多线程以及异步任务

1. Android的多线程

1.1 Android app的主线程 & ANR

Android中有主线程的概念，也就是UI线程。在App启动的时候，运行在一个进程中，然后在其中创建一条主线程，主要负责四大组件与用户交互的事情(比如UI与页面交互相关的工作)。然而因为用户随时都会与界面交互，为了用户体验，主线程要保持响应。如果我们在主线程中进行了耗时的操作，阻塞了主线程，此时用户与界面进行交互，UI一段时间内没有响应。为了体验的一致，系统就会报出ANR(Application NOT Responding)异常提示用户。一般这个超时的时间是Broadcast中10秒，按键无响应5秒，前台Service20秒以及后台Service200秒。

1.2 开启子线程(工作线程)

为了避免阻塞到主线程，我们需要使用多线程编程。也就是要将耗时的任务放到子线程中运行，那么我们就需要开启子线程。一般有两种方法开启一个新的线程并运行。

- **继承Thread类**

```
class MyThread extends Thread{
    @Override
    public void run(){
        //子线程处理的事情
    }
}

new MyThread().start();
```

- **实现Runnable接口**

```
class MyThread implement Runnable{
    @Override
    public void run(){
        //子线程处理的事情
    }
}

MyThread myThread = new MyThread();
new Thread(myThread).start();
```

- **与第二种方法其实一样，还经常使用一种匿名类的写法**
  这里的匿名类其实是定义并且实例化了一个继承Runnable接口的类，作为参数传入了Thread的构

造函数中。

```
new Thread(new Runnable(){
    @Override
    public void run(){
        //子线程处理的事情
    }
}).start();
```

1.3 线程间通信

在Android中，为了避免多线程同事修改UI造成的线程同步以及线程安全问题。设计成只有主线程才能修改UI。(ViewRootImpl创建之后才能检查线程是否是主线程，在这之前用子线程去访问UI控件并不会报错，但是也不好。)
而我们的子线程完成了任务之后，需要如果需要更新UI，则需要将内容传递回主线程，这时候就需要完成线程中的通信。我们一般有多重方法来传递消息给主线程，他们原理上都是基于Android的handler机制完成的。handler机制的使用在前面的四大组件与handler学习笔记中有相关的介绍。

- **handler.sendXXXMessage()等方法**

在需要接受消息的线程A(可以使主线程)中创建一个Handler（在这之前需要先启动Loop循环，主线程中已经启动了这个循环），并且重写handleMessage方法，处理收到的Message。
然后在发送消息的线程B中取得线程A的handler的引用执行sendXXXMessage方法，将消息传递给线程A的handler处理。
比如下面

```
//先在Acitivy中创建一个handler
Handler mHandler = new Handler(){
    @Override
    public void handlerMessage(Message msg){
        //这里接受message后修改UI
    }
}
//这里创建Handler的方法应该使用静态内部类加上弱引用的方式完成。

//再这样子创建工作线程
mRunnable = new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);//模拟耗时任务
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                mHandler.sendXXXMessage(0x123);

            }
        };
```

- **handler.post(Runnable)**

将sendXXmessage方法改成这个post方法即可。

```
mHandler.post(new Runnable() {
    @Override
    public void run() {
        text.setText("Task Done!!");
        }
 });
```

- **Activity.runOnUIThread(Runnable runnable)**

直接在子线程中调用这个方法将Runnable对象发送给主线程执行。不用定义Handler

- **View.post(Runnable)**

```
text.post(new Runnable() {
    @Override
    public void run() {
        text.setText("Task Done!!");
    }
});
```

这种方式也不用定义Handler。

注意事项： 在子线程中使用Looper的时候，应该是使用完后调用quit()方法退出loop循环，否则会因为无法退出循环而导致内存泄漏。

1.4 线程间同步

除去在Android中使用其独有的handler机制来进行线程间的消息传递。我们还可以使用java中自带的线程同步方式来在让Android中的多线程完成同步。

1.4.1 Synchronized关键字

在java中，每一个对象都有内置锁，使用synchronized关键字可以控制对象锁。

- **锁方法(对象锁)**
  使用synchronized关键字修饰方法，会锁住这个调用这个方法的对象。每个synchronized方法都必须获得调用该方法的类实例的”锁“方能执行，否则所属线程阻塞。
- **锁代码块(对象锁)**
  锁代码块会锁住括号内指定的对象实例，线程必须要获得对应的对象锁才能执行代码块中的代码。
- **锁静态方法(类锁)**
  因为静态方法不需要实例化就可以调用，静态方法需要的是这个类的.class对象。当锁住静态方法，那么线程必须要获得这个class对象的锁才能运行这个静态方法。

- **锁代码块.class(类锁)**
  同理，如果所代码块中的括号中写的是.class对象，那么锁住的是这个类。

synchronized关键字的缺点： 不支持中断，获取锁时不能设定超时，每个锁只能有单一的条件。

1.4.2 对象锁的notify和wait方法

锁池中的线程可以去竞争锁，等待池中的线程只能等待notify或者notifyAll的唤醒。

如果对象A调用了wait方法，那么现在持有A的对象锁的线程会放出这个锁，然后进入这个对象的等待池中。
如果对象A调用了notify方法，就会通知某个正在等待这个对象锁的线程可以去锁池中可以竞争这个对象锁并开始运行。
如果对象A调用了notifyAll方法，就会通知所有正在等待这个对象锁的线程可以进入锁池中开始竞争锁。
锁池中竞争失败的线程并不会回到等待池，而是等待前面持有锁的对象执行完放出对象锁后再次竞争。

1.4.3 volatile变量

我们可以使用volatile关键字修饰变量，用于告诉虚拟机，这个变量可能会被其他线程更新。线程会放弃在自己的local memory中维持这个变量的副本。而是每次使用都去main memory中读取和修改。但是Volatile只能保证可见性，但是不能保证原子性。
所以我们一般在 **一个变量更新其值的时候不依赖当前值，而且该变量不会和其他仪器构成一个不可见条件。**

1.4.4 使用重入锁

java中有java.util.concurrent包来支持同步。里面有ReentrantLock来实现线程同步。还能支持Condition。

可以使用lock和unlock方法，加锁和释放锁。一般记得在try语句的finally中使用unlock释放锁。而且锁是可重入的，因为线程可以重复地获取已经持有的锁，它保持一个持有计数（hold count）来跟踪对lock方法的嵌套调用。线程每次调用lock都要调用unlock方法来释放锁，由于这个特性，被一个锁保护的代码可以调用另一个使用相同锁的方法。

- **支持Condition**
  我们可以使用一个条件对象来管理已经获得了一个锁却没有做有用工作的线程。使用lock.newCondition方法可以获取一个条件对象，当发现不满足条件时候，调用condition.await() 方法，那么当前的线程发现不满足条件，被阻塞并且放弃了锁，进入了这个条件的等待池。需要等待同一个条件调用signalAll才能再进入锁池参与锁的竞争。如果所有的线程都进入了等待池而且没有线程来使用signal或者signalAll来唤醒他们。这一点和Synchronized的notify使用很相似。

1.5 线程池

我们有的时候，需要创建的子线程非常多，而频繁的创建和销毁线程需要时间。此时希望对线程进行管理和复用，就可以使用线程池来进行了。

1.5.1 ThreadPoolExecutor类

这个类是线程池中最核心的类，其中最重要的构造方法如下，其他的构造方法都是复用的这个方法：

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit
        BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler
```

- **corePoolSize：** 核心池的大小，在创建了线程池后，默认情况下，线程池中没有任何的线程，而是等待任务的到来才创建线程执行，在没调用prestartCoreThread方法的情况下，一般默认任务来了才创建线程，当创建的线程数量到达corePoolSize的时候，就会把达到的任务放到缓存队列中。
- **maximumPoolSize：** 线程的最大线程数，表示在线程池中最多能够创建多少个线程。
- **keepAliveTime：** 表示线程没有任务到来的时候，最长保持线程不被回收销毁的时间。默认情况下，只有当线程池中的线程数量大于corePoolSize的时候，KeepAliveTime才会起作用，直到线程数不大于corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0。
- **unit：** 参数keepAliveTime的时间单位，七种取值。

```
TimeUnit.DAYS;               //天
TimeUnit.HOURS;             //小时
TimeUnit.MINUTES;           //分钟
TimeUnit.SECONDS;           //秒
TimeUnit.MILLISECONDS;      //毫秒
TimeUnit.MICROSECONDS;      //微妙
TimeUnit.NANOSECONDS;       //纳秒
```

- **workQueue:** 阻塞队列，用来储存等待执行的任务，有几种选择， 比如 `ArrayBlockingQueue` ，`LinkedBlockingQueue` ，`SynchronousQueue` 。阻塞队列与线程池的排队策略有关，一般使用LinkedBlockingQueue和SynchronousQueue。

SynchronousQueue，不对任务进行缓存，而是直接提交，当新的任务进入线程池，却没有空余的线程时，创建一个新线程处理这个任务。这种策略需要线程池具有无限增长的可能。

LinkedBlockingQueue，是无界队列，当所有的corePoolSize都忙时，新任务在队列中等待。创建的线程数量不会超过corePoolSize。此时maximumPoolSize的值是无效的。当每个任务完全独立于其他的任务时，适合使用无界队列。

- **threadFactory：** 线程工厂，主要用于创建线程；
- **handler：** 表示当拒绝任务时候的策略，一般有以下几种：

```
ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务，这个策略不想放弃任务。但是由于线程池已经
```

Executor是一个顶层的接口，里面只有一个excutor(Runnable)方法，用于执行任务。
然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等等。
抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中的所有方法。
然后ThreadPoolexecutor继承了类AbstractExecutorService。
在其中有几个很重要的方法：

```
executor()
//这个方法是在ThreadPoolexecutor中实现的。通过这个方法可以向线程池提交一个任务，交给线程池去完成。
submit()
//在ExecutorService中声明的方法，在AbstractExecutorService中具体实现，在后面并没有重写。
//也可以用来向线程池提交任务，但是能返回执行的结果，其中还是调用了executor方法，只不过使用Future获取
shutdown()
shutdownNow()
//上面的这两个方法都是用来关闭线程池的。
```

1.5.2 线程池的状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量来表示线程的几个状态如下：

```
    private static final int RUNNING    = -1 << COUNT_BITS;
    private static final int SHUTDOWN   =  0 << COUNT_BITS;
    private static final int STOP       =  1 << COUNT_BITS;
    private static final int TIDYING    =  2 << COUNT_BITS;
    private static final int TERMINATED =  3 << COUNT_BITS;
    //runState和workerCount两个数字打包在一起。
```

runState表示线程池当前的状态，是一个volatile变量来保证线程的可见性。下面的static final变量表示runState的几个取值。当线程池创建之后，初始的时候，线程池处于RUNNING状态；
如果调用了shutdown方法，线程池就处于SHUTDOWN状态，此时的线程池不能继续接受新的任务，但是会等待目前还在执行的任务执行完毕。
如果调用了shutdownNow的方法，线程池就会处于STOP状态，此时的线程池不会接受新的任务，而且会尝试终止正在执行的任务。
当线程处于SHUTDOWN或者STOP状态，并且所有的工作线程已经销毁，任务缓存队列已经清空或者执行结束后，线程池会被置为TERMINATED状态。

1.5.3 线程池的主要从成员变量

```
private final BlockingQueue<Runnable> workQueue;
//任务缓存队列，同来存放等待执行的任务。
private final ReentrantLock mainLock = new ReentrantLock();
//线程池的主要状态锁，对任务池的状态，比如大小，runState的更改都需要使用这个锁。
private final HashSet<Worker> workers = new HashSet<Worker>();
//用于存放工作集
private volatile long keepAliveTime; //线程存货时间。
private volatile boolean allowCoreThreadTimeOut; //是否允许为核心线程设置存活时间。
private volatile int corePoolSize; //核心池的大小。
private volatile int maximumPoolSize; //线程池最大能容纳的线程数。
private volatile int poolSize; //线程池当前的线程数。
private volatile RejectedExecutionHandler handler;//任务拒绝策略。
private volatile ThreadFactory threadFactory; // 线程工厂，用于创建线程。
private int largestPoolSize; //用于记录线程池中曾经出现过的最大线程数。
private long completedTaskCount; //用于记录已经执行完毕的任务数量
```

1.5.4 线程池的关键方法实现

线程池执行任务调用的就是execute方法，下面便是 java1.8 中的源码：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
        int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    else if (!addWorker(command, false))
        reject(command);
}
//ctl的定义是这个，当线程池刚创建的时候，状态为RUNNING，有效线程数为0。
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//打包方法ctlOf，解包方法为：
private static int runStateOf(int c)     { return c & ~CAPACITY; }
private static int workerCountOf(int c)  { return c & CAPACITY; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
```

观察源码，可以看到，首先对输入的Runnable对象执行了一次判空，ctl是一个原子integer类，其中打包了两个数，一个是workerCount，表示有效的线程数；第二个是runState，表示目前的线程池状态。其中的COUNT_BITS的值就是32-3=29，然后将CAPACITY是1左移29位并-1，也就成了一个低位29位

的掩码。那么很显然高3位就用来保存有5种状态需要3位二进制来保存的线程池状态runState。
然后分为了三步：

- (1) 首先，判断如果正在运行的线程数小于corePoolSize，那么尝试创建一个新的线程，并且放入这个command作为新线程的第一个任务。addWorker方法在检查runState和workerCount保证了原子性，返回false以避免在不应该添加线程的时候发出错误警报。
- (2)如果一个任务成功进入排队，那么我们仍然需要检查是否应该添加一个新的线程。（因为现有线程自上次检查来已经死掉了）或者这个线程池在进入这个方法的时候刚好已经进入了SHUTDOWN状态。不然就在没有线程的时候，开启一个新线程。
- (3)如果我们不能排队任务，那我们尝试添加一个线程。如果失败了，那我们就知道我们已经SHUTDOWN状态或者是饱和了，就应该拒绝这个任务。

2. AsyncTask (异步任务)

除去手动使用Handler机制来完成异步任务的要求，我们还能使用Android中封装好的一个异步任务类来达到同样的效果。

2.1 主要的相关方法

在使用异步任务前，首先需要继承AsyncTask<Params, progress, Result>

1. `execute(Params...)`
   这个方法是异步任务的入口，应该在主线程中调用，调用这个方法的时候，会让任务以单线程队列的方式或者线程池队列的方式运行，其中Params为前面定义中指定的参数。
2. `void onPreExecute()`
   这个方法是在execute方法执行后立即执行的，我们可以重写，主要用来完成在开启线程执行异步任务之前的操作，比如让一个进度条可见之类的。
3. `Result doInBackground(Params)`
   这个方法我们必须重写，因为真正的异步任务就是放在这个方法中进行的，在onPreExecute方法之后立即执行。因为这个方法内的代码是在子线程中完成的，我们可以在里面执行耗时长的任务，而不应该在其他的方法中执行耗时任务，因为其他方法都是在主线程中的。这个方法接受输入参数，也就是execute中输入的参数。并且可以返回结果，在执行的期间，可以调用publishProgress(Progress... values)方法实时地将信息传入onProgressUpdate(Progress...values)方法。
4. `void onProgressUpdate(Progress...)`
   在doInBackground中调用publishProgress传递的信息传到这个方法接受，用于更新UI。同样的，这个方法执行的线程也在主线程。
5. `void onPostExecute(Result)`
   这个方法是在UI线程中执行的，接收doInBackground方法返回的参数，处理异步任务所得到的的结果。

2.2 注意事项

- 异步任务的子类的实例应该在UI线程中创建。
- 除了execute方法之外，doInBackground、onPreExecute、onProgressUpdate以及onPostExecute方法都不能手动调用。而publicProgress方法只能在DoInBackground方法中调用。
- doInBackground这个方法是在另外的子线程中执行的，所以在这个方法中不能操作UI。只能通过publishProgress将参数发给onProgressUpdate方法在UI线程处理。
- 每一个任务实例只能执行一次。

异步任务主要还是对我们常需要做的异步任务模型做了一层封装，让我们能聚焦于异步任务的每一步操作，而不用操心线程间通信的问题。

3. Java的异步任务回传结果

前面有提到Android中的异步任务，可以手动使用Handler实现多线程之间的消息传递以及回传执行的结果给主线程。
但是Handler是Android中的独有实现，在Java中其实也支持了多线程之间的线程通信。

我们就需要使用Callable、Future和FutureTask，来将子线程执行的结果传回给其他线程。

3.1 Callable

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

这是一个泛型接口，其中唯一的方法call返回的就是泛型的类型。

3.2 Future

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    //用于取消任务，如果取消任务成功则返回true，如果取消失败则返回false。
    //参数表示的是是都允许取消正在执行还没执行完的任务，如果设置true，表示可以。
    //如果任务已经完成，这个方法一定返回false;
    //如果任务正在执行,若mayInterruptIfRunning设置为true，则返回true
    //若mayInterruptIfRunning设置为false，则返回false
    //如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。
    boolean isCancelled();
    //表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
    boolean isDone();
    //表示任务是否已经完成，若任务完成，则返回true;
    V get() throws InterruptedException, ExecutionException;
    //用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回;
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, Timeout
        //用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。
}
```

Future也是一个泛型接口，一般用于对具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成等等。有必要的时候可以通过get方法取得执行的结果，这个方法会阻塞直到任务完成取到结果，返回的结果类型就是泛型的类型。
Future泛型接口一共提供了三种功能：判断任务是否完成；中断任务；获取任务的执行结果。

3.3 FutureTask

FutureTask实现了RunnableFuture接口，而这个接口又继承了Runnable和Future。也就是间接实现了Runnable和Future接口。
Future的构造方法如下：

```
public FutureTask(Callable<V> callable) {
public FutureTask(Runnable runnable, V result)
```

3.4 使用Callable和Future配合ExecutorService获取执行结果

首先新建一个ExecutorService。
定义一个task类继承实现Callable接口，其中定义自己想要完成的任务。
将想要执行的Callable任务传入executor的submit方法执行，使用Future result 接收执行的结果。就能关闭executor了。
调用result.get()方法来取得执行的结果并处理。

3.5 Callable和FutureTask来获取执行结果

首先新建一个ExecutorService。
定义一个task类继承实现Callable接口，其中定义自己想要完成的任务。
新建一个FutureTask并将task实例传入构造函数。

将FutureTask实例作为任务传入executor的submit函数。(这一步也可以换成将FutureTask作为参数传入Thread的构造函数并start。相对应的，就不需要executor了)
然后调用FutureTask实例的get方法获取结果并处理。