**[Android] 任意时刻从子线程切换到主线程的实现**

**引入**

在Android开发中常常会遇到**网络请求**，[**数据库**](http://www.2cto.com/database/)**数据准备**等一些**耗时**的操作；而这些操作是**不允许在主线程中**进行的。因为这样会堵塞主线程导致程序出现未响应情况。

所以只能另起一个子线程进行这些耗时的操作，完成后再显示到界面。众所周知，界面等控件操作只能在主线程中完成；所以不可避免的**需要从子线程切换到主线程**。

**方法**

对于这样的情况在[Android](http://www.2cto.com/kf/yidong/Android/) 中比较常见的是使用**AsynTask**类或者 Handler来进行线程切换；而其中**AsynTask**是官方封装的类，较为简单，效率也比较可以，但是并不适合所有的情况，至少我使用了一两次后就再也没有使用了。使用 Handler可以说是最万能的方式，其原理是消息循环，在主线程中建立Handler 变量时，就会启动Handler消息循环，一个个的处理消息队列中的任务。但是其也有棘手的时候；其棘手的地方就是麻烦。

每次都需要去建立一个 Handler 类，然后使用voidhandleMessage(Messagemsg) 方法把消息取出来进行界面操作，而其中还要遇到参数的传递等问题，说起来真的是挺麻烦的。

**想法**

既然有着这么多的问题，但是又有其的优势，我们何不自行封装一次呢？

**这里我梳理一下思路：**

还是使用 Handler进行线程切换在子线程中能通过简单的调用就切换到主线程进行工作在子线程切换到主线程时，子线程进入阻塞直到主线程执行完成（知道为什么有这样的需求么？）一定要保证其效率主线程的执行要有时间限制，不能执行太长时间导致主线程阻塞

我能想到的就是这些；观众老爷们咋样？可否还有需求？

**说干就干，梳理一下实现方法**

使用Handler 实现，既然这样那么主方法当然就是采用继承Handler 来实现而要简单同时又要能随时进入方法 那么对外采用静态方法是个不错的选择而要保证效率的话，那就不能让Handler 的消息队列过于太多，但是又要满足能随时调用，那么采用外部 Queue更具情况有阻塞与不阻塞子线程两种情况，那么采用两个 Queue吧，分开来好一点要保证不能长时间在主线程执行那么对于队列的执行一定要有时间限制加一个时间变量吧当然最后考虑了一下，既然要简单那么传入参数采用Runnable 是很爽的

万事俱备，只欠东风了；好了进入下一环节。

**CodeTime**

**首先我们建立一个ToolKit类：**

public class ToolKit {

/\*\*

\* Asynchronously

\*

\* @param runnable Runnable Interface

\*/

public static void runOnMainThreadAsync(Runnable runnable) {

}

/\*\*

\* Synchronously

\*

\* @param runnable Runnable Interface

\*/

public static void runOnMainThreadSync(Runnable runnable) {

}

}

两个对外的方法简单来说就是这样了；但是其功能实现就需要使用继承Handler了。

建立类HandlerPoster，继承自Handler：

final class HandlerPoster extends Handler {

private final int ASYNC = 0x1;

private final int SYNC = 0x2;

private final Queue asyncPool;

private final Queue syncPool;

private final int maxMillisInsideHandleMessage;

private boolean asyncActive;

private boolean syncActive;

HandlerPoster(Looper looper, int maxMillisInsideHandleMessage) {

super(looper);

this.maxMillisInsideHandleMessage = maxMillisInsideHandleMessage;

asyncPool = new LinkedList<>();

syncPool = new LinkedList<>();

}

void dispose() {

this.removeCallbacksAndMessages(null);

this.asyncPool.clear();

this.syncPool.clear();

}

void async(Runnable runnable) {

synchronized (asyncPool) {

asyncPool.offer(runnable);

if (!asyncActive) {

asyncActive = true;

if (!sendMessage(obtainMessage(ASYNC))) {

throw new GeniusException(Could not send handler message);

}

}

}

}

void sync(SyncPost post) {

synchronized (syncPool) {

syncPool.offer(post);

if (!syncActive) {

syncActive = true;

if (!sendMessage(obtainMessage(SYNC))) {

throw new GeniusException(Could not send handler message);

}

}

}

}

@Override

public void handleMessage(Message msg) {

if (msg.what == ASYNC) {

boolean rescheduled = false;

try {

long started = SystemClock.uptimeMillis();

while (true) {

Runnable runnable = asyncPool.poll();

if (runnable == null) {

synchronized (asyncPool) {

// Check again, this time in synchronized

runnable = asyncPool.poll();

if (runnable == null) {

asyncActive = false;

return;

}

}

}

runnable.run();

long timeInMethod = SystemClock.uptimeMillis() - started;

if (timeInMethod >= maxMillisInsideHandleMessage) {

if (!sendMessage(obtainMessage(ASYNC))) {

throw new GeniusException(Could not send handler message);

}

rescheduled = true;

return;

}

}

} finally {

asyncActive = rescheduled;

}

} else if (msg.what == SYNC) {

boolean rescheduled = false;

try {

long started = SystemClock.uptimeMillis();

while (true) {

SyncPost post = syncPool.poll();

if (post == null) {

synchronized (syncPool) {

// Check again, this time in synchronized

post = syncPool.poll();

if (post == null) {

syncActive = false;

return;

}

}

}

post.run();

long timeInMethod = SystemClock.uptimeMillis() - started;

if (timeInMethod >= maxMillisInsideHandleMessage) {

if (!sendMessage(obtainMessage(SYNC))) {

throw new GeniusException(Could not send handler message);

}

rescheduled = true;

return;

}

}

} finally {

syncActive = rescheduled;

}

} else super.handleMessage(msg);

}

}

下面来说说这个我花了很大时间弄出来的类。

**类的变量部分：**

两个标识，两个队列，两个执行状态，一个时间限制；很好理解吧？标识为了区别分别是处理那个队列使用；队列当然是装着任务了；执行状态是为了避免重复发送消息导致消息队列过多；时间限制这个最好理解了。

**下面来说说方法部分：**

**构造函数HandlerPoster(Looper\_looper,int\_maxMillisInsideHandleMessage)：**

传入两个参数，分别是 Looper，用于初始化到主线程，后面的是时间限制；然后初始化了两个队列。

**销毁函数void\_dispose()：**首先去除掉没有处理的消息，然后清空队列。

**添加异步执行方法void\_async(Runnable\_runnable)：**

void async(Runnable runnable) {

synchronized (asyncPool) {

asyncPool.offer(runnable);

if (!asyncActive) {

asyncActive = true;

if (!sendMessage(obtainMessage(ASYNC))) {

throw new GeniusException(Could not send handler message);

}

}

}

}

可以看见进入方法后第一件事儿就是进入同步状态，然后调用asyncPool.offer(runnable);把任务写入到队列。

之后判断当前是否处于异步任务执行中，如果不是：立刻改变状态，然后发送一个消息给当前Handler，当然不要忘记了传入标识。

当然为了效率其消息的构造也是通过obtainMessage(ASYNC)方法来完成，为的就是不过多建立新的Message,尽量使用当前队列中空闲的消息。

**添加同步执行方法void\_sync(SyncPost\_post)：**

void sync(SyncPost post) {

synchronized (syncPool) {

syncPool.offer(post);

if (!syncActive) {

syncActive = true;

if (!sendMessage(obtainMessage(SYNC))) {

throw new GeniusException(Could not send handler message);

}

}

}

}

可以看到，这里传入的并不是Runnable 而是SyncPost这是为了同步而对Runnable进行了一次封装后的类；后面介绍。

同样是进入同步，添加，判断，发送消息。

**任务执行者@Override\_void\_handleMessage(Message\_msg)：**

这里是复写的Handler的消息处理方法，当当前Handler消息队列中有消息的时候将会按照顺序一个个的调用该方法。

**分段来看：**

if (msg.what == ASYNC) {

boolean rescheduled = false;

try {

long started = SystemClock.uptimeMillis();

while (true) {

Runnable runnable = asyncPool.poll();

if (runnable == null) {

synchronized (asyncPool) {

// Check again, this time in synchronized

runnable = asyncPool.poll();

if (runnable == null) {

asyncActive = false;

return;

}

}

}

runnable.run();

long timeInMethod = SystemClock.uptimeMillis() - started;

if (timeInMethod >= maxMillisInsideHandleMessage) {

if (!sendMessage(obtainMessage(ASYNC))) {

throw new GeniusException(Could not send handler message);

}

rescheduled = true;

return;

}

}

} finally {

asyncActive = rescheduled;

}

}

进入后首先判断是否是进行异步处理的消息，如果是那么进入该位置。

进入后我们进行了try\_finally有一个变量long\_started用于标识开始时间。

当执行一个任务后就判断一次如果超过了每次占用主线程的时间限制，那么不管队列中的任务是否执行完成都退出，同时发起一个新的消息到Handler循环队列。

在while部分，我们从队列取出一个任务，采用Poll方法；判断是否为空，如果为空进入队列同步块；然后再取一次，再次判断。

如果恰巧在进入同步队列之前有新的任务来了，那么第二次取到的当然就不是 NULL也就会继续执行下去。反之，如果还是为空；那么重置当前队列的状态为false同时跳出循环。

**下面来看第二部分：**

else if (msg.what == SYNC) {

boolean rescheduled = false;

try {

long started = SystemClock.uptimeMillis();

while (true) {

SyncPost post = syncPool.poll();

if (post == null) {

synchronized (syncPool) {

// Check again, this time in synchronized

post = syncPool.poll();

if (post == null) {

syncActive = false;

return;

}

}

}

post.run();

long timeInMethod = SystemClock.uptimeMillis() - started;

if (timeInMethod >= maxMillisInsideHandleMessage) {

if (!sendMessage(obtainMessage(SYNC))) {

throw new GeniusException(Could not send handler message);

}

rescheduled = true;

return;

}

}

} finally {

syncActive = rescheduled;

}

} else super.handleMessage(msg);

首先还是判断，如果是同步任务消息就进入，如果还是不是 那么只有调用super.handleMessage(msg);了。

从上面的处理部分可以看出来其处理的过程与第一部分可以说是完全一样的。

只不过是从不同队列取出不同的类SyncPost，然后判断执行，以及发送不同标识的消息；可以说如果懂了第一部分，这部分是毫无营养的。

这里就有问题了，既然方法操作流程一样，那么同步与异步是在哪里进行区分的？

**这里就要看看SyncPost了：**

final class SyncPost {

boolean end = false;

Runnable runnable;

SyncPost(Runnable runnable) {

this.runnable = runnable;

}

public void run() {

synchronized (this) {

runnable.run();

end = true;

try {

this.notifyAll();

} catch (Exception e) {

e.printStackTrace();

}

}

}

public void waitRun() {

if (!end) {

synchronized (this) {

if (!end) {

try {

this.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

}

**首先看看SyncPost的构造函数：**

是不是传入一个Runnable接口？所以说是对Runnable 的简单封装。

**可以看见其public\_void\_run()方法：**

在该方法中我们进入了同步块，然后调用Runnable接口的run方法。同时在执行完成后将其中的一个状态变量进行了改变boolean\_end=true;

然后调用this.notifyAll();通知等待的部分可以继续了，当然有这样的情况；假如在进入该同步块的时候子线程还未执行到this.wait();部分呢？所以我们为此准备了end和try。

**然后看看public\_void\_waitRun()方法：**

在这个中，我们首先判断状态，如果状态已经变了，那么证明子线程执行到此处时，主线程以及执行了void\_run()。

所以也就不用进入同步块进行等待了，不然那还不等死啊？反之就进入进行等待直到主线程调用this.notifyAll();

**激情部分**

马上进入到完成部分了，组建都完善了那么该进行最后的组装了。

**回到类classToolKit**

public class ToolKit {

private static HandlerPoster mainPoster = null;

private static HandlerPoster getMainPoster() {

if (mainPoster == null) {

synchronized (ToolKit.class) {

if (mainPoster == null) {

mainPoster = new HandlerPoster(Looper.getMainLooper(), 20);

}

}

}

return mainPoster;

}

/\*\*

\* Asynchronously

\* The child thread asynchronous run relative to the main thread,

\* not blocking the child thread

\*

\* @param runnable Runnable Interface

\*/

public static void runOnMainThreadAsync(Runnable runnable) {

if (Looper.myLooper() == Looper.getMainLooper()) {

runnable.run();

return;

}

getMainPoster().async(runnable);

}

/\*\*

\* Synchronously

\* The child thread relative thread synchronization operation,

\* blocking the child thread,

\* thread for the main thread to complete

\*

\* @param runnable Runnable Interface

\*/

public static void runOnMainThreadSync(Runnable runnable) {

if (Looper.myLooper() == Looper.getMainLooper()) {

runnable.run();

return;

}

SyncPost poster = new SyncPost(runnable);

getMainPoster().sync(poster);

poster.waitRun();

}

public static void dispose() {

if (mainPoster != null) {

mainPoster.dispose();

mainPoster = null;

}

}

}

**其中就一个静态变量HandlerPoster**

然后一个初始化部分HandlerPoster\_getMainPoster()这里采用同步的方式进行初始化，用于适应多线程同时调用情况；当然在初始化的时候我们传入了

mainPoster=newHandlerPoster(Looper.getMainLooper(),20); 这里就决定了是在主线程执行的HandlerPoster，同时指定主线程单次运行时间为20毫秒。

**在方法void\_runOnMainThreadAsync(Runnable\_runnable)中：**

首先判断调用该方法的是否是主线程，如果是那还弄到队列中执行干嘛？直接执行啊；如果是子线程就调用getMainPoster().async(runnable);追加到队列中执行。

**而在方法void\_runOnMainThreadSync(Runnable\_runnable)中：**

public static void runOnMainThreadSync(Runnable runnable) {

if (Looper.myLooper() == Looper.getMainLooper()) {

runnable.run();

return;

}

SyncPost poster = new SyncPost(runnable);

getMainPoster().sync(poster);

poster.waitRun();

}

同样是线程判断，然后进行封装，然后丢进队列中等待执行，而在该方法中调用poster.waitRun();进行等待；直到主线程执行了SyncPost类的run方法。  
最后当然留下了一个销毁方法；妈妈说要学会清理不留垃圾：void\_dispose()

**OK，完成了**

// Runnable 类实现其中 run() 方法

// run() 运行在主线程中，可在其中进行界面操作

// 同步进入主线程,等待主线程处理完成后继续执行子线程

ToolKit.runOnMainThreadSync(Runnable runnable);

// 异步进入主线程,无需等待

ToolKit.runOnMainThreadAsync(Runnable runnable);

对外就是这么两个方法，简单便捷啊；大伙试试吧；一个字爽！