network request的创建和处理

## 浏览器的网络资源请求过程

先简述一下从用户输入URL到浏览器渲染图像的过程:

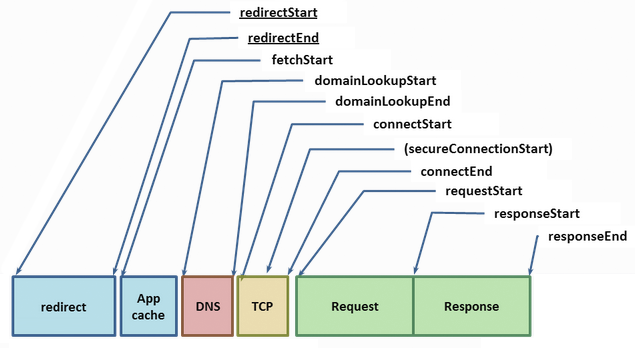
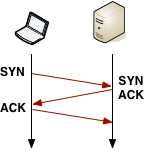


图0

给定一个网络资源的URL, 浏览器首先会检查它本地和应用的缓存(之前访问过的页面). 如果你之前获取过这个资源并提供了合适的缓存头([appropriate cache headers](http://www.aosabook.org/https:/developers.google.com/speed/docs/best-practices/caching), Expires, Cache-Control, etc.), 就可能可以使用本地缓存来填充请求 ---- 最快的请求就是不做请求. 不然的话, 要是我们必须重新验证资源, 或者它过期了, 或者没有见过它, 那就必须来一次昂贵的网络请求了.

有了域名和资源路径, Chrome首先会在它可以重用的连接中查找 -- 以{scheme, host,port}存储的套接字们. 也可以这样, 如果你配置了代理, 或者指定了[proxy auto-config](http://www.aosabook.org/http:/en.wikipedia.org/wiki/Proxy_auto-config)(PAC)脚本, chrome就会通过适当的代理查找连接. PAC脚本允许根据URL或其他指定的规则使用不同的代理, 每种都可以它们自己的套接字池. 最后要是上面的条件都不符合, 就只能以解析域名到IP地址来开始这个请求了 -- 也就是要先做DNS查找.

运气好的话, 主机名已经在缓存中, 也就是说系统查找一下就可以返回. 否则的话, 在发出DNS查询之前什么都做不了. DNS查询要花上一点时间.

[](https://github-camo.global.ssl.fastly.net/7daa3351001248eb2140e3e36225876539e39fcf/687474703a2f2f7777772e616f7361626f6f6b2e6f72672f656e2f706f73612f6368726f6d652d696d616765732f74687265652d7761792e706e67)

有了IP地址, Chrome就可以打开一个新的TCP连接, 意味着我们得执行"三次握手": SYN > SYN-ACK > ACK. 每个新的TCP连接绕不过去.

TCP握手结束后如果我们要连接到一个 (HTTPS)上, 需要再执行一次SSL握手.

这些事情都做完了, 才算建立了一个新的连接.

终于Chrome能发送HTTP请求了(图1.1 requestStart). 服务器一收到请求就可以执行它然后以流的形式向客户端发送返回数据. 到这里我们的事情做完了 – 拿到服务器返回的数据渲染页面, 如果返回数据中含有重定向的URL, 那就要再次发送请求, 得到数据, 渲染页面...

## Chrome的network request

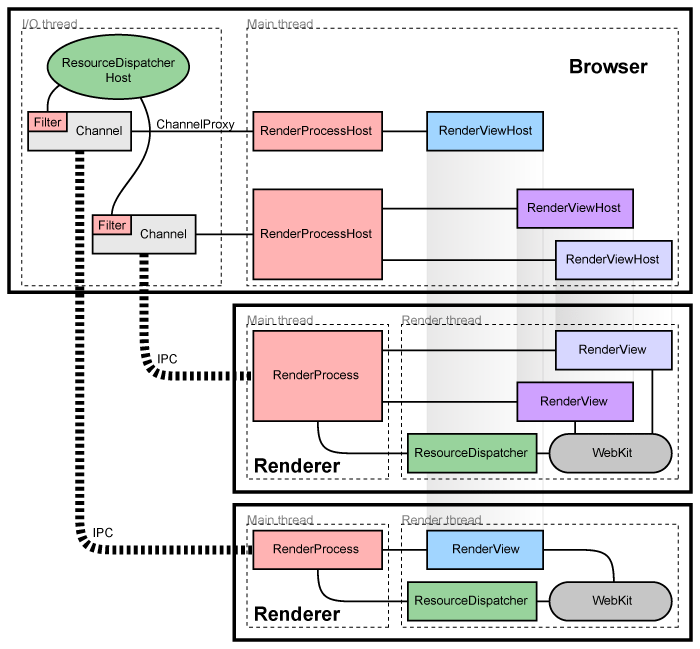


图1 – 多进程模型

先重温一下多进程结构里几个重要的类的角色: RenderView, RenderViewHost, ResourceDispatcherHost, ResourceDispatcher.

用户在地址栏输入URL是浏览器进程的UI线程接收窗口消息并处理, (待矫正)浏览器进程会根据SiteInstance和BrowsingContext等来决定是否要新建一个渲染进程并决定发送到哪个渲染进程来处理请求. 渲染进程的RenderView把请求发给WebKit, 这个RequestResource的消息绕一圈(怎么处理的没有深入)又通过IPC发到浏览器进程(因为渲染进程在沙箱里没有权限访问资源)处理. 官网上的[network stack](http://www.chromium.org/developers/design-documents/network-stack)一节讲的就是这之后的事情.

## 浏览器进程 --> 渲染进程

**关系:**

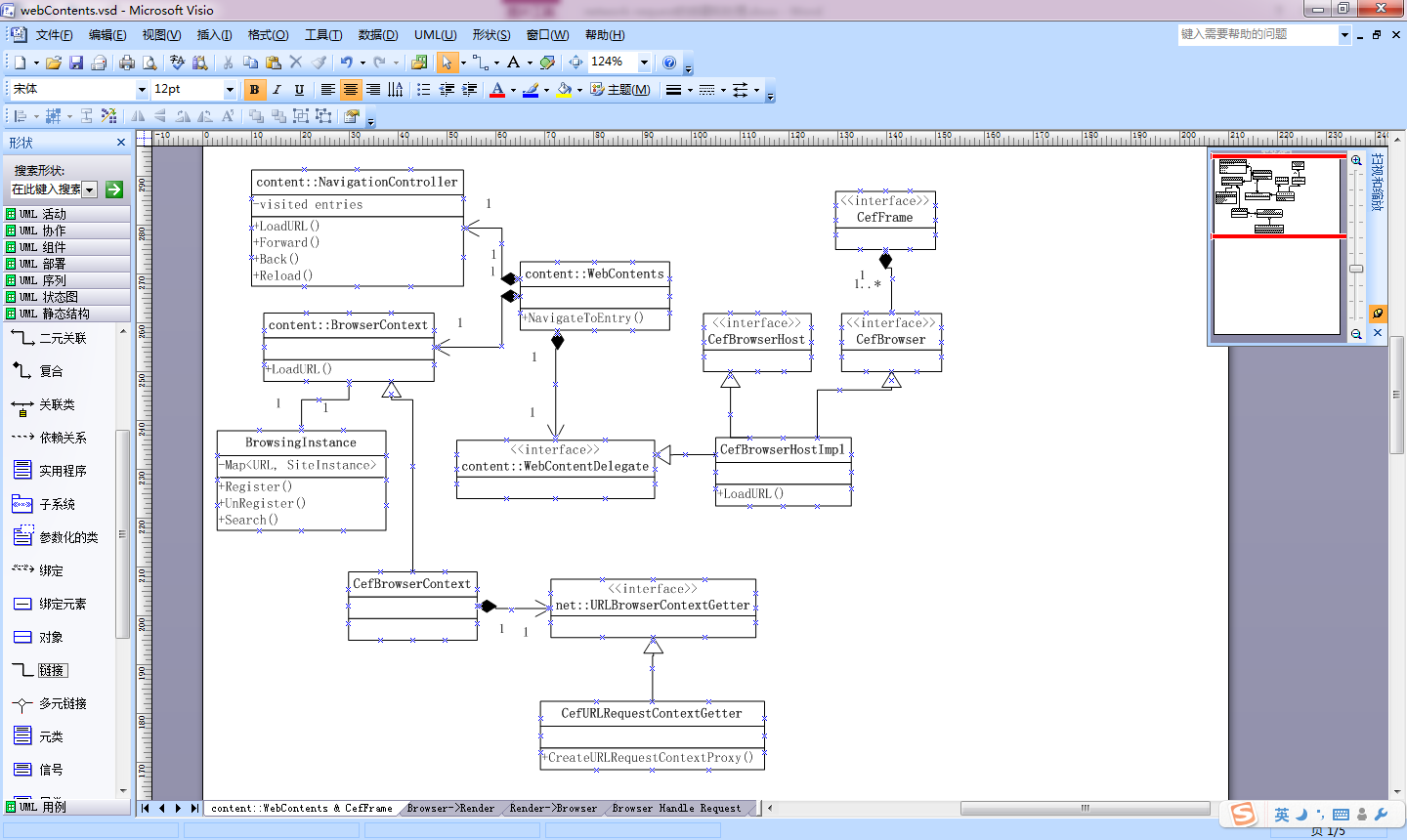
在Cef 3.0中, 一个WebContents对应一个CefFram, 即一个标签页和一个net::WebContents对应, 这是Content的负责渲染的核心类, 每个WebContents对应一个NavigationController用来加载URL和前进后退.

NavigationController接口：负责页面的导航逻辑：前进、后退、刷新等。其内部维护一个导航过的NavigationEntry列表.

WebContents创建的时候指定content::BrowserContext, CEF 3.0提供的CefURLRequestContextXXX几个类实现了net::URLRequestContextXXX几个接口, 用来在BrowserContext中获取URLRequestContext.

SiteInstance在BrowsingInstance注册备案, BrowsingInstance(理论上)与BorwsingContext一一对应. BrowsingInstance内部维护一个站点URL(如:baidu.com)到SiteInstance对象的映射，并提供注册/取消注册、查找的接口。相当于SiteInstance的cache对象。

BrowsingInstance对象为诸多SiteInstance对象所共享, 采用引用计数的方式控制生命周期.

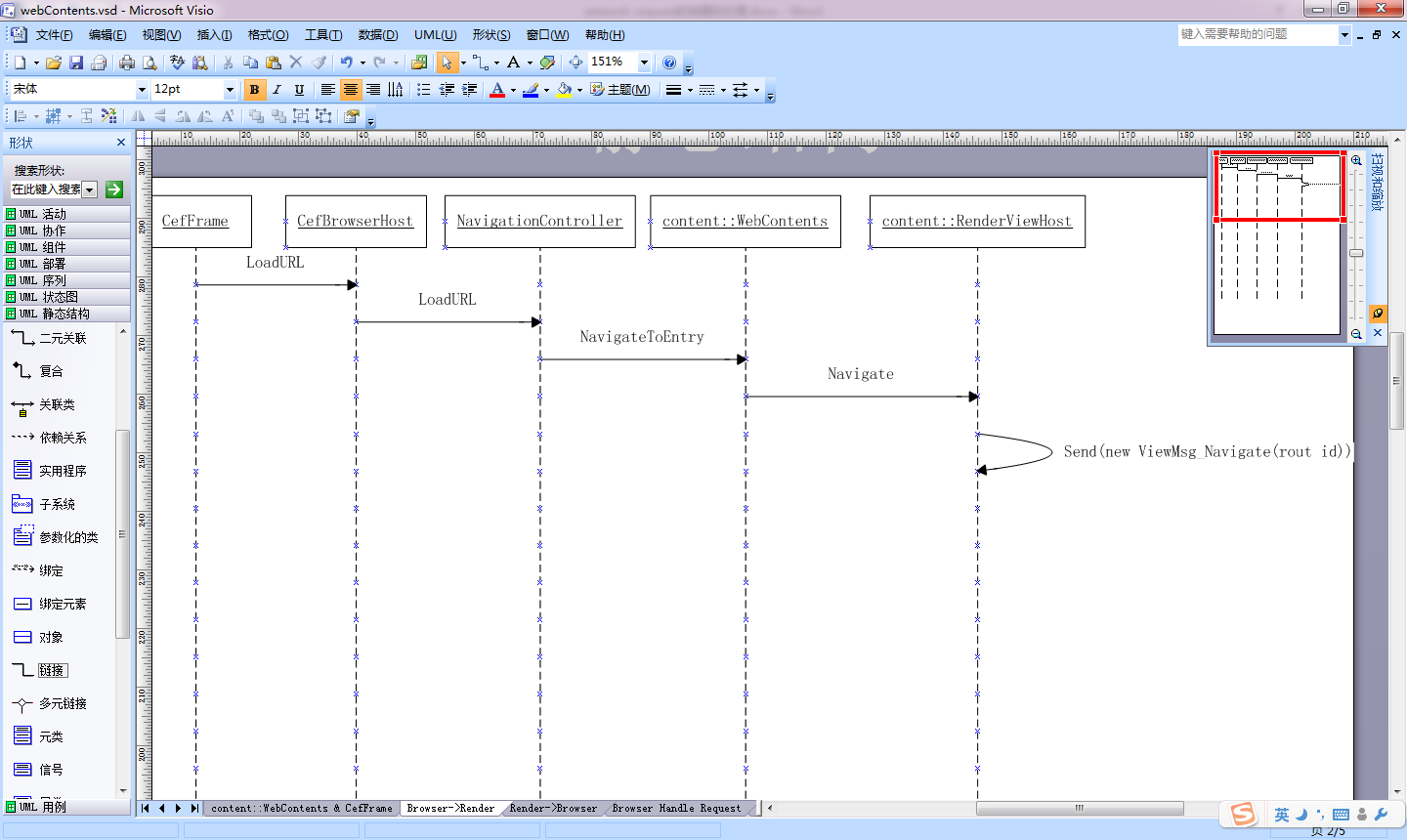


(WebContents和CefFrame的关系)

**过程:**

**使用cef时, CefFrame作为UI控件加载url, 最终调用CefBrowserHostImpl::LoadURL(). 对URL做一些矫正补充后, 发给web\_contents\_->GetController()->LoadURL(), web\_contents->getController()是navigation\_cotnroller, 这个controller会先做一些工作, 如果不是已经访问过的entry, 才会让web\_contents跳转过去.**

**这前面做了好多事情, 都在浏览器进程里, 最终浏览器进程调用WebContentsImpl::NavigateToEntry --> content::RenderViewHostImpl::Navigate(), 后者从浏览器进程向渲染进程发送一个"Send(new ViewMsg\_Navigate(GetRoutingID(), params));"的消息.**



## 渲染进程 --> 浏览器进程

**渲染进程的RenderViewImpl把对这类消息(ViewMsg\_Navigate)的处理放在RenderViewImpl::OnNavigate中. 这时已经在渲染进程里了.**

**在RenderViewImpl中, 我们要把请求的httpBody数据发送到浏览器进程, 并通过FrameLoader::loadFrameRequest在WebKit中提起一个特殊的POST跳转.**

**WebKit被命令加载URL时最终会调用WebURLLoader来异步加载(多数为异步加载), 使用者实现WebURLLoader接口来定制加载实现, 在Chrome中就是通过webkit\_glue::WebURLLoaderImpl::loadAsynchronously(), 这里会通过IPC发个消息给浏览器进程. 这一段的事情, 发生在(图1 – 多进程模型)里渲染进程绿色的ResourceDispatcher那一部分. 。**IPCResourceLoaderBridge继承自ResourceLoaderBridge，其作用是负责请求对象和回复对象的解释工作，实际消息的接受和派发交给ResourceDispatcher来处理

**IPCResourceLoaderBridge::Start(peer)**

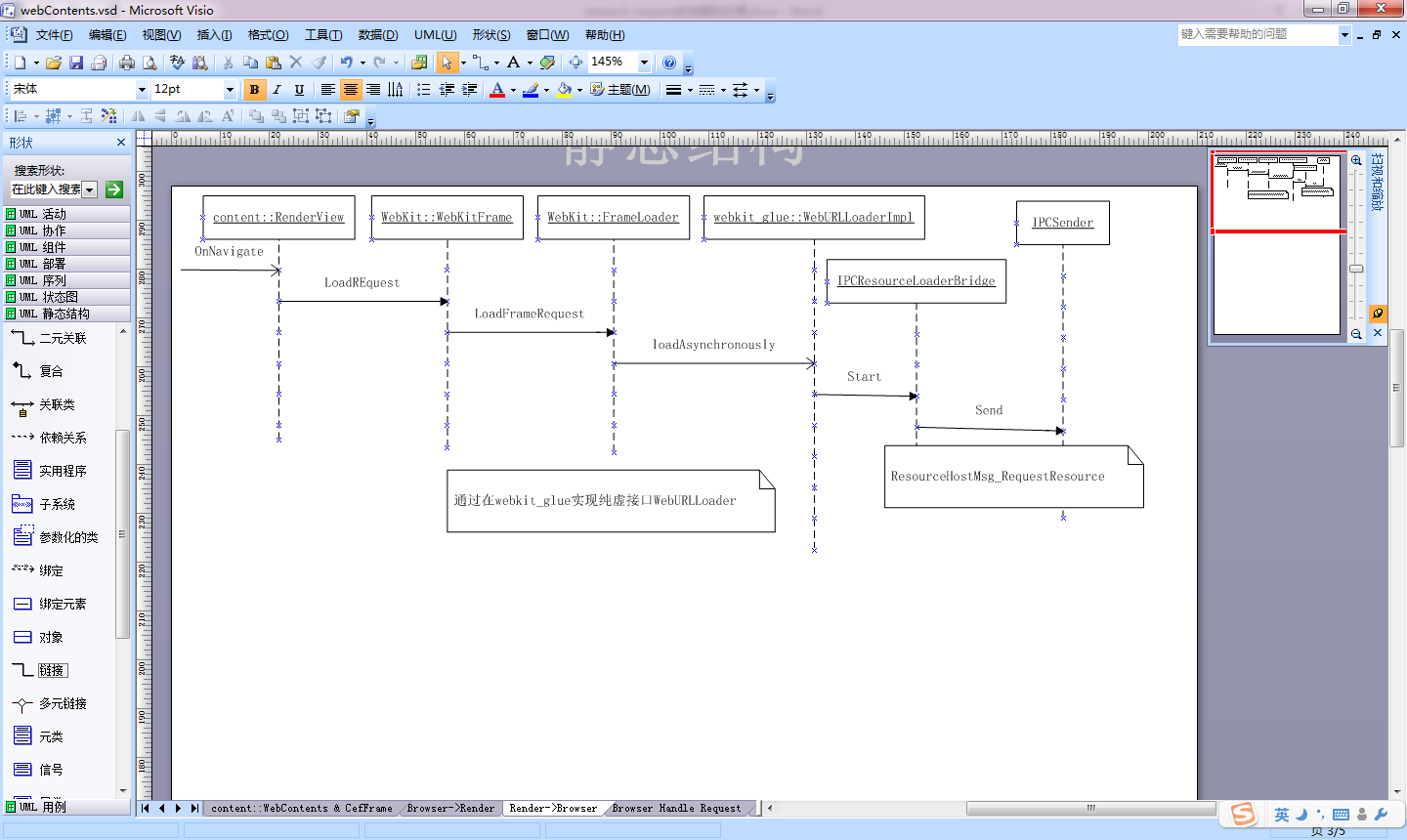
**{**

**...**

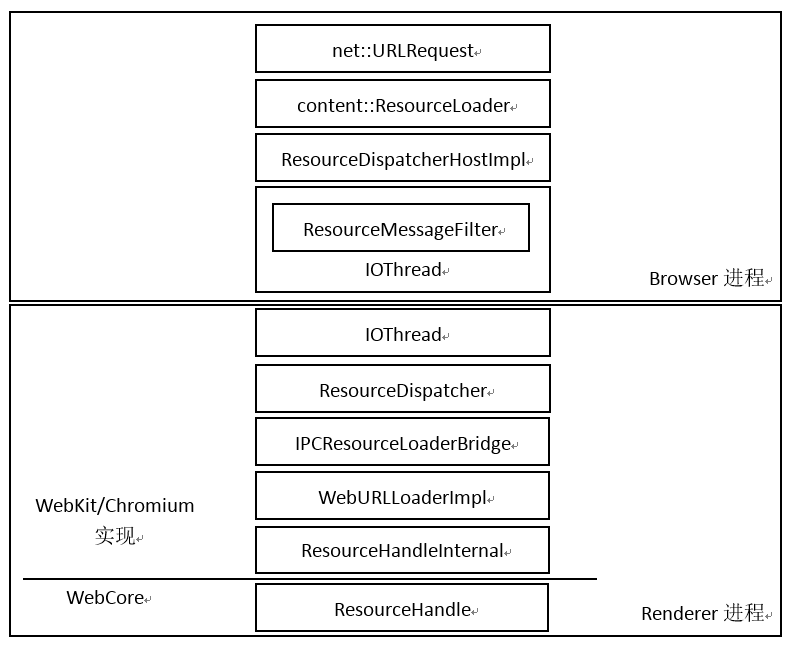
**IPCSender::Send(**

**new ResourceHostMsg\_RequestResource(routing\_id\_, request\_id\_, request\_)) //resource\_dispatcher.cc l:185**

**}**



**更清晰的类的调用关系如下(渲染进程 --> 浏览器进程)**



## ****浏览器进程对URLRequest的处理****

**这段期间在URLRequest下层的重要的类都在**[network stack](http://www.chromium.org/developers/design-documents/network-stack)**里有讲.**

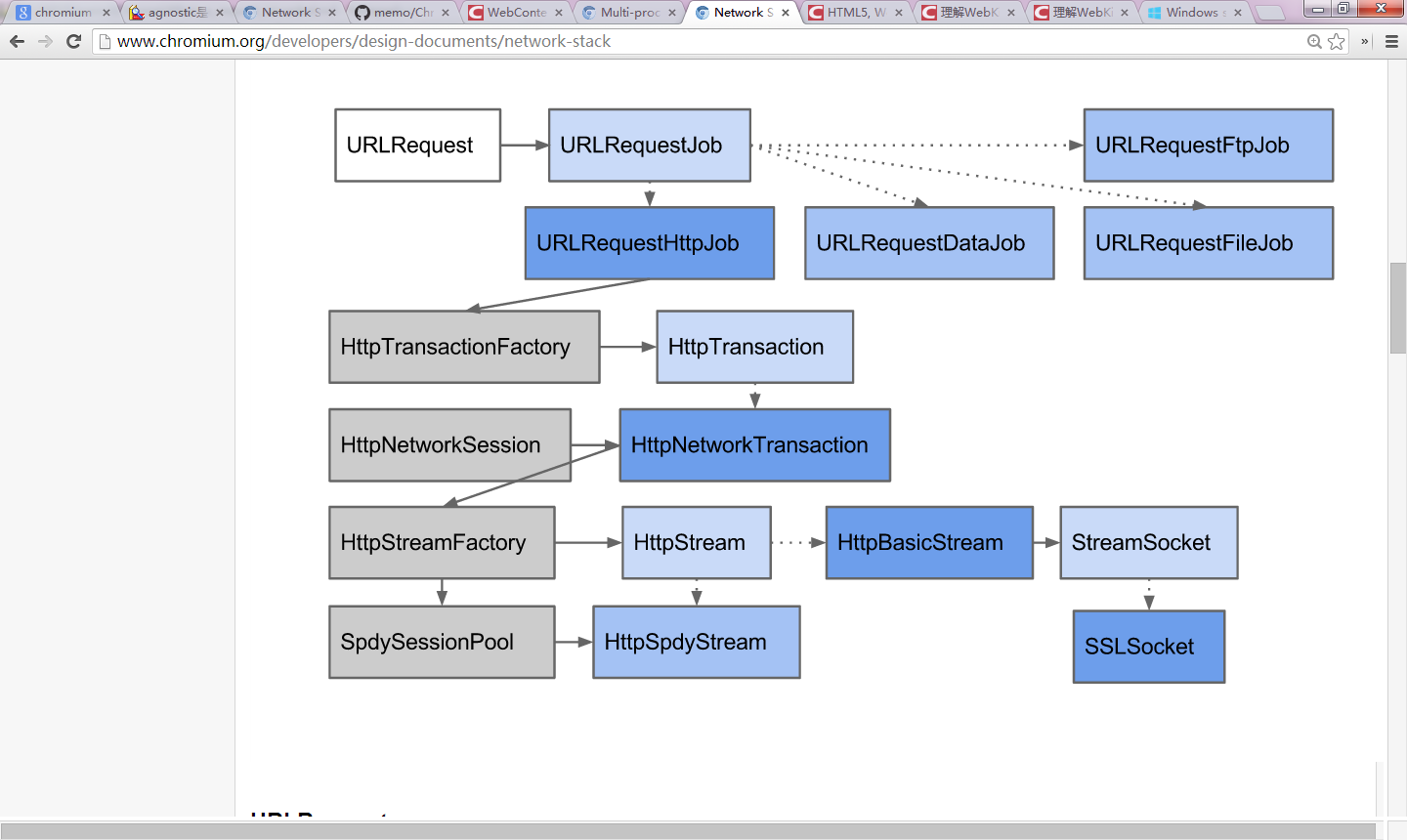
**在URLRequest上层(调用它的人)有几个重要的类要介绍一下.**

**多进程模型图1左上角那个绿色的ResourceDispatcherHost是个很重要的类, 这是一个浏览器进程的单例.** 在Browser进程中，首先由ResourceMessageFilter将Renderer进程的消息过滤，如果与资源请求相关，则转发给 ResourceDispatcherHostImp. **” ResourceHostMsg\_RequestResource ”消息会由 ResourceDispatcherHostImpl :: OnRequestResource会进行处理, 最终启动请求. (貌似同步异步都会调到ResourceDispatcherHostImpl::BeginRequest).**

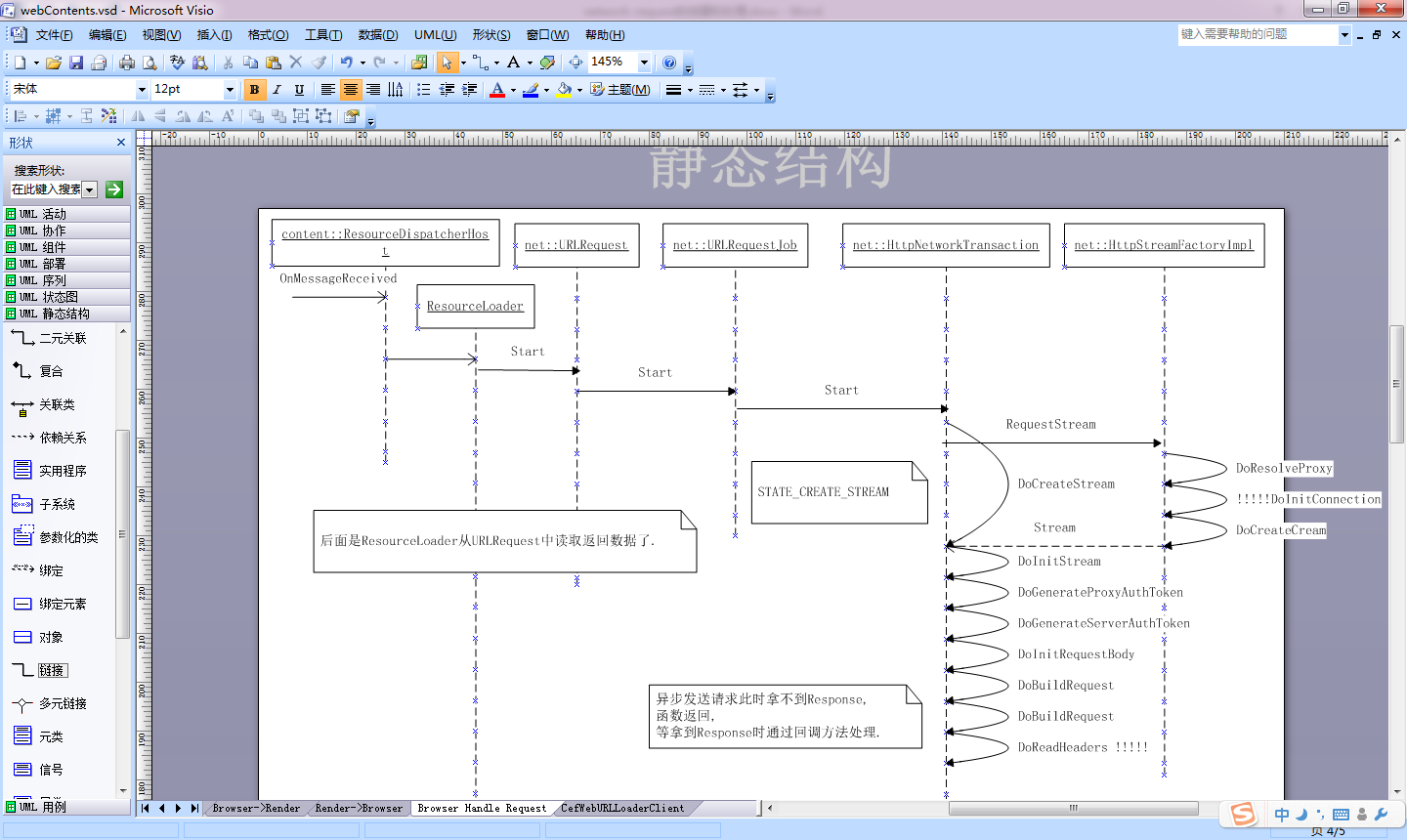
**ResourceDispatcherHostImpl接收到消息后会先做一些处理, scheme判断, 寻找ResourceLoader, 绑定ResourceHandler到URLRequest (ResourceHandler会处理这个URLRequest的网络事件, 使用SyncResourceHandle和AsyncResourceHandle来向渲染进程发送状态消息，并接受渲染进程对这些消息的反馈).**

**ResourceDispatcherHost最后会调用content::ResourceLoader来**承担Browser进程中的对资源的总体管理任务, 对于同步和异步两种资源请求方式**, 包括调用URLRequest->Start(). Content::ResourceLoader是Chrome实际的资源加载类, 它负责管理请求到网络, 从网络过来的认证请求, 服务器的回复等工作. 每项工作都有更细分的类来处理, 但是ResourceLoader来管理这些.**

1. **URLRequest启动之后, 如下图所示, 会先创建一个URLRequestJob, 在浏览网页的时候, 这是一个URLRequestHttpJob. URLRequestJob和它的工厂URLRequestJobFactory的管理工作都是URLRequestJobManager负责。基本的思路是，用户可以在该类中注册多个工厂，当有URLRequest请求时候，先有工厂检查它是否需要处理该scheme，如果没有，继续交由下一个工厂类。最后，如果没有任何工厂能够处理的话，则交给内置的工厂来检查和处理是否是http://，ftp://或者**<file://等>**.**
2. **URLRequestHttpJob被创建后，它首先从Cookie管理器中获取跟该URL相关联的信息。之后，它同样借助于HttpTransactionFactory创建一个HttpTransaction类的对象来表示开启一个HTTP连接的事务. 通常情况下，HttpTransactionFactory对应的是一个它的子类HttpCache的实例。HttpCache类使用本地磁盘缓存机制（稍后会介绍），如果该请求对应的回复已经在磁盘缓存中，那么无需再建立HttpTransaction来发起连接，直接从磁盘中获取即可。如果磁盘中没有，同时如果目前该URL请求对应的HttpTransaction已经建立，那么只要等待它的回复即可。这些条件都不满足后，实际上才会真正创建HttpTransaction。**



1. **HttpNetworkTransaction使用HttpNetworkSession来管理连接会话. HttpNetworkSession通过它的成员HttpStreamFactory来建立TCP Socket连接，之后创建HttpStream对象. HttpStreamFactory将和网络之间的数据读写交给自己新创建的一个HttpStream (其实是它的子类) 对象来处理. HttpStream发送请求后负责读取回复数据.**



### ****优化处理****

chromium为了提升访问速度所做的优化

<http://aosabook.org/en/posa/high-performance-networking-in-chrome.html>