**一、线程和进程**

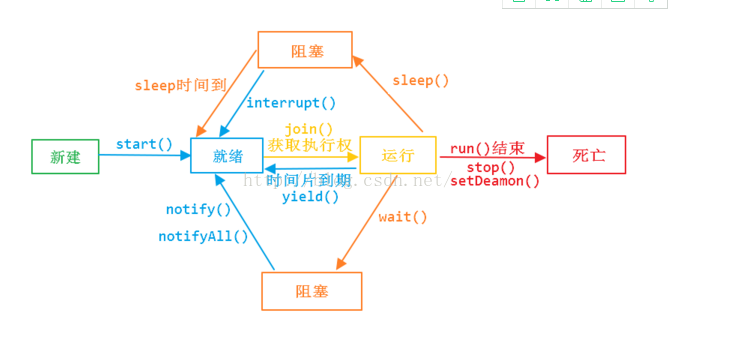
进程是一个具有独立功能的应用程序，能够申请和分配系统资源。程序本身是没有生命的，只有处理器赋予其执行权力时，程序才能够成为一个活动的实体，我们称之为进程。一个进程可以包含多个线程，线程是进程的最小调度单位。

多个进程相互独立，多个线程共享进程资源。不同进程中的线程相互不可见。

线程依赖于进程，每个线程都有进程带来的各自独立的资源。

线程之间切换比进程之间切换节约系统开销。但是多线程的切换，开销仍然很大。

**二、线程的五种状态**



三、线程池

### 为什么要用线程池

使用线程池管理线程有如下优点：

1. 降低资源消耗：通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. 提高响应速度：当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 提高线程的可管理性：线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

### 线程池介绍

#### ThreadPoolExecutor

Java 为我们提供了 ThreadPoolExecutor 来创建一个线程池，其完整构造函数如下所示：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

* int corePoolSize（核心线程数）：线程池新建线程的时候，如果当前线程总数小于corePoolSize，则新建的是核心线程，如果超过corePoolSize，则新建的是非核心线程；核心线程默认情况下会一直存活在线程池中，即使这个核心线程啥也不干(闲置状态)；如果设置了 allowCoreThreadTimeOut 为 true，那么核心线程如果不干活(闲置状态)的话，超过一定时间(时长下面参数决定)，就会被销毁掉。
* int maximumPoolSize（线程池能容纳的最大线程数量）：线程总数 = 核心线程数 + 非核心线程数。
* long keepAliveTime（非核心线程空闲存活时长）：非核心线程空闲时长超过该时长将会被回收，主要应用在缓存线程池中，当设置了 allowCoreThreadTimeOut 为 true 时，对核心线程同样起作用。
* TimeUnit unit（keepAliveTime 的单位）：它是一个枚举类型，常用的如：TimeUnit.SECONDS（秒）、TimeUnit.MILLISECONDS（毫秒）。
* BlockingQueue workQueue（任务队列）：当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果队列满了，则新建非核心线程执行任务，常用的 workQueue 类型：
  1. SynchronousQueue：这个队列接收到任务的时候，会直接提交给线程处理，而不保留它，如果所有线程都在工作怎么办？那就新建一个线程来处理这个任务！所以为了保证不出现 线程数达到了 maximumPoolSize 而不能新建线程 的错误，使用这个类型队列的时候，maximumPoolSize 一般指定成 Integer.MAX\_VALUE，即无限大。
  2. LinkedBlockingQueue：这个队列接收到任务的时候，如果当前线程数小于核心线程数，则新建线程(核心线程)处理任务；如果当前线程数等于核心线程数，则进入队列等待。由于这个队列没有最大值限制，即所有超过核心线程数的任务都将被添加到队列中，这也就导致了 maximumPoolSize 的设定失效，因为总线程数永远不会超过 corePoolSize。
  3. ArrayBlockingQueue：可以限定队列的长度，接收到任务的时候，如果没有达到 corePoolSize 的值，则新建线程(核心线程)执行任务，如果达到了，则入队等候，如果队列已满，则新建线程(非核心线程)执行任务，又如果总线程数到了 maximumPoolSize，并且队列也满了，则发生错误。
  4. DelayQueue：队列内元素必须实现 Delayed 接口，这就意味着你传进去的任务必须先实现 Delayed 接口。这个队列接收到任务时，首先先入队，只有达到了指定的延时时间，才会执行任务。
* ThreadFactory threadFactory（线程工厂）：用来创建线程池中的线程，通常用默认的即可。
* RejectedExecutionHandler handler（拒绝策略）：在线程池已经关闭的情况下和任务太多导致最大线程数和任务队列已经饱和，无法再接收新的任务，在上面两种情况下，只要满足其中一种时，在使用 execute() 来提交新的任务时将会拒绝，线程池提供了以下 4 种策略：
  1. AbortPolicy：默认策略，在拒绝任务时，会抛出RejectedExecutionException。
  2. CallerRunsPolicy：只要线程池未关闭，该策略直接在调用者线程中，运行当前的被丢弃的任务。
  3. DiscardOldestPolicy：该策略将丢弃最老的一个请求，也就是即将被执行的任务，并尝试再次提交当前任务。
  4. DiscardPolicy：该策略默默的丢弃无法处理的任务，不予任何处理。

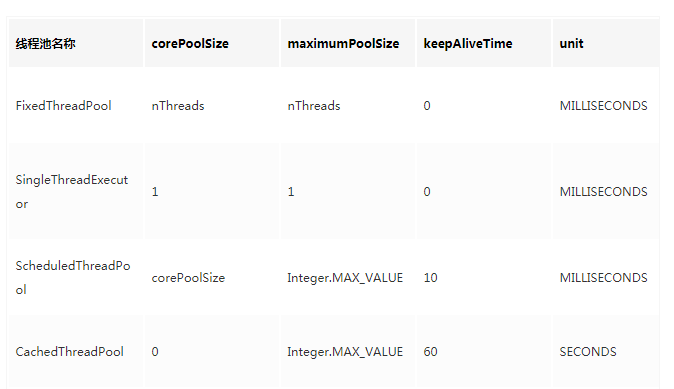
#### 线程池执行策略

当一个任务要被添加进线程池时，有以下四种执行策略：

1. 线程数量未达到 corePoolSize，则新建一个线程（核心线程）执行任务。
2. 线程数量达到了 corePoolsSize，则将任务移入队列等待。
3. 队列已满，新建非核心线程执行任务。
4. 队列已满，总线程数又达到了 maximumPoolSize，就会由 RejectedExecutionHandler 抛出异常。

#### 其流程图如下所示：常见的四类线程池

常见的四类线程池分别有 FixedThreadPool、SingleThreadExecutor、ScheduledThreadPool 和 CachedThreadPool，它们其实都是通过 ThreadPoolExecutor 创建的，其参数如下表所示：



如果你不想自己写一个线程池，那么你可以从上面看看有没有符合你要求的（一般都够用了），如果有，那么很好你直接用就行了，如果没有，那你就老老实实自己去写一个吧。

#### 合理地配置线程池

需要针对具体情况而具体处理，不同的任务类别应采用不同规模的线程池，任务类别可划分为 CPU 密集型任务、IO 密集型任务和混合型任务。

* CPU 密集型任务：线程池中线程个数应尽量少，推荐配置为 (CPU 核心数 + 1)；
* IO 密集型任务：由于 IO 操作速度远低于 CPU 速度，那么在运行这类任务时，CPU 绝大多数时间处于空闲状态，那么线程池可以配置尽量多些的线程，以提高 CPU 利用率，推荐配置为 (2 \* CPU 核心数 + 1)；
* 混合型任务：可以拆分为 CPU 密集型任务和 IO 密集型任务，当这两类任务执行时间相差无几时，通过拆分再执行的吞吐率高于串行执行的吞吐率，但若这两类任务执行时间有数据级的差距，那么没有拆分的意义。

### 线程池工具类封装及使用

为了提升开发效率及更好地使用和管理线程池，我已经为你们封装好了线程工具类----ThreadUtils，依赖 AndroidUtilCode 1.16.1 版本即可使用，其 API 如下所示：

isMainThread : 判断当前是否主线程

getFixedPool : 获取固定线程池

getSinglePool : 获取单线程池

getCachedPool : 获取缓冲线程池

getIoPool : 获取 IO 线程池

getCpuPool : 获取 CPU 线程池

executeByFixed : 在固定线程池执行任务

executeByFixedWithDelay : 在固定线程池延时执行任务

executeByFixedAtFixRate : 在固定线程池按固定频率执行任务

executeBySingle : 在单线程池执行任务

executeBySingleWithDelay: 在单线程池延时执行任务

executeBySingleAtFixRate: 在单线程池按固定频率执行任务

executeByCached : 在缓冲线程池执行任务

executeByCachedWithDelay: 在缓冲线程池延时执行任务

executeByCachedAtFixRate: 在缓冲线程池按固定频率执行任务

executeByIo : 在 IO 线程池执行任务

executeByIoWithDelay : 在 IO 线程池延时执行任务

executeByIoAtFixRate : 在 IO 线程池按固定频率执行任务

executeByCpu : 在 CPU 线程池执行任务

executeByCpuWithDelay : 在 CPU 线程池延时执行任务

executeByCpuAtFixRate : 在 CPU 线程池按固定频率执行任务

executeByCustom : 在自定义线程池执行任务

executeByCustomWithDelay: 在自定义线程池延时执行任务

executeByCustomAtFixRate: 在自定义线程池按固定频率执行任务

cancel : 取消任务的执行

如果你使用 RxJava 很 6，而且项目中已经使用了 RxJava，那么你可以继续使用 RxJava 来做线程切换的操作；如果你并不会 RxJava 或者是在开发 SDK，那么这个工具类再适合你不过了，它可以为你统一管理线程池的使用，不至于让你的项目中出现过多的线程池。

ThreadUtils 使用极为方便，看 API 即可明白相关意思，FixedPool、SinglePool、CachedPool 分别对应了上面介绍的 FixedThreadPool、SingleThreadExecutor、CachedThreadPool 这三种，IoPool 是创建 (CPU\_COUNT \* 2 + 1) 个核心线程数，CpuPool 是建立 (CPU\_COUNT + 1) 个核心线程数；而所有的 execute 都是线程池外围裹了一层 ScheduledThreadPool，这里和 RxJava 线程池的实现有所相似，可以更方便地提供延时任务和固定频率执行的任务，当然也可以更方便地取消任务的执行，下面让我们来简单地来介绍其使用，以从 assets 中拷贝 APK 到 SD 卡为例，其代码如下所示：

public static void releaseInstallApk(final OnReleasedListener listener) {

if (!FileUtils.isFileExists(Config.TEST\_APK\_PATH)) {

ThreadUtils.executeByIo(new ThreadUtils.SimpleTask<Void>() {

@Override

public Void doInBackground() throws Throwable {

ResourceUtils.copyFileFromAssets("test\_install", Config.TEST\_APK\_PATH);

return null;

}

@Override

public void onSuccess(Void result) {

if (listener != null) {

listener.onReleased();

}

}

});

} else {

if (listener != null) {

listener.onReleased();

}

LogUtils.d("test apk existed.");

}

}

看起来还不是很优雅是吧，你可以把相关的 Task 都抽出来放到合适的包下，这样每个 Task 的职责一看便知，如上例子可以改装成如下所示：

public class ReleaseInstallApkTask extends ThreadUtils.SimpleTask<Void> {

private OnReleasedListener mListener;

public ReleaseInstallApkTask(final OnReleasedListener listener) {

mListener = listener;

}

@Override

public Void doInBackground() throws Throwable {

ResourceUtils.copyFileFromAssets("test\_install", Config.TEST\_APK\_PATH);

return null;

}

@Override

public void onSuccess(Void result) {

if (mListener != null) {

mListener.onReleased();

}

}

public void execute() {

ThreadUtils.executeByIo(this);

}

}

public static void releaseInstallApk(final OnReleasedListener listener) {

if (!FileUtils.isFileExists(Config.TEST\_APK\_PATH)) {

new ReleaseInstallApkTask(listener).execute();

} else {

if (listener != null) {

listener.onReleased();

}

LogUtils.d("test apk existed.");

}

}

是不是瞬间清爽了很多，如果执行成功的回调中涉及了 View 相关的操作，那么你需要在 destroy 中取消 task 的执行哦，否则会内存泄漏哦，继续以上面的例子为例，代码如下所示：

public class XXActivity extends Activity {

···

@Override

protected void onDestroy() {

// ThreadUtils.cancel(releaseInstallApkTask);// 或者下面的取消都可以

releaseInstallApkTask.cancel();

super.onDestroy();

}

}

以上是以 SimpleTask 为例，Task 的话会多两个回调，onCancel() 和 onFail(Throwable t)，它们和 onSuccess(T result) 都是互斥的，最终回调只会走它们其中之一，**并且在 Android 端是发送到主线程中执行**，如果是 Java 端的话那就还是会在相应的线程池中执行，这点也方便了我做单元测试。

### 线程池工具类单元测试

如果遇到了异步的单测，你会发现单测很快就跑完呢，并没有等待我们线程跑完再结束，我们可以用 CountDownLatch 来等待线程的结束，或者化异步为同步的做法，这里我们使用 CountDownLatch 来实现，我进行了简单的封装，测试 Fixed 的代码如下所示：

public class ThreadUtilsTest {

@Test

public void executeByFixed() throws Exception {

asyncTest(10, new TestRunnable<String>() {

@Override

public void run(final int index, CountDownLatch latch) {

final TestTask<String> task = new TestTask<String>(latch) {

@Override

public String doInBackground() throws Throwable {

Thread.sleep(500 + index \* 10);

if (index < 4) {

return Thread.currentThread() + " :" + index;

} else if (index < 7) {

cancel();

return null;

} else {

throw new NullPointerException(String.valueOf(index));

}

}

@Override

void onTestSuccess(String result) {

System.out.println(result);

}

};

ThreadUtils.executeByFixed(3, task);

}

});

}

@Test

public void executeByFixedWithDelay() throws Exception {

asyncTest(10, new TestRunnable<String>() {

@Override

public void run(final int index, CountDownLatch latch) {

final TestTask<String> task = new TestTask<String>(latch) {

@Override

public String doInBackground() throws Throwable {

Thread.sleep(500);

if (index < 4) {

return Thread.currentThread() + " :" + index;

} else if (index < 7) {

cancel();

return null;

} else {

throw new NullPointerException(String.valueOf(index));

}

}

@Override

void onTestSuccess(String result) {

System.out.println(result);

}

};

ThreadUtils.executeByFixedWithDelay(3, task, 500 + index \* 10, TimeUnit.MILLISECONDS);

}

});

}

@Test

public void executeByFixedAtFixRate() throws Exception {

asyncTest(10, new TestRunnable<String>() {

@Override

public void run(final int index, CountDownLatch latch) {

final TestScheduledTask<String> task = new TestScheduledTask<String>(latch, 3) {

@Override

public String doInBackground() throws Throwable {

Thread.sleep(500 + index \* 10);

if (index < 4) {

return Thread.currentThread() + " :" + index;

} else if (index < 7) {

cancel();

return null;

} else {

throw new NullPointerException(String.valueOf(index));

}

}

@Override

void onTestSuccess(String result) {

System.out.println(result);

}

};

ThreadUtils.executeByFixedAtFixRate(3, task, 3000 + index \* 10, TimeUnit.MILLISECONDS);

}

});

}

abstract static class TestScheduledTask<T> extends ThreadUtils.Task<T> {

private static final AtomicInteger ATOMIC\_INTEGER = new AtomicInteger();

private int mTimes;

CountDownLatch mLatch;

TestScheduledTask(final CountDownLatch latch, final int times) {

mLatch = latch;

mTimes = times;

}

abstract void onTestSuccess(T result);

@Override

public void onSuccess(T result) {

onTestSuccess(result);

if (ATOMIC\_INTEGER.addAndGet(1) % mTimes == 0) {

mLatch.countDown();

}

}

@Override

public void onCancel() {

System.out.println(Thread.currentThread() + " onCancel: ");

mLatch.countDown();

}

@Override

public void onFail(Throwable t) {

System.out.println(Thread.currentThread() + " onFail: " + t);

mLatch.countDown();

}

}

abstract static class TestTask<T> extends ThreadUtils.Task<T> {

CountDownLatch mLatch;

TestTask(final CountDownLatch latch) {

mLatch = latch;

}

abstract void onTestSuccess(T result);

@Override

public void onSuccess(T result) {

onTestSuccess(result);

mLatch.countDown();

}

@Override

public void onCancel() {

System.out.println(Thread.currentThread() + " onCancel: ");

mLatch.countDown();

}

@Override

public void onFail(Throwable t) {

System.out.println(Thread.currentThread() + " onFail: " + t);

mLatch.countDown();

}

}

<T> void asyncTest(int threadCount, TestRunnable<T> runnable) throws Exception {

CountDownLatch latch = new CountDownLatch(threadCount);

for (int i = 0; i < threadCount; i++) {

runnable.run(i, latch);

}

latch.await();

}

interface TestRunnable<T> {

void run(final int index, CountDownLatch latch);

}

}

### 最后想说的话

感谢大家一起陪伴 AndroidUtilCode 的成长，核心工具类几乎都已囊括，也是汇集了我大量的心血，把开源做到了极致，希望大家可以用的舒心，大大提升开发效率，早日赢取白富美，走上人生巅峰。

欢迎来我的 狗窝 坐坐哈

后文再添加一个个人对 OkHttp 的线程池的使用分析，算是送上个小福利。

### OkHttp 中的线程池使用

查看 OkHttp 的源码发现，不论是同步请求还是异步请求，最终都是交给 Dispatcher 做处理，我们看下该类和线程池有关的的主要代码：

public final class Dispatcher {

// 最大请求数

private int maxRequests = 64;

// 相同 host 最大请求数

private int maxRequestsPerHost = 5;

// 请求执行线程池，懒加载

private @Nullable ExecutorService executorService;

// 就绪状态的异步请求队列

private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();

// 运行中的异步请求队列，包括还没完成的请求

private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();

public Dispatcher(ExecutorService executorService) {

this.executorService = executorService;

}

public Dispatcher() {

}

public synchronized ExecutorService executorService() {

if (executorService == null) {

// 和 CachedThreadPool 很相似

executorService = new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));

}

return executorService;

}

synchronized void enqueue(AsyncCall call) {

// 不超过最大请求数并且不超过 host 最大请求数

if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {

// 添加到运行中的异步请求队列

runningAsyncCalls.add(call);

// 添加到线程池中运行

executorService().execute(call);

} else {

// 添加到就绪的异步请求队列

readyAsyncCalls.add(call);

}

}

// 当该异步请求结束的时候，会调用此方法，用于将运行中的异步请求队列中的该请求移除并调整请求队列

// 此时就绪队列中的请求就可以进入运行中的队列

void finished(AsyncCall call) {

finished(runningAsyncCalls, call, true);

}

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {

int runningCallsCount;

Runnable idleCallback;

synchronized (this) {

if (!calls.remove(call)) throw new AssertionError("Call wasn't in-flight!");

if (promoteCalls) promoteCalls();

runningCallsCount = runningCallsCount();

idleCallback = this.idleCallback;

}

if (runningCallsCount == 0 && idleCallback != null) {

idleCallback.run();

}

}

// 根据 maxRequests 和 maxRequestsPerHost 来调整 runningAsyncCalls 和 readyAsyncCalls

// 使运行中的异步请求不超过两种最大值，并且如果队列有空闲，将就绪状态的请求归类为运行中。

private void promoteCalls() {

// 如果运行中的异步队列不小于最大请求数，直接返回

if (runningAsyncCalls.size() >= maxRequests) return;

// 如果就绪队列为空，直接返回

if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

// 遍历就绪队列并插入到运行队列

for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {

AsyncCall call = i.next();

if (runningCallsForHost(call) < maxRequestsPerHost) {

i.remove();

runningAsyncCalls.add(call);

executorService().execute(call);

}

// 运行队列中的数量到达最大请求数，直接返回

if (runningAsyncCalls.size() >= maxRequests) return;

}

}

}

可以发现 OkHttp 不是在线程池中维护线程的个数，线程是通过 Dispatcher 间接控制，线程池中的请求都是运行中的请求，这也就是说线程的重用不是线程池控制的，通过源码我们发现线程重用的地方是请求结束的地方 finished(AsyncCall call) ，而真正的控制是通过 promoteCalls 方法， 根据 maxRequests 和 maxRequestsPerHost 来调整 runningAsyncCalls 和 readyAsyncCalls，使运行中的异步请求不超过两种最大值，并且如果队列有空闲，将就绪状态的请求归类为运行中。