线程池的优点

1. 重用线程池中的线程，避免因为线程的创建和销毁所带来的性能开销
2. 能有效控制线程池的最大并发数，避免大量的线程之间因互相抢占系统资源而导致的阻塞现象
3. 能够对线程进行简单的管理，并提供定时执行以及指定间隔循环执行等功能。

ThreadPoolExecutor

Android中的线程池概念来源于Java中的Executor，Executor是一个接口，真正的线程池实现为ThreadPoolExecutor。ThreadPoolExecutor提供了一系列参数来配置线程池。

构造方法

ThreadPoolExecutor的构造方法提供了一系列参数来配置线程池，下面介绍ThreadPoolExecutor的构造方法中各个参数的含义，这些参数将会直接影响到线程池的功能特性，下面是ThreadPoolExecutor的一个常用构造方法。

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

参数	说明
corePool	线程池的核心线程数，默认情况下，核心线程会在线程中一直存活，即使处于闲置状态。如果将ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true，那么闲置的核心线程在等待新任务到来时会有超时

Size	策略，这个时间有keepAliveTime指定。当等待时间超过keepAliveTime，核心线程就会终止。
maximumPoolSize	线程池所能容纳的最大线程数，当活动线程数达到这个数值后，后续新任务将会被阻塞。
keepAliveTime	非核心线程闲置时的超时时长，超过这个时长，非核心线程就会被回收。当ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true时，keepAliveTime同样会作用于核心线程。
unit	用于指定keepAliveTime参数的时间单位，这是一个枚举，常用的有TimeUnit.MILLISECONDS(毫秒)、TimeUnit.SECONDS(秒)以及TimeUnit.MINUTES(分钟)等。
workQueue	线程池中的任务队列，通过线程池的execute方法提交的Runnable对象会存储在这个参数中。
threadFactory	线程工厂，为线程池提供创建新线程的功能。ThreadFactory是一个接口，它只有一个方法：Thread newThread(Runnable r)。

RejectedExecutionHandler	不常用。当线程池无法执行新任务时，可能是由于任务队列已满或者是无法成功执行任务，这个时候ThreadPoolExecutor会调用handler的rejectedExecution方法来通知调用者，默认情况下，rejectExecution方法会直接抛出RejectedExecutionException异常。ThreadPoolExecutor为RejectExecutionHandler提供了几个可选值：CallerRunsPolicy、AbortPolicy、DiscardPolicy和DiscardOldestPolicy，其中AbortPolicy是默认值

执行任务规则

1. 如果线程池中的线程数量未达到核心线程的数量，那么会直接启动一个核心线程来执行任务
2. 如果线程池中的线程数量已经达到或者超过核心线程的数量，那么任务会被插入到任务队列中排队等待执行
3. 如果在步骤2中无法将任务插入到任务队列中，这往往是由于任务队列已满，这个时候如果线程数量未达到线程池规定的最大值，那么就会立刻启动一个非核心线程来执行任务
4. 如果步骤3中线程数量已经达到线程池规定的最大值，那么就拒绝执行此任务，ThreadPoolExecutor会调用RejectedExecutionHandler的rejectedExecution方法来通知调用者。

AsyncTask中的线程池配置

```
private static final int CPU_COUNT =

Runtime.getRuntime().availableProcessors();
private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
private static final int KEEP_ALIVE = 1;

private static final ThreadFactory sThreadFactory = new
ThreadFactory() {
    private final AtomicInteger mCount = new AtomicInteger(1);

    public Thread newThread(Runnable r) {
        return new Thread(r, "AsyncTask #" +
mCount.getAndIncrement());
    }
};

private static final BlockingQueue<Runnable> sPoolWorkQueue = new
LinkedBlockingQueue<Runnable>(128);

public static final Executor THREAD_POOL_EXECUTOR = new
ThreadPoolExecutor(CORE_POOL_SIZE, MAXMIMUM_POOL_SIZE, KEEP_ALIVE,
TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
```

可以看出，AsyncTask对THREAD_POOL_EXECUTOR线程池进行了如下配置：

- 核心线程数等于CPU核心数 + 1
- 线程池的最大线程数为CPU核心数 * 2 + 1
- 核心线程无超时机制，非核心线程在闲置时的超时时间为1秒
- 任务队列的容量为128

线程池的分类

Android中最常见的四类具有不同功能特性的线程池，它们都直接或间接地通过配置ThreadPoolExecutor来实现自己的功能特性，这四类线程池分别是FixedThreadPool、CachedThreadPool、ScheduleThreadPool以及SingleThreadExecutor。

FixedThreadPool

通过Executors的newFixedThreadPool方法创建。是一种线程数量固定的线程池，当线程处于空闲状态时，它们并不会被回收，除非线程池被关闭了。当所有的线程都处于活动状态时，新任务都会处于等待状态，直到有线程空闲出来。由于FixedThreadPool只有核心线程并且核心线程不会被回收，这意味着它能够更加快速地响应外界的请求。newFixedThreadPool方法的实现如下，可以发现FixedThreadPool中只有核心线程并且没有超时机制，另外任务队列也是没有大小限制的。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L,
TimeUnit.MILLISECONDS, new LinkedBLockingQueue<Runnable>());
}
```

CachedThreadPool

通过Executors的newCachedThreadPool方法来创建。它是一种线程数量不定的线程池，它只有非核心线程，并且其最大线程数为Integer.MAX_VALUE。由于Integer.MAX_VALUE是一个很大的数，实际上就相当于最大线程数可以任务大。当线程池中的线程都处于活动状态时，线程池会创建新的线程来处理任务，否则就会利用空闲的线程来处理新任务。线程池中的空闲线程都有超时机制，这个超时时长为60秒，超过60秒闲置线程就会被回收。和FixedThreadPool不同，CachedThreadPool的任务队列其实相当于一个空集合，这将导致任何任务都会立即被执行，因为在这种场景下SynchronousQueue是无法插入任务的。SynchronousQueue是一个非常特殊的队列，在很多情况下可以把它简单理解为一个无法存储元素的队列，由于实际中较少使用，这里不深入分析。从CachedThreadPool的特性来看，这类线程池比较适合大量的耗时较少的任务。当整个线程池都处于闲置状态时，线程池中的线程都会超时被被停止，这个时候CachedThreadPool之中实际上是没有任何线程的，它几乎不占用任何系统资源。newCachedThreadPool方法的实现如下：

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

ScheduledThreadPool

通过Executors的newScheduledThreadPool方法来创建。它的核心线程数量是固定的，而非核心线程数是没有限制的，并且当非核心线程闲置时会被立即回收。ScheduledThreadPool这类线程池主要用于执行定时任务和具有固定周期的重复任务，newScheduledThreadPool方法的实现如下：

```
public static ScheduledExecutorService newScheduledThreadPool(int
corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new
DelayedWorkQueue());
}
```

SingleThreadExecutor

通过Executors的newSingleThreadExecutor方法来创建。这类线程池内部只有一个核心线程，它确保所有的任务都在同一个线程中按顺序执行。SingleThreadExecutor的意义在于统一所有的外界任务到一个线程中，这使得在这些任务之间不需要处理线程同步的问题。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService(new
ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue<Runnable>()));
}
```

典型用法

```
private void runThreadPool() {
        Runnable command = new Runnable() {
            @Override
            public void run() {
                SystemClock.sleep(2000);
            }
        };

        ExecutorService fixedThreadPool =
Executors.newFixedThreadPool(4);
        fixedThreadPool.execute(command);

        ExecutorService cachedThreadPool =
Executors.newCachedThreadPool();
        cachedThreadPool.execute(command);

        ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(4);
        // 2000ms后执行command
        scheduledThreadPool.schedule(command, 2000,
TimeUnit.MILLISECONDS);
        // 延迟10ms后，每隔1000ms执行一次command
        scheduledThreadPool.scheduleAtFixedRate(command, 10, 1000,
TimeUnit.MILLISECONDS);

        ExecutorService singleThreadExecutor =
Executors.newSingleThreadExecutor();
        singleThreadExecutor.execute(command);
    }
```