线程

概念

说到线程池，不得不说一下线程。无论是 `java` 还是 `Android` ，线程都是一个非常重要的概念，它是所有基础操作的载体，无论是更新 `UI` ，或是请求网络等耗时操作，都需要在线程中完成。众所周知，`Android` 中将线程分为 **主线程** 和 **工作线程**。那主线程和工作线程有什么区别的，其实本质上没太大区别，主线程因为是要跟用户直接打交道，实时交互性强，不能有其他的耗时操作阻塞其正常流程，不然出现丢帧卡顿的现象，因此 `Android` 是禁止在主线程中进行耗时操作的。

Thread

单纯的线程操作很简单，只需要：

```
    new Thread(new Runnable() {
        @Override
        public void run() {

        }
    }).start();
```

就可以将耗时操作放在子线程中执行。

Handler

如果涉及到与主线程数据的交互，比如需要在子线程中更新 UI 的话，最常见的就是通过 `Handler` 的方式来通知主线程更新 UI。

```
    static class MyHandler extends Handler {
        WeakReference<MainActivity> mActivity;

        MyHandler(MainActivity activity) {
            mActivity = new WeakReference<>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            MainActivity theActivity = mActivity.get();
            if (theActivity == null || theActivity.isFinishing()) {
                return;
            }
            switch (msg.what) {
                case :
                break;
                default:
                    break;
            }
        }
    }
```

考虑到 **内部类对外部类持有引用，可能引发内存泄漏问题**，所以采用静态内部类+持有外部类的弱引用方式来解决此问题。像类似这种常写的代码，通过创建 AS 模板，在下次需要使用的时候直接使用，方便快捷。

AsyncTask

作为轻量级别的异步任务类，内部封装了线程池、线程和 `Handler` ,主要的流程：

1. 耗时操作之前准备 (Main Thread)
2. 处理耗时操作 & 向主线程发送更新进度的 `message`(Work Thread)
3. 获取进度的回调并处理 (Work Thread)
4. 耗时操作结束的处理 (Main Thread)
5. (如果调用cancel),则要处理取消后的相应操作 (Main Thread)

作为抽象类的 `AsyncTask`，需要定义相应的泛型：

```
// Params 参数类型
// Progress 更新进度类型
// Result 执行最终结果类型
public abstract class AsyncTask<Params, Progress, Result>
```

主要涉及到的四个核心方法：

1. **onPreExecute():** 在主线程处理一些准备工作。
2. **doInBackground(Params...params):** 在子线程中处理异步耗时任务，可以通过 `publishProgress` 方法来更新任务的进度。
3. **onProgressUpdate(Progress...values):** 在主线程中执行，当后台任务进度改变触发回调。
4. **onPostExecute(Result result):** 在主线程中，异步任务结束触发回调，其中 result 就是后台任务的返回值。

案例

咱们快速实现一个简单小例子，主要功能：输入一串字符数组，统计所以字符的长度。为了模拟耗时操作，在 `doInBackground` 方法里让 `Thread` 睡一会。

```
    static class CalculateSizeTask extends AsyncTask<String, Float, Long> {

        WeakReference<AsyncTaskActivity> mActivity;

        CalculateSizeTask(AsyncTaskActivity activity) {
            mActivity = new WeakReference<>(activity);
        }

        @Override
        protected Long doInBackground(String... urls) {
            AsyncTaskActivity theActivity = mActivity.get();
            if (theActivity == null || theActivity.isFinishing()) {
                return 0L;
            }
            theActivity.curThreadName = Thread.currentThread().getName();
            int length = urls.length;
            long totalSize = 0;
            for (int i = 0; i < length; i++) {
                publishProgress(i * 1.0f / length);
                try {
                    Thread.sleep(300);
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                totalSize += urls[i].length();
            }
            return totalSize;
        }

        @Override
        protected void onProgressUpdate(Float... values) {
            Log.i(TAG, "onProgressUpdate: " + values[0]);
            updateContent(String.format("%s%%", values[0]));
        }

        @Override
        protected void onPostExecute(Long aLong) {
            updateContent("Complete! totalSize:\t" + aLong);
        }

        @Override
        protected void onPreExecute() {
            updateContent("loading....");
        }

        @Override
        protected void onCancelled(Long aLong) {
            super.onCancelled(aLong);
            updateContent("已取消 " + aLong);
        }

        void updateContent(String content) {
            AsyncTaskActivity theActivity = mActivity.get();
            if (theActivity == null || theActivity.isFinishing()) {
                return;
            }
            theActivity.mTvContent.setText(theActivity.curThreadName + "\n" +
content);
        }
    }
```

与之前 `Handler` 创建方式一样，采用 **静态内部类+外部类的弱引用** 方式避免内存泄漏。最终调用就简单多了。

```
    calculateSizeTask = new CalculateSizeTask(this);
    calculateSizeTask.execute(DATA);
```

看一下效果：

效果很明显不做过多解释，不过有几点需要注意一下：

1. 创建 `AsyncTask` 对象过程 & `execute` 执行过程必须在主线程中完成。
2. 不要调用 `AsyncTask` 内部的回调方法(`doInBackground` ...)。
3. 不能多次执行同一个 `AsyncTask` 对象的 `execute` 方法，否则会直接抛出异常。
4. 因为 `AsyncTask` 内部默认是串行执行任务，因此前一个任务没有执行完，新的任务处于等待过程。

IntentService

`Service` 用于处理长期需要在后台执行的任务，但是不代表着可以执行耗时任务，内部的操作其实都是在主线程中完成的。因此 `IntentService` 诞生，既可以存在于后台，提高任务的存活率，又可以执行耗时操作。相比较于前者 `AsyncTaks` 它毕竟需要依附于 `Activity`,因此生命周期相对较短，而 `IntentService` 内部封装了 `HanderThread` 和 `Handler`。

`HandlerThread` 是 `Thread` 的子类，具体的逻辑都在 `run()` 方法中:

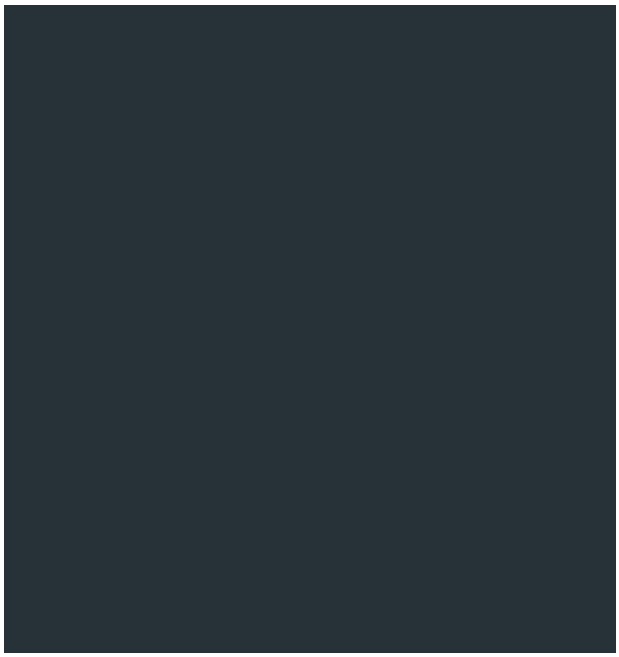
```
    @Override
    public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        synchronized (this) {
            mLooper = Looper.myLooper();
            notifyAll();
        }
        Process.setThreadPriority(mPriority);
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
    }
```

其实就是内部维护了一个无线循环的消息队列，这样就可以在 `HandlerThread` 中创建 `Handler`，外部则通过消息的方式来通知 `HandlerThread` 执行一个具体的任务。

再回到 `IntentService`, 在 `onCreate()` 方法中创建 `HandlerThread` 和 `Handler`。

```
    @Override
    public void onCreate() {
        super.onCreate();
        HandlerThread thread = new HandlerThread("IntentService[" + mName +
"]");
        thread.start();
        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }
```

随后在 `onStart()` 方法中创建 `Message` 交给 `Handler` 处理，`Handler` 收到事件通知后调用需要子类覆盖的 `onHandleIntent()` 方法，具体的耗时任务就在该方法中执行，执行结束调用 `stopSelf()` 关闭自己。这套流程行云流水，丝毫不拖泥带水，默默的执行，默默的关闭，都不用关心 `Service` 的生命周期。



当需要同时执行多个任务时，`IntentService` 会串行执行，直至任务完成才会关闭服务。

线程池

当需要执行的任务增多时，单个线程是满足不了需求的，此时就需要创建多个线程来完成需要。多线程的最大好处就在于 **提高 CPU 的利用率** & **提高执行效率**，同时也存在着一些弊端：频繁的创建和销毁线程会产生很多的性能开销。为了解决这个问题，线程池孕育而生。

1. 复用线程池中的线程，减少创建&销毁线程的性能开销。
2. 控制线程的并发数，避免对资源竞争而导致阻塞现象。
3. 对线程有很好的管理并开启新的姿势。

ThreadPoolExecutor

翻译过来就是 线程池执行器，它是线程池的真正实现，构造方法提供了一些列参数来配置线程池，掌握了这些参数的配置，可以加大对线程池的了解。

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

以这个最常见的构造函数进行分析。

1. **corePoolSize:** 线程池中核心的线程数。默认情况核心线程是一致存活，及时没有任何操作。但是如果执行了 `allowCoreThreaTimeOut()` 方法，核心线程也是有可能被回收的，这取决于是否处于闲置状态 & `keepAliveTime` 。
2. **maximumPollSize:** 线程池所能容纳的最大线程数。超过限制，新线程会被阻塞。
3. **keepAliveTime:** 线程闲置超时等待时间。超过此值会被回收。
4. **unit:** 超时等待时间单位。

5. **workQueue:** 线程池中的任务队列。每次执行 `execute()` 会把 `runnable` 对象存储在这个队列中。
6. **threadFactory:** 线程工厂，为线程池提供创建新线程的功能。

关键点在于 核心线程数、最大线程数和任务队列数，执行流程如下，记住一点，优先级：核心线程数 > 任务队列数 > 最大线程数。

1. 线程池中线程数小于核心线程数，直接创建新的核心线程。
2. 如果线程数超过了超过核心线程数，任务会被插入到任务队列中等待执行。
3. 如果任务队列也满了，这时会进入到最大线程(步骤4)的判断逻辑值。
4. 如果线程池中线程数小于最大线程数，则启动一个非核心线程来执行。
5. 如果果线程池中线程数超过了最大线程数，那没办法了，任务会被拒绝，`ThreadPoolExecutor` 会调用 RejectedExecutionHander 的 rejectedExecution 方法通知调用者。

Executors

`Executors` 提供了创建常用线程池的静态方法，接下也会大概讲解一下常用的四种线程池。

1. 定长线程池 (FixedThreadPool)
2. 单线程化线程池 (SingleThreadExecutor)
3. 定时线程池 (ScheduledThreadPool)
4. 缓存线程池 (CachedThreadPool)

在创建线程池之前，我单独构造了个方法专门用来创建 `Runnnable` 对象。

```
    private Runnable getRunnable() {
        return new Runnable() {
            @Override
            public void run() {
                sb.append(Thread.currentThread().getName()).append("\n");
                printLog();
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
    }

    private void printLog(){
        mHandler.post(new Runnable() {
            @Override
            public void run() {
                mTvContent.setText(sb.toString());
            }
        });
    }
```

该 `Runnable` 对象主要用来打印线程信息，并展示在界面上。

定长线程池 (FixedThreadPool)

源码：

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

特点:

- 线程数量固定
- 只有核心线程，并且不会被回收
- 任务队列无限制，超出核心线程数的线程处于等待中
- 适用于控制线程的最大并发数

案例：

```
    private void fixedThreadPool() {
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        for (int i = 0; i < 10; i++) {
            executorService.execute(getRunnable());
        }
    }
```

效果：

这里我设置了最大并发数为2，所以只会创建两个核心线程，其余为完成的需要进行等待。

单线程化线程池 (SingleThreadExecutor)

源码：

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

特点：

- 只有一个核心线程
- 任务队列无限制

- 不需要考虑线程同步问题
- 适用于一些 因为并发而导致问题的操作

案例：

```
    private void singleThreadPool() {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
            executorService.execute(getRunnable());
        }
    }
```

效果：

简单粗暴，只有一个线程在执行。

定时线程池 (ScheduledThreadPool)

源码：

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

特点：

- 线程数量固定
- 非核心数量无限制
- 适用于定时&周期性任务

案例：

```
    private void scheduledThreadPool() {
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
//        scheduledExecutorService.schedule(getRunnable(), 1, TimeUnit.SECONDS);
//        scheduledExecutorService.scheduleAtFixedRate(getRunnable(), 1, 1,
TimeUnit.SECONDS);
        scheduledExecutorService.scheduleWithFixedDelay(getRunnable(), 1, 1,
TimeUnit.SECONDS);
    }
```

> schedule(): 执行定时任务
>
> scheduleAtFixedRate(): 执行周期任务 (从任务开始计时)

> scheduleWithFixedDelay(): 执行周期任务 (从任务结束开始计时)

效果:

可缓存线程池 (CachedThreadPool)

源码:

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

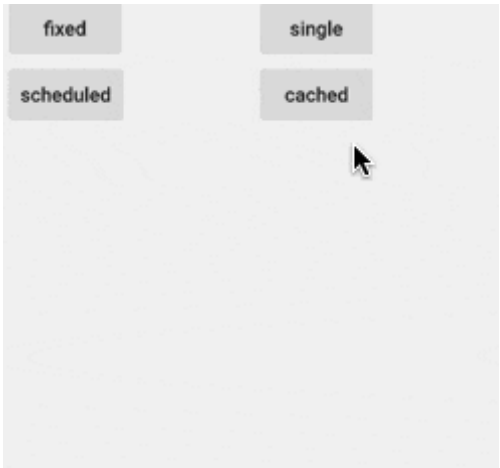
特点:

- 无核心线程
- 非核心线程数量无限制
- 对于空闲线程回收灵活
- 适用于大量&耗时少的任务

案例:

```
    private void cacheThreadPool() {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 50; i++) {
            executorService.execute(getRunnable());
        }
    }
```

效果:



可以看到很多线程是被复用了。