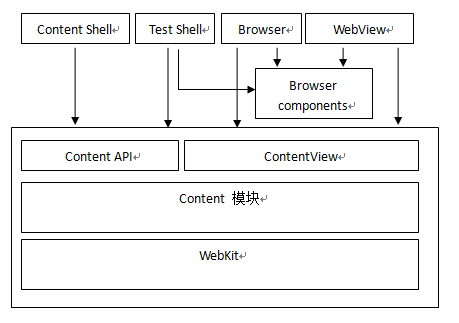
# Chromium for android

本文主要通过对比chromium for android和chromium for linux的不同，对android版本的chromium浏览器进行分析。

1. 工作模式方面，android版本是基于桌面版本的某个稳定版本进行开发的，其也不是全部开源的，而是开源了content模块及以下部分，浏览器及以下部分不会开源。
2. 架构方面：gpu进程不再是一个独立的进程，而是作为browser进程的一个线程，线程化合成已经是android版本的一个默认功能。
3. 渲染部分：特别是gpu硬件加速部分
4. 最后是功能方面，同桌面版比较，一些功能如HTML5，插件，NativeClient，extension等目前都不支持。
5. 事件循环部分。

目前，根据官网给出的信息是，只能通过chromium项目编译出安卓版本的content Shell和Test Shell，以及webview，而不能完整的编译出一个浏览器，然而，根据下图，所有的四个生成的执行程序都是依赖content模块提供的contentAPI和contentView实现的，而content是提供网页渲染和html5支持的核心，所以，分析contentView和contentTest可以很好的可以帮助理解chrome for android的架构和内部原理。



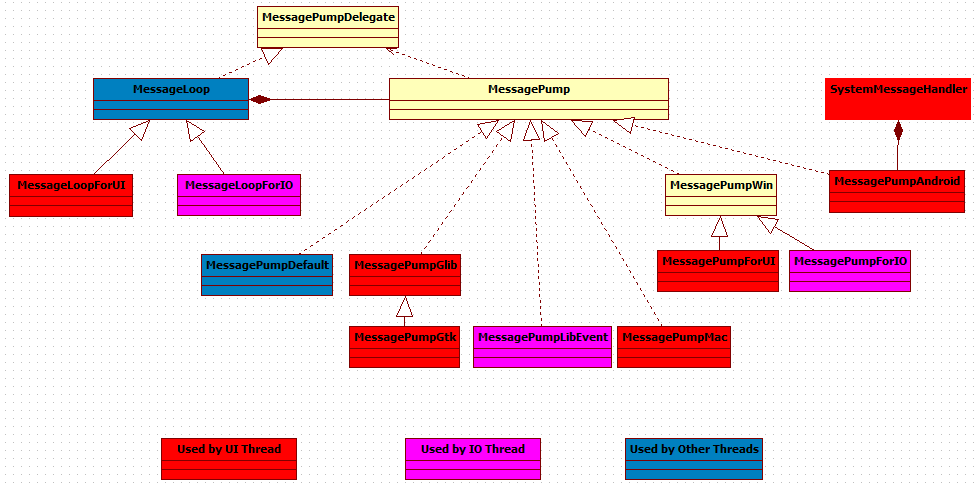
下面就上面提到的关于技术部分做详细介绍：

## 事件循环

### **消息循环**

或叫做事件循环，是异步编程模型中一个重要的概念，用来处理单个线程中发生的异步事件，这些异步事件包括用户输入事件，系统事件，Timer以及线程间派发的异步任务等。

对于事件循环见如下的集成关系：

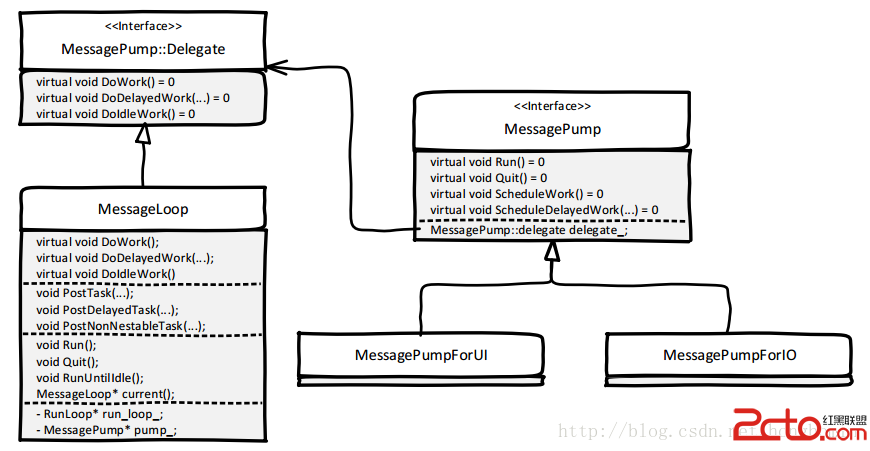


其具体实现可以参见chromiunm的消息循环文档，在此不做多介绍，只要了解一下内容：

1. chromium移动定义了三种类型的任务，为了处理这三种任务，定义了相应的三种消息循环（不包括android），一种只处理自定义任务，一种处理自定义和IO任务，一种处理自定义和UI任务。
2. messageloop用于处理只包含自定义任务的循环，通过调用MessagePumpDelegate来实现对自定义任务的处理，而包括IO任务和UI任务由于涉及到平台相关消息，因此新定义了一个基类messagePump，在不同平台不同实现来处理这两种任务。
3. Android跟其它的稍有不同，那就是它的UI线程，原因在于主循环在Java层，用户界面的事件派发机制都在Java代码来处理，因而需要在Android的消息循环机制中加入对自定义任务的处理。

基于以上的认知，在android系统中，划分了一种全新的消息循环类型**TYPE\_JAVA：专**为Android平台设计的消息循环类型，后端实现是Java层的消息处理器，用来执行添加到MessageLoop中的任务，其行为与TYPE\_UI类型差不多，但创建时不能使用主线程上的MessagePump工厂方法。

消息循环部分的类间关系同如下：



MessagePump的具体实现提供了平台相关的异步事件处理，而MessageLoop提供轮询和调度异步任务的基本框架，两者通过MessagePump::Delegate抽象接口关联起来。

MessagePump:Run()的基本固件是：

1. **for** (;;) {
2. **bool** did\_work =DoInternalWork();   //负责分发下一个UI事件或者是通知下一个IO完成事件
3. **if** (should\_quit\_)
4. **break**;
6. did\_work |= delegate\_->DoWork();
7. **if** (should\_quit\_)
8. **break**;
10. TimeTicks next\_time;
11. did\_work |=delegate\_->DoDelayedWork(&next\_time);
12. **if** (should\_quit\_)
13. **break**;
15. **if** (did\_work)
16. **continue**;
18. did\_work =delegate\_->DoIdleWork();
19. **if** (should\_quit\_)
20. **break**;
22. **if** (did\_work)
23. **continue**;
25. WaitForWork();  //
26. }

MessagePump既要负责响应系统的异步事件，又要给足够的时间片段调度Delegate去执行异步任务，所以MessagePump是以混合交错的方式调用DoInternalWork,DoWork, DoDelayedWork和DoIdleWork，以保证任何一类任务不会因得不到执行而都发生“饥饿”现象；

DoInternalWork和WaitForWork都是MessagePump具体实现的私有方法。DoInternalWork负责分发下一个UI事件或者是通知下一个IO完成事件，而WaitForWork会阻塞MessagePump::Run方法直到当前有任务需要执行；

每个MessagePump都有设置should\_quit\_标记位，一旦MessagePump::Quit被调用了，should\_quit\_将置为true, 每次MessagePump处理完一类任务时都要去检查should\_quit\_标记，以决定是否继续处理后续的任务，当发现should\_quit\_为true时，直接跳出for循环体。

### 嵌套的消息循环：

嵌套的消息循环就是当前消息循环中因执行某个任务而进入另外一个消息循环，并且当前的消息循环被迫等待嵌套消息循环退出，才能继续执行后面的任务，例如，当弹出MessageBox对话框时，意味着进入了一个新的消息循环，直到MessageBox的OK或者Cancel按钮按下之后这个消息循环才能退出。

每个MessageLoop中都有一个指针，指向当前正在运行的RunLoop实例，调用RunLoop::Run方法时，MessageLoop::current()消息循环中的RunLoop指针也会随之修改为当前运行的RunLoop，换句话说，如果此时调用MessageLoop::current()->PostTask，那么将在嵌套的消息循环中执行异步任务。

嵌套消息循环基本使用方法如下代码所示：

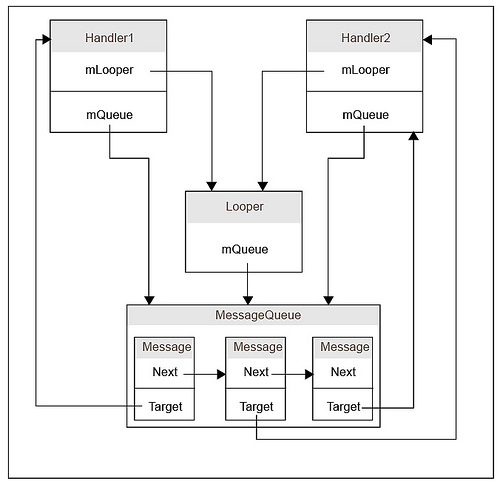
1. **void** RunNestedLoop(RunLoop\* run\_loop) {
2. MessageLoop::ScopedNestableTaskAllower allow(MessageLoop::current());
3. run\_loop->Run();
4. }
5. }
7. // 在栈上创建一个新的RunLoop
8. RunLoop nested\_run\_loop;
10. // 在当前消息循环中异步启动一个嵌套的消息循环；
11. MessageLoop::current()->PostTask(FROM\_HERE, base::Bind(&RunNestedLoop,Unretained(&nested\_run\_loop)));
12. // 一旦RunNestedLoop执行，MessageLoop::current()内部的RunLoop指针会指向nested\_run\_loop, PostTask将在嵌套RunLoop上执行Quit操作
13. MessageLoop::current()->PostTask(FROM\_HERE, nested\_run\_loop.QuitClosure());
14. // 异步执行task\_callback时，嵌套RunLoop此时已经退出了，恢复到原先的消息队列中执行；
15. MessageLoop::current()->PostTask(FROM\_HERE,task\_callback);

### Android系统的消息循环机制：

Android系统中，android.os.Looper和android.os.Handler是消息循环机制中两个非常重要的类。

Looper类似于Chromium的MessageLoop（或者是RunLoop）概念，Android系统中每个线程都可以关联一个Looper，用于处理异步消息或者Runable对象等。Android系统线程默认是没有关联任何一个Looper，开发者可以显式调用Looper.prepare和Looper.loop为线程创建并运行Looper，直到退出Looper。Looper之间的交互则需要通过类Handler完成。

Handler允许开发者向线程的消息队列发送或者处理消息和Runable对象，它有两个用途：1）通过handleMessage方法异步处理自己线程中的消息队列；2）通过sendMessage或post方法系列向其他线程的消息队列发送消息。一个线程可以有多个Handler，Looper轮询消息队列时，负责将消息派发给目标Handler，由目标Handler的handleMessage来处理这个消息。



### Android的消息循环的使用：

这里所讨论的主要是针对主线程的消息循环，即UI线程，因为IO线程是在native层创建的，并没有涉及到与UI元素的交互事件，并且Android也是POSIX系统，不用考虑IO线程消息循环的整合问题。

如上所述，Android系统的消息循环是构建在Java层的，Chromium需要解决的问题，如何在主线程上将C++层负责管理和调度异步任务的MessageLoop整合到Android系统的控制路径上。答案当然是使用android.os.Handler。

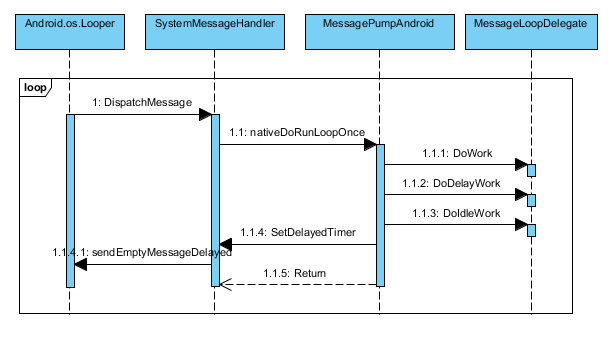
首先来看看Android平台上MessagePumpForUI具体实现。与其他平台不同的是，MessagePumpForUI的实现中，启动整个消息循环处理逻辑的Run方法居然不做任何事情，取而代之是，新增了一个Start方法在Chromium中启用Android系统的消息循环，如下代码所示：

1. **void** MessagePumpForUI::Run(Delegate\* delegate) {
2. NOTREACHED()<< "UnitTests should rely on MessagePumpForUIStub in  test\_stub\_android.h";
3. }
4. **void** MessagePumpForUI::Start(Delegate\* delegate) {
5. run\_loop\_ = newRunLoop();
6. // Since the RunLoop was just created above, Before Run should be guaranteed to
7. // return true (itonly returns false if the RunLoop has been Quit already).
8. **if**(!run\_loop\_->BeforeRun())
9. NOTREACHED();
10. JNIEnv\* env =base::android::AttachCurrentThread();
11. system\_message\_handler\_obj\_.Reset(
12. Java\_SystemMessageHandler\_create(
13. env,**reinterpret\_cast**<**intptr\_t**>(delegate)));
14. }

Start方法会通过JNI向Java层请求创建一个继承于android.os.Handler的SystemMessageHandler，向当前UI线程的Looper增加一个新的Handler类型，它提供了自己的handleMessage方法：

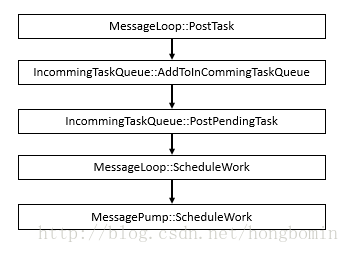
1. **class** SystemMessageHandler **extends** android.os.Handler {
2. **private** staticfinal **int** SCHEDULED\_WORK = 1;
3. **private** staticfinal **int** DELAYED\_SCHEDULED\_WORK = 2;
4. privateSystemMessageHandler(**long** messagePumpDelegateNative) {
5. mMessagePumpDelegateNative = messagePumpDelegateNative;
6. }
7. @Override
8. **public** voidhandleMessage(Message msg) {
9. **if** (msg.what== DELAYED\_SCHEDULED\_WORK) {
10. mDelayedScheduledTimeTicks = 0;
11. }
12. nativeDoRunLoopOnce(mMessagePumpDelegateNative,mDelayedScheduledTimeTicks);
13. }
14. …
15. }

MessagePumpAndroid其实是被Looper调用的。当它被调用时，其会把执行任务的工作交给MessagePumpDelegate，这里也就是MessageLoopForUI来完成，如下图所示的UI线程的消息循环主逻辑。Java层的SystemMessageHandler和C++层的MessagePumpAndroid其实都是辅助Looper调用执行chromium的自定义类的。



### C++层将异步任务发送给Java层:

每当C++层通过调用MessageLoop::current()->PostTask\*向UI线程的MessageLoop添加新的异步任务时，MessageLoop::ScheduleWork都发起调度任务执行的动作，而MessageLoop::ScheduleWork则是请求MessagePump来完成这个动作，其调用链为：



所以MessagePumpForUI的ScheduleWork需要将来自C++层请求发送给Java层的SystemMessageHandler：

//c++

void MessagePumpForUI::ScheduleWork() {

JNIEnv\* env =base::android::AttachCurrentThread();

Java\_SystemMessageHandler\_scheduleWork(env,

system\_message\_handler\_obj\_.obj());

}

//java

class SystemMessageHandler extends Handler {

...

@CalledByNative

private voidscheduleWork() {

sendEmptyMessage(SCHEDULED\_WORK);

}

...

}

SystemMessageHandler.scheduleWork唯一要做的事情就是向UI线程的消息队列发送一个类型SCHEDULED\_WORK的消息。

### Java层Handler处理来自C++层的异步任务：

当UI线程的Looper收到这个SCHEDULED\_WORK异步消息后，它会准确无误的派发给SystemMessageHandler，由SystemMessageHandler.handleMessage重载方法去处理这个消息，如上代码所示，handleMessage执行了一个定义在MessagePumpForUI中的native方法DoRunLoopOnce，其实现代码如下：

static void DoRunLoopOnce(JNIEnv\* env, jobject obj, jlong native\_delegate, jlong delayed\_scheduled\_time\_ticks) {

base::MessagePump::Delegate\* delegate =

reinterpret\_cast(native\_delegate);

bool did\_work =delegate->DoWork();

base::TimeTicksnext\_delayed\_work\_time;

did\_work |=delegate->DoDelayedWork(&next\_delayed\_work\_time);

if(!next\_delayed\_work\_time.is\_null()) {

if(delayed\_scheduled\_time\_ticks == 0 ||

next\_delayed\_work\_time < base::TimeTicks::FromInternalValue(

delayed\_scheduled\_time\_ticks)) {

Java\_SystemMessageHandler\_scheduleDelayedWork(env, obj,

next\_delayed\_work\_time.ToInternalValue(),

(next\_delayed\_work\_time -

base::TimeTicks::Now()).InMillisecondsRoundedUp());

}

}

if (did\_work)

return;

delegate->DoIdleWork();

}

DoRunLoopOnce方法使C++层的MessageLoop有机会去处理自己的异步任务，包括延时任务和Idle任务。

至此，通过MessagePumpForUI和SystemMessageHandler的实现，Chromium系统在主线程上已经可以无缝整合到Android的消息循环中，

## 参考文档：

<http://blog.csdn.net/hongbomin/article/details/41258469>

<http://blog.csdn.net/milado_nju/article/details/8451730>