浏览 Chromium 源代码目录结构（Getting Around the Chromium Source Code Directory Structure）

浏览 Chromium 源代码目录结构

目录

高级概述

Chromium 分为两个主要部分（不包括其他库）：浏览器和渲染器（包括 Blink，即网页引擎）。浏览器是主要进程，代表所有的 UI 和 I/O。渲染器是（通常是）每个标签页的子进程，由浏览器驱动。它嵌入 Blink 来进行布局和渲染。

你需要阅读并熟悉我们的多进程架构multi-process architecture以及 Chromium 如何显示网页how Chromium displays web pages。

顶级项目

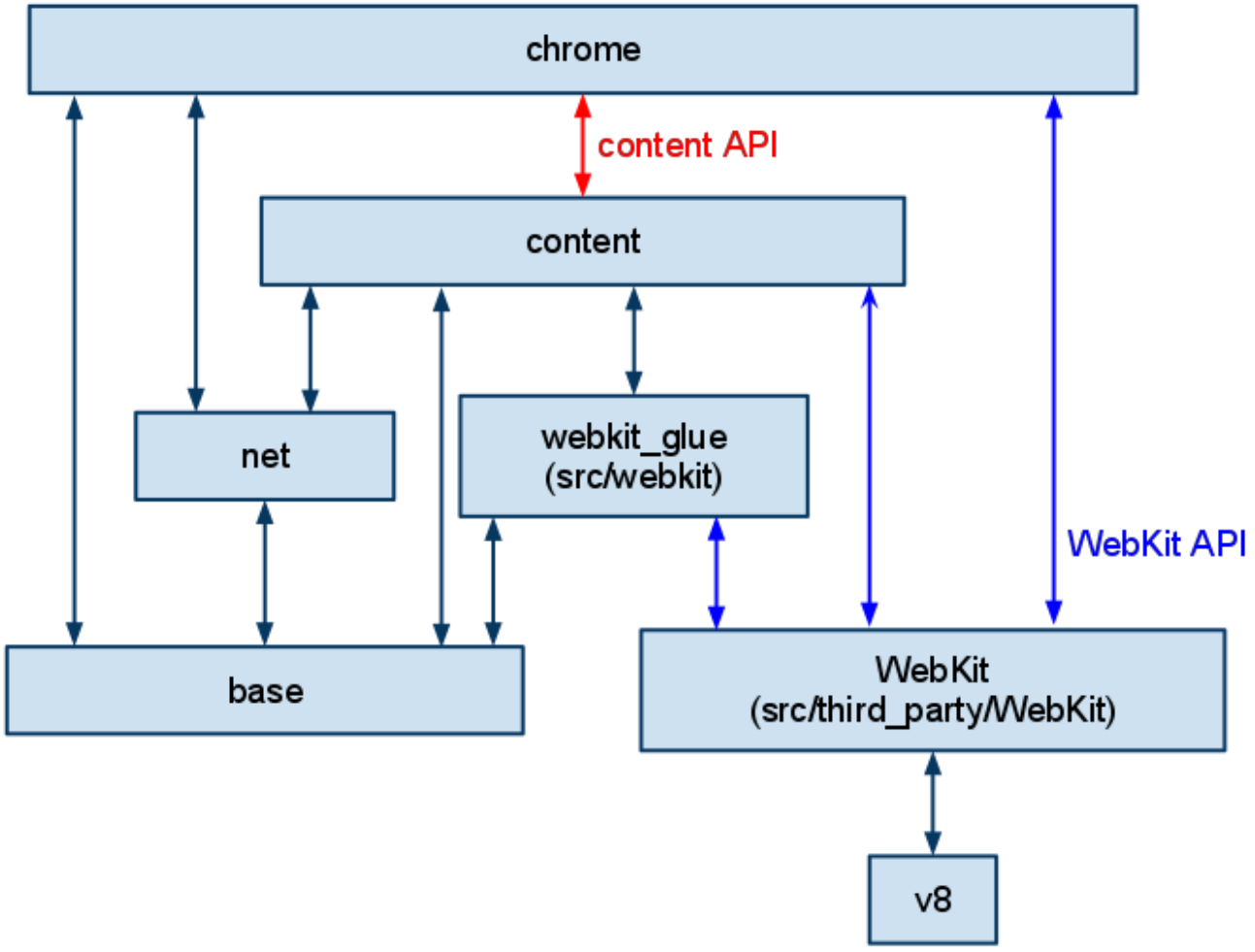
当你浏览和搜索 Chromium 代码或检出 Chromium 代码时，你会注意到一些顶级目录：

- **android_webview**：为 src/content 提供适合集成到 Android 平台的外观。不打算用于单个 Android 应用程序（APK）。更多More information关于 Android WebView 源代码组织的信息。
- **apps**：Chrome 打包应用程序。
- **base**：所有子项目之间共享的通用代码。这里包含字符串操作、通用实用程序等。仅当它必须在多个顶级项目之间共享时，才添加到这里。
- **breakpad**：Google 的开源崩溃报告项目。直接从 Google Code 的 Subversion 存储库中提取。
- **build**：所有项目共享的构建相关配置。
- **cc**：Chromium 合成器实现。
- **chrome**：Chromium 浏览器（见下文）。
- **chrome/test/data**：用于运行某些测试的数据文件。
- **components**：包含 Content 模块作为它们依赖的最高层的组件目录。
- **content**：多进程沙箱浏览器所需的核心代码（见下文）。关于More information我们为什么要分离这些代码的更多信息。
- **device**：跨平台的常见低级硬件 API 抽象。
- **net**：为 Chromium 开发的网络库。当在 webkit 存储库中运行我们的简单测试外壳时，可以单独使用此库。另见 chrome/common/net。
- **sandbox**：沙箱项目，试图防止被入侵的渲染器修改系统。
- **skia + third_party/skia**：Google 的 Skia 图形库。我们在 ui/gfx 中的附加类封装了 Skia。
- **sql**：我们对 sqlite 的封装。
- **testing**：包含 Google 开源的 GTest 代码，我们用于单元测试。
- **third_party**：200 多个小型和大型“外部”库，例如图像解码器、压缩库和网页引擎 Blink（因为它继承了 WebKit 的许可证限制）。添加新包Adding new packages。
    - **.../blink/renderer**：负责将 HTML、CSS 和脚本转换为绘制命令和其他状态变化的网页引擎。
- **tools**

- **ui/gfx**：共享图形类。这些类构成了 Chromium UI 图形的基础。
- **ui/views**：用于 UI 开发的简单框架，提供渲染、布局和事件处理。大多数浏览器 UI 是在此系统中实现的。此目录包含基础对象。一些更具体的浏览器对象位于 chrome/browser/ui/views 中。
- **url**：Google 的开源 URL 解析和规范化库。
- **v8**：V8 Javascript 库。直接从 Google Code 的 Subversion 存储库中提取。

由于历史原因，有一些小的顶级目录。未来，新顶级目录的指导方针是用于应用程序（例如 Chrome、Android WebView、Ash）。即使这些应用程序有多个可执行文件，代码也应位于应用程序的子目录中。

这是一个稍微过时的依赖关系图。特别是，WebKit 被 blink/renderer 取代。较低模块不能直接包含较高模块的代码（即 content 不能包含 chrome 的头文件），但可以使用嵌入 API 与之通信。



content/ 目录树的快速参考

- **browser**：处理所有 I/O 和与子进程通信的应用程序后端。这与渲染器通信以管理网页。
- **common**：多个进程之间共享的文件（即浏览器和渲染器，渲染器和插件等）。这是 Chromium 特有的代码（不适合放在 base 中）。
- **gpu**：用于 GPU 进程的代码，用于 3D 合成和 3D API。
- **plugin**：在其他进程中运行浏览器插件的代码。
- **ppapi_plugin**：用于 Pepper 插件进程的代码。
- **renderer**：每个标签页中的子进程代码。它嵌入了 WebKit 并与浏览器通信进行 I/O。
- **utility**：在沙箱进程中运行随机操作的代码。浏览器进程在希望对不受信任的数据执行操作时使用它。
- **worker**：运行 HTML5 Web Workers 的代码。

chrome/ 目录树的快速参考

- **app**：程序的最基本级别。在启动时运行，并根据当前进程的能力分派到浏览器或渲染器代码。它包含 chrome.exe 和 chrome.dll 的项目。除非是资源（如图像和字符串），通常不需要更改这些内容。
  - **locales**：构建本地化 DLL 的项目。
  - **resources**：图标和光标。
  - **theme**：窗口主题的图像。
- **browser**：包括主窗口、UI 和处理所有 I/O 和存储的应用程序后端。它与渲染器通信以管理网页。
  - **ui**：UI 功能和功能的模型、视图和控制器代码。
- **common**：浏览器和渲染器之间共享的特定于 Chrome 模块的文件。
  - **net**：在 net 顶级模块之上的一些 Chromium 特有内容。这应该与 browser/net 合并。
- **installer**：制作安装程序（MSI 包）的源文件和项目。
- **renderer**：在渲染器进程中运行的 Chrome 特定代码。它向 content 模块添加了 Chrome 功能，如自动填充、翻译等。
- **test**：

    - **automation**：用于驱动浏览器 UI 的测试，例如在 test/ui、test/startup 等中。它与浏览器中的 browser/automation 通信。
    - **page_cycler**：用于运行页面循环测试（用于性能测量）的代码。参见 tools/perf/dashboard。
    - **reliability**：用于分布式测试页面加载的可靠性测试，用于可靠性指标和崩溃查找。
    - **selenium**：用于运行 selenium 测试的代码，这是一个第三方测试套件，用于 Ajaxy 和 JavaScript 内容。参见 test/third_party/selenium_core。
    - **startup**：用于测量启动性能的测试。参见 tools/perf/dashboard 和 tools/test/reference_build。
    - **ui**：用于浏览器 UI 的 UI 测试，打开标签页等。它使用 test/automation 来执行大多数操作。
    - **unit**：单元测试的基础代码。单个测试的测试代码通常与被测试代码一起放在 *_unittest.cc 文件中。
- **third_party**：特定于 Chromium 的第三方库。其他一些第三方库位于顶级 third_party 库中。
- **tools**
    - **build**：与构建相关的工具和随机内容。
    - **memory**：与内存相关的工具。目前包括用于设置页面堆选项的 gflags。
    - **perf/dashboard**：用于将性能日志（例如 test/startup_test）转换为数据和图表的代码。
    - **profiles**：用于生成随机历史数据的生成器。用于制作测试配置文件。

个人学习计划

最终你会设置好构建环境并开始工作。在一个理想的世界中，我们会有足够的时间阅读每一行代码并在编写第一行代码之前理解它。但在实践中，如果我们什么都不做，只阅读一天内发生的所有检入代码，我们会很难完成，所以我们中的任何人都不可能阅读所有代码。那么，我们该怎么办呢？我们建议你制定自己的学习计划，以下是一些建议的起点。

幸运的是，Chromium 有一些高质量的设计文档。虽然这些文档可能有点过时（例如，跟随阅读时，你可能会发现引用了已被移动、重命名或重构的文件），但能够理解代码整体如何组合在一起是非常有用的。

- **阅读最重要的开发文档**
  - multi-process-architecture
  - displaying-a-web-page-in-chrome
  - inter-process-communication
  - threading
- **查看你的组是否有任何启动文档**
  可能有一些文档是同一代码组的人会关心的，而其他人不需要知道那么多细节。
- **学习一些代码习惯：**
  - important-abstractions-and-data-structures
  - smart-pointer-guidelines
  - chromium-string-usage
- **以后有时间时，浏览所有设计文档，阅读相关内容。**
- **熟练使用代码搜索（或你选择的代码浏览工具）。**
- **学习向谁询问代码工作原理。**
- **如果可以的话，使用调试器调试你需要学习的代码，如果不能，使用日志语句和 grep。**
- **查看你需要了解和当前了解的差异。例如，如果你的组做了很多 GUI 编程，那么你可以投入时间学习 GTK+、Win32 或 Cocoa 编程。**
- **使用source.chromium.org 搜索源代码。如果代码发生变化而我们的文档不再准确，这会特别有帮助。**

常见操作的代码路径

关于 Chromium 如何显示网页how Chromium displays web pages，有更多的信息和示例。

应用启动

1. **WinMain** 函数位于 chrome/app/main.cc，并链接在 chrome 项目中。
2. **WinMain** 启动 Google 更新客户端，这是安装程序/自动更新程序。它将找到当前版本的子目录，并从那里加载 chrome.dll。
3. 它调用 chrome_main.cc 中 chrome_dll 项目的 ChromeMain。

4. **ChromeMain** 为通用组件进行初始化，然后根据命令行标志指示是否应为子进程，转发到 chrome/renderer/renderer_main.cc 中的 RendererMain，或者不这样做则加载应用程序的新副本。在启动时，我们正在启动浏览器。
5. **BrowserMain** 进行常见的浏览器初始化。它有不同的模式，用于运行已安装的 web 应用程序、连接到自动化系统（如果正在测试浏览器）等。
6. 它调用 browser_init.cc 中的 LaunchWithProfile，这在 chrome/browser/ui/browser.cc 中创建一个新的 Browser 对象。这个对象封装了应用程序中的一个顶级窗口。此时添加了第一个标签页。

标签页启动和初始导航

- **BrowserImpl::AddTab** 在 weblayer/browser/browser_impl.cc 中被调用以添加新标签页。
- 它将从 browser/tab_contents/tab_contents.cc 创建一个新的 TabContents 对象。
- **TabContents** 通过 RenderViewHostManager 的 Init 函数（位于 chrome/browser/tab_contents/render_view_host_manager.cc）创建一个 RenderViewHost（chrome/browser/renderer_host/render_view_host.cc）。根据 SiteInstance，RenderViewHost 要么生成一个新的渲染器进程，要么重用现有的 RenderProcessHost。**RenderProcessHost** 是浏览器中表示单个渲染器子进程的对象。
- 由标签页内容拥有的 NavigationController（位于 chrome/browser/tab_contents/navigation_controller.cc）被指示导航到新标签页的 URL。导航控制器在 **NavigationController::LoadURL** 中加载 URL。从第 3 步起，“从 URL 栏导航”描述了从此点开始的内容。

从 URL 栏导航

- 当用户在 URL 栏中键入或接受条目时，自动完成编辑框确定最终目标 URL 并将其传递给 **AutocompleteEdit::OpenURL**。（这可能不是用户输入的确切内容，例如，在搜索查询的情况下生成 URL。）
- 导航控制器被指示导航到该 URL，在 **NavigationController::LoadURL** 中加载 URL。
- **NavigationController** 使用它为此特定页面转换创建的 **NavigationEntry** 调用 **TabContents::Navigate**。如果需要，它将创建一个新的 **RenderViewHost**，这将导致在渲染器进程中创建一个 **RenderView**。如果这是第一次导航或渲染器崩溃，则不会存在 RenderView，因此这也将从崩溃中恢复。

- **Navigate** 转发到 **RenderViewHost::NavigateToEntry**。导航控制器存储此导航条目，但由于它不确定转换是否会发生（例如，主机可能无法解析），因此该条目被标记为“挂起”。
- **RenderViewHost::NavigateToEntry** 将 **ViewMsg_Navigate** 发送到渲染器进程中的新 **RenderView**。
- 当被告知导航时，**RenderView** 可能会导航，可能会失败，或者可能会导航到其他地方（例如，如果用户点击链接）。**RenderViewHost** 等待 **RenderView** 的 **ViewHostMsg_FrameNavigate** 消息。
- 当 WebKit 提交加载（服务器响应并向我们发送数据）时，**RenderView** 发送此消息，该消息在 **RenderViewHost::OnMsgNavigate** 中处理。
- **NavigationEntry** 更新加载信息。在链接点击的情况下，浏览器从未见过此 URL。如果导航是浏览器发起的，如启动情况，可能会有重定向改变了 URL。
- **NavigationController** 更新其导航列表以反映此新信息。

导航和会话历史

每个 **NavigationEntry** 都存储一个页面 ID 和一块历史状态数据。页面 ID 用于唯一标识页面加载，因此我们知道它对应于哪个 **NavigationEntry**。它在页面提交时分配，因此挂起的 **NavigationEntry** 的页面 ID 为 -1。历史状态数据只是一个 **WebCore::HistoryItem** 序列化为字符串。此项包括页面 URL、子帧 URL 和表单数据。

- **当浏览器发起请求时**（在 URL 栏中键入或点击后退/前进/重新加载）
  - 代表导航发出一个 **WebRequest**，以及用于簿记的额外信息，如页面 ID。新导航的 ID 为 -1。导航到旧条目时，其 ID 为首次访问页面时分配给 **NavigationEntry** 的 ID。加载提交后将查询此额外信息。
  - 主 **WebFrame** 被告知加载新请求。
- **当渲染器发起请求时**（用户点击链接，JavaScript 改变位置等）：
  - **WebCore::FrameLoader** 被告知通过其众多不同的加载方法之一加载请求。
- 无论哪种情况，当从服务器接收到第一个数据包时，加载被提交（不再“挂起”或“临时”）。
  - 如果这是新的导航，**WebCore** 将创建一个新的 **HistoryItem** 并将其添加到 **BackForwardList**（WebCore 类）。这样，我们可以区分哪些导航是新的，哪些是会话历史导航。

- **RenderView::DidCommitLoadForFrame** 处理加载提交。在这里，通过 **ViewHostMsg_UpdateState** 消息将前一个页面的状态存储在会话历史中。这将通知浏览器使用新历史状态更新相应的 **NavigationEntry**（由 **RenderView** 当前的页面 ID 标识）。
- **RenderView** 当前的页面 ID 更新为反映提交的页面。对于新导航，将生成一个新的唯一页面 ID。对于会话历史导航，它将是首次访问时分配的页面 ID，我们在启动导航时将其存储在 **WebRequest** 上。
- 一个 **ViewHostMsg_FrameNavigate** 消息发送到浏览器，使用新 URL 和其他信息更新相应的 **NavigationEntry**（由 **RenderView** 新更新的页面 ID 标识）。