面试流程总结

笔记本:	android面试总结		
创建时间:	2019/10/8 15:20	**更新时间:**	2019/10/15 16:02
作者:	冯建伟		

一、自我介绍一下

面试官您好，我是毕业于电子科技大学，计算机专业，并在深圳康冠科技集团担任Android工程师职务，有两年工作经验。
在前公司，我主要负责做Android应用层的开发，同时兼职做部分Android系统层开发。
Android应用层主要是参与实现一些APP的代码模块，主要是：Tv管家中的升级模块和网络测速、网络诊断模块。
独立开发的壁纸商城、
协同开发的Launcher等。
平时主要使用的开源框架为：okhttp3、retrofit和glide。

二、问答环节：
1.Activity的生命周期：
正常流程是：onCreate——》onStart——》onResume——》onPause——》onStop——》onDestroy
onCreate：做一些初始化工作，使用setContentView加载布局资源，使用Bundle对象恢复异常情况下Activity结束的状态。
onStart：activity正在被启动，但是activity还在后台，onResume的时候才会切换到前台
onResume：activity切换到前台并可以交互
onPause：activity正在被停止，正常会马上调用onStop，特殊会回到onResume状态。
onPause不能执行耗时操作，因为onPause执行完才会执行新的Activity的onResume，会造成卡死。
onStop：activity已经切换到了后台，可以做一些微量的回收工作，也不能做太耗时的工作
onDestroy：activity即将被销毁，可以做资源回收工作。

onRestart：activity重新被启动，后面开始执行onStart等方法。

普通操作下的变化：finish方法和按返回键都会onDestroy销毁activity，其他都会再次调用onRestart方法。

横竖屏切换：
正常来说，会导致activity的销毁和重建。
但是使用onSaveInstanceState和onRestoreInstanceState可以来恢复状态。
在onStop之前，调用onSaveInstanceState方法来保存activity状态；
然后重新创建之后，调用onRestoreSaveInstanceState来恢复状态。
流程是：onPause——》onSaveInstanceState——》onDestroy——》onCreate——》onStart——》onRestoreInstanceState——》onResume

避免横竖屏切换时，Activity的销毁和重建，在AndroidManifest文件中指定如下属性：
android：configChanges=“orientation | screenSize”
既然都不销毁和重建了，那么bundle的数据也不需要保存和恢复了！

2.Activity的四种启动模式：
标准模式：standard，创建一个activity往栈里面方一个
栈顶复用模式：singleTop 栈顶的activity不会再次创建
栈内复用模式：singleTask 把要用的activity放置到栈顶，之上的activity全部出栈
栈内单例模式：singleInstance 重新创建一个栈，activity单独放在里面

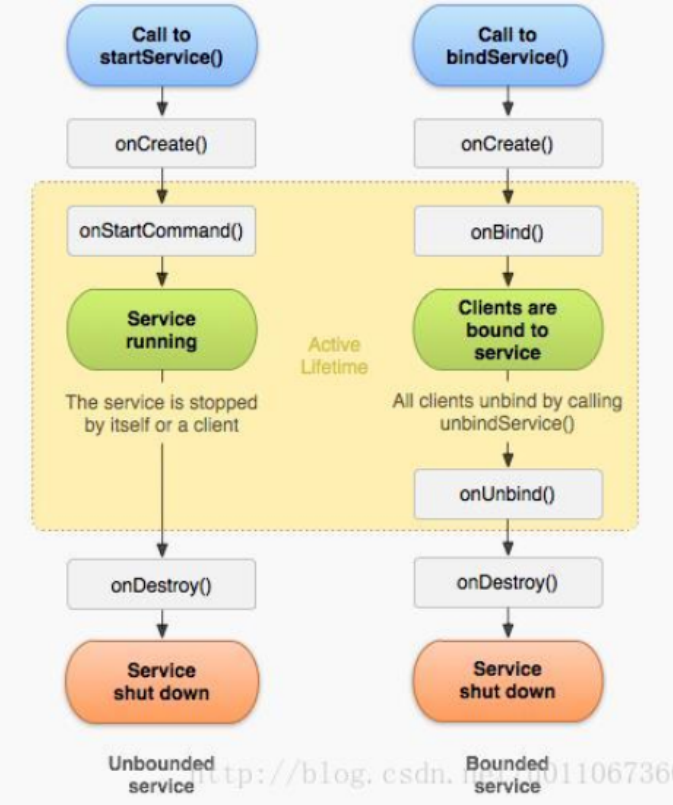
3.service
service也是Context的子类，只不过他没有UI界面，属于后台运行的组件，但是同样运行于UI线程，不能执行耗时操作

按运行地点分类：
分为本地服务（Local Service）和远程服务（Remote Service）
本地服务主要依赖主进程，主进程被kill之后，本地服务也就被杀死了
远程服务主要是系统级服务

按运行类型分类：
分为前台服务和后台服务
前台服务是指由Notification通知的
后台服务是指没有Notification通知的

按使用方式分类：
分为startService启动的服务和bindService启动的服务，还有两者都是用到的服务

Service的生命周期



onCreate 一次调用

onStartCommand（）多次调用
onBind() 一次调用就绑定了，下次再调用bindService也不会回调这个方法
onUnBind（）一次调用就解除绑定了，下次再调用unBindService会抛出异常

onDestroy 一次调用

使用了bindService就不会调用onStartCommand方法

普通使用startService方法的话，要在子线程中执行耗时操作，并且要在onDestroy中关闭子线程
使用bindService启动可交互服务，通过ServiceConnection 来操作返回值

前台调用

通过以下方式绑定服务：

```
bindService(mIntent,con,BIND_AUTO_CREATE);
```

其中第二个参数：

```
private ServiceConnection con = new ServiceConnection() {
@Override
public void onServiceConnected(ComponentName name, IBinder service) {
BackService.MyBinder myBinder = (BackService.MyBinder) service;
myBinder.showTip();
}

@Override
public void onServiceDisconnected(ComponentName name) {

}
};
```

当建立绑定后，onServiceConnected中的service便是Service类中onBind的返回值。如此便可以调用后台服务类的方法，实现交互。

当调用unbindService()停止服务，同时要在onDestory()方法中做好清理工作。

注意：通过bindService启动的Service的生命周期依附于启动它的Context。因此当前台调用bindService的Context销毁后，那么服务会自动停止。

android进程保活的一般套路

进程的优先级越高，那么越不容易被杀死，一般的思路就是想办法提高进程的优先级——最好设置成前台进程。
比如设置一个像素点的activity，监听屏保广播，当屏幕保护的时候，打开这个activity。
或者弄一个前台Service。

IntentService
https://blog.csdn.net/github_39367000/article/details/79797189
主要是重写一个onHandleIntent方法来执行耗时操作。源码是IntentService继承了Service，目的是解决service不能执行耗时操作的作用。
大体上是通过HandlerThread来执行子线程操作，然后把结果发给onHandleIntent的逻辑。
IntentService本身的好处是可以直接实现多线程的异步加载，而且不需要考虑Service的结束问题，会自动执行完Intent就自己关闭。
IntentService不通过bindService启动的原因是bindService返回的是一个null值。

首先是onCreate方法，通过HandlerThread单独开启一个名为IntentService的线程，创建一个名叫ServiceHandler的内部Handler，再将这两者绑定。这样IntentService中就可以使用 HandlerThread，并使用Handle传递消息。在ServiceHandler中的handleMessage方法中调用了抽象方法onHandleIntent()，这样，我们就能通过重写onHandleIntent()处理耗时任务了。
创建的Handler是属于线程的，所以handleMessage调用的抽象方法onHandleIntent就是在线程当中的

```
public void onCreate(){
    super.onCreate();
    //HandlerThread继承自Thread，内部封装了Looper
    //通过实例化HandlerThread新建线程并启动
    //所以使用IntentService时不需要额外新建线程
    HandlerThread thread
= new HandlerThread("IntentService["+mName+"]");
    thread.start();

    //获得工作线程的Looper，并维护自己的工作队列
    mServiceLooper = thread.getLooper();
    //讲上述获得Looper与新建的mServiceHandler进行绑定
    //新建的Handler是属于工作线程的
    mServiceHandler = new ServiceHandler(mServiceLooper);//这个就是
Handler，但是他是属于thread线程的
}
```

```
private final class ServiceHandler extends Handler{
   public ServiceHandler(Looper looper){
      super(looper);
   }

   public void handleMessage(Message msg){
```

```
        //onHandleIntent方法在工作线程中执行
        onHandleIntent((Intent) msg.obj);
        //处理完会自动调用stopSelf停止服务
        stopSelf(msg.arg1);
    }
}
```

https://blog.csdn.net/javazejian/article/details/52426425
IntentService使用案例
————————————————

AsyncTask 用来做一个异步加载
三个参数：Params、Progress、Result
四个方法：onPreExecute、doInBackground（Params...）、onProgressUpdate(Progress...)、onPostExecute(Result)

内部原理：
通过FutureTask来执行线程操作，通过IntentHandler来执行UI更新
————————————————————————————————————

4.ContentProvider
主要是做数据的交互和共享，跨进程通信使用
内容：
统一资源标识符URI ：
自定义URI=content：//com.carson.provider/User/1
主题名 授权信息 表明 记录

然后根据URI返回MIME类型的内容
ContentProvider.getType(uri);

而ContentProvider本质上主要是对数据的增删改查：
insert、update、delete和query

```
public Cursor query(Uri uri, String[] projection, String selection, String[] selecti
// 外部应用 获取 ContentProvider 中的数据
```

query可以通过游标Cursor来直接访问。
ContentResolver resolver = getContentResolver();
Uri uri = Uri.parse("content://cn.scu.myprovider/user");
Cursor cursor = resolver.query(uri, null, null, null, "userid desc");

5.Fragment
Fragment可以理解为Activity中的Activity，生命周期比Activity对了几个：
onAttach——》onCreate——》onCreateView——》onActivityCreated——》onStart——》onResume——》onPause——》onStop——》onDestroyView——》onDestroy——》onDetach

onAttach:当activity和Fragment发生关联的时候调用
onCreateView：当创建fragment的时候调用
onActivityCreateView：回调Activity的onCreate的时候调用
onDestroyView：当移除Fragment的时候调用
onDetach：取消Fragment和Activity关联的时候调用

```
public class MyFragment extends Fragment{
    public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState){
        View v = inflater.inflate(R.layout.item_fragment, container,
flase);
         return v;
    }
}
```

主界面在onCreate中：

```
MyFragment myFragment= new MyFragment（）；
FragmentManager manager= getSupportFragmentManager（）；
FragmentTransaction transaction = manager.beginTransaction();
transaction.add(r.di.myfragment, myFragment).commit;
```

FragmentTransaction是Fragment的管理器对象，通过他来调度Fragment。
方法有replace、add、remove、hide和show
detach和attach是决定是不是显示该视图的方法
最后一定要commit才能显示

Fragment回退栈：FragmentTransaction.addToBackStack(null)
目的是为了防止Fragment重绘

6.Android消息机制
流程：在子线程执行完耗时操作的时候，handler发送消息sendMessage发送Message，然后会调用MessageQuene.enqueueMessage向消息队列添加消息。然后Looper.loop开启循环，调用MessageQueue.next方法从线程池中读取消息，通过Handler的dispatch方法把消息分发给Handler，Handler通过handleMessage接受处理消息。

Looper是唯一的

handler可能造成泄漏的原因:
handler是非静态的，activity要回收，但是handler在别的线程有引用，导致activity没法被回收，因为handler是绑定当前UI线程的，就会导致内存泄漏问题。
解决方法就是把handler声明称static静态的，然后加一个绑定activity的弱引用，只要activity回收了，那么handler也会销毁

```
static class MyHandler extends Handler{
    WeakReference<Activity> mActivityReference;
```

```
    MyHandler(Activity activity){
        mActivityReference = new WeakReference<Activity>(activity);
    }

    @Override
    public void handleMessage(Message msg){
        final Activity activity = mActivityReference.get();
         if(activity != null){
            mInageView,setImageBitmap(mBitmap);
         }
    }

}
```

HandlerThread
HandlerThread是帮我们封装好可以实现多线程而不需要手动多次开启线程的封装类
使用：
HandlerThread thread= new HandlerThread("线程名字");
启动：
thread.start();
讲HandlerThread和Handler绑定的方法

```
mHandler = new Handler(thread.getLooper()){
   @Override
   public void handleMessage(Message msg){//Handler属于线程
      checkForUpdate();//这个里面执行的就是耗时操作
      if(isUpdate){
         mHandler.sendEmptyMessage(MSG_UPDATE_INFO);
      }
   }
}
```

原理：
HandlerThread继承了Thread，并在Runnable方法里面获得了当前线程的Looper方法，也就是在子线程中有一个Looper来循环管理MessageQueue。
然后子线程创建一个Handler，并绑定这个Looper，就相当于子线程有Handler可以使用，handleMessage方法可以执行耗时操作了。原理是HandlerThread中的Run方法重写了handleMessage方法。
https://www.cnblogs.com/ldq2016/p/8192210.html

主线程可以调用子线程的Handler去更新子线程的操作，

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_handler_thread);
    imageView= (ImageView) findViewById(R.id.image);
    //创建异步HandlerThread
    HandlerThread handlerThread = new HandlerThread("downloadImage");
    //必须先开启线程
    handlerThread.start();
    //子线程Handler
    Handler childHandler = new Handler(handlerThread.getLooper(),new ChildCallback());
    for(int i=0;i<7;i++){
        //每个1秒去更新图片
        childHandler.sendEmptyMessageDelayed(i,1000*i);
    }
}
```

然后子线程的Handler可以去更新主线程的UI操作

```
/**
 * 该callback运行于子线程
 */
class ChildCallback implements Handler.Callback {
    @Override
    public boolean handleMessage(Message msg) {
        //在子线程中进行网络请求
        Bitmap bitmap=downloadUrlBitmap(url[msg.what]);
        ImageModel imageModel=new ImageModel();
        imageModel.bitmap=bitmap;
        imageModel.url=url[msg.what];
        Message msg1 = new Message();
        msg1.what = msg.what;
        msg1.obj =imageModel;
        //通知主线程去更新UI
        mUIHandler.sendMessage(msg1);
        return false;
    }
}
```

从源码可以看出HandlerThread继续自Thread,构造函数的传递参数有两个，一个是name指的是线程的名称，一个是priority指的是线程优先级，我们根据需要调用即可。其中成员变量mLooper就是HandlerThread自己持有的Looper对象。onLooperPrepared()该方法是一个空实现，是留给我们必要时可以去重写的，但是注意重写时机是在Looper循环启动前，再看看run方法：

```
@Override
public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        synchronized (this) {
            mLooper = Looper.myLooper();
            notifyAll(); //唤醒等待线程
        }
        Process.setThreadPriority(mPriority);
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
    }
```

前面我们在HandlerThread的常规使用中分析过，在创建HandlerThread对象后必须调用其start()方法才能进行其他操作，而调用start()方法后相当于启动了线程，也就是run方法将会被调用，而我们从run源码中可以看出其执行了**Looper.prepare()**代码，这时Looper对象将被创建，当Looper对象被创建后将绑定在当前线程（也就是当前异步线程），这样我们才可以把Looper对象赋值给Handler对象，进而确保Handler对象中的handleMessage方法是在异步线程执行的。接着将执行代码：

```
synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll(); //唤醒等待线程
    }
```

7.Android事件分发机制
一个事件指的是从按下down到离开屏幕up的一个完整过程，中间包括无数个move事件都算一个事件。
事件传递主要是三个：activity——》viewGroup——》view

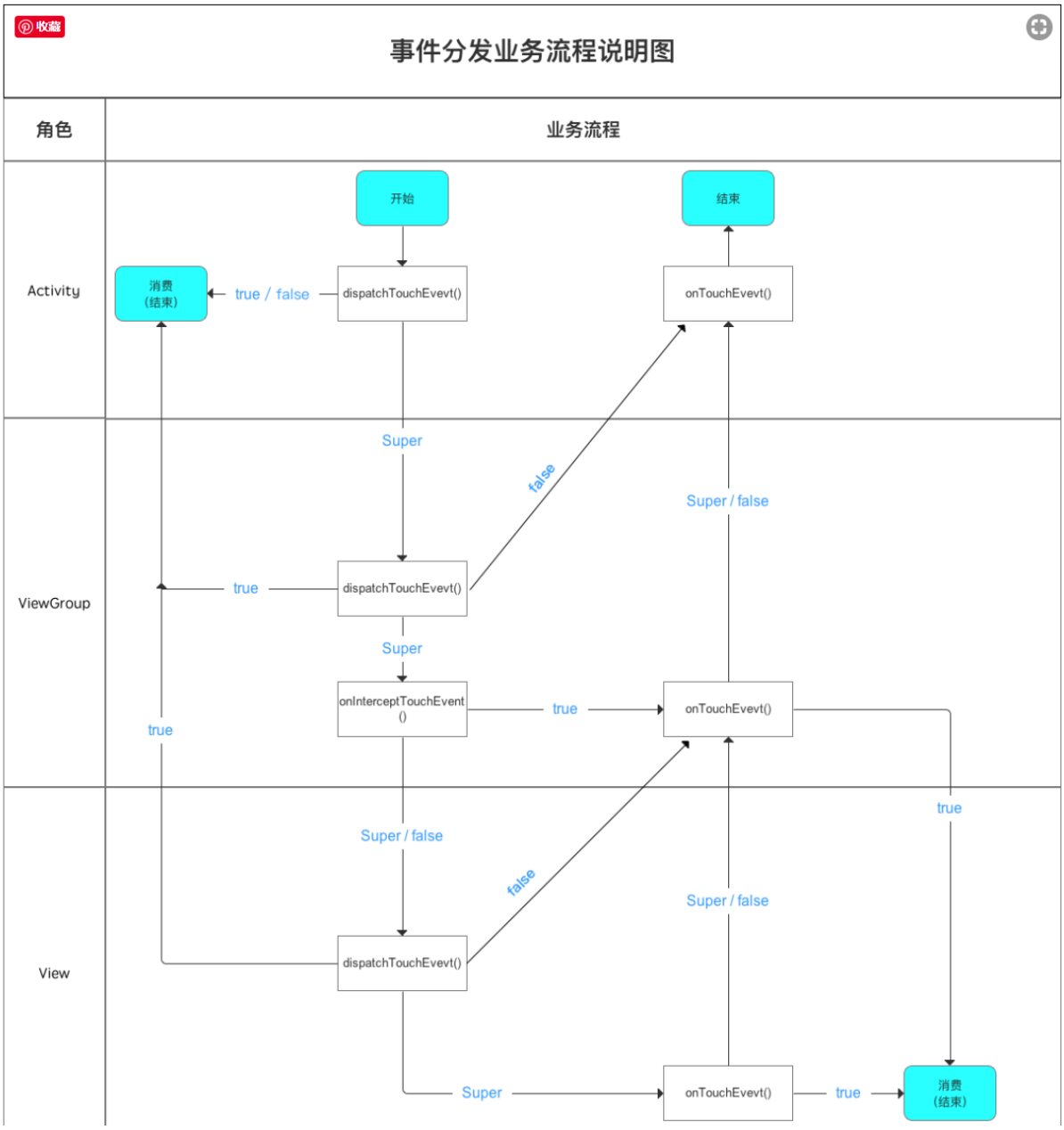
true：表示消费，事件不再传递
false：表示不消费，事件继续传递，调用父类的onTouchEvent方法、
只有分发给onTouchEvent事件才会 调用这个方法。

比如按下屏幕，执行MotionEvent.ACTION_DOWN;
首先是activity的dispatchTouchEvent来处理事件：
如果执行false，那么会调用父类的onTouchEvent方法，但是activity没有父类，所以事件停止
如果执行true，就传给activity本事的onTouchEvent方法消费掉。
如果执行super，就调用ViewGroup的dispatchTouchEvent方法。

ViewGroup的dispatchTouchEvent方法会调用本身的interceptTouchEvent方法，如果interceptTouchEvent是true，那么就调用本身的onTouchEvent方法，后续的move和up也不会继续传递下去，都会直接调用ViewGroup的onTouchEvent方法。
如果inerceptTouchEvent是false，那么就不拦截事件，调用子类view的dispatchTouchEvent方法。

view的dispatchTouchEvent方法一种是false，那么也不处理，就返回viewGroup的onTouchEvent方法。
view的dispatchTouchEvent方法返回true，那么就调用自己的onTouchEvent。

然后onTouchEvent一般两种，返回true就自己消费掉，结束事件，返回false就继续往回传递。



默认都是执行的super方法，view的super方法会执行他本身的onTouchEvent方法。
https://blog.csdn.net/u013637594/article/details/82493350

1.dispatchTouchEvent方法本身的true和false不会影响后续事件
2.onTouch > onTouchEvent > onClick 高优先级的事件返回true，低优先级的事件就不会继续执行了。
3.interceptTouchEvent事件拦截，后续的move和up都不会执行这个方法了，都会执行自身的onTouch方法
4.如果ViewGroup拦截的是Move事件，而不是Down事件，那么Down事件会传递下去，而第一个Move事件不会直接执行本身的onTouchEvent方法，而是传一个Cancel给View，后续的Move事件才会传给自身的onTouch。

8.LurCache原理
最近最少使用的缓存策略

```
int maxMemory = (int) (Runtime.getRuntime().totalMemory() / 1024);
int cacheSize = maxMemory / 8;
mMemoryCache = new LruCache<String, Bitmap>(cacheSize){
    @Override
    protected int sizeOf(String key, Bitmap bitmap){
        return value.getRowBytes() * value.getHeight() / 1024;
    }
}
```

1.设置LruCache缓存大小。一般是当前进程容量的1/8
2.重写sizeOf方法，计算出缓存的每张图片的大小
原理：
LurCache是通过叫LinkedHashMap的双向链表结构+数据的结构实现的。
有一个trimToSize方法判断是不是缓存已满，如果满了就删除最近最少使用的数据【队列尾部的元素】。put方法存的时候才会判断
get方法读取的时候会把读取的数据更新到队列的头部。
【最近访问的会更新到头部，删除的时候会从队列尾部删除】

9.Activity、Window和DecorView
https://lrh1993.gitbooks.io/android_interview_guide/content/android/basis/decorview.html
activity并不负责视图的控制，他只是控制视图的生命周期和处理事件。真正控制视图的是window。一个activity包含了一个Window，Window才是真正代表了一个窗口。Activity就像一个控制器，统筹视图的添加与显示，以及通过其他的回调方法，来与Window以及View进行交互。

Window是视图的承载器，内部有一个DecorView，而这个DecorView才是View的根布局。
Window是一个抽象类，实际在Activity中持有的是其子类PhoneWindow。
PhoneWindow内部有个子类DecorView，通过
创建DecorView来加载Activity中设置的布局R.layout.activity_main。
Window通过WindowManager将DecorView加载其中，并将DecorView交给ViewRoot进行视图绘制以及其他交互。

DecorView
DecorView是FrameLayout 的子类，内部是一个LinearLayout的竖直线性布局

分为三部分：
ViewStub：设施ActionBar、
中间是标题栏：Theme设置
下边的是内容栏（主要是使用这个） setContentView就是设置这个

10.View的绘制流程
自上而下
DecorView ——》 ViewGroup——》View

Measure 测量
主要是用来获取view的宽高、
onMeasure(int widthMeasureSpec, int heightMeasureSpec);

确定View自身的位置
onLayout(int l, int t, int r, int b)

绘制过程
onDraw（Canvas canvas）

10.Android的进程间通信
https://lrh1993.gitbooks.io/android_interview_guide/content/android/basis/ipc.html
1.Intent、Service、Receiver都支持在Intent中传递Bundle数据，而Bundle实现了 Parcelable接口，可以在不同的进程进行传输
2.使用文件共享，SharedPreference在内存中有缓存，高频使用可能导致问题

3.Messager实现跨进程通信
服务端创建一个Service处理客户端的请求，同时通过一个Handler对象来实例化一个Messager对象，然后在Service的onBind方法返回这个Messager对象的底层binder。

服务端：

```
private static class MessagerHandler extends {
    @Override
    public void handleMessage(Message msg){
        switch(msg.what){
            case MyContants.MSG_FROM_CLIENT:
                Messager client = msg.replyTo;//replyTo回调，获得客户端的Messager
                client.send(message);
                break;
            default:
                super.handleMessage(msg);
        }
    }
}

//用创建出来的Handler作为信使
private final Messager mMessager = new
```

```
Messager(new MessagerHandler());

//通过onBind方法返回客户端
@Override
public IBinder onBind(Intent intent){
    return mMessager.getBinder();
}
```

客户端首先绑定Service，回调onServiceConnected方法中获取Message = new Message（iBinder）；
然后读取Message内的信息，最后调用Message.send(msg)调用service进程的逻辑。

```
private ServiceConnection mConnection = new ServiceConnection(){
    @Override
    public void onServiceConnected(ComponentName name, IBinder service){
        mService = new Message(service);
        Message msg = Message.obtain(null, MyContants.MSG_FROM_CLIENT);
        Bundle data = new Bundle();
        data.putString("msg", "hello, this is client");
        msg.setData(data);
        try{
            mService.send(msg);
        }catch(RemoteExcetption e){
           e.printStackTrace();
            }
    }
}
```

客户端和服务端都是通过拿对方的Messager发送消息。
只不过服务端是通过replyTo获得对方的Messager；而客户端通过bindService的回调方法onServiceConnected的IBinder获得。

基本message类型：msg.what 消息类型、msg.data 对应数据类型，对象需要parcelable序列化、msg.replyTo返回一个Message对象，可以直接send回调；
msg.obj 可以传递任意类型，类对象需要parcelable序列化

Message.obtain比直接New一个Message的好处是，减少内存使用；obtain是一个列表里面找，没有才创建。

4.ContentProvider
5.Socket
本质：unix设计哲学，一切皆文件。都可以打开关闭，读写。
Socket是应用层和传输层之间的协议，分为流式Socket（针对TCP服务），数据报式Socket（UDP服务）
【IO操作必须放到子线程执行】
Socket的实现本质上也是分为服务端和客户端。
客户端建立Socket对象，传入IP地址和端口号，然后写入数据发给服务端，并接受服务端发送过来的数据，最后要close。
服务端创建ServerSocket对象，并指定端口号，

然后通过ServerSocket获取客户端的实例（ServerSocket.accept），
通过Socket获取输入流，接受客户端发来的消息；
通过Socket获取输出流，写入数据到客户端，最后关闭close。

服务端代码：

```
public class TCPServerService extends IntentService{
    private static final[] defultMessages = {"xxxxxxx"};
    private int index = 0;
    private boolean isServiceDestroy = false;
    public TCPServerService(){
        super("TCP");
    }

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

    @Override
    protected void onHandleIntent(Intent intent){
        try{
            //1.设置本地端口号
            ServerSocket serverSocket = new ServerSocket(8888);
            Socket sokcet = serverSocket.accept();//建立与客户端的连接

            //2.获取输入流，接受用户发来的消息
            InputStream inputStream = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputSteam));

            //3.获取输出流，向客户端发送消息
            OutputStream outputStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(new
OutputStreamWriter(outputStream));

            //4.通过循环不断读取客户端发送来的消息，并发送
            while(!isServiceDestroy){
                String readLine = reader.readLine();
                if(!TextUtils.isEmpty(readLine)){
                    String sendMsg = index < defaultMessages.length ?
default : defaultMessages[index];
                writer.println(sendMsg + '\r');
                writer.flush();//刷新流
                index ++;
                }
            }
            //关闭流
            inputStream.close();
            reader.close();
            outputStream.close();
            writer.close();
            socket.close();
            serverSocket.close();
        }catch (IOException e){
```

```
            e.printStackTrace();
        }
    }

    @Override
    public void onDestory(){
        super.onDestroy();
        isServiceDestroy = true;
    }

}
```

客户端:

```
public class SocketActivity extends AppCompatActivity{
    private TextView mTvChatContent;
    private EditText mEtSendContent;
    private Intent mIntent;

    private static final int CONNECT_SERVER_SUCCESS = 0；//连接服务器成
功
    private static final int MESSAGE_RECEIVE_SUCCESS = 1;//接受到服务
器的消息
    private static final int MESSAGE_SEND_SUCCESS = 2;//消息发送

    private Handler mHandler = new Handler(new Handler.Callback(){
        @Override
        public boolean handleMessage(Message msg){
            switch(msg.what){
                case CONNECT_SERVER_SUCCESS:
                    mTvChatContent.setText("与聊天室连接成功\n");
                    break;
                case MESSAGE_RECEIVE_SUCCESS:
                    String msgContent =
mTvChatContent.getText.toString();

mTvChatContent.setText(msgContent+msg.obj.toString() + "\n");
                    break;
                case MESSAGE_SEND_SUCCESS:
                    mEtSendContent.setText("");

mTvChatContent.setText(mTvContent.getText().toString());
                    break;
            }
            return false;
        }
    });

    private PrintWriter mPrintWriter;
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_socket);
        mTvChatContent = findViewById(R.id.tv_chat_content);
        mEtSendContent = findViewById(R.id.et_send_content);
        //启动服务
        mIntent = new Intent(this, TCPServerService.class);
```

```
        startService(mIntent);
        new Thread(new Runnable()){
            @Override
            punlic void run(){
                //连接服务端，IO操作必须放在子线程进行
                connectTCPServer();
            }
        }.start();

        private Socket mSocket = null;
        private void connectTCPServer(){
            while(mSocket == null){
                try{
                    //创建Socket
                    mSocket = new Socket("localhost", 8888);
                    mPrintWriter = new PrintWriter(new
OutputStreamWriter(mSocker));
                    if (mSocket.isConnected()){

mHandler.sendEmptyMessage(CONNECT_SERVER_SUCCESS);
                    }
                }catch(IOException e){
                    e.printStackTrace();
                    SystemClock.sleep(1000);
                }
            }

            //通过循环，不断的接受服务端发送的消息
            try{
                BufferedReader reader = new BufferedReader(new
InputStreamReader(mSocket.getInputSteam()));
                while (!SocketActivity.this.isFinishing()){//当前
Activity没有销毁的情况下
                    String msg = reader.readLine();
                    if(!TextUtils.isEmpty(msg){

mHandler.obtainMessage(MESSAGE_RECEIVE_SUCCESS, msg).sendToTarget();
                    }
                }
                //关闭流
                mPrintWriter.close();
                reader.close();
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }

}
https://blog.csdn.net/hzw2017/article/details/81210979

【核心：Socket核心就是客户端通过Socket对象建立，服务端通过
ServerSocket.accept方法获得客户端的Socket，然后都是调用
```

```
Socket.getInputStream()方法来读取数据，通过Socket.getOutputStream()方法来输出数据
写入数据使用PrintWriter对象】
```

6.AIDL
暂时不看

11.Android Bitmap压缩策略
BitmapFactory.Options的参数：
inSampleSize参数：采样率，对图片的像素进行缩放。当inSampleSize>1，采样后的图片会缩放比例为1/inSimpleSize的二次方
例如一张1024x1024像素的图片，采用ARGB_8888的格式存储，内存大小为1024x1024x4=4M
inSimpleSize=2；采样后的图片大小为4/2^2=1M
【inSimpleSize只能选取2的倍数，比如设置为3，那么区附近的值2，但是不能是4
通常的做法是计算出缩放比，通过比较图片的实际大小/所需的宽高大小，然后取值比较小的那个，避免图片不能铺满从而拉伸导致的模糊】

inJustDecodeBounds参数
目的是在不加载图片的时候获得图片的宽高信息，设置为true
当选择好缩放比之后，在设置为false，那么再次加载图片就是加载缩放了之后的。

高效加载Bitmap流程：
设置BitmapFactory.Options的inJustDecodeBounds参数为true，然后加载图片
从BitmapFactory.Options中取出图片的原始宽高属性
根据要求的宽高设置采样率inSimpleSize的值
设置BitmapFactory.Options的inJustDecodeBounds参数为false，然后重新加载图片

代码流程：

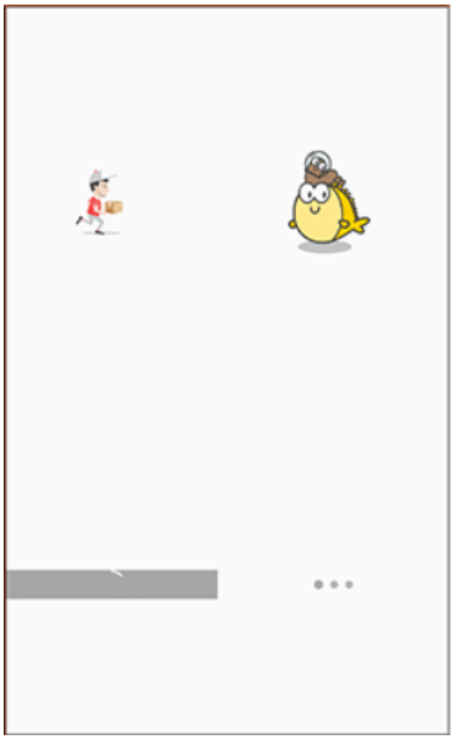
```
public static Bitmap decodeSampledBitmapFromResource(Resources res,
int resId, int reqWidth, int reqHeight){
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = false;
    BitmapFactory.decodeResource(res, resId, options);
    options.inSampleSize = calculateInSimpleSize(options, reqHeigth,
reqWidth);
    options.inJustDecodeBounds = false;
    BitmapFactory.decodeRresource(res, resid, options);
}

private static int calculatInSampleSize(BitmapFactory.Options
options, int reqWidth, int reqHeight){
    int height = options.outHeight;
    int width = options.outWidth;
    int inSimpleSize = 1;
    if(Height > reqHeight || width > reqWidth){
        int halfHeight = height / 2;
        int helfWidth = width / 2;
        //计算缩放比
```

```
        while((helfHeigth/inSampleSize) >= reqHeight &&
             (helfWidth/inSampleSize) >= reqWidth)
        inSimpleSize *= 2;
    }
    return inSimpleSize;
}
```

12.Android 动画总结

1.帧动画

就是把图片按照一个list放置，然后一帧一帧的播放



```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com.apk/res/android">
   <item
        android:drawable="@drawable/a_0"
        android:duration="100"/>
   <item
        android:drawable="@drawable/a_1"
        android:duration="100"/>
   <item
        android:drawable="@drawable/a_2"
        android:duration="100"/>
</animation-list>
```

```
protected void onCreate(Bunlde savedInstanceState){
    super.onCreate(savedInstatnceState);
    setContentView(R.layout.activity_frame_animation);
```

```
    ImageView animationImg1 = (ImageView)
findViewById(R.id.animation1);
    animationImg1.setImageResource(R.drawable.frame_anim1);
    AnimationDrawable animationDrawable1 = (AnimationDrawable)
animationImg1.getDrawable();
    animationDrawable1.start();
}
```

2.补间动画

https://lrh1993.gitbooks.io/android_interview_guide/content/android/basis/animator.html

四种：透明度（alpha）、位移（translate）、缩放（scale）、旋转（rotate）

xml实现：

alpha_anim.xml动画实现

```
<?xml version="1.0" encoding="utf-8">
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:duration="1000"
  android:fromAlpha="1.0"
  android:interpolator="@android:anim.accelerate_decelerate_interpolator"
  android:toAlpha="0.0"/>
```

scale.xml动画实现

```
<?xml version="1.0" encoding="utf-8">
<scale xmlns:android="http://schemas.android.com/apl/res/android"
  android:duration="1000"
  android:fromXScale="0.0"
  android:fromYScale="0.0"
  android:pivotX="50%"
  android:pivotY="50%"
  android:toXScale="1.0"
  android:toYScale="1.0"/>
```

```
Animation animation = AnimationUtils.loadAnimation(mContext,
R.anim.alpha_anim);
img = (ImageView) findViewById(R.id.img);
img.startAnimation(animation);
```

Interpolator:控制动画的变化速率，就是动画进行的快慢节奏

pivot：决定动画执行的参考位置

我们以pivotX为例，

pivotX取值	含义
10	距离动画所在view自身左边缘10像素
10%	距离动画所在view自身左边缘 的距离是整个view宽度的10%
10%p	距离动画所在view父控件左边缘的距离是整个view宽度的10%

pivotY 也是相同的原理，只不过变成的纵向的位置。如果还是不明白可以参考源码，在Tweened Animation中结合seekbar的滑动观察rotate的变化理解。

3.属性动画
入门

```
private void RotateAnimation(){
    ObjectAnimator anim = ObjectAnimator.ofFloat(myView, "rotation",
0f, 360f);
    anim.setRepeatCount(-1);
    anim.setRepeatMode(ObjectAnimator.REVERSE);
    anim.setDuration(1000);
    anim.start();
}

private void AlphaAnimation(){
    ObjectAnimator anim = OjectAnimator.ofFloat(myView, "alpha",
1.0f, 0.8f, 0.6f, 0.4f, 0.2f, 0.0f);
    anim.setRepeatCount(-1);
     anim.setRepeatMode(ObjectAnimator.REVERSE);
     anim.setDuration(1000);
     anim.start();
}
```

组合实现：

```
ObjectAnimator alphaAnim = ObjectAnimator.ofFloat(myView, "alpha",
1.0f, 0.8f, 0.6f, 0.4f, 0.0f);
ObjectAnimator scaleXAnim = ObjectAnimator.onFloat(myView, "scaleX",
0.0f, 1.0f);
ObjectAnimator scaleYAnim = ObjectAnimator.onFloat(myView, "scaleY",
0.0f, 2.0f);
ObjectAnimator rotateAnim = ObjectAnimator.onFloat(myView,
"rotation", 0, 360);
ObjectAnimator transXAnim = ObjectAnimator.onFloat(myView,
"translationX", 100, 400);
ObjectAnimator transYAnim = ObjectAnimator.onFloat(myView,
"translationY", 100, 750);
AnimatorSet set = new AnimatorSet();
set.playTogether(alphaAnim, scaleXAnim, scaleYAnim, rotateAnim,
transXAnim, transYAnim);
```

```
set.setDuration(3000);
set.start();
```

属性动画和补间动画的区别：
1.属性动画真正移动了view的位置，而补间动画并不移动view的位置。
2.属性动画在activity关闭的时候没有结束，会导致activity无法释放而导致内存泄漏，而补间动画不需要考虑这个问题。
所以属性动画在切换activity的时候执行onStop方法顺带停止掉动画。

插值器（Interpolator）：决定值的变化模式
常用的插值器类型9种：一般是AccelerateInterpolator（动画加速进行）、DecelerateInterpolator（减速）、LinearInterpolator（匀速）
估值器（TypeEvaluator）：决定值的具体变化数值

组合动画使用：AnimatorSet对象
.play（）一起播放
.with() 连接多个动画
.after()将现有动画插入到传入的动画之后执行
.before()将现有动画插入到传入的动画之前执行