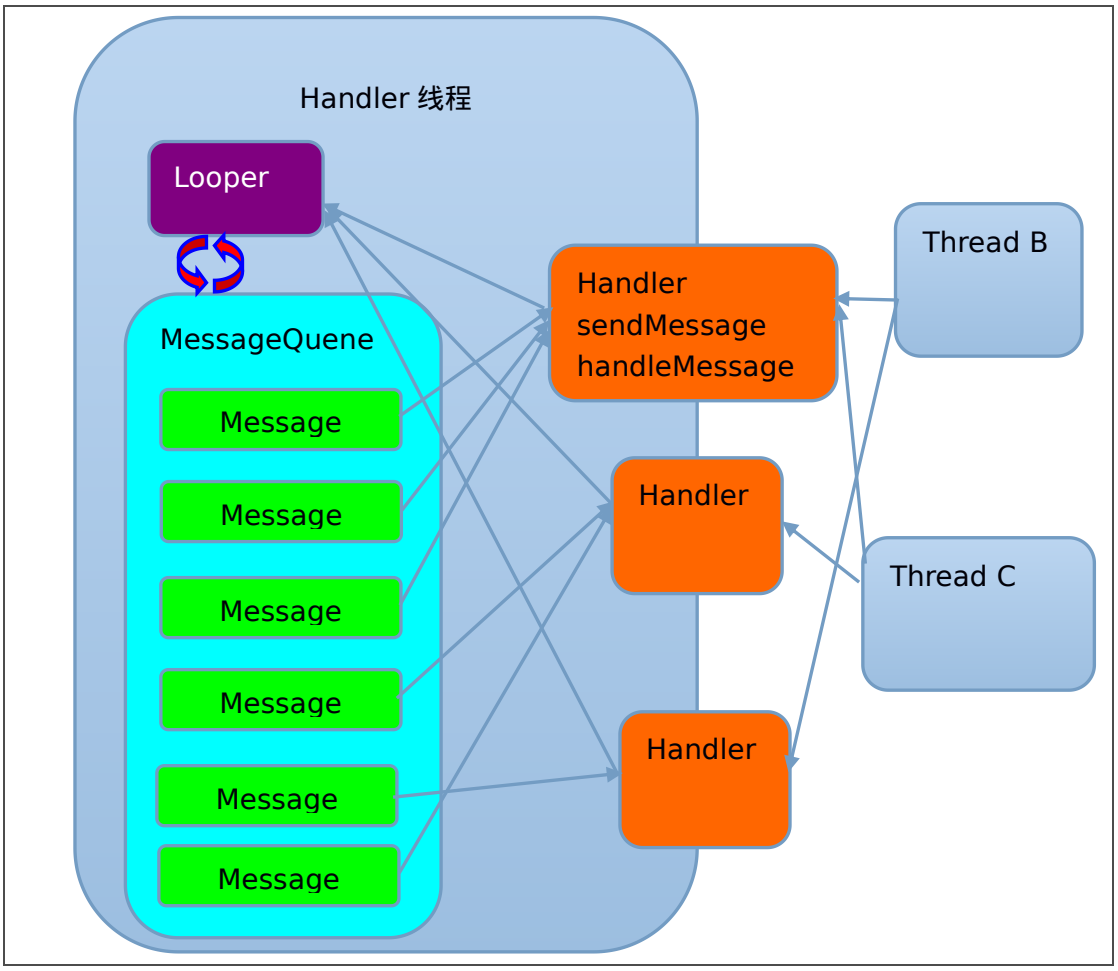
Handler 机制

1 handler 简述

Handler 是 android 应用的子线程之间，最常用的一种消息通信机制。Android 应用的主线程采用的是消息驱动的机制，也是采用了 handler 机制来实现的。以 Android 的主线程为例，下图简单描述描述 Handler 的机制。



Handler 线程中涉及了四个主要类:

Looper：不断的循环遍历整个消息队列，由存在的消息的时候就去处理。

MessageQueue：线程的消息队列，存储其他线程发送过来的未处理的消息。

Message：消息实体

Handler：和 Handler 的线程的 looper 以及 MessageQueue 联系，用来向 MessageQueue 来添加消息，和用来处理 looper 循环出来消息。

2 Handler 的工作流程

在一般的流程中，Handler 生效分为三步：

1，线程 A 建立消息循环机制

2，线程 B 发送消息到线程 A

3，线程 A 响应并处理消息

2.1 线程建立消息循环机制

线程创建消息循环机制，主要是创建 Looper，MessageQueue 和 Handler 的实例。下面是具体的代码示例：

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

    public void run() {
        //创建消息队列
        Looper.prepare();

        //创建 Handler 对象
        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // process incoming messages here
            }
        };
        //循环读取消息
        Looper.loop();
    }
}
```

主要分为两步，创建消息队列和开始循环读取消息。

1．创建消息队列，Looper.prepare()是用来创建线程的消息循环队列

```
    private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be created per thread");
        }
        //创建 looper 对象并，设置为线程唯一
        sThreadLocal.set(new Looper(quitAllowed));
    }
```

每一个线程只允许拥有一个消息循环队列，所以对应只有一个 looper 的对象，就是要线程唯一。进程唯一，可以使用静态变量和单例模式来实现。此时这是一个 java 封装的线程，会拥有唯一的 java 的 Thread 类，把 looper 对象设置为线程的唯一变量。可以解决这个问题，当下次再次创建的时候，将 looper 实例再次 attach 到 Thread 对象的时候，发现已经存在了，此时就会报错，相关的逻辑可以参考 Thread 类的代码。

```
    private Looper(boolean quitAllowed) {
        //创建消息队列
        mQueue = new MessageQueue(quitAllowed);
        mRun = true;
        mThread = Thread.currentThread();
    }
```

创建的消息队列是一个由 Message 组成的一个链表

```
public final class MessageQueue {
    //mMessages 指向链表的第一个消息，若未 null，表示消息队列为空
    Message mMessages;
    MessageQueue(boolean quitAllowed) {
        mQuitAllowed = quitAllowed;
        //创建 C++层的消息队列和 looper
        mPtr = nativeInit();
    }
}
```

创建完成 java 层的 Messagequeue 以后，要创建 JNI 以及 C++层的 MessageQueue 和 Looper 对象。这里先不做过多讲解。

2．开始循环读取消息，Looper.loop 来循环读取消息

```
public static void loop() {
    final Looper me = myLooper();
    //获取线程唯一的消息队列
    final MessageQueue queue = me.mQueue;

    //获取消息的死循环，有消息就处理，没有消息就 block
    for (;;) {
        //获取下一个 Message，如果当前没有消息，此函数会被阻塞，不会返回
        Message msg = queue.next(); // might block
        if (msg == null) {
            // No message indicates that the message queue is quitting.
            return;
        }
        //处理消息
        msg.target.dispatchMessage(msg);
    }
}
```

在创建了新的消息队列之后，此时消息队列中，没有任何的消息，queue.next()方法会被 block，不会返回。此时主线程也阻塞。

2.2 线程 B 发送消息到 Handler 线程

线程 B 要发送消息到 Handler 线程，首先要获取 Handler 的对象的引用，而 Handler 对象要实现发送消息到线程唯一的消息队列，就要获取对应的引用。线程 B 发送消息，需要两步：

- 获取 handler 对象
- 发送消息到 MessageQueue 队列

1．获取 handler 对象

```
//无参构造函数
public Handler() {
    this(null, false);
}

public Handler(Callback callback, boolean async) {
```

```
        //获取当前执行线程的 Looper 对象
        mLooper = Looper.myLooper();
        if (mLooper == null) {
            throw new RuntimeException(
                "Can't create handler inside thread that has not called Looper.prepare()");
        }
        //获取当前线程的消息队列病赋值给 mQueue
        mQueue = mLooper.mQueue;
        mCallback = callback;
        mAsynchronous = async;
}
```

注意在构造 Handler 对象的时候，要获取线程的 Looper 对象。Looper.myLooper 方法只会返回当前执行线程的 looper 对象。所以由此可见所有通过无参构造函数创建的 Handler 对象必须在有消息队列的线程中创建。

2．Handler 发送消息

线程 B 调用 Handler 线程中创建的 Handler 对象的 sendMessage 方法来发送消息。

```
    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;
        }
        //把现有的消息加入到队列 mQueue 中
        return enqueueMessage(queue, msg, uptimeMillis);
    }
```

2.3 Handler 线程处理消息

如在创建线程中看到的，一旦消息队列中存在消息，looper.loop 中 for 循环中 MessageQueue.next 就会返回队列中已有的消息。此时线程也不会阻塞，然后调用 Message.target.dispatchMessage(msg);方法来处理消息。

关于 Message 的 target 属性，可以看一下 Message 类

```
public final class Message implements Parcelable {
    //target 是 Handler 类的引用
    Handler target;

    public static Message obtain(Handler h) {
        Message m = obtain();
        m.target = h;

        return m;
    }

    public static Message obtain(Handler h, Runnable callback) {
        Message m = obtain();
        m.target = h;
        m.callback = callback;

        return m;
    }
}
```

从上面的来看，target 属性就是 Handler 对象，一般就是发送这条 Message 的 Handler 对象。所以消息的处理会调用 Handler 的 DispatchMessage 方法来处理。

```
    public void dispatchMessage(Message msg) {
        //如果 Handler 不是通过 sendMessage 来发送消息，而是通过 post 一个 Runnable 来发送消息，此时就会进入下面的分支
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            //调用自己实现的 HandleMessage 方法来处理 Message
            handleMessage(msg);
        }
    }
```

2.4 小结

Handler 在 java 层的消息机制，大致如下，由此可见：

- 所有的拥有消息队列的线程都拥有唯一的 looper 对象，和唯一的 Messagequeue 对象。
- 拥有消息队列的线程，在有消息的时候，处理消息，没有消息的时候就无限 block
- Android 应用的主线程也是采用这种机制，所以所有的主线程操作都是消息驱动的。包括 UI 消息和 FW 发送过来的消息。
- Handler 对象的创建必须是在已经拥有 Looper 和 MessageQueue 的线程中。并且一般情况下，必须在消息开始循环之前就创建第一个 Handler 对象。

3 Handler 的工作原理

在 Handler 消息队列的创建中，我们提到了在创建过程中还会创建 C++层的消息队列，以及进程会阻塞在 MessageQueue.next 方法。如果有消息返回了，线程会去处理消息。但是线程 如何阻塞以及线程如何唤醒都没有详细介绍。主要的逻辑依赖 C++层，以及 linux 的 epoll 机制来实现。

创建 C++层的 Looper 和 MessageQueue。在创建 java 层的 MessageQueue 的构造函数里，会调用 nativeInit 这个 native 方法来调用 JNI 层的方法。

3.1 C++层创建 native 的 MessageQueue 和 epoll 实例

```
static jint android_os_MessageQueue_nativeInit(JNIEnv* env, jclass clazz) {
    //创建 NativeMessageQueue 的对象
    NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
```

```
    if (!nativeMessageQueue) {
        jniThrowRuntimeException(env, "Unable to allocate native queue");
        return 0;
    }

    nativeMessageQueue->incStrong(env);
     //这个方法返回的引用会作为 Java 对象中在 MessageQueue 的属性
    return reinterpret_cast<jint>(nativeMessageQueue);
}
```

C++层创建了 Looper 对象

```
    NativeMessageQueue::NativeMessageQueue() : mInCallback(false),
mExceptionObj(NULL) {
    //生成 Looper 对象
    mLooper = Looper::getForThread();
    if (mLooper == NULL) {
        mLooper = new Looper(false);
        //设置为线程局部变量
        Looper::setForThread(mLooper);
    }
}
```

在 Loopper 的构造函数里，会创建 Epoll 的实例，并注册。

```
Looper::Looper(bool allowNonCallbacks) :
        mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),
        mResponseIndex(0), mNextMessageUptime(LLONG_MAX) {
    //创建管道
    int wakeFds[2];
    int result = pipe(wakeFds);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not create wake pipe.  errno=%d", errno);

    //分别设置管道的读端和写端
    mWakeReadPipeFd = wakeFds[0];
    mWakeWritePipeFd = wakeFds[1];

    //设置管道两端的文件描述的读写权限
    result = fcntl(mWakeReadPipeFd, F_SETFL, O_NONBLOCK);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not make wake read pipe non-blocking.
errno=%d",
            errno);

    result = fcntl(mWakeWritePipeFd, F_SETFL, O_NONBLOCK);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not make wake write pipe non-blocking.
errno=%d",
```

```
                errno);

    mIdling = false;

    // 向 Linux 系统申请创建 Epoll 的实例，epoll_create 返回的文件描述附指向一个 epoll 实例
    mEpollFd = epoll_create(EPOLL_SIZE_HINT);
    LOG_ALWAYS_FATAL_IF(mEpollFd < 0, "Could not create epoll instance.  errno=%d",
errno);

    struct epoll_event eventItem;
    memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data field
union
    eventItem.events = EPOLLIN;
    eventItem.data.fd = mWakeReadPipeFd;
    //注册 epoll 感兴趣的事件，这里感兴趣的是管道读端的 EPOLLIN 事件。
    result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not add wake read pipe to epoll instance.
errno=%d",
                errno);
  }
```

3.2 Handler 线程读取消息，无消息并阻塞主线程

在之前的 java 层的 Looper.loop 循环会调用 MessageQueue 的 next 方法，如果此时 MessageQueue 中没有消息，这个函数会阻塞，具体看一下，线程是如何阻塞的。

```
Message next() {
        int pendingIdleHandlerCount = -1; // -1 only during first iteration
        int nextPollTimeoutMillis = 0;
        for (;;) {

            //去 C++层去读取消息，nextPollTimeoutMillis 为 0 的时候读取不到消息会立即返
回，nextPollTimeoutMillis 为-1 的时候，读取不到消息，一直不会返回，并会阻塞。
            nativePollOnce(mPtr, nextPollTimeoutMillis);

        }
}
```

在 nativePollOnce 会调用到 C++层的 looper 中，来查询 C++层的 Looper 会查询 Epoll 监听的文件是否有可读的消息。

```
static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jclass clazz,
        jint ptr, jint timeoutMillis) {
    NativeMessageQueue* nativeMessageQueue =
reinterpret_cast<NativeMessageQueue*>(ptr);
    //调用 native 层的 MessageQueue.pollOnce
    nativeMessageQueue->pollOnce(env, timeoutMillis);
    }

void NativeMessageQueue::pollOnce(JNIEnv* env, int timeoutMillis) {
    mInCallback = true;
    //调用 C++中 Looper 的 pollOnce 方法
    mLooper->pollOnce(timeoutMillis);
    mInCallback = false;
    }
```

在 C++层的 Looper 对象中，会来读取 epoll 实例监听的文件是否有可读的消息。

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
    int result = 0;
    for (;;) {
//调用 pollInner 方法
int Looper::pollInner(int timeoutMillis) {
    }
}

int Looper::pollInner(int timeoutMillis) {


    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    //Epoll 会在这个函数中，尝试读取 epoll 实例中感兴趣的事情是否发生，如果发生了就返回。
如果没有发生并且此时 timeoutMillis 为-1，epollWait 就不会返回，导致线程阻塞。
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);


}
```

上述可以清楚，主线程是如何利用 epoll 机制来阻塞的。所以 Android 应用

的主线程如果没有消息来处理的话，所有的堆栈信息都会显示正在 epoll_wait 方法中。

3.3 唤醒线程并处理消息

在 java 层利用 Handler 的 sendMessage 方法会调用 enqueueMessage(queue, msg, uptimeMillis);来把消息加入队列中，但是是如何实现唤醒了线程的？如何使得线程能从阻塞的 epoll_wait 方法中返回的？

```
    boolean enqueueMessage(Message msg, long when) {

        synchronized (this) {

            msg.when = when;
            Message p = mMessages;
            boolean needWake;
            if (p == null || when == 0 || when < p.when) {
                // New head, wake up the event queue if blocked.
                //把消息加入到队列中
                msg.next = p;

                mMessages = msg;
                //needWake 为 true，当线程中无消息的时候，线程阻塞在 epoll_wait 方法中，所以此时 mBlocked=true
                needWake = mBlocked;
            }                   // We can assume mPtr != 0 because mQuitting is false.
            if (needWake) {
                //native 层来唤醒线程
                nativeWake(mPtr);
            }
        }
        return true;
    }
```

所以在 java 层不仅要把消息加入队列，还要去唤醒主线程

```
static void android_os_MessageQueue_nativeWake(JNIEnv* env, jclass clazz, jint ptr) {
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
```

```
    return nativeMessageQueue->wake();
}

void NativeMessageQueue::wake() {
    mLooper->wake();
}
```

调用 C++层线程的 looper 对象的 wake 方法来唤醒线程。

```
void Looper::wake() {

    do {
        //向管道的写端写入字符
        nWrite = write(mWakeWritePipeFd, "W", 1);
    } while (nWrite == -1 && errno == EINTR);

}
```

在 wake 方法中，只是在管道的写端（mWakeWritePipeFd）写入了一个字符,而此时 mWakeReadPipeFd 管道的读端就可以读取到这个字符，当系统检测到 mWakeReadPipeFd 有可读的字符的时候,之前在阻塞的 epoll_wait 就会返回，表示感兴趣的事件已经发生。然后一直返回到 java 层的 MessageQueue 的 next 方法里，由于在 snedMessage 方法里面已经把要发送的消息加入到了队列中，所以此时 next 方法会读取这个消息，并调用发送它的 Handler 来处理消息。

3.4 小结

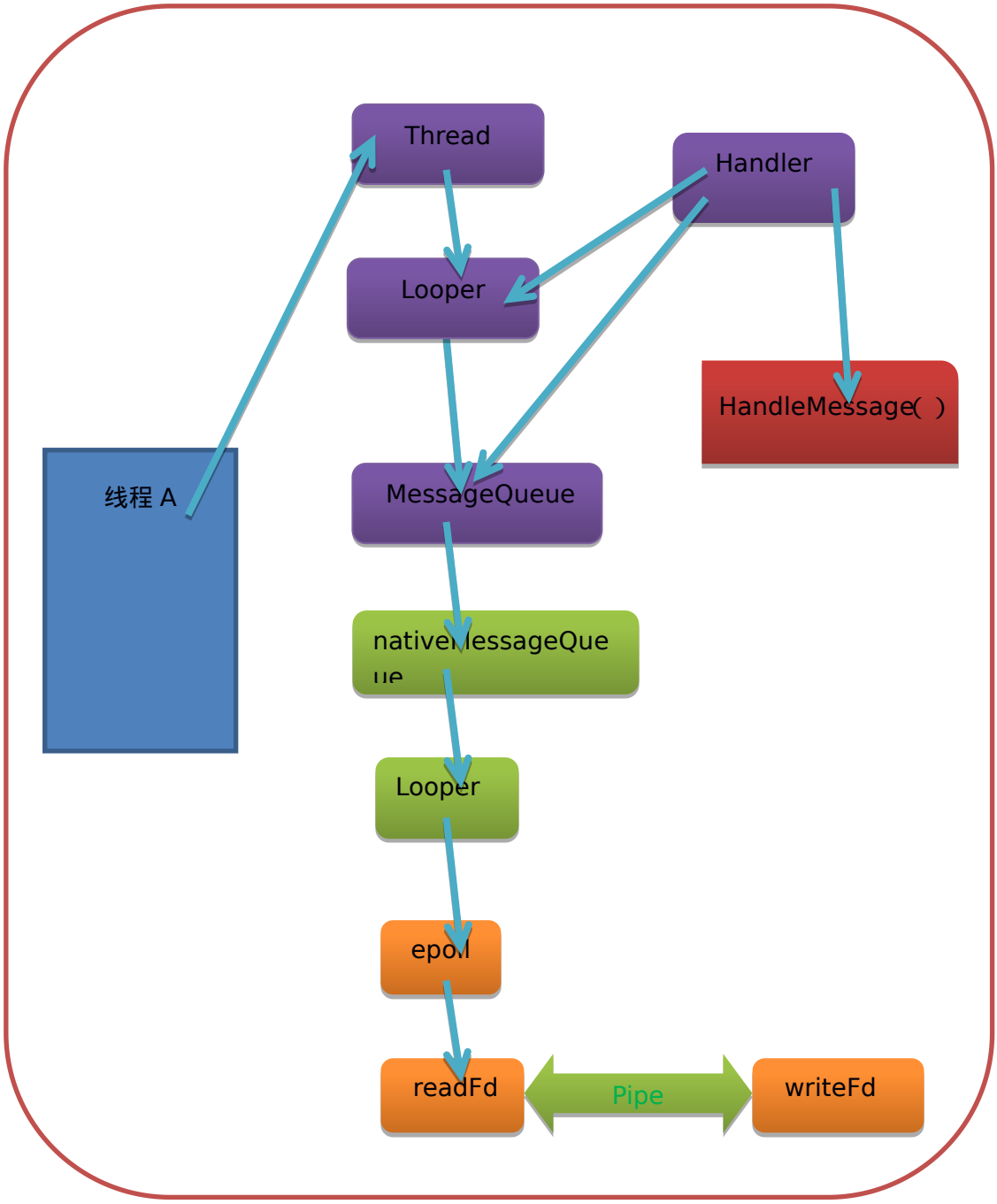
所以主线程的阻塞和唤醒是通过 Epoll 机制来实现的。

- Epoll 会监听读端读端是否有可读内容。
- 当无消息的时候主线程阻塞到 epoll_wait 函数中。
- 如果线程 B 发送消息，不仅要把消息加入到 java 的消息队列之中还会通过 native 层的 looper 对象向管道的写端写入一个字符，此时

epoll_wait 监听的管道读端有字符可读，线程就会唤醒。并处理消息

4 Handler 机制的应用

Handler 机制应用在 Android 应用中的主线程，所以所有的应用进程都是消息驱动的。下图展示 handler 的总体框架



4.1 Handler 线程的两种状态

由于 Handler 线程是消息驱动的，要不在处理消息，要不就是在等待消息阻塞中。所以主线程的调用栈一般只有下面两种类型：

主线程无消息来处理，主线程阻塞在 epoll_wait 中

```
"main" prio=5 tid=1 NATIVE
  | group="main" sCount=1 dsCount=0 obj=0x417dbe58 self=0x417ca8c0
  | sysTid=1136 nice=-2 sched=0/0 cgrp=apps handle=1074004308
  | state=S schedstat=( 27460518191 4308013842 21950 ) utm=2522 stm=224 core=3

  //线程无消息，在 epoll_wait 中

  #00  pc 00021940  /system/lib/libc.so (epoll_wait+12)
  #01  pc 000104ef  /system/lib/libutils.so (android::Looper::pollInner(int)+98)
t+50)
  #22  pc 00000d7c  /system/bin/app_process
  at android.os.MessageQueue.nativePollOnce(Native Method)
  at android.os.MessageQueue.next(MessageQueue.java:138)
  at android.os.Looper.loop(Looper.java:123)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:621)
  at dalvik.system.NativeStart.main(Native Method)
```

主线程正在处理消息，调用栈处于 handleMessage 中

```
DALVIK THREADS (53):
"main" prio=5 tid=1 Native
 | group="main" sCount=1 dsCount=0 obj=0x73b4fb88 self=0xb7524bd0
 | sysTid=1058 nice=-8 cgrp=default sched=0/0 handle=0xb6f09bec
 | state=S schedstat=( 124260927329 144508009965 327805 ) utm=9829 stm=2597 core=0 HZ=100
 | stack=0xbe1b5000-0xbe1b7000 stackSize=8MB
 | held mutexes=

  //栈定不是正在处于 epollwait

 native: #00 pc 0000fa00  /system/lib/libc.so (syscall+28)
com.huawei.bone.HomeActivity$2.onSensorChanged(HomeActivity.java:275)

  //正在处理消息
```

```
      at
android.hardware.SystemSensorManager$SensorEventQueue.dispatchSensorE
vent(SystemSensorManager.java:405)
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:143)
  at android.os.Looper.loop(Looper.java:122)
  at android.app.ActivityThread.main(ActivityThread.java:5254)
  at java.lang.reflect.Method.invoke!(Native method)
  at java.lang.reflect.Method.invoke(Method.java:372)
  at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
962)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:757)
```

4.2 应用的 ANR

ANR 的全称是 Application Not response。一般情况下，Android 中是由 FW 的 AMS 和 WMS 来监控的，如果监控到 AMS 和 WMS 发送的消息一直没有得到迅速处理，就会扔出 AND 的 dialog，直接影响 ANR 的原因有两种：

- 对于输入事件的处理超过 5s
- BroadcastReceiver 处理的时间超时 10s

ANR 的实质就是应用的主线程超时，影响主线程超时的原因才是最终的原因。

4.3 其他问题

Handler 的初始化顺序是否可以改变

Handler 设计中的若干问题

屏幕正在显示你的 activity 是否意味着你的 app 正在运行

WMS 和 AMS 要对你的应用发消息怎么实现的

Binder 相对与 handler 的机制的异同

5 涉及的文件

LINUX/android/frameworks/base/core/java/android/os/Handler.java
LINUX/android/frameworks/base/core/java/android/os/Looper.java
LINUX/android/frameworks/base/core/java/android/os/Message.java
LINUX/android/frameworks/base/core/java/android/os/MessageQueue.java
LINUX/android/frameworks/base/core/jni/android_os_MessageQueue.cpp
LINUX/android/system/core/libutils/Looper.cpp