屏幕旋转后数据还是能够继续使用恢复到旋转前的状态。

ViewModel 在对应的作用域内保持**生命周期内的局部单例**，这样就可以通过ViewModel实现Fragment 与 Fragment之间，Activity 与 Fragment 之间的通讯以及数据共享。

- 通过 onSaveInstanceState() 存储数据，如果数据量大，不方便序列化及反序列化，则这个方法不适合。

如何保证数据不随着屏幕旋转而销毁

`setRetainInstance(boolean)` 是 `Fragment` 中的一个方法。将这个方法设置为true 就可以使当前 `Fragment` 在 `Activity` 重建时存活下来

如何保证在不同的Fragment()中，通过以下代码生成同一个ViewModel的实例呢

```
public <T extends ViewModel> T get(Class<T> modelClass) {  
    // 先从ViewModelStore容器中去找是否存在ViewModel的实例  
    ViewModel viewModel = mViewModelStore.get(key);  
    // 若ViewModel已经存在，就直接返回  
    if (modelClass.isInstance(viewModel)) {  
        return (T) viewModel;  
    }  
    // 若不存在，再通过反射的方式实例化ViewModel，并存储进ViewModelStore  
    viewModel = modelClass.getConstructor(Application.class).newInstance(mApplication);  
    mViewModelStore.put(key, viewModel);  
    return (T) viewModel;  
}
```

- databinding
- livedata

LiveData是一个可观测的数据持有类，当数据发生改变时，并且lifecycle对象处于活跃状态的时候，LiveData将立即通知观察者数据发生了变化，也就是说比普通观察者多了一个**生命周期感知能力**,可以在Activity或Fragment 组件生命周期发生改变时停止或恢复之前的状态。

LiveData 原生的API提供了2种方式供开发者更新数据，分别是 `setValue()` 和 `postValue()`，官方明确 `setValue`方法必须在主线程进行调用，而`postValue()` 方法更适合在执行较重工作的子线程中进行调用(比如网络请求等)，在所有情况下，调用`setValue ()` 或 `postValue()` 都会触发观察者并更新UI

- `postValue()` 之所以能够在子线程更新UI，因为其内部使用Handler

- ButterKnife

```
UPDATE OR ABORT `DeviceGroupModel` SET `primaryKey` = ?, `deviceKey` = ?, `displayName` = ?, `sortIndex` = ? WHERE `primaryKey` = ?
```

Android 版本特性

Android 8.0

<https://weilu.blog.csdn.net/article/details/80965631>

1. 新增权限分组, 可以对权限进行分组
2. 悬浮窗适配 应用必须使用名为 `TYPE_APPLICATION_OVERLAY` 的新窗口类型

用之前判断一下

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) { mWindowParams.type =  
    WindowManager.LayoutParams.TYPE_APPLICATION_OVERLAY } else { mWindowParams.type =  
    WindowManager.LayoutParams.TYPE_SYSTEM_ALERT }
```

还需要添加权限

Android 9.0

1. 通过 `startForegroundService` 来创建一个前台服务的话, 需要申请 `FOREGROUND_SERVICE` 权限
在 `AndroidManifest.xml` 中 添加
2. 9.0 中不能直接在 `Activity` 环境中 (比如 `Service`, `Application`) 启动 `Activity`, 否则会崩溃报错
解决: 在 `Intent` 中添加 `FLAG_ACTIVITY_NEW_TASK`

```
Intent intent = new Intent(this, TestActivity.class); intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
startActivity(intent);
```

- 3.刘海屏适配 官方提供了 `DisplayCutout` 类 进行适配

MIUI 系统, Android P 无法通过这个 API 判断是否刘海屏

<https://dev.mi.com/console/doc/detail?plid=1341>

4. 限制访问通话记录

如果应用需要访问通话记录或者需要处理去电, 则您必须向 `CALL_LOG` 权限组明确请求这些权限。否则会发生 `SecurityException`。

5. 限制访问电话号码

要通过 `PHONE_STATE` Intent 操作读取电话号码，同时需要 `READ_CALL_LOG` 权限和 `READ_PHONE_STATE` 权限。

要从 `PhoneStateListener` 的 `onCallStateChanged()` 中读取电话号码，只需要 `READ_CALL_LOG` 权限。不需要 `READ_PHONE_STATE` 权限

6. 如果您的应用必须在多个进程中使用 `WebView` 实例，则您必须先使用 `WebView.setDataDirectorySuffix()` 方法为每个进程指定唯一的数据目录后缀，然后再在相应进程中使用 `webView` 的给定实例

`WebView.setDataDirectorySuffix()` 多进程添加就是这个方法，记住，`application` 中就要添加。

Android 10

作用域存储

https://blog.csdn.net/guolin_blog/article/details/105419420

Android10之前, 应用可以在SD卡创建一个属于自己的专属目录,可以存放各类文件

好处:存储到SD卡的文件不会记到应用程序的占用空间当中, 即使应用被删除了存储在SD卡的文件也会保留下来,可以保留一些需要永久保留的数据.

坏处:应用删除后,保留的SD卡的垃圾文件可能会一直保存,不会被清理. 同时因为是公共空间,所以其他软件也可以访问, 对数据不安全.

从Android 10 开始 每个应用程序只能有权在自己的外置存储控件关联的目录下读取和创建文件, 获取该关联目录的代码是: `context.getExternalFilesDir()`。关联目录对应的路径大致如下:

`/storage/emulated/0/Android/data/<包名>/files`

将数据放到这个目录下, 你将可以完全使用之前的写法来对文件进行读写, 不需要做任何变更和适配。这个目录中的文件会被计入到应用程序的占用空间中, 同时也会随着应用程序的卸载而被删除。

我们的应用程序向媒体库贡献的图片、音频或视频, 将会自动拥有其读写权限, 不需要额外申请 `READ_EXTERNAL_STORAGE` 和 `WRITE_EXTERNAL_STORAGE` 权限。而如果你要读取其他应用程序向媒体库贡献的图片、音频或视频, 则必须申请 `READ_EXTERNAL_STORAGE` 权限才行。`WRITE_EXTERNAL_STORAGE` 权限将会在未来的Android版本中废弃。, 图片、音频、视频这三类文件将可以通过 **MediaStore API** 来进行访问, 而其他类型的文件则需要使用系统的文件选择器来进行访问。

将图片添加到相册

```
fun addBitmapToAlbum(bitmap: Bitmap, displayName: String, mimeType: String, compressFormat: Bitmap.CompressFormat) {
    ContentResolver resolver = context.getContentResolver();
```

```

val values = ContentValues()
//设置文件名
values.put(MediaStore.MediaColumns.DISPLAY_NAME, displayName)
//设置文件类型 如 image/*
values.put(MediaStore.MediaColumns.MIME_TYPE, mimeType)
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
    values.put(MediaStore.MediaColumns.RELATIVE_PATH, Environment.DIRECTORY_DCIM)
} else {
    //在Android 10中 MediaColumns.DATA 已废弃
    values.put(MediaStore.MediaColumns.DATA,
"${Environment.getExternalStorageDirectory().path}/${Environment.DIRECTORY_DCIM}/$displayName")
}
//第一个参数是外部存储路径
val uri = resolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)
if (uri != null) {
    val outputStream = contentResolver.openOutputStream(uri)
    if (outputStream != null) {
        bitmap.compress(compressFormat, 100, outputStream)
        outputStream.close()
    }
}
}

//最后通过 openOutputStream 获取输出流, 然后将 Bitmap 对象写如输出流中即可.

```

下载文件到Download 目录

targetSdkVersion 低于 29 , 不做任何作用域存储的适配,是没问题的

targetSdkVersion >=29 , 如果不想进行适配,需要 加上 android:requestLegacyExternalStorage="true"

如果需要读取SD卡上非图片,音频,视频类的文件,比如打开一个PDF文件,这时候需要使用手机系统内置的文件选择器.

Android 11

引入了权限过期的机制，本来用户授予了应用程序某个权限，该权限会一直有效，现在如果某应用程序很长时间没有启动，Android系统会自动收回用户授予的权限，下次启动需要重新请求授权。

Android 11针对摄像机、麦克风、地理定位这3种权限提供了单次授权的选项。因为这3种权限都是属于隐私敏感的权限，如果选择了这个选项，整个生命周期内，这个权限一直在，但在下次启动程序的时候，需要重新请求权限。

Android 10系统首次引入了android:foregroundServiceType属性，如果你想要在前台Service中获取用户的位置信息，那么必须在AndroidManifest.xml中进行以下配置声明：

```
<manifest>
    ...
    <service ...
        android:foregroundServiceType="location" />
</manifest>
```

而在Android 11系统中，这个要求扩展到了摄像头和麦克风权限。也就是说，如果你想要在前台Service中获取设备的摄像头和麦克风数据，那么也需要在AndroidManifest.xml中进行声明：

```
<manifest>
    ...
    <service ...
        android:foregroundServiceType="location|camera|microphone" />
</manifest>
```

Android 11 更改了您的应用在读取电话号码时使用的与电话相关的权限。

其实就是这两个API

- TelecomManager 类中的 `getLine1Number()` 方法
- TelecomManager 类中的 `getMsisdn()` 方法

当用到这两个API的时候，原来的 `READ_PHONE_STATE` 权限不管用了，需要 `READ_PHONE_NUMBERS` 权限才行。

APK 需要 v2 签名方案

对于以 Android 11 (API 级别 30) 为目标平台，且目前仅使用 APK 签名方案 v1 签名的应用，现在还必须使用 **APK 签名方案 v2 或更高版本进行签名**。用户无法在搭载 Android 11 的设备上安装或更新仅通过 APK 签名方案 v1 签名的应用

5G

Android 11 添加了在您的应用中支持 5G 的功能

- 检测是否连接到了5G网络
- 检查按流量计费性

软件包可见性 ★

Android 11 中，如果你想去获取其他应用的信息，比如包名，名称等等，不能直接获取了，必须在清单文件中添加 `<queries>` 元素，告知系统你要获取哪些应用信息或者哪一类应用。

前台服务类型

从 Android 9 开始，应用仅限于在前台访问摄像头和麦克风。为了进一步保护用户，Android 11 更改了前台服务访问摄像头和麦克风相关数据的方式。如果您的应用以 Android 11 为目标平台并且在某项前台服务中访问这些类型的数据，您需要在该前台服务的声明的 `foregroundServiceType` 属性中添加新的 `camera` 和 `microphone` 类型。

链接：<https://juejin.im/post/6860370635664261128>

```
<service ... android:foregroundServiceType="location|camera|microphone" />
```

Android 版本适配

<https://juejin.im/post/6844903713178386446#heading-1>

数据库

- sqlite
 - greenDao
 - room

@Entity

表明该类是持久化的类【持久化含义，存入数据库文件中，作本地化处理】 可配置参数：

1. schema： 告知GreenDao当前实体属于哪个schema
2. active： 标记一个实体处于活动状态，活动实体有更新、删除和刷新方法
3. nameInDb： 在数据中使用的别名， 默认使用的是实体的类名
4. indexes： 定义索引，可以跨越多个列
5. createInDb： 标记创建数据库表

@Id

选择一个long或Long类型的属性作为该实体所对应数据库中数据表的主键【类型要是long】

@Generated

写在构造方法前

@Property

对应于表中的字段，如果不设置，则默认java类属性对应相同的表字段

@NotNull

使一个属性不为空在数据库表中

@Transient

指定属性不具有持久性，仅仅使用一个暂时的状态

@index

指定一个属性作为索引中的字段 可配置的参数 1) name： 修改在索引中的字段名，不使用默认的。 2) unique：使此字段的值唯一。

@Unique

给字段一个UNIQUE约束

@ToOne

一对—

可配置的参数：

1. joinProperty

GreenDao数据库操作指令：

```
- eq(): ==
- noteq(): !=
- gt(): >
- lt(): <
- ge: >=
- le:<=
- like(): 包含
- between: 俩者之间
- in:在某个值内
- notIn:不在某个值内
```

Constrainlayout 布局使用

<https://juejin.im/post/6854573221312725000>

进程间通信

在Android中使用多进程只有**一种方法**(还有一种非常规的,通过JNI在native层fork一个新的线程), 那就是给四大组件在 **AndroidManifest** 中指定 **android: process属性。**，我们无法给一个线程或者一个实体类指定其运行所在的进程。

process 属性以 “：“ 开头 表示在当前进程名前面附加上当前的包名，这是一种简写的方法，其次，表示当前应用属于**私有进程**，其他应用的组件**不可以和它跑在同一个进程中**。而不以 “：“ 开头的进程属于**全局进程**，其他应用**通过 ShareUID 方式可以和它跑在同一进程中**。

Android 系统中，每个应用都会分配一个位移的UID，具有相同UID的应用才能共享数据。两个应用通过ShareUID跑在同一个进程中是有要求的，需要这两个应用有**相同的ShareUID并且签名相同**才可以。它们可以互相访问对方的私有数据，比如data目录，组件信息等，不管是否跑在同一个进程中，当然如果它们跑在同一个进程中还能共享内存数据。

所有运行在不同进程的 四大组件，只要它们之间通过内存来共享数据，都会共享失败。

一般来说使用多进程会造成如下几个方面的问题：

1. 静态成员和单例模式完全失效。
2. 线程同步机制完全失效。

不是一块内存，那么不管是锁对象还是锁全局类都无法保证线程同步，因为不同的进程锁不是同一个对象。

3. SharedPreferences的可靠性下降。

SharedPreferences 不支持两个进程同时去执行写操作，否则会导致一定几率的数据丢失。因为并发读写会出现问题。

4. Application会多次创建。

运行在同一个进程中的组件是属于同一个虚拟机和同一个Application的，同理，运行在不同进程中的组件是属于两个不同的虚拟机和Application的。

socket：传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。

消息队列和管道：采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后从内核缓存拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难以使用。

binder：只需要一次拷贝，性能上仅次于共享内存。基于C/S架构，职责明确，架构清晰，稳定性好。为每个App分配UID，用于鉴别进程身份，提高安全性。

序列化

`Serializable` 是Java中的序列化接口，使用起来简单但是开销很大，序列化和反序列化过程需要大量的I/O操作。

`Parcelable` 是Android中的序列化方式，因此更适合在Android平台上使用，缺点就是使用起来比较麻烦，但是效率高。

Serializable实现

```
public class User implements Serializable {  
  
    private static final long serialVersionUID = 519067123721295773L;  
  
    public int userId;  
    public String userName;  
    public boolean isMale;  
}
```

使用`Serializable` 实现序列化很简单，只需要实现`Serializable`接口，然后声明一个`SerialVersionUID`即可，其实不声明这个`UID`也是可以的，不过这样会对反序列话过程产生影响。

序列化和反序列化

```
//序列化过程  
User user = new User(0, "jake", true);  
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("cache.txt"));  
out.writeObject(user);  
out.close();  
  
//反序列化过程  
ObjectInputStream in = new ObjectInputStream(new FileInputStream("cache.txt"));  
User newUser = in.readObject();  
in.close();
```

上述代码演示了采用`Serializable`方式序列化对象的过程，很简单，只需要把实现了`Serializable` 接口的`User` 对象写到文件中就可以快速恢复了，恢复后的对象`newUser` 和 `user` 的内容完全一样，但是两者并不是同一个对象。

为什么需要 `serialVersionUID` 呢, `serialVersionUID` 的详细工作机制是这样的, 序列化的时候系统会把当前类的 `serialVersionUID` 写入序列化文件中(也可能是其他中介), 当反序列化的时候系统会检测文件中的 `serialVersionUID`, 看它是否和当前类的 `serialVersionUID` 一致, 如果一致说明序列化的类和当前类的版本是相同的, 这个时候就可以成功反序列化.

Parcelable 接口实现

只要实现 Parcelable 接口, 一个类的对象就可以实现序列化并通过 Intent 和 Binder 传递.

```
public class User implements Parcelable {

    public int userId;
    public String userName;
    public boolean isMale;

    public Book book;
    public User(int userId, String userName, boolean isMale) {
        this.userId = userId;
        this.userName = userName;
        this.isMale = isMale;
    }

    public int describeContents() {
        return 0;
    }
    //序列化
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(userId);
        out.writeString(userName);
        out.writeInt(isMale ? 1 : 0);
        out.writeParcelable(book, 0);
    }
    //反序列化
    public static final Parcelable.Creator<User> CREATOR = new Parcelable.Creator<User>() {
        public User createFromParcel(Parcel in) {
            return new User(in);
        }

        public User[] newArray(int size) {
            return new User[size];
        }
    };

    private User(Parcel in) {
        userId = in.readInt();
        userName = in.readString();
        isMale = in.readInt() == 1;
        book = in.readParcelable(Thread.currentThread().getContextClassLoader());
    }
}
```

方法	功能	标志位
createFromParcel(Parcel in)	从序列化后的对象中创建原始对象	
newArray(int size)	创建指定长度的原始对象数组	
writeToParcel(Parcel out, int flags)	将当前对象写入序列化结构中,其中flags标志有两种值,0或者1. 为1时标识当前对象需要作为返回值返回,不能立即释放资源,几乎所有情况都为0	PARCELABLE_WRITE_RETURN_VALUE
User(Parcel in)	从序列化后的对象中创建原始对象	
describeContents	返回当前对象的内容描述, 如果含有文件描述符, 返回1, 否则返回0 , 几乎所有情况都返回0	CONTENS_FILE_DESCRIPTOR

Parcelable **主要用在内存序列化上**, 通过Parcelable 将对象序列化到存储设备中或者将对象序列化后通过网络传输也是可以的. 但是这个过程会比较复杂, 因此这两种情况下建议大家使用Serializable.

IPC通信方式

IPC 是 Inter-process Communication 的缩写, 意为**进程间通信或者跨进程通信**.

使用Bundle

Bundle实现了 Parcelable 接口, 所以可以方便在不同进程间进行传输, 当我们在一个进程中启动另一个进程的Activity, Service 和 Receiver, 我们就可以在Bundle 中附加我们需要传输给远程进程的信息并通过 Intent 发送出去.

使用文件共享

共享文件也是一种不错的进程间通信的方式, 两个进程通过读/写同一个文件来交换数据, 比如A进程把数据写入文件, B进程通过读取这个文件来获取数据.

通过文件共享这种方式来共享数据**对文件格式是没有具体要求的**, 只要读/写双方约定数据格式即可.

通过文件共享的方式也是有**局限性的**, 比如并发读/写问题. 所以**文件共享只适合在数据同步要求不高的进程间进行通信, 并处理好读/写并发的问题**.

SharedPreferences 本质上也属于文件的一种, 但是由于系统对它的读/写有一定的缓存策略, 即在内存中会有一份SharedPreferences 文件的缓存, 因此在多线程的模式下, 系统对它的读/写就变得不可靠了, 当面对高并发的读/写访问, SharedPreferences 有很大的几率会丢失数据, 因此**不建议在进程间通信中使用SharedPreferences**.

使用Messenger

Messenger可以翻译为信使, 通过它可以在不同进程中传递Messenger对象, 在Messenger中放入我们需要传递的数据, 就可以轻松的实现数据的进程间传递了. Messenger是一种轻量级的IPC方案, 它底层实现是ADL. Messenger一次处理一个请求, 因此在服务端我们不用考虑线程同步的问题, 这是因为在服务端中不存在并发执行的情形.

在Messenger中进行数据传递必须将数据放入Message 中, 而Messenger 和 Message 都实现了 Parcelable接口, 因此可以进行跨进程传输. 简单来说, Message中所支持的数据类型就是Messenger所支持的传输类型.

Messenger 是以**串行的方式处理客户端发来的消息**,如果大量的消息同时发送到服务端, 服务端仍然只能一个个进行处理,如果有大量的并发请求, 那么Messenger就不太合适了.

使用 AIDL

Messenger不适合处理大量的并发请求, 同时Messenger主要作用的传递消息, 很多时候我们可能需要跨进程调用服务端的方法,这种情形使用Messenger就无法做到了, 但是我们可以通过AIDL 来实现跨进程的方法调用, AIDL 也是Meessenger的底层实现, 因此Meessenger 本质上也是AIDL, 只不过系统为我们做了封装, 从而方便调用而已.

使用ContentProvider

ContentProvider 是 Android中提供专门用于不同应用间进行数据共享的方式, 和 Messenger 一样 ,
ContentProvider的底层实现同样也是Binder

使用Socket

Socket 也称为"套接字", 是网络通信的概念, 分为**流式套接字和用户数据报套接字**两种, 分别对应网络的传输控制层中的 **TCP 和 UDP**

TCP协议是面向连接协议, 提供稳定的双向通信功能, TCP连接建立需要经过 "三次握手" 过程, 为了提供稳定的数据功能, 其本身提供了超时重传机制, 因此具有很高的稳定性.

UDP 是无连接的, 提供不稳定的单向通信功能, 当然UDP也可以实现双向通信功能, 在性能上,UDP 具有更好的效率, 缺点是不保证数据一定能够正确传输.

名称	优点	缺点	使用场景
Bundle	简单易用	只能传输Bundle支持的数据类型	四大组件间进程间的通信
文件共享	简单易用	不适合高并发场景,并且无法做到进程间的即时通信	无并发访问的场景,交换简单的数据实时性不高的场景
AIDL	功能强大,支持一对多并发通信,支持实时通信	使用稍复杂,需要处理好线程同步	一对多通信且有RPC需求
Messenger	功能一般,支持一对多串行通信,支持实时通信	不能很好的处理高并发的情形,不支持RPC,数据通过Message进行传输,因此只能传输Bundle支持的数据类型	低并发一对多即时通信,无RPC需求,获取无需返回结果的RPC需求
ContentProvider	在数据源访问方面功能强大,支持一对多并发数据共享,可通过call方法扩展其他操作	可以理解为受约束的AIDL,主要提供数据源的CRUD操作	一对多的进程间的数据共享
Socket	功能强大,可以通过网络传输字节流,支持一对多并发实时通信	实现细节稍微有点繁琐,不支持直接RPC	网络数据交换

Binder机制

链接: <https://juejin.im/post/5acccf845188255c3201100f#heading-2>

Binder是Android的一种跨进程方式，它实现了IBinder接口，该通信方式在linux中是没有的。

- 一个进程的**Binder线程数默认最大是16**，超过的请求会被阻塞等待空闲的Binder线程。
- 所以，在进程间通信时处理并发问题时，如使用**ContentProvider**时，它的**CRUD**（创建、检索、更新和删除）方法只能同时有16个线程同时工作

内存映射

Binder IPC 机制中涉及到的内存映射通过mmap()来实现，mmap()是操作系统中一种内存映射的方法。内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。

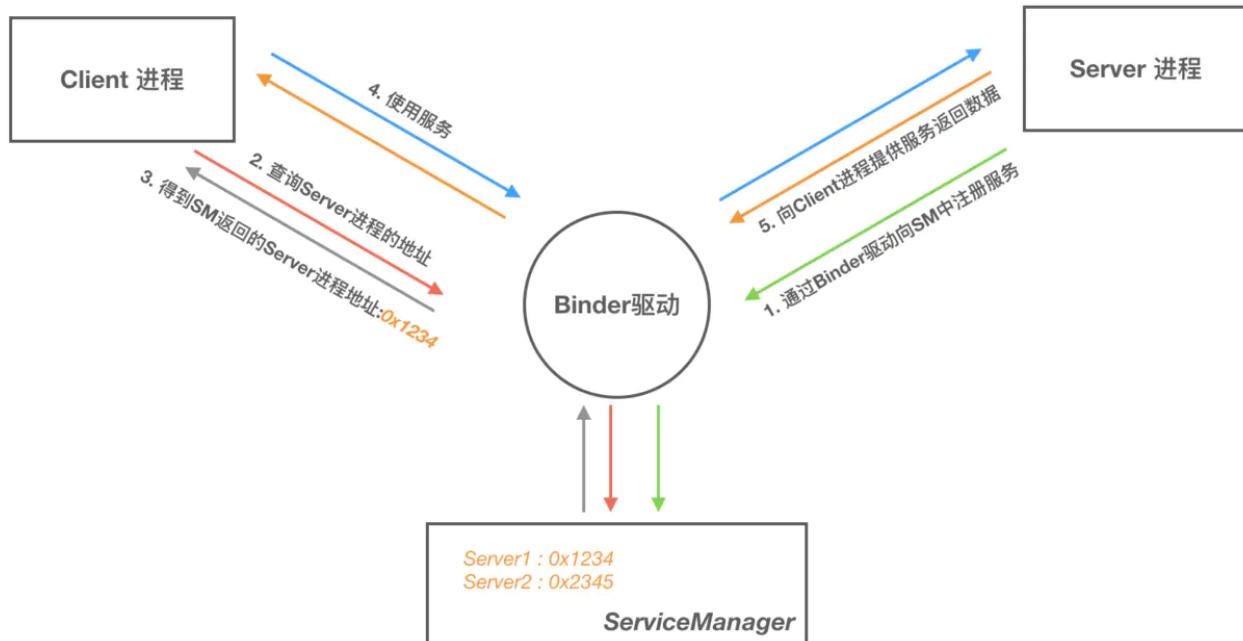
内存映射能减少数据拷贝次数，实现用户空间和内核空间的高效互动。两个空间各自的修改能直接反映在映射的内存区域，从而被对方空间及时感知。也正因为如此，内存映射能够提供对进程间通信的支持。

IBinder: IBinder是一个接口，代表了一种跨进程通信的能力。只要实现了这个借口，这个对象就能跨进程传输。

IInterface : IInterface 代表的就是 Server 进程对象具备什么样的能力（能提供哪些方法，其实对应的就是 AIDL 文件中定义的接口）

Binder : Java 层的 Binder 类，代表的其实就是 Binder 本地对象。BinderProxy 类是 Binder 类的一个内部类，它代表远程进程的 Binder 对象的本地代理；这两个类都继承自 IBinder, 因而都具有跨进程传输的能力；实际上，在跨越进程的时候，Binder 驱动会自动完成这两个对象的转换。

Stub : AIDL 的时候，编译工具会给我们生成一个名为 Stub 的静态内部类；这个类继承了 Binder, 说明它是一个 Binder 本地对象，它实现了 IInterface 接口，表明它具有 Server 承诺给 Client 的能力；Stub 是一个抽象类，具体的 IInterface 的相关实现需要开发者自己实现。



linkToDeath 和 unlinkToDeath

这是Binder很重要的方法,通过linkToDeath 我们可以给Binder 设置一个死亡代理,当Binder死亡的时候我们就会收到通知,这个时候我们就可以重新发起连接请求从而恢复连接.

首先,声明一个DeathRecipient对象, DeathRecipient是一个接口,其内部只有一个方法binderDied , 我们需要实现这个方法.当Binder死亡的时候, 系统就会回调这个方法, 然后我们就可以移除之前绑定的binder代理并重新绑定远程服务.

```
private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient() {  
  
    @Override  
    public void binderDied() {  
        if(mBookManager == null) {  
            return;  
        }  
        mBookManager.asBinder().unlinkToDeath(mDeathRecipient, 0);  
        mBookManager == null;  
        //这里重新绑定远程Service  
    }  
}
```

其次,在客户端绑定远程服务成功后,给binder设置死亡代理.

```
mService = TMessageBoxManager.Stub.asInterface(binder);  
binder.linkToDeath(mDeathRecipient, 0);
```

其中linkToDeath 的第二个参数是个标志位, 我们直接设置0即可, 经过上面的步骤,就给我们的Binder设置了死亡代理, 当Bidner死亡的时候我们就会收到通知. 另外通过Binder的方法 isBinderAlive 也可以判断Binder是否死亡.

设计模式

链接:[Java设计模式](#)

创建型

- 简单工厂模式
- 工厂模式
- 抽象工厂模式
- 建造者模式

创建型主要为了产生实例对象

结构型

- 适配器模式
- 桥梁模式
- 装饰模式
- 外观模式

结构型主要通过改变代码结构来达到解耦的目的, 使我们的代码更容易维护和扩展.

行为型

- 策略模式
- 观察者模式
- 责任链模式
- 模板模式

行为型模式关注的是各个类之间的相互作用,将职责划分清楚,使得我们的代码更加清晰.

模板模式

模板方法只负责定义第一步做什么,第二步做什么,至于怎么做,由子类来实现

- 提高代码复用性
- 将相同的部分放在抽象的父类中,而将不同的代码放入不同的子类中,让子类自己去实现
- 提高拓展性

缺点:引入了抽象类,每一个不同的实现都需要一个子类来实现,导致类的个数增加,从而增加了系统实现的复杂度。

使用方法:

```
/**
 * 模板模式
 * @Author Liang Jingyan
 * 20-7-8
 */
public abstract class AbstractTemplate {

    public void templateMethod() {
        init();
        apply();
        end();
    }

    //相同的代码在父类这里实现
    protected void init() {
        Log.e("AbstractTemplate", "init: 抽象层已经实现, 子类也可以选择覆写" );
    }

    //留给子类实现
    protected abstract void apply();

    protected void end();
}
```

```
/**
 * @Author Liang Jingyan
 * 20-7-8
 */
public class ConcreteTemplate extends AbstractTemplate {

    //不同的方法在子类实现
}
```

```
    @Override
    protected void end() {
        Log.e("ConcreteTemplate", "end: 我们可以把 method3 当做钩子方法来使用, 需要的时候覆写就可以了" );
    }

    @Override
    protected void apply() {
        Log.e("ConcreteTemplate", "apply: 子类实现抽象方法 " );
    }
}

//在需要的地方调用
AbstractTemplate t = new ConcreteTemplate();
// 调用模板方法
t.templateMethod();
```

桥接模式

<https://juejin.im/entry/59b408d7f265da065352b233>

桥接模式是一种很实用的结构型设计模式，如果软件系统中某个类存在两个独立变化的维度，通过该模式可以将这两个维度分离出来，使两者可以独立扩展，让系统更加符合“单一职责原则”。与多层继承方案不同，它将两个独立变化的维度设计为两个独立的继承等级结构，并且在抽象层建立一个抽象关联，该关联关系类似一条连接两个独立继承结构的桥，故名桥接模式

试想一下，当我们在绘画时需要大中小三种型号的画笔，并且能绘制12种颜色。当我们选择蜡笔时，为了满足这个需求，我们需要 $12 \times 3 = 36$ 支蜡笔。而同样的情况，如果我们选择油彩笔时，我们仅需3支不同型号的油彩笔，配合12种不同的颜料就可以了，总共需要 $3 + 12 = 15$ 个物品。而且当我们需要增加一种型号的画笔并且也需要绘制12种颜色，蜡笔需要增加12支，而油彩笔仅需要增加一支不同型号的笔就行。为什么同样一个需求，选择不同的画笔会有不同的结果呢？

类似矩阵

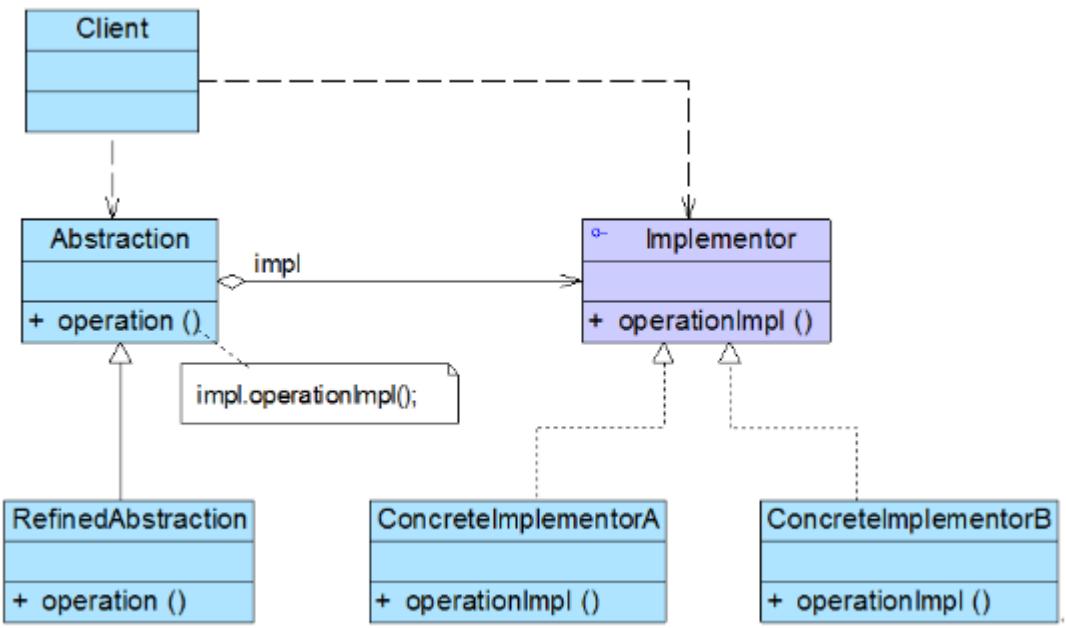
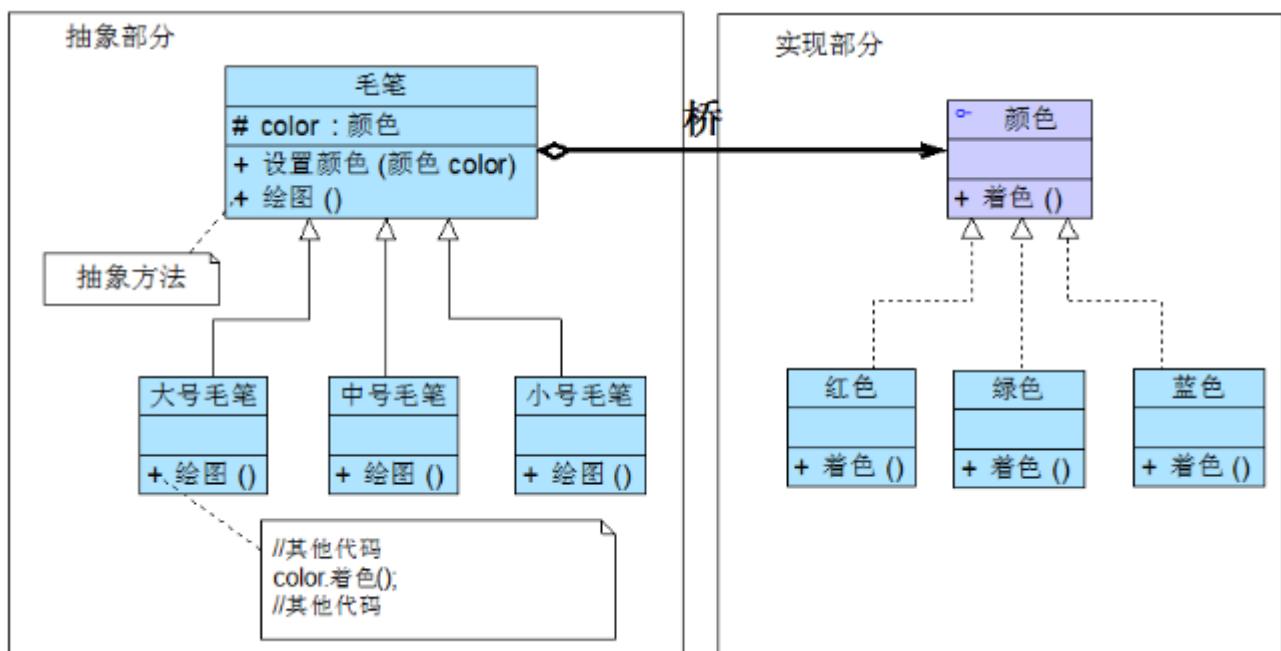


图 10-3 桥接模式结构图。



画笔抽象类

```
Abstraction.java × ClientActivity.java × ConcreteImplementorA.java × DataParser.java ×
1 package com.ljy.ofoview.qiaojiemoshi;
2
3 /**
4 * 毛笔
5 * @Author Liang Jingyan
6 * 20-5-27
7 */
8 public abstract class Abstraction {
9
10     protected Implementor impl;
11     // 设置颜色
12     public void setImpl(Implementor implementor) {
13         impl = implementor;
14     }
15
16     // 抽象操作方法
17     // 绘图
18     public abstract void operation();
19 }
```

颜色接口

```
view > qiaojiemoshi > Implementor
× DataParser.java × DataProviderImp.java × TXTDataParser.java
package com.ljy.ofoview.qiaojiemoshi;


- 1 /**
- 2 * 颜色
- 3 * @Author Liang Jingyan
- 4 * 20-5-27
- 5 */
- 6 public interface Implementor {
- 7
- 8     void operationImpl();
- 9 }

```

画笔实现类

VIEW / qiaojiemoshi / BehindAbstraction

DataParser.javaDataProviderImpl.javaTXTDataParser.javaMysqlDataProvider

```
package com.ljy.ofoview.qiaojiemoshi;

/**
 * @Author Liang Jingyan
 * 20-5-27
 */
public class DefindAbstraction extends Abstraction {

    @Override
    public void operation() {
        impl.operationImpl();
    }
}
```

颜色实现类

ofoview / qiaojiemoshi / ConcreteImplementorA

ConcreteImplementorA.javaDataProviderImpl.javaTXTDataParser.javaMysqlDataProvider.javaImpl

```
package com.ljy.ofoview.qiaojiemoshi;

import android.util.Log;

/**
 * @Author Liang Jingyan
 * 20-5-27
 */
public class ConcreteImplementorA implements Implementor {

    @Override
    public void operationImpl() {
        Log.d(tag: "Lxima", msg: "operationImpl: this is ConcreteImplementorA");
    }
}
```

使用：

```
ClientActivity.java × ConcreteImplementorA.java × DataParser.java × DataProviderImp.java × TXTDataParser.java ×
6 * 桥接模式
7 * @Author Liang Jingyan
8 * 20-5-27
9 */
0 public class ClientActivity extends AppCompatActivity {
1
2     @Override
3     protected void onCreate(@Nullable Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5
6         Abstraction abstraction;
7         Implementor implementor;
8
9         abstraction = new DefindAbstraction();
0
1         implementor = new ConcreteImplementorA();
2         abstraction.setImpl(implementor);
3         abstraction.operation();
4
5         implementor = new ConcreteImplementorB();
6         abstraction.setImpl(implementor);
7         abstraction.operation();
8
9     }
0 }
```

策略模式

```
// 使用策略模式前
public void Share{
    public void shareOptions(String option){
        if(option.equals("微博")){
            //function1();
            //...
        }else if(option.equals("微信")){
            //function2();
            //...
        }else if(option.equals("朋友圈")){
            //function3();
            //...
        }else if(option.equals("QQ")){
            //function4();
            //...
        }
        //...
    }
}
```

```
//使用策略模式后
```

```
//定义策略接口
public interface DealStrategy{
    void dealMythod(String option);
}

//定义具体的策略1
public class DealSina implements DealStrategy{
    @override
    public void dealMythod(String option){
        //...
    }
}

//定义具体的策略2
public class DealWeChat implements DealStrategy{
    @override
    public void dealMythod(String option){
        //...
    }
}

//定义上下文，负责使用DealStrategy角色
public static class DealContext{

    private String type;
    private DealStrategy deal;

    public DealContext(String type,DealStrategy deal){
        this.type = type;
        this.deal = deal;
    }

    public getDeal(){
        return deal;
    }
    public boolean options(String type){
        return this.type.equals(type);
    }
}

public void Share{
    private static List<DealContext> algs = new ArrayList();
    //静态代码块,先加载所有的策略
    static {
        algs.add(new DealContext("Sina",new DealSina()));
        algs.add(new DealContext("WeChat",new DealWeChat()));
    }
    public void shareOptions(String type){
        DealStrategy dealStrategy = null;
        for (DealContext deal : algs) {
```

```
        if (deal.options(type)) {
            dealStrategy = deal.getDeal();
            break;
        }
    }
    dealStrategy.dealMythod(type);
}}
```

代理模式

动态代理

动态代理就是在实现阶段不用关心代理谁, 而在运行阶段才指定代理哪一个对象.

Retrofit

优点:

1. 可以配置不同HTTP client 来实现网络请求, 如okhttp、 httpclient等
2. 请求的方法参数注解可以定制
3. 支持同步、异步和RxJava
4. 超级解耦
5. 可以配置不同的反序列化工具来解析数据, 如json、 xml等
6. 使用方便灵活
7. 采用大量设计模式简化使用
8. 性能最好, 处理快

缺点:

1. 扩展性差
2. 高度封装使解析数据都是使用统一的converter, 如果服务器不能给出统一的APO的形式, 将很难进行处理。
3. 使用方法较多, 原理复杂, 存在一定的门槛

使用的[设计模式](#):

1. Retrofit构建过程
 - 建造者模式、工厂方法模式
2. 创建网络请求接口实例过程
 - 外观模式、代理模式、单例模式、策略模式、装饰模式
3. 生成执行请求过程
 - 适配器模式

Context

https://blog.csdn.net/guolin_blog/article/details/47028975

Activity和Service以及Application的Context是不一样的,Activity继承自ContextThemeWraper.其他的继承自ContextWrapper.

Context数量 = Activity数量 + Service数量 + 1

上面的1代表着Application的数量，因为一个应用程序中可以有多个Activity和多个Service，但是只能有一个Application。

获取Application实例

`getApplicationContext()`

`getApplication()`

两个方法获取到的对象相同，因为Application本身也是一个Context，所以它们是统一对象。

区别在于作用域上，`getApplication()`方法的语义性非常强，一看就知道是用来获取Application实例的，但是这个方法只有在Activity和Service中才能调用的到。

如果在一些其它的场景，比如BroadcastReceiver中也想获得Application的实例，这时就可以借助
`getApplicationContext()`方法

`getBaseContext()` 方法得到的是一个 `ContextImpl` 对象

Application全局只有一个，它本身就是单例了，无需再用单例模式去为它做多重实例保护了

OkHttp

常用状态码：

- 100~199：指示信息，表示请求已接收，继续处理
- 200~299：请求成功，表示请求已被成功接收、理解
- 300~399：重定向，要完成请求必须进行更进一步的操作
- 400~499：客户端错误，请求有语法错误或请求无法实现
- 500~599：服务器端错误，服务器未能实现合法的请求

[OPPO团队 OkHttp文章》》》](#)

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder().url("www.xxx.com").build();
//execute()是同步方法，enqueue()是异步方法
Response response = client.newCall(request).execute();

//实现2
//异步方法
client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        e.printStackTrace();
    }
})
```

```
@Override  
public void onResponse(Call call, Response response) throws IOException {  
    ...  
}
```

OkHttpClient: 这个是整个OkHttp的核心管理类，所有的内部逻辑和对象归OkHttpClient统一来管理，它通过Builder构造器生成，构造参数和类成员很多，这里先不做具体的分析。

Request 和Response: Request 是我们发送请求封装类，内部有url, header , method, body等常见的参数，Response是请求的结果，包含code, message, header, body；这两个类的定义是完全符合Http协议所定义的请求内容和响应内容。

RealCall：负责请求的调度（同步的话走当前线程发送请求，异步的话则使用OkHttp内部的线程池进行），同时负责构造内部逻辑责任链，并执行责任链相关的逻辑，直到获取结果。虽然OkHttpClient是整个OkHttp的核心管理类，但是真正发出请求并且组织逻辑的是RealCall类，它同时肩负了调度和责任链组织的两大重任，接下来我们来着重分析下RealCall类的逻辑。

RealCall类并不复杂，有两个最重要的方法，execute() 和 enqueue()，一个是处理同步请求，一个是处理异步请求。跟进enqueue的源码后发现，它只是通过异步线程和callback做了一个异步调用的封装，最终逻辑还是会调用到execute()这个方法，然后调用了getResponseBodyWithInterceptorChain()获得请求结果

拦截器

- RetryAndFollowUpInterceptor——失败和重定向拦截器
 - BridgeInterceptor——封装request和response拦截器
 - CacheInterceptor——缓存相关的过滤器，负责读取缓存直接返回、更新缓存
 - ConnectInterceptor——连接服务，负责和服务器建立连接 这里才是真正的请求网络
 - CallServerInterceptor——执行流操作(写出请求体、获得响应数据) 负责向服务器发送请求数据、从服务器读取响应数据 进行http请求报文的封装与请求报文的解析
-
- addInterceptor与addNetworkInterceptor有什么区别？

应用拦截器 (addInterceptor) 是最先执行的拦截器，也就是用户自己设置request属性后的原始请求，一个请求只调用一次

而网络拦截器位于ConnectInterceptor和CallServerInterceptor之间，此时网络链路已经准备好，只等待发送请求数据。如果一个请求在 RetryAndFollowUpInterceptor 这个拦截器内部重试或者重定向了N次，那么其内部嵌套的所有拦截器也会被调用N次。

从使用场景看，应用拦截器因为只会调用一次，通常用于统计客户端的网络请求发起情况；而网络拦截器一次调用代表了一定会发起一次网络通信，因此通常可用于统计网络链路上传输的数据。

Connection的连接复用

<https://juejin.im/post/5e7b27a0e51d4567eb6bc8a9#comment>

- 网络缓存如何实现的？

内部使用LUR 缓存算法

OkHttp 默认只支持GET 请求的缓存

线程池

<https://juejin.im/post/5da050c5f265da5b9e0d429c#heading-3>

- **降低资源消耗**: 通过重用已存在的线程，可以有效的降低，由频繁创建线程带来的性能消耗。
- **增加系统的稳定性**: 由线程池统一管理，除了可以降低资源的消耗，还可以有效控制线程的并发数量，不会让线程无限制的创建，增加了系统的稳定性。
- **提高响应速度**: 因为重用线程，所以提高了响应速度。

```
// 获得当前CPU的核心数
private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();

// 设置线程池的核心线程数2-4之间,但是取决于CPU核数
private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1, 4));

// 创建线程池
ExecutorService executorService = Executors.newFixedThreadPool(CORE_POOL_SIZE);
```

作者: 刘洋巴金

链接:<https://juejin.im/post/5eb7b6a1f265da7c0750a9a8>

来源:掘金

线程池生命周期管理

线程池的运行状态，并不是用户显示设置的，而是伴随着线程池的运行，由内部来维护的。

ThreadPoolExecutor

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}
```

- corePoolSize

线程池核心线程数，**默认情况下，核心线程会在线程池中一直存活，即使它们处于闲置状态**，如果将 `ThreadPoolExecutor` 的 `allowCoreThreadTimeOut (true)` 属性设置为true，那么限制的核心线程在等待新任务到来时会有超时策略，这个时间间隔由 `keepAliveTime` 所指定，当等待时间超过 `keepAliveTime` 所指定的时长后，核心线程就会被终止。如果当前的线程总个数 < `corePoolSize`，那么新建的线程为核心线程，如果当前线程总个数 >= `corePoolSize`，那么新建的线程为非核心线程。

- `maximumPoolSize`

核心线程数 + 非核心线程数 = 最大线程数量

线程池所能容纳的最大线程数，当活动线程达到这个数值后，后续的新任务将会被阻塞。

- `keepAliveTime`

非核心线程限制时的超时时长，超过这个时长，非核心线程就会被回收。当`ThreadPoolExecutor` 的 `allowCoreThreadTimeOut` 属性设置为true 时，`keepAliveTime`同样会作用于核心线程。

- `unit`

用于指定`keepAliveTime` 参数的时间单位，这是一个枚举，常用的有`TimeUnit.MILLISECONDS`(毫秒)

- `workQueue`

线程池中任务队列，通过线程池的`execute` 方法提交的`Runnable` 对象会存储在这个参数中。

- `threadFactory`

线程工厂，为线程池提供创建新线程的功能。`ThreadFatory`是一个接口，它只有一个方法，`Thread newThread (Runnable r)`

BlockingQueue(缓存队列)

当有任务到来时，会指派给核心线程去执行，等核心线程都被占用了，那么再有新的任务，就会加入到队列中，等队列满了，再有任务，就再启动非核心线程去执行。常用的队列如下

- **SynchronousQueue** 使用这个队列时，当有任务到来的时候，它并不存任务，而是直接将任务丢给线程去执行，如果线程都在被占用，它就会创建线程去处理这个任务，所以一般使用这个缓存队列的时候，`maximumPoolSize`（线程池能容纳的最大线程数量）设置到 `Integer.MAX_VALUE`，不然任务数超过 `maximumPoolSize`限制而创建不了线程。
- **LinkedBlockingQueue** 使用这个队列是，当有任务到来的时候，如果当前的核心线程数 < `corePoolSize`，它会新建核心线程去执行任务，如果当前核心线程数 >= `corePoolSize`时，它会将还未被执行的任务存储起来，等待执行，但是这个队列，**没有存储上限**，所以尼，这也就造成了，`maximumPoolSize`（总线程数），永远不会超过`corePoolSize`。此队列按 FIFO（先进先出）原则对任务进行操作。
- **ArrayBlockingQueue** 使用这个队列，可以设置队列的长度，那么当任务到来的时候，核心线程数 < `corePoolSize`时，则创建核心线程去执行任务，如果核心线程数 >= `corePoolSize`时，加入到队列里面，等待执行，如果队列也满了，则新建非核心线程去执行任务。此队列按 FIFO（先进先出）原则对元素进行操作。但是线程数不能超过总线程数。
- **DelayQueue (延时队列)** 任务到来时，首先先加入到队列中，只有达到了指定的延时时间，才会执行任务。

RejectedExecutionHandler (拒绝策略) :

当任务过多时，即：当前线程数已经达到了最大线程数，缓冲队列也已经满了，或者线程池关闭了，那么再来的任务请求，我们会拒绝，怎么拒绝尼？有以下几个方案：

- **AbortPolicy** (默认策略)

“当你有了，原配，想再娶个小三时，原配的态度”，默认策略，简单粗暴，直接拒绝抛异常 (RejectedExecutionException)

- **DiscardPolicy**

“原配的太粗暴，没法子，只能把小三的电话，微信都删掉了，再也不来往了”，DiscardPolicy策略就是，直接丢弃，但是不抛异常。如果线程队列已满，则后续提交的任务都会被丢弃。

- **DiscardOldestPolicy**

“但是原配看久了，很腻了，算了，人生短短几十年，何必委屈自己，休妻，腾地方，娶小三”，DiscardOldestPolicy策略就是，直接丢弃掉队伍最前面的任务，再重新提交后面新来的任务。

- **CallerRunsPolicy**

“唉，原配虽然不好看，但是家里有背景，不敢随意抛弃，后面的小三还是哪来的回哪去吧，找个熟人照顾着”，CallerRunsPolicy策略就是不抛弃任务，由调用者运行这个任务，比如主线程启动了线程池去运行这个任务，现在线程池满了，那么这个任务就由主线程进行调用执行了。

总结一下 让任务到来的时候，会执行以下的流程

1. 线程数量未达到 `corePoolSize`，则新建一个线程（核心线程）执行任务。
2. 线程数量达到了 `corePoolsSize`，则将任务移入队列等待。
3. 队列已满，新建非核心线程（先进先出）执行任务。
4. 队列已满，总线程数又达到了 `maximumPoolSize`，就会由 `RejectedExecutionHandler` 抛出异常。

线程池分类

1. **FixedThreadPool**

这是一种线程固定的线程池，当线程处于空闲状态时，他们并不会被回收，除非线程池被关闭。当所有的线程都处于活动状态时，新任务都会处于等待状态，直到有线程空闲出来。由于 `FixedThreadPool` 只有核心线程，并且这些线程不会被回收，这意味着它能够更加快速响应外界的请求。

FixewFixedThreadPool线程池的核心线程数是固定的，它使用了近乎于无界的`LinkedBlockingQueue`阻塞队列。当核心线程用完后，任务会入队到阻塞队列，如果任务执行的时间比较长，没有释放，会导致越来越多的任务堆积到阻塞队列，最后导致机器的内存使用不停的飙升，造成JVM OOM。<https://juejin.im/post/5d6a57ee51d4561e721df30>

2. **CachedThreadPool**

这是一种线程数量不固定的线程池，他只有非核心线程，并且其最大线程数为 `Integer.MAX_VALUE`。由于`Integer.MAX_VALUE`是一个很大的数，实际上就相当于最大线程数可以任意大，当线程池都处于活动状态时，线程池会创建新的线程来处理新的任务，否则就会利用空闲的线程来处理新的任务。线程池中的空闲线程都有超时机制，这个超时时长为60秒，超过60秒闲置线程就会被回收，和 `FixedThreadPool` 不同的是， `CahcedThreadPool` 的任务队列相当于一个空集合，这将导致任何任务都会立即被执行，因为在这种场景下，`Synchronous-Queue`是无法插入任务的，`SynchronousQueue` 是一个非常特殊的队列在很多情况下可以把它简单理解为一个无法存储元素的队列，比较适合执行大量的耗时较少的任务，当整个线程池都处于闲置状态时，线程池中的线程都会超时而被停止，这个时候`CachedThreadPool`之中是没有任何线程的，它几乎是不占用任何系统资源的。在实际中使用较少。

3. **ScheduledThreadPool** 它的核心线程是固定的，而非核心线程数是没有限制的，并且当非核心线程闲置时会被立即回收，这类线程池主要用于执行定时任务和具有固定周期的重复任务。

4. **SingleThreadExecutor**

这类线程池内部只有一个核心线程，它确保所有的任务都在同一个线程中按顺序执行，SingleThreadExecutor的意义在于统一所有的外界任务到一个线程中，这使得这些任务之间不需要处理线程同步问题。

合理的线程池数量计算

```
/**  
 * 获得当前CPU的核心数  
 */  
private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();  
  
/**  
 * 设置线程池的核心线程数2-4之间，但是取决于CPU核数  
 */  
private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1, 4));  
  
/**  
 * 创建线程池  
 */  
ExecutorService mExecutor = Executors.newFixedThreadPool(CORE_POOL_SIZE);
```

RxJava 解析

RxJava是一个使用了观察者模式，能够异步的库。

- 背压策略的前提是异步环境，被观察者和观察者处于不同的线程环境当中
- 在观察者模式中，**被观察者是主动的推送**数据给观察者，观察者是被动接收的
- **响应式拉取**则是反过来的，观察者主动从被观察者那里去拉取数据，而被观察者变成被动的等待通知再发送数据

JsonchaoRxJava源码分析

使用装饰者模式

newThread

该方法返回一个默认、共享的调度器实例，该实例为每个工作单元创建一个新线程，默认实现是创建一个新的**单线程**，要注意的是，每次调用**Scheduler.scheduleDirect**方法（及其重载方法）和**Scheduler.createWorker**方法都可以创建数目无限制的线程，从而造成**内存溢出（OOM）**。

single

该方法会放回一个默认、共享的调度器实例，该实例会创建一个单独的线程

Scheduler类型

Scheduler类型	使用方式	含义	使用场景
IoScheduler	Scheduler.io()	io操作线程	读写SD卡类型,查询数据库,访问网络等io密集型操作
NewThreadScheduler	Schedulers.newThread()	.createNew线程	耗时操作等
SingleScheduler	Schedulers.single()	单例线程	只需一个单例线程时
ComputationScheduler	Schedulers.computation()	CPU计算操作线程	图片压缩取样、xml,json解析等CPU密集型计算
TrampolineScheduler	Schedulers.trampoline()	当前线程	需要在当前线程立即执行任务时
HandlerScheduler	AndroidSchedulers.mainThread()	Android主线程	更新UI等

为什么多次subscribeOn 指定上游线程,为什么只有第一次有效?

因为上游Observable 只有一个任务, 就是subscribe(准确的来说是subscribeActual()) , 而subscribeOn 要做的事情就是把上游任务切换到一个指定线程里, 那么一旦被切换到了某个指定的线程里, 后面的切换不就是没有意义了吗。

Glide 解析

Picasso与Glide相比:

- 相似——api上的调用差不多, 都支持内存缓存, glide是picasso的升级, 性能有一定的提升
- 差异
 - 缓存不同, picasso2级缓存, **没有磁盘缓存**, Glide三级缓存, 内存-磁盘-网络。
 - Picasso默认加载的图片格式是ARGB-8888, Glide默认是RGB-565(**4.x之后改为默认8888**), 内存开销节省小一半
 - Picasso是加载全尺寸图片到内存中, 下次在任何imageView中加载图片时, 先取全尺寸图片, 重新调整大小, 再存缓存。而Glide是按imageView的大小缓存的, 为每种大小缓存一次。 **Glide这种方式是空间换时间**, 大小几乎是picasso的2倍
 - 生命周期问题, Glide的with方法图片加载会和Activity与Fragment的生命周期一致
 - Glide可以加载GIF
 - Picasso包特别小, 100k左右, glide大概500k

Fresco

- 优点：

- 图片存储在安卓系统匿名共享内存区，而不是虚拟机的堆内存，图片的中间缓冲数据也存放在本地堆内存，所以应用有更多的内存空间使用。不容易OOM，减少GC回收
- 大大减少OOM（在更底层的Native层对OOM进行处理，图片将不再占用App的内存）
- 渐进式加载，支持图片从模糊到清晰，用户体验极好
- 图片可以以任意为中心点显示在ImageView
- JPEG图片改变大小也是在native进行，不在虚拟机堆内存，同样减少OOM
- 对GIF支持友好
- 适用于需要高性能加载大量图片的场景
- 最大的优势在于5.0以下(最低2.3)的bitmap加载。在5.0以下系统，Fresco将图片放到一个特别的内存区域(Ashmemp区)

- 缺点

- 包太大，2-3M
- 用法复杂
- 底层都是C/C++，源码阅读有挑战性

为了防止加载大量图片导致OOM，Fresco会更适合一点，因为LruCache初始化是需要指定一个内存大小的，在大量图片的需求下，可能会指定大一点的内存，保证效率，LruCache内存越大，发生OOM的几率就越高，特别是在老设备上，内存本来就小。

小结

Glide能做Picasso能做的所有事情，更快，能处理大型图片流，一般Glide是首选；Fresco当使用图片非常多的应用时，它的优势非常明显

with 可以传入多种类型，可以传入Activity，Fragment或者是Context。传入非Application参数的时候，会向当前的Activity添加一个隐藏的Fragment，因为Fragment的生命周期和Activity是同步的，如果Activity被销毁了，Fragment是可以监听到的，这样Glide就可以捕获这个事件并停止图片加载了。

内存缓存

```
@Nullable
private EngineResource<?> loadFromMemory(
    EngineKey key, boolean isMemoryCacheable, long startTime) {
    if (!isMemoryCacheable) {
        return null;
    }
    //从弱引用中获取缓存
    EngineResource<?> active = loadFromActiveResources(key);
    if (active != null) {
        if (VERBOSE_IS_LOGGABLE) {
            logWithTimeAndKey("Loaded resource from active resources", startTime, key);
        }
        return active;
    }
}
```

```

}
//从Lrucache中获取缓存
EngineResource<?> cached = loadFromCache(key);
if (cached != null) {
    if (VERBOSE_IS_LOGGABLE) {
        logWithTimeAndKey("Loaded resource from cache", startTime, key);
    }
    return cached;
}

return null;
}

```

先从弱引用中获取缓存，如果获取不到，就到Lrucache 获取缓存，使用activeResources来缓存正在使用中的图片，可以保护这些图片不会被**LruCache**算法回收掉。

使用一个弱引用map activeResources 来放项目中正在使用的资源，Lrucache中不含有正在使用的资源，资源内部有个计数器来显示自己是不是还有被引用的情况，把正在使用的资源和没有被使用的资源分开。

弱引用缓存则减轻了 lru缓存的压力,避免lru过满导致频繁gc,并且提高查找效率---如何减轻lru缓存压力?

engine是一个负责加载、管理活跃和缓存资源的引擎类

engineJob (decodejob的回调类，管理下载过程以及状态)

Glide会使用LruCache来对解析后的url来进行缓存，以便后续可以省去解析url的时间

核心思想：

使用一个弱引用map activeResources来盛放项目中正在使用的资源。Lrucache中不含有正在使用的资源。资源内部有个计数器来显示自己是不是还有被引用的情况，把正在使用的资源和没有被使用的资源分开有什么好处呢？因为当Lrucache需要移除一个缓存时，会调用resource.recycle()方法。注意到该方法上面注释写着只有没有任何consumer引用该资源的时候才可以调用这个方法。那么为什么调用resource.recycle()方法需要保证该资源没有任何consumer引用呢？glide中resource定义的recycle () 要做的事情是把这个不用的资源（假设是bitmap或drawable）放到bitmapPool中。bitmapPool是一个bitmap回收再利用的库，在做transform的时候会从这个bitmapPool中拿一个bitmap进行再利用。这样就避免了重新创建bitmap，减少了内存的开支。而既然bitmapPool中的bitmap会被重复利用，那么肯定要保证回收该资源的时候（即调用资源的recycle () 时），要保证该资源真的没有外界引用了。这也是为什么glide花费那么多逻辑来保证Lrucache中的资源没有外界引用的原因。

作者：jsonchao

链接：<https://juejin.im/post/5e65ad276fb9a07cc01a3264>

硬盘缓存

硬盘缓存的实现也是使用的**LruCache**算法，而且Google还提供了一个现成的工具类DiskLruCache。

我们通过 diskCacheStrategy() 方法来设置缓存策略, 可以接收下面四种 参数:

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.SOURCE： 表示只缓存原始图片。

- DiskCacheStrategy.RESULT：表示只缓存转换过后的图片（默认选项）。
- DiskCacheStrategy.ALL：表示既缓存原始图片，也缓存转换过后的图片。

glide缓存在哪写入？

```

@Synthetic
void handleResultOnMainThread() {
    stateVerifier.throwIfRecycled();
    if (isCancelled) {
        resource.recycle();
        release(isRemovedFromQueue: false /*isRemovedFromQueue*/);
        return;
    } else if (cbs.isEmpty()) {
        throw new IllegalStateException("Received a resource without any callbacks to notify");
    } else if (hasResource) {
        throw new IllegalStateException("Already have resource");
    }
    engineResource = engineResourceFactory.build(resource, isCacheable); ← 构建对象
    hasResource = true;

    // Hold on to resource for duration of request so we don't recycle it in the middle of
    // notifying if it synchronously released by one of the callbacks.
    engineResource.acquire();
    listener.onEngineJobComplete(engineJob: this, key, engineResource); ← 回调

    //noinspection ForLoopReplaceableByForEach to improve perf
    for (int i = 0, size = cbs.size(); i < size; i++) {
        ResourceCallback cb = cbs.get(i);
        if (!isInIgnoredCallbacks(cb)) {
            engineResource.acquire();
            cb.onResourceReady(engineResource, dataSource);
        }
    }
}

```

Glide生成缓存key问题

[高级技巧](#)

Glide预加载

使用[Preload\(\)方法](#)， preload() 方法有两个重载的方法，不带参数的表示直接在家原始图片大小，另一个可以通过参数加载指定图片的宽高。

使用：

```
//先提前预加载
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .preload();

//需要加载的时候再加载出来
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .into(imageView);

//需要保证后面加载的 是预加载好的图片， 因为不同尺寸缓存会不同
```

downloadOnly()方法

Glide 4.0 使用 `submit()` 方法， `submit()` 方法必须运行在子线程中。

`downloadOnly()` 方法表示只会下载图片，而不会对图片进行加载。

它有两个方法重载，一个接收图片的宽度和高度，另一个接收一个泛型对象，如下所示：

- `downloadOnly(int width, int height)`
- `downloadOnly(Y target)`

其中`downloadOnly(int width, int height)`是用于在子线程中下载图片的，而`downloadOnly(Y target)`是用于在主线程中下载图片的。

其实`downloadOnly(int width, int height)`方法必须使用在子线程当中，最主要还是因为它在内部帮我们自动创建了一个`RequestFutureTarget`，是这个`RequestFutureTarget`要求必须在子线程当中执行。而`downloadOnly(Y target)`方法则要求我们传入一个自己创建的Target，因此就不受`RequestFutureTarget`的限制了。

自定义变换图片

继承`BitmapTransformation` 类即可，不过这个类只能对静态图片进行变换，如果需要对GIF图进行变换需要实现`Trasform` 接口。

继承`BitmapTransformation` 类之后，只要在`Transform()` 方法中实现具体变换就行了。

```
public class CircleCrop extends BitmapTransformation {

    public CircleCrop(Context context) {
        super(context);
    }

    public CircleCrop(BitmapPool bitmapPool) {
        super(bitmapPool);
    }

    @Override
```

```

public String getId() {
    return "com.example.glide.test.CircleCrop";
}

@Override
protected Bitmap transform(BitmapPool pool, Bitmap toTransform, int outWidth, int
outHeight) {
    int diameter = Math.min(toTransform.getWidth(), toTransform.getHeight());

    final Bitmap toReuse = pool.get(outWidth, outHeight, Bitmap.Config.ARGB_8888);
    final Bitmap result;
    if (toReuse != null) {
        result = toReuse;
    } else {
        result = Bitmap.createBitmap(diameter, diameter, Bitmap.Config.ARGB_8888);
    }

    int dx = (toTransform.getWidth() - diameter) / 2;
    int dy = (toTransform.getHeight() - diameter) / 2;
    Canvas canvas = new Canvas(result);
    Paint paint = new Paint();
    BitmapShader shader = new BitmapShader(toTransform, BitmapShader.TileMode.CLAMP,
                                            BitmapShader.TileMode.CLAMP);
    if (dx != 0 || dy != 0) {
        Matrix matrix = new Matrix();
        matrix.setTranslate(-dx, -dy);
        shader.setLocalMatrix(matrix);
    }
    paint.setShader(shader);
    paint.setAntiAlias(true);
    float radius = diameter / 2f;
    canvas.drawCircle(radius, radius, radius, paint);

    if (toReuse != null && !pool.put(toReuse)) {
        toReuse.recycle();
    }
    return result;
}
}

```

更多的变换效果也可以使用 `glide-transformations` 这个库。

Glide自定义组件

https://blog.csdn.net/guolin_blog/article/details/78179422

原理：

```

@NotNull
Glide build(@NotNull Context context) {
    if (sourceExecutor == null) {

```

```
        sourceExecutor = GlideExecutor.newSourceExecutor();
    }

    if (diskCacheExecutor == null) {
        diskCacheExecutor = GlideExecutor.newDiskCacheExecutor();
    }

    if (animationExecutor == null) {
        animationExecutor = GlideExecutor.newAnimationExecutor();
    }

    if (memorySizeCalculator == null) {
        memorySizeCalculator = new MemorySizeCalculator.Builder(context).build();
    }

    if (connectivityMonitorFactory == null) {
        connectivityMonitorFactory = new DefaultConnectivityMonitorFactory();
    }

    if (bitmapPool == null) {
        int size = memorySizeCalculator.getBitmapPoolSize();
        if (size > 0) {
            bitmapPool = new LruBitmapPool(size);
        } else {
            bitmapPool = new BitmapPoolAdapter();
        }
    }

    if (arrayPool == null) {
        arrayPool = new LruArrayPool(memorySizeCalculator.getArrayPoolSizeInBytes());
    }

    if (memoryCache == null) {
        memoryCache = new LruResourceCache(memorySizeCalculator.getMemoryCacheSize());
    }

    if (diskCacheFactory == null) {
        diskCacheFactory = new InternalCacheDiskCacheFactory(context);
    }

    if (engine == null) {
        engine =
            new Engine(
                memoryCache,
                diskCacheFactory,
                diskCacheExecutor,
                sourceExecutor,
                GlideExecutor.newUnlimitedSourceExecutor(),
                animationExecutor,
                isActiveResourceRetentionAllowed);
    }

    if (defaultRequestListeners == null) {
```

```
    defaultRequestListeners = Collections.emptyList();
} else {
    defaultRequestListeners = Collections.unmodifiableList(defaultRequestListeners);
}

...
}

/**
 * Glide创建的时候会进行空检查，只有在为空的时候才会创建对象，如果我们在检查之前创建好对象，这样Glide就不
 * 重新创建它们的实例了，从而实现更改Glide组件的目的。
*/

```

可用配置项:

- `setMemoryCache()`
用于配置Glide的内存缓存策略，默认配置是LruResourceCache。
- `setBitmapPool()`
用于配置Glide的Bitmap缓存池，默认配置是LruBitmapPool。
- `setDiskCache()`
用于配置Glide的硬盘缓存策略，默认配置是InternalCacheDiskCacheFactory。
- `setDiskCacheService()`
用于配置Glide读取缓存中图片的异步执行器，默认配置是FifoPriorityThreadPoolExecutor，也就是先入先出原则。
- `setResizeService()`
- 用于配置Glide读取非缓存中图片的异步执行器，默认配置也是FifoPriorityThreadPoolExecutor。
- `setDecodeFormat()`
用于配置Glide加载图片的解码模式，默认配置是RGB_565

然后需要创建一个自定义的类

```
/**
 * @Author Liang Jingyan
 * 19-12-17
 * Glide 4.0之后自定义模块需要继承的类
 * Glide 4必须加@GlideModule注解，不然识别不到这个自定义模块
 */

@GlideModule
public class MyAppGlideModule extends AppGlideModule {

    @Override
    public void applyOptions(@NonNull Context context, @NonNull GlideBuilder builder) {
        //修改Glide 配置
    }
}
```

```
        builder.setDiskCache(new ExternalCacheDiskCacheFactory(context));

    }

    @Override
    public void registerComponents(@NonNull Context context, @NonNull Glide glide, @NonNull
Registry registry) {
    super.registerComponents(context, glide, registry);
    //修改Glide 组件
    registry.register(GlideUrl.class, InputStream.class, new
OkHttpGlideUrlLoader.Factory());

}
}
```

Glide 问题

Glide 如何防止ImageView 内存泄漏的?

Glide 通过生命周期监听, 在创建的时候会绑定一个Fragment, 用来监听生命周期, 这样当Activity/Fragment 销毁的时候, 就取消图片的加载, 防止内存泄漏.

如何防止线程池任务过多?

当列表滑动的时候, 会有很多图片进行请求, 如果是第一次进入, 没有缓存, 队列中会有很多任务在等待, 在网络请求图片之前, 我们可以判断队列中是否存在该任务, 存在则不加入到队列中.

为什么大量图片的场景不推荐使用Glide?

LruCache初始化是需要指定一个内存大小的, 在大量图片的需求下, 可能指定大一点的内存, 保证效率, LruCache内存越大, 发生OOM的几率就越高, 特别是在一些老设备上

Glide 4.x版本已经有注册监听内存的使用情况, 如果是内存极度紧张的情况下, Glide会立即清空内存缓存的, 从而释放内存压力, 所以OOM的可能性大大降低了; 不过有一种情况是比较容易触发OOM, 就是图片控件大量使用了wrap_content, 这样子Glide拿不到具体的尺寸, 就会默认为手机屏幕尺寸来对图片进行裁剪压缩, 这样子OOM的可能性就大增了

EventBus 分析

源码分析: <https://blog.csdn.net/u014702653/article/details/100087264>

使用简单, 可以指定接收事件的函数运行的线程, 解耦, 传递大量数据的Event时不会崩溃

为什么EventBus传递大数据的时候不会崩溃

EventBus如何接收指定事件方法运行的线程

EventBus 通过注解的方式指定接收事件的运行线程，在注册的时候将注解信息保存起来，当post事件的时候拿到注册时保存的注解信息（运行线程），通过不同的 ThreadMode 做不同的处理，如开启工作线程执行接收事件的方法，或者通过handler方式将事件运行在主线程中，以此达到指定线程运行方法的目的

一个类如果使用了多个方法对同一个 Event 对象进行注册，那么当该 Event 对象被发射出来的时候，所有的方法都将会得到回调

当子类「显示」实现父类的订阅方法的时候，如果此时发射指定 Event 的话，父类的订阅方法将不会执行，而仅会执行子类的订阅方法。

threadMode() = 函数运行的线程

POSTING

与发送事件同一个线程, 如果post在主线程, **避免耗时操作**

MAIN

主线程

如果post在主线程,直接在主线程中运行, 会阻塞发布线程

如果post在子线程,则事件排队等待传递,使用handler回调到主线程中

MAIN_ORDERED

主线程- 排队

在 Android 中则接收事件方法会被扔进 MessageQueue 中等待执行

意味着发射事件方法是不会阻塞的

BACKGROUND

后台线程

如果post在子线程,则在post所在的线程执行

如果post是主线程, 则使用单个后台线程按顺序执行,由于该子线程是 EventBus 维护的单一子线程, 所以为了解决影响到其他接收事件方法的执行, **该方法不应太耗时**避免该子线程阻塞。

ASYNC

异步线程

总是使用异步线程执行, 通过线程池管理异步线程,不会阻塞发布线程和主线程

源码分析

```

//注册
public void register(Object subscriber) {
    Class<?> subscriberClass = subscriber.getClass();
    //获取当前类中使用@Subscribe注解的方法
    List<SubscriberMethod> subscriberMethods =
    subscriberMethodFinder.findSubscriberMethods(subscriberClass);
    synchronized (this) {
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            //遍历方法
            subscribe(subscriber, subscriberMethod);
        }
    }
}

```

```

//subscribe 方法
private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
    //-----注解方法中的参数类型作为key-----
    Class<?> eventType = subscriberMethod.eventType;
    Subscription newSubscription = new Subscription(subscriber, subscriberMethod);
    CopyOnWriteArrayList<Subscription> subscriptions =
    subscriptionsByEventType.get(eventType);
    if (subscriptions == null) {
        subscriptions = new CopyOnWriteArrayList<>();
        subscriptionsByEventType.put(eventType, subscriptions);
    } else {
        if (subscriptions.contains(newSubscription)) {
            throw new EventBusException("Subscriber " + subscriber.getClass() + " "
already registered to event "
+ eventType);
        }
    }

    int size = subscriptions.size();
    for (int i = 0; i <= size; i++) {
        if (i == size || subscriberMethod.priority >
subscriptions.get(i).subscriberMethod.priority) {
            subscriptions.add(i, newSubscription);
            break;
        }
    }
    //-----当前注册对象作为key-----

    /**
     * 根据当前注册对象作为key保存本类中所有注解方法的事件类型
     * 用于查询是否已经注册了当前对象
     * isRegistered(Object subscriber){ return typesBySubscriber.containsKey(subscriber);
     */
    * 解绑注册 unregister(Object subscriber) 的时候可以找到所有的事件类型，(结合
* subscriptionsByEventType )
    * 然后再根据事件找到 已经注册对象的列表,移除当前注册的对象
    */
}

```

```

List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
if (subscribedEvents == null) {
    subscribedEvents = new ArrayList<>();
    typesBySubscriber.put(subscriber, subscribedEvents);
}
subscribedEvents.add(eventType);

//-----
//判断是否是粘性事件
//粘性事件单独放在一个map里面,只要不清除,它就一直在,所以可以先发粘性事件,然后注册接收
if (subscriberMethod.sticky) {
    if (eventInheritance) {
        Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();
        for (Map.Entry<Class<?>, Object> entry : entries) {
            Class<?> candidateEventType = entry.getKey();
            if (eventType.isAssignableFrom(candidateEventType)) {
                Object stickyEvent = entry.getValue();
                checkPostStickyEventToSubscription(newSubscription, stickyEvent);
            }
        }
    } else {
        Object stickyEvent = stickyEvents.get(eventType);
        checkPostStickyEventToSubscription(newSubscription, stickyEvent);
    }
}
}
}

```

register方法里面,首先获取当前订阅对象的class,通过这个class拿到订阅的事件集合,然后遍历进行订阅,在订阅类subscribe中,首先拿到事件的类型,然后判断是否存在该对象类型,如果已经存在,判断是否重复订阅.最后根据优先级插入到队列中.

post方法

```

public void post(Object event) {
    PostingThreadState postingState = currentPostingThreadState.get();
    List<Object> eventQueue = postingState.eventQueue;
    //将消息添加到消息队列中
    eventQueue.add(event);

    if (!postingState.isPosting) {
        postingState.isMainThread = isMainThread();
        postingState.isPosting = true;
        if (postingState.canceled) {
            throw new EventBusException("Internal error. Abort state was not reset");
        }
        try {
            while (!eventQueue.isEmpty()) {
                // 消息队列不为空,就一直循环发送消息
                //分析1
                postSingleEvent(eventQueue.remove(0), postingState);
            }
        }
    }
}

```

```

        }
    } finally {
        postingState.isPosting = false;
        postingState.isMainThread = false;
    }
}

//分析1
private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {
    //获取当前消息的class对象，如果是发送的 int 类型，这个会获取到Integer类型对象，所有要用Integer对象进行接收
    Class<?> eventClass = event.getClass();
    boolean subscriptionFound = false;
    if (eventInheritance) {
        //获取当前消息class对象所有继承和实现的类
        List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);
        int countTypes = eventTypes.size();
        for (int h = 0; h < countTypes; h++) {
            Class<?> clazz = eventTypes.get(h);
            //发送消息
            //分析2
            subscriptionFound |= postSingleEventForEventType(event, postingState,
clazz);
        }
    } else {
        subscriptionFound = postSingleEventForEventType(event, postingState,
eventClass);
    }
    if (!subscriptionFound) {
        if (logNoSubscriberMessages) {
            logger.log(Level.FINE, "No subscribers registered for event " +
eventClass);
        }
        if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&
            eventClass != SubscriberExceptionEvent.class) {
            post(new NoSubscriberEvent(this, event));
        }
    }
}

//分析2
private boolean postSingleEventForEventType(Object event, PostingThreadState postingState,
Class<?> eventClass) {
    CopyOnWriteArrayList<Subscription> subscriptions;
    synchronized (this) {
        //查找注册的事件类型里面有没有当前的类型
        subscriptions = subscriptionsByEventType.get(eventClass);
    }
}

```

```

    if (subscriptions != null && !subscriptions.isEmpty()) {
        //如果不为空,就遍历该集合发送消息
        for (Subscription subscription : subscriptions) {
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;
            try {
                //发送消息
                //分析点3
                postToSubscription(subscription, event, postingState.isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }
            if (aborted) {
                break;
            }
        }
        return true;
    }
    return false;
}

//分析点3
private void postToSubscription(Subscription subscription, Object event, boolean
isMainThread) {
    //根据订阅线程的线程模式,选择不同发送方式
    switch (subscription.subscriberMethod.threadMode) {
        case POSTING:
            invokeSubscriber(subscription, event);
            break;
        case MAIN:
            if (isMainThread) {
                invokeSubscriber(subscription, event);
            } else {
                mainThreadPoster.enqueue(subscription, event);
            }
            break;
        case MAIN_ORDERED:
            if (mainThreadPoster != null) {
                mainThreadPoster.enqueue(subscription, event);
            } else {
                // temporary: technically not correct as poster not decoupled from
subscriber
                invokeSubscriber(subscription, event);
            }
            break;
        case BACKGROUND:
            if (isMainThread) {
                backgroundPoster.enqueue(subscription, event);
            }
    }
}

```

```
        } else {
            invokeSubscriber(subscription, event);
        }
        break;
    case ASYNC:
        asyncPoster.enqueue(subscription, event);
        break;
    default:
        throw new IllegalStateException("Unknown thread mode: " +
subscription.subscriberMethod.threadMode);
    }
}
```

postSticky()方法

```
public void postSticky(Object event) {
    synchronized (stickyEvents) {
        //将消息添加到粘性事件的map集合当中
        stickyEvents.put(event.getClass(), event);
    }
    // Should be posted after it is putted, in case the subscriber wants to remove
immediately
    post(event);
}
```

unregister()方法

```
public synchronized void unregister(Object subscriber) {
    List<Class<?>> subscribedTypes = typesBySubscriber.get(subscriber);
    if (subscribedTypes != null) {
        for (Class<?> eventType : subscribedTypes) {
            unsubscribeByEventType(subscriber, eventType);
        }
        typesBySubscriber.remove(subscriber);
    } else {
        logger.log(Level.WARNING, "Subscriber to unregister was not registered before:
" + subscriber.getClass());
    }
}
```

1. 根据注解方法对象参数类型 构建对象

```
class Activity1{
    @Subscribe(threadMode = ThreadMode.MAIN)
    public void updateUI(String event) {
    }

    @Subscribe(threadMode = ThreadMode.ASYNC)
    public void updateUI2(int event) {
    }
}

class Activity2{
    @Subscribe(threadMode = ThreadMode.MAIN)
    public void updateUI(String event) {
    }

    @Subscribe(threadMode = ThreadMode.ASYNC)
    public void updateUI2(int event) {
    }
}

map.put("Stirng.class",
List(Subscription(Activity1&updateUI), Subscription(Activity2&updateUI)));
map.put("Int.class",
List(Subscription(Activity1&updateUI2), Subscription(Activity2&updateUI2)));
```

如上述伪代码，Activity1 和 Activity2 中都使用 @Subscribe 注解接收事件的方法，事件的类型为String, int 那么根据事件对象类型作为 key 保存 map 可以得到：`map.put("Stirng.class", List(Subscription(Activity1&updateUI), Subscription(Activity2&updateUI)));` 表示多个Activity绑定了事件对象类型为 String 的事件。构建这个map有何作用？因为后续 **post**（事件）时要根据事件类型发送给所有已经注册该事件类型的对象（Activity）中的注解方法 (updateUI(String event))

2. 根据注册对象Activity保存当前对象存在的事件类型 ,构建map

```

class Activity1{
    @Subscribe(threadMode = ThreadMode.MAIN)
    public void updateUI(String event) {
    }

    @Subscribe(threadMode = ThreadMode.ASYNC)
    public void updateUI2(int event) {
    }
}

map.put("Activity1", List(String.class, Int.class..));

```

主要作用:

1. 用于查询是否已经注册了当前对象
2. 用于解绑注册,释放资源

如果当前注解的方法接收的是粘性事件, 则根据当前方法的参数类型, 从粘性事件map缓存(postSticky时保存)中获取粘性事件信息, 如果存在则执行当前注解方法.

EventBus 中 方法名 + '>' + 事件类型 作为键, 这意味着对于同一个类来说, subscriberClassByMethodKey 肯定不会键重复 (毕竟一个类中不能够方法名相同且方法参数、个数都相同), 因此它最终会返回 true。这意味着一个类如果使用了多个方法对同一个 Event 对象进行注册, 那么当该 Event 对象被发射出来的时候, 所有的方法都会得到回调。

如果子类重写父类的的订阅方法的时候, 如果发射指定Event的话, 父类的订阅方法将不会执行,而仅会执行子类的订阅方法.

为什么EventBus 传递大数据的时候不会崩溃?

Intent 方式实际上底层parcel对象在不同activity直接传递过程中保存在一个叫做“ Binder transaction buffer”的缓冲区, 并且这个缓冲区大小有限制, 不能超过1M。 EventBus 不存在将数据放在什么缓冲区中, 普通事件直接拿到事件对象, 找到需要调用的方法直接调用即可, 简单的方法参数传递; 而粘性事件, 将数据缓存在内存中, 保存在map, 除非是内存爆炸, 不然是没有大小限制的说法。因此说EventBus可以传递大数据

EventBus 如何指定接收事件方法运行的线程?

EventBus 通过注解的方式指定接收事件的运行线程, 在注册的时候将注解信息保存起来, 当post事件的时候拿到注册时保存的注解信息 (运行线程), 通过不同的 ThreadMode 做不同的处理, 如开启工作线程执行接收事件的方法, 或者通过handler方式将事件运行在主线程中, 以此达到指定线程运行方法的目的

LRU 缓存算法

LruCache 内部实现原理为什么要使用LinkedHashMap?

因为LinkHashMap内部是一个数组加双向链表的形式来存储数据，他能够保证插入时候的数据和取出来时候数据的顺序的一致性。也就是说，我们以什么样的顺序插入数据，就会以什么样的顺序取出数据。并且更重要的一点是，当我们通过get方法获取数据的时候，这个获取的数据会从队列中跑到队列头来，从而很好的满足我们LruCache的算法设计思想。

这跟LinkedHashMap的特性有关，LinkedHashMap的构造函数里有个布尔参数**accessOrder**，当它为true时，LinkedHashMap会以访问顺序为序排列元素，否则以插入顺序为序排序元素。

LinkHashMap 继承**HashMap**，在 **HashMap**的基础上，新增了双向链表结构，每次访问数据的时候，会更新被访问的数据的链表指针，具体就是先在链表中删除该节点，然后添加到链表头**header**之前，这样就保证了链表头**header**节点之前的数据都是最近访问的（从链表中删除并不是真的删除数据，只是移动链表指针，数据本身在map中的位置是不变的）。

LruCache 内部用**LinkHashMap**存取数据，在双向链表保证数据新旧顺序的前提下，设置一个最大内存，往里面**put**数据的时候，当数据达到最大内存的时候，将最老的数据移除掉，保证内存不超过设定的最大值。

源码分析

```
public class LruCache<T, Y> {
    //第一个参数是最大容量, 第二个参数是负载因子
    //第三个参数是存储顺序, true:按访问顺序 false:按插入顺序
    private final Map<T, Y> cache = new LinkedHashMap<>(100, 0.75f, true);
    private final long initialMaxSize;
    private long maxSize;
    private long currentSize;

    //初始化最大容量和最大容量
    public LruCache(long size) {
        this.initialMaxSize = size;
        this.maxSize = size;
    }
}
```

设置缓存最大值

```
@Nullable
public synchronized Y put(@NotNull T key, @Nullable Y item) {
    //返回值 是一个 1
    final int itemSize = getSize(item);
    //如果1 大于等于最大值就返回 null
    //所以不能将size设置为1
    if (itemSize >= maxSize) {
        onItemEvicted(key, item);
        return null;
    }
    //容量加1
    if (item != null) {
        currentSize += itemSize;
    }

    @Nullable
    //判断是否已经存在这个键值对，如果有就减去旧的容量
```

```

final Y old = cache.put(key, item);
if (old != null) {
    currentSize -= getSize(old);

    if (!old.equals(item)) {
        onItemEvicted(key, old);
    }
}
//容量判断
evict();

return old;
}

private void evict() {
    trimToSize(maxSize);
}

protected synchronized void trimToSize(long size) {
    Map.Entry<T, Y> last;
    Iterator<Map.Entry<T, Y>> cacheIterator;
    //存储的容量大于最大容量
    while (currentSize > size) {
        cacheIterator = cache.entrySet().iterator();
        //获取头部，并删除头部第一个
        last = cacheIterator.next();
        final Y toRemove = last.getValue();
        currentSize -= getSize(toRemove);
        final T key = last.getKey();
        cacheIterator.remove();
        onItemEvicted(key, toRemove);
    }
}
}

```

LinkedHashMap

LinkHashMap 继承HashMap，跟 HashMap 不同的是 HashMap 是无序的， LinkedHashMap 是有序的。在 HashMap 的基础上，新增了双向链表结构，每次访问数据的时候，会更新被访问的数据的链表指针，具体就是先在链表中删除该节点，然后添加到链表头 header 之前，这样就保证了链表头 header 节点之前的数据都是最近访问的（从链表中删除并不是真的删除数据，只是移动链表指针，数据本身在 map 中的位置是不变的）

内部重写了 HashMap 的 get() 和 put() 方法，

LinkedHashMap 默认的排序方式是按照 插入顺序排序的

有两种排序方式，通过参数的 boolean 值控制，**false 表示 插入排序， true 表示 访问顺序**

```
Map<T, Y> cache = new LinkedHashMap<>(100, 0.75f, true);
```

HashMap

链接：

<https://blog.csdn.net/justloveyou/article/details/62893086>

[美团团队Java8 HashMap](#)

[漫画解析什么是HashMap](#)

[常见问题](#)

在JDK 8.0以前，HashMap采用位桶+链表实现，即使用链表处理冲突，同一个hash的值的链表都存储在一个链表里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。

而在JDK 8.0中，HashMap采用位桶+链表+红黑树实现，当链表长度超过值(8)时，将链表转换为红黑树，这样大大减少查找时间。

在Java中，HashMap的实现采用了（数组 + 链表 + 红黑树）的复杂结构，数组的一个元素又称作桶。

HashMap最多只允许一条Entry的键为Null(多条会覆盖)，但允许多条Entry的值为Null

虽然 HashMap 和 HashSet 实现的接口规范不同，但是它们底层的 Hash 存储机制完全相同。实际上，HashSet 本身就是在 HashMap 的基础上实现的

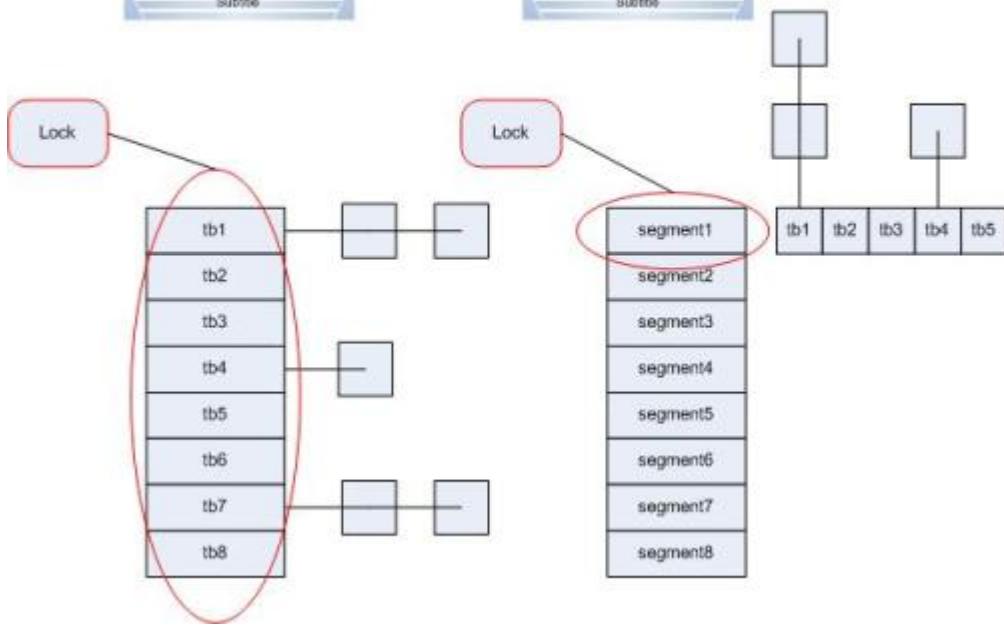
必须指出的是，虽然容器号称存储的是 Java 对象，但实际上并不会真正将 Java 对象放入容器中，只是在容器中保留这些对象的引用。也就是说，Java 容器实际上包含的是引用变量，而这些引用变量指向了我们要实际保存的 Java 对象。

HashMap**非线程安全**，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。

1. 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。
2. 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。
 - 如果内存空间很多，又对时间效率要求高，可以降低负载因子的值。
 - 如果内存空间紧张，而对时间效率要求不高，可以增加负载因子。
3. HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。
4. JDK1.8引入红黑树大程度优化了HashMap的性能。

HashTable 和 ConcurrentHashMap 的区别

HashTable ConcurrentHashMap



HashTable

键值不能为空,HashMap默认容量是16,HashTable 默认容量是11,负载因子都是0.75

HashTable扩容时,长度是原来的两倍+1

把所有的get,put 方法都加上锁实现同步

HashTable 实现方式是锁住整个hash 表, 在线程竞争激烈的情况下,效率会很低

当HashTable的大小增加到一定的时候,性能就会急剧下降,因为迭代时需要被锁定很长时间.

ConcurrentHashMap

JDK8之前, 采用数据分段,把每个分段分别加锁,实现同步,提高效率

JDK8之后,采用CAS算法提高效率

而ConcurrentHashMap 实现的方式是锁住单个桶

ConcurrentHashMap将hash表分为16个桶（默认值）, 诸如get,put,remove等常用操作只锁当前需要用到的桶

原来只能一个线程进入, 现在却能同时16个写线程进入（写线程才需要锁定, 而读线程几乎不受限制, 之后会提到）, 并发性的提升是显而易见的。

- Segment的get操作是不需要加锁的。因为volatile修饰的变量保证了线程之间的可见性
- Segment的put操作是需要加锁的, 在插入时会先判断Segment里的HashEntry数组是否会超过容量(threshold),如果超过需要对数组扩容, 翻一倍。然后在新的数组中重新hash, 为了高效, ConcurrentHashMap只会对需要扩容的单个Segment进行扩容

为什么ConcurrentHashMap 不能完全替代HashTable?

因为ConcurrentHashMap是弱一致性,其get方法没有上锁,会导致get到的元素并不是当前并行还未执行完的put值,读取到的数据并不一定是最终的值,在一些要求强一致性的场景下可能会出错.

为什么HashMap链表插入使用的是 头插入法? (jdk 8 是插入尾部)

因为HashMap的发明者认为, 后插入的Entry被查找的可能性更大。

为什么HashMap不是线程安全的?

HashMap在扩容的时候,元素会重新排列,(同一个桶里面的元素采用头插入法,原来的链表顺序会被倒置),并发场景可能形成循环链表

如何减少hash冲突? 如何增大hash冲突?

HashMap的工作原理, get, put方法的工作原理

如果HashCode值相同, 你如何取值?

get方法:

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

/**
 * Implements Map.get and related methods
 *
 * @param hash hash for key
 * @param key the key
 * @return the node, or null if none
 */
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
```

```

        ((k == e.key) == key || (key != null && key.equals(k)))
        return e;
    } while ((e = e.next) != null);
}
return null;
}

```

- 根据hash值查找到指定位置的数据
- 校验指定位置第一个节点的数据是key是否为传入的key，如果是直接返回第一个节点，否则继续查找第二个节点
- 如果数据是TreeNode（红黑树结构），直接通过红黑树查找节点数据并返回
- 如果是链表结构，循环查找所有节点，返回数据
- 如果没有找到符合要求的节点，返回null

put方法：

```

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //1.判断桶是否为空，为空的话就进行初始化
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    //2.根据当前 key 的 hashCode 值判断是否为空，为空表明没有 Hash 冲突就直接在当前位置创建一个新桶
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        //3.果当前桶有值（Hash 冲突），那么就要比较当前桶中的 key、key 的 hashCode 与写入的 key 是否相等，相等就赋值给 e，在第 8 步的时候会统一进行赋值及返回。
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        //4.如果是红黑树，则写入到红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            //5.如果是链表，就需要将当前的 key、value 封装成一个新节点写入到当前桶的后面（形成链表）。
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    //6.判断当前链表是否大于红黑树预设值，如果大于则转换成红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                //7.遍历中，找到的key相同则直接退出遍历
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
    }
}

```

```

        }
    }
    //8.如果e 不等于空,就相当于存在相同的key,就需要将值覆盖
    if (e != null) { // existing mapping for key
        V oldValue = e.value;
        if (!onlyIfAbsent || oldValue == null)
            e.value = value;
        afterNodeAccess(e);
        return oldValue;
    }
}
++modCount;
//9.是否需要扩容
if (++size > threshold)
    resize();
afterNodeInsertion(evict);
return null;
}

```

为什么用数组 + 链表的方式

数组是用来确定桶的位置的，利用元素的key的hash值对数组长度取模得到。

链表是用来解决hash冲突问题，当出现hash一样的时候，就会在数组的对应位置形成一条链表。

数组的特点：寻址容易，插入和删除困难

链表的特点：寻址困难，插入和删除容易

为什么不将链表全部换成二叉树呢？

- 第一个是链表的结构比红黑树简单，构造红黑树要比构造链表复杂，所以在链表的节点不多的情况下，从整体的性能看来，数组+链表+红黑树的结构不一定比数组+链表的结构性能高。
- 第二个是HashMap频繁的resize（扩容），扩容的时候需要重新计算节点的索引位置，也就是会将红黑树进行拆分和重组其实这是很复杂的，这里涉及到红黑树的着色和旋转，有兴趣的可以看看红黑树的原理，这又是一个比链表结构耗时的操作，所以为链表树化设置一个阀值是非常有必要的。

当链表转换为红黑树之后，什么时候退化为链表？

为6的时候退转为链表。中间有个差值7可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

为什么String, Integer这样的wrapper类适合作为键？

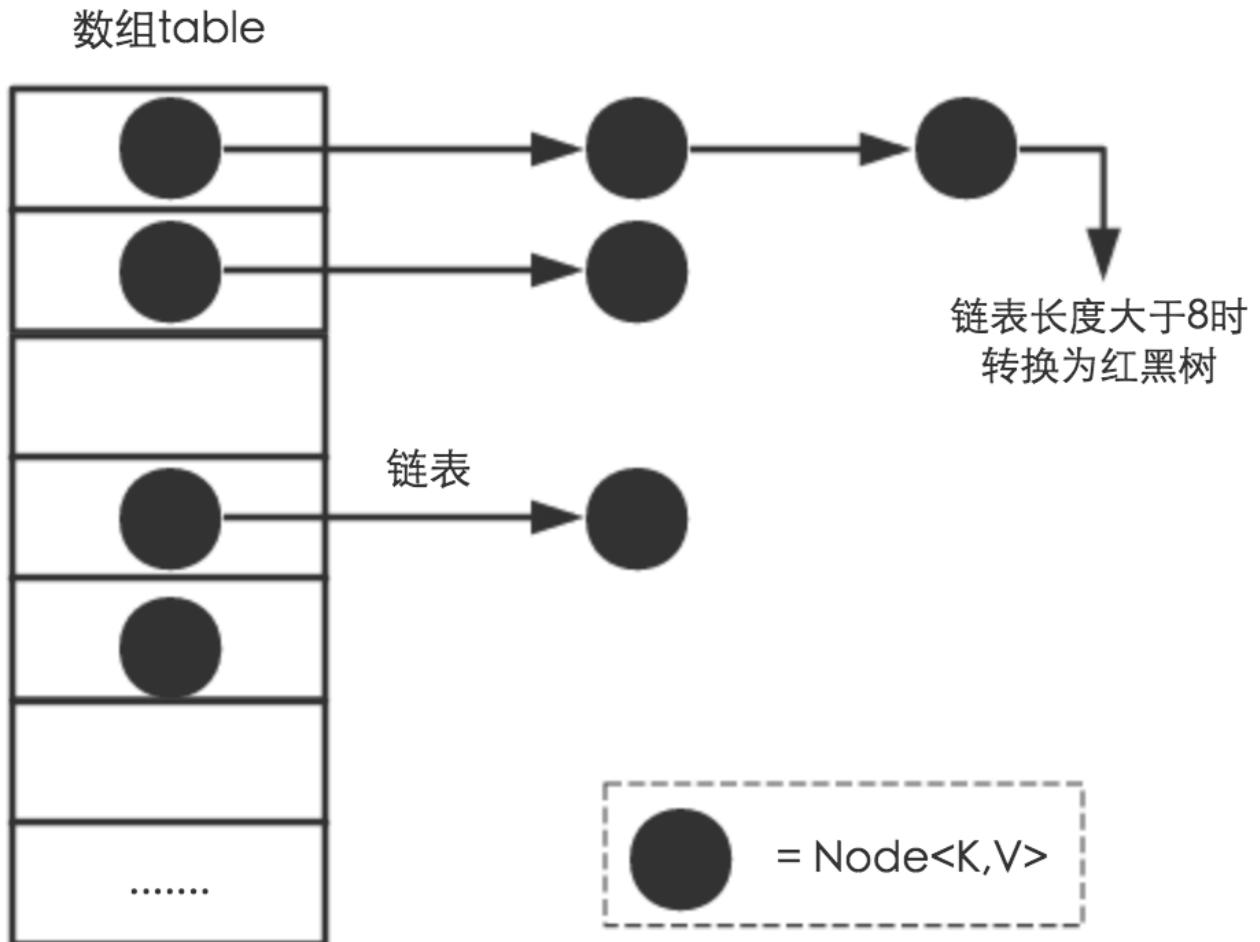
String, Integer这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，

那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能

- **HashMap**默认的初始化大小为16，之后每次扩充为原来的2倍。
- **HashTable**默认的初始大小为11，之后每次扩充为原来的 $2n+1$

HashMap如何解决hash碰撞？

Java中HashMap采用链地址法解决Hash碰撞



`h=hashCode() : 1111 1111 1111 1111 1111 0000 1110 1010` 调用`hashCode()`

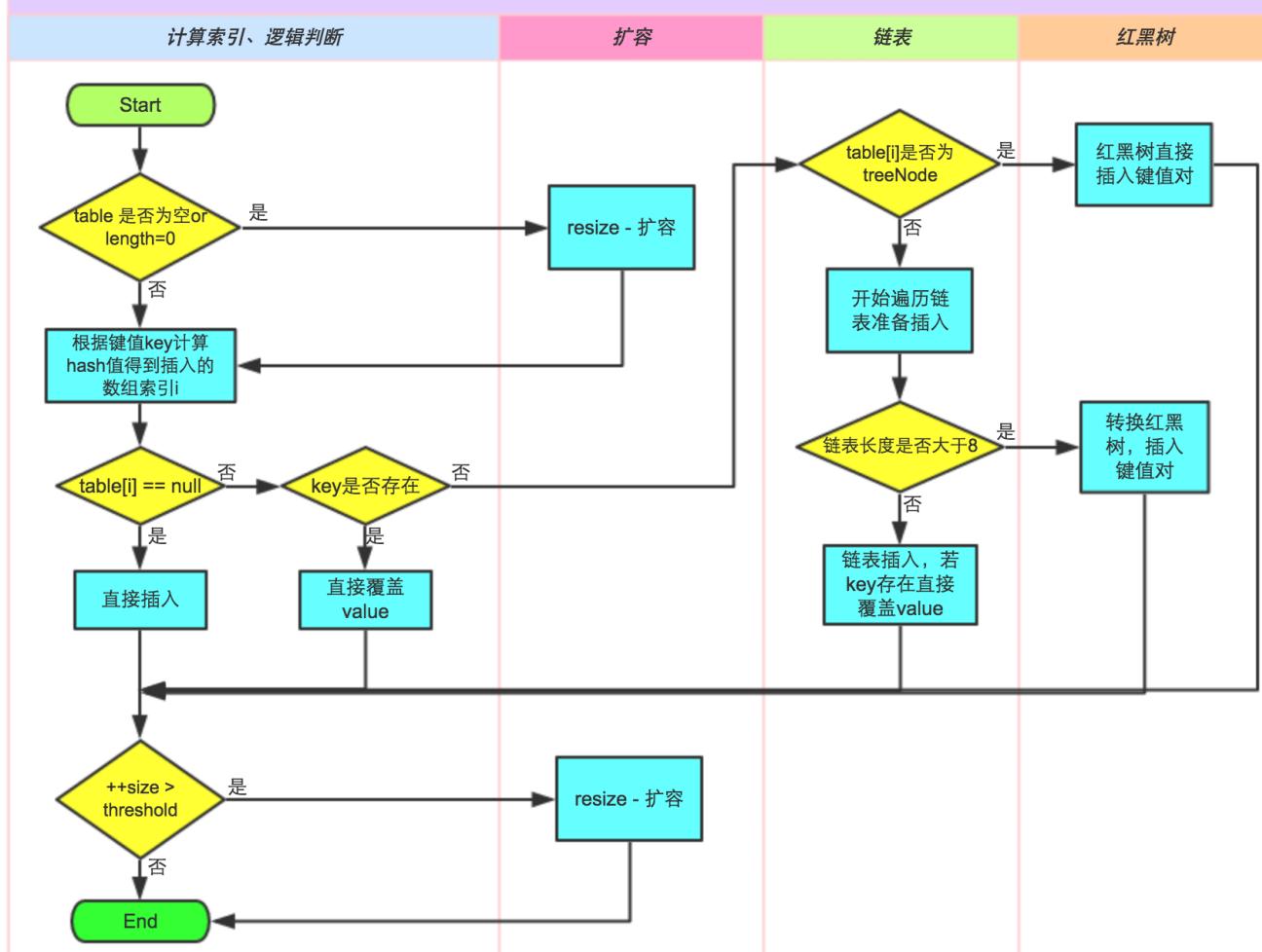
`h : 1111 1111 1111 1111 1111 0000 1110 1010`
`h >>> 16 : 0000 0000 0000 0000 1111 1111 1111 1111` 计算Hash

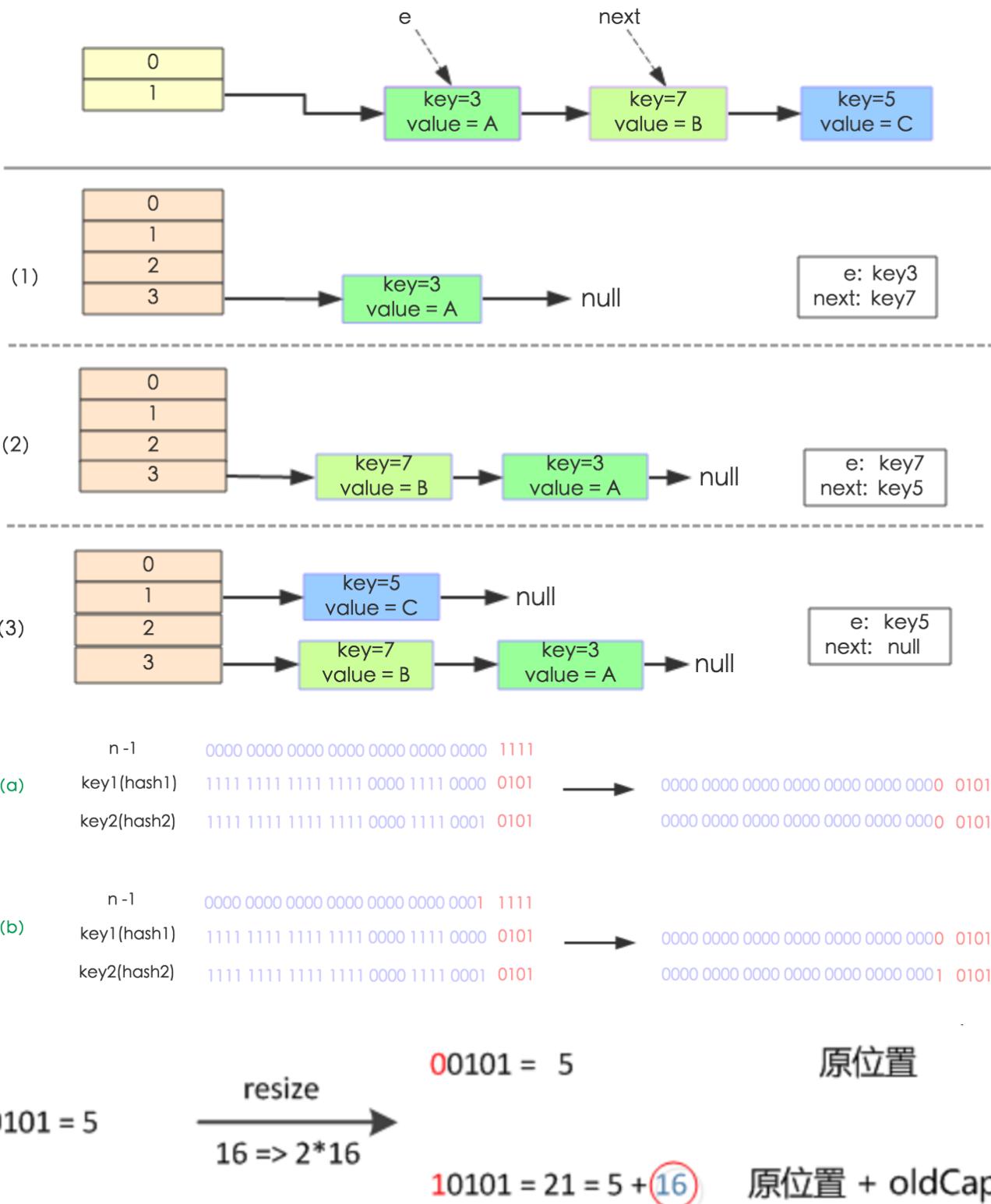
`hash = h ^ (h >>> 16) : 1111 1111 1111 1111 0000 1111 0001 0101`

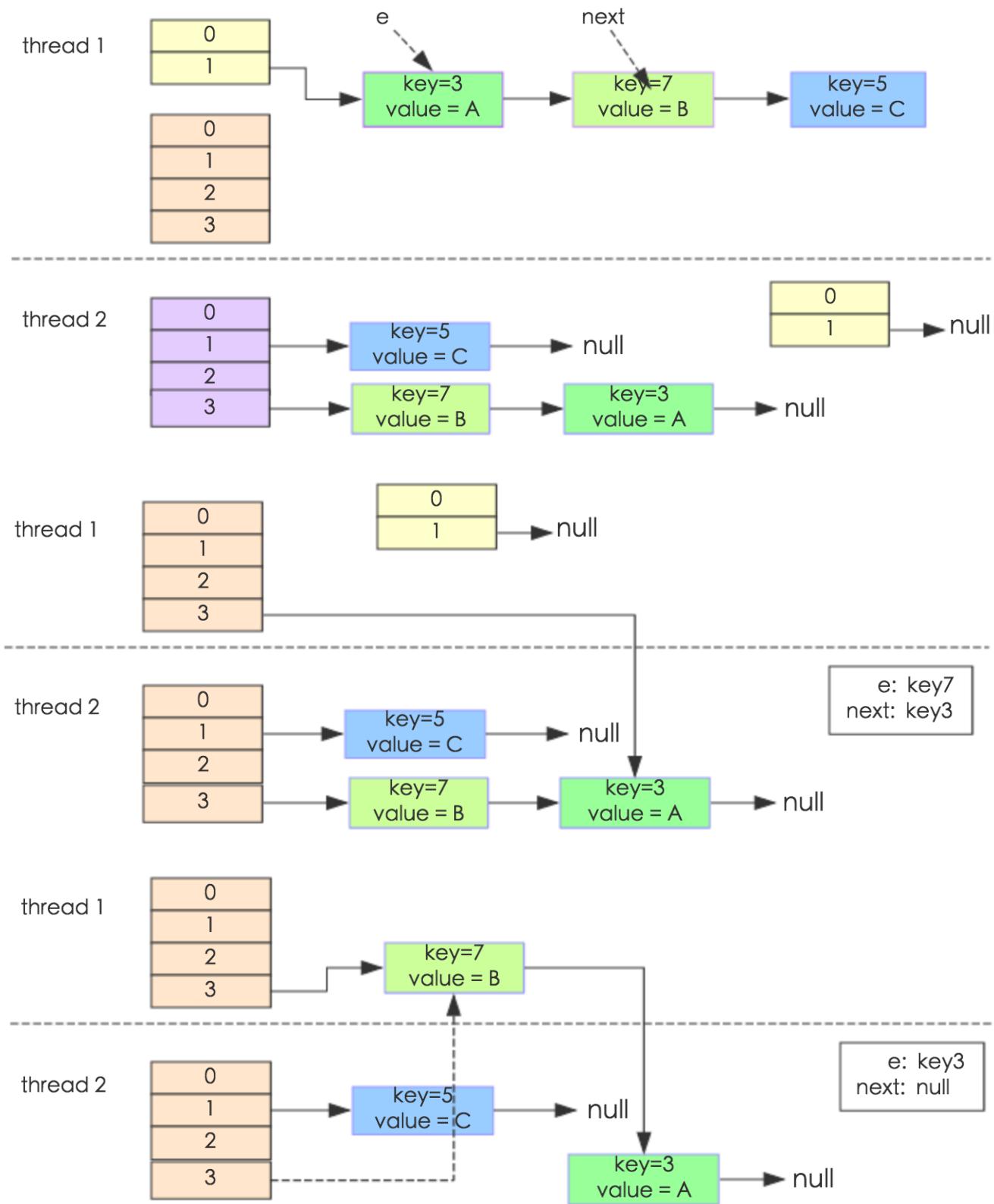
`(n - 1) & hash : 0000 0000 0000 0000 0000 0000 1111
1111 1111 1111 1111 0000 1111 0001 0101` 计算下标

`0101 = 5`

HashMap之`put(K key, V value)`方法







红黑树

<https://juejin.im/post/5a27c6946fb9a04509096248>

红黑树是一种弱平衡的二叉树,由于是弱平衡的,在相同的节点情况下,AVL树的高度低于红黑树.

但是相对与严格的AVL树来说,它的旋转次数少,所以对于搜索,插入和删除操作较多的情况下,我们就使用红黑树.

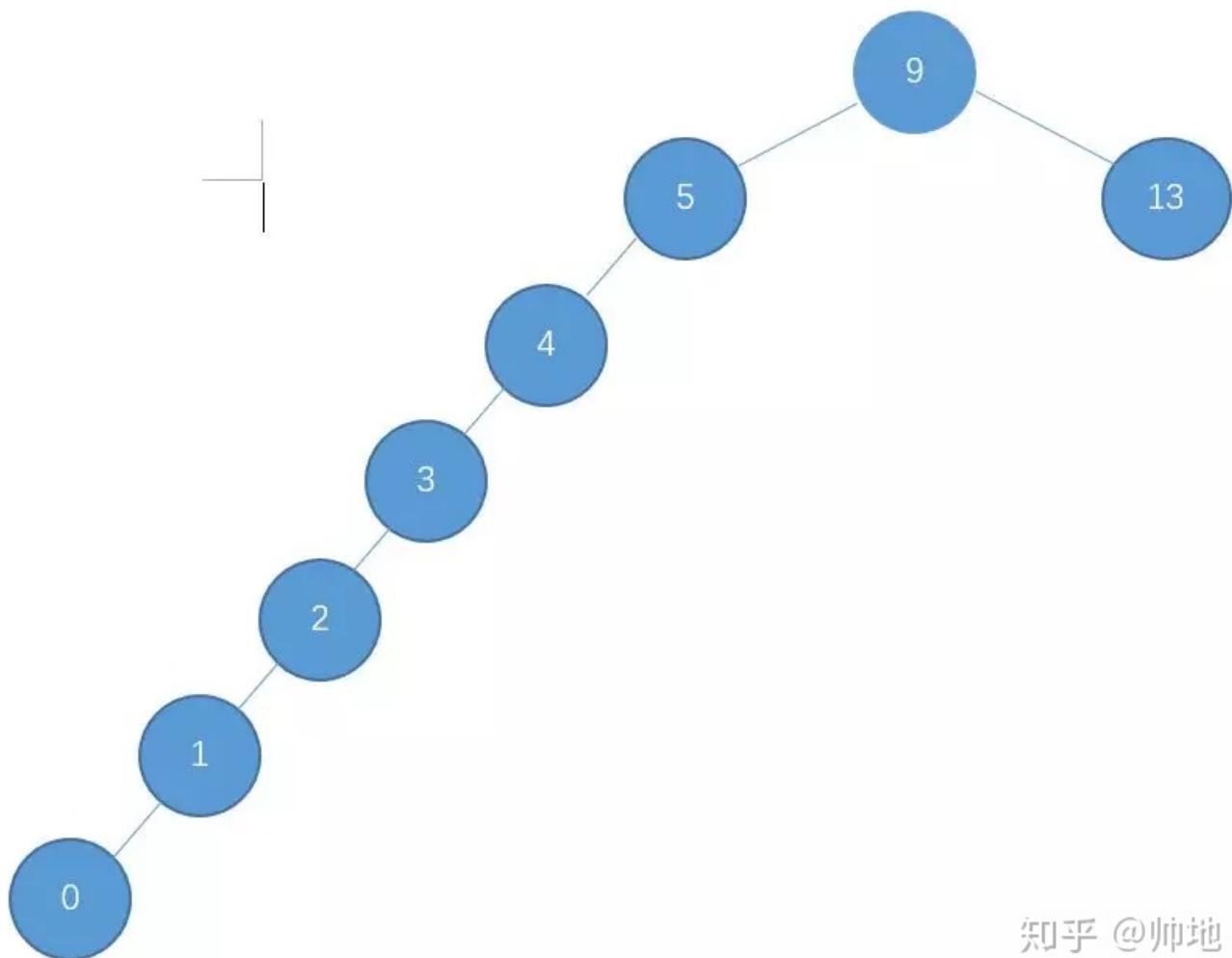
1. 节点是红色或黑色。
2. 根节点是黑色。
3. 每个叶子节点都是黑色的空节点（NIL节点）。
4. 每个红色节点的两个子节点都是黑色。（从每个叶子到根的所有路径上不能有两个连续的红色节点）
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

二叉树、平衡树 和红黑树区别：

链接：<https://zhuanlan.zhihu.com/p/72505589>

二叉树缺点：

如果出现下面这种极端情况性能会很差：



为什么有了平衡树还要红黑树？

因为平衡树要求每个节点的左子树和右子树的高度差不能大于1,导致每次进行插入或者删除节点的时候, 大概率需要通过左旋和右旋来进行调整.频繁的进行调整会使平衡树的性能大打折扣.

如果在那种插入、删除很频繁的场景, 平衡树需要频繁的进行调整, 这会使得平衡树的性能大打折扣。

红黑树在插入和删除等操作的时候，不会像平衡树那样，频繁的破坏红黑树的规则，所以不需要频繁进行调整，也可以说红黑树是一种不太严格的平衡树。

平衡树是为了解决二叉查找树退化为链表的情况，而红黑树是为了解决平衡树在插入、删除等操作需要频繁调整的情况。

C++

虚函数 virtual: `virtual` 是 C++ 中的一个关键字，用于声明函数，表示虚函数。用于告诉编译器不要静态链接到该函数，改为动态链接。

预处理: 在 C++ 中有一个方法，可以让我们在程序编译前，对代码做一些处理，称为预处理。这是 Java 中没有的，在 C++ 中却经常使用到。

预处理是一些指令，但是这些指令并不是 C++ 语句，所以不需要以分号 ; 结束。C++ 中常用的预处理有以下几个 `#include`、`#define`、`#if`、`#else`、`#ifdef`、`#endif` 等。

最常用的一个预处理语句 `#define`，通常称为宏定义。

```
class A {  
public:  
    int i;  
}  
  
int main() {  
    // 不通过new的方式创建指针变量  
    A a = new A();  
    A *c = &a; // 定义指针 c 指向 a  
  
    // 直接new的方式创建指针变量  
    A *a = new A();  
    a->i = 0;  
  
    printf("%d\n", a->i);  
    // 输出: 0  
  
    // 删除指针变量, 回收内存  
    delete a;  
  
    return 0;  
}
```

通过 `new` 的方式创建的指针变量和不通过 `new` 创建的变量最大的区别在于：

通过 `new` 创建的指针需要我们自己手动回收内存，否则将会导致内存泄漏。回收内存则是通过 `delete` 关键字进行的，也就是说 `new` 和 `delete` 是成对调用的。

引用

```
// 声明一个普通变量 i  
int i = 0;  
  
// 声明定义一个引用 j  
int &j = i;  
  
j = 1;  
  
printf("%d, %d\n", i, j)  
  
// 输出: 1, 1
```

引用指的是:为一个变量起一个别名,也就是说,它是某个已存在变量的另一个名字.

引用和指针的不同

不存在空引用. 引用必须连接到一块合法的内存

一旦引用被初始化为一个对象,就不能被指向到另一个对象,指针可以在任何时候指向到另一个对象

引用必须在创建的时候初始化,指针可以在任何时候初始化.

图片处理

- **fresco**

可以指定图片的宽高比例

可以显示图片加载进度

采用ARGB_8888显示图片

先加载小尺寸图片,再加载大尺寸图片(glide只有占位图)

首次需要进行初始化,一般是在Application初始化, `Fresco.initialize()`

必须声明 android: layout_width 和 android:layout_height

Fresco支持渐进式的网路JPEG图,图会从模糊到清晰渐渐呈现,不过仅仅支持网络图

如果需要支持GIF图需要导包 `'com.facebook.fresco:animated-gif:0.12.0'`

```
compile 'com.facebook.fresco:fresco:1.5.0'  
compile 'com.facebook.fresco:animated-gif:1.5.0' //加载gif动图需添加此库  
compile 'com.facebook.fresco:animated-webp:1.5.0' //加载webp动图需添加此库  
compile 'com.facebook.fresco:webpsupport:1.5.0' //支持webp需添加此库  
compile 'com.facebook.fresco:imagepipeline-okhttp3:1.5.0' //网络实现层使用okhttp3需添加此库  
compile 'jp.wasabeef:fresco-processors:2.1.0@aar' //用于提供fresco的各种图片变换
```

不支持相对路径的URI，所有的uri必须是绝对路径

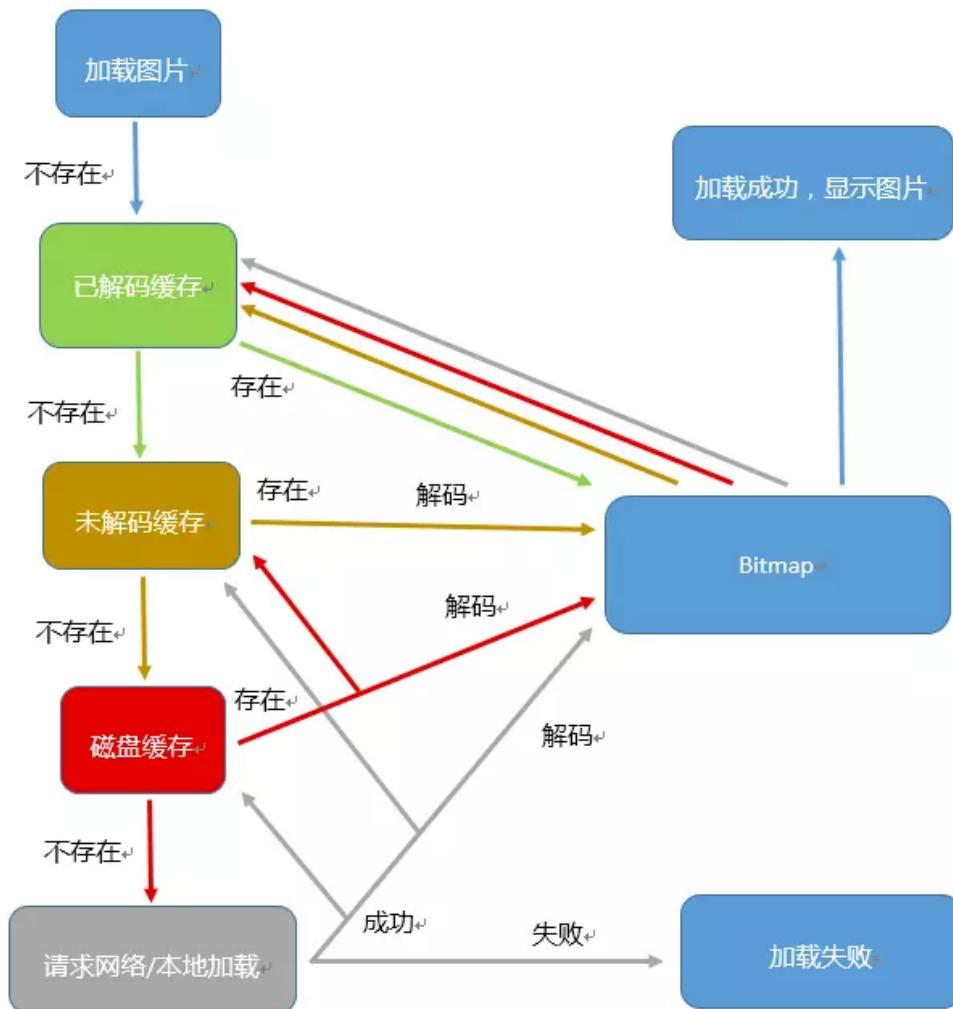
默认使用HttpURLConnection访问网络，可以自己设定

内存自动回收，三级缓存机制（两级内存缓存，一级磁盘缓存）

缓存的时候只会缓存原图（意味着如果控件比图片小的话，加载的时候会比glide更耗内存）

缺点：

体积大（2~3M，glide是500kB），侵入性较强，



Glide

Glide内存缓存的使用的LruCache算法。不过除了LruCache算法之外，Glide还结合了一种弱引用的机制，共同完成了内存缓存功能。这样做的目的是把正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。分工合作，保护正在使用的资源不会被LruCache算法回收掉。（划重点）

Glide 加载 Gif 的原理比较简单，就是将 Gif 解码成多张图片进行无限轮播，每帧切换都是一次图片加载请求，再加载到新的一帧数据之后会对旧的一帧数据进行清除，然后再继续下一帧数据的加载请求，以此类推，使用 Handler 发送消息实现循环播放。

体积小

源码解析

不管你在Glide.with()方法中传入的是Activity、FragmentActivity、v4包下的Fragment、还是app包下的Fragment，最终的流程都是一样的，那就是会向当前的Activity当中添加一个隐藏的Fragment，因为Glide需要知道加载的生命周期，因为Fragment的生命周期和Activity是同步的，如果Activity被销毁了，Fragment是可以监听到的，这样Glide就可以捕获这个事件并停止图片加载了。

如果我们是在非主线程当中使用的Glide，那么不管你是传入的Activity还是Fragment，都会被强制当成Application来处理

序列化和反序列化

Serializable 接口

Serializable接口是Java所提供的，为对象提供标准的序列化和反序列化操作。通常一个对象实现Serializable接口，该对象就具有被序列化和反序列化的能力，而且几乎所有工作有系统自动完成。Serializable接口内serialVersionUID可指定也可以不指定，其作用是用来判断序列化前和反序列化的类版本是否发生变化。该变量如果值不一致，表示类中某些属性或者方法发生了更改，反序列化则出问题。（静态成员变量和transient关键字标记的成员不参与序列化过程）

Parcelable 接口

Parcelable 接口是Android所提供的，其实现相对来说比较复杂。实现该接口的类的对象就可以在Intent和Binder进行传递。

两者的区别

- Serializable是Java提供的接口，使用简单，但序列化与反序列化需要大量的IO操作，所以开销比较大。
- Parcelable是Android提供的序列化方法，使用麻烦但效率高。在Android开发中，将对象序列化到设备或者序列化后通过网络传输建议使用Serializable接口，其他情况建议是用Parcelable接口，尤其在内存的序列化上。例如Intent和Binder传输数据。

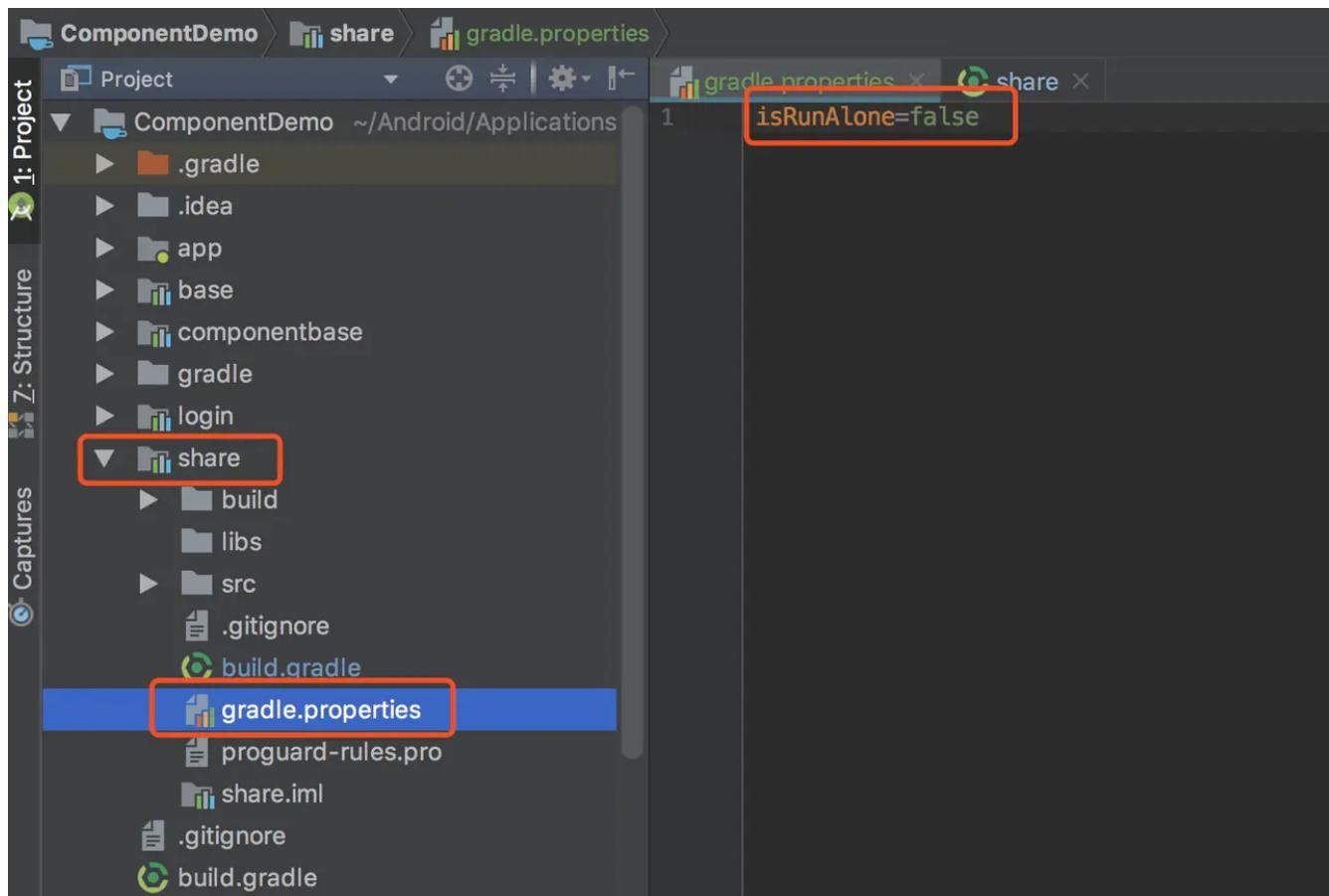
日志

- 网络日志
- debug日志
- 本地文件日志
- 埋点
- aop日志

组件化

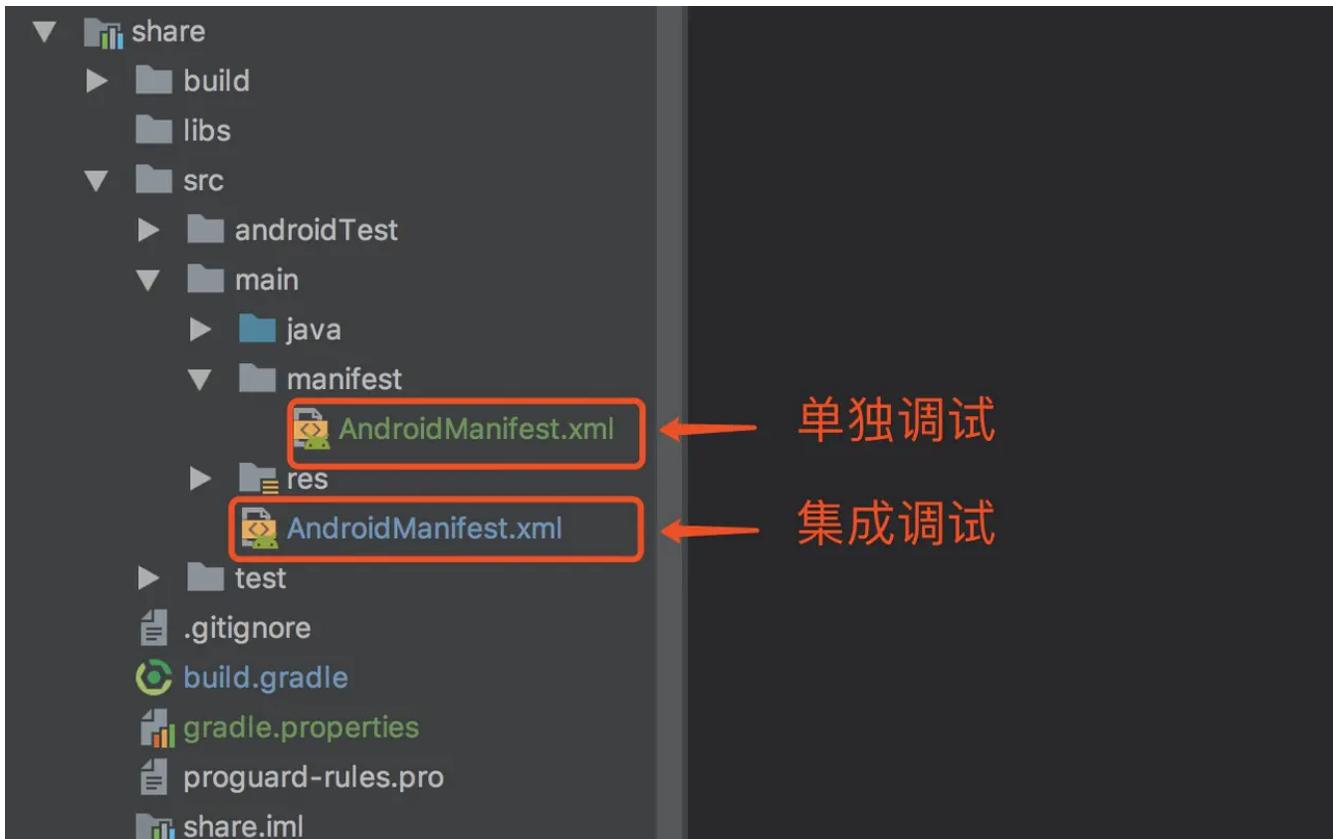
参考 <https://juejin.im/post/6844903649102004231#comment>

动态配置组件类型(是library 还是 Application)



```
if (isRunAlone.toBoolean()) {  
    apply plugin: 'com.android.application'  
} else {  
    apply plugin: 'com.android.library'  
}  
  
android {  
    compileSdkVersion compile_sdk_version.toInt()  
}  
  
defaultConfig {  
    if (isRunAlone.toBoolean()) {  
        applicationId "com.loong.login"  
    }  
    minSdkVersion min_sdk_version.toInt()  
    targetSdkVersion target_sdt_version.toInt()  
    versionCode 1  
    versionName "1.0"  
}  
  
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"  
  
javaCompileOptions {  
    annotationProcessorOptions {  
        arguments = [ moduleName : project.getName() ]  
    }  
}
```

动态配置 AndroidManifest 文件



其中 AndroidManifest 文件中的内容如下：

```
// main/manifest/AndroidManifest.xml 单独调试
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.loong.share">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".ShareActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>

// main/AndroidManifest.xml 集成调试
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.loong.share">
```

```
<application android:theme="@style/AppTheme">
    <activity android:name=".ShareActivity"/>
</application>

</manifest>
```

作者：玉刚说
链接：<https://juejin.im/post/6844903649102004231>

然后在 build.gradle 中通过判断 isRunAlone 的值，来配置不同的 ApplicationId 和 AndroidManifest.xml 文件的路径：

```
// share 组件的 build.gradle

android {
    defaultConfig {
        if (isRunAlone.toBoolean()) {
            // 单独调试时添加 applicationId，集成调试时移除
            applicationId "com.loong.login"
        }
        ...
    }

    sourceSets {
        main {
            // 单独调试与集成调试时使用不同的 AndroidManifest.xml 文件
            if (isRunAlone.toBoolean()) {
                manifest.srcFile 'src/main/manifest/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
            }
        }
    }
}
```

组件间数据传递和方法互相调用

组件Application的动态配置

组件间界面跳转

使用alibaba 开源的 **ARouter** 进行组件间界面的跳转

比如A 跳转 到 B

1. 两个需要跳转的组件都需要添加上 Arouter 的依赖

```
dependencies {  
    ...  
    annotationProcessor 'com.alibaba:arouter-compiler:1.1.4'  
}
```

2. 在B组件的页面添加@Route注解, 并添加地址

```
@Route(path = "/share/share")  
public class ShareActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
  
    }  
}
```

3. 在A 的界面添加跳转

```
@Route(path = "/account/login")  
public class LoginActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_login);  
    }  
    public void loginShare(View view) {  
        ARouter.getInstance().build("/share/share").withString("share_content", "分享数据到微博").navigation();  
    }  
}
```

插件化

Activity 插件化方案使用的大多是占坑的思想, 不同的是如何在检验之前替换, 在生成对象的时候还原.

- 美团插件化
- 360 RePlugin

插件化, 组件化, 模块化

组件化在运行时不具备动态添加或修改组件的功能, 但是插件化是可以的

组件化实现的是解耦与加快编译, 隔离不需要关注的部分。插件化实现的也是解耦与加快编译, 同时实现热插拔也就是热更新。

组件化的灵活性在于按加载时机切换, 分离出独立的业务组件, 比如微信的朋友圈 插件化的灵活性在于是加载 apk, 完全可以动态下载, 动态更新, 比组件化更灵活。

组件化能做的只是, 朋友圈已经有了, 我想单独调试, 维护, 和别人不耦合。但是和整个项目还是有关联的。
插件化可以说朋友圈就是一个app, 我需要整合了, 把它整合进微信这个大的app里面

其实从框架名称就可以看出: 组 和 插。

组本来就是一个系统, 你把微信分为朋友圈, 聊天, 通讯录按意义上划为独立模块, 但并不是真正意义上的独立模块。插本来就是不同的apk, 你把微信的朋友圈, 聊天, 通讯录单独做一个完全独立的app, 需要微信的时候插在一起, 就是一个大型的app了。

插件化的加载是动态的, 这点很重要, 也是灵活的根源

通信方式

模块化的通信方式, 无非是相互引入; 我抽取了common, 其他模块使用自然要引入这个module 组件化的通信方式, 按理说可以划分为多种, 主流的是隐式和路由。隐式的存在使解耦与灵活大大降低, 因此路由是主流 插件化的通信方式, 不同插件本身就是不同的进程了。因此通信方式偏向于Binder机制类似的进程间通信

ListView vs RecyclerView

• 点击事件

ListView原生提供Item单击、长按的事件, 而RecyclerView则需要使用onTouchListener, 相对自定义实现会比较复杂。

分割线

ListView可以很轻松的设置divider属性来显示Item之间的分割线, RecyclerView需要我们自己实现 ItemDecoration, 前者使用简单, 后者可定制性强。

布局类型

AdapterView提供了ListView与GridView两种类型，分别对应流式布局与网格式布局。RecyclerView提供了LinearLayoutManager、GridLayoutManager与之抗衡，相对而言，使用RecyclerView来进行更换布局方式更为轻松。只需要更换一个变量即可，而对于AdapterView而言则是需要更换一个View了。

缓存方式

ListView使用了一个名为RecyclerBin的类负责试图的缓存，而Recycler则使用Recycler来进行缓存，原理上两者基本一致。RecyclerView：里面存储的不是View，而是ViewHolder

局部刷新

这是一个很有用的功能，在ListView中我们想局部刷新某个Item需要自己来编写刷新逻辑，而在RecyclerView中我们可以通过 `notifyItemChanged(position)` 来刷新单个Item，甚至可以通过 `notifyItemChanged(position, payload)` 来传入一个payload信息来刷新单个Item中的特定内容。

动画

作为视觉动物，我相信很多人喜欢RecyclerView都和它简单的动画API有关，因为之前对ListView做动画比较困难，并且不舒服。

嵌套布局

嵌套布局也是最近比较火的一个概念，RecyclerView实现了 NestedScrollingChild 接口，使得它可以和一些嵌套组件很好的工作。我们再来看ListView原生独有的几个特点：

- 头部与尾部的支持
- ListView原生支持添加头部与尾部，虽然RecyclerView可以通过定义不同的Type来做支持，但实际应用中，如果封装的不好，是很容易出问题的，因为Adapter中的数据位置与物理数据位置发生了偏移。

多选

支持多选、单选也是ListView的一大长处，其实如果要我们自己在RecyclerView中去做支持还是需要不少代码量的。多数据源的支持ListView提供了CursorAdapter、ArrayAdapter，可以让我们很方便的从数据库或者数组中获取数据，这在测试的时候很有用。

总结

综上，我们会发现RecyclerView的最大特点就是灵活，正因为这种灵活，因此会牺牲了某些便利性。而AdapterView相对来讲就比较刻板，但它原生为我们提供了很多有用的方法来便于我们快速开发。ListView并不像当年的ActivityGroup，在Fragment出来后就被标记为Deprecated。两者目前还是一种互补的关系，起码在短时间内RecyclerView还并不能完全替代AdapterView，个人感觉原因有两个，一是目前太多的应用使用了ListView，并且ListView向RecyclerView转变也没有无损的方法。第二点，比如我就是想添加个头部，每个item带个点击事件这类简单的需求，ListView完全可以很轻松的胜任，没必要舍近求远来使用RecyclerView。因此，在实际应用中选择更适合自己的就好。

当然，从Google最近几次的更新来看，RecyclerView的进化还是很迅速的，而ListView则几乎没什么变动，所以RecyclerView绝对是大大的潜力股呀。

ListView相比RecyclerView，有一些优点：

- `addHeaderView()`, `addFooterView()` 添加头视图和尾视图。
- 通过"android:divider"设置自定义分割线。
- `setOnItemClickListener()` 和 `setOnItemLongClickListener()` 设置点击事件和长按事件。

这些功能在RecyclerView中都没有直接的接口，要自己实现（虽然实现起来很简单），因此如果只是实现简单的显示功能，ListView无疑更简单。

RecyclerView相比ListView，有一些明显的优点：

- 默认已经实现了View的复用，不需要类似`if(convertView == null)`的实现，而且回收机制更加完善。
- 默认支持局部刷新。
- 容易实现添加item、删除item的动画效果。
- 容易实现拖拽、侧滑删除等功能。

RecyclerView是一个插件式的实现，对各个功能进行解耦，从而扩展性比较好。

缺点：

RecyclerView也不是万能的，它的灵活性也是有一定限制的，比如我就遇到了一不是很好解决的问题：Recyler的缓存级别是一个Item的整个View，而我们没办法自定义缓存级别，这样说比较抽象，举个例子，我的某些Item的某个子View加载很耗时，所以我希望我在上下滑动的时候，Item的其它View是可以被回收利用的，但这个加载很耗时的View是不要重复使用的。即我希望用空间换取时间来获取滑动的流畅性。当然，这样的需求不常见，RecyclerView也不能很好的满足这一点。

性能优化

<https://juejin.im/post/6844904105438134286>

绘制优化 `onDraw`方法会被频繁调用，所以`onDraw`中不要创建新的布局对象，不要做耗时的操作。

线程优化 采用线程池，避免程序中存在大量的Thread，线程池可以重用内部的线程，从而避免了线程的创建和销毁所带来的性能开销，同时线程池还能有效的控制线程池的最大并发数，避免大量的线程因互相抢占系统资源从而导致阻塞现象的发生，因此在实际开发中，我们要尽量使用线程池，而不是每次都要创建一个Thread对象。

安装包大小优化

抖音安装包优化方案：<https://juejin.im/post/5e809cf46fb9a03c763cf348>

1. 对图片进行压缩，优化图片像素
2. 使用McImage 进行图片压缩

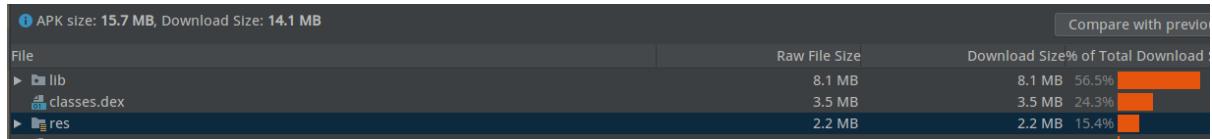
插件地址:<https://github.com/smallSohoSolo/McImage>

使用结果：

压缩前:res 资源文件大小 12MB



压缩后:res 资源文件大小 2.2MB 收益9.8MB



MclImage插件的优势:

- tinypng是商业api，对图片压缩有限额
- github上的主流tinypng的插件并没有压缩项目中依赖的图，仅仅压缩了你项目中自己的图
- MclImage可以对一个Module中所有的依赖进行处理，包括jar中的图和aar中的图
- 采用的算法pngquant仅仅是MclImage的一个插件，可以随时替换算法，如果出现更加好用的算法，MclImage会及时更换。主流的tinypng插件仅仅适用于tinypng

MclImage插件的劣势:

- tinypng采用自研算法，压缩效率高于开源的pngquant算法，大约百分之8左右的压缩度

MclImage插件对全量依赖的压缩可以尽可能的减少着百分之8左右的压缩带来的劣势

3. 开启 shrinkResources 模式

```
buildTypes {  
    release {  
        minifyEnabled true  
        shrinkResources true  
    }  
}
```

minifyEnabled 这个是用来开启删除无用代码，比如没有引用到的代码

shrinkResources 用来开启删除无用资源，也就是没有被引用的文件（**经过实测是drawable,layout，实际并不是彻底删除，而是保留文件名，但是没有内容，等等**），但是因为需要知道是否被引用所以需要配合 **minifyEnabled** 使用，只有当两者都为true的时候才会起到真正的删除无效代码和无引用资源的目的

4. 删除string.xml 下面无用的文案

5. 使用webp压缩

右键 convert to webp , 然后转换

优点:压缩比优于tinypng

缺点: Android 设备对 webp 的支持存在兼容性问题，在 4.3 以上才完全支持

将 png/jpg 转换为 webp 格式只是减少文件大小。而 Bitmap 在 drawable 内存中占用大小的计算是根据 [长 * (设备 dpi / 资源目录对应的 dpi) * 宽 * (设备 dpi / 资源目录对应的 dpi) * 位图格式 = ?] 该公式计算的。固 webp 能减少内存占用不成立。

6. 开启代码混淆

混淆器的 **作用** 不仅仅是 **保护代码**，它也有 **精简编译后程序大小** 的作用，其 **通过缩短变量和函数名以及丢失部分无用信息等方式**，能使得应用包体积减小。

7. 第三方库优化, 只引入部分需要的代码, 比如有些库对各个功能进行剥离, 这时候我们只需要引入我们需要的功能即可,不必引入整个库. 如果引入的第三方库没有做剥离, 只能通过修改源码的方式, 提取需要的功能.
8. 去除不必要的so库, 如架构so库, 可以尝试动态下发,或者只支持一种架构

/home/rs/AndroidStudioProjects/HomeCamera_skin/app/src

D8 优化

D8 的 优化效果 总的来说可以归结为如下 四点：

- 1) 、 Dex的编译时间更短。
- 2) 、 .dex文件更小。
- 3) 、 D8 编译的 .dex 文件拥有更好的运行时性能。
- 4) 、 包含 Java 8 语言支持的处理。

开启 D8

在 **Android Studio 3.0** 需要主动在 **gradle.properties** 文件中新增:

```
android.enableD8 = true
```

Android Studio 3.1 或之后的版本 **D8** 将会被作为默认的 **Dex** 编译器。

屏幕刷新机制及优化

链接: <https://juejin.im/post/5d837cd1e51d4561cb5ddf66#heading-16>

蓝师傅卡顿优化: <https://www.jianshu.com/p/2de6d5fb0eab>

卡顿检测:

TraceView 工具

Systrace 工具

BlockCanary开源库

在 `requestLayout()` 方法中, 往消息队列插入了一个同步屏障, 这时候消息队列中的同步消息不会被处理, 而是优先处理异步消息。这里很好理解, UI相关的操作优先级最高, 比如消息队列有很多没处理完的任务, 这时候启动一个 Activity, 当然要优先处理Activity启动, 然后再去处理其他的消息

应用必须向底层请求vsync 信号,然后下一次vsync 信号来的时候就会通过JNI调用 `DisplayEventReceiver#dispatchVsync`方法,然后接下来才到应用绘制逻辑.

```
//在子类实现dispatchVsync 里面的onVsync 方法
private final class FrameDisplayEventReceiver extends DisplayEventReceiver
    implements Runnable {
```

```
private boolean mHavePendingVsync;
private long mTimestampNanos;
private int mFrame;

public FrameDisplayEventReceiver(Looper looper, int vsyncSource) {
    super(looper, vsyncSource);
}

@Override
public void onVsync(long timestampNanos, int builtInDisplayId, int frame) {

    if (builtInDisplayId != SurfaceControl.BUILT_IN_DISPLAY_ID_MAIN) {
        Log.d(TAG, "Received vsync from secondary display, but we don't support "
            + "this case yet. Choreographer needs a way to explicitly request"
            + "vsync for a specific display to ensure it doesn't lose track "
            + "of its scheduled vsync.");
        scheduleVsync();
        return;
    }

    long now = System.nanoTime();
    if (timestampNanos > now) {
        Log.w(TAG, "Frame time is " + ((timestampNanos - now) * 0.000001f)
            + " ms in the future! Check that graphics HAL is generating vsync"
            + "timestamps using the correct timebase.");
        timestampNanos = now;
    }

    if (mHavePendingVsync) {
        Log.w(TAG, "Already have a pending vsync event. There should only be "
            + "one at a time.");
    } else {
        mHavePendingVsync = true;
    }

    mTimestampNanos = timestampNanos;
    mFrame = frame;
    //如果Handler此时存在耗时操作,那么需要等耗时操作完成后,Looper才会轮询到下一条消息,run方法
    //才会调用,然后才会执行run方法里面的 doFrame 进行绘制
    Message msg = Message.obtain(mHandler, this);
    msg.setAsynchronous(true);
    mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS_PER_MS);
}

@Override
public void run() {
    mHavePendingVsync = false;
    //在这里执行绘制
    doFrame(mTimestampNanos, mFrame);
}
}
```

在doFrame中，计算收到的vsync信号到 doFrame被调用的时间差，vsync信号间隔是16毫秒一次，大于16毫秒就是掉帧了，如果超过了30帧（默认是30），就打印log提示开发者检查主线程是否有耗时操作。

然后调用ViewRootImpl下 TraversalRunnable 的 run方法

```
final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        doTraversal();
    }
}

void doTraversal() {
    if (mTraversalScheduled) {
        mTraversalScheduled = false;
        // 移除同步屏障
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

        if (mProfile) {
            Debug.startMethodTracing("ViewAncestor");
        }
        // 最后在这方法里面调用measure、layout、draw 方法进行绘制
        performTraversals();

        if (mProfile) {
            Debug.stopMethodTracing();
            mProfile = false;
        }
    }
}
```

应用需要主动请求vsync，vsync来的时候才会通过JNI通知到应用，然后才调用View的三个绘制方法。如果没有发起绘制请求，例如没有requestLayout，View的绘制方法是不会被调用的。ViewRootImpl里面的这个View其实是DecorView。

小结

View 的 requestLayout 会调到ViewRootImpl 的 requestLayout方法，然后通过 scheduleTraversals 方法向 Choreographer 提交一个绘制任务，然后再通过 DisplayEventReceiver 向底层请求vsync信号，当vsync信号来的时候，会通过JNI回调回来，通过Handler往主线程消息队列post一个异步任务，最终是 ViewRootImpl 去执行那个绘制任务，调用 performTraversals 方法，里面是View的三个方法的回调。

- 从源码角度分析了屏幕刷新机制，底层每间隔16毫秒会发出vsyn信号，应用界面要更新，必须先向底层请求 vsync信号，这样下一个16毫秒vsync信号来的时候，底层会通过JNI通知到应用，然后通过主线程Handler执行

View的绘制任务。所以两个地方会造成卡顿，一个是主线程在执行耗时操作导致View的绘制任务没有及时执行，还有一个是View绘制太久，可能是层级太多，或者里面绘制算法太复杂，导致没能在下一个vsync信号来临之前准备完数据，导致掉帧卡顿。

2. 介绍目前比较流行的几种卡顿监控方式，基于消息队列的代表 `BlockCanary` 原理，以及通过编译插桩的方式在每个方法入口和出口加入计算方法耗时的代码。

面试中应对卡顿问题，可以围绕**卡顿原理、屏幕刷新机制、卡顿监控这几个方面来回答**，当然，卡顿监控这一块，还可以通过TraceView、SysTrace等工具来找出卡顿代码。在BlockCanary出现之前，TraceView、Systrace是开发者必备的卡顿分析工具，而如今，能把BlockCanary原理讲清楚我认为就很不错了，而对于厂商做系统App开发维护的，不会轻易接入开源库，所以就有必要去了解TraceView、Systrace工具的使用。

卡顿检测方法

ViewRootImpl 是在onResume 才被创建的

启动优化

链接：<https://juejin.im/post/5d95f4a4f265da5b8f10714b>

刘洋巴金<https://juejin.im/post/5eccb949f265da770b40c648>

检测耗时方法：

1. 直接在代码方法的前后加入 `System.currentTimeMillis()` 算出消耗的时间。

缺点，侵入性较强，需要在每个方法都加入，测试完后还要删除。

2. 使用Android Studio 自带的 Profile 查看哪些方法比较耗时，找出耗时方法。

比较直观，可以直接看出什么方法比较耗时

wall time 就是执行这段代码的时间，但是如果线程卡住了，等待的时间也算在内。

cpu time (Thread time) 这是CPU具体花在线程上的时间，它的时间一定小于wall time的时间

如果wall time 时间很长，而cpu time 时间很短，说明需要开启线程了，因为大部分时间cpu都是空闲的，时间都花在了等待上面。

优化耗时方法：

- 开启线程,使用线程池进行优化，将耗时方法放到线程池开启线程进行初始化。

缺点，如果耗时方法之间存在依赖关系，则不好处理。

- 如果初始化的方法需要先后执行顺序(也就是依赖关系)，可以采用启动器来进行优化。

- 一些不用初始化那么快的方法，可以使用IdleHandler 来进行延迟初始化

- 在5.0 一下 可以对 MultiDex 进行优化

- 第三方库懒加载

很多第三方库都是在Application 进行初始化, 当在Application 初始化的库越多, 对冷启动的影响就越大

这时候我们可以考虑按需初始化，例如Glide，可以放在自己封装的图片加载类中，调用到再初始化，其它库也是同理，让Application变得更轻。

- WebView启动优化

1. WebView第一次创建比较耗时，可以预先创建WebView，提前将内核初始化
2. 使用WebView 缓冲池,用到WebView的地方都从缓冲池取,缓冲池中没有再创建,需要注意内存泄漏问题
3. 本地预置html 和 css, WebView创建的时候先预加载本地html, 之后通过js脚本填充内容部分

IdleHandler

```
//getMainLooper().myQueue()或者Looper.myQueue()
Looper.myQueue().addIdleHandler(new IdleHandler() {
    @Override
    public boolean queueIdle() {
        //你要处理的事情
        return false;
    }
});
```

IdleHandler 是Handler机制的一种，可以在Looper事件循环的过程中，**当出现空闲的时候**，允许我们执行任务的一种机制。

既然 **IdleHandler** 主要在 **MessageQueue** 出现空闲的时候被执行，那么何时出现空闲？

1. MessageQueue 为空,没有消息
2. MessageQueue 中最近需要处理的消息，是一个延迟的消息（when>currentTime），需要滞后处理

IdleHandler 有什么用？

1. IdleHandler 是 Handler 提供的一种在消息队列空闲时，执行任务的时机
2. 当 MessageQueue 当前没有立即需要处理的消息时，会执行 IdleHandler

Q：MessageQueue 提供了 add/remove IdleHandler 的方法，是否需要成对使用？

1. 不是必须
2. IdleHandler.queueIdle() 的返回值，可以移除加入 MessageQueue 的 IdleHandler

Q：当 mIdleHandlers 一直不为空时，为什么不会进入死循环？

1. 只有在 pendingIdleHandlerCount 为 -1 时，才会尝试执行 mIdleHandler
2. pendingIdleHandlerCount 在 next() 中初始时为 -1，执行一遍后被置为 0，所以不会重复执行

Q：是否可以将一些不重要的启动服务，搬到 IdleHandler 中去处理？

1. 不建议
2. IdleHandler 的处理时机不可控，如果 MessageQueue 一直有待处理的消息，那么 IdleHandler 的执行时机会很靠后

Q: IdleHandler 的 queueIdle() 运行在那个线程?

1. 陷阱问题, queueIdle() 运行的线程, 只和当前 MessageQueue 的 Looper 所在的线程有关
2. 子线程一样可以构造 Looper, 并添加 IdleHandler

作者: 承香墨影

链接: <https://juejin.im/post/5e4de2f2f265da572d12b78f>

思考:

请求过多的情况, 并发的情况,

如果空闲执行中执行的任务还必须有先后执行的顺序呢。比如A页面, B页面都把自己耗时的方法加入到了空闲执行队列里面, 但是要想执行B页面耗时方法, 必须得先执行页面A中的方法, 你该怎么做?

使用启动器

Bitmap 图片优化

<https://mp.weixin.qq.com/s/RTRkNXOzrtb0T7FuG-vIwA>

Bitmap占用内存 = 宽 * 高 * 一像素所占用字节内存

RGB_565 : R = 5, G = 6, B = 5, 一共16位

ARGB_8888 : A = 8, R = 8, G = 8, B = 8, 一共 32 位

8位 = 1 字节, 所以假如有一张 ARGB_8888 480x800 大小的图片, 内存计算如下:

$480 \times 800 \times 4 = 1536000 \text{B} / 1024 = 1500 \text{ KB} = 1.5 \text{M}$

屏幕适配

$\text{px} = \text{dp}(\text{dpi} / 160)$

1. 通过dp加上自适应布局

可以适配大部分的手机, 但是部分手机仍需要单独适配, 因为不是所以相同像素的手机 dpi 都会相同, 比如普通的1080P手机的DPI是480, Pixel 的手机只有420, 所以会导致两个手机上显示的布局可能有些差别.

通过dp直接适配, 我们只能让UI基本适配不同的手机,但是在设计图和UI代码之间的鸿沟, dp是无法解决的, 因为dp不是真实像素。而且, 设计稿的宽高往往和Android的手机真实宽高差别极大。比如, 设计稿的ImageView是128px128px, 当我们在编写layout文件的时候, 却不能直接写成128dp128dp。在把设计稿向UI代码转换的过程中, 我们需要耗费相当的精力去转换尺寸, 这会极大的降低我们的生产力, 拉低开发效率。

DPI: 像素密度,指在系统软件上指定的单位尺寸的像素数量, 它往往是写在系统出厂配置文件的一个固定值.

$\text{px} / \text{density} = \text{dp}$, $\text{DPI} / 160 = \text{density}$, $\text{px} / (\text{DPI} / 160) = \text{dp}$

屏幕的总 px 宽度 / density = 屏幕的总 dp 宽度

当前设备屏幕总宽度（单位为像素）/ 设计图总宽度（单位为 dp）= density

宽高限定符适配

<https://juejin.im/post/6844903621855805448>

- ▶  **values**
- ▶  **values-480x320**
- ▶  **values-800x480**
- ▶  **values-854x480**
- ▶  **values-960x540**
- ▶  **values-1024x600**
- ▶  **values-1024x768**
- ▶  **values-1184x720**
- ▶  **values-1196x720**
- ▶  **values-1280x720**
- ▶  **values-1280x800**
- ▶  **values-1812x1080**
- ▶  **values-1920x1080**

设定一个基准的分辨率，其他分辨率都根据这个基准分辨率来计算，在不同的尺寸文件夹内部，根据该尺寸编写对应的dimens文件。

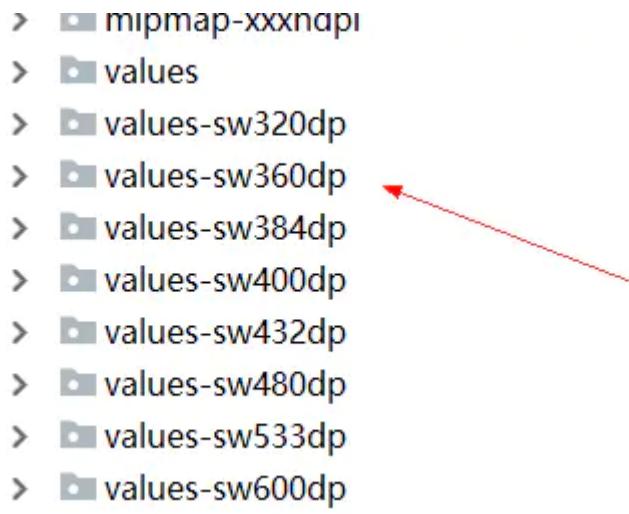
缺点：需要精准命中才能适配，比如1920x1080的手机就一定要找到1920x1080的限定符，否则就只能用统一的默认的dimens文件了。而使用默认的尺寸的话，UI就很可能变形，简单说，就是容错机制很差。

smallestWidth 方案

<https://juejin.im/post/5ba197e46fb9a05d0b142c62>

smallestWidth适配，或者叫sw限定符适配。指的是Android会识别屏幕可用高度和宽度的最小尺寸的dp值（其实质是手机的宽度值），然后根据识别到的结果去资源文件中寻找对应限定符的文件夹下的资源文件。

举个例子，小米5的dpi是480，横向像素是1080px，根据 $\text{px} = \text{dp}(\text{dpi}/160)$ ，横向的dp值是 $1080/(480/160)$ ，也就是360dp，系统就会去寻找是否存在value-sw360dp的文件夹以及对应的资源文件。



根据屏幕的宽度来匹配对应文件夹里面的dimens，然后在布局中引用dimens里面定义好的dp就行了，这样不同宽度的手机就会匹配不同的文件夹，对应里面dimens定义的dp完成适配。

优点：有很好的容错机制，如果找不到当前的dp，就会向下查找，找到离当前最近的dp

缺点：

- 维护麻烦
- 侵入行高，因为每个Layout文件中都存在大量的dimens的引用，修改起来工作量非常巨大
- 每个机型就要生成一个资源文件，生成的越多，占的体积就越大，没有覆盖的机型上会匹配更小宽度的文件夹，这样会出现一定的误差
- 不支持横竖屏切换的适配，如果想自动支持横竖屏切换时的适配，需要在生成一套资源文件
- 不能以高度为基准进行适配

头条屏幕适配方案

<https://mp.weixin.qq.com/s/d9QCoBP6kV9VSwvVldVVwA>

首先要了解以下计算公式

px = 像素, dpi = 手机的dpi

px = density * dp , density = dpi / 160

比如一台1080p,480dpi的手机，如果想要计算出多少dp才能铺满手机宽度

$1080 / (480 * 160) = 360$

头条的屏幕适配解决方案就是通过修改density 来适配不同的手机

```
private static float sNoncompatDensity;  
  
private static float sNoncompatScaleDensity;  
  
private static void setCustomDensity(@NonNull Activity activity, @NonNull final Application application) {
```

```

final DisplayMetrics appDisplayMetrics =
application.getResources().getDisplayMetrics();

if (sNoncompatDensity == 0) {
    sNoncompatDensity = appDisplayMetrics.density;
    sNoncompatScaleDensity = appDisplayMetrics.scaledDensity;
    application.registerComponentCallbacks(new ComponentCallbacks() {
        @Override
        public void onConfigurationChanged(Configuration newConfig) {
            if (newConfig != null && newConfig.fontScale > 0) {
                sNoncompatScaleDensity =
application.getResources().getDisplayMetrics().scaledDensity;
            }
        }
    });

    @Override
    public void onLowMemory() {

    }
});;
}

final float targetDensity = appDisplayMetrics.widthPixels / 411;
final float targetScaledDensity = targetDensity * (sNoncompatScaleDensity /
sNoncompatDensity);
final int targetDensityDpi = (int) (160 * targetDensity);

appDisplayMetrics.density = targetDensity;
appDisplayMetrics.scaledDensity = targetScaledDensity;
appDisplayMetrics.densityDpi = targetDensityDpi;

final DisplayMetrics activityDisplayMetrics =
activity.getResources().getDisplayMetrics();
activityDisplayMetrics.density = targetDensity;
activityDisplayMetrics.scaledDensity = targetScaledDensity;
activityDisplayMetrics.densityDpi = targetDensityDpi;

}

```

通过上面的代码把所有的设备density 都设置成一样, 以此来达到屏幕适配. 然后只需要在Activity的 `onCreate` 方法 在 `setContentView()` 之前调用即可.

优点: 入侵性低,

LeakCanary原理分析

蓝师傅LeakCanary分析：<https://www.jianshu.com/p/eb17a6bfdd9f>

```
@Override public void execute(Retryable retryable) {
    if (Looper.getMainLooper().getThread() == Thread.currentThread()) {
        //1 .如果是主线程直接执行
        waitForIdle(retryable, 0);
    } else {
        //2
        postwaitForIdle(retryable, 0);
    }
}

//1 主线程直接执行,然后开启一个延迟执行 Handler, IdleHandler
void waitForIdle(final Retryable retryable, final int failedAttempts) {
    // This needs to be called from the main thread.
    Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
        @Override public boolean queueIdle() {
            //3
            postToBackgroundWithDelay(retryable, failedAttempts);
            return false;
        }
    });
}

//2 如果不是主线程,就调用主线程的Handler进行post(),切换到主线程,然后在调用 waitForIdle
void postwaitForIdle(final Retryable retryable, final int failedAttempts) {
    //mainHandler 是主线程的Handler
    mainHandler.post(new Runnable() {
        @Override public void run() {
            waitForIdle(retryable, failedAttempts);
        }
    });
}

//3
void postToBackgroundWithDelay(final Retryable retryable, final int failedAttempts) {
    long exponentialBackoffFactor = (long) Math.min(Math.pow(2, failedAttempts),
maxBackoffFactor);
    //initialDelayMillis 是 5s
    long delayMillis = initialDelayMillis * exponentialBackoffFactor;
    backgroundHandler.postDelayed(new Runnable() {
        @Override public void run() {
            Retryable.Result result = retryable.run();
            if (result == RETRY) {
                postwaitForIdle(retryable, failedAttempts + 1);
            }
        }
    }, delayMillis);
}
```

leakCanary 如何判断内存泄露

```
public void watch(Object watchedReference, String referenceName) {
    if (this == DISABLED) {
        return;
    }
    checkNotNull(watchedReference, "watchedReference");
    checkNotNull(referenceName, "referenceName");
    final long watchStartNanoTime = System.nanoTime();
    String key = UUID.randomUUID().toString();
    //将key添加到set集合
    retainedKeys.add(key);
    //弱引用
    final KeyedWeakReference reference =
        new KeyedWeakReference(watchedReference, key, referenceName, queue);

    ensureGoneAsync(watchStartNanoTime, reference);
}
```

```
KeyedWeakReference(Object referent, String key, String name,
    ReferenceQueue<Object> referenceQueue) {
    //将引用队列和弱引用关联起来
    super(checkNotNull(referent, "referent"), checkNotNull(referenceQueue,
"referenceQueue"));
    this.key = checkNotNull(key, "key");
    this.name = checkNotNull(name, "name");
}
```

弱引用和引用队列搭配使用，如果弱引用持有的对象被回收，Java虚拟机就会把这个弱引用加入到与之关联的队列中，也就是说如果 `keyedWeakReference` 持有的Activity 对象被回收，该 `keyedWeakReference` 就会加入到引用队列`queue`中去。

```
Retryable.Result ensureGone(final KeyedWeakReference reference, final long
watchStartNanoTime) {
    long gcStartNanoTime = System.nanoTime();
    long watchDurationMs = NANOSECONDS.toMillis(gcStartNanoTime - watchStartNanoTime);
    //1
    removeWeaklyReachableReferences();

    if (debuggerControl.isDebuggerAttached()) {
        // The debugger can create false leaks.
        return RETRY;
    }
    //2
    if (gone(reference)) {
        return DONE;
    }
}
```

```

//3 手动调用gc
//这里并没有使用System.gc()方法进行回收, 因为system.gc()并不会每次都执行。而是从AOSP中拷贝一段GC回收的代码, 从而相比System.gc()更能够保证进行垃圾回收的工作。
gcTrigger.runGc();
//重复 1,2 步骤, 如果对象还是没有被回收, 说明内存泄漏了。
removeWeaklyReachableReferences();
if (!gone(reference)) {
    long startDumpHeap = System.nanoTime();
    long gcDurationMs = NANOSECONDS.toMillis(startDumpHeap - gcStartNanoTime);

    File heapDumpFile = heapDumper.dumpHeap();
    if (heapDumpFile == RETRY_LATER) {
        // Could not dump the heap.
        return RETRY;
    }
    long heapDumpDurationMs = NANOSECONDS.toMillis(System.nanoTime() - startDumpHeap);
    heapdumpListener.analyze(
        new HeapDump(heapDumpFile, reference.key, reference.name, excludedRefs,
watchDurationMs,
        gcDurationMs, heapDumpDurationMs));
}
return DONE;
}

//步骤1 遍历引用队列, 如果里面不为空, 说明activity被回收了。获取key, 在retainedKeys中移除对应的key(被回收则移除)
private void removeWeaklyReachableReferences() {

    KeyedWeakReference ref;
    while ((ref = (KeyedWeakReference) queue.poll()) != null) {
        retainedKeys.remove(ref.key);
    }
}

//步骤2 判断队列里面是否还有对应的key, 如果有说明没回收
private boolean gone(KeyedWeakReference reference) {
    return !retainedKeys.contains(reference.key);
}

```

总结

LeakCanary 通过监听Activity的生命周期, 在Activity onDestroy 的时候, 创建一个弱引用, key 跟当前 Activity绑定, 将key保存到set里面, 并且关联一个引用队列。LeakCanary 使用的是IdleHandler进行延迟加载, 会在主线程空闲5秒后, 开始检测内存是否泄漏。

1. 遍历引用队列, 判断是否还有该Activity的引用, 如果有, 则说明被回收了(因为被回收后会加入到引用队列中去), 移除队列里面对应的key

2. 判断队列里面是否有当前检测的Activity的key，如果有，说明Activity对象还没移除，可能此时已经被没有被引用了，不一定是内存泄露。
3. 手动调用GC，然后重复1,2 的操作，然后还没有被回收，说明内存泄漏了。

你知道 LeakCanary 中的Idle机制吗？

在Activity onDestroy的时候，LeakCanary 并没有马上去执行检测任务，而是将任务添加到消息队列的一个idle任务列表里，然后当Handler 在消息队列中获取不到消息，也就是主线程空闲的时候，会去idle任务列表里取任务出来执行。

为什么 LeakCanary 需要主动触发GC呢？

LeakCanary 监控泄漏利用了弱引用的特性，为Activity创建弱引用，当Activity对象变成弱可达时(没有强引用)，弱引用会被加入到引用队列中，通过在 Activity.onDestroy() 后连续触发两次GC，并检查引用队列，可以判定Activity是否发生了泄漏。但频繁的GC会造成用户可感知的卡顿

卡顿优化

卡顿检测

```
public class AppContext extends Application {

    @Override
    public void onCreate() {
        super.onCreate();

        Looper.getMainLooper().setMessageLogging(new Printer() {
            @Override
            public void println(String x) {
                Log.e("AppContext", x);
                //在这里判断是否卡顿
            }
        });
    }

    //loop会调用

    if (logging != null) {
        logging.println(">>>> Dispatching to " + msg.target + " " +
                       msg.callback + ": " + msg.what);
    }

    if (logging != null) {
        logging.println("<<<< Finished to " + msg.target + " " + msg.callback);
    }

    //通过这两个 一个开始一个结束，通过判断这个过程的时间判断是否卡顿
}
```

Looper轮循的时候，每次从消息队列取出一条消息，如果logging不为空，就会调用 logging.println，我们可以通过设置Printer，计算Looper两次获取消息的时间差，如果时间太长就说明Handler处理时间过长，直接把堆栈信息打印出来，就可以定位到耗时代码。不过println方法参数涉及到字符串拼接，考虑性能问题，所以这种方式只推荐在Debug模式下使用。

BlockCanary使用

BlockCannary 是一个卡顿检测框架

使用方法

1. 添加依赖方法

```
debugImplementation 'com.github.markzhai:blockcanary-android:1.5.0'  
//只在Debug 模式下使用
```

2. 在Application初始化

```
//注意这框架里面有将log 写入文件的操作,可能需要申请权限  
//第二个参数可以传入 自己定义的方法,继承自Block CanaryContext(),进行自定义设置  
BlockCanary.install(this, new Block CanaryContext()).start();
```

3. 然后就可以进行检测了，下面是测试

```
case R.id.glide:  
    Button button = findViewById(R.id.glide);  
    Log.d("liang", "onClick: test Block");  
    try {  
        //sleep两秒, 因为Block默认超过一秒就会进行检测  
        sleep(2000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    button.setText("test Block");
```

BlockCanary 常用可自定义选项,更多请看源码:

- provideQualifier 获取应用版本信息
- provideUid 获取用户uid
- provideNetworkType 获取网络类型
- provideMonitorDuration 设置监控时长
- provideBlockThreshold 设置监控卡顿阀值(默认1000ms)
- providePath 设置log保存地址
- displayNotification 设置是否显示在桌面展示

内存优化

常用工具

1. Memory Profiler

- 实时图表展示内存使用量
- 识别内存泄露,内存抖动等
- 提供捕获堆转储,强制GC以及根据内存分配的能力
- 优点:方便直观,线下使用

2. Memory Analyzer

- 强大的Java Heap分析工具,查找内存泄露及内存的占用
- 生成整体报告,分析问题等.建议线下深入使用

3. LeakCanary

- 自动内存泄漏检测神器。仅用于线下集成。
- 它的缺点比较明显, 虽然使用了idleHandler与多进程, 但是dumphprof的SuspendAll Thread的特性依然会导致应用卡顿。在三星等手机, 系统会缓存最后一个Activity, 此时应该采用更严格的检测模式。

内存抖动:

内存频繁分配和回收导致内存不稳定, 就会出现内存抖动, 导致页面卡顿

频繁创建对象会造成内存不断地攀升, 在刚回收了之后又迅速涨起来, 那么紧接着就是又一次的回收, 多次内存回收集中在短时间内爆发, 这就造成了比较大的界面卡顿的风险。

这个问题在 **Dalvik虚拟机** 上会 **更加明显**, 而 **ART虚拟机** 在 **内存管理跟回收策略** 上都做了 **大量优化**, **内存分配和GC效率相比提升了5~10倍**, 所以 **出现内存抖动的概率会小很多**。

解决方案:

1. 字符串拼接

- 使用StringBuilder 代替
- 初始化时设置容量, 减少StringBuilder 的扩容。

2. 资源复用

- 使用全局缓存池, 以重用频繁申请和释放的对象。
- 使用结束后, 需要手动释放对象池中的对象。

3. 减少不合理的对象创建

- ondraw、getView 中创建的对象尽量复用
- 避免在循环中不断创建局部变量

4. 使用合理的数据结构

- 使用sparseArray类、ArrayMap 来替代 HashMap

使用 IntDef和StringDef 替代枚举类型

使用枚举类型的dex size是普通常量定义的dex size的13倍以上, 同时, 运行时的内存分配, 一个enum值的声明会消耗至少20bytes。

枚举最大的优点是类型安全，但在Android平台上，枚举的内存开销是直接定义常量的三倍以上。所以Android提供了注解的方式检查类型安全。目前提供了int型和String型两种注解方式：IntDef和StringDef，用来提供编译期的类型检查。

优化图片内存的大小

线下内存检测，及时发现占用内存大的图片

内存溢出(OOM)：

手机出厂后，Java虚拟机会对单个应用进行最大内存分配，一旦应用超过这个最大值就会出现OOM，很多时候都是因为图片处理不当导致的。

图片资源Drawable放在合适密度的文件夹下，如果放在比你手机当前像素密度低的文件夹下，图片就会放大，像素点会变多，这样就会占更多的内存。

内存泄露

所谓内存泄露就是内存中存在已经没有用的对象

避免使用非静态内部类，定义内部类的时候，要么放在单独的类文件中，要么就是使用静态内部类，因为静态的内部类不会持有外部类的引用，所以不会导致外部类实例的内存泄露，当你要在静态内部类调用外部的Activity时，我们可以使用弱引用来处理。

在任何使用到Context的地方，都要多加注意，例如我们常见的Dialog，Menu，悬浮窗，这些控件都需要传入Context作为参数的，如果要使用Activity作为Context参数，那么一定要**保证控件的生命周期跟Activity的生命周期同步**。窗体泄漏也是内存泄漏的一种，就是我们常见的leak window，这种错误就是依赖Activity的控件生命周期跟Activity不同步造成的。一般来说，**对于非控件类型的对象需要Context参数，最好优先考虑全局ApplicationContext，来避免内存泄漏。**

<https://juejin.im/post/5a692377518825734e3e71ab#heading-6>

造成Handler内存泄露有两个关键条件：

1. 存在“未处理、正在处理的消息 -> Handler实例 -> 外部类”的引用关系
2. Handler的生命周期 **大于** 外部生命周期

即Handler消息队列 还有未处理的消息、正在处理消息 而外部类需要销毁

解决方案1：静态内部类 + 弱引用

- 原理 静态内部类 不默认持有外部类的引用，从而使得“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”的引用关系 的引用关系 不复存在。
- 具体方案 将 Handler 的子类设置成 静态内部类
 - 同时，还可加上 使用WeakReference弱引用持有Activity实例

- 原因：弱引用的对象拥有短暂的生命周期。在垃圾回收器线程扫描时，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存

解决方案2：当外部类结束生命周期时，清空Handler内消息队列

- 原理 不仅使得“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”的引用关系 不复存在，同时使得 Handler 的生命周期（即 消息存在的时期）与 外部类的生命周期 同步
- 具体方案 当 外部类（此处以 Activity 为例）结束生命周期时（此时系统会调用 `onDestroy()`），清除 Handler 消息队列里的所有消息（调用 `removeCallbacksAndMessages(null)`）

使用属性动画的时候，在销毁的时候记得调用cancle 取消属性动画，因为属性动画是一个耗时操作，如果页面销毁了而动画没有cancle，虽然我们看不见这个动画，但这个动画还在播放，动画引用的控件引用到Activity，这就会导致Activity 无法释放造成内存泄漏。

常见内存泄漏场景：

资源对象未关闭：

对于资源性对象不再使用时，应该立即调用它的close()函数，将其关闭，然后再置为null。例如Bitmap等资源未关闭会造成内存泄漏，此时我们应该在Activity销毁时及时关闭。

注册对象未注销：

例如BroadcastReceiver、EventBus未注销造成的内存泄漏，我们应该在Activity销毁时及时注销。

类的静态变量持有大数据对象

尽量避免使用静态变量存储数据，特别是大数据对象，建议使用数据库存储。

单例造成的内存泄漏

优先使用Application的Context，如需使用Activity的Context，可以在传入Context时使用弱引用进行封装，然后，在使用到的地方从弱引用中获取Context，如果获取不到，则直接return即可。

非静态内部类的静态实例

该实例的生命周期和应用一样长，这就导致该静态实例一直持有该Activity的引用，Activity的内存资源不能正常回收。此时，我们可以将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，尽量使用Application Context，如果需要使用Activity Context，就记得用完后置空让GC可以回收，否则还是会内存泄漏。

Handler临时性内存泄漏

Message发出之后存储在MessageQueue中，在Message中存在一个target，它是Handler的一个引用，Message在Queue中存在的时问过长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。并且消息队列是在一个Looper线程中不断地轮询处理消息，当这个Activity退出时，消息队列中还有未处理的消息或者正在处理的消息，并且消息队列中的Message持有Handler实例的引用，Handler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏。解决方案如下所示：

- 1、使用一个静态Handler内部类，然后对Handler持有的对象（一般是Activity）使用弱引用，这样在回收时，也可以回收Handler持有的对象。

- 2、在Activity的Destroy或者Stop时，应该移除消息队列中的消息，避免Looper线程的消息队列中有待处理的消息需要处理。

需要注意的是，AsyncTask内部也是Handler机制，同样存在内存泄漏风险，但其一般是临时性的。对于类似 AsyncTask或是线程造成的内存泄漏，我们也可以将AsyncTask和Runnable类独立出来或者使用静态内部类。

容器中的对象没清理造成的内存泄漏

在退出程序之前，将集合里的东西clear，然后置为null，再退出程序

WebView

WebView都存在内存泄漏的问题，在应用中只要使用一次WebView，内存就不会被释放掉。我们可以为WebView开启一个独立的进程，使用AIDL与应用的主进程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，达到正常释放内存的目的。

使用ListView时造成的内存泄漏

在构造Adapter时，使用缓存的convertView。

检测工具

Android Bitmap 内存分配的变化

Android 3.0 ~ Android 7.0

将 **Bitmap** 对象 和 像素数据 统一放到 **Java Heap** 中，即使不调用 recycle，Bitmap 像素数据也会随着对象一起被回收。

但是，Bitmap 全部放在 Java Heap 中的缺点很明显，大致有如下两点：

- 1) 、**Bitmap**是内存消耗的大户，而 **Max Java Heap** 一般限制为 **256、512MB**，**Bitmap** 过大过多容易导致 **OOM**。
- 2) 、容易引起大量 **GC**，没有充分利用系统的可用内存。

Android 8.0 及以后

- 使用了能够辅助回收 Native 内存的 **NativeAllocationRegistry**，以实现将像素数据放到 Native 内存中，并且可以和 **Bitmap** 对象一起快速释放，最后，在 **GC** 的时候还可以考虑到这些 **Bitmap** 内存以防止被滥用。
- Android 8.0 为了解决图片内存占用过多和图像绘制效率过慢的问题新增了 **硬件位图 Hardware Bitmap**。

Android 虚拟机

方法区

这块区域对应**持久代（Permanent Generation）**，一般来说，方法区上执行GC的情况很少，因此方法区被称为持久代的原因之一，但这并不代表方法区上完全没有GC，其上的GC主要针对常量池的回收和已加载类的卸载。

堆区

GC最频繁的地方，堆区由所有线程共享，在虚拟机启动时创建，堆区主要用于存放对象实例及数组，所有new出来的对象都存储在该区域

虚拟机栈

虚拟机栈占用的操作系统内存，每个线程一个虚拟机栈，它的线程是私有的，生命周期和线程一样，每个方法被执行时产生一个**栈帧（Stack Frame）**，栈帧用于存储局部变量表、动态链接、操作数和方法出口等信息，当方法被调用时，栈帧入栈，当方法调用结束时，栈帧出栈。

本地方法栈

本地方法栈用于支持native方法的执行，存储了每个native方法的执行状态。本地方法栈和虚拟机栈他们的运行机制一致，唯一的区别是，**虚拟机栈执行Java方法，本地方法栈执行native方法**。在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将虚拟机栈和本地方法栈一起使用

本地方法栈，程序计数器，虚拟机栈不需要进行垃圾回收，因为他们的生命周期是线程同步的，随着线程的销毁，他们占用的内存会自动释放。所以，**只有方法区和堆区需要进行垃圾回收**，回收那么不存在任何引用的对象。

G1算法

GC机制

链接：[Android GC](#)

[深入理解JVM](#)

标记回收算法（Mark and Sweep GC）

从“GC Roots”集合开始，将内存整个遍历一次，保留所有可以被GC Roots直接或间接引用到的对象，而剩下的对象都当作垃圾对待并回收，这个算法需要中断进程内其它组件的执行并且可能产生内存碎片。

清除阶段清理的是没有被引用的对象，存活的对象被保留。

标记-清除动作不需要移动对象，且仅对不存活的对象进行清理，在空间中**存活对象较多的时候，效率较高**，但由于只是清除，没有重新整理，因此**会造成内存碎片**。

复制算法（Copying）

将现有的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

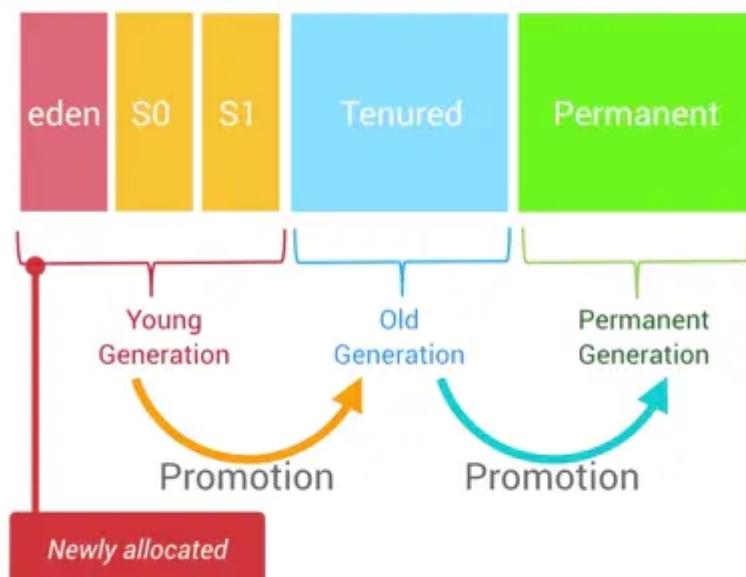
当存活的对象较少时，复制算法会比较高效（新生代的Eden区就是采用这种算法，因为新生代存活的比较少，用复制算法效率高），其带来的成本是需要一块额外的空闲空间和对象的移动。

标记-压缩算法 (Mark-Compact) 先需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。但由于需要进行移动，因此成本也增加了。（该算法适用于旧生代，因为存活的对象比较多）

分代 将所有的新建对象都放入称为年轻代的内存区域，年轻代的特点是对象会很快回收，因此，在年轻代就选择效率较高的复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老生代的内存空间。**对于新生代适用于复制算法，而对于老年代则采取标记-压缩算法。**

三种算法速度对比：

	mark-sweep	Mark-Compact	Copying
速度	中等	最慢	最快
空间开销	少，会堆积碎片	少，不会堆积碎片	通常需要活对象的两倍空间大小，不堆积碎片
移动对象	否	是	是



内存对象的处理过程小结

1. 对象创建后在Eden区。
2. 执行GC后，如果对象仍然存活，则复制到S0区。
3. 当S0区满时，该区域存活对象将复制到S1区，然后S0清空，接下来S0和S1角色互换。
4. 当第3步达到一定次数（系统版本不同会有差异）后，存活对象将被复制到Old Generation。

5. 当这个对象在Old Generation区域停留的时间达到一定程度时，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。

系统在Young Generation、Old Generation上采用不同的回收机制。每一个Generation的内存区域都有固定的大\小。随着新的对象陆续被分配到此区域，当对象总的大小临近这一级别内存区域的阈值时，会触发GC操作，以便腾出空间来存放其他新的对象。

此外，执行GC占用的时间与Generation和Generation中的对象数量有关，如下所示：

- Young Generation < Old Generation < Permanent Generation
- Generation中的对象数量与执行时间成反比。

5、Young Generation GC

由于其对象存活时间短，因此基于**Copying算法**（扫描出存活的对象，并复制到一块新的完全未使用的控件中）来回收。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在Young Generation区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。

6、Old Generation GC

由于其对象存活时间较长，比较稳定，因此采用**Mark（标记）算法**（扫描出存活的对象，然后再回收未被标记的对象，回收后对空出的空间要么合并，要么标记出来便于下次分配，以减少内存碎片带来的效率损耗）来回收。

Dalvik 与 ART 区别

- 1) 、Dalvik 仅固定一种回收算法。
- 2) 、ART 回收算法可运行期选择。
- 3) 、ART 具备内存整理能力，减少内存空洞。

ART优点：

1. 系统性能的显著提升
2. 应用启动更快、运行更快、体验更流畅、触感反馈更及时
3. 更长的电池续航能力
4. 支持更低的硬件

ART缺点：

1. 更大的存储空间占用，可能会增加10%-20%
2. 更长的应用安装时间

GC的类型：

在Android系统中，GC有三种类型：

- **kGcCauseForAlloc**: 分配内存不够引起的GC，会Stop World。由于是并发GC，其它线程都会停止，直到GC完成。
- **kGcCauseBackground**: 内存达到一定阈值触发的GC，由于是一个后台GC，所以不会引起Stop World。
- **kGcCauseExplicit**: 显示调用时进行的GC，当ART打开这个选项时，使用System.gc时会进行GC。

在Dalvik虚拟机下，GC的操作都是**并发的**，也就意味着每次触发GC都会导致其它线程暂停工作（包括UI线程）。而在ART模式下，GC时不像Dalvik仅有一种回收算法，ART在不同的情况下会选择不同的回收算法，比如**Alloc内存不够时会采用非并发GC，但在Alloc后，发现内存达到一定阈值时又会触发并发GC**。所以在ART模式下，并不是所有的GC都是非并发的。

总体来看，在GC方面，与Dalvik相比，ART更为高效，不仅仅是GC的效率，大大地缩短了Pause时间，而且在内存分配上对大内存分配单独的区域，还能有算法在后台做内存整理，减少内存碎片。因此，在ART虚拟机下，可以避免较多的类似GC导致的卡顿问题。

链接：<https://juejin.im/post/6844904096541966350>

Groovy

如果不声明public/private 等访问权限的话，**Groovy中的类及其变量默认都是public**

<https://juejin.im/post/5e97ac34f265da47aa3f6dca#heading-13>

变量定义

- 强类型定义方式：**groovy 像 Java 一样，可以进行强类型的定义，比如上面直接定义的 int 类型的 x，这种方式就称为强类型定义方式，即在声明变量的时候定义它的类型。
- 弱类型定义方式：**不需要像强类型定义方式一样需要提前指定类型，而是通过 def 关键字来定义我们任何的变量，因为编译器会根据值的类型来为它进行自动的赋值。

```
int a1 = 1231231 //强类型
def a2 = 1231231//弱类型，编译器自动推断类型
```

如果你这个类或变量要用于其它模块的，建议不要使用 def，还是应该使用 Java 中的那种强类型定义方式，因为使用强类型的定义方式，它不能动态转换为其它类型，它能够保证外界传递进来的值一定是正确的。

Groovy 中常用的三种字符串定义方式

- 单引号 '' 定义的字符串
- 双引号 "" 定义的字符串
- 三引号 """ 定义的字符串

它们的类型都都是'java.lang.String'

单引号：当我们编写的单引号字符串中有转义字符的时候，需要添加 ''

双引号：可扩展变量

```
def student = "this is groovy ${test}"
```

闭包 如果闭包没定义参数的话，则隐含有一个参数，这个参数名字叫 it，和 this 的作用类似。it 代表闭包的参数。
表示闭包中没有参数的示例代码：

```
def noParamClosure = { -> true} // 没有圆括号
```

this 与 owner、delegate

```
def nestClouser = {
    println "this :" + this //代表闭包定义处的类
    println "owner" + owner //代表闭包定义处的类或对象
    println "delegate :" + delegate // 代表任意对象， 默认跟owner一致
}
//三个都输出 : nestClouser

def nestClouser = {
    def innnerClouser = {

        println "this :" + this //代表闭包定义处的类
        println "owner" + owner //代表闭包定义处的类或对象
        println "delegate :" + delegate // 代表任意对象， 默认跟owner一致
    }
}

//this 输出: nestClouser, owner 和 delegate 输出: innnerClouser
```

如果我们直接在类、方法、变量中定义一个闭包，那么这三种关键变量的值都是一样的，但是，如果我们在闭包中又嵌套了一个闭包，那么，this 与 owner、delegate 的值就不再一样了。换言之，**this** 还会指向我们闭包定义处的类或者实例本身，而 **owner**、**delegate** 则会指向离它最近的那个闭包对象。

delegate 的值是可以修改的，并且仅仅当我们修改 delegate 的值时，delegate 的值才会与 owner 的值不一样。

Groovy 常用的数据结构

- 数组
- List

即链表，其底层对应 Java 中的 List 接口，一般用 ArrayList 作为真正的实现类，List 变量由[] 定义，其元素可以是任何对象。

链表中的元素可以通过索引存取，而且 不用担心索引越界。如果索引超过当前链表长度，List 会自动往该索引添加元素。

- Map
- Range

List

```
// 添加
list.add(6)
list.leftShift(7)
list << 8
// 删除
list.remove(7)
list.removeAt(7)
list.removeElement(6)
```

```

list.removeAll { return it % 2 == 0 }
// 查找
int result = findList.find { return it % 2 == 0 }
def result2 = findList.findAll { return it % 2 != 0 }
def result3 = findList.any { return it % 2 != 0 }
def result4 = findList.every { return it % 2 == 0 }
// 最小值、最大值
list.min()
list.max{return Math.abs(it)}

```

Map

冒号左边是 key，右边是 Value。key 必须是字符串，value 可以是任何对象。另外，key 可以用 " 或 "" 包起来，也可以不用引号包起来

```

//存取
aMap.keyName
aMap['keyName']
aMap.anotherkey = "i am map"
aMap.anotherkey = [a: 1, b: 2]

//each 方法
//如果我们传递的闭包是一个参数，那么它就把 entry 作为参数。如果我们传递
//的闭包是 2 个参数，那么它就把 key 和 value 作为参数。
def result = ""
[a:1, b:2].each {key, value ->
    result += "$key$value"
}

[a:1, b:2].each {entry ->
    result += entry
}

//eachWithIndex 方法
def result = ""
//三个参数， 传递 键 值和索引
[a:1, b:3].eachWithIndex { key, value, index -> result += "$index($key$value)" }
assert result == "0(a1)1(b3)"
//两个参数 传递Map.Entry
[a:1, b:3].eachWithIndex { entry, index -> result += "$index($entry)" }
assert result == "0(a1)1(b3)"

//groupBy 方法
def group = students.groupBy { def student ->
    return student.value.score >= 60 ? '及格' : '不及格'
}

```

Range表示范围，它其实是 List 的一种拓展。其由 begin 值 + 两个点 + end 值表示。如果不想包含最后一个元素，则 begin 值 + 两个点 + < + end 表示。我们可以通过 aRange.from 与 aRange.to 来获对应的边界元素。

allprojects

表示 用于配置当前 project 及其下的每一个子 project

```
allprojects {  
    repositories {  
        google()  
        jcenter()  
        mavenCentral()  
        maven{  
            url "https://jitpack.io"  
        }  
        maven {url "https://plugins.gradle.org/m2/"}
```

subprojects

subprojects 可以 统一配置当前 project 下的所有子 project

```
subprojects {  
    if (project.plugins.hasPlugin("com.android.library")) {  
        apply from: '../publishToMaven.gradle'  
    }  
}
```

在上述示例代码中, 我们会先判断当前 project 旗下的子 project 是不是库, 如果是库才有必要引入 publishToMaven 脚本。

路径获取

- getRootDir()
- getProjectDir()
- getBuildDir()

SourceSet

SourceSet 主要是 用来设置我们项目中源码或资源的位置的, 目前它最常见的两个使用案例就是如下 两类:

- 修改so库存放位置
- 资源文件分包存放

```
android {  
    ...  
    sourceSets {  
        main {  
            // 修改 so 库存放位置  
            jniLibs.srcDirs = ["libs"]  
        }  
    }  
}  
  
// 资源文件分包存放
```

```

    android {
        sourceSets {
            main {
                res.srcDirs = [
                    "src/main/res",
                    "src/main/res-play",
                    "src/main/res-shop"
                ]
            }
        }
    }
}

```

网络

<https://juejin.im/post/5eba5a39e51d454de64e49b1#heading-151>

OSI 七层模型

OSI	功能
应用层	为计算机用户提供接口和服务。
表示层	数据处理：编解码、加解密等等。
会话层	管理（建立、维护、重连）通信会话
传输层	管理端到端的通信连接
网络层	数据路由：决定数据在网络中的路径。
数据链路层	管理相邻节点之间的数据通信
物理层	数据通信的光电物理特性。

DHCP（动态主机设置协议）

网络管理员只需配置一段共享的 IP 地址，每一台新接入的机器都可以通过 DHCP 来这个共享的 IP 地址里面申请 IP 地址，就可以自动配置。等用完还回去其它机器也能使用

- DHCP 是一个局域网协议
- DHCP 是应用 UDP 协议的应用层协议

因为 UDP 是无连接的不可靠传输，所以接收方需要在发送方发送数据之前就启动，否则会接收不到数据，也就是说必须先运行 UDP Socket Receive 再运行 UDP Socket Send。

Socket 通信原理：<https://juejin.im/post/6844904022567026695>

Android 与服务器通信方式主要有两种

- Http 通信

- Socket通信

两者之间的差异:

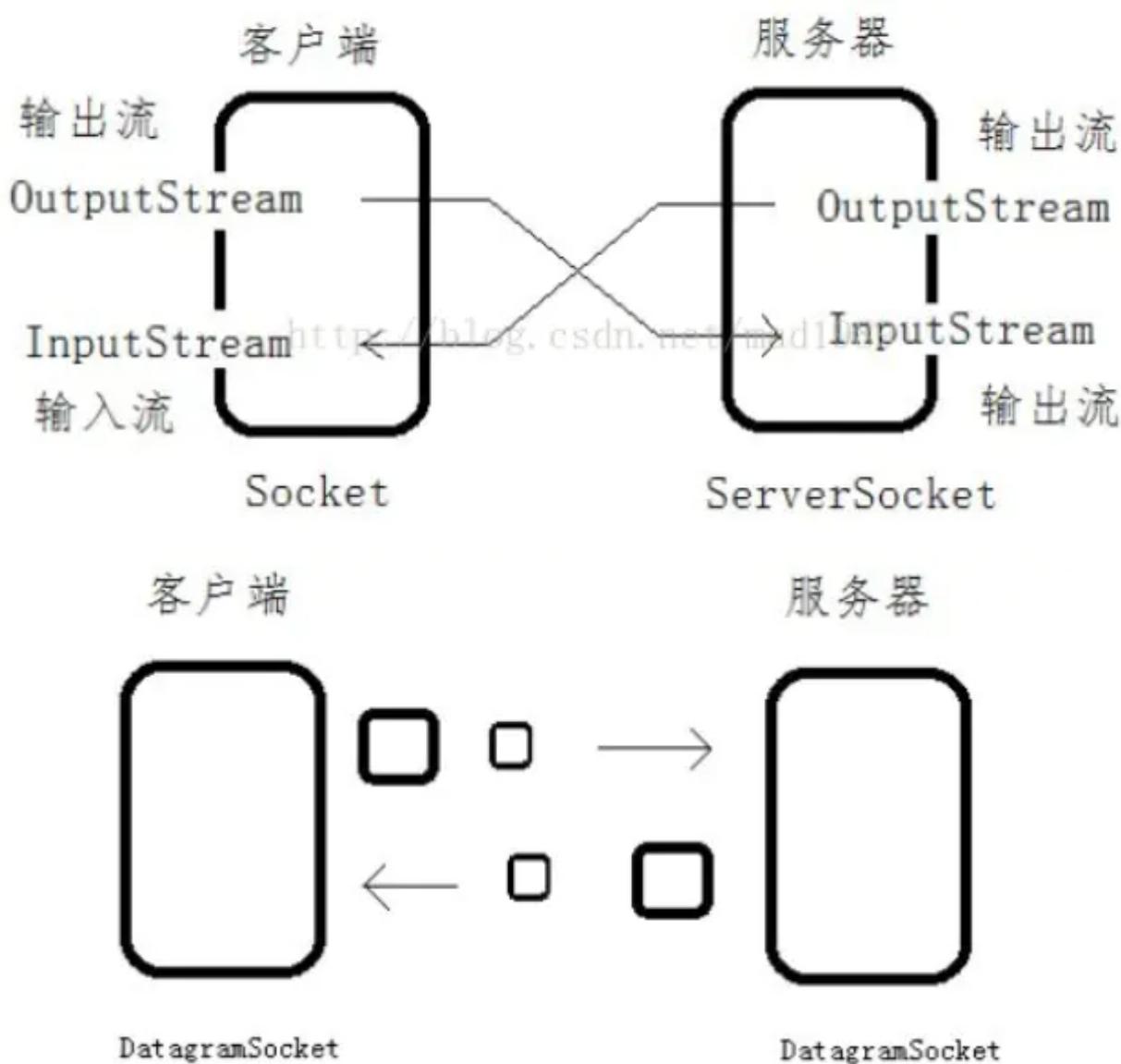
Http使用的是 "请求-响应方式", 即在请求时建立连接通道,当客户端向服务端发送请求后,服务端才能向客户端返回数据.

Socket 通信则是在双方建立连接后,可以直接进行数据的传输,在连接时可实现信息的主动推送,而不需要每次客户端向服务器发送请求.

在TCP/IP协议簇当中主要的Socket类型有:**流套接字(streamsocket)** 和 **数据报套接字(datagramsocket)**

流套接字将**TCP**作为其端对端协议, 提供一个可信赖的字节流服务

数据报套接字使用**UDP** 协议, 提供数据打包数据.



为什么有了HTTP 还需要 webSocket ?

HTTP 只能由客户端主动发起, 如果需要服务端主动通知业务,就需要轮询 , 轮询的效率低, 比较浪费资源. 为了解决 web 端即时通信的需求就出现了 webSocket

网络中的 Socket 并不是什么协议, 而是为了使用TCP, UDP而抽象出来的一层API, 它是位于应用层和传输层之间的一个抽象层.

Kotlin

可见性修饰符

在Kotlin 中有四种可见性修饰符

- private
- protected
- internal
- public

如果不指定任何修饰符默认public

如果声明成 internal 表示相同模块中可见

protected 表示私有和子类可见, 不适用于顶层声明

所有以未超出 Int 最大值的整型值初始化的变量都会推断为 Int 类型。如果初始值超过了其最大值, 那么推断为 Long 类型。如需显式指定 Long 型值, 请在该值后追加 L 后缀

当一个函数中只有一行代码的时候, Kotlin 允许我们不必编写函数体, 可以直接将唯一的一行代码写在函数定义的尾部, 中间用等号连接即可:

```
fun largerNumber(num1: Int, num2: Int) = max(num1, num2)
```

Kotlin 中的 if 语句相比Java有一个额外的功能, 它是可以有返回值的, 返回值就是 if 语句每一个条件中的最后一行代码的返回值。

```
fun largerNumber(num1: Int, num2: Int): Int {
    return if (num1 < num2) {
        num2
    } else {
        num1
    }
}

//里面if 跟 只有一行代码的作用的相同的 所以可以再简化
fun largerNumber(num1: Int, num2: Int) : Int = if (num1 < num2) {
    num2
} else {
    num1
}
//当然还可以更简化
fun largerNumber(num1: Int, num2: Int) : Int = if (num1 < num2) num1 else num2
```

when语句

```
fun getScore(name: String) = when (name) {  
    "zhangsan" -> 84  
    "lisi" -> 55  
    "laowang" -> 22  
    else -> 0  
}  
  
//when 语句还允许进行类型适配  
when (num) {  
    is Int -> println("number is Int")  
    is Double -> println("number is Double")  
    else -> ...  
}
```

when 还有一种不带参数的用法

```
fun getScore(name: String) = when {  
    name == "zhangsan" -> 22  
    name == "lisi" -> 15  
    else -> 0  
}  
  
//还可以这样  
// 判断名字 以 Tom 开头的人  
fun getScore(name: String) = when {  
    name.startsWith("Tom") -> 99  
    name == "zhangsan" -> 22  
    name == "lisi" -> 15  
    else -> 0  
}
```

循环语句

```
// 闭区间 0..10    输出 0 1 2 3 4 5 6 7 8 9 10  
for (i in 0..10) {  
    println(i)  
}  
  
//单端闭区间 0 until 10   输出 0 1 2 3 4 5 6 7 8 9  
for (i in 0 until 10) {  
    println(i)  
}
```

```
//跳过元素
for (i in 0 until 10 step 2) {
    println(i)
    // 输出 0 2 4 6 8
}

//降序
for (i in 0 until 10 downTo 1) {
    println(i)
    // 输出 0 2 4 6 8
}
```

在 Kotlin 中任何一个**非抽象类默认都是不可以被继承的**，相当于 Java 类中给类声明了 **final** 关键字。如果想要继承，在类前面加上**open** 关键字就可以了。

协程

协程是轻量级的线程，基于线程池 API 实现。

线程的调度是基于 CPU 算法和优先级，还是要跟底层打交道的，协程是完全由应用程序来调用的，但是他还是要基于线程来运行的。他比线程更轻量，开销很小。

- 协程就是切线程；
- 挂起就是可以自动切回来的切线程；
- 挂起的非阻塞式指的是它能用看起来阻塞的代码写出非阻塞的操作，就这么简单。

阻塞式和非阻塞式

- 阻塞不阻塞，都是针对单线程讲的，一旦切了线程，肯定是非阻塞的

协程只是在写法上「看起来阻塞」，其实是「非阻塞」的，因为在协程里面它做了很多工作，其中有一个就是帮我们切线程。

协程就是基于线程来实现的一种更上层的工具 API，类似于 Java 自带的 Executor 系列 API 或者 Android 的 Handler 系列 API。在设计思想上是一个**基于线程的上层框架**，协程是为了解决并发问题，让**协作式多任务**实现起来更加方便。

什么时候使用协程？

协程中「挂起」的对象到底是什么？挂起线程，还是挂起函数？都不对，我们挂起的对象是**协程**

那此时又是从哪里挂起？**从当前线程挂起**。换句话说，就是这个协程从正在执行它的线程上脱离（是脱离，不是暂停）

协程的代码块中，线程执行到了 **suspend** 函数这里的时候，就暂时不再执行剩余的协程代码，跳出协程的代码块。

协程在执行到有 **suspend** 标记的函数的时候，会被 **suspend** 也就是被挂起，而所谓的被挂起，就是切个线程；不过区别在于，**挂起函数在执行完成之后，协程会重新切回它原先的线程**。

再简单来讲，在 Kotlin 中所谓的挂起，就是一个稍后会被自动切回来的线程调度操作。

`suspend` 函数只能在协程里或者另一个 `suspend` 函数里被调用，这是为了要让协程能够在 `suspend` 函数切换线程之后还能切回来。

相较于RxJava有什么优势？

用看起来同步的方式写出异步的代码

`suspend` 关键字并不是真正实现挂起，而是一个提醒作用，提醒使用者在协程里调用。

它仅仅起着提醒的作用，比如，当我们的函数中没有需要挂起的操作的时候，编译器会给我们提醒 Redundant suspend modifier，意思是当前的 `suspend` 是没有必要的，可以把它删除

什么时候需要自定义 `suspend` 函数？

- 如果你的某个函数比较耗时，也就是要等的操作，比如文件读写、网络交互、图片的模糊处理等
- 需要等待的情况，比如5秒钟后再做这个操作

总结：

- 协程就是切线程
- 挂起就是可以自动切回来的线程
- 挂起的非阻塞式指的是它能用看起来阻塞的代码，写出非阻塞的操作

可以理解为轻量级的线程

挂起函数：如果某个函数比较耗时，那就把它写成挂起函数

平时使用的线程需要**依靠操作系统的调度**才能实现不同线程之间的切换

而使用协程却可以仅在**编程语言的层面**就能实现不同协程之间的切换

协程允许我们在单线程模式下模拟多线程的编程效果，代码执行时的挂起与恢复完全是由编程语言来控制的，和操作系统无关。

runBlocking：会创建一个协程的作用域，但是它可以保证在协程作用域内所有代码和子协程没有全部执行之前一直阻塞当前线程。

coroutineScope 函数和 **runBlocking** 函数相似，也是保证其作用域所有的代码和子线程。**coroutineScope** 函数只会阻塞当前协程，既不影响其他协程，也不影响任何线程，因此，不会造成任何性能上的问题。

而 **runBlocking** 函数会阻塞当前线程，如果你恰好又在主线程当中调用它的话，那有可能会导致界面卡死的情况。所以只建议在测试环境下使用。

GlobalScope.launch 由于每次创建都是顶层协程，一般也不太建议使用，除非你非常明确要创建顶级协程。

launch 函数只能用于执行一段逻辑，却不能获取执行结果，因为它返回值永远是一个 **Job** 对象。

async 函数必须在协程作用域当中才能调用，它会创建一个新的子协程并返回一个 **Deferred** 对象，如果我们想要获取 **asunc** 函数代码块的执行结果。只需要调用 **Deferred** 对象的 **await()** 方法即可。

Dispatchers.Default：表示会使用一种默认**低并发的线程策略**，当你要执行的代码属于计算密集型任务时，开启过高的并发反而可能会影响任务的运行效率，此时就可以使用该参数。

Dispatchers.IO 表示会使用一种**较高并发的线程策略**，当你要执行的代码大多时间是在阻塞和等待中，比如执行网络请求的时候，为了能够支持更高的并发数量，此时就可以使用该参数。

Dispatchers.Main 则表示不会开启子线程，而是在Android主线程中执行代码，但这个值只能在Android项目中使用，纯kotlin程序使用这种类型的参数会出现错误。

Delay() 函数

delay() 函数可以让当前协程延迟指定时间后再运行，但它和 Thread.sleep() 方法不同。delay() 函数是一个**非阻塞式**的挂起函数，只会挂起当前的协程，并不会影响其他协程的运行。而**Thread.sleep()**方法会阻塞当前的线程，这样运行在该线程下的所有协程都会被阻塞。注意，delay() 函数只能在协程的作用域或其他挂起函数中调用。

runBlocking()函数

```
fun main () {
    runBlocking {
        println("xxxxxxxx")
        delay(1500)
        println("cccccccc")
    }
}
```

runBlocking 函数同样会创建一个协程的作用域，但是它可以保证在协程作用域内的所有代码和子协程没有全部执行之前一直阻塞当前线程。需要注意的是，**runBlocking** 函数通常只应该在测试环境中使用。因为**runBlocking** 会阻塞当前线程，如果你刚好在主线程中调用它，那么就有可能导致界面卡死。

launch() 函数

```
fun main() {
    runBlocking {
        launch {
            delay(1000)
            println("launch1")
        }
        launch {
            delay(1000)
            println("launch2")
        }
    }
}

runBlocking() {
    var test = "1234"
    val job = launch {
        delay(3000)
        test = "4321"
    }
    println(test)
    job.join()
    println(test)
}
```

```
//输出  
// 1234  
// 4321  
  
//调用 join() 后将会挂起当前协程，直到子协程执行完毕或者取消
```

launch 和 GlobalScope.launch 不同，首先它必须在协程的作用域才能调用，其次它会在当前线程的作用域下创建子协程。子协程的特点是如果外层作用域的协程结束了，该作用域下的所有子协程也会一同结束，这样也就不用担心内存泄露的问题了。

launch() 函数的返回值是 Job，我们可以通过 Job.cancel() 来取消当前执行的协程任务。

suspend() 函数

suspend() 关键字只能将一个函数声明成挂起函数，是无法给它提供协程作用域的。

```
suspend fun printDot() {  
    println("")  
    delay(10000)  
}
```

coroutineScope() 函数

coroutineScope 函数也是一个挂起函数，因此可以在任何其他挂起函数中调用。它的特点是会继承外部协程作用域，并创建一个子作用域，借助这个特性，我们可以给任意挂起函数提供协程作用域。

```
suspend fun printDot() = coroutineScope {  
    //因为这里有作用域，所以可以调用launch  
    launch {  
        println("")  
        delay(1000)  
    }  
}
```

coroutineScope 函数和 runBlocking 函数有点类似，它可以保证其作用域内的所有代码和子协程在全部执行之前，会一直阻塞当前协程。但是 coroutineScope 函数只会阻塞当前协程，既不影响其他协程，也不影响其他线程，因此是不会造成任何性能上的问题。

async() 函数

async 函数必须在协程作用域当中才能调用，它会创建一个新的子协程并返回一个 Deferred 对象，如果我们想要获取 async 函数代码块的执行结果，只需要调用 Deferred 对象的 await() 方法即可。

```
fun main() {
    runBlocking {
        val result = async {
            5+5
        }.await()
        println(result)
    }
}
```

在调用async函数之后，代码块中的代码就会立刻开始执行，当调用await()方法时，如果代码块中的代码还没执行完，那么await()方法就会将当前协程阻塞住，直到可以获得async函数的执行结果。也可以理解为一个带返回值的launch()函数。

为了更高的效率，我们通常在需要async函数执行结果时才调用await()方法。

```
fun main() {
    runBlocking {
        val deferred1 = async {
            delay(1000)
            5+5
        }
        val deferred2 = async {
            delay(1000)
            6+6
        }
        //需要的时候才调用，而不是直接在方法后面调用。
        println("result is ${deferred1.await() + deferred2.await()}")
    }
}
```

withContext() 函数

调用withContext()函数后，会立即执行代码块中的代码，同时将当前协程阻塞住。当代码块执行完毕之后，会将最后一行的执行结果作为withContext()函数的返回值返回。相当于async{...}.await()方法。唯一不同的是withContext()函数强制要求我们指定一个线程参数。

有三种值可以选：

- Dispatchers.Default

默认低并发线程策略，当你要执行的代码属于计算密集型任务时，开启过高的并发反而可能影响任务的运行效率，此时就会使用该值。

- Dispatchers.IO

表示会使用一种较高并发的线程策略，当你要执行的代码大多时间都在阻塞和等待中，比如执行网络请求时，为了能够获得更高的并发数量，此时就可以使用该值

- Dispatchers.Main

表示不会开启子线程，而是在Android主线程中执行代码，这个值只能在Android项目中使用，纯Kotlin程序使用这种类型的线程参数会出现错误。

```
fun main() {
    runBlocking {
        val result = withContext(Dispatchers.Default) {
            5+5
        }
        println(result)
    }
}
```

suspendCoroutine()函数

`suspendCoroutine()` 函数必须在协程作用域或挂起函数中才能调用，它接收一个Lambda 表达式参数，主要作用是将当前协程立即挂起，然后在一个普通的线程中执行Lambda表达式中的代码。Lambda表达式的参数列表上会传入一个Continuation 参数，调用它的 `resume()` 方法或 `resumeWithException()` 可以让协程恢复执行。

非阻塞式

== 和 ===

结构相等 `==`：用 `equals()` 判断相等性

引用相等`===`：判断两个引用是否指向同一对象

with, run, apply

with

```
val list = listOf("1", "2", "3")
val result = with(StringBuilder()) {
    append("this is with")
    for(fruit in list) {
        append(fruit).append("\n")
    }
}
println(result)
```

//Lambda 表达式最后一行代码会作为with函数的返回值返回

run函数

用法和使用场景和with差不多， `run`函数只接收一个Lambda参数，并且会在Lambda表达式中提供调用对象的上下文，也是使用Lambda表达式中的最后一行代码作为返回值返回

```

val list = listOf("1", "2", "3")
val result = StringBuilder().run {
    append("this is with")
    for(fruit in list) {
        append(fruit).append("\n")
    }
    append("Ate all fruits")
}
println(result)
//Lambda 表达式最后一行代码会作为with函数的返回值返回

```

apply

也是和run函数差不多，但是apply 函数无法指定返回值， 而是会自动返回调用者对象本身

```

val list = listOf("1", "2", "3")
val result = StringBuilder().apply {
    append("this is with")
    for(fruit in list) {
        append(fruit).append("\n")
    }
    append("Ate all fruits")
}
println(result.toString())

```

let , also

also 返回调用者本身

```

ar name = "hi-dhl"

// 返回调用本身
name = name.also {
    val result = 1 * 1
    "juejin"
}
println("name = ${name}") // name = hi-dhl

//交换两个变量
val a = 1
val b = 2
a = b.also{ b = a }
println("a = ${a} b = ${b}") // a = 2, b = 1

```

let 返回最后一行

```
name = name.let {  
    val result = 1 * 1  
    "hi-dhl.com"  
}  
println("name = ${name}") // name = hi-dhl.com
```

密封类

密封类的关键字是 `sealed class`

```
//使用密封类前  
interface Result  
class Success(val msg: String) : Result  
class Failure(val error: Exception) : Result  
  
fun getResultMsg(result: Result) = when(result) {  
    is Success -> result.msg  
    is Failure -> result.error.message  
    else -> xxxxxxxx  
}  
  
//使用密封类后  
sealed class Result  
class Success(val msg: String) : Result()  
class Failure(val error: Exception) : Result()  
  
fun getResultMsg(result: Result) = when(result) {  
    is Success -> result.msg  
    is Failure -> result.error.message  
    //不需要else  
}
```

当在when语句中传入一个密封类变量作为条件时，Kotlin 编译器会自动检查该密封类有哪些子类，所对应的条件全部处理。

kotlin扩展的应用

```
//一般使用  
Toast.makeText(this, "hello", Toast.LENGTH_SHORT).show()  
  
//加一个扩展  
fun Activity.toast(message: CharSequence) = Toast.makeText(this, message,  
    Toast.LENGTH_SHORT).show()  
  
//扩展后使用  
toast("hello")
```

高阶函数

如果一个函数接收另一个函数作为参数，或者返回值的类型是另一个函数，那么该函数就称为高阶函数。

原理：kotlin编译器会将这些高阶函数的语法转换成Java支持的语法结构，会转换成匿名类的实现方式。

但这意味着每调用一次Lambda表达式，都会创建一个新的匿名类实例，会造成额外的内存和性能开销。

使用**内联函数**可以解决使用Lambda表达式带来的运行时开销

vararg 关键字

vararg对应的就是Java中可变参数列表，我们允许向这个方法传入任意多个参数，这些参数都会被赋值到使用vararg声明的这个变量上，然后使用for-in循环就可以将传入的所有参数遍历出来。

双冒号 ::

在Kotlin 中，一个**函数名**的左边加上双冒号，它就不表示这个函数本身了，而是表示一个对象，或者说一个指向对象的引用，但这个对象可不是函数本身，而是一个和这个函数具有相同功能的对象。

对象是不能加个括号来调用了，但是函数类型的对象可以，这是Kotlin 的语法糖，实际真正调用这个对象的是**invoke()** 函数

```
d(1) // 实际上会调用 d.invoke(1)  
(::b)(1) // 实际上会调用 (::b).invoke(1)
```

Lambda表达式

<https://kaixue.io/kotlin-lambda/>

内联函数

扔物线: <https://juejin.im/post/6869954460634841101#heading-3>

内联函数：在定义的高阶函数时加上 `inline` 关键字的声明即可

内联的函数类型参数在编译的时候会进行代码替换，因此没有真正的参数属性，非内联的函数类型参数可以自由地传递给其他任何函数，因为它就是一个真实的参数，而内联的函数类型参数只允许传递给另外一个内联函数，这就是它最大的局限性。

内联函数和非内联函数最大的区别在于，内联函数所引用的Lambda表达式中是可以使用return关键字来进行函数返回的，而非内联函数只能进行局部返回。

crossinline

局部加强内联优化

内联函数里的函数类型参数不允许间接调用

```
inline fun hello(postAction: () -> Unit) {  
    println("1111")  
    runOnUiThread {  
        postAction() //不允许这样调用  
    }  
}
```

如果真的要这样间接调用，需要在参数上加上**crossinline**关键字

```
inline fun hello(crossinline postAction: () -> Unit) {  
    println("2222")  
    runOnUiThread {  
        postAction() // 加上crossinline 之后就可以调用了  
    }  
}
```

Any 是Kotlin中所有类的基类，相当于java的Object **val** 声明一个不可变变量

双冒号 :: 一个函数名的左边，加上双冒号，就不表示这个函数本身了，而是表示一个对象，这个对象不是函数本身，而是一个和这个函数具有相同功能的对象

Kotlin 不允许我们调用**即是成员函数，又是扩展函数**的函数

Lambda 的返回值不是return 而是 最后一行作为返回值

Kotlin 的匿名函数不是函数，而是一个对象

在Kotlin里匿名函数 和 Lambda 其实都是对象

创建函数类型的对象有三种方式：双冒号加函数名，匿名函数，Lambda

var声明一个可变的变量

```
val range = 0..10 //两端都是闭区间 表示0到10  
val range = 0 until 10 //左闭右开区间 表示0 到 9  
for-in 循环每次执行循环时都会在区间范围内加1 相当于 i++ 可以使用step关键字跳过一些元素  
for (i in 0 until 10 step 2) {  
    println(i) //输出0 2 4 6 8  
}
```

修饰符

	kotlin	java
默认	public	default
protected	当前类，子类	当前类，子类，同包类
internal	同一模块	

集合

`listOf()` 创建一个不可变的集合，不可变集合只能用于读取，不能进行添加，修改或者删除操作

`mutableListOf()` 函数创建一个可变的集合

匿名类 由于kotlin完全舍弃了new关键字，因此创建匿名类实例的时候不能再使用new了，而是改用了**object**关键字

遍历

```
val map = mapOf("Apple" to 1, "Banana" to 2, "Orange" to 3)
for((fruit, number) in map) {
    println("fruit is " + fruit + ", number is "+ number)
}
//通过for in 循环，将Map的键值对变量一起声明到了一对括号里面，这样当循环遍历的时候，每次遍历就会赋值给这两个键值对变量。
```

判空

可空类型	<code>?</code>
判空辅助	<code>?.</code> <code>a?.doSomething()</code> 如果不为空执行后面
	<code>?:</code> <code>val a ?: b</code> 不为空执行左边，否则返回右边
<code>!!</code>	<code>content!!.toUpperCase()</code> 不做空指针检查

高阶函数定义：如果一个函数接收另一个函数作为参数，或者返回值的类型是另一个函数，那么该函数就成为高阶函数。

```
//定义一个函数
// -> 左边的部分就是用来声明该函数接收什么参数的，多个参数之间使用逗号隔开，如果不接收任何参数，写一对空括号就可以了，而 -> 右边的部分用于声明该函数的返回值是什么类型，如果没有返回值就是用Unit，相当于Java的void
(String, Int) -> Unit{
    func("hello", 123)
}
```

内联函数：它可以将使用Lambda表达式带来的运行时开销完全消除。

内联函数：定义高阶函数时加上`inline` 关键字

原理：就是Kotlin编译器会将内联函数中的代码在编译的时候自动替换到调用它的地方，这样就不存在运行时的开销了

缺点：内联的函数类型参数在编译的时候会被进行代码替换，因此它没有真正的参数属性，非内联的函数类型参数可以自由地传递给其他任何函数，因为它就是一个真实的参数，而内联函数的函数类型只允许传递给另外一个内联函数，这也是他最大的局限性。内联函数和非内联函数还有一个重要的区别，那就是内联函数所引用的Lambda表达式中是可以使用return关键字进行函数返回的，而非内联函数只能进行局部返回。

单例

在kotlin中创建一个单例类的方式极其简单，只需要将class关键字改成object关键字即可。

泛型

```
class MyClass {  
    fun <T> method(param : T) : T {  
        return param  
    }  
}
```

在默认情况下，所有的泛型都是可以指定成可空的类型，因为在不手动指定上界的时候，泛型的上界默认是Any？如果想要泛型不可为空，只需要将泛型上界手动指定成Any就可以了

委托

委托是一种设计模式，它的基本理念是：操作对象自己不会去处理某段逻辑，而是会把工作对象委托给另一个辅助对象去处理

类委托的关键字是 by，只需要在接口声明的后面使用by关键字，在接上受委托的辅助对象就可以了

顶级函数

只有不定义在任何类当中的函数才是顶级函数

扩展函数

扩展函数表示即使在不修改某个类源码的情况下，仍然可以打开这个类，向该类添加新的函数。

```
//这是一个String 的扩展函数  
//定义扩展函数的方法只要在扩展的类后面加上 .函数名 就可以了  
fun String.lettersCount(): Int {  
    var count = 0  
    for (char in this) {  
        if (char.isLetter()) {  
            count++  
        }  
    }  
    return count  
}
```

使用扩展函数简化Toast的用法：

```
//第二个参数加入默认值，添加参数的时候就使用参数，不添加的时候就使用默认值，能够更灵活的设置时间
fun String.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}

fun Int.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}
```

委托

kotlin 中委托使用的关键字是 `by`，我们只需要在接口声明的后面使用`by`关键字，再接上受委托的辅助对象，就可以免去一大堆模板代码

```
//委托类
class MySet<T>(val helperSet<T>): Set<T> by helperSet{

}

//委托属性
class MyClass {
    var p by Delegate()
}
```

只有不定义在任何类当中的函数才是顶级函数

infix函数

1. 只有一个参数
2. 在方法前必须加`infix`关键字
3. 必须是成员方法或者扩展方法

- `infix`函数是不能定义成顶层函数的，它必须是某个类的成员函数，可以使用扩展函数的方式将它定义到某个类当中
- `infix`函数必须接收且只能接收一个参数，参数类型没有限制

```

infix fun String.beginsWith(prefix: String) = startsWith(prefix)
//加上infix 即可

//使用
if ("Hello kotlin" beginsWith "Hello") {

}

```

infix 函数允许我们将函数调用时候的小数点、括号等计算机相关的语法去掉，从而使用一种更接近英语的语法来编写程序，让代码看起来更具可读性

限制：

infix函数是不能定义成顶级函数的，它必须是某个类的成员函数，可以使用扩展函数的方式将它定义到某个类当中；其次infix 函数必须接收且只能接收一个参数，至于参数类型是没有限制的

泛型高级用法

内联函数泛型实化：

首先该函数必须是内联函数才行，也就是要用 `inline` 关键字修饰该函数

其次，在声明泛型的地方必须加上 `reified` 关键字来表示该泛型要进行实化

```

inline fun <reified T> getGenericType() = T::class.java

//测试
val result1 = getGenericType<String>()
val result2 = getGenericType<Int>()
println("result1 is $result1")//输出 result1 is class java.lang.String
println("result2 is $result2")//输出 result2 is class java.lang.Integer

```

泛型实化的应用

```

inline fun <reified T> startActivity(context: Context) {
    val intent = Intent(context, T::class.java)
    context.startActivity(intent)
}

//使用
startActivity<TestActivity>(context)

//如果想要附带一些参数，使用高阶函数即可
inline fun <reified T> startActivity(context: Context, block: Intent.() -> Unit) {
    val intent = Intent(context, T::class.java)
    intent.block()
    context.startActivity(intent)
}

//使用
startActivity<TestActivity>(context) {
    putExtra("param1", "data")
    putExtra("param1", "data")
}

```

泛型协变

假如定义了一个MyClass的泛型类，其中 A 是 B 的子类型，同时 MyClass 又是 MyClass的子类型，那么我们可以称 MyClass 在 T 这个泛型上是协变的。

Kotlin-DSL

DSL 全称是领域特定语言，他是编程语言赋予开发者的一种特殊的能力，通过它们可以编写出一些看似脱离其原始语法结构的代码，从而构建出一种专有的语法。

internal 关键字

internal 修饰类的方法，表示这个类方法只适合当前module 使用，如果其他module 使用的话会找不到这个 internal 方法或者报错。

泛型的协变、逆变

在Java 中 使用 泛型上界通配符 ? extends 来表示协变，但是只能读取不能修改，这里的修改指对泛型集合添加元素，如果是remove 以及 clear 是可以的。

使用 泛型下界通配符? super 来表示逆变，但是只能修改不能读取，这个说的读取是指不能按照泛型类型进行读取，你如果按照 Object 读出来再强制转型是可以的。

```
//协变
List<? extends TextView> list = new ArrayList<Button>();
TextView textView = list.get(0);
list.add(textView); //报错
```

```
//逆变
List<? super Button> list = new ArrayList<TextView>();
Object object = list.get(0);
TextView textView = list.get(0); //报错
Button button = (Button)list.get(0);
list.add(button);
```

在 Kotlin 中 使用 `out` 和 `in` 来表示协变和逆变

- 使用关键字 `out` 表示协变，相当于上界通配符 `? extends`
- 使用关键字 `in` 来表示逆变，相当于 Java 的下界通配符 `? super`

```
var textView: List<out TextView>
var TextView: List<int TextView>
```

`out` 表示，我这个变量或者参数只用来输出，不用来输入，你只能读我不能写我；`in` 就反过来，表示它只用来输入，不用来输出，你只能写我不能读我。

```
//out
class Producer<out T> {
    fun produce(): T {
        ...
    }
}

val producer: Producer<TextView> = Producer<Button>()
val textView: TextView = producer.produce() // ← 相当于 'List' 的 `get`


//in
class Consumer<in T> {
    fun consume(t: T) {
        ...
    }
}

val consumer: Consumer<Button> = Consumer<TextView>()
consumer.consume(Button(context)) // ← 相当于 'List' 的 'add'
```

泛型中的 `where` 关键字

```
//在Java中
// T必须是 Animal 的子类
class Monster<T extends Animal>{

}

//多个边界用 & 号连接
//T必须是Animal 和 Food 的子类型
class Monster<T extends Animal & Food> {

}

//在Kotlin 中
class Monster<T: Animal>
```

```
class Minster<T> where T: Animal, T: Food
```

Kotlin 委托

lateinit

```
private lateinit var name: String
```

lateinit 只能用来修饰类属性，不能用来修饰局部变量，并且只能用来修饰对象，不能用来修饰基本类（因为基本类型的属性在类加载后的准备阶段都会被初始化为默认值）

lateinit 的作用就是让编译期在检查时不要因为属性变量而未被初始化而报错。

by lazy

by lazy本身是一种属性委托，委托的关键字是 by

```
val name:String by lazy {
    println("只有第一次才会调用")
    println("后续调用最后一行代码作为返回值")
    "Lima"
}

fun main() {
    println(name)
    println(name)
    println(name)
}
```

by lazy 要求属性声明为 val，即不可变量，{}内的操作就是返回唯一一次初始化的结果。

by lazy 可以使用于类属性或者局部变量

区别：

lateinit 只能用于 变量 var， lazy 只能用于常量 val

lateinit 只是让编译期忽略对属性未初始化的检查，后续在哪里以及何时初始化还需要开发者来决定

by lazy 在声明的同时也指定了延迟初始化行为，在属性第一次被使用的时候就会自动执行{}里面初始化的代码。

Observable

可观察属性 Observable，如果你要观察一个属性的变化过程，那么可以将属性委托给 Delegates.observable

该方法接收两个参数：

- initialValue: 初始值
- onChange: 属性值被修改时的回调处理器，回调有三个参数
 - property (被赋值的属性)

- oldValue (旧值)
- newValue (新值)

```

var name: String by Delegates.observable("默认值"){property, oldValue, newValue ->
    println("property: $property, $oldValue -> $newValue")
}

fun main() {
    name = "第一次修改"
    name = "第二次修改"
}

//打印输出
property: var observableProp: kotlin.String: 默认值 -> 第一次修改值
property: var observableProp: kotlin.String: 第一次修改值 -> 第二次修改值

```

通过委托Observable 可以观察属性值的变化过程。

vetoable 函数

vetoable 与 Observer 一样，可以观察属性值的变化过程，不同的是，vetoable 可以通过 处理器函数来决定属性值是否生效。

```

var vetoableProp: Int by Delegates.vetoable(0){
    _, oldValue, newValue ->
    // 如果新的值大于旧值，则生效
    newValue > oldValue
}

fun main() {
    println("vetoableProp=$vetoableProp")
    vetoableProp = 10
    println("vetoableProp=$vetoableProp")
    vetoableProp = 5
    println("vetoableProp=$vetoableProp")
    vetoableProp = 100
    println("vetoableProp=$vetoableProp")
}

```

打印如下：

```

vetoableProp=0
0 -> 10
vetoableProp=10
10 -> 5
vetoableProp=10
10 -> 100
vetoableProp=100

```

可以看到10 -> 5 的赋值没有生效。

companion object

表示静态代码块, 类似Java 中的 static {} ,其中的代码有且只会执行一次

```
companion object {  
    }  
}
```

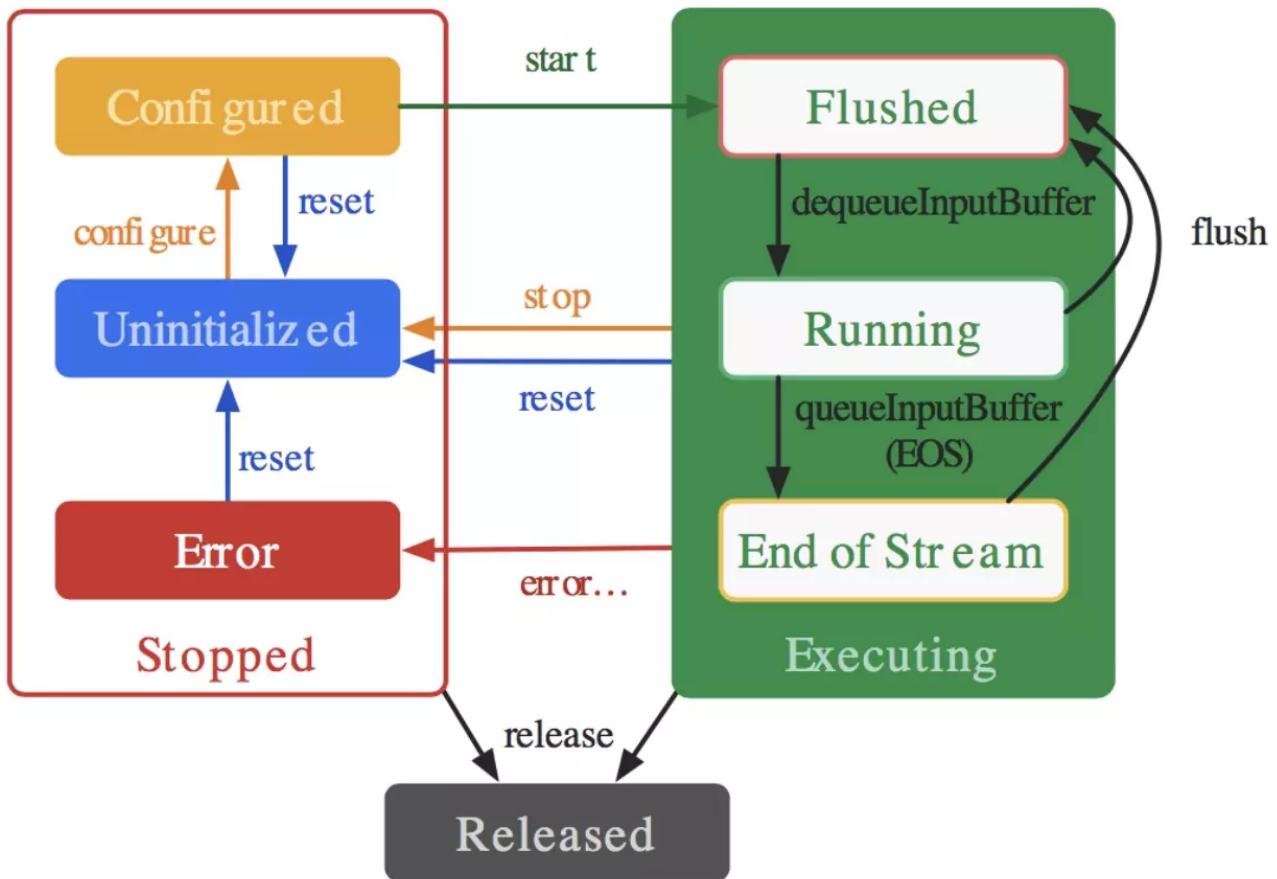
音视频

码率, 是指一个数据流中每秒钟能通过的信息量, 单位bps (bit per second)

码率 = 采样率 * 采样位数 * 声道数

ffmpeg 使用的是软解码, 也即是纯 CPU 解码; 而使用平台的 MediaCodec 播放的是硬解码, 也就是支持 GPU 协助

基本流程:



1. 设置数据源

```
//设置数据源,将数据路径传进去  
//音频视频分开创建对象  
mExtractor =new MediaExtractor();  
mExtractor.setDataSource(path);
```

2. 初始化参数

```
//音频需要设置  
// 获取 通道数, 采样率, 采样位数  
@Override  
void initSpecParams(MediaFormat format) {  
    try {  
        //通道数  
        mChannels = format.getInteger(MediaFormat.KEY_CHANNEL_COUNT);  
        //采样率  
        mSampleRate = format.getInteger(MediaFormat.KEY_SAMPLE_RATE);  
  
        mPCMEncodeBit = format.containsKey(MediaFormat.KEY_PCM_ENCODING)?  
            format.getInteger(MediaFormat.KEY_PCM_ENCODING) :  
        AudioFormat.ENCODING_PCM_16BIT;  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
  
//初始化渲染器,视频不需要  
@Override  
boolean initRender() {  
    //CHANNEL_OUT_MONO 单声道  
    //CHANNEL_OUT_STEREO双声道  
    int channel = mChannels == 1 ? AudioFormat.CHANNEL_OUT_MONO :  
    AudioFormat.CHANNEL_OUT_STEREO;  
    //获取构建对象的最小缓冲区大小  
    int minBufferSize =  
    AudioTrack.getMinBufferSize(mSampleRate, channel, mPCMEncodeBit);  
    mAudioOutTempBuf = new short[minBufferSize / 2];  
    /**  
     * 创建AudioTrack并启动  
     * 1.播放类型: 音乐 2.采样率 3.通道 4.采样位数 5.缓冲区大小 6.播放模式:数据流动态写入, 另一种是一次性写入  
     */  
    mAudioTrack = new AudioTrack(  
        AudioManager.STREAM_MUSIC,  
        mSampleRate,  
        channel,  
        mPCMEncodeBit,  
        minBufferSize,  
        AudioTrack.MODE_STREAM  
    );  
    mAudioTrack.play();  
    return true;  
}
```

3. 创建解码器

```
//1.根据音视频编码格式初始化解码器  
//video/avc 即 h.264 ,audio/mp4a-latm 即 AAC  
String type = mExtractor.getFormat().getString(MediaFormat.KEY_MIME);  
//创建解码器  
mCodec = MediaCodec.createDecoderByType(type);
```

4. 配置解码器

```
//配置解码器  
//如果是视频需要传入surface, 如果是音频不需要surface 直接传入null  
mediaCodec.configure(format, surface, null, 0);  
  
//记得配置解码器,并传入surface对象,否则会黑屏无法显示.
```

5. 启动解码器

```
//3.启动解码器  
mCodec.start();
```

- 读取 Mp4 文件中的编码数据，并送入解码器解码
- 获取解码好的帧数据
- 将一帧画面渲染到屏幕上

解码流程

在解码的时候，通过 `dequeueInputBuffer` 查询到一个空闲的输入缓冲区，在通过 `queueInputBuffer` 将 `未解码` 的数据压入解码器，最后，通过 `dequeueOutputBuffer` 得到 `解码好` 的数据。

```
private fun pushBufferToDecoder(): Boolean {
    var inputBufferIndex: Int = mCodec!!.dequeueInputBuffer(timeoutUs: 1000)
    var isEndOfStream = false

    if (inputBufferIndex >= 0) {
        val inputBuffer: ByteBuffer = mInputBuffers!![inputBufferIndex]
        val sampleSize: Int = mExtractor!!.readBuffer(inputBuffer)

        if (sampleSize < 0) {
            //如果数据已经取完，压入数据结束标志：MediaCodec.BUFFER_FLAG_END_OF_STREAM
            mCodec!!.queueInputBuffer(inputBufferIndex, offset: 0, size: 0,
                presentationTimeUs: 0, MediaCodec.BUFFER_FLAG_END_OF_STREAM)
            isEndOfStream = true
        } else {
            mCodec!!.queueInputBuffer(inputBufferIndex, offset: 0,
                sampleSize, mExtractor!!.getCurrentTimestamp(), flags: 0)
        }
    }
    return isEndOfStream
}

private fun pullBufferFromDecoder(): Int {
    // 查询是否有解码完成的数据，index >=0 时，表示数据有效，并且index为缓冲区索引
    var index: Int = mCodec!!.dequeueOutputBuffer(mBufferInfo, timeoutUs: 1000)
    Log.e(TAG, msg: "pushBufferFromDecoder: index = $index")
    when (index) {
```

编码流程

其实，编码流程和解码流程基本是一样的。不同在于压入 `dequeueInputBuffer` 输入缓冲区的数据是未编码的数据，通过 `dequeueOutputBuffer` 得到的是已编码的数据。

音视频同步

解码有两个重要参数, PTS 和 DTS, 分别表示渲染时间和解码时间. 这里我们使用的是 PTS

一般可以采用三个时间源 来实现同步:

- 视频时间戳
- 音频时间戳
- 系统时间戳

视频时间戳

通常情况下，由于人类对声音比较敏感，并且视频解码的PTS通常不是连续，而音频的PTS是比较连续的，如果以视频为同步信号源的话，基本上声音都会出现异常，而画面的播放也会像倍速播放一样。

音频时间戳

以音频的PTS作为同步源，让画面适配音频是比较不错的一种选择。

但是这里不采用，而是使用系统时间作为同步信号源。因为如果以音频PTS作为同步源的话，需要比较复杂的同步机制，音频和视频两者之间也有比较多的耦合。

系统时间戳

而系统时间作为统一信号源则非常适合，音视频彼此独立互不干扰，同时又可以保证基本一致。

在解码数据出来以后，检查PTS时间戳和当前系统流过的时间差距，快则延时，慢则直接播放

- 基本原理、概念
 - 音视频格式
 - 编解码
- C++
- FFmpeg
- protobuf for c++ :客户端数据传输
- OpenGL ES : 视频渲染
 - OpenGL 是间接操作GPU的工具，是一组定义好的跨平台和跨语言的图形API，是可用于2D和3D画面渲染的底层图形库，是由各个硬件厂家具体实现的编程接口
- OpenSL ES : 音频播放
- gpuimage : 基于opengl的库

<https://juejin.im/post/5ed4c55451882542f44894b1>

• SurfaceView

继承自view，本质上是一个view，但是与普通的view不同，他有自己的surface，在WMS中有对应的WindowState，在SurfaceFlinger中有Layer

优点：可以在一个独立的进程中进行绘制，不影响主线程

缺点：Surface不能在view hierarchy中，它显示也不受view的属性控制，所以不能进行平移，缩放等变换，也不能放在其他的viewGroup中，Surface不能嵌套使用。

• TextureView

继承自View，和Surface不同的是它不会在WMS中单独创建窗口，而是作为view hierarchy中的一个普通的View，因此可以和其他普通的View一样进行移动，旋转，缩放，动画等变化。占用内存比SurfaceView高，5.0以前在主线程渲染，5.0以后有单独的渲染线程。

TextureView专门用来渲染像视频或OpenGL场景之类的数据的，而且TextureView只能用在具有硬件加速的Window中，如果使用的是软件渲染，TextureView将什么也不显示。也就是说对于没有GPU的设备，TextureView完全不可用。

• GLSurfaceView

GLSurfaceView作为SurfaceView的补充，可以看作是SurfaceView的一种典型使用模式。在SurfaceView的基础上，它加入了EGL的管理，并自带了渲染线程。另外它定义了用户需要实现的Render接口，提供了用Strategy pattern更改具体Render行为的灵活性。作为GLSurfaceView的Client，只需要将实现了渲染函数的Renderer的实现类设置给GLSurfaceView即可。

- **SurfaceTexture**

SurfaceTexture 和 SurfaceView 不同的是，它对图像流的处理并不直接显示，而是转为 GL 外部纹理，因此可用于图像流数据的二次处理 (如 Camera 滤镜，桌面特效等)。比如 Camera 的预览数据，变成纹理后可以交给 GLSurfaceView 直接显示，也可以通过 SurfaceTexture 交给 TextureView 作为 View heirachy 中的一个硬件加速层来显示。首先，SurfaceTexture 从图像流 (来自 Camera 预览，视频解码，GL 绘制场景等) 中获得帧数据，当调用 updateTexImage() 时，根据内容流中最近的图像更新SurfaceTexture 对应的 GL 纹理对象，接下来，就可以像操作普通 GL 纹理一样操作它了

<https://blog.csdn.net/leixiaohua1020/article/details/18893769>

播放器原理：视频播放原理： (mp4、flv) ->解协议（网络协议，本地播放不需要）-> 解封装 (mp3/aac、h264/h265) -> 解码 (pcm、yuv) -> 音视频同步 -> 渲染播放

音视频同步：

- 选择参考时钟源：音频时间戳、视频时间戳和外部时间三者选择一个作为参考时钟源（一般选择音频，因为人对音频更敏感，ijk 默认也是音频）
- 通过等待或丢帧将视频流与参考时钟源对齐，实现同步

OpenGL ES

OpenGL 是一组可以操作GPU 的 API, 然而仅仅能够操作GPU , 并不能够将图像渲染到设备上进行显示.

在Android中 OpenGL ES 通常配合GLSurfaceView使用, 在GLSurfaceView中, Google 已经封装好了渲染的基础流程

OpenGL 是基于线程的一个状态机,有关OpenGL的操作,比如创建纹理ID, 初始化, 渲染等,都必须在同一个线程中完成, 否则会造成异常.

1. 初始化顶点坐标, 在绘制之前就需要把OpenGL的世界坐标和纹理坐标确定好.

OpenGL所有的画面都是由三角形构成的

- GL_TRIANGLES: 独立顶点的构成三角形
- GL_TRIANGLE_STRIP: 复用顶点构成三角形
- GL_TRIANGLE_FAN: 复用第一个顶点构成三角形

一般采用GL_TRIANGLE_STRIP 模式进行绘制

因为底层不能直接接收数组, 所以将数组转换为ByteBuffer

```
//分配地址空间, 一个float是4个字节, 所以 * 4
    ByteBuffer bb = ByteBuffer.allocateDirect(mVertexCoors.length * 4);
    bb.order(ByteOrder.nativeOrder());
    //将坐标数据转换为FloatBuffer, 用以传入给OpenGL 程序
    mVerTexBuffer = bb.asFloatBuffer();
    mVerTexBuffer.put(mVertexCoors);
    mVerTexBuffer.position(0);
```

2. 创建, 编译,并启动OpenGL着色器

```

private fun createGLPrg() {
    if (mProgram == -1) {
        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, getVertexShader())
        val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, getFragmentShader())

        //创建OpenGL ES程序,注意:需要在OpenGL渲染线程中创建,否则无法渲染
        mProgram = GLES20.glCreateProgram()
        //将顶点着色器加入到程序
        GLES20.glAttachShader(mProgram, vertexShader)
        //将片元着色器加入到程序中
        GLES20.glAttachShader(mProgram, fragmentShader)
        //连接到着色器程序
        GLES20.glLinkProgram(mProgram)

        //Java和GLSL交互的通道,通过属性可以给GLSL设置相关的值
        mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
        mTexturePosHandler = GLES20.glGetAttribLocation(mProgram, "aCoordinate")
    }
    //使用OpenGL程序
    GLES20.glUseProgram(mProgram)
}

private fun getVertexShader(): String {
    return "attribute vec4 aPosition;" +
        "void main() {" +
        "    gl_Position = aPosition;" +
        "}"
}

private fun getFragmentShader(): String {
    return "precision mediump float;" +
        "void main() {" +
        "    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);;" +
        "}"
}

private fun loadShader(type: Int, shaderCode: String): Int {
    //根据type创建顶点着色器或者片元着色器
    val shader = GLES20.glCreateShader(type)
    //将资源加入到着色器中,并编译
    GLES20.glShaderSource(shader, shaderCode)
    GLES20.glCompileShader(shader)

    return shader
}

```

3. 激活并绑定纹理单元

```

//注意注意
private fun activateTexture() {
    //激活指定纹理单元

```

```

//OpenGL中内置的很多纹理单元，并且是连续的，GL_TEXTURE0, GL_TEXTURE1, GL_TEXTURE2...
GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
//绑定纹理ID到纹理单元
GLES20 glBindTexture(GLES20.GL_TEXTURE_2D, mTextureId)
//将激活的纹理单元传递到着色器里面
//注意，第二个参数索引必须和上面的激活的纹理单元索引保持一致，否则无法显示
GLES20 glUniform1i(mTextureHandler, 0)
//配置边缘过渡参数
//GLES20.GL_TEXTURE_2D是用来渲染图片的，渲染视频要改成拓展纹理单元
GLES11Ext.GL_TEXTURE_EXTERNAL_OES
//配置纹理过滤模式，GL_LINEAR(线性过滤)
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR);
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
//配置纹理环绕方式，第三个参数是环绕方式
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_CLAMP_TO_EDGE);
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_T, GLES20.GL_CLAMP_TO_EDGE);

}

```

4. 绑定图片到纹理单元

```

private fun bindBitmapToTexture() {
    if (!mBitmap.isRecycled) {
        //绑定图片到被激活的纹理单元
        GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, mBitmap, 0)
    }
}

```

5. 开始渲染

```

private fun doDraw() {
    //启用顶点的句柄
    GLES20 glEnableVertexAttribArray(mVertexPosHandler)
    GLES20 glEnableVertexAttribArray(mTexturePosHandler)
    //设置着色器参数，第二个参数表示一个顶点包含的数据数量，这里为xy，所以为2
    GLES20 glVertexAttribPointer(mVertexPosHandler, 2, GLES20.GL_FLOAT, false, 0,
    mVertexBuffer)
    GLES20 glVertexAttribPointer(mTexturePosHandler, 2, GLES20.GL_FLOAT, false, 0,
    mTextureBuffer)
    //开始绘制
    GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4)
}

```

激活着色器的顶点坐标和纹理坐标属性,然后把初始化好的坐标传递给着色器,最后进行绘制

绘制的方式有两种:

glDrawArrays和glDrawElements, 两者区别在于glDrawArrays是直接使用定义好的顶点顺序进行绘制; 而glDrawElements则是需要定义另外的索引数组, 来确认顶点的组合和绘制顺序。

着色器

```
private fun getVertexShader(): String {
    return //顶点坐标
        "attribute vec2 aPosition;" +
        //纹理坐标
        "attribute vec2 aCoordinate;" +
        //用于传递纹理坐标给片元着色器, 命名和片元着色器中的一致
        "varying vec2 vCoordinate;" +
        "void main() {" +
        "    gl_Position = aPosition;" +
        "    vCoordinate = aCoordinate;" +
        "}"
}

private fun getFragmentShader(): String {
    return
        //如果是渲染视频, 需要加上拓展纹理
        //拓展纹理的作用就是把YUV转换为RGB显示到屏幕上
        "#extension GL_OES_EGL_image_external : require\n" +
        //配置float精度, 使用了float数据一定要配置: lowp(低)/mediump(中)/highp(高)
        "precision mediump float;" +
        //从Java传递进入来的纹理单元
        "uniform sampler2D uTexture;" +
        //从顶点着色器传递进来的纹理坐标
        "varying vec2 vCoordinate;" +
        "void main() {" +
        //根据纹理坐标, 从纹理单元中取色
        "    vec4 color = texture2D(uTexture, vCoordinate);" +
        "    gl_FragColor = color;" +
        "}"
}
```

操作GLSL流程总结

比如设置一个透明度

1. 创建OpenGL ES 程序

```
private fun createGLPrg() {
    if (mProgram == -1) {
        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, getVertexShader())
```

```
    val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
getFragmentShader())

    //创建OpenGL ES程序,注意:需要在OpenGL渲染线程中创建,否则无法渲染
    mProgram = GLES20.glCreateProgram()
    //将顶点着色器加入到程序
    GLES20.glAttachShader(mProgram, vertexShader)
    //将片元着色器加入到程序中
    GLES20.glAttachShader(mProgram, fragmentShader)
    //连接到着色器程序
    GLES20.glLinkProgram(mProgram)
    //alpha 对应顶点着色器里面对应的名称
    mAlphaHandler = GLES20.glGetAttribLocation(mProgram, "alpha")
}
//使用OpenGL程序
GLES20.glUseProgram(mProgram)
}
```

```
private fun getVertexShader(): String {
    return "attribute vec4 aPosition;" +
        "precision mediump float;" +
        "uniform mat4 uMatrix;" +
        "attribute vec2 aCoordinate;" +
        "varying vec2 vCoordinate;" +
        "attribute float alpha;" +
        "varying float inAlpha;" +
        "void main() {" +
        "    gl_Position = uMatrix*aPosition;" +
        "    vCoordinate = aCoordinate;" +
        "    inAlpha = alpha;" +
        "}"
}

private fun getFragmentShader(): String {
    //一定要加换行"\n",否则会和下一行的precision混在一起,导致编译出错
    return "#extension GL_OES_EGL_image_external : require\n" +
        "precision mediump float;" +
        "varying vec2 vCoordinate;" +
        "varying float inAlpha;" +
        "uniform samplerExternalOES uTexture;" +
        "void main() {" +
        "    vec4 color = texture2D(uTexture, vCoordinate);" +
        "    gl_FragColor = vec4(color.r, color.g, color.b, inAlpha);"
    }
}
```

2. 对OpenGL 进行设置

```
//第一个参数对应上面创建着色器时候的名称  
//第二个参数就是我们要传进去的透明值  
GLES20.glVertexAttrib1f(mAlphaHandler, mAlpha)  
  
// glVertexAttrib1f对应的就是 着色器里面的 varying 类似的还有 glUniform1i 对应 uniform, 其他  
也差不多
```

这样就完成对OpenGL 的操作, 通过 glVertexAttrib1f 传入着色器 和 透明值, 着色器会把透明值传递给顶点着色器, 顶点着色器又通过 varying 传递给片元着色器进行颜色设置.

要把Java中的值传递到片元着色器, 直接传值是不行的, 需要通过顶点着色器通过varying 进行间接传递.

画面比例矫正

OpenGL 提供两种方式可以对画面进行矫正, 分别是 透视投影 和 正交投影

透视投影

类似人眼成像原理, 距离越远, 投影就会越小, 经常用在3D渲染中

正交投影

Matrix.orthoM 函数表示正交投影

所有在裁剪空间中的物体, 无论远近,只要大小一样, 在近平面上的投影都是一样的, 不再有近大远小的效果.比较适合用来渲染2D画面.

渲染多视频画面

渲染多视频画面 其实就是生成多个纹理ID, 利用这个ID生成一个surface渲染表面, 最后把这个Surface给到解码器MediaCodec 进行渲染即可.

OpenGL 之 EGL

- **EGLDisplay**

EGL定义的一个抽象系统显示类, 用于操作设备窗口.

- **EGLConfig**

EGL配置, 如RGBA 位数

- **EGLSurface**

渲染缓存,一块内存空间,所有要渲染到屏幕上的图像数据,都要先缓存在EGLSurface上.

- **EGLContext**

OpenGL上下文, 用于存储OpenGL绘制状态信息, 数据.

进行初始化EGL 要配置以上过程.

为什么OpenGL可以在多个GLSurfaceView中正确绘制，在EGL初始化以后，即渲染环境（EGLDisplay、EGLContext、GLSurface）准备就绪以后，需要在渲染线程（绘制图像的线程）中，明确的调用glMakeCurrent。这时，系统底层会将OpenGL渲染环境绑定到当前线程。

在这之后，只要你是在渲染线程中调用任何OpenGL ES的API（比如生产纹理ID的方法GLES20 glGenTextures），OpenGL会自动根据当前线程，切换上下文（也就是切换OpenGL的渲染信息和资源）。

换而言之，如果你在非调用glMakeCurrent的线程中去调用OpenGL的API，系统将找不到对应的OpenGL上下文，也就找不到对应的资源，可能会导致异常出错。

这也就是为什么有文章说，OpenGL渲染一定要在OpenGL线程中进行。

影响编码速度的因素

https://segmentfault.com/a/1190000021223837?utm_source=tag-newest

1. 画面尺寸,画面数据大小决定进行每帧编码的任务量
2. 帧速率,帧数越高,耗时越多
3. Encoder Profile, Profile越高, 就说明采用越高级的压缩特性, 更高级的压缩特性,通常需要耗费更多的压缩时间.

影响画面清晰度的因素

1. Encoder Profile Level , Profile 越高,同样配置下所编码得到的视频文件清晰度就越高

MediaRecorder 和 AudioRecord

Android SDK 中有两套音频采集的API，分别是：MediaRecorder 和 AudioRecord。

1. MediaRecorder是一个更加上层一点的API，它可以直接把手机麦克风录入的音频数据进行编码压缩（如AMR、MP3等）并存成文件
2. AudioRecord则更接近底层，能够更加自由灵活地控制，可以得到原始的一帧帧PCM音频数据。

如果只是想简单地做一个录音机，录制音频文件，就使用 MediaRecorder，而如果需要对音频做进一步的算法处理、或者采用第三方的编码库进行压缩、以及网络传输、直播等应用，则建议使用 AudioRecord。

交叉编译

GCC

老牌编译工具，不仅可以编译C/C++，也可以编译Java, Object-C , GO 等语言

CLANG

是一个更高效的C/C++编译工具，并且兼容GCC, Google在NDK 17以后就把GCC移除了，全面推行使用CLANG

FFmpeg 相关库

avcodec	音视频编解码核心库
avformat	音视频容器格式封装和解析
avutil	核心工具库
swscal	图像格式转换的模块
swresampel	音频重采样
avfilter	音视频滤镜库，比如加水印，音频变声
avdevice	输入输出设备库，提供设备数据的输入与输出

Linux

- shell
- vim

枚举

优点：枚举本身线程安全，能够防止通过反射和反序列化创建实例。

缺点：对JDK版本有限制要求，非懒加载。

volatile

https://blog.csdn.net/SEU_Calvin/article/details/52370068

被volatile关键字修饰之后，任何一个线程对它进行修改，都会让所有其他的CPU高速缓存中的值过期，这样其他的线程就必须去内存中重新获取新的值，这样也就解决了可见性的问题。

被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象。

保证可见性，不保证原子性

可见性是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。

禁止指令重排序

CPU在执行代码时，并不一定会严格按照我们编写的顺序取执行，而是可能会考虑一些效率方面的原因，对哪些先后顺序无关紧要的代码进行重新拍排序，这个操作就称为指令重排。

指令重排序是指编译器和处理器为了优化程序性能对指令进行排序的一种手段，需要遵守一定规则：

1. 不会对存在依赖关系的指令重排序，例如 $a = 1; b = a$; a 和 b 存在依赖关系，不会被重排序
2. 不能影响单线程下的执行结果。比如： $a=1; b=2; c=a+b$ 这三个操作，前两个操作可以重排序，但是 $c=a+b$ 不会被重排序，因为要保证结果是3

使用场景：

对于一个变量，只有一个线程执行写操作，其他线程都是读操作，这时候可以用 **volatile** 修饰这个变量。

场景：

比如一个下载功能

```
public class DownloadTask {

    boolean isCanceled = false;

    public void download() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (!isCanceled) {
                    byte[] bytes = readBytesFromNetwork();
                    if (bytes.length == 0) {
                        break;
                    }
                    writeBytesToDisk(bytes);
                }
            }
        }).start();
    }

    public void cancel() {
        isCanceled = true;
    }
}
```

这里我们有两个方法，一个 `download` 下载，一个 `cancel` 取消下载。调用 `cancel()` 方法时将 `isCanceled` 变量设置为 `true`，表示下载已取消。

然后在 `download()` 方法当中，如果发现 `isCanceled` 变量为 `true`，就跳出循环不再继续执行下载任务，这样也就实现了取消下载的功能。

但是这种写法真的安全吗？不，因为你会发现 `download()` 方法和 `cancel()` 方法是运行在两个线程当中的，因此 `cancel()` 方法对于 `isCanceled` 变量的修改，未必对 `download()` 方法就立即可见。

所以，存在着这样一种可能，就是我们明明已经将 `isCanceled` 变量设置成了 `true`，但是 `download()` 方法所使用的 CPU 高速缓存中记录的 `isCanceled` 变量还是 `false`，从而导致下载无法被取消的情况出现。

因此，最安全的写法就是对 `isCanceled` 变量声明 **volatile** 关键字

十六进制状态管理

- 通过状态集的注入，一行代码即可完成模式的切换。
- 无论再多的状态，都只需要一个字段来存储。状态被存放在 int 类型的状态集中，可以直接向数据库写入或读取。

采用按位计算的方式来进行操作：

`a & b, a | b, a ^ b, ~a`

按位与 &

```
A = 0001  
B = 0011  
A & B = 0001
```

按位 或 |

```
//可以用来进行状态的添加  
A = 0001  
B = 0011  
A | B = 0011
```

按位 非 ~

```
A = 0010  
~ = 1101
```

左移 <<

移动一位相当乘2， 右移相反

```
A = 001101 // 13  
A << 1 = 011010 // 26
```

<https://www.zhihu.com/question/34021773/answer/118589857>

WorkManager

使用：

- 定义一个后台任务，并实现具体的任务逻辑
- 配置该后台任务的运行条件和约束信息，并构建后台任务请求。
- 将该后台任务请求传入WorkManager的enqueue()方法中，系统会在合适的时间运行

doWork() 方法不会运行在主线程当前，所以可以在这里执行耗时操作

OneTimeWorkRequest.Builder 是WorkRequest.Builder 用于构建单次运行的后台任务请求

PeriodicWorkRequest.Builder 可用于构建周期性运行的后台任务请求，当时为了降低设备性能消耗，构建函数中传入的运行周期事件不能短于15分钟

```
//添加约束:网络连接
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .build();

//构建后台任务
OneTimeWorkRequest request = new OneTimeWorkRequest.Builder(DeviceSyncWorker.class)
    .setConstraints(constraints)
    .build();

//执行后台任务
//将任务加入任务队列,需要等满足环境条件下才会执行。
WorkManager.getInstance().enqueue(request);
```

//链式任务，beginWith 用于开启一个链式任务，后面接上什么后台任务，只要使用then方法来连接即可，必须在一个后台任务运行成功后下一个任务才会运行

比如：A , B , C 按顺序执行

```
WorkManager.getInstance()
    .beginWith(workA)
    .then(workB)
    .then(workC)
    .enqueue()
```

A和B 同时执行，他们执行完成后再执行C

```
WorkManager.getInstance()
    .beginWith(workA, workB)
    .then(workC)
    .enqueue()
```

```

val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED) // 网络状态
    .setRequiresBatteryNotLow(true) // 不在电量不足时执行
    .setRequiresCharging(true) // 在充电时执行
    .setRequiresStorageNotLow(true) // 不在存储容量不足时执行
    .setRequiresDeviceIdle(true) // 在待机状态下执行，需要 API 23
    .build()

val request = OneTimeWorkRequest.Builder(MainWorker::class.java)
    .setConstraints(constraints)
    .build()

```

这个很好理解，除了网络状态，其他设置项都是传入一个布尔值，网络状态可选值如下：

状态	说明
NOT_REQUIRED	没有要求
CONNECTED	网络连接
UNMETERED	连接无限流量的网络
METERED	连接按流量计费的网络
NOT_ROAMING	连接非漫游网络

```

//任务状态监听
WorkManager.getInstance().getWorkInfoByIdLiveData(request.getId())
    .observe((LifecycleOwner) context, new android.arch.lifecycle.Observer<WorkInfo>() {
        @Override
        public void onChanged(@Nullable WorkInfo workInfo) {
            if (workInfo != null && workInfo.getState().isFinished()) {
                if (workInfo.getState() == WorkInfo.State.SUCCEEDED) {
                    //
                } else if (workInfo.getState() == WorkInfo.State.FAILED) {
                    //
                } else if (workInfo.getState() == WorkInfo.State.CANCELLED) {
                    //
                }
            }
        }
    });
return Observable.just(true);

```

Handler 原理

Android定义的一套子线程与主线程间通讯的消息传递机制。

子线程更新UI也行，但是只能更新自己创建的View,换句话说：Android的UI更新被设计成了单线程

```
void checkThread() {
    if (mThread != Thread.currentThread()) {
        throw new CalledFromWrongThreadException(
            "Only the original thread that created a view hierarchy can touch its views.");
    }
}
```

图中 mThread 是 ViewRootImpl 的成员变量，是在 ViewRootImpl 构造的时候赋值的，赋值的就是当前 Thread 的对象。所以说，你的 ViewRootImpl 在哪个线程创建的，你后续的 UI 更新就需要在那个线程执行，跟是不是 UI 线程毫无关系。

```
public ViewRootImpl(Context context, Display display) {
    mContext = context;
    mWindowSession = WindowManagerGlobal.getWindowSession();
    mDisplay = display;
    mBasePackageName = context.getPackageName();
    mThread = Thread.currentThread(); ←
    mLocation = new WindowLeaked(msg: null);
    mLocation.fillInStackTrace();
}
```

UI 线程其实就是 ActivityThread 中创建的线程，用来分发 Activity 生命周期和 UI 消息等，因为为了线程安全人为定义的一个线程。对于 View 操作的线程必须是 View 创建的线程，所以理论上不需要在 UI 线程中操作 View，只要保证创建 View 和 操作 View 的线程保持一致就行了。

UI 线程的概念应该是“视图绑定时所在的线程”，其实质和创建绑定时所在的线程相关联，并不能和主线程 MainThread 混为一谈，我们日常那么说只是因为便于理解罢了。

为什么子线程中不能直接 new Handler()，而主线程可以？

主线程与子线程不共享同一个 Looper 实例，主线程的 Looper 在启动时就通过 prepareMainLooper() 完成了初始化，而子线程还需要调用 Looper.prepare() 和 Looper.loop() 启开轮询，否则会报错

```
// Looper.prepare() 方法源码
private static void prepare(boolean quitAllowed) {
    // 判断 sThreadLocal 是否为 null，所以一个线程只能调用一次 Looper.prepare()
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }

    sThreadLocal.set(new Looper(quitAllowed));
}
```

Looper 的构造方法

```
//在创建Looper 的时候,直接创建一个MessageQueue
private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}
```

生成 `Looper` 和 `MessageQueue` 对象后，则自动进入消息循环：`Looper.loop()`

使用

```
private Handler mHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {

        switch (msg.what) {
            case xxxx:
                //处理回调
                break;
            default:
                break;
        }
    }
};

public void HandlerTest() {
    //创建Message对象, 建议使用Message.obtain(),而不是直接 new Meessage();
    Message msg = Message.obtain();
    msg.what = "发送一条message消息";
    msg.obj = param;
    mHandler.sendMessage(msg);
}
```

源码分析

```
public Handler() {
    this(null, false);
}
```

```

public Handler(@Nullable Callback callback, boolean async) {
    if (FIND_POTENTIAL_LEAKS) {
        final Class<? extends Handler> klass = getClass();
        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass())
&&
           (klass.getModifiers() & Modifier.STATIC) == 0) {
            Log.w(TAG, "The following Handler class should be static or leaks might
occur: " +
                      klass.getCanonicalName());
        }
    }
    //分析1 获取Looper对象
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
                "Can't create handler inside thread " + Thread.currentThread()
                + " that has not called Looper.prepare()");
    }
    //Looper对象里面创建的队列
    mQueue = mLooper.mQueue;
    //回调,如果传有Callback 进来就赋值, 没有就为null
    mCallback = callback;

    mAsynchronous = async;
}

//分析1
@UnsupportedAppUsage
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

public static @Nullable Looper myLooper() {
    //这里Thradlocal获取一个Looper对象
    return sThreadLocal.get();
}

```

我们看到分析1 需要获取一个Looper对象, 如果Looper为空,那就直接抛异常了, 接下来我们看看这个ThraedLocal是在哪里赋值进去的

```

#Looper.perpare

public static void prepare() {
    prepare(true);
}

private static void prepare(boolean quitAllowed) {

```

```

if (sThreadLocal.get() != null) {
    throw new RuntimeException("Only one Looper may be created per thread");
}
//我们看到ThreadLocal是在这里赋值的
sThreadLocal.set(new Looper(quitAllowed));
}

//这个方法实在ActivityThread.main() 方法里面调用的，这就是为什么我们在UI线程创建Handler 不需要调用
//prepare() 的原因
public static void prepareMainLooper() {
    //这里调用上面的prepare 创建Looper对象
    //里面传入false 表示主线程的Loop 是不能停止的，如果我们在子线程调用perpare()里面传入的是true,
    //所以子线程可以停止loop
    prepare(false);
    synchronized (Looper.class) {
        if (sMainLooper != null) {
            throw new IllegalStateException("The main Looper has already been
prepared.");
        }
        sMainLooper = myLooper();
    }
}

private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}

```

在Looper 对象里会创建MessageQueue队列

通过上面的代码,我们可以知道, 在主线程创建Handler的话, 系统已经帮我们创建好了 Looper 对象, 如果在子线程的创建Handler的话,需要我们调用Looper.prepare() 否则 Looper对象就为空, 就会抛出异常.

同时 new Looper() 的时候会创建MessageQueue()队列, 后续消息发送和获取都是通过这个队列进行的.

```

public static void loop() {
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this
thread.");
    }
    final MessageQueue queue = me.mQueue;

    // Make sure the identity of this thread is that of the local process,
    // and keep track of what that identity token actually is.
}

```

```
Binder.clearCallingIdentity();
final long ident = Binder.clearCallingIdentity();

// Allow overriding a threshold with a system prop. e.g.
// adb shell 'setprop log.looper.1000.main.slow 1 && stop && start'
final int thresholdOverride =
    SystemProperties.getInt("log.looper."
        + Process.myUid() + "."
        + Thread.currentThread().getName()
        + ".slow", 0);

boolean slowDeliveryDetected = false;

for (;;) {
    //从队列中获取消息，如果没有，则退出
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is quitting.
        return;
    }

    final Printer logging = me.mLogging;
    if (logging != null) {
        logging.println(">>>> Dispatching to " + msg.target + " " +
            msg.callback + ": " + msg.what);
    }
    final Observer observer = sObserver;

    final long traceTag = me.mTraceTag;
    long slowDispatchThresholdMs = me.mSlowDispatchThresholdMs;
    long slowDeliveryThresholdMs = me.mSlowDeliveryThresholdMs;
    if (thresholdOverride > 0) {
        slowDispatchThresholdMs = thresholdOverride;
        slowDeliveryThresholdMs = thresholdOverride;
    }
    final boolean logSlowDelivery = (slowDeliveryThresholdMs > 0) && (msg.when >
        0);
    final boolean logSlowDispatch = (slowDispatchThresholdMs > 0);

    final boolean needStartTime = logSlowDelivery || logSlowDispatch;
    final boolean needEndTime = logSlowDispatch;

    if (traceTag != 0 && Trace.isTagEnabled(traceTag)) {
        Trace.traceBegin(traceTag, msg.target.getTraceName(msg));
    }

    final long dispatchStart = needStartTime ? SystemClock.uptimeMillis() : 0;
    final long dispatchEnd;
    Object token = null;
    if (observer != null) {
        token = observer.messageDispatchStarting();
    }
    long origWorkSource = ThreadLocalWorkSource.setUid(msg.workSourceUid);
```

```

try {
    //在这里分发消息，这里的target就是发送发送消息的时候设置的，就是Handler对象，所以这里
//会调用Handler的dispatchMessage() 方法
    msg.target.dispatchMessage(msg);
    if (observer != null) {
        observer.messageDispatched(token, msg);
    }
    dispatchEnd = needEndTime ? SystemClock.uptimeMillis() : 0;
} catch (Exception exception) {
    if (observer != null) {
        observer.dispatchingThrewException(token, msg, exception);
    }
    throw exception;
} finally {
    ThreadLocalWorkSource.restore(origWorkSource);
    if (traceTag != 0) {
        Trace.traceEnd(traceTag);
    }
}
if (logSlowDelivery) {
    if (slowDeliveryDetected) {
        if ((dispatchStart - msg.when) <= 10) {
            Slog.w(TAG, "Drained");
            slowDeliveryDetected = false;
        }
    } else {
        if (showSlowLog(slowDeliveryThresholdMs, msg.when, dispatchStart,
"delivery",
msg)) {
            // Once we write a slow delivery log, suppress until the queue
drains.
            slowDeliveryDetected = true;
        }
    }
}
if (logSlowDispatch) {
    showSlowLog(slowDispatchThresholdMs, dispatchStart, dispatchEnd,
"dispatch", msg);
}

if (logging != null) {
    logging.println("<<<< Finished to " + msg.target + " " + msg.callback);
}

msg.recycleUnchecked();
}
}

```

主线程调用完prepareMainLooper() 创建Looper对象之后，就会调用Loop 不断的从队列中获取消息，如果为空就退出等待消息..

创建完了Handler() 对象，现在我们就可以发送消息了，接着往下看

```

#Handler.sendMessage()

public final boolean sendMessage(@NonNull Message msg) {
    return sendMessageDelayed(msg, 0);
}

public final boolean sendMessageDelayed(@NonNull Message msg, long delayMillis) {
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(@NonNull Message msg, long uptimeMillis) {
    //这个mQueue 就是创建Handler对象时候，里面进行赋值的
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}

private boolean enqueueMessage(@NonNull MessageQueue queue, @NonNull Message msg,
    long uptimeMillis) {
    //这个target 在loop循环里面用来调用dispatchMessage()
    msg.target = this;
    msg.workSourceUid = ThreadLocalWorkSource.getUid();

    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    //调用MessageQueue.enqueueMessage(),把消息放入消息队列中
    return queue.enqueueMessage(msg, uptimeMillis);
}

```

上面的代码使用sendMessage()发送消息，里面判断延时和队列是否为空，把target设置为this，后面在loop循环里面会用来调用dispatchMessage() 和判断同步屏障。

继续往下看

```

boolean enqueueMessage(Message msg, long when) {
    if (msg.target == null) {
        throw new IllegalArgumentException("Message must have a target.");
    }
}

```

```

    if (msg.isInUse()) {
        throw new IllegalStateException(msg + " This message is already in use.");
    }

    synchronized (this) {
        if (mQuitting) {
            IllegalStateException e = new IllegalStateException(
                msg.target + " sending message to a Handler on a dead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }

        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        //如果队列里面没有消息 或 不使用延时, 或者延时时间小于队列里面消息的延时时间
        //插入到队列头部
        if (p == null || when == 0 || when < p.when) {

            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {

            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p; // invariant: p == prev.next
            prev.next = msg;
        }

        //是否需要唤醒
        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}

```

在enqueueMessage 中根据判断条件把消息插入到队列当中, 然后消息就会在loop循环中获取到, 并调用dispatchMessage()方法

```
#Handler.dispatchMessage()

public void dispatchMessage(@NonNull Message msg) {
    //msg.callback 是在post中赋值了,所以如果使用post发送消息,msg.callback就不为空
    if (msg.callback != null) {
        //调用post的run方法
        handleCallback(msg);
    } else {
        //mCallback是创建Handler的时候传进来的,如果没有传callback参数,为null
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        //调用匿名函数里面的handleMessage()
        handleMessage(msg);
    }
}
```

常规的消息 Message 是如何创建的?

我们经常会在 `Handler` 的使用中创建消息对象 `Message`，创建方式也有两个 `new Message()` 或者 `Message.obtain()`。我们通常都更青睐于 `Message.obtain()` 这种方式，因为这样的方式，可以有效避免重复创建 `Message` 对象。实际上在代码中也是显而易见的。

```
public static Message obtain() {
    //用sPoolSync作为Message池,对Message进行复用
    synchronized (sPoolSync) {
        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null;
            m.flags = 0; // clear in-use flag
            sPoolSize--;
            return m;
        }
    }
    //如果消息池为空,则创建一个新的对象
    return new Message();
}
```

Handle.post()

```
public final boolean post(Runnable r)
{
```

```

        return sendMessageDelayed(getPostMessage(r), 0);
    }

//将Runnable 加入到消息队列，用于在dispatchMessage 方法判断是否是post 消息
private static Message getPostMessage(Runnable r) {
    //这个也是使用Message.obtain()方法,而不是直接创建消息
    Message m = Message.obtain();
    m.callback = r;
    return m;
}

public void dispatchMessage(Message msg) {
    //判断是否是post 消息，里面的callback 就是上面保存的
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        //如果创建Handler的时候传入Callback，则这里的mCallback不为空，
        //回调到传入的Callback的handlerMessage方法，否则调用我们实现的handlerMessage方法
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}

```

为什么在oncreate() 方法 可以在子线程更新UI?

因为线程检测方法 checkThread() 是在 ViewRootImpl 中的 requestLayout() 方法中执行的，

ViewRootImpl 是在 activity 的 onResume 方法调用后 才由 WindowManagerGlobal 的 addView 方法创建的，所以在onCreate() 中可以在子线程更新UI，因为 当时还没有创建ViewRootImpl()

Handler、Thread 和 HandlerThread的差别

- **Handler:** 在android中负责发送和处理消息，通过它可以实现其他支线任务与主线程之间的消息通讯。
- **Thread:** Java进程中执行运算的最小单位，亦即执行处理机调度的基本单位。某一进程中一路单独运行的程序。
- **HandlerThread:** 一个继承自Thread的类HandlerThread，Android中没有对Java中的Thread进行任何封装，而是提供了一个继承自Thread的类HandlerThread类，这个类对Java的Thread做了很多便利的封装。HandlerThread继承于Thread，所以它本质就是个Thread。与普通Thread的差别就在于，它在内部直接实

现了Looper的实现，这是Handler消息机制必不可少的。有了自己的looper，可以让我们在自己的线程中分发和处理消息。如果不用HandlerThread的话，需要手动去调用Looper.prepare()和Looper.loop()这些方法。

同步屏障

postSyncBarrier() 方法就是用来插入一个屏障到消息队列

- 屏障消息和普通消息的区别在于屏障没有target，普通消息有target是因为它需要将消息分发给对应的target，而屏障不需要被分发，它就是用来挡住普通消息来保证异步消息优先处理的。
- 屏障和普通消息一样可以根据时间来插入到消息队列中的适当位置，并且只会挡住它后面的同步消息的分发
- postSyncBarrier()返回一个int类型的数值，通过这个数值可以撤销屏障即removeSyncBarrier()。
- postSyncBarrier()是私有的，如果我们想调用它就得使用反射。插入普通消息会唤醒消息队列，但是插入屏障不会。

多线程的优缺点

优点：

- 方便高效的内存共享 - 多进程中内存共享比较不便，且会抵消掉多进程编程的好处
- 较轻的上下文切换开销 - 不用切换地址空间，不用更改CR3寄存器，不用清空TLB
- 线程上的任务执行完后自动销毁

缺点：

- 开启线程需要占用一定的内存空间(默认情况下,每一个线程都占512KB)
- 如果开启大量的线程,会占用大量的内存空间,降低程序的性能
- 线程越多,cpu在调用线程上的开销就越大
- 程序设计更加复杂,比如线程间的通信、多线程的数据共享

结论：

- 不要频繁创建，销毁线程，使用线程池
- 减少线程间同步和通信（最为关键）
- 避免需要频繁共享写的数据
- 合理安排共享数据结构，避免伪共享（false sharing）
- 使用非阻塞数据结构/算法
- 避免可能产生可伸缩性问题的系统调用（比如mmap）
- 避免产生大量缺页异常，尽量使用Huge Page
- 可以的话使用用户态轻量级线程代替内核线程

在生成消息的时候，最好是用 `Message.obtain()` 来获取一个消息，这是为什么呢？

```
// Message.java  
  
public static Message obtain() {  
    synchronized (sPoolSync) {
```

```

    if (sPool != null) {
        Message m = sPool;
        sPool = m.next;
        m.next = null;
        m.flags = 0; // clear in-use flag
        sPoolSize--;
        return m;
    }
}
return new Message();
}

链接:https://juejin.im/post/5eeb06f151882565cd468957

```

可以看到，这是一个消息池，如果消息池不为空，就会从池中获取一个消息，达到复用的效果。

实际开发中应该注意：

多线程可能会引起数据获取异常问题，比如在子线程里面获取数据，在子线程外返回获取的数据，会因为异步原因导致数据未获取到就直接返回预料外的值。特别需要注意需要在子线程获取数据并返回的情况。

开启过多的线程引起卡顿问题，避免开启过多的线程

同步屏障

同步屏障是用来阻挡同步消息执行的。

View.post()原理

```

// view.java
public boolean post(Runnable action) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action);
    }

    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    getRunQueue().post(action);
    return true;
}

```

<https://www.jianshu.com/p/101e2edd29d1>

`view.post()` 中 ,`HandlerActionQueue#executeActions` 在View绘制流程之前执行的, 为什么能够准确的获取到宽高?

Android 系统是基于消息机制运行的, 所有的事件、行为, 都是基于 Handler 消息机制在运行的。所以, 当 `ViewRootImpl#performTraversals` 在执行的时候, 也一定是基于某个消息的。而且, `HandlerActionQueue#executeActions` 执行的时候, 也只是通过 Handler 将 Runnable post 到了 UI 线程等待执行

总结:

1. 在当前 View attach 到 Window 之前, 会自己先维护一个 HandlerActionQueue 对象, 用来存储当前的 Runnable 对象, 然后等到 Attach 到 Window 的时候(也就是 ViewRootImpl 执行到 `performTraversals` 方法时), 会统一将 Runnable 转交给 ViewRootImpl 处理;
2. 而在 View#dispatchAttachedToWindow 时, 也会为当前 View 初始化一个 AttachInfo 对象, 该对象持有 ViewRootImpl 的引用, 当 View 有此对象后, 后续的所有 Runnable 都将直接交给 ViewRootImpl 处理;
3. 而 ViewRootImpl 也会在执行 `performTraversals` 方法, 也会调用 ViewRootImpl#`getRunQueue`, 利用 ThreadLocal 来为主线程维护一个 HandlerActionQueue 对象, 至此, ViewRootImpl 内部都将使用该队列来进行 Runnable 任务的短期维护;
4. 但需要注意的是, 各个 View 调用的 post 方法, 仍然是由各自的 HandlerActionQueue 对象来入队任务的, 然后在 View#`dispatchAttachedToWindow` 的时候转移给 ViewRootImpl 去处理。

大图加载

https://blog.csdn.net/guolin_blog/article/details/9316683

1. 对图片进行压缩, 根据控件大小进行合适的压缩
2. 一次性加载大量的大图时, 使用缓存

View机制

Activity,Window,DecorView的关系

- Activity包含一个PhoneWindow
- PhoneWindow 是 Window的实现类
- Activity通过`setContentView`将View 设置到PhoneWindow上
- PhoneWindow里面包含DecorView, 最终被添加到DecorView上

`getX / getY` 返回的是相对于当前View左上角的x 和 y 坐标, 而`getRawX / getRawY`返回的是相对于手机屏幕左上角的x 和 y 坐标

view的滑动

三种方式:

1. 通过View本身提供的`ScrollTo / ScrollBy` 方法来实现滑动
2. 通过动画给View施加平移效果来实现滑动
3. 通过改变View的LayoutParams 使得 View 重新布局从而实现滑动

事件分发

`dispatchTouchEvent` : 用来事件分发, 表示是否消耗当前事件

`onInterceptTouchEvent` : 用来判断是否拦截某事件，如果当前View拦截了某个事件，那么在同一个事件序列当中，此方法不会被再次调用，返回结果表示是否拦截当前事件。

`onTouchEvent` : 在 `dispatchTouchEvent` 方法中调用，用来处理事件，返回结果表示是否消耗当前事件，如果不消耗，则在同一个事件序列中，当前View无法再次接收到事件。

伪代码

```
public boolean dispatchTouchEvent(MotionEvent ev) {  
    boolean consume = false;  
    if (onInterceptTouchEvent(ev)) {  
        // 拦截 调用当前 onTouchEvent方法  
        consume = onTouchEvent(ev);  
    } else {  
        // 不拦截 传递给子类的 dispatchTouchEvent()方法  
        consume = child.dispatchTouchEvent(ev);  
    }  
    return consume;  
}
```

View处理事件时，如果它设置了OnTouchListener 那么 OnTouchListener () 中的onTouch方法就会被调用，如果返回false，则当前的View的 onTouchEvent方法会被调用，如果返回true 那么onTouchEvent方法将不会被调用。

在onTOuchEvent 方法中，如果当前设置的有OnClickListener 那么它的onClick方法会被调用，可以看出，OnClickListener 其优先级最低，处于事件传递的尾端。

如果一个View的onTouchEvent返回false, 那么它的父容器的 onTouchEvent将被会调用。

View 没有 onInterceptTOuchEvent 方法，一旦有点击事件传递给它，那么它的 onTouchEvent就会被调用。View 的onTouchEvent默认都会消耗事件，除非它是不可点击的 clickable 和 longClickable 同时为 false；

onInterceptTouchEvent 不是每次事件都会被调用的，如果我们想提前处理所有的点击事件，要选择 dispatchTouchEvent 方法，只有这个方法能确保每次都会被调用，当前前提是事件能够传递到当前的ViewGroup

View 原理

measure 过程决定了View的宽、高。Measure 完成后，可以通过 `getMeasuredWidth` 和 `getMeasuredHeight` 方法来获取到View 测量后的宽高。在几乎所有情况下都等于View的最终的宽高。

viewGroup 并没有定义其测量的具体过程，因为viewGroup 是一个抽象类，其测量过程需要各个子类去实现。比如LinearLayout 和 RelativeLayout .

LinearLayout的 measure 中会调用 `measureChildBeforeLayout`，在这个方法中会调用子类的 `measure` 方法，这样每次子元素就依次进入 `measure` 方法测量，每测量一个子元素，就会通过 `mTotalLength` 这个变量来存储高度，当子元素测量完之后，LinearLayout 会测量自己的大小。

Layout 作用是用来确定View在父容器中的放置位置，Layout过程决定了View 的四个顶点的坐标和实际View的宽高，可以通过 `getTop`, `getBottom` , `getLeft`, `getRight` 来拿到View的四个顶点的位置。`onLayout` 方法的用途是父容器确定子元素的位置。

view 和 viewGroup 均没有实现 onLayout 方法,

Draw 过程决定了 View 的显示, 只有 draw 方法完成后 View 的内容才能现在在屏幕上。

- 绘制背景 background.draw(canvas)
- 绘制自己 (onDraw)
- 绘制 children (dispatchDraw)
- 绘制装饰 (onDrawScrollBars)

view 有一个特殊的方法 setWillNotDraw, 如果一个 View 不需要绘制任何内容, 那么设置这个标志位为 true 以后, 系统会进行相应的优化。默认情况下 View 没有启用这个优化标志位, 但是 ViewGroup 会默认启用这个优化标志位。

在实际开发中, 当我们自定义控件继承与 ViewGroup 并且自身不具备绘制功能时, 就可以开启这个标志位 从而便于系统进行后续的优化。当明确一个 ViewGroup 需要通过 onDraw 来绘制内容时, 我们需要显式的关闭 WILL_NOT_DRAW 这个标志位。

在某些极端情况下, 系统可能需要多次 measure 才能确定最终的测量宽/高, 在这种情形下, 在 onMeasure 方法中拿到的测量宽/高可能是不准确的, 一个比较好的习惯是在 onLayout 方法中取获取 View 测量宽/高或者最终 宽/高

如果需要获取某个 View 的宽高, 在 onCreate onStart onResume 方法中均无法正确的得到某个 View 的宽高信息, 这是因为 View 的 measure 过程和 Activity 的生命周期方法不是同步执行的, 因此无法保证执行完 onCreate onStart 或者 onResume 执行完毕时 某个 View 已经测量完毕。

解决办法:

onWindowFocusChanged, 这时候 View 已经初始化完毕, 可以正确的取获取宽高。但是要注意的是 onWindowFocusChanged 会被调用多次, 当 Activity 得到焦点 和 失去焦点时都会被调用一次

使用方法

```
public void onWindowFocusChanged(boolean hasFocus) {  
    super.onWindowFocusChanged(hasFocus);  
    if (hasFocus) {  
        int width = view.getMeasuredWidth();  
        int height = view.getMeasuredHeight();  
    }  
}
```

2. view.post(runnable)

通过 post 可以将一个 runnable 投递到消息队列的尾部, 然后等待 Looper 调用此 runnable 的时候, View 也已经初始化好了。

```

protected void onStart() {
    super.onStart();
    view.post(new Runnable() {
        @Override
        public void run() {
            int width = view.getMeasuredWidth();
            int height = view.getMeasuredHeight();
        }
    });
}

```

view 的 getMeasuredWidth 和 getWidth 这个方法有什么区别?

在view的默认实现中, View的 getMeasuredWidth 和 getWidth 是相等的, 只不过测量宽高(getMeasuredWidth)形成于View的measure 过程, 而最终宽高(getWidth)形成于 layout 过程, 即两者的赋值时机不同, getMeasuredWidth 更早一些.

MeasureSpec

MeasureSpec 参与了 View 的 measure 过程

对于顶级View(DecorView), 其MeasureSpec由窗口尺寸和其自身的LayoutParams 来共同决定的, 对于普通View, 其MeasureSpec由父容器的MeasureSpec和自身的LayoutParams来共同决定的。所以在继承View的自定义控件的时候, 需要重写onMeasure方法并设置wrap_content时的自身大小, 否则在布局中使用wrap_content就相当于使用match_parent.

specMode : 测量模式

- UNSPECIFIED : 父容器不对View有任何限制, 要多大给多大, 这种情况一般用于系统内部, 表示一种测量状态
- EXACTLY : 父容器已经检测出View 所需要的精确大小, 这个时候View的最终大小就是SpecSize 所指定的值, 它对应于LayoutParams中的match_parent和具体的数值这两种模式
- AT_MOST : 父容器指定了一个可用大小即SpecSize , View 的大小不能大于这个值, 具体是什么值要看不同的View的具体实现, 它对应于LayoutParams 中的 wrap_content

SpecSize : 某种测量模式下的规格大小

直接继承View的自定义控件需要重写onMeasure 方法 并设置wrap_content 时的自身大小, 否则在布局中使用wrap_content就相当于使用match_parent

```

public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    switch (specMode) {
        case MeasureSpec.UNSPECIFIED:
            result = size;
            break;
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
            result = specSize;
    }
}

```

```

        break;
    }
    return result;
}

```

ViewGroup默认情况下，会被设置成WILL_NOT_DRAW，这是从性能考虑，这样一来，onDraw就不会被调用了。

2) 如果我们要重写一个ViewGroup的onDraw方法，有两种方法：

1. 在构造函数里面，给其设置一个颜色，如#00000000
2. 在构造函数里面，调用setWillNotDraw(false)，去掉其WILL_NOT_DRAW flag。

parentSpecMode	EXACTLY	AT_MOST	UNSPECIFIED
childLayoutParams			
dp/px	EXACTLY childSize	EXACTLY childSize	EXACTLY childSize
match_parent	EXACTLY parentSize	AT_MOST parentSize	UNSPECIFIED 0
wrap_content	AT_MOST parentSize	AT_MOST parentSize	UNSPECIFIED 0

注：parentSize 为父容器中目前可使用的大小 <http://blog.csdn.net/singwhatiwanna>

自定义View

让View支持wrap_content

如果直接继承View 或者 ViewGroup控件，如果不重写 onMeasure 并对 wrap_content 做特殊处理，那么当外界在布局中使用wrap_content 时，就无法达到预期效果。 (详见源码)

如果有必要，让你的View支持padding

直接继承View的控件，如果不在draw方法中处理padding，那么padding属性是无法起作用的。另外，直接继承自 ViewGroup 的控件需要在 onMeasure 和 onLayout 中考虑 padding 和 子元素 的 margin 对其造成的影响，不然将导致padding 和 子元素的 margin失效。

尽量不要在View中使用Handler， 没必要

这是因为View 内部本身就提供了post系列的方法，完全可以起到Handler的作用，当然除非你很明确的要使用 Handler来发送消息。

View中如果有线程或者动画，需要及时停止。 参考View #onDetachedFromWindow

如果有线程或者动画需要停止时，那么 `onDetacheFromWindow` 是一个很好的时机，当包含此View 的 Activity 退出或者当前View被remove时，View的 `onDetacheFromWindow` 方法会被调用，和此方法对应的是 `onAttachedToWindow`，当包含此View的Activity启动时，View 的`onAttacheToWindow` 方法会被调用。同时，当View变得不可见时我们也要停止线程和动画，如果不及时处理这种问题，有可能会造成内存泄露。

view 的绘制过程的传递是通过`dispatchDraw` 来实现的, `dispatchDraw` 会遍历调用所有的元素的`draw` 方法，这样 `draw` 事件就一层一层的传递下去.

自定义View和自定义ViewGroup的区别

如果只是一个原始的view, 那么通过 `measure` 方法就完成其测量过程, 如果是一个 `ViewGroup` , 除了完成自己的测量过程外, 还要去遍历所有的子元素的 `measure` 方法, 各个子元素再递归回去执行这个流程.

自定义属性

1. 在`values`目录下创建自定义属性的XML, 名字不限制, 不过最好使用`attrs`开头的名字比较规范. 里面的内容如下:

```
<declare-styleable name="CricleView">
    <attr name="cricle_color" format="color"/>
</declare-styleable>
```

2. 在View 的构造方法中解析自定义属性值并做相应处理,

```
public CricleView(Context context, AttributeSet attrs) {
    super(context, attrs);
    TypedArray cl = context.obtainStyledAttributes(attrs, R.styleable.CricleView);
    mColor = cl.getColor(R.styleable.CricleView_cricle_color, Color.BLUE);
    cl.recycle();
}
```

3. 在布局中使用自定义属性

```
<com.ljy.ofoview.utils.View.CricleView
    android:layout_width="match_parent"
    android:layout_height="100dp"
    app:cricle_color="@color/colorAccent"
/>
```

SurfaceView和View的区别?

- View需要在UI线程对画面进行刷新，而SurfaceView可在子线程进行页面的刷新
- View适用于主动更新的情况，而SurfaceView适用于被动更新，如频繁刷新，这是因为如果使用View频繁刷新会阻塞主线程，导致界面卡顿
- SurfaceView在底层已实现双缓冲机制，而View没有，因此SurfaceView更适用于需要频繁刷新、刷新时数据处理量很大的页面（如视频播放界面）

如何实现View的弹性滑动？

View 滑动冲突解决

外部拦截法

在父类进行拦截处理，需要重写 `onInterceptTouchEvent` 方法，在方法内部做相应的拦截即可

```
public boolean onInterceptTouchEvent(MotionEvent event) {  
    boolean intercepted = false;  
    int x = event.getX();  
    int y = event.getY();  
    switch (event.getAction()) {  
  
        case MotionEvent.ACTION_DOWN: {  
            intercepeted = false;  
            break;  
        }  
  
        case MotionEvent.ACTION_MOVE: {  
            if(父类需要点击事件) {  
                intercepted = true;  
            } else {  
                intercepted = false;  
            }  
            break;  
        }  
        case MotionEvent.ACTION_UP: {  
            intercepted = false;  
            break;  
        }  
        default:  
            break;  
    }  
    mLastXIntercept = x;  
    mLastYIntercept = y;  
    return intercepted;  
}
```

内部拦截法

父容器不拦截任何事件，所有的事件都传递给子元素，如果子元素需要此事件就直接消耗掉。否则由父容器进行处理。需要在子类使用 `requestDisallowInterceptTouchEvent` 方法才能正常工作

重写子元素的`dispatchTouchEvent()` 方法

```
public boolean dispatchTOuchEvent(MotionEvent event) {  
    int x = event.getX();  
    int y = event.getY();
```

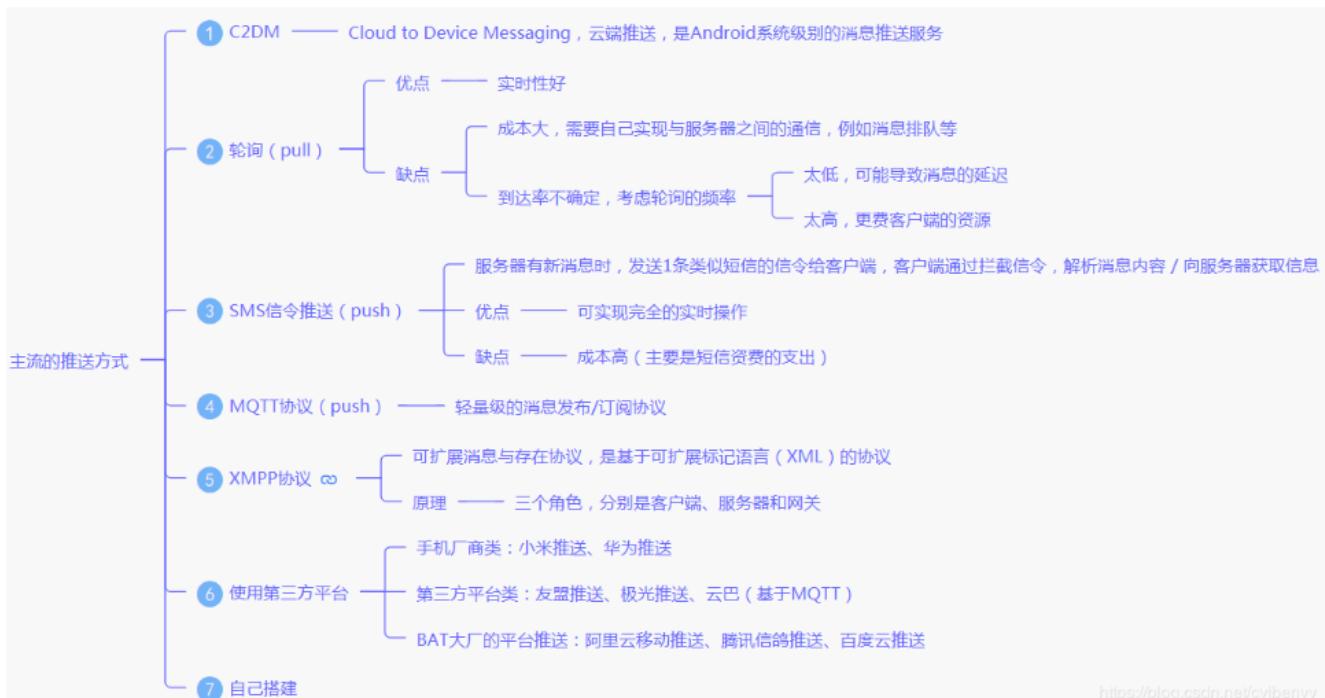
```
switch(event.getAction()) {  
    case MotionEvent.ACTION_DOWN: {  
        parent.requestDisallowInterceptTouchEvent(true);  
        break;  
    }  
    case MotionEvent.ACTION_MOVE: {  
        int deltaX = x - mLastX;  
        int deltaY = y - mLastY;  
        if(父容器需要此点击事件) {  
            parent.requestDisallowInterceptTouchEvent(false);  
        }  
        break;  
    }  
  
    case MotionEvent.ACTION_UP: {  
        break;  
    }  
    default:  
        break;  
}  
mLastX = x;  
mLastY = y;  
return super.dispatchTouchEvent(event);  
}
```

推送

Android平台在不使用GCM的情况下就需要将自己的服务器或是第三方推送服务提供商的服务器与设备建立一条长连接，通过长连接进行推送。

AlarmManager 是Android 系统封装的用于管理RTC 的模块，RTC (Real Time Clock) 是一个独立的硬件时钟，可以在CPU 休眠时正常运行，在预设的时间到达时，通过中断唤醒CPU。这意味着，如果我们用AlarmManager 来定时执行任务，CPU 可以正常的休眠，只有在需要运行任务时醒来一段很短的时间。极光推送的Android SDK 就是基于这种技术实现的。

在android客户端使用Push推送时，应该使用AlarmManager来实现心跳功能，使其真正实现长连接。



短连接：通讯双方有数据的时候建立连接，数据传输完毕就断开连接。

长连接：大家建立连接后，不主动断开，双方互相发送数据，发完了也不断开，之后有需要发送的数据就继续通过这个连接发送。

TCP连接在默认的情况下就是所谓的**长连接**, 也就是说连接双方都不主动关闭连接, 这个连接就应该一直存在.

Android 动画 (ObjectAnimator)

View动画

支持四种动画效果：

- 平移动画 (TranslateAnimation)
- 缩放动画 (ScaleAnimation)
- 旋转动画 (RotateAnimation)
- 透明动画 (AlphaAnimation)

`android:interpolator` : 动画集合所采用的插值器，插值器影响动画的速度

`android:shareInterpolator` : 表示集合中的动画是否和集合共享同一个插值器，如果集合不指定插值器，那么子动画就需要单独指定所需的插值器或者使用默认值。

更多方法请查看文档.

LayoutAnimation

LayoutAnimation 作用于 **ViewGroup**，为 **ViewGroup** 指定一个动画，这样当它的子元素出场时都会具有这种动画效果。这种效果常常被使用在 **ListView** 上

定义 LayoutAnimation

```
<layoutAnimation
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="0.5"
    android:animationOrder="normal"
    android:animation="anim/anim_item"/>
```

android:delay： 表示子元素动画延迟时间，比如子元素入场动画的时间周期为300ms, 那么0.5 表示每个子元素都需要延迟150ms 才能播放入场动画

android:animationOrder : 表示子元素的动画顺序： normal （顺序显示） , reverse(逆向显示), random(随机播放)

android:animation: 指定具体的入场动画

Activity 切换效果

```
Intent intent = new Intent(this, testActivity.class);
startActivity(intent);
overridePendingTransition(R.anim.enter_anim, R.anim.exit_anim);
//这个方法必须在startActivity之后被调用才能生效

//Activity 退出时
@Override
public void finish() {
    super.finish();
    overridePendingTransition(R.anim.enter_anim, R.anim.exit_anim);
}
```

overridePendingTransition这个方法必须位于startActivity 或者 finish 的后面 否则动画效果将不起作用。

帧动画

通过 AnimationDrawable 来使用帧动画

使用帧动画的时候，尽量避免使用过多尺寸较大的图片，否则容易引起OOM

补间动画

缺点：

- 只能够实现移动、缩放、旋转和淡入淡出这四种动画操作。
- 只能够作用在View上面。
- 只是改变View的显示效果，而不会真正的去改变View的属性。

属性动画

属性动画是API 11 新加入的特性， 属性动画可以对任意对象的属性进行动画，而不仅仅是View

ValueAnimator

整个属性动画机制中最核心的一个类。负责管理动画的播放次数、播放模式、以及对动画设置监听器等

ObjectAnimator

继承自ValueAnimator，可以直接对任意对象任意属性进行动画操作，底层动画实现机制是由ValueAnimator实现。

ObjectAnimator内部的工作机制并不是直接对我们传入的属性名进行操作的，而是会去寻找这个属性名对应的get和set方法，因此alpha属性所对应的get和set方法应该就是：

```
ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "scaleY", 1f, 3f, 1f);
animator.setDuration(5000);
animator.start();

public void setAlpha(float value);
public float getAlpha()
```

AnimatorSet

组合动画

- after(Animator anim) 将现有动画插入到传入的动画之后执行
- after(long delay) 将现有动画延迟指定毫秒后执行
- before(Animator anim) 将现有动画插入到传入的动画之前执行
- with(Animator anim) 将现有动画和传入的动画同时执行

```
ObjectAnimator moveIn = ObjectAnimator.ofFloat(textview, "translationX", -500f, 0f);
ObjectAnimator rotate = ObjectAnimator.ofFloat(textview, "rotation", 0f, 360f);
ObjectAnimator fadeInOut = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f);
AnimatorSet animSet = new AnimatorSet();
animSet.play(rotate).with(fadeInOut).after(moveIn);
animSet.setDuration(5000);
animSet.start();
```

TypeEvaluator

ViewPropertyAnimator

```
textview.animate().x(500).y(500).setDuration(5000);
```

`ViewPropertyAnimator`是支持连缀用法的，我们想让`textview`移动到横坐标500这个位置上时调用了`x(500)`这个方法，然后让`textview`移动到纵坐标500这个位置上时调用了`y(500)`这个方法，将所有想要组合的动画通过这种连缀的方式拼接起来，这样全部动画就都会一起被执行。接口中使用了隐式启动动画的功能，只要我们将动画定义完成后，动画就会自动启动。

插值器 和 估值器

插值器(TimeInterpolator)：根据时间流逝的百分比来计算出当前属性值改变的百分比

- `LinearInterpolator` 线性（匀速）
- `AccelerateInterpolator` 持续加速
- `DecelerateInterpolator` 持续减速
- `AccelerateDecelerateInterpolator` 先加速后减速
- `OvershootInterpolator` 结束时回弹一下
- `AnticipateInterpolator` 开始回拉一下
- `BounceInterpolator` 结束时Q弹一下
- `CycleInterpolator` 来回循环

估值器(TypeEvaluator)：根据当前属性改变的百分比计算改变后的属性值

先由插值器根据时间流逝的百分比计算出目标对象的属性改变的百分比，再由估值器根据插值器计算出来的属性改变的百分比计算出目标对象属性对应类型的值。

属性动画监听器

`AnimatorListener` 可以监听动画的开始、结束、取消以及重复播放

`AnimatorUpdateListener` 它会监听整个动画的过程，动画是由许多帧组成的，每播放一帧，`onAnimationUpdate` 就会被调用一次。

属性动画原理：

属性动画要求动画作用的对象提供该属性的 `get` 和 `set` 方法，属性动画根据外界传递的该属性的初始值和最终值，以动画的效果多次去调用`set`方法，每次传递给`set`方法的值都不一样，随着时间的推移，所传递的值越来越接近最终值。

对`object` 的属性 `abc` 做动画，如果想要动画生效，要同时满足下面两个条件：

1. `object` 方法 必须要提供`setAbc` 方法，如果动画的时候没有传递初始值，那么还要提供`getAbc` 方法，因为系统要去取`abc`属性的初始值（如果不满足，程序会直接Crash）
2. `object` 的 `setAbc` 对属性 `abc` 所做的改变必须能够通过某种方法反映出来，比如会带来UI的改变之类的

`ValueAnimator` : `ValueAnimator` 本身不作用于任何对象，也就是说直接使用它没有任何效果，它可以对一个值做动画，然后我们可以监听其动画过程，在动画过程中修改我们的对象的属性值。

使用动画注意事项

1. 尽量避免使用帧动画，当图片数量较多且图片较大时就极容易出现OOM
2. 开启无限循环的动画之后，要在Activity退出时及时停止，否则将导致Activity无法释放从而造成内存泄露。
3. View动画并不是真正的改变View的状态，因此有时候会出现动画完成后View无法隐藏的现象，即 `setVisibility(View.GONE)`无效，这时候只需要调用`view.clearAnimation()`清除View动画即可解决此问题。
4. 开启硬件加速会提高动画的流畅性

泛型擦除

在泛型类被类型擦除的时候，之前泛型类中的类型参数部分如果没有指定上限，如 `<T>` 则会被转译成普通的 Object 类型，如果指定了上限如 `<T extends String>` 则类型参数就被替换成类型上限

泛型类或者泛型方法中，不接受 8 种基本数据类型

Zxing扫码

<https://github.com/jenly1314/ZXingLite>

JNI 和 NDK

JNI，即 Java Native Interface (Java本地接口)，它是为了方便Java调用C/C++等本地代码所封装的一层接口，意味着 Java 和 其他语言之间可以相互调用，不止局限于 Java 调用其他语言，其他语言也可以主动调用 Java .

NDK是Android 所提供的工具集合, 通过NDK 可以在Android中更加方便的通过JNI来访问本地代码, 比如C/C++ , NDK还提供交叉编译器, 开发人员只需要简单的修改就能生成特定的CPU平台的动态库, 使用NDK有以下优点:

1. 提高代码的安全性, 因为so库反编译比较困难,所以NDK提高了Android 程序的安全性
2. 可以很方便的使用目前已有的C/C++开源库

JNIEnv:

JNIEnv 表示 Java 调用 native 语言的环境，是一个封装了几乎全部 JNI 方法的指针。

JNIEnv 只在创建它的线程生效，不能跨线程传递，不同线程的 JNIEnv 彼此独立。

native 环境中创建的线程，如果需要访问 JNI，必须要调用 `AttachCurrentThread` 关联，并使用 `DetachCurrentThread` 解除链接。

JavaVm

JavaVM 是虚拟机在 JNI 层的代表，一个进程只有一个 JavaVM，所有的线程共用一个 JavaVM

Java类型	签名（描述符）
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
void	V
其它引用类型	L + 全类名 + ;
type[]	[
method type	(参数)返回值*

* 和 &

* 用于定义一个指针： `type *var_name;`， `var_name` 是一个指针变量，如 `int *p;`

用于对一个指针取内容： `*var_name`， 如 `*p` 的值是 `1`。

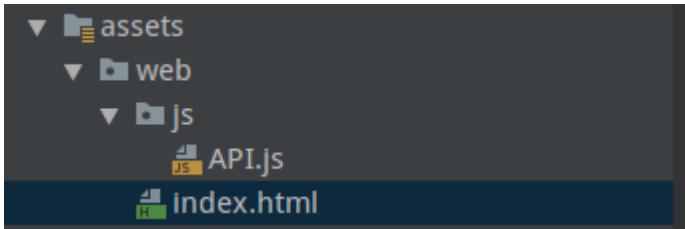
& 是一个取值符号

其用于获取一个变量所在的内存地址。如 `&a;` 的值是 `a` 所在内存的位置，即 `a` 的地址。

shell 命令 无权限 请添加 `chmod +x test.sh`

Android WebView 和 JS 交互

1. 在assets下创建一个HTML文件



2. 用web加载这个html

```
@Override  
protected void onCreate(@Nullable Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_web_to_js);  
  
    setWebview();  
    initView();  
}  
private void setWebview() {  
    webview = new WebView(getApplicationContext());  
    FrameLayout.LayoutParams params = new FrameLayout.LayoutParams  
        (ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.MATCH_PARENT);  
    webview.setLayoutParams(params);  
    FrameLayout content = findViewById(R.id.webview);  
    content.addView(webview);  
    WebSettings webSettings = webview.getSettings();  
    //是否开启JavaScript  
    webSettings.setJavaScriptEnabled(true);  
    webview.loadUrl("file:///android_asset/web/index.html");  
    //把当前对象和js对象进行映射，js可以通过里面的name调用当前对象的方法  
    webview.addJavascriptInterface(object: this, name: "Android");  
}
```

```
private void initView() {  
    tvJs = findViewById(R.id.tv_androidcalljs);  
    tvJsArgs = findViewById(R.id.tv_androidcalljsargs);  
    tvShowmsg = findViewById(R.id.tv_showmsg);  
  
    tvJs.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            webview.evaluateJavascript(script: "javascript:JSSendMessageToWeb()", resultCallback: null);  
        }  
    });  
  
    tvJsArgs.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            webview.evaluateJavascript(script: "javascript:javacalljswith(" + "'Android传过来的参数'" + ")", resultCallback: null);  
        }  
    });  
}
```

然后 `web.load()` 或者 `webview.evaluateJavascript()` 调用JS的方法

```
API.js x
*.js files are supported by IntelliJ IDEA Ultimate Edition
Check IntelliJ IDEA
1 //JS调用Java的方法
2     function JSSendMessageToApp() {
3         try {
4             Android.JSSendMessageToApp();
5         } catch (e){
6         }
7
8
9 }
10 //Java调用js的方法
11     function JSSendMessageToWeb(data) { //提供给APP的交互接口
12
13         document.getElementById("showmsg").innerHTML = "JAVA调用了JS";
14
15 }
16
```

通过上面的 `webView.evaluateJavascript()` 就可以加载到 JS 中对应的方法了

```
//JS调用Java的方法
function JSSendMessageToApp() {
    try {
        Android.JSSendMessageToApp(); 
    } catch (e){
    }

}

//Java调用js的方法
function JSSendMessageToWeb(data) { //提供给APP的交互接口

    document.getElementById("showmsg").innerHTML = "JAVA调用了JS";
}
```

JS中调用Java的方法

JS中调用Java的方法

只需要在JS的方法中，使用映射时候设置的name，加上Java中带有 `@JavascriptInterface` 标志的方法名 即可以调用。

```
tvJs.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        webview.evaluateJavascript(script: "javascript:JSSendMessageToApp();");
    }
});

tvJsArgs.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        webview.evaluateJavascript(script: "javascript:jsCallAndroidArgs('Hello');");
    }
}

/***
 * js调用此方法返回数据
 */
@JavascriptInterface
public void JSSendMessageToApp(){
    tvShowmsg.setText("Js调用Android方法");
}

@JavascriptInterface
public void jsCallAndroidArgs(String args) { tvShowmsg.setText(args); }
```

Java中要通过JS调用的方法，必须加上 `@JavascriptInterface` 标志，才能够进行调用

加密

对称加密：就是客户端和服务端共用同一个密钥，该密钥可以用于加密一段内容，同时也可用于解密这段内容。对称加密的优点是加解密效率高，但是在安全性方面可能存在一些问题，因为密钥存放在客户端有被窃取的风险。

非对称加密：它将密钥分成了两种：公钥和私钥。公钥通常存放在客户端，私钥通常存放在服务器。使用公钥加密的数据只有用私钥才能解密，反过来使用私钥加密的数据也只有用公钥才能解密。非对称加密的优点是安全性更高，因为客户端发送给服务器的加密信息只有用服务器的私钥才能解密，因此不用担心被别人破解，但缺点是加解密的效率相比于对称加密要差很多。

https加密原理：

//<https://mp.weixin.qq.com/s/DGIkZT26CBafJzpQgrqqdQ>

HTTP1.0 和 HTTP 2.0 的区别

1. 缓存处理，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag, If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
2. 带宽优化及网络连接的使用，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它

允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3. 错误通知的管理，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
4. Host头处理，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。
5. 长连接，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection: keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

区别：

HTTP	HTTPS
	需要到ca申请证书，一般免费证书较少，因而需要一定费用
是超文本传输协议，信息是明文传输	具有安全性的ssl加密传输协议
80端口	443端口
连接简单,是无状态的	

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输

https是通过非对称加密和对称加密两种加密方式来进行数据传输的。

网站为了把对称加密的密钥安全的给到浏览器，首次会使用非对称加密来传输密钥，之后就采用对称加密来传输数据。

那如何保证非对称加密的公钥不会被篡改呢？

首先网站向CA机构申请数字证书，证书制作完成后，CA机构则会使用我们提交的公钥，再加上一系列其他的信息，如网站域名、有效时长等，来制作证书。CA机构会使用自己的私钥对其加密，并将加密后的数据返回给我们，我们只需要将获得的加密数据配置到网站服务器上即可。

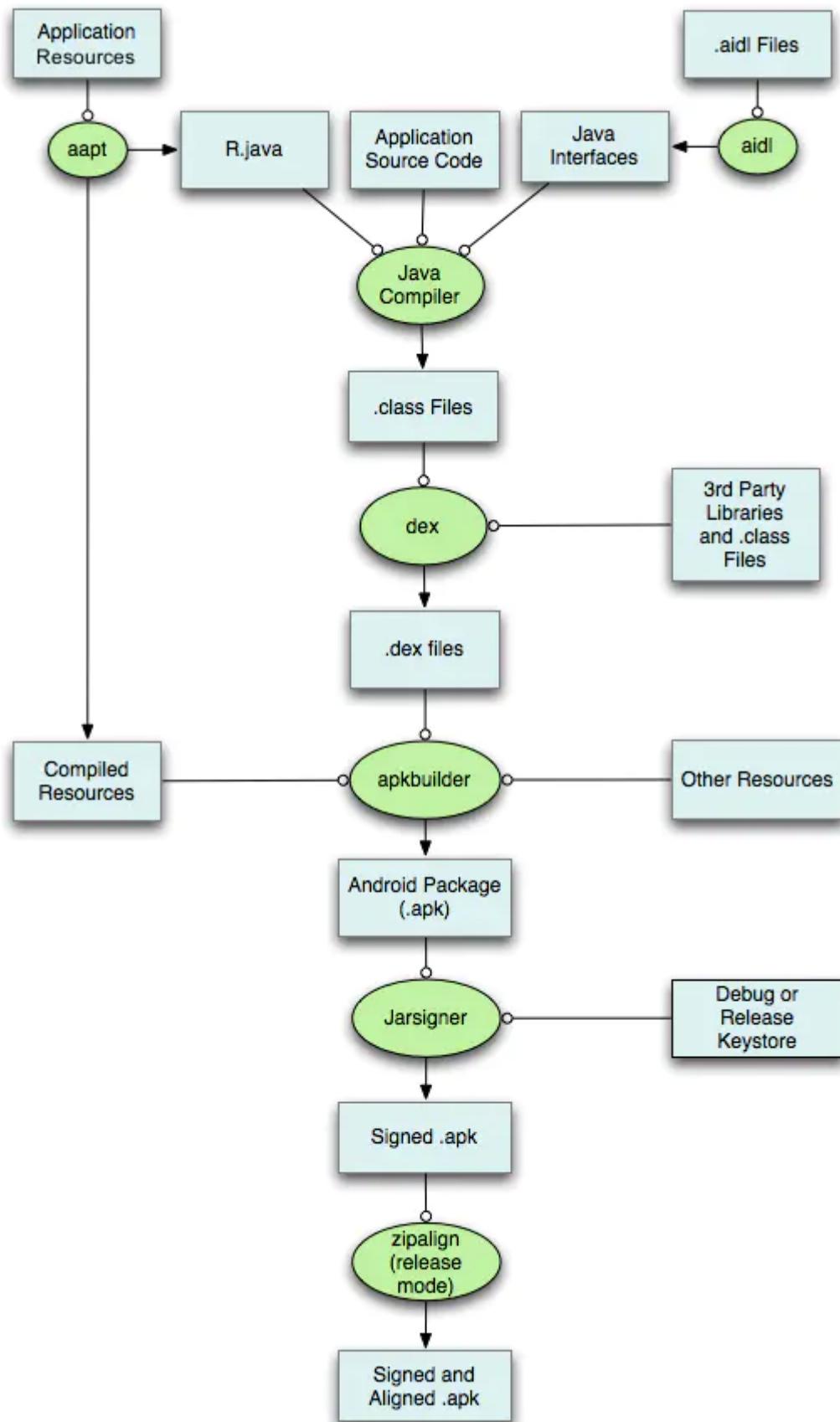
然后浏览器请求网站的时候就可以获得网站返回给我们的加密数据，此时浏览器会使用CA机构的公钥来对这段数据进行解密。如果解密成功就可以获得CA机构给该网站颁发的证书，其中包括网站给我们的公钥。这时候就可以用这公钥拿到网站对称加密的密钥，然后使用对称加密进行数据传输。

如何安全的获取到CA机构给我们的公钥？

任何正版操作系统都会将所有主流CA机构的公钥内置到操作系统当中，所以我们不用额外获取，解密时只需遍历系统中所有内置的CA机构的公钥，只要有任何一个公钥能够正常解密出数据，就说明它是合法的。

当然CA机构为了防止攻击者使用正规的CA机构的证书来替换浏览器请求时获取的证书，证书中加入了多项数据进行辅助校验，比如网站的域名。这样即使加密数据被成功解密，但是最终解密出来的证书中包含的域名和浏览器的请求地址对不上，那么此时浏览器就会显示异常页面。

Android 打包流程



1. 打包资源文件 生成R.java

应用中的资源文件会通过 aapt 工具 打包 , 在这个过程中,项目中的AndroidManifest.xml 文件和 布局文件 都会编译, 然后生成R.java 文件, AndroidMainifrst.xml 会被编译成二进制

应用中的 res 目录下的资源文件, 在APP 打包前大多会被编译, 变成二进制文件, 并会为每个该类文件赋予一个 resource id. 对于该类资源文件的访问, 应用层的代码则是通过resource id 进行访问的. 通过aapt 工具编译后会生成一个resource.arsc文件, resource.arsc 文件相当于一个文件的索引表, 记录很多资源相关的信息.

2. 处理aidl 文件 生成相应的java文件

使用aidl 工具 处理 aidl文件, aidl 工具解析接口定义文件然后 生成相应的Java 代码接口 供程序调用

3. 编译项目源代码 生成 class 文件

项目中的所有Java 代码, R.java 文件 ,aidl 文件 都会被Java编译器(Javac) 编译成 .class 文件

4. 将所有 .class 文件 转换生成classes.dex 文件

使用dex 工具生成 可供Android dalvik虚拟机 执行的classes.dex 文件

任何第三方的libraries 和 .class 文件都会转换成 .dex 文件 , dx工具的主要工作是将 Java字节码 转成 Dalvik 字节码、压缩常量池、消除冗余信息等。

5. 打包生成 APK 文件

打包的工具是 apkbuilder , 所有没有编译的资源, 如images , assets目录下的资源 (该类文件是一些原始文件, APP打包时并不会对其进行编译, 而是直接打包到APP中, 对于这一类资源文件的访问, 应用层代码需要通过文件名对其进行访问)

编译过的资源文件和 .dex 文件都会被 apkbuilder 工具打包到最终的 .apk 文件中.

6. 对APK 文件进行签名

只有被签名过的apk文件才能被安装到设备上

7. 对签名后的 APK 文件进行对齐处理

如果发布的是正式版的 apk 需要使用 zipalign工具进行对齐.

对齐的主要过程是将apk包中所有的资源文件距离文件起始偏移4字节整数倍, 这样通过内存映射访问 apk 文件时的速度会快一些, 对齐的作用就是减少运行时的内存的使用.

Android 打包 v1 v2 区别

<https://zhuanlan.zhihu.com/p/89126018>

v1签名：基于JAR签名，仅针对单个ZIP条目进行验证，

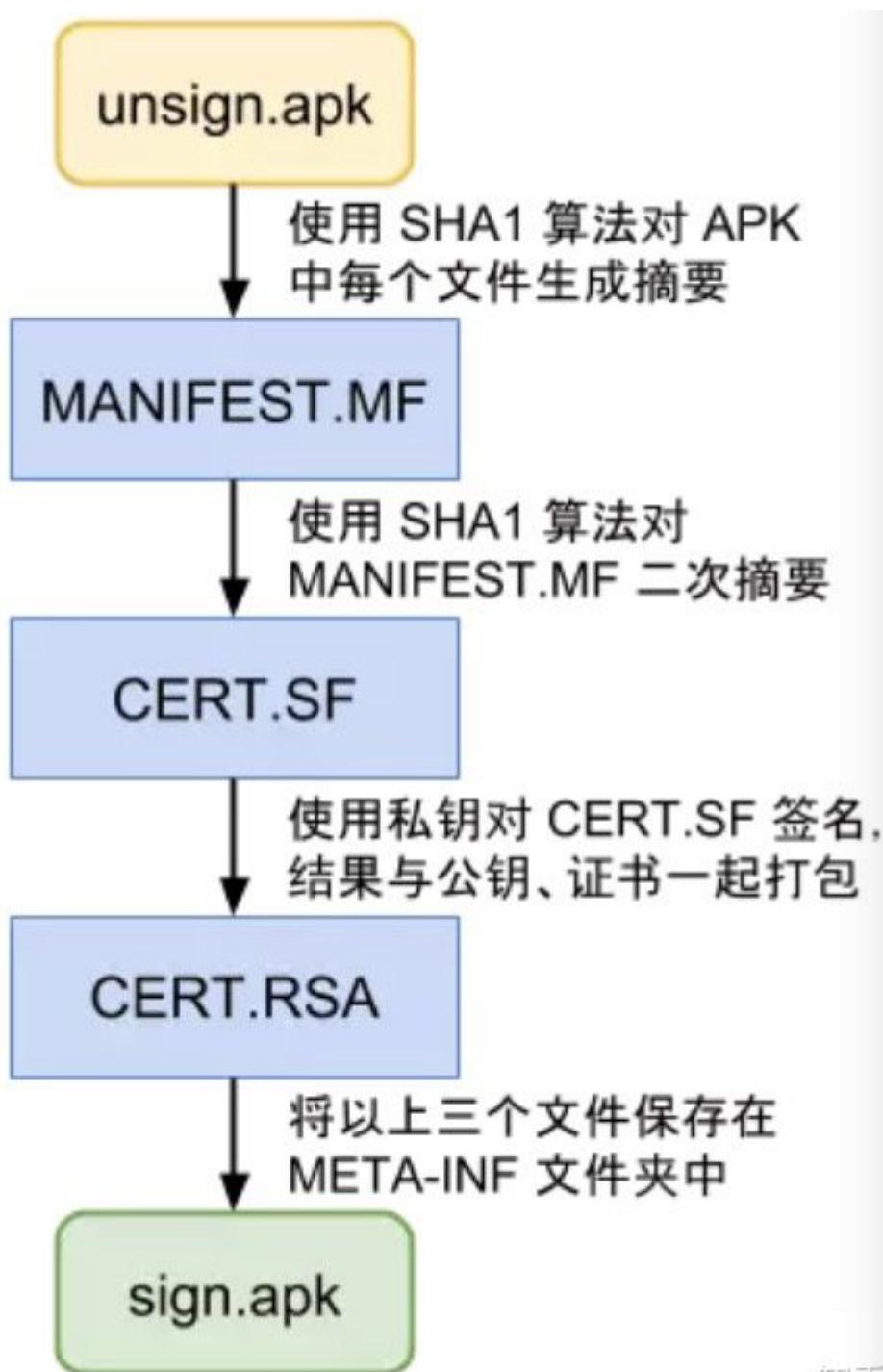
v2签名：将验证归档中的所有字节，而不是单个ZIP，因此，在签署后无法再运行 ZIPalign (必须在签名之前执行)

v3签名：在Android 9.0 引入，在v2的基础上，仍然采用检查整个压缩包的校验方式。不同的是在签名部分增加可以添加新的证书（Attr块）。在这个新块中，会记录我们之前的签名信息以及新的签名信息，以密钥转轮的方案，来做签名的替换和升级。这意味着，只要旧签名证书在手，我们就可以通过它在新的 APK 文件中，更改签名。

v2是7.0引进的，如果只设置v2签名，在7.0以下会直接显示未安装

如果只设置v1签名，在7.0上就不会使用更安全的验证方式

消息摘要只能保证消息的完整性，并不能保证消息的不可篡改



美团多渠道打包原理

在打好的Release包的META-INF目录下添加一个空文件(v2中已经不可用)，利用文件名来标示渠道，然后在代码中去读取这个文件的文件名。签名的时候不会检验这个文件夹的其他内容。

在v2签名方式下，META-INF 已经被列入了保护区了，向 META-INF 添加空文件的方案会对上面受保护的三个区块都有影响。

v2中新增加的区块（APK Signing Block）不受签名校验规则保护，Walle就是通过这个写入渠道信息的。

具体步骤

- 对新的应用签名方案生成的APK包中区块（APK Signing Block）写入渠道信息，并保存在APK中
- APK在安装过程中进行的签名校验，是忽略我们添加的渠道信息的，这样就能正常安装了
- 在App运行阶段，可以通过ZIP的End of central directory、Central directory等结构中的信息找到我们自己添加的渠道信息，从而实现获取渠道信息的功能

速度快，只需要打一次签名包，后续只需要拷贝，新增渠道文件即可，

交叉编译

IntentService的使用

IntentService 是一个基于Service的一个类，用来处理异步的请求。你可以通过startService(Intent)来提交请求，该Service会在需要的时候创建，当完成所有的任务以后自己关闭，且请求是在工作线程处理的。

Room

<https://www.jianshu.com/p/0ed8b17a199e>

<https://juejin.im/post/5a4228036fb9a044ff31b8ca>

<https://juejin.im/post/5d9fdacaf265da5bb86ac12c#heading-2> 外键约束

- 比SQLite API更简单的使用方式
- 省略了许多重复代码
- 能在编译时校验sql语句的正确性
- 数据库相关的代码分为Entity, DAO, Database三个部分，结构清晰
- 简单安全的数据库升级方案

默认情况下，Room使用类名作为数据库表名。如果希望表具有不同的名称，请设置@Entity注解的 tableName 属性

Room使用字段名称作为数据库中的列名。如果希望列具有不同的名称，请将 @ColumnInfo 注解添加到字段中，

```

@Entity(tableName = "users")
public class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}

```

@Query

每个@Query方法在编译时被验证，因此，如果存在查询问题，则会发生编译错误而不是运行时故障。

Room还验证查询的返回值，这样如果返回对象中字段的名称与查询响应中的相应列名不匹配，则Room将以以下两种方式之一提醒您：

- 如果只有一些字段名匹配，则发出警告。
- 如果没有字段名匹配，则会出错。

有时插入数据和更新数据会产生冲突,所以就有了冲突之后要怎么解决,SQLite对于事务冲突定义了5个方案

OnConflictStrategy

- REPLACE:见名知意,替换,违反的记录被删除,以新记录代替之
- ignore: 违反的记录保持原貌,其它记录继续执行
- fail: 终止命令,违反之前执行的操作得到保存
- abort 终止命令,恢复违反之前执行的修改
- rollback 终止命令和事务,回滚整个事务

NO_ACTION: 当person中的uid有变化的时候clothes的father_id不做任何动作

RESTRICT: 当person中的uid在clothes里有依赖的时候禁止对person做动作,做动作就会报错。

SET_NULL: 当person中的uid有变化的时候clothes的father_id会设置为NULL。

SET_DEFAULT: 当person中的uid有变化的时候clothes的father_id会设置为默认值,我这里是int型,那么会设置为0

CASCADE: 当person中的uid有变化的时候clothes的father_id跟着变化,假如我把uid = 1的数据删除,那么clothes表里, father_id = 1的都会被删除。

数据库升级

```

Room.databaseBuilder(getApplicationContext(),
    RoomDemoDatabase.class, "database_name")
    .addCallback(new RoomDatabase.Callback() {
        //第一次创建数据库时调用,但是在创建所有表之后调用的
        @Override

```

```
        public void onCreate(@NonNull SupportSQLiteDatabase db) {
            super.onCreate(db);
        }

        //当数据库被打开时调用
        @Override
        public void onOpen(@NonNull SupportSQLiteDatabase db) {
            super.onOpen(db);
        }
    )
    .allowMainThreadQueries()//允许在主线程查询数据
    .addMigrations()//迁移数据库使用,下面会单独拿出来讲
    .fallbackToDestructiveMigration()//迁移数据库如果发生错误,将会重新创建数
据库,而不是发生崩溃
    .build();
}
```

作者: simplepeng

链接:<https://juejin.im/post/5a4228036fb9a044ff31b8ca>

来源:掘金

常用sql语句

```
//使用 in 可以传入整个数组
//删除不在组里的数据
@Query("DELETE FROM user WHERE uid not in(:ids)")

/**
 * 删除 DeviceGroupModel 表里面所有的数据
 */
@Query("DELETE FROM user")

/**
 * 查询所有数据并通过 id 排序
 * @return
 */
@Query("SELECT * FROM user ORDER BY id")

/**
 * 返回查询到的第一条数据
 * @param
 * @return
 */
@Query("SELECT * FROM user WHERE id = :uid LIMIT 1")

/**
 * 判断name是否存在数据表里
 * @param channelNo

```

```

    * @return
    */
@Query("SELECT EXISTS (SELECT * FROM user WHERE name = :name)")

```

GreenDao	Room
开发者只需要规定Entity的属性即可	需要规定Entity的属性，需要规定Dao接口
每次更新Entity需要重新build以生成代码，大型项目build耗时会比较久	更新Entity不需要重新build，因为使用的是Dao接口规定的方法，但是需要根据情况更新Dao接口
只有进行复杂操作时才需要写SQL语句	即使是进行简单的条件查询，也要写SQL语句
有一定的学习成本，需要学习注解、查询、条件语句等API	学习成本低，基本只需要掌握Room的注解即可

当在 `User.class` 增加完字段，编译后全局搜索 `1.json`。你会发现sql语句已经被生成出来了。

`room` 在插入数据的时候不会更新当前的实体类，如果主键是自增的，并且需要在插入之后获取主键，需要从数据库获取。

```

for (RSDevice device : group.getDeviceList()) {
    HomeItemDeviceIPC item = new HomeItemDeviceIPC(device);
    if (device.getChannelList().isEmpty()) {
        ChannelModel model = new ChannelModel();
        model.setChannelNO(0);
        model.setChannelName("Channel01");
        model.setDeviceKey(device.getModel().getPrimaryKey());
        RSChannel rsChannel = new RSChannel(model);
        rsChannel.setDevice(device);
        device.getChannelList().add(rsChannel);
        Observable.create(emitter -> {
            rsChannel.insertChannelModel(model);
            Long channelPrimaryKey =
rsChannel.getChannelPrimaryKey(device.getModel().getPrimaryKey());
            model.setPrimaryKey(channelPrimaryKey);
            item.setChannel(rsChannel);
        })
        .subscribeOn(Schedulers.io())
        .subscribe();
    }
}

```

Room配合RxJava使用

可以通过 `Maybe`, `Single`, `Flowable` 对象来执行异步查询

`Maybe`

```
@Query("SELECT * FROM Users WHERE id = :userId")
Maybe<User> getUserId(String userId);
```

1. 当数据库中没有user，查询没有返回行时，Maybe调用complete。
2. 当数据库中有一个user时，Maybe将触发onSuccess并调用complete。
3. 如果在Maybe的complete调用之后user被更新，什么也不会发生

Single

1. 当数据库中没有user，查询没有返回行时，Single触发onError (EmptyResultSetException.class)。
2. 当数据库中有一个user时，Single触发onSuccess。
3. 如果在Single.onComplete调用之后user被更新，什么也不会发生。

Flowable

1. 当数据库中没有user，查询没有返回行时，Flowable不会发射，也不会触发onNext, 或者 onError。
2. 当数据库中有一个user时，Flowable会触发onNext。
3. 每当user更新之后，Flowable将自动发射，这样你就可以根据最新的数据来更新UI。

这里的getUserById返回的是Flowable，如果查询没有结果的话，什么信息也收不到，这在实际开发中是很蛋疼的。但是如果我们把返回类型改成Flowable<List>的话，如果查询没有结果是可以得到一个空list的。

System.currentTimeMillis()与SystemClock.uptimeMillis()

1. System.currentTimeMillis()获取的是系统的时间，可以使用SystemClock.setCurrentTimeMillis(long millis)进行设置。如果使用System.currentTimeMillis()来获取当前时间进行计时，应该考虑监听ACTION_TIME_TICK, ACTION_TIME_CHANGED 和 ACTION_TIMEZONE_CHANGED这些广播ACTION，如果系统时间发生了改变，可以通过监听广播来获取。
2. SystemClock.uptimeMillis()表示系统开机到当前的时间总数，单位是毫秒，但是，当系统进入深度睡眠（CPU休眠、屏幕休眠、设备等待外部输入）时间就会停止，但是不会受到时钟缩放、空闲或者其他节能机制的影响。
3. SystemClock.elapsedRealtime()和SystemClock.elapsedRealtimeNanos()表示系统开机到当前的时间总数。它包括了系统深度睡眠的时间。这个时钟是单调的，它保证一直计时，即使CPU处于省电模式，所以它是推荐使用的时间计时器。

Android 调试

Device supports, but APK only supports armeabi rmeabi-v7a

1. 在控制台输入adb kill-server
 2. 重启ADB 在控制台输入adb start-server
 3. 重新刷新Android Studio 问题解决
-

在Android 8.0 之后调用了startForegroundService() 需要在五秒内调用该服务的startForeground()方法，如果在此时间限制内未调用，则会出现ANR

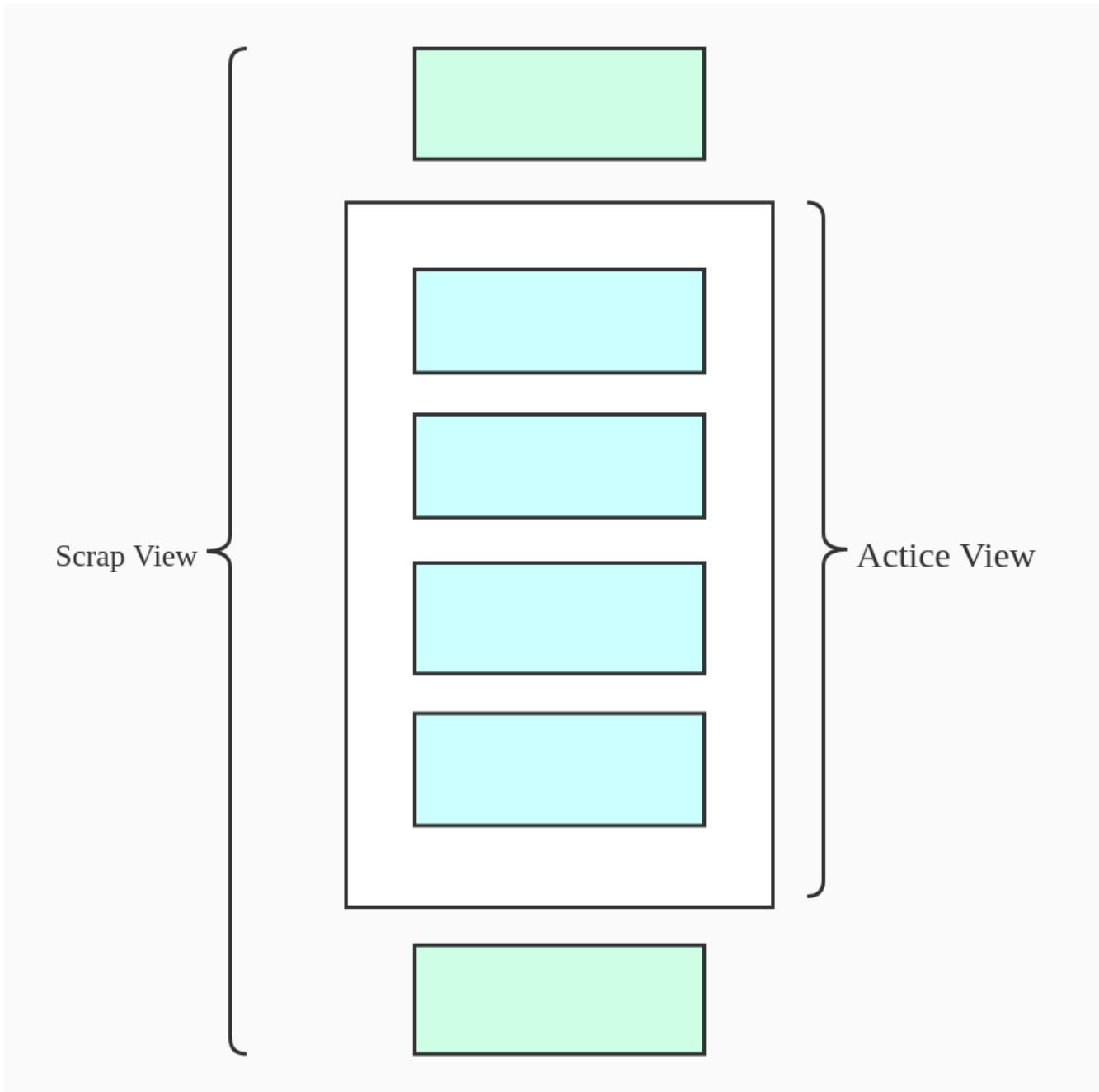
RecyclerView 缓存机制

参考：https://www.jianshu.com/p/3e9aa4bdaef?utm_source=desktop&utm_medium=timeline

ListView有两级缓存，分别是Active View 和 Scrap View 缓存的对象是ItemView

Active View：是缓存在屏幕内的ItemView，当列表数据发生变化时，屏幕内的数据可以直接拿来复用，无须进行数据绑定。

Scrap view：缓存屏幕外的ItemView，这里所有的缓存的数据都是"脏的"，也就是数据需要重新绑定，也就是说屏幕外的所有数据在进入屏幕的时候都要走一遍getView () 方法。再来一张图，看看ListView的缓存流程



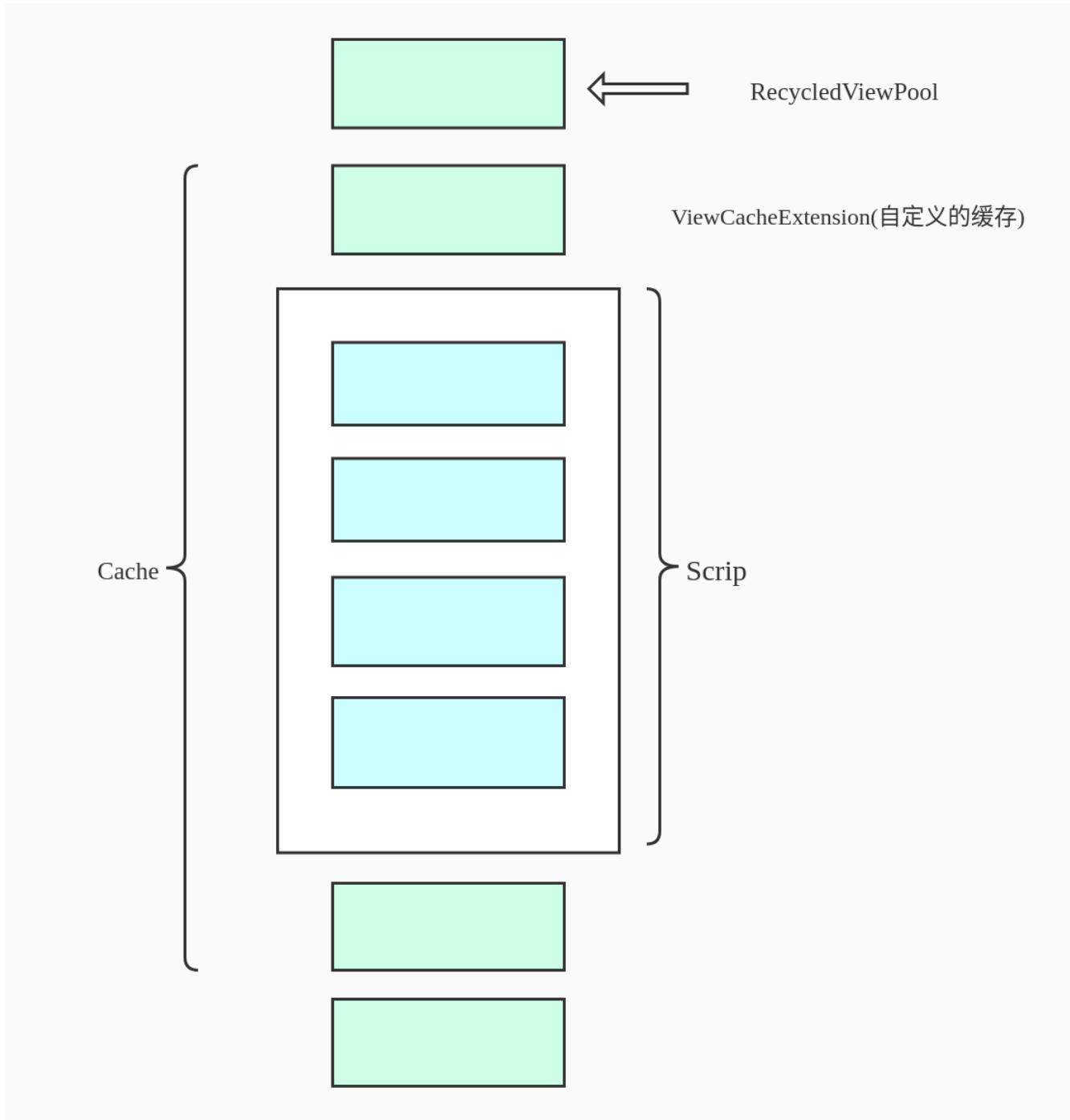
RecyclerView分为四级缓存

- Scrap
- Cache
- ViewCacheExtension
- RecycledViewPool

RecycledViewPool默认的缓存数量是5个

Cache默认的缓存数是2个

当Cache缓存满了之后就会



RecycledViewPool与Cache相比不同的是，从Cache里面移出的ViewHolder再存入RecycledViewPool之前 ViewHolder的数据会被全部重置，相当于一个新的ViewHolder，而且Cache是根据position来获取 ViewHolder，而RecycledViewPool是根据itemType获取的，如果没有重写getItemType () 方法， itemType 就是默认的。因为RecycledViewPool缓存的ViewHolder是全新的，所以取出来的时候需要走 onBindViewHolder () 方法。

RecyclerView中mCacheViews 获取缓存的时候，是通过匹配pos获取目标位置的缓存,这样做的好处是，当数据源数据不变的情况下,无需重新bindView

而同样是离屏缓存,ListView 从 mScrapViews根据pos获取相应的缓存，但是并没有直接使用，而是重新getView (即必定会bindView)

RecyclerView中的 mAttachedScrap 和 mChangedScrap 容器, 为什么需要两个?

mAttachedScrap 在整个布局过程中都能使用, 但是 changed scrap — 只能在预布局阶段使用。

ViewHolder 只有在满足下面情况才会被添加到 mChangedScrap: 当它关联的 item 发生了变化 (notifyItemChanged 或者 notifyItemRangeChanged 被调用), 并且 ItemAnimator 调用 ViewHolder#canReuseUpdatedViewHolder 方法时, 返回了 false。否则, ViewHolder 会被添加到 AttachedScrap 中。

canReuseUpdatedViewHolder 返回 “false” 表示我们要执行用一个 view 替换另一个 view 的动画, 例如淡入淡出动画。“true”表示动画在 view 内部发生。

优化

<https://juejin.im/post/6844903685655363598>

总结

列表展示页面, 需要支持动画, 或频繁更新, 局部刷新, 建议使用 RecyclerView, 更加强大完善, 易扩展;

其他情况(如微信卡包列表)两者都ok, 但 ListView 使用上会更加方便快捷.

CPU架构

1. armeabiv7a: 第7代及以上的 ARM 处理器。2011年12月以后的生产的大部分Android设备都使用它。
2. arm64-v8a: 第8代、64位ARM处理器, 很少设备, 三星 Galaxy S6是其中之一。
3. armeabi: 第5代、第6代的ARM处理器, 早期的手机用的比较多。
4. x86: 平板、模拟器用得比较多。
5. x86_64: 64位的平板。

arm64-v8a是可以向下兼容的, 但前提是你的项目里面没有arm64-v8a的文件夹, 如果你有两个文件夹armeabi和arm64-v8a, 两个文件夹, armeabi里面有a.so 和 b.so, arm64-v8a里面只有a.so, 那么arm64-v8a的手机在用到b的时候发现有arm64-v8a的文件夹, 发现里面没有b.so, 就报错了, 所以这个时候删掉arm64-v8a文件夹, 这个时候手机发现没有适配arm64-v8a, 就会直接去找armeabi的so库, 所以要么你别加arm64-v8a, 要么armeabi里面有的so库, arm64-v8a里面也必须有

GLSL语法

.vert - 顶点着色器 .tesc - 曲面细分控制着色器 .tese - 曲面细分评估着色器 .geom - 几何着色器 .frag - 片元着色器 .comp - 计算着色器

向量: 存储及操作颜色、位置、纹理坐标等

vec2 二维向量型-浮点型

vec3 三维向量型-浮点型

vec4 四维向量型-浮点型

ivec2 二维向量型-整型

ivec3 三维向量型-整型

ivec4 四维向量型-整型

bvec2 二维向量型-布尔型

bvec3 三维向量型-布尔型

bvec4 四维向量型-布尔型

矩阵：根据矩阵的运算进行变换操作

mat2 2X2矩阵-浮点型 **mat3** 3X3矩阵-浮点型 **mat4** 4X4矩阵-浮点型

采样器

sampler2D 二维纹理 **sampler3D** 三维纹理 **samplerCube** 立方贴图纹理

限定符

attribute：顶点的变量,如顶点位置,颜色 (变量是这个需要用**GLES20.glGetAttribLocation** 获取)

uniform：一般用于对于3D物体中所有顶点都相同的量 (要用 **GLES20.glGetUniformLocation** 获取)

varying：用于从顶点着色器传递到片元着色器的变量

const：常量 **precision** 精度 |**--lowp** |**--mediump** |**--highp**

变量

gl_Position: 顶点坐标

gl_PointSize: 点的大小, 没有赋值则默认值为1

gl_FragColor: 当前片元的颜色

```
precision mediump float; //片元的精度
uniform vec4 vColor; // 声明片元的颜色
attribute vec4 vPosition;//定义一个四维向量
void main() {
    //gl_FragColor 是 gl的内定名, 将vColor的值赋给它
    gl_FragColor = vColor;
    gl_Position = vPosition;
}
```

世界坐标：是用于显示的坐标，即像素点应该显示在哪个位置由世界坐标决定。

纹理坐标：表示世界坐标指定的位置点想要显示的颜色，应该在纹理上的哪个位置获取。即颜色所在的位置由纹理坐标决定。

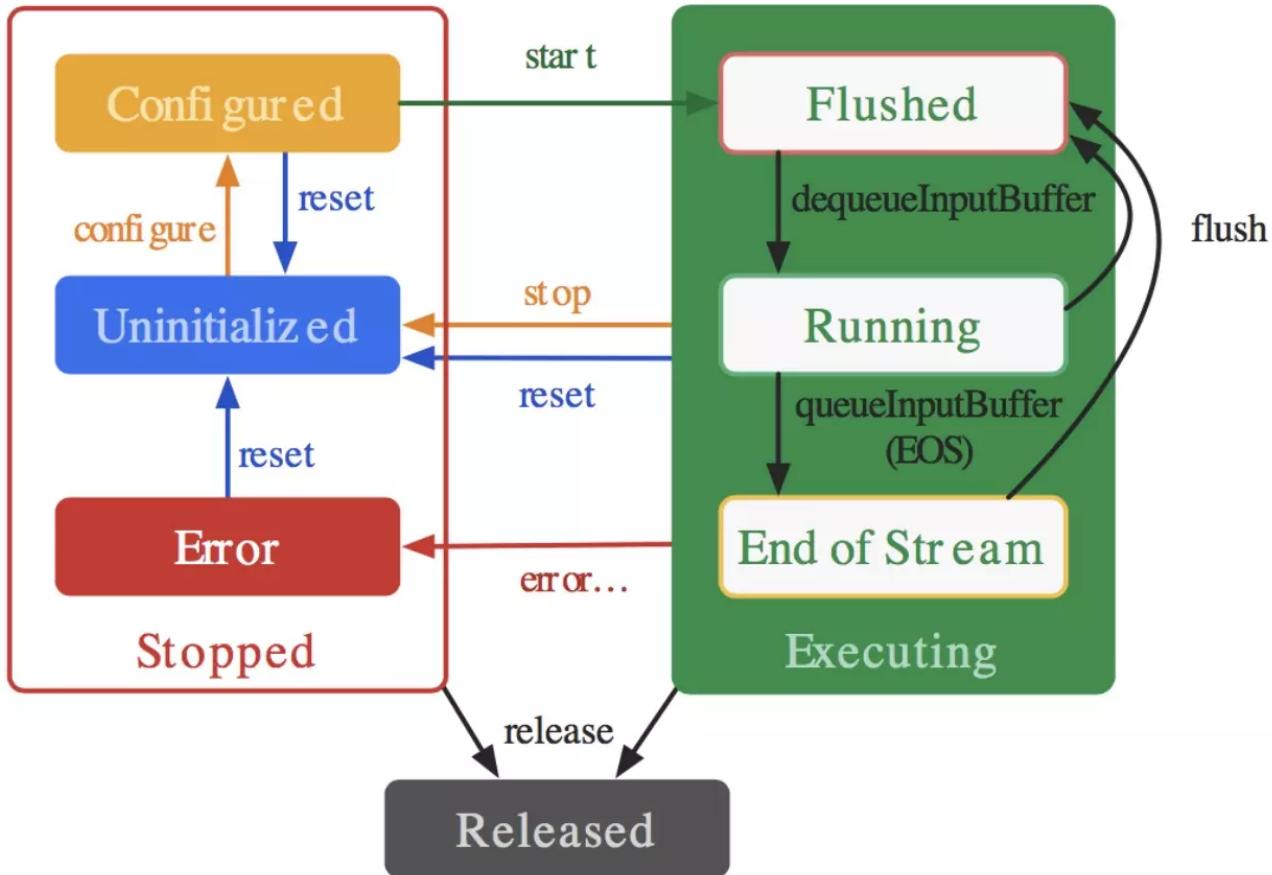
顶点着色器对应世界坐标，片元着色器对应纹理坐标。

Amazon 投屏注册

登录Amazon - > <https://developer.amazon.com/loginwithamazon/console/site/lwa/overview.html>

The screenshot shows the 'Login with Amazon' interface. At the top, there are links for 'Home', 'Documentation', and 'Login with Amazon Console'. On the right, there are icons for 'R', '?', and a magnifying glass. Below the header, the title '安全配置文件管理' is displayed. A note says '命名您的新安全配置文件' (Name your new security configuration file). It explains that you can create multiple security configuration files and share data between them. A link to '了解更多信息' (Learn more) is provided. There are three input fields: '安全配置文件名称*' (Security Configuration File Name*) containing '应用名' (App Name), '安全配置文件描述*' (Security Configuration File Description*) containing '随意' (Any), and '同意隐私声明URL*' (Agree Privacy Statement URL*) containing 'https:// or http:// 应用的隐私政策' (The app's privacy policy). Below these fields is a placeholder for '同意徽标图像' (Agree Logo Image) with a button to '上传图片' (Upload Picture). At the bottom right are '保存' (Save) and '取消' (Cancel) buttons.

获取密钥--进入本地 keystore 目录下 执行命令



openGL 回调一个 SurfaceTexture ----> MediaCodec 启动解码器

1. openGL生成纹理ID
2. 将纹理ID绑定到SurfaceTexture上
3. 用SurfaceTexture做参数创建Surface
4. 创建一个回调函数，把Surface回调
5. 将回调的参数和视频的path一起发送给MediaCodec进行解码，然后就可以显示画面了

```

Matrix.orthoM(float[], mOffset, left, right, bottom, top, near, far)
float[]:16位数组, 用来存储
mOffset:
left: x轴的最小值
right: x轴的最大值
bottom: y轴的最小值
top: y轴的最大值
near: z轴最小值
far: z轴最大值

```

Android 自定义注解

元注解

@Target	表明我们注解可以出现的地方，是一个ElementType枚举
@Retention	这个注解的存活时间
@Document	表明注解可以被javadoc此类的工具文档化
@Inherited	是否允许子类继承该注解，默认为false

<https://blog.csdn.net/wuyuxing24/article/details/81139846>

@Target ElementType 描述的范围

CONSTRUCTOR	用于描述构造器
FIELD	用于描述域
LOCAL_VARIABLE	用于描述局部变量
METHOD	用于描述方法
PACKAGE	用于描述包
PARAMETER	用于描述参数
TYPE	用于描述类、接口（包括注解类型）或enum声明

@Retention

表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）

SOURCE	在源文件有效
CLASS	在class文件有效
RUNTIME	在运行时有效

运算符

>> 右移运算符，num >> 1 相当于 num 除以 2。5 >> 2 等于 1

<< 左移运算符， num << 1 相当于 num 乘以 2。5 << 2 等于 20

使用Battery Historian 检测耗电

1. 更新apt包的索引

```
sudo apt update
```

2. 安装软件包以允许apt通过HTTPS使用存储库

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

3. 添加Docker官方的密钥

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

4. 将Docker存储库添加到APT源

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
```

5. 更新apt包索引

```
sudo apt-get update
```

6. 确认从 Docker repo 安装

```
apt-cache policy docker-ce
```

如果还没有安装，接下来就安装Docker

```
sudo apt install docker-ce
```

这时候如果显示这样的错误

下列软件包有未满足的依赖关系：

```
docker-ce : 依赖: containerd.io (>= 1.2.2-3) 但是它将不会被安装  
           依赖: libseccomp2 (>= 2.3.0) 但是 2.1.1-1ubuntu1~trusty5 正要被安装  
           依赖: libdevmapper1.02.1 (>= 2:1.02.97) 但是 2:1.02.77-6ubuntu2 正要被安装  
           依赖: libsystemd0 但无法安装它
```

E: 无法修正错误，因为您要求某些软件包保持现状，就是它们破坏了软件包间的依赖关系

请输入

```
sudo apt-cache madison docker-ce
```

查看可以安装的依赖包，比如

```
docker-ce | 17.03.2~ce-0~ubuntu-trusty | http://mirrors.aliyun.com/docker-ce/linux/ubuntu/trusty/stable amd64 Packages  
docker-ce | 17.03.2~ce-0~ubuntu-trusty | https://download.docker.com/linux/ubuntu/trusty/stable amd64 Packages  
docker-ce | 17.03.1~ce-0~ubuntu-trusty | http://mirrors.aliyun.com/docker-ce/linux/ubuntu/trusty/stable amd64 Packages  
docker-ce | 17.03.1~ce-0~ubuntu-trusty | https://download.docker.com/linux/ubuntu/trusty/stable amd64 Packages  
docker-ce | 17.03.0~ce-0~ubuntu-trusty | http://mirrors.aliyun.com/docker-ce/linux/ubuntu/trusty/stable amd64 Packages
```

这时候选择一个安装即可，如果一个不行，就换另一个

```
sudo apt-get install docker-ce="版本"  
比如:  
sudo apt-get install docker-ce=17.03.1~ce-0~ubuntu-trusty
```

安装完成后输入

```
sudo docker run hello-world
```

如果显示下面这样说明安装成功了

```
rs@rsyf:~$ sudo docker run hello-world  
Hello from Docker!  
This message shows that your installation appears to be working correctly.
```

接下来输入

```
sudo docker run -p 9999:9999 gcr.io/android-battery-historian/stable:3.0 --port 9999
```

```
s@rsyf:~$ sudo docker run -p 9998:9998 gcr.io/android-battery-historian/stable:3.0 --port 9998
Unable to find image 'gcr.io/android-battery-historian/stable:3.0' locally
3.0: Pulling from android-battery-historian/stable
72777927d38a: Pull complete
d3eac894db4: Pull complete
df72af6d627: Pull complete
e4f86211d23: Pull complete
c5d6e8f41a3: Pull complete
9e1ccca7c291: Pull complete
c6b579f97fc: Pull complete
73d3370a7dc: Pull complete
2c70f621a64: Pull complete
687e759f67f: Pull complete
f056f369588: Pull complete
2683cf72ae5: Pull complete
Digest: sha256:7711d28f2d95b6858688acd52350fa5fd420c4a5114e670b559d77cfe37f90f2
Status: Downloaded newer image for gcr.io/android-battery-historian/stable:3.0
2020/03/31 03:49:48 Listening on port: 9998
2020/03/31 04:02:22 Trace starting analysisServer processing for: GET
2020/03/31 04:02:22 Trace finished analysisServer processing for: GET
2020/03/31 04:04:11 Trace starting analysisServer processing for: GET
```

会出现这样的页面就说明成功了，如果没有出现，可能网络有问题，多试几次就好了

配置完成后在浏览器输入

```
localhost:9998
```

9998是你刚刚配置的端口，你配置的是什么就填什么就好了，如果没有意外的话这时候网页就会显示下图这样就可以了。

The screenshot shows a web browser displaying the 'Battery Historian' application. The title bar says 'Battery Historian'. Below it, the main heading is 'Upload Bugreport'. A sub-instruction states 'Both .txt and .zip bug reports are accepted.' There are three input fields: 'Choose a Bugreport File' (with a 'Browse' button), 'Kernel Wakesource Trace' (with a '+' button), and 'Power Monitor File' (with a '+' button). A link 'Switch to Bugreport Comparison' is also visible.

接下来就是获取手机耗电信息了。

插入数据线，先初始化一下数据采集，重置好后就可以拔出数据线进行测试了，为了测试准确性，测试途中请不要充电。

```
:~$ adb kill-server
:~$ adb start-server
n not running; starting now at tcp:5037
n started successfully
:~$ adb shell dumpsys batterystats --enable full-wake-history
: full-wake-history
:~$ adb shell dumpsys batterystats --reset
```

adb shell dumpsys batterystats --enable full-wake-history

adb shell dumpsys batterystats --reset

adb bugreport bugreport.zip

adb bugreport > bugreport.txt

https://blog.csdn.net/qq_40423339/article/details/87885086

<https://cloud.tencent.com/developer/article/1167995>

The screenshot shows the 'Battery History' application interface. On the left, there's a sidebar with 'App Selection' and a dropdown for 'Sort apps by Name'. Below it is a list of tables: 'System Stats', 'Aggregated Checkin Stats', 'Device's Power Estimates', and 'Userspace Wakelocks'. The main area has tabs for 'System Stats', 'History Stats' (selected), and 'App Stats'. Under 'History Stats', there's a section titled 'Wakeup alarm info:' with a table showing three entries: '*walarm*:com.redlinecloud.client/com.baidu.android', '*walarm*:job.delay*', and '*walarm*:job.deadline*'. A note at the bottom says 'Showing 1 to 3 of 3 entries'.

System Stats History Stats App Stats			
Showing 2 of 3 entries			
- Wakeup alarm info:			
Show 5 entries			
Wakeup Alarm Name			
walarm.com.redlinecloud.client/com.baidu.android.pushservice.PushService	3	Search: <input type="text"/>	Count <input type="button" value="Copy"/>
walarm.*job.delay*	3		
walarm.*job.deadline*	1		
Showing 1 to 3 of 3 entries			
Previous <input type="button" value="1"/> Next			
- Services:			
Show 10 entries			
Service Name			
com.baidu.android.pushservice.PushService	5m 6s 545ms	1	1
com.gyf.cactus.service.CactusJobService	4m 27s 282ms	1	1
com.gyf.cactus.service.LocalService	4m 27s 205ms	1	1
com.gyf.cactus.service.RemoteService	4m 27s 145ms	1	1
com.squareup.leakcanary.internal.HeapAnalyzerService	12s 310ms	1	1
com.gyf.cactus.service.HideForegroundService	2s 61ms	1	1
com.raysharp.camviewplus.notification.service.PushRegisterIntentService	184ms	1	1
com.baidu.android.pushservice.CommandService	175ms	4	4
com.squareup.leakcanary.DisplayLeakService	146ms	1	1
androidx.work.impl.background.systemjob.SystemJobService	0	3	
Showing 1 to 10 of 10 entries			
Previous <input type="button" value="1"/> Next			

Upload Bugreport

Both .txt and .zip bug reports are accepted.

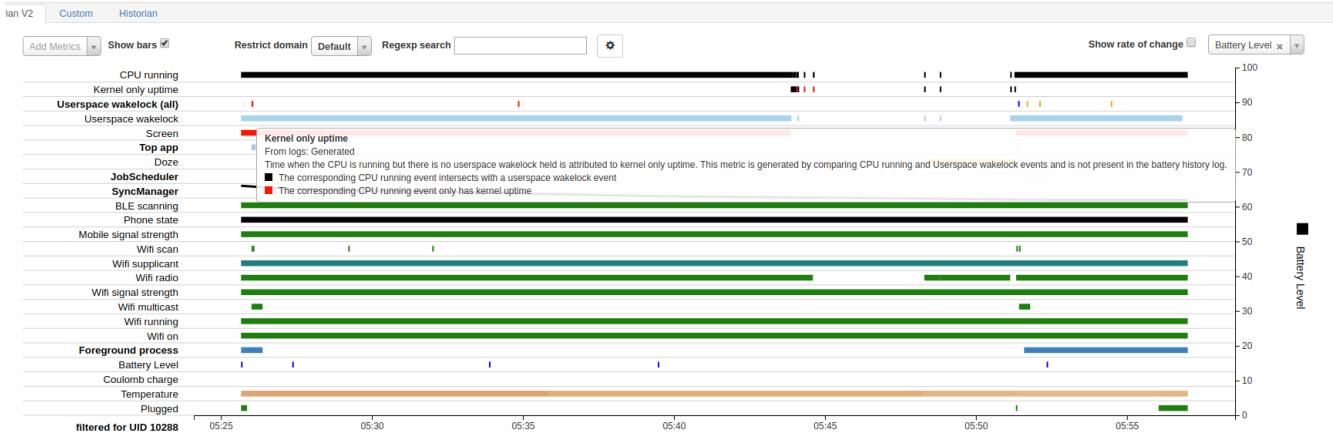
<input type="button" value="Browse"/>	bugreport.txt
<input type="button" value="Kernel Wakesource Trace"/>	<input type="button" value="Power Monitor File"/>
<input type="button" value="Switch to Bugreport Comparison"/>	
<input type="button" value="Submit"/>	

Upload Bugreport

Both .txt and .zip bug reports are accepted.

<input type="button" value="Browse"/>	bugreport.txt
<input type="button" value="Browse"/>	Choose a Second Bugreport File
<input type="button" value="X"/>	

System Stats		History Stats	App Stats	
Application	com.redlinecloud.client			<input type="button" value="Copy"/>
Version Name	2.0.4			
Version Code	339			
UID	10288			
Device estimated power use	0.21%			
Foreground	5 times over 10s 881ms			
CPU user time	2m 0s 450ms			
CPU system time	3m 29s 640ms			
Device estimated power use due to CPU usage	0.00%			
Total number of wakeup alarms	3			



HashMap 扩容条件 $\text{HashMap} \text{ 中的数据量} > \text{容量} * \text{加载因子}$ **HashMap默认的加载因子是0.75**

HashMap获取数据是通过遍历Entry[]数组来得到对应的元素，在数据量很大时候会比较慢

SparseArray 它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间

SparseArray 只能存储key为int类型的数据，同时SparseArray在存储和读取数据时候，使用的是二分法

所以，在获取数据的时候非常快，比HashMap快的多，因为HashMap获取数据是通过遍历Entry[]数组来得到对应的元素。

setOnClickListener

多线程

Java 内存模型规定了所有的变量都存储在主内存中，每条线程有自己的工作内存。

线程的工作内存中保存了该线程中用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。

线程访问一个变量，首先将变量从主内存拷贝到工作内存，对变量的写操作，不会马上同步到主内存。

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

并发三要素

原子性: 在一个操作中，CPU 不可以在中途暂停然后再调度，即不被中断操作，要么执行完成，要么就不执行。

可见性: 多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性: 程序执行的顺序按照代码的先后顺序执行。

总结

- 当只有一个线程写，其它线程都是读的时候，可以用 `volatile` 修饰变量
- 当多个线程写，那么一般情况下并发不严重的话可以用 `Synchronized`，`Synchronized`并不是一开始就是重量级锁，在并发不严重的时候，比如只有一个线程访问的时候，是偏向锁；当多个线程访问，但不是同时访问，这时候锁升级为轻量级锁；当多个线程同时访问，这时候升级为重量级锁。所以在并发不是很严重的情况下，使用 `Synchronized` 是可以的。不过 `Synchronized` 有局限性，比如不能设置锁超时，不能通过代码释放锁。
- `ReentrantLock` 可以通过代码释放锁，可以设置锁超时。
- 高并发下，`Synchronized`、`ReentrantLock` 效率低，因为同一时刻只有一个线程能进入同步代码块，如果同时有很多线程访问，那么其它线程就都在等待锁。这个时候可以使用并发包下的数据结构，例如 `ConcurrentHashMap`，`LinkBlockingQueue`，以及原子性的数据结构如：`AtomicInteger`。

链接：<https://juejin.im/post/6844903941830869006>

线程等待

需要等待的其他线程执行完毕再执行的场景

`sleep` 属于限时等待状态，限时等待一段时间，时间到了就会自动转换为可运行状态。

`wait` 属于等待状态，需要使用 `notify()`、`notifyAll()` 等方法进行唤醒。

线程池

BlockingQueue（缓存队列）：

当有任务到来时，会指派给核心线程去执行，等核心线程都被占用了，那么再有新的任务，就会加入到队列中，等队列满了，再有任务，就再启动非核心线程去执行。

- `SynchronousQueue` 使用这个队列时，当有任务到来的时候，它并不存任务，而是直接将任务丢给线程去执行，如果线程都在被占用，它就会创建线程去处理这个任务，所以一般使用这个缓存队列的时候，`maximumPoolSize`（线程池能容纳的最大线程数量）设置到 `Integer.MAX_VALUE`，不然任务数超过 `maximumPoolSize` 限制而创建不了线程。
- `LinkedBlockingQueue` 使用这个队列时，当有任务到来的时候，如果当前的核心线程数 < `corePoolSize`，它会新建核心线程去执行任务，如果当前核心线程数 \geq `corePoolSize` 时，它会将还未被执行的任务存储起来，等待执行，但是这个队列，没有存储上限，所以尼，这也就造成了，`maximumPoolSize`（总线程数），永远不会超过 `corePoolSize`。此队列按 `FIFO`（先进先出）原则对任务进行操作。

- **ArrayBlockingQueue** 使用这个队列，可以设置队列的长度，那么当任务到来的时候，核心线程数 < `corePoolSize` 时，则创建核心线程去执行任务，如果核心线程数 \geq `corePoolSize` 时，加入到队列里面，等待执行，如果队列也满了，则新建非核心线程去执行任务。此队列按 FIFO（先进先出）原则对元素进行操作。但是线程数不能超过总线程数。
- **DelayQueue**（延时队列） 任务到来时，首先先加入到队列中，只有达到了指定的延时时间，才会执行任务。

锁

链接：<https://juejin.im/post/5d7da37d6fb9a06b0202f156>

volatile

Volatile 保证可见性 不保证原子性

可见性就是说一旦某个线程修改了**volatile**关键字修饰的变量，则该变量将会立即保存修改后的值到物理内存中，其他线程读取该变量时，也可以立即获取修改后的值。

对于一个变量，只有一个线程执行写操作，其它线程都是读操作，这时候可以用 **volatile** 修饰这个变量。

volatile和synchronized的区别

1. **volatile**本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；
synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. **volatile**仅能使用在变量级别；**synchronized**则可以使用在变量、方法、和类级别的
3. **volatile**仅能实现变量的修改可见性，不能保证原子性；而**synchronized**则可以保证变量的修改可见性和原子性
4. **volatile**不会造成线程的阻塞；**synchronized**可能会造成线程的阻塞。
5. **volatile**标记的变量不会被编译器优化；**synchronized**标记的变量可以被编译器优化

Synchronized

在1.6以前是重量级锁，在1.6以后引入了偏向锁,和轻量级锁

偏向锁： 大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。

当一个线程A访问加了同步锁的代码块时，会在对象头中存储当前线程的id，后续这个线程进入和退出这段加了同步锁的代码块时，不需要再次加锁和释放锁。

轻量级锁： 在偏向锁情况下，如果线程B也访问了同步代码块，比较对象头的线程id不一样，会升级为轻量级锁，并且通过自旋的方式来获取轻量级锁。

重量级锁： 如果线程A和线程B同时访问同步代码块，则轻量级锁会升级为重量级锁，线程A获取到重量级锁的情况下，线程B只能入队等待，进入BLOCK状态。

缺点：

1. 不能设置锁超时时间
2. 不能通过代码释放锁
3. 容易造成死锁

当只有一个线程写，其它线程都是读的时候，可以用 `volatile` 修饰变量

当多个线程写，那么一般情况下并发不严重的话可以用 `Synchronized`，`Synchronized`并不是一开始就是重量级锁，在并发不严重的时候，比如只有一个线程访问的时候，是偏向锁；当多个线程访问，但不是同时访问，这时候锁升级为轻量级锁；当多个线程同时访问，这时候升级为重量级锁。所以在并发不是很严重的情况下，使用 `Synchronized` 是可以的。不过 `Synchronized` 有局限性，比如不能设置锁超时，不能通过代码释放锁。

`ReentrantLock` 可以通过代码释放锁，可以设置锁超时。

高并发下，`Synchronized`、`ReentrantLock` 效率低，因为同一时刻只有一个线程能进入同步代码块，如果同时有很多线程访问，那么其它线程就都在等待锁。这个时候可以使用并发包下的数据结构，例如 `ConcurrentHashMap`，`LinkBlockingQueue`，以及原子性的数据结构如：`AtomicInteger`。

`synchronized` 修饰实例方法和修饰静态方法有啥不一样

修饰静态方法相当于修饰这个类，就算 `new` 几个不同的实例，但还是属于这个类，所以还是互斥状态。

修饰实例方法，如果 `new` 出两个对象，相当与两个方法，之间不互斥

参考：<https://www.jianshu.com/p/27f5935cafd8>

ReentrantLock

在多个条件变量和高度竞争锁的地方，用 `ReentrantLock` 更合适，`ReentrantLock` 还提供了 `Condition`，对线程的等待和唤醒等操作更加灵活，一个 `ReentrantLock` 可以有多个 `Condition` 实例，所以更有扩展性。

`ReentrantLock` 构造函数传 `true` 表示公平锁。

1. `ReentrantLock` 使用 `lock` 和 `unlock` 来获得锁和释放锁
2. `unlock` 要放在 `finally` 中，这样正常运行或者异常都会释放锁
3. 使用 `condition` 的 `await` 和 `signal` 方法之前，必须调用 `lock` 方法获得对象监视器

原子操作类 `AtomicInteger`

内部采用 CAS (compare and swap) 保证原子性

公平锁和非公平锁

公平锁

指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获取锁。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大。

非公平锁

多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁，所以非公平锁有可能出现后申请锁的线程先获取锁的场景。非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

可重入锁 和 不可重入锁

可重入锁

又名递归锁，指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者同一个class），不会因为之前已经获取过还没释放而阻塞。Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。

```
public class Widget {  
    public synchronized void doSomething() {  
        System.out.println("方法1执行...");  
        doOthers();  
    }  
  
    public synchronized void doOthers() {  
        System.out.println("方法2执行...");  
    }  
}  
//doSomething()方法中调用doOthers()方法。因为内置锁是可重入的，所以同一个线程在调用doOthers()时可以直接获得当前对象的锁，进入doOthers()进行操作
```

独享锁 和 共享锁

独享锁

也叫排他锁，是指该锁一次只能被一个线程所持有。如果线程T对数据A加上排它锁后，则其他线程不能再对A加任何类型的锁。获得排它锁的线程即能读数据又能修改数据。

共享锁

指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。

乐观锁 和 悲观锁

锁从宏观上分类，可以分成乐观锁和悲观锁

<https://juejin.im/post/6844903639207641096>

悲观锁

总是假设最坏的情况，每次取数据的时候都认为数据会被别人修改，所以每次取数据都会上锁，这样别人想拿数据的话就会阻塞。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

适用于多写的场景

乐观锁

总是假设最好的情况，每次取数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。在Java中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

适用于读多写少的情况，这样可以省去锁的开销，加大系统的吞吐量

版本号机制

一般在数据表会加上一个 版本号 `version` 字段，读取数据的时候会读取 `version` 值，当数据被修改进行更新的时候 `version` 值会加 1，在提交更新的时候提交的 `version` 值要大于 数据库读取到的 `version` 值才会更新，否则重试。也就是 提交的版本必须大于数据库记录的版本

CAS算法

什么是CAS

- CAS(compare and swap) 比较并替换，比较和替换是线程并发算法时用到的一种技术
- CAS是原子操作，保证并发安全，而不是保证并发同步
- CAS是CPU的一个指令
- CAS是非阻塞的、轻量级的乐观锁

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

比如A,B两个线程，主内存中的值为1，A,B 线程要对这个值 进行 +1 处理

此时A,B 中的工作内存 值为1，那么 读取的内存值=1，期望值 = 1，需要 期望值 == 读取的内存值 才会进行修改。

当且仅当 V 的值等于 A 时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

缺点：

- ABA问题 线程C、D,线程D将A修改为B后又修改为A,此时C线程以为A没有改变过，java的原子类 `AtomicStampedReference`，通过控制变量值的版本来保证CAS的正确性。
- 自旋时间过长，消耗CPU资源，如果资源竞争激烈，多线程自旋长时间消耗资源

SparseArray应用场景：

okhttp原理

缓存：

Http 的缓存分为两种：强制缓存和对比缓存。强制缓存优先于对比缓存。

1. 强制缓存：客户端第一次请求数据时，服务端返回缓存的过期时间（通过字段 `Expires` 与 `Cache-Control` 标识），后续如果缓存没有过期就直接使用缓存，无需请求服务端；否则向服务端请求数据。
 - `Expires`：服务端返回的到期时间。下一次请求时，请求时间小于 `Expires` 的值，直接使用缓存数据。
由于到期时间是服务端生成，客户端和服务端的时间可能存在误差，导致缓存命中的误差。
 - `Cache-Control`：Http1.1 中采用了 `Cache-Control` 代替了 `Expires`，常见 `Cache-Control` 的取值有：
 - `private`: 客户端可以缓存
 - `public`: 客户端和代理服务器都可缓存
 - `max-age=xxx`: 缓存的内容将在 `xxx` 秒后失效
 - `no-cache`: 需要使用对比缓存来验证缓存数据，并不是字面意思
 - `no-store`: 所有内容都不会缓存，强制缓存，对比缓存都不会触发

2. 对比缓存：

RxJava解析

gradle多渠道打包

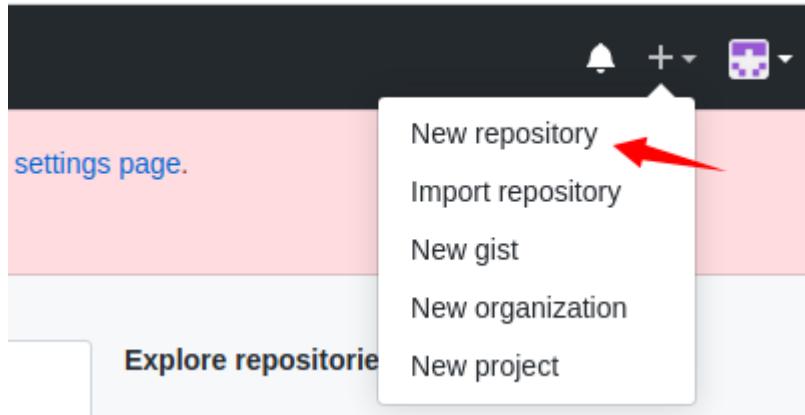
###

webview怎么封装

gitHub + PicGo 图床配置

1. 创建仓库

首先进入gitHub 创建一个仓库



点击new repository创建一个新的仓库

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner **Repository name ***

Lnima-hub

Great repository names are short and memorable. Need inspiration? How about [supreme-umbrella](#)?

Description (optional)

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** | Add a license: **None** |

Create repository

填写好点击创建就可以了。

2. 获取Token

在github 右上角找到settings -> Developer Settings -> Personal access tokens

GitHub Apps
OAuth Apps
Personal access tokens

Personal access tokens

Tokens you have generated that can be used to access the GitHub API.

Lnima PicGo — repo Last used within the last week Delete

Generate new token Revoke all

点击生成 Token

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Lnima PicGo

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> write:packages	Upload packages to github package registry

名称随意填一个，勾上repo，然后点击底下Generate token 就可以拿到一个一串数字的Token了，复制保存好，等下需要用到。

3.安装PicGo

下载地址：<https://github.com/Molunerfinn/PicGo>

下载对应系统的版本，然后安装就可以了。



打开点击图床设置，这里我们用的是GitHub进行配置，所以选择GitHub图床

- 第一个是你仓库名
- 第二个是分支直接写master
- 第三个就是我们刚刚生成的Token
- 第四项就是在仓库下创建一个文件夹，名称随意，非必填
- 第五个就是<https://raw.githubusercontent.com/>用户名/仓库名/分支，填了之后你的图片路径就是这个路径+图片的名称了。

配置完这个就可以了，

linux 系统下快捷键上传需要 安装 xclip

```
sudo apt -y install xclip
```

ObservaleField 防抖，第一次 set true，就有 true 为 value 了，第二次再 set true，就不 notify 视图刷新了

Window、 WindowManager

window: 一个抽象类，它的具体实现类是PhoneWindow

windowManager: 是外界访问Window的入口

普通的Dialog 必须采用Activity的 Context，如果采用Application的Context 就会报错。这是因为没有应用 token所导致的，而应用的token一般只有Activity 拥有

异常处理

Caused by: com.android.tools.r8.CompilationFailedException: Compilation fail: 异常处理

Clean Project -> Rebuild Project

ERROR: *** Android NDK: APP_STL gnuStl_static is no longer supported. Please switch to either c++static or c++shared. See <https://developer.android.com/ndk/guides/cpp-support.html> for more information.. Stop.

1. 一般是NDK版本过高引起的，下载低版本的NDK，把路径切换到低版本的NDK上

或者：

2. 对于ndk-build，删除设置NDK_TOOLCHAIN或NDK_TOOLCHAIN_VERSION的行。
3. 对于cmake，请删除设置ANDROID_TOOLCHAIN的行。

undefined reference to 'eglGetDisplay'

添加下面这个

```
target_link_libraries( # Specifies the target library.
    native-lib
        # 连接FFmpeg相关的库
        avutil
        swresample
        avcodec
        avfilter
        swscale
        avformat
        avdevice

        -landroid
        # 打开OpenSL ES 支持
        OpenSLES
        -lEGL
        -lGLESv2
        # Links the target library to the log library
        # included in the NDK.
        ${log-lib} )
```

java.lang.OutOfMemoryError: pthread_create (1040KB stack) failed: Out of memory

一般出现在华为手机上，由于华为手机系统给程序分配的线程数比较少，所以华为手机容易出现这个bug

猫眼团队问题解决

解决： 对线程进行管理， 使用线程池， 减少线程创建的数量。

在dex2jar目录下执行sudo sh d2j-dex2jar.sh classes.dex时报错如下 d2j-dex2jar.sh: 36: d2j-dex2jar.sh: ./d2j_invoke.sh: Permission denied

解决方案

sudo chmod +x d2j_invoke.sh

You need to use a Theme.AppCompat theme (or descendant) with this activity错误

解决方法有两种：

1. 把当前Activity继承的AppCompatActivity 换成 Activity
2. 在AndroidManifest.xml 文件下，把当前Activity的 android: theme 设置的主题换成 Theme.AppCompat类的主题。

Cmake

<https://juejin.im/post/5b9879976fb9a05d330aa206>

- CMAKE_CURRENT_SOURCE_DIR
当前CMake 文件所在的文件夹路径
- CMAKE_SOURCE_DIR
指当前工程的 CMake 文件所在路径
- CMAKE_CURRENT_LIST_FILE
指当前 CMake 文件的完整路径
- PROJECT_SOURCE_DIR
指当前工程的路径

在文件的操作中，还有两个很重要的指令 GLOB 和 GLOB_RECURSE 。

```
# GLOB 的使用
file(GLOB ROOT_SOURCE *.cpp)
# GLOB_RECURSE 的使用
file(GLOB_RECURSE CORE_SOURCE ./detail/*.cpp)
```

复制代码

其中， GLOB 指令会将所有匹配 *.cpp 表达式的文件组成一个列表，并保存在 ROOT_SOURCE 变量中。

而 GLOB_RECURSE 指令和 GLOB 类似，但是它会遍历匹配目录的所有文件以及子目录下面的文件。

使用 `GLOB` 和 `GLOB_RECURSE` 有好处，就是当添加需要编译的文件时，不用再一个一个手动添加了，同一目录下的内容都被包含在对应变量中了，但也有弊端，就是新建了文件，但是 CMake 并没有改变，导致在编译时也会重新产生构建文件，要解决这个问题，就是动一动 CMake，让编译器检测到它有改变就好了。

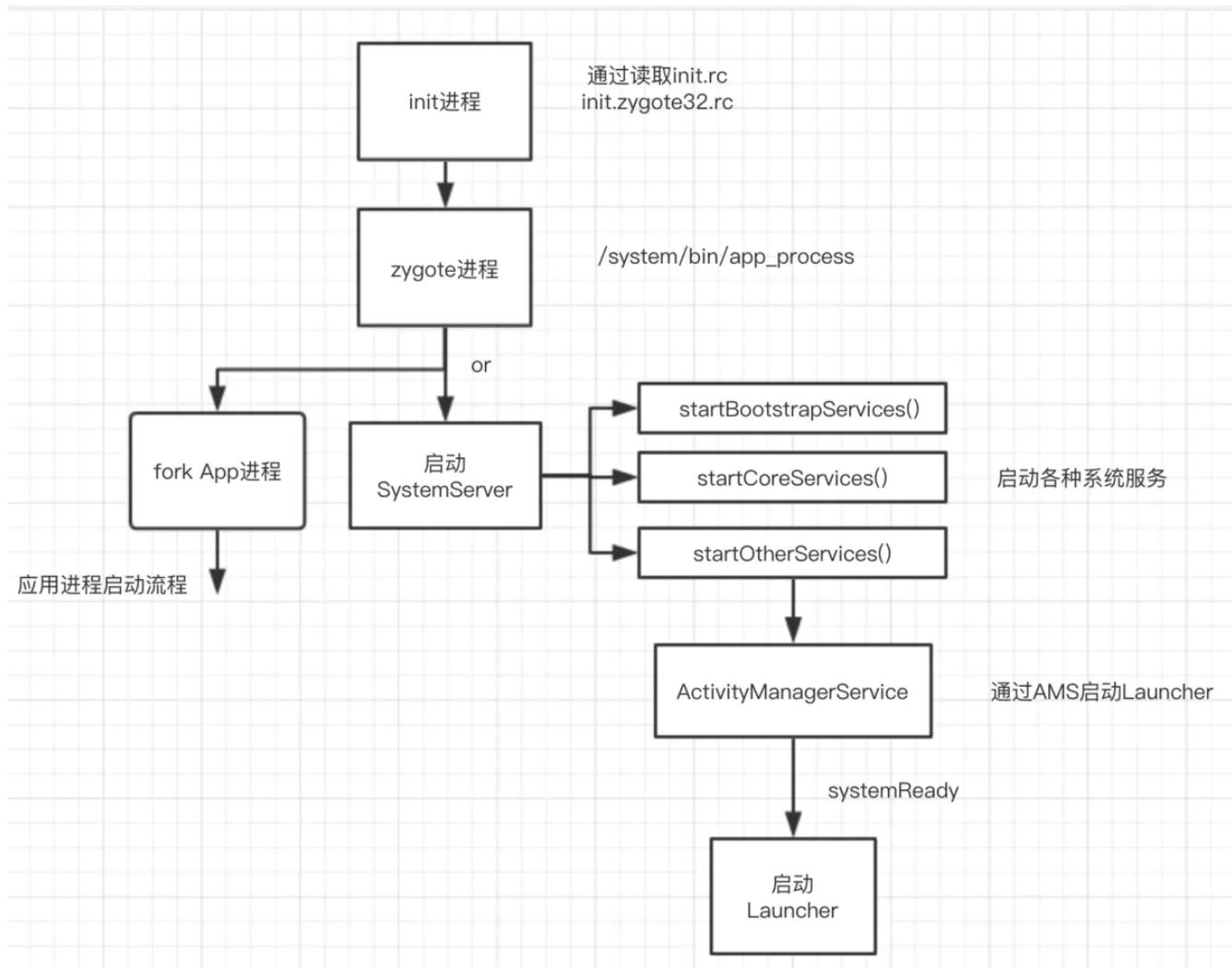
Hilt

只要初始化工作是外部做的，都叫依赖注入

底层是基于Dagger

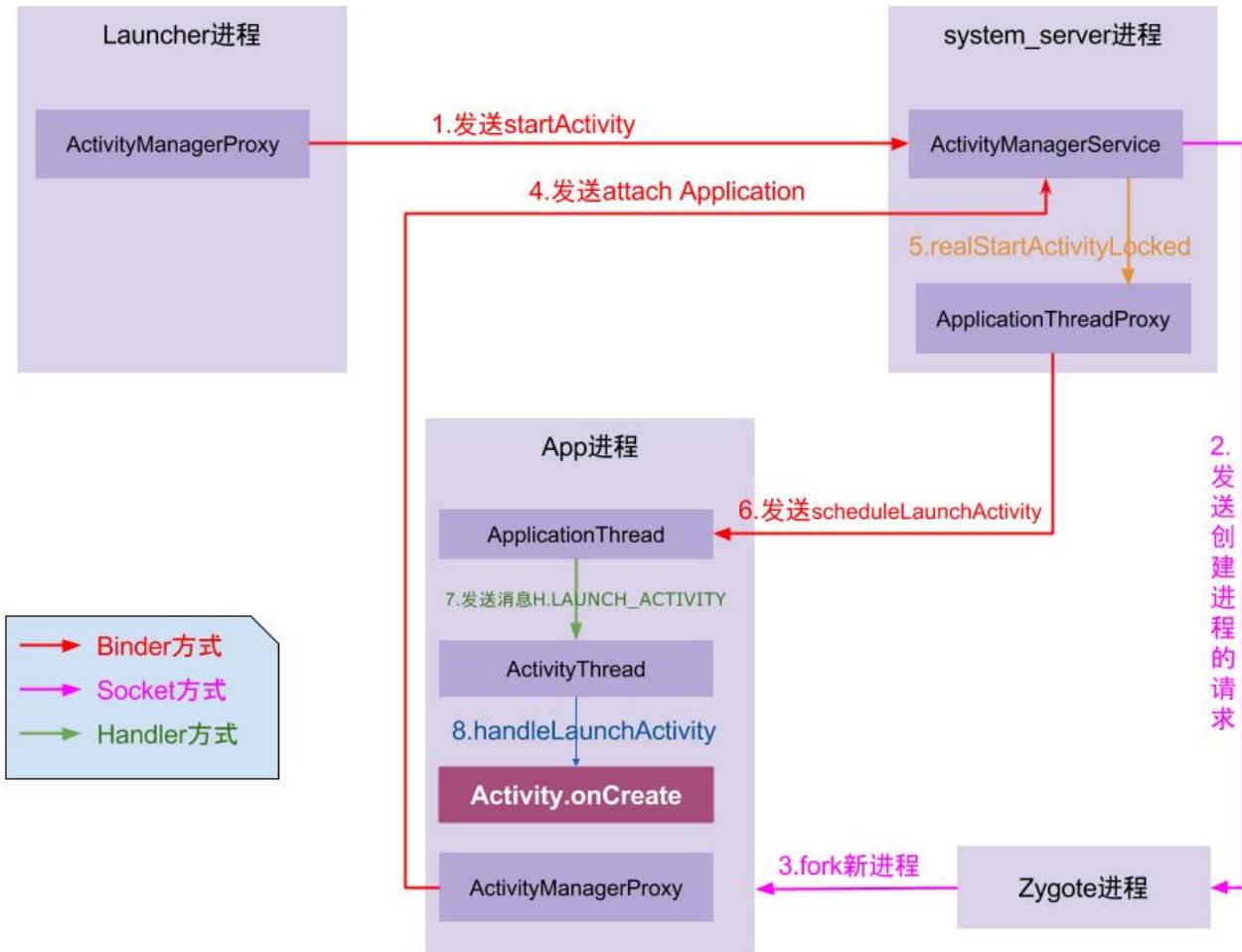
Android 系统启动流程

蓝师傅Android系统启动流程分析:<https://www.jianshu.com/p/9d8508a0982c> (这是系统的启动流程)



Activity 启动流程

链接：https://github.com/LRH1993/android_interview/blob/master/android/advance/app-launch.md



在Android系统中，任何一个Activity的启动都是由AMS和应用程序进程（主要是ActivityThread）相互配合来完成的。AMS服务统一调度系统中所有进程的Activity启动，而每个Activity的启动过程则由其所属的进程具体来完成。每个Activity都持有Instrumentation对象的一个引用，但是整个进程只会存在一个Instrumentation对象。Instrumentation这个类里面的方法大多数和Application和Activity有关，这个类就是完成对Application和Activity初始化和生命周期的工具类。Instrumentation这个类很重要，对Activity生命周期方法的调用根本就离不开他，他可以说是一个大管家。

Zygote 通过 `zygote.forkSystemServer` 创建SystemServer

AMS通过 socket 请求 Zygote for出一个新的应用程序进程。

Activity 启动模式

Standard

不管是否存在，在栈顶创建新的实例

SingleTask

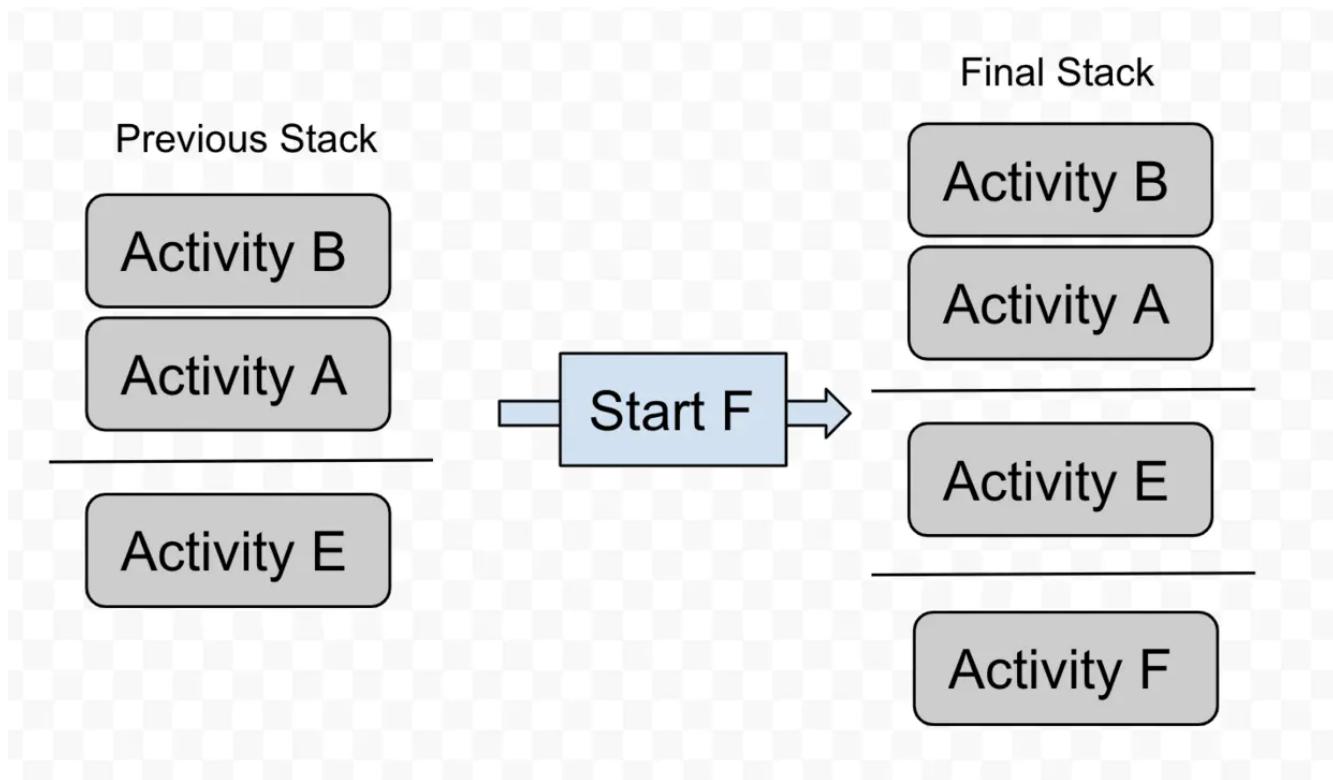
如果已经存在，则把该实例顶上的实例全部出栈，如果不存在则创建一个新的实例

SingleTop

如果栈顶存在该实例，则直接使用，如果不存在则创建新的实例

SingleInstance

会创建一个新的任务栈，并且里面只有它一个实例，如果已经存在这个实例，则复用已经创建的这个实例



因为 `singleInstance` 的属性是禁止与其他 Activities 共享任务栈，所以启动模式为 `SingleInstance` 的 Activity 启动其他 Activity 时会默认带有 `FLAG_ACTIVITY_NEW_TASK` 属性。所以 Activity E 启动 Activity F 后，最后会存在三个任务栈，Activity F 会单独存在于一个任务栈中

新增

HashMap 和 HashTable 以及 CurrentHashMap 区别

HashTable

数组+链表实现，key和value都不能为空，线程安全。

实现线程安全的方式是在修改数据时锁住整个HashTable。效率低。

初始大小为 11，每次扩容 = $\text{oldsize} * 2 + 1$

HashMap

数组+链表+红黑树，可以存储null键、值，线程不安全

ConcurrentHashMap

分段数组 + 链表，线程安全

MVVM

MVC 软件可以分为三部分

- 视图（View）: 用户界面
- 控制器（Controller）: 业务逻辑
- 模型（Model）: 数据保存

各部分之间的通信方式如下：

1. View 传送指令到 Controller
2. Controller 完成业务逻辑后，要求 Model 改变状态
3. Model 将新的数据发送到 View，用户得到反馈

Tips：所有的通信都是单向的。

#互动模式 接受用户指令时，MVC可以分为两种方式。一种是通过View接受指令，传递给Controller。

另一种是直接通过Controller接受指令

#MVP

MVP模式将Controller改名为Presenter，同时改变了通信方向。

1. 各部分之间的通信，都是双向的
2. View和Model不发生联系，都通过Presenter传递
3. View非常薄，不部署任何业务逻辑，称为"被动视图"(Passive View)，即没有任何主动性，而Presenter非常厚，所有逻辑都部署在那里。

缺点：

1. Presenter中除了应用逻辑以外，还有大量的View->Model，Model->View的手动同步逻辑，造成Presenter比较笨重，维护起来会比较困难。
2. 由于对视图的渲染放在了Presenter中，所以视图和Presenter的交互会过于频繁。
3. 如果Presenter过多地渲染了视图，往往会使它与特定的视图的联系过于紧密。一旦视图需要变更，那么Presenter也需要变更了。
4. 额外的代码复杂度及学习成本。

#MVVM

MVVM模式将Presenter改名为ViewModel，基本上与MVP模式完全一致。

唯一的区别是，它采用双向绑定(data-binding)：View的变动，自动反映在ViewModel，反之亦然。

Git 操作

- 创建本地仓库

```
git init
```

- 获取远程仓库

```
git clone [url]  
例:git clone https://github.com/you/yourpro.git
```

- 创建远程仓库

```
// 添加一个新的 remote 远程仓库  
git remote add [remote-name] [url]  
例:git remote add origin https://github.com/you/yourpro.git  
origin:相当于该远程仓库的别名
```

```
// 列出所有 remote 的别名  
git remote
```

```
// 列出所有 remote 的 url  
git remote -v
```

```
// 删除一个 remote  
git remote rm [name]
```

```
// 重命名 remote  
git remote rename [old-name] [new-name]
```

- 从本地仓库中删除

```
git rm file.txt          // 从版本库中移除, 删除文件  
git rm file.txt -cached // 从版本库中移除, 不删除原始文件  
git rm -r xxx           // 从版本库中删除指定文件夹
```

- 从本地仓库中添加新的文件

```
git add .                // 添加所有文件  
git add file.txt        // 添加指定文件
```

- 提交, 把缓存内容提交到 HEAD 里

```
git commit -m "注释"
```

- 撤销

```
// 撤销最近的一个提交。  
git revert HEAD  
  
// 取消 commit + add  
git reset --mixed  
  
// 取消 commit  
git reset --soft  
  
// 取消 commit + add + local working  
git reset --hard
```

- 把本地提交 push 到远程服务器

```
git push [remote-name] [loca-branch]:[remote-branch]  
例:git push origin master:master
```

- 查看状态

```
git status
```

- 从远端库中下载新的改动

```
git fetch [remote-name]/[branch]
```

- 合并下载的改动到分支

```
git merge [remote-name]/[branch]
```

- 从远端库中下载新的改动

```
pull = fetch + merge
```

```
git pull [remote-name] [branch]  
例:git pull origin master
```

- 分支

```
// 列出分支  
git branch  
  
// 创建一个新的分支  
git branch (branch-name)  
  
// 删除一个分支  
git branch -d (branch-nam)  
  
// 删除 remote 的分支  
git push (remote-name) :(remote-branch)
```

- 切换分支

```
// 切换到一个分支  
git checkout [branch-name]  
  
// 创建并切换到该分支  
git checkout -b [branch-name]
```

##与github建立ssh通信，让Git操作免去输入密码的繁琐。

- 首先呢，我们先建立ssh密匙。

```
ssh key must begin with 'ssh-ed25519', 'ssh-rsa', 'ssh-dss', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'. -- from github
```

根据以上文段我们可以知道github所支持的ssh密匙类型，这里我们创建ssh-rsa密匙。

在command line 中输入以下指令：``ssh-keygen -t rsa``去创建一个ssh-rsa密匙。如果你并不需要为你的密匙创建密码和修改名字，那么就一路回车就OK，如果你需要，请您自行Google翻译，因为只是英文问题。

```
$ ssh-keygen -t rsa Generating public/private rsa key pair. //您可以根据括号中的路径来判断你的.ssh文件放在了什么地方 Enter file in which to save the key (/c/Users/Liang Guan Quan/.ssh/id_rsa):
```

- 到 <https://github.com/settings/keys> 这个地址中去添加一个新的SSH key，然后把你的xx.pub文件下的内容文本都复制到Key文本域中，然后就可以提交了。
- 添加完成之后 我们用 `ssh git@github.com` 命令来连通一下github，如果你在response里面看到了你github账号名，那么就说明配置成功了。 *let's enjoy github ;)*

gitignore

在本地仓库根目录创建 .gitignore 文件。Win7 下不能直接创建，可以创建 ".gitignore." 文件，后面的标点自动被忽略；

```
/.idea      // 过滤指定文件夹  
/fd/*       // 忽略根目录下的 /fd/ 目录的全部内容;  
*.iml       // 过滤指定的所有文件  
.gitignore // 不忽略该文件
```

giffun

DvProgressView --- 摄像机加载样式进度条类

GifFunModule --- 自定义Glide模块，用来进行修改Glide 配置，进行Glide 自定义组件

ProgressResponseBody --- 实现Glide 加载进度条

待处理：

RecyclerView

Gradle 构建

第三方登录

Rxjava 原理以及优缺点

SVN + Git

Android Framework

下半月重点：



性能优化（内存优化，启动优化）



适配处理



Map系列源码



音视频复习

HashMap 数据结构是怎样的，可以多线程使用吗，多线程需要使用哪些集合？

数组作为哈希桶，扩容时候已链表形式，超过8个节点结构改为红黑树。不能多线程使用，多线程可以用 ConcurrentHashMap

ConcurrentHashMap 为什么可以多线程使用， jdk 1.7 和 1.8 有何区别

HashMap 有哪些高效的运算，如何扩容

答：取key的hashCode值、高位运算、取模运算当length总是2的n次方时， $h \& (length-1)$ 运算等价于对length取模，也就是 $h \% length$ ，但是 $\&$ 比%具有更高的效率。可以参考美团的：[Java 8系列之重新认识HashMap](#)

为什么 setContentView 要放在 onCreate 里面？能不能放在 onStart 或者 attachBaseContext？

如果放在 onStart 或者 onResume，我们平时切换布局会来回调用生命周期，setContentView 会调用多次，这是没必要的。如果放在 attachBaseContext 那也是不行的，因为 setContentView 需要获取一个 mWindow 对象，而这个对象是在 attach 里面才进行初始化的，所以会获取不到对象。

通过代码设置全屏、去标题栏等属性的时候，需要在 setContentView 前面调用，因为 setContentView 方法里面进行布局属性设置的时候会进行判断，如果在之后设置的话就没效果了。

```
getWindow().requestFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(FLAG_FULLSCREEN, FLAG_FULLSCREEN );
```

XML解析有哪些方式

SAX (Simple API XML)

是一种基于事件的解析器，事件驱动的流式解析方式是，从文件的开始顺序解析到文档的结束，不可暂停或倒退

DOM

即对象文档模型，它是将整个XML文档载入内存(所以效率较低，不推荐使用)，每一个节点当做一个对象

Pull

Android解析布局文件所使用的方式。Pull与SAX有点类似，都提供了类似的事件，如开始元素和结束元素。不同的是，SAX的事件驱动是回调相应方法，需要提供回调的方法，而后在SAX内部自动调用相应的方法。而Pull解析器并没有强制要求提供触发的方法。因为他触发的事件不是一个方法，而是一个数字。它使用方便，效率高。

ArrayList 和 LinkedList 的区别

<https://cloud.tencent.com/developer/article/1451016>

ArrayList

初始化大小为 10，插入新元素的时候，会判断是否需要扩容，扩容的长度是原来容量的1.5倍，扩容的方式是复制，因此有一定的开销。

ArrayList在进行插入元素的时候，需要移动插入位置之后的所有元素，位置越靠前，需要位移的元素越多，开销越大，相反，插入位置越靠后，开销就越小，如果在最后面进行插入，那就不需要进行位移。

```
public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

    ensureCapacityInternal(minCapacity: size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, destPos: index + 1,
                    length: size - index);
    elementData[index] = element;
    size++;
}
```

插入到指定位置，进行复制操作，所以效率比较低。

```

    public E get(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

        return (E) elementData[index];
    }

```

直接返回数组该位置的元素，所以查询比较快，时间复杂度为O(1)

扩容

```

/*
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);           ← 右移一位相当于除以2
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}

private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}

```

如果数组空间不足，则进行扩容，容量是扩容前的1.5倍，扩容完之后需要进行复制操作，比较耗时

LinkedList

内部使用双向链表结构，每一个元素（结点）的地址不连续，通过引用找到当前结点的上一个结点和下一个结点，即插入和删除效率较高，只需要常数时间，而get和set则较为低效。LinkedList的方法和使用和ArrayList大致相同，由于LinkedList是链表实现的，所以额外提供了在头部和尾部添加/删除元素的方法，也没有ArrayList扩容的问题了。另外，ArrayList和LinkedList都可以实现栈、队列等数据结构，但LinkedList本身实现了队列的接口，所以更推荐用LinkedList来实现队列和栈。

```

//LinkedList 源码

public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}

```

```

Node<E> node(int index) {
    // assert isElementIndex(index);
    //如果 index 小于总数的一半, 则从first节点向后遍历, 直到找到index节点, 然后返回该节点的值。
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        //index 大于总数的一半, 从last节点向前遍历, 直到找到index节点, 然后返回该节点的值。
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}

```

采用双向链表存储数据, 查询的时候需要从链表头 或者 链表尾 进行遍历, 直到找到要查询的元素, 数据多的时候需要遍历较长时间, 所以查询效率比较低, 时间复杂度为O(n/2)

ArrayList在插入或者删除时, 需要移动插入位置之后的所有元素, 因此速度较慢, 时间复杂度为O(n)。而LinkedList只需要找到该位置, 移动“指针”即可, 时间复杂度为O(1)。

扩容

LinkedList 是双向链表结构, 添加数据直接在后面新添加一个node对象, 不需要扩容机制

因此, 在使用ArrayList 的时候, 如果能够预估大小, 可以直接初始化容量, 避免频繁进行扩容带来的性能消耗.

在需要频繁读取集合中的元素时, 使用ArrayList效率较高, 而在插入和删除操作较多时, 使用LinkedList效率较高。

如果只在尾部频繁添加数据, 那ArrayList效率高一点, 因为ArrayList 直接把数据放到数组里面, LinkedList 则需要创建一个node对象来进行存储.

ArrayMap 和 HashMap 的区别:

1. 查找效率 HashMap因为其根据hashcode的值直接算出index, 所以其查找效率是随着数组长度增大而增加的。 ArrayMap使用的是二分法查找, 所以当数组长度每增加一倍时, 就需要多进行一次判断, 效率下降
2. 扩容数量 HashMap初始值16个长度, 每次扩容的时候, 直接申请双倍的数组空间。 ArrayMap每次扩容的时候, 如果size长度大于8时申请size*1.5个长度, 大于4小于8时申请8个, 小于4时申请4个。这样比较 ArrayMap其实是申请了更少的内存空间, 但是扩容的频率会更高。因此, 如果数据量比较大的时候, 还是使用HashMap更合适, 因为其扩容的次数要比ArrayMap少很多。
3. 扩容效率 HashMap每次扩容的时候重新计算每个数组成员的位置, 然后放到新的位置。 ArrayMap则是直接使用System.arraycopy, 所以效率上肯定是ArrayMap更占优势。

4. 内存消耗 以ArrayMap采用了一种独特的方式，能够重复的利用因为数据扩容而遗留下来的数组空间，方便下一个ArrayMap的使用。而HashMap没有这种设计。由于ArrayMap之缓存了长度是4和8的时候，所以如果频繁的使用到Map，而且数据量都比较小的时候，ArrayMap无疑是相当的是节省内存的。
5. 总结 综上所述，数据量比较小，并且需要频繁的使用Map存储数据的时候，推荐使用ArrayMap。而数据量比较大的时候，则推荐使用HashMap。

- 数据结构
 - ArrayMap和SparseArray采用的都是两个数组，Android专门针对内存优化而设计的
 - HashMap采用的是数据+链表+红黑树
- 内存优化
 - ArrayMap比HashMap更节省内存，综合性能方面在数据量不大的情况下，推荐使用ArrayMap
 - Hash需要创建一个额外对象来保存每一个放入map的entry，且容量的利用率比ArrayMap低，整体更消耗内存
 - SparseArray比ArrayMap节省1/3的内存，但SparseArray只能用于key为int类型的Map，所以int类型的Map数据推荐使用SparseArray
- 性能方面：
 - ArrayMap查找时间复杂度O(logN) ArrayMap增加、删除操作需要移动成员，速度相比较慢，对于个数小于1000的情况下，性能基本没有明显差异
 - HashMap查找、修改的时间复杂度为O(1)
 - SparseArray适合频繁删除和插入来回执行的场景，性能比较好
- 缓存机制
 - ArrayMap针对容量为4和8的对象进行缓存，可避免频繁创建对象而分配内存与GC操作，这两个缓存池大小的上限为10个，防止缓存池无限增大；
 - HashMap没有缓存机制
 - SparseArray有延迟回收机制，提供删除效率，同时减少数组成员来回拷贝的次数
- 扩容机制
 - ArrayMap是在容量满的时机触发容量扩大至原来的1.5倍，在容量不足1/3时触发内存收缩至原来的0.5倍，更节省的内存扩容机制
 - HashMap是在容量的0.75倍时触发容量扩大至原来的2倍，且没有内存收缩机制。HashMap扩容过程有hash重建，相对耗时。所以能大致知道数据量，可指定创建指定容量的对象，能减少性能浪费。
- 并发问题
 - ArrayMap是非线程安全的类，大量方法中通过对mSize判断是否发生并发，来决定抛出异常。但没有覆盖到所有并发场景，比如大小没有改变而成员内容改变的情况就没有覆盖
 - HashMap是在每次增加、删除、清空操作的过程将modCount加1，在关键方法内进入时记录当前mCount，执行完核心逻辑后，再检测mCount是否被其他线程修改，来决定抛出异常。这一点的处理比ArrayMap更有全面。
- ArrayList的使用，ArrayList使用过程中有没有遇到过坑



Handler中loop方法为什么不会导致线程卡死

<https://juejin.im/post/5c5694b951882562e5441e71>



还有就是[动画的原理](#)，问他有没有实战过补间动画



[卡顿优化](#)

- MVVM优势



[MeasureSpec](#)的意义，一般怎样计算MeasureSpec；自定义View和自定义ViewGroup的区别
onmeasure, onLayout, onDraw的调度流程；自定义View的measure时机；有没有写过自定义View

- 问了Glide使用过程中的坑
- EventBus使用过程中的坑



[多线程](#)开发中你都在哪些地方使用过sleep, wait, 分别怎么使用的

- 网络协议okhttp中的缓存机制
- dex加载流程
- 组件化的原理



Kotlin优缺点，跟Java的区别，为什么说编译慢

优点:不用findid查找布局,协程更简洁的线程写法, 高阶函数, with apply let 关键字带来的简便等

缺点:apk 会变大, 会增加一些kotlin的库文件,编译的类增加了一些方法



Fragment的生命周期管理过程中遇到的坑和解决办法

https://github.com/LRH1993/android_interview/blob/master/android/basis/Fragment.md

- 排序算法，还有观察者模式和单例模式，还问了抽象类和接口的关系
- 应用启动流程，activity 启动流程
- 为什么要每个应用有一个自己的虚拟机，这个虚拟机和JVM的关系。
- databinding原理，binder原理
- 多进程通信问题；binder优势；aidl生成的java类细节；多进程遇到过哪些问题？
- 子线程中维护的looper，消息队列无消息时候的处理节省性能的处理方案

handler.getLooper().quitSafely(); 释放内存 释放线程

- 你遇到的最难的技术问题和解决方案

你知道广播有哪几种吗？

那么广播的注册方式有哪几种？

那广播是怎么传送的？

问题的答案就是binder，你敢想象吗？

2) 你知道activity的生命周期管理协调方式么? (如果你看过, 对是handler)

那么它为什么用handler来协调管理 (傻眼了没?)



[Handler机制](#) 子线程可以创建handler吗? 一个线程是否只有一个Looper? 如何保证一个线程只有一个Looper?

recyclerView缓存原理?



[Http和Https的区别?](#)

看过哪些源码? Glide原理?

OkHttp原理?

Retrofit原理? 为何用代理? 代理的作用是什么?

ButterKnife原理? 用到反射吗? 为什么?

eventbus的原理。

[Handler原理](#)

Binder原理

ANR(应用程序无响应)异常如何查找并分析?

JVM内存模型? 性能调优?



[垃圾收集算法有哪些?](#) [G1算法?](#)



[加密算法](#)有哪些? 对称加密和非对称加密的区别?

TCP的三次握手? 两次行不行? 为什么? TCP攻击知道吗? 如何进行攻击?

实现Glide 加载 带进度条-

Glide 内存缓存, 磁盘缓存是在哪里写入的?

当图片加载完成后, 会在EngineJob 当中通过 Handler 发送一条消息, 将执行逻辑切回到主线程当中, 从而执行 handlerResultOnMainThread()方法

在里面会构建一个包含图片资源的EngineResoure 对象, 然后会将这个对象回调到Engine 的 onEngineJobComplete() 方法当中, 就是在里面写入的弱引用缓存

弱引用是通过acquired变量来记录图片引用的, 当acquired 为0 ,也就是图片不再使用了的时候, 就会进行释放, 从弱引用中移除, 并添加到 Lru缓存中去

弱引用缓存则减轻了 lru缓存的压力, 避免lru过满导致频繁gc, 并且提高查找效率---如何减轻lru缓存压力?

Glide 三级缓存是 弱引用, Lrucache, 磁盘缓存 还是 内存缓存, 磁盘缓存, 网络缓存?

Okhttp 自定义拦截器

HashMap 源码分析--



为什么 [view.post](#) 在布局加载完后才执行

<https://www.cnblogs.com/dasusu/p/8047172.html>

<https://www.jianshu.com/p/101e2edd29d1>

网络连接怎么实现复用?

OkHttp如何做网络监控?

OkHttp网络缓存如何实现的?



[glide](#) 为什么使用弱引用 而不是 软引用



[Fresco](#) [为什么适合加载大量图片](#)的场景



谈谈消息机制[Handler](#)作用? 有哪些要素? 流程是怎样的?

组件化 和 插件化的区别



什么是内存屏障?

在安卓中, 你在哪些实际场景中被多线程问题折磨过?

android 打包流程



说一下[自定义View](#)需要注意哪些细节 主要说了一下View绘制的三大过程onDraw()不要做耗时view的事件冲突的解决办法 ps: 重写dispatchEvent()或者touch()方法



什么是乐观锁, 什么是悲观锁 --> [锁](#)



[Activity的启动流程](#)



[Activity的启动模式](#), 应用场景



如何减少对第三方框架的耦合(进行二次封装)



[hashmap](#)的基本原理, [CocurrentHashMap](#)和[Hashtable](#) 有啥区别



Handler机制

- Binder机制 (IPC、AIDL 的使用)
- 写一个单例模式,DCL为什么要加V关键字
- 直接在Activity Sleep 5000ms,再post一个runnable会不会ANR。
- 如何监听ANR



[View的绘制流程](#),每个都是干什么的



事件传递机制

- 组件化如何实现组件通信



[HashMap基本原理](#) 如何减少hash冲突 如何增大hash冲突



[安装包怎么优化的](#)



如果有A,B,C,D,E五个步骤,每个步骤都需要操作对应请求,用哪种设计模式。(责任链模式)



[JVM垃圾回收](#)有哪几种算法,有哪几种垃圾回收处理器



静态类的静态方法能不能被子类重写(不能)



[HashMap,ConcurrentHashMap,HashTable有什么异同](#)

- Binder通信机制
- 红包随机算法 是怎么做到的
- 写一个二叉树的深度优先遍历,递归 非递归
- 为什么android使用了Binder机制没采用共享内存
- 线程池是如何管理线程状态的



Kotlin的访问权限



Kotlin为什么比Java更安全



Kotlin比Java好在哪



[ArrayList跟LinkedList区别](#)



ArrayList如何扩容的,如何提高ArrayList的效率

- 动态代理静态代理区别



[apk为什么要签名,v1跟v2签名的区别](#)



[apk从编译打包流程](#)



apk 安装流程

- dex到odex为什么不能在编译期优化
- http2.0 1.1 1.0的区别



View的绘制流程,每个方法干什么的,如果要获取View的宽高,在哪个方法里获取

- 如何应对弱网环境
- 如果一个app无法访问网络,你怎么做数据存储。
- Android 沉浸式状态栏 怎么实现的
- eventbus为什么要用CopyOnWriteList?
- 插件化怎么加载资源的?
- volatile 跟 synchronized区别?
- 接口暴露,但是不让别人调用,有哪些办法?
- 正则表达式使用
- Java虚拟机, 垃圾回收
- GSON
- RxJava+Retrofit



图片缓存, 三级缓存

- 四大组件
- Fragment生命周期, 嵌套
- AsyncTask机制
 - 网络框架的搭建
- 写个图片浏览器, 说出你的思路
- Toolbar的使用
- SharedPreference

问题

systrace , traeview, MAT 等工具的使用

生产者消费者模型



内存泄漏及解决方法



[线程池](#)



[LruCache](#)



大图片处理

- ~~Activity A 跳转 B 生命周期~~
- ~~retrofit源码~~



[okhttp源码](#)

- MVVM优势
- MVP接口爆炸怎么解决



卡顿解决



[线程池以及执行策略](#)

- 泛型
- invalidate原理



锁

- ThreadLocal
- handler.postDelay原理
- 组件化优势



[协程和线程区别](#)



[rxjava源码，线程切换原理](#)

- 协程和线程区别



kotlin inline关键字



kotlin 高阶函数

- 内存优化，内存泄漏
- jvm内存区域

- 滑动冲突



gc机制

- message数据结构，排序

- SparseArray

- lifecycle原理



协程async和launch区别

launch() 更多是用来发起一个无需结果的耗时任务(如批量删除文件, 创建文件), 不需要返回值的场景.

async() 函数 主要用于异步执行耗时任务, 并且需要返回值(如网络请求, 数据库读写), 执行完毕后可以通过 .await() 函数获取返回值.



kotlin let和apply区别



kotlin 高阶函数



[synchronized](#) 修饰实例方法和修饰静态方法有啥不一样

- invalidate

- synchronized和lock区别

- 堆和栈

- lifecycle原理

- Arouter原理

- bugly崩溃处理, 性能优化相关指标

- 两个字符串最大公共子串

- 快排

- 创建对象内存分配

- 什么是多态、多态底层原理

- 视频流协议

- 写得最好的自定义view

- tcp/ip握手挥手

- view, window, activity的关系, 问了很多源码

- view属性动画

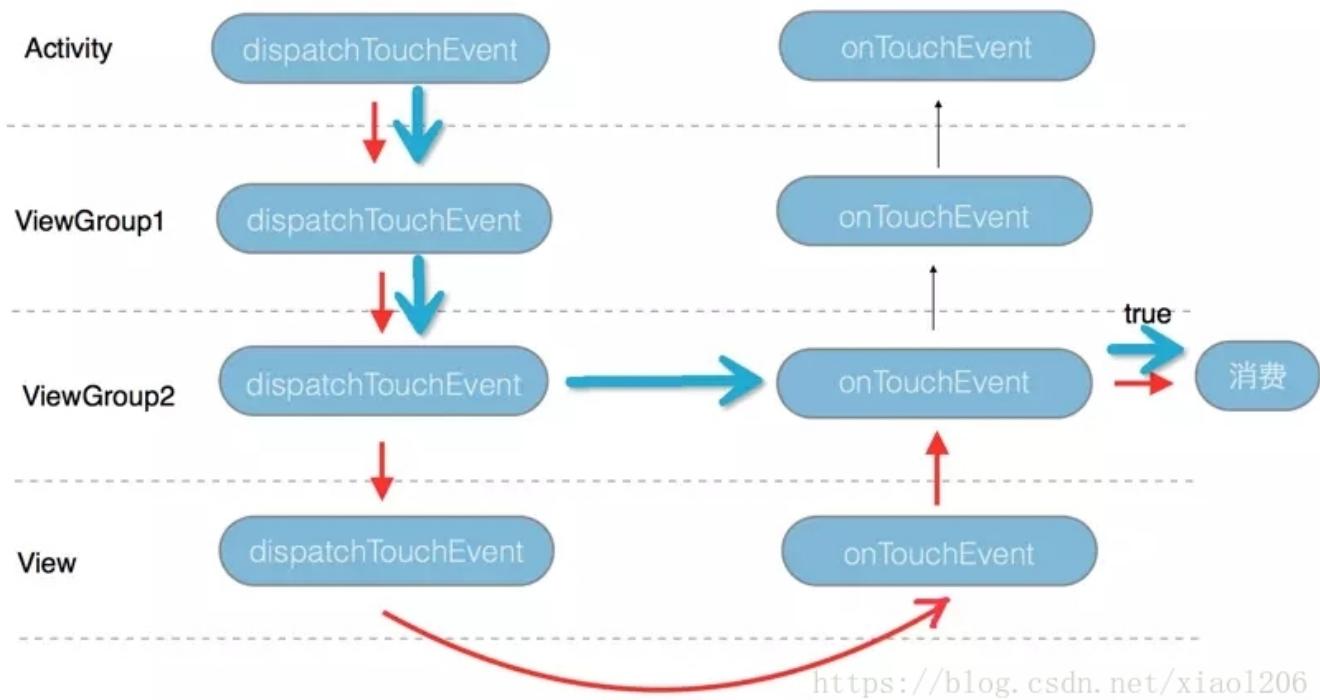


kotlin inline



kotlin 扩展函数

- SparseArray、ArrayMap
- RecyclerView性能优化
- MVP和MVVM MVP：大致就说下项目中MVP怎么写的。View层负责更新ui，presenter持有view层引用，通过model层获取数据回调到view层。 MVVM：使用ViewModel和LiveData处理生命周期问题，观察者模式解耦view层引用问题。然后讲讲lifecycle源码。
- 协程 依赖线程，轻量线程，然后说说挂起恢复机制。结合ViewModel协程扩展库和retrofit使用。关于协程扩展库，自己写个job，在ViewModel.onCleared()里cancle()也是一样的。
- view绘制流程 Activity.makeVisible()--->WindowManager.addView()--->ViewRootImpl.performTraversals() --->performMeasure()、performLayout()、performDraw() ---顶层view--->DecorView--->onMeasure()、onLayout()、onDraw()



一个线程能否创建多个Handler，Handler跟Looper之间的对应关系？

- 参考回答：
 - 一个Thread只能有一个Looper，一个MessageQueue，可以有多个Handler
 - 以一个线程为基准，他们的数量级关系是： Thread(1) : Looper(1) : MessageQueue(1) : Handler(N)

、Looper死循环为什么不会导致应用卡死？

- 参考回答：
 - 主线程的主要方法就是消息循环，一旦退出消息循环，那么你的应用也就退出了，Looper.loop () 方法可能会引起主线程的阻塞，但只要它的消息循环没有被阻塞，能一直处理事件就不会产生ANR 异常。

- 造成ANR的不是主线程阻塞，而是主线程的Looper消息处理过程发生了任务阻塞，无法响应手势操作，不能及时刷新UI。
- 阻塞与程序无响应没有必然关系，虽然主线程在没有消息可处理的时候是阻塞的，但是只要保证有消息的时候能够立刻处理，程序是不会无响应的。

链接：<https://juejin.im/post/5c85cead5188257c6703af47>

Android中还了解哪些方便线程切换的类？

- 参考回答：
 - AsyncTask：底层封装了线程池和Handler，便于执行后台任务以及在子线程中进行UI操作。
 - HandlerThread：一种具有消息循环的线程，其内部可使用Handler。
 - IntentService：是一种异步、会自动停止的服务，内部采用HandlerThread。

IntentService有什么用？

- 参考回答：
 - IntentService可用于执行后台耗时的任务，当任务执行完成后会自动停止，同时由于IntentService是服务的原因，不同于普通Service，IntentService可自动创建子线程来执行任务，这导致它的优先级比单纯的线程要高，不容易被系统杀死，所以IntentService比较适合执行一些高优先级的后台任务。

直接在Activity中创建一个thread跟在service中创建一个thread之间的区别？

- 参考回答：
 - 在Activity中被创建：该Thread的就是为这个Activity服务的，完成这个特定的Activity交代的任务，主动通知该Activity一些消息和事件，Activity销毁后，该Thread也没有存活的意义了。
 - 在Service中被创建：这是保证最长生命周期的Thread的唯一方式，只要整个Service不退出，Thread就可以一直在后台执行，一般在Service的onCreate()中创建，在onDestroy()中销毁。所以，在Service中创建的Thread，适合长期执行一些独立于APP的后台任务，比较常见的就是：在Service中保持与服务器端的长连接。

Handler、Thread和HandlerThread的差别？

- 参考回答：
 - Handler：在android中负责发送和处理消息，通过它可以实现其他支线任务与主线程之间的消息通讯。
 - Thread：Java进程中执行运算的最小单位，亦即执行处理机调度的基本单位。某一进程中一路单独运行的程序。
 - HandlerThread：一个继承自Thread的类HandlerThread，Android中没有对Java中的Thread进行任何封装，而是提供了一个继承自Thread的类HandlerThread类，这个类对Java的Thread做了很多便利的封装。HandlerThread继承于Thread，所以它本质就是个Thread。与普通Thread的差别就在于，它在内部直接实现了Looper的实现，这是Handler消息机制必不可少的。有了自己的looper，可以让我们在自己的线程中分发和处理消息。如果不用HandlerThread的话，需要手动去调用Looper.prepare()和Looper.loop()这些方法。

ListView跟RecyclerView的区别

- 参考回答：
 - 动画区别：

- 在RecyclerView中，内置有许多动画API，例如：`notifyItemChanged()`, `notifyDataInserted()`, `notifyItemMoved()`等等；如果需要自定义动画效果，可以通过实现（`RecyclerView.ItemAnimator`类）完成自定义动画效果，然后调用`RecyclerView.setItemAnimator()`；
- 但是ListView并没有实现动画效果，但我们在Adapter自己实现item的动画效果；
- 刷新区别：
 - ListView中通常刷新数据是用全局刷新`notifyDataSetChanged()`，这样一来就会非常消耗资源；本身无法实现局部刷新，但是如果要在ListView实现局部刷新，依然是可以实现的，当一个item数据刷新时，我们可以在Adapter中，实现一个`onItemChanged()`方法，在方法里面获取到这个item的position（可以通过`getFirstVisiblePosition()`），然后调用`getView()`方法来刷新这个item的数据；
 - RecyclerView中可以实现局部刷新，例如：`notifyItemChanged()`；
- 缓存区别：
 - RecyclerView比ListView多两级缓存，支持多个离ItemView缓存，支持开发者自定义缓存处理逻辑，支持所有RecyclerView共用同一个RecyclerViewPool(缓存池)。
 - ListView和RecyclerView缓存机制基本一致，但缓存使用不同

http有哪几种版本,版本间有什么区别。https跟http有什么区别

1. HTTP 0.9

- 只支持`GET`请求方式：由于不支持其他请求方式，因此客户端是没办法向服务端传输太多的信息
- 没有请求头概念：所以不能在请求中指定版本号，服务端也只具有返回HTML字符串的能力
Android
- 服务端相响应之后，立即关闭TCP连接

2. HTTP 1.0

- 请求方式新增了`POST`, `DELETE`, `PUT`, `HEADER`等方式
- 增添了请求头和响应头的概念，在通信中指定了HTTP协议版本号，以及其他的一些元信息（比如：状态码、权限、缓存、内容编码）
- 扩充了传输内容格式，图片、音视频资源、二进制等都可以进行传输

安装包,内存,卡顿,启动优化

耗电检测

屏幕刷新机制及原理

UI原理及优化方式

Glide, Okhttp, EventBus, RxJava(线程切换原理) 原理

LiveData, Room, ViewModel, DataBinding使用

视频编解码,封装和解封装

Android版本适配

设计模式

屏幕适配

Handler

反射

消息推送

排序算法

listView 和 RecyclerView 的区别

view 的绘制原理

2. 反射是什么，在哪里用到，怎么利用反射创建一个对象

3. 代理模式与装饰模式的区别，手写一个静态代理，一个动态代理

6. 自定义View,事件分发机制讲一讲

7. 有做过什么Bitmap优化的实际经验

13. 讲一下RecyclerView的缓存机制，滑动10个，再滑回去，会有几个执行onBindViewHolder

14. 如何实现RecyclerView的局部更新，用过payload吗，notifyItemChange方法中的参数？

3. ViewModel为什么在旋转屏幕后不会丢失状态

14. kotlin与Java互相调用有什么问题？

5. 有用过什么加密算法？AES,RSA什么原理？

7. SharedParence可以跨进程通信吗？如何改造成可以跨进程通信的.commit和apply的区别。

4. 说说你对volatile字段有什么用途？

. 说说你对屏幕刷新机制的了解，双重缓冲，三重缓冲

18. 有用DSL,anko写过布局吗？

20. 阿里编程规范不建议使用线程池，为什么？

1. 泛型有什么优点？

2. 动态代理有什么作用？

Socket 与 Http 对比

- `Socket` 属于传输层，因为 `TCP / IP` 协议属于传输层，解决的是数据如何在网络中传输的问题
- `HTTP` 协议 属于 应用层，解决的是如何包装数据

由于二者不属于同一层面，所以本来是没有可比性的。但随着发展，默认的Http里封装了下面几层的使用，所以才会出现 `Socket & HTTP` 协议的对比：（主要是工作方式的不同）：

- `Http`：采用 请求—响应 方式。

1. 即建立网络连接后，当 客户端 向 服务器 发送请求后，服务器端才能向客户端返回数据。

2. 可理解为：是客户端有需要才进行通信

- **Socket**：采用 服务器主动发送数据 的方式

1. 即建立网络连接后，服务器可主动发送消息给客户端，而不需要由客户端向服务器发送请求

2. 可理解为：是服务器端有需要才进行通信