Android消息机制-Handler(上篇)

Dec 26, 2015

> 本文基于Android 6.0的源代码，来分析Java层的消息处理机制

相关源码

```
framework/base/core/java/andorid/os/Handler.java
framework/base/core/java/andorid/os/Looper.java
framework/base/core/java/andorid/os/Message.java
framework/base/core/java/andorid/os/MessageQueue.java
```

一、概述

在整个Android的源码世界里，有两大利剑，其一是Binder IPC机制，，另一个便是消息机制(由Handler/Looper/MessageQueue等构成的)。关于Binder在Binder系列 (http://www.yuanhh.com/2015/10/31/binder-prepare/)中详细讲解过，有兴趣看看。

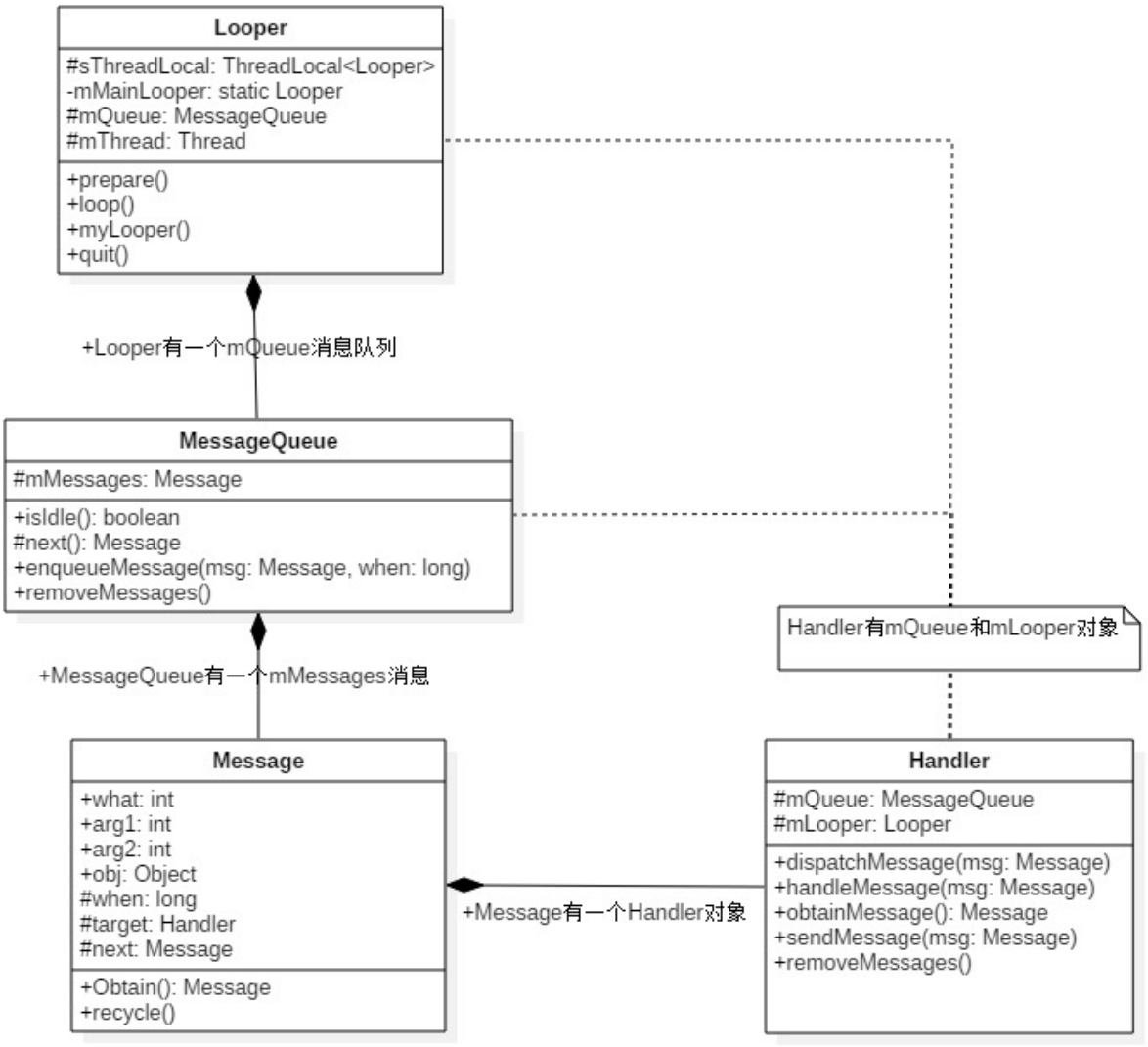
Android有大量的消息驱动方式来进行交互，比如Android的四剑客Activity, Service, Broadcast, ContentProvider的启动过程的交互，都离不开消息机制，Android某种意义上也可以说成是一个以消息驱动的系统。消息机制设计MessageQueue/Message/Looper/Handler。

1.1 模型

消息机制主要包含：

- **消息(Message)**：消息分为硬件产生的消息(如按钮、触摸)和软件生成的消息；
- **消息队列(MessageQueue)**：主要功能向消息池投递消息(`enquueMessage`)和取走消息池的消息(`next`)；
- **消息辅助类(Handler)**：主要功能向消息池发送各种消息事件(`sendMessage`)和处理相应消息事件(`handleMessage`)；
- **消息分发(Looper)**：不断循环执行(`loop`)，用于分发消息至target Handler。

1.2 架构图



- Looper有一个MessageQueue消息队列；
- MessageQueue有一系列的待处理的Message；
- Message中有一个用于处理消息的Handler；
- Handlder中有Looper和MessageQueue。

另外，由于本文是讲述Java层的Handler消息处理机制，其实MessageQueue更多的核心功能都是由native层来完成的，后面再讲述native的情况。想深入研究Native层的Handler机制可查看Android消息机制-Handler(下篇)(http://www.yuanhh.com/2016/01/01/handler-message-3)

1.3 典型实例

先展示一个典型的关于Handler/Looper的线程

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

    public void run() {
        Looper.prepare();   //【见 2.2】

        mHandler = new Handler() {  【见 5.1】
            public void handleMessage(Message msg) {
                //处理即将发送过来的消息    【见 5.3】
            }
        };

        Looper.loop();  //【见 2.3】
    }
}
```

下面会后面围绕着这个线程来展开详细分析

二、Looper

2.1 new Looper()

```
private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);  //创建MessageQueue对象
    mThread = Thread.currentThread();  //保存当前线程到mThread变量
}
```

Looper对象有两个比较重要的成员变量： `mQueue` , `mThread`

- 创建MessageQueue对象，代表消息队列
- 初始化mThread变量，代表当前所在线程；

myLooper()获取TLS存储的Looper对象

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

2.2 Looper.prepare()

对于无参的情况，默认调用prepare(true)

```
private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) { //该方法只允许执行一次，第二执行是TLS中已有数据，则会抛出异常
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));   //Looper对象，保存到sThreadLocal(线程局部变量) 【见 2.1】
}
```

ThreadLocal： TLS(Thread Local Storage），叫线程局部存储，每个线程都有自己的私有的局部存储区域，线程之间彼此不能访问对方的TLS区域。

- `sThreadLocal.get()` 作用是用于获取当前线程的TLS区域的数据；
- `sThreadLocal.set(new Looper(quitAllowed))` 的作用是将创建的Looper对象存储在当前线程的TLS区域

特别注意，每个线程只允许执行一次prepare()方法，该方法主要工作是创建Looper对象，并保存到线程局部变量。

2.3 Looper.loop()

```
public static void loop() {
    final Looper me = myLooper();  //获取TLS存储的Looper对象 【见2.1】
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't
called on this thread.");
    }
    final MessageQueue queue = me.mQueue;  //获取Looper对象中的消息队列

    Binder.clearCallingIdentity();
    final long ident = Binder.clearCallingIdentity(); //确保在权限检查时
基于本地进程，而不是基于最初调用进程。

    for (;;) { //进入Loop的主循环方法
        Message msg = queue.next(); //可能会阻塞 【见4.2】
        if (msg == null) { //没有消息，则退出循环
            return;
        }

        Printer logging = me.mLogging;  //默认为null，可通过setMessageLo
gging()方法来指定输出，用于debug功能
        if (logging != null) {
            logging.println(">>>>> Dispatching to " + msg.target + " "
+
                    msg.callback + ": " + msg.what);
        }
        msg.target.dispatchMessage(msg); //用于分发Message 【见5.3】
        if (logging != null) {
            logging.println("<<<<< Finished to " + msg.target + " " +
msg.callback);
        }

        final long newIdent = Binder.clearCallingIdentity(); //确保分发
过程中identity不会损坏
        if (ident != newIdent) {
            ... //打印identity改变的log
        }
        msg.recycleUnchecked();  //将Message放入消息池 【见3.2.2】
    }
}
```

loop()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
- 再把分发后的Message，回收到消息池

这是这个消息处理的核心部分。另外，上面代码中可以看到有logging方法，这是用于debug的，默认情况下 `logging == null` ，通过设置setMessageLogging()用来开启debug工作。

消息循环，涉及到消息，下一节会讲解Message。

2.4 quit()

```
public void quit() {
    mQueue.quit(false); //消息移除 【见 4.4】
}

public void quitSafely() {
    mQueue.quit(true); //安全地消息移除 【见 4.4】
}
```

2.5 post()

发送消息，并设置变量callback，用于处理消息

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

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

三、 Message

3.1 消息体

每个消息用 Message 表示， Message 主要包含以下内容：

Item	解释
what	消息类别
arg1	参数1
arg2	参数2
obj	消息内容
target	消息响应方
when	触发时间

3.2 消息池

在代码中，可能经常看到recycle()方法，咋一看，可能是在做虚拟机的gc()相关的工作，其实不然，这是用于把消息加入到消息池的作用。这样的好处是，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。

静态变量 sPool 代表消息池；静态变量 MAX_POOL_SIZE 代表消息池的可用大小；消息池的默认大小为50。

3.2.1 obtain()

从消息池中获取消息

```
public static Message obtain() {
    synchronized (sPoolSync) {
        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null; //从sPool中取出一个Message对象，并消息链表断开
            m.flags = 0; // 清除in-use flag
            sPoolSize--; //消息池的可用大小进行减1操作
            return m;
        }
    }
    return new Message(); // 当消息池为空时，直接创建Message对象
}
```

obtain()，从消息池取Message，都是把消息池表头的Message取走，再把表头指向next;

3.2.2 recycle()

把不再使用的消息加入消息池

```
public void recycle() {
    if (isInUse()) { //判断消息是否正在使用中
        if (gCheckRecycle) { //Android 5.0以后的版本默认为true,之前的版本默认为false.
            throw new IllegalStateException("This message cannot be recycled because it is still in use.");
        }
        return;
    }
    recycleUnchecked();
}

//对于不再使用的消息，加入到消息池中
void recycleUnchecked() {
    //将消息标示位置为IN_USE，并清空消息所有的参数。
    flags = FLAG_IN_USE;
    what = 0;
    arg1 = 0;
    arg2 = 0;
    obj = null;
    replyTo = null;
    sendingUid = -1;
    when = 0;
    target = null;
    callback = null;
    data = null;
    synchronized (sPoolSync) {
        if (sPoolSize < MAX_POOL_SIZE) { //当消息池没有满时，将Message对象加入消息池
            next = sPool;
            sPool = this;
            sPoolSize++; //消息池的可用大小进行加1操作
        }
    }
}
```

recycle()，将Message加入到消息池的过程，都是把Message加到链表的表头；

四、MessageQueue

native方法如下：

```
private native static long nativeInit();
private native static void nativeDestroy(long ptr);
private native void nativePollOnce(long ptr, int timeoutMillis); //注意，唯独该方法不是static
private native static void nativeWake(long ptr);
private native static boolean nativeIsPolling(long ptr);
private native static void nativeSetFileDescriptorEvents(long ptr, int fd, int events);
```

这些native方法，会再下一篇文章中，详细说明。

4.1 new MessageQueue()

```
MessageQueue(boolean quitAllowed) {
    mQuitAllowed = quitAllowed;
    mPtr = nativeInit(); //通过native方法，来初始化消息队列
}
```

`mPtr` 是用于native代码

4.2 next()

提取下一条message

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) { //当消息循环已经退出，则直接返回
        return null;
    }
    int pendingIdleHandlerCount = -1; // 循环迭代的首次为-1
    int nextPollTimeoutMillis = 0;
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }
        //阻塞操作，当等待nextPollTimeoutMillis时长，或者消息队列被唤醒，都
会返回。
        nativePollOnce(ptr, nextPollTimeoutMillis);
        synchronized (this) {
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                //查询MessageQueue中的下一条异步消息
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    //设置下一次轮询消息的超时时间
                    nextPollTimeoutMillis = (int) Math.min(msg.when -
now, Integer.MAX_VALUE);
                } else {
                    // 获取一条消息，并返回
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    //设置消息flag成使用状态
                    msg.markInUse();
                    return msg;   //成功地获取MessageQueue中的下一条即将要
执行的消息
                }
            } else {
                //没有消息
                nextPollTimeoutMillis = -1;
            }
```

```
            //消息正在退出，返回null
            if (mQuitting) {
                dispose();
                return null;
            }
            //当消息队列为空，或者消息队列的第一个消息时
            if (pendingIdleHandlerCount < 0 && (mMessages == null || n
ow < mMessages.when)) {
                pendingIdleHandlerCount = mIdleHandlers.size();
            }
            if (pendingIdleHandlerCount <= 0) {
                //没有idle handlers 需要运行，则循环并等待。
                mBlocked = true;
                continue;
            }
            if (mPendingIdleHandlers == null) {
                mPendingIdleHandlers = new IdleHandler[Math.max(pendin
gIdleHandlerCount, 4)];
            }
            mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleH
andlers);
        }
        //只有第一次循环时，会运行idle handlers，执行完成后，重置pendingIdle
HandlerCount为0.
        for (int i = 0; i < pendingIdleHandlerCount; i++) {
            final IdleHandler idler = mPendingIdleHandlers[i];
            mPendingIdleHandlers[i] = null; //去掉handler的引用
            boolean keep = false;
            try {
                keep = idler.queueIdle();  //idle时执行的方法
            } catch (Throwable t) {
                Log.wtf(TAG, "IdleHandler threw exception", t);
            }
            if (!keep) {
                synchronized (this) {
                    mIdleHandlers.remove(idler);
                }
            }
        }
        //重置idle handler个数为0，以保证不会再次重复运行
        pendingIdleHandlerCount = 0;
        //当调用一个空闲handler时，一个新message能够被分发，因此无需等待可以
直接查询pending message.
        nextPollTimeoutMillis = 0;
    }
} nativePollOnce(ptr, nextPollTimeoutMillis)是一个native方法，是一个阻塞
操作。其中nextPollTimeoutMillis代表下一个消息到来前，还需要等待的时长；当next
PollTimeoutMillis = -1时，表示消息队列中无消息，会一直等待下去。空闲后，往往
会执行IdleHandler中的方法。当nativePollOnce()返回后，next()从mMessages中提
```

取一个消息。nativePollOnce()在**native**做了大量的工作，想深入研究可查看 [Android消息机制-Handler(下篇)](http://www.yuanhh.com/2016/01/01/handler-message-3)。

4.3 enqueueMessage

添加一条消息到消息队列

```
boolean enqueueMessage(Message msg, long when) {
    // 每一个Message必须有一个target
    if (msg.target == null) {
        throw new IllegalArgumentException("Message must have a targe
t.");
    }
    if (msg.isInUse()) {
        throw new IllegalStateException(msg + " This message is alread
y in use.");
    }
    synchronized (this) {
        if (mQuitting) {  //正在退出时，回收msg，加入到消息池
            IllegalStateException e = new IllegalStateException(
                    msg.target + " sending message to a Handler on a d
ead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }
        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        if (p == null || when == 0 || when < p.when) {
            //p为null(代表MessageQueue没有消息） 或者msg的触发时间是队列中
最早的， 则进入该该分支
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked; //当阻塞时需要唤醒
        } else {
            //将消息按时间顺序插入到MessageQueue。一般地，不需要唤醒事件队
列，除非
            //消息队头存在barrier，并且同时Message是队列中最早的异步消息。
            needWake = mBlocked && p.target == null && msg.isAsynchron
ous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p;
            prev.next = msg;
```

```
        }
        //消息没有退出，我们认为此时mPtr != 0
        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}
```

MessageQueue一直是按照Message触发的时间先后顺序排列的，队头的消息是将要最早触发的消息。当有消息需要加入消息队列时，会从队列头开始遍历，直到找到消息应该插入的合适位置，以保证所有消息的时间顺序。

4.4 quit

```
void quit(boolean safe) {
        if (!mQuitAllowed) {
            throw new IllegalStateException("Main thread not allowed t
o quit.");
        }
        synchronized (this) {
            if (mQuitting) {
                return;
            }
            mQuitting = true;
            if (safe) {
                removeAllFutureMessagesLocked(); //移除尚未触发的所有消息
            } else {
                removeAllMessagesLocked(); //移除所有的消息
            }
            //mQuitting=false，那么认定为 mPtr != 0
            nativeWake(mPtr);
        }
    }
```

消息退出的方式：

- 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 =flase时，移除所有的消息

4.5 removeMessages

```
void removeMessages(Handler h, int what, Object object) {
    if (h == null) {
        return;
    }
    synchronized (this) {
        Message p = mMessages;
        //从消息队列的头部开始，移除所有符合条件的消息
        while (p != null && p.target == h && p.what == what
               && (object == null || p.obj == object)) {
            Message n = p.next;
            mMessages = n;
            p.recycleUnchecked();
            p = n;
        }
        //移除剩余的符合要求的消息
        while (p != null) {
            Message n = p.next;
            if (n != null) {
                if (n.target == h && n.what == what
                    && (object == null || n.obj == object)) {
                    Message nn = n.next;
                    n.recycleUnchecked();
                    p.next = nn;
                    continue;
                }
            }
            p = n;
        }
    }
}
```

这个移除消息的方法，采用了两个while循环，第一个循环是从队头开始，移除符合条件的消息，第二个循环是从头部移除完连续的满足条件的消息之后，再从队列后面继续查询是否有满足条件的消息需要被移除。

五、Handler

```
final MessageQueue mQueue;
final Looper mLooper;
final Callback mCallback;
final boolean mAsynchronous;
IMessenger mMessenger;
```

5.1 new Handler()

(1) Handler()

```
public Handler() {
    this(null, false);
}

public Handler(Callback callback, boolean async) {
    //匿名类、内部类或本地类都必须申明为static，否则会警告可能出现内存泄露
    if (FIND_POTENTIAL_LEAKS) {
        final Class<? extends Handler> klass = getClass();
        if ((klass.isAnonymousClass() || klass.isMemberClass() || klas
s.isLocalClass()) &&
                (klass.getModifiers() & Modifier.STATIC) == 0) {
            Log.w(TAG, "The following Handler class should be static o
r leaks might occur: " +
                klass.getCanonicalName());
        }
    }
    //必须先执行Looper.prepare()，才能获取Looper对象，否则会抛出异常
    mLooper = Looper.myLooper();  //从当前线程的TLS中获取Looper对象【见代
码 2.1】
    if (mLooper == null) {
        throw new RuntimeException(
            "Can't create handler inside thread that has not called Lo
oper.prepare()");
    }
    mQueue = mLooper.mQueue;
    mCallback = callback;
    mAsynchronous = async;
}
```

当创建Handler时，并没有传递Looper对象时，则直接从当前线程的TLS中尝试获取Looper对象。只要执行的Looper.prepare()方法，那么便可以获取有效的Looper对象。另外还有一个prepareMainLooper()方法，也有雷同功能，这些暂不介绍，后续的文章再详细说明。

(2) Handler(Looper looper)

```
public Handler(Looper looper) {
    this(looper, null, false);
}

public Handler(Looper looper, Callback callback, boolean async) {
    mLooper = looper;
    mQueue = looper.mQueue;  // 将Looper的MessageQueue赋给Handler的Messa
geQueue
    mCallback = callback;
    mAsynchronous = async;
}
```

5.2 obtainMessage

获取消息

```
public final Message obtainMessage()
    {
        return Message.obtain(this); 【见 3.2.1】
    }
```

Handler.obtainMessage() 方法，最终调用 Message.obtainMessage(this) ，其中this为当前的Handler对象。

5.3 dispatchMessage

分发消息

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);  //Message有回调方法
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) { //Handler有Callback对象
来处理
                return;
            }
        }
        handleMessage(msg);  //Handler自身的回调方法
    }
}
```

(1) Message的回调方法

```
//当Message存在回调方法时，直接由Message的CallBack方法来处理，CallBack是一个
Runnable类型
private static void handleCallback(Message message) {
    message.callback.run();
}
```

(2) Handler的mCallback的回调方法

```
//消息回调接口，用于处理消息
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

(3) Handler的回调方法

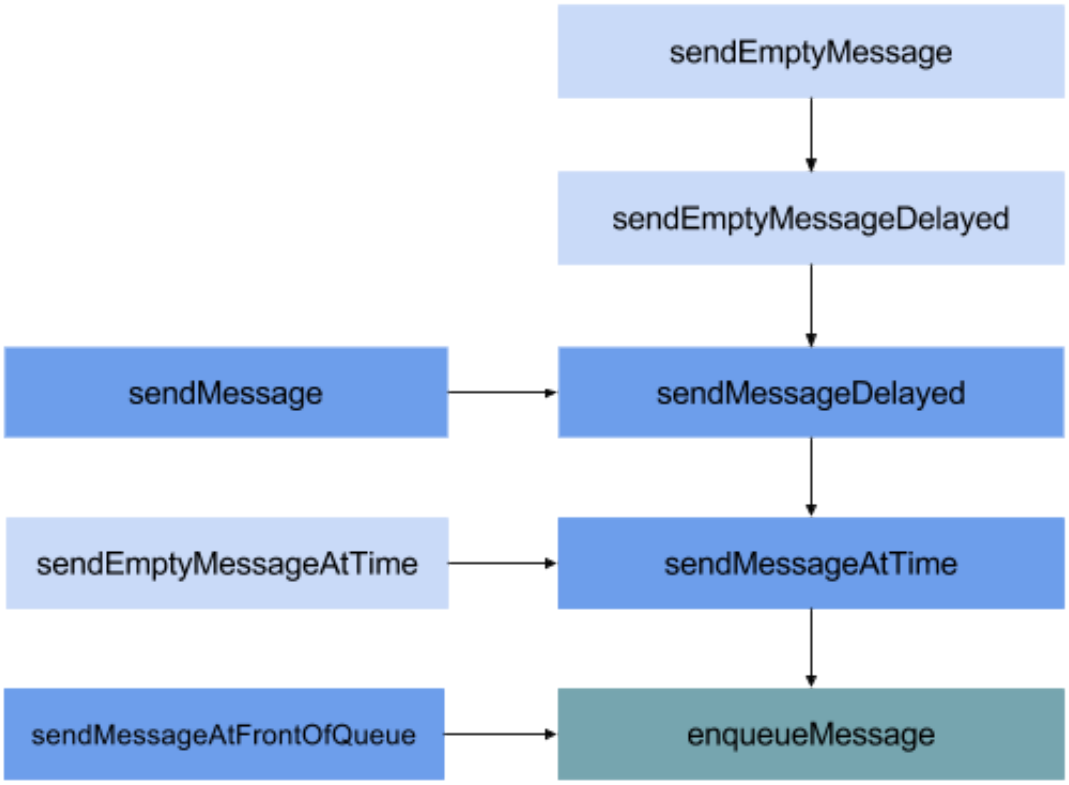
```
// Handler自身的回调方法
public void handleMessage(Message msg) {
        //空方法，子类实现时需要覆写的地方
}
```

消息分发的优先级：

1. 当Message有回调方法，那么由 `message.callback.run()` 来处理消息并返回；否则继续执行；
2. 当Handler设置了mCallback成员变量，那么由 `mCallback.handleMessage(msg)` 来处理消息，处理完的返回值为true则直接返回；否则继续执行；
3. 调用Handler自身的回调方法 `handleMessage(msg)` 来处理.

5.4 sendMessage

发送消息调用链：



从上图，可以发现所有的发消息方式，最终都是调用MessageQueue.enqueueMessage();

(1) sendEmptyMessage

```
public final boolean sendEmptyMessage(int what)
{
    return sendEmptyMessageDelayed(what, 0);
}
```

(2) sendEmptyMessageDelayed

```
public final boolean sendEmptyMessageDelayed(int what, long delayMilli
s) {
    Message msg = Message.obtain();
    msg.what = what;
    return sendMessageDelayed(msg, delayMillis);
}
```

(3) sendMessageDelayed

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

(4) sendMessageAtTime

```
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);
}
```

(5) sendMessageAtFrontOfQueue

```
public final boolean sendMessageAtFrontOfQueue(Message msg) {
        MessageQueue queue = mQueue;
        if (queue == null) {
            RuntimeException e = new RuntimeException(this + " sendMes
sageAtTime() called with no mQueue");
            return false;
        }
        return enqueueMessage(queue, msg, 0);
    }
```

该方法将Message加入到消息队列的队头

(6) enqueueMessage

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long u
ptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    return queue.enqueueMessage(msg, uptimeMillis); 【见4.3】
}
```

Handler.sendEmptyMessage()方法，最终调用MessageQueue.enqueueMessage(msg, uptimeMillis)，其中uptimeMillis为系统当前的运行时间。

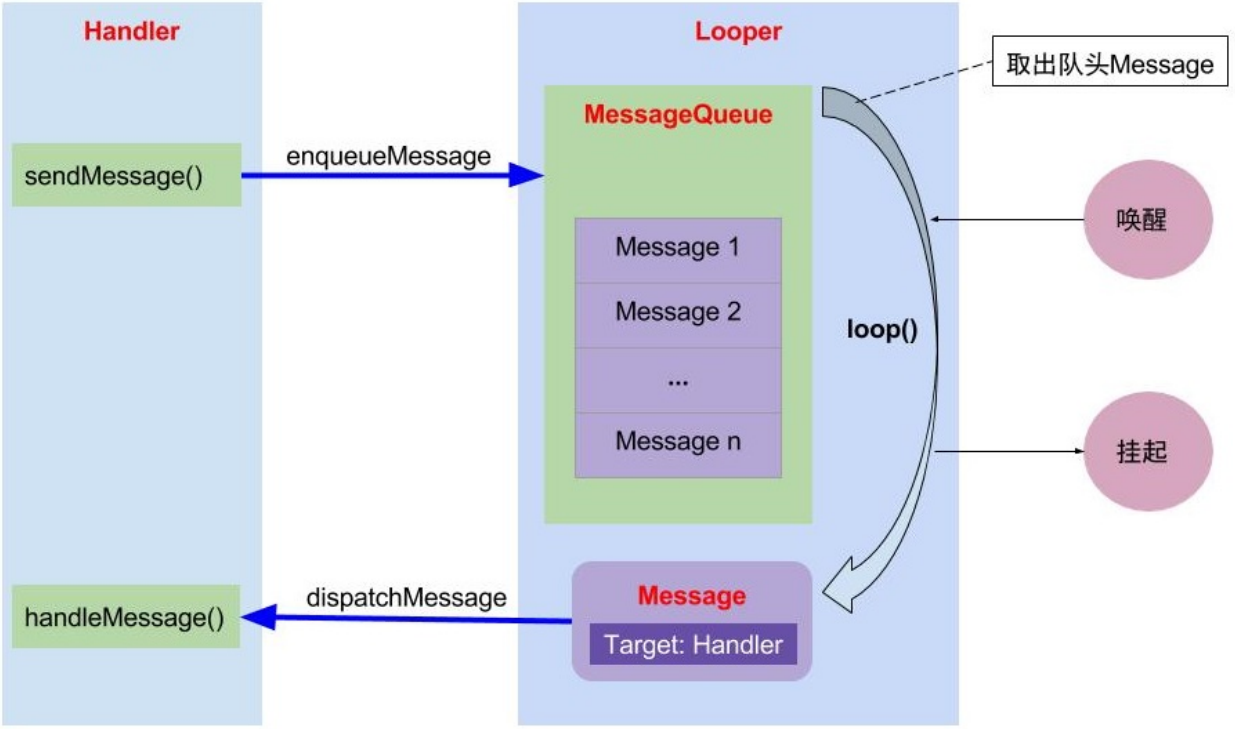
5.5 removeMessages

```
public final void removeMessages(int what) {
    mQueue.removeMessages(this, what, null); // 【见 4.5】
}
```

Handler类似于辅助类，更多的实现都是MessageQueue, Message中的方法。Handler的目的是为了更加方便的使用消息机制。

总结

最后用一张图，来表示整个消息机制



流程说明：

- Handler通过sendMessage()发送Message到MessageQueue队列；

- Looper通过loop()，不断提取出达到触发条件的Message，并将Message交给target来处理；
- 经过dispatchMessage()后，交回给Handler的handleMessage()来进行相应地处理。
- 将Message加入MessageQueue时，处往管道写入字符，可以会唤醒loop线程；如果MessageQueue中没有Message，并处于Idle状态，则会执行IdelHandler接口中的方法，往往用于做一些清理性地工作。

---

gityuan@gmail.com (mailto:gityuan@gmail.com) · Github

(https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll

(https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)