Android消息机制-Handler(下篇)

Jan 1, 2016

- 一、概述
- 二、MessageQueue
  - 2.1 nativeInit()
  - 2.2 nativeDestroy()
  - 2.3 nativePollOnce()
  - 2.4 nativeWake()
  - 2.5 Native sendMessage
  - 2.6 小结
- 三、Native结构体和类
  - 3.1 Message结构体
  - 3.2 消息处理类
  - 3.3 回调类
  - 3.4 Looper类
  - 3.5 ALooper类
- 总结

> 本文基于Android 6.0的源代码，来分析native层的消息处理机制

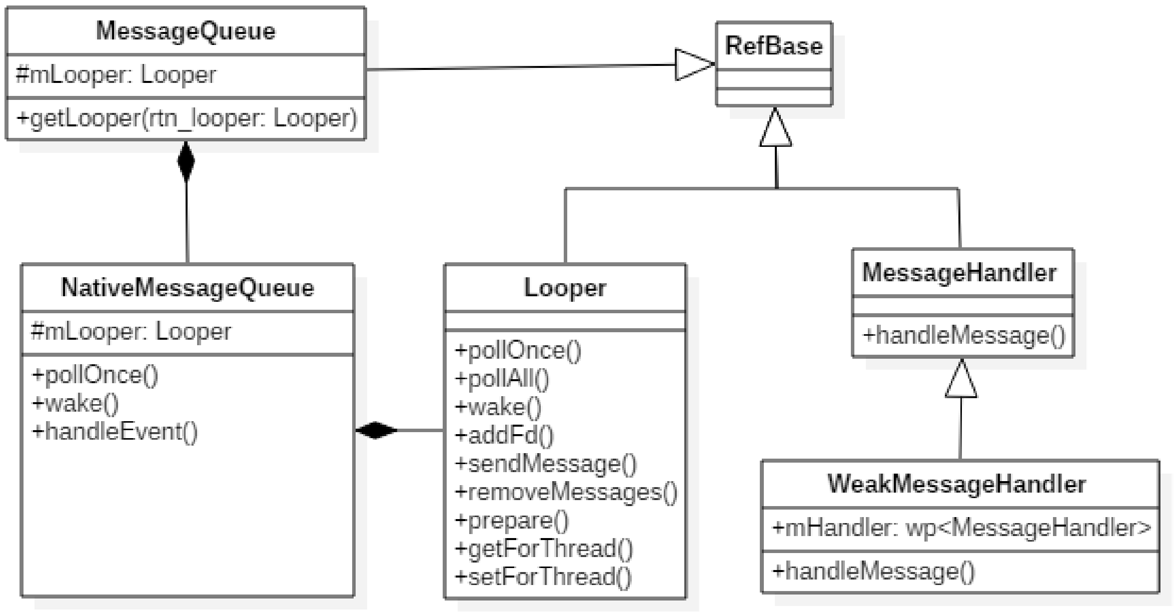
相关源码

```
framework/base/core/java/andorid/os/MessageQueue.java

framework/base/core/jni/android_os_MessageQueue.h
framework/base/core/jni/android_os_MessageQueue.cpp

system/core/include/utils/Looper.h
system/core/libutils/Looper.cpp
system/core/libutils/RefBase.cpp

framework/native/include/android/looper.h
framework/base/native/android/looper.cpp
```

一、概述

在文章Android消息机制-Handler(上篇)(http://www.yuanhh.com/2015/12/26/handler-message/#looper-1)中讲解了Java层的消息处理机制，其中 `MessageQueue` 类里面涉及到多个native方法，除了MessageQueue的native方法，native层本身也有一套完整的消息机制，用于处理native的消息。在整个消息机制中，而 `MessageQueue` 是连接Java层和Native层的纽带，换言之，Java层可以向 `MessageQueue` 消息队列中添加消息，Native层也可以向 `MessageQueue` 消息队列中添加消息。

Native层的关系图：



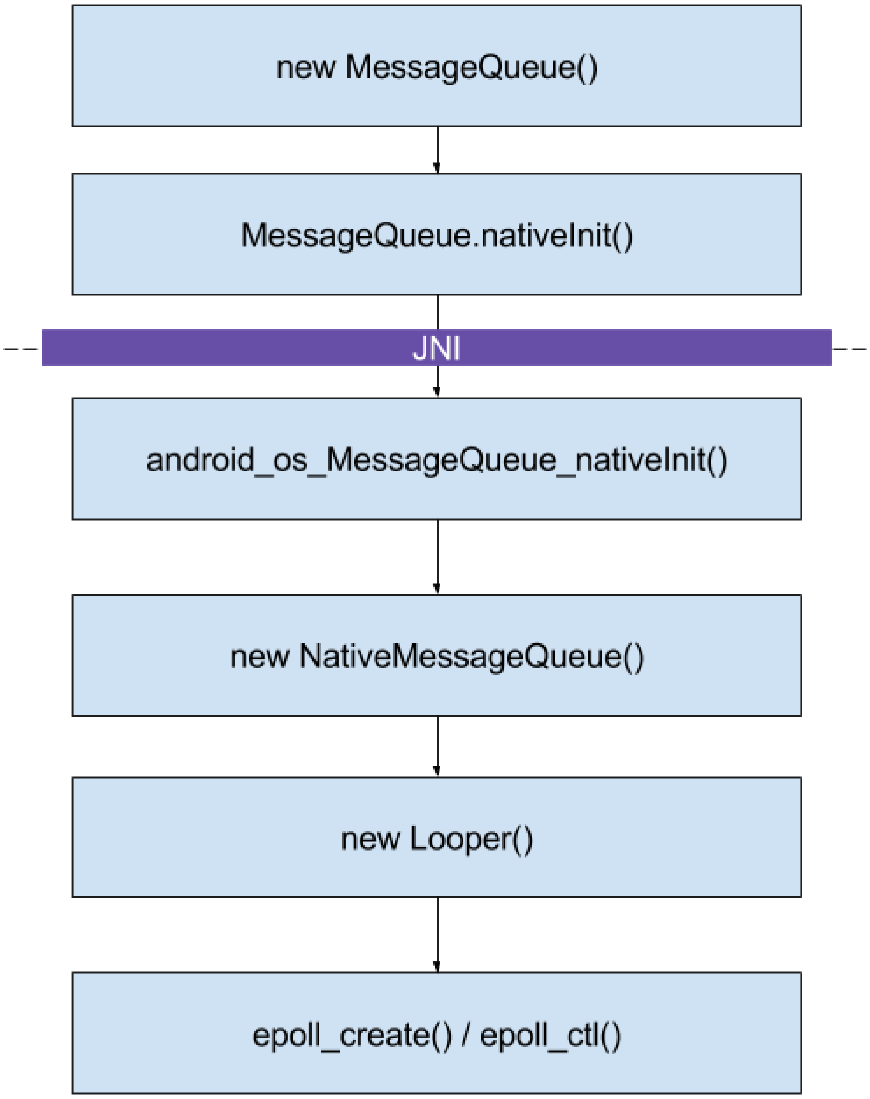
二、MessageQueue

在MessageQueue中的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); //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);
```

2.1 nativeInit()

初始化过程的调用链如下：



下面来进一步来看看调用链的过程：

【1】 new MessageQueue()

==> MessageQueue.java

```
MessageQueue(boolean quitAllowed) {
    mQuitAllowed = quitAllowed;
    mPtr = nativeInit();  //mPtr记录native消息队列的信息 【2】
}
```

【2】android_os_MessageQueue_nativeInit()

==> android_os_MessageQueue.cpp

```
static jlong android_os_MessageQueue_nativeInit(JNIEnv* env, jclass cl
azz) {
    NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
//初始化native消息队列 【3】
    if (!nativeMessageQueue) {
        jniThrowRuntimeException(env, "Unable to allocate native queu
e");
        return 0;
    }
    nativeMessageQueue->incStrong(env);
    return reinterpret_cast<jlong>(nativeMessageQueue);
}
```

【3】new NativeMessageQueue()

==> android_os_MessageQueue.cpp

```
NativeMessageQueue::NativeMessageQueue() : mPollEnv(NULL), mPollObj(NU
LL), mExceptionObj(NULL) {
    mLooper = Looper::getForThread(); //获取TLS中的Looper对象
    if (mLooper == NULL) {
        mLooper = new Looper(false); //创建native层的Looper 【4】
        Looper::setForThread(mLooper); //保存native层的Looper到TLS中
    }
}
```

- Looper::getForThread()，功能类比于Java层的Looper.myLooper();
- Looper::setForThread(mLooper)，功能类比于Java层的ThreadLocal.set();

MessageQueue是在Java层与Native层有着紧密的联系，但是此次Native层的Looper与Java层的Looper没有任何的关系，可以发现native基本等价于用C++重写了Java的Looper逻辑，故可以发现很多功能类似的地方。

【4】new Looper()

==> Looper.cpp

```
Looper::Looper(bool allowNonCallbacks) :
        mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),
        mPolling(false), mEpollFd(-1), mEpollRebuildRequired(false),
        mNextRequestSeq(0), mResponseIndex(0), mNextMessageUptime(LLON
G_MAX) {
    mWakeEventFd = eventfd(0, EFD_NONBLOCK); //构造唤醒事件的fd
    AutoMutex _l(mLock);
    rebuildEpollLocked();  //重建Epoll事件【5】
}
```

【5】epoll_create/epoll_ctl

==> Looper.cpp

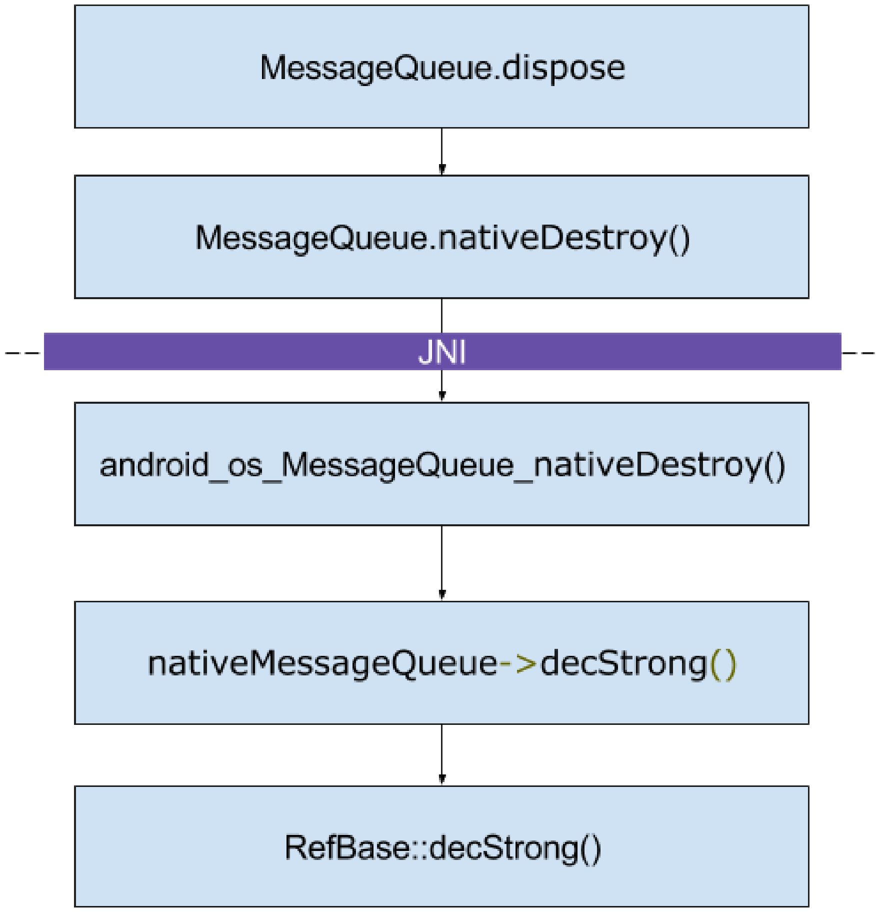
```
void Looper::rebuildEpollLocked() {
    if (mEpollFd >= 0) {
        close(mEpollFd); //关闭旧的epoll实例
    }
    mEpollFd = epoll_create(EPOLL_SIZE_HINT); //创建新的epoll实例，并注册wake管道
    struct epoll_event eventItem;
    memset(& eventItem, 0, sizeof(epoll_event)); //把未使用的数据区域进行置0操作
    eventItem.events = EPOLLIN; //可读事件
    eventItem.data.fd = mWakeEventFd;
    //将唤醒事件(mWakeEventFd)添加到epoll实例(mEpollFd)
    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);
        //将request队列的事件，分别添加到epoll实例
        int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, request.fd, & eventItem);
        if (epollResult < 0) {
            ALOGE("Error adding epoll events for fd %d while rebuilding epoll set, errno=%d", request.fd, errno);
        }
    }
}
```

关于epoll的原理以及为什么选择epoll的方式，可查看文章select/poll/epoll对比分析 (http://www.yuanhh.com/2015/12/06/linux_epoll/)。

另外，需要注意 `Request` 队列，也添加到epoll的监控范围内。

2.2 nativeDestroy()

清理回收的调用链如下：



下面来进一步来看看调用链的过程：

【1】MessageQueue.dispose()

==> MessageQueue.java

```
private void dispose() {
    if (mPtr != 0) {
        nativeDestroy(mPtr); 【2】
        mPtr = 0;
    }
}
```

【2】android_os_MessageQueue_nativeDestroy()

==> android_os_MessageQueue.cpp

```
static void android_os_MessageQueue_nativeDestroy(JNIEnv* env, jclass
clazz, jlong ptr) {
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMe
ssageQueue*>(ptr);
    nativeMessageQueue->decStrong(env); 【3】
}
```

nativeMessageQueue继承自RefBase类，所以decStrong最终调用的是RefBase.decStrong().

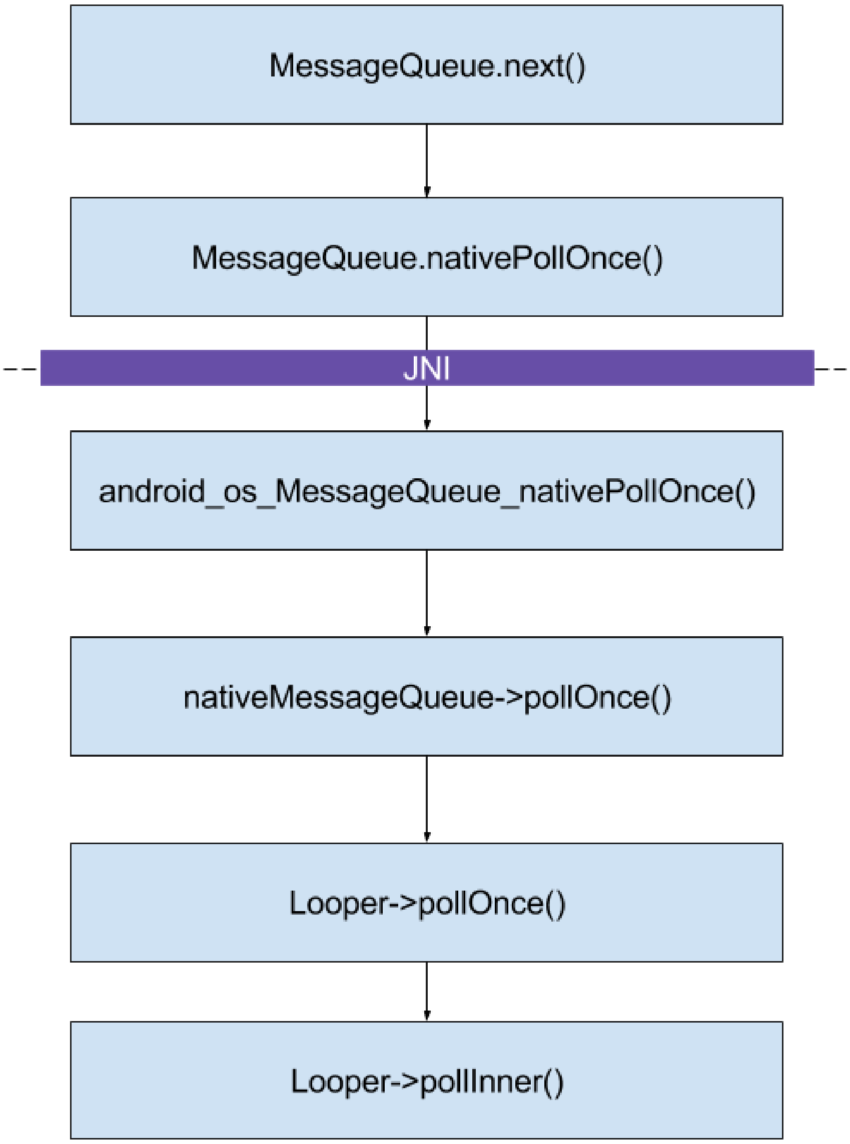
【3】RefBase::decStrong()

==> RefBase.cpp

```
void RefBase::decStrong(const void* id) const
{
    weakref_impl* const refs = mRefs;
    refs->removeStrongRef(id); //移除强引用
    const int32_t c = android_atomic_dec(&refs->mStrong);
    if (c == 1) {
        refs->mBase->onLastStrongRef(id);
        if ((refs->mFlags&OBJECT_LIFETIME_MASK) == OBJECT_LIFETIME_STR
ONG) {
            delete this;
        }
    }
    refs->decWeak(id); // 移除弱引用
}
```

2.3 nativePollOnce()

nativePollOnce用于提取消息队列中的消息，提取消息的调用链，如下：



下面来进一步来看看调用链的过程：

【1】MessageQueue.next()

==> MessageQueue.java

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) {
        return null;
    }

    for (;;) {
        ...
        nativePollOnce(ptr, nextPollTimeoutMillis); //阻塞操作 【2】
        ...
    }
```

【2】android_os_MessageQueue_nativePollOnce()

==> android_os_MessageQueue.cpp

```
static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jobject obj, jlong ptr, jint timeoutMillis) {
    //将Java层传递下来的mPtr转换为nativeMessageQueue
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->pollOnce(env, obj, timeoutMillis); 【3】
}
```

【3】NativeMessageQueue::pollOnce()

==> android_os_MessageQueue.cpp

```
void NativeMessageQueue::pollOnce(JNIEnv* env, jobject pollObj, int timeoutMillis) {
    mPollEnv = env;
    mPollObj = pollObj;
    mLooper->pollOnce(timeoutMillis); 【4】
    mPollObj = NULL;
    mPollEnv = NULL;
    if (mExceptionObj) {
        env->Throw(mExceptionObj);
        env->DeleteLocalRef(mExceptionObj);
        mExceptionObj = NULL;
    }
}
```

【4】Looper::pollOnce()

==> Looper.h

```
inline int pollOnce(int timeoutMillis) {
    return pollOnce(timeoutMillis, NULL, NULL, NULL); 【5】
}
```

【5】 Looper::pollOnce()

==> Looper.cpp

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, vo
id** outData) {
    int result = 0;
    for (;;) {
        // 先处理没有Callback方法的 Response事件
        while (mResponseIndex < mResponses.size()) {
            const Response& response = mResponses.itemAt(mResponseInde
x++);
            int ident = response.request.ident;
            if (ident >= 0) { //ident大于0，则表示没有callback, 因为POL
L_CALLBACK = -2,
                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;
        }
        // 再处理内部轮询
        result = pollInner(timeoutMillis); 【6】
    }
}
```

参数说明：

- timeoutMillis：超时时长
- outFd：发生事件的文件描述符
- outEvents：当前outFd上发生的事件，包含以下4类事件
    - EVENT_INPUT 可读
    - EVENT_OUTPUT 可写
    - EVENT_ERROR 错误
    - EVENT_HANGUP 中断
- outData：上下文数据

【6】Looper::pollInner()

==> Looper.cpp

```
int Looper::pollInner(int timeoutMillis) {
    ...
    int result = POLL_WAKE;
    mResponses.clear();
    mResponseIndex = 0;
    mPolling = true; //即将处于idle状态
    struct epoll_event eventItems[EPOLL_MAX_EVENTS]; //fd最大个数为16
    //等待事件发生或者超时，在nativeWake()方法，向管道写端写入字符，则该方法会
返回；
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENT
S, timeoutMillis);

    mPolling = false; //不再处于idle状态
    mLock.lock();  //请求锁
    if (mEpollRebuildRequired) {
        mEpollRebuildRequired = false;
        rebuildEpollLocked();  // epoll重建，直接跳转Done;
        goto Done;
    }
    if (eventCount < 0) {
        if (errno == EINTR) {
            goto Done;
        }
        result = POLL_ERROR; // epoll事件个数小于0，发生错误，直接跳转Don
e;
        goto Done;
    }
    if (eventCount == 0) {  //epoll事件个数等于0，发生超时，直接跳转Done;
        result = POLL_TIMEOUT;
        goto Done;
    }

    //循环遍历，处理所有的事件
    for (int i = 0; i < eventCount; i++) {
        int fd = eventItems[i].data.fd;
        uint32_t epollEvents = eventItems[i].events;
        if (fd == mWakeEventFd) {
            if (epollEvents & EPOLLIN) {
                awoken(); //已经唤醒了，则读取并清空管道数据【7】
            }
        } else {
            ssize_t requestIndex = mRequests.indexOfKey(fd);
            if (requestIndex >= 0) {
                int events = 0;
                if (epollEvents & EPOLLIN) events |= EVENT_INPUT;
                if (epollEvents & EPOLLOUT) events |= EVENT_OUTPUT;
                if (epollEvents & EPOLLERR) events |= EVENT_ERROR;
                if (epollEvents & EPOLLHUP) events |= EVENT_HANGUP;
```

```
                //处理request，生成对应的reponse对象，push到响应数组
                pushResponse(events, mRequests.valueAt(requestIndex));
            }
        }
    }
Done: ;
    //再处理Native的Message，调用相应回调方法
    mNextMessageUptime = LLONG_MAX;
    while (mMessageEnvelopes.size() != 0) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        const MessageEnvelope& messageEnvelope = mMessageEnvelopes.ite
mAt(0);
        if (messageEnvelope.uptime <= now) {
            {
                sp<MessageHandler> handler = messageEnvelope.handler;
                Message message = messageEnvelope.message;
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();  //释放锁
                handler->handleMessage(message);  // 处理消息事件
            }
            mLock.lock();  //请求锁
            mSendingMessage = false;
            result = POLL_CALLBACK; // 发生回调
        } else {
            mNextMessageUptime = messageEnvelope.uptime;
            break;
        }
    }
    mLock.unlock(); //释放锁

    //处理带有Callback()方法的Response事件，执行Reponse相应的回调方法
    for (size_t i = 0; i < mResponses.size(); i++) {
        Response& response = mResponses.editItemAt(i);
        if (response.request.ident == POLL_CALLBACK) {
            int fd = response.request.fd;
            int events = response.events;
            void* data = response.request.data;
            // 处理请求的回调方法
            int callbackResult = response.request.callback->handleEven
t(fd, events, data);
            if (callbackResult == 0) {
                removeFd(fd, response.request.seq); //移除fd
            }
            response.request.callback.clear(); //清除reponse引用的回调方
法
            result = POLL_CALLBACK;  // 发生回调
        }
    }
```

```
    return result;
}
```

pollOnce返回值说明：

- POLL_WAKE： 表示由wake()触发，即pipe写端的write事件触发；
- POLL_CALLBACK： 表示某个被监听fd被触发。
- POLL_TIMEOUT： 表示等待超时；
- POLL_ERROR：表示等待期间发生错误；

【7】Looper::awoken()

```
void Looper::awoken() {
    uint64_t counter;
    //不断读取管道数据，目的就是为了清空管道内容
    TEMP_FAILURE_RETRY(read(mWakeEventFd, &counter, sizeof(uint6
4_t)));
}
```

poll小结

pollInner()方法的处理流程：

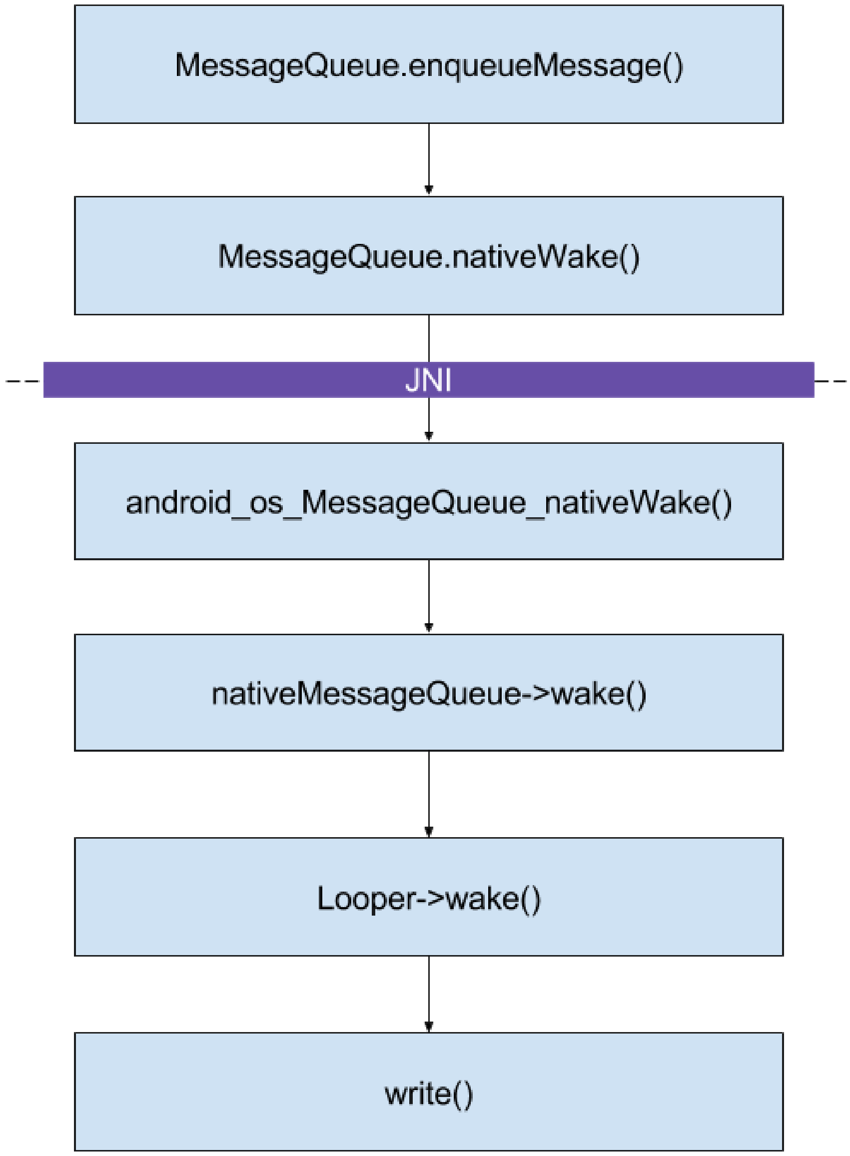
1. 先调用epoll_wait()，这是阻塞方法，用于等待事件发生或者超时；
2. 对于epoll_wait()返回，当且仅当以下3种情况出现：
    - POLL_ERROR，发生错误，直接跳转到Done；
    - POLL_TIMEOUT，发生超时，直接跳转到Done；
    - 检测到管道有事件发生，则再根据情况做相应处理：
        - 如果是管道读端产生事件，则直接读取管道的数据；
        - 如果是其他事件，则处理request，生成对应的reponse对象，push到reponse数组；
3. 进入Done标记位的代码段：
    - 先处理Native的Message，调用Native 的Handler来处理该Message;
    - 再处理Response数组，POLL_CALLBACK类型的事件；

从上面的流程，可以发现对于Request先收集，一并放入reponse数组，而不是马上执行。真正在Done开始执行的时候，是先处理native Message，再处理Request，说明native Message的优先级高于Request请求的优先级。

另外pollOnce()方法中，先处理Response数组中不带Callback的事件，再调用了pollInner()方法。

2.4 nativeWake()

nativeWake用于唤醒功能，在添加消息到消息队列 `enqueueMessage()`，或者把消息从消息队列中全部移除 `quit()` ，再有需要时都会调用 `nativeWake` 方法。包含唤醒过程的添加消息的调用链，如下：



下面来进一步来看看调用链的过程：

【1】MessageQueue.enqueueMessage()

==> MessageQueue.java

```
boolean enqueueMessage(Message msg, long when) {
    ... //将Message按时间顺序插入MessageQueue
    if (needWake) {
            nativeWake(mPtr); 【2】
        }
}
```

往消息队列添加Message时，需要根据mBlocked情况来决定是否需要调用nativeWake。

【2】android_os_MessageQueue_nativeWake()

==> android_os_MessageQueue.cpp

```
static void android_os_MessageQueue_nativeWake(JNIEnv* env, jclass clazz, jlong ptr) {
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->wake(); 【3】
}
```

【3】NativeMessageQueue::wake()

==> android_os_MessageQueue.cpp

```
void NativeMessageQueue::wake() {
    mLooper->wake();  【4】
}
```

【4】Looper::wake()

==> Looper.cpp

```
void Looper::wake() {
    uint64_t inc = 1;
    // 向管道mWakeEventFd写入字符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);
        }
    }
}
```

其中 `TEMP_FAILURE_RETRY` 是一个宏定义， 当执行 `write` 失败后，会不断重复执行，直到执行成功为止。

2.5 Native sendMessage

在Android消息机制-Handler(上篇)(http://www.yuanhh.com/2015/12/26/handler-message/#sendmessage)文中，讲述了Java层如何向MessageQueue类中添加消息，那么接下来讲讲Native层如何向MessageQueue发送消息。

【1】sendMessage

```
void Looper::sendMessage(const sp<MessageHandler>& handler, const Message& message) {
    nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
    sendMessageAtTime(now, handler, message);
}
```

【2】sendMessageDelayed

```
void Looper::sendMessageDelayed(nsecs_t uptimeDelay, const sp<MessageH
andler>& handler,
        const Message& message) {
    nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
    sendMessageAtTime(now + uptimeDelay, handler, message);
}
```

sendMessage(),sendMessageDelayed() 都是调用sendMessageAtTime()来完成消息插入。

【3】sendMessageAtTime

```
void Looper::sendMessageAtTime(nsecs_t uptime, const sp<MessageHandle
r>& handler,
        const Message& message) {
    size_t i = 0;
    { //请求锁
        AutoMutex _l(mLock);
        size_t messageCount = mMessageEnvelopes.size();
        //找到message应该插入的位置i
        while (i < messageCount && uptime >= mMessageEnvelopes.itemA
t(i).uptime) {
            i += 1;
        }
        MessageEnvelope messageEnvelope(uptime, handler, message);
        mMessageEnvelopes.insertAt(messageEnvelope, i, 1);
        //如果当前正在发送消息，那么不再调用wake()，直接返回。
        if (mSendingMessage) {
            return;
        }
    } //释放锁
    //当把消息加入到消息队列的头部时，需要唤醒poll循环。
    if (i == 0) {
        wake();
    }
}
```

2.6 小结

本节介绍MessageQueue的native()方法，经过层层调用：

- nativeInit()方法，最终实现由epoll机制中的epoll_create()/epoll_ctl()完成；
- nativeDestroy()方法，最终实现由RefBase::decStrong()完成；
- nativePollOnce()方法，最终实现由Looper::pollOnce()完成；
- nativeWake()方法，最终实现由Looper::wake()调用write方法，向管道字符完成；

- nativeIsPolling()，nativeSetFileDescriptorEvents()这两个方法类似，此处就不一一列举。

三、Native结构体和类

Looper.h/ Looper.cpp文件中，定义了Message结构体，消息处理类，回调类，Looper类。

3.1 Message结构体

```
struct Message {
    Message() : what(0) { }
    Message(int what) : what(what) { }
    int what; // 消息类型
};
```

3.2 消息处理类

MessageHandler类

```
class MessageHandler : public virtual RefBase {
protected:
    virtual ~MessageHandler() { }
public:
    virtual void handleMessage(const Message& message) = 0;
};
```

WeakMessageHandler类，继承于MessageHandler类

```
class WeakMessageHandler : public MessageHandler {
protected:
    virtual ~WeakMessageHandler();
public:
    WeakMessageHandler(const wp<MessageHandler>& handler);
    virtual void handleMessage(const Message& message);
private:
    wp<MessageHandler> mHandler;
};

void WeakMessageHandler::handleMessage(const Message& message) {
    sp<MessageHandler> handler = mHandler.promote();
    if (handler != NULL) {
        handler->handleMessage(message); //调用MessageHandler类的处理方
法()
    }
}
```

3.3 回调类

LooperCallback类

```
class LooperCallback : public virtual RefBase {
protected:
    virtual ~LooperCallback() { }
public:
    //用于处理指定的文件描述符的poll事件
    virtual int handleEvent(int fd, int events, void* data) = 0;
};
```

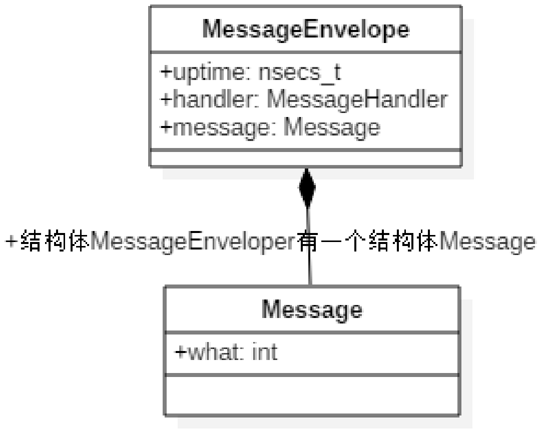
SimpleLooperCallback类，继承于LooperCallback类

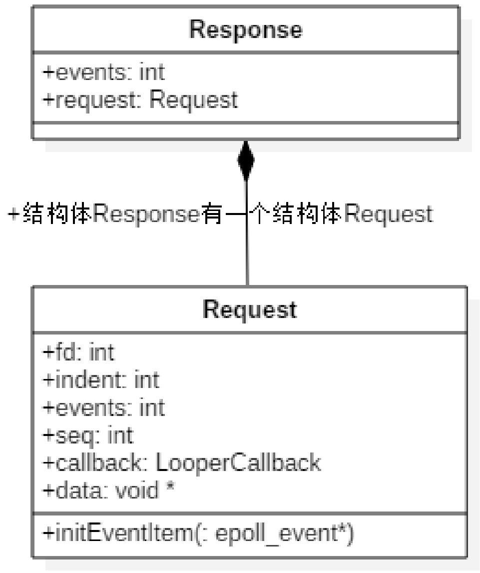
```
class SimpleLooperCallback : public LooperCallback {
protected:
    virtual ~SimpleLooperCallback();
public:
    SimpleLooperCallback(Looper_callbackFunc callback);
    virtual int handleEvent(int fd, int events, void* data);
private:
    Looper_callbackFunc mCallback;
};

int SimpleLooperCallback::handleEvent(int fd, int events, void* data)
{
    return mCallback(fd, events, data); //调用回调方法
}
```

3.4 Looper类

```
static const int EPOLL_SIZE_HINT = 8; //每个epoll实例默认的文件描述符个数
static const int EPOLL_MAX_EVENTS = 16; //轮询事件的文件描述符的个数上限
```

其中Looper类的内部定义了Request，Response，MessageEnvelope这3个结构体，关系图如下：





代码如下：

```
struct Request { //请求结构体
    int fd;
    int ident;
    int events;
    int seq;
    sp<LooperCallback> callback;
    void* data;
    void initEventItem(struct epoll_event* eventItem) const;
};

struct Response { //响应结构体
    int events;
    Request request;
};

struct MessageEnvelope { //信封结构体
    MessageEnvelope() : uptime(0) { }
    MessageEnvelope(nsecs_t uptime, const sp<MessageHandler> handler,
            const Message& message) : uptime(uptime), handler(handler), message(message) {
    }
    nsecs_t uptime;
    sp<MessageHandler> handler;
    Message message;
};
```

MessageEnvelope正如其名字，信封。MessageEnvelope里面记录正收信人(handler)，发信时间(uptime)，信件内容(message)

3.5 ALooper类

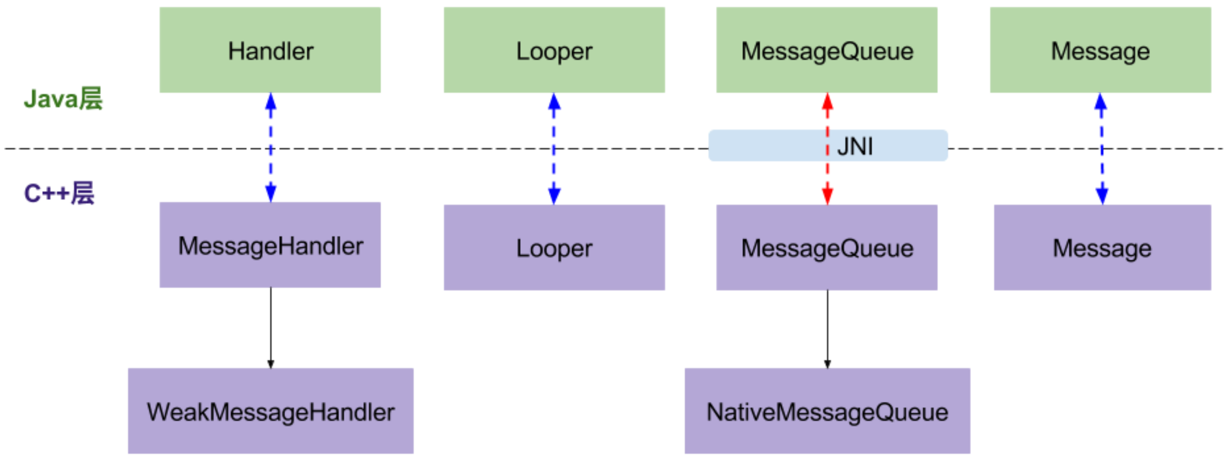
ALooper类定义在通过looper.cpp/looper.h（注意此文件是小写字母开头，与Looper.cpp不同，具体源码路径，可通过查看文章最开头的 相关源码）

```
static inline Looper* ALooper_to_Looper(ALooper* alooper) {
    return reinterpret_cast<Looper*>(alooper);
}
static inline ALooper* Looper_to_ALooper(Looper* looper) {
    return reinterpret_cast<ALooper*>(looper);
}
```

ALooper类 与前面介绍的Looper类，更多的操作是通过ALooper_to_Looper(), Looper_to_ALooper()这两个方法转换完成的，也就是说ALooper类中定义的所有方法，都是通过转换为Looper类，再执行Looper中的方法。

总结

MessageQueue通过mPtr变量保存NativeMessageQueue对象，从而使得MessageQueue成为Java层和Native层的枢纽，既能处理上层消息，也能处理native层消息；下面列举Java层与Native层的对应图：



- 其中MessageQueue在Java层和Native层通过JNI建立关联，图中以红色虚线代表这种关系；
- 而Handler/Looper/Message这些在Java层与Native层都是彼此独立的，没有任何的关联，图中只是以蓝色虚线代表这种关系。
- WeakMessageHandler继承于MessageHandler类，NativeMessageQueue继承于MessageQueue类

另外，消息处理流程是先处理Native Message，再处理Native Request，最后处理Java Message。理解了该流程，也就明白有时上层消息很少，但相应时间却比较长的真正缘由。

喜欢

0条评论

最新 最早 最热

还没有评论，沙发等你来抢

嘿嘿参北斗哇 (http://www.baidu.com/p/嘿嘿参北斗哇) 帐号管理

(http://duoshuo.com/settings/avatar/)

说点什么吧…

分享到: 发布

多说 (http://duoshuo.com)