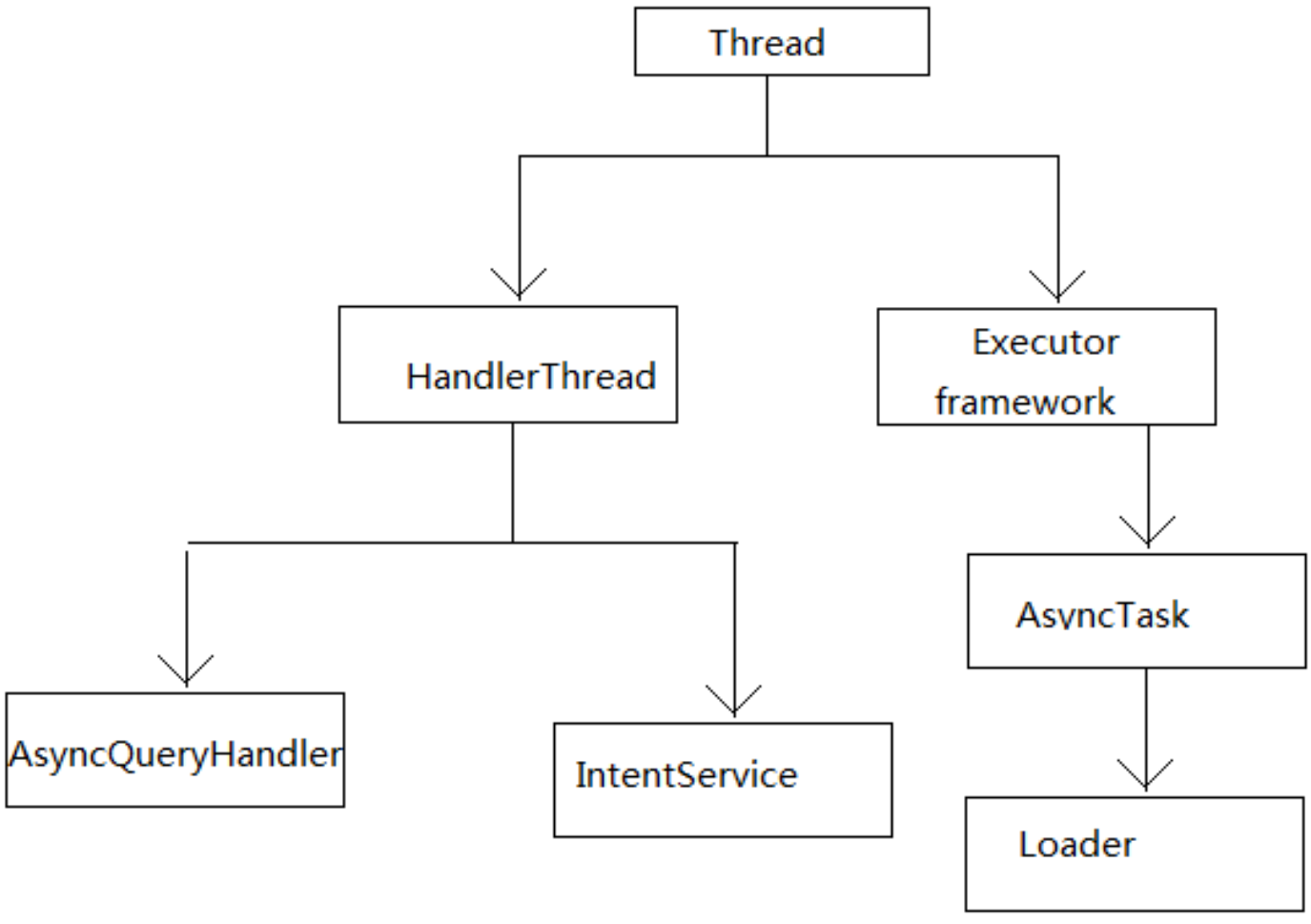
1008711
2017年05月02日 阅读 963

棒棒糖之——Android中全套异步处理的详细讲解

一、前言

在应用的开发中我们正确处理好主线程和子线程之间的关系，耗时的操作都放到子线程中处理，避免阻塞主线程，导致ANR。异步处理技术是提高应用性能，解决主线程和子线程之间通信问题的关键。

首先看一个异步技术链：



二、Thread

Thread是Android中异步处理技术的基础，创建线程有两种方法。

- 继承Thread类并重写run方法，如下：

```
public class MyThread extends Thread {

    @Override
    public void run() {
        super.run();
    }

    public void startThread() {
        MyThread myThread = new MyThread();
        myThread.start();
    }
```

```
}
```

- 实现Runnable接口并实现run方法，如下：

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {

    }

    public void startThread() {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }

}
```

Android应用各种类型的线程本质上基于linux系统的pthreads，在应用层可以分为三种类型线程。

- 主线程：主线程也称为UI线程，随着应用启动而启动，主线程用来运行Android组件，同时刷新屏幕上的UI元素。Android系统如果检测到非主线程更新UI组件，那么就会抛出CalledFromWrongThreadException异常，只有主线程才能操作UI，是因为Android的UI工具包不是线程安全的。主线程中创建的Handler会顺序执行接受到的消息，包括从其他线程发送的消息。因此，如果消息队列中前面的消息没有很快执行完，那么它可能会阻塞队列中的其他消息的及时处理。
- Binder线程：Binder线程用于不通过进程之间线程的通信，每个进程都维护了一个线程池，用来处理其他进程中线程发送的消息，这些进程包括系统服务、Intents、ContentProviders和Service等。在大部分情况下，应用不需要关心Binder线程，因为系统会优先将请求转换为使用主线程。一个典型的需要使用Binder线程的场景是应用提供一个给其他进程通过AIDL接口绑定的Service。
- 后台线程：在应用中显式创建的线程都是后台线程，也就是当刚创建出来时，这些线程的执行体是空的，需要手动添加任务。在Linux系统层面，主线程和后台线程是一样的。在Android框架中，通过WindowManager赋予了主线程只能处理UI更新以及后台线程不能直接操作UI的限制。

三、HandlerThread

HandlerThread是一个集成了Looper和MessageQueue的线程，当启动HandlerThread时，会同时生成Looper和MessageQueue，然后等待消息进行处理，它的run方法源码：

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

使用HandlerThread的好处是开发者不需要自己去创建和维护Looper，它的用法和普通线程一样，如下：

```
HandlerThread handlerThread = new HandlerThread("HandlerThread");
handlerThread.start();

handler = new Handler(handlerThread.getLooper()){
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);

        //处理接受的消息
    }
};
```

HandlerThread中只有一个消息队列，队列中的消息是顺序执行的，因此是线程安全的，当然吞吐量自然受到一定影响，队列中的任务可能会被前面没有执行完的任务阻塞。HandlerThread的内部机制确保了在创建Looper和发送消息之间不存在竞态条件（是指一个在设备或者系统试图同时执行两个操作的时候出现的不希望的状况，但是由于设备和系统的自然特性，为了正确地执行，操作必须按照合适顺序进行），这个是通过将HandlerThread.getLooper()实现为一个阻塞操作实现的，只有当HandlerThread准备好接受消息之后才会返回，源码如下：

```
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }

    // 如果线程已经启动，那么在Looper准备好之前应先等待
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}
```

如果具体业务要求在HandlerThread开始接受消息之前要进行某些初始化操作的话，可以重写HandlerThread的onLooperPrepared函数，例如可以在这个函数中创建于HandlerThread关联的Handler实例，这同时也可以对外隐藏我们的Handler实例，代码如下：

```
public class MyHandlerThread extends HandlerThread {

    private Handler mHandler;

    public MyHandlerThread() {
        super("MyHandlerThread", Process.THREAD_PRIORITY_BACKGROUND);
    }


    @Override
    protected void onLooperPrepared() {
        super.onLooperPrepared();
        mHandler = new Handler(getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    case 1:

                        break;
                    case 2:

                        break;
                }
```

```
            }
        };
    }


    public void publishedMethod1() {
        mHandler.sendEmptyMessage(1);
    }


    public void publishedMethod2() {
        mHandler.sendEmptyMessage(2);
    }

}
```

四、AsyncQueryHandler

AsyncQueryHandler是用于在ContentProvider上面执行异步的CRUD操作的工具类，CRUD操作会被放到一个单独的子线程中执行，当操作结束获取到结果后，将通过消息的方式传递给调用AsyncQueryHandler的线程，通常就是主线程。AsyncQueryHandler是一个抽象类，集成自Handler，通过封装ContentResolver、HandlerThread、AsyncQueryHandler等实现对ContentProvider的异步操作。

AsyncQueryHandler封装了四个方法操作ContentProvider，对应CRUD如下：

```
public final void startDelete(int token, Object cookie, Uri uri,String selection, String[] selectio
```

```
public final void startInsert(int token, Object cookie, Uri uri,ContentValues initialValues);
```

```
public void startQuery(int token, Object cookie, Uri uri,
        String[] projection, String selection, String[] selectionArgs,
        String orderBy);
```

```
public final void startUpdate(int token, Object cookie, Uri uri,
        ContentValues values, String selection, String[] selectionArgs);
```

AsyncQueryHandler的子类可以根据实际需求实现下面的回调函数，对应上面操作的CRUD操作的返回结果。

```
/**
 * Called when an asynchronous query is completed.
 *
 * @param token the token to identify the query, passed in from
 *            {@link #startQuery}.
 * @param cookie the cookie object passed in from {@link #startQuery}.
 * @param cursor The cursor holding the results from the query.
 */
protected void onQueryComplete(int token, Object cookie, Cursor cursor) {
    // Empty
}

/**
 * Called when an asynchronous insert is completed.
 *
 * @param token the token to identify the query, passed in from
 *        {@link #startInsert}.
 * @param cookie the cookie object that's passed in from
 *        {@link #startInsert}.
 * @param uri the uri returned from the insert operation.
 */
protected void onInsertComplete(int token, Object cookie, Uri uri) {
    // Empty
}

/**
 * Called when an asynchronous update is completed.
 *
 * @param token the token to identify the query, passed in from
 *        {@link #startUpdate}.
 * @param cookie the cookie object that's passed in from
 *        {@link #startUpdate}.
 * @param result the result returned from the update operation
 */
protected void onUpdateComplete(int token, Object cookie, int result) {
    // Empty
}

/**
 * Called when an asynchronous delete is completed.
 *
 * @param token the token to identify the query, passed in from
 *        {@link #startDelete}.
```

```
     * @param cookie the cookie object that's passed in from
     *        {@link #startDelete}.
     * @param result the result returned from the delete operation
     */
    protected void onDeleteComplete(int token, Object cookie, int result) {
        // Empty
    }
```

五、IntentService

Service的各个生命周期函数是运行在主线程，因此它本身并不是一个异步处理技术。为了能够在Service中实现在子线程中处理耗时任务，Android引入了一个Service的子类：IntentService。IntentService具有Service一样的生命周期，同时也提供了在后台线程中处理异步任务的机制。与HandlerThread类似，IntentService也是在一个后台线程中顺序执行所有的任务，我们通过给Context.startService传递一个Intent类型的参数可以启动IntentService的异步执行，如果此时IntentService正在运行中，那么这个新的Intent将会进入队列进行排队，直到后台线程处理完队列前面的任务；如果此时IntentService没有在运行，那么将会启动一个新的IntentService，当后台线程队列中所有任务处理完成之后，IntentService将会结束它的生命周期，因此IntentService不需要开发者手动结束。

IntentService本身是一个抽象类，因此，使用前需要继承它并实现onHandlerIntent方法，在这个方法中实现具体的后台处理业务逻辑，同时在子类的构造方法中需要调用super(String name)传入子类的名字，如下：

```
public class SimpleIntentService extends IntentService {


    public SimpleIntentService() {
        super(SimpleIntentService.class.getName());
        setIntentRedelivery(true);

    }

    @Override
    protected void onHandleIntent(Intent intent) {
        //该方法在后台调用
    }
}
```

上面代码中的setIntentRedelivery方法如果设置为true，那么IntentService的onStartCOmmand方法将会返回START_REDELIVER_INTENT。这时，如果onHandlerIntent方法返回之前进程死掉了，那么进程将会重新启动，intent将会重新投递。

当然，类似Service，不要忘记在AndroidManifest.xml文件中注册SimpleIntentService 。

```
<service android:name=".SimpleIntentService" />
```

通过查看IntentService 的源码，我们可以发现事实上IntentService 是通过HandlerThread来实现后台任务的处理的，代码逻辑很简单：

```
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }


    public IntentService(String name) {
        super();
        mName = name;
    }


    public void setIntentRedelivery(boolean enabled) {
        mRedelivery = enabled;
    }

    @Override
    public void onCreate() {
```

```
        super.onCreate();
        HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
        thread.start();

        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }

    @Override
    public void onStart(Intent intent, int startId) {
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent;
        mServiceHandler.sendMessage(msg);
    }


    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        onStart(intent, startId);
        return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
    }

    @Override
    public void onDestroy() {
        mServiceLooper.quit();
    }


    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }


    @WorkerThread
    protected abstract void onHandleIntent(Intent intent);
}
```

六、Executor Framework

创建和销毁对象是存在开销的，在应用中频繁出现线程的创建和销毁，那么会影响到应用的性能，使

用Executor框架可以通过线程池机制解决这个问题，改善应用的体验。Executor框架为开发者提供了如下：

- 创建工作线程池，同时通过队列来控制能够在这些线程执行的任务的个数。
- 检测导致线程意外终止的错误。
- 等待线程执行完成并获取执行结果。
- 批量执行线程，并通过固定的顺序获取执行结构。
- 在合适的时机启动后台线程，从而保证线程执行结果可以很快反馈给用户。

Executor框架的基础是一个名为Executor的接口定义，Executor的主要目的是分离任务的创建和它的执行，最终实现上述功能点。

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

开发者通过实现Executor接口并重写execute方法从而实现自己的Executor类，最简单的是直接在这个方法中创建一个线程来执行Runnable。

```
public class SimpleExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        new Thread(command).start();
    }
}
```

线程池是任务队列和工作线程的集合，这两者组合起来实现生产者消费者模式。Executor框架为开发者提供了预定义的线程池实现。

- 固定大小的线程池：

```
Executors.newFixedThreadPool(3);
```

- 可变大小的线程池：

```
Executors.newCachedThreadPool();
```

当有新任务需要执行时，线程池会创建新的线程来处理它，空闲的线程池会等待60秒来执行新任务，当没有任务可执行时就自动销毁，因此可变大小线程池会根据任务队列的大小而变化。

- 单个线程的线程池：

```
Executors.newSingleThreadExecutor();
```

这个线程池中永远只有一个线程来串行执行任务队列中的任务。

预定义的线程池都是基于ThreadPoolExecutor类之上构建的，而通过ThreadPoolExecutor开发者可以自定义线程池的一些行为，我们主要来看看这个类的构造函数：

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

- corePoolSize：核心线程数，核心线程会一直在于线程池中，即使当前没有任务需要处理；当线程数小于核心线程数时，即使当前有空闲的线程，线程池也会优先创建新的线程来处理任务。
- maximumPoolSize：最大线程数，当线程数大于核心线程数，且任务队列已经满了，这时线程池就会创建新的线程，直到线程数量达到最大线程数为止。
- keepAliveTime：线程的空闲存活时间，当线程的空闲时间超过这个之时，线程会被撤毁，直到线程数等于核心线程数。
- unit：keepAliveTime的单位，可选的有TimeUnit 类中的

```
NANOSECONDS,//微秒
MICROSECONDS,//毫秒
MILLISECONDS,// 毫微秒
SECONDS // 秒
```

- workQueue：线程池所有使用的任务缓冲队列。

七、AsyncTask

AsyncTask是在Executor框架基础上进行的封装，它实现将耗时任务移动到工作线程中执行，同时提供方便的接口实现工作线程和主线程的通信，使用AsyncTask一般会用到如下：

```
public class FullTask extends AsyncTask<String,String,String> {

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        //主线程执行
    }

    @Override
    protected String doInBackground(String... params) {
        return null;
        //子线程执行
    }

    @Override
    protected void onProgressUpdate(String... values) {
        super.onProgressUpdate(values);
        //主线程执行
    }

    @Override
    protected void onPostExecute(String s) {
        super.onPostExecute(s);
        //主线程执行
    }

    @Override
    protected void onCancelled() {
        super.onCancelled();
        //主线程执行
```

```
    }

}
```

一个应用中使用的所有AsyncTask实例会共享全局的属性，也就是说如果AsnycTask中的任务是串行执行，那么应用中所有的AsyncTask都会进行排队，只有等前面的任务执行完成之后，才会接着执行下一个AsnycTask中的任务，在executeOnExecutor(AsyncTask.SERIAL_EXECUTOR)或者API大于13的系统上面执行execute()方法，都会是这个效果；如果AsyncTask是异步执行，那么在四核的CPU系统上，最多只有五个任务可以同时进行，其他任务需要在队列中排队，等待空闲的线程。之所以会出现这种情况是由于AsyncTask中的ThreadPoolExecutor指定核心线程数是系统CPU核数+1，如下：

```
public abstract class AsyncTask<Params, Progress, Result> {
    private static final String LOG_TAG = "AsyncTask";

    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);

    /**
     * An {@link Executor} that can be used to execute tasks in parallel.
     */
    public static final Executor THREAD_POOL_EXECUTOR
            = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE,
                    TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
}
```

八、Loader

Loader是Android3.0开始引入的一个异步数据加载框架，它使得在Activity或者Fragment中异步加载数据变得简单，同时它在数据源发生变化时，能够及时发出消息通知。Loader框架涉及的API如下：

- Loader：加载器框架的基类，封装了实现异步数据加载的接口，当一个加载器被激活后，它就会开始监听数据源并在数据发生改变时发送新的结果。
- AsyncTaskLoader：Loader的子类，它是基于AsyncTask实现的异步数据加载，它是一个抽象类，子类必须实现loadInBackground方法，在其中进行具体的数据加载操作。
- CursorLoader：AsyncTaskLoader的子类，封装了对ContentResolver的query操作，实现从ContentProvider中查询数据的功能。
- LoaderManager：抽象类，Activity和Fragment默认都会关联一个LoaderManager的对象，开发者只需要通过getLoaderManager即可获取。LoaderManager是用来管理一个或者多个加载器对象的。
- LoaderManager.LaoderCallbacks：LoaderManager的回调接口，有以下三个方法
  - onCreateLoader（）：初始化并返回一个新的Loader实例。
  - onLoadFinished（）：当一个加载器完成加载过程之后会回调这个方法。
  - onLoaderReset（）：当一个加载器被重置并且数据无效时会回调这个方法。

```
public class ContactActivity extends ListActivity implements LoaderManager.LoaderCallbacks<Cursor>

    private static final int CONTACT_NAME_LOADER_ID = 0;


    static final String[] CONTACTS_SUMMARY_PROJECTION = new String[]{ContactsContract.Contacts._ID,

    SimpleCursorAdapter mAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        initAdapter();
        //通过LoaderManger初始化Loader，这会回调到onCreateLoader
        getLoaderManager().initLoader(CONTACT_NAME_LOADER_ID, null, this);

    }

    private void initAdapter() {
        mAdapter = new SimpleCursorAdapter(this, android.R.layout.simple_list_item_1, null, new Str
        setListAdapter(mAdapter);
    }

    @Override
```

```
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        //实际创建Loader的地方  此处使用CursorLoader
        return new CursorLoader(this, ContactsContract.Contacts.CONTENT_URI, CONTACTS_SUMMARY_PROJE
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        //后台线程中加载完数据后，回调这个方法将数据传递给主线程
        mAdapter.swapCursor(data);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        //Loader 被重置后的回调，在这里可以重新刷新页面数据
        mAdapter.swapCursor(null);
    }
}
```

九、总结

根据以上列出的异步处理技术，使用的时候需要根据以下结果因素：

- 尽量使用更少的系统资源，例如cpu和内存等。
- 为应用提供更好的性能和响应度。
- 实现和使用起来不复杂。
- 写出来的代码是否符合好的设计，是否易于理解和维护。