android 中 任务、进程和线程的区别

https://blog.csdn.net/xujingcheng123/article/details/79925661

任务

在SDK中关于Task（guide/topics/fundamentals.html#acttask），有一个很好的比方，说，Task就相当于应用（application）的概念。在开发人员眼中，开发一个Android程序，是做一个个独门独户的组件，但对于一般用户而言，它们感知到的，只是一个运行起来的整体应用，这个整体背后，就是Task。

Task，简单的说，就是一组以栈的模式聚集在一起的Activity组件集合。它们有潜在的前后驱关联，新加入的Activity组件，位于栈顶，并仅有在栈顶的Activity，才会有机会与用户进行交互。而当栈顶的 Activity完成使命退出的时候，Task会将其退栈，并让下一个将跑到栈顶的Activity来于用户面对面，直至栈中再无更多 Activity，Task结束。

事件 Task栈

点开Email应用，进入收件箱（Activity A） A
选中一封邮件，点击查看详情（Activity B） AB
点击回复，开始写新邮件（Activity C） ABC
写了几行字，点击选择联系人，进入选择联系人界面（Activity D） ABCD
选择好了联系人，继续写邮件 ABC
写好邮件，发送完成，回到原始邮件 AB
点击返回，回到收件箱 A
退出Email程序 null

如上表所示，是一个实例。从用户从进入邮箱开始，到回复完成，退出应用整个过程的Task栈变化。这是一个标准的栈模式，对于大部分的状况，这样的Task 模型，足以应付，但是，涉及到实际的性能、开销等问题，就会变得残酷许多。比如，启动一个浏览器，在Android中是一个比较沉重的过程，它需要做很多初始化的工作，并且会有不小的内存开销。但与此同时，用浏览器打开一些内容，又是一般应用都会有的一个需求。设想一下，如果同时有十个运行着的应用（就会对应着是多个Task），都需要启动浏览器，这将是一个多么残酷的场面，十个Task栈都堆积着很雷同的浏览器Activity，是多么华丽的一种浪费啊。于是你会有这样一种设想，浏览器Activity，可不可以作为一个单独的Task而存在，不管是来自哪个Task的请求，浏览器的Task，都不会归并过去。这样，虽然浏览器Activity本身需要维系的状态更多了，但整体的开销将大大的减少，这种舍小家为大家的行为，还是很值得歌颂的。

如此值得歌颂的行为，Android当然会举双手支持的。在Android中，每一个Activity的Task模式，都是可以由Activity提供方)(通过配置文件...）和Activity使用方（通过Intent中的flag信息...）进行配置和选择。当然，使用方对Activity的控制力，是限定在提供方允许的范畴内进行，提供方明令禁止的模式，使用方是不能够越界使用的。

在SDK中（guide/topics/fundamentals.html#acttask），将两者实现Task模式配置的方式，写的非常清晰了。提供方对组件的配置，是通过配置文件(Manifest)<activity>项来进行的，而调用方，则是通过Intent对象的flag进行抉择的。相对于标准的Task栈的模式，配置的主要方向有两个：一则是破坏已有栈的进出规则，或样式；另一则是开辟新Task栈完成本应在同一Task栈中完成的任务。

对于应用开发人员而言，<activity>中的launchMode属性，是需要经常打交道的。它有四种模式："standard", "singleTop", "singleTask", "singleInstance"。

standard模式，是默认的也是标准的Task模式，在没有其他因素的影响下，使用此模式的Activity，会构造一个Activity的实例，加入到调用者的Task栈中去，对于使用频度一般开销一般什么都一般的Activity而言，standard模式无疑是最合适的，因为它逻辑简单条理清晰，所以是默认的选择。

而singleTop模式，基本上于standard一致，仅在请求的Activity正好位于栈顶时，有所区别。此时，配置成singleTop的Activity，不再会构造新的实例加入到Task栈中，而是将新来的Intent发送到栈顶Activity中，栈顶的Activity可以通过重载onNewIntent来处理新的Intent（当然，也可以无视...）。这个模式，降低了位于栈顶时的一些重复开销，更避免了一些奇异的行为（想象一下，如果在栈顶连续几个都是同样的Activity，再一级级退出的时候，这是怎么样的用户体验...），很适合一些会有更新的列表Activity展示。一个活生生的实例是，在 Android默认提供的应用中，浏览器（Browser）的书签Activity（BrowserBookmarkPage），就用的是singleTop。

singleTop模式，虽然破坏了原有栈的逻辑（复用了栈顶，而没有构造新元素进栈...），但并未开辟专属的Task。而singleTask，和singleInstance，则都采取的另辟Task的蹊径。标志为singleTask的Activity，最多仅有一个实例存在，并且，位于以它为根的Task中。所有对该Activity的请求，都会跳到该Activity的Task中展开进行。singleTask，很象概念中的单件模式，所有的修改都是基于一个实例，这通常用在构造成本很大，但切换成本较小的Activity中。在Android源码提供的应用中，该模式被广泛的采用，最典型的例子，还是浏览器应用的主Activity（名为Browser...），它是展示当前tab，当前页面内容的窗口。它的构造成本大，但页面的切换还是较快的，于 singleTask相配，还是挺天作之合的。

相比之下，singleInstance显得更为极端一些。在大部分时候singleInstance与singleTask完全一致，唯一的不同在于，singleInstance的Activity，是它所在栈中仅有的一个Activity，如果涉及到的其他Activity，都移交到其他Task中进行。这使得singleInstance的Activity，像一座孤岛，彻底的黑盒，它不关注请求来自何方，也不计较后续由谁执行。在Android默认的各个应用中，很少有这样的Activity，在我个人的工程实践中，曾尝试在有道词典的快速取词Activity中采用过，是因为我觉得快速取词入口足够方便（从notification中点选进入），并且会在各个场合使用，应该做得完全独立。

除了launchMode可以用来调配Task，<activity>的另一属性taskAffinity，也是常常被使用。taskAffinity，是一种物以类聚的思想，它倾向于将taskAffinity属性相同的Activity，扔进同一个Task中。不过，它的约束力，较之launchMode而言，弱了许多。只有当<activity>中的allowTaskReparen ting设置为true，抑或是调用方将Intent的flag添加FLAG_ACTIVITY_NEW_TASK属性时才会生效。如果有机会用到Android的Notification机制就能够知道，每一个由notification进行触发的Activity，都必须是一个设成FLAG_ACTIVITY_NEW_TASK的Intent来调用。这时候，开发者很可能需要妥善配置taskAffinity属性，使得调用起来的Activity，能够找到组织，在同一taskAffinity的Task中进行运行。

进程

在大多数其他平台的开发中，每个开发人员对自己应用的进程模型都有非常清晰的了解。比如，一个控制台程序，你可以想见它从main函数开始启动一个进程，到 main函数结束，进程执行完成退出；在UI程序中，往往是有一个消息循环在跑，当接受到Exit消息后，退出消息循环结束进程。在该程序运行过程中，启动了什么进程，和第三方进程进行通信等等操作，每个开发者都是心如明镜一本帐算得清清楚楚。进程边界，在这里，犹如国界一般，每一次穿越都会留下深深的印迹。

在Android程序中，开发人员可以直接感知的，往往是Task而已。倍感清晰的，是组件边界，而进程边界变得难以琢磨，甚至有了进程托管一说。Android中不但剥夺了手工锻造内存权力，连手工处置进程的权责，也毫不犹豫的独占了。

当然，Android隐藏进程细节，并不是刻意为之，而是自然而然水到渠成的。如果，我们把传统的应用称为面向进程的开发，那么，在Android中，我们做得就是面向组件的开发。从前面的内容可以知道，Android组件间的跳转和通信，都是在第三方介入的前提下进行，正由于这种介入，使得两个组件一般不会直接发生联系（于Service的通信，是不需要第三方介入的，因此Android把它全部假设成为穿越进程边界，统一基于RPC来通信，这样，也是为了掩盖进程细节...），其中是否穿越进程边界也就变得不重要。因此，如果这时候，还需要开发者关注进程，就会变得很奇怪，很费解，干脆，Android将所有的进程一并托管去了，上层无须知道进程的生死和通信细节。

在Android的底层，进程构造了底部的一个运行池，不仅仅是Task中的各个Activity组件，其他三大组件Service、Content Provider、Broadcast Receiver，都是寄宿在底层某个进程中，进行运转。在这里，进程更像一个资源池（概念形如线程池，上层要用的时候取一个出来就

好，而不关注具体取了哪一个...），只是为了承载各个组件的运行，而各个组件直接的逻辑关系，它们并不关心。但我们可以想象，为了保证整体性，在默认情况下，Android肯定倾向于将同一Task、同一应用的各个组件扔进同一个进程内，但是当然，出于效率考虑，Android也是允许开发者进行配置。

在Android中，整体的<application>（将影响其中各个组件...）和底下各个组件，都可以设置<process>属性，相同<process>属性的组件将扔到同一个进程中运行。最常见的使用场景，是通过配置<application>的process属性，将不同的相关应用，塞进一个进程，使得它们可以同生共死。还有就是将经常和某个Service组件进行通信的组件，放入同一个进程，因为与Service通信是个密集操作，走的是RPC，开销不小，通过配置，可以变成进程内的直接引用，消耗颇小。

除了通过<process>属性，不同的组件还有一些特殊的配置项，以Content Provider为例（通过<provider>项进行配置...）。<provider>项有一个mutiprocess的属性，默认值为false，这意味着Content Provider，仅会在提供该组件的应用所在进程构造一个实例，第三方想使用就需要经由RPC传输数据。这种模式，对于构造开销大，数据传输开销小的场合是非常适用的，并且可能提高缓存的效果。但是，如果是数据传输很大，抑或是希望在此提高传输的效率，就需要将mutiprocess设置成true，这样，Content Provider就会在每一个调用它的进程中构造一个实例，避免进程通信的开销。

既然，是Android系统帮助开发人员托管了进程，那么就需要有一整套纷繁的算法去执行回收逻辑。Android中各个进程的生死，和运行在其中的各个组件有着密切的联系，进程们依照其上组件的特点，被排入一个优先级体系，在需要回收时，从低优先级到高优先级回收。Android进程共分为五类优先级，分别是：Foreground Process, Visible Process, Service Process, Background Process, Empty Process。顾名思义不难看出，这说明，越和用户操作紧密相连的，越是正与用户交互的，优先级越高，越难被回收。----------

线程

读取数据，后台处理，自然少不了线程的参与。在Android核心的调度层面，是不屑于考量线程的，它关注的只有进程，每一个组件的构造和处理，都是在进程的主线程上做的，这样可以保证逻辑的足够简单。多线程，往往都是开发人员需要做的。

Android的线程，也是通过派生Java的Thread对象，实现Run方法来实现的。但当用户需要跑一个具有消息循环的线程的时候，Android有更好的支持，来自于Handler和Looper。Handler做的是消息的传送和分发，派生其handleMessage函数，可以处理各种收到的消息，和win开发无异。Looper的任务，则是构造循环，等候退出或其他消息的来临。在Looper的SDK页面，有一个消息循环线程实现的标准范例，当然，更为标准的方式也许是构造一个HandlerThread线程，将它的Looper传递给Handler。

在Android中，Content Provider的使用，往往和线程挂钩，谁让它和数据相关呢。在前面提到过，Content Provider为了保持更多的灵活性，本身只提供了同步调用的接口，而由于异步对Content Provider进行增删改查是一个常做操作，Android通过AsyncQueryHandler对象，提供了异步接口。这是一个Handler的子类，开发人员可以调用startXXX方法发起操作，通过派生onXXXComplete方法，等待执行完毕后的回调，从而完成整个异步调用的流程，十分的简约明了。 ProcessRecord，是整个进程托管实现的核心，它存放有运行在这个进程上，所有组件的信息，根据这些信息，系统有一整套的算法来决议如何处置这个进程，如果对回收算法感兴趣，可以从ActivityManagerService的trimApplications函数入手来看。

对于开发者来说，去了解这部分实现，主要是可以帮助理解整个进程和任务的概念，如果觉得这块理解的清晰了，就不用去碰ActivityManagerService这个庞然大物了。