在 Android 中，线程内部或者线程之间进行信息交互时经常会使用消息，这些基础的东西如果我们熟悉其内部的原理，将会使我们容易、更好地架构系统，避免一些低级的错误。在学习 Android 中消息机制之前，我们先了解与消息有关的几个类：

1.Message

消息对象，顾名思义就是记录消息信息的类。这个类有几个比较重要的字段：

a.arg1 和 arg2：我们可以使用两个字段用来存放我们需要传递的整型值，在 Service 中，我们可以用来存放 Service 的 ID。

b.obj：该字段是 Object 类型，我们可以让该字段传递某个多项到消息的接受者中。

c.what：这个字段可以说是消息的标志，在消息处理中，我们可以根据这个字段的不同的值进行不同的处理，类似于我们在处理 Button 事件时，通过 switch（v.getId（））判断是点击了哪个按钮。

在使用 Message 时，我们可以通过 new Message（）创建一个 Message 实例，但是 Android 更 推荐我们通过 Message.obtain()或者 Handler.obtainMessage（）获取 Message 对象。这并不一定是直接创建一个新 的实例，而是先从消息池中看有没有可用的 Message 实例，存在则直接取出并返回这个实例。反之如果消息池中没有可用的 Message 实例，则根据给定 的参数 new 一个新 Message 对象。通过分析源码可得知，Android 系统默认情况下在消息池中实例化 10 个 Message 对象。

2.MessageQueue

消息队列，用来存放 Message 对象的数据结构，按照“先进先出”的原则存放消息。存放并非实际意义的保存，而是将 Message 对象以链表的方 式串联起来的。MessageQueue 对象不需要我们自己创建，而是有 Looper 对象对其进行管理，一个线程最多只可以拥有一个 MessageQueue。我们可以通过 Looper.myQueue（）获取当前线程中的 MessageQueue。

3.Looper

MessageQueue 的管理者，在一个线程中，如果存在 Looper 对象，则必定存在 MessageQueue 对象，并且只存在一个 Looper 对象和一个 MessageQueue 对象。在 Android 系统中，除了主线程有默认的 Looper 对象，其它线程默认是没有 Looper 对象。如果想让我们新创建的线程拥有 Looper 对象时，我们首先应调用 Looper.prepare（）方法，然后再调用 Looper.loop（）方法。典型的用法如下：

view plaincopy to clipboardprint?

```
class LooperThread extends Thread

{

public Handler mHandler;

public void run ()

{

Looper.prepare () ;

//其它需要处理的操作

Looper.loop () ;

}

}
```

倘若我们的线程中存在 Looper 对象，则我们可以通过 Looper.myLooper() 获取，此外我们还可以通过 Looper.getMainLooper () 获取当前应用系统中主线程的 Looper 对象。在这个地方有一点需要注意，假如 Looper 对象位于应用程序主线程中，则 Looper.myLooper () 和 Looper.getMainLooper () 获取的是同一个对象。

4.Handler

消息的处理者。通过 Handler 对象我们可以封装 Message 对象，然后通过 sendMessage (msg) 把 Message 对象添加到 MessageQueue 中；当 MessageQueue 循环到该 Message 时，就会调用该 Message 对象对应的 handler 对象的 handleMessage () 方法对其进行处理。由于是在 handleMessage () 方法中处理消息，因此我们应该编写一个类继承自 Handler，然后在 handleMessage () 处理我们需要的操作。

```
view plaincopy to clipboardprint?

# public class MessageService extends Service

#

# {
```

```
# private static final String TAG = “MessageService”;

# private static final int KUKA = 0;

# private Looper looper;

# private ServiceHandler handler;

# /**

# * 由于处理消息是在 Handler 的 handleMessage（）方法中，因此我们需要自己编写类

# * 继承自 Handler 类，然后在 handleMessage（）中编写我们所需要的功能代码

# * @author coolszy

# *

# */

# private final class ServiceHandler extends Handler

# {

# public ServiceHandler (Looper looper)

# {

# super (looper) ;

# }

#

# @Override

# public void handleMessage (Message msg)

# {

# // 根据 what 字段判断是哪个消息

# switch (msg.what)
```

```
    # {
    # case KUKA:
    # //获取 msg 的 obj 字段。我们可在此编写我们所需要的功能代码
    # Log.i（TAG， “The obj field of msg：” + msg.obj）;
    # break;
    # // other cases
    # default:
    # break;
    # }
    # // 如果我们 Service 已完成任务，则停止 Service
    # stopSelf（msg.arg1）;
    # }
    # }
    #
    # @Override
    # public void onCreate（）
    # {
    # Log.i（TAG， “MessageService-->onCreate（）”）;
    # // 默认情况下 Service 是运行在主线程中，而服务一般又十分耗费时间，如果
    # // 放在主线程中，将会影响程序与用户的交互，因此把 Service
    # // 放在一个单独的线程中执行
    # HandlerThread thread = new HandlerThread（“MessageDemoThread”，Process.THREAD_PRIORITY_BACKGROUND）;
```

```
# thread.start（）；
# // 获取当前线程中的 looper 对象
# looper = thread.getLooper（）；
# //创建 Handler 对象，把 looper 传递过来使得 handler、
# //looper 和 messageQueue 三者建立联系
# handler = new ServiceHandler（looper）；
# }
#
# @Override
# public int onStartCommand(Intent intent， int flags， int startId)
# {
# Log.i（TAG，  “MessageService-->onStartCommand（）”）；
#
# //从消息池中获取一个 Message 实例
# Message msg = handler.obtainMessage（）；
# // arg1 保存线程的 ID，在 handleMessage（）方法中
# // 我们可以通过 stopSelf（startId）方法，停止服务
# msg.arg1 = startId;
# // msg 的标志
# msg.what = KUKA;
# // 在这里我创建一个 date 对象，赋值给 obj 字段
# // 在实际中我们可以通过 obj 传递我们需要处理的对象
# Date date = new Date（）；
# msg.obj = date;
```

```
# // 把 msg 添加到 MessageQueue 中
# handler.sendMessage（msg）;
# return START_STICKY;
# }
#
# @Override
# public void onDestroy（）
# {
# Log.i（TAG,  “MessageService-->onDestroy（）”）;
# }
#
# @Override
# public IBinder onBind（Intent intent）
# {
# return null;
# }
# }
```

注：在测试代码中我们使用了 HandlerThread 类，该类是 Thread 的子类，该类运行时将会创建 looper 对象，使用该类省去了我们自己编写 Thread 子类并且创建 Looper 的麻烦。

下面我们通过查看源码，分析下程序的运行过程：

1.onCreate（）

首先启动服务时将会调用 onCreate（）方法，在该方法中我们 new 了一个 HandlerThread 对象，提供了线程的名字和优先级。

紧接着我们调用了 start（）方法，执行该方法将会调用 HandlerThread 对象的 run（）方法：

```
view plaincopy to clipboardprint?

public void run（） {

mTid = Process.myTid（）；

Looper.prepare（）；

synchronized （this） {

mLooper = Looper.myLooper（）；

notifyAll（）；

}

Process.setThreadPriority（mPriority）；

onLooperPrepared（）；

Looper.loop（）；

mTid = -1;

}
```

在 run（）方法中，系统给线程添加的 Looper，同时调用了 Looper 的 loop（）方法：

```
view plaincopy to clipboardprint?

1. public static final void loop（） {

2.

3. Looper me = myLooper（）；

4. MessageQueue queue = me.mQueue;

5. while （true） {
```

```
6. Message msg = queue.next () ; // might block
7. //if (! me.mRun) {
8. // break;
9. //}
10. if (msg ! = null) {
11. if (msg.target == null) {
12. // No target is a magic identifier for the quit message.
13. return;
14. }
15. if (me.mLogging! = null) me.mLogging.println (
16. “>>>>> Dispatching to ” + msg.target + “ ”
17. + msg.callback + “: ” + msg.what
18. ) ;
19. msg.target.dispatchMessage (msg) ;
20. if (me.mLogging! = null) me.mLogging.println (
21. “<<<<< Finished to ” + msg.target + “ ”
22. + msg.callback) ;
23. msg.recycle () ;
24. }
25. }
26. }
```

通过源码我们可以看到 loop（）方法是个死循环，将会不停的从 MessageQueue 对象中获取 Message 对象，如果 MessageQueue 对象中不存在 Message 对象，则结束本次循环，然后继续循环；如果存在 Message 对象，则执行 msg.target.dispatchMessage（msg），但是这个 msg 的.target 字段的值是

什么呢？我们先暂时停止跟踪源码，返回到 onCreate（）方法中。线程执行完 start（）方法后，我们可以获取线程的 Looper 对象，然后 new 一个 ServiceHandler 对象， 我们把 Looper 对象传到 ServiceHandler 构造函数中将使 handler、looper 和 messageQueue 三者建立联系。

2.onStartCommand（）

执行完 onStart（）方法后，将执行 onStartCommand（）方法。首先我们从消息池中获取一个 Message 实例，然后给 Message 对象的 arg1、what、obj 三个字段赋值。紧接着调用 sendMessage（msg）方法，我们跟踪源代码，该方法将会调用 sendMessageDelayed（msg， 0）方法，而 sendMessageDelayed（）方法又会调用 sendMessageAtTime（msg， SystemClock.uptimeMillis（） + delayMillis）方法，在该方法中我们要注意该句代码 msg.target = this，msg 的 target 指向了 this，而 this 就是 ServiceHandler 对象，因此 msg 的 target 字段指向了 ServiceHandler 对象，同时该方法又调用 MessageQueue 的 enqueueMessage（msg， uptimeMillis）方法：

```
view plaincopy to clipboardprint?

# final boolean enqueueMessage（Message msg， long when） {

# if （msg.when ! = 0） {

# throw new AndroidRuntimeException（msg

# + “ This message is already in use.”）；

# }

# if （msg.target == null && ! mQuitAllowed） {

# throw new RuntimeException（“Main thread not allowed to quit”）；

# }

# synchronized （this） {

# if （mQuiting） {

# RuntimeException e = new RuntimeException（

# msg.target + “ sending message to a Handler on a dead thread”）；
```

```
# Log.w (“MessageQueue”, e.getMessage (), e);
# return false;
# } else if (msg.target == null) {
# mQuiting = true;
# }
# msg.when = when;
# //Log.d (“MessageQueue”, “Enqueing: ” + msg);
# Message p = mMessages;
# if (p == null || when == 0 || when < p.when) {
# msg.next = p;
# mMessages = msg;
# this.notify ();
# } else {
# Message prev = null;
# while (p ! = null && p.when <= when) {
# prev = p;
# p = p.next;
# }
# msg.next = prev.next;
# prev.next = msg;
# this.notify ();
# }
# }
# return true;
```

```
# }
```

该方法主要的任务就是把 Message 对象的添加到 MessageQueue 中（数据结构最基础的东西，自己画图理解下）。

handler.sendMessage（）-->handler.sendMessageDelayed（）-->handler.sendMessageAtTime（）-->msg.target = this;queue.enqueueMessage==>把 msg 添加到消息队列中

3.handleMessage（msg）

onStartCommand（）执行完毕后我们的 Service 中的方法就执行完毕了，那么 handleMessage（）是怎么调用的呢？在前 面分析的 loop（）方法中，我们当时不知道 msg 的 target 字段代码什么，通过上面分析现在我们知道它代表 ServiceHandler 对 象，msg.target.dispatchMessage（msg）;则表示执行 ServiceHandler 对象中的 dispatchMessage（）方法：

view plaincopy to clipboardprint?

```
1. public void dispatchMessage（Message msg） {
2. if （msg.callback ! = null） {
3. handleCallback（msg）;
4. } else {
5. if （mCallback ! = null） {
6. if （mCallback.handleMessage（msg））{
7. return;
8. }
9. }
10. handleMessage（msg）;
11. }
12. }
```

该方法首先判断 callback 是否为空，我们跟踪的过程中未见给其赋值，因此 callback 字段为空，所以最终将会执行 handleMessage（）方法，也就是我

们 ServiceHandler 类中复写的方法。在该方法将根据 what 字段的值判断执行哪段代码。

至此，我们看到，一个 Message 经由 Handler 的发送，MessageQueue 的入队，Looper 的抽取，又再一次地回到 Handler 的怀抱中。而绕的这一圈，也正好帮助我们将同步操作变成了异步操作。