

38 | 浏览器原理（二）：浏览器进程通信与网络渲染详解

2022-10-28 LMOS 来自北京



天下无鱼

<https://shikey.com/>

《计算机基础实战课》

[课程介绍 >](#)



讲述：陈晨

时长 15:00 大小 13.70M



你好，我是 LMOS。

通过前面的学习，你应该对浏览器内的进程和线程已经有了一个大概的印象，也知道了为了避免一些问题，现代浏览器采用了多进程架构。

这节课，我们首先要说的是 Chrome 中的进程通信。这么多的进程，它们之间是如何进行 IPC 通信的呢？要知道，如果 IPC 通信设计得不合理，就会引发非常多的问题。

Chrome 如何进行进程间的通信

[上节课](#)我们提了一下 Chrome 进程架构，Chrome 有很多类型的进程。这些进程之间需要进行数据交换，其中有一个浏览器主进程，每个页面会使用一个渲染进程，每个插件会使用一个插件进程。除此之外，还有网络进程和 GPU 进程等功能性进程。

进程之间需要进程通信，渲染进程和插件进程需要同网络和 GPU 等进程通信，借助操作系统的功能来完成部分功能。其次，同一类进程（如多个渲染进程）之间不可以直接通信，需要依赖主进程进行调度中转。

进程与进程之间的通信，也离不开操作系统的支持。在前面讲 IPC 的时候，我们了解过多种实现方式。这里我们来看看 Chrome 的源码，Chrome 中 IPC 的具体实现是通过 `IPC::Channel` 这个类实现的，具体在 `ipc/ipc_channel.cc` 这个文件中封装了实现的细节。

但是在查阅代码的过程中，我发现 Chrome 已经不推荐使用 `IPC::Channel` 机制进行通信了，Chrome 实现了一种新的 IPC 机制——Mojo。

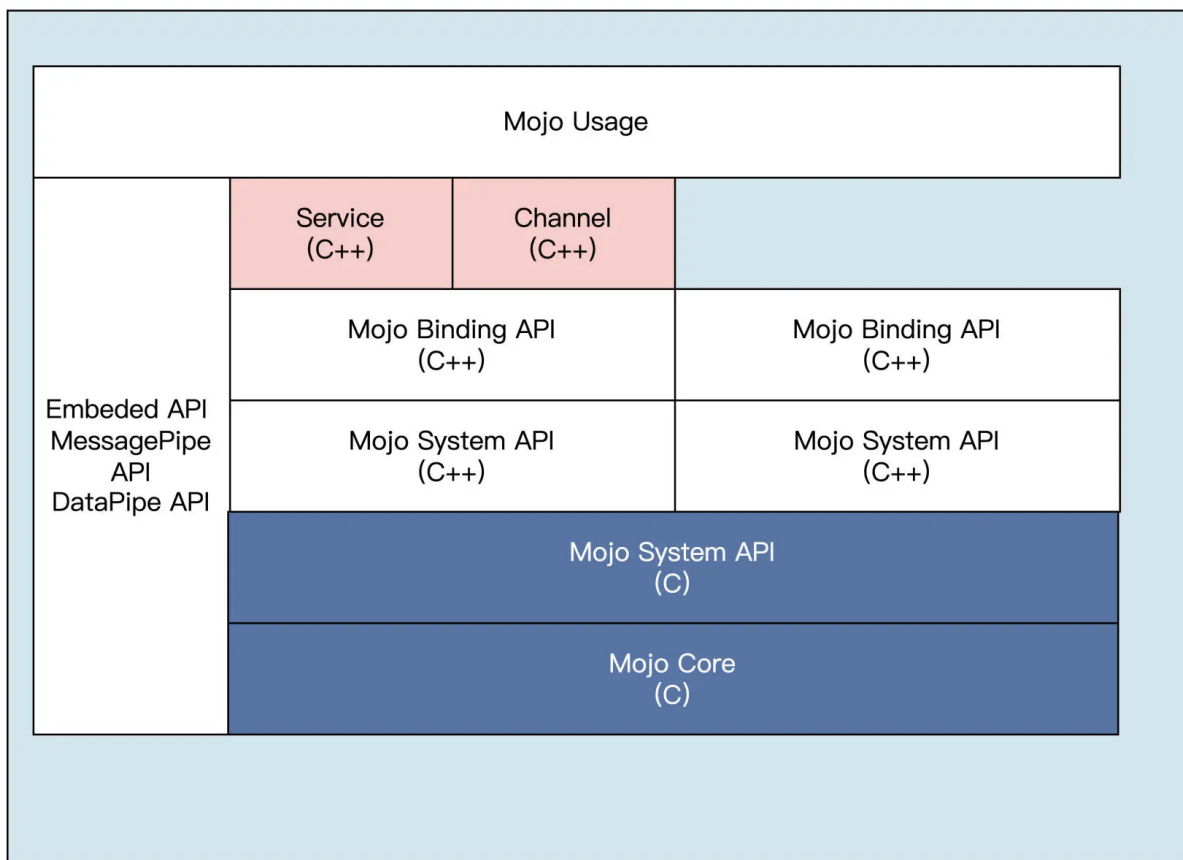
目前 `IPC::Channel` 底层也是基于 Mojo 来实现的，但是上层接口和旧的 Chrome IPC 保持兼容，`IPC::Channel` 这种方式即将被淘汰，所以这里我们先重点关注 Mojo，后面我们再简单了解一下 Chrome IPC 接口。

Mojo

Mojo 是一个跨平台 IPC 框架，它源于 Chromium 项目，主要用于进程间的通信，ChromeOS 用的也是 Mojo 框架。

Mojo 官方文档给出的定义是这样的：

“Mojo 是运行时库的集合，这些运行时库提供了与平台无关的通用 IPC 原语抽象、消息 IDL 格式以及具有用于多重目标语言的代码生成功能的绑定库，以方便在任意跨进程、进程内边界传递消息。”



在 Chromium 中，有两个基础模块使用 Mojo，分别是 Services 和 IPC::Channel。

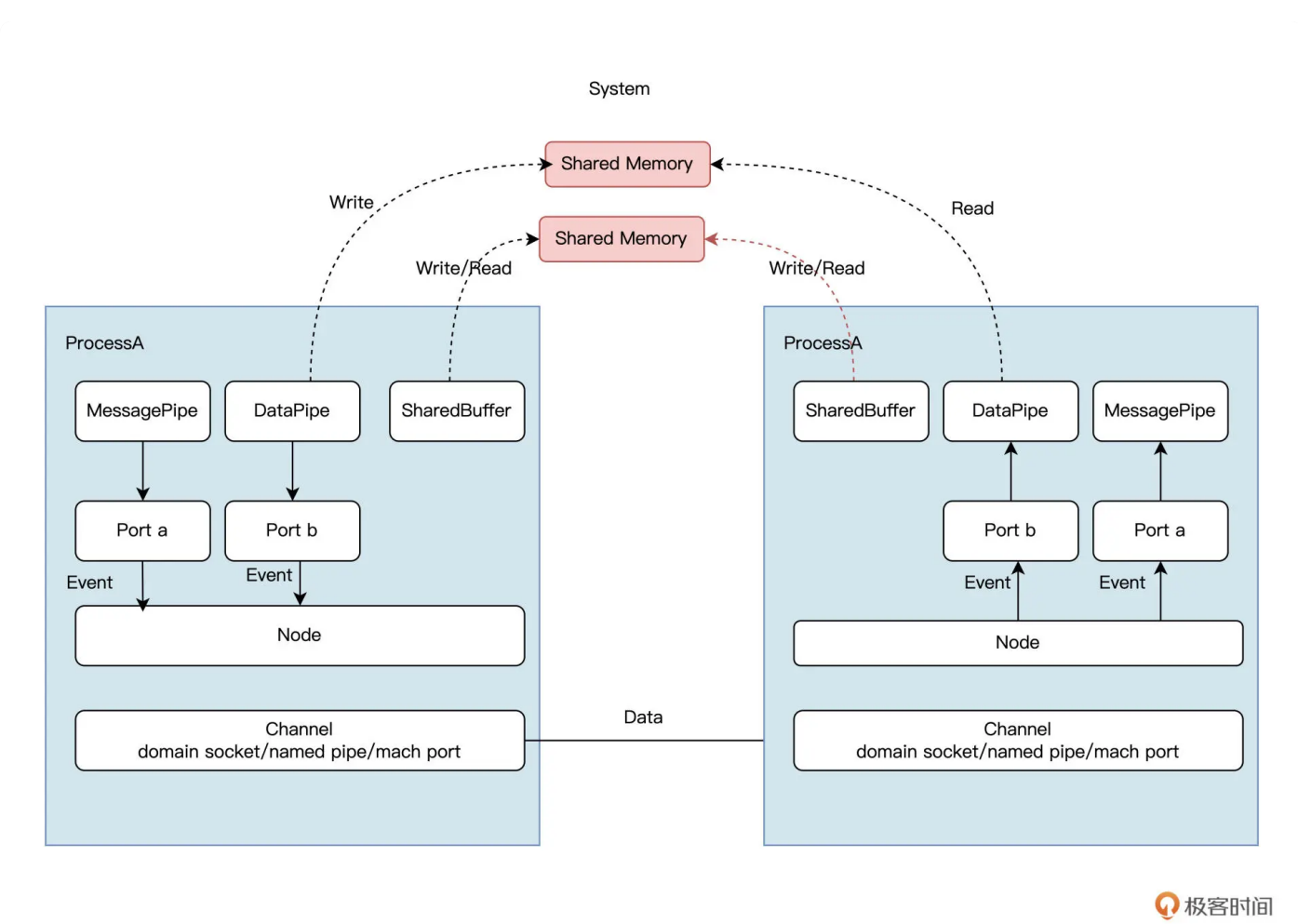
Services 是一种更高层次的 IPC 机制，底层通过 Mojo 来实现。Chromium 大量使用这种 IPC 机制来包装各种功能服务，用来取代 IPC::Channel，比如 device 服务，performance 服务，audio 服务，viz 服务等。

Mojo 支持在**多个**进程之间互相通信，这一点和其他的 IPC 有很大的不同，其他 IPC 大多只支持 2 个进程之间进行通信。

这些由 Mojo 组成的、可以互相通信的进程就形成了一个网络。在这个网络内，任意两个进程都可以进行通信，并且每个进程只能处于一个 Mojo 网络中，每一个进程内部有且只有一个 Node，每一个 Node 可以提供多个 Port，每个 Port 对应一种服务，这点类似 TCP/IP 中的 IP 地址和端口的关系。一个 Node:port 对可以唯一确定一个服务。

Node 和 Node 之间通过 Channel 来实现通信，在不同平台上 Channel 有不同的实现方式：在 Linux 上是 Domain Socket；在 Windows 上是 Named Pipe；在 macOS 平台上是 Mach Port。

在 Port 的上一层，Mojo 封装了 3 个“应用层协议”，分别为 MessagePipe，DataPipe 和 SharedBuffer（这里你是不是感觉很像网络栈，在 TCP 上封装了 HTTP）。整体结构如下图：



我们在 Chromium 代码中使用 Mojo，是不必做 Mojo 初始化相关工作的，因为这部分 Chromium 代码已经做好了。如果我们在 Chromium 之外的工程使用 Mojo，还需要做一些初始化的工作，代码如下：

复制代码

```
1 int main(int argc, char** argv) {
2     // 初始化CommandLine, DataPipe 依赖它
3     base::CommandLine::Init(argc, argv);
4     // 初始化 mojo
5     mojo::core::Init();
6     // 创建一个线程，用于Mojo内部收发数据
7     base::Thread ipc_thread("ipc!");
8     ipc_thread.StartWithOptions(
9         base::Thread::Options(base::MessageLoop::TYPE_IO, 0));
10    // 初始化 Mojo 的IPC支持，只有初始化后进程间的Mojo通信才能有效
11    // 这个对象要保证一直存活，否则IPC通信就会断开
12    mojo::core::ScopedIPCSupport ipc_support(
```

```

13     ipc_thread.task_runner(),
14     mojo::core::ScopedIPCSupport::ShutdownPolicy::CLEAN);
15     // ...
16 }


```

MessagePipe 用于进程间的双向通信，类似 **UDP**，消息是基于数据报文的，底层使用 **Channel** 通道；**SharedBuffer** 支持双向块数据传递，底层使用系统 **Shared Memory** 实现；**DataPipe** 用于进程间单向块数据传递，类似 **TCP**，消息是基于数据流的，底层使用系统的 **Shared Memory** 实现。

一个 **MessagePipe** 中有一对 **handle**，分别是 **handle0** 和 **handle1**，**MessagePipe** 向其中一个 **handle** 写的的数据可以从另外一个 **handle** 读出来。如果把其中的一个 **handle** 发送到另外一个进程，这一对 **handle** 之间依然能够相互收发数据。

Mojo 提供了多种方法来发送 **handle** 到其他的进程，其中最简单的是使用 **Invitation**。要在多个进程间使用 **Mojo**，必须先通过 **Invitation** 将这些进程“连接”起来，这需要一个进程发送 **Invitation**，另一个进程接收 **Invitation**。

发送 **Invitation** 的方法如下：

 复制代码

```

1 // 创建一条系统级的IPC通信通道
2 // 在Linux上是 Domain Socket, Windows 是 Named Pipe, macOS是Mach Port, 该通道用于支持
3 mojo::PlatformChannel channel;
4 LOG(INFO) << "local: "
5     << channel.local_endpoint().platform_handle().GetFD().get()
6     << " remote: "
7     << channel.remote_endpoint().platform_handle().GetFD().get();
8 mojo::OutgoingInvitation invitation;
9 // 创建1个Message Pipe用来和其他进程通信
10 // 这里的 pipe 就相当于单进程中的pipe.handle0
11 // handle1 会被存储在invitation中, 随后被发送出去
12 // 可以多次调用, 以便Attach多个MessagePipe到Invitation中
13 mojo::ScopedMessagePipeHandle pipe =
14     invitation.AttachMessagePipe("my raw pipe");
15 LOG(INFO) << "pipe: " << pipe->value();
16 base::LaunchOptions options;
17 base::CommandLine command_line(
18     base::CommandLine::ForCurrentProcess()->GetProgram());
19 // 将PlatformChannel中的RemoteEndpoint的fd作为参数传递给子进程
20 // 在posix中, fd会被复制到新的随机的fd, fd号改变
21 // 在windows中, fd被复制后会直接进行传递, fd号不变
22 channel.PrepareToPassRemoteEndpoint(&options, &command_line);

```

```

23 // 启动新进程
24 base::Process child_process = base::LaunchProcess(command_line, options);
25 channel.RemoteProcessLaunchAttempted();
26 // 发送Invitation
27 mojo::OutgoingInvitation::Send(
28     std::move(invitation), child_process.Handle(),
29     channel.TakeLocalEndpoint(),
30     base::BindRepeating(
31         [](const std::string& error) { LOG(ERROR) << error; }));

```

在新进程中接收 Invitation 的方法如下：

 复制代码

```

1 // Accept an invitation.
2 mojo::IncomingInvitation invitation = mojo::IncomingInvitation::Accept(
3     mojo::PlatformChannel::RecoverPassedEndpointFromCommandLine(
4         *base::CommandLine::ForCurrentProcess()));
5 // 取出 Invitation 中的pipe
6 mojo::ScopedMessagePipeHandle pipe =
7     invitation.ExtractMessagePipe("my raw pipe");
8 LOG(INFO) << "pipe: " << pipe->value();

```

上面使用 Mojo 的方法是通过读写原始的 buffer，还是比较原始的。

Chromium 里面使用了更上层的 bindings 接口来进行 IPC 通信。它先定义了一个 mojom 的接口文件，然后生成相关的接口 cpp 代码。发送方调用 cpp 代码接口，接收方去实现 cpp 代码接口。这种用法类似 Protocol Buffers。

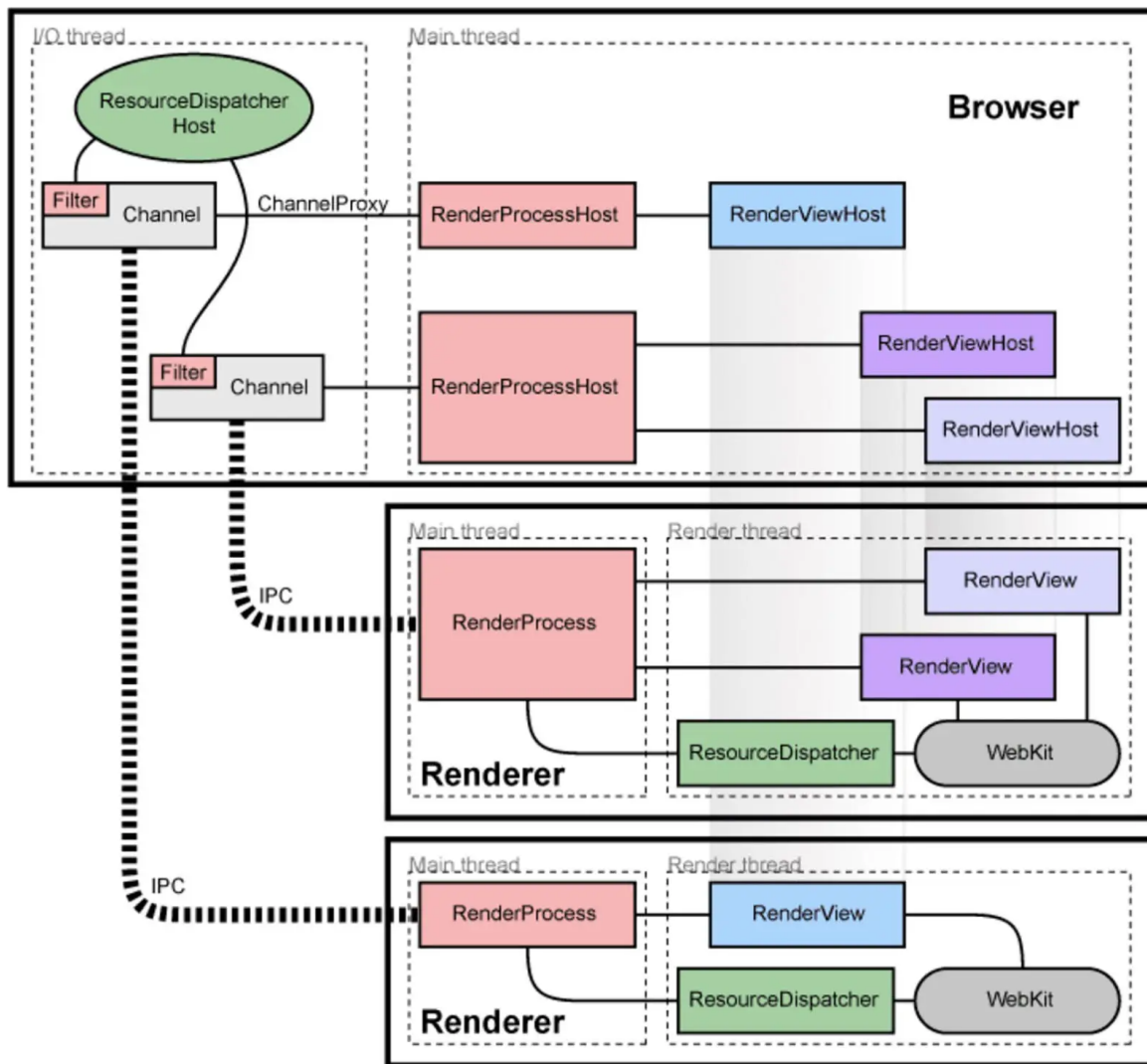
我们不需要显式地去建立进程间的 IPC 连接，因为这些 Chromium 代码已经做好了。

Chromium 的每个进程都有一个 Service Manager，它管理着多个 Service。每个 Server 又管理着多个 Mojo 接口。在 Chromium 中，我们只需要定义 Mojo 接口，然后在恰当的地方去注册接口、实现接口即可。

legacy IPC

说完 Mojo，我还想带你简单看一下 legacy IPC。虽然它已经被废弃掉，但是目前还有不少逻辑仍在用它，你可以在 [这里](#) 看到目前还在使用它的部分，都是一些非核心的消息。所以，我们还是要大致理解这种用法。

后面这张图是官方的经典图解：



图片来源 <https://www.chromium.org/developers/design-documents/multi-process-architecture/>

我们看到：每个 Render 进程都有一条 Legacy IPC 通过 Channel 和 Browser 连接，ResourceDispatcher 通过 Filter 同 Channel 进行连接。IPC 里面有几个重要的概念：

- **IPC::Channel**：一条数据传输通道，提供了数据的发送和接收接口；
- **IPC::Message**：在 Channel 中传输的数据，主要通过宏来定义新的 Message；
- **IPC::Listener**：提供接收消息的回调，创建 Channel 必须提供一个 Listener；
- **IPC::Sender**：提供发送 **IPC::Message** 的 **Send** 方法，**IPC::Channel** 就实现了 **IPC::Sender** 接口；
- **IPC::MessageFilter**：也就是 Filter，用来对消息进行过滤，类似管道的机制，它所能过滤的消息必须由其他 Filter 或者 Listener 传给它；
- **IPC::MessageRouter**：一个用来处理 **Routed Message** 的类。

Legacy IPC 的本质就是**通过 IPC::Channel 接口发送 IPC::Message**，IPC::Channel 是封装好的类，IPC::Message 需要用户自己定义。

IPC::Message 有两类，一类是路由消息“routed message”，一类是控制消息“control message”。

唯一不一样的就是 routing_id() 不同，每一个 IPC::Message 都会有一个 routing_id，控制消息的 routing_id 始终是 MSG_ROUTING_CONTROL，这是一个常量。除此之外，所有 routing_id 不是这个常量的消息，都是路由消息。

网页渲染的流程

前面我们讲了浏览器的架构，进程 / 线程模型以及浏览器内的 IPC 通信实现，有了这些铺垫，我们再来理解浏览器内部的进程模型的工作机制，就更容易了。进程通信会伴随着网络渲染的过程，所以，我推荐你从实际的渲染过程来观察，也就是搞明白浏览器是怎么借助计算机进行页面图像渲染的。

浏览器接收到用户在地址栏输入的 URL 以后，浏览器的网络进程会利用操作系统内核网络栈进行资源获取。在第一季的网络篇，我们曾经用了一节课的时间讲解 [网络数据包是在网络中如何流转的](#)。如果你想要详细了解，可以去看看。这里我们着重关注浏览器收到响应后的渲染过程。

在浏览器启动后，浏览器会通过监听系统的某个指定端口号，监听数据的变化。在浏览器收到网络数据包后，会根据返回的 Content-Type 字段决定后续的操作，如果是 HTML，那么浏览器则会进入渲染的流程。

在渲染过程中，主要工作交由渲染进程处理，我们可以简要分为几个部分：建立数据传输管道、构建 DOM 树、布局阶段、绘制以及合成渲染。下面，我们分别进行讲解。

建立数据传输管道

当网络进程接收到网络上出来的 HTML 数据包的时候，渲染进程不会等网络进程完全接受完数据，才开始渲染流程。为了提高效率，渲染进程会一边接收一边解析。所以，渲染进程在收到主进程准备渲染的消息后，会使用 Mojo 接口，通过边解析边接收数据的方式，和网络进行 IPC 通信，建立数据传输的管道，将数据提交到渲染进程。

构建 DOM 树

渲染进程收到的是 HTML 的字符串，是一种无法进行结构化操作的数据，于是我们需要将纯文本转为一种容易操作、有结构的数据 —— DOM 树。

DOM 树本质上是一个以 document 为根节点的多叉树，DOM 树是结构化、易操作的，同样浏览器也会提供接口给到开发者，浏览器通过 JS 语言来操作 DOM 树，这样就可以动态修改页面内容了。

在渲染进程的主线程内部，存在一个叫 **HTML 解析器（HTMLParser）** 的东西，想要将文本解析为 **DOM**，离不开它的帮助。**HTML 解析器**会将 **HTML** 的字节流，通过分词器转为 **Token 流**，其中维护了一个栈结构，通过不断的压栈和出栈，生成对应的节点，最终生成 DOM 结构。

在 DOM 解析的过程中当解析到 `<script>` 标签时，它会暂停 HTML 的解析，渲染进程中的 JS 引擎加载、解析和执行 JavaScript 代码完成后，才会继续解析。

在 JS 解析的过程中，JS 是可能进行 CSS 操作的，所以在执行 JS 前还需要解析引用的 CSS 文件，生成 **CSSOM** 后，才能进行 JS 的解析。**CSSOM** 是 DOM 树中每个节点的具体样式和规则对应的树形结构，在构建完 **CSSOM** 后，要先进行 JS 的解析执行，然后再进行 DOM 树的构建。

布局阶段 —— layout

这时已经构建完 DOM 树和 CSSOM 树，但是还是无法渲染，因为目前渲染引擎拿到的只是一个树形结构，并不知道具体在浏览器中渲染的具体位置。

布局就是寻找元素几何形状的过程，具体就是主线程遍历 DOM 和计算样式，并创建包含 xy 坐标和边界框大小等信息的布局树。

布局树可能类似于 DOM 树的结构，但它只包含与页面上可见内容相关的信息。比如说，布局树构建会剔除掉内容，这些内容虽然在 DOM 树上但是不会显示出来，如属性为 `display: none` 的元素；其次，布局树还会计算出布局树节点的具体坐标位置。

绘制

渲染进程拿到布局树已经有具体节点的具体位置，但是还缺少一些东西，就是层级。我们知道，页面是类似 **PS** 的图层，是有图层上下文顺序的，而且还有一些 **3D** 的属性，浏览器内核还需要处理这些专图层，并生成一棵对应的图层树（**LayerTree**）。

有了图层的关系，就可以开始准备绘制了，渲染进程会拆分出多个小的绘制指令，然后组装成一个有序的待绘制列表。

合成渲染

从硬件层面看，渲染操作是由显卡进行的，于是浏览器将具体的绘制动作，转化成待绘制指令列表。

浏览器渲染进程中的合成线程，会将数据传输到栅格化线程池，从而实现图块的栅格化，最终把生成图块的指令发送给 **GPU**。然后，在 **GPU** 中执行生成图块的位图，并保存在 **GPU** 的内存中。

此时显示器会根据显示器的刷新率，定期从显卡的内存中读取数据。这样，图像就可以正常显示，被我们看到了。

浏览器渲染的流程比较复杂，其中的细节也比较多，如果要详细分析，还可以拆成一篇超长篇幅，所以这里我们只是了解简单过程。你如果想要了解完整过程，可以阅读拓展材料中的 **Chrome** 开发者的官方博客。

Chromium 的文件结构解析

前面课程里，我们通过一些概念和例子简单了解了 **WebKit** 和 **Chromium** 的架构，不过这两者是非常庞大的项目，代码量也是非常的巨大，除去其中依赖的第三方库，这两个项目的代码量都是百万级别的，如果直接阅读的话是非常困难的。

但是良好的代码组织结构，很好地帮助了开发者和学习者们。下面我大致介绍一下它们的目录结构及其用处，方便你快速地理解整个项目。

因为里面的一二级目录非常多和深，所以我们把焦点放在核心的部分即可。我们可以通过 [🔗 GitHub](#) 将 **Chromium** 的源码下载下来阅读，但是源码非常大，如果你不想下载，可以通过 [🔗 这个链接](#) 访问在线版本。

```

1 |— android_webview - 安卓平台webview的 `src/content` 目录所需要的接口
2 |— apps - chrome打包 apps 的代码
3 |— base - 基础工具库，所有的子工程公用
4 |— build - 公用的编译配置
5 |— build_overrides //
6 |— cc - 合成器
7 |— chrome - chrome 相关的稳定版本实现比如渲染进程中的某些API 的回调函数和某些功能实现
8 |   |— app - 程序入口
9 |   |— browser - 主进程
10 |   |— renderer - 渲染进程
11 |   ...
12 |— chromecast
13 |— chromeos - chromeos 相关
14 |— components - content层调用的一些组件模块；
15 |— content - 多进程模型和沙盒实现的代码
16 |   |— app - contentapi 的部分 app 接口
17 |   |— browser - 主进程的实现
18 |   |— common - 基础公共库
19 |   |— gpu - gpu 进程实现
20 |   |— ppapi_plugin - plugin 进程实现
21 |   |— public - contentapi 接口
22 |   |— renderer - 渲染进程实现
23 |   ...
24 |— courgette
25 |— crypto - 加密相关
26 |— device - 硬件设备的api抽象层
27 |— docs - 文档
28 |— gpu - gpu 硬件加速的代码
29 |— headless - 无头模式，给 puppeteer 使用
30 |— ipc - ipc 通信的实现，包括 mojo 调用和 ChromeIPC
31 |— media - 多媒体相关的模块
32 |— mojo - mojo 底层实现
33 |— native_client_sdk
34 |— net - 网络栈相关
35 |— pdf - pdf 相关
36 |— ppapi - ppapi 代码
37 |— printing - 打印相关
38 |— sandbox - 沙箱项目，安全用防止利用漏洞攻击操作系统和硬件
39 |— services
40 |— skia - Android 图形库，直接从 Android 代码树中复制过来的
41 |— sql - 本地数据库实现
42 |— storage - 本地存储实现
43 |— third_party - 三方库
44 |   |— Webkit
45 |   ...
46 |— tools
47 |— ui - 渲染布局的基础框架
48 |— url - url 解析和序列化
49 |— v8 - V8 引擎

```

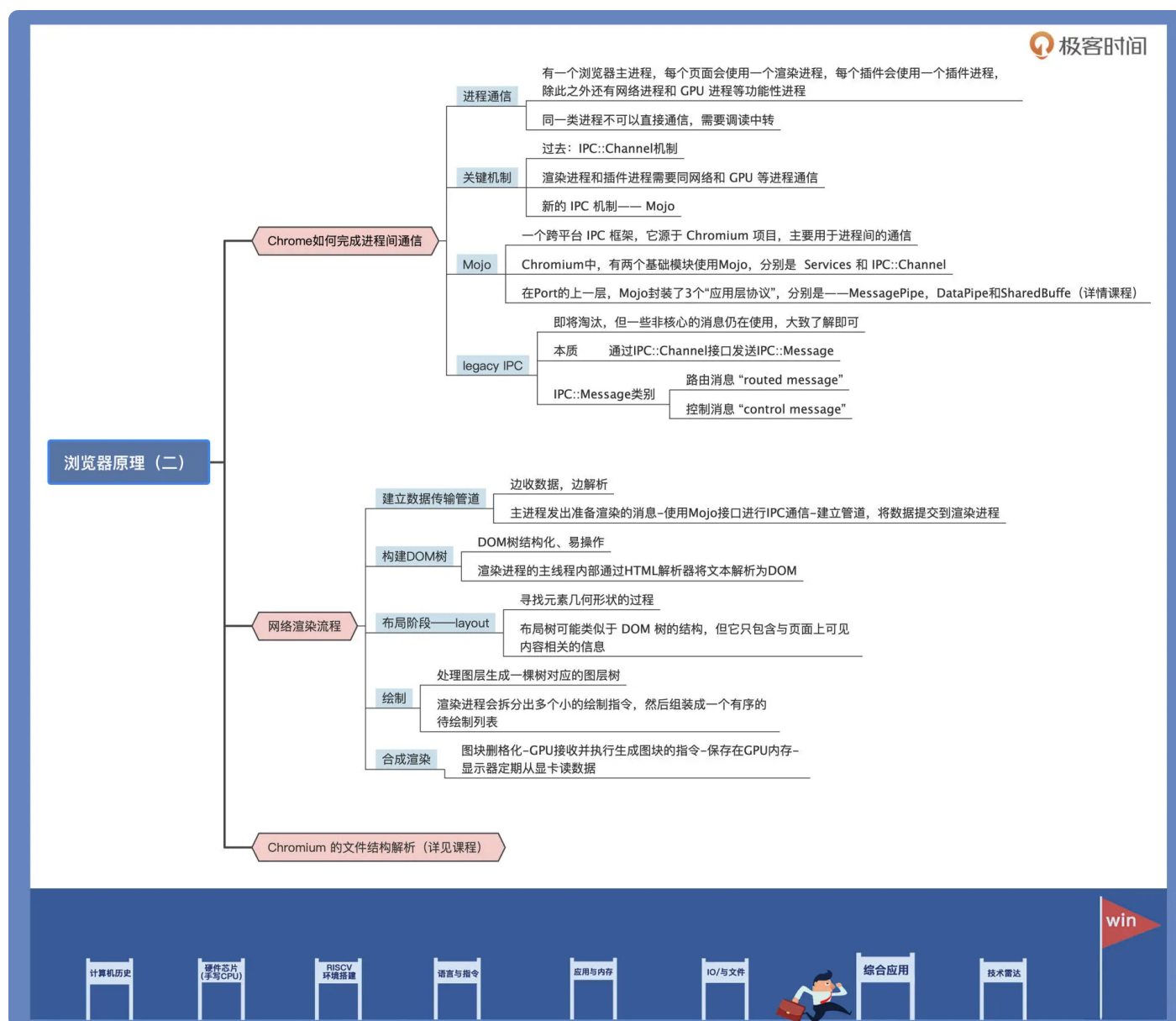
重点回顾

今天，我们学习了 Chrome 下的多进程之间的协作方式。

老版本的 Chrome 使用 Legacy IPC 进行 IPC 通信，它的本质就是通过 IPC::Channel 接口发送 IPC::Message。而新版本的 Chrome 使用了 Mojo 进行 IPC 通信，Mojo 是源于 Chrome 的 IPC 跨平台框架。Chrome 在不同的操作系统下的 IPC 实现方式有所不同，在 Linux 上是 Domain Socket，Windows 是 Named Pipe，macOS 是 Mach Port。

之后，我们通过网页渲染的例子深入了解了，不同进程之间如何协作来进行渲染。最后我给你列举了 Chrome 项目的基本目录结构，如果你对其感兴趣，可以自行下载源码，深入探索。

这节课的导图如下，供你参考：



扩展阅读

浏览器是一个极为庞大的项目，仅仅通过两节课的内容，想要完全了解浏览器的特性是不太可能的。希望这两节课能抛砖引玉，更多的内容需要你自己去进行探索。

这里我为你整理了一些参考资料，如果你能够认真阅读，相信会获得意想不到的收获。


1. 首先是 Chromium 官方的 [🔗 设计文档](#)，包含了 Chromium and Chromium OS 的设计思维以及对应源码。
2. 其次是 Chrome 开发者的 [🔗 官方博客](#)，里面的系列文章详细介绍了 Chrome 渲染页面的工作流程。
3. 还有 Mojo 的 [🔗 官方文档](#)，从这里你可以了解 Mojo 的简单使用以及实现。
4. 最后就是 [🔗 《🔗 WebKit 技术内幕 🔗》](#)，这本书详细介绍了 WebKit 的渲染引擎和 JavaScript 引擎的工作原理

思考题

为什么 JS 代码会阻塞页面渲染？从浏览器设计的角度看，浏览器可以做哪些操作来进行优化？在开发前端应用过程中又可以做哪些优化呢？

欢迎你在留言区和我交流讨论。如果这节课对你有启发，别忘了分享给身边更多朋友。

分享给需要的人，Ta 购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3)

写留言



LockedX

2022-10-28 来自湖北

浏览器无法知道JS代码会对DOM和CSSDOM操作什么，所以要等JS执行完再渲染页面。尽量把<script/>放在HTML最后执行，对用户来说可能体验好一点。

作者回复: 好的



1



peter

2022-10-28 来自湖北

请教老师几个问题：

Q1: 一个网络还是多个网络？

文中“这些由 Mojo 组成的、可以互相通信的进程就形成了一个网络。在这个网络内，任意两个进程都可以进行通信，并且每个进程只能处于一个 Mojo 网络中”，这句话的前面部分是说是一个网络。但后面说“每个进程处于一个网络”，给人感觉就是多个进程就会处于多个网络中。

Q2: 多个“mojo system + mojo binding”是什么意思？

第一个图中，画了两个mojo system API + mojo binding API”，为什么画多个，有什么含义？

Q3: mach port是什么通信方式？

Q4: chrome中进程间通信没有采用消息队列，是吗？

作者回复: 一个网络



苏流郁宓

2022-10-28 来自湖北

由于cpu（单核），始终实时只能指向一个栈的入口（可以切换不同的栈）

对于浏览器来说，由于网络重要数据存放在内存条上。需要通过链表的形式来引导cpu在不同栈中切换！也就是说链表存放的内存所在位置会被反复的引用（cpu访问）

那么浏览器可以优化的部分，就是可以制定一个统一的接口标准，方便不同网页的插件进行应用。从而更好的通过数据分流，既流畅也用得舒服的 啊

作者回复: 嗯嗯

