Handler源码和9个常见问题的解答，这些你都掌握了吗？

2020-11-19阅读 3140

Handler是Android中的消息处理机制，是一种线程间通信的解决方案，同时你也可以理解为它天然的为我们在主线程创建一个队列，队列中的消息顺序就是我们设置的延迟的时间，如果你想在Android中实现一个队列的功能，不妨第一时间考虑一下它。

**目录：**

1. 一个线程中最多有多少个Handler，Looper，MessageQueue？
2. Looper死循环为什么不会导致应用卡死，会耗费大量资源吗？
3. 子线程的如何更新UI，比如Dialog，Toast等？系统为什么不建议子线程中更新UI?
4. 主线程如何访问网络？
5. 如何处理Handler使用不当造成的内存泄漏？
6. Handler的消息优先级，有什么应用场景？
7. 主线程的Looper何时退出？能否手动退出？
8. 如何判断当前线程是安卓主线程？
9. 正确创建Message实例的方式？

**Handler的源码和常见问题的解答**

下面来看一下[官方](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Fos%2FHandler)对其的定义：

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler it is bound to a Looper. It will deliver messages and runnables to that Looper's message queue and execute them on that Looper's thread.

大意就是Handler允许你发送Message/Runnable到线程的[消息队列](https://cloud.tencent.com/product/cmq?from=10680)(MessageQueue)中，每个Handler实例和一个线程以及那个线程的消息队列相关联。当你创建一个Handler时应该和一个Looper进行绑定（主线程默认已经创建Looper了，子线程需要自己创建Looper），它向Looper的对应的消息队列传送Message/Runnable同时在那个Looper所在线程处理对应的Message/Runnable。下面这张图就是Handler的工作流程

**Handler工作流程图**

可以看到在Thread中，Looper的这个传送带其实就一个死循环，它不断的从消息队列MessageQueue中不断的取消息，最后交给Handler.dispatchMessage进行消息的分发，而Handler.sendXXX,Handler.postXXX这些方法把消息发送到消息队列中MessageQueue，整个模式其实就是一个生产者-消费者模式，源源不断的生产消息，处理消息，没有消息时进行休眠。**MessageQueue是一个由单链表构成的优先级队列（取的都是头部，所以说是队列）**。

前面说过，当你创建一个Handler时应该和一个Looper进行绑定（绑定也可以理解为创建，主线程默认已经创建Looper了，子线程需要自己创建Looper），因此我们先来看看主线程中是如何处理的：

//ActivityThread.java

public static void main(String[] args) {

···

Looper.prepareMainLooper();

···

ActivityThread thread = new ActivityThread();

thread.attach(false, startSeq);

if (sMainThreadHandler == null) {

sMainThreadHandler = thread.getHandler();

}

if (false) {

Looper.myLooper().setMessageLogging(new

LogPrinter(Log.DEBUG, "ActivityThread"));

}

// End of event ActivityThreadMain.

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

Looper.loop();

throw new RuntimeException("Main thread loop unexpectedly exited");

}

可以看到在ActivityThread中的main方法中，我们先调用了Looper.prepareMainLooper()方法，然后获取当前线程的Handler，最后调用Looper.loop()。先来看一下Looper.prepareMainLooper()方法

//Looper.java

/\*\*

\* Initialize the current thread as a looper, marking it as an

\* application's main looper. The main looper for your application

\* is created by the Android environment, so you should never need

\* to call this function yourself. See also: {@link #prepare()}

\*/

public static void prepareMainLooper() {

prepare(false);

synchronized (Looper.class) {

if (sMainLooper != null) {

throw new IllegalStateException("The main Looper has already been prepared.");

}

sMainLooper = myLooper();

}

}

//prepare

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));

}

可以看到在Looper.prepareMainLooper()方法中创建了当前线程的Looper，同时将Looper实例存放到线程局部变量sThreadLocal(ThreadLocal)中，也就是每个线程有自己的Looper。在创建Looper的时候也创建了该线程的消息队列，可以看到prepareMainLooper会判断sMainLooper是否有值，如果调用多次，就会抛出异常，所以也就是说主线程的Looper和MessageQueue只会有一个。同理子线程中调用Looper.prepare()时，会调用prepare(true)方法，如果多次调用，也会抛出每个线程只能由一个Looper的异常，总结起来就是**每个线程中只有一个Looper和MessageQueue。**

//Looper.java

private Looper(boolean quitAllowed) {

mQueue = new MessageQueue(quitAllowed);

mThread = Thread.currentThread();

}

再来看看主线程sMainThreadHandler = thread.getHandler()，getHandler获取到的实际上就是mH这个Handler。

//ActivityThread.java

final H mH = new H();

@UnsupportedAppUsage

final Handler getHandler() {

return mH;

}

mH这个Handler是ActivityThread的内部类，通过查看handMessage方法，可以看到这个Handler处理四大组件，Application等的一些消息，比如创建Service，绑定Service的一些消息。

//ActivityThread.java

class H extends Handler {

···

public void handleMessage(Message msg) {

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

switch (msg.what) {

case BIND\_APPLICATION:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "bindApplication");

AppBindData data = (AppBindData)msg.obj;

handleBindApplication(data);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case EXIT\_APPLICATION:

if (mInitialApplication != null) {

mInitialApplication.onTerminate();

}

Looper.myLooper().quit();

break;

case RECEIVER:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "broadcastReceiveComp");

handleReceiver((ReceiverData)msg.obj);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case CREATE\_SERVICE:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, ("serviceCreate: " + String.valueOf(msg.obj)));

handleCreateService((CreateServiceData)msg.obj);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case BIND\_SERVICE:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "serviceBind");

handleBindService((BindServiceData)msg.obj);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case UNBIND\_SERVICE:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "serviceUnbind");

handleUnbindService((BindServiceData)msg.obj);

schedulePurgeIdler();

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case SERVICE\_ARGS:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, ("serviceStart: " + String.valueOf(msg.obj)));

handleServiceArgs((ServiceArgsData)msg.obj);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case STOP\_SERVICE:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "serviceStop");

handleStopService((IBinder)msg.obj);

schedulePurgeIdler();

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

···

case APPLICATION\_INFO\_CHANGED:

mUpdatingSystemConfig = true;

try {

handleApplicationInfoChanged((ApplicationInfo) msg.obj);

} finally {

mUpdatingSystemConfig = false;

}

break;

case RUN\_ISOLATED\_ENTRY\_POINT:

handleRunIsolatedEntryPoint((String) ((SomeArgs) msg.obj).arg1,

(String[]) ((SomeArgs) msg.obj).arg2);

break;

case EXECUTE\_TRANSACTION:

final ClientTransaction transaction = (ClientTransaction) msg.obj;

mTransactionExecutor.execute(transaction);

if (isSystem()) {

// Client transactions inside system process are recycled on the client side

// instead of ClientLifecycleManager to avoid being cleared before this

// message is handled.

transaction.recycle();

}

// TODO(lifecycler): Recycle locally scheduled transactions.

break;

case RELAUNCH\_ACTIVITY:

handleRelaunchActivityLocally((IBinder) msg.obj);

break;

case PURGE\_RESOURCES:

schedulePurgeIdler();

break;

}

Object obj = msg.obj;

if (obj instanceof SomeArgs) {

((SomeArgs) obj).recycle();

}

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

}

}

最后我们查看Looper.loop()方法

//Looper.java

public static void loop() {

//获取ThreadLocal中的Looper

final Looper me = myLooper();

···

final MessageQueue queue = me.mQueue;

···

for (;;) { //死循环

//获取消息

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

if (msg == null) {

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

return;

}

···

msg.target.dispatchMessage(msg);

···

//回收复用

msg.recycleUnchecked();

}

}

在loop方法中是一个死循环，在这里从消息队列中不断的获取消息queue.next()，然后通过Handler(msg.target)进行消息的分发，**其实并没有什么具体的绑定，因为Handler在每个线程中对应只有一个Looper和消息队列MessageQueue，自然要靠它来处理，也就是是调用Looper.loop()方法。在Looper.loop()的死循环中不断的取消息，最后回收复用**。

**这里要强调一下Message中的参数target(Handler)，正是这个变量，每个Message才能找到对应的Handler进行消息分发，让多个Handler同时工作**。

再来看看子线程中是如何处理的，首先在子线程中创建一个Handler并发送Runnable

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_three);

new Thread(new Runnable() {

@Override

public void run() {

new Handler().post(new Runnable() {

@Override

public void run() {

Toast.makeText(HandlerActivity.this,"toast",Toast.LENGTH\_LONG).show();

}

});

}

}).start();

}

运行后可以看到错误日志，可以看到提示我们需要在子线程中调用Looper.prepare()方法，实际上就是要创建一个Looper和你的Handler进行“关联”。

--------- beginning of crash

2020-11-09 15:51:03.938 21122-21181/com.jackie.testdialog E/AndroidRuntime: FATAL EXCEPTION: Thread-2

Process: com.jackie.testdialog, PID: 21122

java.lang.RuntimeException: Can't create handler inside thread Thread[Thread-2,5,main] that has not called Looper.prepare()

at android.os.Handler.<init>(Handler.java:207)

at android.os.Handler.<init>(Handler.java:119)

at com.jackie.testdialog.HandlerActivity$1.run(HandlerActivity.java:31)

at java.lang.Thread.run(Thread.java:919)

添加Looper.prepare()创建Looper，同时调用Looper.loop()方法开始处理消息。

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_three);

new Thread(new Runnable() {

@Override

public void run() {

//创建Looper，MessageQueue

Looper.prepare();

new Handler().post(new Runnable() {

@Override

public void run() {

Toast.makeText(HandlerActivity.this,"toast",Toast.LENGTH\_LONG).show();

}

});

//开始处理消息

Looper.loop();

}

}).start();

}

这里需要注意在所有事情处理完成后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于循环等待的状态，因此不需要的时候终止Looper，调用Looper.myLooper().quit()。

看完上面的代码可能你会有一个疑问，在子线程中更新UI（进行Toast）不会有问题吗，我们Android不是不允许在子线程更新UI吗，实际上并不是这样的，在ViewRootImpl中的checkThread方法会校验mThread != Thread.currentThread(),mThread的初始化是在ViewRootImpl的的构造器中，**也就是说一个创建ViewRootImpl线程必须和调用checkThread所在的线程一致，UI的更新并非只能在主线程才能进行**。

void checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

这里需要引入一些概念，Window是Android中的窗口，每个Activity和Dialog，Toast分别对应一个具体的Window，Window是一个抽象的概念，**每一个Window都对应着一个View和一个ViewRootImpl，Window和View通过ViewRootImpl来建立联系，因此，它是以View的形式存在的**。我们来看一下Toast中的ViewRootImpl的创建过程，调用toast的show方法最终会调用到其handleShow方法

//Toast.java

public void handleShow(IBinder windowToken) {

···

if (mView != mNextView) {

// Since the notification manager service cancels the token right

// after it notifies us to cancel the toast there is an inherent

// race and we may attempt to add a window after the token has been

// invalidated. Let us hedge against that.

try {

mWM.addView(mView, mParams); //进行ViewRootImpl的创建

trySendAccessibilityEvent();

} catch (WindowManager.BadTokenException e) {

/\* ignore \*/

}

}

}

这个mWM（WindowManager）的最终实现者是WindowManagerGlobal，其的addView方法中会创建ViewRootImpl，然后进行root.setView(view, wparams, panelParentView)，通过ViewRootImpl来更新界面并完成Window的添加过程。

//WindowManagerGlobal.java

root = new ViewRootImpl(view.getContext(), display); //创建ViewRootImpl

view.setLayoutParams(wparams);

mViews.add(view);

mRoots.add(root);

mParams.add(wparams);

// do this last because it fires off messages to start doing things

try {

//ViewRootImpl

root.setView(view, wparams, panelParentView);

} catch (RuntimeException e) {

// BadTokenException or InvalidDisplayException, clean up.

if (index >= 0) {

removeViewLocked(index, true);

}

throw e;

}

}

setView内部会通过requestLayout来完成异步刷新请求，同时也会调用checkThread方法来检验线程的合法性。

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

因此，我们的ViewRootImpl的创建是在子线程，所以mThread的值也是子线程，同时我们的更新也是在子线程，所以不会产生异常。同理下面的代码也可以验证这个情况

//子线程中调用

public void showDialog(){

new Thread(new Runnable() {

@Override

public void run() {

//创建Looper，MessageQueue

Looper.prepare();

new Handler().post(new Runnable() {

@Override

public void run() {

builder = new AlertDialog.Builder(HandlerActivity.this);

builder.setTitle("jackie");

alertDialog = builder.create();

alertDialog.show();

alertDialog.hide();

}

});

//开始处理消息

Looper.loop();

}

}).start();

}

在子线程中调用showDialog方法，先调用alertDialog.show()方法，再调用alertDialog.hide()方法，hide方法只是将Dialog隐藏，并没有做其他任何操作(没有移除Window)，然后再在主线程调用alertDialog.show()；便会抛出Only the original thread that created a view hierarchy can touch its views异常了。

2020-11-09 18:35:39.874 24819-24819/com.jackie.testdialog E/AndroidRuntime: FATAL EXCEPTION: main

Process: com.jackie.testdialog, PID: 24819

android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.

at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:8191)

at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:1420)

at android.view.View.requestLayout(View.java:24454)

at android.view.View.setFlags(View.java:15187)

at android.view.View.setVisibility(View.java:10836)

at android.app.Dialog.show(Dialog.java:307)

at com.jackie.testdialog.HandlerActivity$2.onClick(HandlerActivity.java:41)

at android.view.View.performClick(View.java:7125)

at android.view.View.performClickInternal(View.java:7102)

**所以在线程中更新UI的重点是创建它的ViewRootImpl和checkThread所在的线程是否一致**。

**如何在主线程中访问网络**

在网络请求之前添加如下代码

StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitNetwork().build();

StrictMode.setThreadPolicy(policy);

StrictMode（严苛模式）Android2.3引入，用于检测两大问题：ThreadPolicy（线程策略）和VmPolicy(VM策略)，这里把严苛模式的网络检测关了，就可以在主线程中执行网络操作了，一般是不建议这么做的。关于严苛模式可以查看[这里](https://links.jianshu.com/go?to=https%3A%2F%2Fericchows.github.io%2FAndroid-StrictMode-Analysis%2F)。

**系统为什么不建议在子线程中访问UI？**

这是因为 Android 的UI控件不是线程安全的，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态，那么为什么系统不对UI控件的访问加上锁机制呢？缺点有两个：

1. 首先加上锁机制会让UI访问的逻辑变得复杂
2. 锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。

所以最简单且高效的方法就是采用单线程模型来处理UI操作。（安卓开发艺术探索）

**子线程如何通知主线程更新UI（都是通过Handle发送消息到主线程操作UI的）**

1. 主线程中定义 Handler，子线程通过 mHandler 发送消息，主线程 Handler 的 handleMessage 更新UI。
2. 用 Activity 对象的 runOnUiThread 方法。
3. 创建 Handler，传入 getMainLooper。
4. View.post(Runnable r) 。

**Looper死循环为什么不会导致应用卡死，会耗费大量资源吗？**

从前面的主线程、子线程的分析可以看出，Looper会在线程中不断的检索消息，如果是子线程的Looper死循环，一旦任务完成，用户应该手动退出，而不是让其一直休眠等待。（引用自Gityuan）线程其实就是一段可执行的代码，当可执行的代码执行完成后，线程的生命周期便该终止了，线程退出。而对于主线程，我们是绝不希望会被运行一段时间，自己就退出，那么如何保证能一直存活呢？简单做法就是可执行代码是能一直执行下去的，死循环便能保证不会被退出，例如，binder 线程也是采用死循环的方法，通过循环方式不同与 Binder 驱动进行读写操作，当然并非简单地死循环，无消息时会休眠。**Android是基于消息处理机制的，用户的行为都在这个Looper循环中，我们在休眠时点击屏幕，便唤醒主线程继续进行工作**。

主线程的死循环一直运行是不是特别消耗 CPU 资源呢？ 其实不然，这里就涉及到 Linux pipe/epoll机制，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，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。

**主线程的Looper何时退出**

在App退出时，ActivityThread中的mH（Handler）收到消息后，执行退出。

//ActivityThread.java

case EXIT\_APPLICATION:

if (mInitialApplication != null) {

mInitialApplication.onTerminate();

}

Looper.myLooper().quit();

break;

如果你尝试手动退出主线程Looper，便会抛出如下异常

Caused by: java.lang.IllegalStateException: Main thread not allowed to quit.

at android.os.MessageQueue.quit(MessageQueue.java:428)

at android.os.Looper.quit(Looper.java:354)

at com.jackie.testdialog.Test2Activity.onCreate(Test2Activity.java:29)

at android.app.Activity.performCreate(Activity.java:7802)

at android.app.Activity.performCreate(Activity.java:7791)

at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1299)

at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3245)

at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3409)

at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:83)

at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.java:135)

at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:95)

at android.app.ActivityThread$H.handleMessage(ActivityThread.java:2016)

at android.os.Handler.dispatchMessage(Handler.java:107)

at android.os.Looper.loop(Looper.java:214)

at android.app.ActivityThread.main(ActivityThread.java:7356)

at java.lang.reflect.Method.invoke(Native Method)

at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:492)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:930)

为什么不允许退出呢，**因为主线程不允许退出，一旦退出就意味着程序挂了，退出也不应该用这种方式退出**。

**Handler的消息处理顺序**

在Looper执行消息循环loop()时会执行下面这行代码，msg.targe就是这个Handler对象

msg.target.dispatchMessage(msg);

我们来看看dispatchMessage的源码

public void dispatchMessage(@NonNull Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

//如果 callback 处理了该 msg 并且返回 true， 就不会再回调 handleMessage

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

1. 如果Message这个对象有CallBack回调的话，这个CallBack实际上是个Runnable，就只执行这个回调，然后就结束了，创建该Message的CallBack代码如下： Message msgCallBack = Message.obtain(handler, new Runnable() { @Override public void run() { } }); 而handleCallback方法中调用的是Runnable的run方法。 private static void handleCallback(Message message) { message.callback.run(); }
2. 如果Message对象没有CallBack回调，进入else分支判断Handler的CallBack是否为空，不为空执行CallBack的handleMessage方法，然后return，构建Handler的CallBack代码如下：
3. Handler.Callback callback = new Handler.Callback() { @Override public boolean handleMessage(@NonNull Message msg) { //retrun true，就不执行下面的逻辑了，可以用于做优先级的处理 return false; } };
4. 最后才调用到Handler的handleMessage()函数，也就是我们经常去重写的函数，在该方法中做消息的处理。

**使用场景**

可以看到Handler.Callback 有**优先处理消息的权利** ，当一条消息被 Callback 处理**并拦截（返回 true）**，那么 Handler 的 handleMessage(msg) 方法就不会被调用了；如果 Callback 处理了消息，但是并没有拦截，那么就意味着**一个消息可以同时被 Callback 以及 Handler 处理**。我们可以**利用CallBack这个拦截来拦截Handler的消息**。

场景：Hook [ActivityThread.mH](https://links.jianshu.com/go?to=http%3A%2F%2Factivitythread.mh%2F) ， 在 ActivityThread 中有个成员变量 mH ，它是个 Handler，又是个极其重要的类，几乎所有的插件化框架都使用了这个方法。

**Handler.post(Runnable r)方法的执行逻辑**

我们需要分析平时常用的Handler.post(Runnable r)方法是如何执行的，是否新创建了一个线程了呢，实际上并没有，这个Runnable对象只是被调用了它的run方法，根本并没有启动一个线程，源码如下：

//Handler.java

public final boolean post(@NonNull Runnable r) {

return sendMessageDelayed(getPostMessage(r), 0);

}

private static Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

最终该Runnable对象被包装成一个Message对象，也就是这个Runnable对象就是该Message的CallBack对象了，有优先执行的权利了。

**Handler是如何进行线程切换的**

原理很简单，线程间是共享资源的，子线程通过handler.sendXXX，handler.postXXX等方法发送消息，然后通过Looper.loop()在消息队列中不断的循环检索消息，最后交给handle.dispatchMessage方法进行消息的分发处理。

**如何处理Handler使用不当造成的内存泄漏？**

1. 有延时消息，在界面关闭后及时移除Message/Runnable，调用handler.removeCallbacksAndMessages(null)
2. 内部类导致的内存泄漏改为静态内部类，并对上下文或者Activity/Fragment使用弱引用。

具体内存泄漏的分析和解决可以参考这篇[文章](https://www.jianshu.com/p/daffbb9ddc17)。同时还有一个很关键的点，如果有个延时消息，当界面关闭时，该Handler中的消息还没有处理完毕，那么最终这个消息是怎么处理的？经过测试，比如我打开界面后延迟10s发送消息，关闭界面，最终在Handler（匿名内部类创建的）的handMessage方法中还是会收到消息（打印日志）。因为会有一条MessageQueue -> Message -> Handler -> Activity的引用链，所以Handler不会被销毁，Activity也不会被销毁。

**正确创建Message实例**

1. 通过 Message 的静态方法 Message.obtain() 获取；
2. 通过 Handler 的公有方法 handler.obtainMessage()

所有的消息会被回收，放入sPool中，使用享元设计模式。

**结尾**

最后为了帮助大家深刻理解Handler相关知识点的原理以及面试相关知识，这里还为大家整理了[**Android开发相关源码精编解析**](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2FXiaxiao-1024%2FAndroid-P7-%2Fblob%2Fmaster%2FAndroid%25E5%25BC%2580%25E5%258F%2591%25E8%25BF%2598%25E4%25B8%258D%25E4%25BC%259A%25E8%25BF%2599%25E4%25BA%259B%25EF%25BC%259F%25E5%25A6%2582%25E4%25BD%2595%25E9%259D%25A2%25E8%25AF%2595%25E6%258B%25BF%25E9%25AB%2598%25E8%2596%25AA%25EF%25BC%2581.md)：

**深入解析 Handler 源码解析**

* 发送消息
* 消息入队
* 消息循环
* 消息遍历
* 消息的处理
* 同步屏障机制
* 阻塞唤醒机制

**还有Handler相关面试题解析帮助熟练掌握Handler知识：**

最后为了帮助大家深刻理解Android相关知识点的原理以及面试相关知识，这里放上相关的我搜集整理的24套腾讯、字节跳动、阿里、百度**2019-2020BAT 面试真题解析**，我把大厂面试中**常被问到的技术点**整理成了视频和PDF（实际上比预期多花了不少精力），包知识脉络 + 诸多细节。

还有 高级架构技术进阶脑图 帮助大家学习提升进阶，也节省大家在网上搜索资料的时间来学习，也可以分享给身边好友一起学习。

我是之后按着这份资料复习面试题，后来再去面试就过了，帮助还是很大的~

**以上内容均放在了开源项目：【**[**github**](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2FXiaxiao-1024%2FAndroid-P7-%2Fblob%2Fmaster%2FAndroid%25E5%25BC%2580%25E5%258F%2591%25E8%25BF%2598%25E4%25B8%258D%25E4%25BC%259A%25E8%25BF%2599%25E4%25BA%259B%25EF%25BC%259F%25E5%25A6%2582%25E4%25BD%2595%25E9%259D%25A2%25E8%25AF%2595%25E6%258B%25BF%25E9%25AB%2598%25E8%2596%25AA%25EF%25BC%2581.md)**】 中已收录，里面包含不同方向的自学Android路线、面试题集合/面经、及系列技术文章等，资源持续更新中...**