

# Tiley: 基于 Wayland 的平铺式窗口管理器 技术文档

爱吃橘子的 doro 队

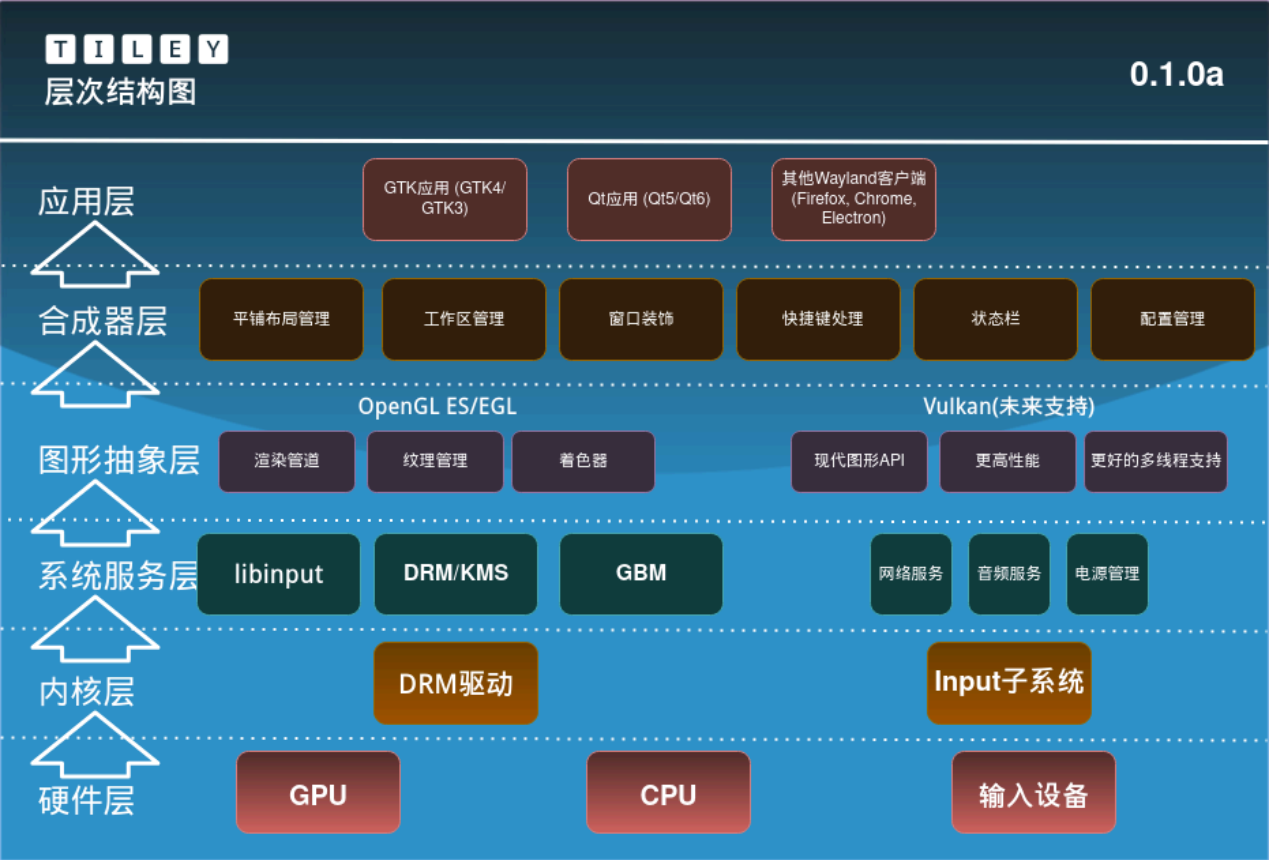
## 目录

- ▶ Tiley: 基于 Wayland 的平铺式窗口管理器技术文档
  - ▶ 1. 架构概览
    - ▶ 项目文件结构
    - ▶ 项目开发周期
    - ▶ 开发过程概述
      - ▶ 1.1 框架使用
        - ▶ 1.1.1 现用框架
        - ▶ 1.1.2 历史依赖
      - ▶ 1.2 设计模式
      - ▶ 1.3 代码实现
      - ▶ 1.4 用户体验问题解决
        - ▶ 1.4.1 用户交互的随机性
        - ▶ 1.4.2 客户端的多样性
        - ▶ 1.4.3 硬件环境的多样性
  - ▶ 2. 设计思路
    - ▶ 2.1 设计目标
      - ▶ 2.1.1 高性能: 构建瞬时响应的桌面核心
      - ▶ 2.1.2 现代化的用户体验: 兼顾设计与使用便利性
      - ▶ 2.1.3 高度可定制性
    - ▶ 2.2 问题解决流程: 从 wlroots 的转变
      - ▶ 2.2.1 转变的目的: 解决 wlroots 的痛点
      - ▶ 2.2.2 转变的过程: 有序的模块化迁移
      - ▶ 2.2.3 迁移中遇到的主要问题
    - ▶ 2.3 架构哲学: 面向对象的实践与展望
      - ▶ 2.3.1 将 Wayland 世界模型化: OOP 的核心实践
      - ▶ 2.3.2 架构驱动的良好循环: 可拓展、可读与用户友好的统一
  - ▶ 3. 核心功能设计
    - ▶ 3.1 Louvre 框架介绍
      - ▶ 3.1.1 虚函数继承: 完全面向对象开发

- ▶ 3.1.2 在现代化硬件上测试并运行
- ▶ 3.1.3 高性能
  - ▶ FPS - 测试图像
  - ▶ CPU/FPS 比 - 测试图像
  - ▶ GPU/FPS 比 - 测试图像
- ▶ 3.2 平铺功能
  - ▶ 3.2.1 窗口分层和浮动模式
  - ▶ 3.2.2 层级关系模型
  - ▶ 3.2.2 分层问题解决: `Surface` 消失的问题
  - ▶ 3.2.3 平铺算法基本逻辑
- ▶ 3.3 窗口管理
  - ▶ 3.3.1 溢出裁剪的问题: 客户端需要的尺寸和平铺分配的冲突
  - ▶ 3.3.2 平铺窗口的视图层级
  - ▶ 3.3.3 管理形式适应窗口类型
  - ▶ 3.3.4 用户和窗口的交互: 移动和调整大小
- ▶ 3.4 用户交互模型
  - ▶ 3.4.1 焦点跟随鼠标
  - ▶ 3.4.2 "活动容器"设计
  - ▶ 3.4.3 全键盘操作
- ▶ 3.5 工作区
  - ▶ 3.5.1 "视图可见性" vs "map/unmap"
  - ▶ 3.5.2 逻辑上数量无限制
  - ▶ 3.5.3 工作区布局记忆
- ▶ 3.6 界面和自定义部分
  - ▶ 3.6.1 图形化设置界面
  - ▶ 3.6.2 快捷键自定义
  - ▶ 3.6.3 窗口效果动画
  - ▶ 3.6.4 窗口装饰自定义
- ▶ 4. 和其他平铺式管理器的比较
  - ▶ 4.1 框架与架构的现代性
    - ▶ 4.1.1 技术栈的代际优势 (同 `Sway` , `i3` 相比)
    - ▶ 4.1.2 渲染自由度与性能的兼顾 (同基于 `wlroots` 的合成器相比):
  - ▶ 4.2 人性化与开箱即用的体验
  - ▶ 4.3 原生的工作区布局记忆
- ▶ 5. 文档完整性说明

# 1. 架构概览

**Tiley** 是一个基于 Wayland 协议的平铺式 Linux 桌面环境, 使用 C++开发, 旨在让用户可以快速上手、免去复杂的配置而开箱即用, 同时兼具美观和技术新颖性。



## 项目文件结构

截至文档编写时, **Tiley** 的项目文件结构如下:

```

.  📁 tiley    // 根目录
├── 📄 README.md    // README
├── 📁 archive/    // 历史文件归档
├── 📁 backup/     // 历史文件备份
├── 📁 design/     // (开发用)设计思路文件
├── 📄 dev_run.sh   // 开发快捷启动脚本
├── 📄 hotkey.json  // 快捷键配置表
├── 📁 include/    // 外部库头文件
├──   ├── 📁 glm/      // GLM数学库
├──   ├── 📁 nlohmann/  // JSON库
├──   └── 📄 json.hpp   // JSON库头文件
├── 📄 stb_image.h  // stb图像处理库
├── 📄 learing.md   // 框架学习记录
├── 📄 learning_2.md
├── 📄 meson.build  // meson配置文件
├── 📄 meson.options // meson配置文件
├── 📁 src/        // 源代码目录
├──   ├── 📁 assets/   // 资源目录
├──   │   ├── 📁 shaders/ // 渲染脚本目录
├──   │   │   ├── 📄 rounded_corners.frag // 片段着色器
├──   │   │   ├── 📄 rounded_corners.vert // 顶点着色器
├──   │   │   └── 📁 wallpaper/ // 默认壁纸目录
├──   │   │       ├── 📄 tiley_16x10.png // 默认壁纸16:10
├──   │   │       ├── 📄 tiley_16x10@2x.png // 默认壁纸16:10, 放大2倍
├──   │   │       └── 📄 tiley_1x1.png // 默认壁纸1:1(方形)
├──   └── 📄 config.h.in // meson预编译配置文件
├──   └── 📁 lib/      // 核心代码
├──       ├── 📄 TileCompositor.cpp // Compositor: 合成器类, 用于完成Wayland协议对象管理
├──       ├── 📄 TileCompositor.hpp
├──       ├── 📄 TileServer.cpp // Server: 服务端类, 用于完成服务器所需资源管理
├──       ├── 📄 TileServer.hpp
├──       ├── 📄 TileWindowStateManager.cpp // WindowStateManager: 窗口状态管理类
├──       ├── 📄 TileWindowStateManager.hpp
├──       ├── 📄 Utils.hpp // 工具文件。包含一些通用工具函数
├──       └── 📁 client/ // 客户端相关代码
├──           ├── 📄 Client.cpp // Client: 客户端类, 用于与客户端进行通信
├──           ├── 📄 Client.hpp
├──           ├── 📄 ToplevelRole.cpp // ToplevelRole: 窗口角色类, 用于管理客户端窗口
├──           ├── 📄 ToplevelRole.hpp
├──           ├── 📄 meson.build
├──           └── 📁 render/ // 渲染相关代码
├──               ├── 📄 SSD.cpp // SSD: 服务端装饰相关
├──               ├── 📄 SSD.hpp
├──               ├── 📄 Shader.cpp // Shader: 代表编译并连接成管线的渲染脚本对象
├──               └── 📄 Shader.hpp
├──           └── 📁 views/ // 视图相关代码
├──               ├── 📄 LayerView.cpp // LayerView: 用于作为容器的视图的公共父类
├──               ├── 📄 LayerView.hpp
├──               ├── 📄 SurfaceView.cpp // SurfaceView: 用于显示一个Surface的视图。
├──               ├── 📄 SurfaceView.hpp // **内含自定义的渲染管线**
├──               └── 📁 extra/ // 视图额外代码

```

- ├── DebugView.cpp // DebugView: 暂未实现的调试视图。
- ├── DebugView.hpp // 将附着在桌面上, 显示各种合成器当前信息
- ├── config/ // 配置相关代码
  - ├── Config.cpp // Config: 配置管理器相关
  - ├── Config.hpp
- ├── core/ // 使用到的数据结构代码
  - ├── Container.cpp // Container: 平铺容器,
  - ├── Container.hpp // 包含平铺状态下的窗口的各种布局信息
  - ├── UserAction.cpp // UserAction: 用户交互对象。
  - ├── UserAction.hpp // 将用于处理用户传入的键盘鼠标事件
- ├── input/ // 输入相关代码
  - ├── Handle.cpp // Handle: 将与UserAction合并
  - ├── Handle.hpp
  - ├── Keyboard.cpp // Keyboard: 键盘输入事件处理相关
  - ├── Keyboard.hpp
  - ├── Pointer.cpp // Pointer: 鼠标输入事件处理相关
  - ├── Pointer.hpp
  - ├── Seat.cpp // Seat: 代表一个操作环境
  - ├── Seat.hpp // 运行在Keyboard和Pointer之前的事件处理相关
  - ├── ShortcutManager.cpp // ShortcutManager: 快捷键管理器
  - ├── ShortcutManager.hpp // 用于加载、配置、更新用户定义的快捷键
- ├── meson.build
- ├── output/ // 显示屏相关代码
  - ├── Output.cpp // Output: 代表显示屏, 处理显示器相关事件
  - ├── Output.hpp
- ├── scene/ // 场景相关代码
  - ├── Scene.cpp // Scene: 代表当前屏幕上的场景
  - ├── Scene.hpp
- ├── surface/ // 表面(Surface)相关代码
  - ├── Surface.cpp // Surface: 用于处理客户端发来的表面相关事件
  - ├── Surface.hpp
- ├── types.hpp // 全局通用枚举定义等
- ├── meson.build
- ├── tiley.cpp // tiley.cpp: 程序入口
- ├── xdg-shell-protocol.c // xdg-shell-protocol: xdg窗口协议相关获取媒介。暂未使用
- ├── subprojects/ // 项目非纯头文件依赖
- ├── useful.md

## 项目开发周期

时间节点	说明
2025-05-20 左右	项目开始开发, 引入 <code>wlroots</code>
2025-06-07	<code>wlroots</code> 模板代码修改完毕
2025-06-19	平铺核心算法设计完成
2025-06-22	<code>wlroots</code> 版 <code>Tiley</code> 遇到开发瓶颈, 遇到无法自定义渲染器和胶水代码过多问题

2025-06-30	初赛代码和文档完善
2025-07-02	开始迁移到 Louvre 框架
2025-07-14	动态平铺容器二叉树核心算法迁移完成
2025-07-21	Louvre 版本的 Tiley 基本迁移完成
2025-07-27	重大进展: 成功实现 Louvre 框架下的自定义渲染
2025-08-09	Louvre 版本的 Tiley 各分支代码合并完成

## 开发过程概述

本项目在开发过程中, 主要以 4 个方面为指导性步骤, 分别是**框架使用**、**设计模式**、**代码实现**和**用户体验**。尤其是代码实现和用户体验部分,项目实现的每个功能都充分考虑了 Wayland 环境的复杂性、用户交互的随机性和用户使用的便捷性。

### 1.1 框架使用

#### 1.1.1 现用框架

开发一个 Wayland 合成器需要考虑到协议通信、用户交互、性能和美观程度。鉴于此, **Tiley** 使用开源框架**Louvre** [\[1\]](#)(**GPL 2.1** 协议) 完成 Wayland 协议通信部分的抽象。该框架将各个协议接口封装为独立的 C++类, 使得在开发过程中, 我们无需手动完成协议通信的处理, 而是转为响应一个个**客户端事件** [\[2\]](#)。这就将协议通信的复杂性转换成了事件发生的随机性, 仅需要在基于"每个事件都是独立发生"的前提下进行合适的事件响应即可。

此外, 为了适应这样的随机性, 项目在开发过程中尤其强调数据模型和视图以及外部环境的分离, 在操作数据模型时不通过协议发送信息, 在响应客户端事件时不操作数据模型, 从而保证了客户端和服务端的同步性。



截至目前, 项目使用的 **Louvre** 版本为 **2.18.1**, 该版本下 **Louvre** 的依赖库包括:

- ▶ **SRM** [\[3\]](#): 实现 **Louvre** 的渲染功能、表面更新(Damage)管理等
- ▶ **GL ES 2.0** [\[4\]](#): **OpenGL ES 2.0**, 实现最底层的渲染。

此外, 为了方便具体的功能实现, 项目还使用了一些常见的第三方库:

- ▶ **glm** [\[5\]](#): 一个用于简化和加速 **OpenGL** 数学计算的工具库
- ▶ **nlohmann/json** [\[6\]](#): 一个用于简化 **JSON** 配置文件的读写和数据类型管理的工具库
- ▶ **stb\_image** [\[7\]](#): 一个 **Linux** 下的纯头文件图像处理库, 提供方便的图像读取、缓冲管理和编码格式抽象

#### 1.1.2 历史依赖

本项目在开发的前期过程中, 曾使用wlroots  作为服务端框架。虽然 wlroots 是一个经典框架, 被诸如 Sway  等市面上已有的平铺式管理器所使用, 但经过权衡, 决定以 Louvre 进行替换。主要原因如下表对比:

	wlroots	Louvre
渲染可自定义性	弱。wlroots 内置的渲染流水线固定, 只能使用内置场景图模型渲染, 无法使用自定义 shader。	强。可根据渲染的对象不同局部替换自定义 shader, 做到外观自由定制。
性能	由于渲染管线固定, 相比 Louvre 较低。详见下方性能对比图。	高。优化的渲染瓶颈代码, 场景图、扫描直出(Direct Scanout)、OpenGL 绘制合用, 为不同渲染情况使用不同的渲染方法, 提高性能。
框架易用性和技术性	使用 C 语言编写, 无直接面向对象支持, 对于 wayland 的各种协议接口抽象无直接的模拟; 使用早期的 C 语言特性, 对于使用 C++进行开发的流程而言需要额外包装层进行兼容, 代码冗余大。	使用 C++20 标准开发, 完全面向对象, 对于各个 wayland 协议接口(如鼠标对象、键盘对象、屏幕对象等)而言更加直观。

## 1.2 设计模式

本项目在开发过程中完全遵守面向对象的设计模式, 包含了一系列子模式:

- ▶ 对于每个成员属性, 决定可访问性( `public` / `private` )根据该成员是否产生副作用而决定。例如, 在"上一个活动容器"设计中, 活动容器成员为私有, 防止意外更改, 确保系统的稳定性; 对于仅仅是标签的属性而言, 则为公有, 因为该成员的值修改不产生副作用。
- ▶ 渐进式开发。 Louvre 框架本身提供了一套完整的 wayland 协议接口(对象列表 [🔗](#)), 因此在开发时逐渐使用自己的子类替换父类返回即可。所有的对象由 `TileyCompositor::createObjectRequest` 方法返回, 对于尚未实现的对象(或无需修改行为的对象), 则使用父类的 `LCompositor::createObjectRequest` 方法返回具体对象:



```

LFactoryObject* TileyCompositor::createObjectRequest(LFactoryObject::Type objectType, const void* params){

    if (objectType == LFactoryObject::Type::LOutput){
        // 显示器对象
        return new Output(params);
    }

    if (objectType == LFactoryObject::Type::LClient){
        // 客户端对象
        return new Client(params);
    }

    if (objectType == LFactoryObject::Type::LSurface){
        // 表面对象
        return new Surface(params);
    }

    if (objectType == LFactoryObject::Type::LToplevelRole){
        // 窗口角色对象
        return new ToplevelRole(params);
    }
    if (objectType == LFactoryObject::Type::LKeyboard){
        // 键盘对象
        return new Keyboard(params);
    }
    if (objectType == LFactoryObject::Type::LPointer){
        // 鼠标(实体)对象
        return new Pointer(params);
    }
    if (objectType == LFactoryObject::Type::LSeat){
        // 操作环境对象(一个Seat包含一套完整的用户与合成器交互所需的工具, 包括键盘、鼠标等)
        return new Seat(params);
    }
    // 随着代码的迁移越来越完善, 下方需要继承的类逐渐完成继承。
    return LCompositor::createObjectRequest(objectType, params);

    //if (objectType == // LFactoryObject::Type::LSubsurfaceRole){
    // TODO: 实现subsurface角色
    //}
    // ...
}

```

- ▶ 数据模型和环境分离的模式。对于平铺式管理器而言, 平铺算法是整个合成器的核心, 使用[容器二叉树](#)数据结构实现。由于该数据结构反映了窗口的布局结构, 而客户端的事件又是随时变化的, [Tiley](#)需要确保操作数据结构和与环境交互之间的相互独立性。独立性还会带来更多好处, 包括可以独立测试容器树算法的稳定性和性能而无需真正处于 [Wayland](#) 环境之下(类似于无头模式), 关于如何实现这样的独立性在下文中进行详述。

### 1.3 代码实现

本项目的代码编写过程同时在虚拟机和真机上进行, 其中虚拟机的操作系统为 [Ubuntu 22.04 LTS](#), 相关的真机的硬件配置和软件配置为:




- ▶ [CPU](#) : Intel® Core i7-12700H @ 4.6GHz

- ▶ GPU : Nvidia® Geforce RTX 3060 Laptop
- ▶ 内存 : 16GB DDR5
- ▶ 显示器 : 2560x1600 @ 165Hz
- ▶ 操作系统 : Archlinux, 内核版本 6.15.9-arch1-1
- ▶ Wayland版本 : 1.24.0-1

代码运行环境有二, 其一是运行在以 `gnome` 或 `hyprland` 为宿主合成器的嵌套模式下, 主要用于开发阶段前期进行调试; 其二是运行在 `TTY` 模式下, 直接成为主合成器与 `Wayland` 进行交互, 用于测试、模拟用户体验等。

`Tiley` 使用 `C++` 语言开发, 语言标准是 `C++20`, 一方面, 这是 `Louvre` 的语言标准, 另一方面是为了在开发过程中使用高版本 `C++` 特性, 例如智能指针、Lambda 表达式、扩充的 `std` 库等。 `Tiley` 的构建系统使用 `meson`, 该现代化的构建系统以脚本化的形式进行指令组织, 在依赖库缺失时可以自动查找、下载并补全, 可以非常方便地完成环境的构建。

在开发时, 主要参考了如下文档:

- ▶ [Louvre 文档主页](#) : 这是 `Louvre` 的文档, 包含 API 文档、案例项目和教程等等;
- ▶ [SRM 文档主页](#) : 这是 `Louvre` 的渲染库 `SRM` 的文档, 主要用于了解渲染流程以及使用 `DRM` (`TTY` 模式)和 `Wayland` (嵌套模式)后端时的不同环境配置;
- ▶ [The Wayland Book](#) : 这是 `Wayland` 协议的学习参考手册。
- ▶ [The Wayland Protocol](#) : 这是 `Wayland` 官方编写的协议文档。
- ▶ [Learn OpenGL CN](#) : 这是 `OpenGL` 的参考教程, 作为查询手册使用。

## 1.4 用户体验问题解决

合成器作为 `Linux` 桌面体验的最重要的一部分, 在开发时除了基本功能外, 尤其注重用户体验。 `Tiley` 在开发过程中, 主要从如下方面考虑:

### 1.4.1 用户交互的随机性

用户在使用桌面环境时, 鼠标或键盘的操作对于合成器而言是随时可能发生的, 间隔不固定、类型不固定成为了我们在处理客户端事件时最棘手的问题。 为此, `Louvre` 通过暴露事件回调接口的形式, 尽可能切分事件到最小单元, 在任何事件发生时都只专注于当前事件的响应而无论上下文。例如下面的表面 (Surface)堆叠顺序发生改变的例子:

```

void Surface::orderChanged()
{
    //Log::debug("顺序改变, surface地址: %d, 层次: %d", this, layer());

    // 调试: 打印surface前后关系
    // TileServer& server = TileServer::getInstance();
    // server.compositor()->printToplevelSurfaceLinklist();

    if (toplevel() && toplevel()->fullscreen())
        return;

    // 由于每个Surface在重排顺序时都会触发该方法, 因此我们
    // 只需知道每个Surface的前一个Surface即可而无需当前是
    // 第几个Surface在排序
    Surface *prevSurface { static_cast<Surface*>(this->prevSurface()) };
    LView *view { getView() };

    while (prevSurface != nullptr)
    {
        if (subsurface() || prevSurface->getView()->parent() == view->parent())
            break;

        prevSurface = static_cast<Surface*>(prevSurface->prevSurface());
    }

    if (prevSurface)
    {
        if (subsurface() || prevSurface->getView()->parent() == getView()->parent())
            view->insertAfter(prevSurface->getView());
    }
    else
        view->insertAfter(nullptr);
}

```

该方法是处理随机性的一个非常直观的体现。用户在打开一个新的窗口后, 会触发表面重排, 以便新的表面(或窗口的子表面)找到所在堆叠层次。

对于平铺管理器而言, 堆叠层次主要是窗口和其子表面(如视频、画布或者弹出菜单等元素)之间的关系。在触发表面重排时, 我们无需访问完整的 surface 列表, 而是采用类似"洗牌"的方法, 将视图按照 surface 顺序, 每个视图插入到 surface 顺序在自己之前的视图之前即可。这样避免了环境污染而导致的无限递归, 更能提高性能, 以局部重排代替全局重排, 期望时间复杂度降低。


#### 1.4.2 客户端的多样性

Linux 平台下, 软件 UI 开发工具众多, 如 **GTK**、**QT**、**electron** 等框架, 使得客户端的行为和显示逻辑具有多样性。例如, 大部分 **GTK 应用程序不支持服务端装饰** [🔗](#), 这在一般的平铺式管理器中会带来显示统一性的问题。平铺式管理器倾向于"一个工作区一个主要窗口, 并且更多是键盘控制", 即以工作区/键盘为中心, 而不像传统的堆叠式管理器那样, 以窗口/鼠标为中心, 用户控制每个窗口, 需要每个窗口都有完整的控制控件(如"最小化"、"最大化"等)。

为了使得 **GTK** 应用程序和其他应用程序显示效果类似, 即都隐藏自己的客户端控件, 需要对窗口进行伪最大化操作。对于其他类型的应用, 只需判断是否支持服务端装饰, 并选择使用伪最大化或不显示客户端装

饰, 视情况而定。

#### 1.4.3 硬件环境的多样性

Linux 平台虽然不是当前世界上最主流的个人桌面操作系统, 但仍然拥有广泛的硬件支持。Tiley 作为一个桌面环境, 需要注重对多型号的显卡支持, 包括 Intel、Nvidia 和 AMD 系列显卡等, 其中最主流的是 Nvidia 显卡, 但由于 [Nvidia 对开源驱动的完全支持刚刚起步](#) , 现在大部分系统仍然使用的是闭源驱动, 在开发中会遇到各种兼容性问题。这就使得 Tiley 需要做到:

- ▶ 尽量使用通用的方法。例如, 在渲染方面, 不使用过于特殊的 OpenGL 拓展、不高估目标硬件的性能等;
- ▶ 需要在不同的环境下进行测试以确保硬件广泛兼容。

## 2. 设计思路

**Tiley** 的开发过程, 不仅仅是功能实现的堆砌, 更是一个在清晰的顶层设计理念指导下, 进行持续迭代的工程实践。本部分将深入阐述项目背后的核心设计思想——即我们“为什么这么做”。

这些设计思路是项目的基石, 确保了 **Tiley** 在技术选型、架构搭建和功能实现上的高度一致性与前瞻性。

### 2.1 设计目标

在项目启动之初, **Tiley** 便确立了核心设计目标。它并非简单地复刻现有的平铺式窗口管理器, 而是旨在成为一个将高性能、高可定制性、用户尊重性融合的 **Wayland** 合成器。

我们尊重并学习了如 **Sway** 在 **Wayland** 生态中的开创性地位, 以及如 **Hyprland** 在视觉效果上的大胆探索。因此, **Tiley** 的目标源于一个更具体的愿景: 解决我们在实际使用中遇到的特定挑战, 并为初级用户提供一个更加均衡和完善的选择。我们可同时面向初级用户和高级用户, 既为初级用户提供开箱即用的体验, 也为高级用户提供一定的可定制性。项目团队的主要代码编写者日常使用的桌面环境就是 **Hyprland**, 在刚开始使用的过程中深有其感, 即需要加强对新用户的引导和便利性。为了将这一宏观愿景转化为可执行、可衡量的工程指标, 我们将其分解为以下三个具体的设计目标:

#### 2.1.1 高性能: 构建瞬时响应的桌面核心

性能直接决定了用户交互的品质。**Tiley** 将性能置于最高优先级, 我们追求的是一种“感觉不到延迟”的瞬时响应体验, 主要分为下面的几个方面:

**交互延迟优化:** 用户的每一次键盘敲击、鼠标移动, 都应得到即时的视觉反馈。我们致力于缩短从物理输入到屏幕像素更新(Input-to-Photon Latency)的完整链路时间, 让用户感觉自己与系统融为一体, 实现真正的“指哪打哪”。

**高帧率的流畅渲染:** 现代硬件, 尤其是高刷新率显示器(如本项目测试用的 165Hz 屏幕), 为流畅的视觉体验提供了基础。**Tiley** 的目标是充分利用硬件能力, 确保在窗口移动、缩放、切换工作区等所有动态操作中, 动画渲染都能稳定匹配显示器的刷新率, 防止撕裂与卡顿。

**高效的资源利用:** 高性能不应以高昂的资源消耗为代价。**Tiley** 的设计目标是在提供丰富视觉效果的同时, 将空闲和负载状态下的 CPU、GPU 及内存占用维持在较低水平。这使得它不仅能在高端工作站上有较好的表现, 也能在依赖电池续航的笔记本电脑上高效运行。例如, 下面是实际运行时对于内存占用的截图(**LLauncher** 是 **Tiley** 的守护进程):

tiley	866811	0.0 %	65.4 MB	-	-
LLauncher	866842	0.0 %	864.3 kB	-	-
LLauncher	866879	0.0 %	864.3 kB	-	-
LLauncher	866881	0.0 %	864.3 kB	-	-

可见, 当合成器被积极使用时, 内存占用不到 70MB, 对于内存的需求量非常小。

#### 2.1.2 现代化的用户体验: 兼顾设计与使用便利性

传统的平铺式管理器往往功能强大但上手门槛高, 需要用户投入大量时间进行学习和配置。Tiley 致力于打破这一局面, 将现代桌面环境的美学标准与直观易用的交互逻辑相结合。

**开箱即用体验:** 提供一套精心设计的默认配置, 包括美观的色彩主题、符合直觉的快捷键布局以及平滑的动画效果。用户在首次启动 Tiley 时, 无需任何修改即可获得一个功能完备且视觉舒适的桌面环境。

**清晰的视觉反馈系统:** 用户的每一个操作都应有明确的视觉响应。无论是窗口焦点的切换(通过阴影变化)、工作区的变换(通过状态栏指示和过渡动画), 还是窗口模式的改变(如从平铺变为浮动), Tiley 都提供了无歧义的视觉提示, 帮助用户时刻掌握当前桌面的状态。

### 2.1.3 高度可定制性

工具应当能适应用户, 而非让用户去适应工具。Tiley 在提供默认体验的同时, 赋予用户最大程度的自由, 让他们可以根据自己的工作习惯和审美偏好定制桌面环境。

**可读性优先的配置文件:** Tiley 的所有可配置项, 从快捷键绑定到颜色主题, 均通过一个结构清晰的 JSON 文件进行管理。我们选择 JSON 是因为它语法简洁、人类可读性强, 且生态系统成熟, 便于用户编辑、校验和分享。此外, JSON 文件可以非常简单地完成序列化/反序列化, 我们还提供了实时读取配置并应用的功能, 使得配置可以自动更新, 用户仅仅修改内容即可。

**模块化的布局引擎:** 项目的核心 —— 动态平铺算法, 在设计上是模块化的。这意味着除了内置的动态平铺算法, 未来可以方便地引入或由社区贡献新的布局方式, 如斐波那契布局、三列式布局等, 用户只需在配置文件中切换即可。

**面向未来的可拓展接口:** Tiley 的长远规划是成为一个可拓展的平台。我们计划在未来版本中暴露脚本接口(例如使用 Lua), 允许用户编写脚本来响应事件、控制窗口行为, 实现超越配置文件能力的终极自动化和个性化定制。

## 2.2 问题解决流程: 从 wlroots 的转变

Tiley 在开发过程中, 技术选型经过过一次非常大的变动。在项目演进历程中, 一次最关键的决策便是从早期使用的 wlroots 框架迁移至 Louvre。转变的核心原因已经在之前的表格中列出, 它标志着 Tiley 从一个功能验证原型, 向一个面向长远发展的成熟项目的正式迈进。

### 2.2.1 转变的目的: 解决 wlroots 的痛点

在项目初期, 了解到 wlroots 作为 Wayland 合成器开发的事实标准, 并且具有非常容易上手的示例代码 [\[1\]](#), 为我们快速搭建原型、验证核心平铺算法提供了坚实的基础。然而, 随着我们对“现代化的用户体验”和“高度可拓展性”的设计目标不断深入, wlroots 的一些核心设计与我们的目标产生了难以调和的冲突。

**首先是渲染管线的局限性。** wlroots 提供了一个高度抽象且固定的渲染管线。这种设计简化了开发, 但牺牲了灵活性。它无法让我们注入自定义的 GLSL 着色器(Shader), 这意味着诸如窗口圆角、动态模糊背景、高亮边框光晕等对提升视觉品质至关重要的现代图形效果, 都无法实现。这直接阻碍了我们达成“现代化体验”这一核心设计目标。



其次是开发范式的冲突。 `Tiley` 自始至终都以 C++ 面向对象的思想进行架构设计, 这符合 `Wayland` 各种对象对应的接口, 以及我们核心的容器二叉树数据模型。而 `wlroots` 是一个纯 C 语言库, 其设计遵循 C 语言的事件驱动和模块化范式。在开发过程中, 我们必须编写大量的“胶水代码”, 一方面将 C 语言的结构体和回调函数包装成 C++ 类和方法, 另一方面则是对旧版本 C 语言标准的一种绕过机制(例如, [C99 标准的定长数组](#) `char array[N]`, 该特性无法和使用 `g++` 编译的 C++ 代码混合)。这种“阻抗不匹配”不仅增加了代码的冗余度, 更严重的是, 它破坏了 C++ RAII(资源获取即初始化)的特性, 使得内存管理变得复杂且容易出错, 与我们追求代码健壮性和可读性的目标背道而驰。

2.2.2 转变的过程: 有序的模块化迁移

为了确保迁移过程的平稳可控, 在有限的时间内尽可能多地复用我们的核心逻辑, 我们采取了“自顶向下、模块替换”的有序迁移策略, 充分利用了 `Louvre` 框架优秀的面向对象设计。整个过程按顺序分为如下步骤:

- 1. 搭建新骨架: 由于 `Louvre` 提供了参考模板代码 `louvre-templates`, 我们首先基于此搭建了一个最简化的合成器程序, 实现了基本的后端初始化(DRM/TTY 和 Wayland 嵌套模式)和初始化主事件循环。可以启动一个完全黑屏的合成器。
- 2. 按照协议对象迁移: 迁移的核心在于第一部分中提到的 `TileyCompositor::createObjectRequest` 工厂方法。在初期, 这个方法将所有对象的创建请求都直接转发给父类 `LCompositor`, 以返回默认对象。这样使得我们有了一个可以运行, 但行为完全是 `Louvre` 默认行为的空壳。
- 3. 逐个模块替换: 随后, 我们开始逐一创建 `Tiley` 的自定义子类来替换 `Louvre` 的基类, 并在这个过程中移植原有的业务逻辑。迁移的顺序遵循从“环境”到“核心”的原则:
  - ▶ 首先是 `Output`、`Seat`、`Keyboard` 和 `Pointer`, 用于首先完成对显示器和输入设备的管理;
  - ▶ 接着是迁移项目中最重要的一部分: `Surface` 和 `ToplevelRole`。将原先在 `wlroots` 信号回调中处理的窗口创建(map)、销毁(unmap)、几何信息设置等逻辑, 重构并移植到这些 C++ 子类的对应虚函数(例如, 将 `xdg_map_changed` 迁移到 `LSurface::mappingChanged` 中);
  - ▶ 然后重构核心算法: 将与框架无关的核心平铺算法(容器二叉树)从旧代码库中剥离出来, 并对其接口进行重构, 重新引入新的 `Container` 类, 使其能与新的 `Tiley::ToplevelRole` 对象无缝协作。
  - ▶ 主要的代替列表如下:

<code>wlroots</code>	<code>Louvre</code>
<code>new_output</code>	在 <code>Compositor::initialized</code> 中初始化
<code>new_toplevel</code>	<code>Surface::mappingChanged</code>
<code>new_popup</code>	<code>Surface::mappingChanged</code>
<code>cursor_motion</code>	<code>Pointer::pointerMoveEvent</code>

cursor_axis	<code>Pointer::pointerButtonEvent</code>
cursor_frame	对 <code>cursor()</code> 相关方法的调用
new_input	<code>Seat::configureInputDevices</code>
request_set_cursor	对 <code>cursor()</code> 相关方法的调用
request_set_selection	在 <code>Pointer</code> 类中相关方法的处理

这种渐进式的迁移方式，使得我们在每个阶段都有一个可编译、可运行、可测试的版本，极大地降低了风险，并使得调试工作可以聚焦在当次迁移的模块上。

### 2.2.3 迁移中遇到的主要问题

尽管迁移策略清晰，但在实践中我们依然遇到了一些挑战，克服这些挑战的过程也加深了我们对 `Wayland` 和图形系统的理解。

**首先是控制流的范式转变。** 最主要的问题是从 `wlroots` 的"信号-槽"事件模型到 `Louvre` 的"虚函数重载"面向对象编程模型的思维转变。例如，在 `wlroots` 中，我们通过监听一个全局的 `new_toplevel` 信号来响应新窗口的创建；而在 `Louvre` 中，则是在 `Surface` 的 `mappingChanged()` 虚函数被调用时进行处理。这要求我们必须从“被动监听事件”转变为“在对象生命周期的特定节点主动处理”，对代码的组织方式和逻辑流的理解提出了新的要求。

**其次是渲染逻辑的完全重写。** `Louvre` 将渲染的控制权完全交给了开发者，这既是优点也是挑战。我们不得不放弃 `wlroots` 内置的场景图，从零开始编写所有渲染代码。这包括使用 `glm` 库进行矩阵变换、管理 `OpenGL` 视口、自行封装 `VBO/EB0` 等缓冲对象、编写用于渲染窗口纹理和自定义边框的着色器程序。这项工作虽然繁重，但也正是这次彻底的重写，为后续 `Tiley` 实现丰富的视觉效果铺平了道路。

**最明显的，是协议细节的抽象差异。** 虽然两个框架都实现了标准的 `Wayland` 协议，但它们对协议的抽象层次和暴露给开发者的接口不尽相同。例如，`wlroots` 通过简单的函数进行窗口位置和大小配置：

```
wlr_xdg_toplevel_set_size(container->toplevel->xdg_toplevel, remaining_area.width, remaining_
wlr_scene_node_set_position_(get_wlr_scene_tree_node(container->toplevel->scene_tree), remaining_
```

其目标是整个 `toplevel`，包含了窗口的各个子表面和弹出窗口等。但在 `Louvre` 中，没有直接对"窗口"操作的方法，我们必须手动建立层级，将窗口的各个部分都跟随窗口移动：



```

if(surface->mapped()){
    // ...

    // 显示部分调整为“内部区域”
    container->containerView->setPos(areaForWindow.pos());
    container->containerView->setSize(areaForWindow.size());

    // 请求客户端也调整为“内部区域”
    surface->setPos(areaRemain.pos());
    container->window->configureSize(areaForWindow.size());
    container->window->setExtraGeometry({GAP, GAP, GAP, GAP});

    LLog::debug("containerView的children数量: %zu", container->containerView->children()
    SurfaceView* surfaceView = static_cast<SurfaceView*>(container->containerView->children().first());

    const LRect& windowGeometry = container->window->windowGeometry();

    // 如果窗口不支持服务端装饰, 并且windowGeometry有偏移
    if(!container->window->supportServerSideDecorations() && (windowGeometry.x() > 0
        // 则说明它大概率会无论如何都要画自己的装饰, 我们移动customPos, 将装饰部分移动到外边(不
        surfaceView->setCustomPos(-windowGeometry.x(), -windowGeometry.y());
    }

    compositor()->repaintAllOutputs();
}

```

这些都使得迁移工作不能简单地复制粘贴，而需要我们对照 Wayland 官方协议文档，仔细理解每个接口背后的协议行为，以确保实现的正确性。

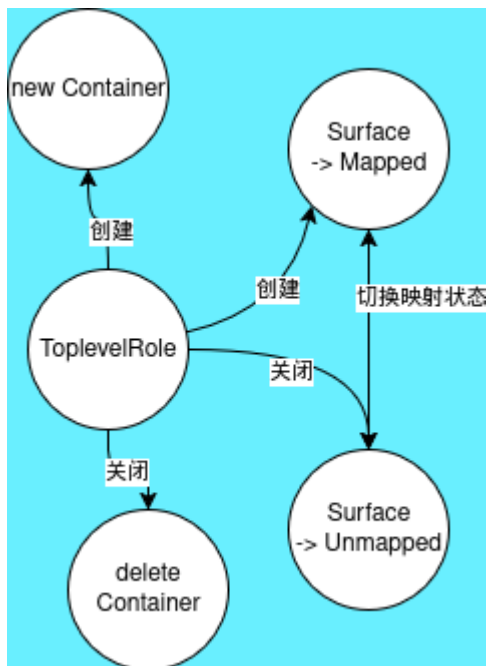
## 2.3 架构哲学: 面向对象的实践与展望

在确定了使用 `Louvre` 框架后，`Tiley` 的架构设计便完全建立在现代 `C++` 的面向对象(Object-Oriented Programming, OOP)哲学之上。这不仅是一种编码风格的选择，更是一种系统化的方法论，它深刻地影响了项目的代码结构、可维护性和未来发展潜力。这里的核心理念是：一个设计优雅的系统，其可拓展性、代码可读性和用户友好性必然同时具有。

### 2.3.1 将 Wayland 世界模型化: OOP 的核心实践

`Wayland` 协议本身就是围绕一系列“对象”及其接口定义的规范。因此，采用 OOP 范式是对 `Wayland` 世界最自然、最直观的建模。`Tiley` 中的每一个核心类，都对应 `Wayland` 协议中的一个实体，这种一一对应的关系极大地降低了开发者的心智负担。

**首先是职责单一原则(Single Responsibility Principle)。**这是我们在实践中严格遵守的首要原则。每个类只做一件事，并把它做好。例如：`Tiley::Surface` 类只负责管理与 `wl_surface` 相关的状态和纹理，它不关心这个表面是窗口、是弹窗还是鼠标指针。`Tiley::ToplevelRole` 类则专门处理“窗口”这一角色的逻辑，如最大化、最小化、全屏状态的管理，以及与平铺布局算法的交互。而平铺布局算法本身，则被封装在独立的 `TileyWindowStateManager` 类中，它对 `Wayland` 协议一无所知，只负责维护一个抽象的容器树结构。这种清晰的职责划分，使得代码高内聚、低耦合，修改一个模块的内部实现，不会意外地影响到其他模块。



然后是组合优于继承 (Composition over Inheritance)。虽然我们通过继承 `Louvre` 的基类来融入框架,但在 `Tiley` 内部的组件设计中,我们更倾向于使用组合。例如, `Tiley::ToplevelRole` 对象内部包含一个指向 `Container` 的指针,而不是让 `ToplevelRole` 去继承 `Container`。这样做的好处是更加灵活,它解耦了窗口的角色逻辑和布局逻辑,使得未来可以轻松地为同一个 `ToplevelRole` 切换不同的布局策略,而无需改变其继承关系。同时,数据模型和实体的分离也使得我们可以非常方便地测试逻辑部分,而无需真正的 Wayland 环境。

最后,是利用 C++20 特性提升代码质量: 我们积极使用 C++20 标准带来的新特性,以编写更安全、更简洁、更具表达力的代码。

- ▶ 智能指针: `std::unique_ptr` 和 `std::shared_ptr` 的使用,从根本上杜绝了手动 `new/delete` 可能导致的内存泄漏和悬垂指针问题。在合成器这种需要长期稳定运行的系统软件中,这一点至关重要。
- ▶ Lambda 表达式与 `std::function`: 在处理异步事件和回调时, Lambda 表达式让我们能够就地定义简洁的处理器,并捕获必要的上下文,极大地提升了代码的可读性和编写效率。例如快捷键绑定和注册系统的一个片段:

```
shortcutManager.registerHandler("launch_terminal",
    [](auto){
        LLog::log("执行: launch_terminal");
        // 打开终端的代码...
    });
```

这里,注册机制和 Lambda 回调的综合,使得我们可以非常方便地调用一些和上下文无关的方法,另一方面保证了和客户端交互的独立性。

### 2.3.2 架构驱动的良好循环: 可拓展、可读与用户友好的统一

**Tiley** 的架构设计旨在创造一个正向的、自我增强的良性循环, 在这个循环中, 可拓展性、代码可读性和用户友好性相互促进, 共同推动项目的健康发展。

**首先确保代码可读性:** 一个开发者如果无法快速读懂现有的代码, 就不可能在此基础上进行有效的拓展。我们通过遵循上述 OOP 原则、保持一致的编码规范、撰写关键模块的文档注释, 致力于让 **Tiley** 的代码库对新贡献者尽可能透明。当需要添加一个新功能时, 通过 **Louvre** 官方文档和 **Tiley** 的类名命名, 清晰的模块划分能让开发者迅速定位到需要修改的代码区域, 从而降低贡献门槛。

**然后就能保障可拓展性:** 用户的需求是不断变化的, 桌面环境的潮流也在不断演进。一个僵化的、难以拓展的系统, 其用户体验很快就会过时。**Tiley** 的模块化架构和未来规划的脚本接口保证了我们能够快速响应用户需求和社区创意。无论是支持一个新的 **Wayland** 协议扩展, 还是实现一种全新的视觉效果, 良好的可拓展性都能让我们以最小的成本、最低的风险完成迭代, 持续提升用户友好性。

**对用户友好性的追求反哺架构设计:** 当我们设定了如“提供流畅的窗口动画”这样一个对用户友好的目标时, 它会反过来对架构提出更高的要求。为了实现这个目的, 我们必须设计一个高效的渲染循环、一个能够解耦逻辑与渲染的事件系统, 以及一套能够精确描述动画过程的数据结构。这种来自用户体验层面的需求, 是驱动我们不断优化底层架构、提升代码质量的最强大动力。

最终, **Tiley** 的架构哲学可以归结为: 通过构建一个对开发者友好的系统, 来最终实现一个对用户友好的产品。我们相信, 一个内部优雅、结构清晰的项目, 才能拥有持续进化的生命力, 才能在不断变化的技术浪潮中, 为社区贡献长久的价值。

## 3. 核心功能设计

### 3.1 Louvre 框架介绍

关于 **Louvre** 框架的基本信息以及其针对协议编写的各个封装类, 前面都有提到, 此处不再赘述, 更多描述关于 **Louvre** 的一些特性。

#### 3.1.1 虚函数继承: 完全面向对象开发

**Louvre** 充分践行现代 C++ 的 **OOP** 开发范式。对于大多数情况而言, 我们需要做的就是明确客户端和服务端的交互流程, 并依次为据重写/实现父类相关函数来达到目的。以实现显示屏相关功能( **LOutput** 类) 为例:

```
class Output final : public LOutput{
public:
    // 屏幕插入
    void initializeGL() override;
    // 绘制到屏幕
    void paintGL() override;
    // moveGL, resizeGL类似wlroots中的request_state事件
    // 移动屏幕的index
    void moveGL() override;
    // 屏幕缩放变化, 尺寸变化
    void resizeGL() override;
    // 类似wlroots中的output_destroy, 屏幕拔出
    void uninitializedGL() override;

    //...
}
```

这里, 所有的非纯虚方法都在父类中有一定实现(例如 **PaintGL** ), 我们参考父类实现编写子类功能即可; 对于纯虚方法(例如 **moveGL** ), 我们根据功能本身(在逻辑上移动一块屏幕的显示位置), 根据移动屏幕的目标位置移动内部绘制的内容即可。这里反映的就是 **Louvre** 的特点: 将 **Wayland** 事件封装到一个个独立的方法中, 在合适的时候触发它们。这基于 **Wayland** 最本质的特性:

**Wayland** 交互的逻辑本质上是一个协议定义的事件循环。

基于这一点, 我们就可以对每个事件分别处理, 仅在服务端合适的位置记录状态即可, 无需依赖客户端稳定的行为。

其他类的逻辑类似, 例如 **Surface** 类就是处理一个表面的生命周期(提交、更改映射状态、顺序发生变化等)、 **Pointer** 类就是处理鼠标的各种按钮和移动事件。

#### 3.1.2 在现代化硬件上测试并运行

这里要说的不是 **Tiley** 的测试和运行环境, 而是 **Louvre** 在文档中说明的测试环境。据[官方文档](#), **Louvre** 本身的测试环境是:

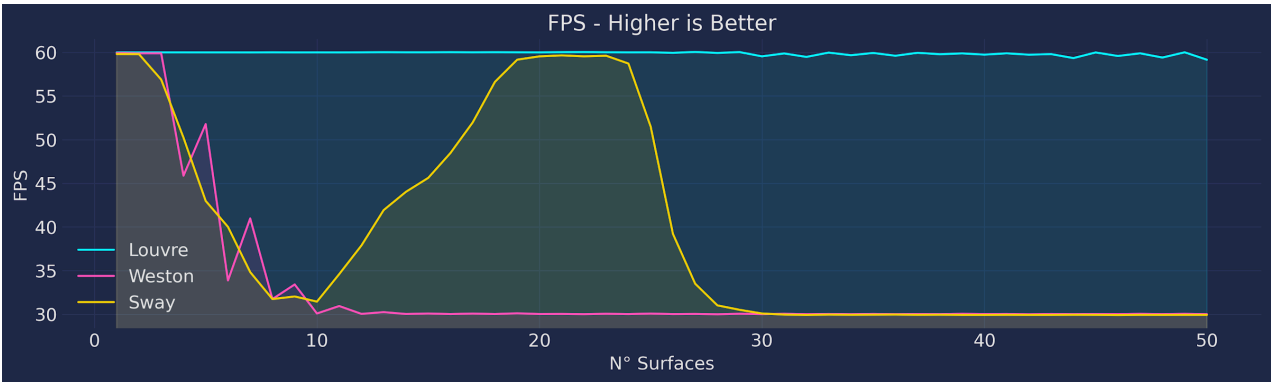
Benchmark Environment	
Machine	MacBook Pro A1398 (Retina, 15-inch, Mid 2015)
CPU	Intel Core i7-4770HQ @ 2.20GHz (up to 3.4GHz) with 6MB shared L3 cache
Memory	16GB of 1600MHz DDR3L
GPU	Intel Iris Pro Graphics - i915 (Intel Graphics) version 1.6.0 (20201103)
Display	15-inch Retina Display with single mode 2880x1800@60Hz
OS	Linux Mint 21 - Linux 5.15.0-86-generic

是一个相对现代的运行环境。在这样的环境下控制变量进行测试, **Louvre** 都取得了不错的成绩:

### 3.1.3 高性能

下面的截图来自 **Louvre** 官方仓库, 对比了 **Louvre** , **Weston** 和 **Sway** 的性能:

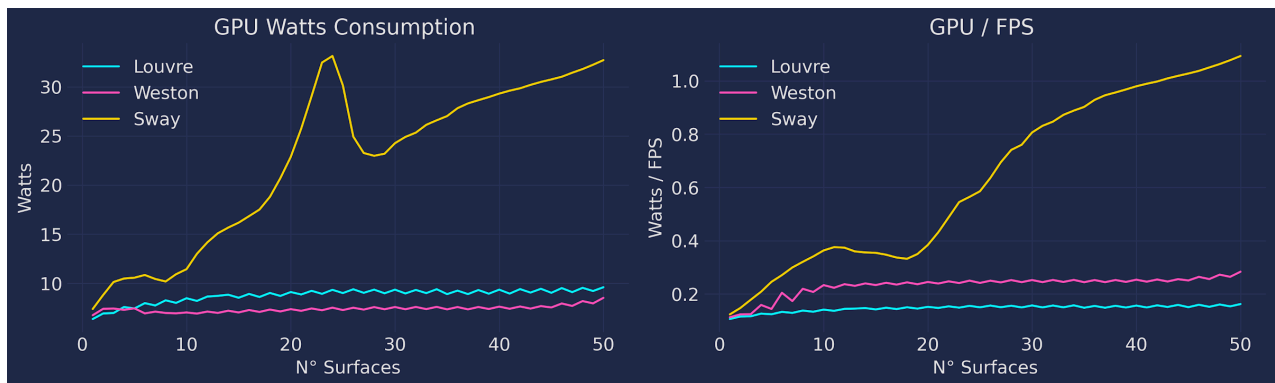
FPS - 测试图像



CPU/FPS 比 - 测试图像



GPU/FPS 比 - 测试图像



## 3.2 平铺功能

### 3.2.1 窗口分层和浮动模式

在 **wayland** 协议和 **Louvre** 框架中, 窗口分层遵循五层协议, 从下到上(下面以 **Louvre** 中的命名表示):

#### 1. LLayerBackground (背景层):

- ▶ 壁纸和桌面背景: 桌面环境的背景图片或颜色
- ▶ 系统级背景元素: 登录界面的背景

#### 2. LLayerBottom (底层):

- ▶ 桌面图标: 文件管理器在桌面上显示的图标和文件
- ▶ 桌面小部件: 如时钟、天气预报等桌面组件

#### 3. LLayerMiddle (中间层):

- ▶ 普通应用窗口: 大部分应用程序的主窗口
- ▶ 文档编辑器、浏览器、文件管理等常规应用
- ▶ 对话框和模态窗口: 应用程序的设置窗口、文件选择器等
- ▶ 子窗口: 应用程序的工具栏、侧边栏等

#### 4. LLayerTop (顶层):

- ▶ 系统通知: 桌面通知弹窗
- ▶ 系统对话框: 如权限请求、系统设置对话框
- ▶ 窗口管理器的 UI 元素
- ▶ 输入法窗口: 候选词列表、输入法状态指示器
- ▶ 上下文菜单: 右键菜单、应用程序菜单

#### 5. LLayerOverlay (覆盖层):

- ▶ 全屏覆盖: 屏幕保护程序、锁屏界面
- ▶ 系统级 UI: 电源管理对话框、紧急模式界面
- ▶ 调试和开发工具: 性能监控覆盖、开发者工具
- ▶ 鼠标指针: 鼠标 **cursor** 在 **Louvre** 中位于此层

因此, 无论是浮动的窗口还是平铺的窗口, 都是合成器内部自己管理的行为, 而非协议规定。这就导致我们一开始的实现是有问题的。

### 3.2.2 层级关系模型

正如上面提到的那样, 在 `Louvre` 的世界中, 层级分为 5 层, 那么相应地, 我们需要有调整层级的方法, 这便是数据模型和视图分离的另一个体现:

#### 提示

`Wayland` 负责发送 `Surface` 到合成器, 而合成器为每个 `Surface` 创建一个 `view` 进行显示。

对于一个窗口内的内容, `Surface` 的顺序是客户端决定的, 而对于不同窗口的内容, 则遵循先来后到的原则, 我们也可以调用 `LSurface::raise` 方法来提升一个 `Surface` 到顶层。对于一般情况而言, 黄金法则就是:

我们只需要遵循 `Wayland` 提供的 `Surface` 顺序即可, `View` (视图)是我们为 `Surface` 创建的显示区域, 只需忠实地反映 `Surface` 的顺序。

在 `Louvre` 中, `View` 同样是数组形式的存在, 且越靠后的越后渲染(在上层), 这和 `Surface` 形式相同。所以, 我们只需利用 `Louvre` 提供的两个方法:

- ▶ `LView::insertAfter`: 用于将一个 `View` 在数组中插入到另一个 `View` 后面, 在视觉上更接近顶层;
- ▶ `LView::setParent`: 用于设置一个 `View` 的父 `View`, 并放置到那个父 `View` 层级下的数组末尾, 也就是视觉上的最顶层。

这可以用图像处理软件里的图层来理解。一个 `Parent` 就是一个图层组, 里面包含的图层一起上移/下移, 相对于其他组外的图层而言关系是同上同下的, 即移动 `Parent` 会移动所有子图层的顺序; 而 `insertAfter` 则是调整单个图层的顺序。最后, 软件按照图层展平后从上到下的顺序绘制。

### 3.2.2 分层问题解决: `Surface` 消失的问题

上面的理论具备之后, 让我们来回顾一个项目初期存在的问题。项目一开始, 在 `Surface` 创建时, 曾采用过将"浮动"的放置到"中间层"的上一层, 也就是 `LLayerTop` 的方案。但这导致了 `Surface` 渲染异常, 常常出现浮动的窗口消失的问题, 下面是当时的代码:



```

void Surface::orderChanged(){
    LLog::log("%d: orderChanged", this);
    LSurface::orderChanged();
    if(toplevel()){
        LLog::log("是窗口");
        TileServer& server = TileServer::getInstance();
        view.setParent(&server.layers()[TILED_LAYER]);
    }else{
        LLog::log("是窗口中的子surface");
        Surface* prev = static_cast<Surface*>(prevSurface());
        view.insertAfter(prev && (prev->layer() == layer()) ? &prev->view : nullptr);
    }
}

```

可以看见, 当时我们将 **窗口** 和 **窗口的子Surface** 区分对待, 导致 **窗口的子Surface** 没有调用对应的 **view.setParent**, **View** 没有严格遵守 **Surface** 的顺序, 因此会出现渲染消失问题, 弹出菜单都不可见。理解了这个模型之后, 我们修改成了下面的代码:

```

void Surface::layerChanged(){
    //LLog::debug("%d: 层次改变: ", this);
    TileServer& server = TileServer::getInstance();
    getView()->setParent(&server.layers()[layer()]);
}

```

通过将逻辑移动到 **Surface::layerChanged** 中(该方法先调用, 在下面的**Surface 的生命周期**章节说明), 确保所有 **Surface** 都有一个 **Parent**, 才能在后续的 **orderChanged** 中正确排序。

### 3.2.3 平铺算法基本逻辑

将窗口放置到正确的层级之后, 对于平铺算法, 我们需要处理的问题就简化了。针对现在市面上已有的平铺管理器( **Hyprland**, **Sway** )存在的功能, 在平铺功能上, 我们需要实现:

#### 信息

- ▶ 能使窗口平铺显示, 即窗口之间不重叠;
- ▶ 能使用户在"平铺"和"浮动"两种状态之间切换一个窗口的状态;
- ▶ 能使窗口以合理的布局显示, 正常情况下不至于太小或太大、难以操作等。

为此, 我们引入了二叉容器树数据结构, 作为我们平铺算法的核心数据结构, 同时实现了对该数据结构的增删改查操作。结合容器树结构和上面解决的层级问题, 我们需要的就是:

- ▶ 我们需要筛选出是 **ToplevelRole** (窗口角色)的 **Surface**, 并对它们进行布局;
- ▶ 我们需要在同一层内正确处理浮动窗口和平铺窗口的逻辑;
- ▶ 我们需要有合适的方法, 将用户对窗口的操作对应到容器树的变化上。



### 提示

关于平铺算法的具体实现, 包括新窗口插入、窗口分离(用于移动或者浮动)等具体方法, 将在后文[平铺算法中的核心: 动态平铺](#)部分详细说明。

## 3.3 窗口管理

关于窗口管理, 我们也是在不断适应 [Wayland](#) 复杂的交互环境的过程中不断更新窗口管理的逻辑。对于窗口管理, 我们实现的是:

### 信息

- ▶ 能正确裁剪窗口大小, 使得窗口显示适应平铺布局分配的区域;
- ▶ 能自动识别窗口的类型, 分配不同的管理策略;
- ▶ 用户可以移动窗口, 调整窗口大小

### 3.3.1 溢出裁剪的问题: 客户端需要的尺寸和平铺分配的冲突

对于窗口管理, 由于我们在项目初期一直使用简单的终端应用进行功能测试, 在引入非终端应用测试后, 窗口尺寸和分配区域不适配的问题就非常明显。终端应用(如 [foot](#)、[alacritty](#) 和 [weston-terminal](#)) 的主要特点是:

- ▶ 尺寸一般可自由调整, 较少受到显示内容的限制
- ▶ 界面内容单一, 一般只有窗口主 [Surface](#) 一个表面

因为这两个特点的存在, 终端类应用成为了 [Tiled](#) 初期时用于测试的最佳应用; 但随着开发的进行, 逐渐成为新功能的障碍:

- ▶ 因为尺寸可自由调整, 所以对于平铺算法分配的区域, 终端类客户端会接受; 但对于内容丰富的客户端(如 [GIMP](#) (一个图像处理软件)等), 由于其需要显示其内部各类控件、用户交互内容, 对于过小的尺寸其不会接受, 导致溢出分配的区域;
- ▶ 因为界面单一, 所以对于有诸多弹出菜单、子表面的应用而言, 我们无法全面保证其弹出菜单也会正常显示。

因此, 在项目的后期, 我们引入了除了终端以外的其他应用作为测试, 以期尽可能反映真实用户使用场景, 并测试功能在各种环境下的稳定性:

- ▶ [weston-terminal](#): 朴素的终端应用, 新功能最优先测试的客户端。
- ▶ [GIMP](#): 开源的图像处理软件, 拥有丰富的画布界面和各式弹出窗口、弹出菜单, 且属于 [GTK](#) 应用(参考上文//TODO)。
- ▶ [Wireshark](#): 开源的网络数据包分析软件, 拥有数据不断变化的界面, 属于 [QT](#) 应用程序。
- ▶ [Firefox](#): 各大 Linux 发行版默认的浏览器软件, 拥有非常丰富的显示内容, 且属于用户常用软件。

随着开发的不断完善, 我们还会引入 `electron` 框架开发的软件等(注: 该框架在 `wayland` 下一直表现不佳, 参考链接: [electron apps are buggy and laggy on wayland with proprietary nvidia drivers](#)), 不断提升合成器的健壮性。

回到溢出裁剪的问题, 我们正是在引入 `GIMP` 后发现窗口会溢出分配的平铺区域。

### 3.3.2 平铺窗口的视图层级

为解决上面提到的溢出问题, 我们引入了一层 `ContainerView`, 继承 `LLayerView`, 用于将窗口裁剪到平铺区域。 `LLayerView` 是一种完全不可见的视图容器, 非常适合作为父容器对其中的子视图进行裁剪。引入后, 对于每个窗口, 显示层级如图所示:

```
SceneView
|- LayerView
   |- ContainerView
      |- SurfaceView
```

其中, `SceneView` 是整个场景的视图, `LayerView` 是 5 层模型中间层的视图, `ContainerView` 即是那个用于将窗口通过一切手段(包括但不限于配置大小、强制裁剪等)裁剪到平铺分配的区域视图, `SurfaceView` 是真正的窗口视图。

#### 提示

关于窗口裁剪的具体实现, 将在后文[窗口尺寸裁剪机制](#)部分详细说明。

### 3.3.3 管理形式适应窗口类型

除了使用一个 `ContainerView` 来对溢出的窗口进行裁剪以外, 对于过小的窗口, 我们也进行了响应的处理。

由于弹出窗口或客户端限制尺寸的窗口等大小不允许调整, 我们匹配了现在通行的做法: 将这类窗口自动提升为"浮动"模式而不插入平铺布局。这样做是考虑到用户体验: 用户不需要所有窗口都是平铺的, 将大小有限制的窗口强行进行平铺反而会损害用户的交互体验。

在代码实现上, 下面的片段简要体现了主要的操作:

```

void ToplevelRole::assignToplevelType(){
    bool userFloat = /*从配置中读取用户之前保存的浮动状态*/
    // 如果有尺寸限制
    if(hasSizeRestrictions()){
        this->type = RESTRICTED_SIZE;
        // 如果是子窗口或者用户指定
    }else if(surface()->parent() || userFloat){
        this->type = FLOATING;
        // 否则就是普通窗口
    }else{
        this->type = NORMAL;
    }
}

```

#### 提示

基本上, 代码通过 `hasSizeRestrictions` 对是否有尺寸限制进行了判断, 具体方法将会在后文[识别窗口类型](#)决定是否加入平铺布局部分详细说明。

### 3.3.4 用户和窗口的交互: 移动和调整大小

有了容器树作为基础数据结构, 用户对窗口的大小调整或移动就变得简单了。此处对我们的实现方式进行简单概括:

- ▶ **移动窗口:** 对客户端传来的"移动窗口"信号进行捕捉, 同时加以判断是否符合移动条件。如果符合, 并且是平铺层的窗口, 则执行: 从容器树分离->准备好移动参数->跟随客户端更新窗口位置->当可移动条件不再满足->合并回容器树->停止移动; 不是平铺层的窗口则省略开始的分离和最后的合并步骤。
- ▶ **调整窗口大小:** 对客户端传来的"调整大小"信号进行捕捉, 同时加以判断是否符合调整大小条件。如果符合, 并且是平铺层的窗口, 则执行: 准备好调整参数->跟随客户端更新窗口大小->同时通知容器树中符合条件的父容器改变大小->当可调整大小条件不再满足->停止调整; 不是平铺层的窗口则省略通知父容器改变大小的步骤。

#### 提示

这部分的交互逻辑稍复杂, 具体逻辑将会在后文[窗口大小调整和移动](#)部分详细说明。

## 3.4 用户交互模型

### 3.4.1 焦点跟随鼠标

市面上大部分的平铺式管理器都引入了"焦点跟随鼠标"(focus-follow-cursor)机制, 我们也需要按惯例引入该功能。此功能在平铺式管理器下非常方便, 因为窗口以平铺形式呈现, 用户移动鼠标进入一个目标窗口多数情况下都不会因为窗口堆叠次序变化而导致副作用(在堆叠式管理器中焦点跟随鼠标会很大程度

上影响体验, 因为窗口以堆叠形式呈现, 移动到后面的窗口可能只是为了拖拽一个文件图标, 并非想要将窗口提前)。

#### 提示

焦点跟随鼠标的实现较为简单, 将在后文[窗口焦点跟随鼠标](#)部分详细说明。

### 3.4.2 "活动容器"设计

动态平铺算法的一个特点是, 新窗口的位置总是跟随用户的鼠标或键盘焦点, 也就是:

- ▶ 用户移动鼠标到一个新窗口上, 焦点改变到对应窗口 A;
- ▶ 此时用户打开新窗口 B, 新窗口 B 将根据 A 的几何尺寸和 A 进行布局平分。

为此, 引入"活动容器"设计是非常必要的。由于窗口环境的多变、用户焦点的多变等等, 我们必须使用一种机制, 使得逻辑容器和用户交互环境之间存在一个更新中介进行缓冲, 而不是在环境发生改变时立即更新逻辑容器。设想下面的情况:

- ▶ 用户在快速移动鼠标, 并快速按下打开终端快捷键, 在鼠标经过的各个位置都打开一个终端;
- ▶ 因为客户端窗口无响应, 导致两个窗口之间出现空隙, 此时用户尝试在空白区域打开一个新的窗口;
- ▶ 用户刚启动合成器或切换到一个从未使用的工作区, 并且尚未移动鼠标或键盘输入等(焦点未刷新), 此时用户尝试开启一个窗口。

上述情况如果处理不当, 将会导致窗口出现意外布局, 甚至因为空指针等导致合成器崩溃。

要在用户多变的操作中寻找一种稳定性, 我们只需要保证:

#### 信息

无论用户的聚焦状态如何, 是否有键盘输入或移动鼠标, 新的窗口总可以找到一个放置的位置。

"活动容器"在用户的各种交互事件中被更新, 并遵循以下规范:

- ▶ 当"活动容器"为 `nullptr` 时, 下一个新窗口的布局目标一定是桌面;
- ▶ 当"活动容器"不为 `nullptr` 时, 下一个新窗口的布局目标容器一定存在而非空。

在代码上, 由于"活动容器"具有副作用, 将使用一个 `setActiveContainer` 方法, 进行适当的入参检查等:

```

void TileWindowStateManager::setActiveContainer(Container* container){
    if(!container){
        activeContainer = container; //设置为空容器
        return;
    }

    UInt32 workspace = getWorkspace(container);
    if(workspace >= 0 && workspace < WORKSPACES){
        workspaceActiveContainers[workspace] = container;
    }

    // 同步更新, 需要随时更新activeContainer和workspaceActiveContainers, 让他们保持同步
    activeContainer = workspaceActiveContainers[workspace];
}

```

其中, `workspaceActiveContainers` 是个工作区的上一个活动容器, 用于解决切换工作区后的目标问题和用户在非当前工作区打开窗口的问题; `activeContainer` 是当前活动的工作区, 是新窗口布局目标的唯一参照物, 需要遵循上面提到的规范。

#### 提示

关于具体的实现, 将在后文["上一个活动容器"系统部分](#)中详细说明。

### 3.4.3 全键盘操作

全键盘操作的引入也是为了与市面上流行的平铺管理器保持一致。在平铺式布局下, 使用键盘的操作便利度将远大于鼠标。各个窗口同时显示, 并且焦点跟随用户操作移动, 非常方便用户并列两个文本编辑器同时编辑等使用场景。

为了全键盘操作(也是快捷键系统的实现), 我们引入了快捷键管理器 `tiley::ShortcutManager`。这个管理器拥有如下功能:

- ▶ 从用户配置加载快捷键列表并存入一个 `map` 数据结构。该 `map` 数据结构是多线程安全的, 防止用户短时间内频繁修改快捷键列表导致加载不完整等情况。
- ▶ 为每个快捷键注册目标操作。目标操作目前使用 `Lambda` 匿名函数传入, 与上下文无关, 保证每个操作的独立性; 后续将会改为 `Action` 类, 提供更加丰富的获取操作名称、可配置的前置操作和后置操作等功能。
- ▶ 动态监控快捷键列表的更新。使用 `Linux` 内核提供的 `<sys/inotify.h>` 头文件中的监控文件变化相关函数, 实时监控用户对快捷键列表的修改并加载。

#### 提示

关于键盘快捷键相关具体实现, 将在后文[全键盘操作](#)部分中详细说明。

## 3.5 工作区

工作区是平铺管理器中一个非常重要的概念。由于窗口在整个平铺空间内平铺放置(除屏幕保留区域外,例如顶栏)而不重叠,且屏幕大小有限,我们无法在一个屏幕内放太多的窗口,需要使用工作区机制将窗口按需显示。

`Tiley` 使用切换窗口 `View` 可见性的机制进行工作区管理。关于这个选择,在项目开发过程中经历过一番波折。

### 3.5.1 "视图可见性" vs "map/unmap"

在项目刚开始实现工作区时,我们曾有一个非常朴素而直观的想法:隐藏一个窗口 = 将窗口 `unmap`。事实证明,这样的方法会带来非常多的问题,包括但不限于:

- ▶ `map / unmap` (映射/取消映射)是协议操作。当客户端通过协议请求映射一个 `Surface` 时,往往代表该应用即将退出,结束其生命周期;但我们只是因为切换工作区而在服务端隐藏一个窗口;
- ▶ 如果我们主动取消映射一个窗口,由于 `Wayland` 的安全协议限制,某些客户端可能不会遵守指令(因为这代表合成器想要主动关闭一个窗口),导致该方法不稳定。

事实上,在我们进行该方法的尝试时,就出现过多数应用意外退出的情况。由此, `Tiley` 转向使用只在合成器端存在的 `View` 系统实现工作区,该方法的好处是:

- ▶ `View` 只在合成器端存在,操作视图的可见性对客户端的 `Surface` 是否提交到合成器无影响,即隐藏 `View` 不会导致 `Surface` 发生变化;
- ▶ 当 `View` 不显示时, `Louvre` 将在后台对该 `View` 的 `buffer` 进行自动管理,停止提交该 `buffer` 到 GPU,同时节约内存。

#### 提示

关于具体实现细节,将在后文[工作区管理](#)详细说明。

### 3.5.2 逻辑上数量无限制

`Tiley` 的工作区在设计时就对数量没有限制。具体而言,我们使用下面两个步骤保证数量是理论无限的:

- ▶ 使用 `vector` 容器保存每个工作区的根节点:

```
std::vector<Container*> workspaceRoots{WORKSPACES};
```

此处的 `WORKSPACES` 默认为 10,是初始工作区的数量,这是 `Hyprland` 的默认设置,较为合理,我们直接沿用;

- ▶ 在切换工作区时,引入缓存机制。将分为两种情况:
  - ▶ 如果切换前的工作区在前 10 个范围内,无论是否拥有窗口,切换后将不会回收内存;
  - ▶ 如果切换前的工作区在前 10 个范围之外,且该工作区没有窗口,切换后将回收内存;如果有,则不会回收内存。

每个工作区都有自己的平铺容器根节点, 这保证了布局的独立性。

### 3.5.3 工作区布局记忆

关于布局记忆, 该功能目前尚处于正在开发状态。但我们已经完成了下面的工作, 使得届时布局记忆可以非常顺畅地引入:

- ▶ 为每个窗口对象分配一个 `workspaceId` 成员属性, 而不是仅有平铺布局的窗口才有该属性(即: 分配工作区/切换工作区时仅仅修改 `Container` 的父节点; 尺寸被限制的窗口没有 `Container`。), 保证该窗口明确属于一个工作区, 且该属性可被序列化到文件中保存, 下次启动可以加载布局。
- ▶ 配置文件的统一管理、加载和保存。我们引入 `ConfigManager` 类, 专门用于管理合成器内部需要的各种配置文件, 包括键盘快捷键、保存的工作区布局、壁纸配置、用户配置等。

## 3.6 界面和自定义部分

### 3.6.1 图形化设置界面

我们计划引入图形化的设置界面, 这是 `Tiley` 走向便于普通用户使用的重要一步。该图形化设置界面将使用 `QT` 开发, 作为用户和配置文件之间的媒介, 避免普通用户直接修改配置文件可能会带来的问题。对于高级用户, 我们仍然保留实时监控配置文件变化的特性, 以便在用户直接修改配置文件时也可以立即刷新配置。

### 3.6.2 快捷键自定义

快捷键自定义是用户自定义的最重要的一个部分。对于该功能上文已进行比较详细的描述, 也将在下文详细说明这部分的两层模块化实现。

### 3.6.3 窗口效果动画

该部分仍然在实现中。不过, 依托 `Louvre` 提供的便捷动画功能, 我们可以轻松实现各种动画, 下面是目前正在使用的屏幕插入时会触发的动画:

```

LAnimation::oneShot(1000, [weakRef](LAnimation* anim){
    if(!weakRef){ //如果中途因为任何问题导致weakRef这个弱指针包装器变空了(比如, 屏幕被拔出), 则立刻
        anim->stop();
        return;
    }

    // 否则正常执行
    weakRef->fadeInView.setPos(weakRef->pos());
    weakRef->fadeInView.setSize(weakRef->size());
    weakRef->fadeInView.setOpacity(1.f - powf(anim->value(), 5.f));
    weakRef->repaint();
},
```

下面是我们即将实现的窗口效果动画列表:



对象	动画类型
窗口	窗口切换平铺/浮动动画
窗口	窗口打开动画
窗口	窗口关闭动画
窗口	窗口切换工作区动画
工作区	工作区切换动画
屏幕	屏幕缩放调整动画

后续, 还将通过暴露 `Lua` 脚本接口的形式或通过纯配置文件的形式, 让用户可以声明式地应用自定义的动画。

### 3.6.4 窗口装饰自定义

由于 `Louvre` 使用 `OpenGL ES 2.0` 作为渲染协议, 我们也可以在 `Tiley` 中使用自定义 `shader` 渲染脚本, 进行渲染自定义。这也是从 `wlroots` 迁移到 `Louvre` 的一个重要目的。目前, `Tiley` 已经实现了窗口边框和窗口圆角, 并即将支持自定义配置。具体步骤如下:

1. 在 `TileyServer` 中, 我们创建了一个渲染脚本管理对象, 该对象将在 `Tiley` 被启动时初始化, 并从指定位置加载渲染脚本, 然后连接成渲染管线。
2. 重写 `SurfaceView::paintEvent` 方法, 当渲染对象是窗口时, 引入自己的渲染管线。

在开发自定义窗口装饰时, 我们遇到了不小的障碍, 包括 TTY 模式和嵌套模式渲染效果不同、窗口出现闪烁、撕裂甚至翻转的情况。具体问题讨论过程和最终的解决方案可查看我们在开发过程中与 `Louvre` 作者在 Github 上的讨论 [🔗](#)

#### 提示

具体实现将在后文[渲染流水线定制](#)详细说明

---

到这里, `Tiley` 的所有基本功能就介绍完毕了。各项功能的具体介绍将在后文详细完善。



## 4. 和其他平铺式管理器的比较

在开始介绍具体功能实现的细节之前, 我们想就 `Tiley` 的开发初衷做出一些说明, 并与现在市面上的合成器进行比较。 `Tiley` 的开发并非意在全盘否定现有方案, 而是希望在巨人肩膀上看得更远。

我们对市面上主流的平铺式管理器, 尤其是经典的 `i3`、`Wayland` 生态的先驱 `Sway` 以及视觉效果丰富的 `Hyprland` 进行了深入的学习。 `Tiley` 的设计和实现, 正是在汲取它们优点的同时, 针对性地解决我们认为可以改进的方面, 从而提供一种差异化的选择。

### 4.1 框架与架构的现代性

桌面环境的底层框架决定了其性能上限、可拓展性以及长期维护的潜力。我们相信, `Tiley` 在这方面做出了具有前瞻性的选择。

#### 4.1.1 技术栈的代际优势 (同 `Sway`, `i3` 相比)

`i3` 作为 `X11` 时代的几乎唯一流行的平铺式管理器, 其架构已历经十余年。 `Sway` 作为其在 `Wayland` 上的继任者, 继承了其大部分设计并基于 `wlroots` 构建。而 `Tiley` 则选择了 2023 年正式发布的 `Louvre` 框架。这一选择带来了显著的优势:

**现代语言特性。** `Louvre` 基于 C++20 标准开发, 使得 `Tiley` 可以全面拥抱现代 C++ 带来的内存安全 (RAII 与智能指针)、高表达力 (Lambda 表达式、Concepts) 和丰富的标准库, 从根本上提升了代码质量和开发效率。

**原生面向对象:** 与 `wlroots` 的 C 语言范式不同, `Louvre` 是一个纯粹的面向对象框架。这使得 `Tiley` 的架构能够与 `Wayland` 协议的对象模型完美契合, 代码结构更清晰, 逻辑更内聚, 极大地降低了项目的复杂度和维护成本。

**高性能设计:** `Louvre` 在设计之初就充分考虑了高性能场景, 其事件处理和渲染机制经过精心优化, 旨在实现最低的延迟和最高的吞吐量。

#### 4.1.2 渲染自由度与性能的兼顾 (同基于 `wlroots` 的合成器相比):

这是 `Tiley` 最核心的技术优势之一。基于 `wlroots` 的合成器 (如 `Sway`) 受限于其固定的渲染管线, 自定义视觉效果方面存在天然的桎梏。而 `Tiley` 依托 `Louvre`, 实现了两种渲染模式的融合:

对于常规窗口内容, 可以使用 `Louvre` 内置的高性能场景图进行渲染, 保证效率。对于需要自定义效果的部分 (如窗口边框、背景模糊), 则可以无缝切换到手动绘制模式, 注入自定义的 `OpenGL` 渲染逻辑和 `GLSL` 着色器。这种“混合渲染”模式, 让 `Tiley` 在保证基础性能的同时, 拥有了几乎无限的视觉定制潜力, 这是其他框架难以比拟的。

为了量化性能优势, 我们进行了一系列基准测试。当测试结果稳定后, 将在此处展示 `Tiley` 与其他合成器在特定场景下的性能对比。

### 4.2 人性化与开箱即用的体验

我们认为, 强大的功能不应以牺牲易用性为代价。 **Tiley** 在设计上致力于降低用户的上手门槛, 提供更加人性化的交互体验。

内置图形化设置界面 (同 **Hyprland**, **Sway** 相比): **Hyprland** 和 **Sway** 都拥有强大的配置能力, 但这通常需要用户直接编辑复杂的文本配置文件。这对于新手用户来说门槛较高, 且容易因配置错误导致问题。

**Tiley** 则计划提供一个官方的、内置的图形化设置应用:

**降低门槛:** 用户可以通过直观的图形界面完成大部分常用设置, 如快捷键绑定、颜色主题、动画效果等, 无需学习配置文件语法。

**保证兼容性:** 由官方提供的设置工具, 可以确保生成的配置项与合成器版本完全兼容, 避免了因使用第三方配置工具可能带来的版本不匹配问题。

**双轨并行:** 对于高级用户, 我们依然保留并支持通过直接编辑 JSON 配置文件进行深度定制。图形化界面与配置文件将实现双向同步, 满足不同层次用户的需求。

## 4.3 原生的工作区布局记忆

在许多平铺管理器中, 重启会话后恢复上一次的窗口布局, 通常需要依赖第三方脚本或插件, 这些方案往往不够稳定或配置复杂。 **Tiley** 将布局记忆作为一项核心的原生功能进行设计:

**无缝恢复:** **Tiley** 会在会话结束时, 自动将被标记的应用窗口布局(包括其所在工作区、大小和浮动状态)序列化并保存。

**智能匹配:** 在下一次启动时, **Tiley** 会在应用窗口创建时识别它们, 并自动将其恢复到上次关闭时的位置和状态, 真正实现无缝的工作环境恢复。

**官方集成:** 作为一项内置功能, 其稳定性和集成度远非第三方脚本可比, 为用户提供了连贯、可靠的多任务工作流体验。

## 5. 文档完整性说明

---

### ! 重要

以上便是 **Tiley** 的功能初步介绍和与其他合成器的对比, 仅为文档的一部分。一旦本轮开发过程最终完善(截止到决赛截止日之前), 将完善接下来的章节, 其中将对各项功能实现进行详细的说明。