Android 异步处理

王应平

2017.8.4

第一部分：线程相关知识

简单说下：

多线程开发是每个程序员不可避免的开发方式，或多或少的，都会遇到多线程问题。采用多线程开发有诸多好处，例如：提高程序运行效率，提升用户体验，使项目结构更加解耦，功能模块代码复用率更高等等。但是开发的过程，也会带来很多复杂的和不易察觉的问题。所以对待线程开发问题，我们要慎重。

实现线程的方式：

- 继承Thread类，并重写 run()方法
- 实现Runnable接口，并重写run()方法

实现Runnable接口比继承Thread类所具有的优势：

1）适合多个相同的程序代码的线程去处理同一个资源

2）可以避免java中的单继承的限制

3）增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

开启线程有很多方式：

- new Thread().start();
- new Thread(new RunnableXXX()).start();
- Executors.newCachedThreadPool().submit(new RunnableXXX());
- ……

ThreadGroup 线程组

- 对一批线程进行分类管理
- 不指定默认为Main线程组
- 只有创建之前才能制定其所在的线程组
- 线程组与线程之间的结构类似于树形的结构
- 统一处理线程组未捕获的异常（ThreadGroup实现了 Thread.UncaughtExceptionHandler 接口）
- 可以设置线程组的最高优先级 （setMaxPriority(int pri)）
- 只能访问自己线程组的信息，不能访问父线程组或者其他线程组信息

ThreadPool 线程池

线程池类似于数据库连接池，是为了提高效率而采用的一种可配置、可动态调节的资源容量池

创建线程池的两种方式：

- 使用Java API 提供的 Executors 工厂类创建
- 自定义创建

自定义创建线程池

线程池中最核心的一个类：

```
public class ThreadPoolExecutor extends AbstractExecutorService {

    .....
    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
            BlockingQueue<Runnable> workQueue);


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

    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
            BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);


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

    .....
}
```

自定义创建线程池

参数详解：

- corePoolSize：核心池的大小，对线程池的实现原理有非常大的关系。相关方法：prestartAllCoreThreads()和prestartCoreThread()
- maximumPoolSize：线程池最大线程数
- keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。相关方法：allowCoreThreadTimeOut(boolean)
- unit：参数keepAliveTime的时间单位 TimeUnit
- workQueue：一个阻塞队列，用来存储等待执行的任务
- threadFactory：线程工厂，主要用来创建线程

- handler：

  表示当拒绝处理任务时的策略，取下面的四个值：

  1. ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
  2. ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
  3. ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
  4. ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

使用API提供的 Executors 工厂类来创建线程池

- newCachedThreadPool()：创建一个具有缓存功能的线程池，系统根据需要创建线程，这些线程将会被缓存在线程池中。
- newFixedThreadPool(int nThreads)：创建一个可重用的、具有固定线程数的线程池
- newScheduledThreadPool(int corePoolSize)：创建具有指定线程数的线程池，它可以在指定延迟后执行线程任务。
- newWorkStealingPool(int parallelism)：创建持有足够的线程的线程池来支持给定的并行级别，该方法还会使用多个队列来减少竞争。

带返回参数的CallBack 和 Future接口

```
public interface Callable<V> {

    V call() throws Exception;

}
```

```
public interface Future<V>{

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

FutureTask

```
public class FutureTask<V> implements RunnableFuture<V> {
        ……
}
```

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

```
public class CallableAndFuture {
    public static void main(String[] args) {
        Callable<Integer> callable = new Callable<Integer>() {
            public Integer call() throws Exception {
                return new Random().nextInt(100);
            }
        };
        FutureTask<Integer> future = new FutureTask<Integer>(callable);
        new Thread(future).start();
        try {
            Thread.sleep(5000);// 可能做一些事情
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

第二部分：Android中的线程运用

Android中能实现操作的方式有哪些：

1、AsyncTask

2、HandlerThread

3、ThreadPool

4、IntentService

5、LoaderManager.LoaderCallbacks

……

AsyncTask

定义：

public abstract class AsyncTask<Params, Progress, Result> {}

对线程池的包装，是对Thread+Handler良好的封装

使用AsyncTask必须注意的几个地方：

1.异步任务的实例必须在UI线程中创建。

2.execute(Params... params)方法必须在UI线程中调用。

3.不要手动调用onPreExecute()，doInBackground(Params... params)，onProgressUpdate(Progress... values)，onPostExecute(Result result)这几个方法。

4.不能在doInBackground(Params... params)中更改UI组件的信息。

5.一个任务实例只能执行一次，如果执行第二次将会抛出异常。

6.在创建匿名内部类的时候，注意内存泄漏

- AsyncTask
  - SerialExecutor
  - Status
  - InternalHandler
  - WorkerRunnable
  - AsyncTaskResult
  - static class initializer ThreadPoolExecutor t...
  - AsyncTask()
  - getHandler(): Handler
  - setDefaultExecutor(Executor): void
  - postResultIfNotInvoked(Result): void
  - postResult(Result): Result
  - getStatus(): Status
  - doInBackground(Params...): Result
  - onPreExecute(): void
  - onPostExecute(Result): void
  - onProgressUpdate(Progress...): void
  - onCancelled(Result): void
  - onCancelled(): void
  - isCancelled(): boolean
  - cancel(boolean): boolean
  - get(): Result
  - get(long, TimeUnit): Result
  - execute(Params...): AsyncTask<Params, Progress, Result>
  - executeOnExecutor(Executor, Params...): AsyncTask<Params, Progress, Result>
  - execute(Runnable): void
  - publishProgress(Progress...): void
  - finish(Result): void

AsyncTask源码浅析

```
//初始状态
private volatile Status mStatus = Status.PENDING;

public enum Status {
    /**
     * Indicates that the task has not been executed yet.
     */
    PENDING,
    /**
     * Indicates that the task is running.
     */
    RUNNING,
    /**
     * Indicates that {@link AsyncTask#onPostExecute} has finished.
     */
    FINISHED,
}

/**
 * Returns the current status of this task.
 *
 * @return The current status.
 */
public final Status getStatus() {
    return mStatus;
}
```

AsyncTask的初始状态为PENDING，代表待定状态，RUNNING代表执行状态，FINISHED代表结束状态

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                //如果该任务正在被执行则抛出异常
                //值得一提的是,在调用cancel取消任务后,状态仍未RUNNING
                throw new IllegalStateException("Cannot execute task:"
                        + " the task is already running.");
            case FINISHED:
                //如果该任务已经执行完成则抛出异常
                throw new IllegalStateException("Cannot execute task:"
                        + " the task has already been executed "
                        + "(a task can be executed only once)");
        }

    //改变状态为RUNNING
    mStatus = Status.RUNNING;

    //调用onPreExecute方法，这个地方也解释了为什么要在UI线程调用这个方法
    onPreExecute();

    mWorker.mParams = params;
    sExecutor.execute(mFuture);

    return this;
}
```

```
private static final int CORE_POOL_SIZE = 5;
private static final int MAXIMUM_POOL_SIZE = 128;
private static final int KEEP_ALIVE = 10;

//新建一个队列用来存放线程
private static final BlockingQueue<Runnable> sWorkQueue =
        new LinkedBlockingQueue<Runnable>(10);
//新建一个线程工厂
private static final ThreadFactory sThreadFactory = new ThreadFactory() {
    private final AtomicInteger mCount = new AtomicInteger(1);
    //新建一个线程
    public Thread newThread(Runnable r) {
        return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
    }
};
//新建一个线程池执行器,用于管理线程的执行
private static final ThreadPoolExecutor sExecutor = new ThreadPoolExecut
or(CORE_POOL_SIZE,  MAXIMUM_POOL_SIZE, KEEP_ALIVE, TimeUnit.S
ECONDS, sWorkQueue, sThreadFactory);
```

AsyncTask总结

优点：

简单、方便快捷、过程可控

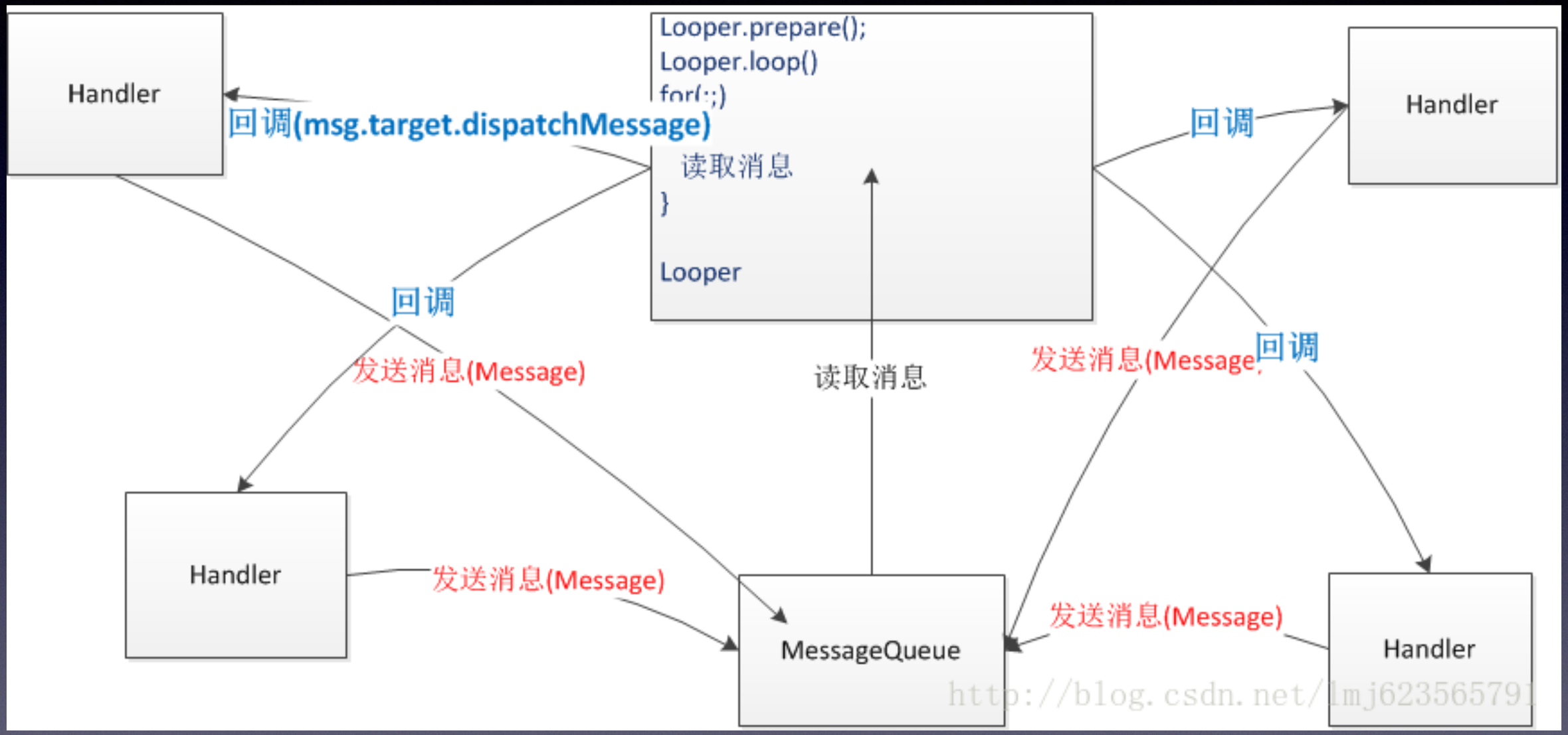
缺点：

1、如果不指定线程池，异步任务会串行执行

2、在使用多个异步操作和并需要进行Ui变更时,就变得复杂起来

3、在doInBackground( ) 方法中，需要做isCancelled( )判断，代码耦合度增高

Looper、Handler、Message三者的关系



Looper 源码浅析

一个线程中只有一个Looper实例：

```
private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));
}
```

在构造方法中，创建了一个MessageQueue（消息队列）：

```
private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}
```

```
public static void loop() {
    //获取    sThreadLocal.get();
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;
     ……
    for (;;) {
        //取出一条消息，可能会阻塞
        Message msg = queue.next(); // might block
        if (msg == null) {
            // No message indicates that the message queue is quitting.
            return;
        }
        ……
        final long traceTag = me.mTraceTag;
        if (traceTag != 0 && Trace.isTagEnabled(traceTag)) {
            Trace.traceBegin(traceTag, msg.target.getTraceName(msg));
        }
        try {
            msg.target.dispatchMessage(msg);
        } finally {
            if (traceTag != 0) {
                Trace.traceEnd(traceTag);
            }
        }
        msg.recycleUnchecked();
    }
}
```

Looper主要作用：

1、与当前线程绑定，保证一个线程只会有一个Looper实例，同时一个Looper实例也只有一个MessageQueue。

2、loop()方法，不断从MessageQueue中去取消息，交给消息的target属性的dispatchMessage去处理。

Handler源码浅析

```
public Handler(Callback callback, boolean async) {
    if (FIND_POTENTIAL_LEAKS) {
        final Class<? extends Handler> klass = getClass();
        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&
                (klass.getModifiers() & Modifier.STATIC) == 0) {
            Log.w(TAG, "The following Handler class should be static or leaks might occur: " +
                klass.getCanonicalName());
        }
    }

    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
            "Can't create handler inside thread that has not called Looper.prepare()");
    }
    //handler的实例与我们Looper实例中MessageQueue关联上
    mQueue = mLooper.mQueue;
    mCallback = callback;
    mAsynchronous = async;
}
```

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0);
}

public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}

//最终把消息保存到消息队列中去
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

流程总结

1、首先Looper.prepare()在本线程中保存一个Looper实例，然后该实例中保存一个MessageQueue对象；因为Looper.prepare()在一个线程中只能调用一次，所以MessageQueue在一个线程中只会存在一个。
2、Looper.loop()会让当前线程进入一个无限循环，不端从MessageQueue的实例中读取消息，然后回调msg.target.dispatchMessage(msg)方法。
3、Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue相关联。
4、Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。
5、在构造Handler实例时，我们会重写handleMessage方法，也就是msg.target.dispatchMessage(msg)最终调用的方法。

典型代码

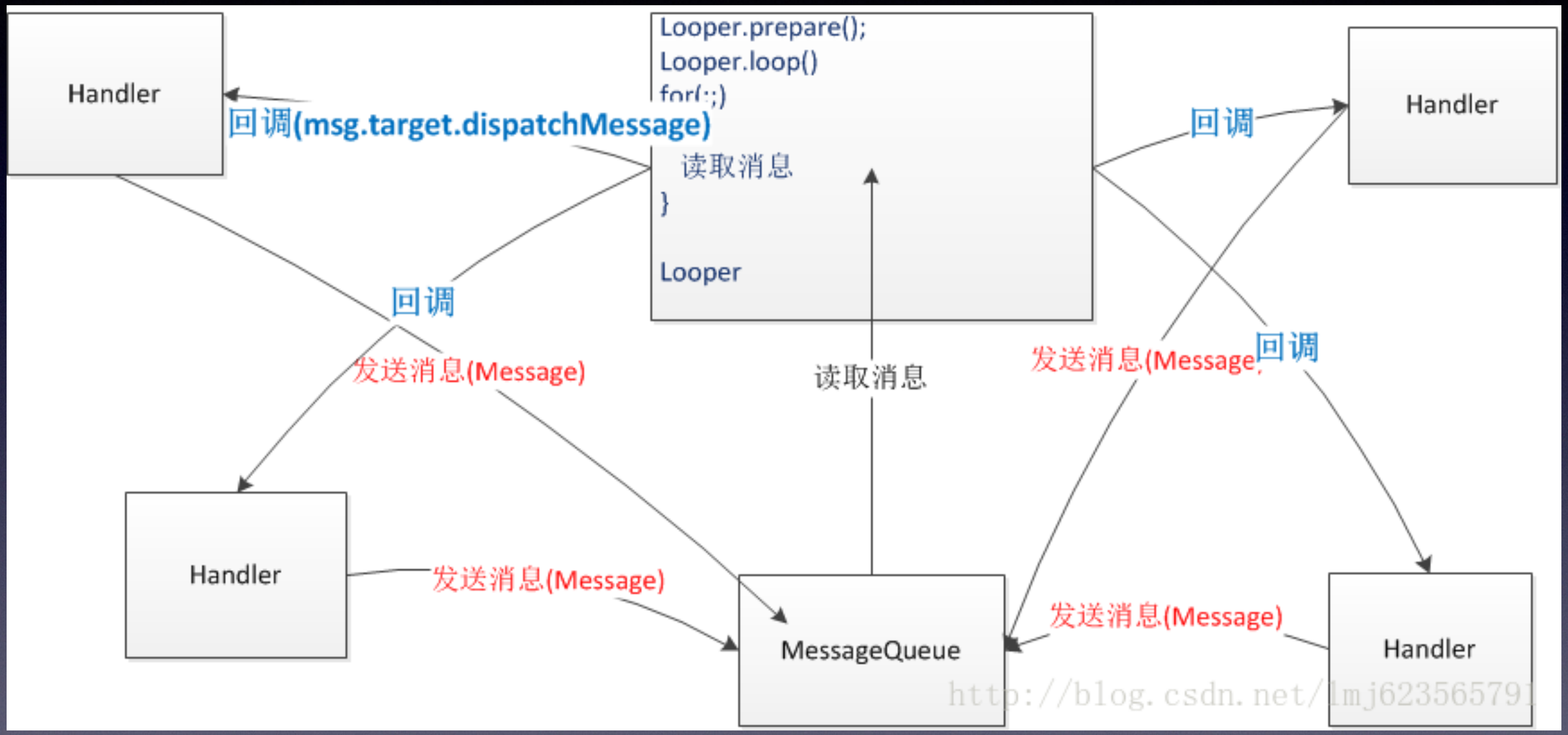
```
class LooperThread extends Thread {
        public Handler mHandler;

        public void run() {
            Looper.prepare();

            mHandler = new Handler() {
                public void handleMessage(Message msg) {
                    // process incoming messages here
                }
            };

            Looper.loop();
        }
}
```

Looper、Handler、Message三者的关系



```
public class HandlerThread extends Thread {
    ……
    protected void onLooperPrepared() {
    }

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

    public Looper getLooper() {
        if (!isAlive()) {
            return null;
        }
        synchronized (this) {
            while (isAlive() && mLooper == null) {
                try {
                    wait();
                } catch (InterruptedException e) {
                }
            }
        }
        return mLooper;
    }
    ……
}
```

```
public abstract class IntentService extends Service {
    ……

    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);
        }
    }

    @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;
    }

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

总结

AsyncTask: 为 UI 线程与工作线程之间进行快速的切换提供一种简单便捷的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的使用场景。

HandlerThread: 为某些回调方法或者等待某些任务的执行设置一个专属的线程，并提供线程任务的调度机制。

ThreadPool: 把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。

IntentService: 适合于执行由 UI 触发的后台 Service 任务，并可以把后台任务执行的情况通过一定的机制反馈给 UI。

……

谢谢大家！