Notzuonotdied 调整HandleThread的编辑细 0d85fdf 9 days ago

2 contributors

389 lines (331 sloc) | 11.8 KB

我们知道在Android系统中，我们执行完耗时操作都要另外开启子线程来执行，执行完线程以后线程会自动销毁。想象一下如果我们在项目中经常要执行耗时操作，如果经常要开启线程，接着又销毁线程，这无疑是很消耗性能的？那有什么解决方法呢？

1. 使用线程池
2. 使用HandlerThread

本篇文章主要讲解一下问题

1. HandlerThread的使用场景以及怎样使用HandlerThread?
2. HandlerThread源码分析

HandlerThread的使用场景以及怎样使用HandlerThread?

使用场景

HandlerThread是Google帮我们封装好的，可以用来执行多个耗时操作，而不需要多次开启线程，里面是采用Handler和Looper实现的。

> Handy class for starting a new thread that has a looper. The looper can then be used to create handler classes. Note that start() must still be called.

怎样使用HandlerThread?

1. 创建HandlerThread的实例对象

```
HandlerThread handlerThread = new HandlerThread("myHandlerThread");
```

该参数表示线程的名字，可以随便选择。

2. 启动我们创建的HandlerThread线程

```
handlerThread.start();
```

1. 将我们的handlerThread与Handler绑定在一起。 还记得是怎样将Handler与线程对象绑定在一起的吗？其实很简单，就是将线程的looper与Handler绑定在一起，代码如下：

```
mThreadHandler = new Handler(mHandlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        checkForUpdate();
        if(isUpdate){
            mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
        }
    }
};
```

注意必须按照以上三个步骤来，下面在讲解源码的时候会分析其原因

完整测试代码如下

```
public class MainActivity extends AppCompatActivity {
    private static final int MSG_UPDATE_INFO = 0x100;
    Handler mMainHandler = new Handler();
    private TextView mTv;
    private Handler mThreadHandler;
    private HandlerThread mHandlerThread;
    private boolean isUpdate = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTv = (TextView) findViewById(R.id.tv);
        initHandlerThread();
    }

    private void initHandlerThread() {
        mHandlerThread = new HandlerThread("xujun");
        mHandlerThread.start();
        mThreadHandler = new Handler(mHandlerThread.getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                checkForUpdate();
                if (isUpdate) {
                    mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
                }
            }
        };
    }

    /**
     * 模拟从服务器解析数据
     */
    private void checkForUpdate() {
        try {
            //模拟耗时
            Thread.sleep(1200);
            mMainHandler.post(new Runnable() {
                @Override
                public void run() {
                    String result = "实时更新中，当前股票行情：<font color='red'>%d</font>";
                    result = String.format(result, (int) (Math.random() * 5000 + 1000));
                    mTv.setText(Html.fromHtml(result));
                }
            });
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onResume() {
        isUpdate = true;
        super.onResume();
        mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
    }

    @Override
    protected void onPause() {
        super.onPause();
        isUpdate = false;
        mThreadHandler.removeMessages(MSG_UPDATE_INFO);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mHandlerThread.quit();
        mMainHandler.removeCallbacksAndMessages(null);
    }
}
```

运行以上测试代码，将可以看到如下效果图(例子不太恰当，主要使用场景是在handleMessage中执行耗时操作)

HandlerThread源码分析

官方源代码如下，是基于sdk23的，可以看到，只有一百多行代码而已。

```
public class HandlerThread extends Thread {
    int mPriority;
    int mTid = -1;
    Looper mLooper;

    public HandlerThread(String name) {
        super(name);
        mPriority = Process.THREAD_PRIORITY_DEFAULT;
    }

    public HandlerThread(String name, int priority) {
        super(name);
        mPriority = priority;
    }

    /**
     * Call back method that can be explicitly overridden if needed to execute some
     * setup before Looper loops.
     */
    protected void onLooperPrepared() {
    }

    @Override
    public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        //持有锁机制来获得当前线程的Looper对象
        synchronized (this) {
            mLooper = Looper.myLooper();
            //发出通知，当前线程已经创建mLooper对象成功，这里主要是通知getLooper方法中的wait
            notifyAll();
        }
        //设置线程的优先级别
        Process.setThreadPriority(mPriority);
        //这里默认是空方法的实现，我们可以重写这个方法来做一些线程开始之前的准备，方便扩展
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
    }

    public Looper getLooper() {
        if (!isAlive()) {
            return null;
        }
        // 直到线程创建完Looper之后才能获得Looper对象，Looper未创建成功，阻塞
        synchronized (this) {
            while (isAlive() && mLooper == null) {
                try {
                    wait();
                } catch (InterruptedException e) {
                }
            }
        }
        return mLooper;
    }
```

```
    public boolean quit() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quit();
            return true;
        }
        return false;
    }

    public boolean quitSafely() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quitSafely();
            return true;
        }
        return false;
    }

    /**
     * Returns the identifier of this thread. See Process.myTid().
     */
    public int getThreadId() {
        return mTid;
    }
}
```

1）首先我们先来看一下它的构造方法

```
public HandlerThread(String name) {
    super(name);
    mPriority = Process.THREAD_PRIORITY_DEFAULT;
}

public HandlerThread(String name, int priority) {
    super(name);
    mPriority = priority;
}
```

有两个构造方法，一个参数的和两个参数的，name代表当前线程的名称，priority为线程的优先级别

2）接着我们来看一下run()方法，在run方法里面我们可以看到我们会初始化一个Looper，并设置线程的优先级别

```
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    //持有锁机制来获得当前线程的Looper对象
    synchronized (this) {
        mLooper = Looper.myLooper();
        //发出通知，当前线程已经创建mLooper对象成功，这里主要是通知getLooper方法中的wait
        notifyAll();
    }
    //设置线程的优先级别
    Process.setThreadPriority(mPriority);
    //这里默认是空方法的实现，我们可以重写这个方法来做一些线程开始之前的准备，方便扩展
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

- 还记得我们前面我们说到使用HandlerThread的时候必须调用 `start()` 方法，接着才可以将我们的HandlerThread和我们的handler绑定在一起吗?其实原因就是我们是在 `run()` 方法才开始初始化我们的looper，而我们调用HandlerThread的 `start()` 方法的时候，线程会交给虚拟机调度，由虚拟机自动调用run方法：

```
mHandlerThread.start();
mThreadHandler = new Handler(mHandlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        checkForUpdate();
        if(isUpdate){
            mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
```

```
        }
    }
};
```

- 这里我们为什么要使用锁机制和`notifyAll()`;，原因我们可以从`getLooper()`方法中知道

```
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }
    // 直到线程创建完Looper之后才能获得Looper对象，Looper未创建成功，阻塞
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}
```

总结：在获得mLooper对象的时候存在一个同步的问题，只有当线程创建成功并且Looper对象也创建成功之后才能获得mLooper的值。这里等待方法wait和run方法中的notifyAll方法共同完成同步问题。

3)接着我们来看一下quit方法和quitSafe方法

```
//调用这个方法退出Looper消息循环，及退出线程
public boolean quit() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quit();
        return true;
    }
    return false;
}
//调用这个方法安全地退出线程
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
public boolean quitSafely() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quitSafely();
        return true;
    }
    return false;
}
```

跟踪这两个方法容易知道只两个方法最终都会调用MessageQueue的`quit(boolean safe)`方法

```
void quit(boolean safe) {
    if (!mQuitAllowed) {
        throw new IllegalStateException("Main thread not allowed to quit.");
    }
    synchronized (this) {
        if (mQuitting) {
            return;
        }
        mQuitting = true;
        //安全退出调用这个方法
        if (safe) {
            removeAllFutureMessagesLocked();
        } else {//不安全退出调用这个方法
            removeAllMessagesLocked();
        }
        // We can assume mPtr != 0 because mQuitting was previously false.
        nativeWake(mPtr);
    }
}
```

不安全的会调用`removeAllMessagesLocked();`这个方法，我们来看这个方法是怎样处理的，其实就是遍历Message链表，移除所有信息的回调，并重置为null。

```
private void removeAllMessagesLocked() {
    Message p = mMessages;
    while (p != null) {
        Message n = p.next;
        p.recycleUnchecked();
        p = n;
    }
    mMessages = null;
}
```

安全地会调用`removeAllFutureMessagesLocked();`这个方法，它会根据Message.when这个属性，判断我们当前消息队列是否正在处理消息，没有正在处理消息的话，直接移除所有回调，正在处理的话，等待该消息处理处理完毕再退出该循环。因此说`quitSafe()`是安全的，而`quit()`方法是不安全的，因为quit方法不管是否正在处理消息，直接移除所有回调。

```
private void removeAllFutureMessagesLocked() {
    final long now = SystemClock.uptimeMillis();
    Message p = mMessages;
    if (p != null) {
        //判断当前队列中的消息是否正在处理这个消息，没有的话，直接移除所有回调
        if (p.when > now) {
            removeAllMessagesLocked();
        } else {//正在处理的话，等待该消息处理处理完毕再退出该循环
            Message n;
            for (;;) {
                n = p.next;
                if (n == null) {
                    return;
                }
                if (n.when > now) {
                    break;
                }
                p = n;
            }
            p.next = null;
            do {
                p = n;
                n = p.next;
                p.recycleUnchecked();
            } while (n != null);
        }
    }
}
```