handler

属性：

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

构造方法：

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;  
}

public Handler(Looper looper, Callback callback, boolean async) {  
 mLooper = looper;  
 mQueue = looper.mQueue;  
 mCallback = callback;  
 mAsynchronous = async;  
}

主要方法：

Message最好复用，不要new

public final Message obtainMessage()  
{  
 return Message.*obtain*(this);  
}

public final Message obtainMessage(int what)  
{  
 return Message.*obtain*(this, what);  
}

……

public final Message obtainMessage(int what, int arg1, int arg2, Object obj)  
{  
 return Message.*obtain*(this, what, arg1, arg2, obj);  
}

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

sendxxx方法最终都是走到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);  
}

public final boolean sendMessageAtFrontOfQueue(Message msg) {  
 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, 0);  
}

public final void removeMessages(int what, Object object) {  
 mQueue.removeMessages(this, what, object);  
}

public final boolean hasMessages(int what, Object object) {  
 return mQueue.hasMessages(this, what, object);  
}

public final boolean hasCallbacks(Runnable r) {  
 return mQueue.hasMessages(this, r, null);  
}

final IMessenger getIMessenger() {  
 synchronized (mQueue) {  
 if (mMessenger != null) {  
 return mMessenger;  
 }  
 mMessenger = new MessengerImpl();  
 return mMessenger;  
 }  
}  
  
private final class MessengerImpl extends IMessenger.Stub {  
 public void send(Message msg) {  
 Handler.this.sendMessage(msg);  
 }  
}

//将Runnable封装成一个msg  
private static Message getPostMessage(Runnable r) {  
 Message m = Message.*obtain*();  
 m.callback = r;  
 return m;  
}  
  
private static Message getPostMessage(Runnable r, Object token) {  
 Message m = Message.*obtain*();  
 m.obj = token;  
 m.callback = r;  
 return m;  
}

Runnable仅仅作为一个回调方法，与线程无关

private static void handleCallback(Message message) {  
 message.callback.run();  
}

Looper

属性：

static final ThreadLocal<Looper> *sThreadLocal* = new ThreadLocal<Looper>();  
private static Looper *sMainLooper*; // guarded by Looper.class  
  
final MessageQueue mQueue;  
final Thread mThread;

构造函数：

private Looper(boolean quitAllowed) {  
 mQueue = new MessageQueue(quitAllowed);  
 mThread = Thread.*currentThread*();  
}

主要方法：

public static void prepare() {  
 *prepare*(true);  
}  
  
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));  
}

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;  
 }  
  
 // This must be in a local variable, in case a UI event sets the logger  
 Printer logging = me.mLogging;  
 if (logging != null) {  
 logging.println(">>>>> Dispatching to " + msg.target + " " +  
 msg.callback + ": " + msg.what);  
 }  
  
 msg.target.dispatchMessage(msg);//分发msg  
  
 if (logging != null) {  
 logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);  
 }  
  
 // Make sure that during the course of dispatching the  
 // identity of the thread wasn't corrupted.  
 final long newIdent = Binder.*clearCallingIdentity*();  
 if (ident != newIdent) {  
 Log.*wtf*(*TAG*, "Thread identity changed from 0x"  
 + Long.*toHexString*(ident) + " to 0x"  
 + Long.*toHexString*(newIdent) + " while dispatching to "  
 + msg.target.getClass().getName() + " "  
 + msg.callback + " what=" + msg.what);  
 }  
  
 msg.recycle();//处理完成后回收到消息池  
 }  
}

public static Looper myLooper() {  
 return *sThreadLocal*.get();  
}

public static MessageQueue myQueue() {  
 return *myLooper*().mQueue;  
}

public void quit() {  
 mQueue.quit(false);  
}

public void quitSafely() {  
 mQueue.quit(true);  
}

public Thread getThread() {  
 return mThread;  
}

Looper调用流程

Looper.prepare()-->Looper.loop()…………Looper.quit()

MessageQueue

属性：

private final boolean mQuitAllowed;  
  
Message mMessages;//队列头的msg  
private final ArrayList<IdleHandler> mIdleHandlers = new ArrayList<IdleHandler>();  
private IdleHandler[] mPendingIdleHandlers;  
private boolean mQuitting;  
  
// Indicates whether next() is blocked waiting in pollOnce() with a non-zero timeout.  
private boolean mBlocked;  
  
// The next barrier token.  
// Barriers are indicated by messages with a null target whose arg1 field carries the token.  
private int mNextBarrierToken;

构造方法：

MessageQueue(boolean quitAllowed) {  
 mQuitAllowed = quitAllowed;  
 mPtr = *nativeInit*();  
}

主要方法：

private native static int nativeInit();  
private native static void nativeDestroy(int ptr);  
private native static void nativePollOnce(int ptr, int timeoutMillis);  
private native static void nativeWake(int ptr);  
private native static boolean nativeIsIdling(int ptr);  
  
*/\*\*  
 \* Callback interface for discovering when a thread is going to block  
 \* waiting for more messages.  
 \*/*public static interface IdleHandler {  
 */\*\*  
 \* Called when the message queue has run out of messages and will now  
 \* wait for more. Return true to keep your idle handler active, false  
 \* to have it removed. This may be called if there are still messages  
 \* pending in the queue, but they are all scheduled to be dispatched  
 \* after the current time.  
 \*/* boolean queueIdle();  
}

public void addIdleHandler(IdleHandler handler) {  
 if (handler == null) {  
 throw new NullPointerException("Can't add a null IdleHandler");  
 }  
 synchronized (this) {  
 mIdleHandlers.add(handler);  
 }  
}  
public void removeIdleHandler(IdleHandler handler) {  
 synchronized (this) {  
 mIdleHandlers.remove(handler);  
 }  
}

Message next() {  
 int pendingIdleHandlerCount = -1; // -1 only during first iteration  
 int nextPollTimeoutMillis = 0;  
 for (;;) {  
 if (nextPollTimeoutMillis != 0) {  
 Binder.*flushPendingCommands*();  
 }  
  
 // We can assume mPtr != 0 because the loop is obviously still running.  
 // The looper will not call this method after the loop quits.  
 *nativePollOnce*(mPtr, nextPollTimeoutMillis);  
  
 synchronized (this) {  
 // Try to retrieve the next message. Return if found.  
 final long now = SystemClock.*uptimeMillis*();  
 Message prevMsg = null;  
 Message msg = mMessages;

//消息屏障，忽略同步方法，执行异步方法（如果存在），  
 if (msg != null && msg.target == null) {  
 // Stalled by a barrier. Find the next asynchronous message in the queue.  
 do {  
 prevMsg = msg;  
 msg = msg.next;  
 } while (msg != null && !msg.isAsynchronous());  
 }

//如果有消息需要处理，先判断时间有没有到，如果没到的话设置一下阻塞时间nextPollTimeoutMillis，进入下次循环的时候会调用nativePollOnce(ptr, nextPollTimeoutMillis);阻塞；

否则把消息返回给调用者，并且设置mBlocked = false代表目前没有阻塞。

如果阻塞了有两种方式唤醒，一种是超时了，一种是被主动唤醒了。根据生产消费模式，生产者有产品的时候一般情况下会唤醒消费者。那么MessageQueue入队列的时候应该会去唤醒

if (msg != null) {  
 if (now < msg.when) {  
 // Next message is not ready. Set a timeout to wake up when it is ready.  
 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 (false) Log.*v*("MessageQueue", "Returning message: " + msg);  
 msg.markInUse();  
 return msg;  
 }  
 } else {  
 // No more messages.  
 nextPollTimeoutMillis = -1;  
 }  
  
 // Process the quit message now that all pending messages have been handled.  
 if (mQuitting) {  
 dispose();  
 return null;  
 }  
  
 // If first time idle, then get the number of idlers to run.  
 // Idle handles only run if the queue is empty or if the first message  
 // in the queue (possibly a barrier) is due to be handled in the future.  
 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);  
 }  
  
 // Run the idle handlers.  
 // We only ever reach this code block during the first iteration.  
 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*("MessageQueue", "IdleHandler threw exception", t);  
 }  
  
 if (!keep) {  
 synchronized (this) {  
 mIdleHandlers.remove(idler);  
 }  
 }  
 }  
  
 // Reset the idle handler count to 0 so we do not run them again.  
 pendingIdleHandlerCount = 0;  
  
 // While calling an idle handler, a new message could have been delivered  
 // so go back and look again for a pending message without waiting.  
 nextPollTimeoutMillis = 0;  
 }  
}  
  
void quit(boolean safe) {  
 if (!mQuitAllowed) {  
 throw new RuntimeException("Main thread not allowed to quit.");  
 }  
  
 synchronized (this) {  
 if (mQuitting) {  
 return;  
 }  
 mQuitting = true;  
  
 if (safe) {  
 removeAllFutureMessagesLocked();  
 } else {  
 removeAllMessagesLocked();  
 }  
  
 // We can assume mPtr != 0 because mQuitting was previously false.  
 *nativeWake*(mPtr);  
 }  
}

//这个方法直接在MessageQueue中插入了一个Message，并且未设置target。它的作用是插入一个消息屏障，这个屏障之后的所有同步消息都不会被执行，即使时间已经到了也不会执行。

可以通过public void removeSyncBarrier(int token)来移除这个屏障，参数是post方法的返回值。

这些方法是隐藏的或者是私有的，具体应用场景可以查看ViewRootImpl中的void scheduleTraversals()方法，它在绘图之前会插入一个消息屏障，绘制之后移除。

int enqueueSyncBarrier(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.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;  
 }  
}  
  
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;  
 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.recycle();  
  
 // If the loop is quitting then it is already awake.  
 // We can assume mPtr != 0 when mQuitting is false.  
 if (needWake && !mQuitting) {  
 *nativeWake*(mPtr);  
 }  
 }  
}

插入队列，时间为0时插入到队列头，时间不为零插入到队列尾

主要就是加入链表的时候按时间顺序从小到大排序，然后判断是否需要唤醒，如果需要唤醒则调用nativeWake(mPtr);来唤醒之前等待的线程。

boolean enqueueMessage(Message msg, long when) {  
 if (msg.isInUse()) {  
 throw new AndroidRuntimeException(msg + " This message is already in use.");  
 }  
 if (msg.target == null) {  
 throw new AndroidRuntimeException("Message must have a target.");  
 }  
  
 synchronized (this) {  
 if (mQuitting) {  
 RuntimeException e = new RuntimeException(  
 msg.target + " sending message to a Handler on a dead thread");  
 Log.*w*("MessageQueue", e.getMessage(), e);  
 return false;  
 }  
  
 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;  
}  
  
boolean hasMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h && p.what == what && (object == null || p.obj == object)) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
}  
  
boolean hasMessages(Handler h, Runnable r, Object object) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h && p.callback == r && (object == null || p.obj == object)) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
}

void removeMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.what == what  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 p.recycle();  
 p = n;  
 }  
  
 // Remove all messages after front.  
 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.recycle();  
 p.next = nn;  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
}  
  
void removeMessages(Handler h, Runnable r, Object object) {  
 if (h == null || r == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.callback == r  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 p.recycle();  
 p = n;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && n.callback == r  
 && (object == null || n.obj == object)) {  
 Message nn = n.next;  
 n.recycle();  
 p.next = nn;  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
}  
  
void removeCallbacksAndMessages(Handler h, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 p.recycle();  
 p = n;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && (object == null || n.obj == object)) {  
 Message nn = n.next;  
 n.recycle();  
 p.next = nn;  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
}

Message

主要属性：

public int what;  
public int arg1;   
public int arg2;  
public Object obj;

int flags;  
  
long when;  
  
Bundle data;  
  
Handler target;   
  
Runnable callback;   
  
Message next;  
  
private static final Object *sPoolSync* = new Object();  
private static Message *sPool*;  
private static int *sPoolSize* = 0;  
  
private static final int *MAX\_POOL\_SIZE* = 50;

构造函数：

*/\*\* Constructor (but the preferred way to get a Message is to call {****@link*** *#obtain() Message.obtain()}).  
\*/*public Message() {  
}

主要方法：

public static Message obtain() {  
 synchronized (*sPoolSync*) {  
 if (*sPool* != null) {  
 Message m = *sPool*;  
 *sPool* = m.next;  
 m.next = null;  
 *sPoolSize*--;  
 return m;  
 }  
 }  
 return new Message();  
}

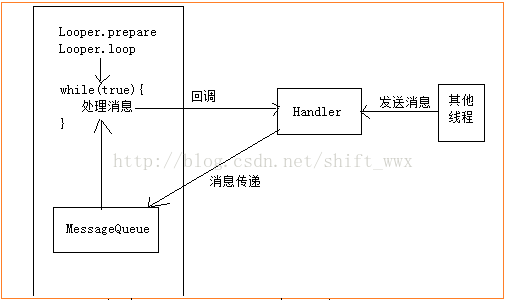
……

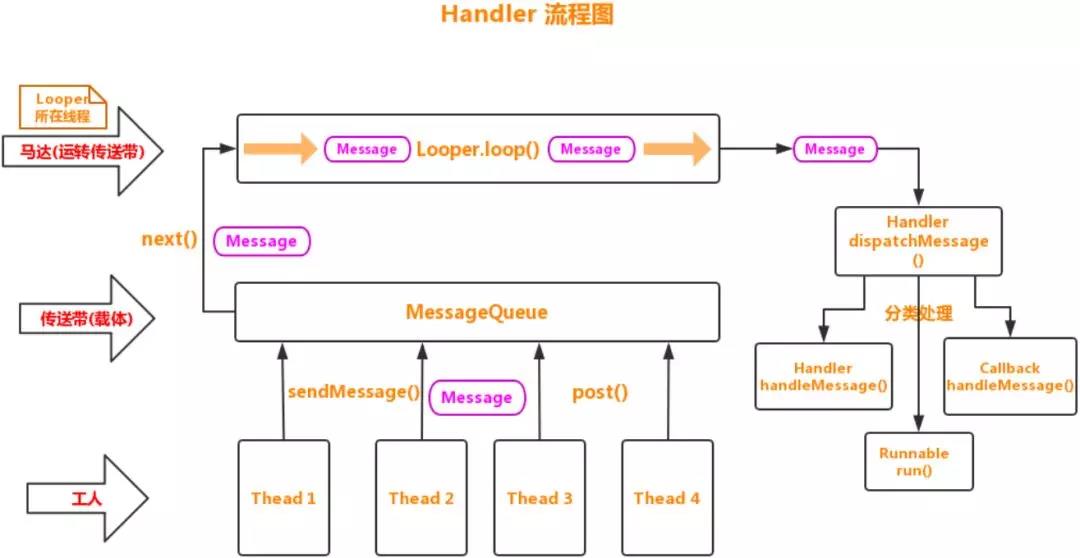
public static Message obtain(Message orig) {  
 Message m = *obtain*();  
 m.what = orig.what;  
 m.arg1 = orig.arg1;  
 m.arg2 = orig.arg2;  
 m.obj = orig.obj;  
 m.replyTo = orig.replyTo;  
 if (orig.data != null) {  
 m.data = new Bundle(orig.data);  
 }  
 m.target = orig.target;  
 m.callback = orig.callback;  
  
 return m;  
}

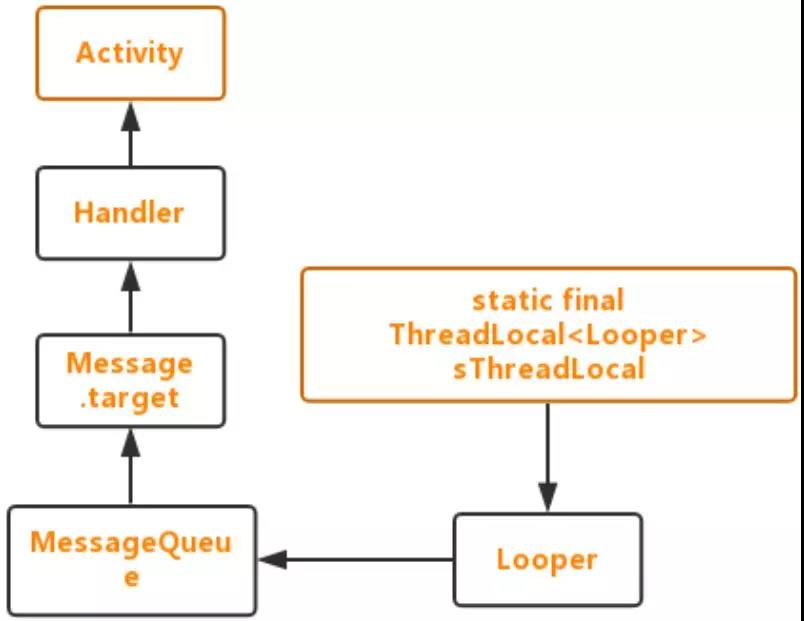
public static Message obtain(Handler h, int what,   
 int arg1, int arg2, Object obj) {  
 Message m = *obtain*();  
 m.target = h;  
 m.what = what;  
 m.arg1 = arg1;  
 m.arg2 = arg2;  
 m.obj = obj;  
  
 return m;  
}

public void recycle() {  
 clearForRecycle();  
  
 synchronized (*sPoolSync*) {  
 if (*sPoolSize* < *MAX\_POOL\_SIZE*) {  
 next = *sPool*;  
 *sPool* = this;  
 *sPoolSize*++;  
 }  
 }  
}

public void sendToTarget() {  
 target.sendMessage(this);  
}







1. Handler 的回调方法是在 Looper.loop()所调用的线程进行的；

2. Handler 的创建需要先调用 Looper.prepare() ，然后再手动调用 loop()方法开启循环；

3. App 启动时会在ActivityThread.main()方法中创建主线程的 Looper ,并开启循环，所以主线程使用 Handler 不用调用第2点的逻辑；

4. 延时消息并不会阻塞消息队列；

5. 异步消息不会马上执行，插入队列的方式跟同步消息一样，唯一的区别是当有消息屏障时，异步消息可以继续执行，同步消息则不行；

6. Callback.handleMessage() 的优先级比 Handler.handleMessage()要高\*

7. Handler.post(Runnable)传递的 Runnale 对象并不会在新的线程执行；

8. Message 的创建推荐使用 Message.obtain() 来获取，内部采用缓存消息池实现；

9. 不要在 handleMessage()中对消息进行异步处理；

10. 可以通过removeCallbacksAndMessages(null)或者静态类加弱引用的方式防止内存泄漏；

11. Looper.loop()不会造成应用卡死，里面使用了 Linux 的 epoll 机制。

Handler

<https://www.jianshu.com/p/13c8a66d3b5c>

messagequeue

<https://pqpo.me/2017/05/03/learn-messagequeue/>

Looper

<https://blog.csdn.net/lilu_leo/article/details/8145320>