多进程架构（Multi-process Architecture）

多进程架构（Multi-process Architecture）

本文档描述了Chromium的高级架构以及它是如何在多种进程类型之间划分的。

问题

几乎不可能构建一个从不崩溃或卡死的渲染引擎。同样，几乎不可能构建一个完全安全的渲染引擎。

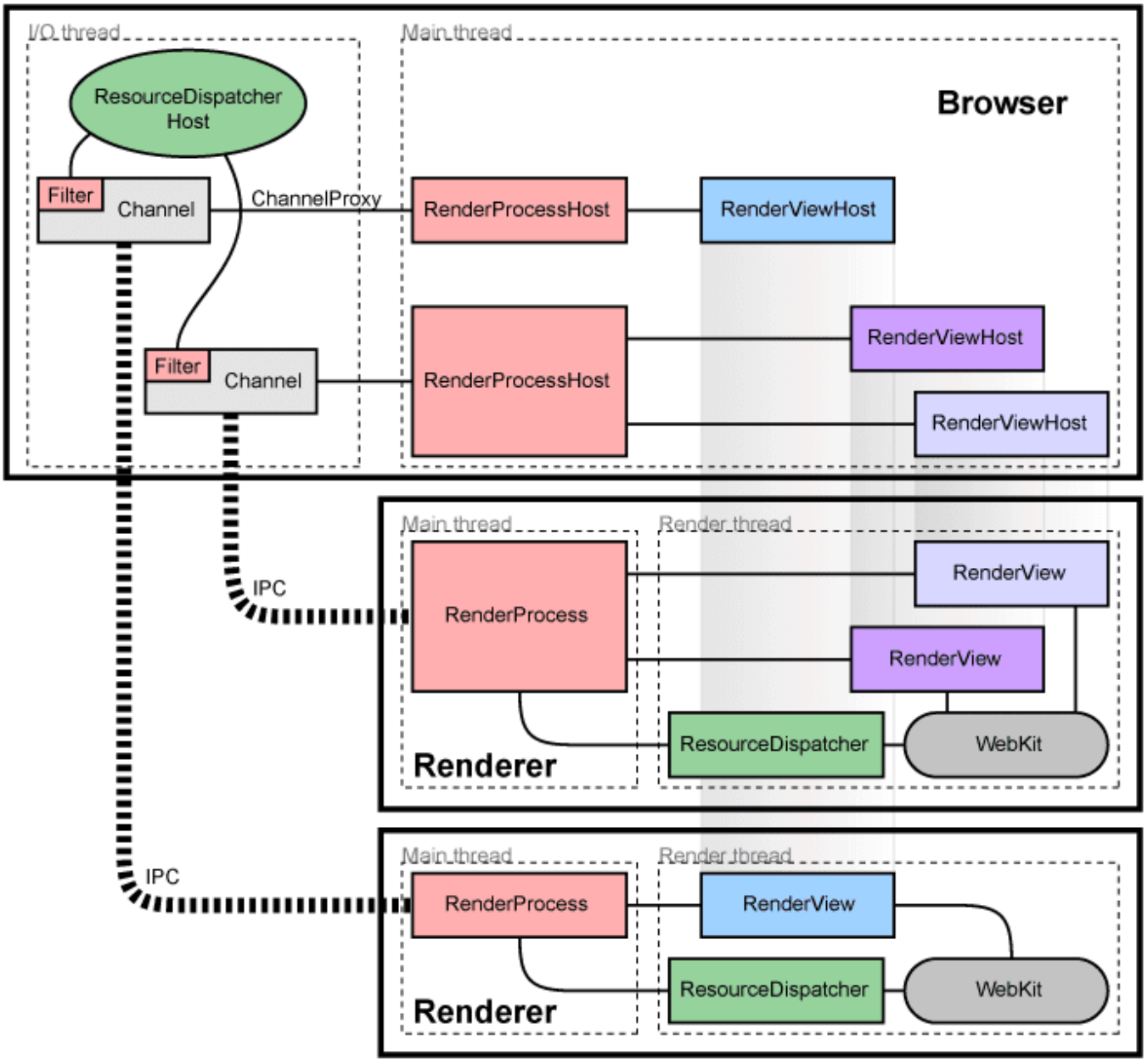
在某些方面，2006年左右的网页浏览器状态类似于过去的单用户协作多任务操作系统。在这样的操作系统中，表现不佳的应用程序可以导致整个系统崩溃，同样，一个表现不佳的网页在网页浏览器中也可以造成同样的后果。只需要一个渲染引擎或插件的漏洞就能使整个浏览器和所有当前运行的标签页崩溃。

现代操作系统更加健壮，因为它们将应用程序放入相互隔离的独立进程中。一个应用程序的崩溃通常不会影响其他应用程序或操作系统的完整性，每个用户对其他用户数据的访问也受到限制。Chromium的架构旨在实现这种更健壮的设计。

架构概述

Chromium使用多个进程来保护整体应用程序免受渲染引擎或其他组件中的错误和故障的影响。它还限制了每个渲染引擎进程对其他进程和系统其余部分的访问。在某些方面，这为网页浏览带来了内存保护和访问控制为操作系统带来的好处。

我们将运行UI并管理渲染器及其他进程的主要进程称为“浏览器进程”或“浏览器”。同样，处理网页内容的进程被称为“渲染器进程”或“渲染器”。渲染器使用 Blink 开源布局引擎来解释和布局HTML。



管理渲染器进程

每个渲染器进程都有一个全局的RenderProcess对象，该对象管理与父浏览器进程的通信并维护全局状态。浏览器为每个渲染器进程维护一个对应的RenderProcessHost，负责管理渲染器的浏览器状态和通信。浏览器和渲染器使用Mojo或Chromium的传统IPC系统进行通信。

管理框架和文档

每个渲染器进程有一个或多个RenderFrame对象，对应包含内容的文档框架。浏览器进程中的对应RenderFrameHost管理与该文档相关的状态。每个RenderFrame都有一个路由ID，用于区分同一渲染器中的多个文档或框架。这些ID在一个渲染器内是唯一的，但在浏览器内不是，因此识别一个框架需要RenderProcessHost和路由ID。浏览器

通过这些RenderFrameHost对象与渲染器中的特定文档进行通信，后者知道如何通过Mojo或传统IPC发送消息。

组件和接口

在渲染器进程中：

- RenderProcess处理与浏览器中对应的RenderProcessHost的Mojo设置和传统IPC。每个渲染器进程中只有一个RenderProcess对象。
- RenderFrame对象通过Mojo与浏览器进程中的对应RenderFrameHost和Blink层通信。该对象表示标签页或子框架中的一个网页文档的内容。

在浏览器进程中：

- Browser对象表示一个顶级浏览器窗口。
- RenderProcessHost对象表示浏览器一侧的单个浏览器 ↔ 渲染器IPC连接。浏览器进程中每个渲染器进程都有一个RenderProcessHost。
- RenderFrameHost对象封装了与RenderFrame的通信，而RenderWidgetHost处理浏览器中RenderWidget的输入和绘制。

  有关这种嵌入工作的更详细信息，请参见 How Chromium displays web pages。

共享渲染器进程

通常，每个新窗口或标签页在一个新进程中打开。浏览器将生成一个新进程并指示其创建一个单独的RenderFrame，如果页面中有更多的iframe（可能在不同的进程中），则可能会创建更多。

有时，在标签页或窗口之间共享渲染器进程是必要的或可取的。例如，Web应用程序可以使用window.open创建另一个窗口，如果新文档属于相同来源，则必须共享同一进程。如果进程总数过多，Chromium也有策略将新标签页分配给现有进程。这些考虑和策略在进程模型中有描述。

检测崩溃或表现不佳的渲染器

每个与浏览器进程的Mojo或IPC连接都会监视进程句柄。如果这些句柄发出信号（signaled），则表示渲染器进程已崩溃，受影响的标签页和框架会收到崩溃通知。Chromium会显示一个“sad tab”或“sad frame”图像，通知用户渲染器已崩溃。可以通过

按下重新加载按钮或开始新的导航来重新加载页面。当发生这种情况时，Chromium会注意到没有渲染器进程并创建一个新进程。

沙箱化渲染器

由于渲染器在一个单独的进程中运行，我们有机会通过沙箱技术sandboxing限制其对系统资源的访问。例如，我们可以确保渲染器只能通过Chromium的网络服务访问网络。同样，我们可以使用主操作系统的内置权限限制其对文件系统、用户显示和输入的访问。这些限制显著减少了被攻陷的渲染器进程能够完成的操作。

释放内存

由于渲染器在单独的进程中运行，将隐藏的标签页视为低优先级变得简单。在Windows上，最小化的进程通常会自动将其内存放入“可用内存”池。在内存不足的情况下，Windows会将这些内存交换到磁盘，然后才会交换出优先级较高的内存，从而帮助保持用户可见程序的响应性。我们可以将这一原则应用于隐藏标签页。当一个渲染器进程没有顶级标签页时，我们可以释放该进程的“工作集”大小，作为提示系统首先将这些内存交换到磁盘。由于我们发现减少工作集大小也会降低用户在两个标签页之间切换时的性能，我们会逐步释放这些内存。这意味着如果用户切换回最近使用的标签页，该标签页的内存比最近不使用的标签页更可能被分页进内存。拥有足够内存运行所有程序的用户不会注意到这个过程：Windows只有在需要时才会真正回收这些数据，因此在内存充足时不会有性能影响。

这有助于我们在低内存情况下获得更优的内存占用。与很少使用的后台标签页相关的内存可以完全交换出去，而前台标签页的数据可以完全加载到内存中。相比之下，单进程浏览器的所有标签页数据都会随机分布在其内存中，不可能如此干净地分离已用和未用的数据，浪费了内存和性能。

其他进程类型

Chromium还将许多其他组件分离到独立进程中，有时是以特定平台的方式。例如，它现在有一个独立的GPU进程、网络服务和存储服务。沙箱化的实用进程也可用于小型或有风险的任务，作为满足安全性“双重规则”Rule of Two的一种方式。