# 从W3c资料中理解事件循环模型

There must be at least one [event loop](http://www.w3.org/TR/html5/webappapis.html#event-loop) per user agent, and at most one [event loop](http://www.w3.org/TR/html5/webappapis.html#event-loop) per [unit of related similar-origin browsing contexts](http://www.w3.org/TR/html5/browsers.html#unit-of-related-similar-origin-browsing-contexts).

events, user interaction, scripts, rendering, networking, and so forth

按照w3c标准，浏览器至少有一个事件循环，最多每个相关的同源浏览器上下文有一个事件循环。

（因为浏览器实现时只有一个事件循环（在Firefox和Safari中，只有一个事件循环（进程）；chrome中则是每个tab一个事件循环（进程）），故此处只讨论一个事件循环的情况）

事件循环负责调度事件、用户交互、脚本、渲染、网络操作等任务。

[unit of related browsing contexts](http://www.w3.org/TR/html5/browsers.html#unit-of-related-browsing-contexts)?

（在浏览器实现中，“事件循环”为一个进程， 渲染引擎是单线程的，除了网络操作以外，几乎所有的事情都在单一的进程（网上资料通常称之为线程，但我理解为应该是进程！还需要继续讨论！）中处理。

网络操作由几个并行线程执行，并行连接的个数是受限的（通常是2－6个）

本文不讨论网络操作，只讨论负责调度其它任务的事件循环。

（网络操作是否也是在其它任务的进程中？只是它开了几个线程去支持并行请求？待继续讨论）

在Firefox和Safari中，浏览器只有一个进程负责调度其它任务；chrome中则是每个tab一个进程，负责调度各自tab的任务（包括网络操作？）

）

事件循环依次从多个任务队列中提取一个任务队列执行

事件循环有1个或多个任务队列，任务队列是一个对任务进行排序了的列表。

**Events**

Asynchronously dispatching an [Event](http://www.w3.org/TR/html5/infrastructure.html#event) object at a particular [EventTarget](http://www.w3.org/TR/html5/infrastructure.html#eventtarget) object is often done by a dedicated task.

Not all events are dispatched using the [task queue](http://www.w3.org/TR/html5/webappapis.html#task-queue), many are dispatched synchronously during other tasks.

**Parsing**

The [HTML parser](http://www.w3.org/TR/html5/syntax.html#html-parser) tokenizing one or more bytes, and then processing any resulting tokens, is typically a task.

**Callbacks**

Calling a callback asynchronously is often done by a dedicated task.

**Using a resource**

When an algorithm [fetches](http://www.w3.org/TR/html5/infrastructure.html#fetch) a resource, if the fetching occurs asynchronously then the processing of the resource once some or all of the resource is available is performed by a task.

**Reacting to DOM manipulation**

Some elements have tasks that trigger in response to DOM manipulation, e.g. when that element is [inserted into the document](http://www.w3.org/TR/html5/infrastructure.html#insert-an-element-into-a-document).

有以下类型的任务：

事件

解析

回调

使用资源

与DOM交互

“浏览器加入任务”是指浏览器将任务加到一个任务队列中。

每个任务都来自于一个特定的任务来源。来自于同一个特定的任务来源的任务会被加入到同一个任务队列中，而来自于不同任务来源的任务会被加入到不同的任务队列。

（比如定时器加入的回调函数、鼠标移动触发的事件、解析html文档，这些任务来自于不同的任务来源，因此会加入到不同的队列中）

（任务队列会对任务进行排序。我的理解是比如用户点击了一个按钮，该按钮绑定了一个clickHandle，那么会产生两个任务：更新按钮的UI和执行clickHandle。因为这两个任务都来自于同一个任务来源，因此会加入到同一个任务队列中。然后任务队列会对任务进行排序，更新按钮的UI会在执行clickHandle之前）（排序的依据是什么？？？）

什么是任务来源task source？如何区分是否为不同的任务来源？

##### *6.1.4.3 Generic task sources*

The following [task sources](http://www.w3.org/TR/html5/webappapis.html#task-source) are used by a number of mostly unrelated features in this and other specifications.

**The DOM manipulation task source**

This [task source](http://www.w3.org/TR/html5/webappapis.html#task-source) is used for features that react to DOM manipulations, such as things that happen asynchronously when an element is [inserted into the document](http://www.w3.org/TR/html5/infrastructure.html#insert-an-element-into-a-document).

**The user interaction task source**

This [task source](http://www.w3.org/TR/html5/webappapis.html#task-source) is used for features that react to user interaction, for example keyboard or mouse input.

Asynchronous events sent in response to user input (e.g. [click](http://www.w3.org/TR/html5/infrastructure.html#event-click) events) must be fired using [tasks](http://www.w3.org/TR/html5/webappapis.html#concept-task) [queued](http://www.w3.org/TR/html5/webappapis.html#queue-a-task) with the [user interaction task source](http://www.w3.org/TR/html5/webappapis.html#user-interaction-task-source). [[DOMEVENTS]](http://www.w3.org/TR/html5/references.html#refsDOMEVENTS)

**The networking task source**

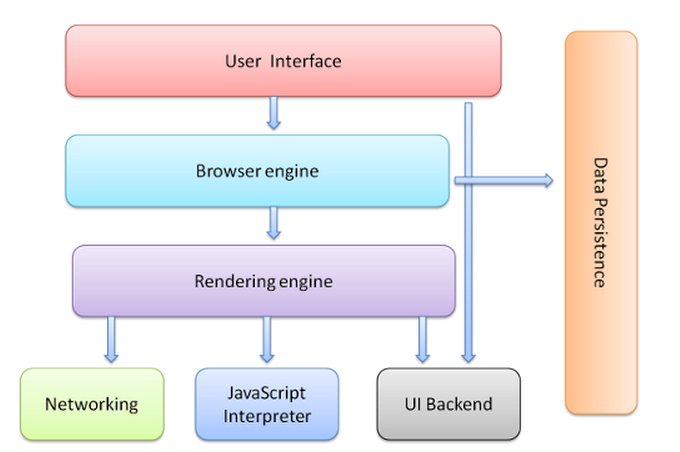
This [task source](http://www.w3.org/TR/html5/webappapis.html#task-source) is used for features that trigger in response to network activity.

**The history traversal task source**

This [task source](http://www.w3.org/TR/html5/webappapis.html#task-source) is used to queue calls to [history.back()](http://www.w3.org/TR/html5/browsers.html#dom-history-back) and similar APIs.

# 浏览器工作模型

  1. 大多数浏览器的组件构成如图



  在最底层的三个组件分别是网络，UI后端和js解释器。作用如下：

    （1）网络－ 用来完成网络调用，例如http请求，它具有平台无关的接口，可以在不同平台上工作

    （2）UI 后端－ 用来绘制类似组合选择框及对话框等基本组件，具有不特定于某个平台的通用接口，底层使用操作系统的用户接口

    （3）JS解释器－ 用来解释执行JS代码

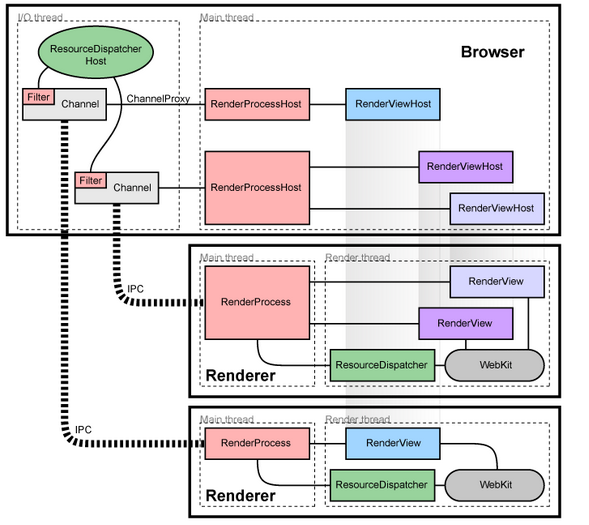
 ps：上图和知识点主要来自《[HOW BROWSERS WORK: BEHIND THE SCENES OF MODERN WEB BROWSERS](http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/)》 想深入了解的同学可以重点看下。

      2. 大多数浏览器（比如chrome）让一个单线程共用于执行javascrip和更新用户界面。这个线程通常被称为“浏览器UI线程”， 每个时刻只能执行其中一种操作，这意味着当Javascript代码正在执行时用户界面无法响应输入，反之亦然。这样做是因为javascript代码的作用就是操作DOM更新用户界面，用同一个线程来做负责这两件事情可以更高效

      3. 浏览器UI线程的工作基于一个简单的队列系统，任务会被保存到队列中直到进程空闲。一旦空闲，队列中的下一个任务就被重新提取出来并运行。这些任务要么是运行javascript代码，要么执行UI更新，包括重绘和重排。

# 关于chrome的具体实现整理

Chrome进程模型：



  相比之下，**Browser进程**既然是老大，小弟自然要多一些，除了大脑般的Main thread，和负责与各Renderer帮派通信的IO thread，其实还包括负责管文件的file thread，负责管数据库的db thread等等，它

大部分的工作是在UI线程（即主线程）上完成的，另外的主要线程有，

Browser进程包括以下线程：

io\_thread 负责进程间通讯（IPC）的消息分发，以及网络资源请求的分发。  
file\_thread 负责文件读写  
db\_thread 负责访问本地的SQLite数据库  
safe\_browsing\_thread等等

Chrome包括以下进程：

CEF3 uses a number of processes for different purposes.

* Browser process - This process is the same as the client application process that calls CefInitialize().
* Render process - Web content (WebKit and V8) is hosted in a this process.
* Plugin process - Plugins like Flash are hosted in this process. More information about the plugin architecture is available [here](http://www.chromium.org/developers/design-documents/plugin-architecture).
* GPU process - Accelerated content is rendered in this process. More information about accelerated compositing is available [here](http://dev.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome).
* Utility process - Some miscellaneous tasks are executed in this process.

All WebKit and V8 interation must take place on the render process RENDERER thread.

渲染引擎webkit和javascript引擎v8都在render进程中

**2. Render进程**    先不扯Plugin的进程，只考虑Render进程。前面说了，一个Process一个tab，只是广告用语，实际上，每一个web页面内容（包括在tab中的和在弹出窗口中的…），在Chrome中，用RenderView表示一个web页面，每一个RenderView可以寄宿在任一一个RenderProcess中，它只是依托RenderProcess帮助它进行通信。每一个RenderProcess进程都可以有1到N个RenderView实例。

    Chrome支持不同的进程模型，可以一个tab一个进程，一个site instance一个进程等等。但基本模式都是一致的，当需要创建一个新的RenderView的时候，Chrome会尝试进行选择或者是创建进程。比如，在one site one process的模式下，如果存在此site，就会选择一个已有的RenderProcessHost，让它管理这个新的RenderView，否则，会创建一个RenderProcessHost（同时也就创建了一个Process），把RenderView交给它。

## 2. Chrome的页面加载和绘制

上面这些UI控件，都是用在窗口上的（比如浏览器的外框，菜单，对话框之类的...）。我们在浏览器中看到的大部分内容，是网页页面。页面的绘制（绘制，就是把一个HTML文件变成一个活灵活现的页面展示的过程...），只有一半轮子是Chrome自己做的，还有一部分来自于**WebKit**，这个Apple打造的Web渲染器。。。

之所以说是一半轮子来源于WebKit，是因为WebKit本身包含两部分主要内容，一部分是做Html渲染的，另一部分是做JavaScript解析的。**在Chrome中，只有Html的渲染采用了WebKit的代码，而在JavaScript上，重新搭建了一个NB哄哄的V8引擎**。目标是，用WebKit + V8的强强联手，打造一款上网冲浪的法拉利，从效果来看，还着实做的不错。。。

当你键入一个Url并敲下回车后，**Chrome会在Browser进程中下载Url对应的页面资源（包括Web页面和Cookie），**而不是直接将Url发送给Render进程让它们自行下载（你会越来越发现，Render进程绝对是100%的名符其实，除了绘制，几乎啥多余的事情都不会干的...）。与各个Render进程各自为站，各自管好自己所需的资源相比，这种策略仿佛会增加大量的进程间通信。之所以采用，按照[这篇文档](http://sites.google.com/a/chromium.org/dev/developers/design-documents/multi-process-resource-loading)的解释，主要有三个优点，一个是避免子进程与网络通信，从而将网络通信的权限牢牢握在主进程手中，Render进程能力弱了，想造反干坏事的可能性就降低了（可以更好控制各个Render进程的权限...）；另一个是有利于Cookie等持久化资源在不同页面中的共享，否则在不同Render进程中传递Cookie这样的事情，做起来更麻烦；还有一点很重要的，是可以控制与网络建立HTTP连接的数量，以Browser为代表与网络各方进行通信，各种优化策略都比较好开展（比如池化）。。。

当然，在Browser进程中进行统一的资源管理，也就意味着不再方便用WebKit进行资源下载（WebKit当然有此能力，不过再次被Chrome抛弃了...），而是依托WinHTTP来做的。**WinHTTP在接受数据的过程中，会不停的把数据和相关的消息通过IPC，发送给负责绘制此页面的Render进程中对应的RenderView**。在这里，路由消息中的那个ID值起了关键的作用，系统依照此ID，能够准确的将相关的消息发送到相关的View头上，这玩意发错了地方还真不是和有人把钱错到你账户上一样，因为错收的进程基本上无福消受这个意外来客，轻者页面显示混乱，重者消化不良直接噎死。。。

RenderView接收到页面信息，会一边绘制一边等待更多的资源到来，在用户看来，所请求的页面正在一点一点显示出来。当然，如果是一个通知传输开始、传输结束这样的消息，通过序列化到消息参数里面，经由IPC发过来，代价还是可以承受的，但是，想资源内容这样大段大段的字节流，如果通过消息发过来，浪费两边进程大量空间和时间，就不合适了。于是这里用到了**共享内存**。Browser进程将下载到的资源写到共享内存中，并将共享内存的句柄和共享区域的大小序列化在消息中发送给Render进程。Render进程拿到这个句柄，就可以通过它访问到共享内存相关的区域，读取信息并进行绘制。通过这样的方式，即享用到了统一资源管理的优点，由避免了很高的进程通信开销，左右逢源，好不快活。。。

All WebKit and V8 interation must take place on the UI thread. Consequently, some CEF API functions can only be called on the UI thread. Functions that have this limitation will be documented accordingly in the associated CEF header file.

# 参考资料

《高性能Javascript》->6.1 浏览器UI线程

event loop <http://www.w3.org/TR/html5/webappapis.html#event-loops>

<http://www.w3.org/TR/html5/webappapis.html#user-interaction-task-source>

<http://www.monring.com/front_end/browser-render-thread.html>

Chromium Embedded Framework

<http://code.google.com/p/chromiumembedded/wiki/Architecture>

**Chrome源码剖析**

详见“chrome收藏夹->web标准->浏览器工作原理”中的网页资源