|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 什么叫Serialization  Serialization是指把类或者基本的数据类型持久化(persistence)到数据流(Stream)中，包括文件、字节流、网络数据流， JAVA中实现serialization主要靠两个类：ObjectOuputStream和ObjectInputStream。他们是JAVA IO系统里的OutputStream和InputStream的子类。 双锁机制 一、饿汉模式  饿汉模式代码     1. /\*\* 2. \* @ClassName: Singleton\_Simple 3. \* @Description: 单例模式——饿汉模式 4. \* @author Ran 5. \* @date 2011-2-4 上午12:46:15 6. \* 7. \*/ 8. public class SpUtils { 10. private static final SpUtils simple = new SpUtils (); 12. private SpUtils (){} 14. public static SpUtils getInstance(){ 15. return simple; 16. } 18. }       说明：顾名思义，饿汉模式就是在jvm进程启动并在我们主动使用该类的时候就会在内存中初始化一个单例对象，当我们调用getInstance()的时候直接获取该对象，他的创建是在我们调用getInstance()静态方法之前！    二、懒汉模式  懒汉模式代码     1. /\*\* 2. \* @ClassName: Singleton\_lazy 3. \* @Description: 单例模式——懒汉模式 4. \* @author Ran 5. \* @date 2011-2-4 上午12:48:41 6. \* 7. \*/ 8. public class Singleton\_lazy { 10. private static Singleton\_lazy lazy = null; 12. private Singleton\_lazy(){} 14. public static synchronized Singleton\_lazy getInstance(){ 15. if( lazy == null ){ 16. lazy = new Singleton\_lazy(); 17. } 18. return lazy; 19. } 20. }       说明：懒汉模式是相对于饿汉模式而言的，在jvm进程启动并在我们主动使用该类的时候不会在内存中初始化一个单例对象，只有当我们调用getInstance()的时候才去创建该对象，他的创建是在我们调用getInstance()静态方法之后，为了并没现象同步问题，我们在getInstance()方法上加了一个锁，这个方法每次只允许一个线程进来，虽然同步问题是解决了，但是相应的性能问题就出现了。    三、双锁机制    双锁机制 代码     1. /\*\* 2. \* @ClassName: Singleton\_DoubleKey 3. \* @Description: 单例模式——双锁机制 4. \* @author Ran 5. \* @date 2011-2-4 上午12:53:50 6. \* 7. \*/ 8. public class SpUtils { 10. private static SpUtils doubleKey = null; 12. private SpUtils (){} 14. public static SpUtils getInstance(){ 15. if( doubleKey == null ){    //① 16. synchronized(SpUtils.class){    //② 17. if( doubleKey == null ){    //③ 18. doubleKey = new SpUtils ();  //④ 19. } 20. } 21. } 22. return doubleKey; 23. } 24. }        说明：双锁机制的出现是为了解决前面同步问题和性能问题，看上面的代码，简单分析下确实是解决了多线程并行进来不会出现重复new对象，而且也实现了懒加载，但是当我们静下来并结合java虚拟机的类加载过程我们就会发现问题出来了，对于JVM加载类过程不熟悉的，这里我简单介绍下，熟悉的跳过这段(当然，既然你熟悉就自然会知道双锁的弊端了)。  jvm加载一个类大体分为三个步骤：   1. 加载阶段：就是在硬盘上寻找java文件对应的class文件，并将class文件中的二进制数据加载到内存中，将其放在运行期数据区的方法区中去，然后在堆区创建一个java.lang.Class对象，用来封装在方法区内的数据结构； 2. 连接阶段：这个阶段分为三个步骤，步骤一：验证，验证什么呢？当然是验证这个class文件里面的二进制数据是否符合java规范咯；步骤二：准备，为该类的静态变量分配内存空间，并将变量赋一个默认值，比如int的默认值为0；步骤三：解析，这个阶段就不好解释了，将符号引用转化为直接引用，涉及到指针，这里不做多的解释； 3. 初始化阶段：当我们主动调用该类的时候，将该类的变量赋于正确的值(这里不要和第二阶段的准备混淆了)，举个例子说明下两个区别，比如一个类里有private static int i = 5; 这个静态变量在"准备"阶段会被分配一个内存空间并且被赋予一个默认值0，当道到初始化阶段的时候会将这个变量赋予正确的值即5，了解了吧！   好了，上面大体介绍了jvm类加载过程，回到我们的双锁机制上来分析下问题出在了哪里？假如有两个并发线程a、b，a线程主动调用了静态方法getInstance()，这时开始加载和初始化该类的静态变量，b线程调用getInstance()并等待获得同步锁，当a线程初始化对象过程中，到了第二阶段即连接阶段的准备步骤时，静态变量doubleKey 被赋予了一个默认值，但是这时还没有进行初始化，这时当a线程释放锁后，b线程判断doubleKey ！= null，则直接返回了一个没有初始化的doubleKey 对象，问题就出现在这里了，b线程拿到的是一个被赋予了默认值但是未初始化的对象，刚刚可以通过锁的检索！    所以对于上面的几个模式还是推荐使用第一种，在jvm加载类的时候就初始化一个对象，也避免了同步问题。 Service的高级用法 Service的基本用法  关于Service最基本的用法自然就是如何启动一个Service了，启动Service的方法和启动Activity很类似，都需要借助Intent来实现，下面我们就通过一个具体的例子来看一下。  新建一个Android项目，项目名就叫ServiceTest，这里我选择使用4.0的API。  然后新建一个MyService继承自Service，并重写父类的onCreate()、onStartCommand()和onDestroy()方法，如下所示：  [java] view plain copy   1. public class MyService extends Service { 3. public static final String TAG = "MyService"; 5. @Override 6. public void onCreate() { 7. super.onCreate(); 8. Log.d(TAG, "onCreate() executed"); 9. } 11. @Override 12. public int onStartCommand(Intent intent, int flags, int startId) { 13. Log.d(TAG, "onStartCommand() executed"); 14. return super.onStartCommand(intent, flags, startId); 15. } 17. @Override 18. public void onDestroy() { 19. super.onDestroy(); 20. Log.d(TAG, "onDestroy() executed"); 21. } 23. @Override 24. public IBinder onBind(Intent intent) { 25. return null; 26. } 28. }   在onCreate()、onStartCommand()和onDestroy()方法中分别编码log打印的话  然后打开或新建MainActivity作为程序的主Activity，在里面加入启动Service和停止Service的逻辑，  调用startService()方法来启动MyService。  调用stopService()方法来停止MyService。  运行起来，并点击一下Start Service按钮，可以看到LogCat的打印日志如下：    再点击一次Start Service按钮呢？这个时候的打印日志如下：    这次只有onStartCommand()方法执行了，onCreate()方法并没有执行，为什么会这样呢？这是由于onCreate()方法只会在Service第一次被创建的时候调用，如果当前Service已经被创建过了，不管怎样调用startService()方法，onCreate()方法都不会再执行。因此你可以再多点击几次Start Service按钮试一次，每次都只会有onStartCommand()方法中的打印日志。  我们还可以到手机的应用程序管理界面来检查一下MyService是不是正在运行，如下图所示：    恩，MyService确实是正在运行的，即使它的内部并没有执行任何的逻辑。  回到ServiceTest程序，然后点击一下Stop Service按钮就可以将MyService停止掉了。  Service和Activity通信  上面我们学习了Service的基本用法，启动Service之后，就可以在onCreate()或onStartCommand()方法里去执行一些具体的逻辑了。不过这样的话Service和Activity的关系并不大，只是Activity通知了Service一下：“你可以启动了。”然后Service就去忙自己的事情了。那么有没有什么办法能让它们俩的关联更多一些呢？比如说在Activity中可以指定让Service去执行什么任务。当然可以，只需要让Activity和Service建立关联就好了。  观察MyService中的代码，你会发现一直有一个onBind()方法我们都没有使用到，这个方法其实就是用于和Activity建立关联的，修改MyService中的代码，如下所示：  [java] view plain copy   1. public class MyService extends Service { 3. public static final String TAG = "MyService"; 5. private MyBinder mBinder = new MyBinder(); 7. @Override 8. public void onCreate() { 9. super.onCreate(); 10. Log.d(TAG, "onCreate() executed"); 11. } 13. @Override 14. public int onStartCommand(Intent intent, int flags, int startId) { 15. Log.d(TAG, "onStartCommand() executed"); 16. return super.onStartCommand(intent, flags, startId); 17. } 19. @Override 20. public void onDestroy() { 21. super.onDestroy(); 22. Log.d(TAG, "onDestroy() executed"); 23. } 25. @Override 26. public IBinder onBind(Intent intent) { 27. return mBinder; 28. } 30. class MyBinder extends Binder { 32. public void startDownload() { 33. Log.d("TAG", "startDownload() executed"); 34. // 执行具体的下载任务 35. } 37. } 39. }   这里我们新增了一个MyBinder类继承自Binder类，然后在MyBinder中添加了一个startDownload()方法用于在后台执行下载任务，当然这里并不是真正地去下载某个东西，只是做个[测试](http://lib.csdn.net/base/softwaretest)，所以startDownload()方法只是打印了一行日志。  然后修改activity\_main.xml中的代码，在布局文件中添加用于绑定Service和取消绑定Service的按钮：   1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" 2. android:layout\_width="match\_parent" 3. android:layout\_height="match\_parent" 4. android:orientation="vertical" > 6. <Button 7. android:id="@+id/start\_service" 8. android:layout\_width="match\_parent" 9. android:layout\_height="wrap\_content" 10. android:text="Start Service" /> 12. <Button 13. android:id="@+id/stop\_service" 14. android:layout\_width="match\_parent" 15. android:layout\_height="wrap\_content" 16. android:text="Stop Service" /> 18. <Button 19. android:id="@+id/bind\_service" 20. android:layout\_width="match\_parent" 21. android:layout\_height="wrap\_content" 22. android:text="Bind Service" /> 24. <Button 25. android:id="@+id/unbind\_service" 26. android:layout\_width="match\_parent" 27. android:layout\_height="wrap\_content" 28. android:text="Unbind Service" 29. /> 31. </LinearLayout>   接下来再修改MainActivity中的代码，让MainActivity和MyService之间建立关联，代码如下所示：   1. public class MainActivity extends Activity implements OnClickListener { 2. private Button startService; 3. private Button stopService; 4. private Button bindService; 5. private Button unbindService; 7. private MyService.MyBinder myBinder; 9. private ServiceConnection connection = new ServiceConnection() { 11. @Override 12. public void onServiceDisconnected(ComponentName name) { 13. } 15. @Override 16. public void onServiceConnected(ComponentName name, IBinder service) { 17. myBinder = (MyService.MyBinder) service; 18. myBinder.startDownload(); 19. } 20. }; 22. @Override 23. protected void onCreate(Bundle savedInstanceState) { 24. super.onCreate(savedInstanceState); 25. setContentView(R.layout.activity\_main); 26. startService = (Button) findViewById(R.id.start\_service); 27. stopService = (Button) findViewById(R.id.stop\_service); 28. bindService = (Button) findViewById(R.id.bind\_service); 29. unbindService = (Button) findViewById(R.id.unbind\_service); 30. startService.setOnClickListener(this); 31. stopService.setOnClickListener(this); 32. bindService.setOnClickListener(this); 33. unbindService.setOnClickListener(this); 34. } 36. @Override 37. public void onClick(View v) { 38. switch (v.getId()) { 39. case R.id.start\_service: 40. Intent startIntent = new Intent(this, MyService.class); 41. startService(startIntent); 42. break; 43. case R.id.stop\_service: 44. Intent stopIntent = new Intent(this, MyService.class); 45. stopService(stopIntent); 46. break; 47. case R.id.bind\_service: 48. Intent bindIntent = new Intent(this, MyService.class); 49. bindService(bindIntent, connection, BIND\_AUTO\_CREATE); 50. break; 51. case R.id.unbind\_service: 52. unbindService(connection); 53. break; 54. default: 55. break; 56. } 57. } 59. }   可以看到，这里我们首先创建了一个ServiceConnection的匿名类，在里面重写了onServiceConnected()方法和onServiceDisconnected()方法，这两个方法分别会在Activity与Service建立关联和解除关联的时候调用。在onServiceConnected()方法中，我们又通过向下转型得到了MyBinder的实例，有了这个实例，Activity和Service之间的关系就变得非常紧密了。现在我们可以在Activity中根据具体的场景来调用MyBinder中的任何public方法，即实现了Activity指挥Service干什么Service就去干什么的功能。  当然，现在Activity和Service其实还没关联起来了呢，这个功能是在Bind Service按钮的点击事件里完成的。可以看到，这里我们仍然是构建出了一个Intent对象，然后调用bindService()方法将Activity和Service进行绑定。bindService()方法接收三个参数，第一个参数就是刚刚构建出的Intent对象，第二个参数是前面创建出的ServiceConnection的实例，第三个参数是一个标志位，这里传入BIND\_AUTO\_CREATE表示在Activity和Service建立关联后自动创建Service，这会使得MyService中的onCreate()方法得到执行，但onStartCommand()方法不会执行。  然后如何我们想解除Activity和Service之间的关联怎么办呢？调用一下unbindService()方法就可以了，这也是Unbind Service按钮的点击事件里实现的逻辑。  现在让我们重新运行一下程序吧，在MainActivity中点击一下Bind Service按钮，LogCat里的打印日志如下图所示：    另外需要注意，任何一个Service在整个应用程序范围内都是通用的，即MyService不仅可以和MainActivity建立关联，还可以和任何一个Activity建立关联，而且在建立关联时它们都可以获取到相同的MyBinder实例。  如何销毁Service  在Service的基本用法这一部分，我们介绍了销毁Service最简单的一种情况，点击Start Service按钮启动Service，再点击Stop Service按钮停止Service，这样MyService就被销毁了，可以看到打印日志如下所示：    那么如果我们是点击的Bind Service按钮呢？由于在绑定Service的时候指定的标志位是BIND\_AUTO\_CREATE，说明点击Bind Service按钮的时候Service也会被创建，这时应该怎么销毁Service呢？其实也很简单，点击一下Unbind Service按钮，将Activity和Service的关联解除就可以了。  先点击一下Bind Service按钮，再点击一下Unbind Service按钮，打印日志如下所示：    以上这两种销毁的方式都很好理解。那么如果我们既点击了Start Service按钮，又点击了Bind Service按钮会怎么样呢？这个时候你会发现，不管你是单独点击Stop Service按钮还是Unbind Service按钮，Service都不会被销毁，必要将两个按钮都点击一下，Service才会被销毁。也就是说，点击Stop Service按钮只会让Service停止，点击Unbind Service按钮只会让Service和Activity解除关联，一个Service必须要在既没有和任何Activity关联又处理停止状态的时候才会被销毁。  为了证实一下，我们在Stop Service和Unbind Service按钮的点击事件里面加入一行打印日志：  [java] view plain copy   1. public void onClick(View v) { 2. switch (v.getId()) { 3. case R.id.start\_service: 4. Intent startIntent = new Intent(this, MyService.class); 5. startService(startIntent); 6. break; 7. case R.id.stop\_service: 8. Log.d("MyService", "click Stop Service button"); 9. Intent stopIntent = new Intent(this, MyService.class); 10. stopService(stopIntent); 11. break; 12. case R.id.bind\_service: 13. Intent bindIntent = new Intent(this, MyService.class); 14. bindService(bindIntent, connection, BIND\_AUTO\_CREATE); 15. break; 16. case R.id.unbind\_service: 17. Log.d("MyService", "click Unbind Service button"); 18. unbindService(connection); 19. break; 20. default: 21. break; 22. } 23. }   然后重新运行程序，先点击一下Start Service按钮，再点击一下Bind Service按钮，这样就将Service启动起来，并和Activity建立了关联。然后点击Stop Service按钮后Service并不会销毁，再点击一下Unbind Service按钮，Service就会销毁了，打印日志如下所示：    我们应该始终记得在Service的onDestroy()方法里去清理掉那些不再使用的资源，防止在Service被销毁后还会有一些不再使用的对象仍占用着内存。  Service和Thread的关系  不少Android初学者都可能会有这样的疑惑，Service和Thread到底有什么关系呢？什么时候应该用Service，什么时候又应该用Thread？答案可能会有点让你吃惊，因为Service和Thread之间没有任何关系！  之所以有不少人会把它们联系起来，主要就是因为Service的后台概念。Thread我们大家都知道，是用于开启一个子线程，在这里去执行一些耗时操作就不会阻塞主线程的运行。而Service我们最初理解的时候，总会觉得它是用来处理一些后台任务的，一些比较耗时的操作也可以放在这里运行，这就会让人产生混淆了。但是，如果我告诉你Service其实是运行在主线程里的，你还会觉得它和Thread有什么关系吗？让我们看一下这个残酷的事实吧。  在MainActivity的onCreate()方法里加入一行打印当前线程id的语句：  [java] view plain copy   1. Log.d("MyService", "MainActivity thread id is " + Thread.currentThread().getId());   然后在MyService的onCreate()方法里也加入一行打印当前线程id的语句：  [java] view plain copy   1. Log.d("MyService", "MyService thread id is " + Thread.currentThread().getId());   现在重新运行一下程序，并点击Start Service按钮，会看到如下打印日志：    可以看到，它们的线程id完全是一样的，由此证实了Service确实是运行在主线程里的，也就是说如果你在Service里编写了非常耗时的代码，程序必定会出现ANR的。  你可能会惊呼，这不是坑爹么！？那我要Service又有何用呢？其实大家不要把后台和子线程联系在一起就行了，这是两个完全不同的概念。Android的后台就是指，它的运行是完全不依赖UI的。即使Activity被销毁，或者程序被关闭，只要进程还在，Service就可以继续运行。比如说一些应用程序，始终需要与服务器之间始终保持着心跳连接，就可以使用Service来实现。你可能又会问，前面不是刚刚验证过Service是运行在主线程里的么？在这里一直执行着心跳连接，难道就不会阻塞主线程的运行吗？当然会，但是我们可以在Service中再创建一个子线程，然后在这里去处理耗时逻辑就没问题了。  额，既然在Service里也要创建一个子线程，那为什么不直接在Activity里创建呢？这是   1. 从获取子线程实例的方面考虑：因为Activity很难对Thread进行控制，当Activity被销毁之后，就没有任何其它的办法可以再重新获取到之前创建的子线程的实例。 2. 从一个或者多个activity对子线程操作上：而且在一个Activity中创建的子线程，另一个Activity无法对其进行操作。 3. 但是Service就不同了，所有的Activity都可以与Service进行关联，然后可以很方便地操作其中的方法，即使Activity被销毁了，之后只要重新与Service建立关联，就又能够获取到原有的Service中Binder的实例。因此，使用Service来处理后台任务，Activity就可以放心地finish，完全不需要担心无法对后台任务进行控制的情况。   一个比较标准的Service就可以写成：  [java] view plain copy   1. @Override 2. public int onStartCommand(Intent intent, int flags, int startId) { 3. new Thread(new Runnable() { 4. @Override 5. public void run() { 6. // 开始执行后台任务 7. } 8. }).start(); 9. return super.onStartCommand(intent, flags, startId); 10. } 12. class MyBinder extends Binder { 14. public void startDownload() { 15. new Thread(new Runnable() { 16. @Override 17. public void run() { 18. // 执行具体的下载任务 19. } 20. }).start(); 21. } 23. }  创建前台Service Service几乎都是在后台运行的，一直以来它都是默默地做着辛苦的工作。但是Service的系统优先级还是比较低的，当系统出现内存不足情况时，就有可能会回收掉正在后台运行的Service。如果你希望Service可以一直保持运行状态，而不会由于系统内存不足的原因导致被回收，就可以考虑使用前台Service。前台Service和普通Service最大的区别就在于，它会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，非常类似于通知的效果。当然有时候你也可能不仅仅是为了防止Service被回收才使用前台Service，有些项目由于特殊的需求会要求必须使用前台Service，比如说墨迹天气，它的Service在后台更新天气数据的同时，还会在系统状态栏一直显示当前天气的信息，如下图所示：    那么我们就来看一下如何才能创建一个前台Service吧，其实并不复杂，修改MyService中的代码，如下所示：  [java] view plain copy   1. public class MyService extends Service { 3. public static final String TAG = "MyService"; 5. private MyBinder mBinder = new MyBinder(); 7. @Override 8. public void onCreate() { 9. super.onCreate(); 10. Notification notification = new Notification(R.drawable.ic\_launcher, 11. "有通知到来", System.currentTimeMillis()); 12. Intent notificationIntent = new Intent(this, MainActivity.class); 13. PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, 14. notificationIntent, 0); 15. notification.setLatestEventInfo(this, "这是通知的标题", "这是通知的内容", 16. pendingIntent); 17. startForeground(1, notification); 18. Log.d(TAG, "onCreate() executed"); 19. } 21. ......... 23. }   这里只是修改了MyService中onCreate()方法的代码。可以看到，我们首先创建了一个Notification对象，然后调用了它的setLatestEventInfo()方法来为通知初始化布局和数据，并在这里设置了点击通知后就打开MainActivity。然后调用startForeground()方法就可以让MyService变成一个前台Service，并会将通知的图片显示出来。  现在重新运行一下程序，并点击Start Service或Bind Service按钮，MyService就会以前台Service的模式启动了，并且在系统状态栏会弹出一个通栏图标，下拉状态栏后可以看到通知的详细内容，如下图所示。    好了，由于篇幅的原因，本篇文章就先写到这里。目前我们已经把关于Service的很多重要知识点都梳理完了，下一篇文章会承接这篇文章，介绍Android Service中剩下的一个非常重要且复杂的知识点 —— 远程Service的使用，感兴趣的朋友请继续阅读 [Android Service完全解析，关于服务你所需知道的一切（下）](http://blog.csdn.net/guolin_blog/article/details/9797169) 。  在上篇文章中我们知道了，Service其实是运行在主线程里的，如果直接在Service中处理一些耗时的逻辑，就会导致程序ANR。  让我们来做个实验验证一下吧，修改上一篇文章中创建的ServiceTest项目，在MyService的onCreate()方法中让线程睡眠60秒，如下所示：  [java] view plain copy   1. public class MyService extends Service { 2. ...... 3. @Override 4. public void onCreate() { 5. super.onCreate(); 6. Log.d(TAG, "onCreate() executed"); 7. try { 8. Thread.sleep(60000); 9. } catch (InterruptedException e) { 10. e.printStackTrace(); 11. } 12. } 14. ...... 16. }   重新运行后，点击一下Start Service按钮或Bind Service按钮，程序就会阻塞住并无法进行任何其它操作，过一段时间后就会弹出ANR的提示框，如下图所示。    之前我们提到过，应该在Service中开启线程去执行耗时任务，这样就可以有效地避免ANR的出现。  那么本篇文章的主题是介绍远程Service的用法，如果将MyService转换成一个远程Service，还会不会有ANR的情况呢？让我们来动手尝试一下吧。  将一个普通的Service转换成远程Service其实非常简单，只需要在注册Service的时候将它的android:process属性指定成:remote就可以了，代码如下所示：  [html] view plain copy   1. <?xml version="1.0" encoding="utf-8"?> 2. <manifest xmlns:android="http://schemas.android.com/apk/res/android" 3. package="com.example.servicetest" 4. android:versionCode="1" 5. android:versionName="1.0" > 7. ...... 9. <service 10. android:name="com.example.servicetest.MyService" 11. android:process=":remote" > 12. </service> 14. </manifest>   现在重新运行程序，并点击一下Start Service按钮，你会看到控制台立刻打印了onCreate() executed的信息，而且主界面并没有阻塞住，也不会出现ANR。大概过了一分钟后，又会看到onStartCommand() executed打印了出来。  为什么将MyService转换成远程Service后就不会导致程序ANR了呢？这是由于，使用了远程Service后，MyService已经在另外一个进程当中运行了，所以只会阻塞该进程中的主线程，并不会影响到当前的应用程序。  为了证实一下MyService现在确实已经运行在另外一个进程当中了，我们分别在MainActivity的onCreate()方法和MyService的onCreate()方法里加入一行日志，打印出各自所在的进程id，如下所示：  [java] view plain copy   1. Log.d("TAG", "process id is " + Process.myPid());   再次重新运行程序，然后点击一下Start Service按钮，打印结果如下图所示：    可以看到，不仅仅是进程id不同了，就连应用程序包名也不一样了，MyService中打印的那条日志，包名后面还跟上了:remote标识。  那既然远程Service这么好用，干脆以后我们把所有的Service都转换成远程Service吧，还省得再开启线程了。其实不然，远程Service非但不好用，甚至可以称得上是较为难用。一般情况下如果可以不使用远程Service，就尽量不要使用它。  下面就来看一下远程Service的弊端吧，首先将MyService的onCreate()方法中让线程睡眠的代码去除掉，然后重新运行程序，并点击一下Bind Service按钮，你会发现程序崩溃了！为什么点击Start Service按钮程序就不会崩溃，而点击Bind Service按钮就会崩溃呢？这是由于在Bind Service按钮的点击事件里面我们会让MainActivity和MyService建立关联，但是目前MyService已经是一个远程Service了，Activity和Service运行在两个不同的进程当中，这时就不能再使用传统的建立关联的方式，程序也就崩溃了。  那么如何才能让Activity与一个远程Service建立关联呢？这就要使用AIDL来进行跨进程通信了（IPC）。  AIDL（Android Interface Definition Language）是Android接口定义语言的意思，它可以用于让某个Service与多个应用程序组件之间进行跨进程通信，从而可以实现多个应用程序共享同一个Service的功能。  下面我们就来一步步地看一下AIDL的用法到底是怎样的。首先需要新建一个AIDL文件，在这个文件中定义好Activity需要与Service进行通信的方法。新建MyAIDLService.aidl文件，代码如下所示：  [java] view plain copy   1. package com.example.servicetest; 2. interface MyAIDLService { 3. int plus(int a, int b); 4. String toUpperCase(String str); 5. }   点击保存之后，gen目录下就会生成一个对应的[Java](http://lib.csdn.net/base/java)文件，如下图所示：    然后修改MyService中的代码，在里面实现我们刚刚定义好的MyAIDLService接口，如下所示：  [java] view plain copy   1. public class MyService extends Service { 3. ...... 5. @Override 6. public IBinder onBind(Intent intent) { 7. return mBinder; 8. } 10. MyAIDLService.Stub mBinder = new Stub() { 12. @Override 13. public String toUpperCase(String str) throws RemoteException { 14. if (str != null) { 15. return str.toUpperCase(); 16. } 17. return null; 18. } 20. @Override 21. public int plus(int a, int b) throws RemoteException { 22. return a + b; 23. } 24. }; 26. }   这里先是对MyAIDLService.Stub进行了实现，重写里了toUpperCase()和plus()这两个方法。这两个方法的作用分别是将一个字符串全部转换成大写格式，以及将两个传入的整数进行相加。然后在onBind()方法中将MyAIDLService.Stub的实现返回。这里为什么可以这样写呢？因为Stub其实就是Binder的子类，所以在onBind()方法中可以直接返回Stub的实现。  接下来修改MainActivity中的代码，如下所示：  [java] view plain copy   1. public class MainActivity extends Activity implements OnClickListener { 3. private Button startService; 5. private Button stopService; 7. private Button bindService; 9. private Button unbindService; 11. private MyAIDLService myAIDLService; 13. private ServiceConnection connection = new ServiceConnection() { 15. @Override 16. public void onServiceDisconnected(ComponentName name) { 17. } 19. @Override 20. public void onServiceConnected(ComponentName name, IBinder service) { 21. myAIDLService = MyAIDLService.Stub.asInterface(service); 22. try { 23. int result = myAIDLService.plus(3, 5); 24. String upperStr = myAIDLService.toUpperCase("hello world"); 25. Log.d("TAG", "result is " + result); 26. Log.d("TAG", "upperStr is " + upperStr); 27. } catch (RemoteException e) { 28. e.printStackTrace(); 29. } 30. } 31. }; 33. ...... 35. }   我们只是修改了ServiceConnection中的代码。可以看到，这里首先使用了MyAIDLService.Stub.asInterface()方法将传入的IBinder对象传换成了MyAIDLService对象，接下来就可以调用在MyAIDLService.aidl文件中定义的所有接口了。这里我们先是调用了plus()方法，并传入了3和5作为参数，然后又调用了toUpperCase()方法，并传入hello world字符串作为参数，最后将调用方法的返回结果打印出来。  现在重新运行程序，并点击一下Bind Service按钮，可以看到打印日志如下所示：    由此可见，我们确实已经成功实现跨进程通信了，在一个进程中访问到了另外一个进程中的方法。  不过你也可以看出，目前的跨进程通信其实并没有什么实质上的作用，因为这只是在一个Activity里调用了同一个应用程序的Service里的方法。而跨进程通信的真正意义是为了让一个应用程序去访问另一个应用程序中的Service，以实现共享Service的功能。那么下面我们自然要学习一下，如何才能在其它的应用程序中调用到MyService里的方法。  在上一篇文章中我们已经知道，如果想要让Activity与Service之间建立关联，需要调用bindService()方法，并将Intent作为参数传递进去，在Intent里指定好要绑定的Service，示例代码如下：  [java] view plain copy   1. Intent bindIntent = new Intent(this, MyService.class); 2. bindService(bindIntent, connection, BIND\_AUTO\_CREATE);   这里在构建Intent的时候是使用MyService.class来指定要绑定哪一个Service的，但是在另一个应用程序中去绑定Service的时候并没有MyService这个类，这时就必须使用到隐式Intent了。现在修改AndroidManifest.xml中的代码，给MyService加上一个action，如下所示：  [html] view plain copy   1. <?xml version="1.0" encoding="utf-8"?> 2. <manifest xmlns:android="http://schemas.android.com/apk/res/android" 3. package="com.example.servicetest" 4. android:versionCode="1" 5. android:versionName="1.0" > 7. ...... 9. <service 10. android:name="com.example.servicetest.MyService" 11. android:process=":remote" > 12. <intent-filter> 13. <action android:name="com.example.servicetest.MyAIDLService"/> 14. </intent-filter> 15. </service> 17. </manifest>   这就说明，MyService可以响应带有com.example.servicetest.MyAIDLService这个action的Intent。  现在重新运行一下程序，这样就把远程Service端的工作全部完成了。  然后创建一个新的Android项目，起名为ClientTest，我们就尝试在这个程序中远程调用MyService中的方法。  ClientTest中的Activity如果想要和MyService建立关联其实也不难，首先需要将MyAIDLService.aidl文件从ServiceTest项目中拷贝过来，注意要将原有的包路径一起拷贝过来，完成后项目的结构如下图所示：    然后打开或新建activity\_main.xml，在布局文件中也加入一个Bind Service按钮：  [html] view plain copy   1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" 2. android:layout\_width="match\_parent" 3. android:layout\_height="match\_parent" 4. android:orientation="vertical" 5. > 7. <Button 8. android:id="@+id/bind\_service" 9. android:layout\_width="match\_parent" 10. android:layout\_height="wrap\_content" 11. android:text="Bind Service" 12. /> 14. </LinearLayout>   接下来打开或新建MainActivity，在其中加入和MyService建立关联的代码，如下所示：  [java] view plain copy   1. public class MainActivity extends Activity { 3. private MyAIDLService myAIDLService; 5. private ServiceConnection connection = new ServiceConnection() { 7. @Override 8. public void onServiceDisconnected(ComponentName name) { 9. } 11. @Override 12. public void onServiceConnected(ComponentName name, IBinder service) { 13. myAIDLService = MyAIDLService.Stub.asInterface(service); 14. try { 15. int result = myAIDLService.plus(50, 50); 16. String upperStr = myAIDLService.toUpperCase("comes from ClientTest"); 17. Log.d("TAG", "result is " + result); 18. Log.d("TAG", "upperStr is " + upperStr); 19. } catch (RemoteException e) { 20. e.printStackTrace(); 21. } 22. } 23. }; 25. @Override 26. protected void onCreate(Bundle savedInstanceState) { 27. super.onCreate(savedInstanceState); 28. setContentView(R.layout.activity\_main); 29. Button bindService = (Button) findViewById(R.id.bind\_service); 30. bindService.setOnClickListener(new OnClickListener() { 31. @Override 32. public void onClick(View v) { 33. Intent intent = new Intent("com.example.servicetest.MyAIDLService"); 34. bindService(intent, connection, BIND\_AUTO\_CREATE); 35. } 36. }); 37. } 39. }   这部分代码大家一定会非常眼熟吧？没错，这和在ServiceTest的MainActivity中的代码几乎是完全相同的，只是在让Activity和Service建立关联的时候我们使用了隐式Intent，将Intent的action指定成了com.example.servicetest.MyAIDLService。  在当前Activity和MyService建立关联之后，我们仍然是调用了plus()和toUpperCase()这两个方法，远程的MyService会对传入的参数进行处理并返回结果，然后将结果打印出来。  这样的话，ClientTest中的代码也就全部完成了，现在运行一下这个项目，然后点击Bind Service按钮，此时就会去和远程的MyService建立关联，观察LogCat中的打印信息如下所示：    不用我说，大家都已经看出，我们的跨进程通信功能已经完美实现了。  不过还有一点需要说明的是，由于这是在不同的进程之间传递数据，Android对这类数据的格式支持是非常有限的，基本上只能传递Java的基本数据类型、字符串、List或Map等。那么如果我想传递一个自定义的类该怎么办呢？这就必须要让这个类去实现Parcelable接口，并且要给这个类也定义一个同名的AIDL文件。这部分内容并不复杂，而且和Service关系不大，所以就不再详细进行讲解了，感兴趣的朋友可以自己去查阅一下相关的资料。  好了，结合上下两篇，这就是关于Service你所需知道的一切。 List、map、set 的区别 java集合的主要分为三种类型：   * Set（集） * List（列表） * Map（映射）   要深入理解集合首先要了解下我们熟悉的数组：  数组是大小固定的，并且同一个数组只能存放类型一样的数据（基本类型/引用类型），而JAVA集合可以存储和操作数目不固定的一组数据。 所有的JAVA集合都位于 java.util包中！ JAVA集合只能存放引用类型的的数据，不能存放基本数据类型。  简单说下集合和数组的区别：(参考文章：[《Thinking In Algorithm》03.数据结构之数组](http://blog.csdn.net/speedme/article/details/18180817))   1. <span style="font-family:Microsoft YaHei;font-size:12px;">世间上本来没有集合,(只有数组参考C语言)但有人想要,所以有了集合 2. 有人想有可以自动扩展的数组,所以有了List 3. 有的人想有没有重复的数组,所以有了set 4. 有人想有自动排序的组数,所以有了TreeSet,TreeList,Tree\*\* 6. 而几乎有有的集合都是基于数组来实现的. 7. 因为集合是对数组做的封装,所以,数组永远比任何一个集合要快 8. 但任何一个集合,比数组提供的功能要多 9. 一：数组声明了它容纳的元素的类型，而集合不声明。这是由于集合以object形式来存储它们的元素。 10. 二：一个数组实例具有固定的大小，不能伸缩。集合则可根据需要动态改变大小。 11. 三：数组是一种可读/可写数据结构－－－没有办法创建一个只读数组。然而可以使用集合提供的ReadOnly方法，以只读方式来使用集合。该方法将返回一个集合的只读版本。</span>   Java所有“存储及随机访问一连串对象”的做法，array是最有效率的一种。  1、 效率高，但容量固定且无法动态改变。 array还有一个缺点是，无法判断其中实际存有多少元素，length只是告诉我们array的容量。    2、Java中有一个Arrays类，专门用来操作array。      arrays中拥有一组static函数， equals()：比较两个array是否相等。array拥有相同元素个数，且所有对应元素两两相等。 fill()：将值填入array中。 sort()：用来对array进行排序。 binarySearch()：在排好序的array中寻找元素。 System.arraycopy()：array的复制。  若撰写程序时不知道究竟需要多少对象，需要在空间不足时自动扩增容量，则需要使用容器类库，array不适用。所以就要用到集合。  那我们开始讨论java中的集合。  集合分类：  Collection：List、Set Map：HashMap、HashTable 1.1 Collection接口 Collection是最基本的集合接口，声明了适用于JAVA集合（只包括Set和List）的通用方法。 Set 和List 都继承了Conllection,Map。   1.1.1  Collection接口的方法：  1. <span style="font-weight: normal;">boolean add(Object o)      ：向集合中加入一个对象的引用 3. void clear()：删除集合中所有的对象，即不再持有这些对象的引用 5. boolean isEmpty()    ：判断集合是否为空 7. boolean contains(Object o) ： 判断集合中是否持有特定对象的引用 9. Iterartor iterator()  ：返回一个Iterator对象，可以用来遍历集合中的元素 11. boolean remove(Object o) ：从集合中删除一个对象的引用 13. int size()       ：返回集合中元素的数目 15. Object[] toArray()    ： 返回一个数组，该数组中包括集合中的所有元素 </span>     关于：Iterator() 和toArray() 方法都用于集合的所有的元素，前者返回一个Iterator对象，后者返回一个包含集合中所有元素的数组。 1.1.2  Iterator接口声明了如下方法：  1. hasNext()：判断集合中元素是否遍历完毕，如果没有，就返回true 3. next() ：返回下一个元素 5. remove()：从集合中删除上一个有next()方法返回的元素。  1.2  Set(集合) Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 Set接口主要实现了两个实现类：   * HashSet： HashSet类按照哈希算法来存取集合中的对象，存取速度比较快 * TreeSet ：TreeSet类实现了SortedSet接口，能够对集合中的对象进行排序。   Set 的用法：存放的是对象的引用，没有重复对象   1. Set set=new HashSet(); 3. String s1=new String("hello"); 5. String s2=s1; 7. String s3=new String("world"); 9. set.add(s1); 11. set.add(s2); 13. set.add(s3); 15. System.out.println(set.size());//打印集合中对象的数目 为 2。   Set 的 add()方法是如何判断对象是否已经存放在集合中？   1. boolean isExists=false; 3. Iterator iterator=set.iterator(); 5. while(it.hasNext())           { 7. String oldStr=it.next(); 9. if(newStr.equals(oldStr)){ 11. isExists=true; 13. } 15. }   Set的功能方法  Set具有与Collection完全一样的接口，因此没有任何额外的功能，不像前面有两个不同的List。实际上Set就是Collection,只 是行为不同。(这是继承与多态思想的典型应用：表现不同的行为。)Set不保存重复的元素(至于如何判断元素相同则较为复杂)  Set : 存入Set的每个元素都必须是唯一的，因为Set不保存重复元素。加入Set的元素必须定义equals()方法以确保对象的唯一性。Set与Collection有完全一样的接口。Set接口不保证维护元素的次序。   * HashSet：为快速查找设计的Set。存入HashSet的对象必须定义hashCode()。 * TreeSet： 保存次序的Set, 底层为树结构。使用它可以从Set中提取有序的序列。   LinkedHashSet：具有HashSet的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历Set时，结果会按元素插入的次序显示。 1.3  List(列表) List的特征是其元素以线性方式存储，集合中可以存放重复对象。  List接口主要实现类包括：（参考文章：[ArrayList与LinkedList的区别](http://blog.csdn.net/speedme/article/details/8199418)）   * ArrayList() : 代表长度可以改变得数组。可以对元素进行随机的访问，向ArrayList()中插入与删除元素的速度慢。 * LinkedList(): 在实现中采用链表数据结构。插入和删除速度快，访问速度慢。   对于List的随机访问来说，就是只随机来检索位于特定位置的元素。 List 的 get(int index) 方法放回集合中由参数index指定的索引位置的对象，下标从“0” 开始。最基本的两种检索集合中的所有对象的方法：        1： for循环和get()方法：   1. for(int i=0; i<list.size();i++){ 3. System.out.println(list.get(i)); 5. }     2： 使用 迭代器（Iterator）:   1. Iterator it=list.iterator(); 3. while(it.hashNext()){ 5. System.out.println(it.next()); 7. }   List的功能方法  实际上有两种List：一种是基本的ArrayList,其优点在于随机访问元素，另一种是更强大的LinkedList,它并不是为快速随机访问设计的，而是具有一套更通用的方法。   * List：次序是List最重要的特点：它保证维护元素特定的顺序。List为Collection添加了许多方法，使得能够向List中间插入与移除元素(这只推 荐LinkedList使用。)一个List可以生成ListIterator,使用它可以从两个方向遍历List,也可以从List中间插入和移除元 素。 * ArrayList：由数组实现的List。允许对元素进行快速随机访问，但是向List中间插入与移除元素的速度很慢。ListIterator只应该用来由后向前遍历 ArrayList,而不是用来插入和移除元素。因为那比LinkedList开销要大很多。 * LinkedList ：对顺序访问进行了优化，向List中间插入与删除的开销并不大。随机访问则相对较慢。(使用ArrayList代替。)还具有下列方 法：addFirst(), addLast(), getFirst(), getLast(), removeFirst() 和 removeLast(), 这些方法 (没有在任何接口或基类中定义过)使得LinkedList可以当作堆栈、队列和双向队列使用。  1.4 Map(映射) Map 是一种把键对象和值对象映射的集合，它的每一个元素都包含一对键对象和值对象。 Map没有继承于Collection接口 从Map集合中检索元素时，只要给出键对象，就会返回对应的值对象。  Map 的常用方法：  1 添加，删除操作：   1. Object put(Object key, Object value)： 向集合中加入元素 2. Object remove(Object key)： 删除与KEY相关的元素 3. void putAll(Map t)：  将来自特定映像的所有元素添加给该映像 4. void clear()：从映像中删除所有映射   2 查询操作：  Object get(Object key)：获得与关键字key相关的值 。Map集合中的键对象不允许重复，也就说，任意两个键对象通过equals()方法比较的结果都是false.，但是可以将任意多个键独享映射到同一个值对象上。  Map的功能方法  方法put(Object key, Object value)添加一个“值”(想要得东西)和与“值”相关联的“键”(key)(使用它来查找)。方法get(Object key)返回与给定“键”相关联的“值”。可以用containsKey()和containsValue()测试Map中是否包含某个“键”或“值”。 标准的Java类库中包含了几种不同的Map：HashMap, TreeMap, LinkedHashMap, WeakHashMap, IdentityHashMap。它们都有同样的基本接口Map，但是行为、效率、排序策略、保存对象的生命周期和判定“键”等价的策略等各不相同。  执行效率是Map的一个大问题。看看get()要做哪些事，就会明白为什么在ArrayList中搜索“键”是相当慢的。而这正是HashMap提高速 度的地方。HashMap使用了特殊的值，称为“散列码”(hash code)，来取代对键的缓慢搜索。“散列码”是“相对唯一”用以代表对象的int值，它是通过将该对象的某些信息进行转换而生成的。所有Java对象都 能产生散列码，因为hashCode()是定义在基类Object中的方法。  HashMap就是使用对象的hashCode()进行快速查询的。此方法能够显着提高性能。  Map : 维护“键值对”的关联性，使你可以通过“键”查找“值”  HashMap：Map基于散列表的实现。插入和查询“键值对”的开销是固定的。可以通过构造器设置容量capacity和负载因子load factor，以调整容器的性能。  LinkedHashMap： 类似于HashMap，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用(LRU)的次序。只比HashMap慢一点。而在迭代访问时发而更快，因为它使用链表维护内部次序。  TreeMap ： 基于红黑树数据结构的实现。查看“键”或“键值对”时，它们会被排序(次序由Comparabel或Comparator决定)。TreeMap的特点在 于，你得到的结果是经过排序的。TreeMap是唯一的带有subMap()方法的Map，它可以返回一个子树。  WeakHashMao ：弱键(weak key)Map，Map中使用的对象也被允许释放: 这是为解决特殊问题设计的。如果没有map之外的引用指向某个“键”，则此“键”可以被垃圾收集器回收。  IdentifyHashMap： : 使用==代替equals()对“键”作比较的hash map。专为解决特殊问题而设计。 1.4 区别 1.4.1、Collection 和 Map 的区别  容器内每个为之所存储的元素个数不同。 Collection类型者，每个位置只有一个元素。 Map类型者，持有 key-value pair，像个小型数据库。  1.4.2、各自旗下的子类关系  Collection      --List：将以特定次序存储元素。所以取出来的顺序可能和放入顺序不同。            --ArrayList / LinkedList / Vector      --Set ： 不能含有重复的元素            --HashSet / TreeSet       Map      --HashMap      --HashTable      --TreeMap  1.4.3、其他特征  List，Set，Map将持有对象一律视为Object型别。  Collection、List、Set、Map都是接口，不能实例化。     继承自它们的 ArrayList, Vector, HashTable, HashMap是具象class，这些才可被实例化。 vector容器确切知道它所持有的对象隶属什么型别。vector不进行边界检查。  总结  1. 如果涉及到堆栈，队列等操作，应该考虑用List，对于需要快速插入，删除元素，应该使用LinkedList，如果需要快速随机访问元素，应该使用ArrayList。 2. 如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个类，应该使用同步的类。 3. 在除需要排序时使用TreeSet,TreeMap外,都应使用HashSet,HashMap,因为他们 的效率更高。 4. 要特别注意对哈希表的操作，作为key的对象要正确复写equals和hashCode方法。   5. 容器类仅能持有对象引用（指向对象的指针），而不是将对象信息copy一份至数列某位置。一旦将对象置入容器内，便损失了该对象的型别信息。 6. 尽量返回接口而非实际的类型，如返回List而非ArrayList，这样如果以后需要将ArrayList换成LinkedList时，客户端代码不用改变。这就是针对抽象编程。    注意：  1、Collection没有get()方法来取得某个元素。只能通过iterator()遍历元素。  2、Set和Collection拥有一模一样的接口。  3、List，可以通过get()方法来一次取出一个元素。使用数字来选择一堆对象中的一个，get(0)...。(add/get)  4、一般使用ArrayList。用LinkedList构造堆栈stack、队列queue。  5、Map用 put(k,v) / get(k)，还可以使用containsKey()/containsValue()来检查其中是否含有某个key/value。        HashMap会利用对象的hashCode来快速找到key。  6、Map中元素，可以将key序列、value序列单独抽取出来。  使用keySet()抽取key序列，将map中的所有keys生成一个Set。  使用values()抽取value序列，将map中的所有values生成一个Collection。  为什么一个生成Set，一个生成Collection？那是因为，key总是独一无二的，value允许重复。 JNI 实现 准备工作：下载NDK。  简单的说，要用到C/C++，就要用NDK。直接百度搜索然后去官网下载就行，位置可以随便放。    1、新建一个类，声明native方法。这个类是java与C/C++交互的中介，方法由java声明，由C/C++实现。  不在Activity类里面写是为了避免编译时报错：找不到android.support.v7.app.AppCompatActivity  public class myJNI {  　　//加载so库  static {  System.loadLibrary("JniTest");  }  　　//native方法  public static native String sayHello();  }  2、打开android studio终端，使用javac编译上述文件，生成class文件。      命令：  D:\AndroidStudioProject\testJNI\app\src\main\java\com\example\binbin\testjni>javac myJNI.java    3、确认自己类的包名！然后在上一级的位置使用javah生成.h头文件  一定要在上一级哦，不然不会报错：找不到xxx类    比如我的myJNI.class在  D:\AndroidStudioProject\testJNI\app\src\main\java\com\example\binbin\testjni  我的包名是：  package com.example.binbin.testjni;  所以我的javah命令应该这么写:  D:\AndroidStudioProject\testJNI\app\src\main\java>javah -jni com.example.binbin.testjni.myJNI      然后就能看到生成了一个h文件。    4、新建一个jni文件夹，新建main.c，把.h里面的内容复制进去，并实现里面的函数。  生成的h文件  /\* DO NOT EDIT THIS FILE - it is machine generated \*/  #include <jni.h>  /\* Header for class com\_example\_binbin\_testjni\_myJNI \*/  #ifndef \_Included\_com\_example\_binbin\_testjni\_myJNI  #define \_Included\_com\_example\_binbin\_testjni\_myJNI  #ifdef \_\_cplusplus  extern "C" {  #endif  /\*  \* Class: com\_example\_binbin\_testjni\_myJNI  \* Method: sayHello  \* Signature: ()Ljava/lang/String;  \*/  JNIEXPORT jstring JNICALL Java\_com\_example\_binbin\_testjni\_myJNI\_sayHello  (JNIEnv \*, jclass);  #ifdef \_\_cplusplus  }  #endif  #endif  拷贝修改后的main.c  /\* DO NOT EDIT THIS FILE - it is machine generated \*/  #include <jni.h>  /\* Header for class com\_example\_binbin\_testjni\_myJNI \*/  #ifndef \_Included\_com\_example\_binbin\_testjni\_myJNI  #define \_Included\_com\_example\_binbin\_testjni\_myJNI  #ifdef \_\_cplusplus  extern "C" {  #endif  /\*  \* Class: com\_example\_binbin\_testjni\_myJNI  \* Method: sayHello  \* Signature: ()Ljava/lang/String;  \*/  //就实现了个函数 JNIEnv的用法可以自行百度，这里就不展开了  JNIEXPORT jstring JNICALL Java\_com\_example\_binbin\_testjni\_myJNI\_sayHello  (JNIEnv \*env, jclass jobj){  　　//返回一句话  return (\*env)->NewStringUTF(env,"JNI hahahahahahahaha");  }  #ifdef \_\_cplusplus  }  #endif  #endif    还要在jni文件夹下添加一个空白的util.c文件，不然会报错，我也不知道为什么。      5、配置NDK  打开Project的local.properties文件添加NDK路径    打开app Module的build.gradle文件，在defaultConfig节点里添加以下代码  注意这里的moduleName，是我们在之前自己编写的类里面加载的so库名  ndk {  moduleName "JniTest"  ldLibs "log", "z", "m"  abiFilters "armeabi", "armeabi-v7a", "x86"  }    还要在gradle.properties里面加上这么一句话：  android.useDeprecatedNdk=true      6、生成SO库  完成以上步骤之后，我们rebuild一下就可以生成so库了  在项目的app\build\intermediates\ndk\debug\lib路径下      7、配置so库  在src\main下新建文件夹jniLIB，并将生成的SO文件拷贝到该文件夹下      8、使用JNI  因为native方法声明成了静态的方法，在Activity里面直接调用myJNI类就行  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  text = (TextView)findViewById(R.id.text);  　　　　　//调用  text.setText(myJNI.sayHello());  }  }    ok！  总结一下流程：  编写静态方法（用java声明）-->编译生成class文件--->编译生成h文件---->编写C文件（用C/C++实现）  ---->配置NDK---->配置so库---->在Activity调用（Java调用C/C++）。  android studio虽然很多小问题，但也算很方便了，mk文件编辑器帮你弄好了，直接可以生成so库。 Android studio 实现 AIDL前言 [Android](http://lib.csdn.net/base/android)中夸进程间通信方式有很多种方式，比如：aidl，Messenger，文件共享，广播（BroadCast），ContentProvider，Socket（网络通信）。每种方式都有自己的使用场景和优缺点，接下来几篇博客我们一一学习他们是怎么使用的。这篇博客主要是介绍[android](http://lib.csdn.net/base/android) studio下实现aidl编程。 AIDL的使用步骤 aidl远程调用传递的参数和返回值支持[Java](http://lib.csdn.net/base/java)的基本类型（int long booen char byte等）和String，List，Map等。当然也支持一个自定义对象的传递，不过此时就需要做一些特别处理了，后续会介绍aidl传递对象怎么处理。  有关aidl的相关概念这里就不详细解释了，我们来看看在Android Studio下怎么来实现aidl编程的。  服务端 生成aidl文件 新建一个MyAidlDemoServer工程，然后在main目录下右键新建一个aidl目录，然后在该目录下新建一个IMyAidlInterface.aidl文件，代码如下：  // IMyAidlInterface.aidl  package com.example.xjp.aidla;  interface IMyAidlInterface {  int add(int arg1, int arg2);  }  定义了一个IMyAidlInterface接口，接口里面定义了一个add方法用于计算两个数的和。然后Build当前工程，在app/build/generated/source/aidl/debug目录下会生成一个与IMyAidlInterface.aidl文件同样包名的一个文件，该文件下面自动生成IMyAidlInterface文件，该文件里面自动实现了一些方法用于远程调用，IMyAidlInterface代码如下：  /\*  \* This file is auto-generated. DO NOT MODIFY.  \* Original file: D:\\WorkPlace\\MyAidlDemoServer\\app\\src\\main\\aidl\\com\\example\\xjp\\aidla  \* \\IMyAidlInterface.aidl  \*/  package com.example.xjp.aidla;  public interface IMyAidlInterface extends android.os.IInterface {  /\*\*  \* Local-side IPC implementation stub class.  \*/  public static abstract class Stub extends android.os.Binder implements com.example.xjp.aidla  .IMyAidlInterface {  private static final java.lang.String DESCRIPTOR = "com.example.xjp.aidla.IMyAidlInterface";  /\*\*  \* Construct the stub at attach it to the interface.  \*/  public Stub() {  this.attachInterface(this, DESCRIPTOR);  }  /\*\*  \* Cast an IBinder object into an com.example.xjp.aidla.IMyAidlInterface interface,  \* generating a proxy if needed.  \*/  public static com.example.xjp.aidla.IMyAidlInterface asInterface(android.os.IBinder obj) {  if ((obj == null)) {  return null;  }  android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);  if (((iin != null) && (iin instanceof com.example.xjp.aidla.IMyAidlInterface))) {  return ((com.example.xjp.aidla.IMyAidlInterface) iin);  }  return new com.example.xjp.aidla.IMyAidlInterface.Stub.Proxy(obj);  }  @Override  public android.os.IBinder asBinder() {  return this;  }  @Override  public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)  throws android.os.RemoteException {  switch (code) {  case INTERFACE\_TRANSACTION: {  reply.writeString(DESCRIPTOR);  return true;  }  case TRANSACTION\_add: {  data.enforceInterface(DESCRIPTOR);  int \_arg0;  \_arg0 = data.readInt();  int \_arg1;  \_arg1 = data.readInt();  int \_result = this.add(\_arg0, \_arg1);  reply.writeNoException();  reply.writeInt(\_result);  return true;  }  }  return super.onTransact(code, data, reply, flags);  }  private static class Proxy implements com.example.xjp.aidla.IMyAidlInterface {  private android.os.IBinder mRemote;  Proxy(android.os.IBinder remote) {  mRemote = remote;  }  @Override  public android.os.IBinder asBinder() {  return mRemote;  }  public java.lang.String getInterfaceDescriptor() {  return DESCRIPTOR;  }  @Override  public int add(int arg1, int arg2) throws android.os.RemoteException {  android.os.Parcel \_data = android.os.Parcel.obtain();  android.os.Parcel \_reply = android.os.Parcel.obtain();  int \_result;  try {  \_data.writeInterfaceToken(DESCRIPTOR);  \_data.writeInt(arg1);  \_data.writeInt(arg2);  mRemote.transact(Stub.TRANSACTION\_add, \_data, \_reply, 0);  \_reply.readException();  \_result = \_reply.readInt();  } finally {  \_reply.recycle();  \_data.recycle();  }  return \_result;  }  }  static final int TRANSACTION\_add = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);  }  public int add(int arg1, int arg2) throws android.os.RemoteException;  }  以上代码都是AS自动生成的，如此一来，开发者更加容易来实现AIDL跨进程间通信。如有对以上代码感兴趣者，可自行学习。 编写远程服务 新建MyServer类实现远程服务代码如下：  public class MyServer extends Service {  IMyAidlInterface.Stub mStub = new IMyAidlInterface.Stub() {  public int add(int arg1, int arg2) {  return arg1 + arg2;  }  };  @Override  public IBinder onBind(Intent intent) {  return mStub;  }  }  服务代码也很简单，仅仅实现了IMyAidlInterface.Stub类中的 add方法，然后重写了Service的onBind方法。  记得在AndroidManifest.xml中配置MyServer服务，代码如下：  <?xml version="1.0" encoding="utf-8"?>  <manifest package="com.example.xjp.aidl"  xmlns:android="http://schemas.android.com/apk/res/android">  <application  android:allowBackup="true"  android:icon="@mipmap/ic\_launcher"  android:label="@string/app\_name"  android:theme="@style/AppTheme">  <service  android:name="com.example.xjp.aidl.MyServer"  android:process=":remote">  <intent-filter>  <action android:name="com.xjp.myService"></action>  </intent-filter>  </service>  </application>  </manifest>  以上代码给当前service设置了 android:process属性为“:remote”,这个属性有两种赋值，一是:”:remote”,一是：”remote”,区别在于字符串前面带有”:”，后面的字符串不一定是remote，开发者可以随意设置。  两者代表不同意思，带有”:”的表示该服务所在的进程是私有的，即只要有客户端去启动该服务，系统就会创建一个新的进程来运行该服务。  不带有”:”的表示该服务所在的进程是共享的，即当前系统不管有几个客户端去启动该服务，系统中只有一个进程来运行该服务。  客户端  客户端代码相对简单些，新建MyAidlDemoCustomer工程，然后直接把服务端的aidl目录直接拷贝到客户端的main目录下。这么一来客户端的aidl就无需编写了，直接和服务端的一模一样。包括路径的包名等。虽然你可以自己在客户端在写一遍aidl代码，为了不出错，请直接将服务端的aidl代码直接拷贝过来。 客户端调用代码如下 package com.example.xjp.myaidldemocustomer;  import android.app.Activity;  import android.content.ComponentName;  import android.content.Context;  import android.content.Intent;  import android.content.ServiceConnection;  import android.os.Bundle;  import android.os.IBinder;  import android.os.RemoteException;  import android.util.Log;  import android.widget.ImageView;  import android.widget.TextView;  import com.example.xjp.aidla.IMyAidlInterface;  public class MainActivity extends Activity {  IMyAidlInterface mStub;  TextView txt;  private ServiceConnection serviceConnection = new ServiceConnection() {  @Override  public void onServiceConnected(ComponentName name, IBinder service) {  Log.e("xjp", "the Connected====>" + System.currentTimeMillis());  mStub = IMyAidlInterface.Stub.asInterface(service);  if (mStub == null) {  Log.e("xjp", "the mStub is null");  } else {  try {  int value = mStub.add(1, 8);  txt.setText(value + "");  } catch (RemoteException e) {  e.printStackTrace();  }  }  }  @Override  public void onServiceDisconnected(ComponentName name) {  }  };  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  txt = (TextView) findViewById(R.id.text);  Intent intent = new Intent();  //android 5.0以后直设置action不能启动相应的服务，需要设置packageName或者Component。  intent.setAction("com.xjp.myService");  intent.setComponent(new ComponentName("com.example.xjp.aidl", "com.example.xjp.aidl.MyServer"));  //绑定服务  bindService(intent, serviceConnection, Context.BIND\_AUTO\_CREATE);  Log.e("xjp", "the bindServer start..====>" + System.currentTimeMillis());  }  @Override  protected void onDestroy() {  //解绑服务  unbindService(serviceConnection);  super.onDestroy();  }  }  如此一来，客户端也写好了。先安装服务端，在安装客户端。进入客户端之后实现了远程调用Service。 AIDL跨进程传递Bitamp对象 以上是最简单的 aidl跨进程传递java基本类型数据，最近在项目中遇到这么个需求，跨进程传递bitmap图片。当时看到这个需求就懵逼了，后来仔细分析其实是可以实现Android 利用Aidl实现跨进程传递 Bitmap对象。那么该怎么实现呢？  我们都知道Aidl支持跨进程传递 Java基本数据类型中包括byte这个类型，那么思路是不是来了呢？我们可以在服务端将需要传递的图片由bitmap转换成 byte[]类型。代码如下：  public byte[] getBitmap() {  Bitmap bitmap = BitmapFactory.decodeResource(MyServer.this.getResources(), R.drawable.bg\_top);  ByteArrayOutputStream baos = new ByteArrayOutputStream();  bitmap.compress(Bitmap.CompressFormat.PNG, 90, baos);//压缩位图  return baos.toByteArray();//创建分配字节数组  }  如此一来就将Bitmap对象转换成byte[]类型了，现在aidl就可以跨进程传递byte[]类型数据。  在客户端你只需要将远程调用得到的byte[]类型转换成Bitmap对象，然后显示即可。客户端代码如下：  byte[] bytes = mStub.getBitmap();  Bitmap bitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.length);  img.setImageBitmap(bitmap);  是不是很简单，其实Aidl传递图片只是将Bitamp对象转换成byte[]类型，然后由byte[]类型在转换回Bitmap对象的一个过程。   * 注意：在亲测的过程中发现，利用aidl传递Bitamp对象是有一定的限制的，当Bitmap达到一定大小时是Aidl跨进程传递会失败。所以，利用Aidl跨进程可以传递较小的图片，至于多大可以根据你需求测试以免出错。这也是由于Aidl跨进程传递数据是有限的，数据过大就不适合利用aidl夸进程传递了，应该用Socket传递。因为Aidl跨进程间通信是一个实时的，同步的的一个过程，即Aidl的远程调用方法不能被阻塞，因此不能传递大数据。   总结  这是一个在AS下最简单的一个AIDL编程：  1.服务端创建一个aidl目录，然后在该目录下新建一个.aidl为后缀的接口类，该类定义远程调用的接口方法。  2.build编译之后会在app/build/generated/source/aidl/debug目录下会生成aidl远程实现类，该类是AS自动生成的。  3.在AndroidManifest.xml下配置Service的action和process属性。  4.将服务端的aidl目录拷贝到客户端相应的目录下，然后编写客户端调用代码，AS下简单的aidl编程就ok了。  5.利用Aidl实现跨进程传递较小的Bitmap对象。 子线程Looper.prepare做了哪些事情  |  | | --- | | 当你的线程想拥有自己的MessageQueue的时候先Looper.prepare(),然后Looper.loop(); 参照源码:  [java] [view plain](http://blog.csdn.net/lwuit/article/details/6796464) [copy](http://blog.csdn.net/lwuit/article/details/6796464)   1. public static final void prepare() { 2. if (sThreadLocal.get() != null) { 3. throw new RuntimeException("Only one Looper may be created per thread"); 4. } 5. sThreadLocal.set(new Looper()); 6. }   这段代码就是通过ThreadLocal来产生一个Looper对象做为线程局部变量，然后调用Looper.loop()则是取出Looper对象中的MessageQueue进行消息循环了，这样形成了这个线程的消息队列。 一般情况下只会有主线程会调用prepare方法(ActivityThread的main函数)。 | |  |   使线程拥有自己的消息列队，主线程拥有自己的消息列队，一般线程创建时没有自己的消息列队，消息处理时就在主线程中完成，如果线程中使用Looper.prepare()和Looper.loop()创建了消息队列就可以让消息处理在该线程中完成 一个Activity中多个Handler和消息处理过程 能否有多个handler  handler的构造方法  public Handler() {           ….  mLooper = Looper.myLooper();  mQueue = mLooper.mQueue;  mCallback = null;  …  }      因为几乎主要的成员变量都是从Looper中拿出来的，最初以为一个线程中只能有一个handler。后来看构造方法也没有限制，尝试的写了下，发现的确可以实例化多个handler      后来又想，如果两个handler都重写了handleMessage方法，而handleMessage方法之后，消息会从消息队列中移除。那么假设A发送了一个消息，想实现一些功能，但是B却对这个消息进行了处理，那么A中的功能不是无法实现了吗？      重新看了一下消息的分法机制，终于恍然大悟  1、 Handler发送消息的时候  public boolean sendMessageAtTime(Message msg, long uptimeMillis){  ...          if (queue !=null) {              msg.target = this;// 发送消息的时候会把target设为当前的Handler              sent = queue.enqueueMessage(msg, uptimeMillis);          }  ...  }  2、而Looper中的循环  public static final void loop() {          Looper me = myLooper();          MessageQueue queue = me.mQueue;          while (true) {          ...              Message msg = queue.next();          ...          // 我汗啊… 竟然直接交给了target即发送这个消息的handler处理             msg.target.dispatchMessage(msg);             msg.recycle();              }          }  }  3、这么简单的问题竟然纠结了好久，只能感叹菜鸟真可怕        既然看到这了，就稍微看下dispatchMessage的逻辑吧  dispatchMessage     |  | | --- | | public void dispatchMessage(Message msg) {          if (msg.callback !=null) {              handleCallback(msg);          } else {            if (mCallback !=null) {                  if (mCallback.handleMessage(msg)) {                      return;                  }             }              handleMessage(msg);          }  } |   ²  --  如果msg.callback不为空，则调用handleCallback(msg);  callback是一个Runnable的实例，那么什么时候不为空呢？  |-  当调用 message 的obtain静态方法来实例化Message的时候，会对这个Runnable赋值      public static Message obtain(Handler h, Runnable callback) {          Message m = obtain();          m.target = h;          m.callback = callback;          return m;      }  |-  更常用的方式      当Handler.post(Runnable r)的时候   |  | | --- | | public final boolean post(Runnable r)       return  sendMessageDelayed(getPostMessage(r), 0);  }      private final Message getPostMessage(Runnable r) {          Message m = Message.obtain();          m.callback = r;          return m;  } |       这时候，handler  dispatch这个消息会直接调用Runnable中的run方法  这也是为什么重写的handlerMessage不对这种形式发送的消息进行处理    ²  如果mCallback不为空，则调用mCallback.handleMessage方法  而这个Callback默认情况下为空，只有当调用  public Handler(Callback callback)或者public Handler(Looper looper, Callback callback)这两种构造方法的时候才会被赋值，其实也就是要直接通过new hanlder （）构造实例使用,而不是通过内部类重写handlerMessage 方式来处理的时侯采用的手段。      当然，一般使用handler处理消息都是为了与ui线程通信，而ui的looper是系统维护的，所以推荐第一种方式。      当dispatch消息的时候，会直接调用callback.handleMessage()方法    ²  前边两个都为空的时候，才会调用Handler的handleMessage方法，如果没有重写，则调用系统默认的handleMessage，即什么也不做 属性动画 在手机上去实现一些动画效果算是件比较炫酷的事情，因此[Android](http://lib.csdn.net/base/android)系统在一开始的时候就给我们提供了两种实现动画效果的方式，逐帧动画(frame-by-frame animation)和补间动画(tweened animation)。逐帧动画的工作原理很简单，其实就是将一个完整的动画拆分成一张张单独的图片，然后再将它们连贯起来进行播放，类似于动画片的工作原理。补间动画则是可以对View进行一系列的动画操作，包括淡入淡出、缩放、平移、旋转四种。  然而自[android](http://lib.csdn.net/base/android) 3.0版本开始，系统给我们提供了一种全新的动画模式，属性动画(property animation)，它的功能非常强大，弥补了之前补间动画的一些缺陷，几乎是可以完全替代掉补间动画了。对于逐帧动画和补间动画的用法，我不想再多讲，它们的技术已经比较老了，而且网上资料也非常多，那么今天我们这篇文章的主题就是对Android属性动画进行一次完全解析。  为什么要引入属性动画？  Android之前的补间动画机制其实还算是比较健全的，在android.view.animation包下面有好多的类可以供我们操作，来完成一系列的动画效果，比如说对View进行移动、缩放、旋转和淡入淡出，并且我们还可以借助AnimationSet来将这些动画效果组合起来使用，除此之外还可以通过配置Interpolator来控制动画的播放速度等等等等。那么这里大家可能要产生疑问了，既然之前的动画机制已经这么健全了，为什么还要引入属性动画呢？  其实上面所谓的健全都是相对的，如果你的需求中只需要对View进行移动、缩放、旋转和淡入淡出操作，那么补间动画确实已经足够健全了。但是很显然，这些功能是不足以覆盖所有的场景的，一旦我们的需求超出了移动、缩放、旋转和淡入淡出这四种对View的操作，那么补间动画就不能再帮我们忙了，也就是说它在功能和可扩展方面都有相当大的局限性，那么下面我们就来看看补间动画所不能胜任的场景。  注意上面我在介绍补间动画的时候都有使用“对View进行操作”这样的描述，没错，补间动画是只能够作用在View上的。也就是说，我们可以对一个Button、TextView、甚至是LinearLayout、或者其它任何继承自View的组件进行动画操作，但是如果我们想要对一个非View的对象进行动画操作，抱歉，补间动画就帮不上忙了。可能有的朋友会感到不能理解，我怎么会需要对一个非View的对象进行动画操作呢？这里我举一个简单的例子，比如说我们有一个自定义的View，在这个View当中有一个Point对象用于管理坐标，然后在onDraw()方法当中就是根据这个Point对象的坐标值来进行绘制的。也就是说，如果我们可以对Point对象进行动画操作，那么整个自定义View的动画效果就有了。显然，补间动画是不具备这个功能的，这是它的第一个缺陷。  然后补间动画还有一个缺陷，就是它只能够实现移动、缩放、旋转和淡入淡出这四种动画操作，那如果我们希望可以对View的背景色进行动态地改变呢？很遗憾，我们只能靠自己去实现了。说白了，之前的补间动画机制就是使用硬编码的方式来完成的，功能限定死就是这些，基本上没有任何扩展性可言。  最后，补间动画还有一个致命的缺陷，就是它只是改变了View的显示效果而已，而不会真正去改变View的属性。什么意思呢？比如说，现在屏幕的左上角有一个按钮，然后我们通过补间动画将它移动到了屏幕的右下角，现在你可以去尝试点击一下这个按钮，点击事件是绝对不会触发的，因为实际上这个按钮还是停留在屏幕的左上角，只不过补间动画将这个按钮绘制到了屏幕的右下角而已。  也正是因为这些原因，Android开发团队决定在3.0版本当中引入属性动画这个功能，那么属性动画是不是就把上述的问题全部解决掉了？下面我们就来一起看一看。  新引入的属性动画机制已经不再是针对于View来设计的了，也不限定于只能实现移动、缩放、旋转和淡入淡出这几种动画操作，同时也不再只是一种视觉上的动画效果了。它实际上是一种不断地对值进行操作的机制，并将值赋值到指定对象的指定属性上，可以是任意对象的任意属性。所以我们仍然可以将一个View进行移动或者缩放，但同时也可以对自定义View中的Point对象进行动画操作了。我们只需要告诉系统动画的运行时长，需要执行哪种类型的动画，以及动画的初始值和结束值，剩下的工作就可以全部交给系统去完成了。  既然属性动画的实现机制是通过对目标对象进行赋值并修改其属性来实现的，那么之前所说的按钮显示的问题也就不复存在了，如果我们通过属性动画来移动一个按钮，那么这个按钮就是真正的移动了，而不再是仅仅在另外一个位置绘制了而已。  好了，介绍了这么多，相信大家已经对属性动画有了一个最基本的认识了，下面我们就来开始学习一下属性动画的用法。  ValueAnimator  ValueAnimator是整个属性动画机制当中最核心的一个类，前面我们已经提到了，属性动画的运行机制是通过不断地对值进行操作来实现的，而初始值和结束值之间的动画过渡就是由ValueAnimator这个类来负责计算的。它的内部使用一种时间循环的机制来计算值与值之间的动画过渡，我们只需要将初始值和结束值提供给ValueAnimator，并且告诉它动画所需运行的时长，那么ValueAnimator就会自动帮我们完成从初始值平滑地过渡到结束值这样的效果。除此之外，ValueAnimator还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，确实是一个非常重要的类。  但是ValueAnimator的用法却一点都不复杂，我们先从最简单的功能看起吧，比如说想要将一个值从0平滑过渡到1，时长300毫秒，就可以这样写：  [java] view plain copy   1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f); 2. anim.setDuration(300); 3. anim.start();   怎么样？很简单吧，调用ValueAnimator的ofFloat()方法就可以构建出一个ValueAnimator的实例，ofFloat()方法当中允许传入多个float类型的参数，这里传入0和1就表示将值从0平滑过渡到1，然后调用ValueAnimator的setDuration()方法来设置动画运行的时长，最后调用start()方法启动动画。  用法就是这么简单，现在如果你运行一下上面的代码，动画就会执行了。可是这只是一个将值从0过渡到1的动画，又看不到任何界面效果，我们怎样才能知道这个动画是不是已经真正运行了呢？这就需要借助监听器来实现了，如下所示：  [java] view plain copy   1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f); 2. anim.setDuration(300); 3. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 4. @Override 5. public void onAnimationUpdate(ValueAnimator animation) { 6. float currentValue = (float) animation.getAnimatedValue(); 7. Log.d("TAG", "cuurent value is " + currentValue); 8. } 9. }); 10. anim.start();   可以看到，这里我们通过addUpdateListener()方法来添加一个动画的监听器，在动画执行的过程中会不断地进行回调，我们只需要在回调方法当中将当前的值取出并打印出来，就可以知道动画有没有真正运行了。运行上述代码，控制台打印如下所示：    从打印日志的值我们就可以看出，ValueAnimator确实已经在正常工作了，值在300毫秒的时间内从0平滑过渡到了1，而这个计算工作就是由ValueAnimator帮助我们完成的。另外ofFloat()方法当中是可以传入任意多个参数的，因此我们还可以构建出更加复杂的动画逻辑，比如说将一个值在5秒内从0过渡到5，再过渡到3，再过渡到10，就可以这样写：  [java] view plain copy   1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 5f, 3f, 10f); 2. anim.setDuration(5000); 3. anim.start();   当然也许你并不需要小数位数的动画过渡，可能你只是希望将一个整数值从0平滑地过渡到100，那么也很简单，只需要调用ValueAnimator的ofInt()方法就可以了，如下所示：  [java] view plain copy   1. ValueAnimator anim = ValueAnimator.ofInt(0, 100);   ValueAnimator当中最常用的应该就是ofFloat()和ofInt()这两个方法了，另外还有一个ofObject()方法，我会在下篇文章进行讲解。  那么除此之外，我们还可以调用setStartDelay()方法来设置动画延迟播放的时间，调用setRepeatCount()和setRepeatMode()方法来设置动画循环播放的次数以及循环播放的模式，循环模式包括RESTART和REVERSE两种，分别表示重新播放和倒序播放的意思。这些方法都很简单，我就不再进行详细讲解了。  ObjectAnimator  相比于ValueAnimator，ObjectAnimator可能才是我们最常接触到的类，因为ValueAnimator只不过是对值进行了一个平滑的动画过渡，但我们实际使用到这种功能的场景好像并不多。而ObjectAnimator则就不同了，它是可以直接对任意对象的任意属性进行动画操作的，比如说View的alpha属性。  不过虽说ObjectAnimator会更加常用一些，但是它其实是继承自ValueAnimator的，底层的动画实现机制也是基于ValueAnimator来完成的，因此ValueAnimator仍然是整个属性动画当中最核心的一个类。那么既然是继承关系，说明ValueAnimator中可以使用的方法在ObjectAnimator中也是可以正常使用的，它们的用法也非常类似，这里如果我们想要将一个TextView在5秒中内从常规变换成全透明，再从全透明变换成常规，就可以这样写：  [java] view plain copy   1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f); 2. animator.setDuration(5000); 3. animator.start();   可以看到，我们还是调用了ofFloat()方法来去创建一个ObjectAnimator的实例，只不过ofFloat()方法当中接收的参数有点变化了。这里第一个参数要求传入一个object对象，我们想要对哪个对象进行动画操作就传入什么，这里我传入了一个textview。第二个参数是想要对该对象的哪个属性进行动画操作，由于我们想要改变TextView的不透明度，因此这里传入"alpha"。后面的参数就是不固定长度了，想要完成什么样的动画就传入什么值，这里传入的值就表示将TextView从常规变换成全透明，再从全透明变换成常规。之后调用setDuration()方法来设置动画的时长，然后调用start()方法启动动画，效果如下图所示：    学会了这一个用法之后，其它的用法我们就可以举一反三了，那比如说我们想要将TextView进行一次360度的旋转，就可以这样写：  [java] view plain copy   1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "rotation", 0f, 360f); 2. animator.setDuration(5000); 3. animator.start();   可以看到，这里我们将第二个参数改成了"rotation"，然后将动画的初始值和结束值分别设置成0和360，现在运行一下代码，效果如下图所示：    那么如果想要将TextView先向左移出屏幕，然后再移动回来，就可以这样写：  [java] view plain copy   1. float curTranslationX = textview.getTranslationX(); 2. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "translationX", curTranslationX, -500f, curTranslationX); 3. animator.setDuration(5000); 4. animator.start();   这里我们先是调用了TextView的getTranslationX()方法来获取到当前TextView的translationX的位置，然后ofFloat()方法的第二个参数传入"translationX"，紧接着后面三个参数用于告诉系统TextView应该怎么移动，现在运行一下代码，效果如下图所示：    然后我们还可以TextView进行缩放操作，比如说将TextView在垂直方向上放大3倍再还原，就可以这样写：  [java] view plain copy   1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "scaleY", 1f, 3f, 1f); 2. animator.setDuration(5000); 3. animator.start();   这里将ofFloat()方法的第二个参数改成了"scaleY"，表示在垂直方向上进行缩放，现在重新运行一下程序，效果如下图所示：    到目前为止，ObjectAnimator的用法还算是相当简单吧，但是我相信肯定会有不少朋友现在心里都有同样一个疑问，就是ofFloat()方法的第二个参数到底可以传哪些值呢？目前我们使用过了alpha、rotation、translationX和scaleY这几个值，分别可以完成淡入淡出、旋转、水平移动、垂直缩放这几种动画，那么还有哪些值是可以使用的呢？其实这个问题的答案非常玄乎，就是我们可以传入任意的值到ofFloat()方法的第二个参数当中。任意的值？相信这很出乎大家的意料吧，但事实就是如此。因为ObjectAnimator在设计的时候就没有针对于View来进行设计，而是针对于任意对象的，它所负责的工作就是不断地向某个对象中的某个属性进行赋值，然后对象根据属性值的改变再来决定如何展现出来。  那么比如说我们调用下面这样一段代码：  [java] view plain copy   1. ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f);   其实这段代码的意思就是ObjectAnimator会帮我们不断地改变textview对象中alpha属性的值，从1f变化到0f。然后textview对象需要根据alpha属性值的改变来不断刷新界面的显示，从而让用户可以看出淡入淡出的动画效果。  那么textview对象中是不是有alpha属性这个值呢？没有，不仅textview没有这个属性，连它所有的父类也是没有这个属性的！这就奇怪了，textview当中并没有alpha这个属性，ObjectAnimator是如何进行操作的呢？其实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：  [java] view plain copy   1. public void setAlpha(float value); 2. public float getAlpha();   那么textview对象中是否有这两个方法呢？确实有，并且这两个方法是由View对象提供的，也就是说不仅TextView可以使用这个属性来进行淡入淡出动画操作，任何继承自View的对象都可以的。  既然alpha是这个样子，相信大家一定已经明白了，前面我们所用的所有属性都是这个工作原理，那么View当中一定也存在着setRotation()、getRotation()、setTranslationX()、getTranslationX()、setScaleY()、getScaleY()这些方法，不信的话你可以到View当中去找一下。  组合动画  独立的动画能够实现的视觉效果毕竟是相当有限的，因此将多个动画组合到一起播放就显得尤为重要。幸运的是，Android团队在设计属性动画的时候也充分考虑到了组合动画的功能，因此提供了一套非常丰富的API来让我们将多个动画组合到一起。  实现组合动画功能主要需要借助AnimatorSet这个类，这个类提供了一个play()方法，如果我们向这个方法中传入一个Animator对象(ValueAnimator或ObjectAnimator)将会返回一个AnimatorSet.Builder的实例，AnimatorSet.Builder中包括以下四个方法：   1. after(Animator anim)   将现有动画插入到传入的动画之后执行 2. after(long delay)   将现有动画延迟指定毫秒后执行 3. before(Animator anim)   将现有动画插入到传入的动画之前执行 4. with(Animator anim)   将现有动画和传入的动画同时执行   好的，有了这四个方法，我们就可以完成组合动画的逻辑了，那么比如说我们想要让TextView先从屏幕外移动进屏幕，然后开始旋转360度，旋转的同时进行淡入淡出操作，就可以这样写：  [java] view plain copy   1. ObjectAnimator moveIn = ObjectAnimator.ofFloat(textview, "translationX", -500f, 0f); 2. ObjectAnimator rotate = ObjectAnimator.ofFloat(textview, "rotation", 0f, 360f); 3. ObjectAnimator fadeInOut = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f); 4. AnimatorSet animSet = new AnimatorSet(); 5. animSet.play(rotate).with(fadeInOut).after(moveIn); 6. animSet.setDuration(5000); 7. animSet.start();   可以看到，这里我们先是把三个动画的对象全部创建出来，然后new出一个AnimatorSet对象之后将这三个动画对象进行播放排序，让旋转和淡入淡出动画同时进行，并把它们插入到了平移动画的后面，最后是设置动画时长以及启动动画。运行一下上述代码，效果如下图所示：    Animator监听器  在很多时候，我们希望可以监听到动画的各种事件，比如动画何时开始，何时结束，然后在开始或者结束的时候去执行一些逻辑处理。这个功能是完全可以实现的，Animator类当中提供了一个addListener()方法，这个方法接收一个AnimatorListener，我们只需要去实现这个AnimatorListener就可以监听动画的各种事件了。  大家已经知道，ObjectAnimator是继承自ValueAnimator的，而ValueAnimator又是继承自Animator的，因此不管是ValueAnimator还是ObjectAnimator都是可以使用addListener()这个方法的。另外AnimatorSet也是继承自Animator的，因此addListener()这个方法算是个通用的方法。  添加一个监听器的代码如下所示：  [java] view plain copy   1. anim.addListener(new AnimatorListener() { 2. @Override 3. public void onAnimationStart(Animator animation) { 4. } 6. @Override 7. public void onAnimationRepeat(Animator animation) { 8. } 10. @Override 11. public void onAnimationEnd(Animator animation) { 12. } 14. @Override 15. public void onAnimationCancel(Animator animation) { 16. } 17. });   可以看到，我们需要实现接口中的四个方法，onAnimationStart()方法会在动画开始的时候调用，onAnimationRepeat()方法会在动画重复执行的时候调用，onAnimationEnd()方法会在动画结束的时候调用，onAnimationCancel()方法会在动画被取消的时候调用。  但是也许很多时候我们并不想要监听那么多个事件，可能我只想要监听动画结束这一个事件，那么每次都要将四个接口全部实现一遍就显得非常繁琐。没关系，为此Android提供了一个适配器类，叫作AnimatorListenerAdapter，使用这个类就可以解决掉实现接口繁琐的问题了，如下所示：  [java] view plain copy   1. anim.addListener(new AnimatorListenerAdapter() { 2. });   这里我们向addListener()方法中传入这个适配器对象，由于AnimatorListenerAdapter中已经将每个接口都实现好了，所以这里不用实现任何一个方法也不会报错。那么如果我想监听动画结束这个事件，就只需要单独重写这一个方法就可以了，如下所示：  [java] view plain copy   1. anim.addListener(new AnimatorListenerAdapter() { 2. @Override 3. public void onAnimationEnd(Animator animation) { 4. } 5. });   使用XML编写动画  我们可以使用代码来编写所有的动画功能，这也是最常用的一种做法。不过，过去的补间动画除了使用代码编写之外也是可以使用XML编写的，因此属性动画也提供了这一功能，即通过XML来完成和代码一样的属性动画功能。  通过XML来编写动画可能会比通过代码来编写动画要慢一些，但是在重用方面将会变得非常轻松，比如某个将通用的动画编写到XML里面，我们就可以在各个界面当中轻松去重用它。  如果想要使用XML来编写动画，首先要在res目录下面新建一个animator文件夹，所有属性动画的XML文件都应该存放在这个文件夹当中。然后在XML文件中我们一共可以使用如下三种标签：   1. <animator>  对应代码中的ValueAnimator 2. <objectAnimator>  对应代码中的ObjectAnimator 3. <set>  对应代码中的AnimatorSet   那么比如说我们想要实现一个从0到100平滑过渡的动画，在XML当中就可以这样写：  [html] view plain copy   1. <animator xmlns:android="http://schemas.android.com/apk/res/android" 2. android:valueFrom="0" 3. android:valueTo="100" 4. android:valueType="intType"/>   而如果我们想将一个视图的alpha属性从1变成0，就可以这样写：  [html] view plain copy   1. <objectAnimator xmlns:android="http://schemas.android.com/apk/res/android" 2. android:valueFrom="1" 3. android:valueTo="0" 4. android:valueType="floatType" 5. android:propertyName="alpha"/>   其实XML编写动画在可读性方面还是挺高的，上面的内容相信不用我做解释大家也都看得懂吧。  另外，我们也可以使用XML来完成复杂的组合动画操作，比如将一个视图先从屏幕外移动进屏幕，然后开始旋转360度，旋转的同时进行淡入淡出操作，就可以这样写：  [html] view plain copy   1. <set xmlns:android="http://schemas.android.com/apk/res/android" 2. android:ordering="sequentially" > 4. <objectAnimator 5. android:duration="2000" 6. android:propertyName="translationX" 7. android:valueFrom="-500" 8. android:valueTo="0" 9. android:valueType="floatType" > 10. </objectAnimator> 12. <set android:ordering="together" > 13. <objectAnimator 14. android:duration="3000" 15. android:propertyName="rotation" 16. android:valueFrom="0" 17. android:valueTo="360" 18. android:valueType="floatType" > 19. </objectAnimator> 21. <set android:ordering="sequentially" > 22. <objectAnimator 23. android:duration="1500" 24. android:propertyName="alpha" 25. android:valueFrom="1" 26. android:valueTo="0" 27. android:valueType="floatType" > 28. </objectAnimator> 29. <objectAnimator 30. android:duration="1500" 31. android:propertyName="alpha" 32. android:valueFrom="0" 33. android:valueTo="1" 34. android:valueType="floatType" > 35. </objectAnimator> 36. </set> 37. </set> 39. </set>   这段XML实现的效果和我们刚才通过代码来实现的组合动画的效果是一模一样的，每个参数的含义都非常清楚，相信大家都是一看就懂，我就不再一一解释了。  最后XML文件是编写好了，那么我们如何在代码中把文件加载进来并将动画启动呢？只需调用如下代码即可：  [java] view plain copy   1. Animator animator = AnimatorInflater.loadAnimator(context, R.animator.anim\_file); 2. animator.setTarget(view); 3. animator.start();   调用AnimatorInflater的loadAnimator来将XML动画文件加载进来，然后再调用setTarget()方法将这个动画设置到某一个对象上面，最后再调用start()方法启动动画就可以了，就是这么简单。  ValueAnimator的高级用法  在上篇文章中介绍补间动画缺点的时候有提到过，补间动画是只能对View对象进行动画操作的。而属性动画就不再受这个限制，它可以对任意对象进行动画操作。那么大家应该还记得在上篇文章当中我举的一个例子，比如说我们有一个自定义的View，在这个View当中有一个Point对象用于管理坐标，然后在onDraw()方法当中就是根据这个Point对象的坐标值来进行绘制的。也就是说，如果我们可以对Point对象进行动画操作，那么整个自定义View的动画效果就有了。OK，下面我们就来学习一下如何实现这样的效果。  在开始动手之前，我们还需要掌握另外一个知识点，就是TypeEvaluator的用法。可能在大多数情况下我们使用属性动画的时候都不会用到TypeEvaluator，但是大家还是应该了解一下它的用法，以防止当我们遇到一些解决不掉的问题时能够想起来还有这样的一种解决方案。  那么TypeEvaluator的作用到底是什么呢？简单来说，就是告诉动画系统如何从初始值过度到结束值。我们在上一篇文章中学到的ValueAnimator.ofFloat()方法就是实现了初始值与结束值之间的平滑过度，那么这个平滑过度是怎么做到的呢？其实就是系统内置了一个FloatEvaluator，它通过计算告知动画系统如何从初始值过度到结束值，我们来看一下FloatEvaluator的代码实现：  [java] view plain copy   1. public class FloatEvaluator implements TypeEvaluator { 2. public Object evaluate(float fraction, Object startValue, Object endValue) { 3. float startFloat = ((Number) startValue).floatValue(); 4. return startFloat + fraction \* (((Number) endValue).floatValue() - startFloat); 5. } 6. }   可以看到，FloatEvaluator实现了TypeEvaluator接口，然后重写evaluate()方法。evaluate()方法当中传入了三个参数，第一个参数fraction非常重要，这个参数用于表示动画的完成度的，我们应该根据它来计算当前动画的值应该是多少，第二第三个参数分别表示动画的初始值和结束值。那么上述代码的逻辑就比较清晰了，用结束值减去初始值，算出它们之间的差值，然后乘以fraction这个系数，再加上初始值，那么就得到当前动画的值了。  好的，那FloatEvaluator是系统内置好的功能，并不需要我们自己去编写，但介绍它的实现方法是要为我们后面的功能铺路的。前面我们使用过了ValueAnimator的ofFloat()和ofInt()方法，分别用于对浮点型和整型的数据进行动画操作的，但实际上ValueAnimator中还有一个ofObject()方法，是用于对任意对象进行动画操作的。但是相比于浮点型或整型数据，对象的动画操作明显要更复杂一些，因为系统将完全无法知道如何从初始对象过度到结束对象，因此这个时候我们就需要实现一个自己的TypeEvaluator来告知系统如何进行过度。  下面来先定义一个Point类，如下所示：  [java] view plain copy   1. public class Point { 3. private float x; 5. private float y; 7. public Point(float x, float y) { 8. this.x = x; 9. this.y = y; 10. } 12. public float getX() { 13. return x; 14. } 16. public float getY() { 17. return y; 18. } 20. }   Point类非常简单，只有x和y两个变量用于记录坐标的位置，并提供了构造方法来设置坐标，以及get方法来获取坐标。接下来定义PointEvaluator，如下所示：  [java] view plain copy   1. public class PointEvaluator implements TypeEvaluator{ 3. @Override 4. public Object evaluate(float fraction, Object startValue, Object endValue) { 5. Point startPoint = (Point) startValue; 6. Point endPoint = (Point) endValue; 7. float x = startPoint.getX() + fraction \* (endPoint.getX() - startPoint.getX()); 8. float y = startPoint.getY() + fraction \* (endPoint.getY() - startPoint.getY()); 9. Point point = new Point(x, y); 10. return point; 11. } 13. }   可以看到，PointEvaluator同样实现了TypeEvaluator接口并重写了evaluate()方法。其实evaluate()方法中的逻辑还是非常简单的，先是将startValue和endValue强转成Point对象，然后同样根据fraction来计算当前动画的x和y的值，最后组装到一个新的Point对象当中并返回。  这样我们就将PointEvaluator编写完成了，接下来我们就可以非常轻松地对Point对象进行动画操作了，比如说我们有两个Point对象，现在需要将Point1通过动画平滑过度到Point2，就可以这样写：  [java] view plain copy   1. Point point1 = new Point(0, 0); 2. Point point2 = new Point(300, 300); 3. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), point1, point2); 4. anim.setDuration(5000); 5. anim.start();   代码很简单，这里我们先是new出了两个Point对象，并在构造函数中分别设置了它们的坐标点。然后调用ValueAnimator的ofObject()方法来构建ValueAnimator的实例，这里需要注意的是，ofObject()方法要求多传入一个TypeEvaluator参数，这里我们只需要传入刚才定义好的PointEvaluator的实例就可以了。  好的，这就是自定义TypeEvaluator的全部用法，掌握了这些知识之后，我们就可以来尝试一下如何通过对Point对象进行动画操作，从而实现整个自定义View的动画效果。  新建一个MyAnimView继承自View，代码如下所示：  [java] view plain copy   1. public class MyAnimView extends View { 3. public static final float RADIUS = 50f; 5. private Point currentPoint; 7. private Paint mPaint; 9. public MyAnimView(Context context, AttributeSet attrs) { 10. super(context, attrs); 11. mPaint = new Paint(Paint.ANTI\_ALIAS\_FLAG); 12. mPaint.setColor(Color.BLUE); 13. } 15. @Override 16. protected void onDraw(Canvas canvas) { 17. if (currentPoint == null) { 18. currentPoint = new Point(RADIUS, RADIUS); 19. drawCircle(canvas); 20. startAnimation(); 21. } else { 22. drawCircle(canvas); 23. } 24. } 26. private void drawCircle(Canvas canvas) { 27. float x = currentPoint.getX(); 28. float y = currentPoint.getY(); 29. canvas.drawCircle(x, y, RADIUS, mPaint); 30. } 32. private void startAnimation() { 33. Point startPoint = new Point(RADIUS, RADIUS); 34. Point endPoint = new Point(getWidth() - RADIUS, getHeight() - RADIUS); 35. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 36. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 37. @Override 38. public void onAnimationUpdate(ValueAnimator animation) { 39. currentPoint = (Point) animation.getAnimatedValue(); 40. invalidate(); 41. } 42. }); 43. anim.setDuration(5000); 44. anim.start(); 45. } 47. }   基本上还是很简单的，总共也没几行代码。首先在自定义View的构造方法当中初始化了一个Paint对象作为画笔，并将画笔颜色设置为蓝色，接着在onDraw()方法当中进行绘制。这里我们绘制的逻辑是由currentPoint这个对象控制的，如果currentPoint对象不等于空，那么就调用drawCircle()方法在currentPoint的坐标位置画出一个半径为50的圆，如果currentPoint对象是空，那么就调用startAnimation()方法来启动动画。  那么我们来观察一下startAnimation()方法中的代码，其实大家应该很熟悉了，就是对Point对象进行了一个动画操作而已。这里我们定义了一个startPoint和一个endPoint，坐标分别是View的左上角和右下角，并将动画的时长设为5秒。然后有一点需要大家注意的，就是我们通过监听器对动画的过程进行了监听，每当Point值有改变的时候都会回调onAnimationUpdate()方法。在这个方法当中，我们对currentPoint对象进行了重新赋值，并调用了invalidate()方法，这样的话onDraw()方法就会重新调用，并且由于currentPoint对象的坐标已经改变了，那么绘制的位置也会改变，于是一个平移的动画效果也就实现了。  下面我们只需要在布局文件当中引入这个自定义控件：  [html] view plain copy   1. <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android" 2. android:layout\_width="match\_parent" 3. android:layout\_height="match\_parent" 4. > 6. <com.example.tony.myapplication.MyAnimView 7. android:layout\_width="match\_parent" 8. android:layout\_height="match\_parent" /> 10. </RelativeLayout>   最后运行一下程序，效果如下图所示：    OK！这样我们就成功实现了通过对对象进行值操作来实现动画效果的功能，这就是ValueAnimator的高级用法。  ObjectAnimator的高级用法  ObjectAnimator的基本用法和工作原理在上一篇文章当中都已经讲解过了，相信大家都已经掌握。那么大家应该都还记得，我们在吐槽补间动画的时候有提到过，补间动画是只能实现移动、缩放、旋转和淡入淡出这四种动画操作的，功能限定死就是这些，基本上没有任何扩展性可言。比如我们想要实现对View的颜色进行动态改变，补间动画是没有办法做到的。  但是属性动画就不会受这些条条框框的限制，它的扩展性非常强，对于动态改变View的颜色这种功能是完全可是胜任的，那么下面我们就来学习一下如何实现这样的效果。  大家应该都还记得，ObjectAnimator内部的工作机制是通过寻找特定属性的get和set方法，然后通过方法不断地对值进行改变，从而实现动画效果的。因此我们就需要在MyAnimView中定义一个color属性，并提供它的get和set方法。这里我们可以将color属性设置为字符串类型，使用#RRGGBB这种格式来表示颜色值，代码如下所示：  [java] view plain copy   1. public class MyAnimView extends View { 3. ... 5. private String color; 7. public String getColor() { 8. return color; 9. } 11. public void setColor(String color) { 12. this.color = color; 13. mPaint.setColor(Color.parseColor(color)); 14. invalidate(); 15. } 17. ... 19. }   注意在setColor()方法当中，我们编写了一个非常简单的逻辑，就是将画笔的颜色设置成方法参数传入的颜色，然后调用了invalidate()方法。这段代码虽然只有三行，但是却执行了一个非常核心的功能，就是在改变了画笔颜色之后立即刷新视图，然后onDraw()方法就会调用。在onDraw()方法当中会根据当前画笔的颜色来进行绘制，这样颜色也就会动态进行改变了。  那么接下来的问题就是怎样让setColor()方法得到调用了，毫无疑问，当然是要借助ObjectAnimator类，但是在使用ObjectAnimator之前我们还要完成一个非常重要的工作，就是编写一个用于告知系统如何进行颜色过度的TypeEvaluator。创建ColorEvaluator并实现TypeEvaluator接口，代码如下所示：  [java] view plain copy   1. public class ColorEvaluator implements TypeEvaluator { 3. private int mCurrentRed = -1; 5. private int mCurrentGreen = -1; 7. private int mCurrentBlue = -1; 9. @Override 10. public Object evaluate(float fraction, Object startValue, Object endValue) { 11. String startColor = (String) startValue; 12. String endColor = (String) endValue; 13. int startRed = Integer.parseInt(startColor.substring(1, 3), 16); 14. int startGreen = Integer.parseInt(startColor.substring(3, 5), 16); 15. int startBlue = Integer.parseInt(startColor.substring(5, 7), 16); 16. int endRed = Integer.parseInt(endColor.substring(1, 3), 16); 17. int endGreen = Integer.parseInt(endColor.substring(3, 5), 16); 18. int endBlue = Integer.parseInt(endColor.substring(5, 7), 16); 19. // 初始化颜色的值 20. if (mCurrentRed == -1) { 21. mCurrentRed = startRed; 22. } 23. if (mCurrentGreen == -1) { 24. mCurrentGreen = startGreen; 25. } 26. if (mCurrentBlue == -1) { 27. mCurrentBlue = startBlue; 28. } 29. // 计算初始颜色和结束颜色之间的差值 30. int redDiff = Math.abs(startRed - endRed); 31. int greenDiff = Math.abs(startGreen - endGreen); 32. int blueDiff = Math.abs(startBlue - endBlue); 33. int colorDiff = redDiff + greenDiff + blueDiff; 34. if (mCurrentRed != endRed) { 35. mCurrentRed = getCurrentColor(startRed, endRed, colorDiff, 0, 36. fraction); 37. } else if (mCurrentGreen != endGreen) { 38. mCurrentGreen = getCurrentColor(startGreen, endGreen, colorDiff, 39. redDiff, fraction); 40. } else if (mCurrentBlue != endBlue) { 41. mCurrentBlue = getCurrentColor(startBlue, endBlue, colorDiff, 42. redDiff + greenDiff, fraction); 43. } 44. // 将计算出的当前颜色的值组装返回 45. String currentColor = "#" + getHexString(mCurrentRed) 46. + getHexString(mCurrentGreen) + getHexString(mCurrentBlue); 47. return currentColor; 48. } 50. /\*\* 51. \* 根据fraction值来计算当前的颜色。 52. \*/ 53. private int getCurrentColor(int startColor, int endColor, int colorDiff, 54. int offset, float fraction) { 55. int currentColor; 56. if (startColor > endColor) { 57. currentColor = (int) (startColor - (fraction \* colorDiff - offset)); 58. if (currentColor < endColor) { 59. currentColor = endColor; 60. } 61. } else { 62. currentColor = (int) (startColor + (fraction \* colorDiff - offset)); 63. if (currentColor > endColor) { 64. currentColor = endColor; 65. } 66. } 67. return currentColor; 68. } 70. /\*\* 71. \* 将10进制颜色值转换成16进制。 72. \*/ 73. private String getHexString(int value) { 74. String hexString = Integer.toHexString(value); 75. if (hexString.length() == 1) { 76. hexString = "0" + hexString; 77. } 78. return hexString; 79. } 81. }   这大概是我们整个动画操作当中最复杂的一个类了。没错，属性动画的高级用法中最有技术含量的也就是如何编写出一个合适的TypeEvaluator。好在刚才我们已经编写了一个PointEvaluator，对它的基本工作原理已经有了了解，那么这里我们主要学习一下ColorEvaluator的逻辑流程吧。  首先在evaluate()方法当中获取到颜色的初始值和结束值，并通过字符串截取的方式将颜色分为RGB三个部分，并将RGB的值转换成十进制数字，那么每个颜色的取值范围就是0-255。接下来计算一下初始颜色值到结束颜色值之间的差值，这个差值很重要，决定着颜色变化的快慢，如果初始颜色值和结束颜色值很相近，那么颜色变化就会比较缓慢，而如果颜色值相差很大，比如说从黑到白，那么就要经历255\*3这个幅度的颜色过度，变化就会非常快。  那么控制颜色变化的速度是通过getCurrentColor()这个方法来实现的，这个方法会根据当前的fraction值来计算目前应该过度到什么颜色，并且这里会根据初始和结束的颜色差值来控制变化速度，最终将计算出的颜色进行返回。  最后，由于我们计算出的颜色是十进制数字，这里还需要调用一下getHexString()方法把它们转换成十六进制字符串，再将RGB颜色拼装起来之后作为最终的结果返回。  好了，ColorEvaluator写完之后我们就把最复杂的工作完成了，剩下的就是一些简单调用的问题了，比如说我们想要实现从蓝色到红色的动画过度，历时5秒，就可以这样写：  [java] view plain copy   1. ObjectAnimator anim = ObjectAnimator.ofObject(myAnimView, "color", new ColorEvaluator(), 2. "#0000FF", "#FF0000"); 3. anim.setDuration(5000); 4. anim.start();   用法非常简单易懂，相信不需要我再进行解释了。  接下来我们需要将上面一段代码移到MyAnimView类当中，让它和刚才的Point移动动画可以结合到一起播放，这就要借助我们在上篇文章当中学到的组合动画的技术了。修改MyAnimView中的代码，如下所示：  [java] view plain copy   1. public class MyAnimView extends View { 3. ... 5. private void startAnimation() { 6. Point startPoint = new Point(RADIUS, RADIUS); 7. Point endPoint = new Point(getWidth() - RADIUS, getHeight() - RADIUS); 8. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 9. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 10. @Override 11. public void onAnimationUpdate(ValueAnimator animation) { 12. currentPoint = (Point) animation.getAnimatedValue(); 13. invalidate(); 14. } 15. }); 16. ObjectAnimator anim2 = ObjectAnimator.ofObject(this, "color", new ColorEvaluator(), 17. "#0000FF", "#FF0000"); 18. AnimatorSet animSet = new AnimatorSet(); 19. animSet.play(anim).with(anim2); 20. animSet.setDuration(5000); 21. animSet.start(); 22. } 24. }   可以看到，我们并没有改动太多的代码，重点只是修改了startAnimation()方法中的部分内容。这里先是将颜色过度的代码逻辑移动到了startAnimation()方法当中，注意由于这段代码本身就是在MyAnimView当中执行的，因此ObjectAnimator.ofObject()的第一个参数直接传this就可以了。接着我们又创建了一个AnimatorSet，并把两个动画设置成同时播放，动画时长为五秒，最后启动动画。现在重新运行一下代码，效果如下图所示：    OK，位置动画和颜色动画非常融洽的结合到一起了，看上去效果还是相当不错的，这样我们就把ObjectAnimator的高级用法也掌握了。  Interpolator的用法  Interpolator这个东西很难进行翻译，直译过来的话是补间器的意思，它的主要作用是可以控制动画的变化速率，比如去实现一种非线性运动的动画效果。那么什么叫做非线性运动的动画效果呢？就是说动画改变的速率不是一成不变的，像加速运动以及减速运动都属于非线性运动。  不过Interpolator并不是属性动画中新增的技术，实际上从[android](http://lib.csdn.net/base/android) 1.0版本开始就一直存在Interpolator接口了，而之前的补间动画当然也是支持这个功能的。只不过在属性动画中新增了一个TimeInterpolator接口，这个接口是用于兼容之前的Interpolator的，这使得所有过去的Interpolator实现类都可以直接拿过来放到属性动画当中使用，那么我们来看一下现在TimeInterpolator接口的所有实现类，如下图所示：    可以看到，TimeInterpolator接口已经有非常多的实现类了，这些都是Android系统内置好的并且我们可以直接使用的Interpolator。每个Interpolator都有它各自的实现效果，比如说AccelerateInterpolator就是一个加速运动的Interpolator，而DecelerateInterpolator就是一个减速运动的Interpolator。  我觉得细心的朋友应该早已经发现了，在前面两篇文章当中我们所学到的所有属性动画，其实都不是在进行一种线程运动。比如说在“上”篇文章中使用ValueAnimator所打印的值如下所示：    可以看到，一开始的值变化速度明显比较慢，仅0.0开头的就打印了4次，之后开始加速，最后阶段又开始减速，因此我们可以很明显地看出这一个先加速后减速的Interpolator。  那么再来看一下在“中”篇文章中完成的小球移动加变色的功能，如下图所示：    从上图中我们明显可以看出，小球一开始运动速度比较慢，然后逐渐加速，中间的部分运动速度就比较快，接下来开始减速，最后缓缓停住。另外颜色变化也是这种规律，一开始颜色变化的比较慢，中间颜色变化的很快，最后阶段颜色变化的又比较慢。  从以上几点我们就可以总结出一个结论了，使用属性动画时，系统默认的Interpolator其实就是一个先加速后减速的Interpolator，对应的实现类就是AccelerateDecelerateInterpolator。  当然，我们也可以很轻松地修改这一默认属性，将它替换成任意一个系统内置好的Interpolator。就拿“中”篇文章中的代码来举例吧，MyAnimView中的startAnimation()方法是开启动画效果的入口，这里我们对Point对象的坐标稍做一下修改，让它变成一种垂直掉落的效果，代码如下所示：  [java] view plain copy   1. private void startAnimation() { 2. Point startPoint = new Point(getWidth() / 2, RADIUS); 3. Point endPoint = new Point(getWidth() / 2, getHeight() - RADIUS); 4. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 5. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 6. @Override 7. public void onAnimationUpdate(ValueAnimator animation) { 8. currentPoint = (Point) animation.getAnimatedValue(); 9. invalidate(); 10. } 11. }); 12. anim.setDuration(2000); 13. anim.start(); 14. }   这里主要是对Point构造函数中的坐标值进行了一下改动，那么现在小球运动的动画效果应该是从屏幕正中央的顶部掉落到底部。但是现在默认情况下小球的下降速度肯定是先加速后减速的，这不符合物理的常识规律，如果把小球视为一个自由落体的话，那么下降的速度应该是越来越快的。我们怎样才能改变这一默认行为呢？其实很简单，调用Animator的setInterpolator()方法就可以了，这个方法要求传入一个实现TimeInterpolator接口的实例，那么比如说我们想要实现小球下降越来越快的效果，就可以使用AccelerateInterpolator，代码如下所示：  [java] view plain copy   1. private void startAnimation() { 2. Point startPoint = new Point(getWidth() / 2, RADIUS); 3. Point endPoint = new Point(getWidth() / 2, getHeight() - RADIUS); 4. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 5. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 6. @Override 7. public void onAnimationUpdate(ValueAnimator animation) { 8. currentPoint = (Point) animation.getAnimatedValue(); 9. invalidate(); 10. } 11. }); 12. anim.setInterpolator(new AccelerateInterpolator(2f)); 13. anim.setDuration(2500); 14. anim.start(); 15. }   代码很简单，这里调用了setInterpolator()方法，然后传入了一个AccelerateInterpolator的实例，注意AccelerateInterpolator的构建函数可以接收一个float类型的参数，这个参数是用于控制加速度的。现在运行一下代码，效果如下图所示：    OK，效果非常明显，说明我们已经成功替换掉了默认的Interpolator，AccelerateInterpolator确实是生效了。但是现在的动画效果看上去仍然是怪怪的，因为一个小球从很高的地方掉落到地面上直接就静止了，这也是不符合物理规律的，小球撞击到地面之后应该要反弹起来，然后再次落下，接着再反弹起来，又再次落下，以此反复，最后静止。这个功能我们当然可以自己去写，只不过比较复杂，所幸的是，Android系统中已经提供好了这样一种Interpolator，我们只需要简单地替换一下就可以完成上面的描述的效果，代码如下所示：  [java] view plain copy   1. private void startAnimation() { 2. Point startPoint = new Point(getWidth() / 2, RADIUS); 3. Point endPoint = new Point(getWidth() / 2, getHeight() - RADIUS); 4. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 5. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 6. @Override 7. public void onAnimationUpdate(ValueAnimator animation) { 8. currentPoint = (Point) animation.getAnimatedValue(); 9. invalidate(); 10. } 11. }); 12. anim.setInterpolator(new BounceInterpolator()); 13. anim.setDuration(3000); 14. anim.start(); 15. }   可以看到，我们只是将设置的Interpolator换成了BounceInterpolator的实例，而BounceInterpolator就是一种可以模拟物理规律，实现反复弹起效果的Interpolator。另外还将整体的动画时间稍微延长了一点，因为小球反复弹起需要比之前更长的时间。现在重新运行一下代码，效果如下图所示：    OK！效果还是非常不错的。那么这里我们只是选了几个系统实现好的Interpolator，由于内置Interpolator非常多，就不一一进行讲解了，大家可以自己去使用一下其它的几种Interpolator来看一看效果。  但是，只会用一下系统提供好的Interpolator，我们显然对自己的要求就太低了，既然是学习属性动画的高级用法，那么自然要将它研究透了。下面我们就来看一下Interpolator的内部实现机制是什么样的，并且来尝试写一个自定义的Interpolator。  首先看一下TimeInterpolator的接口定义，代码如下所示：  [java] view plain copy   1. /\*\* 2. \* A time interpolator defines the rate of change of an animation. This allows animations 3. \* to have non-linear motion, such as acceleration and deceleration. 4. \*/ 5. public interface TimeInterpolator { 7. /\*\* 8. \* Maps a value representing the elapsed fraction of an animation to a value that represents 9. \* the interpolated fraction. This interpolated value is then multiplied by the change in 10. \* value of an animation to derive the animated value at the current elapsed animation time. 11. \* 12. \* @param input A value between 0 and 1.0 indicating our current point 13. \*        in the animation where 0 represents the start and 1.0 represents 14. \*        the end 15. \* @return The interpolation value. This value can be more than 1.0 for 16. \*         interpolators which overshoot their targets, or less than 0 for 17. \*         interpolators that undershoot their targets. 18. \*/ 19. float getInterpolation(float input); 20. }   OK，接口还是非常简单的，只有一个getInterpolation()方法。大家有兴趣可以通过注释来对这个接口进行详解的了解，这里我就简单解释一下，getInterpolation()方法中接收一个input参数，这个参数的值会随着动画的运行而不断变化，不过它的变化是非常有规律的，就是根据设定的动画时长匀速增加，变化范围是0到1。也就是说当动画一开始的时候input的值是0，到动画结束的时候input的值是1，而中间的值则是随着动画运行的时长在0到1之间变化的。  说到这个input的值，我觉得有不少朋友可能会联想到我们在“中”篇文章中使用过的fraction值。那么这里的input和fraction有什么关系或者区别呢？答案很简单，input的值决定了fraction的值。input的值是由系统经过计算后传入到getInterpolation()方法中的，然后我们可以自己实现getInterpolation()方法中的[算法](http://lib.csdn.net/base/datastructure)，根据input的值来计算出一个返回值，而这个返回值就是fraction了。  因此，最简单的情况就是input值和fraction值是相同的，这种情况由于input值是匀速增加的，因而fraction的值也是匀速增加的，所以动画的运动情况也是匀速的。系统中内置的LinearInterpolator就是一种匀速运动的Interpolator，那么我们来看一下它的源码是怎么实现的：  [java] view plain copy   1. /\*\* 2. \* An interpolator where the rate of change is constant 3. \*/ 4. @HasNativeInterpolator 5. public class LinearInterpolator extends BaseInterpolator implements NativeInterpolatorFactory { 7. public LinearInterpolator() { 8. } 10. public LinearInterpolator(Context context, AttributeSet attrs) { 11. } 13. public float getInterpolation(float input) { 14. return input; 15. } 17. /\*\* @hide \*/ 18. @Override 19. public long createNativeInterpolator() { 20. return NativeInterpolatorFactoryHelper.createLinearInterpolator(); 21. } 22. }   这里我们只看getInterpolation()方法，这个方法没有任何逻辑，就是把参数中传递的input值直接返回了，因此fraction的值就是等于input的值的，这就是匀速运动的Interpolator的实现方式。  当然这是最简单的一种Interpolator的实现了，我们再来看一个稍微复杂一点的。既然现在大家都知道了系统在默认情况下使用的是AccelerateDecelerateInterpolator，那我们就来看一下它的源码吧，如下所示：  [java] view plain copy   1. /\*\* 2. \* An interpolator where the rate of change starts and ends slowly but 3. \* accelerates through the middle. 4. \* 5. \*/ 6. @HasNativeInterpolator 7. public class AccelerateDecelerateInterpolator implements Interpolator, NativeInterpolatorFactory { 8. public AccelerateDecelerateInterpolator() { 9. } 11. @SuppressWarnings({"UnusedDeclaration"}) 12. public AccelerateDecelerateInterpolator(Context context, AttributeSet attrs) { 13. } 15. public float getInterpolation(float input) { 16. return (float)(Math.cos((input + 1) \* Math.PI) / 2.0f) + 0.5f; 17. } 19. /\*\* @hide \*/ 20. @Override 21. public long createNativeInterpolator() { 22. return NativeInterpolatorFactoryHelper.createAccelerateDecelerateInterpolator(); 23. } 24. }   代码虽然没有变长很多，但是getInterpolation()方法中的逻辑已经明显变复杂了，不再是简单地将参数中的input进行返回，而是进行了一个较为复杂的数学运算。那这里我们来分析一下它的算法实现，可以看到，算法中主要使用了余弦函数，由于input的取值范围是0到1，那么cos函数中的取值范围就是π到2π。而cos(π)的结果是-1，cos(2π)的结果是1，那么这个值再除以2加上0.5之后，getInterpolation()方法最终返回的结果值还是在0到1之间。只不过经过了余弦运算之后，最终的结果不再是匀速增加的了，而是经历了一个先加速后减速的过程。我们可以将这个算法的执行情况通过曲线图的方式绘制出来，结果如下图所示：    可以看到，这是一个S型的曲线图，当横坐标从0变化到0.2的时候，纵坐标的变化幅度很小，但是之后就开始明显加速，最后横坐标从0.8变化到1的时候，纵坐标的变化幅度又变得很小。  OK，通过分析LinearInterpolator和AccelerateDecelerateInterpolator的源码，我们已经对Interpolator的内部实现机制有了比较清楚的认识了，那么接下来我们就开始尝试编写一个自定义的Interpolator。  编写自定义Interpolator最主要的难度都是在于数学计算方面的，由于我数学并不是很好，因此这里也就写一个简单点的Interpolator来给大家演示一下。既然属性动画默认的Interpolator是先加速后减速的一种方式，这里我们就对它进行一个简单的修改，让它变成先减速后加速的方式。新建DecelerateAccelerateInterpolator类，让它实现TimeInterpolator接口，代码如下所示：  [java] view plain copy   1. public class DecelerateAccelerateInterpolator implements TimeInterpolator{ 3. @Override 4. public float getInterpolation(float input) { 5. float result; 6. if (input <= 0.5) { 7. result = (float) (Math.sin(Math.PI \* input)) / 2; 8. } else { 9. result = (float) (2 - Math.sin(Math.PI \* input)) / 2; 10. } 11. return result; 12. } 13. }   这段代码是使用正弦函数来实现先减速后加速的功能的，因为正弦函数初始弧度的变化值非常大，刚好和余弦函数是相反的，而随着弧度的增加，正弦函数的变化值也会逐渐变小，这样也就实现了减速的效果。当弧度大于π/2之后，整个过程相反了过来，现在正弦函数的弧度变化值非常小，渐渐随着弧度继续增加，变化值越来越大，弧度到π时结束，这样从0过度到π，也就实现了先减速后加速的效果。  同样我们可以将这个算法的执行情况通过曲线图的方式绘制出来，结果如下图所示：    可以看到，这也是一个S型的曲线图，只不过曲线的方向和刚才是相反的。从上图中我们可以很清楚地看出来，一开始纵坐标的变化幅度很大，然后逐渐变小，横坐标到0.5的时候纵坐标变化幅度趋近于零，之后随着横坐标继续增加纵坐标的变化幅度又开始变大，的确是先减速后加速的效果。  那么现在我们将DecelerateAccelerateInterpolator在代码中进行替换，如下所示：  [java] view plain copy   1. private void startAnimation() { 2. Point startPoint = new Point(getWidth() / 2, RADIUS); 3. Point endPoint = new Point(getWidth() / 2, getHeight() - RADIUS); 4. ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint); 5. anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() { 6. @Override 7. public void onAnimationUpdate(ValueAnimator animation) { 8. currentPoint = (Point) animation.getAnimatedValue(); 9. invalidate(); 10. } 11. }); 12. anim.setInterpolator(new DecelerateAccelerateInterpolator()); 13. anim.setDuration(3000); 14. anim.start(); 15. }   非常简单，就是将DecelerateAccelerateInterpolator的实例传入到setInterpolator()方法当中。重新运行一下代码，效果如下图所示：    OK！小球的运动确实是先减速后加速的效果，说明我们自定义的Interpolator已经可以正常工作了。通过这样一个程度的学习，相信大家对属性动画Interpolator的理解和使用都达到了一个比较深刻的层次了。  ViewPropertyAnimator的用法  ViewPropertyAnimator其实算不上什么高级技巧，它的用法格外的简单，只不过和前面所学的所有属性动画的知识不同，它并不是在3.0系统当中引入的，而是在3.1系统当中附增的一个新的功能，因此这里我们把它作为整个属性动画系列的收尾部分。  我们都知道，属性动画的机制已经不是再针对于View而进行设计的了，而是一种不断地对值进行操作的机制，它可以将值赋值到指定对象的指定属性上。但是，在绝大多数情况下，我相信大家主要都还是对View进行动画操作的。Android开发团队也是意识到了这一点，没有为View的动画操作提供一种更加便捷的用法确实是有点太不人性化了，于是在Android 3.1系统当中补充了ViewPropertyAnimator这个机制。  那我们先来回顾一下之前的用法吧，比如我们想要让一个TextView从常规状态变成透明状态，就可以这样写：  [java] view plain copy   1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "alpha", 0f); 2. animator.start();   看上去复杂吗？好像也不怎么复杂，但确实也不怎么容易理解。我们要将操作的view、属性、变化的值都一起传入到ObjectAnimator.ofFloat()方法当中，虽然看上去也没写几行代码，但这不太像是我们平时使用的面向对象的思维。  那么下面我们就来看一下如何使用ViewPropertyAnimator来实现同样的效果，ViewPropertyAnimator提供了更加易懂、更加面向对象的API，如下所示：  [java] view plain copy   1. textview.animate().alpha(0f);   果然非常简单！不过textview.animate()这个方法是怎么回事呢？animate()方法就是在Android 3.1系统上新增的一个方法，这个方法的返回值是一个ViewPropertyAnimator对象，也就是说拿到这个对象之后我们就可以调用它的各种方法来实现动画效果了，这里我们调用了alpha()方法并转入0，表示将当前的textview变成透明状态。  怎么样？比起使用ObjectAnimator，ViewPropertyAnimator的用法明显更加简单易懂吧。除此之外，ViewPropertyAnimator还可以很轻松地将多个动画组合到一起，比如我们想要让textview运动到500,500这个坐标点上，就可以这样写：  [java] view plain copy   1. textview.animate().x(500).y(500);   可以看出，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。  那么怎样去设定动画的运行时长呢？很简单，也是通过连缀的方式设定即可，比如我们想要让动画运行5秒钟，就可以这样写：  [java] view plain copy   1. textview.animate().x(500).y(500).setDuration(5000);   除此之外，本篇文章第一部分所学的Interpolator技术我们也可以应用在ViewPropertyAnimator上面，如下所示：  [java] view plain copy   1. textview.animate().x(500).y(500).setDuration(5000) 2. .setInterpolator(new BounceInterpolator());   用法很简单，同样也是使用连缀的方式。相信大家现在都已经体验出来了，ViewPropertyAnimator其实并没有什么太多的技巧可言，用法基本都是大同小异的，需要用到什么功能就连缀一下，因此更多的用法大家只需要去查阅一下文档，看看还支持哪些功能，有哪些接口可以调用就可以了。  那么除了用法之外，关于ViewPropertyAnimator有几个细节还是值得大家注意一下的：   1. 整个ViewPropertyAnimator的功能都是建立在View类新增的animate()方法之上的，这个方法会创建并返回一个ViewPropertyAnimator的实例，之后的调用的所有方法，设置的所有属性都是通过这个实例完成的。 2. 大家注意到，在使用ViewPropertyAnimator时，我们自始至终没有调用过start()方法，这是因为新的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之后，动画就会自动启动。并且这个机制对于组合动画也同样有效，只要我们不断地连缀新的方法，那么动画就不会立刻执行，等到所有在ViewPropertyAnimator上设置的方法都执行完毕后，动画就会自动启动。当然如果不想使用这一默认机制的话，我们也可以显式地调用start()方法来启动动画。 3. ViewPropertyAnimator的所有接口都是使用连缀的语法来设计的，每个方法的返回值都是它自身的实例，因此调用完一个方法之后可以直接连缀调用它的另一个方法，这样把所有的功能都串接起来，我们甚至可以仅通过一行代码就完成任意复杂度的动画功能。   好的，那么到这里为止，整个Android属性动画完全解析的系列就全部结束了，感谢大家有耐心看到最后。 Arraylist基于数组，是怎么实现动态增删功能 [深入Java集合学习系列：ArrayList的实现原理](http://zhangshixi.iteye.com/blog/674856)  1. ArrayList概述：     ArrayList是List接口的可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。     每个ArrayList实例都有一个容量，该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向ArrayList中不断添加元素，其容量也自动增长。自动增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造ArrayList时指定其容量。在添加大量元素前，应用程序也可以使用ensureCapacity操作来增加ArrayList实例的容量，这可以减少递增式再分配的数量。     注意，此实现不是同步的。如果多个线程同时访问一个ArrayList实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。    2. ArrayList的实现：     对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析ArrayList的源代码：     1) 底层使用数组实现：  private transient Object[] elementData;      2) 构造方法：     ArrayList提供了三种方式的构造器，可以构造一个默认初始容量为10的空列表、构造一个指定初始容量的空列表以及构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。   1. public ArrayList() { 2. this(10); 3. } 5. public ArrayList(int initialCapacity) { 6. super(); 7. if (initialCapacity < 0) 8. throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity); 9. this.elementData = new Object[initialCapacity]; 10. } 12. public ArrayList(Collection<? extends E> c) { 13. elementData = c.toArray(); 14. size = elementData.length; 15. // c.toArray might (incorrectly) not return Object[] (see 6260652) 16. if (elementData.getClass() != Object[].class) 17. elementData = Arrays.copyOf(elementData, size, Object[].class); 18. }       3) 存储：     ArrayList提供了set(int index, E element)、add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)这些添加元素的方法。下面我们一一讲解：  Java代码     1. // 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。 2. public E set(int index, E element) { 3. RangeCheck(index); 5. E oldValue = (E) elementData[index]; 6. elementData[index] = element; 7. return oldValue; 8. }   Java代码     1. // 将指定的元素添加到此列表的尾部。 2. public boolean add(E e) { 3. ensureCapacity(size + 1); 4. elementData[size++] = e; 5. return true; 6. }   Java代码     1. // 将指定的元素插入此列表中的指定位置。 2. // 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。 3. public void add(int index, E element) { 4. if (index > size || index < 0) 5. throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size); 6. // 如果数组长度不足，将进行扩容。 7. ensureCapacity(size+1);  // Increments modCount!! 8. // 将 elementData中从Index位置开始、长度为size-index的元素， 9. // 拷贝到从下标为index+1位置开始的新的elementData数组中。 10. // 即将当前位于该位置的元素以及所有后续元素右移一个位置。 11. System.arraycopy(elementData, index, elementData, index + 1, size - index); 12. elementData[index] = element; 13. size++; 14. }   Java代码     1. // 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。 2. public boolean addAll(Collection<? extends E> c) { 3. Object[] a = c.toArray(); 4. int numNew = a.length; 5. ensureCapacity(size + numNew);  // Increments modCount 6. System.arraycopy(a, 0, elementData, size, numNew); 7. size += numNew; 8. return numNew != 0; 9. }   Java代码     1. // 从指定的位置开始，将指定collection中的所有元素插入到此列表中。 2. public boolean addAll(int index, Collection<? extends E> c) { 3. if (index > size || index < 0) 4. throw new IndexOutOfBoundsException( 5. "Index: " + index + ", Size: " + size); 7. Object[] a = c.toArray(); 8. int numNew = a.length; 9. ensureCapacity(size + numNew);  // Increments modCount 11. int numMoved = size - index; 12. if (numMoved > 0) 13. System.arraycopy(elementData, index, elementData, index + numNew, numMoved); 15. System.arraycopy(a, 0, elementData, index, numNew); 16. size += numNew; 17. return numNew != 0; 18. }       4) 读取：  Java代码     1. // 返回此列表中指定位置上的元素。 2. public E get(int index) { 3. RangeCheck(index); 5. return (E) elementData[index]; 6. }       5) 删除：     ArrayList提供了根据下标或者指定对象两种方式的删除功能。如下：  Java代码     1. // 移除此列表中指定位置上的元素。 2. public E remove(int index) { 3. RangeCheck(index); 5. modCount++; 6. E oldValue = (E) elementData[index]; 8. int numMoved = size - index - 1; 9. if (numMoved > 0) 10. System.arraycopy(elementData, index+1, elementData, index, numMoved); 11. elementData[--size] = null; // Let gc do its work 13. return oldValue; 14. }   Java代码     1. // 移除此列表中首次出现的指定元素（如果存在）。这是应为ArrayList中允许存放重复的元素。 2. public boolean remove(Object o) { 3. // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。 4. if (o == null) { 5. for (int index = 0; index < size; index++) 6. if (elementData[index] == null) { 7. // 类似remove(int index)，移除列表中指定位置上的元素。 8. fastRemove(index); 9. return true; 10. } 11. } else { 12. for (int index = 0; index < size; index++) 13. if (o.equals(elementData[index])) { 14. fastRemove(index); 15. return true; 16. } 17. } 18. return false; 19. }       注意：从数组中移除元素的操作，也会导致被移除的元素以后的所有元素的向左移动一个位置。     6) 调整数组容量：     从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。  Java代码     1. public void ensureCapacity(int minCapacity) { 2. modCount++; 3. int oldCapacity = elementData.length; 4. if (minCapacity > oldCapacity) { 5. Object oldData[] = elementData; 6. int newCapacity = (oldCapacity \* 3)/2 + 1; 7. if (newCapacity < minCapacity) 8. newCapacity = minCapacity; 9. // minCapacity is usually close to size, so this is a win: 10. elementData = Arrays.copyOf(elementData, newCapacity); 11. } 12. }      从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。     ArrayList还给我们提供了将底层数组的容量调整为当前列表保存的实际元素的大小的功能。它可以通过trimToSize方法来实现。代码如下：  Java代码     1. public void trimToSize() { 2. modCount++; 3. int oldCapacity = elementData.length; 4. if (size < oldCapacity) { 5. elementData = Arrays.copyOf(elementData, size); 6. } 7. }      7) Fail-Fast机制：  ArrayList也采用了快速失败的机制，通过记录modCount参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。具体介绍请参考我之前的文章[深入Java集合学习系列：HashMap的实现原理](http://zhangshixi.iteye.com/blog/672697) 中的Fail-Fast机制。 |