目录

1. 概述

Linux DRM 层包含旨在支持复杂图形设备的需求的代码，该图形设备通常包含适合 3D 图形加速度的可编程管道。内核中的图形驱动程序可以利用 DRM 函数，来使内存管理，中断处理和 DMA 等任务更加容易，并为应用程序提供统一的接口。

注：本指南涵盖了 DRM 树中发现的功能，包括 TTM Memory Manager，输出配置和模式设置以及新的 VBLANK Internals，此外还包括当前内核中的所有常规功能。

英文在线文档：

http://landley.net/kdocs/htmldocs/drm.html

2. DRM 内部

本章记录 DRM 内部与 driver 作者和开发人员有关的 DRM 内部设备，以增加对现有驱动程序的最新功能的支持。

首先，我们遵循一些典型的驱动程序初始化要求，例如设置命令缓冲区、创建初始输出配置和初始化核心服务。随后的部分更详细地涵盖了核心内部内容，提供了实施说明和示例。

DRM 层为图形驱动程序提供了多种服务，其中许多是由它通过 LIBDRM 提供的，应用程序界面驱动的。Libdrm 是包含大量 DRM IOCTLS 的接口。其中包括 vblank 事件处理、内存管理、输出管理、FrameBuffer 管理、命令提交

和 fence，休眠/唤醒的支持和 DMA 服务。

2.1 驱动初始化

每个 DRM 驱动程序的核心是 drm_driver 结构。 驱动程序通常会静态地初始化 drm_driver 结构，然后将其传递到 drm _*_ init（ ）函数之一，以将其注册为 DRM 子系统。

drm_driver 结构包含描述驱动程序和功能支持的静态信息，并指示 DRM Core 将调用以实现 DRM API 的方法。 我们将首先浏览 drm_driver 静态信息字段，然后在以后的部分中使用详细信息来描述单个操作。

2.1.1 驱动信息

- **驱动功能**

驱动通过在 driver_features 字段中设置适当的标志来告知 DRM 核心的需求和支持的功能。 由于这些标志自注册时间以来会影响 DRM 核心行为，因此必须将其中大多数设置为注册 drm_driver 实例(这里的意思是要初始化，不要处于未初始化的不确定状态)。

```
u32 driver_features;
```

- **驱动功能标识**

DRIVER_USE_AGP - 驱动使用 AGP 接口，drm core 会管理 AGP 资源

DRIVER_REQUIRE_AGP - 驱动需要 AGP 接口函数。AGP 初始化失败会是一个严重错误。

注：AGP(Accelerated Graphics Port)即加速图形端口。它用于连接显示设备的接口，是为了提高视频带宽而设计的一种接口规范。早期的显示接口卡通过 ISA 总线或者 PCI 总线与主板连接，但是 ISA、PCI 显卡均不能满足 3D 图形/视频技术的发展要求。PCI 显卡处理 3D 图形有两个主要缺点，一是 PCI 总线最高数据传输速度仅为 133MB/s，不能满足处理 3D 图形对数据传输率的要求。二是需要足够多的显存来进行图像运算，这将导致显示卡的成本很高。AGP 接口把显示部分从 PCI 总线上拿掉，使其它设备可以得到更多的带宽，并为显示卡提供高达 1064MB/s(AGP 4x)的数据传输速率。**AGP 以系统内存为帧缓冲(Frame Buffer)，可将纹理数据存储在其中，从而减少了显存的消耗，实现了高速存取，有效地解决了 3D 图形处理的瓶颈问题**。

DRIVER_PCI_DMA - 驱动程序能够使用 PCI DMA，将启用 PCI DMA 缓冲区对用户空间的映射。已弃用。

DRIVER_SG - 驱动程序可以执行 scatter/gather DMA，将启用 scatter/gather 缓冲区的分配和映射。已弃用。

DRIVER_HAVE_DMA - 驱动支持 DMA，支持用户层的 DMA 接口。已弃用。

DRIVER_HAVE_IRQ - 指示驱动是否具有由 DRM 核心管理的 IRQ 处理程序。设置标志时，核心将支持简单的 IRQ 处理程序安装。安装过程在称为“ IRQ 注册”部分中描述。

DRIVER_IRQ_SHARED - 指示设备和处理程序是否支持共享 IRQ（请注意，这是 PCI 驱动程序所需的）。

DRIVER_GEM - 驱动使用 GEM 内存管理器

DRIVER_MODESET - 驱动支持模式设置接口（ KMS ）

DRIVER_PRIME - 驱动实现了 DRM PRIME 缓冲区的共享，这个是为实现多显卡准备的

DRIVER_RENDER - 驱动支持专用的渲染节点，即 render 和 kms 在用户层使用不同的设备节点。

- Major，Minor 和 PatchLevel

```
int major;
int minor;
int patchlevel;
```

DRM Core 通过 major，minor 和 patchlevel 的三重态标识驱动程序版本。该信息在初始化时打印到内核日志，并通过 drm_ioctl_version ioctl 传递给用户空间。

Major 和 minor 数字还用于验证传递给 DRM_IOCTL_SET_VERSION 的请求的驱动程序 API 版本。当驱动程序 API 更改次要版本时，应用程序可以调用 DRM_IOCTL_SET_VERSION 选择 API 的特定版本。如果请求的 major 不等于驱动的 major，或者所请求的 minor 大于驱动的 minor，则 DRM_IOCTL_SET_VERSION 调用将返回错误。 否则，将使用请求的版本调用驱动程序的 set_version（ ）方法。

- Name，Descriptin 和 Date

```
char *name;
char *desc;
char *date;
```

驱动程序名称在初始化时打印到内核日志，用于 IRQ 注册，并通过 DRM_IOCTL_VERSION 传递给用户空间。

驱动程序描述是通过 DRM_IOCTL_VERSION ioctl 传递给用户空间的纯粹有用的字符串，否则将不被内核使用。

格式为 YYYYMMDD 的驱动日期旨在确定驱动的最新修改日期。 但是，由于大多数驱动程序无法更新它，因此其价值大多是没有用的。 DRM Core 在初始化时间将其打印到内核日志,并通过 DRM_IOCTL_VERSION ioctl 将其传递给用户空间。

2.1.2 驱动加载

load 方法是驱动程序和设备初始化入口点。该方法负责分配和初始化驱动程序的私人数据，指定支持的性能计数器，执行资源分配和映射（例如获取时钟，映射寄存器或分配命令缓冲区），初始化内存管理器（“存储器管理”部分），安装 IRQ 处理程序(称为“ IRQ 注册”的部分),设置垂直空白处理(称为“Vertical Blanking”），模式设置（称为“KMS”）和初始输出配置（该部分称为“ KMS 初始化”部分 和清理”）。

注：如果兼容性是一个问题（例如，将驱动程序从用户模式设置转换为内核模式设置），则必须注意防止设备初始化和控制与当前活动的用户空间驱动程序不相容的设备初始化和控制。 例如，如果使用了用户级别模式设置驱动程序，则在加载时执行输出发现和配置是有问题的。同样，如果不使用内存管理的用户级驱动程序,则可能需要省略内存管理和命令缓冲区设置。这些要求是特定于驱动的，需要注意保持旧应用程序、新应用程序和库的工作。

```
int (*load) (struct drm_device *, unsigned long flags);
```

该方法采用两个参数：指向新创建的 DRM_DEVICE 的指针和创建标志。标志用于传递与传递给 drm _*_ init（ ）的设备相对应的设备 ID 的 driver_data 字段。目法的标志为 0。

- Driver Private & Performance Counters

驱动程序私有悬挂在主 drm_device 结构上，可用于跟踪各种特定于设备的信息，例如寄存器偏移，命令缓冲区状态，休眠/唤醒的寄存器状态等。在加载时，驱动可以简单地分配一个，并适当设置 drm_device.dev_priv；当驱动卸载时，应将其释放，并将其设置为 null。

DRM 支持几个用于粗糙性能表征的计数器。 此统计计数器系统已弃用，不应使用。如果需要性能监控，开发人员应调查并有可能增强内核 perf 并追踪基础架构，以通过性能监控工具和应用程序导出相关的绩效信息，以供使用。

- IRQ 注册

DRM Core 试图通过提供 drm _irq_install 和 drm_irq_uninstall 功能来处理 IRQ 处理程序的注册和未注册。这些功能仅支持每个设备的单个中断，需要手动处理多个 IRQ 的设备。

- 托管的 IRQ 注册

drm_irq_install 和 drm_irq_uninstall 功能都通过调用 drm_dev_to_irq 获取设备 IRQ。 此内联函数将调用特定于 BUS 的操作以检索 IRQ 号码。 对于平台设备，使用 platform_get_irq（...，0）来检索 IRQ 号码。

drm_irq_install 首先调用 irq_preinstall 驱动程序操作。 该操作是可选的，必须确保通过清除所有待处理的中断标志或禁用中断不会被打开。

然后，将通过呼叫 request_irq 请求 IRQ。 如果设置了

DRIVER_IRQ_SHARED 驱动程序功能标志，则将请求共享（IRQF_SHARED）IRQ 处理程序。

必须提供 IRQ 处理程序功能作为强制性 IRQ_HANDLER 驱动程序操作。它将直接传递给 request_irq，因此具有与所有 IRQ 处理程序相同的原型。它将用指向 DRM 设备的指针作为第二个参数被调用。

最后，该功能调用可选的 irq_postinsetall 驱动程序操作。 该操作通常会启用中断（不包括 vblank 中断，vblank 中断分开启用），但是驱动程序可以选择在不同时间启用/禁用中断。

drm_irq_uninstall 类似地用于卸载 IRQ 处理程序。首先要唤醒所有在 vblank 中断的过程，以确保它们不会悬挂，然后调用可选的 IRQ_UNINSTALL 驱动程序操作。该操作必须禁用所有硬件中断。最后，该功能通过调用 free_irq 来释放 IRQ。

- 手动 IRQ 注册

需要多个中断处理程序的驱动程序无法使用托管的 IRQ 注册功能。 在这种情况下，必须手动注册 IRQ 并未注册（通常使用 request_irq 和 free_irq 函数，或其 devm_*等价）。

当手动注册 IRQ 时，驱动程序不得设置 DRIVER_HAVE_IRQ 驱动程序功能标志，并且不得提供 irq_handler 驱动程序操作。他们必须在注册 IRQ 时将 drm_device_irq_enabled 字段设置为 1，并在取消注册 IRQ 后将其清除为 0。

2.1.3 内存管理器的初始化

每个 DRM 驱动程序都需要一个内存管理器，该内存管理器必须在加载时间

初始化。DRM 当前包含两个内存管理器，即 Translation Table Manager（ TTM ）和 Graphics Execution Manager（ GEM ）。 本文档仅描述仅使用 GEM 内存管理器。 有关详细信息，请参见称为“内存管理”的部分。

2.1.4 混杂设备配置

配置过程中 PCI 设备可能需要的另一个任务是映射 vBIOS。在许多设备上，VBIOS 描述了设备配置，LCD 面板正时（如果有），并包含指示设备状态的标志。 可以使用 pc_map_rom（ ）调用来映射 BIOS，这是一种便利函数，它可以照顾实际的 ROM，无论是已被遮蔽到内存中（通常在地址 0xC0000 ）还是在 ROM 栏中的 PCI 设备上存在。 请注意，在映射了 ROM 并提取了任何必要的信息后，它应该是未覆盖的； 在许多设备上，ROM 地址解析与 BAR 共享，因此将其映射到可能会导致不希望的行为，例如悬挂或内存损坏。

2.2 内存管理

现代 Linux 系统需要大量的图形内存来存储框架缓冲区、纹理、顶点和其他与图形相关的数据。鉴于许多数据的非常动态的性质，因此有效地管理图形存储器对于图形堆栈至关重要，并且在 DRM 基础架构中起着核心作用。

DRM 核心包括两个内存管理器，即翻译表映射（ TTM ）和 Graphics Execution Manager（ GEM ）。 TTM 是第一个开发的 DRM 内存管理器，并试图成为所有解决方案。它提供了一个单个用户空间 API，可满足所有硬件的需求，并使用专用的视频 RAM（即最离散的视频卡）支持统一的内存体系结构（ UMA ）设备和离散内存设备。这导致了一块大型、复杂的代码，事实证明很难用于驱动开发。

GEM 最初是由 Intel 赞助的项目，应对 TTM 的复杂性。它的设计理念是完全不同的：GEM 并没有为每个图形内存有关的问题提供解决方案，而是确定了驱动程序之间的常见代码，并创建了一个支持库来共享它。 GEM 比 TTM 具有更简单的初始化和执行要求，但没有视频 RAM 管理可容纳，因此仅限于 UMA 设备。

2.2.1 翻译表映射（TTM）

- TTM 初始化

希望支持 TTM 的驱动必须填写 drm_bo_driver 结构。 该结构包含一些带有功能指针的字段，用于初始化 TTM，分配和释放内存，等待命令完成和 fence 同步以及内存迁移。有关用法的示例，请参见 Radeon_ttm.c 文件。

```
struct ttm_global_reference {
        enum ttm_global_types global_type;
        size_t size;
        void *object;
        int (*init) (struct ttm_global_reference *);
        void (*release) (struct ttm_global_reference *);
};
```

整个内存管理器应该有一个全局参考结构，并且在运行时内存管理器创建的每个对象都会有其他参考结构。您的全局 TTM 应该具有 ttm_global_ttm_mem 的类型。全局对象的大小字段应为 sizeof(struct ttm_mem_global)，并且 init 和 release

挂钩函数应指向您的特定于驱动程序的初始化和释放例程，该例程最终可能分别调用 ttm_mem_global_init 和 ttm_mem_global_rease。

通过在其上调用 ttm_global_item_ref（）设置和初始化的全局 TTM 会计结构后，您需要创建一个缓冲对象 ttm 来为客户和内核本身提供缓冲对象分配的池。该对象的类型应为 ttm_global_ttm_bo，其大小应为大小（struct ttm_bo_global）。同样，可以提供特定于驱动程序的初始化和发布功能，最终可能分别拨打 ttm_bo_bo_global_init（）和 ttm_bo_global_release（）。 同样，与先前的对象一样，ttm_global_item_ref（）也用于创建 TTM 的初始参考计数，该参考计数将调用您的初始化函数。

2.2.2 图形执行管理器（GEM）

GEM 设计方法产生了一个内存管理器，该内存管理器并未在其用户空间或内核 API 中完全覆盖所有（甚至所有常见）用例。 GEM 向用户空间公开了一组与标准内存相关的操作，并向驱动程序公开了一组辅助功能，并让驱动使用自己的私有 API 实施特定于硬件的操作。

GEM 用户空间 API 在 LWN 上的 GEM-执行管理器文章中进行了描述。虽然过时，但该文档对 GEM API 原理的做了很好的概述。当前使用特定于驱动程序的 IOCTL 实现缓冲区分配和读写操作，这些操作是通用 GEM API 的一部分。

GEM 是数据不敏感的。它在不知道各个缓冲器包含内容的情况下管理抽象缓冲对象。因此，了解缓冲内容或目的的需求，例如缓冲区分配或同步原语，是在 GEM 的范围之外的，必须使用特定于驱动程序的 IOCTL 来实现。

在基本层面上，GEM 涉及多个操作：

- 内存分配和释放
- 命令执行
- 命令执行时间的光圈管理

缓冲对象分配相对简单，并且很大程度上由 Linux 的 shmem 层提供，该层提为每个对象分配内存。

设备特定的操作，例如命令执行、固定（pinning）、缓冲区读写，映射和域所有权转移，将留给特定于驱动程序的 IOCTL。

- GEM 初始化

使用 GEM 的驱动程序必须在 struct drm_driver driver_features 字段中设置 DRIVER_GEM 位。 然后，DRM Core 将在调用加载操作之前自动初始化 GEM Core。这将创建一个 DRM 内存管理器对象，该对象为对象分配提供地址空间池。

在 KMS 配置中，如果硬件需要，驱动程序需要遵循核心 GEM 初始化的方式，分配和初始化命令环缓冲区（command ring buffer）。 UMA 设备通常具有所谓的“被盗”内存区域，该区域为设备所需的初始框架和大型连续内存区域提供了空间。 该空间通常不受 GEM 管理，必须单独初始化为其自己的 DRM MM 对象。

- GEM 对象创建

GEM 拆分了 GEM 对象的创建和内存的分配，在两个不同操作中支持它们。

GEM 对象由 struct drm_gem_object 的实例表示。 驱动程序通常需要使用私有信息扩展 GEM 对象，从而创建一个特定于驱动程序的 GEM 对象结构类型，该类型嵌入了 struct drm_gem_object 的实例。

要创建一个 GEM 对象，驱动程序为其特定的 GEM 对象类型的实例分配内存，并用呼叫 drm_gem_object_init 初始化嵌入式的 struct drm_gem_object。该函数将一个指针指向 DRM 设备，一个指向 GEM 对象的指针和字节中的缓冲区大小。

GEM 使用 shmem 分配匿名的分页内存。 drm_gem_object_init 将创建请求大小的 shmfs 文件，并将其存储到 struct drm_gem_object filp 字段中。 当图形硬件直接使用系统内存时就在系统内存上分配。否则，将作为备用的存储。

通过调用每个页面的 shmem_read_mappe_page_gfp，驱动程序负责实际的物理页面分配。 请注意，他们可以在初始化 GEM 对象时决定分配页面，或者延迟分配直到需要内存（例如，当由于用户空间访问或驱动程序需要启动涉及的 DMA 传输时，在页面故障时发生页面故障时）。

并非总是需要匿名的分页内存分配，例如，当硬件需要物理连续的系统存储器时，比如在嵌入式设备中经常这样做。驱动程序可以通过呼叫 drm_gem_private_object_init 而不是 drm_gem_object_init 来初始化它们来创建没有 shmfs 备份（称为私有 GEM 对象）的 GEM 对象。 私人 GEM 对象的存储必须由驱动程序管理。

不需要使用私有信息扩展 GEM 对象的驱动程序可以调用 drm_gem_object_alloc 函数以分配和初始化 struct drm_gem_object 实例。 GEM Core 用 drm_gem_object_init 初始化 GEM 对象后，可以调用驱动程序 drm_gem_object_init 操作。

```
int (*gem_init_object) (struct drm_gem_object *obj);
```

对于私有 GEM 对象，不存在分配和初始化功能。

- GEM 对象生存周期

所有 GEM 对象均由 GEM 核心引用。可以分别调用 drm_gem_object_reference 和 drm_gem_object_unreference 来获取和发布引用。 调用者必须持有 drm_device 的 struct_mutex 锁。 为了方便起见，GEM 提供了 drm_gem_object_reference_unlocked 和 drm_gem_object_unreference_unlocked 函数，可以在不持有锁的情况下调用。

当释放 GEM 对象的最后一个引用时，GEM Core 调用 drm_driver 的 gem_free_object 操作。该操作是针对 GEM 驱动程序的强制性操作，必须释放 GEM 对象和所有相关资源。

```
void（ *gem_free_object ）（ struct drm_gem_object *obj ）;
```

驱动程序负责释放所有 GEM 对象资源，包括 GEM Core 创建的资源。 如果为对象创建了 mmap 偏移量（在这种情况下，drm_gem_object :: map_list :: map 不是 null ），则必须通过调用 drm_gem_free_mmap_offset 释放。必须通过调用 drm_gem_object_release（如果未创建 SHMFS 备份商店，调用这个函数也是安全的）来释放 SHMFS 备份池。

- GEM 对象命名

用户空间与内核之间的通信是指使用本地句柄、全局名称或最近的文件描述符的 GEM 对象。所有这些都是 32 位整数值。通常的 Linux 内核限制适用于文件描述符。

GEM 句柄是 DRM 文件的本地化。应用程序通过特定于驱动程序的 IOCTL 获得 GEM 对象的句柄，并可以使用该句柄在其他标准或特定于驱动程序的

IOCTL 中引用 GEM 对象。关闭 DRM 文件句柄释放其所有 GEM 句柄并取消对 GEM 对象的引用计数。

调用 drm_gem_handle_create 为 GEM 对象驱动程序创建一个句柄。 该功能将指针指向 DRM 文件和 GEM 对象，并返回本地化的唯一句柄。 当不再需要句柄时，驱动程序用 drm_gem_handle_delete 的调用将其删除。最后，可以通过调用 drm_gem_object_lookup 检索与句柄关联的 GEM 对象。

句柄不具有 GEM 对象的所有权，它们只会引用当句柄被摧毁时将丢弃的对象。为了避免泄漏 GEM 对象，驱动程序必须确保丢弃自己拥有的参考（例如在对象创建时间的初始参考值），而无需对句柄进行任何特殊考虑。例如，在混合 GEM 对象的特定情况下，在实现 dumb_create 操作中的创建中，驱动程序必须在返回句柄之前将 GEM 对象的初始引用删除（这里通常是指引用计数减一）。

GEM 名称与句柄相似，但不是 DRM 文件本地化属性。它们可以在过程之间传递，以在全局引用 GEM 对象。名称不能直接用于引用 DRM API 中的对象，应用程序必须使用 DRM_IOCTL_GEM_FLINK 和 DRM_IOCTL_GEM_OPEN IOCTLS 实现句柄和名称的转换。转换由 DRM 核心处理，没有任何特定驱动程序的支持。

与全局名称类似，GEM 文件描述符也用于跨进程共享 GEM 对象。 他们提供了其他安全性：由于必须在应用程序之间共享的 UNIX domain socket 上明确发送文件描述符，因此无法像全局唯一的 GEM 名称一样被猜测。

支持 GEM 文件描述符（也称为 DRM Prime API）的驱动程序必须在 struct drm_driver driver_features 字段中设置 DRIVER_PRIME 位，并实现 prime_handle_to_fd 和 prime_fd_to_to_handle 操作。

```
int (*prime_handle_to_fd)(struct drm_device *dev,
                          struct drm_file *file_priv, uint32_t handle,
                          uint32_t flags, int *prime_fd);
 int (*prime_fd_to_handle)(struct drm_device *dev,
                          struct drm_file *file_priv, int prime_fd,
                          uint32_t *handle);
```

这两个操作将句柄转换为 Prime 文件描述符，反之亦然。 驱动程序必须使用内核 DMA-BUF 缓冲区共享框架来管理 Prime 文件描述符。

虽然非 GEM 驱动程序必须本身实施操作，但 GEM 驱动程序必须使用 drm_gem_prime_handle_to_fd 和 drm_gem_prime_fd_to_handle 帮助函数功能。这些帮助者依靠驱动程序 gem_prime_export 和 gem_prime_import 操作从 GEM 对象（DMA-BUF 导出者角色）创建 DMA-BUF 实例，并从 DMA-BUF 实例（DMA-BUF 导入者角色）创建 GEM 对象。

```
struct dma_buf * (*gem_prime_export)(struct drm_device *dev,
                                     struct drm_gem_object *obj,
                                     int flags);
struct drm_gem_object * (*gem_prime_import)(struct drm_device *dev,
                                            struct dma_buf *dma_buf);
```

对于支持 DRM Prime 的 GEM 驱动程序，这两个操作是必须的。

- DRM PRIME 帮助函数

驱动程序可以通过使用助手函数 drm_gem_prime_export 和 drm_gem_prime_import 来实现 API gem_prime_export 和 gem_prime_import。这些函数以五个底层驱动程序回调来实现 DMA-BUF 支持：

Export 回调函数:

- gem_prime_pin (optional):为导出准备一个 GEM 对象

- gem_prime_get_sg_table: 为固定的页面提供 scatter/gather 表

- gem_prime_vmap: 对你驱动导出的 buffer 做 vmap

- gem_prime_vunmap: 对你驱动导出的 buffer 做 vunmap

Import 回调函数:

- gem_prime_import_sg_table (import): 从另一个去的的 scatter/gather 表中生成一个 GEM 对象

- GEM 对象的映射

因为映射操作是通过特定于驱动程序的 ioctls 实现的重量级 GEM 读/写式访问，而不是将缓冲区映射到用户空间。但是，当需要随机访问缓冲区（例如执行软件渲染）时，直接访问对象可能会更有效。

MMAP 系统调用不能直接用于映射 GEM 对象，因为它们没有自己的文件句柄。目前，两种替代方法将 GEM 对象映射到用户空间：

1）第一种方法使用特定于驱动程序的 IOCTL 执行映射操作，并在引擎盖下调用 do_mmap。 这通常被认为是不可靠的，GEM 驱动程序开发者并不热衷，因此在这里不再描述。

2）第二种方法在 DRM 文件句柄上使用 MMAP 系统调用。

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
                        off_t offset);
```

DRM 标识了通过 MMAP 偏移参数传递的假偏移来映射的 GEM 对象。在映射之前，必须将 GEM 对象与假偏移相关联。为此，驱动程序必须在对象上调用 drm_gem_create_mmap_offset。该函数分配了一个从池的假偏移范围，并在 obj-> map_list.hash.key 中以 page_size 划分的偏移量。如果已经为对象分配了假偏移，则必须注意不要调用 drm_gem_create_mmap_offset。

分配后，必须以特定于驱动程序的方式将假偏移值（obj-> map_list.hash.key << page_shift）传递给该应用程序，然后可以用作 MMAP 偏移参数。

GEM 核心提供了辅助方法 drm_gem_mmap 来处理对象映射。 该方法可以直接设置为 mmap 文件操作处理程序。它将根据偏移值查找 GEM 对象，并将 VMA 操作设置为 drm_driver gem_vm_ops 字段。 请注意，drm_gem_mmap 不会将内存映射到用户空间，但依靠驱动程序提供的故障处理程序单独映射页面。

要使用 drm_gem_mmap，驱动程序必须用指向 VM 操作的指针填充 struct drm_driver gem_vm_ops 字段。

```
struct vm_operations_struct *gem_vm_ops

struct vm_operations_struct {
        void (*open)(struct vm_area_struct * area);
        void (*close)(struct vm_area_struct * area);
        int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
}
```

打开和关闭操作必须更新 GEM 对象参考计数。 驱动程序可以使用 drm_gem_vm_open 和 drm_gem_vm_close 助手直接用作打开和关闭处理程序的功能。

故障操作处理程序负责在发生页面故障时将单个页面映射到用户空间。根据内存分配方案，驱动程序可以在故障时分配页面，或者决定在创建对象时为 GEM 对象分配内存。

想要预先映射 GEM 对象而不是处理页面故障的驱动程序可以实现自己的 mmap 文件操作处理程序。

- Dumb GEM 对象

GEM API 不会标准化 GEM 对象创建，并将其留在特定于驱动程序的 IOCTL 上。 虽然对于包括设备特定的用户空间组件（例如在 LIBDRM 中）的全面图形堆栈而言，这不是问题，但此限制使得基于 DRM 的设备启动过程中启动图形变得复杂。

dumb 的 GEM 对象可以通过提供标准 API 来创建适合扫描的 dumb 缓冲区，从而部分缓解了问题，然后可以将其用于创建 KMS 框架缓冲区。

为了支持 dumb 的 GEM 对象，驱动程序必须实现 dumb_create，dumb_destroy 和 dumb_map_offset 操作。

```
int (*dumb_create)(struct drm_file *file_priv, struct drm_device *dev,
                   struct drm_mode_create_dumb *args);
```

dumb_create 操作创建了一个基于 struct drm_mode_create_dumb 参数的宽度、高度和深度的 GEM 对象。它用新创建的 GEM 对象、间距和字节单位的尺寸填充了参数的句柄、线的间距和大小字段。

```
int (*dumb_destroy)(struct drm_file *file_priv, struct drm_device *dev,
                    uint32_t handle);
```

dumb_destroy 用来销毁有 dumb_create 创建的 dumb GEM 对象。

```
int (*dumb_map_offset)(struct drm_file *file_priv, struct drm_device *dev,
                       uint32_t handle, uint64_t *offset);
```

dumb_map_offset 操作将 mmap 假偏移与句柄给出的 GEM 对象相关联并返回。驱动程序必须使用 drm_gem_create_mmap_offset 函数来关联假偏移量，如“ GEM 对象映射”部分所述。

- 内存的一致性

当一个对象的页面被映射到设备或在命令缓冲区中使用时，它将被会更新写入内存并将标记为写入合并，以便与 GPU 一致。 同样，如果 CPU 在 GPU 完成对象渲染后访问对象，则必须使对象与 CPU 的内存视图一致，通常涉及各种 GPU CACHE 刷新。 在核心 CPU <-> GPU 一致性的管理由设备特定都 IOCTL 提供，该设备特定的 IOCTL 评估对象的当前域并执行任何必要的 flush 或同步以将对象放入所需的连贯域（请注意对象可能忙，比如正在主动渲染目标，在这种情况下，设置域将阻止客户端，并等待渲染完成，然后再执行任何必要的 flush 操作）。

- 命令的执行

GPU 设备最重要的 GEM 函数也许是向客户端提供命令执行接口。 客户端程序构造包含对先前分配的内存对象的引用，然后将其提交给 GEM。到那时，GEM 小心将所有对象绑定到 GTT 中，执行缓冲区，并在访问相同缓冲区的客户端之间提供必要的同步。这通常涉及驱逐 GTT 中的某些对象并重新绑定其他对象（这是相当昂贵的操作），并提供了迁移支持，从而使固定的 GTT 偏移在客户处隐藏起来。 客户必须注意不要提交引用多个对象的命令缓冲区，该命令缓冲区引用的对象超过了 GTT 中能容纳的对象；否则，GEM 将拒绝它们，不会发生任何渲染。同样，如果缓冲区中的几个对象需要分配 fence 寄存器以进行正确的渲染（例如，在 965 芯片前的 2D 片段），则必须注意不要需要比客户可用的更多围栏寄存器。 该资源管理应从 LIBDRM 中的客户中抽象。

2.3 模式设置

驱动程序必须通过在DRM设备上调用drm_mode_config_init来初始化模式设置核心。 该函数初始化了 drm_device mode_config 字段，并且永远不会失败。完成后，必须通过初始化以下字段来设置模式配置。

```
int min_width, min_height;
int max_width, max_height;
```

framebuffer 的属性，单位是像素。

```
struct drm_mode_config_funcs *funcs;
```

模式设置的函数。

2.3.1 framebuffer 的创建

```
struct drm_framebuffer *(*fb_create)(struct drm_device *dev,
                                     struct drm_file *file_priv,
                                     struct drm_mode_fb_cmd2 *mode_cmd);
```

framebuffer 是抽象存储器对象，可做为扫描输出到 CRTC 上的像素来源。应用程序明确要求通过 DRM_IOCTL_MODE_ADDFB（2）IOCTLS 创建帧缓冲区，并接收一个可以传递到 KMS CRTC 控件、平面配置和页面翻转功能的不透明句柄。

帧缓冲区依靠底层的内存管理器进行底层内存操作。创建帧缓冲区应用程序时，通过 drm_mode_fb_cmd2 参数传递内存句柄（或多平面格式的内存列表）。本文档假设驱动程序使用 GEM，这些就是引用 GEM 对象。

驱动程序必须首先验证通过 mode_cmd 参数传递的请求的帧缓冲区参数。特别是可以捕获无效的大小、像素格式或间距。

如果将参数视为有效，驱动程序就创建、始化并返回 struct drm_framebuffer 的实例。如果需要，该实例可以嵌入到较大的驱动程序特定结构中。驱动程序必须从通过 drm_mode_fb_cmd2 参数传递的值中填充其宽度、高度、俯仰、偏移、深度，bits_per_pixel 和 pixel_format 字段。他们应该调用 drm_helper_mode_fill_fb_struct 助手函数进行此操作。

新的 FrameBuffer 实例的启动通过调用 drm_framebuffer_init 完成，该实例将指向 DRM 帧缓冲区操作（struct drm_framebuffer_funcs)）。 请注意，此功能发布了帧缓冲程序，因此从此开始，可以从其他线程同时访问它。因此，它必须是驱动框架初始化序列中的最后一步。帧缓冲区操作是：

```
int（ *create_handle ）（ struct drm_framebuffer *fb，
struct drm_file *file_priv，unsigned int *handle ）;
```

在内存对象下面的帧缓冲区上创建一个句柄。如果帧缓冲区使用多平面格式，则句柄将引用与第一平面关联的内存对象。

驱动调用 drm_gem_handle_create 创建句柄。

```
void（ *destroy ）（ struct drm_framebuffer *framebuffer ）;
```

销毁帧缓冲对象并释放所有相关资源。驱动程序必须调用 drm_framebuffer_cleanup 以释放由 DRM Core 分配给帧缓冲区对象的资源，并且必须确保取消与帧缓冲区关联的所有内存对象。DRM Core 释放由 create_handle 操作创建的句柄。

```
int (*dirty)(struct drm_framebuffer *framebuffer,
             struct drm_file *file_priv, unsigned flags, unsigned color,
```

```
                struct drm_clip_rect *clips, unsigned num_clips);
```

此可选操作通知驱动程序帧缓冲区的区域已响应 DRM_IOCTL_MODE_DIRTYFB ioctl 调用而发生了变化。

DRM FrameBuffer 的生存周期由参考计数控制，驱动程序可以使用 drm_framebuffer_reference 获取其他参考。并用 drm_framebuffer_unreference 再次取消参考它们。对于驱动程序私有的帧缓冲器，最后一个引用将永远不被删除，驱动程序可以用 drm_framebuffer_unregister_private 在模块卸载时手动清理帧缓冲器。

2.3.2 输出轮询

```
void (*output_poll_changed)(struct drm_device *dev);
```

此操作通知驱动程序是否已更改一个或多个连接器的状态。 使用 FB 助手的驱动程序只能调用 drm_fb_helper_hotplug_event 函数来处理此操作。

2.3.3 锁

除了某些带有自己锁定的查找结构（隐藏在接口功能内）外，大多数模式状态都受到 dev- <mode_config.lock mutex 的保护，另外是每个 crtc 锁允许 cursor 更新，pagelfips 和类似操作发生，以及诸如输出检测之类的背景任务。像完整模式设置（full modeset）之类的跨域的操作总是持有所有锁。那里的驱动需要保护 CRTC 之间共享的资源，并具有额外的锁定。 如果模式设置影响了 CRTC 状态，例如：为了进行负载检测（这仅获得了 mode_config.lock，以允许在实时 CRTC 上进行并发屏幕更新）。

2.4 KMS 的初始化和清除

KMS 设备被抽象和暴露为一组 planes, CRTCs, encoders 和 connectors。 因此，KMS 驱动程序必须在初始化模式设置后在加载时创建并初始化所有这些对象。

2.4.1 CRTC（struct drm_crtc）

CRTC 是代表芯片一部分的抽象，其中包含指向扫描缓冲区的指针。 因此，可用的 CRTC 的数量决定了可以活跃的、独立的扫描缓冲区。 CRTC 结构包含几个字段来支持这一点：

- 指向某些视频内存的指针（作为帧缓冲区对象抽象）
- 显示模式
- （x，y）偏移到视频内存中，以支持 panning 或配置，其中一部分内存跨越多个 CRTC

2.4.1.1 CRTC 初始化

KMS 设备必须创建和注册至少一个 struct drm_crtc 实例。该实例由驱动程序分配和销毁，可能是较大结构的一部分，并通过调用 drm_crtc_init 进行注册，并用指针指向 CRTC 函数。

2.4.1.2 CRTC 的操作

- 设置参数

```
int (*set_config)(struct drm_mode_set *set);
```

将新的 CRTC 配置应用于设备。该配置指定了一个 CRTC，一个可以从在帧缓冲区中的一个点 a（x，y）位置扫描出来的缓冲区，显示模式和一系列连接器，以便使用 CRTC 驱动。

如果配置中指定的帧缓冲区为空，则驱动程序必须卸载连接到这个 CRTC 的所有编码器，以及连接到这些编码器的所有连接器并禁用它们。

此操作是在持有模式配置锁定的情况下调用的。

- 页面翻转

```
int (*page_flip)(struct drm_crtc *crtc, struct drm_framebuffer *fb,
                 struct drm_pending_vblank_event *event);
```

将页面翻转到 CRTC 的给定帧缓冲区，这个过程中持有模式设置的 Mutex。

页面翻转是一种同步机制，它在垂直空白期间用新的帧缓冲区替换用要扫描出去的帧缓冲区，从而避免了撕裂。当应用程序请求页面翻转时，DRM Core 验证了新的帧缓冲区足够大，可以在当前配置的模式下由 CRTC 扫描，然后调用 CRTC page_flip 操作指向新帧缓冲区。

page_flip 调度页面翻转这个操作。一旦对针对新帧缓冲区的任何待处理渲染完成，将对 CRTC 进行重新编程，以在下一个垂直刷新后显示该帧缓冲区。该操作必须立即返回，而无需等待渲染或页面翻转完成，并且必须阻止帧缓冲区的任何新渲染，直到页面翻转完成。

如果可以成功安排页面翻转，则驱动程序必须将 drm_crtc->fb 字段设置为 FB 指向的新 FrameBuffer。这很重要，以便对帧缓冲器的参考计数保持平衡。

如果页面翻转已经待处理，则 page_flip 操作必须返回-EBUSY。

为了同步页面翻转垂直同步，驱动程序可能需要启用垂直空白中断。 它应为此调用 drm_vblank_get,并在页面翻转完成后调用 drm_vblank_put。

如果已要求在页面翻转完成时通知应用程序，则将调用 page_flip 函数指向 drm_pending_vblank_event 实例的非 NULL 事件参数。在页面翻转完成后，驱动程序必须调用drm_send_vblank_event填写事件并发送、以唤醒所有等待过程。这可以通过如下方式：

```
spin_lock_irqsave(&dev->event_lock, flags);
...
drm_send_vblank_event(dev, pipe, event);
spin_unlock_irqrestore(&dev->event_lock, flags);
```

- 其它

```
void（ *set_property ）（ struct drm_crtc *crtc，
                struct drm_property *属性，uint64_t value ）;
```

将给定 CRTC 属性的值设置为给定值。有关属性的更多信息，请参见称为“ KMS 属性”的部分。

```
void（ *gamma_set ）（ struct drm_crtc *crtc，u16 *r，u16 *g，u16 *b，
                uint32_t start，uint32_t size ）;
```

在设备上应用伽马表，这个操作是可选的。

```
void（ *destroy ）（ struct drm_crtc *crtc ）;
```

在不需要的情况下销毁 CRTC。请参阅称为“ KMS 初始化和清理”的部分。

2.4.2 Planes(struct drm_plane)

Plane 表示可以在扫描过程中与 CRTC 顶层混合或覆盖的图像源。 Plane 与帧缓冲区相关联，以裁剪图像存储区，并可选地将其扩展到目标大小。然后将结果与 CRTC 顶层的或覆盖层混合或叠加。

2.4.2.1 Planes 的初始化

Planes 是可选的，要创建 planes，KMS 驱动程序分配和释放一个 struct drm_plane（可能是较大结构的一部分）的实例，是通过调用 drm_plane_init 进行注册。该函数参数包括与 plane 关联的 CRTC、指向 plane 操作函数的指针以及支持格式的列表。

```
int drm_plane_init(struct drm_device *dev, struct drm_plane *plane,
                   uint32_t possible_crtcs,
                   const struct drm_plane_funcs *funcs,
                   const uint32_t *formats, unsigned int format_count,
                   bool is_primary)
```

2.4.2.2 Planes 的操作

```
int（ *update_plane）（struct drm_plane *plane，struct drm_crtc *crtc，
                       struct drm_framebuffer *fb，int crtc_x，int crtc_y，
                       unsigned int crtc_w，unsigned int crtc_h，
                       uint32_t src_x，uint32_t src_y，
                       uint32_t src_w，uint32_t src_h）;
```

启用并配置 plane 以使用给定的 CRTC 和帧缓冲区。

帧缓冲器内存坐标中的源矩形由 SRC_X，SRC_Y，SRC_W 和 SRC_H 参数给出。不支持子像素 plane 坐标的设备可以忽略小数部分。

CRTC 坐标中的目标矩形由 crtc_x、crtc_y、crtc_w、crtc_h 参数（作为整数值）给出。 设备将源矩形扩展到目标矩形。 如果不支持缩放，并且源矩形大小与目标矩形大小不匹配，则驱动程序必须返回-EINVAL 错误。

int（ *disable_plane）（struct drm_plane *plane）;

禁用 plane。DRM Core 以响应 DRM_IOCTL_MODE_SETPLANE IOCTL 调用，将帧缓冲区 ID 设置为 0。

void（ *destroy）（struct drm_plane *plane）;

不再需要时销毁 plane。 请参阅称为“ KMS 初始化和清理”的部分。

2.4.3 Encoders(struct drm_encoder)

编码器从 CRTC 中获取像素数据，并将其转换为适用于任何连接器的格式。在某些设备上，有可能将 CRTC 发送到多个编码器的数据。 在这种情况下，两个编码器都会从相同的扫描缓冲区接收数据，从而在每个编码器上连接的连接器上产生“克隆”显示配置。

2.4.3.1 Encoder 初始化

至于 CRTC，必须创建、初始化和注册一个 struct drm_encoder 实例。该实例由驱动程序分配和释放，可能是较大结构的一部分。

在注册编码器之前，驱动程序必须初始化 struct drm_encoder 的 possible_crtcs 和 possible_clones 字段。 这两个字段分别是可以连接到编码器的 CRTC 的位掩码，而这些可用的 crtc 链接在 clones 字段。

初始化后，必须在调用 drm_encoder_init 的情况下注册编码器。 该函数将指针指向编码器函数和编码器类型。 支持类型包括：

- DRM_MODE_ENCODER_DAC - VGA and analog on DVI-I/DVI-A
- DRM_MODE_ENCODER_TMDS - DVI, HDMI 和(embedded) DisplayPort
- DRM_MODE_ENCODER_LVDS - display panels
- DRM_MODE_ENCODER_TVDAC - TV output (Composite, S-Video, Component, SCART)
- DRM_MODE_ENCODER_VIRTUAL - virtual machine displays

必须将编码器附加到要使用的 CRTC 上。 DRM 驱动程序在初始化时将编码器设置为未绑定状态。 应用程序（或实现的 FBDEV 兼容层）负责将他们要使用的编码器附加到 CRTC 上。

2.4.3.2 Encoder 的操作

```
void（ *destroy）（struct drm_encoder *encoder）;
```

在不需要的情况下调用次方法销毁编码器。请参阅称为“ KMS 初始化和清理”的部分。

```
void（ *set_property）（struct drm_plane *plane，
                struct drm_property *attribute，uint64_t value）;
```

将给定 plane 属性的值设置为给定值。有关属性的更多信息，请参见称为“KMS 属性”的部分。

2.4.4 Connector(struct drm_connector)

连接器是设备上像素数据的最终目的地，通常直接连接到显示器或笔记本电脑面板等外部显示设备。连接器一次只能连接到一个编码器。 连接器也是保留有关显示信息的结构，因此它包含用于显示数据、EDID 数据，DPMS 和连接状态的字段，以及有关显示屏上支持的模式的信息。

2.4.4.1 Connector 初始化

最后，KMS 驱动程序必须创建、初始化、注册和附加至少一个 struct drm_connector 实例。该实例是作为其他 KMS 对象创建的，并通过设置以下字段来初始化。

- interlace_allowed - 连接器是否可以处理交错模式。
- doublescan_allowed - 连接器是否可以处理 dublescan。
- display_info - 检测到显示时，显示信息是从 EDID 信息中填写的。 对于非热插拔的显示，例如嵌入式系统中的平板面板，驱动程序使用显示屏的物理大小初始化 display_info.width_mm 和 display_info.height_mm 字段。
- polled -连接器轮询模式，可以是如下组合：
  - DRM_CONNECTOR_POLL_HPD - 连接器会生成热插拔事件，不需要定期进行轮询。连接和断开标志不得与 HPD 标志设置在一起。
  - DRM_CONNECTOR_POLL_CONNECT - 定期轮询连接器是否连接。
  - DRM_CONNECTOR_POLL_DISCONNECT - 定期轮询连接器是否断开
  - 如果不支持连接状态发现的连接器，设置为 0。。

然后，将连接器注册，并通过调用 drm_connector_init 的调用，并用指向连接器功能和连接器类型的指针进行注册，使用 drm_sysfs_connector_add 通过 sysfs 将属性输出到用户空间。

支持的连接器类型：

- DRM_MODE_CONNECTOR_VGA
- DRM_MODE_CONNECTOR_DVII
- DRM_MODE_CONNECTOR_DVID
- DRM_MODE_CONNECTOR_DVIA
- DRM_MODE_CONNECTOR_Composite
- DRM_MODE_CONNECTOR_SVIDEO
- DRM_MODE_CONNECTOR_LVDS
- DRM_MODE_CONNECTOR_Component
- DRM_MODE_CONNECTOR_9PinDIN
- DRM_MODE_CONNECTOR_DisplayPort
- DRM_MODE_CONNECTOR_HDMIA
- DRM_MODE_CONNECTOR_HDMIB
- DRM_MODE_CONNECTOR_TV
- DRM_MODE_CONNECTOR_eDP
- DRM_MODE_CONNECTOR_VIRTUAL

连接器必须连接到要使用的编码器上。以 1：1 方式将连接器映射到编码器，应在初始化时调用 drm_mode_connector_attach_encoder 附加连接器。驱动程序还必须将 drm_connector 编码器字段设置为指向附加的编码器。

最后，驱动程序必须通过调用 drm_kms_helper_poll_init 来初始化连接器状态更改检测。如果至少一个连接器是可以进行 poll，但无法生成热插拔中断（由 DRM_CONNECTOR_POLL_CONNECT 和 DRM_CONNECTOR_POLL_DISCONNECT 连接器标志表示），则将自动排队以定期进行更改。可以使用 DRM_CONNECTOR_POLL_HPD 标志标记可以生成热插入中断的连接器，并且其中断处理程序必须调用

drm_helper_hpd_irq_event。该功能用排队延迟的工作方式检查所有连接器的状态，但不会进行定期轮询。

2.4.4.2 Connector 的操作

- DPMS

```
void (*dpms)(struct drm_connector *connector, int mode);
```

DPMS 操作设置了连接器的功率状态。 模式参数如下：

- DRM_MODE_DPMS_ON
- DRM_MODE_DPMS_STANDBY
- DRM_MODE_DPMS_SUSPEND
- DRM_MODE_DPMS_OFF

除 DPMS_ON 模式外，所有连接器所在的编码器都应通过适当地驱动其信号来将显示器置于低功率模式。如果将多个连接器连接到编码器，则一个连接器状态的变化不应该影响其它的（面效应）。当所有相关连接器都以低功率模式放置时，应将低功率模式传播到编码器和 CRTC。

- 模式

```
int (*fill_modes)(struct drm_connector *connector, uint32_t max_width,
                  uint32_t max_height);
```

用连接器的所有支持模式填写模式列表。如果 max_width 和 max_height 参数非零，则实现必须忽略所有比 max_width 或更宽的模式、比 max_height 更高的模式。

连接器还必须以毫米为单位，用连接的显示器的尺寸填充 display_info 的 width_mm 和 Height_mm 字段。如果值不知道或不适用，则应将字段设置为

0（例如，用于投影仪设备）。

- 连接状态

```
enum drm_connector_status (*detect)(struct drm_connector *connector,
                                    bool force);
```

通过轮询或热插拔事件更新连接状态（如果支持）。 状态值通过 IOCTL 报告给用户空间，不得在驱动程序内使用，因为它只能通过用户空间对 drm_mode_getconnector 的调用来初始化。

检查连接器是否附加了任何设备。在使用 poll 方式是，force 设置为 false；当由用户请求才检查时，force 设置为 true。驱动可以使用 force 来避免自动探测过程中昂贵的破坏性操作。

驱动程序只能返回 connector_status_connected，如果已连接的连接状态确实进行了探测。 无法检测连接状态或失败连接状态探针的连接器应返回 connector_status_unknown。

- 其它

```
void（ *set_property）（struct drm_connector *connector，
                        struct drm_property *attribute，uint64_t value）；
```

将给定连接器属性的值设置为给定值。有关属性的更多信息，请参见称为“ KMS 属性”的部分。

```
void（ *destroy）（struct drm_connector *connector）；
```

在不需要的情况下销毁连接器。请参阅称为“ KMS 初始化和清理”的部分。

DRM 核心管理其对象的生命周期。 当不再需要对象时，核心调用其销毁功能，该功能必须清理并释放分配给对象的每个资源。每个 drm _*_ init 调用都

必须与相应的 drm_*_cleanup 匹配，以清理 CRTC（drm_crtc_cleanup），Planes（drm_plane_cleanup），编码器（drm_encoder_cleanup）和连接器（drm_connector_cleanup）。此外，在调用 drm_sysfs_connector_remove 的呼叫拨打。在调用 drm_connector_cleanup 之前，必须通过 drm_sysfs_connector_remove 将 connectros 从 sysfs 里卸载掉。

2.4.5 KMS API

2.5 模式设置助手函数

驱动程序提供的 CRTC、编码器和连接器功能实现了 DRM API。DRM Core 和 IOCTL 处理程序将调用以处理设备状态更改和配置请求。由于实施这些功能通常需要逻辑不是针对驱动程序的逻辑，因此可以使用中层辅助辅助功能来避免复制样板代码。

DRM 核心包含一个中层实现。 中层提供了几个 CRTC，编码器和连接器功能（从中层的顶层调用）的实现，这些功能预先处理请求并调用驱动程序提供的下级功能（在中间层的底部）。 例如，drm_crtc_helper_set_config 函数可用于填充 struct drm_crtc_funcs set_config 字段。 当调用时，它将以较小，更简单的操作将 SET_CONFIG 操作拆分，并调用驱动程序来处理它们。

在注册相应的 KMS 对象后，安装中层底部操作处理程序最好立即完成。

中层不分配 CRTC，编码器和连接器操作。要使用它，驱动程序必须为所有三个实体提供底部功能。

- Helper Functions
- CRTC Helper Operations
- Encoder Helper Operations
- Connector Helper Operations
- Modeset Helper Functions Reference
- fbdev Helper Functions Reference
- Display Port Helper Functions Reference
- EDID Helper Functions Reference
- Rectangle Utilities Reference
- Flip-work Helper Reference
- VMA Offset Manager

2.6 KMS 属性

驱动程序可能需要将其他参数暴露给上一节中描述的应用程序。KMS 支持将属性附加到 CRTC、connector 和 plane，并为用户空间提供 list、get 和 set 的 API。

属性由唯一定义名称标识并存储关联的值。对于除 Blob 属性以外的所有属性类型，该值是一个 64 位的无符号整数。

KMS 区分属性和属性实例。驱动程序首先创建属性，然后创建实例并将其与对象相关联。属性可以多次实例化并与不同的对象相关联。值存储在属性实例中，所有其他属性信息都存储在属性中，并在属性的所有实例之间共享。

每个属性都是用一种影响 KMS 核心处理属性的类型创建的。支持的类型有：

- DRM_MODE_PROP_RANGE

范围属性报告其最小和最大可接受的值。KMS 核心验证该范围内按应用程序设置的值。

- DRM_MODE_PROP_ENUM

枚举属性采用一个数值，范围从 0 到由属性减去一个定义的枚举值的数量，并将一个自由形式的字符串名称与每个值相关联（这里说的就是枚举名字和值）。应用程序可以检索定义的 （值 - 名称） 对的列表，并使用数值来获取和设置属性实例值。

- DRM_MODE_PROP_BITMASK

Bitmask 属性是枚举属性，将所有枚举值限制在（0..63）范围内。Bitmask 属性实例值结合了该属性定义的一个或多个枚举位（就是按位掩码）。

- DRM_MODE_PROP_BLOB

Blob 属性存储一个无需任何格式限制的二进制 blov。 二进制 blob 是作为 kms 独立对象创建的，并且 BLOB 属性实例值存储其关联的 BLOB 对象的 ID。

BLOB 属性仅用于连接器 EDID 属性，驱动程序不能创建。

要创建属性驱动程序，请根据属性类型调用以下功能之一。 所有属性创建功能都采用属性标志和名称以及特定于类型的参数。

```
struct drm_property *drm_property_create_range( struct drm_device *dev,int flags,
                                                const char *name,
                                                Uint64_t min，uint64_t max）;
```

创建具有给定最小值和最大值的范围属性。

```
struct drm_property *drm_property_create_enum( struct drm_device *dev,int flags,
                                               const char *name,
                                               const struct drm_prop_enum_list *props,
                                               int num_values）;
```

创建一个枚举的属性。 props 参数指向 num_values 值 name 对的数组。

struct drm_property *drm_property_create_bitmask （ struct drm_device *dev，

int flag，const char *name，
const struct drm_prop_enum_list *props，
int num_values）；

创建一个 BitMask 属性。 Props 参数指向 num_values 值 name 对的数组。

可以另外创建属性作为不可变的，在这种情况下应用程序只读，但可以由驱动程序修改。要创建一个不变的属性驱动程序，必须在属性创建时间设置 DRM_MODE_PROP_IMMUTABLE 标志。

如果在属性创建时间内不容易获得列举或范围属性的值，则可以使用 DRM_PROPERTY_CREATE 函数创建属性，并通过调用 DRM_PROPERTY_ADD_ENUM 函数来手动添加枚举值 - 名称对。 必须注意通过标志参数正确指定属性类型。

创建属性后，驱动程序可以通过调用 drm_object_attach_property 将属性实例连接到 CRTC，连接器和 plane。该函数参数：指向目标对象，指向先前创建的属性的指针和初始实例值。

```
void drm_object_attach_property(struct drm_mode_object *obj,
                                struct drm_property *property,
                                uint64_t init_val)
```

2.7 垂直遮挡（ vblank ）

垂直隐蔽在图形渲染中起主要作用。为了实现无撕裂显示，用户必须将页面翻转渲和染同步到垂直空白。 DRM API 提供的 IOCTL 可以执行与垂直掩盖同步的页面翻转，并等待垂直空白。

DRM 核心处理大多数垂直空白管理逻辑，其中涉及过滤杂乱无章的中断，保持无竞争的空白计数器，应对计数器环绕和重置以及保持使用计数。它依靠驱动程序来生成垂直空白中断，并选择提供硬件垂直抛弃计数器。驱动必须实施以下操作：

int（ *enable_vblank）（struct drm_device *dev，int crtc）；

void（ *disable_vblank）（struct drm_device *dev，int crtc）；

为给定的 CRTC 启用或禁用垂直隐蔽中断。

u32（ *get_vblank_counter）（struct drm_device *dev，int crtc）；

为给定的 CRTC 检索垂直遮挡计数器的值。如果硬件维护垂直空白，则应返回其值。否则，驱动程序可以使用 drm_vblank_count 助手功能来处理此操作。

驱动程序必须在加载操作中调用 drm_vblank_init 初始化垂直遮挡处理核心。该函数将 struct drm_device vblank_disable_lowered 字段设置为 0。这将使垂直空白中断永久启用，直到第一个模式设置操作为止。在第一个模式设置操作中，vblank_disable_lowled 设置为 1。背后的原因尚不清楚。驱动程序可以在调用 drm_vblank_init 之后将字段设置为 1，以使垂直空白从头开始动态管理。

DRM 核心或驱动程序本身可以启用垂直空白中断（例如处理页面翻转操作）。DRM Core 保持垂直空白使用计数，以确保在用户仍然需要时未禁用中断。 要增加使用计数，驱动程序调用 drm_vblank_get。 返回后，垂直空白被保证可以启用。

要减少使用计数驱动程序调用 drm_vblank_put。只有当使用计数下降到零时，DRM 核心才能通过安排一个计时器在延迟一段后禁用垂直空白。可以通

过 vblankoffdelay 模块参数或 drm_vblank_offdelay 全局变量设定这个延迟时间，并以毫秒为单位。其默认值为 5000 ms。

当垂直掩盖中断时，驱动程序只需要调用 drm_handle_vblank 函数即可解释中断。

由 drm_vblank_init 分配的资源必须在驱动程序卸载操作处理程序中的调用 drm_vblank_cleanup。

2.8 文件操控（open、close、ioctl）

对文件句柄的操作。

2.8.1 打开关闭

```
int (*firstopen) (struct drm_device *);
void (*lastclose) (struct drm_device *);
int (*open) (struct drm_device *, struct drm_file *);
void (*preclose) (struct drm_device *, struct drm_file *);
void (*postclose) (struct drm_device *, struct drm_file *);
```

打开和关闭处理程序。 这些方法都不是强制性的。仅当应用程序打开没有其他打开的文件句柄的设备时，DRM Core 为 Legacy UMS（用户模式设置）驱动程序调用了第一个 open 方法。 UMS 驱动程序可以实施以获取设备资源。KMS 驱动程序无法使用该方法，必须在加载方法中获取资源。

同样，对于 UMS 和 KMS 驱动程序，当持有设备上打开的文件句柄的最后一个应用程序将其关闭时，最后一个 lastclose 方法也会调用。 此外，该方法还在模块卸载时间或热插拔设备的设备拔出时调用。 因此，firstopen 和

lastclose 可能是不平衡的。

每次通过应用程序打开设备时，都会调用打开方法。驱动程序可以在此方法中分配每个文件的私人数据，并将其存储在 struct drm_file driver_priv 字段中。请注意，open 方法在 firstopen 之前被调用。

关闭操作被分为 preclose 和 postclose 方法。驱动必须在 preclose 停止并清理每个文件的操作。例如，必须取消等待垂直空白和页面翻转事件。从 preclose 方法返回后，在文件手柄上不允许使用每文件操作。

最后，如果设备上没有打开的文件句柄了，在调用 lastclose 方法之前，postclose 方法作为关闭操作的最后一步被调用。 在 open 方法中分配了每文件的私人数据的驱动程序应在此处释放分配的私人数据。

lastclose 方法应将 CRTC 和 plane 属性还原为默认值，以便设备的后续打开将不会从先前的用户继承状态。它也可以用于执行延迟的电源切换状态更改，例如：结合 VGA-Switcheroo 基础架构。 除此之外，KMS 驱动不应进一步清理。只有传统的 UMS 驱动程序可能需要清理设备状态，以便 VGA 控制台或独立的 FBDEV 驱动程序可以接管。

2.8.2 文件操作

```
const struct file_operations *fops
```

DRM 设备节点上的文件操作。

驱动程序必须定义形成 DRM 用户空间 API 入口点的文件操作结构，即使大多数操作都在 DRM Core 中实现。

```
.owner = THIS_MODULE,
```

```
        .open = drm_open,
        .release = drm_release,
        .unlocked_ioctl = drm_ioctl,
  #ifdef CONFIG_COMPAT
        .compat_ioctl = drm_compat_ioctl,
  #endif
```

实施需要 32/64 位兼容性支持的私有 IOCTL 的驱动程序必须提供自己的 compat_ioctl 处理程序，该处理程序可以处理私有 IOCTLS 并将 drm_compat_ioctl 调用 core ioctls。

read 和 poll 的操作为读取 DRM 事件并进行轮询提供了支持。

```
.poll = drm_poll,
.read = drm_read,
.llseek = no_llseek,
```

内存映射实现取决于驱动程序如何管理内存。 Pre-GEM 驱动程序将使用 drm_mmap，而 Gem-aware 驱动程序将使用 drm_gem_mmap。

```
.mmap = drm_gem_mmap,
```

2.8.3 IOCTL

```
struct drm_ioctl_desc *ioctls;
int num_ioctls;
```

驱动提供的 ioctl 描述符表。

特定于驱动程序的 IOCTLS 数字从 DRM_COMMAND_BASE 开始。

IOCTLS 描述符表用 IOCTL 编号偏移从基本值开始进行索引。驱动程序可以使用 DRM_IOCTL_DEF_DRV() 宏来初始化表条目。

```
DRM_IOCTL_DEF_DRV(ioctl, func, flags)
```

参数 IOCTL 是 IOCTL 名称。驱动程序必须将 DRM _ ## ioctl 和 DRM_IOCTL _ ## ioctl 宏分别从 DRM_COMMAND_BASE 和 IOCTL 数字定义为 IOCTL 数字偏移。 第一个宏是该设备的私有，而第二个宏必须在公共标题中暴露于用户空间。

func 是指向与 drm_ioctl_t 类型兼容的 IOCTL 处理程序功能的指针。

```
typedef int drm_ioctl_t(struct drm_device *dev, void *data,
                        struct drm_file *file_priv);
```

标志是以下值的位置组合。它限制了如何调用 IOCTL。

- DRM_AUTH - 只有认证用户可调用
- DRM_MASTER - ioctl 只能在 master 的文件句柄才能调用
- DRM_ROOT_ONLY - 仅允许具有 sysadmin 功能的调用者
- DRM_CONTROL_ALLOW - 仅在控制设备上调用 ioctl
- DRM_UNLOCKED - 在不持有 DRM 全局互斥锁的情况下进行 ioctl 调用

2.9 命令提交和 fencing

这应该涵盖一些特定于设备的命令提交实现。

2.10 休眠唤醒

DRM Core 提供了一些休眠/唤醒的代码，但是想要完全休眠/唤醒支持的驱动程序应提供 Save（ ）和 Restore（ ）功能。 在这些被称为休眠，睡眠和唤醒的时间，应在休眠、睡眠眠状态下执行设备所需的任何状态保存或还原。

```
int (*suspend) (struct drm_device *, pm_message_t state);
int (*resume) (struct drm_device *);
```

这些是传统的 suspend 和 resume 方法。新驱动程序应使用其总线类型提供的电源管理接口（通常是通过 struct device_driver dev_pm_ops），并将这些方法设置为 null。

2.11 DMA 服务

这应该涵盖核心如何支持 DMA 映射等。 这些功能被弃用，不应使用。

3. 用户接口

DRM Core 将几个接口导出到应用程序，通常打算通过相应的 LIBDRM 包装器功能使用。此外，驱动程序通过 IOCTLS 和 SYSFS 文件导出了用户空间驱动程序和设备感知应用程序使用的设备特定接口。

外部接口包括：内存映射，上下文管理，DMA 操作，AGP 管理，VBLANK 控制，fence 管理，内存管理和输出管理。

在此处介绍通用的 IOCTL 和 SYSFS 布局。 我们只需要高级信息，因为帮助手册应涵盖其余信息。

3.1 渲染接口

DRM Core 为用户空间提供了多个字符设备。根据打开设备的不同，用户空间可以执行不同的操作集（主要是 IOCTL）。主节点始终创建并称为“card <Num>”。 此外，还创建了当前未使用的控制节点，称为“ controlD <num>”。主节点提供所有旧操作，从历史上看，这是用户空间使用的唯一接口。使用 KMS，引入了控制节点。但是，计划的 KMS 控制接口从未编写过，因此控制节点迄今尚未使用。

随着离屏渲染器和 GPGPU 应用程序的使用越来越多，客户不再需要使用 GPU 来运行的合成器或图形服务器。但是 DRM API 要求非私人的客户在获得 GPU 访问之前对 DRM-Master 进行身份验证。为了避免此步骤并授予客户 gpu 访问而无需进行身份验证，引入了渲染节点。渲染节点仅服务于渲染客户端，也就是说，可以在渲染节点上发出任何模式或特权的 IOCTL。仅允许非全局渲染命令。如果驱动程序支持渲染节点，则必须通过“DRIVER_RENDER” DRM 驱动程序功能进行声明。如果不支持，则必须将主节点与旧的 DRMAUTH 身份验证过程一起渲染客户端。

如果驱动程序声明对渲染节点支持，DRM Core 将创建一个单独的渲染节点，称为“renderD <Num>”。每个设备将有一个渲染节点。除了与 Prime 相关的 IOCTL 以外，其它 IOCTL 都允许在该节点将允许使用。特别是 GEM_OPEN 将被明确禁止。渲染节点旨在避免缓冲泄露，例如客户在旧接口上

猜测 flink 名称或 mmap 偏移，就会发生这种情况。 除此基本接口外，驱动程序必须将其驱动程序依赖性仅限的 IOCTL 标记为“DRM_RENDER_ALLOW”，以便渲染客户端可以使用它们。 驱动开发者必须小心，不要在渲染节点上允许任何特权的 IOCTL。

使用渲染节点，用户空间现在可以通过基本文件系统访问模型，控制对渲染节点的访问。 不再需要在特权初级/旧版节点上身份验证客户端的运行图形服务器。相反，客户端可以打开渲染节点，并立即授权 GPU 访问。客户（或服务器）之间的通信是通过 Prime 完成的。 不支持从渲染节点到传统节点的 FLINK 操作。新客户不得使用不安全的 Flink 接口。

除了放弃所有模式/全局 IOCTL 之外，渲染节点还去除了 DRM-Master 的概念。没有理由将客户端与 DRM 主机相关联，因为它们独立于任何图形服务器。此外，无论如何，它们必须在没有任何执行的 master 的情况下工作。如果驱动程序支持渲染节点，则必须能够在没有 master 对象的情况下运行。另一方面，如果驱动程序需要在客户空间可见的客户之间共享状态，并且在开放式边界之外可访问，则他们无法支持渲染节点。

3.2 vblank 事件处理

DRM core 暴露除了两个垂直遮盖相关的 ioctl：

DRM_IOCTL_WAIT_VBLANK

这将 struct drm_wait_vblank 结构作为其参数，用于在指定的 VBlank 事件发生时阻止或请求信号。

DRM_IOCTL_MODESET_CTL

这个应该通过在模式设置之前和之后的应用级驱动程序调用，因为在许多设备

上，垂直空白计数器当时已重置。在内部，DRM 快照最后一个 Vblank 计数，使用_DRM_PRE_MODESET 命令调用 IOCTL，以便计数器不会向后移动（当使用_DRM_POST_MODESET 时，该计数器已处理）。