**同步屏障？阻塞唤醒？和我一起重读 Handler 源码**

[](https://juejin.cn/user/3790771822270920)

你好，我是 N0tExpectErr0r，一名热爱技术的 Android 开发

我的个人博客：[blog.N0tExpectErr0r.cn](https://link.juejin.cn/?target=https%3A%2F%2Fblog.N0tExpectErr0r.cn)

大家应该都知道，Android 的消息机制是基于 Handler 实现的。还记得一年前的自己就看了几篇博客，知道了 Handler、Looper、MessageQueue 就自以为自己知道了 Handler 的原理。但其实看源码的过程中慢慢就会发现，Handler 的内容可不止这点， 像同步屏障以及 Handler 在 Native 层的实现这些知识以前就没有理解清楚。因此写下此篇文章，从头开始重塑对 Handler 的印象 。

Handler 采用的是一种**生产者-消费者模型**，Handler 的用户就是生产者，通过 Handler 可以生产需要执行的任务。而 Looper 则是消费者，不断从 MessageQueue 中取出 Message 对这些消息进行消费并交给对应的 Handler，下面我们看一下其具体的实现。

**发送消息**

**post & sendMessage**

首先我们都知道，Handler 对外主要有两种方式来实现在其所在 Looper 所在线程执行指定 Runnable——post 及 sendMessage，它们都有对应的 delay 方法 。而不论是 post 还是 sendMessage，都会调用到 sendMessageDelayed方法。比如下面是 post 方法的实现：

public final boolean post(Runnable r) {

return sendMessageDelayed(getPostMessage(r), 0);

}

复制代码

可以看到它其实调用的仍然是 sendMessageDelayed 方法，只是通过 getPostMessage 方法将这个 Runnable 包装成了一个 Message 对象。

private static Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

复制代码

这个包装出来的 Message 将 callback 设置为了对应的 Runnable。

而所有的 sendMessage 和 post 方法，实际上最后都通过 sendMessageDelayed 方法调用到了 sendMessageAtTime 方法：

public final boolean sendMessageDelayed(Message msg, long delayMillis) {

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

复制代码

在 sendMessageAtTime 中，它首先通过 mQueue 拿到了对应的 MessageQueue 对象，然后调用了 enqueueMessage 方法将 Message 发送至 MessageQueue 中。

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);

}

最后实际上是调用到了 MessageQueue 的 enqueueMessage 方法将这个消息传入了 MessageQueue。它将 Message 的 target 设置为了当前的 Handler，同时要注意看到，这里在 enqueueMessage 之前先判断了一下 mAsynchronous 是否为 true，若为 true 则将该 Message 的 Asynchronous 置为 true。

private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {

msg.target = this;

if (mAsynchronous) {

msg.setAsynchronous(true);

}

return queue.enqueueMessage(msg, uptimeMillis);

}

那这个 mAsynchronous 是什么时候被赋值的呢？点进去看可以发现它的赋值是在 Handler 的构造函数中进行的。也就是说创建的 Handler 时若将 async 置为 true 则该 Handler 发出的 Message 都会被设为 Async，也就是『异步消息』。

* public Handler(Callback callback, boolean async)
* public Handler(Looper looper, Callback callback, boolean async)

关于异步消息和同步消息是什么，我们放在后面讨论。

Handler 有很多种构造函数，但其他的构造函数最后仍然会调用到上述的两种构造函数，其 async 默认会被设置为 false。

让我们看看上述的两种构造函数：

public Handler(Callback callback, boolean async) {

if (FIND\_POTENTIAL\_LEAKS) {

final Class<? extends Handler> klass = getClass();

if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&

(klass.getModifiers() & Modifier.STATIC) == 0) {

Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

klass.getCanonicalName());

}

}

mLooper = Looper.myLooper();

if (mLooper == null) {

throw new RuntimeException(

"Can't create handler inside thread that has not called Looper.prepare()");

}

mQueue = mLooper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

可以看到这个构造函数主要是对 mLooper、mQueue、mCallback、mAsynchronous 进行赋值，其中 mLooper 是通过 Looper.myLooper 方法获取到的，另一种构造函数除了 Looper 是通过外部传入以外和这个构造函数的实现差不多。同时我们还能看出，mQueue 这个 MessageQueue 是 Looper 对象内部的一个成员变量。

**消息入队**

**enqueueMessage**

我们接着看看 Handler 发送了消息后 MessageQueue 的 enqueueMessage 方法：

boolean enqueueMessage(Message msg, long when) {

if (msg.target == null) {

throw new IllegalArgumentException("Message must have a target.");

}

if (msg.isInUse()) {

throw new IllegalStateException(msg + " This message is already in use.");

}

synchronized (this) {

if (mQuitting) {

IllegalStateException e = new IllegalStateException(

msg.target + " sending message to a Handler on a dead 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) {

// New head, wake up the event queue if blocked.

msg.next = p;

mMessages = msg;

needWake = mBlocked;

} else {

// Inserted within the middle of the queue. Usually we don t have to wake

// up the event queue unless there is a barrier at the head of the queue

// and the message is the earliest asynchronous message in the queue.

needWake = mBlocked && p.target == null && msg.isAsynchronous();

Message prev;

for (;;) {

prev = p;

p = p.next;

if (p == null || when < p.when) {

break;

}

if (needWake && p.isAsynchronous()) {

needWake = false;

}

}

msg.next = p; // invariant: p == prev.next

prev.next = msg;

}

// We can assume mPtr != 0 because mQuitting is false.

if (needWake) {

nativeWake(mPtr);

}

}

return true;

}

可以看到， MessageQueue 实际上里面维护了一个 Message 构成的链表，每次插入数据都会按时间顺序进行插入，也就是说 MessageQueue 中的 Message 都是按照时间排好序的，这样的话就使得循环取出 Message 的时候只需要一个个地从前往后拿即可，这样 Message 都可以按时间先后顺序被消费。

最后在需要唤醒的情况下会调用 nativeWake 这个 native 方法用于进行唤醒，这些和唤醒机制有关的代码我们后面再进行讨论，先暂时放在一边。

**消息循环**

那么我们看看 Looper.myLooper 方法是如何获取到 Looper 对象的呢？

public static @Nullable Looper myLooper() {

return sThreadLocal.get();

}

可以看出来，这个 Looper 对象是通过 sThreadLocal.get 方法获取到的，也就是说这个 Looper 是一个线程独有的变量，每个线程具有一个不同的 Looper。

那么这个 Looper 对象是何时创建又何时放入这个 ThreadLocal 中的呢？

我们通过跟踪可以发现它实际上是通过 Looper.prepare 方法放入 ThreadLocal 中的：

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));

}

那按道理来说我们在应用程序中并没有调用 Looper.prepare 方法，为何还能通过主线程的 Handler 发送 Message 到主线程呢？

其实这个 Looper.prepare 方法在主线程创建时就已经被创建并调用了 prepare 方法进行设置，具体我们可以看到 ActivityThread 类的 main 函数：

public static void main(String[] args) {

// ...

Looper.prepareMainLooper();

// ...

Looper.loop();

// ...

}

这个 main 函数其实就是我们进程的入口，可以看出来它首先调用了 Looper.prepareMainLooper 创建了主线程的 Looper 并传入 ThreadLocal，自此我们就可以在主线程发送消息了。为什么要这样设计呢？因为其实我们的 View 绘制事件等都是通过主线程的 Handler 来进行调度的。

我们接着看到 Looper.loop 方法：

public static void loop() {

final Looper me = myLooper();

if (me == null) {

throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");

}

final MessageQueue queue = me.mQueue;

// Make sure the identity of this thread is that of the local process,

// and keep track of what that identity token actually is.

Binder.clearCallingIdentity();

final long ident = Binder.clearCallingIdentity();

for (;;) {

Message msg = queue.next(); // might block

if (msg == null) {

// No message indicates that the message queue is quitting.

return;

}

// ...

final long start = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();

final long end;

try {

msg.target.dispatchMessage(msg);

end = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();

} finally {

// ...

}

// ...

msg.recycleUnchecked();

}

}

这里其实是一个死循环，它的主要作用是遍历 MessageQueue，获取到 Looper 及 MessageQueue 后，不断通过 MessageQueue 的 next 方法获取到消息列表中的下一个 Message，之后调用了 Message 的 target 的 dispatchMessage 方法对 Message 进行消费，最后对 Message 进行了回收。

通过上面的代码可以看出，Looper 主要的作用是遍历 MessageQueue，每找到一个 Message 都会调用其 target 的 dispatchMessage 对该消息进行消费，这里的 target 也就是我们之前发出该 Message 的 Handler。

**消息遍历**

我们接着看到消息的遍历过程，它不断地从 MessageQueue 中调用 next 方法拿到消息，并对其进行消费，那我们具体看看 next 的过程：

Message next() {

final long ptr = mPtr;

if (ptr == 0) {

return null;

}

int pendingIdleHandlerCount = -1;

int nextPollTimeoutMillis = 0;

for (;;) {

if (nextPollTimeoutMillis != 0) {

Binder.flushPendingCommands();

}

// 1

nativePollOnce(ptr, nextPollTimeoutMillis);

synchronized (this) {

final long now = SystemClock.uptimeMillis();

Message prevMsg = null;

Message msg = mMessages;

// 2

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

do {

prevMsg = msg;

msg = msg.next;

} while (msg != null && !msg.isAsynchronous());

}

if (msg != null) {

// 3

if (now < msg.when) {

nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX\_VALUE);

} else {

// Got a message.

mBlocked = false;

if (prevMsg != null) {

prevMsg.next = msg.next;

} else {

mMessages = msg.next;

}

msg.next = null;

if (DEBUG) Log.v(TAG, "Returning message: " + msg);

msg.markInUse();

return msg;

}

} else {

// No more messages.

nextPollTimeoutMillis = -1;

}

if (mQuitting) {

dispose();

return null;

}

// 4

if (pendingIdleHandlerCount < 0

&& (mMessages == null || now < mMessages.when)) {

pendingIdleHandlerCount = mIdleHandlers.size();

}

if (pendingIdleHandlerCount <= 0) {

// No idle handlers to run. Loop and wait some more.

mBlocked = true;

continue;

}

if (mPendingIdleHandlers == null) {

mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];

}

mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);

}

for (int i = 0; i < pendingIdleHandlerCount; i++) {

final IdleHandler idler = mPendingIdleHandlers[i];

mPendingIdleHandlers[i] = null; // release the reference to the handler

boolean keep = false;

try {

keep = idler.queueIdle();

} catch (Throwable t) {

Log.wtf(TAG, "IdleHandler threw exception", t);

}

if (!keep) {

synchronized (this) {

mIdleHandlers.remove(idler);

}

}

}

pendingIdleHandlerCount = 0;

nextPollTimeoutMillis = 0;

}

}

上面的代码比较长，我们一步步来进行分析

首先在 next 方法内，它在不断地进行着循环，在 1 处它先调用了一次 nativePollOnce 这个 native 方法，它与 Handler 的阻塞唤醒机制有关，我们后面再进行介绍。

之后，在 2 处，它进行了一个非常特殊的处理。这里判断当前的消息是否是 target 为 null 的消息，若 target 为 null，则它会不断地向下取 Message，直到遇到一个异步的消息。到这里可能会有读者觉得很奇怪了，明明在 enqueueMessage 中避免了 Message 的 target 为 null，为什么这里还会存在 target 为 null 的消息呢？其实这与 Handler 的同步屏障机制有关，我们稍后介绍

之后便在注释 3 处判断判断当前消息是否到了应该发送的时间，若到了应该发送的时间，就会将该消息取出并返回，否则仅仅是将 nextPollTimeoutMillis 置为了剩余的时间（这里为了防止 int 越界做了防越界处理）

之后在注释 4 处，第一次循环的前提下，若 MessageQueue 为空或者消息未来才会执行，则会尝试去执行一些 idleHandler，并在执行后将 pendingIdleHandlerCount 置为 0 避免下次再次执行。

若这一次拿到的消息不是现在该执行的，那么会再次调用到 nativePollOnce，并且此次的 nextPollTimeoutMillis 不再为 0 了，这与我们后面会提到的阻塞唤醒机制有关。

**消息消费**

消息的消费是通过 Handler 的 dispatchMessage 实现的：

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

它优先调用了 Message 的 callback，若没有 callback 则会调用 Handler 中 Callback 的 handleMessage 方法，若其仍没定义则最终会调用到 Handler 自身所实现的 handleMessage 方法。

**同步屏障机制**

Handler 中存在着一种叫做同步屏障的机制，它可以实现异步消息优先执行的功能，让我们看看它是如何实现的。

**加入同步屏障**

在 Handler 中还存在了一种特殊的消息，它的 target 为 null，并不会被消费，仅仅是作为一个标识处于 MessageQueue 中。它就是 SyncBarrier （同步屏障）这种特殊的消息。我们可以通过 MessageQueue::postSyncBarrier 方法将其加入消息队列。

private int postSyncBarrier(long when) {

// Enqueue a new sync barrier token.

// We don t need to wake the queue because the purpose of a barrier is to stall it.

synchronized (this) {

final int token = mNextBarrierToken++;

final Message msg = Message.obtain();

msg.markInUse();

msg.when = when;

msg.arg1 = token;

Message prev = null;

Message p = mMessages;

if (when != 0) {

while (p != null && p.when <= when) {

prev = p;

p = p.next;

}

}

if (prev != null) { // invariant: p == prev.next

msg.next = p;

prev.next = msg;

} else {

msg.next = p;

mMessages = msg;

}

return token;

}

}

可以看到，这里并没有什么特殊的，只是将一个 target 为 null 的消息加入了消息队列中，但我们在前面的 enqueueMessage 方法中也看到了，普通的 enqueue 操作是没有办法在消息队列中放入这样一个 target 为 null 的消息的。因此这种同步屏障只能通过这个方法发出。

**移除同步屏障**

我们可以通过 removeSyncBarrier 方法来移除消息屏障。

public void removeSyncBarrier(int token) {

// Remove a sync barrier token from the queue.

// If the queue is no longer stalled by a barrier then wake it.

synchronized (this) {

Message prev = null;

Message p = mMessages;

// 找到 target 为 null 且 token 相同的消息

while (p != null && (p.target != null || p.arg1 != token)) {

prev = p;

p = p.next;

}

if (p == null) {

throw new IllegalStateException("The specified message queue synchronization "

+ " barrier token has not been posted or has already been removed.");

}

final boolean needWake;

if (prev != null) {

prev.next = p.next;

needWake = false;

} else {

mMessages = p.next;

needWake = mMessages == null || mMessages.target != null;

}

p.recycleUnchecked();

// If the loop is quitting then it is already awake.

// We can assume mPtr != 0 when mQuitting is false.

if (needWake && !mQuitting) {

nativeWake(mPtr);

}

}

}

这里主要是将同步屏障从 MessageQueue 中移除，一般执行完了异步消息后就会通过该方法将同步屏障移除。

最后若需要唤醒，调用了 nativeWake 方法进行唤醒。

**同步屏障的作用**

而看了前面 MessageQueue::next 的代码我们知道，当 MessageQueue 中遇到了一个同步屏障，则它会不断地忽略后面的同步消息直到遇到一个异步的消息，这样设计的目的其实是为了使得当队列中遇到同步屏障时，则会使得异步的消息优先执行，这样就可以使得一些消息优先执行。比如 View 的绘制过程中的 TraversalRunnable 消息就是异步消息，在放入队列之前先放入了一个消息屏障，从而使得界面绘制的消息会比其他消息优先执行，避免了因为 MessageQueue 中消息太多导致绘制消息被阻塞导致画面卡顿，当绘制完成后，就会将消息屏障移除。

**阻塞唤醒机制**

同时 Handler 中其实还存在着一种阻塞唤醒机制，我们都知道不断地进行循环是非常消耗资源的，有时我们 MessageQueue 中的消息都不是当下就需要执行的，而是要过一段时间，此时如果 Looper 仍然不断进行循环肯定是一种对于资源的浪费。因此 Handler 设计了这样一种阻塞唤醒机制使得在当下没有需要执行的消息时，就将 Looper 的 loop 过程阻塞，直到下一个任务的执行时间到达或者一些特殊情况下再将其唤醒，从而避免了上述的资源浪费。

**epoll**

这个阻塞唤醒机制是基于 Linux 的 I/O 多路复用机制 epoll 实现的，它可以同时监控多个文件描述符，当某个文件描述符就绪时，会通知对应程序进行读/写操作。

epoll 主要有三个方法，分别是 epoll\_create、epoll\_ctl、epoll\_wait。

**epoll\_create**

int epoll\_create(int size)

其功能主要是创建一个 epoll 句柄并返回，传入的 size 代表监听的描述符个数（仅仅是初次分配的 fd 个数）

**epoll\_ctl**

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

其功能是对 epoll 事件进行注册，会对该 fd 执行指定的 op 操作，参数含义如下：

* epfd：epoll 的句柄值（也就是 epoll\_create 的返回值）
* op：对 fd 执行的操作
  + EPOLL\_CTL\_ADD：注册 fd 到 epfd
  + EPOLL\_CTL\_DEL：从 epfd 中删除 fd
  + EPOLL\_CTL\_MOD：修改已注册的 fd 的监听事件
* fd：需要监听的文件描述符
* epoll\_event：需要监听的事件

epoll\_event 是一个结构体，里面的 events 代表了对应文件操作符的操作，而 data 代表了用户可用的数据。

其中 events 可取下面几个值：

* EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
* EPOLLOUT：表示对应的文件描述符可以写；
* EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外部数据来）；
* EPOLLERR：表示对应的文件描述符发生错误；
* EPOLLHUP：表示对应的文件描述符被挂断；
* EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
* EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

**epoll\_wait**

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

其功能是等待事件的上报，参数含义如下：

* epfd：epoll 的句柄值
* events：从内核中得到的事件集合
* maxevents：events 数量，不能超过 create 时的 size
* timeout：超时时间

当调用了该方法后，会进入阻塞状态，等待 epfd 上的 IO 事件，若 epfd 监听的某个文件描述符发生前面指定的 event 时，就会进行回调，从而使得 epoll 被唤醒并返回需要处理的事件个数。若超过了设定的超时时间，同样也会被唤醒并返回 0 避免一直阻塞。

而 Handler 的阻塞唤醒机制就是基于上面的 epoll 的阻塞特性，我们来看看它的具体实现。

**native 初始化**

在 Java 中的 MessageQueue 创建时会调用到 nativeInit 方法，在 native 层会创建 NativeMessageQueue 并返回其地址，之后都是通过这个地址来与该 NativeMessageQueue 进行通信（也就是 MessageQueue 中的 mPtr，类似 MMKV 的做法），而在 NativeMessageQueue 创建时又会创建 Native 层下的 Looper，我们看到 Native 下的 Looper 的构造函数：

Looper::Looper(bool allowNonCallbacks) :

mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),

mPolling(false), mEpollFd(-1), mEpollRebuildRequired(false),

mNextRequestSeq(0), mResponseIndex(0), mNextMessageUptime(LLONG\_MAX) {

mWakeEventFd = eventfd(0, EFD\_NONBLOCK); //构造唤醒事件的fd

AutoMutex \_l(mLock);

rebuildEpollLocked();

}

可以看到，它调用了 rebuildEpollLocked 方法对 epoll 进行初始化，让我们看看其实现

void Looper::rebuildEpollLocked() {

if (mEpollFd >= 0) {

close(mEpollFd);

}

mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT);

struct epoll\_event eventItem;

memset(& eventItem, 0, sizeof(epoll\_event));

eventItem.events = EPOLLIN;

eventItem.data.fd = mWakeEventFd;

int result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mWakeEventFd, & eventItem);

for (size\_t i = 0; i < mRequests.size(); i++) {

const Request& request = mRequests.valueAt(i);

struct epoll\_event eventItem;

request.initEventItem(&eventItem);

int epollResult = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, request.fd, & eventItem);

}

}

可以看到，这里首先关闭了旧的 epoll 描述符，之后又调用了 epoll\_create 创建了新的 epoll 描述符，然后进行了一些初始化后，将 mWakeEventFd 及 mRequests 中的 fd 都注册到了 epoll 的描述符中，注册的事件都是 EPOLLIN。

这就意味着当这些文件描述符其中一个发生了 IO 时，就会通知 epoll\_wait 使其唤醒，那么我们猜测 Handler 的阻塞就是通过 epoll\_wait 实现的。

同时可以发现，Native 层也是存在 MessageQueue 及 Looper 的，也就是说 ative 层实际上也是有一套消息机制的，这些我们到后面再进行介绍。

**native 阻塞实现**

我们看看阻塞，它的实现就在我们之前看到的 MessageQueue::next 中，当发现要返回的消息将来才会执行，则会计算出当下距离其将要执行的时间还差多少毫秒，并调用 nativePollOnce 方法将返回的过程阻塞到指定的时间。

nativePollOnce 很显然是一个 native 方法，它最后调用到了 Looper 这个 native 层类的 pollOnce 方法。

int Looper::pollOnce(int timeoutMillis, int\* outFd, int\* outEvents, void\*\* outData) {

int result = 0;

for (;;) {

while (mResponseIndex < mResponses.size()) {

const Response& response = mResponses.itemAt(mResponseIndex++);

int ident = response.request.ident;

if (ident >= 0) {

int fd = response.request.fd;

int events = response.events;

void\* data = response.request.data;

if (outFd != NULL) \*outFd = fd;

if (outEvents != NULL) \*outEvents = events;

if (outData != NULL) \*outData = data;

return ident;

}

}

if (result != 0) {

if (outFd != NULL) \*outFd = 0;

if (outEvents != NULL) \*outEvents = 0;

if (outData != NULL) \*outData = NULL;

return result;

}

result = pollInner(timeoutMillis);

}

}

前面主要是一些对 Native 层消息机制的处理，我们先暂时不关心，这里最后调用到了 pollInner 方法：

int Looper::pollInner(int timeoutMillis) {

// ...

int result = POLL\_WAKE;

mResponses.clear();

mResponseIndex = 0;

mPolling = true;

struct epoll\_event eventItems[EPOLL\_MAX\_EVENTS];

// 1

int eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

// ...

return result;

}

可以发现，这里在 1 处调用了 epoll\_wait 方法，并传入了我们之前在 natviePollOnce 方法传入的当前时间距下个任务执行时间的差值。这就是我们的阻塞功能的核心实现了，调用该方法后，会一直阻塞，直到到达我们设定的时间或之前我们在 epoll 的 fd 中注册的几个 fd 发生了 IO。其实到了这里我们就可以猜到，nativeWake 方法就是通过对注册的 mWakeEventFd 进行操作从而实现的唤醒。

后面主要是一些对 Native 层消息机制的处理，这篇文章暂时不关注，它的逻辑和 Java 层是基本一致的。

**native 唤醒**

nativeWake 方法最后通过 NativeMessageQueue 的 wake 方法调用到了 Native 下 Looper 的 wake 方法：

void Looper::wake() {

uint64\_t inc = 1;

ssize\_t nWrite = TEMP\_FAILURE\_RETRY(write(mWakeEventFd, &inc, sizeof(uint64\_t)));

if (nWrite != sizeof(uint64\_t)) {

if (errno != EAGAIN) {

ALOGW("Could not write wake signal, errno=%d", errno);

}

}

}

这里其实就是调用了 write 方法，对 mWakeEventFd 中写入了 1，从而使得监听该 fd 的 pollOnce 方法被唤醒，从而使得 Java 中的 next 方法继续执行。

那我们再回去看看，在什么情况下，Java 层会调用 natvieWake 方法进行唤醒呢？

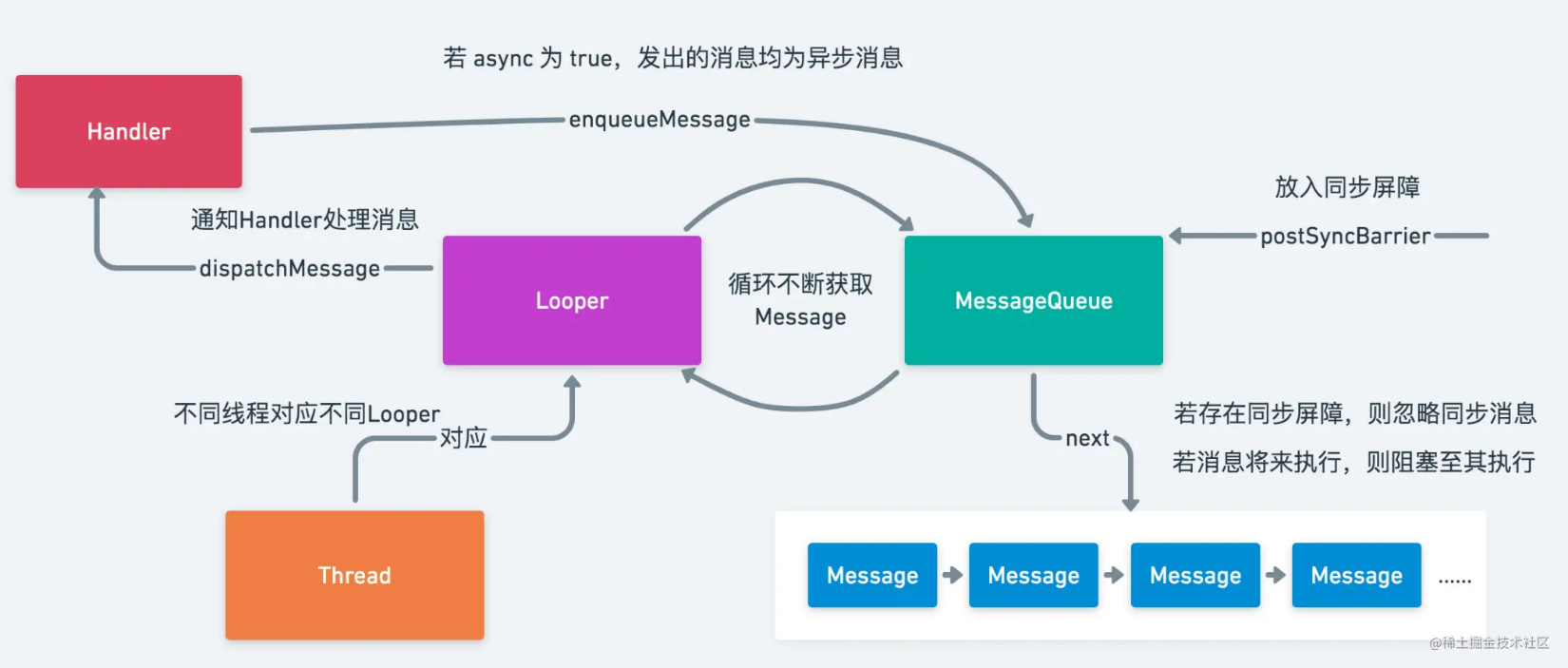
MessageQueue 类中调用 nativeWake 方法主要有下列几个时机：

* 调用 MessageQueue 的 quit 方法进行退出时，会进行唤醒
* 消息入队时，若插入的消息在链表最前端（最早将执行）或者有同步屏障时插入的是最前端的异步消息（最早被执行的异步消息）
* 移除同步屏障时，若消息列表为空或者同步屏障后面不是异步消息时

可以发现，主要是在可能不再需要阻塞的情况下进行唤醒。（比如加入了一个更早的任务，那继续阻塞显然会影响这个任务的执行）

**总结**

Android 的消息机制在 Java 层及 Native 层均是由 Handler、Looper、MessageQueue 三者构成



* Handler：事件的发送及处理者，在构造方法中可以设置其 async，默认为 false。若 async 为 true 则该 Handler 发送的 Message 均为异步消息，有同步屏障的情况下会被优先处理。
* Looper：一个用于遍历 MessageQueue 的类，每个线程有一个独有的 Looper，它会在所处的线程开启一个死循环，不断从 MessageQueue 中拿出消息，并将其发送给 target 进行处理
* MessageQueue：用于存储 Message，内部维护了 Message 的链表，每次拿取 Message 时，若该 Message 离真正执行还需要一段时间，会通过 nativePollOnce 进入阻塞状态，避免资源的浪费。若存在消息屏障，则会忽略同步消息优先拿取异步消息，从而实现异步消息的优先消费。

**相关问题**

下面还有一些与 Handler 相关的常见问题，可以结合前面的内容得到答案。

**问题 1**

Looper 是在主线程创建，同时其 loop 方法也是在主线程执行，为什么这样一个死循环却不会阻塞主线程呢？

我们看到 ActivityThread 中，它实际上是有一个 handleMessage 方法，其实 ActivityThread 就是一个 Handler，我们在使用的过程中的很多事件（如 Activity、Service 的各种生命周期）都在这里的各种 Case 中，也就是说我们平时说的主线程其实就是依靠这个 Looper 的 loop 方法来处理各种消息，从而实现如 Activity 的声明周期的回调等等的处理，从而回调给我们使用者。

public void handleMessage(Message msg) {

if (DEBUG\_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));

switch (msg.what) {

case XXX:

// ...

}

Object obj = msg.obj;

if (obj instanceof SomeArgs) {

((SomeArgs) obj).recycle();

}

if (DEBUG\_MESSAGES) Slog.v(TAG, "<<< done: " + codeToString(msg.what));

}

}

因此不能说主线程不会阻塞，因为主线程本身就是阻塞的，其中所有事件都由主线程进行处理，从而使得我们能在这个循环的过程中作出自己的各种处理（如 View 的绘制等）。

而这个问题的意思应该是为何这样一个死循环不会使得界面卡顿，这有两个原因：

1. 界面的绘制本身就是这个循环内的一个事件
2. 界面的绘制是通过了同步屏障保护下发送的异步消息，会被主线程优先处理，因此使得界面绘制拥有了最高的优先级，不会因为 Handler 中事件太多而造成卡顿。

**问题 2**

Handler 的内存泄漏是怎么回事？如何产生的呢？

首先，造成 Handler 的内存泄漏往往是因为如下的这种代码：

public class XXXActivity extends BaseActivity {

// ...

private Handler mHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

// 一些处理

}

};

// ...

}

那这样为什么会造成内存泄漏呢？

我们都知道，匿名内部类会持有外部类的引用，也就是说这里的 Handler 会持有其外部类 XXXActivity 的引用。而我们可以回忆一下 sendMessage 的过程中，它会将 Message 的 target 设置为 Handler，也就是说明这个 Message 持有了 mHandler 的引用。那么我们假设通过 mHandler 发送了一个 2 分钟后的延时消息，在两分钟还没到的时候，我们关闭了界面。按道理来说此时 Activity 可以被 GC 回收，但由于此时 Message 还处于 MessageQueue 中，MessageQueue 这个对象持有了 Message 的引用，Message 又持有了我们的 Handler 的引用，同时由于 Handler 又持有了其外部类 XXXActivity 的引用。这就导致此时 XXXActivity 仍然是可达的，因此导致 XXXActivity 无法被 GC 回收，这就造成了内存泄漏。

因此我们使用 Handler 最好将其定义为 static 的，避免其持有外部类的引用导致类似的内存泄漏问题。如果此时还需要用到 XXXActivity 的一些信息，可以通过 WeakReference 来使其持有 Activity 的弱引用，从而可以访问其中的某些信息，又避免了内存泄漏问题。

**参考资料**

[Android消息机制2-Handler(Native层) —— Gityuan](https://link.juejin.cn/?target=http%3A%2F%2Fgityuan.com%2F2015%2F12%2F27%2Fhandler-message-native%2F)

[IO多路复用之epoll总结](https://link.juejin.cn/?target=https%3A%2F%2Fwww.cnblogs.com%2FAnker%2Fp%2F3263780.html)

[ActivityThread.java](https://link.juejin.cn/?target=https%3A%2F%2Fandroid.googlesource.com%2Fplatform%2Fframeworks%2Fbase%2F%2B%2Fmaster%2Fcore%2Fjava%2Fandroid%2Fapp%2FActivityThread.java)

目录

* [发送消息](https://juejin.cn/post/6844904036718624782#heading-0)
  + [post & sendMessage](https://juejin.cn/post/6844904036718624782#heading-1)
* [消息入队](https://juejin.cn/post/6844904036718624782#heading-2)
  + [enqueueMessage](https://juejin.cn/post/6844904036718624782#heading-3)
* [消息循环](https://juejin.cn/post/6844904036718624782#heading-4)
* [消息遍历](https://juejin.cn/post/6844904036718624782#heading-5)
* [消息消费](https://juejin.cn/post/6844904036718624782#heading-6)
* [同步屏障机制](https://juejin.cn/post/6844904036718624782#heading-7)
  + [加入同步屏障](https://juejin.cn/post/6844904036718624782#heading-8)
  + [移除同步屏障](https://juejin.cn/post/6844904036718624782#heading-9)
  + [同步屏障的作用](https://juejin.cn/post/6844904036718624782#heading-10)
* [阻塞唤醒机制](https://juejin.cn/post/6844904036718624782#heading-11)
  + [epoll](https://juejin.cn/post/6844904036718624782#heading-12)
    - [epoll\_create](https://juejin.cn/post/6844904036718624782#heading-13)
    - [epoll\_ctl](https://juejin.cn/post/6844904036718624782#heading-14)
    - [epoll\_wait](https://juejin.cn/post/6844904036718624782#heading-15)
  + [native 初始化](https://juejin.cn/post/6844904036718624782#heading-16)
  + [native 阻塞实现](https://juejin.cn/post/6844904036718624782#heading-17)
  + [native 唤醒](https://juejin.cn/post/6844904036718624782#heading-18)
* [总结](https://juejin.cn/post/6844904036718624782#heading-19)
* [相关问题](https://juejin.cn/post/6844904036718624782#heading-20)
  + [问题 1](https://juejin.cn/post/6844904036718624782#heading-21)
  + [问题 2](https://juejin.cn/post/6844904036718624782#heading-22)
* [参考资料](https://juejin.cn/post/6844904036718624782#heading-23)