Wayland与Weston简介

发布时间：2018-01-19 来源：网络 上传者：用户
关键字: Wayland 简介 Weston

摘要：简单地说,Wayland是一套displayserver(Waylandcompositor)与client间的通信协议,而Weston是Waylandcompositor的参考实现。其官网为http://wayland.freedesktop.org/。它们定位于在Linux上替换X图形系统。X图形系统经历了30年左右的发展,其设计在今天看来已略显陈旧。在X系统中,XServer作为中心服务,连接clien和硬件以及compositor。但时至今日,原本在XServer中做

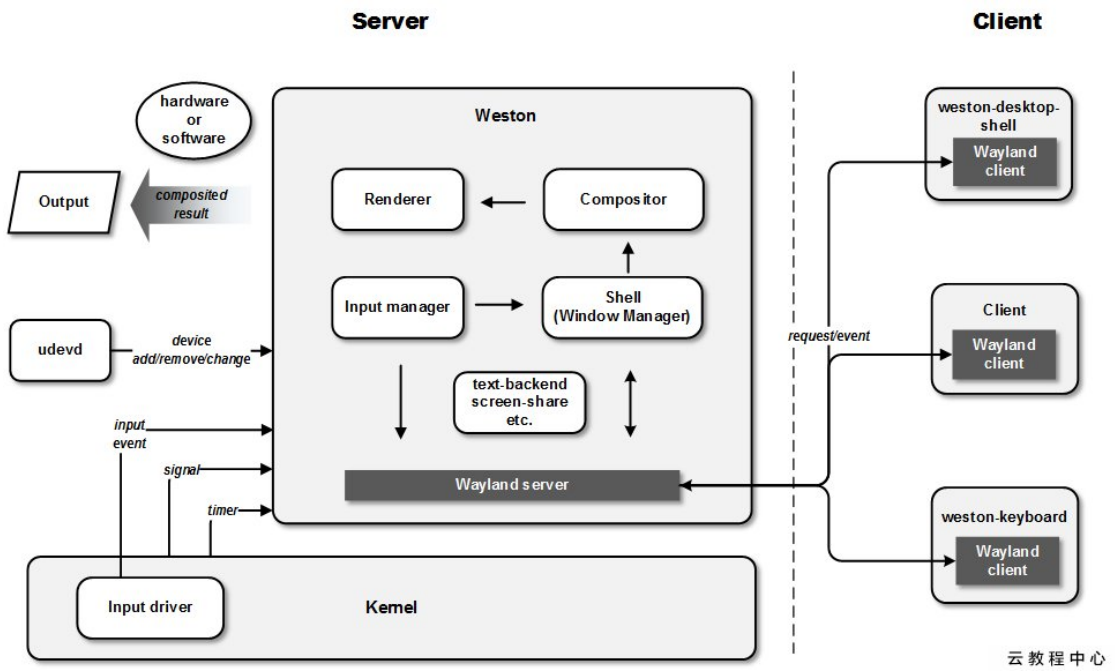
简单地说,Wayland是一套display server(Wayland compositor)与client间的通信协议,而Weston是Wayland compositor的参考实现。其官网为http://wayland.freedesktop.org/。它们定位于在Linux上替换X图形系统。X图形系统经历了30年左右的发展,其设计在今天看来已略显陈旧。在X系统中,X Server作为中心服务,连接clien和硬件以及compositor。但时至今日,原本在X Server中做的事很多已被移到kernel或者单独的库中,因此X Server就显得比较累赘了。Wayland在架构上去掉了这个中间层,将compositor作为display server,使client与compositor直接通信,从而在灵活性和性能等方面上能够比前辈更加出色。

Wayland既可以用于传统的桌面又适用于移动设备,已经被用于Tizen,Sailfish OS等商业操作系统,同时越来越多的窗口和图形系统开始兼容Wayland协议。Wayland基于domain socket实现了一套display server与client间通信的库(简单的基于例子的介绍可以参见http://blog.csdn.NET/jinzhuojun/article/details/40264449),并且以XML形式定义了一套可扩展通信协议。这个协议分为Wayland核心协议和扩展协议(位于Weston中)。Weston作为Wayland compositor的参考实现,一般和Wayland同步发布。其它Wayland compositor实现还有如mutter, Kwin, Lipstick, Enlightenment, Clayland等。

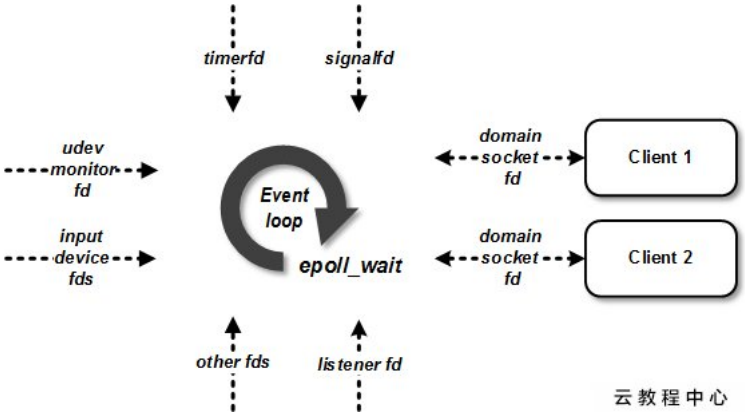
下面分别从架构,源码及模块结构,渲染流水线,窗口和输入管理几个方面介绍下Wayland/Weston的实现。

架构

Wayland的系统体系架构可以参见官方文档,不再累述。Weston从内部体系结构来看,主要分为窗口管理(shell),合成器(compositor)和输入管理几个部分。可见,如果拿Android作类比,从功能上看它约等同于InputManagerService,WindowManagerService和SurfaceFlinger。从大体的流程上来看,输入管理模块接受用户输入,然后一方面shell作出相应的窗口管理操作(如窗口堆栈的改变,focus的变化等),另一方面将该input event传给之前注册了相应输入事件的client。client收到后会在handler中做相应动作,如调整视图然后重绘。如有重绘发生,新buffer渲染完成后client将其handle传给server,接着server端生成z-order序的窗口列表,之后compositor用renderer进行合成,最后输出(比如到framebuffer)。



Weston是主要服务进程,它的事件处理模型采用的是典型的Reactor模式。根据linux中万物皆文件的原则,主循环通过epoll机制等待在一系列的文件fd上。这种模型与基于线程的binder不同,是一种串行的事件处理模型。在此模型上的过程调用在不加额外同步机制的情况下是异步的。好处是不会有竞争问题,数据同步开销较小。缺点是当有一个事件处理比较耗时或者在等待IO,则有可能使整个系统性能下降或响应不及时。



主循环上等待的几个核心fd包括:

• Server/Client通信:listener fd在Weston启动时建立,并一直监听新的client连接。一个client连接后会与Weston建立一对domain socket,Wayland就是基于它来通信的。

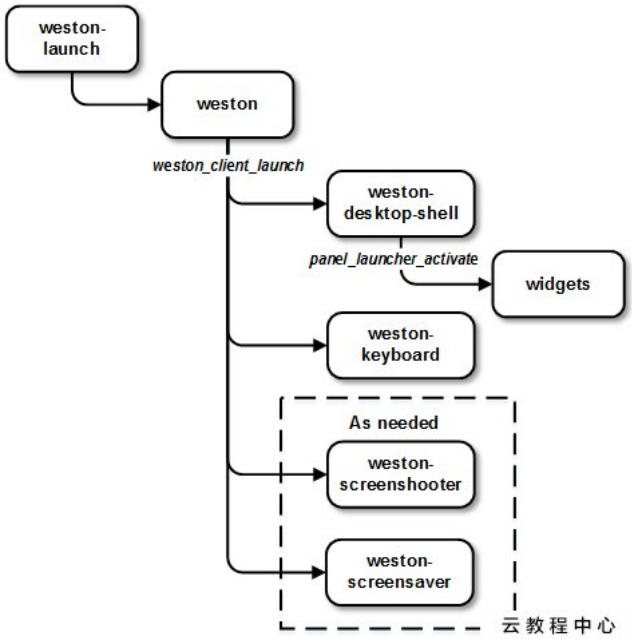
• 输入处理:一方面通过udev monitor监听设备的添加删除事件。另一方面如有新设备添加时会将该设备打开并监听该fd来得到输入事件。

• 其它:监听如timer(用于如睡眠锁屏等场景)和signal(如收到SIGINT, SIGTERM, SIGQUIT时退出主循环)等事件。timer和signal可以分别用timerfd和signalfd来用fd来表示。另外还有logind的dbus连接等。

除这些外,在event loop中还会维护一个idle list。Weston中需要异步处理的操作可以放在其中。每轮循环都会检查其中是否有任务,有的话拿出来执行。

下面看下Weston的运行时进程模型。Weston设计时是可以以一般用户运行的,但就需要用weston-launch来启动。当需要进行一些需要root权限的工作,比如关于DRM, TTY, input device的相关操作,就交由weston-launch去做。

Weston会在启动时或按需起一些子进程,它们本质上是Weston的client,它们会通过专用的协议做一些系统应用的工作。如系统应用weston-desktop-shell负责一些系统全局的界面,比如panel, background, cursor, app launcher, lock screen等。它不作为Weston服务本身的一部分,而是作为一个client。其作用有点类似于android中的SystemUI。这样便可方便地替换成定制的界面。weston-keyboard是软键盘面板。weston-screenshooter和weston-screensaver分别用于截屏和屏保,它们都是按需才由Weston启动的。前者在截屏快捷键按下时启动,后者在需要锁屏时启动。

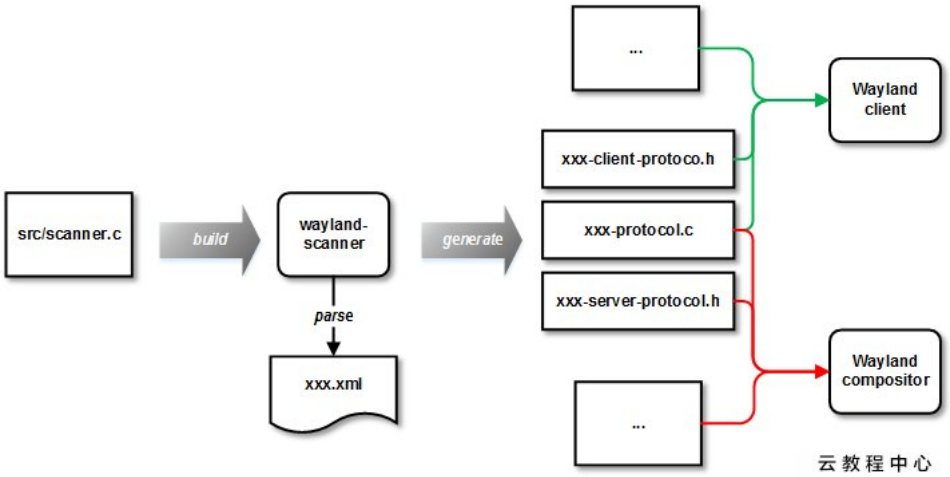


另外,Weston启动时会读取weston.ini这个配置文件,其中可以配置桌面,动画和后端等等信息。详细配置见http://manpages.ubuntu.com/manpages/raring/man5/weston.ini.5.html。

源码与模块结构

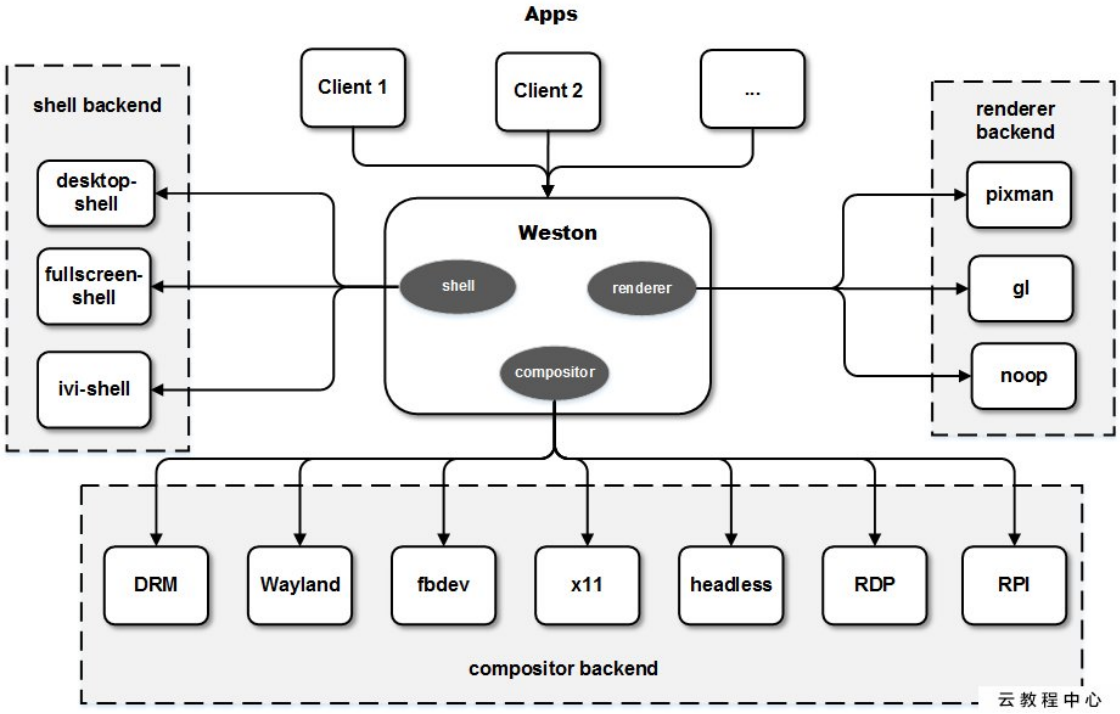
wayland/weston/libinput等项目源码位于http://cgit.freedesktop.org/wayland下。

Wayland的协议定义在protocol目录,通信协议实现在src目录。它主要编译出三个库,libwayland-client,libwayland-server和libwayland-cursor。第一个为协议的client端实现,第二个为协议的server端实现。第三个是协议中鼠标相关处理实现。编译时会首先编译出wayland-scanner这个可执行文件,它利用expat这个库来解析xml文件,将wayland.xml生成相应的wayland-protocol.c,wayland-client-protocol.h和wayland-server-protocol.h。它们会被Wayland的client和server在编译时用到。同时wayland-scanner也需要在生成Weston中的Wayland扩展协议中起同样作用。



Wayland主要依赖于两个库,一个上面提到的expat协议,另一个libffi用于在跨进程过程调用中根据函数描述生成相应calling convention的跳板代码。

Weston的主要实现在src目录下。与Wayland类似,protocol目录下放着Wayland协议定义。在clients目录下是一些client的例子,也包括了desktop-shell和keyboard等核心client的例子,也包含了如simple-egl, simple-shm, simple-touch等针对性的简单用例。Weston启动过程中会分别加载几个backend:shell backend, render backend和compositor backend。它们分别用于窗口管理,合成渲染和合成内容输出。



由于这些后端都可有不同的实现,为了逻辑上的独立性和结构上的灵活性,他们都编译成动态链接库从而可以在Weston初始化时被加载进来。这种方式在Weston中被广泛采用,一些功能比如屏幕共享等都是以这种形式加载的。

举例来说,compositor backend主要决定了compositor合成完后的结果怎么处置。从数据结构上,weston_output是output设备的抽象,而下面的backend会实现具体的output设备。

• fbdev:直接输出至linux的framebuffer设备。接口通用。

• headless:和noop-renderer配合使用,可以在没有窗口系统的机子(比如server上)测试逻辑。

• RPI:用于Raspberry Pi平台。

• RDP:合成后通过RDP传输到RDP peer显示,用于远程桌面。

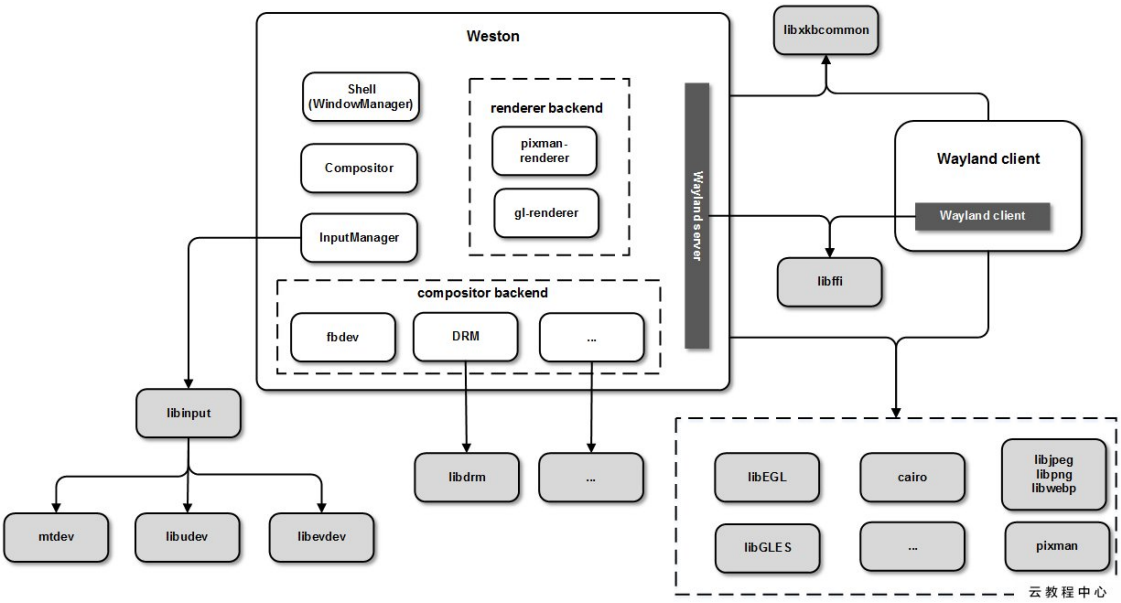
• DRM:Direct redering manager,桌面上一般用这个。

• x11:Wayland compositor作为X server的client。它可以让Wayland client运行在X11上。

• wayland:Wayland composiotr作为server同时,也作为另一个Wayland compositor的client。用于nested compositor。

Renderer backend主要用于compositor的合成之用,除去noop-renderer外,有gl-renderer和pixman-renderer两种。前者为GPU硬件渲染,后者为软件渲染。shell backend用于实现具体的窗口管理。相应的实现分别在desktop-shell,fullscreen-shell和ivi-shell目录中。

Wayland/Weston运行时依赖的库主要有下面几个,其相互关系大体如下。



• libEGL, libGLES:本地窗口系统与图形driver的glue layer,mesa提供了开源版本的实现。

• libdrm:封装KMS,GEM等图形相关接口。平台相关。

• libffi:用于在运行时根据调用接口描述生成函数跳板并调用。

• pixman:用于像素操作的库,包括region, box等计算。用了许多平台相关的优化。

• cairo:软件渲染库,类似于skia。也有OpenGL后端。

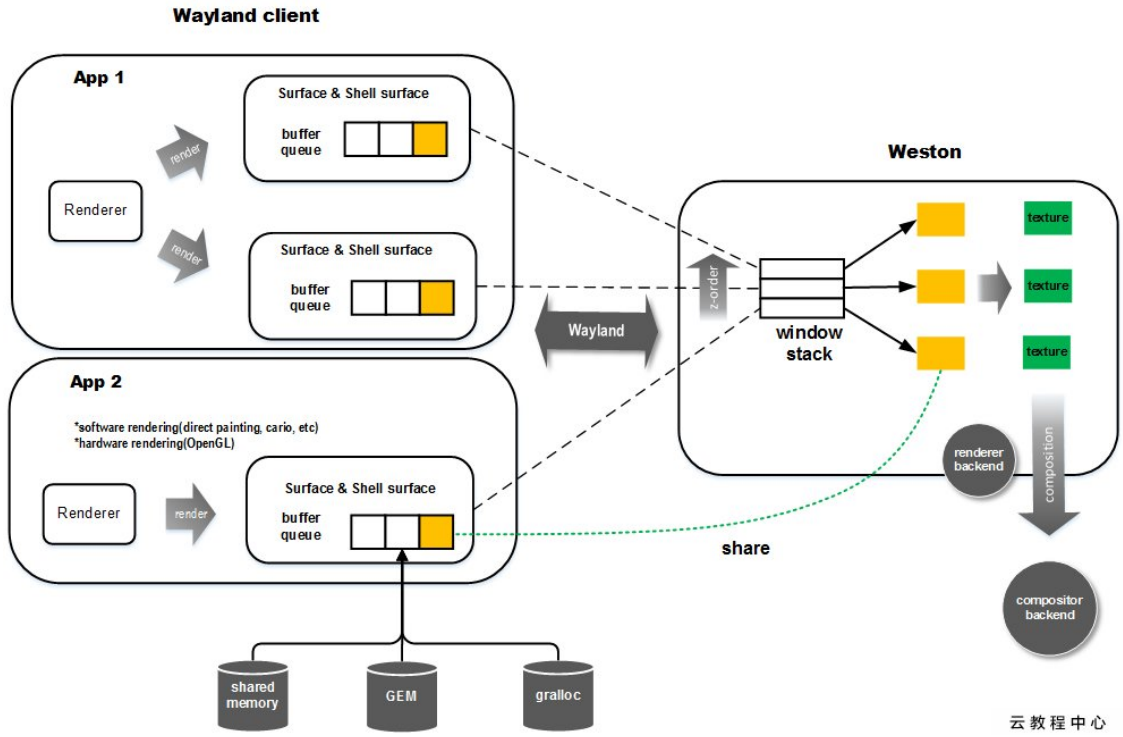
• libinput:输入处理,依赖于mtdev, libudev, libevdev等库。

• libxkbcommon:主要用于键盘处理。

• libjpeg, libpng, libwebp:用于加载各种图片文件,像壁纸,面板和鼠标等都需要。

渲染流水线

一个Wayland client要将内存渲染到屏幕上,首先要申请一个graphic buffer,绘制完后传给Wayland compositor并通知其重绘。Wayland compositor收集所有Wayland client的请求,然后以一定周期把所有提交的graphic buffer进行合成。合成完后进行输出。本质上,client需要将窗口内容绘制到一个和compositor共享的buffer上。这个buffer可以是普通共享内存,也可以是DRM中的GBM或是gralloc提供的可供硬件(如GPU)操作的graphic buffer。在大多数移动平台上,没有专门的显存,因此它们最终都来自系统内存,区别在于图形加速硬件一般会要求物理连续且符合对齐要求的内存。如果是普通共享内存,一般是物理不连续的,多数情况用于软件渲染。有些图形驱动也支持用物理不连续内存做硬件加速,但效率相对会低一些。根据buffer类型的不同,client可以选择自己绘制,或是通过Cairo,OpenGL绘制,或是更高层的如Qt,GTK+这些widget库等绘制。绘制完后client将buffer的handle传给server,以及需要重绘的区域。在server端,compositor将该buffer转为纹理(如果是共享内存使用glTexImage2D上传纹理,硬件加速buffer用GL_OES_EGL_image_external扩展生成外部纹理)。最后将其与其它的窗口内容进行合成。下面是抽象的流程图。

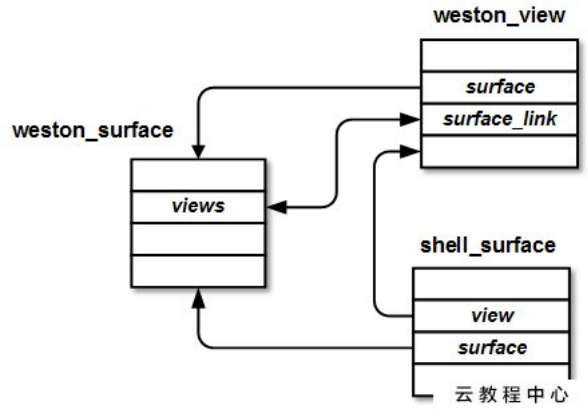


注意Wayland设计中默认buffer是从client端分配的。这和Android中在server端分配buffer的策略是不同的。这样,client可以自行设计buffer的管理策略。理论上,client可以始终只用一块buffer,但因为这块buffer在client和server同时访问会产生竞争,所以一般client端都会实现buffer queue。流水线上比较关键的一环是buffer跨进程的传输,也就是client和server间的有效传递。buffer当然不可能通过拷贝传输,因此这里只会传handle,本质上是传buffer的fd。我们知道fd是per-process的。而可以传递fd的主要IPC机制有binder, domain socket和pipe等。Wayland底层用的是domain socket,因此可以用于传fd。

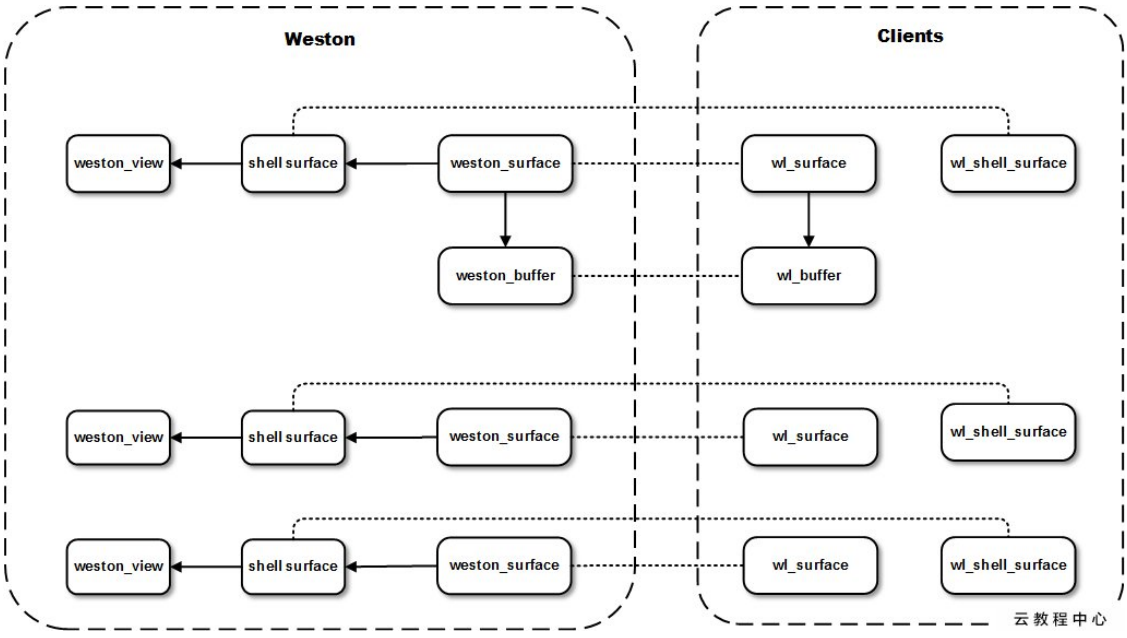
在这条流水线上,可以看到,client和server端都会发生绘制。client绘制本地的窗口内容,server端主要用于合成时渲染。注意两边都可独立选择用软件或者硬件渲染。现在的商用设备上,多是硬件加速渲染。和Android上的SurfaceFlinger和Ubuntu上的Mir一样,Wayland同样基于EGL接口。EGL用于将本地窗口系统与OpenGL关联起来,与WGL, GLX等作用类似,只是它是用于Embedded platform的。在Wayland/Weston系统中,Wayland定义了用于EGL的窗口抽象,来作为EGL stack(也就是厂商的图形驱动)和Wayland协议的glue layer。它对EGL进行了扩展,增加了比如eglBindWaylandDisplayWL, eglUnbindWaylandDisplayWL, eglQueryWaylandBufferWL等接口,对Wayland友好的EGL库应该提供它们的实现,也就是说要提供Wayland EGL platform,比如mesa(src/egl/main/eglapi.c中)。另一种做法是像libhybris中eglplatform一样通过EGL wrapper的方式加上这些支持(hybris/egl/platforms/common/eglplatformcommon.cpp)。同时,EGL stack需要提供厂商相关的协议扩展使client能与compositor共享buffer。wayland-egl库提供了Wayland中surface和EGL粘合的作用。一个要用硬件加速的EGL window可以基于Wayland的surface创建,即通过wayland-egl提供的接口创建wl_egl_window。wl_egl_window结构中包含了wl_surface,然后wl_egl_window就可以被传入EGL的eglCreateWindowSurface()接口。这样就将Wayland surface与EGL stack联系在一起了。

窗口管理

前面提到,buffer需要有surface为载体,这个surface可以理解为一个窗口的绘制表面。如果一个Wayland client的窗口要被窗口管理器(Shell)所管理,则需要为这个surface创建相应的shell surface。理一下这里相关的几个核心概念:surface,view,shell surface。首先,surface是Weston中的核心数据结构之一。Surface代表Wayland client的一个绘图表面。Client通过将画好的buffer attach到surface上来更新窗口,因此说surface是buffer的载体。在Weston中,shell是窗口管理器,因此一个surface要想被窗口管理器管理,需要创建相应的shell surface。同时shell surface对应的其实是surface的一个view。view是surface的一个视图。换句话说,同一个surface理论上可以有多个view,因此weston_surface结构中有view的列表。这里和我们逻辑上的窗口的概念最近似的是view,因为它对应用户所看到的一个窗口。而当surface与view是1:1关系时(绝大多数情况下),surface也可近似等同于窗口。在server端它们的数据结构是相互关联的。



如果从Server/Client角度,它们的相互对应关系如下:

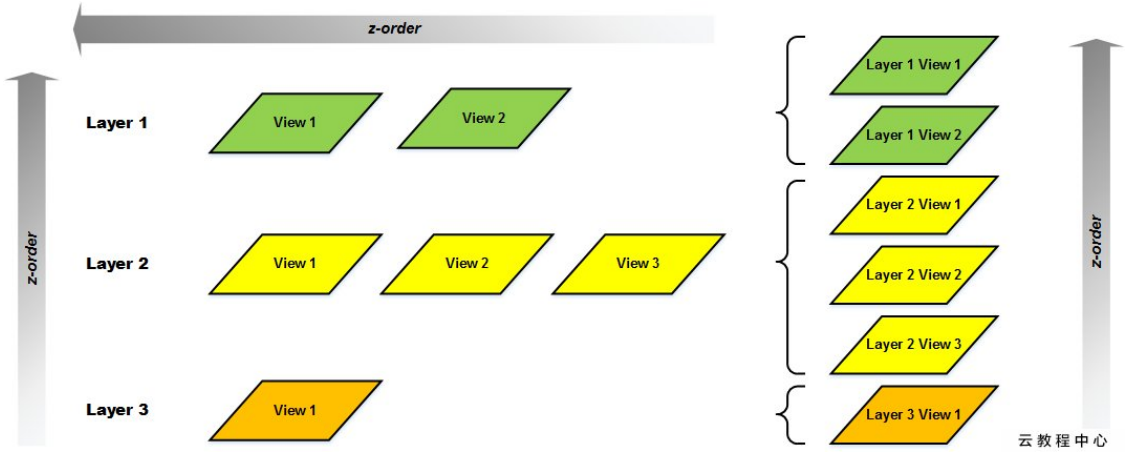


另外subsurface可以作为surface的附属绘图表面,它与父surface保持关联,但拥有独立的绘图surface,类似于Android中的SurfaceView,作用也是类似。主要用于Camera,Video等应用。

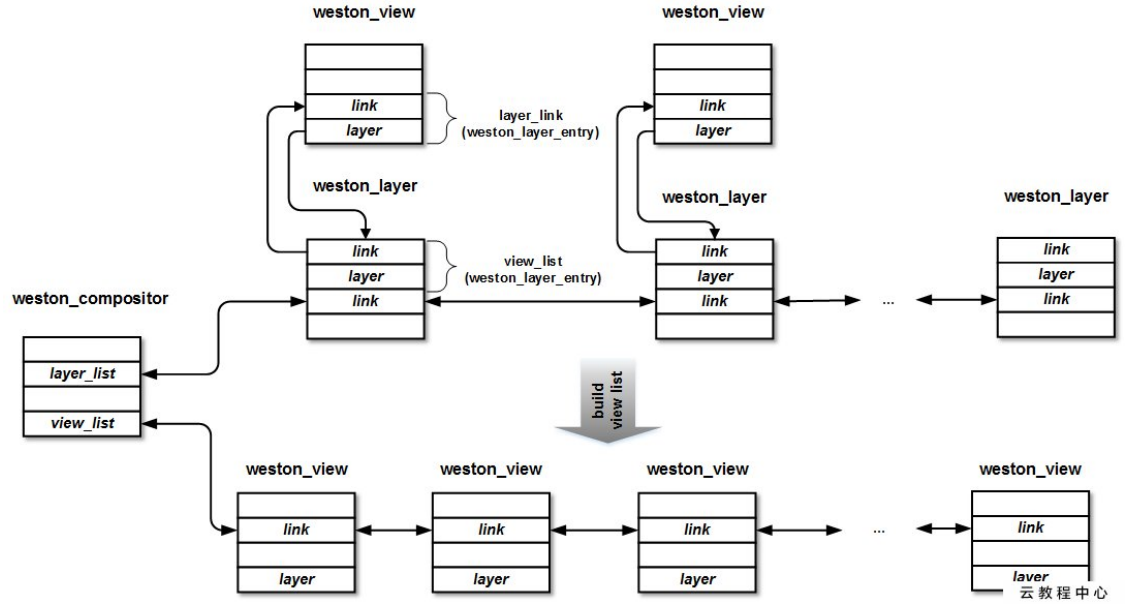
窗口管理不是直接管理view,而是分为两层进行管理。Layer是中间层,系统中的layer大体有下面几个,按从上到下顺序为:

- Fade layer
- Lock layer
- Cursor layer
- Input panel layer
- Fullscreen layer
- Panel layer
- Workspace layers
- Background layer

其中的workspace layer是一个数组,默认是一个,也可以有多个,其数量可以在weston.ini中指定。大部分的普通应用窗口就是在这个layer中。这些layer被串成list。每次在要做合成时,会首先将这些layer的view按顺序合并到一个view list中。这样,在composition过程中compositor只需访问这个view list。



可以看到,这是一个二层有序列表合成一个有序列表的过程。这和Android中的WMS通过为每种类型的窗口定义一段z轴区域的原理类似。WMS中每个类型的窗口对定一个基数(定义在WindowManager.Java),它会乘以乘数再加上偏移从而得到z轴上的区域边界。区别在于Weston中不是以数值而是有序列表表示z-order。结合具体的数据结构:



输入管理

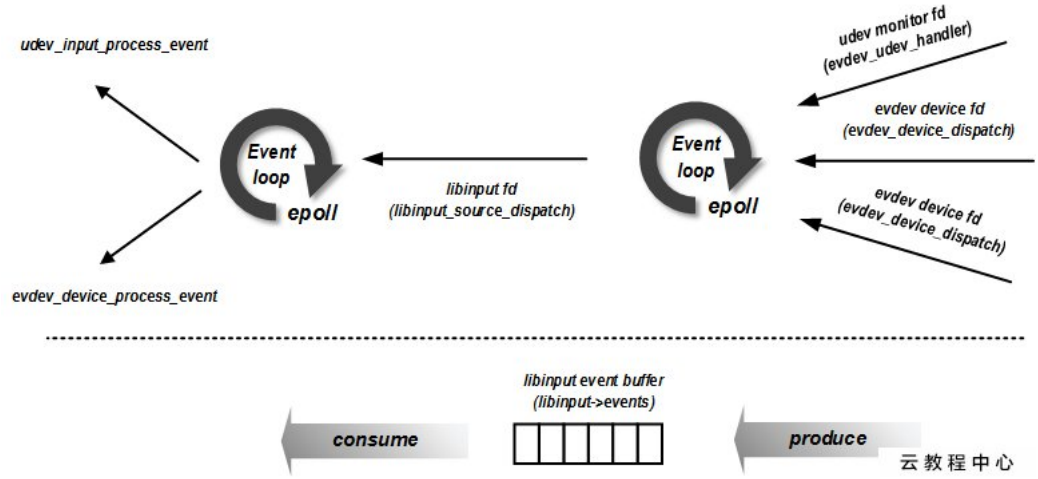
为了提高输入管理部分的重用性和模块性。Weston将对输入设备(键盘,鼠标,触摸屏等)的处理分离到一个单独的库,也就是libinput中。这样,其它的图形处理系统也可以共用这部分,比如X.Org Server和Mir。具体地,它提供了设备检测,设备处理,输入事件处理等基本功能,类似于Android中的EventHub。此外它还有pointer acceleration,touchpad support及gesture recognition等功能。libinput更像是一个框架,它将几个更底层的库的功能整合起来。它主要依赖于以下几个库:

• mtdev:Multi-touch设备处理,比如它会将不带tracking ID的protocol A转化为protocol B。

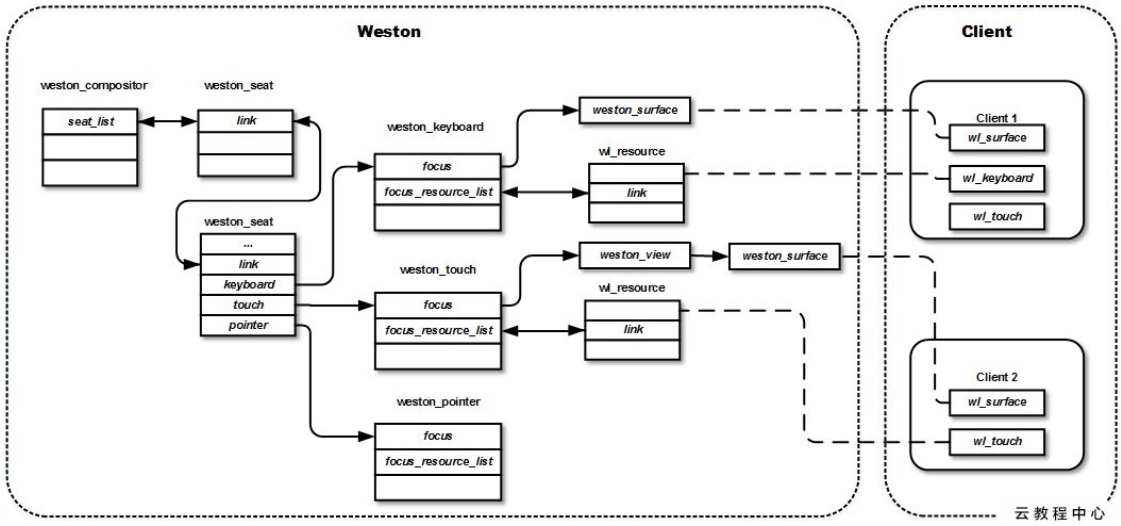
• libevdev:处理kernel中evdev模块对接。

• libudev:主要用于和udevd的通信,从而获取设备的增加删除事件。也可从kernel获取。

Weston中的输入管理模块与libinput对接,它实现了两大部分的功能:一是对输入设备的维护,二是对输入事件的处理。对于输入事件既会在Weston中做处理,也会传给相应的client。从事件处理模型上来看,libinput的主循环监听udev monitor fd,它主要用于监听设备的添加删除事件。如果有设备添加,会打开该设备并把fd加入到libinput的主循环上。另一方面,Weston中会将libinput的epoll fd加入主循环。这样形成级联的epoll,无论是udev monitor还是input device的fd有事件来,都会通知到Weston和libinput的主循环。这些事件通过libinput中的事件缓冲队列存储,而Weston会作为消费者从中拿事件并根据事件类型进行处理。



Weston中支持三种输入设备,分别是键盘,触摸和鼠标。一套输入设备属于一个seat(严格来说,seat中包括一套输入输出设备)。因此,weston_seat中包含weston_keyboard,weston_pointer和weston_touch三个结构。系统中可以有多个seat,它们的结构被串在weston_compositor的seat_list链表中。相应的数据结构如下。



可以看到,对于焦点处理,每个设备有自己的focus,它指向焦点窗口,用于拖拽和输入等。成员focus_resource_list中包含了焦点窗口所在client中输入设备proxy对应的resource对象。在这个list中意味着可以接收到相应的事件。

最后,Wayland/Weston作为正在活跃开发的项目,还有其它很多功能已被加入或正被加入进来。本文只是粗线条的介绍了Wayland/Weston主要结构及功能,具体细节之后再展开。

以上是Wayland与Weston简介的内容，更多 Wayland 简介 Weston 的内容，请您使用右上方搜索功能获取相关信息。

相关文章

Android-SlideSupport-ListLayouts 使用简介

Ubuntu 17.10调整：Dock始终可见 默认使用Wayland

Wayland中的跨进程过程调用浅析

揭开Wayland的面纱（一）：X Window的前生今世

Red Hat和Fedora团队欢迎Ubuntu重新回归至GNOME和Wayland

wayland环境搭建起步之虚拟机。

Fedora 25 将是第一个默认采用 Wayland 显示服务器的发行版

Fedora 25 将默认使用 Wayland

X11 Wayland 及 Mir 比较

pycharm 在wayland的gnome下提示没有JDK_HOME

热门活动

你可能感兴趣

阿里云教程中心为您免费提供 Waylar
介相关信息，包括 Wayland 简介 W
息，所有Wayland与Weston简介相关
阿里云的意见！投稿删除文章请联系
zixun-group@service.aliyun.com，工
个工作日内答复

移动版： Wayland与Westo

咨询·建议