**（本文从网上整理得到，源地址**<http://blog.csdn.net/caimouse/archive/2008/09/07/2893806.aspx>**）**

**谷歌浏览器的源码分析(1)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

随着网络技术的发展，越来越多应用都已经离不开网络，特别像人类大脑一样的知识库的搜索引擎，更加是离不开功能强大的云计算。不过，即便云计算非常强大，但它还不能直接地把结果呈现给用户，这样就需要一个客户端来呈现出来，这个客户端就是浏览器。现在越来越多人上网，他们每一次上网，都离不开浏览的使用，这已经是一个不可缺少的软件了。这里介绍和分析谷歌推出有创新的浏览器，它的速度比其它浏览器快很多，那么它是怎么实现的呢？又采用了什么样的技术能达到这样呢？又比如它的标签页是每一个进程进行显示的，这到底又是怎么样实现的呢？下面来通过分析它的源码，一一地解开这种高新技术的使用，以及这种高效算法的奥秘。

谷歌浏览器的英语名称为Chrome，它的意义是铬。铬是一种有光泽的、蓝灰色的坚硬金属元素。不失光泽，抗腐蚀，最早在铬铁矿中发现。用作催化剂，可加强钢合金的强度和生产不锈钢，可以做防腐镀层和玻璃制品中的颜料。原子序数24；原子量51.996；比重7.18；化合价2，3，6。谷歌起这个名称，可能是想让这个浏览器永远不失去光泽，永远那么吸引人。铬是无毒，化学性质很稳定，有延展性，含杂质时硬而脆。熔点1857C，沸点2672C，密度单晶为7.22克/厘米3，多晶为7.14克/厘米3；铬，原子序数24，原子量51.9961。铬的名称来自希腊文Chroma，意为颜色。因为这种元素以多种不同颜色的化合物存在，故被称为“多彩的元素”。可用于制不锈钢，汽车零件，工具，磁带和录像带等。铬镀在金属上可以防锈，也叫**可多米**，坚固美观。红、绿宝石的色彩也来自于铬。作为现代科技中最重要的金属，以不同百分比熔合的铬镍钢千变万化，种类繁多，令人难以置信。

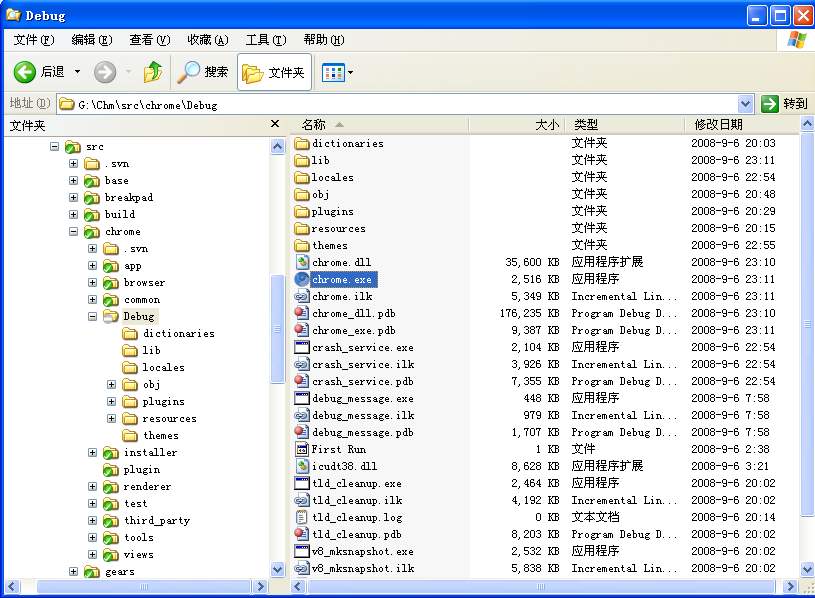
谷歌的开发人员称，虽然网络的发展日新月异，但作为网络平台的浏览器，却没有跟上网络发展的步伐。谷歌倾心打造的免费浏览器就是希望能跟随着网络的发展而不断升级换代，完美的切合网络时代的潮流。

据了解，谷歌员工每天使用的最多的应用程序就是浏览器，通过浏览器，查看新闻资讯，观看视频聊天，玩网络游戏。谷歌的员工说，如果能够开发出一种全新的浏览器，才能够满足人们使用应用程序和网站管理员的要求。谷歌希望能够提供一种**速度更快，稳定性更高，安全性更强**的浏览器。因此Google Chrome诞生了！

为了学习这个浏览器，需要通过网络把这份达到1G以上的代码下载下来，需要的时间就需要好几个小时，然后再把硬盘空间清空为10G左右大小，最后配置好VC 2005，就可以编译这个“可多米”了。在我的电脑上编译，共需要两个小时左右，才完全编译完成，最后生成下面的可多米，如下图：



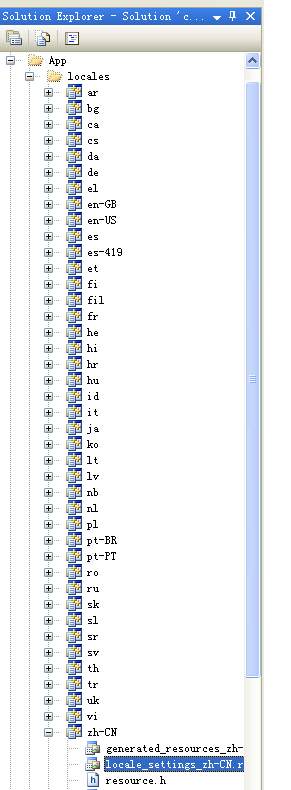
缺省编译出来的可多米是英语版本的，从关于对话框里就可以看到。下面是编译出来的目录图片，如下：



上面是调试版本的输出文件，所以程序大小都比较大，没有经过优化的处理。整个程序的大小，需要编译137个工程，共1G多的源码大小，这是一个非常旁大的一个工程。

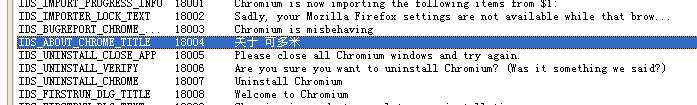
# 谷歌浏览器的源码分析(2) [收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

这么大的工程，我从哪里开始呢？我认为从界面开始，这样才可以快速地深入研究。下面就可以先尝试修改一个chrome的关于对话框，上一次看到它是英语的，那么我就来把它改成中文的吧，这样有目标了。从chrome的工程里可以看到它是支持多种语言的，在Windows平台上支持多语言的标准做法，就是写多个语言的DLL。因此，chrome也不例外，从app工程集里，就可以看到如下图所示：



上面显示了多种语言的动态连接库资源，其中zh-CN是简体中文的。

接着打开资源文件的字符串编辑，如下图：



把上面的字符串修改为 “关于 可多米”，然后把这个工程重新编译一下，就会生成下面的文件：



然后运行自己编译的可多米，就会显示出修改的成果，如下图：



可以看到关于对话框的标题，就变成我上面修改的了。这样学习它的修改，就是几分钟的事情，哈哈......

这样就学习了可多米开发汉化的工作，这是本地化的重要做的一件事情，也学习到怎么样支持多语言的实现方式。那么它的关于对话框是从那里显示出来的呢？怎么样把字符串更新到上面的呢？下一次再告诉你。

# 谷歌浏览器的源码分析(3) [收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

前面修改了chrome关于对话框，并且编译显示出来了，那么它是在那里调用显示的呢？现在就带你去了解它。由于它是界面显示，那么不用想，直接到界面的工程里查找它，也就是到目录src\chrome\browser\views里查看到文件about\_chrome\_view.cc。

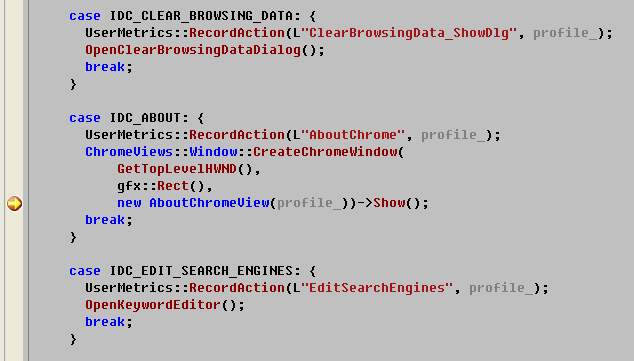
这个文件里声明了一个类AboutChromeView，它就是主要负责初始化对话框、布局、显示字符串等等，比如显示“关于可乐米”的字符串，就是这样实现的，先调用函数：

#001  std::wstring AboutChromeView::GetWindowTitle() const {

  return l10n\_util::GetString(IDS\_ABOUT\_CHROME\_TITLE);

}

获取资源里的对话框标题，接着：

在上面的断点里就是响应菜单，然后创建关于对话框，主要调用函数CreateChromeWindow来创建窗口，把AboutChromeView窗口绑定到这个窗口类型里。由于可多米都是统一的窗口样式，那么它是通过创建一样的窗口类CustomFrameWindow来实现的。

为了显示窗口的标题，是通过下面的函数关系调用：

1．  Browser::ExecuteCommand  浏览器执行菜单命令。

2．  ChromeViews::Window::CreateChromeWindow  创建窗口。

3．  ChromeViews::CustomFrameWindow::Init 初始化窗口。

4．  ChromeViews::Window::Init  初始化窗口标题。

5．  AboutChromeView::GetWindowTitle  从关于对话框获取标题。

理解上面的函数关系调用就知道怎么样显示标题了，因此也知道关于对话框所有内容是由类AboutChromeView来管理的，但窗口的样式是由CustomFrameWindow类来管理的。

通过上面的分析，了解了关于对话框的标题显示过程，你想修改成什么样的内容，就要看你的需要了。下一次再仔细地分析有关于对话框怎么样组织其它信息，比如重要的升级功能。

**谷歌浏览器的源码分析(4)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

关于对话框，主要实现了让用户查看当前软件的版本、软件信息和检查升级的功能。因此这个类主要继续ChromeViews::View类、ChromeViews::DialogDelegate和GoogleUpdateStatusListener。其中ChromeViews::View实现窗口的布局和显示问题，ChromeViews::DialogDelegate实现了事件响应，或者窗口某时是否可以显示按钮的问题，GoogleUpdateStatusListener是用来实现接收更新程序状态信息。

这个类的声明如下：

class AboutChromeView : public ChromeViews::View,

                        public ChromeViews::DialogDelegate,

                        public GoogleUpdateStatusListener {

 public:

  explicit AboutChromeView(Profile\* profile);

  virtual ~AboutChromeView();

  // Initialize the controls on the dialog.

  void Init();

  // Overridden from ChromeViews::View:

  virtual void GetPreferredSize(CSize \*out);

  virtual void Layout();

  virtual void ViewHierarchyChanged(bool is\_add,

                                    ChromeViews::View\* parent,

                                    ChromeViews::View\* child);

  // Overridden from ChromeViews::DialogDelegate:

  virtual int GetDialogButtons() const;

  virtual std::wstring GetDialogButtonLabel(DialogButton button) const;

  virtual bool IsDialogButtonEnabled(DialogButton button) const;

  virtual bool IsDialogButtonVisible(DialogButton button) const;

  virtual bool CanResize() const;

  virtual bool CanMaximize() const;

  virtual bool IsAlwaysOnTop() const;

  virtual bool HasAlwaysOnTopMenu() const;

  virtual bool IsModal() const;

  virtual std::wstring GetWindowTitle() const;

  virtual bool Accept();

  virtual ChromeViews::View\* GetContentsView();

  // Overridden from GoogleUpdateStatusListener:

  virtual void OnReportResults(GoogleUpdateUpgradeResult result,

                               GoogleUpdateErrorCode error\_code,

                               const std::wstring& version);

 private:

  // The visible state of the Check For Updates button.

  enum CheckButtonStatus {

    CHECKBUTTON\_HIDDEN = 0,

    CHECKBUTTON\_DISABLED,

    CHECKBUTTON\_ENABLED,

  };

  // Update the UI to show the status of the upgrade.

  void UpdateStatus(GoogleUpdateUpgradeResult result,

                    GoogleUpdateErrorCode error\_code);

  Profile\* profile\_;

  // UI elements on the dialog.

  ChromeViews::ImageView\* about\_dlg\_background\_;

  ChromeViews::Label\* about\_title\_label\_;

  ChromeViews::TextField\* version\_label\_;

  ChromeViews::TextField\* main\_text\_label\_;

  // UI elements we add to the parent view.

  scoped\_ptr<ChromeViews::Throbber> throbber\_;

  ChromeViews::ImageView success\_indicator\_;

  ChromeViews::ImageView update\_available\_indicator\_;

  ChromeViews::ImageView timeout\_indicator\_;

  ChromeViews::Label update\_label\_;

  // Keeps track of the visible state of the Check For Updates button.

  CheckButtonStatus check\_button\_status\_;

  // The class that communicates with Google Update to find out if an update is

  // available and asks it to start an upgrade.

  GoogleUpdate\* google\_updater\_;

  // Our current version.

  std::wstring current\_version\_;

  // The version Google Update reports is available to us.

  std::wstring new\_version\_available\_;

  DISALLOW\_EVIL\_CONSTRUCTORS(AboutChromeView);

};

通过关于对话框的分析，可以理解到chrome浏览器窗口基本组成，以及窗口继承关系，还有事件的响应方式。前面三次分析，主要是入门的分析，也是了解这么一个大工程的一种手段。比如测试整个工程是否可以编译，是否可以修改代码等等。后面的分析会以浏览器输入HTTP连接开始，直到打开网页显示为一个主线，做一个基本的分析。

**谷歌浏览器的源码分析(5)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

当用户打开浏览器之后，最希望输入的地方，是浏览器的连接框。目前谷歌浏览器把输入连接框与搜索引擎输入合并到一起，可以说完美的组合，让界面更加简洁，方便实用，并且它自动完成的功能更加强劲，如下图所示：



上面输入了[www.c](http://www.c/)时，它就会自动地在后面添加智能选择的连接，并且可以GOOGLE里搜索输入的内容，又如下面：



上面在输入框里输入我的名字，就会自动弹出查找的内容，或者可能搜索的连接。这些功能都比较完美的实现，这可以说是史无前例的输入创新，真正人性化的体现，那么它又是怎么样实现的呢？其主要功能是在文件src\chrome\browser\autocomplete\autocomplete\_edit.cc里实现，具体的实现方式内容等下一次再去分析。

**谷歌浏览器的源码分析(6)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

前面已经介绍了这么引人的输入自动完成功能，并且可以在输入超级连接框里直接通过GOOGLE搜索所有的内容，这是比较大的创新，不但可以节省界面的占用面积，还很方便大家查询的需要，比如记不住的连接，根本不需要去记了，只要你记住需要的内容就行了。这样既不需要到什么门户网站去找连接，也不需要去记住众多的网站，这个功能是非常方便的。

这个输入框的自动完成的功能，是比较智能化的。因为它会根据以往的输入自动完成，或者智能提示所需要的连接或者内容。

下面就来先看这个类的定义：

#001  // Provides the implementation of an edit control with a drop-down

#002  // autocomplete box. The box itself is implemented in autocomplete\_popup.cc

#003  // This file implements the edit box and management for the popup.

#004  //

#005  // This implementation is currently appropriate for the URL bar, where the

#006  // autocomplete dropdown is always displayed because there is always a

#007  // default item. For web page autofill and other applications, this is

#008  // probably not appropriate. We may want to add a flag to determine which

#009  // of these modes we're in.

#010  class AutocompleteEdit

#011      : public CWindowImpl<AutocompleteEdit,

#012                           CRichEditCtrl,

#013                           CWinTraits<WS\_CHILD | WS\_VISIBLE | ES\_AUTOHSCROLL |

#014                                      ES\_NOHIDESEL> >,

#015        public CRichEditCommands<AutocompleteEdit>,

#016        public Menu::Delegate {

类AutocompleteEdit继承了类CWindowImpl、类CRichEditCommands、类Menu::Delegate。其中类CWindowImpl实现了Windows窗口，它是WTL里的窗口模板类，主要用来创建窗口界面类，并且使用类CRichEditCtrl作为基类，类CRichEditCtrl主要调用Windows里的编辑类。类CRichEditCommands实现RichEdit的命令功能。Menu::Delegate类是实现智能下拉式菜单的提示界面。因此，要学习开发chrome，需要先学习WTL的开发，它是一套基于模板的窗口框架。下一次再仔细地分析自动完成的实现过程。

**谷歌浏览器的源码分析(7)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

当我们键入字母或者文字开始时，那么类AutocompleteEdit就会从窗口消息里获取到相应的字母或者文字，然后根据输入的信息到本地或者网络上保存的信息库里查找相应的输入提示，这就是自动完成的实现。下面就来先分析输入的函数：

#001  void AutocompleteEdit::OnChar(TCHAR ch, UINT repeat\_count, UINT flags) {

#002    // Don't let alt-enter beep.  Not sure this is necessary, as the standard

#003    // alt-enter will hit DiscardWMSysChar() and get thrown away, and

#004    // ctrl-alt-enter doesn't seem to reach here for some reason?  At least not on

#005    // my system... still, this is harmless and maybe necessary in other locales.

下面把alt-enter组合键消息过滤掉。

#006    if (ch == VK\_RETURN && (flags & KF\_ALTDOWN))

#007      return;

#008

#009    // Escape is processed in OnKeyDown.  Don't let any WM\_CHAR messages propagate

#010    // as we don't want the RichEdit to do anything funky.

下面把ESC键的消息过滤掉。

#011    if (ch == VK\_ESCAPE && !(flags & KF\_ALTDOWN))

#012      return;

#013

下面把TAB键的消息过滤掉。

#014    if (ch == VK\_TAB) {

#015      // Don't add tabs to the input.

#016      return;

#017    }

#018

这里处理其它有用的按键消息。

#019    HandleKeystroke(GetCurrentMessage()->message, ch, repeat\_count, flags);

#020  }

AutocompleteEdit::OnChar函数是WTL里的WM\_CHAR消息处理，当用户键入字母时就会触发这个消息。这个函数先跳过几个不要处理的消息，最后调用函数HandleKeystroke来处理，如下：

#001  void AutocompleteEdit::HandleKeystroke(UINT message, TCHAR key,

#002                                         UINT repeat\_count, UINT flags) {

冻结RichEdit的更新。

#003    ScopedFreeze freeze(this, GetTextObjectModel());

处理消息变化前的动作。

#004    OnBeforePossibleChange();

处理消息

#005    DefWindowProc(message, key, MAKELPARAM(repeat\_count, flags));

处理消息变化后的动作。

#006    OnAfterPossibleChange();

#007  }

在这里为什么要进行窗口的消息冻结呢？又为什么需要进行消息处理和消息变化后处理呢？

**谷歌浏览器的源码分析(8)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到处理WM\_CHAR消息，当用户每键入一个字符时，万能连接框就会去进行一次查找的过程，然后把智能提示信息显示出来。说到AutocompleteEdit::HandleKeystroke函数的操作，那么它为什么需要冻结这个函数的使用呢？现在就来分析这部份的内容。如下：

ScopedFreeze freeze(this, GetTextObjectModel());

在这行代码里，首先会调用函数GetTextObjectModel()来获取一个文档ITextDocument接口，然后再使用它的功能。这个函数的代码如下：

#001  ITextDocument\* AutocompleteEdit::GetTextObjectModel() const {

先判断这个接口是否获取到，如果已经获取到就不再去重复获取了。

#002    if (!text\_object\_model\_) {

#003      // This is lazily initialized, instead of being initialized in the

#004      // constructor, in order to avoid hurting startup performance.

这里使用了智能指针来获取IRichEditOle接口。

#005      CComPtr<IRichEditOle> ole\_interface;

获取到的IRichEditOle接口绑定到智能指针里。

#006      ole\_interface.Attach(GetOleInterface());

下面通过=操作符获取ITextDocument接口，如果你深入去分析这个赋值操作符，会看到它自动去调用IRichEditOle的接口IUnknown::QueryInterface来查询到ITextDocument接口，这个过程对于程序员来说是完全不用关心的，这就是使用mutable CComQIPtr<ITextDocument> text\_object\_model\_定义的作用。

#007      text\_object\_model\_ = ole\_interface;

#008    }

#009    return text\_object\_model\_;

#010  }

通过上面的分析，可见使用CComQIPtr<ITextDocument>智能指针可以省了很多COM调用的操作，这真是模板类的强大功能的使用之处。当把ITextDocument接口获取回来之后，对于RichEdit操作就可以轻松访问了，ScopedFreeze类生成一个局部对象，这个对象实现了对RichEdit自动冻结和解冻结的功能，这个过程是通过局部对象在栈里生命周期的特性应用。如下面的代码：

#001  AutocompleteEdit::ScopedFreeze::ScopedFreeze(AutocompleteEdit\* edit,

#002                                               ITextDocument\* text\_object\_model)

#003      : edit\_(edit),

#004        text\_object\_model\_(text\_object\_model) {

#005    // Freeze the screen.

#006    if (text\_object\_model\_) {

#007      long count;

**#008      text\_object\_model\_->Freeze(&count);**

#009    }

#010  }

#011

#012  AutocompleteEdit::ScopedFreeze::~ScopedFreeze() {

#013    // Unfreeze the screen.

#014    // NOTE: If this destructor is reached while the edit is being destroyed (for

#015    // example, because we double-clicked the edit of a popup and caused it to

#016    // transform to an unconstrained window), it will no longer have an HWND, and

#017    // text\_object\_model\_ may point to a destroyed object, so do nothing here.

#018    if (edit\_->IsWindow() && text\_object\_model\_) {

#019      long count;

**#020      text\_object\_model\_->Unfreeze(&count);**

#021      if (count == 0) {

这里需要手动地更新窗口的显示。

#022        // We need to UpdateWindow() here instead of InvalidateRect() because, as

#023        // far as I can tell, the edit likes to synchronously erase its background

#024        // when unfreezing, thus requiring us to synchronously redraw if we don't

#025        // want flicker.

#026        edit\_->UpdateWindow();

#027      }

#028    }

#029  }

从上面的代码可以看到构造函数里冻结，析构造函数里解冻结，如果需要就会自动更新窗口的显示。

通过上面的分析，学会使用RichEdit的冻结窗口的输入，并且解冻结和更新窗口的显示，也同时学会使用智能指针来操作COM接口的方便性，最后还学会了使用栈对象的生命周期来方便对加锁和解锁的操作，以便降低代码的出错率。

**谷歌浏览器的源码分析(9)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

为了处理字符消息实现自动完成的功能，这是怎么样实现的呢？其实是先记录字符消息响应前的字符串以及选中状态，接着再处理消息，最后才查询可能的输入，做出智能提示。

#001  void AutocompleteEdit::OnBeforePossibleChange() {

#002    // Record our state.

记录当前已经输入的字符串。

#003    text\_before\_change\_ = GetText();

记录当前选中的字符位置。

#004    GetSelection(sel\_before\_change\_);

#005    select\_all\_before\_change\_ = IsSelectAll(sel\_before\_change\_);

#006  }

上面就保存字符消息响应前的状态，接着下来就是消息响应后的处理了，如下：

#001  bool AutocompleteEdit::OnAfterPossibleChange() {

#002    // Prevent the user from selecting the "phantom newline" at the end of the

#003    // edit.  If they try, we just silently move the end of the selection back to

#004    // the end of the real text.

判断用户新选中状态。

#005    CHARRANGE new\_sel;

#006    GetSelection(new\_sel);

#007    const int length = GetTextLength();

#008    if ((new\_sel.cpMin > length) || (new\_sel.cpMax > length)) {

#009      if (new\_sel.cpMin > length)

#010        new\_sel.cpMin = length;

#011      if (new\_sel.cpMax > length)

#012        new\_sel.cpMax = length;

#013      SetSelectionRange(new\_sel);

#014    }

判断用户是否输入字符有变化。

#015    const bool selection\_differs = (new\_sel.cpMin != sel\_before\_change\_.cpMin) ||

#016        (new\_sel.cpMax != sel\_before\_change\_.cpMax);

#017

#018    // See if the text or selection have changed since OnBeforePossibleChange().

#019    const std::wstring new\_text(GetText());

#020    const bool text\_differs = (new\_text != text\_before\_change\_);

#021

#022    // Update the paste state as appropriate: if we're just finishing a paste

#023    // that replaced all the text, preserve that information; otherwise, if we've

#024    // made some other edit, clear paste tracking.

#025    if (paste\_state\_ == REPLACING\_ALL)

#026      paste\_state\_ = REPLACED\_ALL;

#027    else if (text\_differs)

#028      paste\_state\_ = NONE;

#029

如果输入没有任何变化，就返回去。

#030    // If something has changed while the control key is down, prevent

#031    // "ctrl-enter" until the control key is released.  When we do this, we need

#032    // to update the popup if it's open, since the desired\_tld will have changed.

#033    if ((text\_differs || selection\_differs) &&

#034        (control\_key\_state\_ == DOWN\_WITHOUT\_CHANGE)) {

#035      control\_key\_state\_ = DOWN\_WITH\_CHANGE;

#036      if (!text\_differs && !popup\_->is\_open())

#037        return false;  // Don't open the popup for no reason.

#038    } else if (!text\_differs &&

#039        (inline\_autocomplete\_text\_.empty() || !selection\_differs)) {

#040      return false;

#041    }

#042

#043    const bool had\_keyword = !is\_keyword\_hint\_ && !keyword\_.empty();

#044

下面开始设置新的显示字符串。

#045    // Modifying the selection counts as accepting the autocompleted text.

#046    InternalSetUserText(UserTextFromDisplayText(new\_text));

#047    has\_temporary\_text\_ = false;

#048

#049    if (text\_differs) {

#050      // When the user has deleted text, don't allow inline autocomplete.  Make

#051      // sure to not flag cases like selecting part of the text and then pasting

#052      // (or typing) the prefix of that selection.  (We detect these by making

#053      // sure the caret, which should be after any insertion, hasn't moved

#054      // forward of the old selection start.)

#055      just\_deleted\_text\_ = (text\_before\_change\_.length() > new\_text.length()) &&

#056        (new\_sel.cpMin <= std::min(sel\_before\_change\_.cpMin,

#057                                   sel\_before\_change\_.cpMax));

#058

#059      // When the user doesn't have a selected keyword, deleting text or replacing

#060      // all of it with something else should reset the provider affinity.  The

#061      // typical use case for deleting is that the user starts typing, sees that

#062      // some entry is close to what he wants, arrows to it, and then deletes some

#063      // unnecessary bit from the end of the string.  In this case the user didn't

#064      // actually want "provider X", he wanted the string from that entry for

#065      // editing purposes, and he's no longer looking at the popup to notice that,

#066      // despite deleting some text, the action we'll take on enter hasn't changed

#067      // at all.

这里删除已经选择的提示。

#068      if (!had\_keyword && (just\_deleted\_text\_ || select\_all\_before\_change\_)) {

#069        popup\_->manually\_selected\_match\_.Clear();

#070      }

#071    }

#072

#073    // Disable the fancy keyword UI if the user didn't already have a visible

#074    // keyword and is not at the end of the edit.  This prevents us from showing

#075    // the fancy UI (and interrupting the user's editing) if the user happens to

#076    // have a keyword for 'a', types 'ab' then puts a space between the 'a' and

#077    // the 'b'.

#078    disable\_keyword\_ui\_ = (is\_keyword\_hint\_ || keyword\_.empty()) &&

#079        ((new\_sel.cpMax != length) || (new\_sel.cpMin != length));

#080

更新智能提示菜单。

#081    UpdatePopup();

#082

#083    if (!had\_keyword && !is\_keyword\_hint\_ && !keyword\_.empty()) {

#084      // Went from no selected keyword to a selected keyword. Set the affinity to

#085      // the keyword provider.  This forces the selected keyword to persist even

#086      // if the user deletes all the text.

#087      popup\_->manually\_selected\_match\_.Clear();

#088      popup\_->manually\_selected\_match\_.provider\_affinity =

#089          popup\_->autocomplete\_controller()->keyword\_provider();

#090    }

#091

当自动完成框字符串发生变化，就需要更新URL重点显示。

#092    if (text\_differs)

#093      TextChanged();

#094

#095    return true;

#096  }

在这个函数里，先判断字符串是否发生变化，然后根据变化来决定是否更新编辑框的显示，同时还需要UpdatePopup更新智能提示菜单，最后判断是否有一个URL地址，如果有就重点显示出来。

其实这里最关键的问题就是智能菜单的数据从那里来的呢？怎么样根据用户的输入查找到最合适的提示呢？下一次我们再来分析这方面的问题。

**谷歌浏览器的源码分析(10)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次已经分析到输入字符后，就需要把这些关键字去查找历史的连接，或者相关的内容，那么可多米的浏览器又是从那里去找到这些数据呢？现在就来分析这方面相关的内容。它主要通下面的函数来实现：

#001  void AutocompleteEdit::UpdatePopup() {

冻结输入。

#002    ScopedFreeze freeze(this, GetTextObjectModel());

设置正在输入过程中。

#003    SetInputInProgress(true);

#004

如果输入的EDIT框没有焦点，就直接返回。

#005    if (!has\_focus\_) {

#006      // When we're in the midst of losing focus, don't rerun autocomplete.  This

#007      // can happen when losing focus causes the IME to cancel/finalize a

#008      // composition.  We still want to note that user input is in progress, we

#009      // just don't want to do anything else.

#010      //

#011      // Note that in this case the ScopedFreeze above was unnecessary; however,

#012      // we're inside the callstack of OnKillFocus(), which has already frozen the

#013      // edit, so this will never result in an unnecessary UpdateWindow() call.

#014      return;

#015    }

#016

#017    // Figure out whether the user is trying to compose something in an IME.

判断是否从输入法打开，如果是就从输入法窗口里获取字符串。

#018    bool ime\_composing = false;

#019    HIMC context = ImmGetContext(m\_hWnd);

#020    if (context) {

#021      ime\_composing = !!ImmGetCompositionString(context, GCS\_COMPSTR, NULL, 0);

#022      ImmReleaseContext(m\_hWnd, context);

#023    }

#024

#025    // Don't inline autocomplete when:

#026    //   \* The user is deleting text

#027    //   \* The caret/selection isn't at the end of the text

#028    //   \* The user has just pasted in something that replaced all the text

#029    //   \* The user is trying to compose something in an IME

获取当前选择的字符串。

#030    CHARRANGE sel;

#031    GetSel(sel);

根据用户输入的字符串来查找智能提示菜单的内容。

#032    popup\_->StartAutocomplete(user\_text\_, GetDesiredTLD(),

#033        just\_deleted\_text\_ || (sel.cpMax < GetTextLength()) ||

#034        (paste\_state\_ != NONE) || ime\_composing);

#035  }

在这个函数里主要调用类AutocompletePopupModel的函数StartAutocomplete来完成智能提示。而类AutocompletePopupModel的声明如下：

class AutocompletePopupModel : public ACControllerListener, public Task {

 public:

  AutocompletePopupModel(const ChromeFont& font,

                         AutocompleteEdit\* editor,

                         Profile\* profile);

  ~AutocompletePopupModel();

从这个类里可以看到它是继承类ACControllerListener，说明它是响应一个返回结果的监听器；继承类Task说明它是一个任务线程类。由这两个类可以看出，它是把关键字给一个线程，然后让这个线程去查询结果，当结果返回时，就再更新到显示窗口里。

虽然上面理解它的查询过程了，但是向谁查询呢？这是一个一定需要了解的问题。现在就来分析类AutocompleteController，它在构造函数时，就会创建三个查询的对象：

#001  AutocompleteController::AutocompleteController(ACControllerListener\* listener,

#002                                                 Profile\* profile)

#003      : listener\_(listener) {

#004    providers\_.push\_back(new **SearchProvider**(this, profile));

#005    providers\_.push\_back(new **HistoryURLProvider**(this, profile));

#006    keyword\_provider\_ = new **KeywordProvider**(this, profile);

#007    providers\_.push\_back(keyword\_provider\_);

#008    if (listener) {

#009      // These providers are async-only, so there's no need to create them when

#010      // we'll only be doing synchronous queries.

#011      history\_contents\_provider\_ = new **HistoryContentsProvider**(this, profile);

#012      providers\_.push\_back(history\_contents\_provider\_);

#013    } else {

#014      history\_contents\_provider\_ = NULL;

#015    }

#016    for (ACProviders::iterator i(providers\_.begin()); i != providers\_.end(); ++i)

#017      (\*i)->AddRef();

#018  }

从上面的代码，可以看到它是向**SearchProvider、HistoryURLProvider、KeywordProvider和HistoryContentsProvider**来查找到合适的智能提示。类**SearchProvider**是从搜索引擎里查找合适的内容；类**HistoryURLProvider**是从历史的URL里查找合适的内容；类**KeywordProvider**是从关键字搜索引擎里查找合适的内容；类**HistoryContentsProvider**是从历史内容里查找合适内容。从上面四种智能提示里，在以前的浏览器里一般只能做到从历史的URL里提示，现在“可多米”可以做到从搜索引擎和关键字引擎里查找到相应的结果回来，可见它是智能提示完美的体现，智能的水平可想而知了。这就是强大的云计算典型应用，如果没有强大的服务器群是做不到几亿人输入关键字时，还能快速返回结果的。

分析到这里，也许知道为什么GOOGLE开发浏览器的原因了吧，如果其它浏览是不可能采用这样的技术来分析用户的输入的，顶多是到历史记录里查找一下就算了。

**谷歌浏览器的源码分析(11)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次介绍到怎么样从其它地方返回搜索到的超级连接，现在就来分析一下使用搜索引擎去查找的类**SearchProvider**，它是通过搜索引擎来查找出来的，在这里是通过GOOGLE搜索引擎来查找出来。它的声明如下：

#001  // Autocomplete provider for searches and suggestions from a search engine.

#002  //

#003  // After construction, the autocomplete controller repeatedly calls Start()

#004  // with some user input, each time expecting to receive a small set of the best

#005  // matches (either synchronously or asynchronously).

#006  //

#007  // Initially the provider creates a match that searches for the current input

#008  // text.  It also starts a task to query the Suggest servers.  When that data

#009  // comes back, the provider creates and returns matches for the best

#010  // suggestions.

SearchProvider类是继承AutocompleteProvider和URLFetcher类，AutocompleteProvider提供一个自动完成的结果，URLFetcher主要提供从URL获取数据和状态。

#011  class SearchProvider : public AutocompleteProvider,

#012                         public URLFetcher::Delegate {

#013   public:

#014    SearchProvider(ACProviderListener\* listener, Profile\* profile)

#015        : AutocompleteProvider(listener, profile, "Search"),

#016          last\_default\_provider\_(NULL),

#017          fetcher\_(NULL),

#018          history\_request\_pending\_(false),

#019          have\_history\_results\_(false),

#020          suggest\_results\_pending\_(false),

#021          have\_suggest\_results\_(false) {

#022    }

#023

开始获取。

#024    // AutocompleteProvider

#025    virtual void Start(const AutocompleteInput& input,

#026                       bool minimal\_changes,

#027                       bool synchronous\_only);

停止查找。

#028    virtual void Stop();

#029

当获取到数据回来时响应。

#030    // URLFetcher::Delegate

#031    virtual void OnURLFetchComplete(const URLFetcher\* source,

#032                                    const GURL& url,

#033                                    const URLRequestStatus& status,

#034                                    int response\_code,

#035                                    const ResponseCookies& cookies,

#036                                    const std::string& data);

#037

#038   private:

#039    struct NavigationResult {

#040      NavigationResult(const std::wstring& url, const std::wstring& site\_name)

#041          : url(url),

#042            site\_name(site\_name) {

#043      }

#044

#045      // The URL.

#046      std::wstring url;

#047

#048      // Name for the site.

#049      std::wstring site\_name;

#050    };

#051

保存返回的结果。

#052    typedef std::vector<std::wstring> SuggestResults;

#053    typedef std::vector<NavigationResult> NavigationResults;

#054    typedef std::vector<history::KeywordSearchTermVisit> HistoryResults;

#055    typedef std::map<std::wstring, AutocompleteMatch> MatchMap;

#056

运行获取搜索引擎数据。

#057    // Called when timer\_ expires.

#058    void Run();

#059

#060    // Determines whether an asynchronous subcomponent query should run for the

#061    // current input.  If so, starts it if necessary; otherwise stops it.

#062    // NOTE: These functions do not update |done\_|.  Callers must do so.

#063    void StartOrStopHistoryQuery(bool minimal\_changes, bool synchronous\_only);

#064    void StartOrStopSuggestQuery(bool minimal\_changes, bool synchronous\_only);

#065

#066    // Functions to stop the separate asynchronous subcomponents.

#067    // NOTE: These functions do not update |done\_|.  Callers must do so.

#068    void StopHistory();

#069    void StopSuggest();

#070

#071    // Called back by the history system to return searches that begin with the

#072    // input text.

#073    void OnGotMostRecentKeywordSearchTerms(

#074        CancelableRequestProvider::Handle handle,

#075        HistoryResults\* results);

#076

#077    // Parses the results from the Suggest server and stores up to kMaxMatches of

#078    // them in server\_results\_.  Returns whether parsing succeeded.

#079    bool ParseSuggestResults(Value\* root\_val);

#080

#081    // Converts the parsed server results in server\_results\_ to a set of

#082    // AutocompleteMatches and adds them to |matches\_|.  This also sets |done\_|

#083    // correctly.

#084    void ConvertResultsToAutocompleteMatches();

#085

#086    // Determines the relevance for a particular match.  We use different scoring

#087    // algorithms for the different types of matches.

#088    int CalculateRelevanceForWhatYouTyped() const;

#089    // |time| is the time at which this query was last seen.

#090    int CalculateRelevanceForHistory(const Time& time) const;

#091    // |suggestion\_value| is which suggestion this is in the list returned from

#092    // the server; the best suggestion is suggestion number 0.

#093    int CalculateRelevanceForSuggestion(size\_t suggestion\_value) const;

#094    // |suggestion\_value| is same as above.

#095    int CalculateRelevanceForNavigation(size\_t suggestion\_value) const;

#096

#097    // Creates an AutocompleteMatch for "Search <engine> for |query\_string|" with

#098    // the supplied relevance.  Adds this match to |map|; if such a match already

#099    // exists, whichever one has lower relevance is eliminated.

#100    void AddMatchToMap(const std::wstring& query\_string,

#101                       int relevance,

#102                       int accepted\_suggestion,

#103                       MatchMap\* map);

#104    // Returns an AutocompleteMatch for a navigational suggestion.

#105    AutocompleteMatch NavigationToMatch(const NavigationResult& query\_string,

#106                                        int relevance);

#107

#108    // Trims "http:" and up to two subsequent slashes from |url|.  Returns the

#109    // number of characters that were trimmed.

#110    // TODO(kochi): this is duplicate from history\_autocomplete

#111    static size\_t TrimHttpPrefix(std::wstring\* url);

#112

#113    // Don't send any queries to the server until some time has elapsed after

#114    // the last keypress, to avoid flooding the server with requests we are

#115    // likely to end up throwing away anyway.

#116    static const int kQueryDelayMs;

#117

#118    // The user's input.

#119    AutocompleteInput input\_;

#120

#121    TemplateURL default\_provider\_;  // Cached across the life of a query so we

#122                                    // behave consistently even if the user

#123                                    // changes their default while the query is

#124                                    // running.

#125    const TemplateURL\* last\_default\_provider\_;

#126                                    // TODO(pkasting): http://b/1162970  We

#127                                    // shouldn't need this.

#128

#129    // An object we can use to cancel history requests.

#130    CancelableRequestConsumer history\_request\_consumer\_;

#131

#132    // Searches in the user's history that begin with the input text.

#133    HistoryResults history\_results\_;

#134

#135    // Whether history\_results\_ is valid (so we can tell invalid apart from

#136    // empty).

#137    bool have\_history\_results\_;

#138

#139    // Whether we are waiting for a history request to finish.

#140    bool history\_request\_pending\_;

#141

#142    // True if we're expecting suggest results that haven't yet arrived.  This

#143    // could be because either |timer\_| or |fetcher| is still running (see below).

#144    bool suggest\_results\_pending\_;

#145

#146    // A timer to start a query to the suggest server after the user has stopped

#147    // typing for long enough.

#148    base::OneShotTimer<SearchProvider> timer\_;

#149

#150    // The fetcher that retrieves suggest results from the server.

#151    scoped\_ptr<URLFetcher> fetcher\_;

#152

#153    // Suggestions returned by the Suggest server for the input text.

#154    SuggestResults suggest\_results\_;

#155

#156    // Navigational suggestions returned by the server.

#157    NavigationResults navigation\_results\_;

#158

#159    // Whether suggest\_results\_ is valid.

#160    bool have\_suggest\_results\_;

#161

#162    DISALLOW\_EVIL\_CONSTRUCTORS(SearchProvider);

#163  };

#164

在这个类里先调用函数SearchProvider::Start来获取缺省的搜索引擎，然后停止以前的搜索，接着SearchProvider::Run()函数里使用URLFetcher获取数据回来，它的代码如下：

#001   void SearchProvider::Run() {

#002    // Start a new request with the current input.

#003    DCHECK(!done\_);

获取搜索的URL。

#004    const TemplateURLRef\* const suggestions\_url =

#005        default\_provider\_.suggestions\_url();

建议代替的字符。

#006    DCHECK(suggestions\_url->SupportsReplacement());

开始新的搜索。

#007    fetcher\_.reset(new URLFetcher(GURL(suggestions\_url->ReplaceSearchTerms(

#008        default\_provider\_, input\_.text(),

#009        TemplateURLRef::NO\_SUGGESTIONS\_AVAILABLE, std::wstring())),

#010        URLFetcher::GET, this));

#011    fetcher\_->set\_request\_context(profile\_->GetRequestContext());

#012    fetcher\_->Start();

#013  }

当前上面的搜索完成时，就会通知SearchProvider::OnURLFetchComplete函数来分析返回的结果，最后调用SearchProvider::ConvertResultsToAutocompleteMatches()函数来把结果转换自动完成的列表项。

**谷歌浏览器的源码分析(12)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

当我们根据自动提示，或者完全自己输入连接后，再按下回车键，这时浏览器就会去打开网页，或者去查找内容。那么浏览器是怎么样实现这些功能的呢？又是怎么判断去打开网页，还是去打开GOOGLE的搜索引擎的呢？下面就来分析这部份的代码，如下：

#001  bool AutocompleteEdit::OnKeyDownOnlyWritable(TCHAR key,

#002                                               UINT repeat\_count,

#003                                               UINT flags) {

#004    // NOTE: Annoyingly, ctrl-alt-<key> generates WM\_KEYDOWN rather than

#005    // WM\_SYSKEYDOWN, so we need to check (flags & KF\_ALTDOWN) in various places

#006    // in this function even with a WM\_SYSKEYDOWN handler.

#007

#008    int count = repeat\_count;

**#009    switch (key) {**

**#010      case VK\_RETURN:**

**#011        AcceptInput((flags & KF\_ALTDOWN) ? NEW\_FOREGROUND\_TAB : CURRENT\_TAB,**

**#012                    false);**

**#013        return true;**

#014

#015      case VK\_UP:

#016        count = -count;

#017        // FALL THROUGH

#018      case VK\_DOWN:

当用户按下回车键，就会调用上面的函数OnKeyDownOnlyWritable，并且在**VK\_RETURN**按键处理回车的事件，接着就是调用函数**AcceptInput**来处理。

这个函数的代码如下：

#001  void AutocompleteEdit::AcceptInput(WindowOpenDisposition disposition,

#002                                     bool for\_drop) {

#003    // Get the URL and transition type for the selected entry.

#004    PageTransition::Type transition;

#005    bool is\_history\_what\_you\_typed\_match;

#006    std::wstring alternate\_nav\_url;

保存当前获取的URL连接串。

#007    const std::wstring url(GetURLForCurrentText(&transition,

#008                                                &is\_history\_what\_you\_typed\_match,

#009                                                &alternate\_nav\_url));

#010    if (url.empty())

#011      return;

#012

判断是否重新加载当前的网页。

#013    if (url == permanent\_text\_) {

#014      // When the user hit enter on the existing permanent URL, treat it like a

#015      // reload for scoring purposes.  We could detect this by just checking

#016      // user\_input\_in\_progress\_, but it seems better to treat "edits" that end

#017      // up leaving the URL unchanged (e.g. deleting the last character and then

#018      // retyping it) as reloads too.

#019      transition = PageTransition::RELOAD;

#020    } else if (for\_drop || ((paste\_state\_ != NONE) &&

#021                            is\_history\_what\_you\_typed\_match)) {

下面是打开一个新的连接。

#022      // When the user pasted in a URL and hit enter, score it like a link click

#023      // rather than a normal typed URL, so it doesn't get inline autocompleted

#024      // as aggressively later.

#025      transition = PageTransition::LINK;

#026    }

#027

这里是调用OpenURL函数打开这个连接的内容。

#028    OpenURL(url, disposition, transition, alternate\_nav\_url,

#029            AutocompletePopupModel::kNoMatch,

#030            is\_keyword\_hint\_ ? std::wstring() : keyword\_);

#031  }

这段代码的流程很清楚，就是先通过判断按键输入，是否按下回车键，如果是回车键就调用函数**AcceptInput**处理，然后在这个函数就判断这个连接是否已经打开了，如果已经打开，只需要重新加载就行了，如果不是当前的，就是打开一个新的连接。下一次再来分析OpenURL函数是怎么样通过连接来重新加载，还是打开一个新网页。

**谷歌浏览器的源码分析(13)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到调用函数OpenURL来打开网络连接，这仅是网络浏览的开始，现在再来分析它怎么样去下载网页数据，然后再显示出来。

#001  void AutocompleteEdit::OpenURL(const std::wstring& url,

#002                                 WindowOpenDisposition disposition,

#003                                 PageTransition::Type transition,

#004                                 const std::wstring& alternate\_nav\_url,

#005                                 size\_t selected\_line,

#006                                 const std::wstring& keyword) {

#007    if (url.empty())

#008      return;

#009

#010    ScopedFreeze freeze(this, GetTextObjectModel());

#011    SendOpenNotification(selected\_line, keyword);

#012

#013    if (disposition != NEW\_BACKGROUND\_TAB)

#014      RevertAll();  // Revert the box to its unedited state

#015    controller\_->OnAutocompleteAccept(url, disposition, transition,

#016                                      alternate\_nav\_url);

#017  }

在这个函数里第一个参数url是要打开的网络连接；第二个参数disposition是显示位置，比如新添加一个TAB显示，还是在原来的TAB显示；第三个参数transition是下载的类型，比如是重新加载，还是新的连接下载；第四个参数alternate\_nav\_url是候选的连接；第五个参数是选择那一行提示菜单；第六个参数keyword是关键字。

第7行判断打开的连接是否为空，如果为空就不用打开连接了。

第10行锁定输入框。

第11行通知选中的关键字。

第13行，14行关闭当前输入提示，直接在当前窗口显示，清除一些状态和内容。

第15行调控制实例来打开输入的连接。

上面的代码在AutocompleteEdit类里处理完成了，然后就把打开网络连接放到别的类里来执行，这样可以提高代码的复用性，降低了代码复杂程度。那么这个controller\_是何方神圣呢？一看AutocompleteEdit类有Controller类的定义，肯定就是它了，但再仔细一看它，它只是一个接口类，所有函数都是纯虚函数，真是百思不得其解时，突然想起，既然它是接口类，肯定就有其它类继承它的，那么再通过搜索，一查看，果然找到一个继承它的类LocationBarView，接着查看它的声明，如下：

#001  /////////////////////////////////////////////////////////////////////////////

#002  //

#003  // LocationBarView class

#004  //

#005  //   The LocationBarView class is a View subclass that paints the background

#006  //   of the URL bar strip and contains its content.

#007  //

#008  /////////////////////////////////////////////////////////////////////////////

#009  class LocationBarView : public ChromeViews::View,

#010                          public AutocompleteEdit::Controller {

#011   public:

可见类LocationBarView是继承ChromeViews::View类，并且继承AutocompleteEdit::Controller类。说明它是一个窗口类，并且是控制类，那么就是说所有接口的功能都是在这个类里实现的，只需要分析这个类里的内容，就知道它是怎么样打开连接的了，下一次再来分析它。

**谷歌浏览器的源码分析(14)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到控制类的派生类LocationBarView，现在就来分析这个函数的功能，看看它又把URL连接传到那里去，立即就去看代码，在这行代码controller\_->OnAutocompleteAccept里，可以看到调用函数OnAutocompleteAccept,它的代码如下：

#001  void LocationBarView::OnAutocompleteAccept(

#002      const std::wstring& url,

#003      WindowOpenDisposition disposition,

#004      PageTransition::Type transition,

#005      const std::wstring& alternate\_nav\_url) {

判断输入的URL连接是否为空。

#006    if (url.empty())

#007      return;

#008

保存相应的参数。

#009    location\_input\_ = url;

#010    disposition\_ = disposition;

#011    transition\_ = transition;

#012

调用控制器controller\_来打开这个连接。

#013    if (controller\_) {

#014      if (alternate\_nav\_url.empty()) {

#015        controller\_->ExecuteCommand(IDC\_OPENURL);

#016        return;

#017      }

#018

打开候选的连接。

#019      scoped\_ptr<AlternateNavURLFetcher> fetcher(

#020          new AlternateNavURLFetcher(alternate\_nav\_url));

#021      // The AlternateNavURLFetcher will listen for the pending navigation

#022      // notification that will be issued as a result of the "open URL." It

#023      // will automatically install itself into that navigation controller.

#024      controller\_->ExecuteCommand(IDC\_OPENURL);

#025      if (fetcher->state() == AlternateNavURLFetcher::NOT\_STARTED) {

#026        // I'm not sure this should be reachable, but I'm not also sure enough

#027        // that it shouldn't to stick in a NOTREACHED().  In any case, this is

#028        // harmless; we can simply let the fetcher get deleted here and it will

#029        // clean itself up properly.

#030      } else {

#031        fetcher.release();  // The navigation controller will delete the fetcher.

#032      }

#033    }

#034  }

上面的代码主要保存传入来的参数，然后紧接着又调用了控制器controller\_的函数ExecuteCommand来执行命令，这个命令是IDC\_OPENURL。为什么要使用命令的方式呢？仔细地思考一下，原来这种方式是便于使用自动化测试，测试时可以自动使用程序来不断传入命令来执行。

我们再来分析这行代码：

controller\_->ExecuteCommand(IDC\_OPENURL)；

controller\_是类CommandController的实例，它主要是由MVC设计模式的控制类，可见这里可以学习怎么样把MVC设计模式应用到实际例子里，使用这种模式主要是分离面渲染、逻辑控制和不同的数据来源，这样方便维护代码。

其实所有的命令并不是CommandController来处理，它只是一个中传站，把命令发往不同的浏览器对象，如下面的代码：

#001  void CommandController::ExecuteCommand(int id) {

#002    if (IsCommandEnabled(id))

#003      handler\_->ExecuteCommand(id);

#004  }

这样就把命令发送到handler\_处理了，而这里的handler\_是什么呢？其实它就是浏览器对象类Browser的实例，因此命令就是发送给浏览器对象来处理，它是怎么样处理命令的呢？下一次再来分析。

**谷歌浏览器的源码分析(15)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到发送命令给浏览器对象打开网页显示，但还没有分析它是怎么实现的，现在就来分析这方面的内容，如下：

#001  void Browser::ExecuteCommand(int id) {

#002    if (!IsCommandEnabled(id)) {

#003      NOTREACHED() << id;

#004      return;

#005    }

#006    // This might happen during QMU testing.

#007    if (!GetSelectedTabContents())

#008      return;

#009

#010    switch (id) {

#011      case IDC\_BACK:

#012        UserMetrics::RecordAction(L"Back", profile\_);

#013        GoBack();

#014        break;

#015

#016        ...

#017

#018

#019         case IDC\_OPENURL:

#020        UserMetrics::RecordAction(L"LoadURL", profile\_);

#021        {

#022          LocationBarView\* lbv = GetLocationBarView();

#023          if (lbv) {

#024            OpenURL(GURL(lbv->location\_input()), lbv->disposition(),

#025                    lbv->transition());

#026          } else {

#027            OpenURL(GURL(), CURRENT\_TAB, PageTransition::TYPED);

#028          }

#029        }

#030        break;

#031

可以看到这段代码，第19行就是处理打开网页的命令处理，当然在这个函数里不仅仅处理打开网页的连接命令，还有很多其它的命令，目前先分析这个命令的代码。

第20行里先记录当前执行的动作。

第22行里查找到当前显示BAR窗口。

如果找到LocationBarView窗口，就把当前的输入连接生成GURL对象，从窗口获取显示位置，传送的类型。否则，就是使用空的连接，并打开当前页面。

下面再来分析函数OpenURL的实现，它是调用TabContentsDelegate类里的OpenURL函数：

#001    virtual void OpenURL(const GURL& url,

#002                         WindowOpenDisposition disposition,

#003                         PageTransition::Type transition) {

#004      OpenURLFromTab(NULL, url, disposition, transition, std::string());

#005    }

在这个函数继续调用OpenURLFromTab函数来实现打开网页的功能，这个函数比较复杂，下一次再来分析它。

**谷歌浏览器的源码分析(16)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到需要调用这个OpenURLFromTab函数，那么这个函数是做什么的呢？从名称上可能猜到它是打开网页，但是是从目前TAB页里打开呢？还是新建一个？或者使用每个TAB页一个进程呢？这些疑惑，只能通过代码的分析来理解它的实现，代码如下：

这个函数的参数意思：

source是TAB内容。

url是网络连接地址。

disposition是窗口打开的位置。

transition是连接传送的类型。

override\_encoding是编码类型。

#001  void Browser::OpenURLFromTab(TabContents\* source,

#002                               const GURL& url,

#003                               WindowOpenDisposition disposition,

#004                               PageTransition::Type transition,

#005                               const std::string& override\_encoding) {

调试时检查代码。

#006    // No code for these yet

#007    DCHECK((disposition != NEW\_POPUP) && (disposition != SAVE\_TO\_DISK));

#008

获取当前的TAB页。

#009    TabContents\* current\_tab = source ? source : GetSelectedTabContents();

判断是否当前TAB页选中。

#010    bool source\_tab\_was\_frontmost = (current\_tab == GetSelectedTabContents());

#011    TabContents\* new\_contents = NULL;

#012

#013    // If the URL is part of the same web site, then load it in the same

#014    // SiteInstance (and thus the same process).  This is an optimization to

#015    // reduce process overhead; it is not necessary for compatibility.  (That is,

#016    // the new tab will not have script connections to the previous tab, so it

#017    // does not need to be part of the same SiteInstance or BrowsingInstance.)

#018    // Default to loading in a new SiteInstance and BrowsingInstance.

#019    // TODO(creis): should this apply to applications?

保存打开连接的实例指针。

#020    SiteInstance\* instance = NULL;

如果不使用每个TAB页一个进程的方式，就不需要进行下面的处理。因为同一个连接在一个进程里打开是比较快，这里主要做优化。

#021    // Don't use this logic when "--process-per-tab" is specified.

判断是否有每一个TAB一个进程的方式。

#022    if (!CommandLine().HasSwitch(switches::kProcessPerTab)) {

有当前进程页。

#023      if (current\_tab) {

#024        const WebContents\* const web\_contents = current\_tab->AsWebContents();

判断是否相同的网络连接地址。

#025        if (web\_contents) {

#026          const GURL& current\_url = web\_contents->GetURL();

如果相同的网络地址，并且有实例打开，就返回这个实例在instance。

#027          if (SiteInstance::IsSameWebSite(current\_url, url))

#028            instance = web\_contents->site\_instance();

#029        }

#030      }

#031    }

#032

#033    // If this is an application we can only have one tab so a new tab always

#034    // goes into a tabbed browser window.

下面进行不打开新窗口的处理。

#035    if (disposition != NEW\_WINDOW && type\_ == BrowserType::APPLICATION) {

#036      // If the disposition is OFF\_THE\_RECORD we don't want to create a new

#037      // browser that will itself create another OTR browser. This will result in

#038      // a browser leak (and crash below because no tab is created or selected).

#039      if (disposition == OFF\_THE\_RECORD) {

#040        OpenURLOffTheRecord(profile\_, url);

#041        return;

#042      }

#043

#044      Browser\* b = GetOrCreateTabbedBrowser();

#045      DCHECK(b);

#046

#047      // If we have just created a new browser window, make sure we select the

#048      // tab.

#049      if (b->tab\_count() == 0 && disposition == NEW\_BACKGROUND\_TAB)

#050        disposition = NEW\_FOREGROUND\_TAB;

#051

#052      b->OpenURL(url, disposition, transition);

#053      b->Show();

#054      b->MoveToFront(true);

#055      return;

#056    }

#057

#058    if (profile\_->IsOffTheRecord() && disposition == OFF\_THE\_RECORD)

#059      disposition = NEW\_FOREGROUND\_TAB;

#060

这里开始处理打开一个新窗口显示网络连接。

#061    if (disposition == NEW\_WINDOW) {

创建一个新的Browser浏览器对象。

#062      Browser\* new\_browser = new Browser(gfx::Rect(), SW\_SHOWNORMAL, profile\_,

#063                                         BrowserType::TABBED\_BROWSER, L"");

创建一个TAB内容。

#064      new\_contents = new\_browser->AddTabWithURL(url, transition, true, instance);

这里开始显示这个网络连接的内容。

#065      new\_browser->Show();

#066    } else if ((disposition == CURRENT\_TAB) && current\_tab) {

下面开始在当前TAB页里打开连接，同时判断处理的类型。

#067      if (transition == PageTransition::TYPED ||

#068          transition == PageTransition::AUTO\_BOOKMARK ||

#069          transition == PageTransition::GENERATED ||

#070          transition == PageTransition::START\_PAGE) {

#071        // Don't forget the openers if this tab is a New Tab page opened at the

#072        // end of the TabStrip (e.g. by pressing Ctrl+T). Give the user one

#073        // navigation of one of these transition types before resetting the

#074        // opener relationships (this allows for the use case of opening a new

#075        // tab to do a quick look-up of something while viewing a tab earlier in

#076        // the strip). We can make this heuristic more permissive if need be.

#077        // TODO(beng): (http://b/1306495) write unit tests for this once this

#078        //             object is unit-testable.

#079        int current\_tab\_index =

#080            tabstrip\_model\_.GetIndexOfTabContents(current\_tab);

#081        bool forget\_openers =

#082            !(current\_tab->type() == TAB\_CONTENTS\_NEW\_TAB\_UI &&

#083            current\_tab\_index == (tab\_count() - 1) &&

#084            current\_tab->controller()->GetEntryCount() == 1);

#085        if (forget\_openers) {

#086          // If the user navigates the current tab to another page in any way

#087          // other than by clicking a link, we want to pro-actively forget all

#088          // TabStrip opener relationships since we assume they're beginning a

#089          // different task by reusing the current tab.

#090          tabstrip\_model\_.ForgetAllOpeners();

#091          // In this specific case we also want to reset the group relationship,

#092          // since it is now technically invalid.

#093          tabstrip\_model\_.ForgetGroup(current\_tab);

#094        }

#095      }

这里开始在当前TAB页里加载网络地址连接。

#096      current\_tab->controller()->LoadURL(url, transition);

#097      // The TabContents might have changed as part of the navigation (ex: new tab

#098      // page can become WebContents).

获取当前显示的内容。

#099      new\_contents = current\_tab->controller()->active\_contents();

隐藏最下面状态提示窗口。

#100      GetStatusBubble()->Hide();

#101

#102      // Synchronously update the location bar. This allows us to immediately

#103      // have the URL bar update when the user types something, rather than

#104      // going through the normal system of ScheduleUIUpdate which has a delay.

更新本地的工具条。

#105      UpdateToolBar(false);

后面的内容先不分析，主要分析目前打开当前连接的内容。

#106    } else if (disposition == OFF\_THE\_RECORD) {

#107      OpenURLOffTheRecord(profile\_, url);

#108      return;

#109    } else if (disposition != SUPPRESS\_OPEN) {

#110      new\_contents =

#111          AddTabWithURL(url, transition, disposition != NEW\_BACKGROUND\_TAB,

#112                        instance);

#113    }

#114

#115    if (disposition != NEW\_BACKGROUND\_TAB && source\_tab\_was\_frontmost) {

#116      // Give the focus to the newly navigated tab, if the source tab was

#117      // front-most.

#118      new\_contents->Focus();

#119    }

#120

#121    if (!override\_encoding.empty()) {

#122      // The new tab needs a special encoding, such as a view source page

#123      // which should use the same encoding as the original page.

#124      WebContents\* web\_contents = new\_contents->AsWebContents();

#125      if (web\_contents)

#126        web\_contents->set\_override\_encoding(override\_encoding);

#127    }

#128  }

#129

上面函数的过程是这样的：主要根据打开网页的方式来选择窗口，比如是创建新窗口，还是只是打开一个TAB页，然后在TAB页显示。在这里还做了同一个网页地址的优化，不让它打开两个相同的进程来处理。

Browser对象是创建一个浏览器对象，接着调用LoadURL函数来加载输入的网页连接，隐藏当前窗口最下面的加载状态条，更新当前窗口的状态条，就完成了这个函数对当前TAB页加载网页的功能。下一次再来分析LoadURL函数是怎么样打开网页连接了。

**谷歌浏览器的源码分析(17)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

当输入的网页连接传送给LoadURL函数之后，还需要处理很多内容，其实这是由类NavigationController来管理的，NavigationController类主要就是管理加载网页、退回、前进等等控制。

#001  void NavigationController::LoadURL(const GURL& url,

#002                                     PageTransition::Type transition) {

#003    // The user initiated a load, we don't need to reload anymore.

#004    needs\_reload\_ = false;

#005

#006    NavigationEntry\* entry = CreateNavigationEntry(url, transition);

#007

#008    LoadEntry(entry);

#009  }

这个函数的第一个参数url是网络连接地址，第二个参数transition是传送的类型。

第4行代码里设置不是重新加载。

第6行里创建了一个处理网页浏览的入口对象，它是由类NavigationEntry管理。

第8行里就调用函数LoadEntry来加载网页。

LoadEntry函数更进一步去加载网页的内容，它的代码如下：

#001  void NavigationController::LoadEntry(NavigationEntry\* entry) {

#002    // When navigating to a new page, we don't know for sure if we will actually

#003    // end up leaving the current page.  The new page load could for example

#004    // result in a download or a 'no content' response (e.g., a mailto: URL).

#005

#006    // TODO(pkasting): http://b/1113085 Should this use DiscardPendingEntry()?

清除内部变量。

#007    DiscardPendingEntryInternal();

保存当前的入口对象。

#008    pending\_entry\_ = entry;

通知服务器有一个浏览器对象加入。

#009    NotificationService::current()->Notify(

#010        NOTIFY\_NAV\_ENTRY\_PENDING,

#011        Source<NavigationController>(this),

#012        NotificationService::NoDetails());

下面开始进入加载网页的动作。

#013    NavigateToPendingEntry(false);

#014  }

第9行里的类NotificationService是使用OBSERVER的设计模式来实现一对多的显示关系。这个设计模式也是跟MVC与生具来的，显然设计这个浏览器代码的人，已经是对设计模式是专家式的人物了。

第13行里调用函数NavigateToPendingEntry，下一次再来分析它的功能。

**谷歌浏览器的源码分析(18)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

现在继续分析浏览器去下载网页的过程，上一次说到需要分析函数NavigateToPendingEntry，在这个函数又是怎么去处理下载网页的呢？那就需要分析它的源码了，如下：

#001  void NavigationController::NavigateToPendingEntry(bool reload) {

#002    TabContents\* from\_contents = active\_contents\_;

#003

#004    // For session history navigations only the pending\_entry\_index\_ is set.

下面从历史里找到入口选项。

#005    if (!pending\_entry\_) {

#006      DCHECK(pending\_entry\_index\_ != -1);

#007      pending\_entry\_ = entries\_[pending\_entry\_index\_].get();

#008    }

#009

复位当前的SSL状态。

#010    // Reset the security states as any SSL error may have been resolved since we

#011    // last visited that page.

#012    pending\_entry\_->ssl() = NavigationEntry::SSLStatus();

#013

设置内容是否可以显示。

#014    if (from\_contents && from\_contents->type() != pending\_entry\_->tab\_type())

#015      from\_contents->SetActive(false);

#016

获取当前的父窗口的句柄。

#017    HWND parent =

#018        from\_contents ? GetParent(from\_contents->GetContainerHWND()) : 0;

获取当前显示的TAB内容对象。

#019    TabContents\* contents =

#020        GetTabContentsCreateIfNecessary(parent, \*pending\_entry\_);

#021

#022    contents->SetActive(true);

#023    active\_contents\_ = contents;

#024

修改委托对象。

#025    if (from\_contents && from\_contents != contents) {

#026      if (from\_contents->delegate())

#027        from\_contents->delegate()->ReplaceContents(from\_contents, contents);

#028    }

#029

现在开始打开入口对象里指定的网站。

#030    if (!contents->Navigate(\*pending\_entry\_, reload))

#031      DiscardPendingEntry();

#032  }

TabContents类主要描述主显示区的内容，在第30行里就调用它的函数Navigate去浏览网页的内容。pending\_entry\_成员变量是NavigationEntry类的对象，它主要保存所有创建浏览时需要的信息，比如网络连接地址。

当我再跟踪contents->Navigate这行代码时，它不是运行TabContents类的Navigate，这比较奇怪，但回过头来再看一下它的声明如下：

virtual bool Navigate(const NavigationEntry& entry, bool reload);

可见，它前面加了virtual关键字，说明它是虚函数，也就是说设计时，就让它是多态的出现，因此在什么情况下运行什么样的函数内容是不定的，对于这些样的函数，就需要小心一点了，只有实际运行的类才知道它是什么内容。由于我是输入URL关键字，所以它调用的函数是类WebContents里的Navigate函数。下一次再来分析类WebContents里的Navigate函数。

**谷歌浏览器的源码分析(19)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

继续上一次分析到类WebContents的Navigate函数，在这个函数里通过参数entry传送入来，这样它只需要根据这个参数去下载网页回来显示，应就可以了吧，但到底是怎么样工作的呢？这需要深入去分析它，才知道它是什么样的结果。

#001

#002  bool WebContents::Navigate(const NavigationEntry& entry, bool reload) {

从渲染显示管理器里获取当前连接渲染显示对象。

#003    RenderViewHost\* dest\_render\_view\_host = render\_manager\_.Navigate(entry);

#004

设置开始下载计时的时钟。

#005    // Used for page load time metrics.

#006    current\_load\_start\_ = TimeTicks::Now();

#007

在渲染显示对象里进行浏览处理。

#008    // Navigate in the desired RenderViewHost

#009    dest\_render\_view\_host->NavigateToEntry(entry, reload);

#010

#011    if (entry.page\_id() == -1) {

#012      // HACK!!  This code suppresses javascript: URLs from being added to

#013      // session history, which is what we want to do for javascript: URLs that

#014      // do not generate content.  What we really need is a message from the

#015      // renderer telling us that a new page was not created.  The same message

#016      // could be used for mailto: URLs and the like.

#017      if (entry.url().SchemeIs("javascript"))

#018        return false;

#019    }

#020

判断是否重新加载旧的连接处理。

#021    if (reload && !profile()->IsOffTheRecord()) {

#022      HistoryService\* history =

#023          profile()->GetHistoryService(Profile::IMPLICIT\_ACCESS);

#024      if (history)

#025        history->SetFavIconOutOfDateForPage(entry.url());

#026    }

#027

#028    return true;

#029  }

在这个函数最主要的工作，就是调用类RenderViewHost函数NavigateToEntry，这个函数的代码如下：

#001  void RenderViewHost::NavigateToEntry(const NavigationEntry& entry,

#002                                       bool is\_reload) {

创建浏览参数。

#003    ViewMsg\_Navigate\_Params params;

#004    MakeNavigateParams(entry, is\_reload, &params);

#005

授权渲染进程可以显示这个连接。

#006    RendererSecurityPolicy::GetInstance()->GrantRequestURL(

#007        process()->host\_id(), params.url);

#008

发送浏览下载连接参数给进程处理。

#009    DoNavigate(new ViewMsg\_Navigate(routing\_id\_, params));

#010

更新列表计数。

#011    UpdateBackForwardListCount();

#012  }

在这个函数里，主要创建浏览参数，然后调用函数DoNavigate来发送一个消息ViewMsg\_Navigate给RHV进程，在UpdateBackForwardListCount函数里也发送一个消息ViewMsg\_UpdateBackForwardListCount给RHV进程。

继续分析函数DoNavigate：

#001  void RenderViewHost::DoNavigate(ViewMsg\_Navigate\* nav\_message) {

#002    // Only send the message if we aren't suspended at the start of a cross-site

#003    // request.

如果已经挂起，就开始重新复位这个消息。

#004    if (navigations\_suspended\_) {

#005      // Shouldn't be possible to have a second navigation while suspended, since

#006      // navigations will only be suspended during a cross-site request.  If a

#007      // second navigation occurs, WebContents will cancel this pending RVH

#008      // create a new pending RVH.

#009      DCHECK(!suspended\_nav\_message\_.get());

#010      suspended\_nav\_message\_.reset(nav\_message);

#011    } else {

或者直接发送这个消息出去。

#012      Send(nav\_message);

#013    }

#014  }

函数UpdateBackForwardListCount的代码如下：

#001  void RenderViewHost::UpdateBackForwardListCount() {

#002    int back\_list\_count, forward\_list\_count;

#003    delegate\_->GetHistoryListCount(&back\_list\_count, &forward\_list\_count);

#004    Send(new ViewMsg\_UpdateBackForwardListCount(

#005        routing\_id\_, back\_list\_count, forward\_list\_count));

#006  }

可以从函数DoNavigate和UpdateBackForwardListCount里看到，最后都把这些事件变成消息，通过类RenderProcessHost来发送出去，主要使用IPC的通讯机制。具体是怎么样通讯的呢？下一次再来分析它。

**谷歌浏览器的源码分析(20)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次介绍到把网页连接地址生成一个消息通过IPC机制把消息发送出去，那么IPC的通讯机制是怎么样的呢？又是发送给谁呢？

由于这个浏览器是使用多进程的架构来工作的，所以进程之间就需要相互交流，这种交流是就是通讯，可以从源码里看到它是使用IPC的机制来通讯，实际采用的技术，就是Windows的命名管道的方式。可以看到这段代码：

#001  bool Channel::CreatePipe(const wstring& channel\_id, Mode mode) {

#002    DCHECK(pipe\_ == INVALID\_HANDLE\_VALUE);

#003    const wstring pipe\_name = PipeName(channel\_id);

#004    if (mode == MODE\_SERVER) {

#005      SECURITY\_ATTRIBUTES security\_attributes = {0};

#006      security\_attributes.bInheritHandle = FALSE;

#007      security\_attributes.nLength = sizeof(SECURITY\_ATTRIBUTES);

#008      if (!win\_util::GetLogonSessionOnlyDACL(

#009          reinterpret\_cast<SECURITY\_DESCRIPTOR\*\*>(

#010              &security\_attributes.lpSecurityDescriptor))) {

#011        NOTREACHED();

#012      }

#013

#014      pipe\_ = CreateNamedPipeW(pipe\_name.c\_str(),

#015                               PIPE\_ACCESS\_DUPLEX | FILE\_FLAG\_OVERLAPPED |

#016                                  FILE\_FLAG\_FIRST\_PIPE\_INSTANCE,

#017                               PIPE\_TYPE\_BYTE | PIPE\_READMODE\_BYTE,

#018                               1,         // number of pipe instances

#019                               BUF\_SIZE,  // output buffer size (XXX tune)

#020                               BUF\_SIZE,  // input buffer size (XXX tune)

#021                               5000,      // timeout in milliseconds (XXX tune)

#022                               &security\_attributes);

#023      LocalFree(security\_attributes.lpSecurityDescriptor);

#024    } else {

#025      pipe\_ = CreateFileW(pipe\_name.c\_str(),

#026                          GENERIC\_READ | GENERIC\_WRITE,

#027                          0,

#028                          NULL,

#029                          OPEN\_EXISTING,

#030                          SECURITY\_SQOS\_PRESENT | SECURITY\_IDENTIFICATION |

#031                              FILE\_FLAG\_OVERLAPPED,

#032                          NULL);

#033    }

上面这段代码通过WINDOWS API函数CreateNamedPipeW函数来创建命名管道的服务器端，而通过WINDOWS API函数CreateFileW来打开客户端，这样两个进程之间就建立起来通讯的管道，两个进程之间的消息就可以相互发送了。

在浏览网页连接的消息，就是通过IPC的机制，让类RenderProcessHost把消息发送出去，那么这个消息是谁在另一端接收的呢？按照IPC的机制可知是另外一个进程在接收，这个进程就是类RenderProcess。

类RenderProcessHost把所有的消息通过类IPC::ChannelProxy发送出去，在另一个子进程里通过类RenderThread和类RenderView来接收消息，然后在类RenderThread和类RenderView把消息分发处理。

**谷歌浏览器的源码分析(21)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到类RenderThread和类RenderView把消息处理，那么这两个类是怎么样处理消息的呢？又是怎么样处理浏览的消息呢？现在就带着这两个问题去分析它的源码，理解它处理消息的方法。类RenderThread处理消息的代码如下：

#001  void RenderThread::OnMessageReceived(const IPC::Message& msg) {

#002    // NOTE: We could subclass router\_ to intercept OnControlMessageReceived, but

#003    // it seems simpler to just process any control messages that we care about

#004    // up-front and then send the rest of the messages onto router\_.

#005

下面判断是控制消息，如果是控制消息就在本类里处理，否则就分发到别的地方处理，主要是转到类RenderView处理。

#006    if (msg.routing\_id() == MSG\_ROUTING\_CONTROL) {

#007      IPC\_BEGIN\_MESSAGE\_MAP(RenderThread, msg)

#008        IPC\_MESSAGE\_HANDLER(ViewMsg\_VisitedLink\_NewTable, OnUpdateVisitedLinks)

#009        IPC\_MESSAGE\_HANDLER(ViewMsg\_SetNextPageID, OnSetNextPageID)

#010        IPC\_MESSAGE\_HANDLER(ViewMsg\_New, OnCreateNewView)

#011        IPC\_MESSAGE\_HANDLER(ViewMsg\_SetCacheCapacities, OnSetCacheCapacities)

#012        IPC\_MESSAGE\_HANDLER(ViewMsg\_GetCacheResourceStats,

#013                            OnGetCacheResourceStats)

#014        // send the rest to the router

#015        IPC\_MESSAGE\_UNHANDLED(router\_.OnMessageReceived(msg))

#016      IPC\_END\_MESSAGE\_MAP()

#017    } else {

这里是分发消息到别的地方处理。

#018      router\_.OnMessageReceived(msg);

#019    }

#020  }

在浏览器里，消息分为两大类：控制消息和路由消息。像使用IPC\_MESSAGE\_CONTROL宏定义的消息，就是控制消息；使用IPC\_MESSAGE\_ROUTED宏定义的消息，就是路由消息。

路由消息分发是由类MessageRouter来负责的，主要处理的代码如下：

#001

#002  void MessageRouter::OnMessageReceived(const IPC::Message& msg) {

#003    if (msg.routing\_id() == MSG\_ROUTING\_CONTROL) {

#004      OnControlMessageReceived(msg);

#005    } else {

#006      RouteMessage(msg);

#007    }

#008  }

在这里又分为MSG\_ROUTING\_CONTROL消息和其它路由消息，再一次通过函数RouteMessage分发之后，如下：

#001  bool MessageRouter::RouteMessage(const IPC::Message& msg) {

#002    IPC::Channel::Listener\* listener = routes\_.Lookup(msg.routing\_id());

#003    if (!listener)

#004      return false;

#005

#006    listener->OnMessageReceived(msg);

#007    return true;

#008  }

上面这个函数里又把消息通过发送到listener里去，其实listener是根据消息的目标routing\_id来选择的，那么就是说它是选择发送到不同的窗口里去，因为每个TAB一个窗口。消息经过这样的处理之后，就到达了终点地--- RenderView::OnMessageReceived函数。下一次再来分析RenderView::OnMessageReceived函数的代码和后继处理。

**谷歌浏览器的源码分析(22)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到消息转发，并分析了RenderThread类里处理消息的函数，其实大部份的消息都是在RenderView类里的OnMessageReceived函数处理，比如浏览的消息也是在这里处理。它的代码如下：

#001  void RenderView::OnMessageReceived(const IPC::Message& message) {

#002    // Let the resource dispatcher intercept resource messages first.

如果是资源消息，就直接分发去处理，以便提高效率。

#003    if (resource\_dispatcher\_->OnMessageReceived(message))

#004      return;

下面开始处理RenderView类里所有的消息。

#005    IPC\_BEGIN\_MESSAGE\_MAP(RenderView, message)

#006      IPC\_MESSAGE\_HANDLER(ViewMsg\_CreatingNew\_ACK, OnCreatingNewAck)

#007      IPC\_MESSAGE\_HANDLER(ViewMsg\_CaptureThumbnail, SendThumbnail)

#008      IPC\_MESSAGE\_HANDLER(ViewMsg\_GetPrintedPagesCount, OnGetPrintedPagesCount)

#009      IPC\_MESSAGE\_HANDLER(ViewMsg\_PrintPages, OnPrintPages)

#010      IPC\_MESSAGE\_HANDLER(ViewMsg\_Navigate, OnNavigate)

#011      IPC\_MESSAGE\_HANDLER(ViewMsg\_Stop, OnStop)

#012      IPC\_MESSAGE\_HANDLER(ViewMsg\_LoadAlternateHTMLText, OnLoadAlternateHTMLText)

#013      IPC\_MESSAGE\_HANDLER(ViewMsg\_StopFinding, OnStopFinding)

#014      IPC\_MESSAGE\_HANDLER(ViewMsg\_Undo, OnUndo)

#015      IPC\_MESSAGE\_HANDLER(ViewMsg\_Redo, OnRedo)

#016      IPC\_MESSAGE\_HANDLER(ViewMsg\_Cut, OnCut)

#017      IPC\_MESSAGE\_HANDLER(ViewMsg\_Copy, OnCopy)

#018      IPC\_MESSAGE\_HANDLER(ViewMsg\_Paste, OnPaste)

#019      IPC\_MESSAGE\_HANDLER(ViewMsg\_Replace, OnReplace)

#020      IPC\_MESSAGE\_HANDLER(ViewMsg\_Delete, OnDelete)

#021      IPC\_MESSAGE\_HANDLER(ViewMsg\_SelectAll, OnSelectAll)

#022      IPC\_MESSAGE\_HANDLER(ViewMsg\_CopyImageAt, OnCopyImageAt)

#023      IPC\_MESSAGE\_HANDLER(ViewMsg\_Find, OnFind)

#024      IPC\_MESSAGE\_HANDLER(ViewMsg\_AlterTextSize, OnAlterTextSize)

#025      IPC\_MESSAGE\_HANDLER(ViewMsg\_SetPageEncoding, OnSetPageEncoding)

#026      IPC\_MESSAGE\_HANDLER(ViewMsg\_InspectElement, OnInspectElement)

#027      IPC\_MESSAGE\_HANDLER(ViewMsg\_ShowJavaScriptConsole, OnShowJavaScriptConsole)

#028      IPC\_MESSAGE\_HANDLER(ViewMsg\_DownloadImage, OnDownloadImage)

#029      IPC\_MESSAGE\_HANDLER(ViewMsg\_ScriptEvalRequest, OnScriptEvalRequest)

#030      IPC\_MESSAGE\_HANDLER(ViewMsg\_AddMessageToConsole, OnAddMessageToConsole)

#031      IPC\_MESSAGE\_HANDLER(ViewMsg\_DebugAttach, OnDebugAttach)

#032      IPC\_MESSAGE\_HANDLER(ViewMsg\_DebugDetach, OnDebugDetach)

#033      IPC\_MESSAGE\_HANDLER(ViewMsg\_ReservePageIDRange, OnReservePageIDRange)

#034      IPC\_MESSAGE\_HANDLER(ViewMsg\_UploadFile, OnUploadFileRequest)

#035      IPC\_MESSAGE\_HANDLER(ViewMsg\_FormFill, OnFormFill)

#036      IPC\_MESSAGE\_HANDLER(ViewMsg\_FillPasswordForm, OnFillPasswordForm)

#037      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragTargetDragEnter, OnDragTargetDragEnter)

#038      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragTargetDragOver, OnDragTargetDragOver)

#039      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragTargetDragLeave, OnDragTargetDragLeave)

#040      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragTargetDrop, OnDragTargetDrop)

#041      IPC\_MESSAGE\_HANDLER(ViewMsg\_AllowDomAutomationBindings,

#042                          OnAllowDomAutomationBindings)

#043      IPC\_MESSAGE\_HANDLER(ViewMsg\_AllowBindings, OnAllowBindings)

#044      IPC\_MESSAGE\_HANDLER(ViewMsg\_SetDOMUIProperty, OnSetDOMUIProperty)

#045      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragSourceEndedOrMoved, OnDragSourceEndedOrMoved)

#046      IPC\_MESSAGE\_HANDLER(ViewMsg\_DragSourceSystemDragEnded,

#047                          OnDragSourceSystemDragEnded)

#048      IPC\_MESSAGE\_HANDLER(ViewMsg\_SetInitialFocus, OnSetInitialFocus)

#049      IPC\_MESSAGE\_HANDLER(ViewMsg\_FindReplyACK, OnFindReplyAck)

#050      IPC\_MESSAGE\_HANDLER(ViewMsg\_UpdateTargetURL\_ACK, OnUpdateTargetURLAck)

#051      IPC\_MESSAGE\_HANDLER(ViewMsg\_UpdateWebPreferences, OnUpdateWebPreferences)

#052      IPC\_MESSAGE\_HANDLER(ViewMsg\_SetAltErrorPageURL, OnSetAltErrorPageURL)

#053      IPC\_MESSAGE\_HANDLER(ViewMsg\_InstallMissingPlugin, OnInstallMissingPlugin)

#054      IPC\_MESSAGE\_HANDLER(ViewMsg\_RunFileChooserResponse, OnFileChooserResponse)

#055      IPC\_MESSAGE\_HANDLER(ViewMsg\_EnableViewSourceMode, OnEnableViewSourceMode)

#056      IPC\_MESSAGE\_HANDLER(ViewMsg\_UpdateBackForwardListCount,

#057                          OnUpdateBackForwardListCount)

#058      IPC\_MESSAGE\_HANDLER(ViewMsg\_GetAllSavableResourceLinksForCurrentPage,

#059                          OnGetAllSavableResourceLinksForCurrentPage)

#060      IPC\_MESSAGE\_HANDLER(ViewMsg\_GetSerializedHtmlDataForCurrentPageWithLocalLinks,

#061                          OnGetSerializedHtmlDataForCurrentPageWithLocalLinks)

#062      IPC\_MESSAGE\_HANDLER(ViewMsg\_GetApplicationInfo, OnGetApplicationInfo)

#063      IPC\_MESSAGE\_HANDLER(ViewMsg\_ShouldClose, OnMsgShouldClose)

#064      IPC\_MESSAGE\_HANDLER(ViewMsg\_ClosePage, OnClosePage)

#065      IPC\_MESSAGE\_HANDLER(ViewMsg\_ThemeChanged, OnThemeChanged)

#066  #ifdef CHROME\_PERSONALIZATION

#067      IPC\_MESSAGE\_HANDLER(ViewMsg\_PersonalizationEvent, OnPersonalizationEvent)

#068  #endif

#069      IPC\_MESSAGE\_HANDLER(ViewMsg\_HandleMessageFromExternalHost,

#070                          OnMessageFromExternalHost)

这里对于没有处理的消息进行提示。

#071      // Have the super handle all other messages.

#072      IPC\_MESSAGE\_UNHANDLED(RenderWidget::OnMessageReceived(message))

#073    IPC\_END\_MESSAGE\_MAP()

#074  }

从上面这个函数可以看到，它的消息处理是非常多的，下面来分析一个浏览网络连接的消息，它就是ViewMsg\_Navigate，可以看到这个消息后面响应函数是OnNavigate，也就是说，当你输入网络地址之后按回车，就会通过上说过的IPC机制把消息发送到这里，接着来看这个函数OnNavigate的代码，如下：

#001  void RenderView::OnNavigate(const ViewMsg\_Navigate\_Params& params) {

判断窗口是否关闭，如果关闭就不用去打开连接地址了。

#002    if (!webview())

#003      return;

#004

处理一些about的连接处理，比如about:crash。

#005    AboutHandler::MaybeHandle(params.url);

#006

保存是否重新加载网页。

#007    bool is\_reload = params.reload;

#008

获取WEB的显示框架。

#009    WebFrame\* main\_frame = webview()->GetMainFrame();

判断当是重新加载时，而当前又不是历史网页的情况。

#010    if (is\_reload && !main\_frame->HasCurrentState()) {

#011      // We cannot reload if we do not have any history state.  This happens, for

#012      // example, when recovering from a crash.  Our workaround here is a bit of

#013      // a hack since it means that reload after a crashed tab does not cause an

#014      // end-to-end cache validation.

#015      is\_reload = false;

#016    }

#017

下面设置缓冲策略。

#018    WebRequestCachePolicy cache\_policy;

#019    if (is\_reload) {

#020      cache\_policy = WebRequestReloadIgnoringCacheData;

#021    } else if (params.page\_id != -1 || main\_frame->GetInViewSourceMode()) {

#022      cache\_policy = WebRequestReturnCacheDataElseLoad;

#023    } else {

#024      cache\_policy = WebRequestUseProtocolCachePolicy;

#025    }

#026

下面创建一个下载请求，并把相关参数设置到请求里面。

#027    scoped\_ptr<WebRequest> request(WebRequest::Create(params.url));

#028    request->SetCachePolicy(cache\_policy);

#029    request->SetExtraData(new RenderViewExtraRequestData(

#030        params.page\_id, params.transition, params.url));

#031

设置WEBKIT的请求状态。

#032    // If we are reloading, then WebKit will use the state of the current page.

#033    // Otherwise, we give it the state to navigate to.

#034    if (!is\_reload)

#035      request->SetHistoryState(params.state);

#036

让主WEB显示框架去下载请求显示。

#037    main\_frame->LoadRequest(request.get());

#038  }

分析这个函数，就可以知道处理浏览消息的过程，下一次来分析WebFrame里的接口函数LoadRequest处理过程，到底它是怎么样处理javascript脚本网页下载的呢？

**谷歌浏览器的源码分析(23)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

继续上一次来分析LoadRequest的代码，在分析这个函数代码之前，先看看WebFrame类的继承层次关系，如下：

class WebFrame : public base::RefCounted<WebFrame> {

WebFrame是一个接口类，但它先继承引用计数类RefCounted，这样对于这个对象多次访问，就可以使用引用计数来判断对象的生命周期了。对于base::RefCounted<WebFrame>的语法，其实它是一种模板实现的多态特性，这种方案是最高效的实现方式，比使用虚函数更少占内存，并且运行的速度也更快。它就是解决如下的问题：

  void Release() {

    if (subtle::RefCountedBase::Release()) {

      delete static\_cast<T\*>(this);

    }

  }

上面的函数里static\_cast<T\*>(this)，它就是一种多态的实现方法，由于base::RefCounted类并没有声明为虚析构函数，如下：

template <class T>

class RefCounted : public subtle::RefCountedBase {

 public:

  RefCounted() { }

  ~RefCounted() { }

既然没有把类RefCounted声明为虚析构函数，又想在基类里调用派生类的析构函数，只好使用static\_cast和类型转换了，这是一种比较好的模板使用方法，在WTL里就大量使用这种技术。

接着可以看到：

class WebFrameImpl : public WebFrame {

 public:

  WebFrameImpl();

  ~WebFrameImpl();

类WebFrameImpl是继承接口类WebFrame，这里是使用接口与实现分析的设计模式，这样更方便代码灵活地复用。可见设计Chrome的设计师和写代码的程序员，都是顶尖的模板高手，大部的思想与WTL库的设计是一脉相承。也难怪Chrome的浏览器使用WTL库来设计界面。

#001  void WebFrameImpl::LoadRequest(WebRequest\* request) {

#002    SubstituteData data;

#003    InternalLoadRequest(request, data, false);

#004  }

在WebFrame里调用函数LoadRequest，实际上是调用实现类WebFrameImpl函数LoadRequest，而在这个函数又是调用InternalLoadRequest来实现的，它的代码如下：

#001  void WebFrameImpl::InternalLoadRequest(const WebRequest\* request,

#002                                         const SubstituteData& data,

#003                                         bool replace) {

//转换请求参数。

#004    const WebRequestImpl\* request\_impl =

#005        static\_cast<const WebRequestImpl\*>(request);

#006

获取请求的资源。

#007    const ResourceRequest& resource\_request =

#008        request\_impl->frame\_load\_request().resourceRequest();

#009

#010    // Special-case javascript URLs.  Do not interrupt the existing load when

#011    // asked to load a javascript URL unless the script generates a result.

#012    // We can't just use FrameLoader::executeIfJavaScriptURL because it doesn't

#013    // handle redirects properly.

获取需要下载网页的地址。

#014    const KURL& kurl = resource\_request.url();

处理加载javascript的连接情况。

#015    if (!data.isValid() && kurl.protocol() == "javascript") {

#016      // Don't attempt to reload javascript URLs.

#017      if (resource\_request.cachePolicy() == ReloadIgnoringCacheData)

#018        return;

#019

#020      // We can't load a javascript: URL if there is no Document!

#021      if (!frame\_->document())

#022        return;

#023

#024      // TODO(darin): Is this the best API to use here?  It works and seems good,

#025      // but will it change out from under us?

#026      DeprecatedString script =

#027          KURL::decode\_string(kurl.deprecatedString().mid(sizeof("javascript:")-1));

#028      bool succ = false;

加载执行脚本。

#029      WebCore::String value =

#030          frame\_->loader()->executeScript(script, &succ, true);

#031      if (succ && !frame\_->loader()->isScheduledLocationChangePending()) {

#032        // TODO(darin): We need to figure out how to represent this in session

#033        // history.  Hint: don't re-eval script when the user or script navigates

#034        // back-n-forth (instead store the script result somewhere).

#035        LoadDocumentData(kurl, value, String("text/html"), String());

#036      }

#037      return;

#038    }

#039

停止上一次没有完成的加载情况。

#040    StopLoading();  // make sure existing activity stops

#041

#042    // Keep track of the request temporarily.  This is effectively a way of

#043    // passing the request to callbacks that may need it.  See

#044    // WebFrameLoaderClient::createDocumentLoader.

保存当前的请求连接。

#045    currently\_loading\_request\_ = request;

#046

#047    // If we have a current datasource, save the request info on it immediately.

#048    // This is because WebCore may not actually initiate a load on the toplevel

#049    // frame for some subframe navigations, so we want to update its request.

获取当前数据源，如果已经存在就可以保存它。

#050    WebDataSourceImpl\* datasource = GetDataSourceImpl();

#051    if (datasource)

#052      CacheCurrentRequestInfo(datasource);

#053

如果数据有效就可以直接替换就行了。

#054    if (data.isValid()) {

#055      frame\_->loader()->load(resource\_request, data);

#056      if (replace) {

#057        // Do this to force WebKit to treat the load as replacing the currently

#058        // loaded page.

#059        frame\_->loader()->setReplacing();

#060      }

如果是历史网页选择，就判断是否出错的加载处理。

#061    } else if (request\_impl->history\_item()) {

#062      // Use the history item if we have one, otherwise fall back to standard

#063      // load.

#064      RefPtr<HistoryItem> current\_item = frame\_->loader()->currentHistoryItem();

#065

#066      // If there is no current\_item, which happens when we are navigating in

#067      // session history after a crash, we need to manufacture one otherwise

#068      // WebKit hoarks. This is probably the wrong thing to do, but it seems to

#069      // work.

#070      if (!current\_item) {

#071        current\_item = new HistoryItem(KURL("about:blank"), "");

#072        frame\_->loader()->setCurrentHistoryItem(current\_item);

#073        frame\_->page()->backForwardList()->setCurrentItem(current\_item.get());

#074

#075        // Mark the item as fake, so that we don't attempt to save its state and

#076        // end up with about:blank in the navigation history.

#077        frame\_->page()->backForwardList()->setCurrentItemFake(true);

#078      }

#079

#080      frame\_->loader()->goToItem(request\_impl->history\_item().get(),

#081                                 WebCore::FrameLoadTypeIndexedBackForward);

重新加载网页。

#082    } else if (resource\_request.cachePolicy() == ReloadIgnoringCacheData) {

#083      frame\_->loader()->reload();

下面开始调用load来加载新下载的网页资源。

#084    } else {

#085      frame\_->loader()->load(resource\_request);

#086    }

#087

#088    currently\_loading\_request\_ = NULL;

#089  }

上面通过几种情况来分别实现了加载javascript网页的处理，还有历史选项处理，还有重新加载网页和加载新网页的处理。下一次再来分析加载新网页的函数frame\_->loader()->load的实现。

**谷歌浏览器的源码分析(24)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

继续上一次的分析，这里开始把连接址和其它相关的信息传送frame\_->loader()->load函数里面，那么在这个函数里面到底是怎么样处理的呢，只有去分析它的代码，我们才能找到它的答案，现在就来开始看吧，如下：

#001  void FrameLoader::load(const ResourceRequest& request)

#002  {

#003      load(request, SubstituteData());

#004  }

在这个函数也只是一个中间者，它又调用函数load函数的重载函数来实现了。

#001  void FrameLoader::load(const ResourceRequest& request, const SubstituteData& substituteData)

#002  {

#003      if (m\_inStopAllLoaders)

#004          return;

#005

#006      // FIXME: is this the right place to reset loadType? Perhaps this should be done after loading is finished or aborted.

#007      m\_loadType = FrameLoadTypeStandard;

#008      load(m\_client->createDocumentLoader(request, substituteData).get());

#009  }

#010

在这个函数里，第一个参数request是连接相关的信息，第二个参数substituteData是一些状态数据。然后在第7行里设置加载的类型，第8行里调用WebFrameLoaderClient::createDocumentLoader函数来创建WebDocumentLoaderImpl对象，然后再通过get函数返回来，这样就知道load函数又调用那个重载函数了，原来它是调用这个函数，如下：

#001  void FrameLoader::load(DocumentLoader\* newDocumentLoader)

#002  {

#003      ResourceRequest& r = newDocumentLoader->request();

#004      addExtraFieldsToRequest(r, true, false);

#005      FrameLoadType type;

#006

#007      if (shouldTreatURLAsSameAsCurrent(newDocumentLoader->originalRequest().url())) {

#008          r.setCachePolicy(ReloadIgnoringCacheData);

#009          type = FrameLoadTypeSame;

#010      } else

#011          type = FrameLoadTypeStandard;

#012

#013      // Do not use original encoding override since it is not loaded by user

#014      // selecting encoding.

#015      if (m\_documentLoader)

#016          newDocumentLoader->setOverrideEncoding(String());

#017

#018      // When we loading alternate content for an unreachable URL that we're

#019      // visiting in the b/f list, we treat it as a reload so the b/f list

#020      // is appropriately maintained.

#021      if (shouldReloadToHandleUnreachableURL(newDocumentLoader)) {

#022          ASSERT(type == FrameLoadTypeStandard);

#023          type = FrameLoadTypeReload;

#024      }

#025

#026      load(newDocumentLoader, type, 0);

#027  }

上面只对newDocumentLoader做一些准备工作，并没有真正地去加载任何东西，接着又调用函数：

void FrameLoader::load(DocumentLoader\* loader, FrameLoadType type, PassRefPtr<FormState> formState)

在上面这个函数进行安全策略的处理，然后再经过N个函数处理之后，就调用下面的函数：

void FrameLoader::continueLoadAfterWillSubmitForm(PolicyAction)

在这个函数开始使用类DocumentLoader来设置下载请求，主要通过函数

bool DocumentLoader::startLoadingMainResource(unsigned long identifier)

来实现的，紧跟后面调用加载函数：

bool MainResourceLoader::load(const ResourceRequest& r, const SubstituteData&  substituteData)

在这个函数里又开始分为两种情况处理，一种是延进加载数据，一种是立即加载数据，下面主要介绍立即加载数据函数：

bool MainResourceLoader::loadNow(ResourceRequest& r)

在类MainResourceLoader是主要资源下载的管理类，loadNow函数是把资源请求ResourceRequest变成一个IPC消息又发送给资源下载进程去处理。它的简略代码如下：

#001  bool MainResourceLoader::loadNow(ResourceRequest& r)

#002  {

......

#011      willSendRequest(r, ResourceResponse());

#012

......

#015      if (!frameLoader())

#016          return false;

#017

......

#023

#024      if (m\_substituteData.isValid())

#025          handleDataLoadSoon(r);

#026      else if (shouldLoadEmpty || frameLoader()->representationExistsForURLScheme(url.protocol()))

#027          handleEmptyLoad(url, !shouldLoadEmpty);

#028      else

#029          m\_handle = ResourceHandle::create(r, this, m\_frame.get(), false, true, true);

#030

#031      return false;

#032  }

在这个函数的第29行里，就会通过ResourceHandle::create函数创建一个资源消息，并把这个消息发送出去，到底它是怎么样实现的呢？下一次再来分析它。

**谷歌浏览器的源码分析(25)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到需要把显示的网络连接地址变成一个资源的消息发送出去，它是通过函数ResourceHandle::create来实现的，但这个函数到底是怎么样实现的呢？现在就分析它的实现代码，了解它怎么样把资源变换成消息，并且通过IPC机制把消息发送到资源下载进程去。数ResourceHandle::create的代码如下：

#001  PassRefPtr<ResourceHandle> ResourceHandle::create(const ResourceRequest& request,

#002                                                    ResourceHandleClient\* client,

#003                                                    Frame\* deprecated,

#004                                                    bool defersLoading,

#005                                                    bool shouldContentSniff,

#006                                                    bool mightDownloadFromHandle) {

上面的参数request是把所有请求网络连接地址信息传进来了。

#007    RefPtr<ResourceHandle> newHandle(

#008        new ResourceHandle(request, client, defersLoading, shouldContentSniff,

#009                           mightDownloadFromHandle));

这里创建资源类ResourceHandle对象，通过它来生成一个消息发送出去。

#010

#011    if (newHandle->start(NULL))

#012      return newHandle.release();

上面的代码里，调用函数start来处理资源请求下载。

#013

#014    return NULL;

#015  }

在这个函数里调用newHandle->start函数来处理，其实它是调用下面的函数来工作的：

bool ResourceHandle::start(Frame\* deprecated) {

  return d->Start(NULL);

}

那么这里的d实例是什么呢？可以通过ResourceHandle的构造函数来看到它的类，如下：

ResourceHandle::ResourceHandle(const ResourceRequest& request,

                               ResourceHandleClient\* client,

                               bool defersLoading,

                               bool shouldContentSniff,

                               bool mightDownloadFromHandle)

#pragma warning(suppress: 4355)  // it's okay to pass |this| here!

      : **d(new ResourceHandleInternal(this, request, client))** {

  // TODO(darin): figure out what to do with the two bool params

}

可以看到d是类ResourceHandleInternal的实例，这就是说调用d->Start函数，其实就是调用下面的函数：

#001  bool ResourceHandleInternal::Start(

#002      ResourceLoaderBridge::SyncLoadResponse\* sync\_load\_response) {

#003    DCHECK(!bridge\_.get());

#004

#005    // The WebFrame is the Frame's FrameWinClient

#006    WebFrameImpl\* webframe =

#007        request\_.frame() ? WebFrameImpl::FromFrame(request\_.frame()) : NULL;

......

#154

#155    if (sync\_load\_response) {

#156      bridge\_->SyncLoad(sync\_load\_response);

#157      return true;

#158    }

#159

通过上面的处理，然后就调用桥连接成员bridge\_来创建消息。

**#160    bool rv = bridge\_->Start(this);**

#161    if (rv) {

#162      pending\_ = true;

#163      job\_->ref();  // to be released when we get a OnCompletedRequest.

#164    } else {

#165      bridge\_.reset();

#166    }

#167

#168    return rv;

#169  }

在这里使用一个设计模式，叫桥连接模式。函数**bridge\_->Start**的代码如下：

// Writes a footer on the message and sends it

bool IPCResourceLoaderBridge::Start(Peer\* peer) {

  if (request\_id\_ != -1) {

    NOTREACHED() << "Starting a request twice";

    return false;

  }

  RESOURCE\_LOG("Starting request for " << url\_);

保存当前接收的连接端点。

  peer\_ = peer;

生成请求ID，以便返回数据时可以找到相应的显示进程和窗口。

  // generate the request ID, and append it to the message

  request\_id\_ = dispatcher\_->AddPendingRequest(peer\_, request\_.resource\_type,

                                               request\_.mixed\_content);

找到IPC的消息发送对象，然后创建ViewHostMsg\_RequestResource消息并发送出去。

  IPC::Message::Sender\* sender = dispatcher\_->message\_sender();

  bool ret = false;

  if (sender)

    ret = sender->Send(new ViewHostMsg\_RequestResource(MSG\_ROUTING\_NONE,

                                                       request\_id\_,

                                                       request\_));

  return ret;

}

通过上面漫长的分析，总算搞清楚了这个过程：

从界面开始输入URL地址，然后界面把URL发送到渲染进程，渲染进程再进行处理，把这个URL连接请求再次发送到资源下载进程去处理。串起来是一个极其简单的过程，但在这个浏览器里比较复杂的，因为它是多进程的浏览器，进程之间相互消息传送，就比其它浏览器复杂，并且它还有很多安全策略的使用和优化处理，导致这个处理过程是比较复杂的。

OK，资源下载请求消息已经发送出去，那么这个消息又往何处而去呢？又怎么样通过网络连接下载回来呢？欲知后事如何，请继续看下一篇！

**谷歌浏览器的源码分析(26)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

消息的流通过程，是一个不同类相互交流的过程，如果不了解这个过程，根本就不知道这些类是怎么样相互协作的。由于上一次说到ViewHostMsg\_RequestResource消息已经发送出来，它的处理过徎其实就是一般资源的消息处理过程，下面就来看看这个消息的处理过程，如下：

1.       base::MessagePumpWin::Run函数消息

2.       base::MessagePumpWin::RunWithDispatcher

3.       base::MessagePumpWin::DoRunLoop

4.       base::MessagePumpWin::WaitForWork

5.       base::MessagePumpWin::SignalWatcher

6.       IPC::Channel::OnObjectSignaled 通道的消息处理。

7.       IPC::Channel::ProcessIncomingMessages

8.       IPC::SyncChannel::SyncContext::OnMessageReceived

9.       IPC::ChannelProxy::Context::TryFilters

10.    ResourceMessageFilter::OnMessageReceived

11.    IPC::MessageWithTuple<Tuple2<int,ViewHostMsg\_Resource\_Request> >::Dispatch<ResourceMessageFilter,int,ViewHostMsg\_Resource\_Request const &>

12.    ResourceMessageFilter::OnRequestResource

13.    ResourceDispatcherHost::BeginRequest

消息通过上面12个函数的流转，到达到处理这个消息的函数ResourceDispatcherHost::BeginRequest，它在里面就会把消息变成一个网络下载请求，函数代码如下：

#001  void ResourceDispatcherHost::BeginRequest(

#002      Receiver\* receiver,

#003      HANDLE render\_process\_handle,

#004      int render\_process\_host\_id,

#005      int render\_view\_id,

#006      int request\_id,

#007      const ViewHostMsg\_Resource\_Request& request\_data,

#008      URLRequestContext\* request\_context,

#009      IPC::Message\* sync\_result) {

是否已经传送完成，或者关闭。

#010    if (is\_shutdown\_ ||

#011        !ShouldServiceRequest(render\_process\_host\_id, request\_data)) {

#012      // Tell the renderer that this request was disallowed.

#013      receiver->Send(new ViewMsg\_Resource\_RequestComplete(

#014          render\_view\_id,

#015          request\_id,

#016          URLRequestStatus(URLRequestStatus::FAILED, net::ERR\_ABORTED)));

#017      return;

#018    }

#019

保证所有插件已经加载。

#020    // Ensure the Chrome plugins are loaded, as they may intercept network

#021    // requests.  Does nothing if they are already loaded.

#022    // TODO(mpcomplete): This takes 200 ms!  Investigate parallelizing this by

#023    // starting the load earlier in a BG thread.

#024    plugin\_service\_->LoadChromePlugins(this);

#025

构造事件处理器。

#026    // Construct the event handler.

#027    scoped\_refptr<EventHandler> handler;

#028    if (sync\_result) {

#029      handler = new SyncEventHandler(receiver, request\_data.url, sync\_result);

#030    } else {

#031      handler = new AsyncEventHandler(receiver,

#032                                      render\_process\_host\_id,

#033                                      render\_view\_id,

#034                                      render\_process\_handle,

#035                                      request\_data.url,

#036                                      this);

#037    }

#038

#039    if (HandleExternalProtocol(request\_id, render\_process\_host\_id, render\_view\_id,

#040                               request\_data.url, request\_data.resource\_type,

#041                               handler)) {

#042      return;

#043    }

#044

构造下载请求。

#045    // Construct the request.

#046    URLRequest\* request = new URLRequest(request\_data.url, this);

#047    request->set\_method(request\_data.method);

#048    request->set\_policy\_url(request\_data.policy\_url);

#049    request->set\_referrer(request\_data.referrer.spec());

#050    request->SetExtraRequestHeaders(request\_data.headers);

#051    request->set\_load\_flags(request\_data.load\_flags);

#052    request->set\_context(request\_context);

#053    request->set\_origin\_pid(request\_data.origin\_pid);

#054

设置上传数据。

#055    // Set upload data.

#056    uint64 upload\_size = 0;

#057    if (!request\_data.upload\_content.empty()) {

#058      scoped\_refptr<net::UploadData> upload = new net::UploadData();

#059      upload->set\_elements(request\_data.upload\_content);  // Deep copy.

#060      request->set\_upload(upload);

#061      upload\_size = upload->GetContentLength();

#062    }

#063

安装一个CrossSiteEventHandler事件处理器。

#064    // Install a CrossSiteEventHandler if this request is coming from a

#065    // RenderViewHost with a pending cross-site request.  We only check this for

#066    // MAIN\_FRAME requests.

#067    // TODO(mpcomplete): remove "render\_process\_host\_id != -1"

#068    //                   when http://b/viewIssue?id=1080959 is fixed.

#069    if (request\_data.resource\_type == ResourceType::MAIN\_FRAME &&

#070        render\_process\_host\_id != -1 &&

#071        Singleton<CrossSiteRequestManager>::get()->

#072            HasPendingCrossSiteRequest(render\_process\_host\_id, render\_view\_id)) {

#073      // Wrap the event handler to be sure the current page's onunload handler

#074      // has a chance to run before we render the new page.

#075      handler = new CrossSiteEventHandler(handler,

#076                                          render\_process\_host\_id,

#077                                          render\_view\_id,

#078                                          this);

#079    }

#080

#081    if (safe\_browsing\_->enabled() &&

#082        safe\_browsing\_->CanCheckUrl(request\_data.url)) {

#083      handler = new SafeBrowsingEventHandler(handler,

#084                                             render\_process\_host\_id,

#085                                             render\_view\_id,

#086                                             request\_data.url,

#087                                             request\_data.resource\_type,

#088                                             safe\_browsing\_,

#089                                             this);

#090    }

#091

创建一个缓冲区处理。

#092    // Insert a buffered event handler before the actual one.

#093    handler = new BufferedEventHandler(handler, this, request);

#094

#095    // Make extra info and read footer (contains request ID).

#096    ExtraRequestInfo\* extra\_info =

#097        new ExtraRequestInfo(handler,

#098                             request\_id,

#099                             render\_process\_host\_id,

#100                             render\_view\_id,

#101                             request\_data.mixed\_content,

#102                             request\_data.resource\_type,

#103                             upload\_size);

#104    extra\_info->allow\_download =

#105        ResourceType::IsFrame(request\_data.resource\_type);

#106    request->set\_user\_data(extra\_info);  // takes pointer ownership

#107

开始调用内部处理请求函数。

#108    BeginRequestInternal(request, request\_data.mixed\_content);

#109  }

通过上面的分析，已经知道消息转换为一个请求任务URLRequest，这个任务就需要交给后面的工作进程来处理了，它是通过函数BeginRequestInternal来把任务进一步发送出去。

**谷歌浏览器的源码分析(27)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到怎么样开始把任务发送出去，也就是调用函数BeginRequestInternal来把URL请求发送，它的代码如下：

#001  void ResourceDispatcherHost::BeginRequestInternal(URLRequest\* request,

#002                                                    bool mixed\_content) {

获取请求信息。

#003    ExtraRequestInfo\* info = ExtraInfoForRequest(request);

生成全局ID，然后保存到正在下载请求队列里。

#004    GlobalRequestID global\_id(info->render\_process\_host\_id, info->request\_id);

#005    pending\_requests\_[global\_id] = request;

#006    if (mixed\_content) {

#007      // We don't start the request in that case.  The SSLManager will potentially

#008      // change the request (potentially to indicate its content should be

#009      // filtered) and start it itself.

#010      SSLManager::OnMixedContentRequest(this, request, ui\_loop\_);

#011      return;

#012    }

这里开始处理请求。

#013    request->Start();

#014

启动上传状态更新定时器。

#015    // Make sure we have the load state monitor running

#016    if (!update\_load\_states\_timer\_.IsRunning()) {

#017      update\_load\_states\_timer\_.Start(

#018          TimeDelta::FromMilliseconds(kUpdateLoadStatesIntervalMsec),

#019          this, &ResourceDispatcherHost::UpdateLoadStates);

#020    }

#021  }

通过上面的函数可以看到主要调用URLRequest::Start()来处理下载的请求，它的代码如下：

#001  void URLRequest::Start() {

#002    DCHECK(!is\_pending\_);

#003    DCHECK(!job\_);

#004

创建一个下载的工作任务。

#005    job\_ = GetJobManager()->CreateJob(this);

#006    job\_->SetExtraRequestHeaders(extra\_request\_headers\_);

#007

判断是否有数据需要上传。

#008    if (upload\_.get())

#009      job\_->SetUpload(upload\_.get());

#010

设置请下开始下载的时间，以便后面检查超时的状态。

#011    is\_pending\_ = true;

#012    response\_info\_.request\_time = Time::Now();

#013

#014    // Don't allow errors to be sent from within Start().

#015    // TODO(brettw) this may cause NotifyDone to be sent synchronously,

#016    // we probably don't want this: they should be sent asynchronously so

#017    // the caller does not get reentered.

这里把工作任务启动运行。

#018    job\_->Start();

#019  }

由于这里是对URL的HTTP请求下载数据，所以这里的job\_是类URLRequestHttpJob的实例，也就是调用函数URLRequestHttpJob::Start()，在函数URLRequestHttpJob::Start()的处理过程序如下：

1.       URLRequestHttpJob::StartTransaction()

2.       net::HttpCache::Transaction::Start

3.       net::HttpCache::Transaction::BeginNetworkRequest()

4.       net::HttpTransactionWinHttp::Start

5.       net::HttpTransactionWinHttp::DidResolveProxy()

6.       net::HttpTransactionWinHttp::OpenRequest

7.       net::HttpTransactionWinHttp::SendRequest()

8.       net::WinHttpRequestThrottle::SubmitRequest

9.       net::WinHttpRequestThrottle::SendRequest

通过上面9个函数的调用处理，然后就会通过Windows的HTTP API进行发送请求和下载数据。我们来分析一下最后的函数WinHttpRequestThrottle::SendRequest，看看怎么样调用Windows HTTP API函数来获取数据的，它的代码如下：

#001  BOOL WinHttpRequestThrottle::SendRequest(HINTERNET request\_handle,

#002                                           DWORD total\_size,

#003                                           DWORD\_PTR context,

#004                                           bool report\_async\_error) {

下面就是调用Windows的API函数WinHttpSendRequest来发送请求，当然在调用这个函数之前，需要调用函数WinHttpOpenRequest先打开一个TCP连接。

#005    BOOL ok = WinHttpSendRequest(request\_handle,

#006                                 WINHTTP\_NO\_ADDITIONAL\_HEADERS,

#007                                 0,

#008                                 WINHTTP\_NO\_REQUEST\_DATA,

#009                                 0,

#010                                 total\_size,

#011                                 context);

#012    if (!ok && report\_async\_error) {

#013      WINHTTP\_ASYNC\_RESULT async\_result = { API\_SEND\_REQUEST, GetLastError() };

出错处理，就调用外面的回调函数。

#014      HttpTransactionWinHttp::StatusCallback(

#015          request\_handle, context,

#016          WINHTTP\_CALLBACK\_STATUS\_REQUEST\_ERROR,

#017          &async\_result, sizeof(async\_result));

#018    }

#019    return ok;

#020  }

通过前面一系列的分析学会chrome浏览器怎么样输入URL地址，以及怎么样进行URL自动完成，然后把URL发送到渲染进程去处理，最后渲染进程又把资源下载请求发送到资源下载进程里处理，最后资源下载进程通过Windows HTTP API函数进行TCP连接，以及HTTP数据的上传和下载。浏览器向网站发送请求的过程已经分析完成了，那么HTTP API收到网页的数据后，又是怎么样处理的呢？下一次再来分析这个问题。

**谷歌浏览器的源码分析(28)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到使用WinHTTP函数来创建HTTP连接，并且使用它来发送请求，那么数据接收回来，当然也是从它那里接收的，因此下面来分析它的接收数据的几个函数，然后再来看看chrome里收到数据后发送到那里处理。

在WinHTTP里使用WinHttpOpen函数来创建HTTP连接，然后使用WinHttpConnect连接需要下载数据的网站，接着调函数WinHttpOpenRequest来创建一个请求，使用函数WinHttpSendRequest发送HTTP请求，如果有数据发送到服务器，就通过WinHttpWriteData函数来发送，最后调用函数WinHttpReceiveResponse来查询状态码，WinHttpQueryDataAvailable函数查询可以接收到的数据大小，调用函数WinHttpQueryHeaders来查询传送回来的html头的大小，调用函数WinHttpReadData来接收到WEB服务器发送回来的数据。

chrome就是通过下面的函数来实现数据接收的，如下：

#001  void HttpTransactionWinHttp::HandleStatusCallback(DWORD status,

#002                                                    DWORD\_PTR result,

#003                                                    DWORD error,

#004                                                    DWORD secure\_failure) {

#005    int rv = ERR\_FAILED;

#006

#007    switch (status) {

#008      case WINHTTP\_CALLBACK\_STATUS\_REQUEST\_ERROR:

#009        rv = DidReceiveError(error, secure\_failure);

#010        break;

#011      case WINHTTP\_CALLBACK\_STATUS\_SENDREQUEST\_COMPLETE:

#012        PopulateSSLInfo(secure\_failure);

#013        rv = DidSendRequest();

#014        break;

#015      case WINHTTP\_CALLBACK\_STATUS\_WRITE\_COMPLETE:

#016        rv = DidWriteData(static\_cast<DWORD>(result));

#017        break;

接收到HTTP协议头的数据 。

#018      case WINHTTP\_CALLBACK\_STATUS\_HEADERS\_AVAILABLE:

#019        rv = DidReceiveHeaders();

#020        break;

接收HTTP协议的数据 。

#021      case WINHTTP\_CALLBACK\_STATUS\_READ\_COMPLETE:

#022        rv = DidReadData(static\_cast<DWORD>(result));

#023        break;

#024      default:

#025        NOTREACHED() << "unexpected status code";

#026    }

#027

#028    if (rv == ERR\_IO\_PENDING) {

#029      session\_callback\_->AddRef();  // balanced when callback runs.

#030    } else if (callback\_) {

#031      DoCallback(rv);

#032    }

#033  }

通过上面的分析知道数据怎么样从WinHTTP接收到，并且发送到下一个阶段处理。

**谷歌浏览器的源码分析(29)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到通过WinHTTP来接收网络数据，但没有具体介绍怎么样接收，现在就来分析这方面的代码。首先是通过函数WinHttpQueryHeaders来查询HTTP协议头的大小，接着还是通过函数WinHttpQueryHeaders把数据接收到缓冲区里。下面这段代码，就是做这样的事情：

#001  int HttpTransactionWinHttp::DidReceiveHeaders() {

#002    session\_callback\_->set\_load\_state(LOAD\_STATE\_IDLE);

#003

**第一次调用函数WinHttpQueryHeaders查看接收到协议头的大小。**

#004    DWORD size = 0;

#005    if (!WinHttpQueryHeaders(request\_handle\_,

#006                             WINHTTP\_QUERY\_RAW\_HEADERS,

#007                             WINHTTP\_HEADER\_NAME\_BY\_INDEX,

#008                             NULL,

#009                             &size,

#010                             WINHTTP\_NO\_HEADER\_INDEX)) {

#011      DWORD error = GetLastError();

#012      if (error != ERROR\_INSUFFICIENT\_BUFFER) {

#013        DLOG(ERROR) << "WinHttpQueryHeaders failed: " << GetLastError();

#014        return TranslateLastOSError();

#015      }

#016      // OK, size should tell us how much to allocate...

#017      DCHECK(size > 0);

#018    }

#019

**第二次调用函数WinHttpQueryHeaders来接收协议头的数据。**

#020    std::wstring raw\_headers;

#021

#022    // 'size' is the number of bytes rather than the number of characters.

#023    DCHECK(size % 2 == 0);

#024    if (!WinHttpQueryHeaders(request\_handle\_,

#025                             WINHTTP\_QUERY\_RAW\_HEADERS,

#026                             WINHTTP\_HEADER\_NAME\_BY\_INDEX,

#027                             WriteInto(&raw\_headers, size/2 + 1),

#028                             &size,

#029                             WINHTTP\_NO\_HEADER\_INDEX)) {

#030      DLOG(ERROR) << "WinHttpQueryHeaders failed: " << GetLastError();

#031      return TranslateLastOSError();

#032    }

#033

**设置回应的一些状态。**

#034    response\_.response\_time = Time::Now();

#035

#036    // From experimentation, it appears that WinHttp translates non-ASCII bytes

#037    // found in the response headers to UTF-16 assuming that they are encoded

#038    // using the default system charset.  We attempt to undo that here.

#039    response\_.headers =

#040        new HttpResponseHeaders(base::SysWideToNativeMB(raw\_headers));

#041

#042    // WinHTTP truncates a response longer than 2GB.  Perhaps it stores the

#043    // response's content length in a signed 32-bit integer.  We fail rather

#044    // than reading a truncated response.

#045    if (response\_.headers->GetContentLength() > 0x80000000)

#046      return ERR\_FILE\_TOO\_BIG;

#047

#048    response\_.vary\_data.Init(\*request\_, \*response\_.headers);

#049    PopulateAuthChallenge();

#050

#051    // Unfortunately, WinHttp does not close the connection when a non-keepalive

#052    // response is \_not\_ followed by the server closing the connection.  So, we

#053    // attempt to hack around this bug.

#054    if (!response\_.headers->IsKeepAlive())

#055      content\_length\_remaining\_ = response\_.headers->GetContentLength();

#056

#057    return OK;

#058  }

通过上面的函数处理，就可以收到HTTP协议头的数据，这样就可以进一步处理了。那么接着下来就是收到HTTP协议里的数据，这个主要通过下面的函数来接收到的，如下：

#001  BOOL HttpTransactionWinHttp::SessionCallback::ReadData(

#002      HINTERNET request\_handle) {

#003    DCHECK(bytes\_available\_ >= 0);

#004    char\* buf = read\_buf\_;

#005    read\_buf\_ = NULL;

#006    int bytes\_to\_read = std::min(bytes\_available\_, read\_buf\_len\_);

#007    read\_buf\_len\_ = 0;

#008    if (!bytes\_to\_read)

#009      bytes\_to\_read = 1;

#010

#011    // Because of how WinHTTP fills memory when used asynchronously, Purify isn't

#012    // able to detect that it's been initialized, so it scans for 0xcd in the

#013    // buffer and reports UMRs (uninitialized memory reads) for those individual

#014    // bytes. We override that to avoid the false error reports.

#015    // See http://b/issue?id=1173916.

#016    base::MemoryDebug::MarkAsInitialized(buf, bytes\_to\_read);

#017    return **WinHttpReadData**(request\_handle, buf, bytes\_to\_read, NULL);

#018  }

上面通过判断可以接收到多少字节，然后通过函数**WinHttpReadData**把数据保存到缓冲区read\_buf\_里，在这个缓冲区里保存了所有网络接收到的数据，那么这些数据又将要流向何方呢？下一次再来分析这个问题。

**谷歌浏览器的源码分析(30)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上次说到函数**WinHttpReadData**接收数据到缓冲区里，那么这些数据又是怎么样传送给下一步处理的呢？带着这个问题，我们来分析下面这段代码，如下：

#001  void HttpTransactionWinHttp::HandleStatusCallback(DWORD status,

#002                                                    DWORD\_PTR result,

#003                                                    DWORD error,

#004                                                    DWORD secure\_failure) {

#005    int rv = ERR\_FAILED;

#006

#007    switch (status) {

#008      case WINHTTP\_CALLBACK\_STATUS\_REQUEST\_ERROR:

#009        rv = DidReceiveError(error, secure\_failure);

#010        break;

......

#027

#028    if (rv == ERR\_IO\_PENDING) {

#029      session\_callback\_->AddRef();  // balanced when callback runs.

#030    } else if (callback\_) {

**#031      DoCallback(rv);**

#032    }

#033  }

通过上面的函数可以看到，当数据接收完成后，就会调用**DoCallback**函数处理接收到的数据。**DoCallback**函数的代码如下：

#001  void HttpTransactionWinHttp::DoCallback(int rv) {

#002    DCHECK(rv != ERR\_IO\_PENDING);

#003    DCHECK(callback\_);

#004

#005    // since Run may result in Read being called, clear callback\_ up front.

**#006    CompletionCallback\* c = callback\_;**

**#007    callback\_ = NULL;**

**#008    c->Run(rv);**

#009  }

看到这里又是一个回调函数**c->Run**的通知，它是调用开始创建这个连接时设置的回调对象。如果是HTTP请求，那么这个请求回调函数是对象URLRequestHttpJob里的函数，也就是调用URLRequestHttpJob::OnReadCompleted函数，这个函数是当数据接收完成，或接收失败，或者接收还没有完成时都会调用。这个函数代码如下：

#001  void URLRequestHttpJob::OnReadCompleted(int result) {

#002    read\_in\_progress\_ = false;

#003

这里是接收数据完成。

#004    if (result == 0) {

#005      NotifyDone(URLRequestStatus());

#006    } else if (result < 0) {

这里是接收数据出错划。

#007      NotifyDone(URLRequestStatus(URLRequestStatus::FAILED, result));

#008    } else {

这里是接收数据还没有完成。

#009      // Clear the IO\_PENDING status

#010      SetStatus(URLRequestStatus());

#011    }

#012

#013    NotifyReadComplete(result);

#014  }

当上面读取数据完成时，就开始把接收到数据通过类ResourceDispatcherHost来发送出去，而类ResourceDispatcherHost发送数据的方式比较特别，它不是通过消息把整块数据用命名管道发送的，而是通过共享内存的方式让另一个进程来读取数据，这样达到速度快的特点，可见可多米处理处理考虑的都是速度，它的代码如下：

#001  bool OnReadCompleted(int request\_id, int\* bytes\_read) {

#002      if (!\*bytes\_read)

#003        return true;

#004      DCHECK(read\_buffer\_.get());

#005

#006      if (!rdh\_->WillSendData(render\_process\_host\_id\_, request\_id)) {

#007        // We should not send this data now, we have too many pending requests.

#008        return true;

#009      }

#010

这里创建共享内存。

#011      SharedMemoryHandle handle;

#012      if (!read\_buffer\_->GiveToProcess(render\_process\_, &handle)) {

#013        // We wrongfully incremented the pending data count. Fake an ACK message

#014        // to fix this. We can't move this call above the WillSendData because

#015        // it's killing our read\_buffer\_, and we don't want that when we pause

#016        // the request.

#017        rdh\_->OnDataReceivedACK(render\_process\_host\_id\_, request\_id);

#018        return false;

#019      }

#020

把共享内存通过管道消息发送给渲染进程。

#021      receiver\_->Send(new ViewMsg\_Resource\_DataReceived(

#022          routing\_id\_, request\_id, handle, \*bytes\_read));

#023

#024      return true;

#025    }

#026

共享内存是使用Windows API函数CreateFileMapping来创建内存共享文件实现的，具体实现方法请参考类SharedMemory的实现。这里既然把消息通过管道发送出去了，那么在另一个线程里肯定就处理的这个消息，下一次再来分析那部份代码。

**谷歌浏览器的源码分析(31)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到通过管道把接收到的HTTP数据通知另一个线程处理，它不是直接发送数据过去，而是把数据在共享内存里的句柄发送过去，达到高效通讯的目的。下面就来分析资源处理进程里，接收到这个消息之后，做些什么处理。这个消息的处理代码如下：

#001  void ResourceDispatcher::OnReceivedData(int request\_id,

#002                                          SharedMemoryHandle shm\_handle,

#003                                          int data\_len) {

#004    // Acknowlegde the reception of this data.

**回应这个消息，说已经收到数据了。**

#005    IPC::Message::Sender\* sender = message\_sender();

#006    if (sender)

#007      sender->Send(

#008          new ViewHostMsg\_DataReceived\_ACK(MSG\_ROUTING\_NONE, request\_id));

#009

#010    DCHECK((shm\_handle && data\_len > 0) || (!shm\_handle && !data\_len));

**打开共享内存文件，使用只读的方式。**

#011    SharedMemory shared\_mem(shm\_handle, true);  // read only

#012

**查找到请求下载的资源的请求标识号。**

#013    PendingRequestList::iterator it = pending\_requests\_.find(request\_id);

**如果没有找到相应的请求标识号，就直接返回，不用处理这些数据。**

#014    if (it == pending\_requests\_.end()) {

#015      // this might happen for kill()ed requests on the webkit end, so perhaps

#016      // it shouldn't be a warning...

#017      DLOG(WARNING) << "Got data for a nonexistant or finished request";

#018      return;

#019    }

#020

**这里找到相应的请求标识号，就把数据放到请求信息里处理。**

#021    PendingRequestInfo& request\_info = it->second;

#022

#023    if (data\_len > 0 && shared\_mem.Map(data\_len)) {

#024      RESOURCE\_LOG("Dispatching " << data\_len << " bytes for " <<

#025                   request\_info.peer->GetURLForDebugging());

#026      const char\* data = static\_cast<char\*>(shared\_mem.memory());

#027      request\_info.peer->OnReceivedData(data, data\_len);

#028    }

#029  }

上面这个函数实现接收到HTTP数据，并且把数据放到请求的缓冲区里，但它没有知道什么时候接收数据完成，显然有另外一个消息来做这些的工作，就是下面类ResourceDispatcherHost的函数：

#001    bool OnResponseCompleted(int request\_id, const URLRequestStatus& status) {

#002      receiver\_->Send(new ViewMsg\_Resource\_RequestComplete(

#003          routing\_id\_, request\_id, status));

#004

#005      // If we still have a read buffer, then see about caching it for later...

#006      if (spare\_read\_buffer\_) {

#007        read\_buffer\_.reset();

#008      } else if (read\_buffer\_.get() && read\_buffer\_->memory()) {

#009        spare\_read\_buffer\_ = read\_buffer\_.release();

#010      }

#011      return true;

#012    }

这个函数里通过发送消息ViewMsg\_Resource\_RequestComplete来通知资源进程已经把网络的数据接收完成了，可以进入下一步处理。然后在资源进程里就会处理这个消息，下一次再来分析这方面的代码。

**谷歌浏览器的源码分析(32)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到在类ResourceDispatcher会收到接收HTTP数据消息，并进一步处理数据。那么ResourceDispatcher类又把接收到的数据发往何处呢？这是需要我们去搞懂它的。通过进一步的跟踪，会发现在ResourceDispatcher::OnReceivedData函数调用WebCore::ResourceHandleInternal类来处理，也就是把接收到的数据抛给WebCore来处理了。如下面的代码：

#001  void ResourceDispatcher::OnReceivedData(int request\_id,

#002                                          SharedMemoryHandle shm\_handle,

#003                                          int data\_len) {

#004    // Acknowlegde the reception of this data.

#005    IPC::Message::Sender\* sender = message\_sender();

......

#023    if (data\_len > 0 && shared\_mem.Map(data\_len)) {

#024      RESOURCE\_LOG("Dispatching " << data\_len << " bytes for " <<

#025                   request\_info.peer->GetURLForDebugging());

#026      const char\* data = static\_cast<char\*>(shared\_mem.memory());

**#027      request\_info.peer->OnReceivedData(data, data\_len);**

#028    }

#029  }

上面第27行代码就是调用webcore里类ResourceHandleInternal::OnReceivedData函数，这样就把数据保存到webcore里面，也就是webkit里面了。经过如下面的调用过程：

1）  WebCore::ResourceLoader::didReceiveData

2）  WebCore::SubresourceLoader::didReceiveData

3）  WebCore::Loader::didReceiveData

4）  WebCore::CachedImage::data

这里就是把图像的数据缓存起来，以便后面调用渲染引擎来显示。这一次就分析到这里，总算把数据怎么样放到webkit里搞清楚了，下一次再来看看webkit是怎么样把数据显示出来的。

**谷歌浏览器的源码分析(33)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

上一次说到图像缓存起来，其实很多情况下是文本的显示，也就是HTML的解释。要把网页显示出来，肯定是先从HTTP里收到网页数据，然后再使用HTML分析器来解释HTML语言，最后根据HTML来生成所有可以显示的元素，再由于这些元素生成BMP位图，这样只需要把BMP位置显示到窗口里就万事大吉了。这个过程看起来简单，其实是一个非常复杂的过程，现在就带你去深入地分析这个过程，就基本把Webkit的过程搞清楚了，同时也把chrome分析网页的过程搞清楚了。这个过程如下：

1）  ResourceDispatcher::OnReceivedData()  资源分派类接收到网页数据。

2）  WebCore::ResourceHandleInternal::OnReceivedData()  WebCore::ResourceHandleInternal类接收到数据。

3）  WebCore::ResourceLoader::didReceiveData()  资源加载类接收到数据。

4）  WebCore::MainResourceLoader::didReceiveData()  主资源类接收到数据。

5）  WebCore::MainResourceLoader::addData() 主资源类保存数据。

6）  WebCore::FrameLoader::receivedData() 框架加载类保存数据。

7）  WebCore::DocumentLoader::receivedData() 文档加载类保存数据。

8）  WebCore::DocumentLoader::commitLoad() 文档加载类提交所有接收的数据。

9）  WebCore::FrameLoader::committedLoad() 框架加载类提交数据。

10）    WebFrameLoaderClient::committedLoad() 网页框架加载类提交数据。

11）    WebFrameImpl::DidReceiveData() 网页框架实现类保存提交的数据。

12）    WebCore::FrameLoader::addData() 框架加载类保存数据。

13）    WebCore::FrameLoader::write() 把网页数据写入HTML缓冲。

14）    WebCore::HTMLTokenizer::write()  HTML终结符分析器进行保存。

15）    WebCore::HTMLTokenizer::processToken()  HTML终结符分析器分析HTML数据。

16）    WebCore::HTMLParser::parseToken()  HTML分析器分析网页数据。

17）    WebCore::HTMLParser::insertNode() 分析到一个网页里的节点，开始插入。

18）    WebCore::Text::attach()  发现一个文本节点并保存。

19）    WebCore::Node::createRendererIfNeeded()  创建可以渲染的节点。

20）    WebCore::Text::createRenderer() 开始创建文本渲染对象。

21）    WebCore::RenderText::RenderText() 创建文本渲染对象RenderText。

从上面的过程，可以看到分析过程是比较复杂的，不过，总算把分析网页数据这个主线抓住了，其它的东西，都是为了这条主线而进行的。只要跟着这条主线，把相应的类再进一步分析，就可以把整个程序搞得一清二楚了。在最后一步里，就会生成RenderObject对象，而所有的RenderObject对象是根据分析HMTL生成一棵树来保存起来。当界面上要显示出来时，其实就是去遍历整个RenderObject对象树。下一次再来分析界面怎么样显示这些对象的。

**谷歌浏览器的源码分析(34)**[收藏](javascript:d=document;t=d.selection?(d.selection.type!='None'?d.selection.createRange().text:''):(d.getSelection?d.getSelection():'');void(saveit=window.open('http://wz.csdn.net/storeit.aspx?t='+escape(d.title)+'&u='+escape(d.location.href)+'&c='+escape(t),'saveit','scrollbars=no,width=590,height=300,left=75,top=20,status=no,resizable=yes'));saveit.focus();)

通过上一次的分析，我们看到所有网页数据经过HTML分析器之后，都会变成一个一个RenderObject对象，那么这些RenderObject对象又是怎么样显示到界面上面的呢？现在就带着这个疑问来分析下面的代码，这样肯定会找到解决方法的。怎么样找到入口呢？其实可以先从界面显示的类开始，可以看到显示界面的窗口类名称叫做Chrome\_RenderWidgetHostHWND,有了这个类名称，就可以到代码里查看它在那里了。

#001  class RenderWidgetHost;

#002  class WebMouseEvent;

#003  class WebCursor;

#004

#005  typedef CWinTraits<WS\_CHILD | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS, 0>

#006      RenderWidgetHostHWNDTraits;

#007

**#008  static const wchar\_t\* const kRenderWidgetHostHWNDClass =**

**#009      L"Chrome\_RenderWidgetHostHWND";**

可看到这个窗口类名称是定义在这里，再跟着kRenderWidgetHostHWNDClass来查找，就会找到显示窗口，如下：

#001  class RenderWidgetHostHWND :

#002    public CWindowImpl<RenderWidgetHostHWND,

#003                       CWindow,

#004                       RenderWidgetHostHWNDTraits>,

#005    public RenderWidgetHostView {

#006   public:

#007    RenderWidgetHostHWND(RenderWidgetHost\* render\_widget\_host);

#008    virtual ~RenderWidgetHostHWND();

#009

#010    void set\_close\_on\_deactivate(bool close\_on\_deactivate) {

#011      close\_on\_deactivate\_ = close\_on\_deactivate;

#012    }

#013

#014    void set\_parent\_hwnd(HWND parent) { parent\_hwnd\_ = parent; }

#015

**#016    DECLARE\_WND\_CLASS\_EX(kRenderWidgetHostHWNDClass, CS\_DBLCLKS, 0);**

通过上面的分析，就可以找到显示网页的窗口类RenderWidgetHostHWND，在这个类里，主要显示的位置是在void RenderWidgetHostHWND::OnPaint(HDC dc)函数里面，它的代码如下：

#001  void RenderWidgetHostHWND::OnPaint(HDC dc) {

#002    DCHECK(render\_widget\_host\_->process()->channel());

#003

#004    CPaintDC paint\_dc(m\_hWnd);

#005    HBRUSH white\_brush = reinterpret\_cast<HBRUSH>(GetStockObject(WHITE\_BRUSH));

#006

**#007    RenderWidgetHost::BackingStore\* backing\_store =**

**#008        render\_widget\_host\_->GetBackingStore()**;

#009

#010    if (backing\_store) {

#011      gfx::Rect damaged\_rect(paint\_dc.m\_ps.rcPaint);

#012

#013      gfx::Rect bitmap\_rect(

#014          0, 0, backing\_store->size().width(), backing\_store->size().height());

#015

#016      gfx::Rect paint\_rect = bitmap\_rect.Intersect(damaged\_rect);

#017      if (!paint\_rect.IsEmpty()) {

#018        BitBlt(paint\_dc.m\_hDC,

#019               paint\_rect.x(),

#020               paint\_rect.y(),

#021               paint\_rect.width(),

#022               paint\_rect.height(),

#023               backing\_store->dc(),

#024               paint\_rect.x(),

#025               paint\_rect.y(),

#026               SRCCOPY);

#027      }

......

#058  }

其实这个函数是通过如下发送消息给另一个进程进行渲染成BMP的图片，

Send(new ViewMsg\_Repaint(routing\_id\_, view\_size));

那么谁来接收ViewMsg\_Repaint消息呢？继续细心地查找，就到在如下类函数里处理：

void RenderWidget::OnMsgRepaint(const gfx::Size& size\_to\_paint)

在这个函数，并不是最终的结果，它又会调用其它线程来处理渲染，以便达到异步的结果。它的调用过程如下：

1）  RenderWidget::DoDeferredPaint()  线程里开始渲染网页显示

2）  RenderWidget::PaintRect() 窗口里开始进行显示

3）  WebViewImpl::Paint() web视类开始显示。

4）  WebFrameImpl::Paint() web框架类开始显示。

5）  WebCore::ScrollView::paint() 滚动窗口显示。

6）  WebCore::Frame::paint() WebCore里的框架显示。

7）  WebCore::RenderLayer::paint() 分层显示。

8）  WebCore::RenderLayer::paintLayer()

9）  WebCore::RenderBlock::paint()  在每一层里显示每一块区域。

10）    WebCore::RenderBlock::paintObject() 显示这一区域的对象。

11）    WebCore::RenderBlock::paintContents() 显示需要显示的内容。

12）    WebCore::RenderFlow::paintLines() 这里需要显示文字。

13）    WebCore::RootInlineBox::paint() 开始显示一行文字。

14）    WebCore::InlineFlowBox::paint() 进行一行文字排列。

15）    WebCore::InlineTextBox::paint()

16）    WebCore::GraphicsContext::drawText()  进行一个一个文字显示。

17）    WebCore::Font::drawText()  这里调用字体类来把文字的编码变成位图。

18）    WebCore::Font::drawSimpleText()  这里把位图显示到界面内存里。

通过上面的分析，可以看到显示一串文字的过程是如此复杂的过程。其它图片显示的过程也是一样，都把它们变成位图，然后再分层显示出来。那么JavaScript是怎么样显示的呢？这个会比上面的过程更加复杂，后面再仔细地分析它。下一次，主要仔细地看看这些过程里的一些类功能。