Android图形绘制原理

1、图形系统整体架构

应用层 (App)

│

├── Canvas / OpenGL ES → 生产图形数据

│

系统层 (System Server)

├── WindowManagerService (WMS)：管理窗口层级、位置

├── SurfaceFlinger (SF)：合成所有Layer

│

硬件抽象层 (HAL)

├── Gralloc：分配图形缓冲区内存

├── HWC (Hardware Composer)：决策合成策略

│

内核层 (Kernel)

├── DRM/KMS 驱动：控制显示控制器扫描输出

│

显示硬件 (Display)

└── 屏幕物理输出

数据流向：

应用绘制 → Surface → BufferQueue → SurfaceFlinger → HWC → 屏幕

关键流程解析：

* 1. 应用层绘制流程（CPU/GPU协作）

流程如下：

ViewRootImpl

│

└─ performTraversals()

├─ measure() → 树形递归测尺寸

├─ layout() → 递归计算位置

└─ draw() → 生成DisplayList → 提交GPU绘制

每一帧的生成需经历三个阶段，均在ViewRootImpl.performTraversals()中触发递归：

Measure：递归计算所有View的宽高（深度优先）

Layout：递归确定所有View的位置

Draw：分两种模式：

软件绘制（CPU）：主线程操作，通过Skia库直接修改Bitmap像素（已逐渐淘汰）

硬件加速（GPU）：主流方式，包括两部分

主线程生成绘制指令（Display List）

渲染线程（RenderThread）通过OpenGL ES/Vulkan提交GPU绘制

（相关名词：Display List是一个绘制命令缓冲区。当View的成员函数onDraw被调用时, 我们调用通过参数传递进来的Canvas的drawXXX成员函数绘制图形时, 我们实际上只是将对应的绘制命令以及参数保存在Display List中。）

渲染线程执行的时间线：

|---- CPU侧 draw() ----|----- GPU执行延迟 -----|--显示--|

| | | |

|-- 生成DrawCall |-- 顶点处理 | |

|-- 提交命令到驱动队列 |-- 栅格化 | |

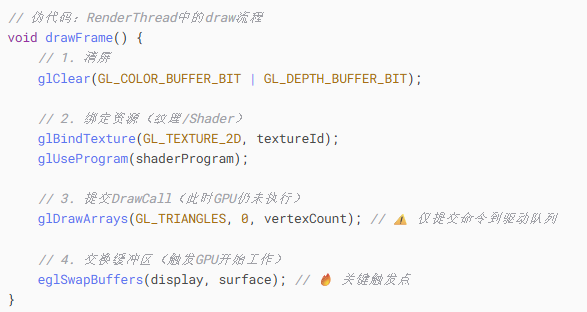
|-- 调用eglSwapBuffers |-- 片段着色 | |

|-- 写入帧缓冲 | |

| |--VSync显示帧

渲染线程详细流程：

阶段1：渲染线程的draw()过程（CPU侧工作）



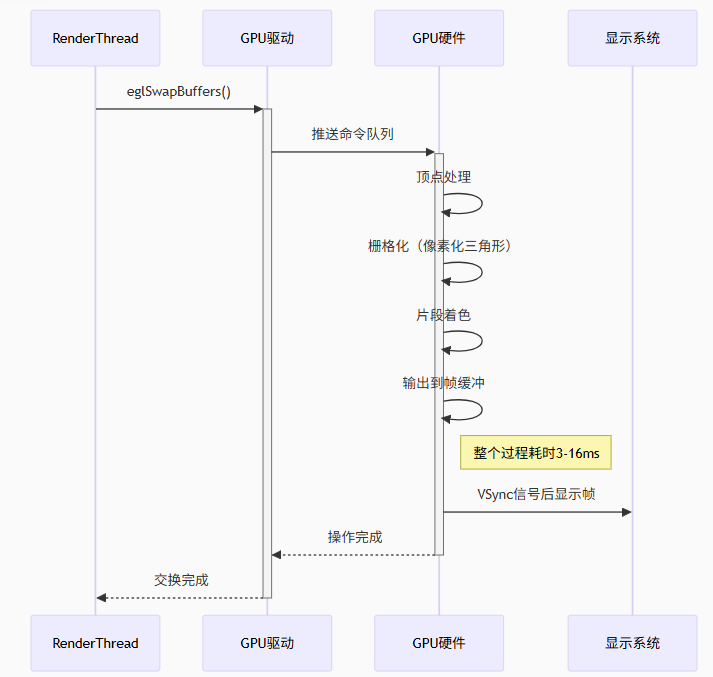
此时GPU状态：

命令进入GPU驱动的等待队列

GPU可能仍在处理前一帧任务（异步执行）

栅格化尚未开始

阶段2：GPU绘制启动（eglSwapBuffers()之后）



栅格化发生位置：

在GPU硬件的固定功能单元中执行

位于顶点处理之后、片段着色之前

完全由GPU硬件控制，CPU无法干预

（注: Android系统级监控）



可能需要关注：

移动端GPU的特殊优化（与RenderThread协作）



* 1. 系统层合成（SurfaceFlinger + HWC）

核心目标： 将应用程序生成的多个图形缓冲区高效、正确地合并（合成）成一个最终的帧，并按时送到显示设备（屏幕）进行显示。

SurfaceFlinger：

接收所有Layer（每个Surface对应一个Layer）

每16ms收到VSYNC信号后启动合成

HWC (硬件合成器)：

决策哪些Layer可直接用Overlay硬件叠加（省电高效）

无法叠加的Layer由GPU预合成到中间Buffer

合成策略优先级：HWC Overlay > GPU合成 → 目标是最小化GPU负载

合成时序：

应用进程 SurfaceFlinger HWC 显示硬件

| | | |

|--- 1. 绘制完成 ------->| | |

| (queueBuffer) | | |

| | | |

|<---- VSYNC-app信号 ----| | |

| (触发下一帧绘制) | | |

| | | |

| |<------- VSYNC-sf信号 -----| |

| | (触发合成周期) | |

| | | |

| |-- 2. 收集图层状态 ------>| |

| | (可见区域/Z-order等) | |

| | | |

| |-- 3. prepare() -------->| |

| | | |

| |<---- 3.1 合成类型决策 ----| |

| | (Device/Overlay/Client) | |

| | | |

|--- 4. 释放旧缓冲区 <----| | |

| (当fence触发后) | | |

| | | |

| |-- 5. GPU预合成? ---------| |

| | (若Client类型则执行) | |

| | | |

| |-- 6. set() ------------>| |

| | (配置图层参数) | |

| | | |

| |-- 7. present() -------->| |

| | | |

| | |-- 8. 硬件合成执行 --->|

| | | (Overlay引擎操作) |

| | | |

| |<------ 9. 返回fence -----| |

| | (present完成信号) | |

| | | |

| | |<--- VSYNC-vblank信号 --|

| | | |

| | |-- 10. 扫描输出帧 ----->|

| | | (显示到屏幕) |

| | | |

关键步骤说明：

1. 应用提交 (queueBuffer)
   * 应用完成帧渲染，通过 BufferQueue 提交给 SurfaceFlinger
2. VSYNC 同步
   * VSYNC-app：触发应用开始下一帧绘制
   * VSYNC-sf：唤醒 SurfaceFlinger 开始合成
3. 合成决策阶段 (prepare)
   * HWC 为每个图层分配合成类型：
     + Device/Overlay：硬件直接合成（最优路径）
     + Client：需 GPU 预合成
     + SolidColor/Cursor：特殊图层
4. 资源回收
   * 当 GPU/HWC 的释放 fence 触发后，回收应用旧缓冲区
5. GPU 回退合成
   * 仅当 HWC 返回 Client 类型时执行
   * 合成结果作为新图层提交给 HWC
6. 硬件配置 (set)
   * SurfaceFlinger 向 HWC 发送：
     + 图层缓冲区句柄
     + 位置/混合模式/变换矩阵
     + 可见区域
7. 执行合成 (present)
   * HWC 启动显示控制器：
     + Overlay 图层直通显示
     + 混合 GPU 合成结果
   * 返回 present fence 表示完成时机
8. 帧显示 (vblank)
   * 在 VSYNC-vblank 期间扫描输出帧
   * 避免屏幕撕裂

关键时序特征：

1. 流水线并行

帧 N 显示 → 帧 N+1 合成 → 帧 N+2 绘制

* + 三帧并行处理提高吞吐量

1. 决策-执行分离
   * prepare() 阶段：纯决策（无硬件操作）
   * present() 阶段：实际硬件编程
2. 硬件优化窗口
   * set() 到 vblank 之间时间用于：
     + 显示控制器预加载图层
     + 内存带宽优化

性能关键点：HWC 在 prepare() 阶段的决策质量直接影响性能。理想情况下应有 >70% 的图层被标记为 Overlay 类型，避免 GPU 回退合成消耗带宽（1080p@60Hz 下 GPU 合成额外消耗 ~1.2GB/s 带宽）。

* 1. 其他层级介绍

硬件抽象层（Hardware Abstraction Layer，HAL），将不同厂商的硬件实现细节（如芯片型号、寄存器操作、驱动程序）封装在 HAL 层中，向上为 Framework 层提供硬件抽象接口（如 Java 层通过 JNI 调用 HAL 功能），向下通过 Linux 内核驱动访问实际硬件（如通过 ioctl 操作硬件寄存器）。

内核层（Linux Kernel），硬件与软件之间的桥梁，直接管理手机的硬件资源。主要功能包括：运行驱动程序（如 Binder驱动、显示驱动），管理硬件资源（CPU、内存、I/O 设备），提供系统调用接口（如 open()、read()、write()）。

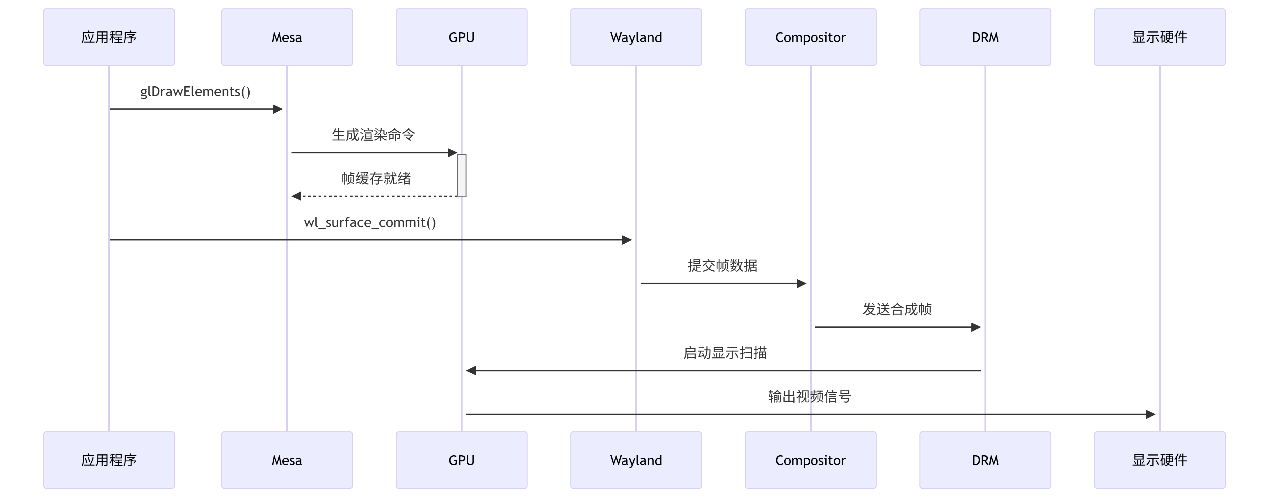
内核层相关名词：

DRM/KMS (Direct Rendering Manager/Kernel Mode Setting)，作用：统一管理GPU与显示控制器，在Android上通过libdrm库与HWC交互，驱动屏幕刷新

ION/DMA-BUF 内存管理，作用：解决CPU/GPU/Display跨设备内存共享，在Android上Gralloc HAL通过ion\_alloc分配图形缓冲区

Fence机制：通过轻量级的文件描述符（fd）同步，高效解决了异步操作中的顺序性和资源竞争问题

Linux图形绘制原理



2.1 流程解析：

应用程序层

功能：用户图形程序（如Firefox、LibreOffice）通过GTK/Qt工具包构建UI元素

调用OpenGL/Vulkan执行渲染指令（如3D模型绘制）

图形API层（Mesa）

功能：将高级渲染命令（如glDrawElements()）转换为GPU可执行指令

显示服务器协议层（Wayland）

功能：应用直接与合成器通信，提交已渲染的缓冲区

通过wl\_surface提交图像，wl\_commit()原子化更新

合成器层：

功能：混合多个应用的Surface（如叠加窗口/半透明效果）

将鼠标/键盘事件分发给焦点窗口

控制分辨率、多屏布局

主流实现：

Weston：Wayland参考合成器

Mutter（GNOME）/KWin（KDE）：支持X11/Wayland双协议

内核图形子系统

包括：DRM（Direct Rendering Manager）：

管理GPU显存分配、命令队列提交

提供/dev/dri/cardX接口供用户空间（Mesa）访问GPU

KMS（Kernel Mode Setting）：

控制显示器参数（分辨率/刷新率）

消除X11时代切换分辨率时的屏幕闪烁

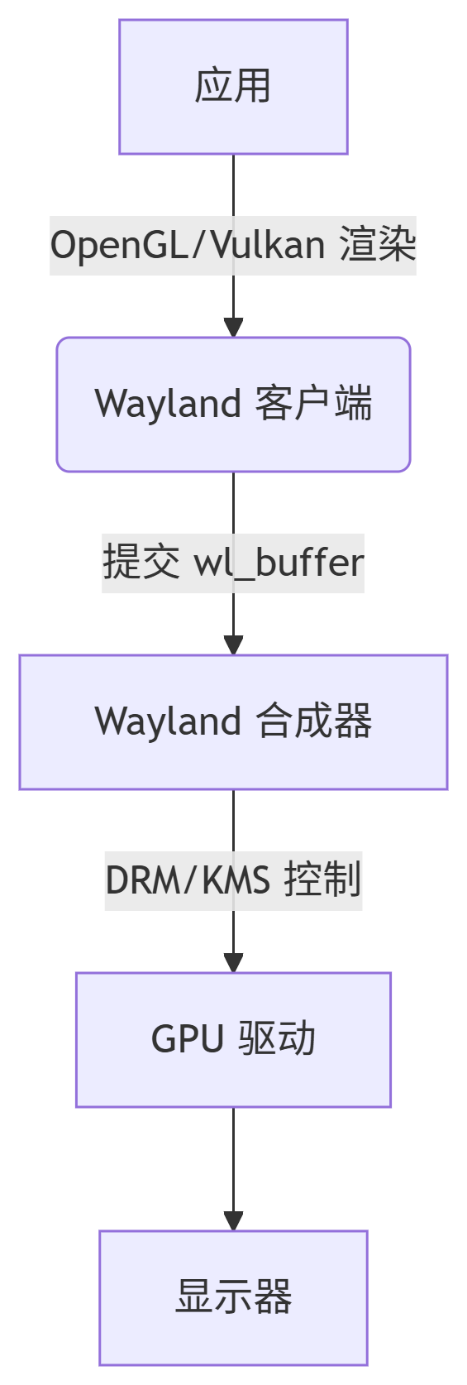
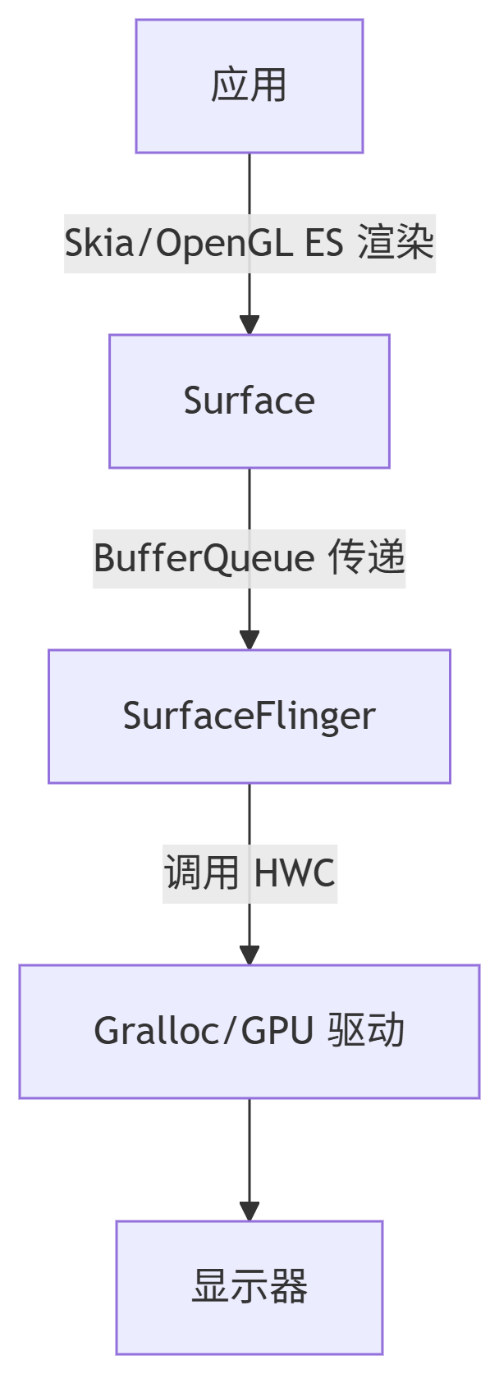
Linux vs Android 图形系统对比

设计理念的差异：

| **维度** | **Linux 图形系统** | **Android 图形系统** |
| --- | --- | --- |
| **目标场景** | 通用桌面/嵌入式设备，强调灵活性与标准化 | 移动设备优先，强调度触控交互与能效优化 |
| **架构哲学** | 模块化分层（协议+合成器+驱动） | 垂直整合（应用→系统服务→HAL） |
| **开放性** | 协议开源（如 Wayland），允许定制合成器 | 部分闭源（如 HWC 厂商实现），框架层开放 |
| **兼容性** | 需兼容 X11 历史生态（XWayland） | 仅支持 Android 应用，无原生 X11 兼容 |

Wayland强调标准化与扩展性，Android追求封闭可控

Linux图形栈： Android 图形栈：

**3.1 渲染机制对比**

**Linux (Wayland)**

* **客户端自主渲染**  
  应用程序直接调用OpenGL/Vulkan（通过Mesa3D）渲染内容到**wl\_buffer**（共享内存或DMA-BUF），合成器仅接收最终帧，不干预渲染过程。
* **低延迟优先**  
  无强制垂直同步（VSYNC），依赖应用自主管理帧提交，可能引发画面撕裂但延迟更低。
* **扩展性支持**  
  支持复杂渲染API（如Vulkan），适合高性能桌面应用（3D建模/游戏）。

**Android**

* **系统管控渲染**  
  应用通过Skia/OpenGL ES渲染到**Surface**，由**BufferQueue**管理双/三缓冲，严格遵循VSYNC信号提交帧，避免撕裂但增加延迟。
* **移动场景优化**  
  渲染指令经Android View系统封装，优先保障触控响应能效，牺牲部分灵活性。
* **后备软件渲染**  
  无GPU设备时，SwiftShader提供软件渲染支持，确保基础图形功能。

**关键差异**：  
Linux渲染更自由，延迟低但需应用自律；Android渲染受系统强管控，延迟可控但灵活性受限。

**3.2 合成机制对比**

**Linux (Wayland合成器如Weston/wlcom)**

* **GPU主导合成**  
  合成器（如Weston）混合所有**wl\_surface**，依赖GPU计算透明度、阴影等效果，消耗较多算力。
* **无硬件叠加层**  
  缺乏类似Android HWC的专用硬件，多窗口合成完全依赖GPU，高负载时性能下降明显。
* **插件式扩展**  
  如wlcom支持防截屏协议、圆角/毛玻璃特效插件，通过模块化增强功能。

**Android (SurfaceFlinger + HWC)**

* **硬件叠加层优先**  
  Hardware Composer（HWC）直接利用显示处理器（DPU）合成静态图层（如状态栏），仅动态内容由GPU处理，大幅降低功耗。
* **图层数量限制**  
  DPU通常支持4-7层硬件叠加（如Mali-DP550），超限时回退GPU合成。
* **严格帧同步**  
  所有合成操作按VSYNC节奏执行，确保无撕裂但可能因等待信号增加延迟。

**关键差异**：  
Android通过HWC实现硬件级能效优化；Linux依赖GPU通用计算，灵活性高但能效比低。

**3.3 底层架构对比**

**公共基础**

* **共享内核子系统**  
  两者均依赖DRM/KMS管理显示模式设置与缓冲区交换。
* **内存传递优化**  
  均采用零拷贝技术：
  + Wayland：**wl\_shm\_pool**共享内存或DMA-BUF
  + Android：**Gralloc**分配BufferQueue传递DMA-BUF

可行性分析：

中兴通讯正式商用Linux图形系统方案Tifeagle （未见其发布的产品和技术细节）

<https://www.zte.com.cn/china/about/news/20171127.html>

从渲染、合成、驱动与底层机制分析：

渲染：

1. **渲染模型差异**
   * **Wayland渲染**：采用直接渲染架构（DRI），客户端（应用）通过EGL/OpenGL ES/Vulkan API直接操作GPU，将内容渲染到共享缓冲区（DMA-BUF），再由合成器（Compositor）合成输出。其核心依赖Linux内核的**DRM/KMS**和**GEM**内存管理。
   * **Android渲染**：应用通过**Surface**接口提交图形缓冲区（GraphicBuffer），由**SurfaceFlinger**合成，并通过**Hwcomposer**处理硬件叠加层（Overlay）以降低GPU负载。
   * **关键矛盾**：Wayland的渲染流程需直接对接DRM/KMS，而Android的渲染链依赖HAL层抽象（如Gralloc），两者缓冲区管理协议（如DMA-BUF vs Android Native Buffer）需深度适配。
2. **硬件加速兼容性**
   * Wayland默认要求**OpenGL ES**或**Vulkan**支持，而Android的GPU驱动（尤其闭源驱动如高通Adreno、NVIDIA）需实现Wayland EGL平台扩展（如EGL\_WL\_bind\_wayland\_display）

合成：

1. **合成器角色冲突**
   * **Wayland合成器**：集显示服务器、窗口管理、合成器于一体（如Weston、KWin），直接控制KMS/evdev。
   * **Android合成器**：SurfaceFlinger仅负责合成，窗口管理由WindowManagerService处理，输入事件独立传递。
   * **整合路径：**
     + **方案A**：用Wayland合成器（如wlroots）替代SurfaceFlinger，直接接管Hwcomposer（需重写HAL层）。
     + **方案B：**将Wayland作为Android的“客户端”，其合成器输出作为Android的一个Surface（类似Waydroid容器化方案），但引入额外性能损耗。
2. **多合成层性能损耗**  
   Android的图形栈已包含应用→SurfaceFlinger→Hwcomposer多层合成，若嵌套Wayland合成器（如方案B），将导致**缓冲区复制**和**同步延迟**，实测在低端GPU上帧率下降可达30%

驱动与底层

1. 内核层依赖  
   Wayland强依赖DRM/KMS进行模式设置与缓冲区交换，而Android内核通常定制化DRM驱动（如msm\_drm高通驱动），需确保KMS接口兼容性。部分闭源驱动未开源KMS实现，阻碍Wayland直接控制显示管线。
2. 输入与权限管理
   * Wayland合成器需直接接管输入设备（通过libinput），但Android的输入事件由InputReader/InputDispatcher处理，需重写事件分发链路。
   * 安全隔离：Android的SELinux策略可能限制Wayland合成器的DRM设备访问权限，需新增安全上下文规则

Wayland protocol

git下载：git clone https://gitlab.freedesktop.org/wayland/wayland.git

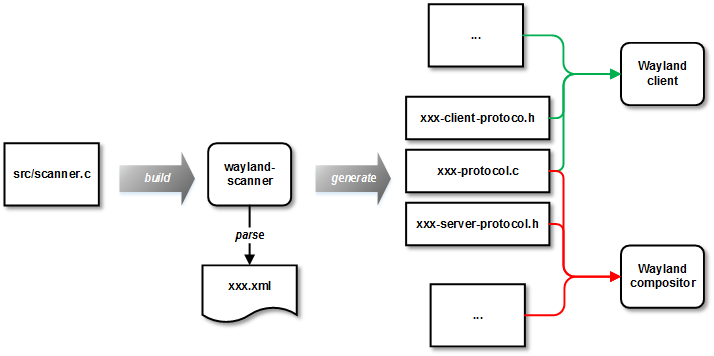
协议xml：**wayland.xml**

wayland-scanner 工具解析这个xml生成Server、Client的协议头文件以及一个协议C文件

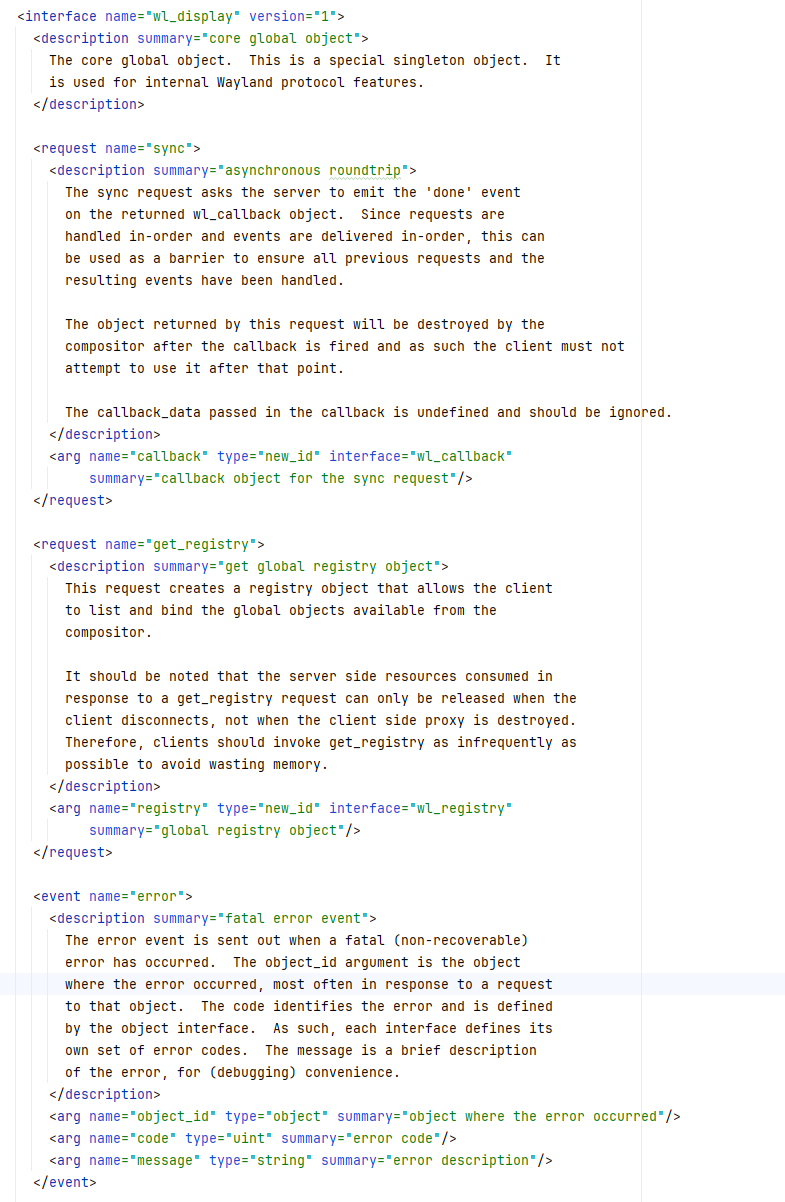
wayland-scanner server-header xxx.xml xxx\_server\_header.h

wayland-scanner client-header xxx.xml xxx\_client\_header.h

wayland-scanner private-code xxx.xml xxx\_protocol.c



协议xml解析



以tests/data/目录为例：

example-client.h/ example-server.h/example-code.c为wayland.xml协议解析的产物

example-code.c为协议C文件，将xml中的interface解析为结构体wl\_interface，包含六个属性：

name（名称）

version（版本号）,

method\_count（request个数）

methods（wl\_message结构体）

event\_count（event个数）

events（wl\_message结构体）

结构体wl\_message包含三个属性：

name（名称）

signature(参数类型)

types（wl\_interface类型）



example-client.h为客户端协议头文件，将xml中的request和event转成对应的函数。

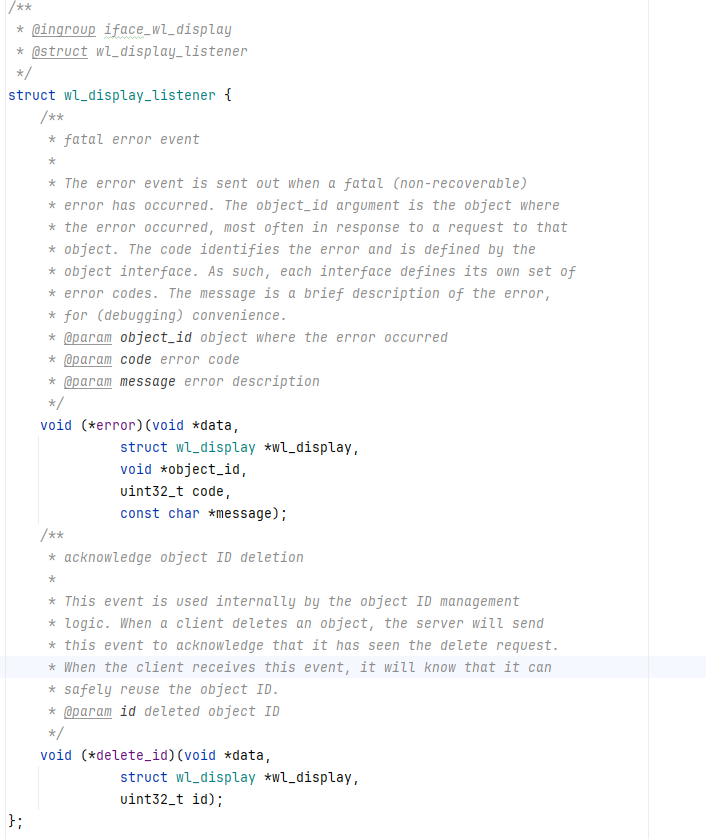
还是以上述xml中的interface：wl\_display为例：

request：get\_registry函数为：

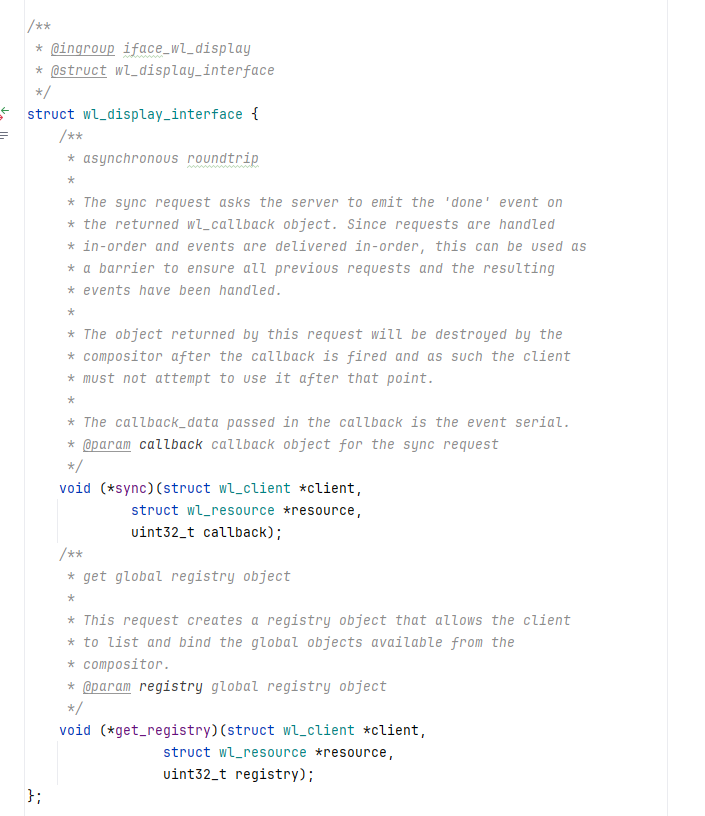


xml中arg类型是new\_id的会作为函数的返回值类型。并且所有的函数添加一个默认的参数，就是它所在的interface，作为第一个参数传入进去，就相当于给一个类成员函数传递this指针一样。告诉是哪个具体对象调用的这个接口。

event则统一定义成interface：wl\_display的listener。

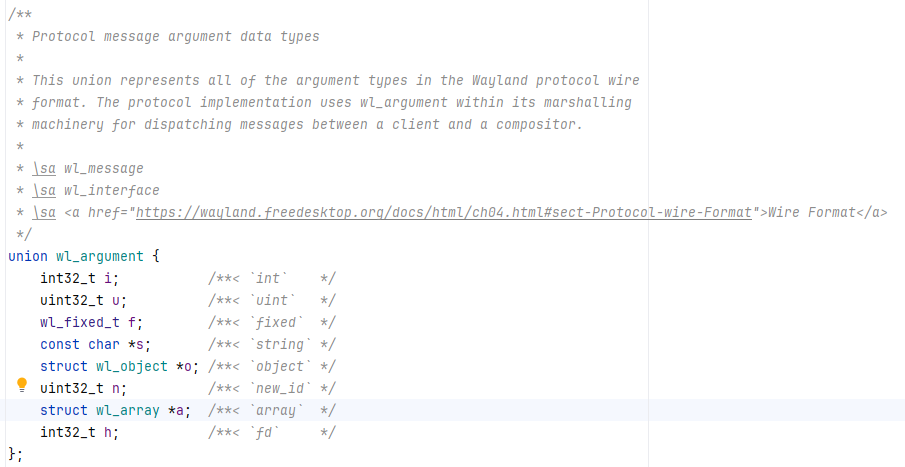


example-server.h为服务端协议头文件，对于request和event的解析与client端对应

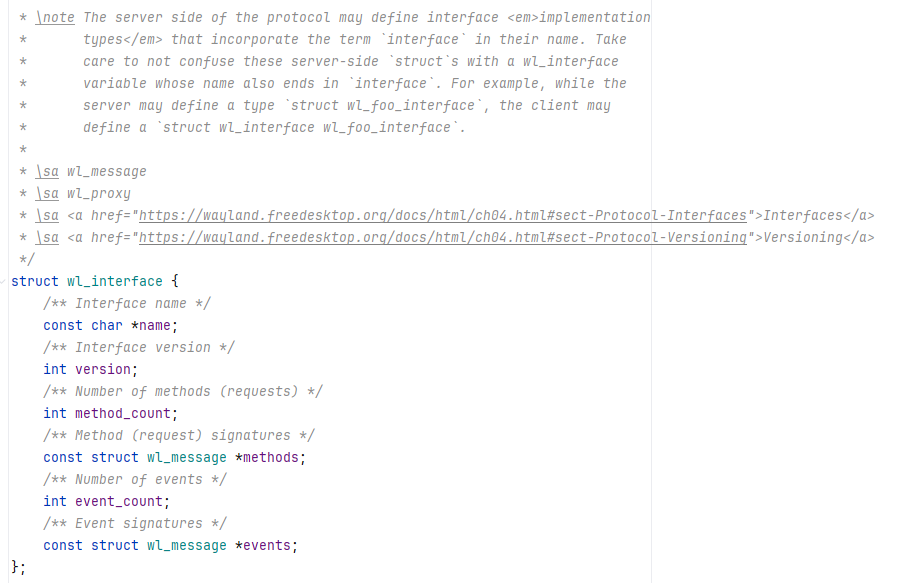


数据结构

wl\_argument（src/wayland-util.h）：参数类型

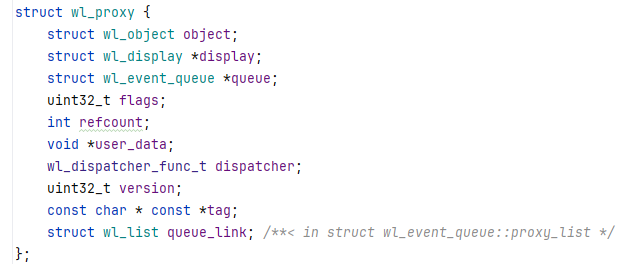


wl\_interface：interface的结构体



wl\_proxy（src/wayland-client.c）：客户端真正的对象结构体

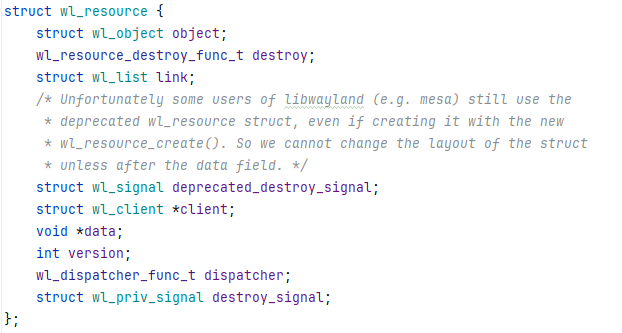
Wayland协议里面的那些interface对象在客户端其实真正的结构体是wl\_proxy，而那些结构体都是不存在的，只是一个声明而已，根本不存在。从源码可以看到wayland协议里面的那些interface对象，都不是创建出来的，而是通过wayland的一些接口返回回来的。



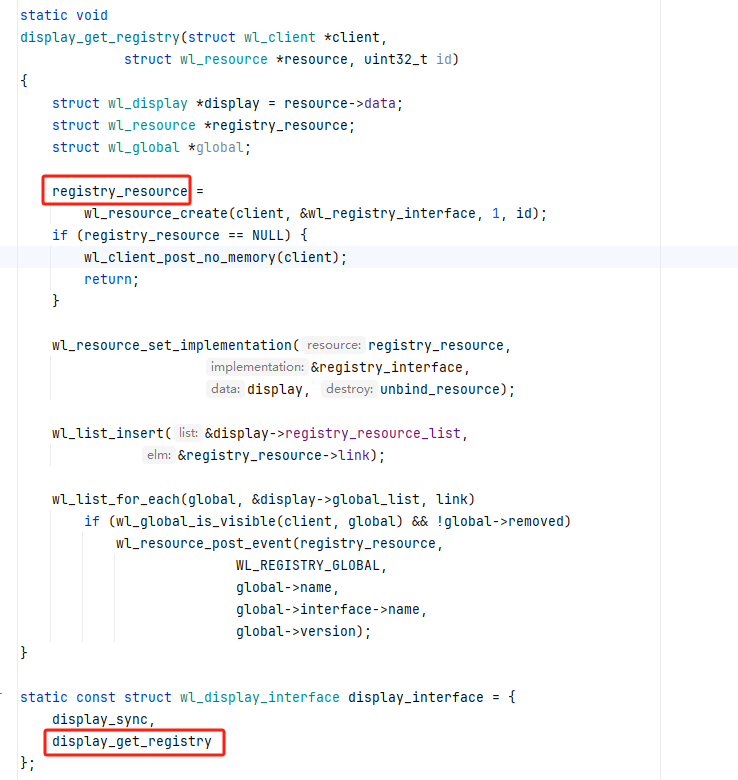
以wl\_display\_get\_registry为例，



同理，服务器端真正的对象结构体是wl\_resource（src/wayland-server.c）



以display\_get\_registry为例：



wl\_proxy/wl\_resource同wl\_interface的对应关系



从上图中可知，所有的对象都有个wl\_object成员来记录它是属于哪个interface的，并用implementation成员来保存真正需要调用的函数指针，成员变量id则是用来进行客户端和服务器端跨进程对象映射的关键。

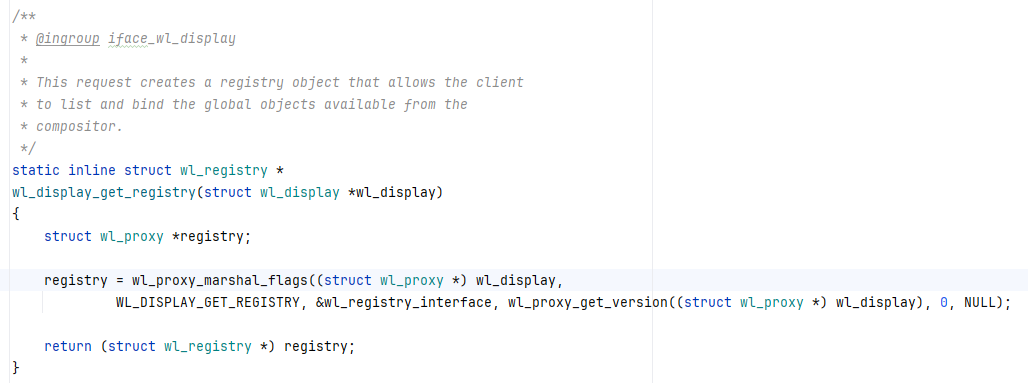
然后wl\_interface就是wayland协议里面的interface。

跨进程调用

以wl\_display\_get\_registry()为例：

调用 wl\_proxy\_marshal\_flags()，创建wl\_proxy，然后将该wl\_proxy插入到display->objects的wl\_map中，返回值为id，其实就是在wl\_map中数组中的索引值。这个值是会被发到Server端的，这样Server端就可以在Server端的wl\_map数组的相同索引值的位置插入相应的wl\_resource。这样逻辑上，就创建了wl\_proxy和wl\_resource的映射关系。

相关代码：



相关流程为：

调用 wl\_proxy\_marshal\_flags()

|

v

检查是否需要创建新代理 (interface != NULL?)

|是

v

创建 wl\_proxy 结构体 (wl\_proxy\_create)

|

v

分配唯一ID (wl\_map\_insert\_new)

|

v

序列化请求（将新ID作为 new\_id 参数）

|

v

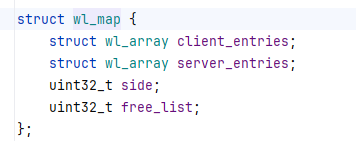
发送请求到服务端 (Unix socket)

|

v

返回新 wl\_proxy 给调用者

数据结构：wl\_map



这个数据结构是wayland的核心，用来保存进程间通信的实际对象指针，并得到对于指针的ID，用于进程间传递。

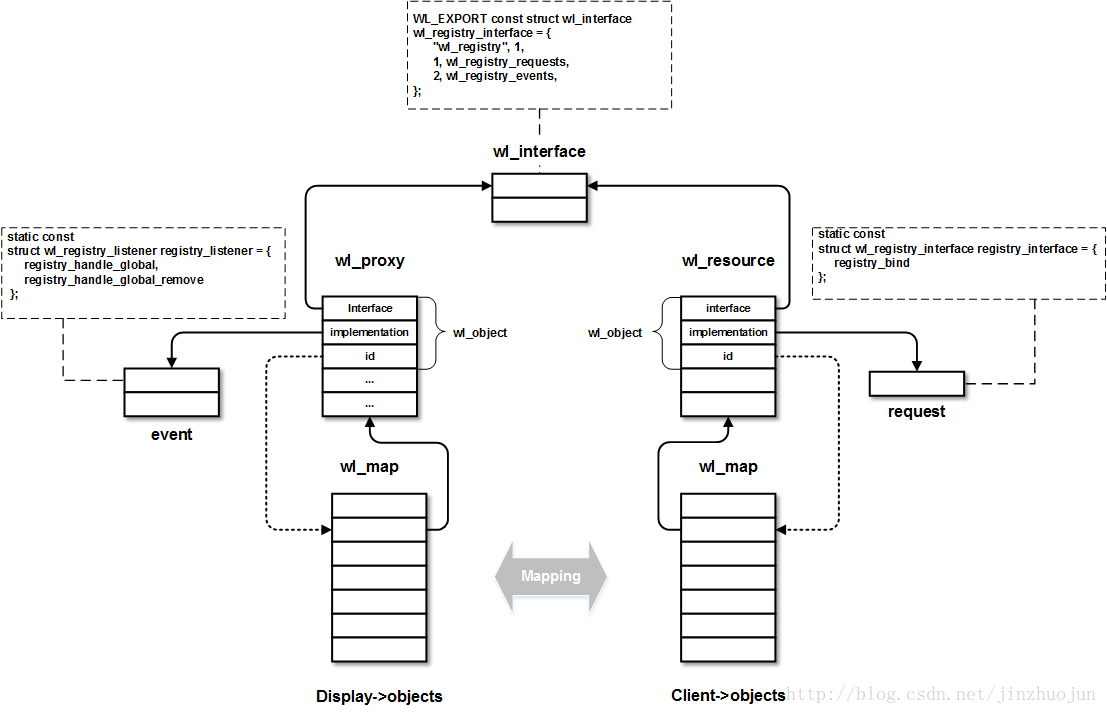
wl\_map结构体存放wayland客户端和服务器端对应的对象。其中：

client\_entries： 用wl\_array数组保存客户端的对象。 （这种情况server\_entries不使用）

server\_entries：用wl\_array数组保存服务器端的对象。（这种情况client\_entries不使用）

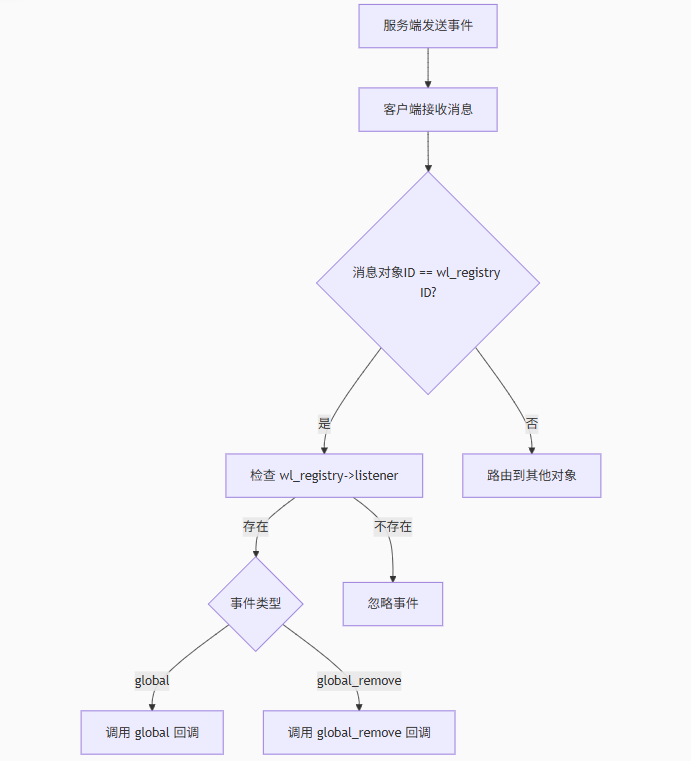
side：表明当前map保存的是客户端还是服务器端对象 （通过这个变量，确定client\_entries/server\_entries里面保存的对象，并且这个变量只在客户端初始化为WL\_MAP\_CLIENT\_SIDE，在服务器端初始化为WL\_MAP\_SERVER\_SIDE）。

wayland协议实现基本决定了，创建对象都是客户端发起的，流程是客户端请求创建对象，通过wl\_map\_insert\_new函数插入对象，并返回对象的ID（其实是下标），然后把ID传递到服务器端，在服务器端通过wl\_map\_insert\_at函数把创建的对象插入到指定的下标（ID），这样就建立了两个对象的对应关系。只有一个对象是特殊的，就是wl\_display.这个对象是写死的下标（为1），因为这个对象肯定是第一个创建的。

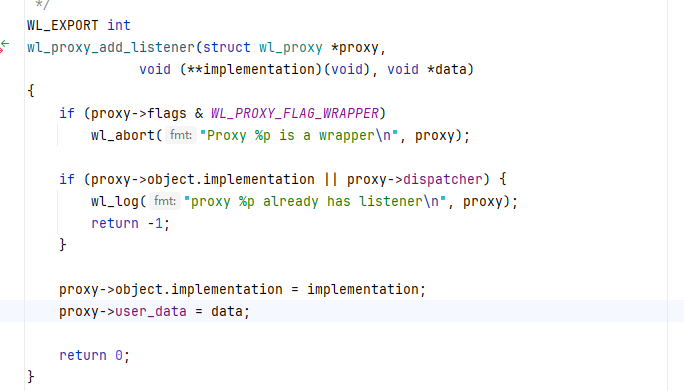


事件接收与回调触发流程（event/ listener）

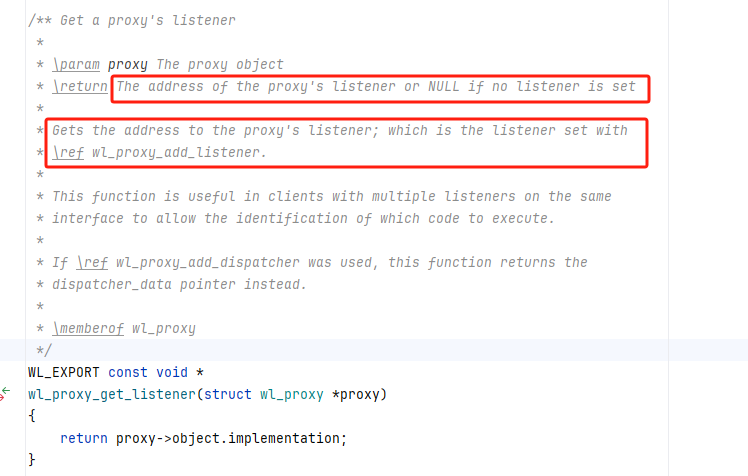
以wl\_registry\_add\_listener为例：



add\_listener：纯客户端操作，不直接与服务端通信，只是在wl\_proxy结构中保存回调函数指针



get\_listener：通过对象ID映射表找到对应的wl\_proxy，检查该wl\_proxy的 listener字段是否非空。如果非空，则回调对应的函数。



同步性要求：监听器必须在事件循环wl\_display\_dispatch前设置。

Wayland global 事件

lobal 事件是 Wayland 协议中的核心事件，由服务端发送给客户端，用于宣告服务端提供的全局对象。这些对象代表了服务端的核心功能，例如：

wl\_compositor：创建界面元素

wl\_shm：共享内存支持

wl\_output：显示器信息

wl\_seat：输入设备（键盘、鼠标）

以客户端获取 wl\_compositor 对象为例，展示完整的 global 事件流程：

