Activity:

- [ ] Activity是什么?

Android 四大基本组件, 是一种可视化的, 可以做各种点击或滑动操作的View.

- [ ] Avtivity 四种状态:

Running: 位于前台, 可见, 有焦点. (onResume)

Paused: 位于前台, 可见, 无焦点 (onPause)

stopped: 位于后台, 不可见, 无焦点(onStop)

killed: 结束, 不可见, 无焦点(onDestroy)

- [ ] Activity 生命周期

1. onCreate()
2. onStart()
3. onResume()
4. onPause()
5. onStop()
6. onDestroy()
7. onRestart() ->(onStop -> onStart)

- [ ] Activity任务栈

- 先进后出

- [ ] Activity启动模式

- 标准模式(standard), 默认加载模式. 总是创建新的Activity实例
- 栈顶模式(singletop), 如何Activity实例不在栈顶, 则创建新实例, 否则直接复用栈顶的Activity实例.
- 单例模式(singletask), 同一个task只能有一个Activity实例. 非栈顶则移除栈顶Activity后在置顶.
- 全局单例模式(singleinstance), 在一个Task中仅有一个Activity实例.该实例总是位于task栈顶

- [ ] sheme跳转协议

Android中的scheme是一种页面内跳转协议, 通过定义自己的sheme协议, 可以跳转到app的各个Activity页面.

- 服务器可以定制化告诉app跳转哪个页面
- app可以跳转到另外app的页面
- 可以通过H5页面跳转.

- [ ] Context, Activity, Application之间的关系

Activity与Application都是Context的子类,.

- Context是上下文的意思, 在实际应用中 可以起到管理上下文环境汇总各个参数和变量的作用, 方便访问各种资源,
- Activity虽然也是上下文管理, 但是其生命周期仅仅维护一个Activity.
- Application 虽然也是上下文管理, 但维护一个app的生命周期.

Fragement

Fragment 比 Activity更加节省内存, 且切换模式也更加简单, 并且其有自己的生命周期, 且必须依附于Activity.

Activity创建Fragment的方式

- 静态创建

```
1. 布局文件left_layout.xml

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:background="#6f6669">
    <Button
        android:id="@+id/panhouye"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="潘侯爷"/>
    <Button
        android:id="@+id/bikonghai"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="碧空海"/>
</LinearLayout>

2. 对应的继承Fragment的java类

public class LeftFragment extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        //通过参数中的布局填充获取对应布局
        View view =inflater.inflate(R.layout.left_layout,container,false);
```

```
        return view;
    }
    @Override
    public void onPause() {
        super.onPause();
    }
}

3. main Activity 加载

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    tools:context="com.example.administrator.fragmenttest.MainActivity">
    <fragment
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:name="com.example.administrator.fragmenttest.LeftFragment"

        tools:layout="@layout/left_layout" />
    <fragment
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3"

 android:name="com.example.administrator.fragmenttest.RightFragment"
        tools:layout="@layout/right_layout" />
</LinearLayout>
```

- 动态加载

```
1. fragmeLayout布局

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    tools:context="com.example.administrator.testfragment.MainActivity">
    <FrameLayout
```

```
        android:id="@+id/right"  <!--用于加载Fragment-->
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3"></FrameLayout>
</LinearLayout>

2. frament的自定义View
public class RightFragment extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        //通过参数中的布局填充获取对应布局
        View view
=inflater.inflate(R.layout.right_layout,container,false);
        return view;
    }
    @Override
    public void onPause() {
        super.onPause();
    }
}

3. mainjava 加载
public class Main2Activity extends AppCompatActivity {
    //声明本次使用到的java类
    FragmentManager fragmentManager;
    FragmentTransaction fragmentTransaction;
    RightFragment rightFragment;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
        fragmentManager=getFragmentManager();
        fragmentTransaction = fragmentManager.beginTransaction();
        rightFragment = new RightFragment();
        fragmentTransaction.add(R.id.right,rightFragment);
        //事务处理完需要提交
        fragmentTransaction.commit();
    }
}
```

- [ ] FragmentPageAdapter 和 FragementStateAdapter 的区别

- FragmentPageAdapter 在每次切换页面的时候, 将fragement进行分离, 适合页面较少的fragment使用, 以保存一些内存, 对内存的影响较小
- FragementStateAdapter 在每次切换页面时, 将fragment进行会后, 适合页面较多的fragment使用, 这样就不会消耗更多的内存.

- [ ] Fragment生面周期

- onAttach()
- onCreate() : 创建Fragment时调用.
- onCreateView() : 绘制组件时, 用于调用.
- onActivityCreated()
- onStart()
- onResume()
- onPause() : 离开Fragement时调用.
- onStop()
- onDestroyView()
- onDestroy()
- onDetach()

- [ ] Fragment的通信

- Fragment调用Activity中的方法: getActivity.
- Activity调用Fragment 中的方法, 接口调用
- Fragment 调用Fragment的方法: FragmentManager.findFragmentById.
- http://blog.csdn.net/u012702547/article/details/49786417

- [ ] Fragment的replace、add、remove方法

- replace：替代Fragment的栈顶页面
- add：添加Fragment到栈顶页面
- remove：移除Fragment栈顶页面

2018年2月26日 13:40:46

Service:

- [ ] Service是什么

Service是四大组件之一，可以在后台执行长时间运行操作, 且没有用户界面的应用组件.

- [ ] Service和Thread的区别

- Service是安卓中系统的组件，它运行在独立进程的主线程中，不可以执行耗时操作。
- Thread是程序执行的最小单元，分配CPU的基本单位，可以开启子线程执行耗时操作.
- Service在不同Activity中可以获取自身实例，可以方便的对Service进行操作。
- Thread在不同的Activity中难以获取自身实例，如果Activity被销毁，Thread实例就很难再获取得到.

- [ ] Service启动方式

- startService: Activity与Service的消亡没有必然关系.

```
1. 显示启动Service.
        bt_start = findViewById(R.id.bt_start);
        final Intent intent1 = new Intent(this,StartServiceDemo.class);
        bt_start.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startService(intent1);
            }
        });

2. 集成Service的java类
public class StartServiceDemo extends Service {

    @Override
    public void onCreate() {
        super.onCreate();
        MyLog.d("startService onCreate");
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        MyLog.d("startService onBind");
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        MyLog.d("startService onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        MyLog.d("startService onDestroy");
        super.onDestroy();
```

```
    }
}
```

- bindService(), Activity消亡, 则Service消亡. 绑定关系.

```
1. 绑定Service
        bt_bind = findViewById(R.id.bt_bind);
        final Intent intent2= new Intent(this,BindServiceDemo.class);
        bt_bind.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                bindService(intent2,conn, Service.BIND_AUTO_CREATE);

                //Toast.makeText(getApplicationContext(),"count:
"+binder.getCount(),Toast.LENGTH_LONG);
            }
        });

2. 集成Service的java类
public class BindServiceDemo extends Service {

    private int count;
    private boolean quit;
    private MyBinder binder = new MyBinder();

    @Override
    public void onCreate() {
        super.onCreate();
        MyLog.d("bindService onCreate");

        new Thread(){
            @Override
            public void run() {
                while(!quit){
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    count++;
                }
            }
        }.start();
    }

```

```
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        MyLog.d("BindService onBind");
        return binder; //该binder可以被Activity接收,
    }

    @Override
    public boolean onUnbind(Intent intent) {
        MyLog.d("bindService onUnbind");
        return true;
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        this.quit=true;
        MyLog.d("bindService onDestroy");
    }

    public class MyBinder extends Binder {
        public int getCount(){
            return count;
        }
    }
}

3. Activity与Service通信
       private ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder
service) {
            MyLog.d("bindService conn");
            binder = (BindServiceDemo.MyBinder) service; //该出的service是
Service的onBind()方法返回的binder对象的映射，此时Activity就获操作Service 的
Binder方法.
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };
```

- [ ] Service生命周期

1. startService:

- onCreate()
- onStartCommand()
- onDestroy()

2. bindService

- onCreate()
- onBind()
- onUnbind()
- onDestroy()

BroadCast Receiver:

- [ ] BroadcastReceiver是什么

Broadcast是四大组件之一，是一种广泛运用在应用程序之间传输信息的机制，通过发送Intent来传送我们的数据

- [ ] 使用场景

- 同一App具有多个进程的不同组件之间的消息通信
- 不同App之间的组件之间的消息通信

- [ ] 广播种类

- 普通广播: 同一时刻, 所有用户都可以收到消息, 但不能将消息往下传.
- 有序广播: 按优先级顺序开始传输数据.
- 本地广播: Android中的广播可以跨App直接通信

- [ ] 实现方法

- 静态注册: 注册后一直运行，尽管Activity、进程、App被杀死还是可以接收到广播.

```
1. AndroidManife.xml

        <receiver android:name="java类">
            <intent-filter>
                <action
android:name="android.provider.Telephony.SMS_RECEIVER"/>
            </intent-filter>
        </receiver>
```

- 动态注册: 跟随Activity的生命周期

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(String); //为BroadcastReceiver指定action，即要监听的消息名字。
registerReceiver(MyBroadcastReceiver,intentFilter); //注册监听
unregisterReceiver(MyBroadcastReceiver); //取消监听
```

- [ ] Broadcast Receiver 实现机制

1. 自定义广播类继承BroadcastReceiver，复写onReceiver()
2. 通过Binder机制向AMS进行注册广播
3. 广播发送者通过Binder机制向AMS发送广播
4. AMS查找符合相应条件的广播发送到BroadcastReceiver相应的循环队列中
5. 消息队列执行拿到广播，回调BroadcastReceiver的onReceiver()

- [ ] LocalBroadcastManager特点

- 本地广播只能在自身App内传播，不必担心泄漏隐私数据
- 本地广播不允许其他App对你的App发送该广播，不必担心安全漏洞被利用
- 本地广播比全局广播更高效
- 以上三点都是源于其内部是用Handler实现的

WebView:

- [ ] WebView安全漏洞

- API16之前存在远程代码执行安全漏洞，该漏洞源于程序没有正确限制使用
- WebView.addJavascriptInterface方法，远程攻击者可通过使用Java反射机制利用该漏洞执行任意Java对象的方法

- [ ] WebView销毁步骤

1. 先移除容器上的WebView
2. 将WebView.destroy()

- [ ] WebView的jsbridge

客户端和服务端之间可以通过Javascript来互相调用各自的方法

- [ ] WebViewClient的onPageFinished

WebViewClient的onPageFinished在每次完成页面的时候调用，但是遇到未加载完成的页面跳转其他页面时，就会一直调用，使用WebChromeClient.onProgressChanged可以替代

- [ ] WebView后台耗电

在WebView加载页面的时候，会自动开启线程去加载，如果不很好的关闭这些线程，就会导致电量消耗加大，可以采用暴力的方法，直接在onDestroy方法中System.exit(0)结束当前正在运行中的java虚拟机

- [ ] WebView硬件加速

Android3.0引入硬件加速，默认会开启，WebView在硬件加速的情况下滑动更加平滑，性能更加好，但是会出现白块或者页面闪烁的副作用，建议WebView暂时关闭硬件加速

- [ ] WebView内存泄漏

由于WebView是依附于Activity的，Activity的生命周期和WebView启动的线程的生命周期是不一致的，这会导致WebView一直持有对这个Activity的引用而无法释放，解决方案如下

- 独立进程，简单暴力，不过可能涉及到进程间通信（推荐）
- 动态添加WebView，对传入WebView中使用的Context使用弱引用

Binder:

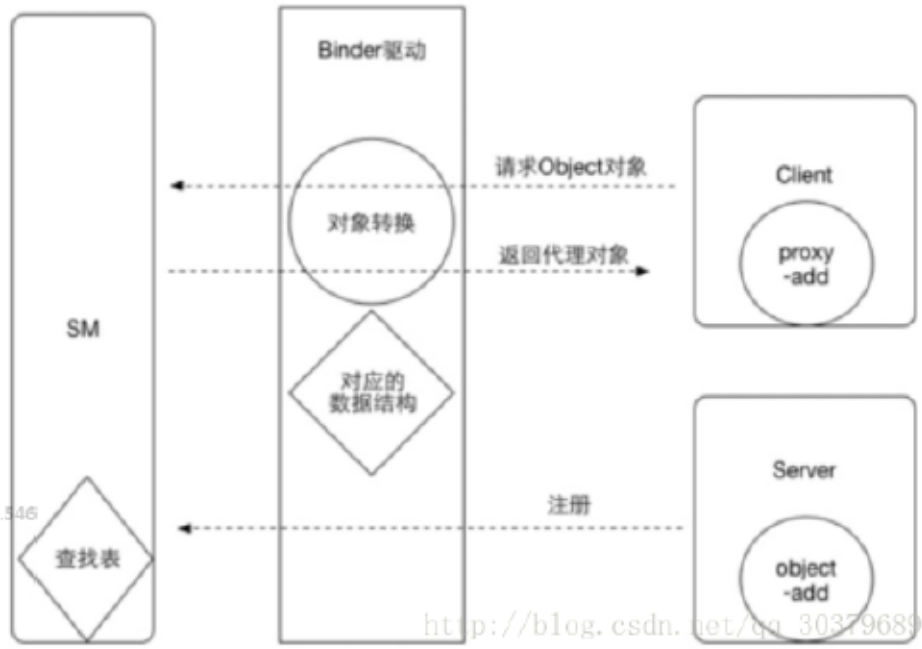
- [ ] Linux内核的基本知识

- 进程隔离/虚拟地址空间：进程间是不可以共享数据的，相当于被隔离，每个进程被分配到不同的虚拟地址中
- 系统调用：Linux内核对应用有访问权限，用户只能在应用层通过系统调用，调用内核的某些程序
- binder驱动：它负责各个用户的进程，通过binder通信内核来进行交互的模块

- [ ] 原因

- 性能上，相比传统的Socket更加高效
- 安全性高，支持协议双方互相校验

- [ ] 通信模型



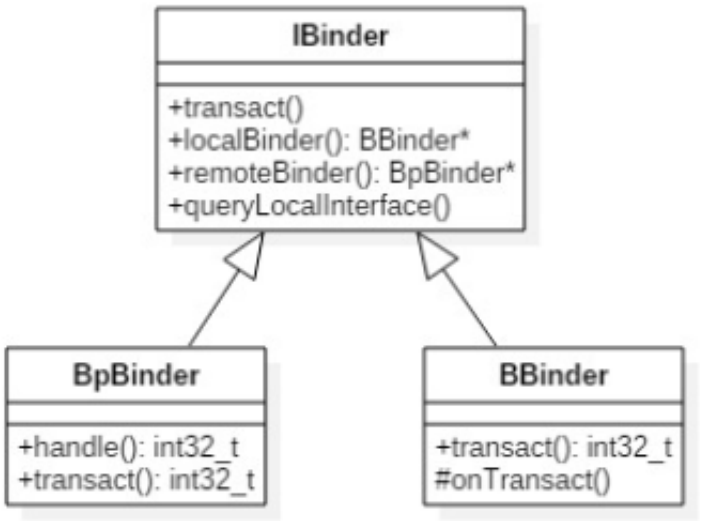
- Service服务端通过Binder驱动在ServiceManager的查找表中注册Object对象的add方法
- Client客户端通过Binder驱动在ServiceManager的查找表中找到Object对象的add方法，并返回proxy的add方法，add方法是个空实现，proxy也不是真正的Object对象， 是通过Binder驱动封装好的代理类的add方法
- 当Client客户端调用add方法时，Client客户端通过Binder驱动将proxy的add方法，请求ServiceManager来找到Service服务端真正对象的add方法，进行调用.

- [ ] AIDL

- 客户端通过aidl文件的Stub.asInterface()方法，拿到Proxy代理类
- 通过调用Proxy代理类的方法，将参数进行封包后，调用底层的transact()方法
- transact()方法会回调onTransact()方法，进行参数的解封
- 在onTransact()方法中调用服务端对应的方法，并将结果返回

- [ ] BpBinder 和BBinder

BpBinder(客户端)对象和BBinder(服务端)对象，它们都从IBinder类中派生而来，
BpBinder(客户端)对象是BBinder(服务端)对象的代理对象.



- client端：BpBinder.transact()来发送事务请求
- server端：BBinder.onTransact()会接收到相应事务

Handler:

- [ ] Handler是什么

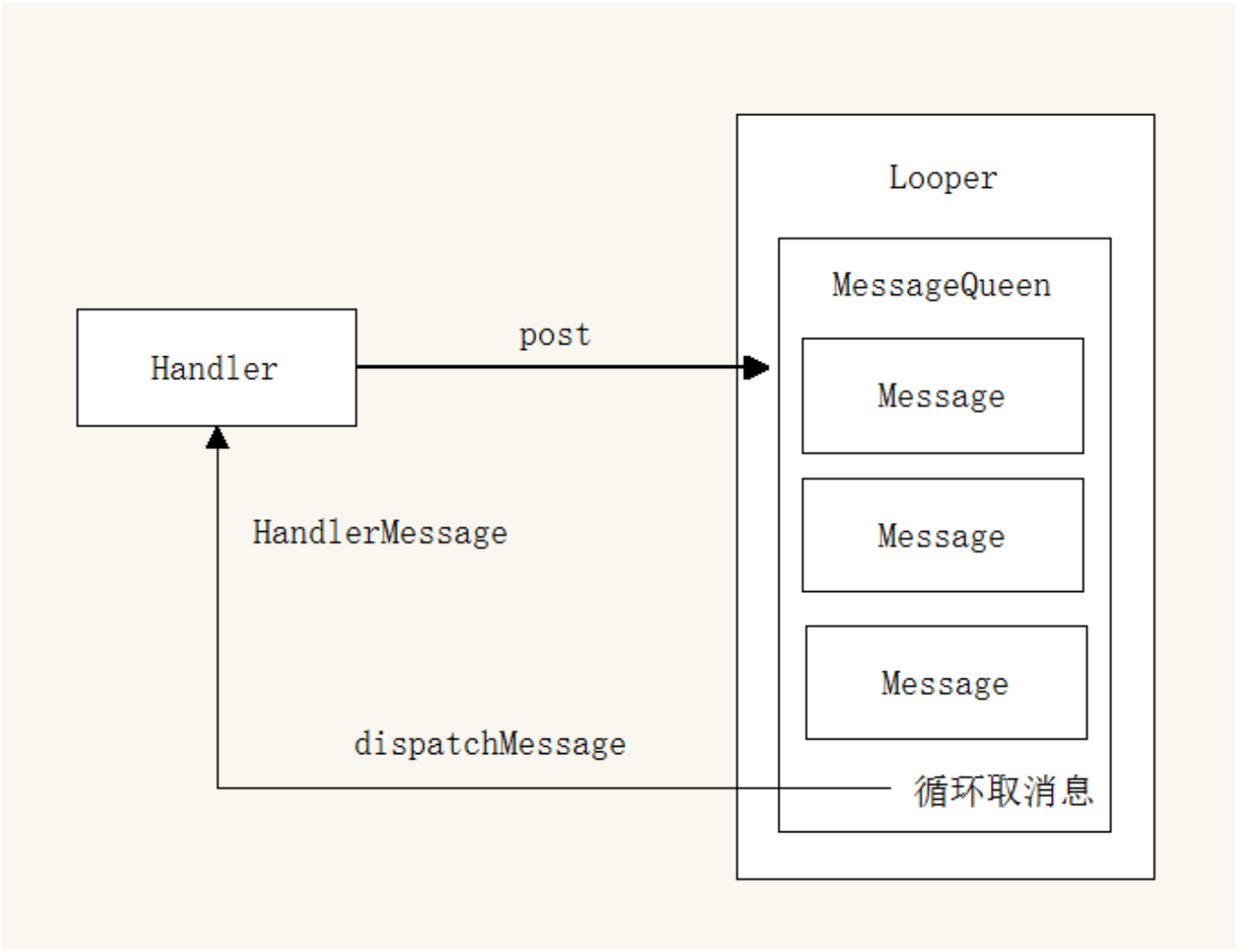
Handler通过发送和处理Message和Runnable对象来关联相对应线程的MessageQueue

- [ ] Handler使用方法

- post(runnable)
- sendMessage(message)

- [ ] Handler工作原理

Android进阶——Android消息机制之Looper、Handler、MessageQueen



请解释下Android通信机制中Message、Handler、MessageQueen、Looper的之间的关系?

首先，是这个MessageQueen，MessageQueen是一个消息队列，它可以存储Handler发送过来的消息，其内部提供了进队和出队的方法来管理这个消息队列，其出队和进队的原理是采用单链表的数据结构进行插入和删除的，即enqueueMessage()方法和next()方法。这里提到的Message，其实就是一个Bean对象，里面的属性用来记录Message的各种信息。

然后，是这个Looper，Looper是一个循环器，它可以循环的取出MessageQueen中的Message，其内部提供了Looper的初始化和循环出去Message的方法，即prepare()方法和loop()方法。在prepare()方法中，Looper会关联一个MessageQueen，而且将Looper存进一个ThreadLocal中，在loop()方法中，通过ThreadLocal取出Looper，使用MessageQueen的next()方法取出Message后，判断Message是否为空，如果是则Looper阻塞，如果不是，则通过dispatchMessage()方法分发该Message到Handler中，而Handler执行handlerMessage()方法，由于handlerMessage()方法是个空方法，这也是为什么需要在Handler中重写handlerMessage()方法的原因。这里要注意的是Looper只能在一个线程中只能存在一个。这里提到的ThreadLocal，其实就是一个对象，用来在不同线程中存放对应线程的Looper。

最后，是这个Handler，Handler是Looper和MessageQueen的桥梁，Handler内部提供了发送Message的一系列方法，最终会通过MessageQueen的enqueueMessage()方法将Message存进MessageQueen中。我们平时可以直接在主线程中使用Handler，那是因为在应用程序启动时，在入口的main方法中已经默认为我们创建好

了Looper。

MessageQueen(将Message对象列队) -> loop(循环MessageQueen, 取出执行) -> Handler(发送Message给MessageQueen对象).

- [ ] Handler引起的内存泄漏

原因：非静态内部类持有外部类的匿名引用，导致Activity无法释放 解决：

- Handler内部持有外部Activity的弱引用
- Handler改为静态内部类
- Handler.removeCallback()

ASyncTask:

- [ ] AsyncTask是什么

本质上就是一个封装了线程池和Handler的异步框架

- [ ] AsyncTask使用方法

1. 三个参数
   - Params：表示后台任务执行时的参数类型，该参数会传给AysncTask的doInBackground()方法
   - Progress：表示后台任务的执行进度的参数类型，该参数会作为onProgressUpdate()方法的参数
   - Result：表示后台任务的返回结果的参数类型，该参数会作为onPostExecute()方法的参数
2. 五个方法
   - onPreExecute()：异步任务开启之前回调，在主线程中执行
   - doInBackground()：执行异步任务，在线程池中执行
   - onProgressUpdate()：当doInBackground中调用publishProgress时回调，在主线程中执行
   - onPostExecute()：在异步任务执行之后回调，在主线程中执行
   - onCancelled()：在异步任务被取消时回调

- [ ] AsyncTask工作原理

[Android进阶——多线程系列之异步任务AsyncTask的使用与源码分析]

- [ ] AsyncTask引起的内存泄漏

原因：非静态内部类持有外部类的匿名引用，导致Activity无法释放

解决：

- AsyncTask内部持有外部Activity的弱引用
- AsyncTask改为静态内部类

- AsyncTask.cancel()

- [ ] AsyncTask生命周期

在Activity销毁之前，取消AsyncTask的运行，以此来保证程序的稳定

- [ ] AsyncTask结果丢失

由于屏幕旋转、Activity在内存紧张时被回收等情况下，Activity会被重新创建，此时，旧的 AsyncTask持有旧的Activity引用，这个时候会导致AsyncTask的onPostExecute()对UI更新无效

- [ ] AsyncTask并行or串行

AsyncTask在Android 2.3之前默认采用并行执行任务，AsyncTask在Android 2.3之后 默认采用串行执行任务
如果需要在Android 2.3之后采用并行执行任务，可以调用AsyncTask的 executeOnExecutor()

HandlerThread:

- [ ] HandlerThread产生背景

当系统有多个耗时任务需要执行时，每个任务都会开启一个新线程去执行耗时任务，这样会导 致系统多次创建和销毁线程，从而影响性能。为了解决这一问题，Google提供了HandlerThread，HandlerThread是在线程中创建一个Looper循环器，让Looper轮询消息队列，当有耗时任务进入队列时，则不需要开启新线程，在原有的线程中执行耗时任务即可，否则线程阻塞.

- [ ] HanlderThread的特点

- HandlerThread本质上是一个线程，继承自Thread
- HandlerThread有自己的Looper对象，可以进行Looper循环，可以创建Handler
- HandlerThread可以在Handler的handlerMessage中执行异步方法
- HandlerThread优点是异步不会堵塞，减少对性能的消耗
- HandlerThread缺点是不能同时继续进行多任务处理，需要等待进行处理，处理效率较低
- HandlerThread与线程池不同，HandlerThread是一个串行队列，背后只有一个线程

- [ ] IntentService

IntentService是继承自Service并处理异步请求的一个类，其内部采用HandlerThread和Handler实现的，在IntentService内有一个工作线程来处理耗时操作，
其优先级比普通Service高。当任务完成后，IntentService会自动停止，而不需要手动调用stopSelf()。
另外，可以多次启动IntentService，每个耗时操作都会以工作队列的方式在IntentService中onHandlerIntent()回调方法中执行，并且每次只会执行一个工作线程

- [ ] IntentService使用方法

1. 创建Service继承自IntentService

2. 覆写构造方法和onHandlerIntent()方法

3. 在onHandlerIntent()中执行耗时操作

视图工作机制

Android进阶——Android视图工作机制之measure、layout、draw

Android视图工作机制按顺序分为以下三步：

- measure：确定View的宽高
- layout：确定View的位置
- draw：绘制出View的形状

子View的MeasureSpec由父容器的MeasureSpec和自身的LayoutParams来共同决定的

1. invalidate()和requestLayout()

- invalidate方法只会执行onDraw方法
- requestLayout方法只会执行onMeasure方法和onLayout方法，并不会执行onDraw方法。

事件分发机制

Android事件分发机制之dispatchTouchEvent、onInterceptTouchEvent、onTouchEvent

- dispatchTouchEvent：用于进行点击事件的分发
- onInterceptTouchEvent：用于进行点击事件的拦截
- onTouchEvent：用于处理点击事件

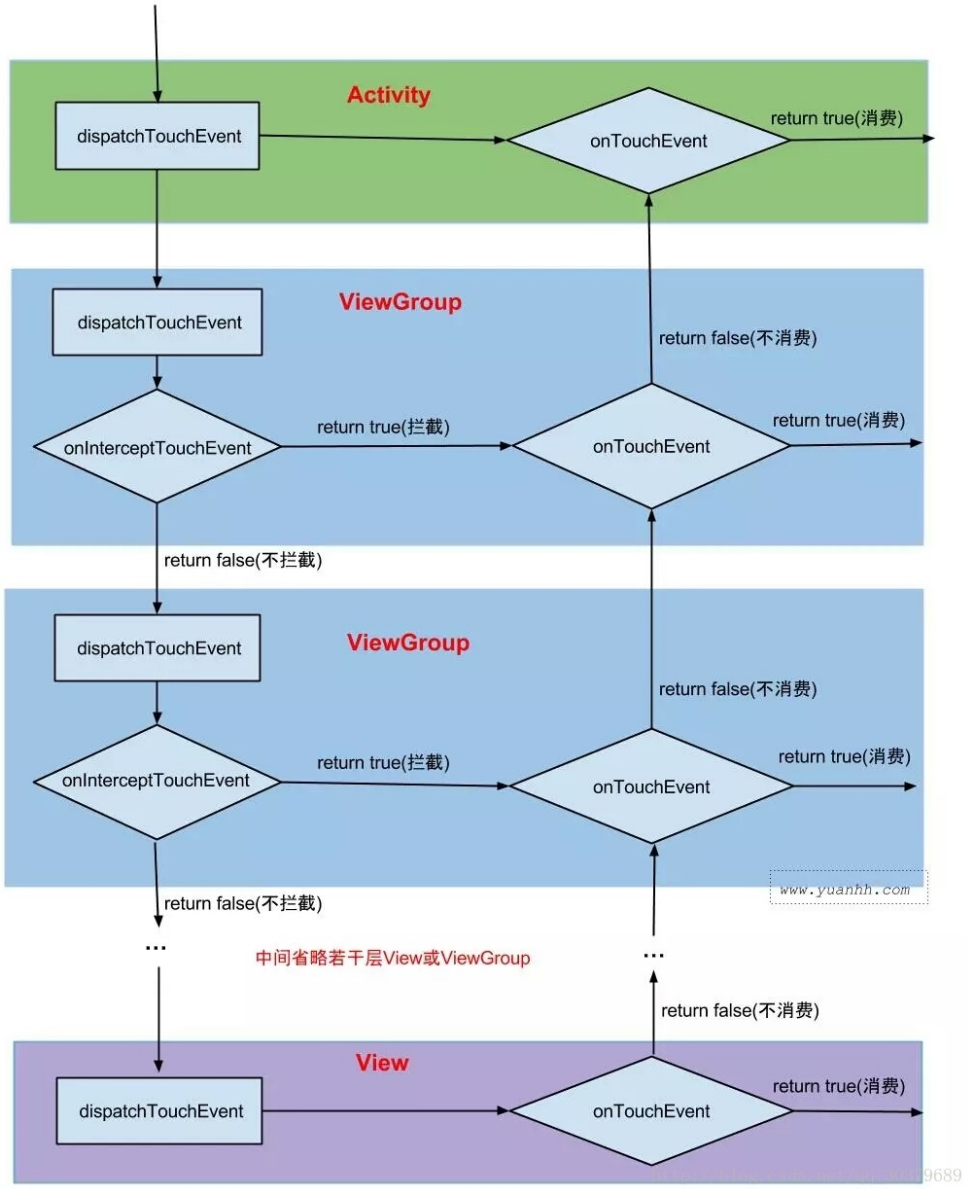
总结:

- dispatchTouchEvent
    - return true：表示该View内部消化掉了所有事件
    - return false：表示事件在本层不再继续进行分发，并交由上层控件的onTouchEvent方法进行消费
    - return super.dispatchTouchEvent(ev)：默认事件将分发给本层的事件拦截onInterceptTouchEvent方法进行处理

- onInterceptTouchEvent
    - return true：表示将事件进行拦截，并将拦截到的事件交由本层控件的onTouchEvent进行处理
    - return false：表示不对事件进行拦截，事件得以成功分发到子View
    - return super.onInterceptTouchEvent(ev)：默认表示不拦截该事件，并将事件传递给下一层View的dispatchTouchEvent

- onTouchEvent
- return true：表示onTouchEvent处理完事件后消费了此次事件

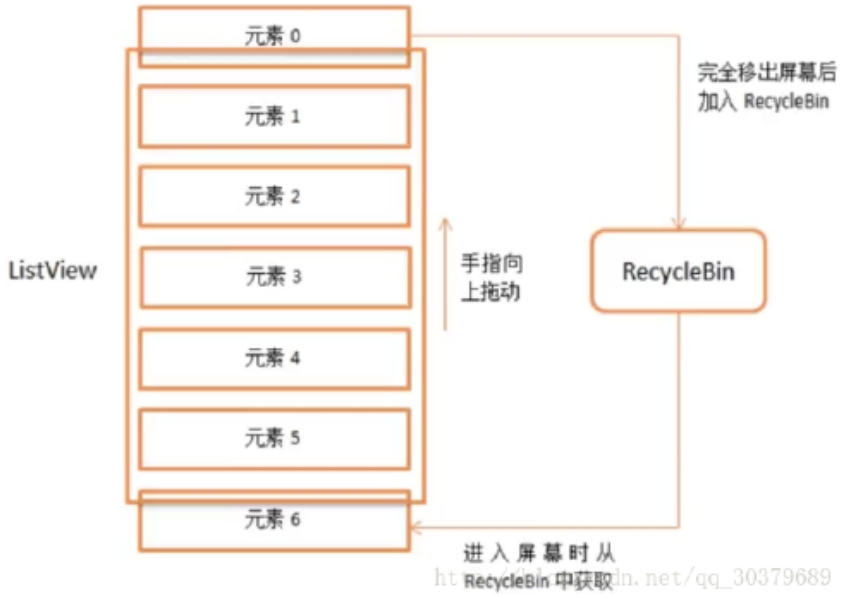
- return fasle：表示不响应事件，那么该事件将会不断向上层View的onTouchEvent方法传递，直到某个View的onTouchEvent方法返回true
- return super.dispatchTouchEvent(ev)：表示不响应事件，结果与return false一样



- [ ] ListView

ListView是能将一个数据集合以动态滚动的方式展示到用户界面上的View

- [ ] ListView的RecycleBin机制



- [ ] ListView的优化

- 重用convertView
- 使用ViewHolder
- 图片三级缓存
- 监听滑动事件
- 少用透明View
- 开启硬件加速

jenkins持续集成构建

http://www.pgyer.com/doc/view/jenkins

proguard

ProGuard工具是用于压缩、优化和混淆我们的代码，其主作用是移除或混淆代码中无用类、字段、方法和属性

- [ ] proguard技术功能

- 压缩
- 优化
- 混淆
- 预检测

- [ ] proguard工作原理

将无用的字段或方法存入到EntryPoint中，将非EntryPoint的字段和方法进行替换

- [ ] 为什么要混淆

由于Java是一门跨平台的解释性语言，其源代码被编译成class字节码来适应其他平台，而class文件包含了Java源代码信息，很容易被反编译

ANR

Application Not Responding，页面无响应的对话框

- [ ] 发生ANR的条件

应用程序的响应性是由ActivityManager和WindowManager系统服务监视的，当ANR发生条 件满足时，就会弹出ANR的对话框

- Activity超过5秒无响应
- BroadcastReceiver超过10秒无响应
- Service超过20秒无响应

- [ ] 造成ANR的主要原因

- 主线程被IO操作阻塞
- Activity的所有生命周期回调都是执行在主线程的
- Service默认执行在主线程中
- BoardcastReceiver的回调onReceive()执行在主线程中
- AsyncTask的回调除了doInBackground，其他都是在主线程中
- 没有使用子线程Looper的Handler的handlerMessage，post(Runnable)都是执行在主线程中

- [ ] 如何解决ANR

- 使用AsyncTask处理耗时IO操作
- 使用Thread或HandlerThread提高优先级
- 使用Handler处理工作线程的耗时操作
- Activity的onCreate和onResume回调尽量避免耗时操作

OOM

OOM指Out of memory（内存溢出），当前占用内存加上我们申请的内存资源超过了Dalvik 虚拟机的最大内存限制就会抛出Out of memory异常.

- [ ] OOM相关概念

- 内存溢出：指程序在申请内存时，没有足够的空间供其使用
- 内存泄漏：指程序分配出去的内存不再使用，无法进行回收
- 内存抖动：指程序短时间内大量创建对象，然后回收的现象

- [ ] 解决OOM

1. Bitmap相关

- 图片压缩
- 加载缩略图
- 在滚动时不加载图片
- 回收Bitmap
- 使用inBitmap属性
- 捕获异常

2. 其他相关

- listview重用convertView、使用lru
- 避免onDraw方法执行对象的创建
- 谨慎使用多进程

Bitmap

- [ ] recycle

1. 在安卓3.0以前Bitmap是存放在堆中的，我们只要回收堆内存即可
2. 在安卓3.0以后Bitmap是存放在内存中的，我们需要回收native层和Java层的内存
3. 官方建议我们3.0以后使用recycle方法进行回收，该方法也可以不主动调用，因为垃圾回收器会自动收集不可用的Bitmap对象进行回收
4. recycle方法会判断Bitmap在不可用的情况下，将发送指令到垃圾回收器，让其回收native层和Java层的内存，则Bitmap进入dead状态
5. recycle方法是不可逆的，如果再次调用getPixels()等方法，则获取不到想要的结果

- [ ] LruCache原理

LruCache是个泛型类，内部采用LinkedHashMap来实现缓存机制，它提供get方法和put方法来获取缓存和添加缓存，其最重要的方法trimToSize是用来移除最少使用的缓存和使用最久的缓存，并添加最新的缓存到队列中

- [ ] 三级缓存

- 网络缓存
- 本地缓存
- 内存缓存

- [ ] 计算采样

```
public static int calculateInSampleSize(BitmapFactory.Options options, int reqWidth,
    final int height = options.outHeight;
    final int width = options.outWidth;
```

```
    int inSampleSize = 1;
    if (height > reqHeight || width > reqWidth) {
        if (width > height) {
            inSampleSize = Math.round((float)height / (float)reqHeight);
        } else {
            inSampleSize = Math.round((float)width / (float)reqWidth);
        }
    }
    return inSampleSize;
}
```

- [ ] 采样率压缩

```
public static Bitmap thumbnail(String path,int maxWidth, int maxHeight) {
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    Bitmap bitmap = BitmapFactory.decodeFile(path, options);
    options.inJustDecodeBounds = false;
    int sampleSize = calculateInSampleSize(options, maxWidth, maxHeight);
    options.inSampleSize = sampleSize;
    options.inPreferredConfig = Bitmap.Config.RGB_565;
    options.inPurgeable = true;
    options.inInputShareable = true;
    if (bitmap != null && !bitmap.isRecycled()) {
        bitmap.recycle();
    }
    bitmap = BitmapFactory.decodeFile(path, options);
    return bitmap;
}
```

- [ ] 质量压缩

```
public static String save(Bitmap bitmap,Bitmap.CompressFormat format, int
quality, File destFile) {
    try {
        FileOutputStream out = new FileOutputStream(destFile);
        if (bitmap.compress(format, quality, out)) {
            out.flush();
            out.close();
        }
        if (bitmap != null && !bitmap.isRecycled()) {
            bitmap.recycle();
```

```
        }
        return destFile.getAbsolutePath();
    } catch (Exception e) {
        e.printStackTrace();
    }
    retrun null;
}
```

- [ ] 尺寸压缩

```
public static void reSize(Bitmap bmp,File file,int ratio){
    Bitmap result = Bitmap.createBitmap(bmp.getWidth()/ratio,
bmp.getHeight()/ratio,Bitmap
    Canvas canvas = new Canvas(result);
    RectF rect = new RectF(0, 0, bmp.getWidth()/ratio, bmp.getHeight()/ratio);
    canvas.drawBitmap(bmp, null, rect , null);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    result.compress(Bitmap.CompressFormat.JPEG, 100, baos);
    try {
        FileOutputStream fos = new FileOutputStream(file);
        fos.write(baos.toByteArray());
        fos.flush();
        fos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

- [ ] 保存到SD卡

```
public static String save(Bitmap bitmap,Bitmap.CompressFormat format, int
quality, Context context) {
    if (!Environment.getExternalStorageState()
            .equals(Environment.MEDIA_MOUNTED)) {
        return null;
    }
    File dir = new File(Environment.getExternalStorageDirectory()
            + "/" + context.getPackageName() + "/save/");
    if (!dir.exists()) {
        dir.mkdirs();
    }
```

```
    File destFile = new File(dir, UUID.randomUUID().toString());
    return save(bitmap, format, quality, destFile);
}

```

- [ ] UI卡顿

- View的绘制帧数保持60fps是佳，这要求每帧的绘制时间不超过16ms（1000/60），如果 安卓不能在16ms内完成界面的渲染，那么就会出现卡顿现象

原因分析

- 在UI线程中做轻微的耗时操作，导致UI线程卡顿
- 布局Layout过于复杂，无法在16ms内完成渲染
- 同一时间动画执行的次数过多，导致CPU和GPU负载过重
- overDraw，导致像素在同一帧的时间内被绘制多次，使CPU和GPU负载过重
- View频繁的触发measure、layout，导致measure、layout累计耗时过多和整个View频 繁的重新渲染
- 频繁的触发GC操作导致线程暂停，会使得安卓系统在16ms内无法完成绘制
- 冗余资源及逻辑等导致加载和执行缓慢
- ANR

优化:

- 使用include、ViewStub、merge
- 不要出现过于嵌套和冗余的布局
- 使用自定义View取代复杂的View

ListView优化;

- 复用convertView
- 滑动不加载

背景和图片优化:

- 缩略图
- 图片压缩

- [ ] 内存泄漏

1. Java内存泄漏引起的主要原因

   长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏

1. Java内存分配策略

- 静态存储区：又称方法区，主要存储全局变量和静态变量，在整个程序运行期间都存在
- 栈区：方法体的局部变量会在栈区创建空间，并在方法执行结束后会自动释放变量的空 间和内存
- 堆区：保存动态产生的数据，如：new出来的对象和数组，在不使用的时候由Java回收 器自动回收

1. Android解决内存泄漏的例子

- 单例造成的内存泄漏：在单例中，使用context.getApplicationContext()作为单例的 context
- 匿名内部类造成的内存泄漏：由于非静态内部类持有匿名外部类的引用，必须将内部类 设置为static
- Handler造成的内存泄漏：使用static的Handler内部类，同时在实现内部类中持有 Context的弱引用
- 避免使用static变量：由于static变量会跟Activity生命周期一致，当Activity退出后台被 后台回收时，static变量是不安全，所以也要管理好static变量的生命周期
- 资源未关闭造成的内存泄漏：比如Socket、Broadcast、Cursor、Bitmap、ListView 等，
- 使用完后要关闭 AsyncTask造成的内存泄漏：由于非静态内部类持有匿名内部类的引用而造成内存泄漏，可以通过AsyncTask内部持有外部Activity的弱引用同时改为静态内部类或在 onDestroy()中执行AsyncTask.cancel()进行修复

☐ 内存管理

1. Android内存管理机制

- 分配机制
- 管理机制

1. 内存管理机制的特点

- 更少的占用内存
- 在合适的时候，合理的释放系统资源
- 在系统内存紧张的时候，能释放掉大部分不重要的资源
- 能合理的在特殊生命周期中，保存或还原重要数据

1. 内存优化方法

- Service完成任务后应停止它，或用IntentService（因为可以自动停止服务）代替 Service
- 在UI不可见的时候，释放其UI资源
- 在系统内存紧张的时候，尽可能多的释放非重要资源
- 避免滥用Bitmap导致内存浪费
- 避免使用依赖注入框架
- 使用针对内存优化过的数据容器
- 使用ZIP对齐的APK
- 使用多进程

☐ 冷启动与热启动

- 冷启动：在启动应用前，系统中没有该应用的任何进程信息

- 热启动：在启动应用时，在已有的进程上启动应用（用户使用返回键退出应用，然后马 上又重新启动应用

1. 区别

- 冷启动：创建Application后再创建和初始化MainActivity
- 热启动：创建和初始化MainActivity即可

1. 冷启动的时间计算

这个时间值从应用启动（创建进程）开始计算，到完成视图的第一次绘制为止

1. 冷启动流程

- Zygote进程中fork创建出一个新的进程
- 创建和初始化Application类、创建MainActivity
- inflate布局、当onCreate/onStart/onResume方法都走完
- contentView的measure/layout/draw显示在界面上
-

总结：Application构造方法->attachBaseContext()->onCreate()->Activity构造方法>onCreate()->配置主题中背景等属性->onStart()->onResume()->测量布局绘制显示在界 面上

1. 冷启动优化

- 减少第一个界面onCreate()方法的工作量
- 不要让Application参与业务的操作 不
- 要在Application进行耗时操作
- 不要以静态变量的方式在Application中保存数据
- 减少布局的复杂性和深度
- 不要在mainThread中加载资源
- 通过懒加载方式初始化第三方SDK

- [ ] 内存对象序列化

- Serializeble：是java的序列化方式，Serializeble在序列化的时候会产生大量的临时对 象，从而引起频繁的GC
- Parcelable：是Android的序列化方式，且性能比Serializeble高，Parcelable不能使用在要将数据存储在硬盘上的情况

- [ ] 进程保活

- 利用系统广播拉活
- 利用系统Service机制拉活
- 利用Native进程拉活

- 利用JobScheduler机制拉活
- 利用账号同步机制拉活
-

1. Android进程回收策略

- Low memory Killer（定时执行）：通过一些比较复杂的评分机制，对进程进行打分， 然后将分数高的进程判定为bad进程，杀死并释放内存
- OOM_ODJ：判别进程的优先

1. 进程的优先级

- 空进程
- 后台进程
- 服务进程
- 可见进程
- 前台进程

- [ ] Dalvik与JVM

1、Dalvik与JVM不同

- 执行的文件不同，class和dex
- 类加载系统区别比较大
- Dalvik可以同时存在多个，即使一个退出了也不会影响其他程序
- Dalvik是基于寄存器的，JVM是基于栈的

2、Dalvik与ART不同

- Dalvik使用JIT（Just In Time 运行时编译）来将字节码转换成机器码，效率低
- ART采用AOT（Ahead Of Time 运行前编译）预编译技术，执行速度更快
- ART会占用更多的应用安装时间和存储空间

1. 发送与接收短信流程:

SmsManager -> ISms.aidl -> UiccSmsController(获取对应卡槽SIM)
-> IccSmsInterfaceManager (提供可访问SMS的接口) -> SMSDispatcher (分派并调用RIL层)
-> RIL(底层封装并在modem层进行发送)

RIL(socket从modem获取到上层,) -> Registrant(handler处理分派) -> InboundSmsHandler(发送广播)

1. 应用的启动

Launcher -> Activity(intent) -> Instrumentation(程序与系统的监控) -> ActivityManagerService
->ActivityThread(开启进程) ->

Keyguard启动

SystemServer-> WindowManagerService-> mKeyguardDelegate

-> KeyguardService -> KeyguardViewMediator(SystemUI) -> KeyguardDisplayguard

-> KeyguardBouncer(绘制) -> KeyguardSecurityContainer