Android消息机制（JAVA层）

Last edited by **caoquanli** 1 month ago

Android消息机制（JAVA层）

Table of Contents

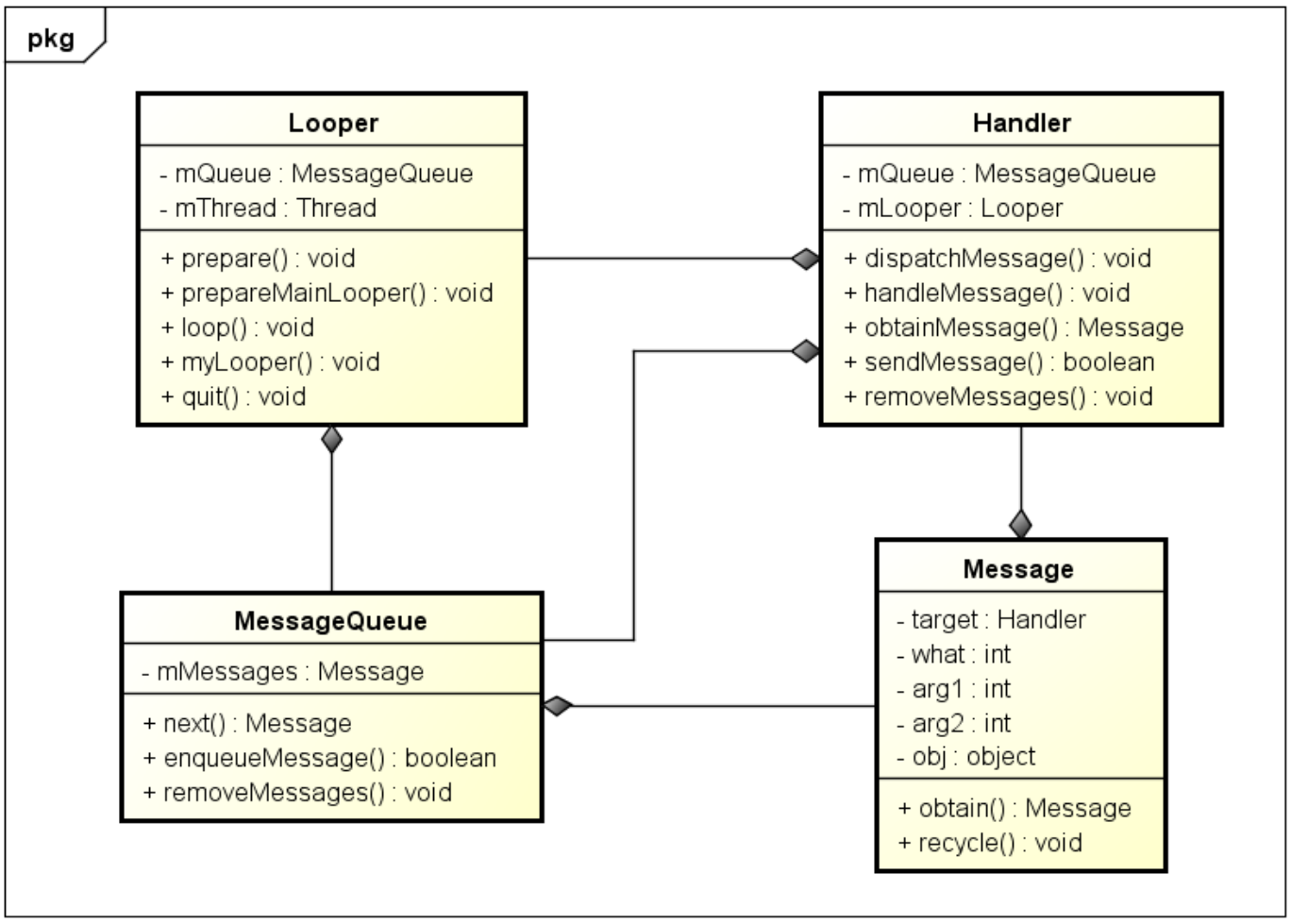
- 一、概述
  - 1.1 相关概念
  - 1.2 模型类图
- 二、工作原理
- 三、Looper
  - 3.1 prepare()
  - 3.2 loop()
  - 3.3 quit()
  - 3.4 mylooper()
- 四、Handler
  - 4.1 post()
  - 4.2 dispatchMessage()
- 五、Message
- 六、使用案例
  - 6.1、在主线程创建内部类
  - 6.2 创建匿名内部类(重写handleMessage)
  - 6.3 创建匿名内部类(重写Callback)
  - 6.4 通过 `Handler.post()` 处理信息

一、概述

1.1 相关概念

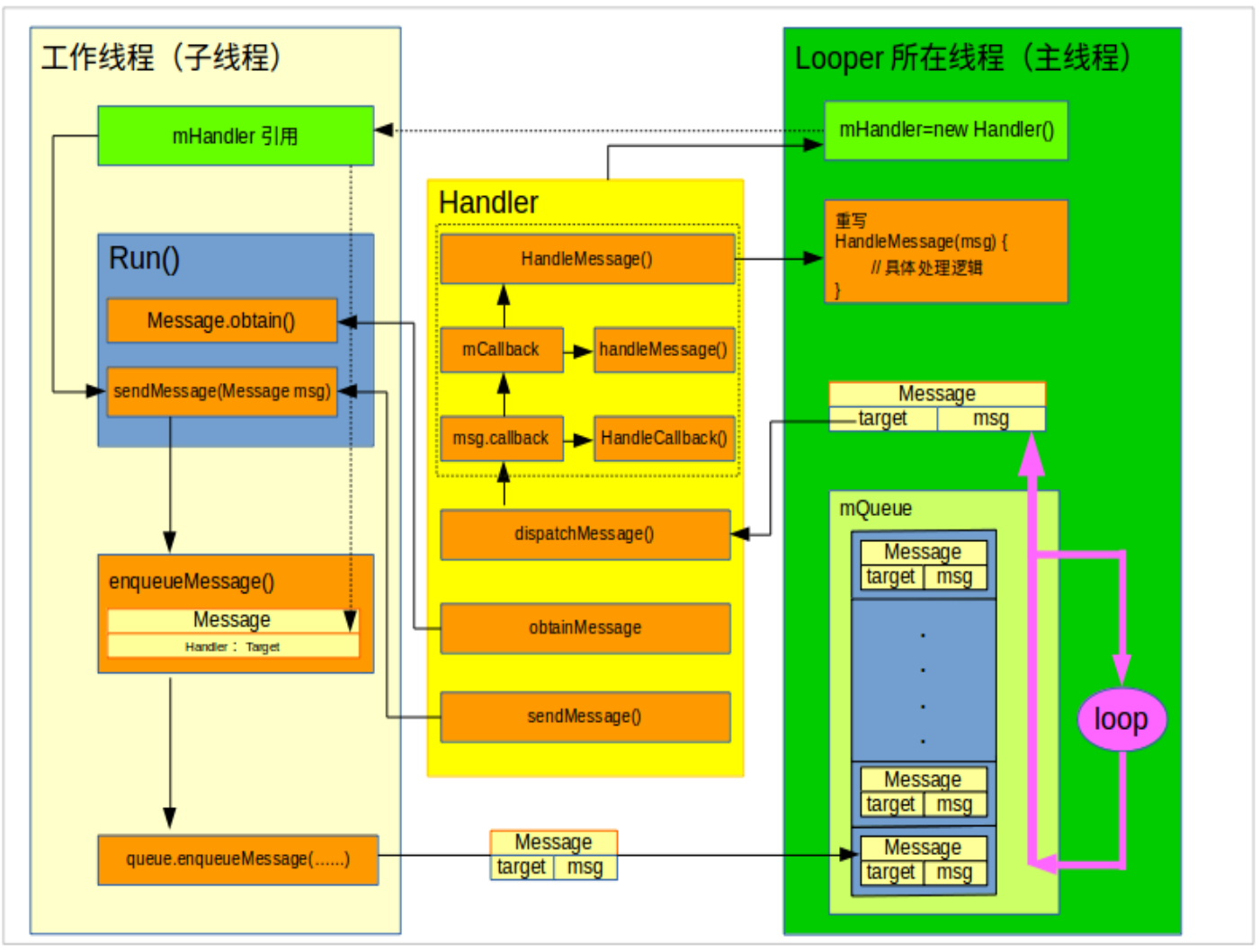
- **Message：** 线程中通信的数据单元（信息载体），存储需要传递的信息，包括硬件产生的信息（按键、触摸）和软件产生的信息。
- **MessageQueue：** 一种数据结构，特点是先进先出，主要功能是向消息池投递消息（enqueueMessage）和从消息池取出消息（next）
- **Handler：** 线程消息的处理者，用于将消息从一个线程发送到另外一个线程和处理从Looper分配过来的消息。
- **Looper：** 消息循环器，消息队列的管理者，按分发机制将消息分发给目标线程的处理者（Handler）

1.2 模型类图



- 一个Looper对象只有一个MessageQueue消息队列
- MessageQueue有一组待处理的Message
- Message中有一个用于处理消息的Handler
- Handler中有一个Looper和MessageQueue

二、工作原理



消息机制的工作流程主要包括四个过程

步骤	具体描述	备注
1. 异步通信准备	在主线程中创建： • 处理器 对象（Looper） • 消息队列 对象（Message Queue） • Handler 对象	• Looper、Message Queue均属于主线程 • 创建Message Queue后，Looper则自动进入消息循环 • 此时，Handler自动绑定了主线程的Looper、Message Queue
2. 消息入队	工作线程 通过 Handler 发送消息（Message）到消息队列（Message Queue）中	该消息内容 = 工作线程对UI的操作
3. 消息循环	• 消息出队：Looper循环取出 消息队列（Message Queue） 中的的消息（Message） • 消息分发：Looper将取出的消息（Message） 发送给 创建该消息的处理者(Handler)	在消息循环过程中，若消息队列为空，则线程阻塞
4. 消息处理	• 处理者(Handler）接收 处理器（Looper）发送过来的消息（Message） • 处理者(Handler）根据消息（Message）进行UI操作	

线程（Thread）、循环器（Looper）、处理者（Handler）之间的对应关系如下：

- 一个Thread只能绑定一个Looper，但可以有多个处理者；
- 一个Looper可以绑定多个Handler
- 一个Handler只能绑定一个Looper

三、Looper

3.1 prepare()

- 创建Looper对象，并将其保存在当前线程的TLS中，代码如下：

```
 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));
}
```

ThreadLocal： 线程本地存储区（Thread Local Storage，简称：TLS），每个线程都有自己私有的本地存储区域，不同的线程之间彼此不能访问对方的TLS区域，TLS常用的操作方法：

- ThreadLocal.set(T value)
- ThreadLocal.get()

Looper.prepare()在每个线程只允许调用一次，该方法会创建一个Looper对象，Looper的构造方法中会创建一个MessageQueue对象，再将Looper对象保存到当前线程TLS中。

与Looper.prepare()具有相似功能的是prepareMainLooper()方法，该方法主要在ActivityThread类中使用，这也解释了在一个Activity中，不需要使用Looper.prepare()就可以创建一个Looper,实际上是UI线程中已经封装了Looper对象。

3.2 loop()

```
public static void loop() {
    final Looper me = myLooper();//获取TLS中存储的Looper对象
        ......
    final MessageQueue queue = me.mQueue;//获取Looper中的消息队列
        //确保在权限检查时基于本进程，而不是基于最初调用进程
    Binder.clearCallingIdentity();
    final long ident = Binder.clearCallingIdentity();

    for (;;) {
        Message msg = queue.next(); // 可能会出现阻塞
        if (msg == null) {
                return;//没有消息队列中没有消息时，则退出循环
            }
                        ......
            msg.target.dispatchMessage(msg);//将消息派发给target
                        ......
        // 确保在分发过程中identity不会损坏
        final long newIdent = Binder.clearCallingIdentity();
        msg.recycleUnchecked();
```

```
        }
    }
```

loop()方法的主要作用是进入循环模式，不断重复下面的操作，直到没有消息时退出循环

- 读取MessageQueue的下一条Message；
- 把Message分发给相应的target(Handler);
- 再把分发后的Message回收到消息池，以便重复使用

3.3 quit()

Looper.quit()主要作用是将消息从消息队列中移除，有两种模式

```
public void quit() {//强制删除
    mQueue.quit(false);
}

public void quitSafely() {//安全删除
    mQueue.quit(true);
}
```

quit()和quitSafely()方法最终都会调用mQueue.quit()方法，如下：

```
void quit(boolean safe) {
    if (!mQuitAllowed) {
        throw new IllegalStateException("Main thread not allowed to quit.");
    }
    synchronized (this) {
        if (mQuitting) {//防止多次退出
            return;
        }
        mQuitting = true;
        if (safe) {
            removeAllFutureMessagesLocked();//移除尚未触发的所有消息
        } else {
            removeAllMessagesLocked();//移除所有消息
        }
        // We can assume mPtr != 0 because mQuitting was previously false.
        nativeWake(mPtr);
    }
}
```

消息退出的方式有两种

- 当safe = true时，只移除尚未触发的所有消息，对于正在触发的消息并不移除；
- 当safe = false时，直接移除所有消息，包括正在触发的消息

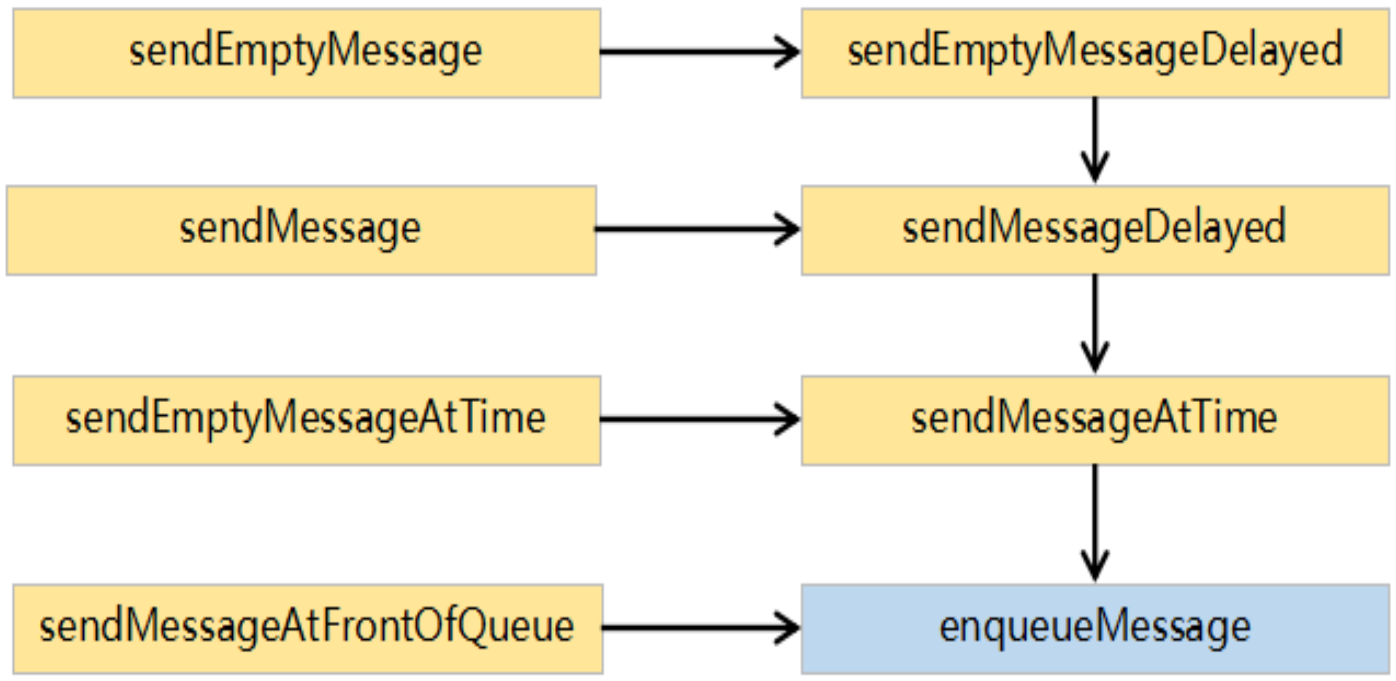
3.4 mylooper()

用于获取存储在TLS中的Looper对象

```
public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

四、Handler

Handler用于消息的发送、处理，其发送消息的调用链如下所示：



4.1 post()

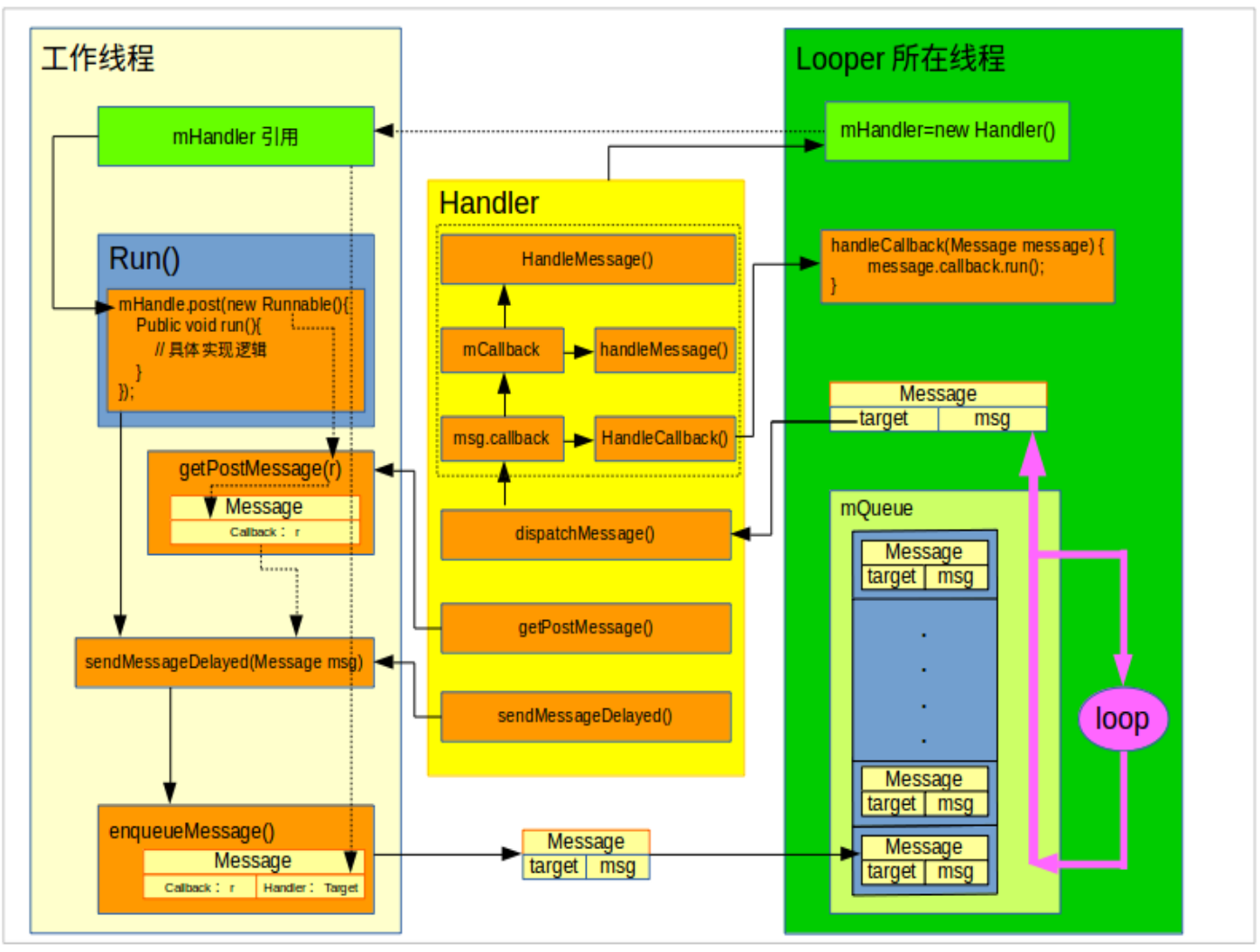
发送消息，并设置消息的callback，用于处理消息

```
 public final boolean post(Runnable r)
{
   return  sendMessageDelayed(getPostMessage(r), 0);
}
```

post()方法调用了sendMessageDelayed()方法，sendMessageDelayed()方法又调用了getPostMessage(r)，代码如下：

```
 private static Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;
    return m;
}
```

post的工作原理如下图所示：



post()方法与HandleMessage()处理消息的区别

post方法是子线程将想让主线程做的事传到主线程，让主线程去执行，主线程中并没有对应的解决方案，只是将子线程写好的方案在主线程中运行，而HandleMessage()是在主线程中写好各种情况对应的解决方法，子线程发送消息去触发，子线程只是选择执行哪个方法，方法的内容由主线程决定。

4.2 dispatchMessage()

dispatchMessage用于消息的派发，代码如下：

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

在dispatchMessage方法中，对于消息的派发具有一定的优先级，首先是判断Message中的Callback接口有没有被赋值，如果赋值，则使用handleCallback对消息进行处理，代码如下，post()方法就是利用该思路进行；

```
private static void handleCallback(Message message) {
    message.callback.run();
}
```

如果Message中的Callback接口没有被赋值，再对Handler中的Callback接口进行判断，若不为空则调用mCallback中的handleMessage方法，Callback代码如下：

```
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

这种方法可以使用匿名内部类的方法避免实现子类问题

```
Handler mHandler = new Handler(new Handler.Callback() {
    @Override
    public boolean handleMessage(Message msg) {
        return false;
    }
});
```

这种方法的好处是发一次消息可以被处理两次，callback.handleMessage()返回false时，会继续调用Handler.handleMessage()

在前两种callback接口都为空的情况下，才最终调用Handler.handleMessage()

五、Message

Message主要包含以下内容：

成员变量	成员变量	说明
what	long	消息触发的时间
arg1	int	参数1
arg2	int	参数2
obj	Object	消息内容
target	Handler	消息处理者
callback	Runnable	回调方法

Table 1. Message成员变量

recycle方法用于把消息加入到消息池中，这样做的好处是，当消息池不为空时，可以直接从消息池中获取Message对象，而不是直接创建，提高效率

创建Message对象的三种方式：

- Message msg = new Message();
- Message msg = Message.obtain();
- Message msg = MyHandler.obtainMessage();

六、使用案例

6.1、在主线程创建内部类

- 创建Handler子类（内部类）

```
class MyHandler extends extends Handler{
   @Override
   public void handleMessage(Message msg){
     //根据不同的线程发过来的消息
     //根据Message对象的what属性标识不同的信息
     switch(msg.what){
       case 1:
           //具体的操作逻辑
           break;
       case 2:
           //具体的操作逻辑
           break;
     }
   }
}
```

- 创建子线程

```
new  Thread(){
       @Override
        public void run(){
            //创建所需的消息对象
            Message msg = Message.obtain();
            msg.what = 1; // 消息标识
            mHandler.sendMessage(msg);
        }
}).start() ;
```

6.2 创建匿名内部类(重写handleMessage)

```
public  Handler mHandler = new Handler(){
   @Override
   public void handleMessage(Message msg){
     //根据不同的线程发过来的消息
     //根据Message对象的what属性标识不同的信息
     switch(msg.what){
       case 1:
           //具体的操作逻辑
           break;
       case 2:
           //具体的操作逻辑
           break;
     }
   }
}
```

6.3 创建匿名内部类(重写Callback)

```
Handler mHandler = new Handler(new Handler.Callback() {
    @Override
    public boolean handleMessage(Message msg) {
        return false;
    }
});
```

6.4 通过 `Handler.post()` 处理信息

```
// 创建线程
new  Thread(){
    @Override
```

```
    public void run(){
        //创建所需的消息对象
        mHandler.post(new Runnable(){
            @Override
            public void run(){
                //指定主线程操作内容
            }
       });
    }
}).start() ;
```