线程、多线程与线程池总结

(转自http://www.jianshu.com/p/b8197dd2934c)

先看几个概念：

线程：进程中负责程序执行的执行单元。一个进程中至少有一个线程。

多线程：解决多任务同时执行的需求，合理使用CPU资源。多线程的运行是根据CPU切换完成，如何切换由CPU决定，因此多线程运行具有不确定性。

线程池：基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

如果对线程概念不清晰的话，不妨先看看我是一个线程这篇文章，拟人化的故事阐述线程的工作原理。

● 线程

创建线程的两种方式：

一、继承Thread类，扩展线程(实现Runnable接口)。

```
class DemoThread extends Thread {

    @Override
    public void run() {
        super.run();
        // Perform time-consuming operation...
    }
}

DemoThread t = new DemoThread();
t.start();
```

- 继承Thread类，覆盖run()方法。
- 创建线程对象并用start()方法启动线程。

面试题

- 1）线程和进程有什么区别？

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

- 2）如何在Java中实现线程？

创建线程有两种方式：

一、继承 Thread 类，扩展线程。

二、实现 Runnable 接口。

- 3）Thread 类中的 start() 和 run() 方法有什么区别？

调用 start() 方法才会启动新线程；如果直接调用 Thread 的 run() 方法，它的行为就会和普通的方法一样；为了在新的线程中执行我们的代码，必须使用 Thread.start() 方法。

扩展

Android 系统接口 HandlerThread 继承了 Thread，它是一个可以使用 Handler 的 Thread，一个具有消息循环的线程。run()方法中通过 Looper.prepare() 来创建消息队列，通过 Looper.loop() 来开启消息循环。可以在 run() 方法中执行耗时的任务，而 HandlerThread 内部创建了消息队列外界需要通过 Handler 的方式来通知 HandlerThread 执行一个具体任务；HandlerThread 的 run() 方法是一个无限的循环，可以通过它的 quite() 或 quitSafely() 方法来终止线程的执行；

二、实现Runnable接口。

```
public class DemoActivity extends BaseActivity implements Runnable {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Thread t = new Thread(this);
        t.start();
    }

    @Override
    public void run() {

    }
}
```

面试题

- 1）用 Runnable 还是 Thread？

我们都知道可以通过继承 Thread 类或者调用 Runnable 接口来实现线程，问题是，创建线程哪种方式更好呢？什么情况下使用它？这个问题很容易回答，如果你知道Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口更好了。

- 2）Runnable 和 Callable 有什么不同？

Runnable 和 Callable 都代表那些要在不同的线程中执行的任务。Runnable 从 JDK1.0 开始就有了，Callable 是在 JDK1.5 增加的。它们的主要区别是 Callable 的 call() 方法可以返回值和抛出异常，而 Runnable 的 run() 方法没有这些功能。Callable 可以返回装载有计算结果的 Future 对象。

注意：这面第二个面试题主要是为了引出下面的扩展，原谅我这样为难人的出场。

扩展

先看一下 Runnable 和 Callable 的源码

```
public interface Runnable {
    public void run();
}

public interface Callable<V> {
    V call() throws Exception;
}
```

可以得出：

1）Callable 接口下的方法是 call()，Runnable 接口的方法是 run()。

2）Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。

3）call() 方法可以抛出异常，run()方法不可以的。

4）运行 Callable 任务可以拿到一个 Future 对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待

计算的完成，并检索计算的结果。通过 Future 对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。
但是，但是，凡事都有但是嘛...
单独使用 Callable，无法在新线程中(new Thread(Runnable r))使用，Thread 类只支持 Runnable。不过 Callable 可以使用 ExecutorService（又抛出一个概念，这个概念将在下面的线程池中说明）。
上面又提到了 Future，看一下 Future 接口：

```
public interface Future<V> {

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

Future 定义了5个方法：
1）boolean cancel(boolean mayInterruptIfRunning)：试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel() 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。此方法返回后，对 isDone() 的后续调用将始终返回 true。如果此方法返回 true，则对 isCancelled() 的后续调用将始终返回 true。
2）boolean isCancelled()：如果在任务正常完成前将其取消，则返回 true。
3）boolean isDone()：如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。(TODO 如果任务都返回true 那么该方法的意义是什么?)
4）V get()throws InterruptedException,ExecutionException：如有必要，等待计算完成，然后获取其结果 注意:该方法的返回值就是call()的返回结果(如果有的话)。
5）V get(long timeout,TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException：如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。
看来 Future 接口也不能用在线程中，那怎么用，谁实现了 Future 接口呢？答：FutureTask。

```
public class FutureTask<V> implements RunnableFuture<V> {
    ...
}

public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

FutureTask 实现了 Runnable 和 Future，所以兼顾两者优点，既可以在 Thread 中使用，又可以在 ExecutorService 中使用。

看完上面啰哩啰嗦的介绍后，下面我们具体看一下具体使用的示例：

```
Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return "个人博客：sunfusheng.com";
    }
```

```
};

FutureTask<String> task = new FutureTask<String>(callable);

Thread t = new Thread(task);
t.start(); // 启动线程
task.cancel(true); // 取消线程
```

使用 FutureTask 的好处是 FutureTask 是为了弥补 Thread 的不足而设计的，它可以让程序员准确地知道线程什么时候执行完成并获得到线程执行完成后返回的结果。FutureTask 是一种可以取消的异步的计算任务，它的计算是通过 Callable 实现的，它等价于可以携带结果的 Runnable，并且有三个状态：等待、运行和完成。完成包括所有计算以任意的方式结束，包括正常结束、取消和异常。

查看具体操作代码请参考 《个人学习项目DroidStudy》，感谢您的关注。

FutureTask 类是 Future 的一个实现， Future 可实现 Runnable，所以可通过 Executor 来执行。例如，可用下列内容替换上面带有 submit 的构造：

```
FutureTask<String> future =
  new FutureTask<String>(new Callable<String>() {
    public String call() {
      return searcher.search(target);
  }});
executor.execute(future);
```

线程的执行:

在线程中执行的

● 多线程

多线程的概念很好理解就是多条线程同时存在，但要用好多线程确不容易，涉及到多线程间通信，多线程共用一个资源等诸多问题。

使用多线程的优缺点：

优点：

1）适当的提高程序的执行效率（多个线程同时执行）。

2）适当的提高了资源利用率（CPU、内存等）。

缺点：

1）占用一定的内存空间。

2）线程越多CPU的调度开销越大。

3）程序的复杂度会上升。

对于多线程的示例代码感兴趣的可以自己写Demo啦，去运行体会，下面我主要列出一些多线程的技术点。

synchronized

同步块大家都比较熟悉，通过 synchronized 关键字来实现；所有加上 synchronized 的方法和块语句，在多线程访问的时候，同一时刻只能有一个线程能够访问。

wait()、notify()、notifyAll()

这三个方法是 java.lang.Object 的 final native 方法，任何继承 java.lang.Object 的类都有这三个方法。它们是Java语言提供的实现线程间阻塞和控制进程内调度的底层机制，平时我们会很少用到的。

wait()：

导致线程进入等待状态，直到它被其他线程通过notify()或者notifyAll唤醒，该方法只能在同步方法中调用。

notify()：

随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态，该方法只能在同步方法或同步块内部调用。

notifyAll()：

解除所有那些在该对象上调用wait方法的线程的阻塞状态，同样该方法只能在同步方法或同步块内部调用。

调用这三个方法中任意一个，当前线程必须是锁的持有者，如果不是会抛出一个 IllegalMonitorStateException 异常。

wait() 与 Thread.sleep(long time) 的区别

sleep()：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），该线程不丢失任何监视器的所属权，sleep() 是 Thread 类专属的静态方法，针对一个特定的线程。

wait() 方法使实体所处线程暂停执行，从而使对象进入等待状态，直到被 notify() 方法通知或者 wait() 的等待的时间到。sleep() 方法使持有的线程暂停运行，从而使线程进入休眠状态，直到用 interrupt 方法来打断他的休眠或者 sleep 的休眠的时间到。

wait() 方法进入等待状态时会释放同步锁，而 sleep() 方法不会释放同步锁。所以，当一个线程无限 sleep 时又没有任何人去 interrupt 它的时候，程序就产生大麻烦了，notify() 是用来通知线程，但在 notify() 之前线程是需要获得 lock 的。另个意思就是必须写在 synchronized(lockobj) {...} 之中。wait() 也是这个样子，一个线程需要释放某个 lock，也是在其获得 lock 情况下才能够释放，所以 wait() 也需要放在 synchronized(lockobj) {...} 之中。

volatile 关键字

volatile 是一个特殊的修饰符，只有成员变量才能使用它。在Java并发程序缺少同步类的情况下，多线程对成员变量的操作对其它线程是透明的。volatile 变量可以保证下一个读取操作会在前一个写操作之后发生。线程都会直接从内存中读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

ThreadLocal 变量

ThreadLocal 是Java里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

join() 方法

join() 方法定义在 Thread 类中，所以调用者必须是一个线程，join() 方法主要是让调用该方法的 Thread 完成 run() 方法里面的东西后，再执行 join() 方法后面的代码，看下下面的"意思"代码：

```
Thread t1 = new Thread(计数线程一);
Thread t2 = new Thread(计数线程二);
t1.start();
t1.join(); // 等待计数线程一执行完成，再执行计数线程二
t2.start();
```

启动 t1 后，调用了 join() 方法，直到 t1 的计数任务结束，才轮到 t2 启动，然后 t2 才开始计数任务，两个线程是按着严格的顺序来执行的。如果 t2 的执行需要依赖于 t1 中的完整数据的时候，这种方法就可以很好的确保两个线程的同步性。

Thread.yield() 方法

Thread.sleep(long time)：线程暂时终止执行（睡眠）一定的时间。

Thread.yield()：线程放弃运行，将CPU的控制权让出。

这两个方法都会将当前运行线程的CPU控制权让出来，但 sleep() 方法在指定的睡眠时间内一定不会再得到运行机会，直到它的睡眠时间完成；而 yield() 方法让出控制权后，还有可能马上被系统的调度机制选中来运行，比如，执行 yield()方法的线程优先级高于其他的线程，那么这个线程即使执行了 yield() 方法也可能不能起到让出CPU控制权的效果，因为它让出控制权后，进入排队队列，调度机制将从等待运行的线程队列中选出一个等级最高的线程来运行，那么它又（很可能）被选中来运行。

扩展

线程调度策略

(1) 抢占式调度策略

Java运行时系统的线程调度算法是抢占式的。Java运行时系统支持一种简单的固定优先级的调度算法。如果一个优先级比其他任何处于可运行状态的线程都高的线程进入就绪状态，那么运行时系统就会选择该线程运行。新的优先级较高的线程抢占了其他线程。但是Java运行时系统并不抢占同优先级的线程。换句话说，Java运行时系统不是分时的。然而，基于Java Thread类的实现系统可能是支持分时的，因此编写代码时不要依赖分时。当系统中的处于就绪状态的线程都具有相同优先级时，线程调度程序采用一种简单的、非抢占式的轮转的调度顺序。

(2) 时间片轮转调度策略

有些系统的线程调度采用时间片轮转调度策略。这种调度策略是从所有处于就绪状态的线程中选择优先级最高的线程分配一定的CPU时间运行。该时间过后再选择其他线程运行。只有当线程运行结束、放弃(yield)CPU或由于某种原因进入阻塞状态，低优先级的线程才有机会执行。如果有两个优先级相同的线程都在等待CPU，则调度程序以轮转的方式选择运行的线程。

以上把这些技术点总结出来，后面我会把多线程这块的示例代码更新到 《个人学习项目DroidStudy》，感谢您的关注。

● 线程池

创建多个线程不光麻烦而且相对影响系统性能，接下来让我看看使用线程池来操作多线程。我把自己的 《个人学习项目DroidStudy》 中线程池转成一个 gif 效果图，大家可以实际把玩下去感受线程池的原理，有兴趣的同学也可以先 star 下，我会在接下来的几个月把这个学习的Demo工程完善好。

线程池.gif

线程池的优点

1）避免线程的创建和销毁带来的性能开销。

2）避免大量的线程间因互相抢占系统资源导致的阻塞现象。

3）能够对线程进行简单的管理并提供定时执行、间隔执行等功能。

再撸一撸概念

Java里面线程池的顶级接口是 Executor，不过真正的线程池接口是 ExecutorService， ExecutorService 的默认实现是 ThreadPoolExecutor；普通类 Executors 里面调用的就是 ThreadPoolExecutor。

照例看一下各个接口的源码：

```
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
```

```
    boolean isShutdown();
    boolean isTerminated();

    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);
    ...
}

public class Executors {
    public static ExecutorService newCachedThreadPool() {
            return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,
TimeUnit.SECONDS,
                              new SynchronousQueue<Runnable>());
    }
    ...
}
```

下面我创建的一个线程池：

```
ExecutorService pool = Executors.newCachedThreadPool();
```

Executors 提供四种线程池：

- 1）newCachedThreadPool 是一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute() 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。注意，可以使用 ThreadPoolExecutor 构造方法创建具有类似属性但细节不同（例如超时参数）的线程池。
- 2）newSingleThreadExecutor 创建是一个单线程池，也就是该线程池只有一个线程在工作，所有的任务是串行执行的，如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- 3）newFixedThreadPool 创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小，线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
- 4）newScheduledThreadPool 创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

通过 ThreadPoolExecutor 的构造函数，撸一撸线程池相关参数的概念：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        threadFactory, defaultHandler);
}
```

- 1）corePoolSize：线程池的核心线程数，一般情况下不管有没有任务都会一直在线程池中一直存活，只有在 ThreadPoolExecutor 中的方法 allowCoreThreadTimeOut(boolean value) 设置为 true 时，闲置的核心线程会存在超时机制，如果在指定时间没有新任务来时，核心线程也会被终止，而这个时间间隔由第3个属性 keepAliveTime 指定。
- 2）maximumPoolSize：线程池所能容纳的最大线程数，当活动的线程数达到这个值后，后续的新任务将会被阻塞。

- 3）keepAliveTime：控制线程闲置时的超时时长，超过则终止该线程。一般情况下用于非核心线程，只有在 ThreadPoolExecutor 中的方法 allowCoreThreadTimeOut(boolean value) 设置为 true时，也作用于核心线程。
- 4）unit：用于指定 keepAliveTime 参数的时间单位，TimeUnit 是个 enum 枚举类型，常用的有：TimeUnit.HOURS(小时)、TimeUnit.MINUTES(分钟)、TimeUnit.SECONDS(秒) 和 TimeUnit.MILLISECONDS(毫秒)等。
- 5）workQueue：线程池的任务队列，通过线程池的 execute(Runnable command) 方法会将任务 Runnable 存储在队列中。
- 6）threadFactory：线程工厂，它是一个接口，用来为线程池创建新线程的。

线程池的关闭

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown() 和 shutdownNow()。

shutdown()：不会立即的终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务。

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务。

面试题

1）什么是 Executor 框架？

Executor框架在Java 5中被引入，Executor 框架是一个根据一组执行策略调用、调度、执行和控制的异步任务的框架。无限制的创建线程会引起应用程序内存溢出，所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executor 框架可以非常方便的创建一个线程池。

2）Executors 类是什么？

Executors为Executor、ExecutorService、ScheduledExecutorService、ThreadFactory 和 Callable 类提供了一些工具方法。Executors 可以用于方便的创建线程池。

查看具体操作代码请参考 《个人学习项目DroidStudy》，感谢您的关注