**Chromium和NW.js消息循环总结**

目 录

[一、类图关系 1](#_Toc418064737)

[二、简单调用关系 2](#_Toc418064738)

[三、根据TYPE\_DEFAULT分析过程 3](#_Toc418064739)

[3.1 MessageLoop的创建 3](#_Toc418064740)

[3.2、Thread A向MessageLoop投递任务 5](#_Toc418064741)

[3.3、Thread A 开始消息循环 8](#_Toc418064742)

[3.4、Thread B 向MessageLoop投递任务 16](#_Toc418064743)

[3.5 整体流程图 16](#_Toc418064744)

[四、Android Chromium消息循环分析 17](#_Toc418064745)

[4.1 Android消息循环基础知识 17](#_Toc418064746)

[4.2 Thread A创建messageloop 19](#_Toc418064747)

[4.3 Thread A开始消息循环 20](#_Toc418064748)

[4.4 Thread A 向messageloop投递任务 21](#_Toc418064749)

[4.4 Thread A异步处理消息 22](#_Toc418064750)

[4.6 DoRunLoopOnce() 函数分析 23](#_Toc418064751)

[五、NW.js中Render进程消息循环分析 27](#_Toc418064752)

[5.1 Node.js的消息循环 27](#_Toc418064753)

[5.2 Node.js与chromium消息循环结合分析 28](#_Toc418064754)

[5.3 Android与Node.js结合的猜想 33](#_Toc418064755)

# 一、类图关系

在chromium和NW.js的简单类图关系如图1.1所示：

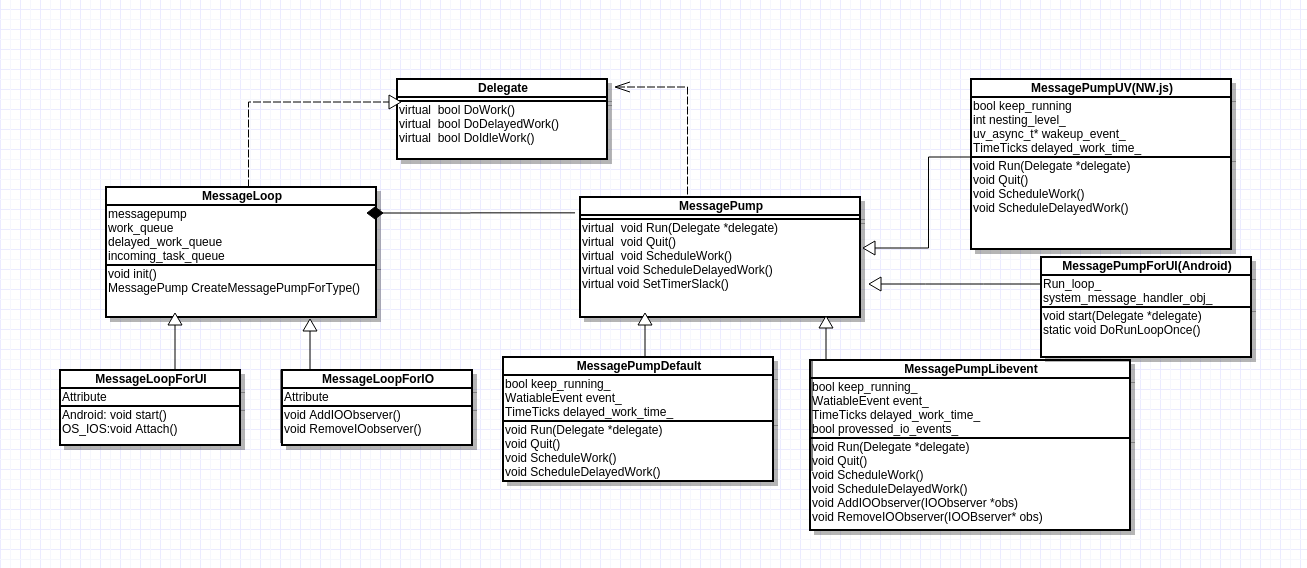


图 1.1

在图中可以看出，各个类之间的继承依赖关系，其中delegate是一个借口类，由messageloop类来实现，有两个类MessageLoopForUI和MessageLoopForIO继承于messageloop类，用来实现一些UI和IO的消息循环。

每一个messageloop对象里面会有一个messagePump对象，其中messagePump对象中的Run()函数需要传入delegate对象，也就是messageloop对象。messagePump是为了隔离各个平台来实现的。其中主要分析的类有四个，分别是messagePumpDefault、messagePumpLibevent、messagePumpUV和messagePumpForUI。messagePumpForUV就是NW.js针对node.js实现的messagepump，messagePumpUI就是针对Android系统实现的messagePump，从类图中可以看出，这四个类都是继承messagePump类。

# 二、简单调用关系

Messageloop和messagePump的调用关系很复杂，但是可以先简单的描述一下调用关系，让大家对调用关系有一个简单的了解，具体的详细说明会在下文中描述。简单的调用关系如图2.1所示：

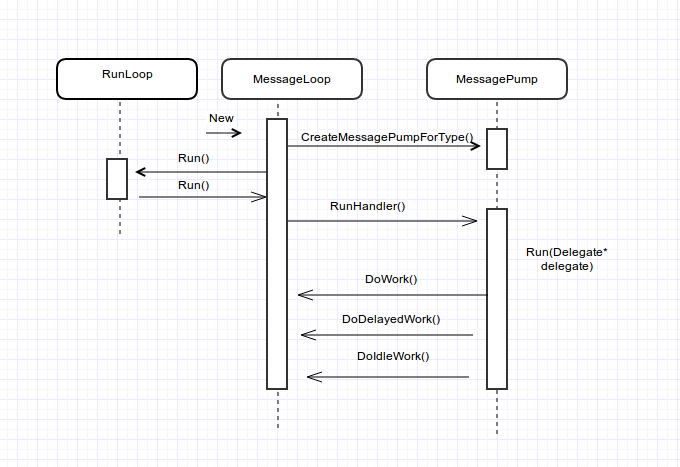


图 2.1

从图中可以看出，首先在创建messageloop对象的时候，会根据messageloop对象的类型创建一个messagePump对象，当messageloop启动循环的时候，会调用自身的RunHandler()函数，然后此函数调用messagePump对象的Run()函数，并且把自身传入当做参数传入，messagePump的Run()函数会调用messageloop对象的DoWork()、DoDelayedWork()和DoIdleWork()函数，三个函数的作用就是调用messageloop对象中任务队列里面的任务，然后执行。

# 三、根据TYPE\_DEFAULT分析过程

## MessageLoop的创建

（1）messageloop的实现主要是在message\_loop.h和message\_loop.cc文件中（全局搜索一下就可以找到）。MessageLoop默认的构造函数中，默认传入的参数就是TYPE\_DEFAULT。构造函数可以通过一个TYPE构造，也可以通过传入对应的MessagePump构造。以下从传入TYPE开始分析。

messageloop构造函数代码如图3.1所示:

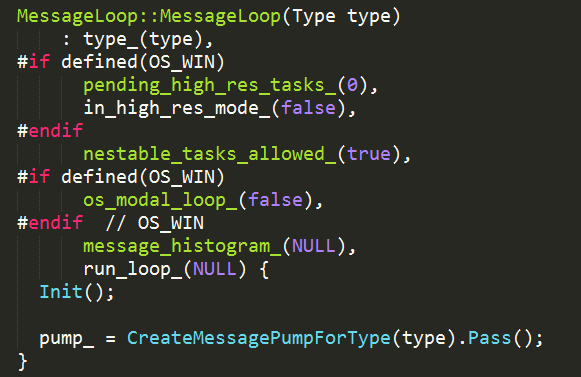


图 3.1

从中可以看出，构造函数主要是做了两件事：一、运行Init()函数，初始化一些条件，包括任务进入队列等。二、构造对应的messagePump。

Init函数的代码截图如3.2所示：

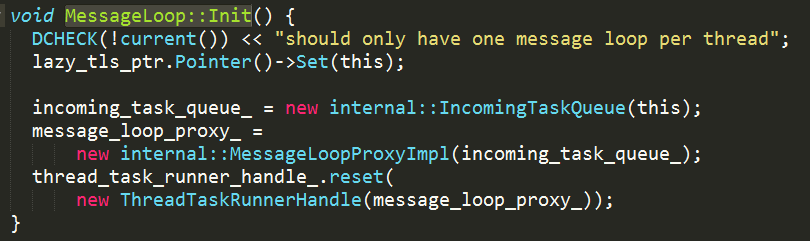


图 3.2

从代码中可以看出，他是创建了一个任务进入队列、messageloop代理和一个任务运行handle。任务进入队列的实现是在incoming\_task\_queue.h文件和incoming\_task\_queue.cc文件中实现，在以后会具体讨论，只要知道，任务是先进入到这个income\_queue，然后在加载到work\_queue，在work\_queue中在执行就可以了。其中那个代理的作用就是以后和messagel交互不是直接和messageloop交互，而是通过代理。Task\_runner\_handle的作用就是执行任务。后两者的作用和分析消息循环联系不是很紧密，所以就没有仔细分析。

（2）当执行Init()函数，初始化环境之后，就会调用CreateMessagePumpForType()函数，创建一个messagepump。CreateMessagePumpForType()函数的分析分两部分进行分析，第一部分的代码如图3.3所示：

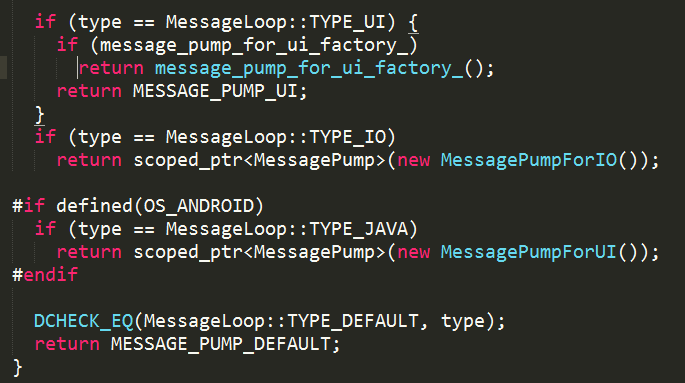


图 3.3

可以看出，是根据不同type，来返回不同的messagepump对象，其中如果定义的是TYPE\_DEFAULT,则会返回MESSAGE\_PUMP\_DEFAULT。在这里已经有了区分的创建负责Android的Messagepump还是负责Node.js的MessagePump，后两者的分析会在下面的文章中讲到。但是MESSAGE\_PUMP\_DEFAULT具体代表的是什么，它利用的是C++的typedef函数。具体的代码截图如图3.4所示：

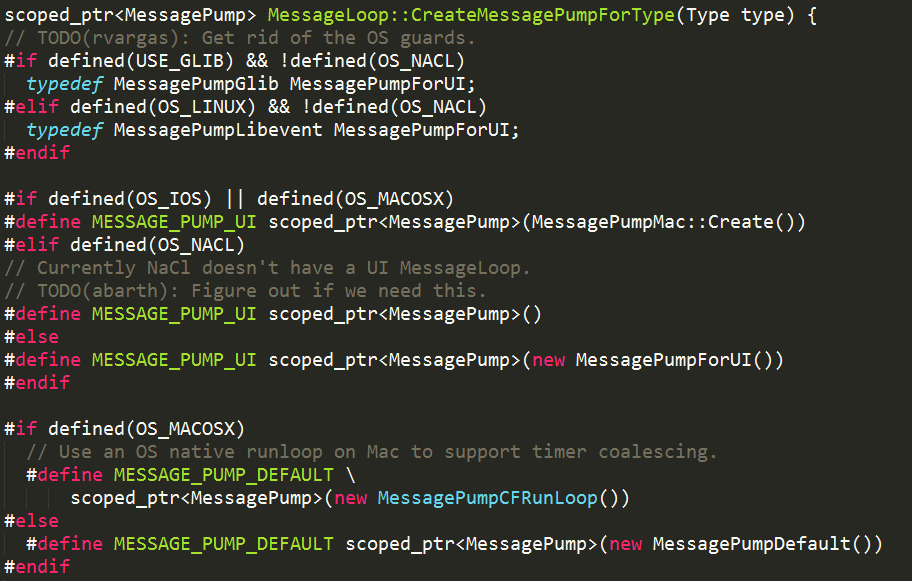


图 3.4

从代码的最后一行可以看出，最后返回的是一个MessagePumpDefault对象。

（3） Messagepumpdefault的实现是在messagepumpdefault.h和messagepumpdefault.cc文件中，它继承于messagepump类。其构造函数很简单，只是给自己的变量赋值。其具体实现代码如图3.5所示：

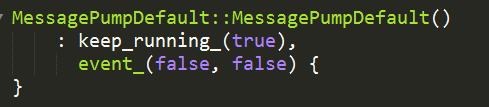


图 3.5

到目前为止，一个线程的MessageLoop便创建完成。

## 3.2、Thread A向MessageLoop投递任务

（1）现在假设Thread A的messageloop就是刚才创建的messageloop，那么当自己线程有任务需要调度的时候，就会获取到自身的messageloop，然后调用其PostTask()函数。PostTask函数的代码截图如图3.6所示：

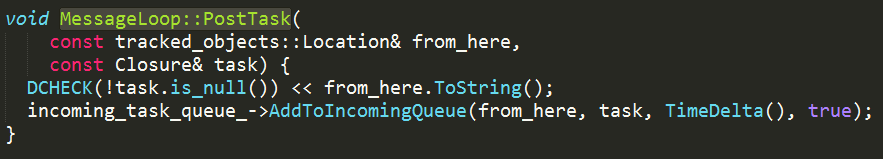


图 3.6

从代码中可以看出是调用的incoming\_task\_queue\_对象的AddToIncomingQueue()函数。其作用就是把投递的任务，添加到incoming\_task\_queue\_中去，注意，此时添加的时候就会给incoming\_task\_queue\_加锁。

（2）前面已经讲过任务进入队列的实现是在incoming\_task\_queue.h文件和incoming\_task\_queue.cc文件中实现。所以可以在文件中找到AddToIncomingQueue()函数的代码，其代码截图如图3.7所示：

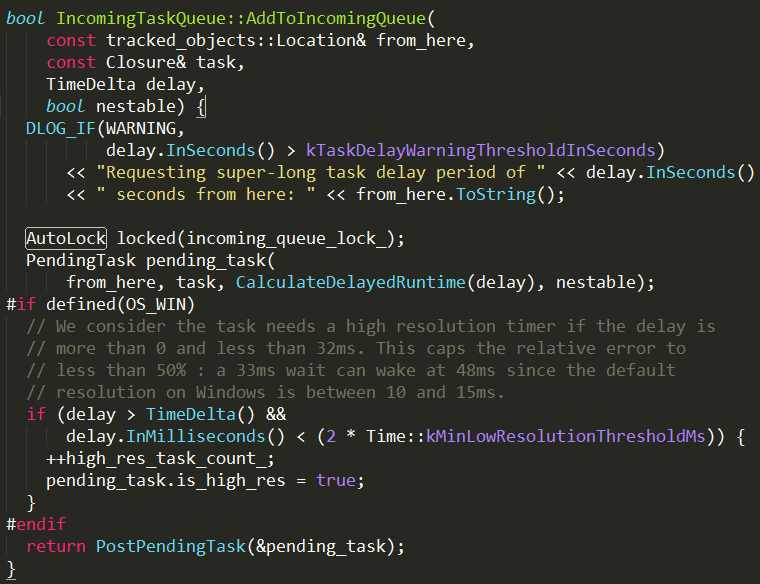


图 3.7

从代码中可以看出，首先给incoming\_task\_queue加锁，然后根据传入的task，又创建了一个pending\_task，对传入的task的延迟时间和嵌套进行了封装。最后调用的是PostPendingTask()函数，将封装好的task放到incoming\_task\_queue的最后。

（3）对于PostPendingTask()函数主要有两个作用：一、将任务放入到incoming\_task\_queue判断是不是执行ScheduleWork函数。其代码截图如图3.8所示：

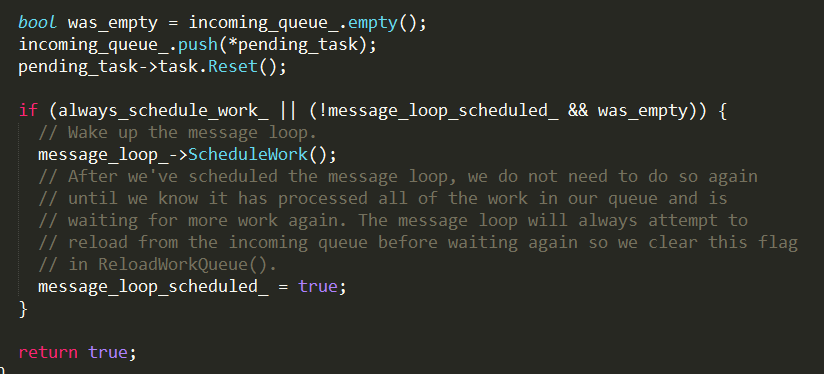


图 3.8

从代码中可以看出，在将任务放入到incoming\_queue\_之后就会判断是否调用message\_loop\_的ScheduleWork()函数，其中要是Android系统，每次都会调用。这个在以后会提到。在Linux系统中，之后当message\_loop\_scheduled\_是false，并且incoming\_queue\_是空的时候才会调用。

（4）下面分析一个messageloop的ScheduleWork()函数的作用。当messageloop开始运行之后，就会执行，所有的任务，那么当任务执行完之后，就不会占用cup资源，然后messageloop就会进入阻塞状态，那么当又有新的任务的时候，必然需要一个ScheduleWork()函数,告诉messageloop又有新的任务了，又要开始进行调度了。Messageloop的ScheduleWork()函数的代码截图如图3.9所示：

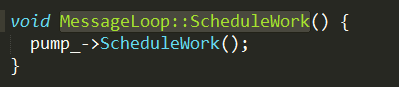


图 3.9

很简单，就是调用MessagePump的ScheduleWork()函数，其函数截图如图3.10所示：

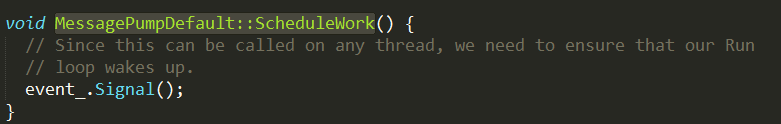


图 3.10

这个也很简单，只是将event\_的信号唤醒就可以了。至于什么时候开始睡眠的，在以后就会讲到。整体的流程图如图3.11所示：

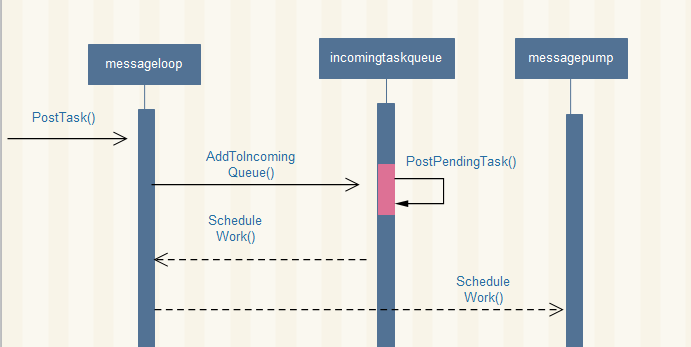


图 3.11

## 3.3、Thread A 开始消息循环

（1）Thread A通过调用messageloop的Run()函数来开始messageloop的执行。其函数截图如图3.12所示：

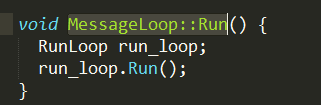


图 3.12

Messageloop是根据自身，创建了一个RunLoop对象，然后调用RunLoop对象的Run函数来开始运行。RunLoop对象也是一个messageloop对象的代理，操作的时候是操作messageloop对象对应的RunLoop对象。至于原理，现在还没有分析清楚，但是RunLoop对象的Run函数，就会调用messageloop对象的RunHandler()函数，所以就从RunHandler()函数分析，其代码截图如图 3.13所示：

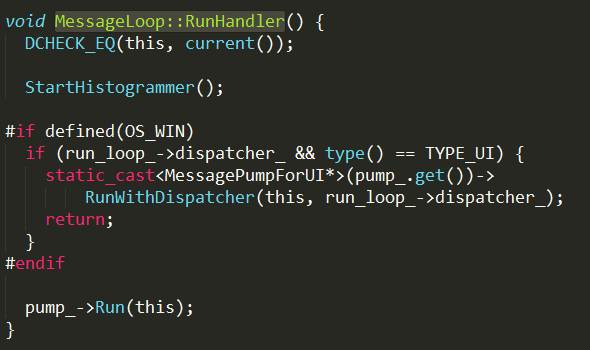


图 3.13

从代码中可以看出，是先开始调用StartHistogrammer()函数，作用就是用来记录一些历史消息，具体的先不做分析。然后可以看出是调用的messageloop对象中的messagepump对象的Run()函数，然后把自身当做参数传入。

（2）分析messagepump对象的Run函数，这个函数很重要，主要分为两部分进行分析，第一部分就是怎么样进行messageloop的消息循环，第二部分就是怎样进入睡眠状态。具体代码截图如3.14所示：



图 3.14

其中传入的delegate就是当前的messageloop对象，因为messageloop对象是实现的delegate结构，所以子啊调用DoWork() DoDelayedWOrk() DoDelayedWork()函数的时候执行的是messageloop对应的函数。

首先执行的是messageloop的DoWork()函数，DoWork()函数的部分代码截图如图3.15所示：

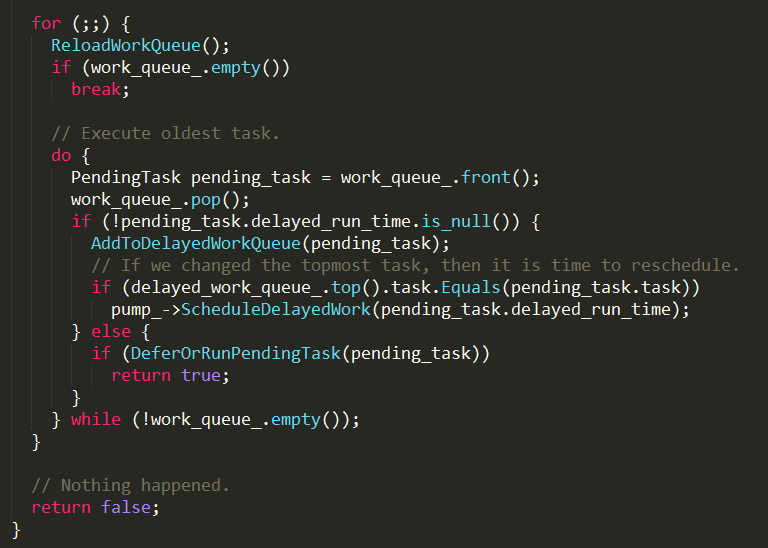


图 3.15

首先，他是一个for语句的死循环，里面先执行ReloadWorkQueue()函数，其ReloadWorkQueue()函数就是将incoming\_task\_queue\_中的任务加载到work\_queue中，里面有一个技巧就是，只有当当前的work\_queue为空的时候才加载，否则不加载。ReloadWorkQueue()函数的截图如图3.16所示：

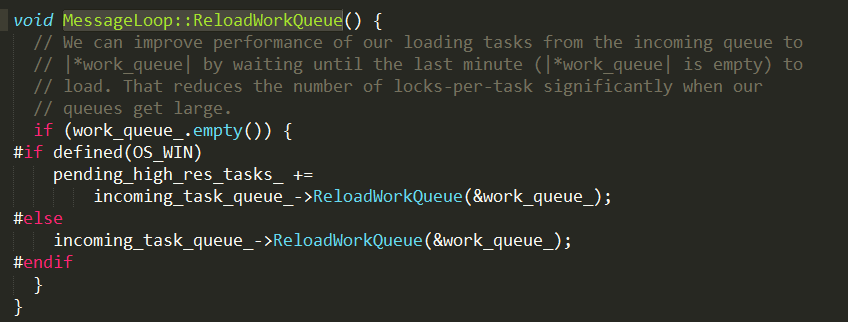


图 3.16

从代码中可以看出，调用的是incoming\_task\_queue\_对象的ReloadWorkQueue()函数，并且把messageloop对象下的work\_queue\_的引用传入进去。incoming\_task\_queue\_对象的ReloadWorkQueue()函数的代码截图如图3.17所示：

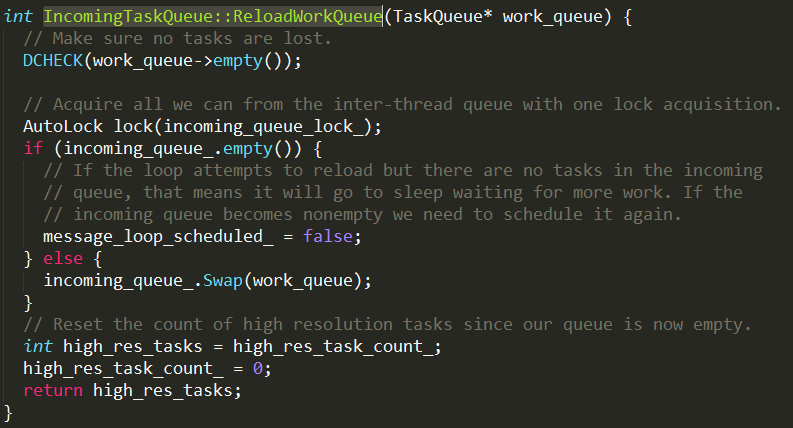


图 3.17

注意这时候incoming\_queue\_是需要加锁的。

在ReloadWorkQueue()函数执行后，判断work\_queue是不是为空，如果是，则返回false，不是的话，执行do while语句。首先取出work\_queue\_的首任务，然后判断是不是延迟任务，如果是延迟任务的话，那么通过调用AddToDelayedWorkQueue(pending\_task)函数将任务放入到延迟任务队列，然后判断，如果延迟任务队列以前为空，现在只有一个任务的话，就调用messagepump对象的ScheduleDelayedWork(pending\_task.delayed\_run\_time)函数。messagepump对象的ScheduleDelayedWork(pending\_task.delayed\_run\_time)函数的主要作用就是通知messagepump对象什么时候执行延迟队列中的首任务，如果时间没有到并且messageloop中没有任务的时候就可以睡眠。其实现代码如图3.18所示：

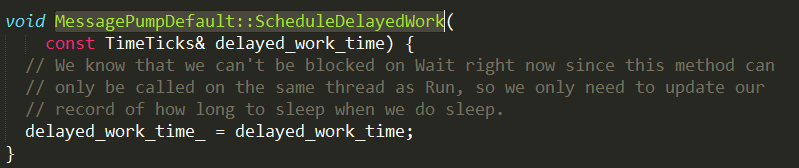


图 3.18

主要就是设置一个delayed\_work\_time\_参数，这个参数用来表示最近的延迟任务需要等待多长时间进行调度，在控制线程睡眠的时候会用到。

开始不是延迟任务的话，那么执行DeferOrRunPendingTask(pending\_task)函数，来判断现在的任务是需要执行还是需要延迟以后执行，DeferOrRunPendingTask(pending\_task)函数的代码截图如图3.19所示：

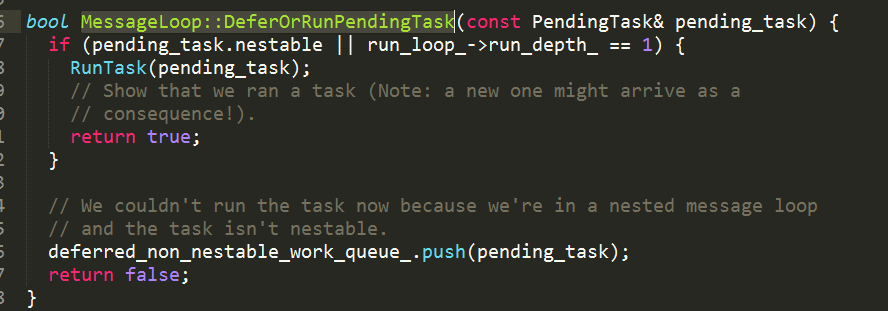


图3.19

首先将一个嵌套任务的概念，就是说，在执行一个消息循环的时候，可能会执行到另一个消息循环中去。也就是说消息循环中嵌套消息循环，并且之后嵌套的消息循环执行完之后，这个父消息循环才能够执行。但是如果父消息循环中的任务是需要嵌套的，那么就执行，如果不是，那么就需要等待，子消息循环退出再执行。这里不能执行的任务将会放到deferred\_non\_nestable\_work\_queue\_队列中。

从代码中可以看出，首先判断任务是不是嵌套的，如果是就执行，如果不是判断当前的消息循环是不是存在嵌套的消息循环，如果不存在，那么就执行任务，如果存在，就先把当前的任务放入到deferred\_non\_nestable\_work\_queue\_队列中。

返回到DoWork()函数，在for循环里面会一直这样循环，直到找到一个正确执行的任务，返回true，否则一直执行，直到work\_queue为空，此时返回false。最后在执行完DoWork()函数之后就会返回一个bool值，如果为true，就说明已经执行了一个任务，如果为false，说明由于所有的任务没有被执行。这个变量对于以后的调用会起到很重要的作用。

（3）继续回到messagepump对象的Run函数，当执行完messageloop对象的DoWork()函数时，就会判断keep\_running\_参数，如果为true的话，就继续执行。那么就会执行到messageloop的DoDelayedWork()函数，传入的参数就是delayed\_work\_time\_参数，此参数就是在图3.17中设置的参数，用来表示最近的延迟任务需要等待多长时间进行调度。Messageloop类的DoDelayedWork()函数的截图如图3.20所示：

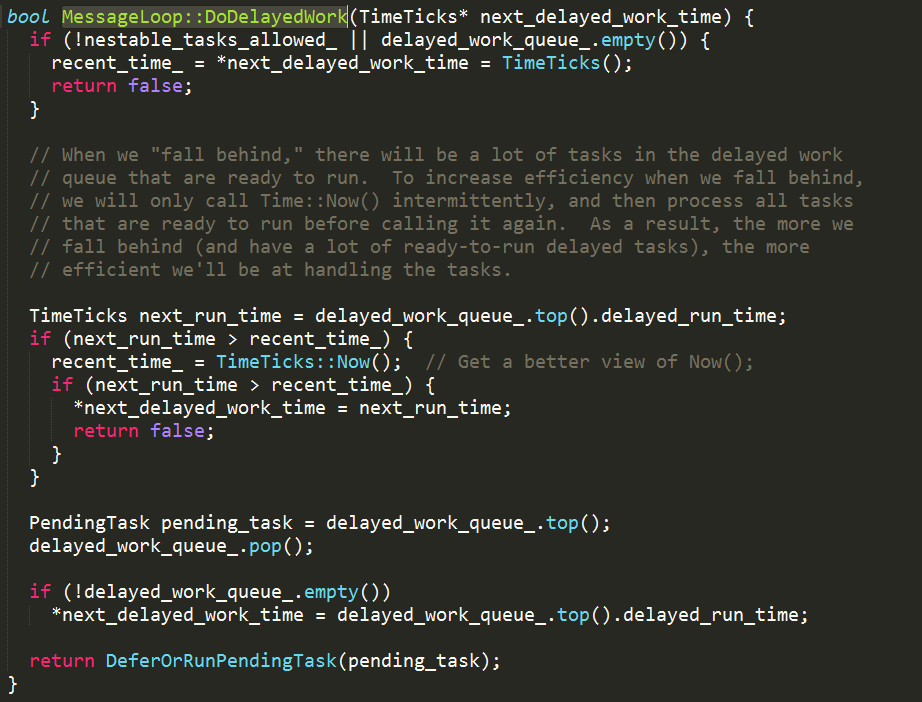


图 3.20

从代码中可以看出，首先判断delayed\_work\_queue\_是不是为空，如果为空的话直接返回false，在下面有一段说明，就是说，如果延迟任务中已经有了很多的任务可以执行了，那么就要把所有可以执行的延迟任务都执行，具体的实现接着看下面代码。

首先定义一个时间戳参数，记录延迟任务队列中首任务的延迟时间，然进行判断。如果首任务的延迟执行时间比现在还晚，就说明现在的延迟任务都还没有准备好，那么直接返回false，如果首任务可以执行，那么就取出延迟任务队列里面的首任务，进行执行，同时修改传入进来的next\_delayed\_work\_time变量，记录下次需要执行延迟任务的时间。至于DeferOrRunPendingTask()函数在DoWork()函数中也用到，这里就不做具体的分析了。

继续分析图3.13中的代码。执行完messageloop对象的DoDelayedWork()函数之后，会返回一个bool 值，如果为true就说明已经执行了一个延迟任务，如果为false，则说明所有的延迟任务都没有被执行。这个变量和执行完messageloop对象的DoWork()函数返回的值取或值，如果为true，则说明至少有一个正常的任务或者是延迟任务被执行。那么分析图3.13的代码，如果did\_work为true，则继续下个for循环，继续执行DoWork()函数和DoDelayedWork()函数。直到两者返回的都是false，说明延迟任务和普通任务要么不需要被执行，要么都执行完了。那么就需要这行messageloop的DoIdleWork()函数。DoIdleWork()函数的截图如图3.21所示：

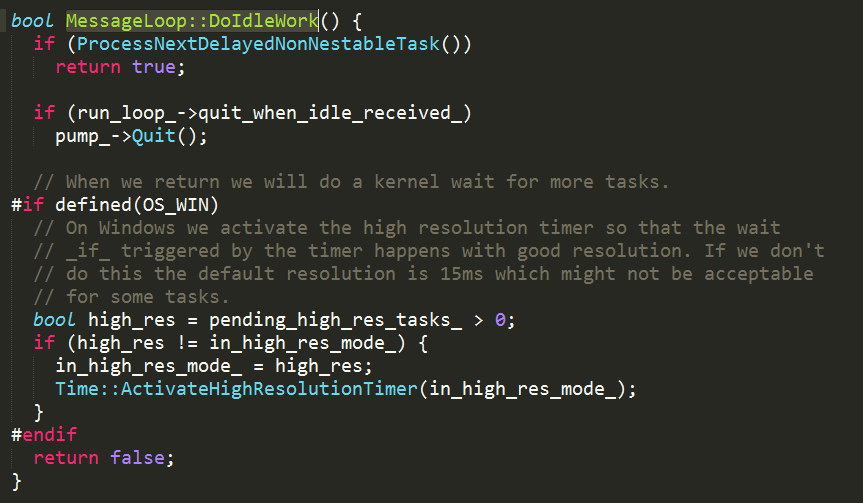


图 3.21

首先会执行messageloop的ProcessNextDelayedNonNestableTask()函数，其函数截图如图3.22所示：

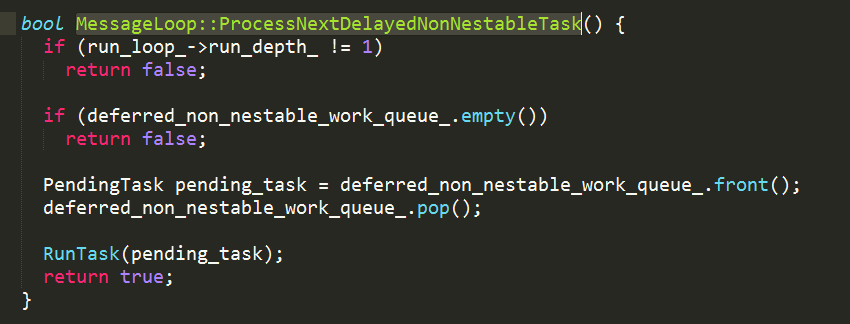


图 3.22

首先会判断当前消息循环是不是有嵌套的消息循环，如果有就是深度不等于1，则返回false，如果返回false，那么DoIdleWork() 函数就会判断是不是需要将messagePump对象停止掉。如果ProcessNextDelayedNonNestableTask()函数中返回true，就是说名当前消息循环没有嵌套的循环，并且deferred\_non\_nestable\_work\_queue\_不是空的，并且执行了一个任务，那么返回true。此时DoIdleWork()函数也返回true，那么看图3.13中的代码，messagePump种的for循环会进入另一次循环。

最后，如果DoWork() DoDelayedWork() DoIdleWork() 都返回false，说明当前所有的任务都是不能够执行，或者当前没有任务，那么messagePump就是判断进入睡眠状态。

（4）看图3.13最后的代码，如果delayed\_work\_time\_是空的话，就说明当前的messageloop中延迟任务也没有，那么就直接睡眠，然后等待唤醒就可以了。但是如果不为空，就说明存在延迟任务，那么就会判断一个延迟时间delay，如果延迟时间比较长，就是大于TimeDelta，那么就设置睡眠一段时间，然后继续执行。如果delay短，就说明当前的延迟任务可以执行了。那么就要将delayed\_work\_time\_设置为当前时间，继续for循环。然后执行延迟任务。

## 3.4、Thread B 向MessageLoop投递任务

Thread B 向Thread A投递任务，就是先要找到Thread A对应的messageloop，然后调用其PostTask() 函数，以后的执行过程和Thread A调用PostTask()之后的过程是一样的，这里就不在做分析。

## 3.5 整体流程图

其整体的流程图如图3.23所示，这个流程图是比较简单，没有把所有的函数都写出来，只是写了比较重要的几部分。大家可以有一个整体的思路。

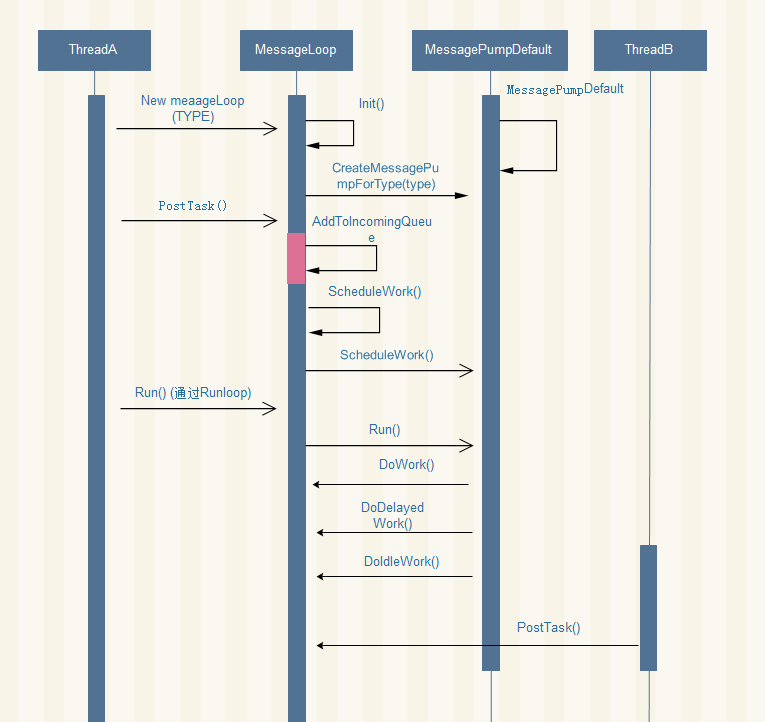


图 3.23

# 四、Android Chromium消息循环分析

## 4.1 Android消息循环基础知识

要想知道Android系统的chromium是怎么样进行消息循环的，需要了解Android系统的消息循环原理。

首先对于一个Android系统的应用程序来说，一旦开始启动，那么就会有一个Java层的消息循环来处理用户输入等事件。其中Android系统提供两个有关的类，Android.os.Looper和Android.os.Handler。其中Android系统中，每一个线程都会关联到一个Looper，它会调用自己的loop() 函数不断的轮询线程的消息队列，当访问到一个消息的时候，消息会记录处理自己的handler，然后由Looper发送到相应的Handler，由Handler的HandleMessage()函数处理该消息。

至于Handler是允许开发人员通过handler向线程发送消息或者可执行对象，当然可以跨线程发送消息，如果A线程向B线程发送消息或者可执行对象，那么A线程需要获取B线程的handler，然后通过handler的sendMessage函数，向B线程的消息队列发送任务。由此可见，handler的作用主要有两个：一、异步处理消息或者可执行对象；二、把一个消息发送到线程的消息队列中。

Handler和Looper合作的关系图如图4.1所示：

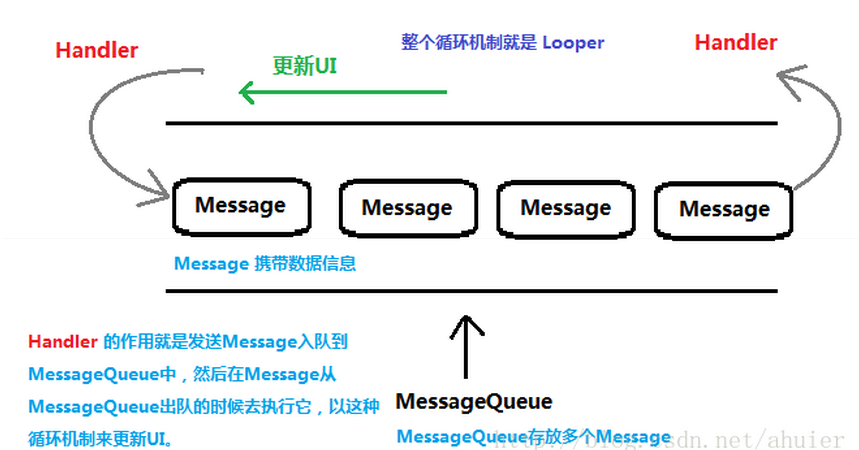


图 4.1

总体就是Handler通过自身的sendMessage函数，将一个消息或者任务队列放入到线程的消息队列中。消息队列有一个looper，不断的轮询消息队列，如果队列有消息，那么就取出，将消息发送给对应的handler，由handler对象的handlemessage()函数处理。如果消息队列为空，那么Looper就会阻塞，等待有消息进入到消息队列，将其唤醒。

注意，在一个线程中只有一个Looper和一个消息队列，但是可以创建多个handler。每一个handler都和线程的消息队列和Looper关联。其组成关系如图4.2所示：

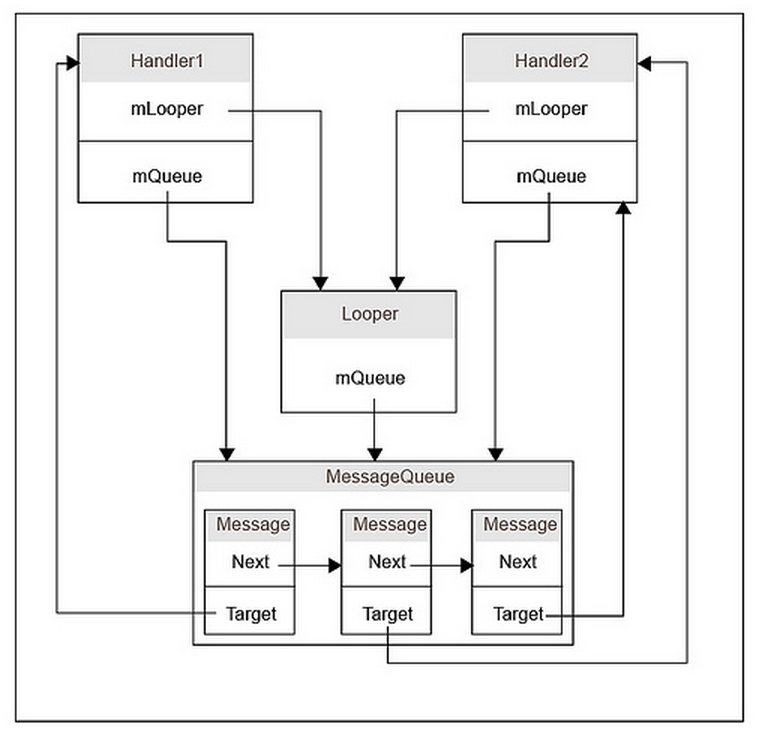


图4.2

## 4.2 Thread A创建messageloop

所有的东西都从创建的过程开始分析，这个会比较容易明了。

首先创建messageloopUI对象，但是创建的时候定义了OS\_ANDROID,所以创建的messageloopUI会有一个start()函数，以后启动的时候是运行这个start() 函数，而不是以前的Run()函数，这点需要注意。在messageloopForUI里面的start()函数会调用相应的messagePump的start()函数，大家需要知道这个就可以了，以后start() 函数就不做分析了。因为这个是在JAVA端整体管理的，但是至于是什么时候真正启动的，这个还没有找到。。。

创建的过程和第三节中讲的创建messageloopDefault的过程大同小异，都是先调用init函数，这点都是相同的，然后调用createMessagePumpForType()函数，createMessagePumpForType()函数在第三节已经分析了，这里不作具体的分析，需要知道的是，这时传入的type是TYPE\_JAVA并且定义了OS\_ANDROID系统，那么就会创建一个messagePumpForUI对象。

此对象的具体实现时在message\_pump\_android.h文件和message\_pump\_android.cc文件中实现，messagePumpForUI对象的构造函数很简单，只是将run\_loop\_对象设置为NULL，其代码截图如图4.3所示：

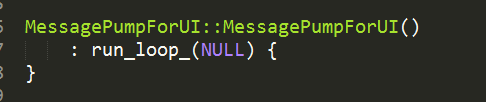


图 4.3

至此C++端的messageloop已经创建完成，但是Java端的消息循环是在开始的时候就创建的所以就没有将创建过程。

## 4.3 Thread A开始消息循环

为什么这次先讲Thread A开始消息循环因为，Thread A的消息循环是由JAVA端控制，开始的时间还没有确定，所以我猜想是启动的创建之后就启动了。前面已经讲过启动的时候会调用Messageloop对象的Start() 函数，然后由其调用messagePumpForUI对象的Start()函数。其函数截图如图 4.4所示：

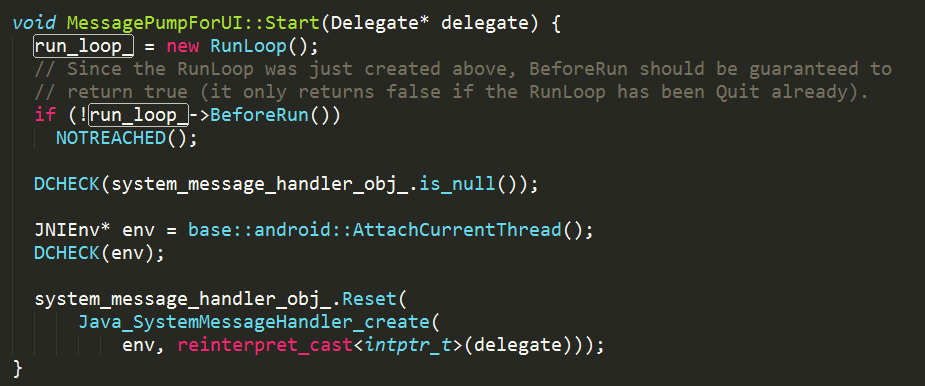


图 4.4

从函数中可以看出，start()函数将messagePumpForUI对象中的run\_loop\_对象初始化，然后将system\_message\_handler\_obj\_对象初始化，这里用到了JNIEnv技术，但是具体怎么操作就不需要理解了。

其中SystemMessageHandler类的实现时在SystemMessageHandler.java文件中实现，它继承Android.os.Handler，用法和Android.os.Handler相同，用于接收java端looper给分配的消息和向线程Java端的消息队列发送消息。

因为Java端的Looper是程序一启动变开始执行的，现在只是向这个Looper注册了这个handler对象，专门用来处理c++端的消息循环。

## 4.4 Thread A 向messageloop投递任务

当Android系统的messageloop创建完成之后，Thead A会向messageloop投递任务，其函数调用的过程和MessageLoopDefault调用的过程大体相同，具体可参见图3.11，但是不同之处是，在调用PostPendingTask() 函数的时候，每次都会调用messageloop对象的ScheduleWork() 函数。然后messageloop对象的ScheduleWork() 函数调用messagePumpForUI对象的ScheduleWork()函数，其函数截图如图4.5所示

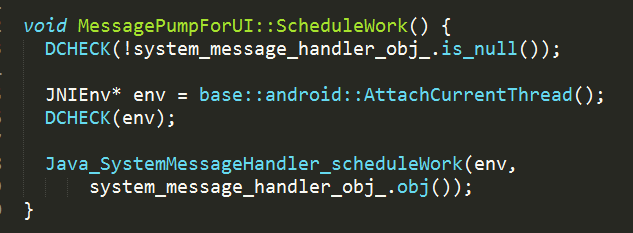


图 4.5

从代码中可以看出，该函数时调用Java端SystemMessageHandler对象的ScheduleWork() 函数，其实现的代码如图4.6所示：

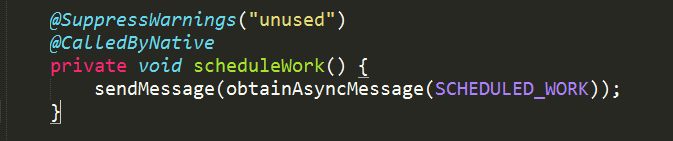


图 4.6

从代码中可以看出是通过调用sendMessage()函数向Java端looper的消息循环发送一个消息，消息很小，只是记录一个int值，表示消息是进行消息循环。

## 4.4 Thread A异步处理消息

（1）在分析完4.3节后，就会发现，每次调用messageloop对象的PostTask()函数，都会通过SystemMessageHandler对象向线程的Java端发送一个消息。如4.1节所讲，Java端的Looper会不断的轮询自己的队列，然后将消息发送给对应的handler，由handler对象的handleMessage()函数处理。在这里handler就是SystemMessageHandler，然后其handleMessage()函数截图如图4.7所示：

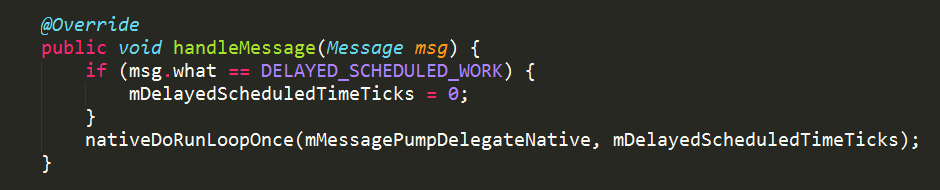


图 4.7

看代码可以看出，是调用本地的DoRunLoopOnce()函数，它起到的作用和messagePump对象的Run()函数相同，就是调度messageloop对象任务队列里面的任务。其具体的实现是在message\_pump\_android.cc文件中。

至此整体从C++端的消息发送到Java端的消息队列，然后由Java端的消息调度到C++端已经将清楚了，具体的整体流程图如图4.8所示：

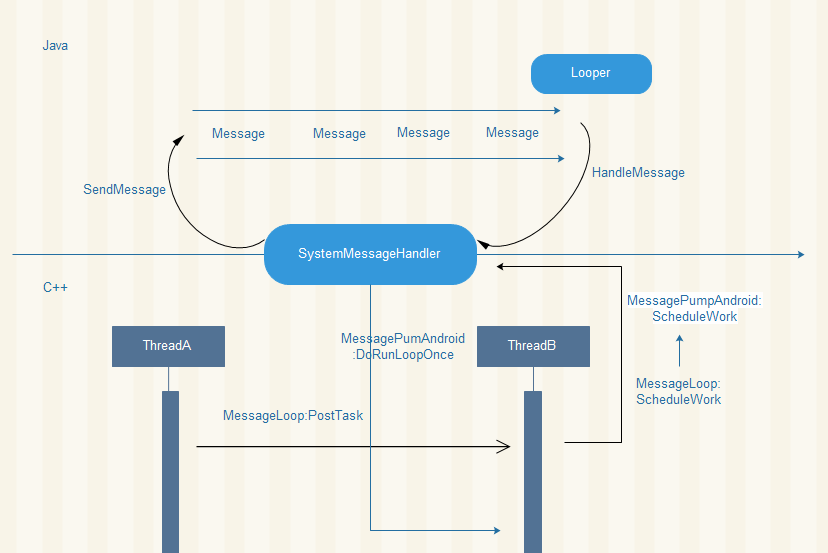


图 4.8

整体流程就是无论线程A还是线程B调用MessageLoop对象的PostTask() 函数，都会做两件事：一、将任务放入到messageloop对象中的消息队列；二、调用messageloop对象的ScheduleWork()函数，然后由其调用messageloop对象中messagepump对象的ScheduleWork()函数，然后由其调用SystemMessageHandler对象的ScheduleWork() 函数，然后调用其SendMessage()函数，将消息发送给Java端的消息队列。Looper会一直轮询自己的消息队列，然后将发现的消息发送给对应的SystemMessageHandler对象，然后由其HandleMessage() 函数处理，由该函数调用C++端的DoRunLoopOnce() 函数，通过此函数调度C++端messageloop对象的消息队列里面的任务。

## 4.6 DoRunLoopOnce() 函数分析

首先需要了解一个知识点在Java端的消息队列中对应SystemMessageHandler的消息只有两种：一个是普通的SCHEDULED\_WORK消息，这种消息可以在队列中存在多个，它的存在是由SystemMessageHandler调用ScheduleWork() 函数将消息发送给消息队列的。另一个是DELAYED\_SCHEDULED\_WORK消息，这个消息用来记录需要执行延迟任务，但是这个消息在队列中只存在一个，如果要放入的话，就需要将队列中以前的任务删除，但是删除操作是很费时的，所以调度的时候需要判断是不是需要放入。该消息的放入是由于SystemMessageHandler对象调用scheduleDelayedWork()函数将消息放入到消息队列的。其函数的具体实现代码如图4.9所示：

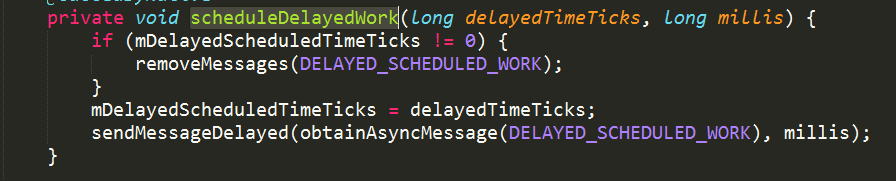


图 4.9

从代码中可以看出，他会判断当前任务队列中是不是有DELAYED\_SCHEDULED\_WORK消息，判断标准就是mDelayedScheduledTimeTicks参数，如果为0则表示队列中没有该消息，然后将此值赋值，值为需要多长时间会执行这个延迟任务。然后调用SystemMessageHandler对象的sendMessageDelayed()函数，该函数可以实现控制在经过一段时间之后，向消息队列发送DELAYED\_SCHEDULED\_WORK消息。

然后分析DoRunLoopOnce()函数，其中传入的参数有三个：一、JNIEnv运行环境对象，二、本地的delegate对象，也就是对应的messageloop对象，三，delayed\_scheduled\_time\_ticks参数，这个参数就是上面讲的控制Java端消息队列是否有延迟任务的参数，如果为0表示没有延迟任务，如果不为0表示有延迟任务，并且值代码最近的需要执行的延迟任务的等待时间。

DoRunLoopOnce()函数的截图如图4.10所示：

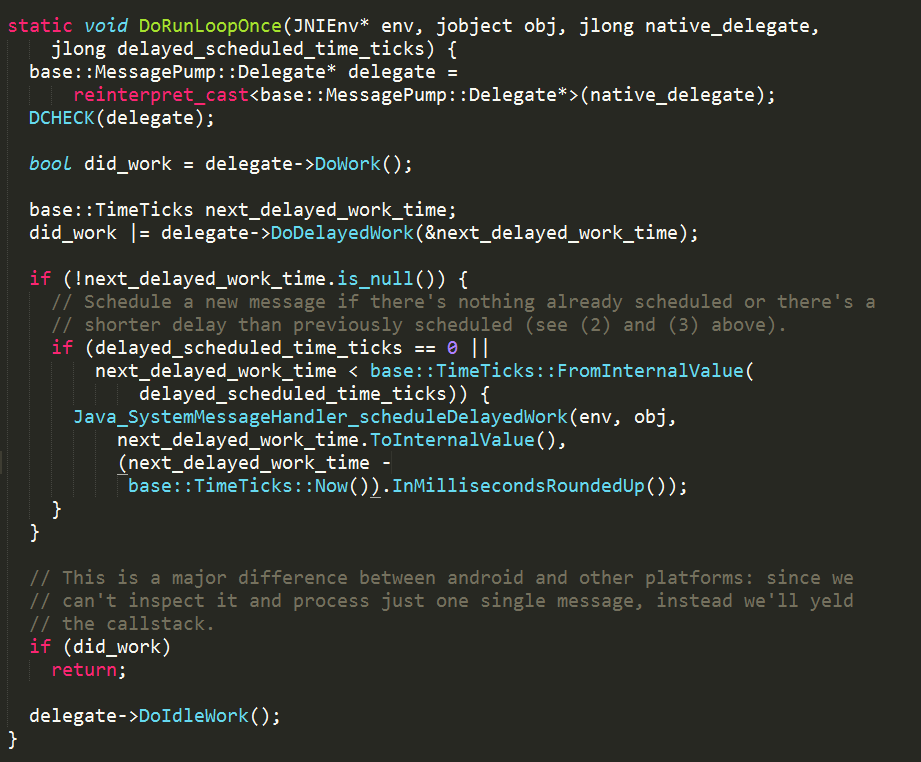


图 4.10

注意，里面有的一些注释，我删掉了。从代码中可以看出，他的整体调度策略和Linux平台上是差不多的，都是先调用DoWork()函数，DoDelayedWork()函数，然后如果两者都是返回false，也就是说没有任务可以被执行，那么就调用DoIdleWork()函数。

首先调用DoWork()、DoDelayedWork()函数，这个分析和Linux的代码是一样的，所以就不做具体的分析了，参见第三章。接着在执行完DoDelayedWork()函数之后，next\_delayed\_work\_time参数代表的就是下一次要执行的延迟任务的时间，因为前面讲过，在Android平台上的消息队列中只能有一个延迟任务的消息，如果每次都调用scheduleDelayedWork()函数的话，会把以前的延迟任务消息删除，这样很浪费时间，那么就需要一个调度策略。

什么时候调度scheduleDelayedWork()函数呢，有两种情况：一、在Java端的消息队列中没有延迟任务消息，这时候表示函数中参数delayed\_scheduled\_time\_ticks等于0；二、在执行DoDelayedWork()函数之后，下一个将要执行的延迟任务的时间比Java端消息队列中下一个执行延迟任务的时间更短，在函数中表现的就是next\_delayed\_work\_time小于delayed\_scheduled\_time\_ticks。注意，因为两者一个是C++端的时间，一个是Java端的时间，所以比较和传入参数的时候都需要转化一下。

如果存在上述两种情况，那么就会调用Java端的scheduleDelayedWork()函数，具体的函数代码分析前面已经讲过，请参照图4.9。

最后因为Java与C++之间的调度关系是C++只要有一个任务，就需要向Java端发送一个消息。Java端轮询到一个消息之后就会调用一下C++端的DoRunLoopOnce()函数，所以，最后如果did\_work为true，表示已经正确执行了一个任务，所以就可以返回，等待下一次的调用。如果did\_work为false，表示前面没有一个任务正确执行，那么就可以执行DoIdleWork ()函数，具体的此函数的调用过程前面也已经分析过，这里就不在分析。

至此DoRunLoopOnce()函数已经分析清楚，为了更加清楚的表达，我将一些调用过程画了一个图，如图4.11所示：

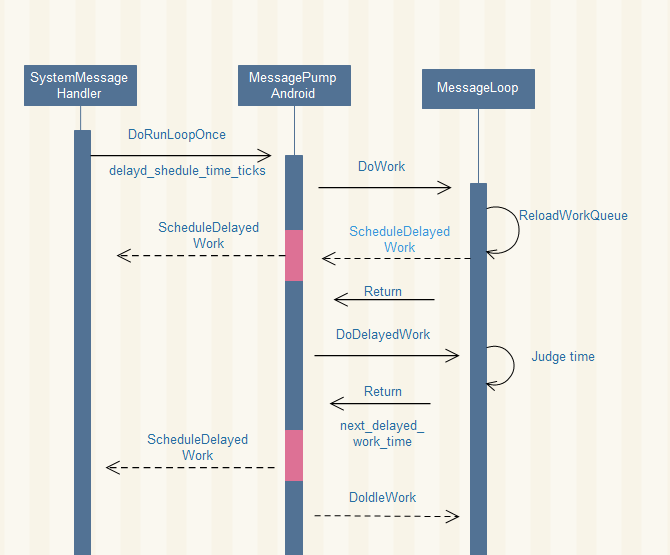


图 4.11

由SystemMessageHandler调用MessagePumpAndroid中的DoRunLoopOnce()函数，传入的参数里面有一个delayed\_shedule\_tme\_ticks，如果为0表示Java端的消息队列中没有延迟任务，如果不为0则表示Java端最近的需要执行延迟任务所需要的时间。然后由DoRunLoopOnce() 函数调用messageloop对象的DoWork()函数，如果一个work是延迟任务的话，会判断是不是需要调用ScheduleDelayedWork()函数，如果刚开始延迟任务队列为0，则需要调度，最后会DoWork()函数会返回一个bool值，表示是否正确执行了一个任务。然后调用DoDelayedWork()函数，也会返回一个bool值，同时会修改next\_delayed\_work\_time参数，此参数代表C++端最近的需要执行的延迟任务的时间，然后由此判断是否需要调用Java端SystemMessageHandler的ScheduleDelayedWork()函数。最后根据DoWork()函数和DoDelayedWork()函数的返回结果判断是不是需要执行DoIdleWork()函数。

# 五、NW.js中Render进程消息循环分析

## 5.1 Node.js的消息循环

（1）首先需要了解Node.js的消息循环，因为现在我也没有对Node.js的源代码进行分析，现在讲的消息循环只是参照各种技术博客上和官网上的知识，所以如果大家有觉得不对的地方，希望指出来。

Node.js启动时，会创建一个无限循环，不断轮询着自己任务队列，每一次执行循环体的过程成为一次Tick。过程就是查看是否有事件等待处理，如果有则取出事件及相关的回调函数，如果存在相关联的回调函数就执行他们。然后进入下一次循环。

观察者：每个事件循环中有若干个观察者，通过向观察者询问来判断是否有事件要处理。事件来源于网络请求、文件I/O等，不同类型事件对应不同观察者。事件循环就是从观察者那取出事件并处理。

观察者的类型有很多，通过uv\_TYPE\_init(uv\_TYPE\_t\*)函数创建。如果让观察者监听事件，则调用uv\_TYPE\_start(uv\_TYPE\_\*, callback)，如果停止监听则调用：uv\_TYPE\_stop(uv\_TYPE\_t\*)。注意：uv\_async\_s观察者不需要start函数，只要init之后就默认开启了。这就是messagepumpuv中的观察者。

只要有活动的观察者，事件循环就会一直进行，如果没有活动的事件观察者，事件循环就会退出。如果任何一个观察者都没有将要处理的请求对象，那么循环就会进入睡眠状态。当有观察者收到需要执行的请求对象的时候，就会通过UV\_EXTERN int uv\_async\_send(uv\_async\_t\* async) 用来唤醒libuv的线程。

请求对象：从JavaScript发起调用到内核执行完操作的过度过程中，存在一种中间产物，叫请求对象。从JS曾传入的参数和方法以及回调函数封装在请求对象中。对象封装完毕后将对象放入线程池中等待进行。

其整体的流程图如图5.1所示：

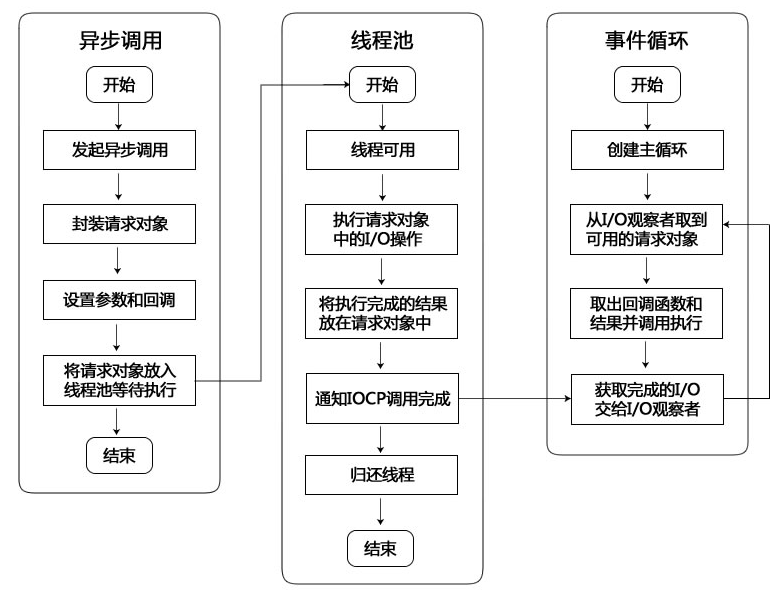


图 5.1

图中以一个IO的异步调用为例，首先发起一个异步调用，然后Node.js会自动的把这个请求调用封装成一个请求对象，包括一些参数和等IO完成后的回调函数，然后就把它放入到线程池中，如果线程池中有空闲的线程，那么就执行请求对象中的IO操作，等IO操作完之后，将结果把请求对象封装，交给IO观察者。这时候的请求对象里面有IO的结果，也有需要执行的回调函数。

事件循环轮询每一个观察者，查看是否有请求对象需要执行。轮询到IO观察者的时候，就是执行刚准备好的请求对象，分析里面的IO结果和回调函数，然后执行回调函数。

## 5.2 Node.js与chromium消息循环结合分析

（1）前面已经讲过，chromium中为了分别各个平台将messagepump分离出来，所以为了实现一个Node.js平台上的消息循环，实现了一个MessagePumpUV类，该类继承于messagepump类，其具体的实现在NW.js的代码中的message\_pump\_uv.h文件和message\_pump\_uv.cc文件中实现。为了更加清晰，作者定义了一个TYPE\_NODE类型的messageloop类MessageLoopForUV类，它继承于messageloop类。其实现和messageloop相同，只是对应的messagepump为MessagePumpUV。

其中MessageLoopForUV的代码实现如图 5.2所示：

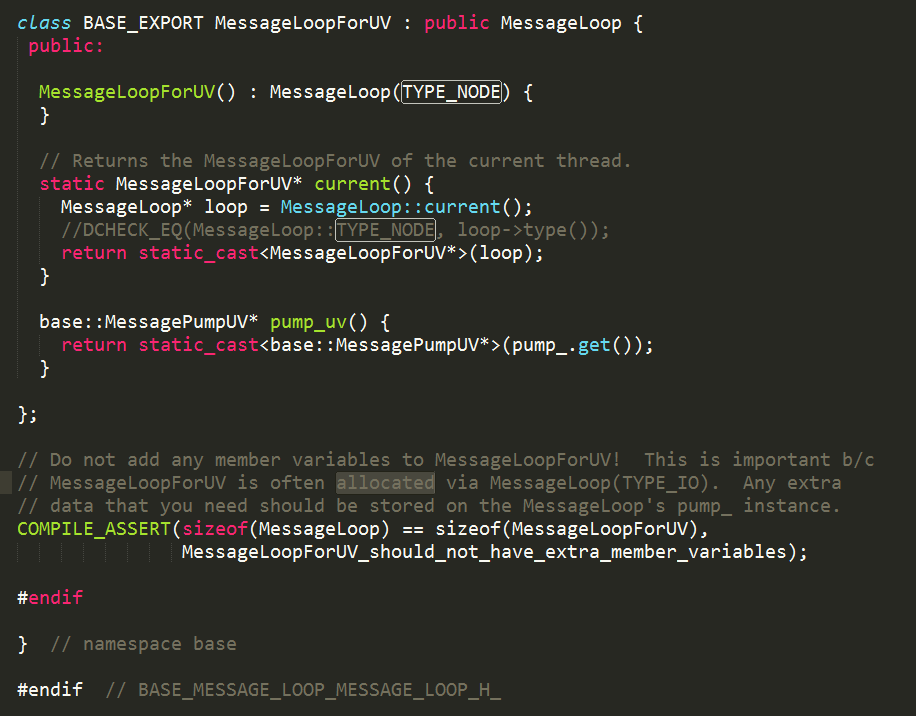


图 5.2

从代码中可以看出，MessageLoopForUV没有多余的变量，只是重新实现了messageloop的几个函数。

MessagePumpForUV类的构造函数如图5.3所示：

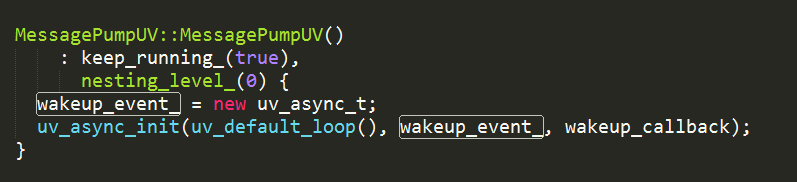


图 5.3

从图中可以看出，构造函数创建了一个uv\_async的观察者，这个观察者只是观察是不是有wakeup\_event\_事件。从整体的代码中看，这个观察者的作用只是当Node.js的循环睡眠的时候，messageloop有任务进入的时候唤醒一下。

（2）分析MessagePumpUV的Run(Delegate\* delegate)函数，因为从上面的文章中可以看出MessagePumpUV的Run(Delegate\* delegate)函数是其主要函数。它负责调度messageloop中的任务队列怎样执行。

此函数的分析分为三部分进行分析，第一部分就是建立初始化的条件，其代码的截图如图5.4所示：

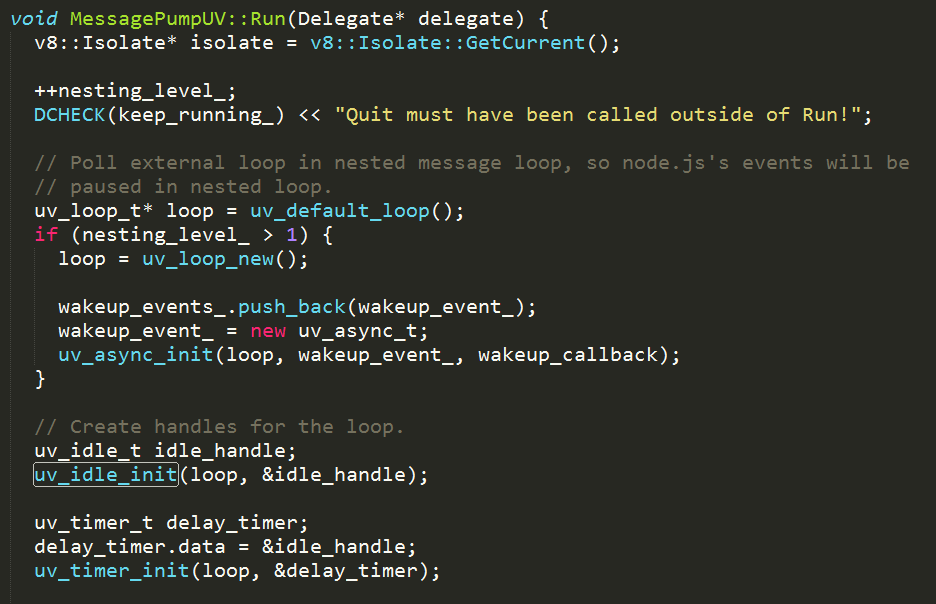


图 5.4

从代码中可以看出，首先获取当前V8的运行环境，然后每次调用run函数的时候都会把nesting\_level\_加1，如果此时是1的话，表示是在正常的messageloop里面，如果大于1表示是在嵌套的messageloop里面，如果在嵌套的messageloop里面，Node.js的主循环就需要停止，此时所有的wakeup\_event\_会被保存起来。然后建立一个新的循环，嵌套的messageloop退出的时候，会把新建的循环停止，然后恢复以前的wakeup\_event\_。

同时还定义了两个监听者，idle和timer，这两个主要是用来处理以后的延迟任务。

第二部分就是进入循环，在代码中的表示就是进入一个for语句。其代码如图5.5所示：



图 5.5

这个是MessagePumpUV的主要部分，负责调度messageloop任务队列里面的任务。其大体结构和messagePumpDefault差不多，都是先执行messageloop的DoWork()函数，然后执行DoDelayedWork()函数，然后根据两者返回的did\_work变量，判断是进行下一次循环或者是往下进行，执行DoIdleWork()函数。

这中间有一个node：CallTickCallback()函数，具体的作用我还没有分析清楚，我觉得是让node的消息队列轮询一次，不能让此线程的消息队列一直占有CPU资源。

最后执行DoIdleWork()函数，最后所有任务都执行完毕之后，开始判断是不是进入睡眠状态。如果此时没有延迟任务，那么就执行uv\_run(loop, UV\_RUN\_ONCE)函数，该函数作用是让Node.js的消息队列在轮询一次，如果所有的监听者都没有要执行的请求，那么Node.js的消息轮询就会进入睡眠状态，如果存在监听者有可以执行的请求对象，那么就会执行请求对象。

如果有延迟任务，分为两种情况，延迟任务现在就可以执行，那么就会进行下一次for循环，执行延迟任务。如果延迟任务还需要一段时间进行，那么就设置一个timer的监听者，并且执行uv\_run(loop, UV\_RUN\_ONCE)函数，将Node.js的loop循环睡眠。放过了一段时间之后会自动唤醒uv\_run，然后接着进行for循环。

第三部分就是退出for循环之后的代码，其代码截图如图5.6所示：

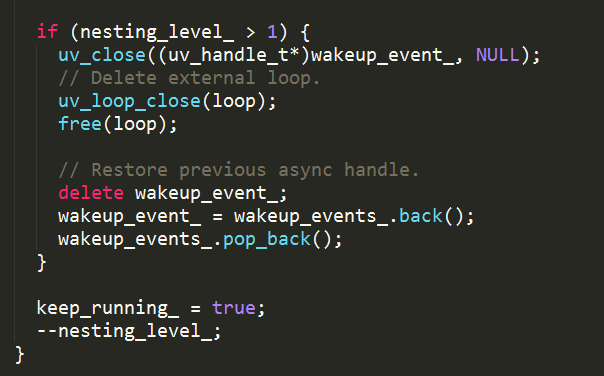


图 5.6

其主要作用就是处理嵌套循环的部分，如果以前是嵌套的，也就是nesting\_level\_大于1，那么就需要把现在的loop和wakeup\_event\_去掉，最后把nesting\_level\_减去1.

（3）分析MessagePumpUV的ScheduleWork()函数，其函数截图如图5.7所示：

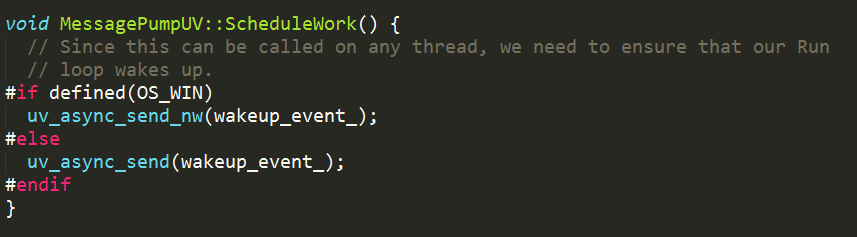


图 5.7

此函数的调用时间和以前messagepumpdefault的调用时机是一样的，这里就不在分析。作用都是讲睡眠的消息循环唤醒，只是这时候的唤醒是通过调用函数uv\_asynv\_send(wakeup\_event\_)函数。

## 5.3 Android与Node.js结合的猜想

根据前面对messagepumpdefault和messagepumpuv的分析，其实两者有很大的相同的，都是执行messageloop里面DoWork()、DoDelayedWork()、DoIdleWork()函数，然后进行阻塞，只是唤醒的方式不同，一个是自己的event，一个是Node.js的libuv。因为NW.js采用Chromium这部分只是用他的渲染引擎，Chromium的IO处理等部分都没有采用，IO应该是采用的Node.js自己的IO处理。所以可以理解其实NW.js中chromium的消息循环只是自己的default类型的。Libuv只是起到了一个简单的阻塞和唤醒作用。

那么在Android中Chromium的消息循环是不同的，它不是一个for的死循环，只是每次执行一次函数DoRunLoopOnce() 函数，它在C++端是没有阻塞和唤醒的。所以可能Android中的消息循环和Node.js的消息循环结合的地方不是很多。他们能不能Node.js自己用自己的循环，然后Android的Java端用自己的消息循环。但是两者之间的通知需要在哪里添加还是需要考虑。

以上的只是通过查看源代码和技术博客得到的结论，但是我觉得还不够深入。我觉得可以通过编译Debug版本，然后通过gdb调试进一步验证。