Doro 爱吃橘子队-Tiley 技术文档

Tiley 以“简洁、高效、可扩展”的理念，从底层到高层构建了一套完整的 Wayland 平铺式窗口管理器(wlroots 库）。下面从整体架构、主要功能模块以及模块间协作三方面，描述其实现思路与功能要点。(都是现阶段已经实现的功能)

文档以以下内容展开：

一、架构概览

核心依赖

Wayland + wlroots：负责底层协议栈、渲染抽象、输出与输入管理。

C++11 单例与智能指针：管理全局状态，确保线程安全与自动析构。

STB Image、inotify：分别用于壁纸加载与配置文件热重载。

主要组件

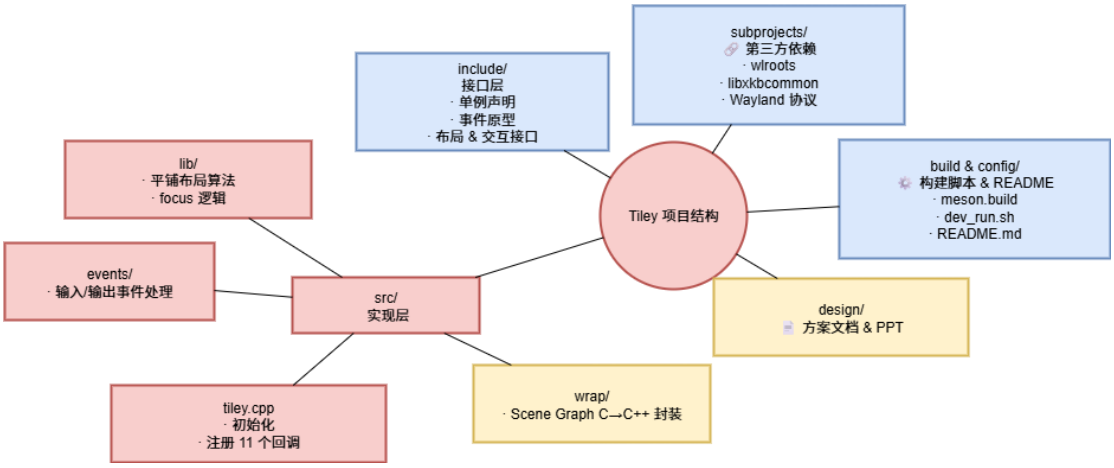
TileyServer:全局单例,持有 Wayland Display、Backend、Renderer、Allocator、Scene Graph、Cursor、Seat、输出与输入事件监听器链表。

WindowStateManager：平铺布局管理器，维护每个工作区的二叉树 area_container，提供插入、删除、重排（reflow）等接口。

cene Graph 封装：C→C++ wrapper，将所有可视元素（壁纸、窗口、装饰、弹窗）组织到同一棵树中，统一渲染与命中测试。

快捷键模块：基于 JSON 配置与 inotify 热加载，实现自定义组合键映射到动作的扩展框架。

附简单目录结构图：



附项目整体架构详细版图：

二、核心功能模块

1 平铺布局（Tiling）

二叉树分区：每个工作区根节点初始为空桌面；新窗口插入时定位到目标叶子，按水平或垂直对半分割，维护双向指针；删除时将兄弟节点提升，保持树结构连贯。

重排（Reflow）：在任何插入 / 删除 / 拖拽完成后，递归地根据分割标记将屏幕空间划分给每个窗口，并通过 WLRoots API 向客户端下发新的大小与位置。

槽位拖拽：拖动窗口时不自由移动坐标，而是在已分割槽位间移动：每次鼠标落点对应的叶子不同，就执行一次“删除+插入+重排”，实现“直观可控”的平铺

移动。

2 渲染与命中测试

Scene Graph：所有元素（壁纸、窗口内容、浮动装饰、弹窗）均作为节点插入同一棵场景树，渲染时调用 wlr_scene_output_commit 一次性提交。

输入路由：鼠标每次移动触发 wlr_scene_node_at 命中测试，定位到最前端的缓冲区节点，再回溯到对应的 surface_toplevel，实现“坐标→窗口”精确映射。

多输出与工作区映射

动态映射：display_to_workspace_map 将每块物理输出与某个工作区编号绑定；新屏幕接入、拔出或用户命令均可改变此映射。

分区作用域：每块输出只渲染对应工作区的窗口树，切换映射后自动重排并显示新的窗口集合。

3 焦点管理与链表切换

双向链表：所有激活窗口在 server.toplevels 链表中维护顺序；focus_toplevel 在场景图和链表中同时提升目标窗口；focus_next_window 遍历链表实现循环切换。

键盘与鼠标聚焦：点击窗口自动激活并将键盘、光标输入路由到该窗口；释放拖拽重置状态。

4 壁纸与浮动装饰

壁纸：启动或新输出时加载图像到 SHM 缓冲区，注入场景图底层，提供原始与等比缩放模式；输出大小变化时可重新生成。

浮动边框：在“浮动”或“调整大小”模式下，为目标窗口动态创建四条高亮矩形边框，显示可拖拽/缩放区域，完成后隐藏或销毁。

5 自定义快捷键

配置驱动：JSON 文件定义“组合键→动作”映射，inotify 后台线程监控文件修改，实时重载映射表。

事件拦截：在 keyboard_handle_key 回调最前端生成标准化字符串，查表命中时执行对应动作（如切换窗口、启动终端、关闭窗口），并拦截，不再透传给客户端。

三、模块协作与启动流程

初始化

main() 中调用 wl_display_create 、 wlr_backend_autocreate 、 wlr_renderer_autocreate、wlr_allocator_autocreate;

创建 server.scene 并绑定 output_layout;

创建光标与 Seat，注册所有 11 条核心回调。

壁纸注入

在每个 new_output 回调内，先调用 create_wallpaper_buffer_scaled，再将其作为最底层 Scene Buffer 注入对应输出。

平铺管理

第一扇窗口在 new_toplevel 时插入根节点；

后续窗口按鼠标落点分割插入；
在 commit 或删除后调用 reflow 完成布局。

交互与输入

鼠标移动 / 点击等事件由光标回调分发至命中测试、拖拽或插件化处理；
键盘事件在拦截快捷键后可传给客户端，确保高优先级组合键。

渲染与退出

每个输出的 frame 回调统一调用 wlr_scene_output_commit；
程序退出时反注册所有监听器，依次销毁 Scene Graph、光标、渲染器、Backend 与 Display。
通过上述整体实现与功能模块的紧密协作，Tiley 构建了一个具有平铺、浮动、高度可配置与多输出支持的现代 Wayland 窗口管理器，兼顾性能、稳定性与可扩展性。

四．对比其它管理器：

Sway

技术栈

C 语言、基于 wlroots。
配置文件采用纯文本，使用类似 i3 的指令式语法。

优点

成熟稳定：社区活跃、文档齐全、插件生态完善。
兼容 i3：对熟悉 i3 的用户几乎零学习成本。
热重载配置：多数配置项可在运行时动态更新，无需重启 compositor。

局限

扩展性：大部分核心功能写在 C 里，插件接口较为有限；想添加新布局或自定义算法，需要修改源码并重新编译。
布局算法：默认仅提供单一的二叉树平铺，对多分区或复杂切分比例支持有限。

与 Tiley 对比

Tiley 采用 C++11 和单例+智能指针管理，全局状态更易维护。
提供模块化的 Scene Graph 封装、动态热加载快捷键、壁纸和装饰框等现代特性；可插拔式算法设计让未来更容易替换或扩展布局逻辑。

bspwm

技术栈

C 语言，使用 XCB 与 X11 协议。
通过 shell 脚本和外部工具（如 bspc）进行控制与配置。

优点

纯平铺：强烈的二叉空间分割模型，支持灵活的窗口树操作。
脚本驱动：将大多数决策逻辑放在外部脚本中，极度可定制。

局限

老平台：基于 X11，不适应新一代 Wayland 的生态；性能和扩展性受限。
重复造轮子：需要大量 shell 逻辑来实现现代 compositor 的核心功能（输入

路由、渲染、帧同步等）。

与 Tiley 对比

Tiley 完全基于 Wayland，为现代 Linux 桌面原生设计。

内建输入与渲染管理、Scene Graph、直接在 compositor 代码层实现布局算法，无需外部脚本桥接，减少延迟与故障点。

Hyprland

技术栈

C++17，基于 wlroots；广泛使用现代 C++ 特性与模板。

配置文件使用独特的键值对语法，支持动态热加载。

优点

灵活布局：支持比例分割、标签式、动态工作区以及混合浮动/平铺模式。

丰富特性：内置动画、模糊、截屏等多媒体效果；插件生态快速增长。

高性能：代码高度优化，对 GPU 渲染与输入延迟做了细致调优。

局限

复杂度高：功能众多但也带来较大代码量和学习曲线。

依赖较多：需要额外的图形库（例如 cairo、glm）来支持动画和特效。

与 Tiley 对比

Tiley 专注于“轻量级平铺”，核心只依赖 wlroots 和少量第三方库（STB、nlohmann/json），上手更简单。

算法与模块以最精简形式实现，适合对性能与定制性有极高要求的场景；未来可按需引入特效或动画插件。

五：设计思路

Tiley 的设计源自对 TinyWL 极简示例的深入拆解与强化扩展，遵循“从最简可运行到模块化可插拔”的演进路径。整个系统被划分为四大相互独立、职责单一的层次，各层之间通过清晰的接口契约进行通信，既保留了 TinyWL 对协议细节的精简掌控，又实现了针对真实场景的丰富功能扩展。

协议层

职责：直接与 Wayland 核心协议及 wlroots 封装交互，负责创建 wl_display、wlr_backend、wlr_renderer、wlr_allocator、wlr_seat 等底层对象。

设计要点：

使用 TinyWL 中的参数处理与日志初始化方式，确保对 Wayland 事件循环的最**简接口；**

所有对 wlroots C API 的调用都被限制在此层或其封装的 wrapper 模块中，上层不直接包含原生 C 头文件。

可替换性：未来若切换到 libweston 或其他 Wayland 库，只需重写本层少数工厂函数，无需触及布局与交互逻辑。
渲染层（Scene Graph 封装）
职责：将 Surface、Popup、壁纸、装饰边框等可视元素组织成单一的树形结构，统一管理渲染提交与命中测试（hit-testing）。
设计要点：在 src/wrap/ 提供对 wlr_scene_* 系列 C 函数的轻量级包装，消除 extern "C" 与 C++ 头文件隔阂；
所有可视节点由统一的 SceneManager（即 server.scene）调度在每个输出的 frame 回调中一次性提交；
鼠标落点通过 wlr_scene_node_at 映射到缓冲区节点，再由封装函数回溯到业务层的 surface_toplevel，避免手工计算坐标冲突。
布局层（平铺算法）
职责：维护每个工作区的 area_container 二叉树，并提供插入、删除、定位、重排（reflow）接口，实现平铺式空间分割与重组。
设计要点：
将所有树操作封装在 WindowStateManager，包括 insert、remove、desktop_container_at、reflow；
当前实现采用对半切分（水平/垂直），并在拖拽模式下以“移除+插入”形式切换槽位；
引入双向链表与父指针维护，确保树结构在动态操作中始终保持一致而无孤立节点。
可替换性：任何其他 C++ 平铺或网格布局算法，只需实现同样的接口即可插拔，例如三分网格、KD-Tree 分区、用户自定义比例切分等。
交互层（输入与扩展）
职责：处理所有用户输入与自定义扩展，包括光标/键盘事件分发、快捷键模块、浮动装饰、壁纸热重载等。
设计要点：
事件分层：11 个核心回调仅做单一职责，不跨层调用；如鼠标移动只关心输入与命中测试，布局层不参与事件绑定细节；
快捷键框架：后台 inotify 热加载 JSON 配置，通过统一 keyboard_handle_key 接口查表并执行动作，无侵入式地植入到输入流；
装饰与壁纸作为可选插件：默认不开启，可在配置中按需启用；它们通过渲染层接口插入场景图，不影响平铺算法核心。
可替换性：未来可将 JSON 换成 Lua、TOML 或数据库，或扩充快捷键动作集合；也可替换装饰模块为更复杂的动画效果插件。

6 各个模块算法详细

6.1：11 个启动事件

在 Tiley 的启动阶段，共有 11 类核心回调。下面按逻辑依赖的顺序，用文字说明它们的触发时机、职责，以及为什么要这样设计。

1 新输出设备接入 (Backend new_output)

触发时机：操作系统或用户插入 / 启用新的显示器时。

主要职责：

为该输出创建一个 output_display 管理对象，绑定到渲染器和缓冲分配器。
将此输出添加到全局的输出布局（OutputLayout）中，并插入到 server.outputs 的双向链表。
注册该输出的三类事件：帧同步（frame）、状态改变（request_state）、销毁（destroy），以便后续渲染、布局调整和资源清理。

2 新顶级窗口创建 (XDG Shell new_toplevel)

触发时机：客户端通过 XDG Shell 协议请求打开一个新窗口。

主要职责：

分配并初始化一个 surface_toplevel 对象。
在场景树（Scene Graph）中为它创建一颗子树，并挂载到主场景。
调用布局管理器，基于当前鼠标位置和分割策略，为它分配或插入一个 area_container。
注册与该窗口相关的生命周期和交互事件

3 新弹出窗口创建 (XDG Shell new_popup)

触发时机：客户端创建临时弹出界面（菜单、工具提示、弹出对话框等）。

主要职责：

分配一个简化版的窗口管理结构 surface_popup。
在其父窗口对应的场景子树下，创建 popup 对应的 scene 节点。
注册 commit（初始渲染提交）和 destroy（关闭）事件，用以在弹出内容出现和消失时更新场景。

4 光标相对移动 (Cursor motion)

触发时机：用户通过鼠标或触控板产生的相对位移。

主要职责：

将光标在内部坐标系中移动指定偏移。
根据当前的 cursor_mode（透传 / 拖动 / 缩放）分发到对应的处理流程：正常模式下进行命中测试并转发给客户端；拖动模式则调用平铺专用的槽位切换逻辑；缩放模式则触发调整窗口大小的算法。

5 光标绝对移动 (Cursor motion_absolute)

触发时机：设备（如数位板、某些手势设备）报告绝对坐标。

主要职责：

将光标直接指定到归一化位置对应的实际屏幕坐标。
同样调用相对移动后的统一处理接口，保持后续逻辑一致。

6 光标按键按下 / 释放 (Cursor button)

触发时机：鼠标按钮发生按下或释放操作。

主要职责：

当前焦点客户端发送按钮事件（按下或释放）。
如果是按下事件且位于某窗口之上，调用聚焦逻辑将其置于最前；
如果是释放事件，并且之前处于拖动或缩放模式，则重置为正常模式。
将点击和拖拽状态切换集中到一个回调中处理，可保证按下和释放的状态机一致性。

7 光标滚轮滚动 (Cursor axis)

触发时机：鼠标滚轮滚动（垂直或水平）。

主要职责：

将滚动方向、滚动幅度和来源等信息，精确地封装后转发给当前焦点客户端，实

现滚动内容（如网页、文本列表）的交互。
设计合理性：将所有滚轮逻辑集中处理，方便后续添加滚动加速、自然滚动等可选特性。
8 光标帧同步 (Cursor frame)
触发时机：Wayland 定时器触发的“frame”事件，用于同步一系列高频输入。
主要职责：
在一次 frame 周期内，将累计的相对移动和滚轮事件一次性批量处理，减少中间抖动和重复回调。
最后调用 wlr_seat_pointer_notify_frame,通知客户端这一帧的所有 pointer 事件已发送完毕。。
9 新输入设备接入 (Backend new_input)
触发时机：系统检测到新的物理输入设备（键盘、鼠标、触控板等）可用。
主要职责：
区分设备类型：
键盘：创建 input_keyboard，为其注册 modifiers（修饰键）、key（按键）、destroy（拔出）等回调，并将其添加到 wl_list keyboards。**指针**：将设备绑定到已有的光标对象，使光标响应该设备的移动和点击。
根据当前已连接的键盘数量，更新 seat 的能力集合（keyboard/pointer）。
只有在检测到设备后再注册回调，可避免无用资源注册；同时实现动态热插拔。
10 客户端请求更改光标形状 (Seat request_set_cursor)
触发时机:当前聚焦客户端通过协议请求改变指针图标(例如在文本框、拖拽时)。
主要职责：
验证请求者是否为当前焦点窗口，只有合规请求才能生效。
调用 wlr_cursor_set_surface 将自定义表面绑定到光标，实现光标图标切换。
设计合理性：仅允许焦点窗口更改光标，防止后台或恶意窗口干扰用户指针。
11 客户端请求设置选区 (Seat request_set_selection)
触发时机：客户端希望将数据放入剪贴板或改变选区（例如文本复制）。
主要职责：
通过 wlr_seat_set_selection 将数据源注册到全局剪贴板，使其他应用能够获取。：将剪贴板管理交给 seat，符合 Wayland 协议设计，将所有 selection 请求集中处理。
整体设计合理性
依赖链清晰：输出必须先就绪，才能渲染窗口和光标；窗口必须创建后，才能响应交互；输入设备注册在光标处理之前；Seat 请求总是在所有核心组件初始化完毕后才允许。
职责单一:整体的设计充分解耦,每个回调只做与自身事件类型高度相关的工作，不会“跨级”操作其他逻辑。
性能与可维护兼顾：Frame 同步、批量处理高频事件；双向链表管理所有对象；状态机集中管理拖动 / 缩放；易于阅读和测试。
这样一套完整的启动与事件注册流程，为后续平铺布局、输入交互和渲染逻辑奠定了坚实基础。

6.2 平铺式算法：

Tiley 的平铺式布局基于一棵简单却功能强大的二叉树结构，使用高度可维护的“插入－删除－重排”三步走策略，将屏幕空间智能地分割、合并与重绘。下面结合你的思路草图，用专业的术语与详实的流程把这套算法的每个细节都梳理清楚。

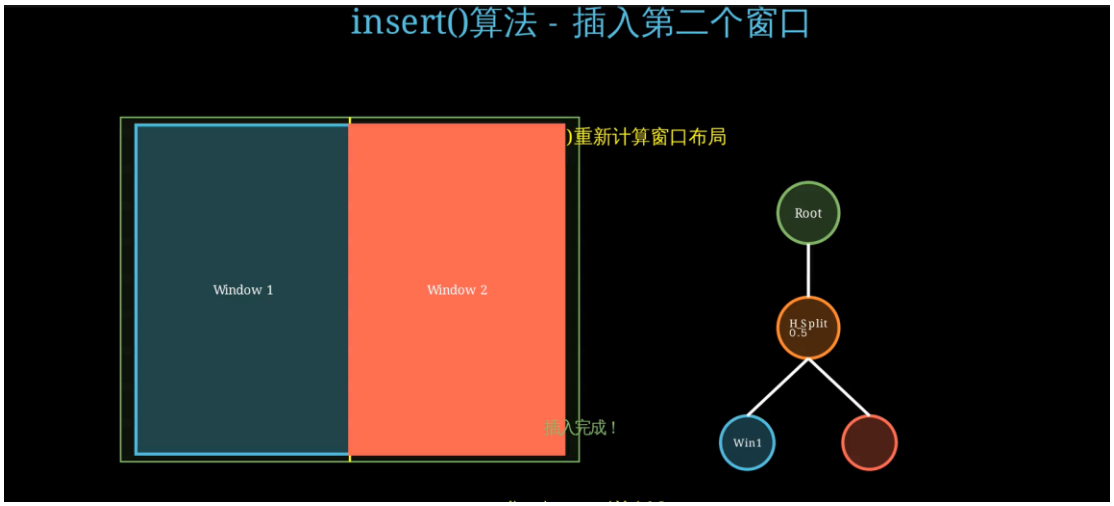
二叉树节点与初始状态

节点定义：每个 area_container 节点记录了
split：当前节点的分割方式（SPLIT_NONE、水平或垂直，通过 enum 记录）。
child1 与 child2：两条子分支，代表屏幕一区与二区。
toplevel：仅当 split == SPLIT_NONE 时不为空，指向一个真实窗口；否则为 nullptr。
初始画布：
程序启动或切换到空工作区时，根节点即为唯一的叶子，此时 toplevel == nullptr，代表纯菜单栏+桌面背景。
这一步骤为之后的所有插入、删除和重排提供了一个干净的“根容器”。

窗口插入流程

目标叶子定位
根据鼠标在屏幕上的坐标，用 desktop_container_at(x, y) 从根开始先序递归，快速定位到承载当前焦点或桌面的叶子容器。
判断与升级
若该叶子为“纯桌面”（无窗口承载），则直接将新窗口以叶子形式挂到其 child1，将 split 保持为 NONE。
若该叶子已有窗口：
将它自身转换为一个内部分割节点，split 标记为“水平方向”或“垂直方向”（默认对半）。
原窗口保留为 child1，新窗口作为 child2，并在每个子节点上更新 parent 指针。
双向指针维护
每次改写 child1、child2 或 split，都立即修正被影响节点的 parent 指向，杜绝孤立或环状结构。

图示：



。

窗口删除流程

叶子节点判断

仅当容器节点 toplevel != nullptr 且存在父节点时，才可进入删除。

若这是根下的唯一叶子，则重置为“纯桌面”状态，释放该叶子。

叶子兄弟高度上升

从父节点获取被删除叶子的兄弟 sibling。

从祖父节点重新挂载：

若父节点也有父节点（即非根内部节点），将 sibling 接到祖父的对应分支。

若父节点为根，则直接将 sibling 提升为根节点。

内存回收

删除叶子节点及其无效的父内部节点，避免内存泄漏。

全局重排（Reflow）

每当树结构变化后，调用一次 reflow(root, full_screen_box)，递归地将屏幕空间分配给每个活跃窗口：

根节点快速判断

没有任何窗口，直接跳过。

只有单个叶子，则其大小与位置直接等于整个工作区。

分区递归

遇到内部节点，根据 split 水平或垂直对半切分当前矩形区域，生成两个子区域。

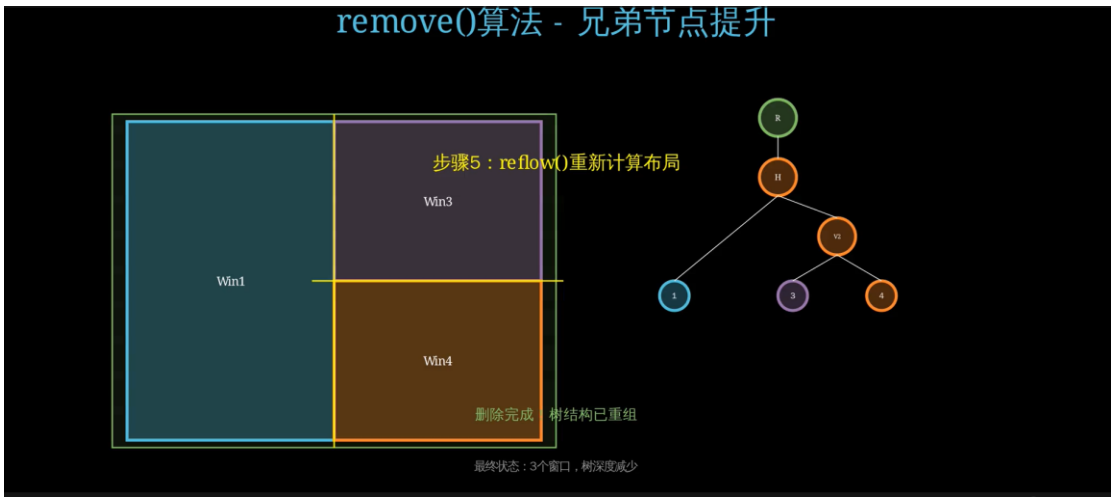
先对 child1 区域递归 _reflow，再对 child2 执行相同操作。

对叶子直接向客户端发送新的大小与位置：

平滑无闪烁

批量执行所有窗口的大小与位置更新后，触发一次渲染提交，使画面在一帧内整体更新，用户体验流畅。

图示：



拖拽与“槽位”切换

Tiley 的拖拽并非自由浮动，而是“在已分割好的槽位之间切换”：

模式进入

在 request_move 回调中，将 cursor_mode 置为 MOVE，并记录窗口原始位置与鼠标偏移。

连续命中

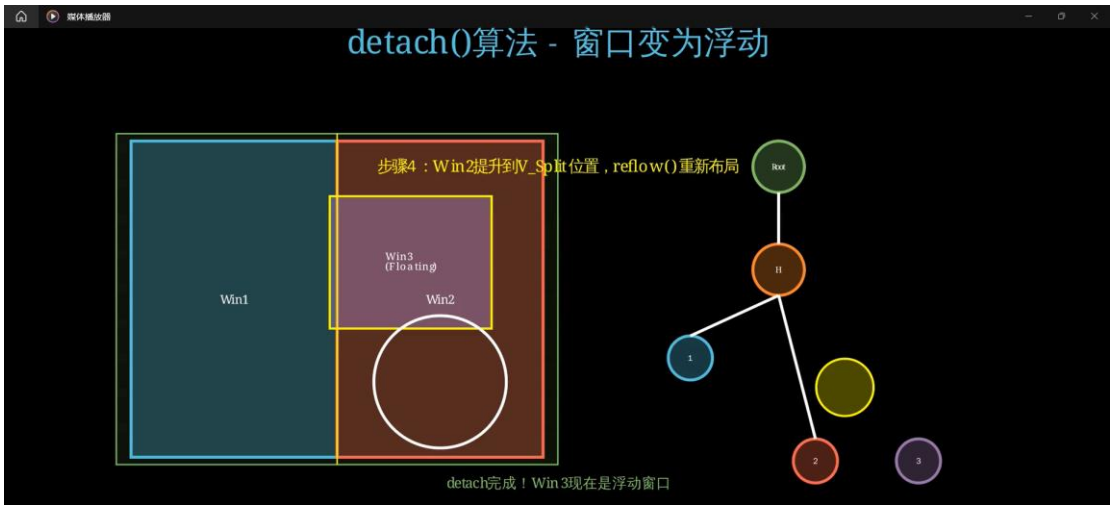
在后续的 cursor_motion 中，每次根据最新坐标调用 desktop_container_at 确定新的目标叶子。

仅当目标叶子与原始叶子不同时，执行一次“删除+插入”以将该窗口移到新槽位，并触发 reflow。

模式退出

在 cursor_button 释放时或 cursor_frame 结束后，复位 cursor_mode 为普通透传模式。

图示：



Tiley 平铺式优点：

清晰可维护：三大基本操作（插入、删除、重排）逻辑单一，配合双向指针，代码易读且无冗余。
性能高效：对半分割与先序遍历保证 O(N) 级别的更新，配合鼠标帧合并，响应迅捷。
扩展友好：未来可在 split_info 里增加不同比例、网格或树型切分；也可引入优先级或最小尺寸约束，完全不需改动核心遍历与链接代码。
这套算法既保持了平铺管理器一贯的简洁与高效，也为后续功能创新（可配置分割、插件化布局）做好了充足准备。它深入浅出、思路明确，是 Tiley 项目最值得骄傲的一环！

6.3 双向链表

在 Tiley 的实现中，wl_list（Wayland 提供的双向链表结构）被广泛用于管理各类全局对象：窗口、输入设备和输出设备。它不但承载了对象的注册与注销，还为遍历、批量操作和生命周期管理提供了统一的、高效的机制。

wl_list 简介
结构定义

wl_list 本质上包含两个指针：prev 和 next，形成一个环形双向链表。每个需要被管理的结构体在其中嵌入一个 wl_list link; 字段，用以链接到全局链表。

核心操作

插入：wl_list_insert(&head, &node->link) 将 node 插入到链表头部。

移除：wl_list_remove(&node->link) 将 node 从所在链表中摘除（并清除其 prev/next）。

初始化：wl_list_init(&head) 用于创建一个空表头，head.next = head.prev = &head。

全局对象链表

Tiley 定义并维护了三条核心链表，每条都对应一种全局可变集合。

窗口链表（server.toplevels）

用途：跟踪所有当前激活的顶级窗口（surface_toplevel）。

插入时机：在 XDG 窗口 map 回调中调用，表示该窗口已准备好首次渲染；示例中使用 wl_list_insert(&server.toplevels, &toplevel->link);。

移除时机：在窗口 unmap 或 destroy 回调中移除，通过 wl_list_remove(&toplevel->link);。

遍历用途：

在切换工作区时，遍历链表重置窗口的可见性或重新分区。

在程序退出前，通过 wl_list_for_each 依次销毁所有窗口资源。

键盘设备链表（server.keyboards）

用途：当多个物理键盘插入时，集中管理它们的修饰键与按键回调。

插入时机：server_new_input 回调检测到键盘后，用 wl_list_insert(&server.keyboards, &keyboard->link); 添加。

移除时机：键盘设备拔出或销毁时，在 keyboard_handle_destroy 回调中调用 wl_list_remove 并释放对应 input_keyboard 对象。

遍历用途：在更新 seat 的能力时（如启用或禁用键盘输入），遍历链表判断当前是否有可用键盘。

输出设备链表（server.outputs）

用途：管理所有当前所在工作区的物理显示器（output_display），以便在多屏环境下统一调整布局与渲染。

插入时机：在 server_new_output 回调中，创建 output_display 后执行 wl_list_insert(&server.outputs, &output->link);。

移除时机：屏幕拔出时，在 output_destroy 回调中 wl_list_remove(&output->link); 并释放对象。

遍历用途：

当窗口需要定位到特定显示器时，从链表中查找或遍历所有 output_display，定位到最合适的输出。

在每次重排前，可根据所有输出的分辨率动态计算每个工作区的分割起始坐标。

链表遍历模式

Tiley 多次使用 wl_list_for_each 或 wl_list_for_each_safe 来遍历链表、执行批量操作：**安全遍历**：在遍历过程中可能会修改链表（如删除节点），使用 wl_list_for_each_safe 可以保证迭代器在节点移除时依然有效。

典型用例：
程序退出时清理：

O(1) 插入删除
插入与删除操作均只修改固定的前后指针，保证了在大量窗口 / 设备频繁变动时也能保持高性能。
内存与生命周期统一管理
每个对象的分配、回调注册、链表插入和最终释放都呈现对称模式，帮助排查内存泄漏和逻辑遗漏。
可扩展性
在后续增加其他全局管理对象（如触控设备、插件模块等）时，只需新建链表头并在相应回调中插入或移除，其他框架不需改动。

通过这一套双向链表管理机制，Tiley 在维护窗口、输入设备和输出设备的全局状态时，既保证了极高的运行效率和内存安全，又让代码逻辑结构清晰、一目了然，堪称 Wayland Compositor 级别的优雅实践。

6.4 内存管理机制与相关处理算法

在高性能的窗口管理器中，健全的内存管理至关重要。Tiley 结合 C++ 智能指针与手动分配 / 释放，构建了一套既安全又灵活的内存管理方案。本节从全局单例、容器树回收、C 数据结构分配与释放，以及事件销毁中的内存清理等方面做诠释。

全局单例的生命周期管理 s
td::unique_ptr + 自定义 Deleter
TileyServer 与 WindowStateManager 均用 std::unique_ptr<ThisClass, Deleter> 存储静态 INSTANCE。
自定义 Deleter 在析构时执行必要的清理：
对于 TileyServer，删除当前对象即可（无额外资源）。
对于 WindowStateManager，自定义 WindowStateManagerDeleter::operator() 进行**后序遍历**，递归删除整棵 area_container 树，避免内存泄漏。
线程安全初始化
通过 std::call_once(onceFlag, …) 确保单例只创建一次，避免多线程环境下重复构造带来的资源竞争和泄露风险。

二叉树节点的批量释放
递归后序删除
```
void delete_node_recursive(area_container* node) const {
    if (node->child1) delete_node_recursive(node->child1);
    if (node->child2) delete_node_recursive(node->child2);
    delete node;  // 释放 C++ 对象
```

}
在 WindowStateManagerDeleter 中使用后序遍历,先销毁子节点,再销毁自身,从根启动回到叶子可确保不访问已释放的内存。

避免内存泄漏与双重删除

插入与删除算法负责维护树结构一致性,确保每个节点只在唯一且正确的时机被删除。

在节点移除（remove()）与单例析构中都有对相应子树的清理，避免遗漏。

结构体对象的分配与释放

Tiley 大量使用 C API（wlroots）和 C 数据结构，因此在关键对象上采用 calloc/free 而非 new/delete，并在相应的销毁回调中准确释放：

窗口管理对象（surface_toplevel）

在 server_new_xdg_toplevel 中：

```
surface_toplevel*  toplevel  =  (surface_toplevel*)calloc(1, sizeof(*toplevel));
```

在 xdg_toplevel_destroy 回调中：

```
wl_list_remove(&toplevel->map.link);
…  // 移除所有监听
free(toplevel);
```

弹出窗口对象（surface_popup）

分配：calloc

销毁：在 xdg_popup_destroy 回调中 free(popup);

输入设备对象（input_keyboard）

分配：calloc

销毁：在 keyboard_handle_destroy 回调中 free(keyboard);

输出设备对象（output_display）

分配：calloc

销毁：在 output_destroy 回调中 free(output);

监听器链剥离

在释放之前，先通过 wl_list_remove(&obj->link) 以及移除事件监听器的 link，确保回调不再触发已释放内存。

主循环退出与全局清理

在 main() 结束前，Tiley 会依次：

销毁所有 Wayland 客户端

```
wl_display_destroy_clients(server.wl_display_);
```

移除所有信号监听

将 wl_signal_add 注册的每条 listener.link 都执行 wl_list_remove，避免 dangling 链表节点。

释放场景树

```
wlr_scene_node_destroy_(get_wlr_scene_root_node(server.scene));
```

删除光标管理器、光标、allocator、renderer、backend

每个对应的 destroy 接口保证底层资源（GPU 缓冲、线程、文件描述符）得到妥善回收。

销毁 Display
wl_display_destroy(server.wl_display_);
通过上述分层、分阶段的清理，Tiley 保证在任何退出场景下都不会留存资源，也能快速定位潜在的内存/资源泄露点。
小结
Tiley 将 C++ 与 C 混合的内存分配方式、智能指针与手动释放结合起来：
智能指针管理全局对象，利用自定义 Deleter 做树状回收。
calloc/free 管理 C 结构体，并在回调中脱链与释放，保持生命周期同步。
有序注销所有 Listener 与资源，防止程序退出时发生未定义行为。

6.5 快捷键自定义功能

Tiley 在原有平铺与事件处理框架之上，预留了灵活的快捷键自定义支持。该模块通过配置文件驱动，运行时可热加载，并无缝集成到键盘事件流中。下面分别从设计思路、实现描述和关键代码说明三个层面进行详细阐述。

设计思路
配置驱动，用户可定制
快捷键与对应动作全由外部 JSON 文件管理，用户只需编辑文本即可添加、删除或修改快捷键，而无需触及 C++ 源码和重启窗口管理器。
热加载，实时生效
使用 Linux 的 inotify 机制监控配置文件，当检测到文件修改后立即重新加载，保证调试与迭代效率。
最小侵入，无缝集成
在原本的 keyboard_handle_key 回调中，只需在透传给客户端前插入一次映射查表与执行动作逻辑；未命中的组合键则正常透传给客户端，不影响原有行为。
线程安全，资源独立
为了不阻塞主事件循环，配置监控与重载在后台独立线程中完成；全局映射表用互斥锁保护，确保加载与查表不会竞态。
实现描述
全局数据结构
g_hotkey_map：std::unordered_map<std::string, std::string>，保存“组合键文本 → 动作名称”映射。
g_hotkey_mutex：std::mutex，保护上述映射的读写安全。
配置加载
load_hotkey_config(const std::string& path)
打开指定 JSON 文件（std::ifstream in{path}）。
解析文件内容到临时 json j。
在持锁状态下，清空 g_hotkey_map 并将 j 中的每对 key:value 插入映射。
在标准错误输出打印加载状态与映射条目数。
热加载线程
watch_hotkey_file(const std::string& path)
调用 inotify_init1(IN_NONBLOCK) 创建非阻塞 inotify 实例，

inotify_add_watch 监视指定文件的 IN_MODIFY 事件。
在 while(true) 循环中，调用 read(fd, buf, sizeof(buf))，若返回长度 > 0，则说明文件被修改，调用 load_hotkey_config(path) 重新加载映射。
每轮循环睡眠 1 秒，平衡响应速度与 CPU 占用。
启动方式：在主程序中使用 std::thread(watch_hotkey_file, path).detach() 创建后台监控。

组合键生成

keycombo_to_string(uint32_t keycode, uint32_t modifiers, xkb_state* state)
根据 modifiers 位掩码依次拼接 "ctrl+"、"alt+"、"Shift+"。
通过 xkb_state_key_get_syms 获取 keycode 对应的 keysym 列表，并用 xkb_keysym_get_name 转为字符。取第一个 keysym 作为最终字符。
返回如 "ctrl+Shift+A" 或 "alt+Return" 的标准化字符串。

动作分发

execute_hotkey_action(const std::string& action)
根据 action 字符串，调用对应的内部函数或外部命令：
"close_window"：调用已有的窗口关闭接口（待实现）。
"launch_terminal"：执行 system("alacritty &")。
focus_right"：调用 focus_next_window() 切换到下一个窗口。
可在此函数中按需扩展更多动作类型。

键盘事件集成

在 keyboard_handle_key(...) 回调最前端：
格式化当前按键和修饰键为组合键字符串 combo。
在持锁状态下查找 g_hotkey_map：
若找到且当前是按下事件，则执行 execute_hotkey_action(it->second)，并 return 拦截，不再透传。
否则跳过，走原有 wlr_seat_keyboard_notify_key 流程，将事件交给客户端。

关键代码说明

```
// 热加载：后台线程不断监控 JSON 文件修改
void watch_hotkey_file(const std::string& path) {
    int fd = inotify_init1(IN_NONBLOCK);
    int wd = inotify_add_watch(fd, path.c_str(), IN_MODIFY);
    char buf[4096];
    while (true) {
        if (read(fd, buf, sizeof(buf)) > 0) {
            load_hotkey_config(path);
        }
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}
```

说明：非阻塞 inotify + 固定延迟，既能保证快速响应配置变更，又避免了主循环阻塞和高频 CPU 占用。

目前该功能在分支上，尚在开发阶段：
尽管当前快捷键功能尚未合并至主分支，但这套框架已具备以下优势：
高度可配置：JSON 驱动，用户可在任意时间自由定义组合键与动作。（后续还会配置对应的前端界面来方便使用者自定义而不是去自己修改文件）
实时可维护：热加载保证对配置的任何修改都能立刻生效，无需重启。
兼容性与安全性：XKB 映射兼容多种键盘布局；全局映射表加锁避免竞态。
易于扩展：仅需在配置文件添加映射，并在 execute_hotkey_action 中添加对应分支，即可支持无限新功能。
Tiley 的快捷键模块，以其简洁优雅和专业安全的实现，为用户提供了极佳的可定制性与交互扩展能力

6.6 场景图（Scene Graph）封装层

Tiley 基于 wlroots 提供的 Scene Graph（场景图）功能，将窗口、弹出框等所有可视元素组织到一棵树中，以便统一管理渲染与命中测试。为方便在 C++ 中使用，我们在 src/wrap/ 目录中对关键 API 进行了简单封装。
核心数据结构
wlr_scene：整棵场景树的根，包含一个 wlr_scene_tree 子树。
wlr_scene_tree：树节点结构，代表可嵌入子树的“容器”。
wlr_scene_node：通用节点类型，可表示子树、矩形、缓冲区（窗口内容）、输出等。
wlr_scene_output：将场景图绑定到物理输出（显示器）后，生成的渲染目标。
封装函数
在 wrap/ 里，每个带下划线后缀的函数简单调用对应的 wlroots C 函数。例如：
wlr_scene_create_() → wlr_scene_create()
wlr_scene_xdg_surface_create_(parent, xdg_surface) → wlr_scene_xdg_surface_create(parent, xdg_surface)
wlr_scene_node_at_(node, x, y, &nx, &ny) → wlr_scene_node_at(node, x, y, &nx, &ny)
渲染流程 程序启动时创建全局 wlr_scene，再通过 wlr_scene_attach_output_layout_ 绑定到 OutputLayout。每当有新窗口、弹窗、矩形或其他元素出现，都往场景树中插入新的 scene node。在输出的 frame 回调里，调用 wlr_scene_output_commit_() 提交渲染，并在完成后发出 frame_done。
命中测试（Hit-Testing）
鼠标事件处理通过 wlr_scene_node_at_ 在场景图中查找给定坐标对应的最前端缓存节点（WLR_SCENE_NODE_BUFFER_）。
找到后，再从该节点向上遍历到最近绑定了 surface_toplevel 的 scene_tree，获取对应窗口。
这一机制将输入定位与布局逻辑彻底解耦，既支持平铺管理也兼容弹窗、重叠等混合场景。

焦点提升

聚焦某个窗口时，先调用 wlr_scene_node_raise_to_top_ 将其对应的 scene node 移到同级链表末端，保证渲染时位于最前。

同时在逻辑链表中也将窗口节点移至表头，保持视觉层级与输入目标一致。

这一层封装保留了 wlroots Scene Graph 的高性能与灵活性，又让 C++ 代码调用时直观、安全，不必担心 extern "C"、符号冲突或头文件隔离问题。

6.7 输出与工作区映射

在多显示器环境下，Tiley 支持将每个物理输出（屏幕）映射到不同的工作区，实现跨屏工作区切换。

数据结构

workspace_roots（std::vector<area_container*>）：存储每个工作区的根容器树，默认数量为 10。

display_to_workspace_map（std::map<std::string,int>）：键为输出名称（wlr_output->name），值为当前映射的工作区编号。

主要接口 insert_display(output_display* new_display)

在新输出接入时调用,为其分配一个新的工作区编号(按映射表当前大小顺序)。

更新 display_to_workspace_map[name] = workspace，并可视化输出日志。

remove_display(output_display* removed)

在输出拔出时调用，删除映射表中的对应项。简单高效地释放该输出与工作区的绑定关系 **move_display_to_workspace(output_display* display, int target_ws)**

支持运行时将某个物理屏切换到另一个工作区（例如快捷键触发、配置命令行接口）。

只需更新映射表，后续调用 reflow 时即可重派该工作区的窗口到此输出。

get_workspace_by_output_display(output_display* display)

返回该输出当前所在的工作区编号，用于在事件或重排阶段，确定应使用哪棵窗口树进行布局。

工作流程

在 new_output 回调中，创建 output_display 并调用 insert_display；

在渲染或布局阶段，根据当前光标落在某个输出上，通过 wlr_output_layout_output_at 获取该输出对象，再用 get_workspace_by_output_display 得到对应工作区,最后对该工作区的根容器执行 reflow。

这一映射机制让 Tiley 在多显示器下表现如同独立管理器一般：每个屏幕各有自己的容器树，窗口只在映射到的工作区内可见；用户通过快捷键或 API 可随时将屏幕与工作区重绑，灵活性与可控性俱佳。

6.8 焦点切换与双向链表遍历

焦点管理负责将用户的键盘与鼠标输入定向到正确的窗口,并同步调整渲染层级。Tiley 结合场景图与双向链表，实现了高效、可循环的焦点切换。

双向链表

wl_list toplevels：在 TileyServer 中维护的链表头，用于链接所有当前激活的 surface_toplevel。

每个 surface_toplevel 结构体都包含一个 wl_list link; 字段，用于链表操作。

聚焦当前窗口

```
void focus_toplevel(surface_toplevel* toplevel) {
    // 1. 失焦上一个
    // 2. 场景图提升：wlr_scene_node_raise_to_top_
    // 3. 链表移动：先从原位置 remove，再插入到表头
    // 4. 调用 wlr_xdg_toplevel_set_activated(true)
    // 5. 键盘聚焦：wlr_seat_keyboard_notify_enter
}
```

切换下一个窗口

```
void focus_next_window() {
    // 从链表头开始遍历 toplevels：
    //   找到当前聚焦的窗口节点后，下一次迭代即为“下一个”
    //   找不到则回到第一个，实现循环切换。
    // 最后调用 focus_toplevel(目标节点)
}
```

遍历使用 wl_list_for_each，O(N) 级别；切换逻辑简单明了，支持在平铺与弹窗混合场景下无缝过渡。

设计价值

链表顺序即为渲染/输入顺序：可直接通过链表逻辑确定“下一个”目标，无需额外索引结构。

与场景图协同：focus_toplevel 同时调用场景图提升和链表移动，保证前端（视觉）与后端（输入）完全一致。

循环与回退：当到达链表末尾时，切换逻辑自动回到表头，使得用户体验流畅自然。

6.9 事件循环与资源清理

Tiley 遵循 Wayland Compositor 的生命周期规范，主循环与退出清理分为以下几个阶段：

启动事件循环

在 main() 的最后，调用 wl_display_run(server.wl_display_)。

这一函数进入无限循环，不断监听并分发 Wayland 与 wlroots 的各类事件（输出、输入、窗口、定时器等），直至收到退出信号。

客户端销毁

在退出流程开始时，调用 wl_display_destroy_clients(server.wl_display_)，强制关闭所有与客户端的连接，清理它们占用的资源。

反注册回调

对所有先前通过 wl_signal_add 注册的监听器，都执行

wl_list_remove(&listener->link)，确保链表不再引用已释放的回调结构。
包括输出、新窗口、弹窗、光标事件、输入设备、Seat 请求等所有 11 条核心回调。

释放核心对象

场景图：wlr_scene_node_destroy_(get_wlr_scene_root_node(server.scene))，递归销毁场景树。

光标管理：wlr_xcursor_manager_destroy(server.cursor_mgr) 、wlr_cursor_destroy(server.cursor)。

渲染与分配器：wlr_allocator_destroy(server.allocator) 、wlr_renderer_destroy(server.renderer)。

Backend：wlr_backend_destroy(server.backend)。

Display：wl_display_destroy(server.wl_display_)。

单例与全局对象清理

程序退出时，TileyServer 与 WindowStateManager 的 unique_ptr 触发自定义 Deleter，释放单例对象与布局树节点。

全局链表管理的 C 结构体（surface_toplevel、output_display、input_keyboard）已在各自的 destroy 回调中释放。

6.10 桌面壁纸（Wallpaper）

1. 功能概览

Tiley 支持在每个物理输出上显示自定义的壁纸图像，呈现为最底层背景，从而丰富桌面视觉效果。壁纸加载仅发生一次，后续渲染均重用已生成的图像缓冲区，性能开销微乎其微。

2. 设计思路

一次性解码，长期复用

利用 STB Image 在启动或新输出接入时，将指定文件解码为原始像素数据，并拷贝到 wlroots 的 SHM 缓冲区中；图像缓冲只创建一次，不随帧刷新重复解码或上传，避免性能浪费。

自动适配输出分辨率

根据每块屏幕的当前宽高，按原始尺寸或等比例缩放（letterbox）两种模式动态调整目标画布大小，将壁纸居中显示或填满屏幕。

Scene Graph 最底层渲染

将壁纸缓冲作为一个 Scene Buffer 节点插入到每个输出的场景子树底层，保证其后续所有窗口、弹窗都能覆盖其上，实现“窗口上层、壁纸下层”的渲染顺序。

主要流程

图像加载

从文件系统读取指定路径，通过 STB Image 解码为 RGBA 原始像素。

缓冲区创建

基于 wlroots SHM allocator 创建一个符合 DRM_FORMAT_XRGB8888 且可 CPU 访问的 wlr_buffer。

像素拷贝

将解码后的像素按行拷贝到缓冲区内存，转换为 XRGB8888 格式；完成后释放解码数据。

注入场景图

在新输出接入或启动时，将生成的 wlr_buffer 通过 Scene Graph API 绑定到对应输出的根节点，固定在最底层。

4. 使用效果

启动过程中或插入新屏幕时，一次性完成壁纸初始化；

切换工作区、窗口重排不影响壁纸，流畅且无闪烁；

当输出分辨率变化（如旋转或外接 4K 屏），壁纸可重新加载或重新缩放，保持最佳显示效果。

6.11 窗口浮动边框装饰

1. 功能概览

在“浮动”或“调整大小”模式下,Tiley 会为当前拖拽或调整的窗口显示一组高亮边框，清晰标识可交互的四条边，提升可视反馈和用户体验。

2. 设计思路

独立边框，灵活可控

将上下左右四条边分别作为 Scene Graph 中的矩形节点，能够单独开关可见性，也可按需改变厚度和颜色。

自动跟随窗口

边框节点附着在窗口的场景子树上，随着窗口位置或大小变化而自动调整，无需额外同步；在每次渲染提交后更新边框几何即可。

最小侵入

装饰逻辑与平铺布局互不干扰，仅在用户进入浮动模式或响应 resize 请求时启用，平铺模式下默认隐藏。

3. 主要流程

创建与隐藏

在窗口首次进入可装饰状态时，分配 4 条矩形节点并插入到窗口 Scene Tree，但默认将它们设为不可见。

启用与显示

当用户开始拖拽或调整大小，调用“启用”接口，将这四条边框打开，提供视觉指示。

动态更新

在每次窗口提交新帧或大小发生变化后，根据当前窗口宽高和预设的边框厚度，重新计算四条矩形的尺寸和相对位置，保证边框紧贴窗口各边。

关闭与销毁

拖拽或调整完成后隐藏边框;窗口销毁时再彻底销毁这四个节点,释放场景资源。

4. 使用效果

浮动模式下可见边框鲜明，帮助用户准确把握拖动与调整的意图；

平铺模式时自动隐藏，保持界面简洁；

边框厚度与颜色可配置，满足不同主题或高对比度需求。