目录

[Android消息机制 1](#_Toc11893527)

[ThreadLocal 的工作原理 2](#_Toc11893528)

[Android消息机制源码分析 2](#_Toc11893529)

[消息阻塞和延时 4](#_Toc11893530)

[唤醒 5](#_Toc11893531)

[延时入队 7](#_Toc11893532)

[Native消息机制 7](#_Toc11893533)

[常见问题分析 8](#_Toc11893534)

[为什么不能在子线程中更新UI,根本原因是什么? 8](#_Toc11893535)

[为什么主线程用Looper死循环不会引发ANR异常? 8](#_Toc11893536)

[为什么Handler构造方法里面的Looper不是直接new？ 8](#_Toc11893537)

[MessageQueue为什么要放在Looper私有构造方法初始化？ 9](#_Toc11893538)

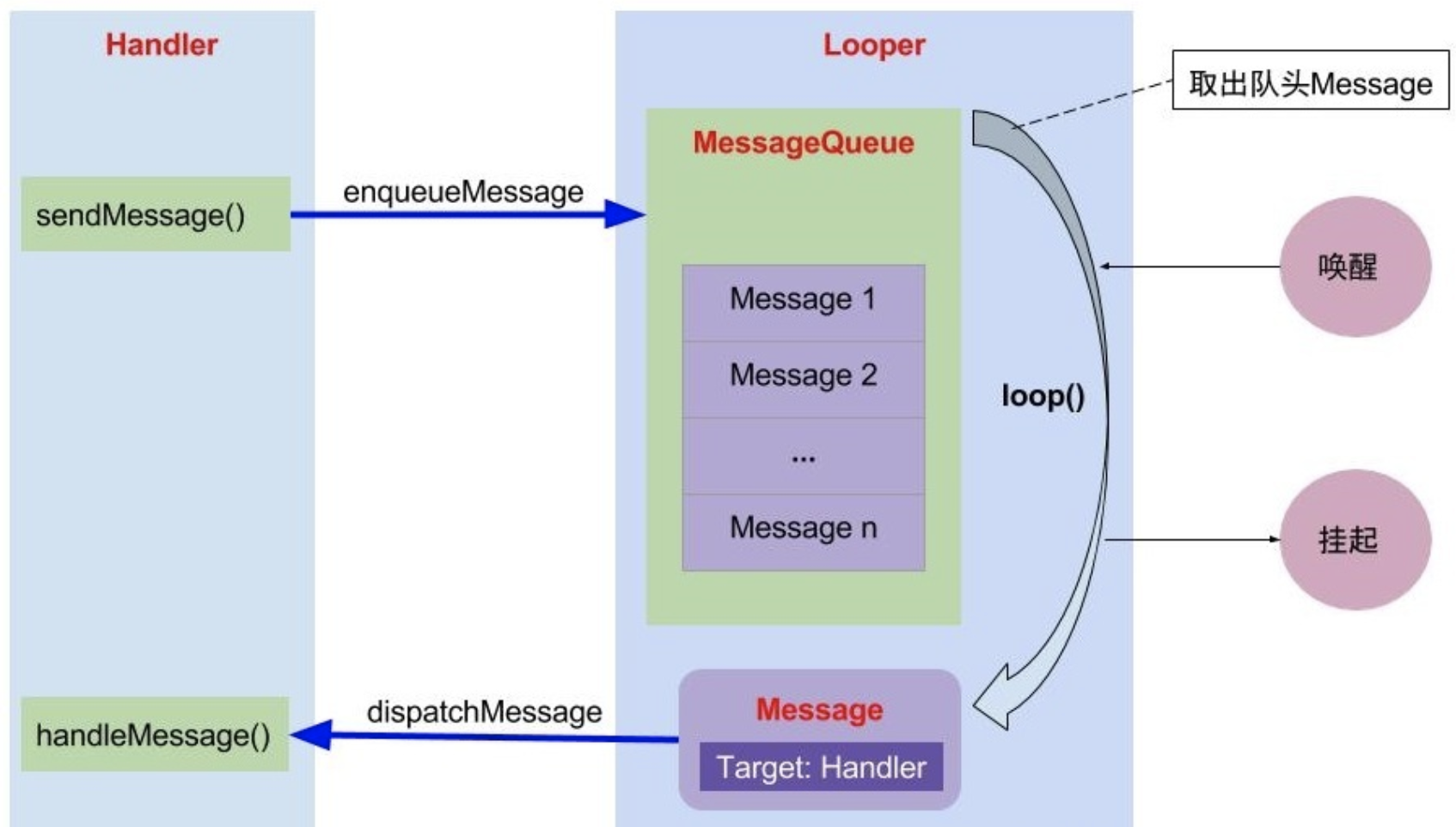
[Handler.post的逻辑在哪个线程执行的，是由Looper所在线程还是Handler所在线程决定的？ 9](#_Toc11893539)

[MessageQueue.next()会因为发现了延迟消息，而进行阻塞。那么为什么后面加入的非延迟消息没有被阻塞呢？ 9](#_Toc11893540)

[Handler的dispatchMessage()分发消息的处理流程？ 9](#_Toc11893541)

# Android消息机制

Android 的消息机制主要是指Handler得运行机制



以上模型的解释：

1.以Handler的sendMessage方法为例，当发送一个消息后，会将此消息加入消息队列MessageQueue中。

2.Looper负责去遍历消息队列并且将队列中的消息分发给对应的Handler进行处理。

3.在Handler的handleMessage方法中处理该消息，这就完成了一个消息的发送和处理过程。

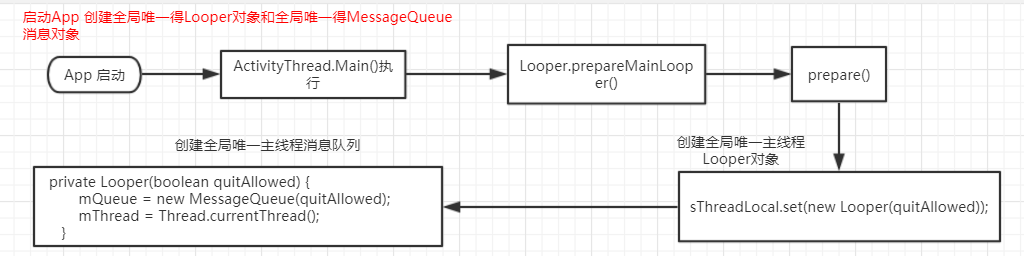
这里从图中可以看到参与消息处理有四个对象，它们分别是 Handler, Message, MessageQueue，Looper。

# ThreadLocal 的工作原理

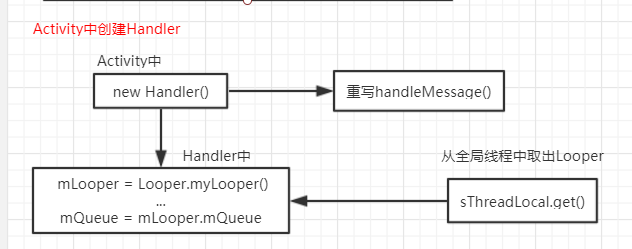
ThreadLocal 是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只有再指定线程中可以获取到存储的数据，对于其他线程来说则无法获取到数据。

# Android消息机制源码分析

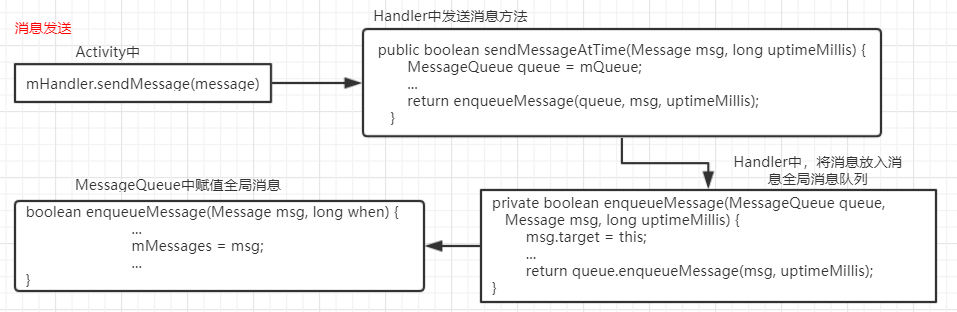
创建全局唯一Looper对象和全局唯一MessageQueue消息对象



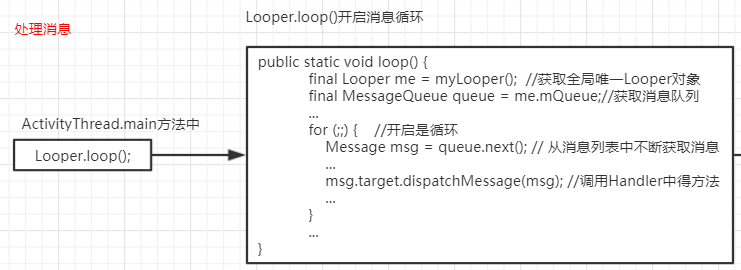
Activity中创建Handler



消息发送



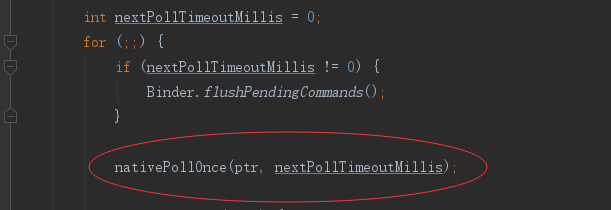
消息处理





# 消息阻塞和延时

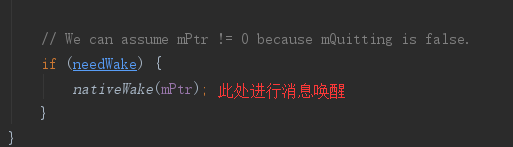
Looper 的阻塞主要是靠 MessageQueue 来实现的，在next()@MessageQuese 进行阻塞，在 enqueueMessage()@MessageQueue 进行唤醒。主要依赖 native 层的 Looper 依靠 epoll 机制进行的。



阻塞和延时，主要是next（）中nativePollOnce(ptr, nextPollTimeoutMillis)调用naive方法操作管道，由nextPollTimeoutMillis决定是否需要阻塞nextPollTimeoutMillis为0的时候表示不阻塞，为-1的时候表示一直阻塞直到被唤醒，其他时间表示延时。

# 唤醒

主要是指enqueueMessage()@MessageQueue 进行唤醒。

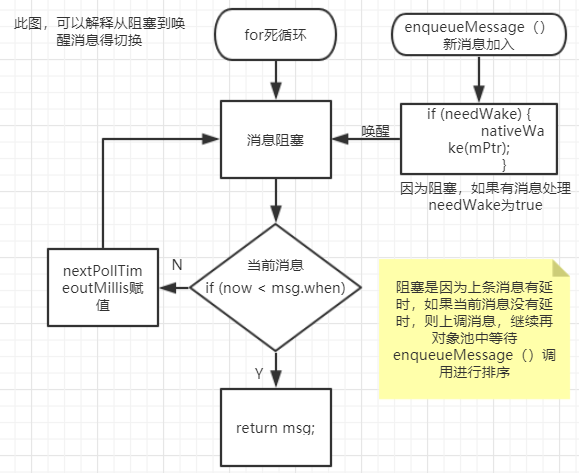


简单理解阻塞和唤醒

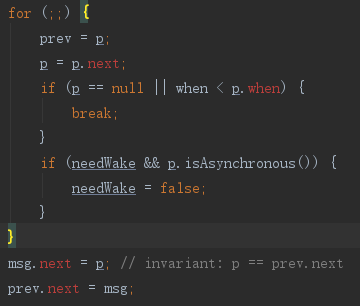
就是在主线程的MessageQueue没有消息时，便阻塞在loop的queue.next()中的nativePollOnce()方法里，此时主线程会释放CPU资源进入休眠状态，直到下个消息到达或者有事务发生，通过往pipe管道写端写入数据来唤醒主线程工作。

这里采用的epoll机制，是一种IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作，本质同步I/O，即读写是阻塞的。 所以说，主线程大多数时候都是处于休眠状态，并不会消耗大量CPU资源。

从阻塞到唤醒，消息切换



# 延时入队



主要指enqueueMessage()消息入列是，上图代码对message对象池得重新排序，遵循规则（when从小到大）。

此处for死循环推出情况分两种

第一种：p==null表示对象池中已经运行到了最后一个，无需再循环。

第二种：碰到下一个消息when小于前一个，立马推出循环（不管对象池中所有message是否遍历完），进行从新排序。

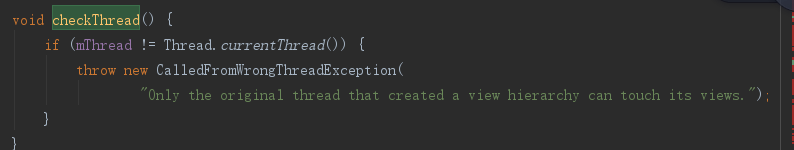
# Native消息机制

参考文档：

<https://blog.csdn.net/chewbee/article/details/78108201#nativewake%E6%96%B9%E6%B3%95>

# 常见问题分析

## 为什么不能在子线程中更新UI,根本原因是什么?



mThread是UI线程，这里会检查当前线程是不是UI线程。那么为什么onCreate里面没有进行这个检查呢。这个问题原因出现在Activity的生命周期中，在onCreate方法中，UI处于创建过程，对用户来说界面还不可视，直到onStart方法后界面可视了，再到onResume方法后界面可以交互。从某种程度来讲，在onCreate方法中不能算是更新UI，只能说是配置UI，或者是设置UI的属性。这个时候不会调用到ViewRootImpl.checkThread()，因为ViewRootImpl没被创建。而在onResume方法后，ViewRootImpl才被创建。这个时候去交互界面才算是更新UI。

setContentView只是建立了View树，并没有进行渲染工作（其实真正的渲染工作是在

onResume之后）。也正是建立了View树，因此我们可以通过findViewById（）来获取到View对象，但是由于并没有进行渲染视图的工作，也就是没有执行ViewRootImpl.performTransversal。同样View中也不会执行onMeasure（），如果在onResume（）方法里直接获取View.getHeight（）/View.getWidth（）得到的结果总是0。

## 为什么主线程用Looper死循环不会引发ANR异常?

简单说就是在主线程的MessageQueue没有消息时，便阻塞在loop的queue.next()中的nativePollOnce()方法里，

此时主线程会释放CPU资源进入休眠状态，直到下个消息到达或者有事务发生，

通过往pipe管道写端写入数据来唤醒主线程工作。这里采用的epoll机制，是一种IO多路复用机制。

## 为什么Handler构造方法里面的Looper不是直接new？

如果在Handler构造方法里面new Looper，怕是无法保证保证Looper唯一，只有用Looper.prepare()才能保证唯一性，具体去看prepare方法。

## MessageQueue为什么要放在Looper私有构造方法初始化？

因为一个线程只绑定一个Looper，所以在Looper构造方法里面初始化就可以保证mQueue也是唯一的Thread对应一个Looper 对应一个 mQueue。

## Handler.post的逻辑在哪个线程执行的，是由Looper所在线程还是Handler所在线程决定的？

由Looper所在线程决定的。逻辑是在Looper.loop()方法中，从MsgQueue中拿出msg，并且执行其逻辑，这是在Looper中执行的，因此有Looper所在线程决定。

## MessageQueue.next()会因为发现了延迟消息，而进行阻塞。那么为什么后面加入的非延迟消息没有被阻塞呢？

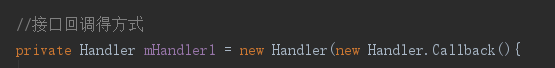
见：唤醒

## Handler的dispatchMessage()分发消息的处理流程？



Msg.callback 在mHandler1.post（）中使用

mCallback在new Handler是通过接口回调



Post()和sendMessage()都是发送消息，加入消息队列得方式也是一样，区别在于处理消息得方式。通过跟踪源码，容易区分。