# OnStart 与 onResume区别

先执行onStart，在执行onResume，onStart时当前activity可见，onResume时用户可以与activity进行交互

# activity view window区别

每一个activity都有一个window，叫做phoneWindow,phoneWindow里面有一个decorView，decorView分为titleView与contentView

# 按钮事件分发

郭霖博客：

http://blog.csdn.net/guolin\_blog/article/details/9097463

普通情况：给一个button添加onClick、OnTouck事件监听，onTouch实现的内容要多一些，包括手指的拿起放下等等

同时监听时，先是onTouch再是onClick，onTouch至少按下抬起这两次

onTouch有返回值，如果返回True，那么后面的onClick便不会再执行

在点击button之后，会首先执行dispatchTouchEvent（），但这个函数button以及父类textView都没有，只能从

view中找到

```

public boolean dispatchTouchEvent(MotionEvent event) {

if (mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED &&

mOnTouchListener.onTouch(this, event)) {

return true;

}

return onTouchEvent(event);

}

```

onClick在onTouchEvent中，这和前面提到的顺序一致

在dispatchEvent中只有前一个action返回true，后面才会接着执行

# viewGroup事件分发

自定义一个布局MyLayout继承LinearLayout，里面添加两个button，都设置onClick监听，给MyLayout设置onTouch监听，点击button1和2会显示onClick，而onTouch没有反应，点击其他地方，会打印出onTouch

首先，touch事件的传递先传到viewGroup，再传到view

先执行viewGroup的dispatchEvent事件，在里面如果

```

if (disallowIntercept || !onInterceptTouchEvent(ev))

```

（disallowIntercept默认为false，可通过```requestDisallowIntercept（）```关闭,而onInterceptTouchEvent（）默认返回false，也可以覆盖重写）

成立，那么就开始执行子view的dispatchEvent事件，如果子view可点击那么一定返回true，则不会执行viewgroup的touch事件，如果不可点击，那么会

接着执行super.dispatchEvent()，也就是view的dispatchEvent，所以onTouch会接着执行

# view绘制过程

measure->layout->draw

对于子view就是一个递归过程

measure过程中系统会将ViewGroup.LayoutParams根据父容器的规则转换为MeasureSpec，在measure中便可以使用这个参数

\*\*ViewGroup.LayoutParams\*\*:match\_parent...

\*\*MeasureSpec\*\*:前两位表示测量模式，后三十位表示该模式下的测量大小：exact、at\_most、不限制

measure() 最终还是会调用onMeasure（）

onMeasure()我们自定义时需要复写的，最终会调用setMeasureDimension()

继承view之后，在其onDraw（）方法内部，如果写在super.onDraw()前面那么，会先于父view绘制，反之

## layout

在viewGroup中onLayout是一个虚函数，

dispatchDraw（）负责处理子view的绘制，写在super.dispatchDraw（）后面，那么子view绘制的内容就会被我们所写的代码所覆盖，写在上面和写在super.onDraw()后面一样

绘制的过程：

- 背景（drawBackground()无法重写）

- 主体（onDraw()）

- 子 View（dispatchDraw()）

- 滑动边缘渐变和滑动条

- 前景

后两个的执行函数是onDrawForeGround（）可以重写，写在super.onDrawForeGround（）下面绘制内容会覆盖滑动渐变、滑动条以及前景，比如绘制new标签



写在上面和写在super.dispatchDraw()下面一样

上述的方法都在```draw()```中执行，将代码写在super.draw()下面和重写super.onDrawForeGround()下面一样，写在super.draw()上面可以被所有绘制内容所覆盖，包括

背景，可以用来给editText加背景



# handler机制

郭霖博客

ttp://blog.csdn.net/guolin\_blog/article/details/9991569

注意：在子线程中创建handler要记得Looper.prepare();

Handler通过TreadLocal保存Looper，每个thread的set、get方法都是针对本Thread

各种sendMessage()函数最终会调用sendMessageAtTime（），里面调用了queue.enQueue(msg,uptimeMills)

在这个函数里面把msg当做链表的一个节点(按upTimeMills排列)，每加入一个msg，将其添加到链表的合适位置

```

Message prev = null;

while (p != null && p.when <= when) {

prev = p;

p = p.next;

}

msg.next = prev.next;

prev.next = msg;

this.notify();

```

Looper.loop()里面是一个循环，将消息队列中的消息一个个取出

```

public static final void loop() {

Looper me = myLooper();

MessageQueue queue = me.mQueue;

while (true) {

Message msg = queue.next(); // might block

if (msg != null) {

if (msg.target == null) {

return;

}

if (me.mLogging!= null) me.mLogging.println(

">>>>> Dispatching to " + msg.target + " "

+ msg.callback + ": " + msg.what

);

msg.target.dispatchMessage(msg);

if (me.mLogging!= null) me.mLogging.println(

"<<<<< Finished to " + msg.target + " "

+ msg.callback);

msg.recycle();

}

}

}

```

\*\*新建进程的标准用法\*\*

```

class LooperThread extends Thread {

public Handler mHandler;

public void run() {

Looper.prepare();

mHandler = new Handler() {

public void handleMessage(Message msg) {

// process incoming messages here

}

};

Looper.loop();

}

}

```

\*\*其他用法\*\*

post(Runnable r)

view中找到中的post就是直接调用了这个函数，还有runOnUIThread(Runnable r)也是

```

private Handler handler;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

handler = new Handler();

new Thread(new Runnable() {

@Override

public void run() {

handler.post(new Runnable() {

@Override

public void run() {

// 在这里进行UI操作

}

});

}

}).start();

}

```

# AsyncTask

http://blog.csdn.net/guolin\_blog/article/details/11711405

封装了线程池和Handler的异步框架，执行异步任务，方便在ui线程和子线程中切换，一般做一些耗时短的操作

同handler一样，容易产生内存泄漏与，不会随Activity销毁而销毁，可以调用cancel，建议使用串行（现阶段默认串行，可以用executeOnExecutor(exec)并行）

# HandlerThread

handler+thread+looper，继承了Thread，可以进行looper循环，不能同时进行多任务的处理，需要等待，处理效率低。

与线程池注重并发不同，handlerThread是一个串行队列，只有一个线程，不适合网络io操作

下面是其典型用法，将handler传入子线程looper，那么这个handleMessage中执行的任务就在子线程中执行，降低主线程压力

```

myHandlerThread = new HandlerThread( "handler-thread") ;

//开启一个线程

myHandlerThread.start();

//在这个线程中创建一个handler对象

handler = new Handler( myHandlerThread.getLooper() ){

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

//这个方法是运行在 handler-thread 线程中的 ，可以执行耗时操作

Log.d( "handler " , "消息： " + msg.what + " 线程： " + Thread.currentThread().getName() ) ;

}

};

```

# IntentService

http://blog.csdn.net/lmj623565791/article/details/47143563

继承自Service，比Service优先级高，可以做耗时操作，不需要手动停止，每次只执行一个工作线程

封装了HandlerThread，其实就是一个封装了多线程的Service，不需要再另外开线程

# ListView RecycleBin机制（来源于其父类AbsListView）

# activity四种状态

running/paused/stopped/killed

栈顶 部分被覆盖，仍可见，上面有一个非全屏或者透明的activity 完全覆盖，不可见 被回收

# activity生命周期

onCreate（）第一个调用方法，setContentView，加载之类</br>

onStart（）处于用户可见，用户不能交互</br>

onResume（）可见，可交互</br>

点home返回时：onPause（） onStop（）</br>

onPause（） 可见，不可触摸</br>

onStop（）已经不可见，</br>

当再次回到原Activity时，onRestart（） onStart（） onResume（）

onRestart（） 重新返回时回调</br>

onStart（） 可见</br>

onResume（）用户可以交互</br>

退出Activity时 onPause（） onStop（） onDestroy（）</br>

onDestroy（）被销毁，可以释放资源

# android进程优先级

前台 正与用户交互的Activity或者前台Activity绑定的service

可见 可见但不能点击

服务 在后台

后台 按home键后进程变为后台

空 优先级最低，没有活跃的组件，出于缓存的目的保留，可以随时被kill

# android任务栈、启动模式

standard 每启动一个Activity都会创建一个Activity实例，不会复用已有的Activity，

消耗资源

singleTop 栈顶复用模式，如果是栈顶就不重建，否则还是会创建

singleTask 只要存在，不管在哪，都不重新创建，把它上面都要清除出去

singleinstance activity只有一个且独享整个栈

# scheme跳转协议

app之间跳转、h5页面调到app、后台链接打开app

# Fragment

1.静态加载，写入xml

2.动态加载，创建FragmentManager、Transanction，commit。。。

# FragmentPagerAdapter与FragmentStatePagerAdapter区别

前者页面较少的情况，后面页面较多的情况，后者切换时回收内存，前者只是把ui分离

# Fragment声明周期（图）

# Fragment通信

Fragment中调用Activity getActivity（）

Activity中调用Fragment 接口回调

Fragment调用Fragment findFragmentById（）

# FragmentManager的add、remove、replace方法

replace替换、add添加

# Service

Service是一个可以在后台执行长时间运行操作而没有用户界面的应用组件，可以与Activity交互，即使创建的Activity被

销毁也可以依然存在，但执行在主线程，不能做耗时操作，只是执行在后台。Activity销毁之后无法获取无法获取子线程实例

而有了service，Activity被销毁也没有任何关系，因为service会依然存在，另一个Activity与其绑定仍能获得binder实例进行操作

Service创建之后，一旦bind了Activity，只有所有的关联的Activity全部解除关系才会被关闭（即使unbind也是）

也就是说service只有在和所有Activity没有bind并且处于停止状态时才会被销毁

Service比如播放音乐、天气等

可以将Service设置成remote，会运行在一个新线程，但一般不使用，在于Activity bind时会因为不在一个线程而导致崩溃，必须使用AIDL进行ipc

# 启动service方法

1.startService（Intent intent），Activity销毁也不受影响</br>

继承Service实现的方法</br>

onBind（）、onCreate（）创建服务时调用、onStartCommand（）startService时调用、onDestroy（）

onStartCommand（）返回int值的含义</br>

2.bindService，允许与Activity进行交互，多个Activity可绑定一个service

创建服务端，继承Service类，onBind回调返回实例，在onServiceConnected（）回调

中接受binder

# 广播

http://blog.csdn.net/fkq\_2016/article/details/57086465

发送的是Intent，在一个app中不同进程中进行交互，或者在不同app间通信

Normal broadcast

system broadcast

local broadcast 使用localBroadcastManager 中的send函数发送和register函数注册 ，通过Handler发送消息实现的，所以更高效，更安全

静态注册：写在Manifest里面，进程被杀死也能收到广播，

动态注册：用registerReceiver()注册，和Activity周期一样，记得在destroy里面销毁，否则会引起内存泄漏，

广播通过action识别，发送广播时注明action，intent-filter（Manifest中）中有该action的广播会调用receive函数

# WebView

webview添加在LinearLayout等中，onDestroy中必须先从里面remove，再销毁

jsbridge（混生）：JSBridge是一座用JavaScript搭建起来的桥，一端是web，一端是native。我们搭建这座桥的目的也很简单，让native可以调用web的js代码，让web可以 “调用” 原生的代码。

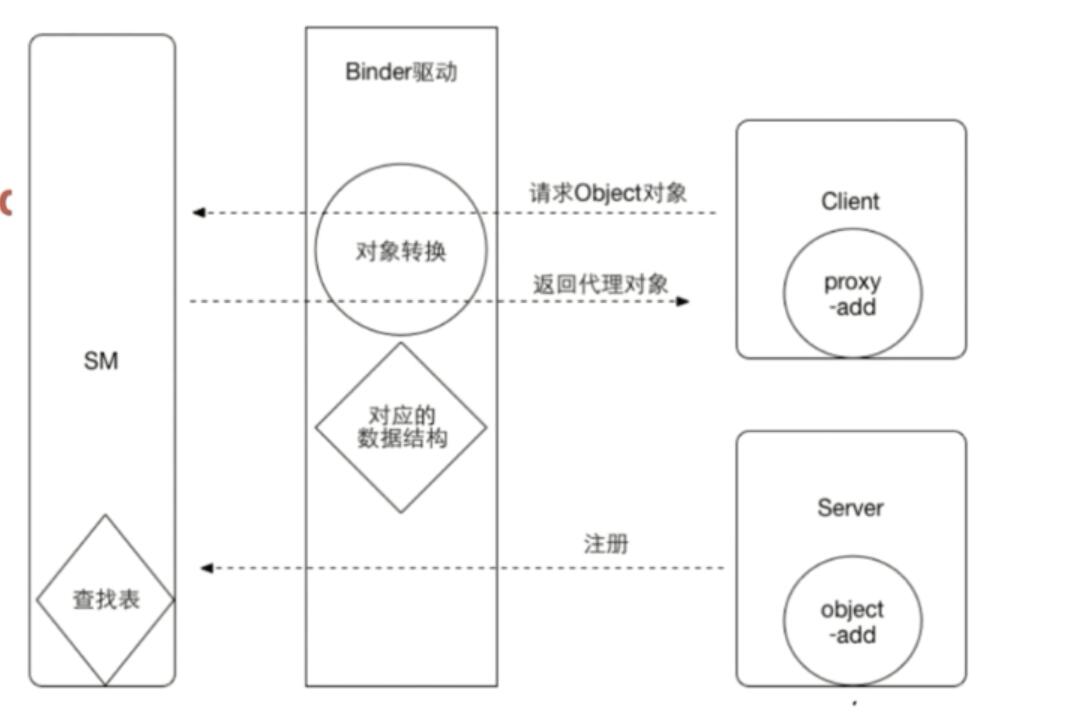
webviewClient.onPageFinished有坑，建议用WebChromeClient.onProgressChanged

WebView内部有Activity引用，开了新线程，导致泄漏，可以动态添加webview，传入Activity弱引用

或者独立进程，用完直接干掉

# Binder

binder是一种通信机制（跨进程），对于server进程binder对象指的是binder本地对象，对于client来说，binder指的是binder代理对象



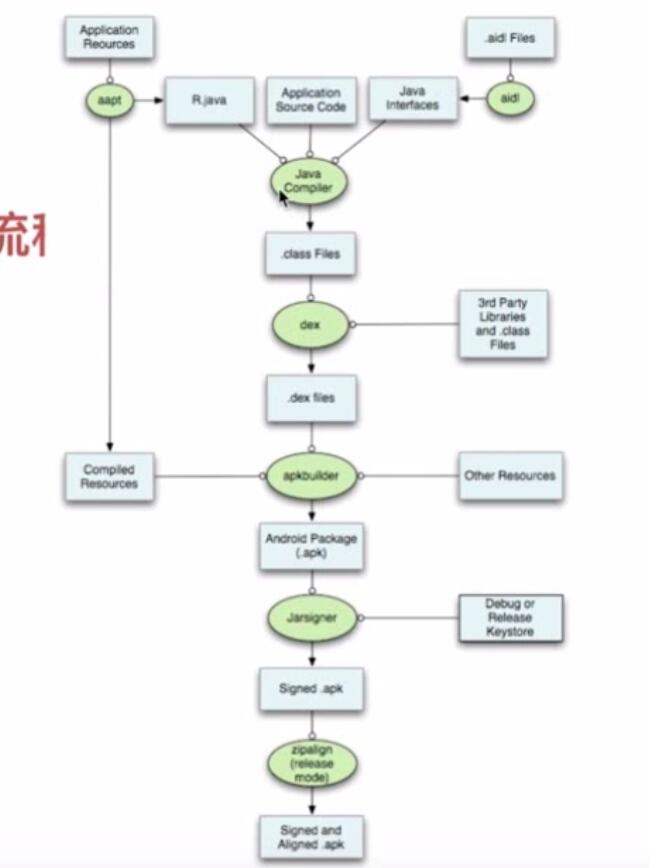
# AIDL

AIDL是Binder的一种实现，关键点同上图，在服务端会调用onTransact（），在客户端会调用return new com.zhy.calc.aidl.ICalcAIDL.Stub.Proxy(obj); 里面又调用了transact（）方法

android studio 每个project包含若干module，每个module就是文件夹带手机符号，能够跑起来的

一个project里有多个module也正是android组件开发

android编译过程



apk zipalign 优化

Jenkins 持续集成构建

git fork复制到自己的远程操作

setting.gradle ：一个project包含一个，里面是project中所有的module模块，include ':app',':new'

多模块开发一定要这个文件

project下build.gradle，说明gradle代码库版本

app内部build.gradle 可覆盖上面的gradle文件

ProGuard 混淆，核心类EntryPoint

# OKHttp

先创建OKHttpClient和Request，然后client.newCall(request)

如果同步则执行execute()返回 Response

异步则执行enqueue（callback）

Request request=new Request.Builder().url(...).build();

这是典型的build模式，可以将复杂的对象的创建与表示相分离

核心：Interceptor拦截器

# Retrofit

封装了OKHttp，动态代理模式

# Volley

打开两个线程，分别是缓存和网络

# ButterKnife

通过java注解机制，性能比反射机制好，不能将变量设为private或static，会使其变为反射

扫描注解-》生成ViewBinder类-》调用bind方法加载ViewBinder类

# ANR

application not responding，主线程做了耗时操作，响应是有Activity Manager和WindowManager系统服务监视的

# OOM

out of Memory(Dalvik虚拟机)当前占用内存加上申请的内存大于dalvik虚拟机分配的

内存抖动：短时间产生大量对象，短时间又回收（GC）

内存泄漏：已经没有用的对象引用GC roots

# Bitmap

recycle()

LRUCache

insampleSize

缩略图

三级缓存

用gone不用invisible

尽量用weight

item嵌套复杂，尽量自定义view

# 内存

后台进程、服务进程区别

避免使用枚举变量，消耗内存多

尽量少使用ioc框架

内存泄漏分析工具 LeakCanary

# 冷启动、热启动

启动前系统没有该应用的任何进程信息，会走Application类，冷启动计算时间：从应用启动开始计算，到完成视图的第一次绘制

zygote进程fork一个新进程

优化：不要让Application参与业务的操作以及耗时操作，不要在Application中以静态变量的方式保存数据

# 其他优化

不要用静态变量保存数据，不安全，因为有时候进程会被杀死（看起来应用却并没有重新启动）

Sharepreference不能跨进程同步

序列化：将对象状态信息转化为可以存储或传输的形式的过程

Serializable产生大量临时变量

Parceble使用内存时性能好，java特有，不能使用在将数据存储在磁盘上的情况

ui卡顿工具strictmode严格模式

# 插件化

类加载机制

DexclassLoader 加载未安装的jar/apk/dex

PathClassLoader 加载系统中已经安装过的apk

# 热更新

Native：

Dexposed hook函数、动态加载技术、反射

AndFix 性能更好，生成补丁包

Java：

Nuwa classLoader，dex包

原理：ClassLoader扫描dexElements，只要新的类放在前面，就不会加载后面那个错的

# 很懂