



杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

Mondrian

Wayland 协议下的平铺式桌面显示系统

小组成员： 林灿， 吴悦怡， 陈序

2025 年 6 月

目录

1 项目概述	3
1.1 项目简介	3
1.2 项目开发进度	4
1.3 项目核心功能	4
2 赛题描述	5
2.1 赛题要求	5
2.2 Linux 图形栈	5
2.3 Wayland 协议与 X11 协议	6
2.4 平铺式布局管理	10
3 项目设计与实现	12
3.1 技术选型	12
3.2 Smithay	12
3.3 Rust	13
3.4 项目最小实现	13
3.5 输入设备事件监听	21
3.6 DRM/KMS 裸机直连	27
3.7 动画效果实现	46
3.8 拓展协议实现	51
4 性能测试与分析(仍在优化, 决赛期间完成)	54
4.1 Rust Tracy 跟踪分析	54
4.2 帧率性能测试	55
4.3 GPU 压力测试	55
5 项目总结	56

1 项目概述

1.1 项目简介

本项目基于 Smithay 使用 Rust 开发了一个使用 Wayland 协议的 平铺式桌面显示系统。项目能够在裸机终端中自行初始化 DRM/KMS 图形管线，并通过 GBM 和 EGL 建立 GPU 渲染上下文，使用 OpenGL ES 进行硬件加速合成显示。启动后该 Compositor 接管系统图形输出，并成为客户端程序（如终端模拟器、浏览器）的 Wayland 显示服务。

“Beauty is the promise of happiness.” — Stendhal

本项目秉持“优雅即力量”的设计哲学，力求在系统结构与用户体验之间取得和谐平衡。无论是内部代码逻辑还是外部交互呈现，都追求简洁、清晰而富有韵律的表达。

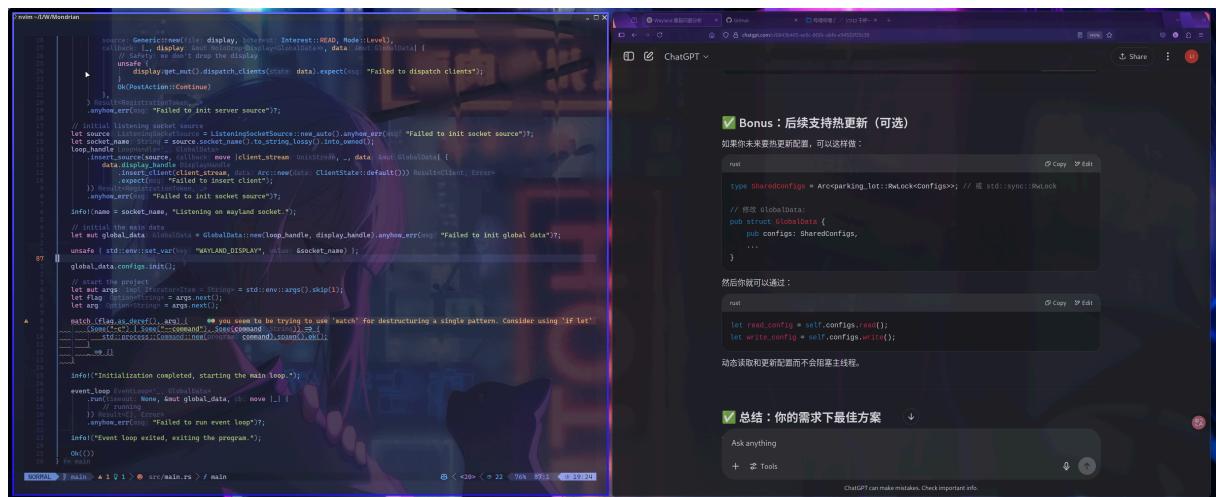


图 1 项目运行效果演示图

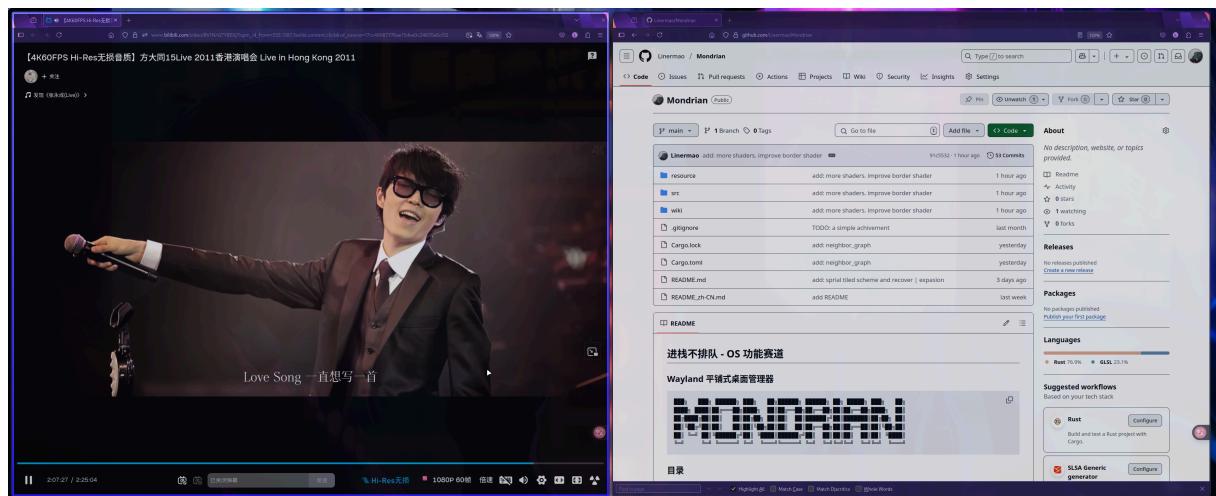


图 2 项目运行效果演示图

1.2 项目开发进度

项目自 3 月启动，初期集中精力对现有 Wayland 合成器框架（如 wlroots、Smithay 等）进行了调研与试验，结合项目可控性与扩展性要求，最终选择 **Rust + Smithay** 作为核心技术栈。

截至初赛阶段(6.31)，本项目已完成主要基础功能的开发，现已能够在主流 Linux 发行版上部署运行，具备日常使用的基本可用性。

在此基础上，为实现更高的性能、更强的可定制性以及更全面的兼容性，后续开发工作将聚焦于以下方向：

XWayland 支持	以兼容传统 X11 应用程序
多显示器管理	支持动态热插拔与独立配置
多输入设备支持	提升对触控板、手写板等外设的适配能力
个性化自定义功能	允许用户通过配置文件自定义风格与快捷操作
更优秀的平铺布局方案	允许自由切换与自定义排版

1.3 项目核心功能

代码体量 累计新增 1.5w 行代码，配套文档逾 2w 字，涵盖架构设计、接口协议与开发细节。

全栈实现 实现双后端架构：winit 支持桌面环境，tty 支持裸机直启，原生 DRM/KMS 渲染流程：直接控制 GPU 输出，无需依赖 X11 Server。

数据结构与算法 引入改造后的容器式二叉树布局方案，实现灵活的平铺与窗口变换；结合 SlotMap 实现节点的常数时间复杂度插入、删除与查找，极大提升动态性能。

个性化与可编程定制能力 使用统一配置文件控制主题样式、键位绑定、布局策略等，用户可通过 shell 脚本绑定快捷键，实现更复杂的自动化操作。

动画与渲染 自定义过渡动画与渲染逻辑，配合手写 GLSL shader，实现流畅、响应式的交互体验，视觉层次统一且精致。

2 赛题描述

2.1 赛题要求

2.1.1 赛题名称

proj340 - 实现一个简单的平铺式管理的 Wayland 合成器

2.1.2 赛题描述

实现一个 Wayland 合成器项目，使用平铺式的窗口布局策略来管理窗口的位置和大小，仅要求支持 XDG Shell 协议，对 XWayland 程序不做要求。

可参照任意已有的平铺式窗口管理器的行为，比如 i3、sway、hyprland 等

可使用基本的开发库作为底层实现，如 wlroots

2.1.3 预期目标

程序运行后自动展示一个终端窗口，在终端窗口中输入命令可启动其它程序，新打开的 Wayland 窗口与已有的窗口无遮挡

2.2 Linux 图形栈

人通过感知和操作与电脑交流，电脑通过硬件设备获取指令、给出反馈，再由系统软件进行翻译和执行。

整个系统围绕着“输入 → 处理 → 输出”的闭环进行工作，人类与计算机通过输入设备产生指令，经内核调度、用户态处理，再输出至显示设备，从而完成一次完整的交互过程。

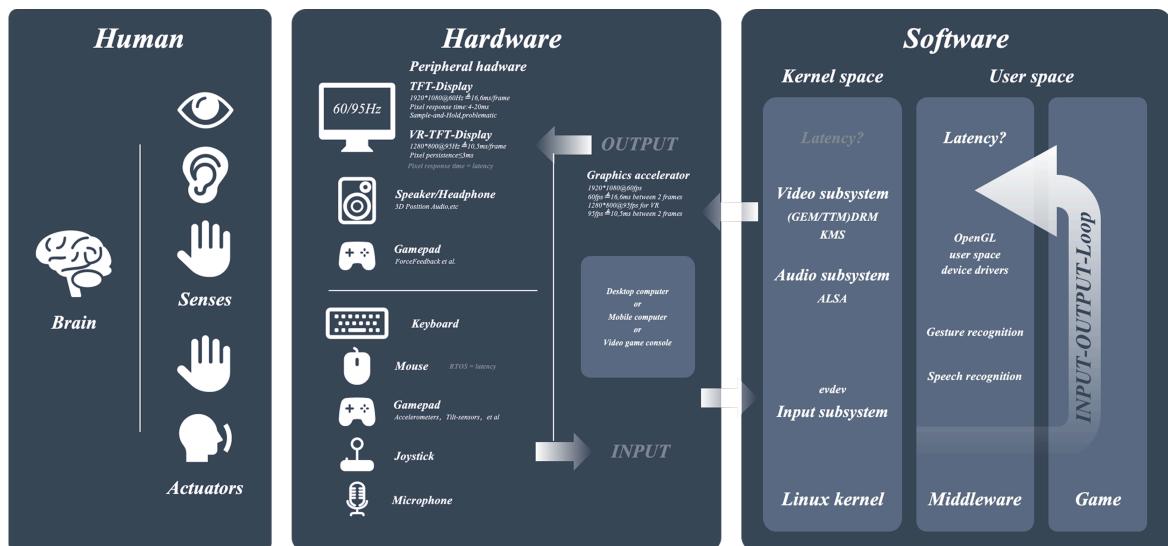


图 3 linux 图形人机交互

2.3 Wayland 协议与 X11 协议

在 Linux 操作系统中，图形显示系统由多个层级组成，从底层的内核显卡驱动到用户态的图形协议，再到最终的 GUI 应用。整个图形栈主要包括以下几部分：

内核层 (*Kernel Space*)：

DRM (Direct Rendering Manager) 管理 GPU 资源与帧缓冲控制。

KMS (Kernel Mode Setting) 用于设置显示模式，如分辨率、刷新率等。

GBM (Generic Buffer Management) 用于创建与管理图形缓冲区。

图形服务器 (*Display Server*)：

Xorg X11 协议的标准实现。

Wayland Compositor Wayland 协议的实现方，集成合成器、窗口管理器、输入系统。

窗口管理器 管理窗口的移动，缩放，堆叠关系等所有的窗口行为。

中间协议层

Mesa 用户态 OpenGL/Vulkan 实现，提供图形驱动接口。

EGL 抽象图形上下文与窗口系统之间的接口，衔接 OpenGL 与窗口系统。

用户层协议 (User Space Protocol) 通信协议。

用户层

- GUI 应用程序，使用 Qt、GTK 等图形库开发。

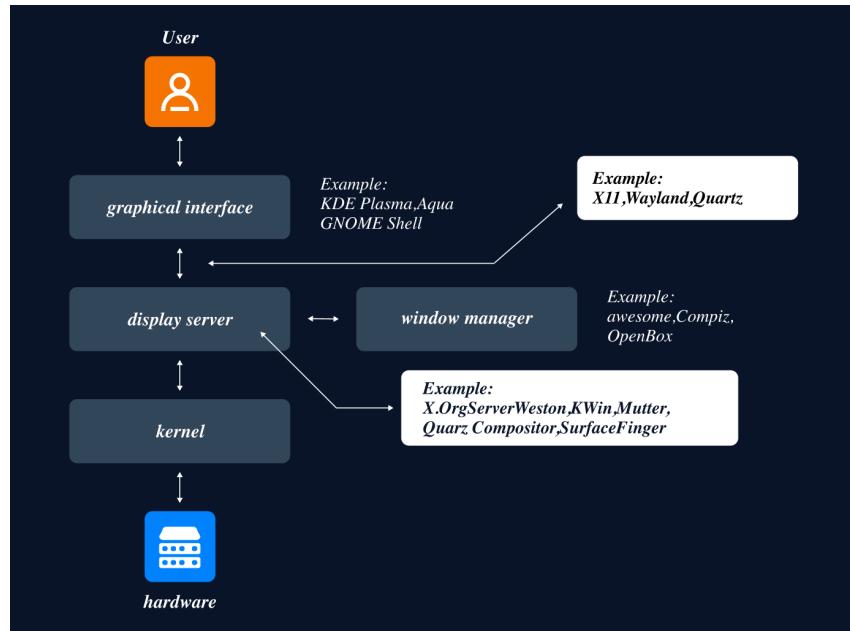


图 4 linux 图形栈

2.3.1 Wayland 协议

Wayland 是设计用于替代 X11 的现代图形协议，由 wayland.freedesktop.org 开发，强调简洁、安全、高性能。其基本架构如下：

Compositor so Display Server:

- 直接管理窗口、图像合成与缓冲交换。
- 处理输入事件，并直接分发到正确的客户端。
- 实现窗口管理逻辑（如平铺、浮动等）。

Client:

- 负责自行渲染窗口内容（通过 GPU 渲染或 CPU 绘图）。
- 使用 wl_surface 等原语将渲染结果提交给 Compositor。
- 与 Compositor 通过共享内存或 DMA Buffer 实现高效图像交换。

Protocols:

- 基于 Unix Domain Socket 通信，使用 wl_display 进行连接。
- 使用对象-事件模型（Object/Interface），类似面向对象远程调用。
- 无需往返确认，大部分请求为异步执行，提高响应效率。

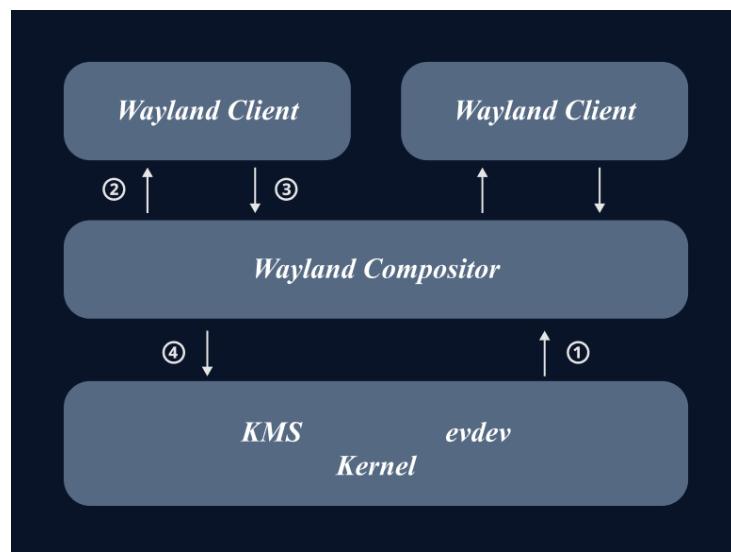


图 5 wayland 协议示意图

2.3.2 X11 协议

X11 是诞生于 1984 年的图形窗口系统，其核心是 client - server 架构：

X Server 运行在用户机器上，控制显示硬件，处理输入事件。

X Clients 运行应用程序，向 X Server 请求窗口资源，并响应事件回调。

X11 协议支持网络透明性，即 X Client 和 X Server 可以运行在不同主机上。但其通信模型较为复杂：

- 每个窗口请求都需往返服务器确认（Round Trip），带来额外延迟。
- 图形渲染与窗口管理被分离为多个组件（如 WM、Compositor、Toolkit），难以协调。
- 输入事件先由 X Server 捕获，后由 Window Manager 转发，路径冗长且易出现冲突。

尽管历经多年优化，X11 的架构问题已难以适应现代图形性能与安全性的需求。

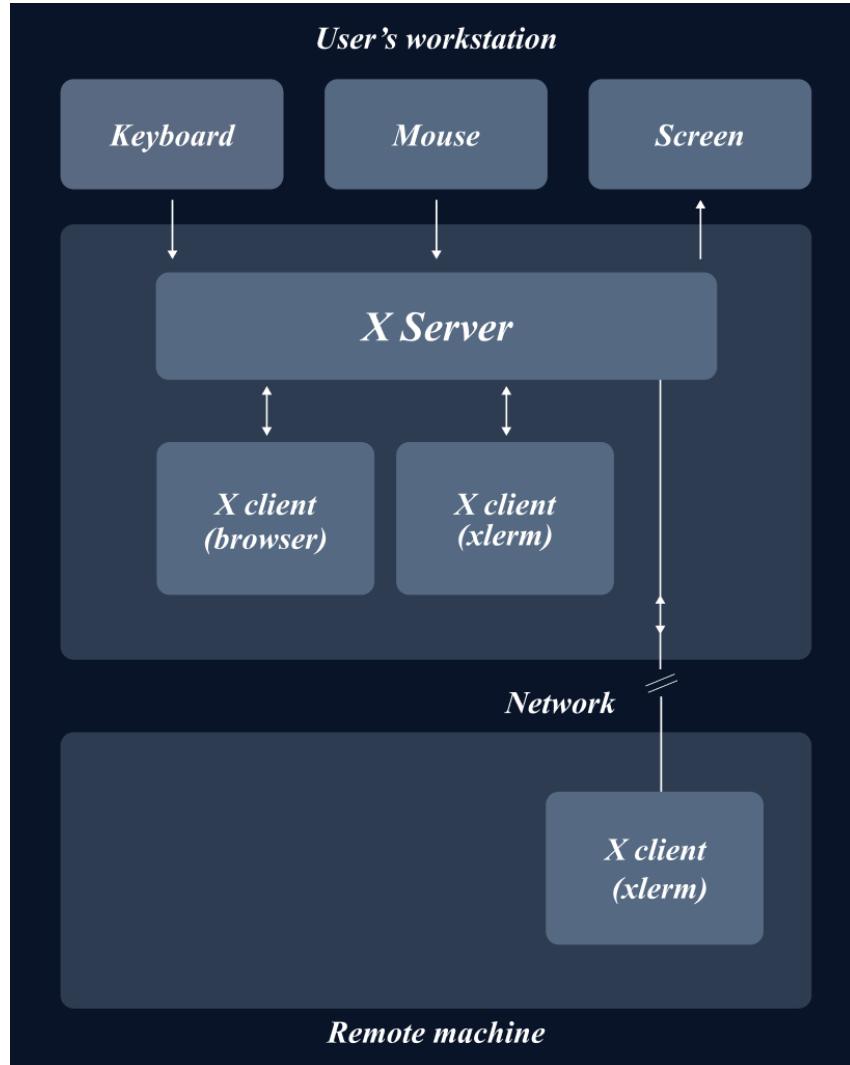


图 6 x11 协议示意图

2.3.3 Wayland 协议的优势

相比 X11，Wayland 协议具有以下核心优势：

简洁的架构设计

Wayland 取消了中间代理（如 Xlib/XCB），让客户端直接负责渲染，Compositor 仅做图像合成与事件路由。这种 **单一控制点设计** 更加清晰易控。

异步通信模型

大多数请求为异步非阻塞，大幅降低绘制窗口所需的 round-trip，提升性能表现，尤其在高帧率与多窗口场景下优势明显。

安全性与隔离性更好

Compositor 全面控制窗口焦点、输入与输出，不再暴露全局窗口信息。各客户端窗口互不可见（无法监听或操作其他窗口）。支持权限隔离（如输入抓取限制、屏幕截图权限控制等）。

动态扩展能力强

Wayland 协议采用模块化设计，核心协议只定义基础对象（如 wl_surface, wl_output），其他功能由扩展协议（Protocol Extensions）提供，例如：

xdg-shell	提供桌面窗口接口（如 toplevel/popup）
wlr-layer-shell	支持桌面元素（如面板、壁纸）
xdg-output	扩展输出信息
pointer-gestures	添加手势支持

原生合成支持

每个窗口的图像由 Client 渲染后交给 Compositor 直接合成，因此减少了冗余图层绘制流程，更容易实现视觉效果（圆角、阴影、动画），支持真正的无撕裂与高刷新率渲染。

2.4 平铺式布局管理

传统的桌面环境普遍采用堆叠式（Stacking）窗口管理模型，其核心思想是通过层叠多个可自由移动和缩放的窗口，来完成用户的窗口组织。窗口的可见性与交互依赖于 Z 轴层级与遮挡关系。在窗口数量增多、频繁切换任务时，这种模型容易导致空间浪费、管理混乱、用户认知负担增大，不利于快速切换和管理窗口。

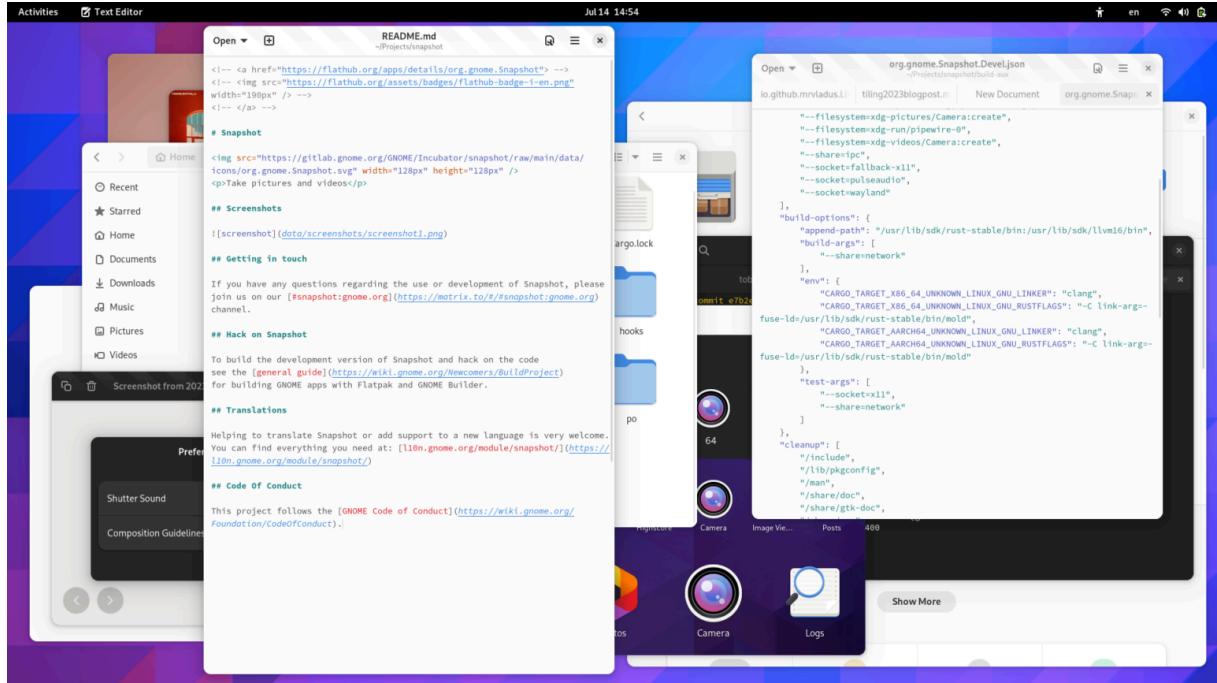


图 7 堆叠式布局示意图（来自 GNOME）

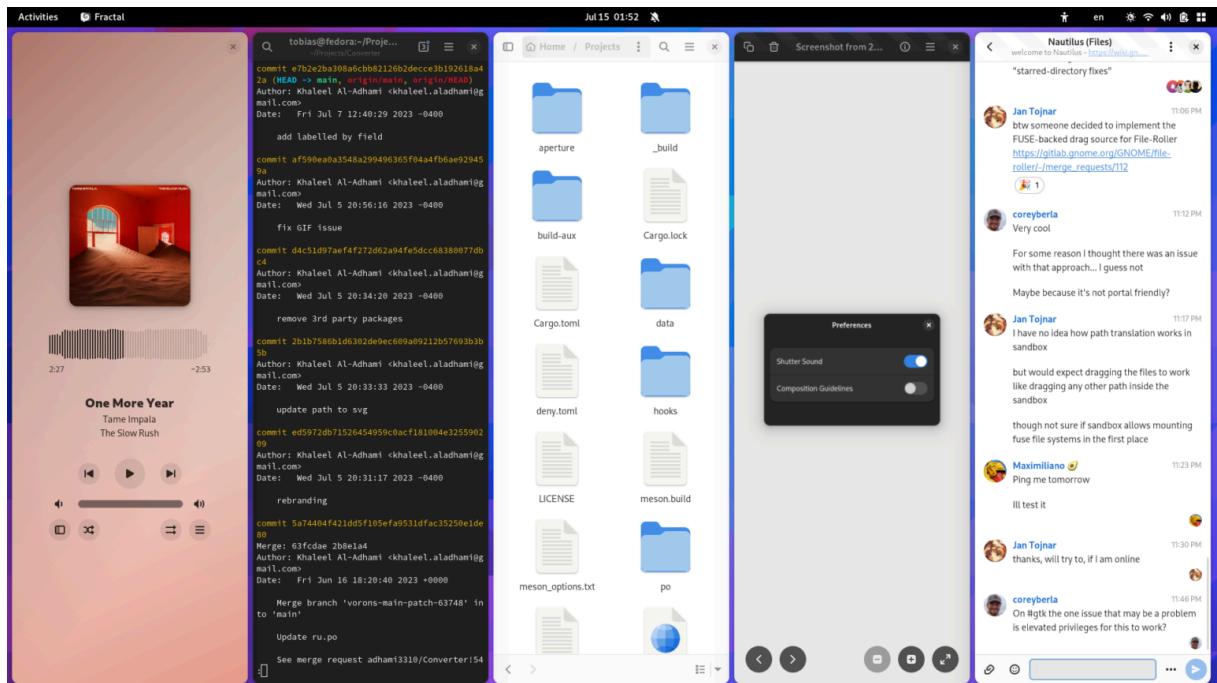


图 8 平铺式布局示意图（来自 GNOME）

平铺式（Tiling）窗口管理是一种高度结构化的布局方式，屏幕被划分为若干区域，每个窗口占据一个不重叠的矩形区域，并根据特定的布局算法自动排列。其核心原则是：

所有活动窗口在空间上无重叠，完全平铺填充屏幕空间。

窗口自动布局 每当有新窗口创建，它不会以浮动方式出现，而是根据当前焦点窗口的位置与所选布局算法（如垂直分裂、水平分裂、主从结构等）自动插入到屏幕分区中。

无重叠区域，最大化利用空间 所有窗口矩形区域互不重叠，窗口大小由布局决定而非用户拖拽（当然存在平铺式与堆叠式一同使用的情况，允许鼠标进行一定的操作），避免空间浪费。

键盘优先交互 平铺管理器强调键盘操控，通过快捷键进行窗口聚焦、移动、交换、调整布局比例等操作，效率远高于传统的鼠标驱动方式。

一致性与可预测性 所有布局变化均可通过布局算法精确复现，不依赖“拖拽”或“随机叠放”这种不可重现的行为，便于自动化与脚本控制。

3 项目设计与实现

3.1 技术选型

Mondrian 的核心目标是实现一个面向未来的、结构可控的平铺式桌面环境，因此我们选择了 Rust 作为主要开发语言，并基于 Smithay 框架进行构建。这一组合在性能、可靠性、安全性和协议支持方面表现出极高的适配性。

3.2 Smithay

Smithay 是一个专为构建 Wayland 合成器而设计的 Rust 框架，提供了协议实现、后端抽象、渲染集成等基础模块。它不是一个完整的 window manager，而是一个合成器构建工具箱。

The screenshot shows the GitHub repository page for Smithay. At the top, there are navigation links for 'README' and 'MIT license'. Below the header, the repository name 'Smithay' is displayed in a large, bold, black font. To the right of the name is a stylized logo consisting of a yellow/orange flame-like shape above a dark grey arch. Below the name, there are several status indicators: 'crates.io v0.6.0' (orange button), 'docs passing' (green button), 'Continuous Integration failing' (red button), '[m] #smithay:matrix.org' (blue button), and 'IRC #Smithay' (blue button). A horizontal line separates this from the main content. The main content area starts with the text 'A smithy for rusty wayland compositors'. Below this, a section titled 'Goals' is shown. The text under 'Goals' states: 'Smithay aims to provide building blocks to create wayland compositors in Rust. While not being a full-blown compositor, it'll provide objects and interfaces implementing common functionalities that pretty much any compositor will need, in a generic fashion.' Further down, it says: 'It supports the [core Wayland protocols](#), the official [protocol extensions](#), and [some external extensions](#), such as those made by and for [wlroots](#) and [KDE](#)'. Finally, there is a bulleted list of advantages:

- **Documented:** Smithay strives to maintain a clear and detailed documentation of its API and its functionalities. Compiled documentations are available on [docs.rs](#) for released versions, and [here](#) for the master branch.
- **Safety:** Smithay will target to be safe to use, because Rust.
- **Modularity:** Smithay is not a framework, and will not be constraining. If there is a part you don't want to use, you should not be forced to use it.
- **High-level:** You should be able to not have to worry about gory low-level stuff (but Smithay won't stop you if you really want to dive into it).

图 9 smithay github 主页截图

其优势主要体现在以下几个方面：

模块化设计 Smithay 拆分为多个可选模块，如 wayland-backend, wayland-protocols, input, output 等。

Wayland 协议支持广泛 支持核心协议如 wl_compositor, wl_seat, xdg-shell, 并集成 xdg-output, layer-shell, wlr-protocols 等常见扩展。可以在合成器层自由定制协议行为，例如限制窗口行为、插入布局钩子等。

后端抽象能力 支持多个图形后端 (EGL, GLES2, WGPU)、输入后端 (Winit、libinput) 与输出设备管理 (DRM/KMS、virtual output)。允许在不同平台 (如嵌入式、TTY、X11) 运行，底层支持度高。

灵活可插拔架构 不像 Weston 或 wlroots 那样强绑定窗口管理逻辑，Smithay 允许合成器设计者自行定义事件循环、窗口模型与渲染策略，非常适合实现平铺式或动态布局系统。

社区活跃、长期演进 Smithay 拥有稳定的维护团队，与 Mesa、wlroots 社区保持良好协作，能持续跟进最新的 Wayland 标准与实践。

3.3 Rust

Rust 是一门强调安全性与并发性能的系统级语言，具备以下几个关键优势，使其特别适合构建图形协议栈与桌面管理器：

内存安全 (Memory Safety) Rust 通过所有权系统与静态借用检查器，在编译期保障内存访问合法性，杜绝 Use-After-Free、空指针解引用等常见错误，无需垃圾回收器。对于一个合成器来说，这意味着在处理 surface 生命周期、buffer 引用、输入事件时可以避免大量运行时错误。

数据并发性 (Fearless Concurrency) Rust 支持无数据竞争的并发操作，允许我们在后台异步处理输入事件、合成器状态更新与渲染流程，确保交互响应流畅且线程安全。

丰富的生态与 tooling cargo、clippy、rust-analyzer 等工具提供了出色的开发体验和持续集成支持，便于维护大型项目。与 Mesa、WGPU、EGL 等图形库的绑定日趋成熟，为集成硬件加速渲染提供了良好基础。

3.4 项目最小实现

3.4.1 架构概览

Smithay 采用 *calloop* 作为主事件循环框架，其优势在于：

- 可插拔式事件源管理 (source registration)
- 高性能的非阻塞式事件分发
- 原生支持定时器、通道等常用异步通信模型

Smithay 为 Winit 后端提供了优秀的兼容模式，可以很方便的进行开发。

3.4.2 EventLoop 事件分发机制

在基于 Smithay 构建的 Wayland Compositor 中，事件循环（EventLoop）是整个系统运行的核心。所有的输入、输出、客户端请求、时间驱动逻辑，乃至后台任务的调度都依赖于该机制完成事件的监听与响应。

定义

在 `main` 函数中定义一个 EventLoop 主体非常简单，直接调用相关的库函数：

```
rust
1 use smithay::reexports::calloop::EventLoop;
2 let mut event_loop: EventLoop<'_, State> = EventLoop::try_new().unwrap();
```

在这里，`State` 类型是全局状态结构体，由我们自己定义，目前暂时不谈论细节，你只需知道这个结构体管理所有的程序状态即可。

通过获取 `LoopHandle` 就来执行事件的插入，删除与执行操作：

```
rust
1 event_loop
2     .handle() // LoopHandle
3     .insert_source(input_backend, move |event, &mut metadata, state| {
4         // action
5     })?;
```

在这里，我们通过 `handle()` 函数获取操作入口，使用 `insert_source` 函数来注册 `EventSource`，其会将一个监听对象添加到主循环中，并且绑定一个处理函数（回调闭包），每当事件产生时，就会调用这个函数。

事件循环可以绑定多个事件源，常见类别如下：

类型	来源	示例事件
输入设备	libinput	PointerMotion、KeyboardKey 等
图形输出	DRM/KMS, Winit	热插拔、显示尺寸改变
Wayland 客户端	WaylandSocket	请求窗口创建、buffer attach
定时器	calloop Timer	动画帧调度、超时
自定义通道	calloop Channel	后台任务返回、信号触发

在 `insert_source` 中绑定的回调闭包具有以下签名：

```
rust
1 FnMut(E, &mut Metadata, &mut State)
```

- `E`: 来自事件源的事件本体，类型依赖于事件源。

- **Metadata**: 事件元信息 (通常是 `calloop::generic::GenericMetadata`)，包含事件触发时的底层 I/O 状态，例如可读/可写标志。大多数情况下你可以忽略该参数，除非你要做更底层的 I/O 操作。
- **State**: 传入的全局状态对象，是你自定义的全局状态结构，也就是一开始定义的类型 `EventLoop<'_, State>` 中的 `State`。

或许你会疑惑我们只是告诉了 `EventLoop` 的 `State` 类型，没有实现 `State` 值的传入，为什么这里可以获取到一个可变借用，别着急，后面就会揭晓答案

最容易理解的就是客户端连接请求的事件处理：

```
rust
1 let source = ListeningSocketSource::new_auto().unwrap();
2 let socket_name = source.socket_name().to_string_lossy().into_owned();
3 loop_handle
4     .insert_source(source, move |client_stream, _, state| {
5         state
6             .display_handle
7             .insert_client(client_stream, Arc::new(ClientState::default()))
8             .unwrap();
9     })
10    .expect("Failed to init wayland socket source.");
```

`Wayland` 是一个基于 `UNIX` 域套接字 (`UNIX domain socket`) 的通信协议，`Client` 与 `Compositor` 之间的所有协议交互，都是通过一个共享的本地套接字进行的。

`ListeningSocketSource::new_auto()` 会自动创建一个新的 `UNIX` 域套接字，并监听客户端连接请求。默认在 `/run/user/UID/` 下创建 `socket` 文件，例如 `wayland-0`。本地调试时我们需要设置环境变量 `WAYLAND_DISPLAY=wayland-0` 来绑定测试的 `Compositor`。

当有客户端连接或请求发生时，对应的事件将触发该回调闭包，并调用 `.display_handle.insert_client` 以执行客户端初始化、资源绑定或协议处理等逻辑。

详细的创建内容在 `Client` 事件源 篇会详细讲解。

事件执行

此前我们只是将需要监听的事件源和需要执行的函数内容加入到了 `EventLoop` 中，但还未真正的下达指令 - 你可以开始监听了，因此，我们还需要以下代码来真正开启循环：

```
rust
1 event_loop
2     .run(None, &mut state, move |_| {
3         // is running
4     })
5     .unwrap();
```

此时，我们可以解答在事件源插入中遗留的问题了，可变借用是此时才被传入其中的，顺序上也许会让人疑惑，但这就是 Rust 的“延迟状态绑定”机制的奇妙之处。

在调用 `insert_source` 时，事件循环尚未开始运行，只是注册了事件源与回调；

所有回调的 `state` 参数类型由 `EventLoop<T>` 的泛型 `T` 决定（例如我们定义的 `State`），但值本身尚未存在；

直到调用 `run(&mut state, ...)` 这一刻，`state` 的实际引用才被注入到事件循环中；

从此刻开始，`calloop` 内部在每次事件分发时，才会将这个 `&mut T` 传入闭包中。

它确保了事件循环中所有 `state` 的使用都在 `run()` 的生命周期范围内发生，且绝不会出现悬垂引用或数据竞争。

至此，核心的框架就已经被我们解决了，接下来就是真正的进行对不同事件源的处理。

3.4.3 Client 事件源

在 Wayland 协议中，客户端的渲染行为是以 `wl_surface` 为基本单位的。每一个客户端窗口本质上都是在创建并提交一个或多个 `surface`，而这些 `surface` 的行为则由其绑定的角色（如 `xdg_toplevel` 或 `xdg_popup`）所定义。

在之前我们已经见过以下的代码：

```
rust
1 let source = ListeningSocketSource::new_auto().unwrap();
2 let socket_name = source.socket_name().to_string_lossy().into_owned();
3 loop_handle
4     .insert_source(source, move |client_stream, _, state| {
5         state
6             .display_handle
7             .insert_client(client_stream, Arc::new(ClientState::default()))
8             .unwrap();
9     })
10    .expect("Failed to init wayland socket source.");
```

这段代码创建了一个新的 UNIX 域套接字，监听客户端的连接请求。Wayland 是一个 拉模型 (`pull model`)，客户端不会主动创建窗口，而是从服务端请求对象并获得引用。其中具体的请求过程如下：

- 连接 `display socket`: 客户端连接 `compositor` 暴露的 UNIX 域套接字（如 `/run/user/1000/wayland-0`）。
- 绑定 `wl_registry`: 连接后，客户端首先获取 `wl_display` 提供的 `wl_registry` 对象，它包含了 `compositor` 所支持的所有全局对象（如 `wl_compositor`、`wl_shm`、`xdg_wm_base` 等）。

- 获取 `wl_compositor` 接口：客户端通过 `wl_registry.bind(...)` 获得 `wl_compositor` 接口，允许创建 `wl_surface`。
- 创建 `wl_surface`：客户端通过 `wl_compositor.create_surface()` 调用，获得一个新的 `surface` 例，这是所有图形内容提交的基础。
- 绑定 `xdg_surface` 与角色：随后，客户端使用 `xdg_wm_base.get_xdg_surface(surface)` 创建 `xdg_surface`，再通过 `get_toplevel()` 等调用为其赋予具体角色。
- 随后就可以通过 `surface.commit()` 向 `compositor` 传递需要显示的内容。

看到如此多的协议信息，不要害怕！让我们一步步的来拆解各个步骤的具体含义，首先有必要介绍一下 `xdg-shell` 协议。

3.4.4 xdg-shell 协议实现

[protocol link: XDG shell protocol | Wayland Explorer](#)

在 `Wayland` 协议体系中，`xdg-shell` 是一项核心协议，扩展了基础的 `wl_surface` 对象，使其能够在桌面环境下扮演窗口的角色。它是现代 `Wayland` 桌面应用窗口管理的标准，涵盖了顶层窗口、弹出窗口、窗口状态控制等一系列行为。

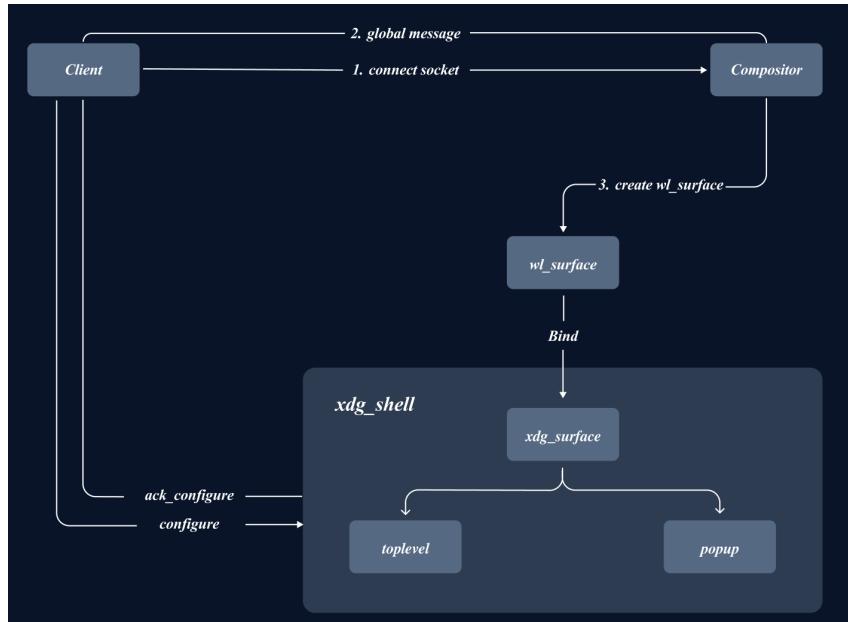


图 10 xdg_shell 协议示意图

`xdg-shell` 协议主要围绕以下对象展开：

- `xdg_wm_base`：客户端首先通过 `wl_registry` 获取 `xdg_wm_base` 接口。
- `xdg_surface`：通过 `xdg_wm_base.get_xdg_surface(wl_surface)`，客户端将一个基础的 `wl_surface` 与 `xdg_surface` 关联起来。
- `xdg_toplevel`：通过 `xdg_surface.get_toplevel()`，该 `surface` 被赋予了「顶层窗口」的角色。

- `xdg_popup`: 替代 `toplevel`, 它赋予窗口「弹出窗口」的角色, 通常用于菜单、右键栏等临时 UI。

一个 `wl_surface` 只能被赋予一个角色, 即它要么是 `xdg_toplevel`, 要么是 `xdg_popup`, 不能同时拥有或重复绑定。

我们可以这样理解: `wl_surface` 是原始画布, `xdg_surface` 是语义包装器, `xdg_toplevel` 或 `xdg_popup` 是具体的行为描述者。

3.4.5 configure / ack 机制

在 `xdg-shell` 协议中, 一个非常重要的机制就是「双向确认机制」:

在有修改需求的时候, `compositor` 发起 `configure` 事件, 告知客户端窗口大小、状态变更请求, 客户端必须回应 `ack_configure`, 明确表示接收到该配置并将进行重绘, 只有在 `ack` 后, 客户端提交的 `surface.commit()` 内容才会被正式展示。

这种机制是 `Wayland` 相对于传统 `X11` 的一大改进点, 确保了服务端与客户端状态始终一致, 不会出现窗口闪动或布局错乱。

rust

```

1 use smithay::{
2     delegate_xdg_shell,
3     wayland::shell::xdg::{XdgShellHandler, XdgShellState},
4 };
5
6 // init in state struct
7 {
8     ...
9     let xdg_shell_state = XdgShellState::new::<Self>(&display_handle);
10    ...
11 }
12
13 impl XdgShellHandler for State {
14     fn xdg_shell_state(&mut self) -> &mut XdgShellState {
15         //
16     }
17
18     fn new_toplevel(&mut self, surface: ToplevelSurface) {
19         //
20     }
21
22     fn new_popup(&mut self, surface: PopupSurface, _positioner: PositionerState)
23     {
24         //
25     }
26
27     fn reposition_request(
28         &mut self,
29         surface: PopupSurface,
30         positioner: PositionerState,
31         token: u32,
32     ) {

```

```

32         //
33     }
34
35     fn toplevel_destroyed(&mut self, surface: ToplevelSurface) {
36         //
37     }
38
39     fn move_request(&mut self, surface: ToplevelSurface, seat: wl_seat::WlSeat,
40                     serial: Serial) {
41         //
42     }
43
44     fn resize_request(
45         &mut self,
46         surface: ToplevelSurface,
47         seat: wl_seat::WlSeat,
48         serial: Serial,
49         edges: xdg_toplevel::ResizeEdge,
50     ) {
51         //
52     }
53
54     fn grab(&mut self, _surface: PopupSurface, _seat: wl_seat::WlSeat, _serial:
55         Serial) { }
56 }
57 delegate_xdg_shell!(State);

```

设置 xdg-shell 协议的相关代码也非常简单，只需要使用 smithay 提供的框架即可。具体函数内部实现的方法，参考基础框架代码。

至此，我们已经完成了核心的 surface 分配机制，相当于给画家提供了画板，还设置了画板最后展出的场馆 - toplevel 或 popup。

3.4.6 input 事件源

compositor 的核心职责之一是处理来自用户的输入事件，如鼠标移动、按键、触摸交互等。而这些输入事件的来源方式依赖于 compositor 所使用的后端类型。Smithay 提供了多个后端支持，其中包括：

- winit 后端：通常用于开发阶段，快速接入图形窗口系统并获取输入；
- TTY + libinput 后端：更贴近生产环境，直接从内核设备文件读取输入事件，适用于 DRM/KMS 渲染路径。

3.4.6.1 使用 winit 后端的 input 事件源

在 winit 模式下，Smithay 提供了高度集成的 WinitInputBackend 类型，开发者可以非常方便地将其作为事件源插入 EventLoop 中，例如：

```
1 event_loop
```

rust

```

2     .handle()
3     .insert_source(winit_backend, move |event, _, state| {
4         state.process_input_event(event);
5     })?;

```

winit 后端封装了窗口事件与输入事件，并提供统一的接口输出 InputEvent。Smithay 内部支持对这些事件进行标准化转换，如：

- PointerEvent
- KeyboardEvent
- TouchEvent

通常在 state.process_input_event 函数中进行分发，Smithay 的 seat 抽象会帮助我们自动处理焦点跟踪、输入分发、键盘修饰等细节。

```

rust
1 use smithay::backend::{delegate_seat, input::{Seat, SeatHandler, SeatState}};
2
3 // init in state struct
4 {
5     let mut seat_state = SeatState::new();
6     let seat_name = String::from("winit");
7     let mut seat: Seat<Self> = seat_state.new_wl_seat(display_handle, seat_name);
8 }
9
10 impl SeatHandler for State {
11     type KeyboardFocus = WlSurface;
12     type PointerFocus = WlSurface;
13     type TouchFocus = WlSurface;
14
15     fn seat_state(&mut self) -> &mut SeatState<State> {
16         //
17     }
18
19     fn cursor_image(
20         &mut self,
21         _seat: &Seat<Self>,
22         image: smithay::input::pointer::CursorImageStatus,
23     ) {
24         //
25     }
26
27     fn focus_changed(&mut self, seat: &Seat<Self>, focused: Option<&WlSurface>)
28     {
29         //
30     }
31 delegate_seat(State);

```

3.4.6.2 使用 TTY 后端的 input 事件源

在没有图形服务器支持的裸机环境下，我们通常使用 TTY 作为图形输出后端，并结合 libinput 获取来自 /dev/input 的事件。此时输入处理方式较为底层，需要我们显式构造事件源：

rust

```
1 let libinput_context = Libinput::new_with_udev(...);
2 let input_backend = LibinputInputBackend::new(libinput_context, seat, ...);
```

与 winit 不同，libinput 后端需要手动处理权限和 seat 初始化，但优点在于：

- 支持更精细的输入设备管理；
- 能兼容热插拔、多用户、多 seat；
- 更贴近真实硬件行为（如 VT 切换、KMS 挂载）；

事件注册仍然可以通过 insert_source 完成。

rust

```
1 event_loop
2     .handle()
3     .insert_source(input_backend, move |event, _, state| {
4         state.process_input_event(event);
5     })?;
```

无论是 winit 还是 TTY 模式，输入事件的处理流程基本保持一致：

- 后端产生 InputEvent；
- 事件被传入 compositor 的状态处理器；
- Smithay 的 Seat 接口会自动更新焦点状态、生成 Wayland 协议事件；
- 若存在活跃客户端，事件会通过 wl_pointer、wl_keyboard、wl_touch 等接口传输至客户端。

具体的状态如：InputEvent::Keyboard, InputEvent::PointerMotion 等这里不再详细讲解，具体参考基础框架代码内容。

至此，我们就得到了一个简单的，可以响应客户端请求，并且支持鼠标、键盘操作的简易 Wayland Compositor。

3.5 输入设备事件监听

在 Mondrian 中，我们实现了一个轻量且可扩展的全局快捷键匹配系统，用于：

- 启动应用程序（如打开终端）
- 执行窗口管理指令（如聚焦切换、窗口平铺方向调整）
- 支持用户自定义的命令绑定

3.5.1 快捷键的输入流程概览

- Wayland 中键盘事件由 `wl_keyboard` (或 `xkb`) 协议触发，最终通过 `InputManager` 处理。
- 快捷键响应链：键盘事件 → 按键状态判定（按下/释放）→ 匹配组合键 → 执行对应指令

3.5.2 正则匹配：解析指令或快捷命令

用户定义的快捷指令存储在 `/keybindings.conf` 文件中，例如：

```
1 # /keybindings.conf
2 bind = Super_L+f, command, "firefox"
3 bind = Super_L+1, exec, "workspace-1"
```

json

为了支持复杂的指令格式，我们在命令解析阶段引入了 Rust 的 `regex` 正则库：

```
1 let re =
2     // bind = Ctrl + t, command, "kitty"
3     // bind = Ctrl + 1, exec, "func1"
4     Regex::new(r#"(?m)^bind\s*=\s*([^,]+?),\s*(command|exec),\s*([^\n]+)"(?:\s*#.*?")#
5             .unwrap();
6
7 for cap in re.captures_iter(&content) {
8     let keybind = &cap[1]; // Ctrl+t / Alt+Enter
9     let action = &cap[2]; // exec / command
10    let command = &cap[3]; // kitty / rofi -show drun
11    ...
12 }
```

rust

对于解析指令与快捷命令，我们使用 `KeyAction` 存储命令内容，对于解析指令，另外使用 `FunctionEnum` 进行存储，方便后续使用：

```
1 #[derive(Debug)]
2 pub enum FunctionEnum {
3     SwitchWorkspace1,
4     SwitchWorkspace2,
5     InvertWindow,
6     Expansion,
7     Recover,
8     Quit,
9     Kill,
10    Json,
11    Up(Direction),
12    Down(Direction),
13    Left(Direction),
14    Right(Direction),
15 }
16
17 #[derive(Debug)]
18 pub enum KeyAction {
19     Command(String, Vec<String>),
```

rust

```
20     Internal(FunctionEnum),  
21 }
```

完整的快捷键识别与匹配，对于 Ctrl Shift 等键，将其设置为左右两键均可触发，保证后续识别执行正确：

rust

```
1 impl InputManager{  
2     fn load_keybindings(path: &str) -> anyhow::Result<HashMap<String,  
3         KeyAction>> {  
4         let content = fs::read_to_string(path).anyhow_err("Failed to load  
5             keybindings config")?;  
6         let mut bindings = HashMap::new();  
7         let re =  
8             // bind = Ctrl + t, command, "kitty"  
9             // bind = Ctrl + 1, exec, "func1"  
10            Regex::new(r#"(?m)^s*bind\s*=\s*(\^,]+?)\s*(command|exec),  
11                \s*"( [^"]+)"(?:\s*#.*)?#"  
12                .unwrap();  
13  
14         let modifier_map: HashMap<&str, Vec<&str>> = [  
15             ("Ctrl", vec!["Control_L", "Control_R"]),  
16             ("Shift", vec!["Shift_L", "Shift_R"]),  
17             ("Alt", vec!["Alt_L", "Alt_R"]),  
18             ("Esc", vec!["Escape"]),  
19             ("[", vec!["bracketleft"])),  
20             ("]", vec!["bracketright"])),  
21             (",", vec!["comma"])),  
22             (".", vec!["period"])),  
23             ("/", vec!["slash"])),  
24             (";", vec!["semicolon"])),  
25             (".", vec!["period"])),  
26             ("'", vec!["apostrophe"])),  
27         ]  
28         .into_iter()  
29         .collect();  
30  
31         for cap in re.captures_iter(&content) {  
32             let keybind = &cap[1]; // Ctrl+t / Alt+Enter  
33             let action = &cap[2]; // exec / command  
34             let command = &cap[3]; // kitty / rofi -show drun  
35  
36             let keys: Vec<String> = keybind  
37                 .split('+')  
38                 .map(|key| {  
39                     if let Some(modifiers) = modifier_map.get(key) {  
40                         modifiers.iter().map(|m| m.to_string()).collect()  
41                     } else {  
42                         vec![key.to_string()]  
43                     }  
44                 })  
45                 .multi_cartesian_product()  
46                 .map(|combination| combination.join("+"))  
47             )  
48         }  
49     }  
50 }
```

```

45         .collect();
46
47     for key in keys {
48         let action_enum = match action {
49             "command" => {
50                 let mut parts = command.split_whitespace();
51                 let cmd = parts.next().unwrap_or("").to_string();
52                 let args: Vec<String> = parts.map(|s|
53                     s.to_string()).collect();
54
55                     KeyAction::Command(cmd, args)
56                 }
57             "exec" => {
58                 let internal_action = match command.trim() {
59                     "workspace-1" => FunctionEnum::SwitchWorkspace1,
60                     "workspace-2" => FunctionEnum::SwitchWorkspace2,
61                     "invert" => FunctionEnum::InvertWindow,
62                     "recover" => FunctionEnum::Recover,
63                     "expansion" => FunctionEnum::Expansion,
64                     "quit" => FunctionEnum::Quit,
65                     "kill" => FunctionEnum::Kill,
66                     "json" => FunctionEnum::Json,
67                     "up" => FunctionEnum::Up(Direction::Up),
68                     "down" => FunctionEnum::Down(Direction::Down),
69                     "left" => FunctionEnum::Left(Direction::Left),
70                     "right" => FunctionEnum::Right(Direction::Right),
71                     _ => {
72                         tracing::info!(
73                             "Warning: No registered function for exec
74
75                             '{}',",
76                             command
77                         );
78                         continue;
79                     }
80                 };
81                 KeyAction::Internal(internal_action)
82             }
83             _ => continue,
84         };
85     }
86
87 #[cfg(feature = "trace_input")]
88 for (key, action) in &bindings {
89     tracing::info!(%key, action = ?action, "Keybinding registered");
90 }
91
92 Ok(bindings)
93 }
94 }

```

3.5.3 Keymap 映射：输入事件的键码与组合键识别

使用 `xkbcommon` 配合 `Smithay::input`，可以将原始键码解析为用户理解的键位，如：

- 原始：`keycode = 38`（硬件码）
- 解析后：`keysym = "a"`

一般快捷键均由功能键发起，为了确保识别正确，我们定义了一个优先级 map，用于设置功能键优先于所有字母键。

```
rust
1 ...
2 // priority: Ctrl > Shift > Alt
3 let priority_map: HashMap<String, i32> = [
4     ("Super_L", 0),
5     ("Control_L", 1),
6     ("Control_R", 1),
7     ("Shift_L", 2),
8     ("Shift_R", 2),
9     ("Alt_L", 3),
10    ("Alt_R", 3),
11 ]
12 .into_iter()
13 .map(|(k, v)| (k.to_string(), v))
14 .collect();
15 ...
```

在按下某个按键时，通过 `keysym_get_name()` 得到硬件码对应的可读 ASCII 码，并且将其按照优先级排序后，排列成当前按下键，交与 `action_keys()` 函数处理快捷键事件。

```
rust
1 ...
2 KeyState::Pressed => {
3     let mut pressed_keys_name: Vec<String> =
4         keyboard.with_pressed_keysyms(|keysym_handles| {
5             keysym_handles
6                 .iter()
7                 .map(|keysym_handle| {
8                     let keysym_value = keysym_handle.modified_sym();
9                     let name = keysym_get_name(keysym_value);
10                    if name == "Control_L" {
11                        #[cfg(feature = "trace_input")]
12                        info!("mainmod_pressed: true");
13                        data.input_manager.set_mainmode(true);
14                    }
15                    name
16                })
17                 .collect()
18            });
19     pressed_keys_name
20         .sort_by_key(|key| priority_map.get(key).cloned().unwrap_or(3));
21     let keys = pressed_keys_name.join("+");
22     #[cfg(feature = "trace_input")]
```

```

23     info!("Keys: {:?}", keys);
24     data.action_keys(keys, serial);
25 }
26 ...
27 ...
28 pub fn action_keys(&mut self, keys: String, serial: Serial) {
29     let keybindings = self.input_manager.get_keybindings();
30     if let Some(command) = keybindings.get(&keys) {
31         match command {
32             KeyAction::Command(cmd, args) => {
33                 #[cfg(feature = "trace_input")]
34                 info!("Command: {} {}", cmd, args.join(" "));
35                 let mut command = std::process::Command::new(cmd);
36                 for arg in args {
37                     command.arg(arg);
38                 }
39                 match command.spawn() {
40                     #[cfg(feature = "trace_input")]
41                     Ok(child) => {
42                         info!("Command spawned with PID: {}", child.id());
43                     }
44                     Err(e) => {
45                         error!(
46                             "Failed to execute command '{} {}': {}",
47                             cmd,
48                             args.join(" "),
49                             e
50                         );
51                     }
52                     #[cfg(not(feature = "trace_input"))]
53                     _ => {}
54                 }
55             }
56             KeyAction::Internal(func) => match func {
57                 FunctionEnum::SwitchWorkspace1 => {
58                     self.set_keyboard_focus(None, serial);
59                     self.workspace_manager.set_activated(WorkspaceId::new(1));
60                 }
61                 ...
62             },
63         }
64     }
65 }
66 ...

```

3.6 DRM/KMS 裸机直连

项目在裸机终端中自行初始化 DRM/KMS 图形管线，并通过 GBM 和 EGL 建立 GPU 渲染上下文，使用 OpenGL ES 进行硬件加速合成显示。启动后该 Compositor 接管系统图形输出，并成为客户端程序（如终端模拟器、浏览器）的 Wayland 显示服务。

3.6.1 Linux 图形栈核心技术组件

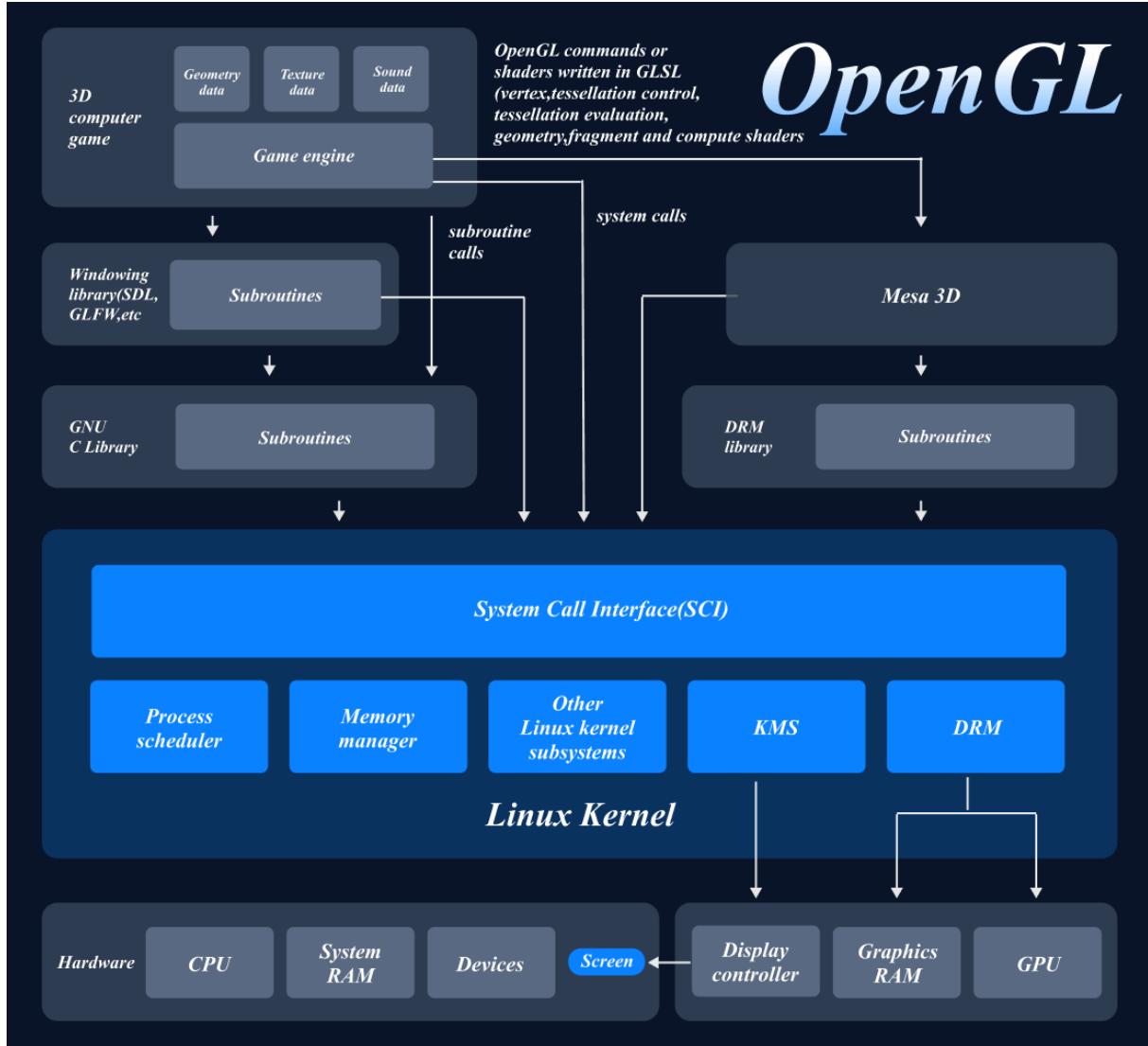


图 11 opengl 渲染过程演示图

用画廊来举例，会比较容易理解。

画师就是 OpenGL/GLES，用于绘制用户提交的绘制需求，在绘制之前，画廊陈列员 (EGL) 会负责与库存管理员 (GBM) 联系，确定好最终需要陈放画框的大小 (buffer size)，位置 (egl buffer 映射) 以及一些其他内容 (egl context)。画师绘制完图画以后，先将图画堆积到队列中 (queue frame)，时机到达后 (VBlank) 就将原先墙上的画拿下，然后挂上新的画 (page flip)。

下面是正式的介绍。

3.6.1.1 OpenGL/GLES

OpenGL (Open Graphics Library) 与其精简版 OpenGL ES (Embedded Systems) 是广泛使用的跨平台图形渲染 API, 用于执行图形计算和渲染操作。在嵌入式或资源受限的环境中, OpenGL ES 更为常用, 其接口更加轻量, 适合直接在 TTY 裸机模式下运行。

在本项目中, OpenGL ES 被用于执行 GPU 加速的图形渲染任务。具体包括:

- 几何图形的绘制 (如窗口、装饰、阴影);
- 着色器程序的编译与执行;
- 将渲染内容输出到帧缓冲 (Framebuffer) 中, 供后续显示。

在 TTY 裸机模式下, 合成器通过 OpenGL ES 执行图形绘制操作, 如几何图元绘制、纹理映射和着色器执行, 最终将图像渲染到 GPU 管理的缓冲区 (Framebuffer) 中。

3.6.1.2 EGL

EGL 是连接 OpenGL ES 与本地窗口系统 (如 X11、Wayland、或裸设备如 GBM) 的中间接口库。其职责包括:

- 初始化图形上下文;
- 创建渲染表面 (EGLSurface);
- 在渲染器与底层硬件 (GBM、DRM) 之间建立连接;
- 管理 buffer swap (如 eglSwapBuffers()) 与同步机制。

在 TTY 环境中, EGL 通常与 GBM 配合使用, 将 GPU buffer 分配出来供 OpenGL ES 使用, 建立渲染到显示设备之间的桥梁。

3.6.1.3 GBM (Generic Buffer Management)

GBM 是 Mesa 提供的一个用于和内核 DRM 系统交互的库, 它的主要功能是:

- 分配可被 GPU 渲染的缓冲区 (bo: buffer object);
- 将这些缓冲区导出为 DMA-BUF handle, 用于与 DRM 或其他进程共享;
- 为 EGL 提供可渲染的 EGLNativeWindowType。

GBM 允许在没有窗口系统的场景下 (如 TTY 模式) 创建 OpenGL 可用的 framebuffer, 从而支持嵌入式系统和裸机合成器的图形输出。

3.6.1.4 Mesa3D

Mesa3D 是开源图形栈的核心, 提供了 OpenGL、OpenGL ES、EGL、GBM 等多个图形接口的完整实现。它在用户空间运行, 并与内核空间的 DRM 驱动协同工作。

Mesa 提供以下功能:

- 实现 OpenGL / GLES API，并将其转译为 GPU 硬件可识别的命令；
- 管理 shader 编译、状态机、纹理、缓冲区等所有渲染细节；
- 实现 GBM 与 DRM 的绑定，支持 buffer 分配与传输；
- 调度 page flip 请求，通过 DRM 与显示硬件同步。

3.6.1.5 DRM (Direct Rendering Manager)

直接渲染管理器 (Direct Rendering Manager, 缩写为 DRM) 是 Linux 内核图形子系统的一部分，负责与 GPU (图形处理单元) 通信。它允许用户空间程序 (如图形服务器或 Wayland compositor) 通过内核公开的接口，完成以下关键任务：

- 分配和管理图形缓冲区 (buffer)
- 设置显示模式 (分辨率、刷新率等)
- 与显示设备 (显示器) 建立连接
- 将 GPU 渲染结果显示到屏幕上 - PageFlip 页面翻转

DRM 是现代 Linux 图形栈的基础，允许程序绕过传统 X Server，直接操作 GPU，形成了“GPU 直连”的渲染路径。

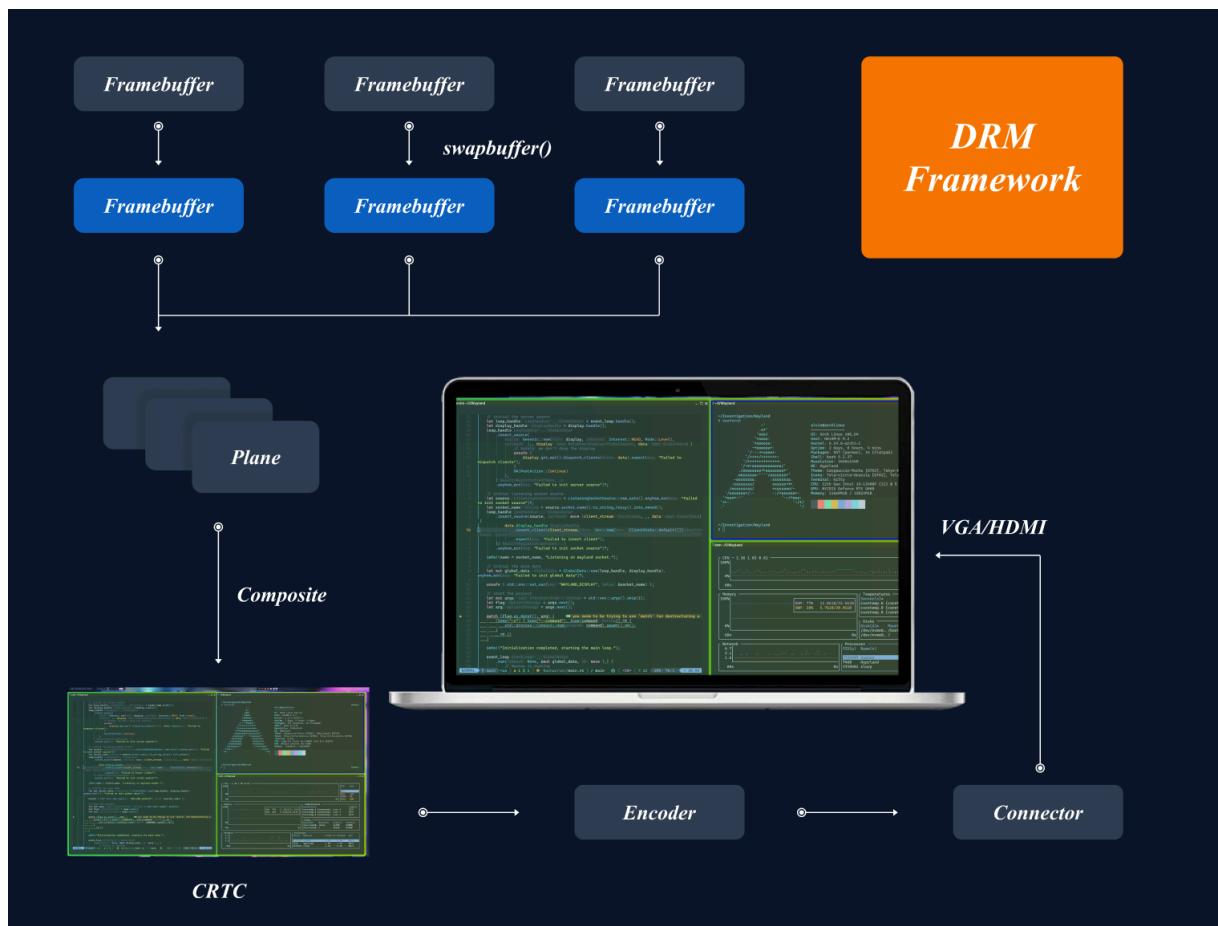


图 12 DRM/KMS 系统演示图

要想理解 DRM，首先要理解两个关键子模块的工作内容：

GEM (Graphic Execution Manager)

图形执行管理器 (Graphics Execution Manager, 简称 GEM) 是 DRM 子系统中的另一个重要模块，主要用于内存管理，即如何分配和管理 GPU 可访问的图形缓冲区 (buffer)。

它提供了如下功能：

- 为用户空间分配 GPU 使用的显存或系统内存缓冲区
- 提供缓冲区在用户空间与内核空间之间的共享与引用机制
- 管理缓冲区的生命周期和同步（避免读写冲突）

帧缓冲区对象 (framebuffer) 是帧内存对象的抽象，它提供了像素源给到 CRTC。帧缓冲区依赖于底层内存管理器分配内存。

在程序中，使用 DRM 接口创建 framebuffer、EGL 创建的渲染目标，底层通常都通过 GEM 管理。GEM 的存在使得多种图形 API (OpenGL ES、Vulkan、视频解码等) 可以统一、高效地访问 GPU 资源。

KMS (Kernel Mode Setting)

内核模式设置 (Kernel Mode Setting, 简称 KMS) 是 DRM 的子系统，用于控制显示设备的“输出路径”，即显示管线。它允许在内核空间完成分辨率设置、刷新率调整、帧缓冲切换等操作，而不依赖用户空间的图形服务器。

KMS 将整个显示控制器的显示 pipeline 抽象成以下几个部分：

- **Plane (图层)**

每个 plane 表示一块可渲染的图像区域，可独立组合显示输出。plane 分为三类：

- Primary：主图层，必需。对应于整个屏幕内容，通常显示整个帧缓冲区。
- Cursor：用于显示鼠标光标，通常是一个小图层，支持硬件加速。
- Overlay：可选的叠加图层，用于视频加速或硬件合成。

- **CRTC (Cathode Ray Tube Controller)**

控制图像从 plane 传送到 encoder，类似一个“图像流控制器”，主要用于管理显示设备的扫描和刷新。一个 CRTC 通常绑定一个主 plane，但也可能支持多个 overlay。

- **Encoder (编码器)**

将图像信号从 GPU 转换为特定格式，如 HDMI、DP、eDP、VGA 等。

- **Connector** (连接器)

表示实际的物理接口 (如 HDMI 接口、DisplayPort 接口), 对应连接的显示设备 (monitor)。

>  工作流程示意: **Plane** → **CRTC** → **Encoder** → **Connector** → **Monitor**

3.6.2 Wayland 通信流程与显示流程

本项目实现了一个独立于 X11、无需任何桌面环境即可运行的 Wayland 合成器 (compositor), 通过直接接管 TTY 并使用 DRM/KMS 完成图形显示。在显示系统的构建中, Wayland 扮演的是图形系统通信协议的角色, 而具体的渲染、显示和输入处理由 DRM、GBM、EGL 与 libinput 等模块协同完成。

Wayland 合成器的主要职责是:

- 接受客户端 (Wayland client) 的连接与绘图请求
- 将客户端 buffer 进行合成、渲染并显示在屏幕上
- 处理来自内核的输入事件

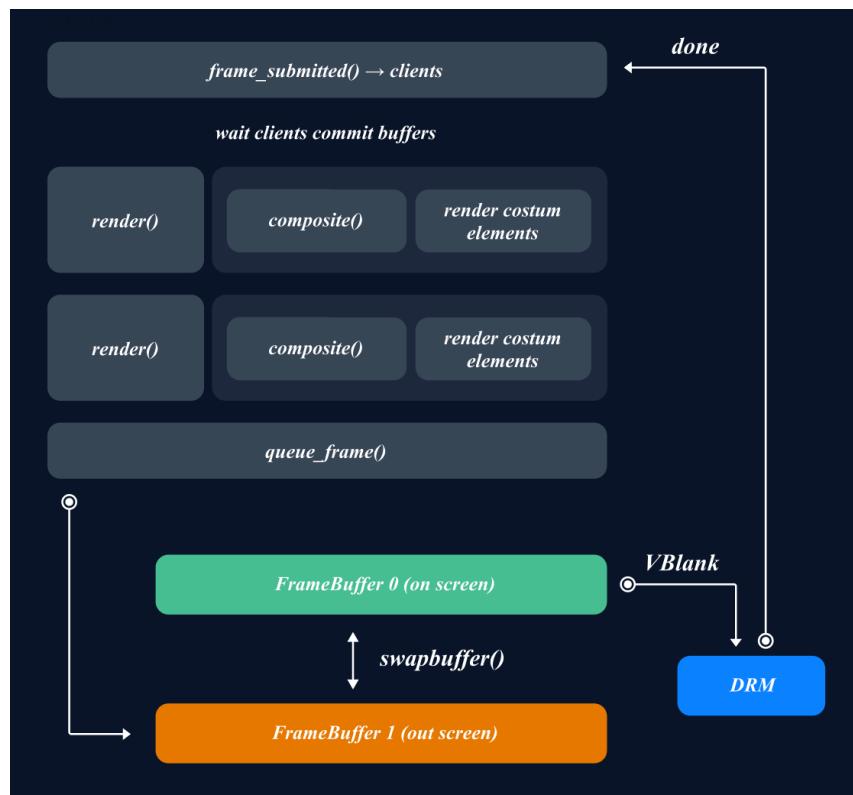


图 13 DRM/KMS 系统演示图

```

1 [Wayland Client]
2     ↓ 提交 buffer (wl_buffer / linux-dmabuf)
3 [Compositor]
4     ↓ OpenGL 合成 (将多个窗口 buffer 组合)
5 [Framebuffer]

```

```
6      ↓ DRM 显示 pipeline (crtc → encoder → connector)
7 [Monitor Output]
```

3.6.2.1 客户端连接与交互

每个 Wayland 客户端通过 Socket 与合成器通信，注册所需协议（如 wl_surface, xdg_surface），并通过共享内存或 GPU buffer 提交其绘制内容。

3.6.2.2 Buffer 获取与提交

客户端通过 wl_buffer 协议提供绘制完成的内容。这个 buffer 可能来自：

- wl_shm: CPU 绘制后的共享内存（较慢）
- linux-dmabuf: GPU 渲染结果，零拷贝

3.6.2.3 合成器接管 buffer 并合成

合成器在服务端接收 attach / commit 请求后，将客户端的 buffer 记录为当前帧的一部分。在下一帧刷新中，所有窗口的 buffer 会被 GPU 合成到一个输出 surface 上。

3.6.2.4 GPU 渲染与提交

使用 OpenGL ES 渲染这些 buffer（如绘制窗口、阴影、边框等），再通过 eglSwapBuffers 提交帧缓冲，交由 DRM 显示。

3.6.2.5 Page Flip 显示与 VBlank 同步

合成后的 framebuffer 通过 drmModePageFlip 提交，等待垂直同步（VBlank）时切换至新帧，防止 tearing。

3.6.3 输入事件处理流程

3.6.3.1 libinput/evdev

evdev (Event Device) 是 Linux 内核提供的一个通用输入事件接口，所有输入设备（键盘、鼠标、触控板、游戏手柄等）在内核中都会以 /dev/input/eventX 设备节点的形式暴露，用户空间可以通过这些节点读取输入事件（如按键、移动、点击等）。

然而，直接与 evdev 接口打交道较为繁琐、底层，且各类设备的事件语义不尽相同。因此，在现代图形系统中，通常借助 libinput 作为更高级的输入事件处理库。

libinput 是一个用户空间库，提供统一的输入设备管理接口，具备以下功能：

- 统一处理来自 evdev 的事件流；
- 解析输入事件，生成高级抽象（如双指滚动、滑动、手势等）；
- 管理输入设备的生命周期（添加、移除）；
- 提供输入设备的识别信息（厂商、型号、功能等）；

- 与 Wayland compositor 无缝集成，支持多种后端（如 udev、seatd）。

输入事件首先由 Compositor 进行解析，无需响应时间时，发送给对应拥有 keyboard, pointer, touch focus 的客户端，通过协议如 wl_pointer.motion, wl_keyboard.key, wl_touch.down 等完成回传。

3.6.4 代码实现细节

Tty 后端部分代码量过大，这里只解释核心的代码部分。

基本数据结构：

```
rust
1 pub struct Tty {
2     pub session: LibSeatSession,
3     pub libinput: Libinput,
4     pub gpu_manager: GpuManager<GbmGlesBackend<GlesRenderer, DrmDeviceFd>>,
5     pub primary_node: DrmNode,
6     pub primary_render_node: DrmNode,
7     pub devices: HashMap<DrmNode, GpuDevice>,
8     pub seat_name: String,
9     pub dmabuf_global: Option<DmabufGlobal>,
10 }
11 pub struct GpuDevice {
12     token: RegistrationToken,
13     render_node: DrmNode,
14     drm_scanner: DrmScanner,
15     surfaces: HashMap<crtc::Handle, Surface>,
16     #[allow(dead_code)]
17     active_leases: Vec<DrmLease>,
18     drm: DrmDevice,
19     gbm: GbmDevice<DrmDeviceFd>,
20
21     // record non_desktop connectors such as VR headsets
22     // we need to handle them differently
23     non_desktop_connectors: HashSet<(connector::Handle, crtc::Handle)>,
24 }
25
26 pub struct Surface {
27     output: Output,
28     #[allow(dead_code)]
29     device_id: DrmNode,
30     render_node: DrmNode,
31     compositor: GbmDrmCompositor,
32     dmabuf_feedback: Option<SurfaceDmabufFeedback>,
33 }
34
35 type GbmDrmCompositor = DrmCompositor<
36     GbmAllocator<DrmDeviceFd>,
37     GbmDevice<DrmDeviceFd>,
38     Option<OutputPresentationFeedback>,
39     DrmDeviceFd,
40 >;
```

这里主要维护三个数据结构，`Tty` 为总后端，其持有多个 `OutputDevice`，也就是 GPU 设备，每个 GPU 设备可能会持有多个 `Surface`，对应的是显示器。

Tty 中还获取记录主 GPU 节点与其渲染节点，输入设备管理器名称等

```

47             device.drm.activate(false).expect("failed to activate
48             drm backend");
49
50             let device: &mut GpuDevice = if let Some(device) =
51                 data.backend.tty().devices.get_mut(&node) {
52                     device
53                 } else {
54                     warn!("not change because of unknown device");
55                     return;
56                 };
57
58             let crtcs: Vec<_> =
59                 device.surfaces.keys().copied().collect();
60                 for crtc in crtcs {
61                     data.backend.tty().render_output(
62                         node,
63                         crtc,
64                         data.clock.now(),
65                         &&mut data.render_manager,
66                         &&data.output_manager,
67                         &&data.workspace_manager,
68                         &&mut data.cursor_manager,
69                         &&data.input_manager,
70                         &&data.clock,
71                         &&data.loop_handle,
72                     );
73                 }
74             }
75             SessionEvent::PauseSession => {
76                 info!("Session paused");
77                 data.backend.tty().libinput.suspend();
78                 for device in data.backend.tty().devices.values_mut() {
79                     device.drm.pause();
80                 }
81             }
82         )
83         .unwrap();
84
85         // Initialize Gpu manager
86         let api =
87             GbmGlesBackend::with_context_priority(ContextPriority::Medium);
88         let gpu_manager = GpuManager::new(api).context("error creating the GPU
89             manager")?;
90
91         let primary_gpu_path = udev::primary_gpu(&seat_name)
92             .context("error getting the primary GPU")?
93             .context("couldn't find a GPU")?;
94
95         info!("using as the primary node: {:?}", primary_gpu_path);
96         let primary_node = DrmNode::from_path(primary_gpu_path)

```

```

96         .context("error opening the primary GPU DRM node")?;
97
98     info!("Primary GPU: {:?}", primary_node);
99
100    // get render node if exit - /renderD128
101    let primary_render_node = primary_node
102        .node_with_type(NodeType::Render)
103        .and_then(Result::ok)
104        .unwrap_or_else(|| {
105            warn!("error getting the render node for the primary GPU;
106            proceeding anyway");
107            primary_node
108        });
109    let primary_render_node_path = if let Some(path) =
110        primary_render_node.dev_path() {
111            format!("{}:{:?}", path)
112        } else {
113            format!("{}:{:?}", primary_render_node)
114        };
115    info!("using as the render node: {}", primary_render_node_path);
116
117    Ok(Self {
118        session,
119        libinput,
120        gpu_manager,
121        primary_node,
122        primary_render_node,
123        devices: HashMap::new(),
124        seat_name,
125        dmabuf_global: None,
126    })
127 }

```

Tty::new() 主要做了以下几件事：

- 监听 libinput 输入事件
- 监听 session 事件
- 初始化 gbm，获取主 GPU 信息

rs

```

1  pub fn init(
2      &mut self,
3      loop_handle: &LoopHandle<'_, GlobalData>,
4      display_handle: &DisplayHandle,
5      output_manager: &mut OutputManager,
6      render_manager: &RenderManager,
7      state: &mut State,
8  ) {
9      let udev_backend = UdevBackend::new(&self.seat_name).unwrap();
10
11     // gpu device

```

```

12     for (device_id, path) in udev_backend.device_list() {
13         if let Ok(node) = DrmNode::from_dev_id(device_id) {
14             if let Err(err) = self.device_added(
15                 node,
16                 &path,
17                 output_manager,
18                 loop_handle,
19                 display_handle,
20             ) {
21                 warn!("erro adding device: {:?}", err);
22             }
23         }
24     }
25     let mut renderer = self
26         .gpu_manager
27         .single_renderer(&self.primary_render_node)
28         .unwrap();
29
30     // initial shader
31     render_manager.compile_shaders(&mut renderer.as_gles_renderer());
32
33     state.shm_state.update_formats(renderer.shm_formats());
34
35     match renderer.bind_wl_display(display_handle) {
36         Ok(_) => info!("EGL hardware-acceleration enabled"),
37         Err(err) => info!(?err, "Failed to initialize EGL hardware-
38             acceleration"),
39     }
40
41     // create dmabuf
42     let dmabuf_formats = renderer.dmabuf_formats();
43     let default_feedback =
44         DmabufFeedbackBuilder::new(self.primary_render_node.dev_id(),
45             dmabuf_formats.clone()
46                 .build()
47                 .expect("Failed building default dmabuf feedback"));
48
49     let dmabuf_global = state
50         .dmabuf_state
51         .create_global_with_default_feedback::<GlobalData>(
52             display_handle,
53             &default_feedback,
54         );
55     self.dmabuf_global = Some(dmabuf_global);
56
57     // Update the dmabuf feedbacks for all surfaces
58     for device in self.devices.values_mut() {
59         for surface in device.surfaces.values_mut() {
60             surface.dmabuf_feedback = surface_dmabuf_feedback(
61                 surface.compositor.surface(),
62                 &self.primary_render_node,
63                 &surface.render_node,
64                 &mut self.gpu_manager
65             )
66         }
67     }

```

```

65      }
66
67      // Expose syncobj protocol if supported by primary GPU
68      if let Some(device) = self.devices.get(&self.primary_node) {
69          let import_device = device.drm.device_fd().clone();
70          if supports_syncobj_eventfd(&import_device) {
71              info!("syncobj enabled");
72              let syncobj_state =
73                  DrmSyncobjState::new::<GlobalData>(&display_handle,
74                      import_device);
75          state.syncobj_state = Some(syncobj_state);
76      }
77
78      loop_handle
79          .insert_source(udev_backend, move |event, _, data| match event {
80              UdevEvent::Added { device_id, path } => {
81                  if let Ok(node) = DrmNode::from_dev_id(device_id) {
82                      if let Err(err) = data.backend.tty().device_added(
83                          node,
84                          &path,
85                          &mut data.output_manager,
86                          &data.loop_handle,
87                          &data.display_handle,
88                      ) {
89                          warn!("erro adding device: {:?}", err);
90                      }
91                  }
92              }
93              UdevEvent::Changed { device_id } => {
94                  if let Ok(node) = DrmNode::from_dev_id(device_id) {
95                      data.backend.tty().device_changed(
96                          node,
97                          &mut data.output_manager,
98                          &data.display_handle,
99                          &data.loop_handle
100                     )
101                 }
102             }
103             UdevEvent::Removed { device_id } => {
104                 if let Ok(node) = DrmNode::from_dev_id(device_id) {
105                     data.backend.tty().device_removed(
106                         &data.loop_handle,
107                         &data.display_handle,
108                         node,
109                         &mut data.output_manager,
110                         &mut data.state,
111                     );
112                 }
113             }
114         })
115         .unwrap();
116     }

```

Tty::init() 主要完成以下几件事：

- 监听 udev，获取所有 GPU 设备以及其对应的显示器信息
- 按照给定帧率执行渲染流程

本项目目前只实现了单 GPU 单显示器固定帧率渲染，渲染部分主要按照此流程重复执行：

```

1 render_output() // 渲染合成指定显示器上的内容
2 ↓
3 queue_frame() // 将渲染好的内容送至等待队列，等待 pageflip
4 ↓
5 VBlank // 垂直同步信号
6 ↓
7 frame_submmit() // 提交帧，执行 pageflip
8 ↓
9 flush_client() // 通知客户端渲染下一帧

```

3.6.5 平铺式布局算法设计(计划修改算法，使用容器多叉树)

在一个窗口管理器中，布局系统扮演着核心角色。为了高效管理窗口的空间排布，本项目采用了一种结构清晰、修改高效的 容器式二叉树（Contain Binary Tree） 结构作为窗口布局的基础数据模型。该树结构基于 SlotMap 构建，结合唯一键值索引（Key-based access），理论上可以将常规操作如插入、删除、定位的时间复杂度优化至常数级别 $O(1)$ ，由于窗口数量一般不超过两位数，本项目综合考量时间与空间复杂度，最终实现 $O(n)$ 时间复杂度。

由于二叉树的方向表达能力不足，本项目还引入了 全局窗口邻接表 作为补充描述数据结构，记录全局所有窗口的临接方向关系。

为了管理和组织当前活动窗口的空间结构，在 Workspace 结构体中维护了两个核心字段：

```

rust
1 #[derive(Debug)]
2 pub struct Workspace {
3     ...
4     pub scheme: TiledScheme,
5     pub tiled_tree: Option<TiledTree>,
6 }

```

- `scheme`: 用于指示当前使用的窗口排列方案。
- `tiled_tree`: 存储当前工作区内窗口的具体排布信息，其核心数据结构即为 `TiledTree`。

3.6.5.1 布局方案枚举

为布局方案定义枚举类型，默认跟随鼠标焦点布局方案。

```

rust
1 #[derive(Debug, Clone)]
2 pub enum TiledScheme {
3     Default,
4     Spiral,
5 }

```

3.6.5.2 节点信息设计

rust

```
1 pub enum Direction {
2     Left,
3     Up,
4     Right,
5     Down,
6 }
7
8 pub enum NodeData {
9     Leaf { window: Window },
10    Split {
11        direction: Direction,
12        rec: Rectangle<i32, Logical>,
13        offset: Point<i32, Logical>,
14        left: NodeId,
15        right: NodeId,
16    }
17 }
```

- 内部节点（父节点）：代表一个区域被划分的逻辑，存储以下信息：
 - 分裂方向：上下左右，当窗口新建时，方向被插入窗口的相对位置
 - 窗口的起点位置与总大小
 - 子节点的索引（左子节点与右子节点）
 - offset：内部分裂的偏差值（用于手动更新窗口大小）
- 叶子节点：表示一个具体窗口的存在，只存储窗口 ID（或 Surface ID），不再包含其他布局信息。
 - > 说明：每次添加新窗口时，目标叶子节点会被替换为一个新的父节点，并且规定左子节点处于布局的上方/左侧，其两个子节点分别为原窗口和新窗口的 ID。

3.6.5.3 SlotMap

为提升树结构的动态操作性能，本项目引入了 [Rust 的 SlotMap](<https://docs.rs/slotmap/>) 作为节点存储的底层容器。相比传统引用或 Box 指针，SlotMap 具有以下优势：

- 快速访问：所有节点通过唯一的 Key 标识，可在 O(1) 时间内访问。
- 插入与删除开销小：不影响其他节点位置，避免指针更新或数据重排。
- 避免悬垂指针问题：因为节点通过 key 而非裸指针引用，内存安全性更高。

每个节点在 SlotMap 中都会分配一个唯一的 NodeId，父节点只需保存左右子节点的 NodeId，大大简化了树的管理和操作逻辑。

以下是 TiledTree 的基础定义：

rust

```
1 use slotmap::{new_key_type, SlotMap};
```

```

3 new_key_type! {
4     pub struct NodeId;
5 }
6
7 #[derive(Debug)]
8 pub struct TiledTree {
9     nodes: SlotMap<NodeId, NodeData>,
10    spiral_node: Option<NodeId>,
11    root: Option<NodeId>,
12    neighbor_graph: NeighborGraph,
13
14    gap: i32,
15 }

```

在这个结构中：

- **nodes**: 维护了整个布局树中所有节点的数据。
- **root**: 指向当前布局树的根节点，如果树为空，则为 None。
- **spiral_node**: 与螺旋布局有关，记录螺旋布局的尾部。
- **neighbor_graph**: 全局邻接表。
- **gap**: 样式设置信息，窗口间距。

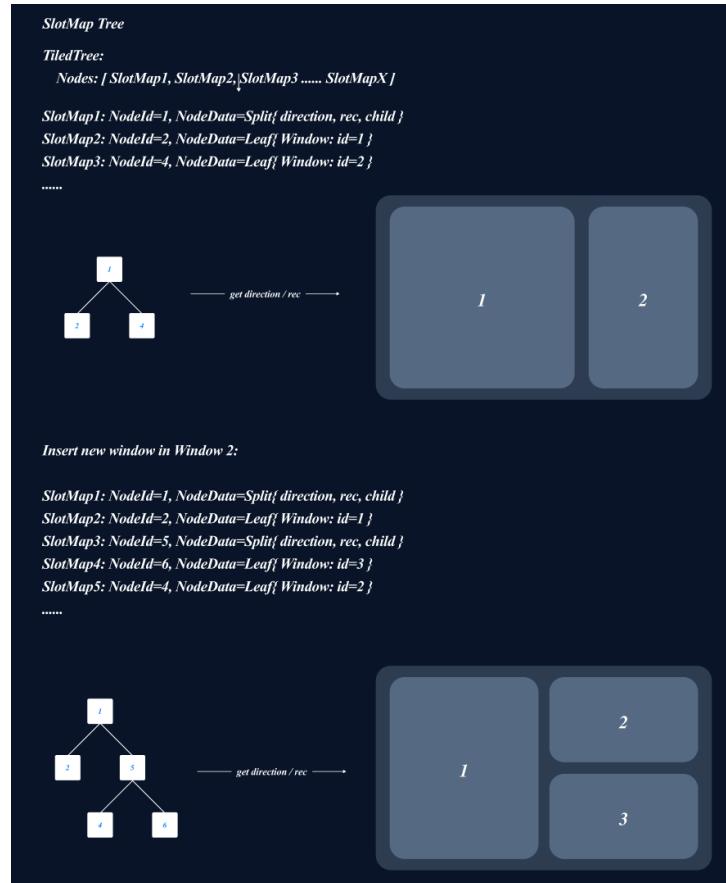


图 14 slotmap 示意图

3.6.5.4 全局窗口邻接表

全局窗口邻接表 用于记录所有窗口与邻居窗口之间的位置关系，表示为 A direction B，使用 HashMap 进行维护。

rust

```
1 #[derive(Debug, Clone)]
2 pub struct NeighborGraph {
3     edges: HashMap<Window, HashMap<Direction, Vec<Window>>>,
4 }
```

全局窗口邻接表 主要完成对新插入窗口，删除窗口，更新窗口后的所有邻接关系的更新与修改。

3.6.5.5 自动布局算法 - 焦点分布 (Focus-Following Mode)

此模式为默认布局策略，所有窗口的插入与删除操作均围绕当前活动窗口 (focus) 展开：

- 插入窗口时：查找当前焦点所在的叶子节点，并将其作为插入位置。该节点将转换为 Split 节点，原窗口与新窗口分别成为左右子节点。
- 删除窗口时：寻找其父节点与兄弟节点，依据兄弟节点类型进行树结构调整（详见删除操作逻辑）。

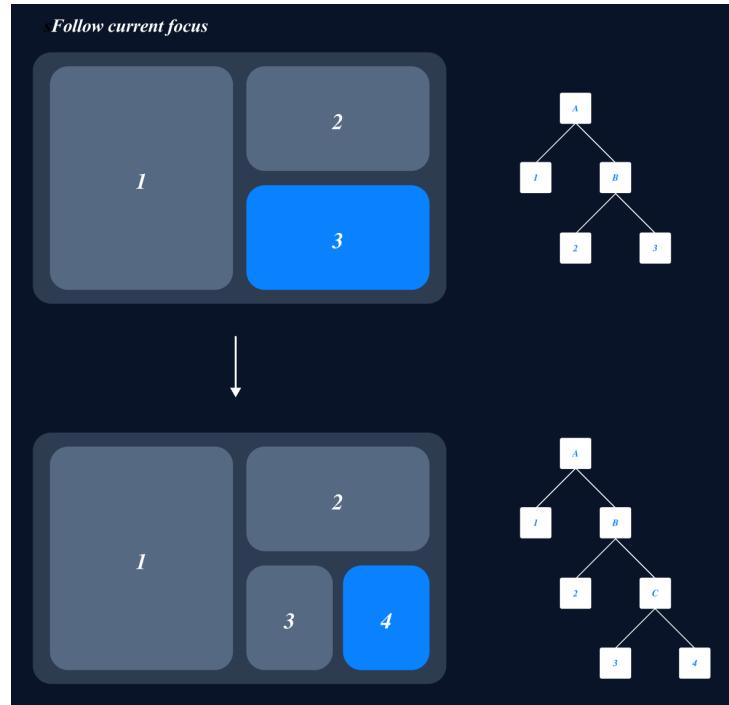


图 15 focus 布局示意图

3.6.5.6 自动布局算法 - 网格分布 (Grid Mode)

此策略试图保持布局树的平衡性，使窗口分布更均匀，避免单边过度嵌套导致的窗口压缩问题：

- 插入窗口时：遍历当前树结构，寻找深度最浅的叶子节点作为插入点，以此保持树结构的对称性与均衡性。
- 删除窗口时：遵循相同的父子结构替换逻辑，但在后续窗口重排时尽可能维持已有平衡性。

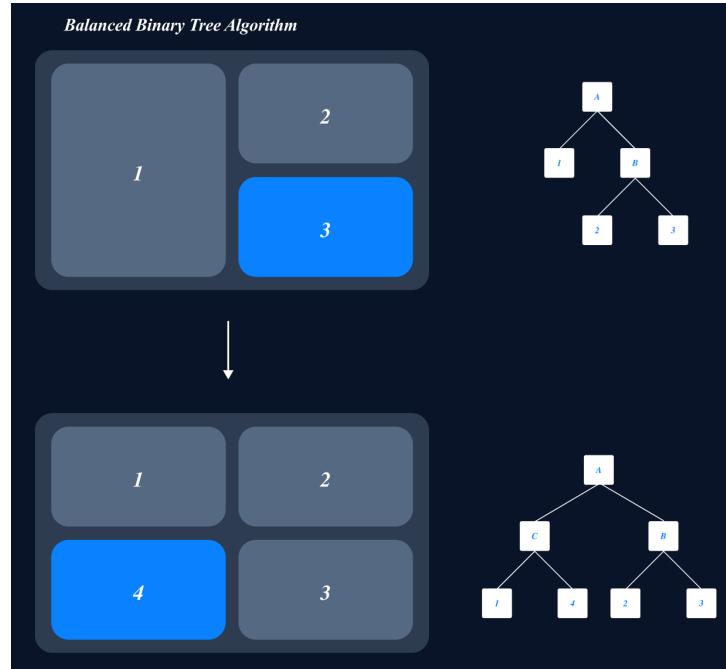


图 16 grid 布局示意图

3.6.5.7 自动布局算法 - 螺旋分布 (Sprial Mode)

此策略试图实现螺旋状的窗口分布，以左侧为起始，实现动态美观的布局效果。

- 插入窗口时：记录的 `sprial_node` 为插入节点，插入方向为右下左上轮换，按照当前窗口的数量计算得到。
- 删除窗口时：若删除窗口为 `sprial_node` 则设置其兄弟节点为新的 `sprial_node`。

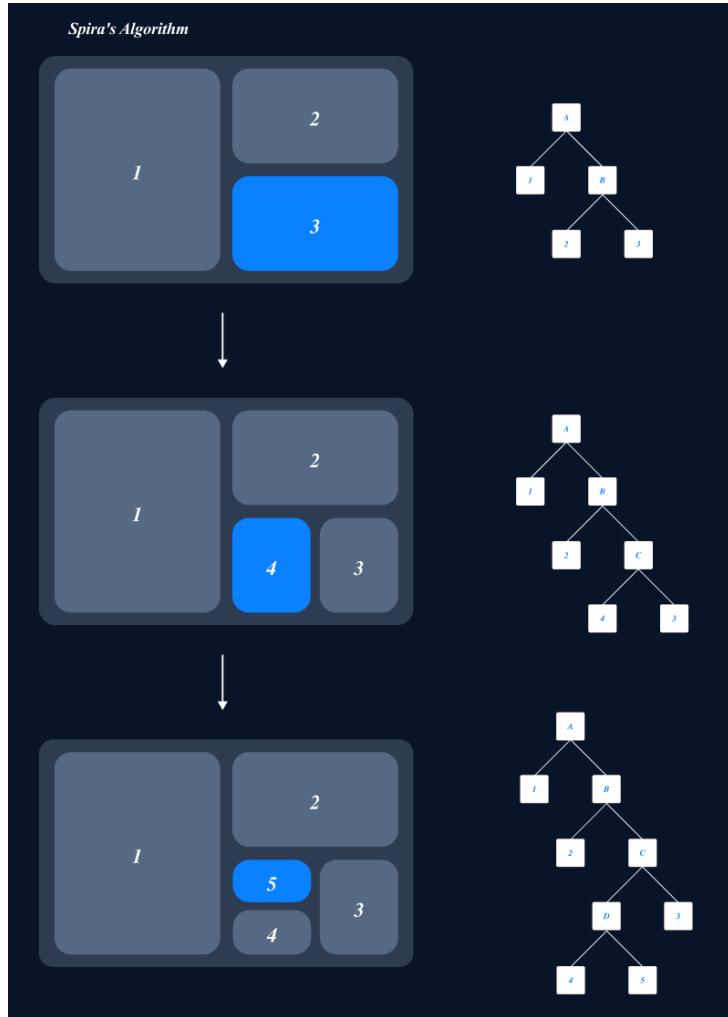


图 17 spriral 布局示意图

3.6.5.8 布局树的基本操作 - 插入窗口

窗口插入遵循当前布局策略，分为以下三个步骤：

1. 确定被插入窗口与插入方向
2. 计算与设置被插入窗口与新插入窗口的大小与位置
3. 修改邻接表

workspace 根据布局策略，给定被插入窗口与插入方向，`insert_window()` 函数会完成计算与更新修改，这里的设计非常符合直觉。

3.6.5.9 布局树的基本操作 - 删 除窗口

窗口删除操作包含以下四个核心步骤：

- 查找关联节点：通过辅助函数 `find_parent_and_sibling` 定位目标窗口的父节点及其兄弟节点。
- 结构调整与继承布局：
 - 若兄弟节点为 Leaf，则继承父节点的 `rec` 并替代父节点位置；

- ▶ 若兄弟节点为 Split，则同样继承 rec，替代父节点后调用 modify 递归更新其子节点的布局信息。
- 清理节点数据：从 SlotMap 中移除被删除的窗口节点，保持结构整洁。
- 更新邻接表

3.6.5.10 布局树的基本操作 - 倒置窗口

倒置操作主要将 Split 类型节点的 direction 参数倒置，视觉效果上为水平变竖直，此操作会导致 rec 的变化，因此还需要更新所有子节点信息。

主要分为以下三步：

- 找到需要倒置的窗口的父节点。
- 倒置 direction 并且使用 modify 递归更新当前父元素为根的树。
- 修改邻接表。

3.7 动画效果实现

动画效果

本吸纳纲目在保持“平铺核心逻辑简洁高效”的前提下，适度引入了 过渡动画机制，用于提升用户在窗口焦点切换、窗口布局变换、标签页切换等场景下的感知连贯性。动画不仅是美学设计的体现，更是信息传递与视觉引导的有效方式。

为了保证动画系统的性能和可控性，*Mondrian* 采用了如下设计原则：

- **最小依赖**：动画系统直接构建在现有渲染框架之上，无引入额外 GUI 框架；
- **状态驱动**：所有动画过渡均由窗口状态变化触发，避免无效重绘；
- **可扩展性强**：动画接口设计后续可插拔，支持不同类型的动画模块（例如弹性缓动、贝塞尔插值等）。

为实现动画效果，*Mondrian* 并不直接修改窗口的最终状态数据，而是采用一种“状态解耦、渲染驱动”的设计模式：

在触发需要动画的操作（如窗口移动、布局切换）时，实际的数据状态立即更新为目标值；

然而，在渲染阶段，窗口的位置与属性并非立刻反映为最终状态，而是通过插值计算出一个中间状态，并随着时间推进逐帧更新，直到过渡完成。

这种做法带来两个显著优势：

1. **逻辑状态与渲染状态分离**：窗口管理逻辑保持简洁，不需要等待动画完成即可进行后续操作；
2. **动画过程可中断、可复用**：新的动画触发可以自然地替换旧的过渡轨迹，增强响应性与一致性。

例如，在窗口移动动画中，我们为每个窗口维护一个 `current_rect`（当前渲染位置）和 `target_rect`（逻辑目标位置），渲染时以时间为参数进行插值过渡，而不是一次性跳转。

3.7.1 Animation 结构体封装

为了实现动画效果，使用 `Animation` 结构体封装所需内容：

```
rust
1 pub struct Animation {
2     from: Rectangle<i32, Logical>,
3     to: Rectangle<i32, Logical>,
4     elapsed: Duration,
5     duration: Duration,
6     animation_type: AnimationType,
7     pub state: AnimationState,
8 }
9
10 pub enum AnimationState {
```

```

11     NotStarted,
12     Running,
13     Completed,
14 }
15
16 pub enum AnimationType {
17     Linear,
18     EaseInOutQuad,
19     OvershootBounce,
20 }

```

在需要触发动画的时刻，将所需内容进行 Animation 被包裹，由 eventloop 异步触发，在每一帧渲染时进行过程状态处理，即可实现动画效果。

rust

```

1 impl Animation {
2     pub fn new(
3         from: Rectangle<i32, Logical>,
4         to: Rectangle<i32, Logical>,
5         duration: Duration,
6         animation_type: AnimationType,
7     ) -> Self {
8         Self {
9             from,
10            to,
11            elapsed: Duration::ZERO,
12            duration,
13            animation_type,
14            state: AnimationState::new(),
15        }
16    }
17
18    pub fn start(&mut self) -> Rectangle<i32, Logical> {
19        self.elapsed = Duration::ZERO;
20        self.state = AnimationState::Running;
21        self.from
22    }
23
24    pub fn tick(&mut self) {
25        self.elapsed += Duration::from_millis(1);
26        if self.elapsed >= self.duration {
27            self.state = AnimationState::Completed;
28        }
29    }
30
31    pub fn current_value(&self) -> Rectangle<i32, Logical> {
32        let progress = (self.elapsed.as_secs_f64() /
33            self.duration.as_secs_f64()).clamp(0.0, 1.0);
34        process_rec(
35            self.from,
36            self.to,
37            self.animation_type.get_progress(progress),
38        )
39    }

```

3.7.2 渲染请求

在窗口渲染阶段，系统首先判断该窗口是否处于动画过程中，即其 AnimationState 是否为 Running：

- 若动画已在进行中，则根据当前时间计算本帧对应的 中间位置与尺寸，用于绘制；
- 若动画尚未启动 (AnimationState == Pending)，则立即初始化动画状态，并将其标记为 Running，以便从下一帧开始正式执行过渡；
- 若动画已经完成，则会在 refresh() 函数下移出动画队列。
- 若动画未绑定动画对象，则直接使用窗口的最终几何状态进行渲染。

通过这种机制，Mondrian 能够在不影响窗口逻辑更新的前提下，实现 按需触发、逐帧驱动 的动画渲染流程，有效提升了动态响应的流畅性和可控性。

rust

```
1 pub struct RenderManager {
2     // no need now
3     start_time: Instant,
4     animations: HashMap<Window, Animation>,
5 }
6
7 ...
8 for window in workspace_manager.elements() {
9     let location = match self.animations.get_mut(window) {
10         Some(animation) => {
11             match animation.state {
12                 AnimationState::NotStarted => {
13                     let rec = animation.start();
14                     window.set_rec(rec.size);
15                     rec.loc
16                 }
17                 AnimationState::Running => {
18                     animation.tick();
19                     let rec = animation.current_value();
20                     window.set_rec(rec.size);
21                     // info!("{}:{?}{:?}", rec);
22                     rec.loc
23                 }
24                 _ => break,
25             }
26         }
27     None => {
28         let window_rec = workspace_manager.window_geometry(window).unwrap();
29         window_rec.loc
30     }
31 };
32 ...
33
34 pub fn refresh(&mut self) {
35     // clean dead animations
```

```
36     self.animations
37         .retain(|_, animation| !matches!(animation.state,
38             AnimationState::Completed));
38 }
```

3.7.3 实现案例

在触发 窗口插入动画 时，操作会涉及到平铺布局树（即二叉树结构）的结构调整。此过程中，我们能够获取：

- 被调整窗口 的旧位置与新位置 (`from → to`)
- 新插入窗口 的初始几何信息（通常为最小化或透明状态）与目标位置

对于这些窗口，我们将其对应的几何变换信息封装为一个 `Animation` 对象，并统一加入到一个 动画任务队列 中，由事件循环（`eventloop`）逐帧驱动执行。

每一帧中，事件循环会对当前活跃的动画组执行插值更新，通过绘制中间状态实现平滑过渡，直到所有动画到达终点并完成移除。

这种方式实现了：

- 解耦逻辑结构修改与渲染过程
- 支持并发多个窗口动画协同
- 为后续扩展缓动函数、过渡风格等提供良好接口

```
rust
1 ...
2 loop_handle.insert_idle(move |data| {
3     data.render_manager.add_animation(
4         target,
5         rec,
6         original_rec,
7         Duration::from_millis(30),
8         crate::animation::AnimationType::EaseInOutQuad,
9     );
10    let mut from = new_rec;
11    match direction {
12        Direction::Right => {
13            from.loc.x += from.size.w;
14        }
15        Direction::Left => {
16            from.loc.x -= from.size.w;
17        }
18        Direction::Up => {
19            from.loc.y -= from.size.h;
20        }
21        Direction::Down => {
22            from.loc.y += from.size.h;
23        }
24    }
25}
```

```
24    }
25    data.render_manager.add_animation(
26        new_window,
27        from,
28        new_rec,
29        Duration::from_millis(30),
30        crate::animation::AnimationType::OvershootBounce,
31    );
32 });
33 ...
```

3.8 拓展协议实现

3.8.1 Layer-Shell 支持

为了实现桌面组件如状态栏、启动器、壁纸容器等持久性窗口，Mondrian 支持 *wlr-layer-shell* 协议。这一协议最初由 wlroots 提出，是实现 Wayland 桌面环境中“系统层级窗口”的标准手段。

layer-shell 允许客户端以特定“图层”（layer）方式向合成器注册自身位置、对齐方式、屏幕边缘锚定，以及交互区域排除等属性，用于构建如：

- 层叠浮动窗口（layer: overlay）
- 顶部面板 / 状态栏（layer: top）
- 底部 dock / launcher（layer: bottom）
- 桌面壁纸容器（layer: background）

Mondrian 对该协议的支持不仅满足了桌面组件的功能需求，还为未来实现更多系统级 UI（如通知气泡、任务视图等）提供了良好的基础。

smithay 中对 *layer-shell* 协议有较好的支持，只需要引用并且构造所需内容即可。

rust

```
1 use smithay::{
2     delegate_layer_shell,
3     desktop::{LayerSurface, WindowSurfaceType, layer_map_for_output},
4     output::Output,
5     reexports::wayland_server::protocol::wl_surface::WlSurface,
6     wayland::{
7         compositor::with_states,
8         shell::wlr_layer::{LayerSurfaceData, WlrLayerShellHandler,
9             WlrLayerShellState},
10    },
11
12 use crate::state::GlobalData;
13
14 impl WlrLayerShellHandler for GlobalData {
15     fn shell_state(&mut self) -> &mut WlrLayerShellState {
16         &mut self.state.layer_shell_state
17     }
18
19     fn new_layer_surface(
20         &mut self,
21         surface: smithay::wayland::shell::wlr_layer::LayerSurface,
22         wl_output:
23             Option<smithay::reexports::wayland_server::protocol::wl_output::WlOutput>,
24             _layer: smithay::wayland::shell::wlr_layer::Layer,
25             namespace: String,
26     ) {
27         let output = if let Some(wl_output) = &wl_output {
28             Output::from_resource(wl_output)
29         } else {
```

```

29         // TODO: output_manager -> Option<Output>
30         Some(self.output_manager.current_output().clone())
31     };
32
33     let Some(output) = output else {
34         warn!("no output for new layer surface, closing");
35         surface.send_close();
36         return;
37     };
38
39     let mut map = layer_map_for_output(&output);
40     map.map_layer(&LayerSurface::new(surface, namespace))
41         .unwrap();
42 }
43
44 fn layer_destroyed(&mut self, surface:
    smithay::wayland::shell::wlr_layer::LayerSurface) {
45     // TODO: outputs
46     let map = layer_map_for_output(self.output_manager.current_output());
47     let layer = map
48         .layers()
49         .find(|&layer| layer.layer_surface() == &surface)
50         .cloned();
51     let (mut map, layer) = layer.map(|layer| (map, layer)).unwrap();
52     map.unmap_layer(&layer);
53 }
54
55 fn new_popup(
56     &mut self,
57     _parent: smithay::wayland::shell::wlr_layer::LayerSurface,
58     popup: smithay::wayland::shell::xdg::PopupSurface,
59 ) {
60     self.unconstrain_popup(&popup);
61 }
62 }
63 delegate_layer_shell!(GlobalData);
64
65 impl GlobalData {
66     pub fn layer_shell_handle_commit(&mut self, surface: &WlSurface) -> bool {
67         let output = self.output_manager.current_output();
68
69         let mut map = layer_map_for_output(output);
70
71         if map
72             .layer_for_surface(surface, WindowSurfaceType::TOPLEVEL)
73             .is_some()
74         {
75             let initial_configure_sent = with_states(surface, |states| {
76                 states
77                     .data_map
78                     .get::<LayerSurfaceData>()
79                     .unwrap()
80                     .lock()
81                     .unwrap()
82                     .initial_configure_sent
83             });

```

```
84
85     map.arrange();
86     if !initial_configure_sent {
87         let layer = map
88             .layer_for_surface(surface, WindowSurfaceType::TOPLEVEL)
89             .unwrap();
90
91         layer.layer_surface().send_configure();
92     }
93
94     return true;
95 }
96
97     false
98 }
99 }
```

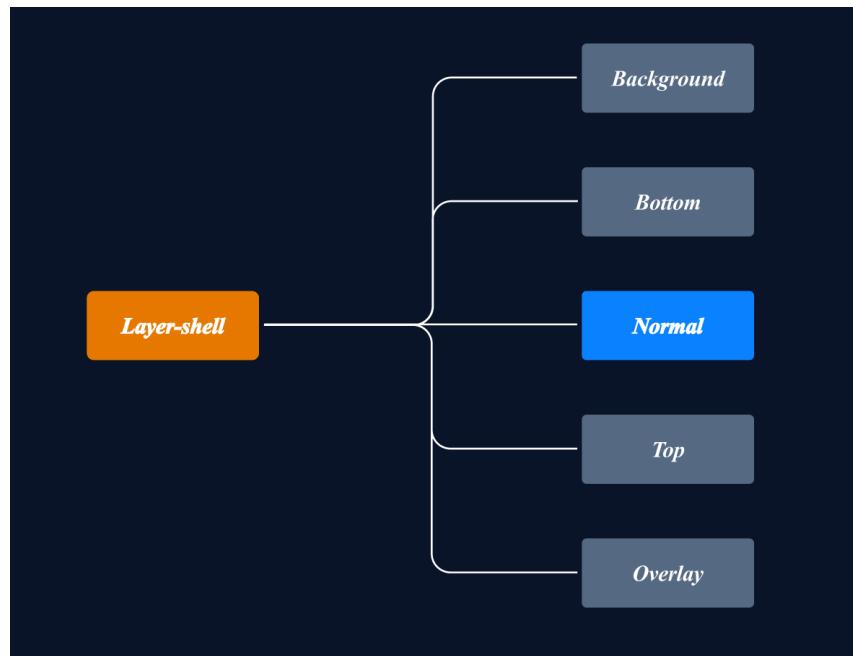


图 18 layer_shell 协议示意图

4 性能测试与分析(仍在优化，决赛期间完成)

4.1 Rust Tracy 跟踪分析

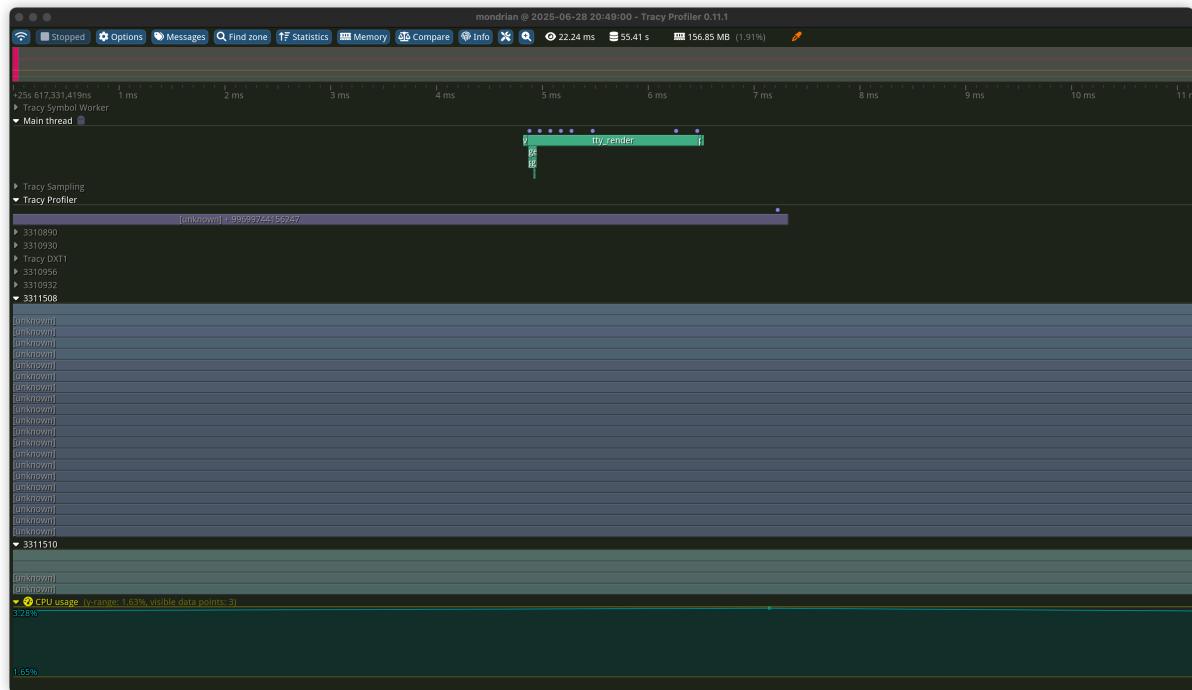


图 19 tty 后端 tracy 跟踪图

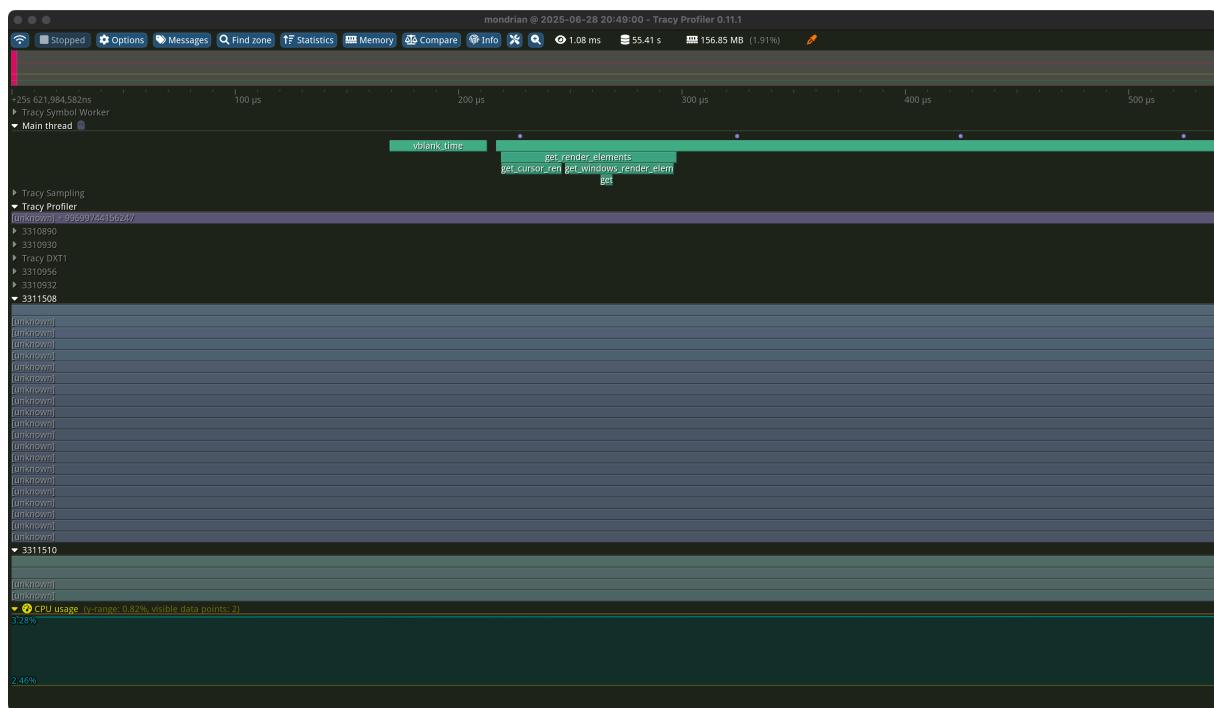


图 20 tty 后端 tracy 跟踪图

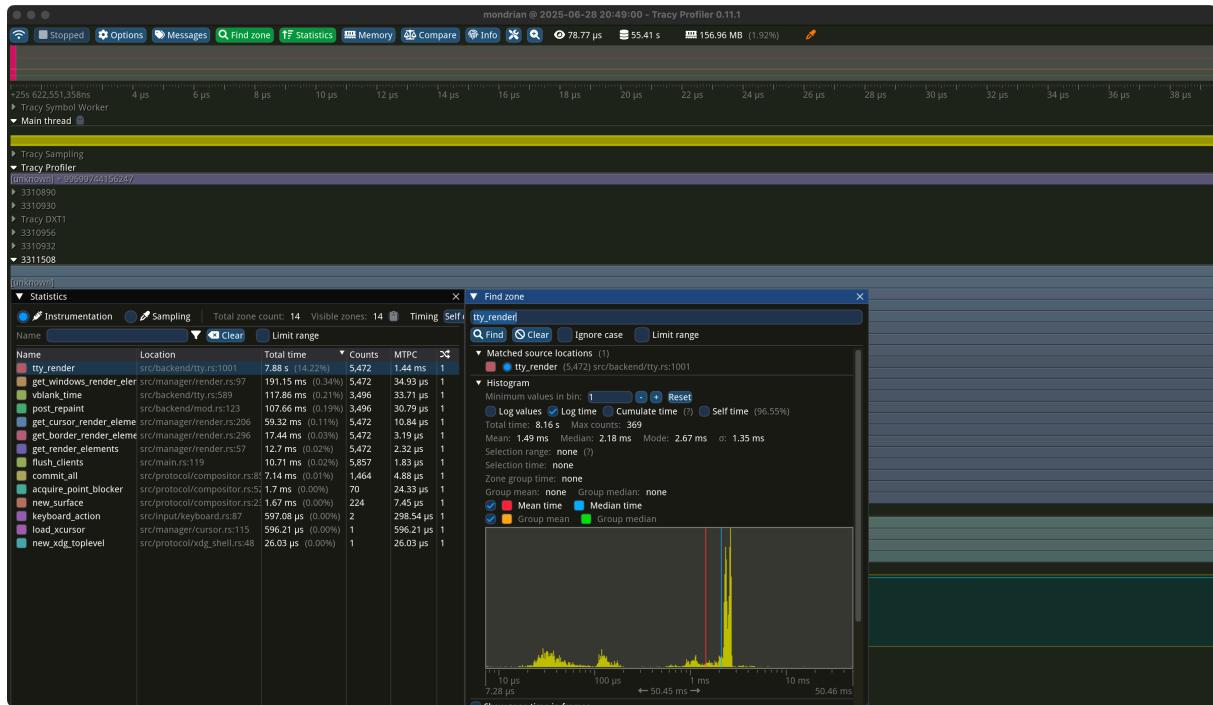


图 21 tty 后端 tracy 跟踪图

4.2 帧率性能测试



图 22 tty 后端 GPU 帧率测试

4.3 GPU 压力测试

5 项目总结

本项目基于 Rust 语言与 Smithay 框架，自主实现了一个完整的 Wayland 合成器，具备显示服务器与窗口管理器的双重功能。通过底层 DRM/KMS 图形接口实现原生渲染管线，支持离屏绘制与缓冲区交换；在输入管理、窗口调度、协议兼容等方面构建了高度模块化的系统架构。

项目采用自定义的平铺式窗口管理算法，支持键盘驱动的高效交互模式，兼顾性能、美学与个性化配置；同时，已成功兼容多种 layer-shell 客户端，具备构建完整桌面环境的基础能力。

在保持稳定运行的基础上，本项目充分体现了现代合成器的核心特性——灵活、可拓展、安全、高效，为探索下一代 Linux 桌面提供了可行路径。后续将在多显示器支持、输入扩展、XWayland 兼容等方向持续推进，朝着高度可定制化与完整生态支持的目标不断完善。