**A Generic and Interactive 2D Rendering Engine for Linux**

**Cocoa Manual for Developers**

1. **Introduction and Overview**

Cocoa是针对于Linux平台的交互式2D渲染引擎。Cocoa的工作方式类似于Web浏览器，它加载本地的XML文档、CSS样式表以及Javascript脚本，这三者将指导Cocoa如何完成渲染和交互任务。XML文档指定要渲染的UI元素的布局，CSS指定UI元素的细节属性和观感，Javascript实现动态交互。Cocoa定义了多种显示后端(Display Backend, Display Driver)，这些后端可以适应不同的底层硬件，而为上层提供统一的接口，即Cocoa中的显示后端本质上是对系统显示设备进行了设备无关的抽象。Cocoa同样从标准HID设备接受输入，具备鼠标指针、键盘等基本元素，支持触摸屏和手势操作[1]。

Cocoa定义了一系列在屏幕上渲染内容的规范(Specification)和相应的实现，本规范是其中一的一部分。“Cocoa Manual for Developers”简述了Cocoa标准实现的核心架构、关键技术以及部分代码的注解。

1. **Core Architecture**

Cocoa是十分复杂的软件系统，包含了显示、交互、硬件抽象、渲染引擎、Javascript引擎等多个子系统组成。因此组织这些系统并使之有序、正确地工作极为重要。“Cocoa Core Framework (Cocoa核心框架)”负责了该工作。Cocoa核心框架是对C++特性的若干拓展，使用了大量的C++11, C++20特性，以维持代码的简洁和高效。后文将详细介绍Cocoa核心框架的规范和实现。

Cocoa中硬件无关的代码占据绝大多数。这些子系统在同一层级互相配合完成渲染任务。它们包括XML解析器、CSS解析器、Javascript引擎、Ciallo渲染引擎等。XML解析器构建DOM树，CSS解析器构建CSS规则集，Ciallo渲染引擎组合DOM树和CSS规则集，进而构建出渲染树。DOM树表达了最原始的UI内容和布局，CSS规则集修饰DOM树上的元素，赋予它们更详细的位置和视觉属性，这些属性由布局引擎映射到渲染树上。Ciallo渲染引擎负责维护渲染树，它自底向上逐个遍历结点，这样就完成了一次渲染。最后调用光栅化器将画面显示到具体设备上。有关Cocoa渲染流水线的详细信息，将在后文介绍。

* 1. **Persistent Object Management**

垃圾回收(Garbage Collection)是大部分基于VM的语言理所当然的功能，C++作为一门编译型语言，除了智能指针之外没有任何类似垃圾回收的机制。这给全局对象的管理带来了困难：我们不希望将需要持久化存在的对象作为全局变量声明，这不符合Cocoa的设计原则；但是若不这样，我们便需要一直传递指向某个或多个持久化全局对象的指针，这十分繁琐且效率低下。

在这样的背景下，Cocoa核心框架设计了类似于垃圾回收机制的全局对象管理机制，也称持久化对象管理机制。该机制的核心思想在于，让每一个全局对象保存指向它自己的指针(作为静态成员)，同时公开一个instance静态方法返回该指针。现在我们可以以T::instance()的形式来访问指向持久化对象的指针了。但是这些对象并不知道自己什么时候应该被析构，因此这里还需要引入一个类GOMCollector来管理它们。GOMCollector作为以全局变量形式定义的全局对象而存在，每新建一个持久化对象，就调用它的GOMCollector::addObject方法使其托管该指针，当GOMCollector自身被析构时(用户调用OCFinalize时)，所有被它托管的对象(指针)都会被正确析构(析构顺序与调用addObject的顺序相反，即最后一个被托管的对象最先被析构)。

所有希望使用GOMCollector的持久化对象都应该继承QObject对象(见下文Chapter 2.3)，并在定义时在类的开头使用GOM\_DECLARE(...)，在实现时使用GOM\_DEFINE(...)，在构造函数末尾使用GOM\_CONSTRUCT\_DONE。

**注意，必须在程序一开始就调用OCInitialize函数初始化，在程序结束时调用OCFinalize函数析构所有对象。**

下列代码演示了一个使用GOMCollector的实例。

class T : public QObject {

GOM\_DECLARE(T)

public:

T();

~T() = default;

void print(int x);

};

GOM\_DEFINE(T)

T::T() {

// Do something...

GOM\_CONSTRUCT\_DONE

}

void T::print(int x) {

std::cout << “T::print(” << x << “)” << std::endl;

}

要使用该持久化对象：

void callPrint() {

T::instance()->print(2233);

}

int main(int argc, char const \*\*argv)

{

OCInitialize(); /\* Initialize object holder \*/

GOM->addObject(new T()); /\* Allocate and construct \*/

callPrint();

OCFinalize(); /\* Garbage collection \*/

return 0;

}

上述代码演示了在不传递指针参数的情况下调用T类的print方法。在Cocoa实现中，大量地使用了这种方式来保存持久化对象。GOMCollector的实现位于src/core/ObjectHolder.cc中。

* 1. **Properties Representation Tree**

除了持久化对象的维护，GOMCollector还解决另一个问题——对象之间的数据共享。不同的对象之间可能共享不同类型、不同内容的数据，它们通常以键值对的形式出现，即由一个字符串类型的名称可以唯一确定一个对应的数据。由此我们联想到可以使用类似文件系统的方式，通过名为“路径”的索引字符串来唯一确定一个数据，而“目录(Directory)”就是一组数据的集合。由此，我们定义如下几条规范：

1. 路径(Path)可以唯一确定一个数据键或一个数据集；
2. 目录(Directory)是一组数据和该目录的子级目录的集合；
3. 路径序列是一个包含若干目录名称和数据键的有序序列，路径是路径序列的字符串表示，正斜杠“/”用以分隔不同的名称和键；
4. 初始状态下，有称为“根目录”的空集，它是所有数据键和目录的全集；
5. 路径序列以根目录“/”开始，以数据键或目录结束，数据键不能出现在路径序列的中间位置。

根据上述几条规则，GOMCollector便实现了名为Properties Representation Tree(PRT)的数据共享机制。PRT是一棵树，它作为数据的容器而存在，之所以选用树这种数据结构是因为目录间的层级关系和自相似性使用树很容易表示。通过GOMCollector提供的registerDirectory方法，可以注册一个目录，注册目录时必须指定该目录的拥有者(即一个QObject继承类的指针)；然后使用setProperty系列方法，可以在已存在的目录下创建一个数据键，并给出其对应的数据值。下列代码演示了如何设置要共享的数据：

class T : public QObject {

public:

T(char const \*name) {

GOM->registerDirectory(this, “/ClassT”);

GOM->setPropertyString(“/ClassT/name”, name);

}

};

要获得数据：

char const \*name = GOM->getPropertyString(“/ClassT/name”);

更详细的API文档会在附录中给出。

* 1. **Signal and Slot (QObject)**

在*Chapter 2.2*中，我们已经介绍了一中对象间通讯的方法，即跨对象的数据共享机制。但面对复杂的显示需求，对象之间不仅要能够互相共享部分数据，还需要能够调用另外的对象内的方法，于是“信号和槽(下文简称为信号槽)”机制应运而生。

按照传统的方法，要调用一个对象的某个方法，必须先获得指向该对象的指针和方法名。这种方法使方法的调用在编译期就被决定，无法在运行时动态地更改，且必须维护对象的指针，使得不同的对象之间依赖关系变得混乱。信号槽机制的核心在于，增加一个中间的表示层，掩盖不同的对象之间的细节差异(例如对象的指针不同)，仅仅保留需要被调用的方法的一致性(参数列表和返回值一致)即可。

要实现这一思想，首先需要了解C++方法调用的本质。类中的方法也被编译器视作一个函数，它并不是依赖于对象而存在，而是依赖于类而存在。也就是说，对于某个类中的某个方法，不管这个类有没有被实例化为对象，该方法的地址是已经确定了的。而方法调用的本质是，**将this指针的值作为方法函数的第一个参数，然后紧跟用户定义的参数列表，进行汇编层面的函数调用。**例如在x86\_64 GCC下，编译器在调用方法之前，将对象指针保存到%rdi寄存器，这等效于将对象指针作为函数调用的第一个参数。即，下面这两段代码在调用原理和过程上是相同的：

struct T {

...

};

void method(T \*pThis, int i0) { ... }

class T {

public:

void method(int i0) { ... }

};

在了解了C++方法调用的本质后，我们就只需要对象指针和目标指向成员函数的指针即可。唯一的问题是，如何在调用时正确填充参数？不同的编译器在不同的平台上，遵循不同的**调用约定**。调用约定规定了在汇编层面，调用一个函数时，栈空间的组织方式和参数、返回值如何在调用者和被调用者之间传递。例如在x86\_64 Linux上，编译器遵循一种fastcall的变体。具体来讲，在调用时，它将前六个参数依次存入%rdi, %rsi, %rdx, %rcx, %r8, %r9六个寄存器中，如果参数列表不足6个，那么就有几个用几个；如果参数列表多余6个，多出来的参数依次存入%rsp指向的地址中(即栈中)。依照该定义，使用JIT(运行时代码生成)或内联汇编便可以实现运行时的参数填充，但适配不同平台不同架构的调用约定十分繁琐，因此Cocoa使用了libffi库完成该工作。本质上这和手动实现在原理和效果上是等效的。

基于以上理论，QObject实现了一套名为“信号槽(Signal and Slots)”的对象间通讯机制。

信号槽分为**信号(Signal)**和**槽(Slot)**两部分。信号具有名称，它是一个字符串，该字符串标识了一个**信号接口(Signal Interface)**。信号接口描述了一个信号的参数列表等关键信息，成员函数的调用就是基于信号接口中的信息进行的。槽是一个QObject子类的某个成员函数(具有qobj\_slot标识)，一个槽标识了一个唯一的槽接口，槽接口保存成员函数的this指针和成员函数本身的指针。槽可以连接(connect)到一个信号，此时它与这个信号就有了绑定关联。当某个信号被发射(emit)时，所有连接到该信号的槽上的方法(即成员函数)都会被调用，调用是串行进行的，且QObject不保证调用的顺序。更具体地，QObject::signal注册一个信号，QObject::connect将槽连接到一个信号，QObject::emit发射一个信号。下面的代码展示了如何使用信号槽进行跨对象的方法调用。

class A : public QObject

{

public:

A()

{

QObject::signal<int, const char\*>(this, “@test-signal”);

}

void call()

{

QObject::emit(“@test-signal”, 2233, “Hello, World!”);

}

};

class B : public QObject

{

public:

B()

{

QObject::connect(this, &B::foo, “@test-signal”);

}

private:

void foo(int a, char const \*str) qobj\_slot

{

std::cout << “a = ” << a << std::endl;

std::cout << “str = ” << str << std::endl;

}

};

int main(int argc, char const \*\*argv)

{

A a;

B b;

a.call();

return 0;

}

信号槽的用法基本是基于C++模板衍生而来的。如下给出了信号槽API的函数原型：

/\* Create a signal with its argument list \*/

template<typename...Args>

QObject::signal(QObject \*owner, const std::string& name);

/\* Create and connect a slot to a certain signal \*/

template<IsQObjectChild \_Tp, typename...Args>

QObject::connect(\_Tp \*\_\_this, void(\_Tp::\*fn)(Args...), const std::string& sig);

/\* Emit a signal with arguments \*/

template<typename...Args>

QObject::emit(const std::string& sig, Args... args);

需要特别注意的是，QObject会检查槽函数的参数列表的信号的参数列表，二者必须是严格匹配的，否则会产生运行时异常。当注册了信号的对象被析构时，这个信号以及它所连接的所有槽都会失效；同理，当一个对象被析构时，它所注册的槽都会被删除。

信号槽机制的引入，彻底分离了不同的类之间的依赖(例如上例中A类和B类没有任何语义上互相的依赖)。这使得Cocoa的模块化更加彻底，也让调试、Hack代码更加容易。

* 1. **General Purpose I/O Dispatcher**

Cocoa需要直接与大量系统设备交互，维护了大量的文件描述符，它们都属于阻塞IO。Cocoa不可能一直阻塞于某个文件描述符而不去处理其它的事件，这时就需要一个基于事件的异步IO框架，在一个运行循环中统一所有的IO操作，并在IO操作可用时通知相应的子系统去处理。由此催生了Cocoa的通用IO分发(调度)框架——IODispatcher类。

用户通过特定的接口向IODispatcher注册一个IO调度对象，它可以是文件描述符、定时器或Linux进程信号量。IODispatcher::schedule方法指导它完成一次IO事件分发(该方法会阻塞直到发生一个或多个IO事件且它们被相应的子系统处理完毕)。IODispatcher的原理和QObject中的信号槽类似，不过IODispatcher是基于回调的。也就是说，IODispatcher只允许一个IO事件对应唯一一个回调函数(处理者)，且参数列表不可自定义。IODispatcher固定向回调函数传递一个CallableArgs结构体指针，包含该IO事件的所有信息。

IODispatcher将回调函数称为Callable，一个Callable必须与一个this指针关联，且该this指针指向的对象必须是IOSchedulable的子类。在语言层面，Callable的调用是使用STL的std::function和std::bind实现的。在注册一个基于文件描述符的Callable时，还必须指定一个或多个触发器(Trigger)。触发器向IODispatcher显式指名文件描述符处于何种状态时才认为是事件发生(调用对应的Callable)。目前我们提供Readable，Writable，Hup和Error四种触发器，分别对应文件描述符可读，文件描述符可写，文件描述符被挂起，文件描述符发生错误四种状态。触发器声明在IOTrigger枚举中，上述四种触发器分别对应kTrigger\_Readable, kTrigger\_Writable, kTrigger\_Hup, kTrigger\_Error四个枚举量。指定多个触发器组合时，使用“逻辑或”运算符连接枚举量。

要创建一个文件描述符Callable，调用IODispatcher::newFdCallable即可，依次传入this指针，回调的成员函数地址(成员函数地址的取法和信号槽相同)，文件描述符，触发器作为参数。Callable被调用时，CallableArgs::trigger指出引发事件的触发器，CallableArgs::fd指出引发事件的文件描述符。

要创建一个定时器Callable，调用IODispatcher::newTimerCallable即可，依次传入this指针，回调的成员函数地址，间隔时间作为参数。注意，定时器并不保证到时间了就一定马上调用Callable，我们只保证向Callable传入定时器的超时次数，该值保存在CallableArgs::timer\_timeoutCount中。

要创建一个Linux信号量Callable，调用IODispatcher::newSoftirqCallable即可，依次传入this指针，回调的成员函数地址，信号量编号的集合作为参数。Callable被调用时，CallableArgs::signal\_siginfo保存了当前信号量的详细信息。

整个Cocoa主程序都是建立在IODispatcher框架上的，但是IODispatcher的具体实现并不复杂，它基于Linux内核的epoll机制，实现了高效的异步IO框架。这里不再对具体实现作过多赘述。下面的代码演示了如何使用IODispatcher监听信号量和文件描述符：

class Listener : public IOSchedulable

{

public:

Listener()

{

fExit = false;

IODispatcher::instance()->newSoftirqCallable(this, &Listener::softirq,

std::set<int>{ SIGTERM, SIGINT });

IODispatcher::instance()->newFdCallable(this, &Listener::input,

STDIN\_FILENO, kTrigger\_Readable);

}

void softirq(CallableArgs \*args) io\_callback

{

fExit = true;

}

void input(CallableArgs\* args) io\_callback

{

/\* Read something from standard input... \*/

}

bool shouldExit() const

{

return fExit;

}

private:

bool fExit;

};

int main(int argc, char const \*\*argv)

{

OCInitialize();

GOM->addObject(new IODispatcher());

Listener listener;

while (!listener.shouldExit())

{

IODispatcher::instance()->schedule();

}

OCFinalize();

return 0;

}

和信号槽相同，当一个对象被析构时，所有以它作为this指针的Callable都会从调度对象列表中被删除。

* 1. **Advanced Runtime Exception**

Advanced Runtime Exception(高级运行时异常)是Cocoa核心框架对C++的拓展。大部分语言提供了较为完整的Exception机制，但C++的异常机制仍然是不完善的。ARTE则提供了较为完善的格式化输出和栈帧回溯功能。ARTE的栈帧回溯是提供GCC提供的unwind功能实现的。希望抛出异常的类只需要继承Throwable类，在构造函数中提供该类的字符串符号名(包含命名空间的完整名字)，然后可以使用artfmt和arthrow两个方法来抛出异常。artfmt方法返回一个std::ostream对象，用它来输出格式化的异常信息，紧接着使用arthrow来抛出异常(栈帧回溯在此时完成)。注意，arthrow必须在artfmt之后被调用，若违背此规定，则ARTE机制本身会抛出std::runtime\_error原生C++异常。

另一种抛出异常的方式被称为“断言(Assertion)”。断言提供了一种if-throw的包装形式。其基本语法是\_\_assert(expr, msg)，若布尔表达式expr的结果为false，则抛出一个包含msg信息的ARTException异常。在语义上，断言表示程序过程在某个特定断点必须满足的某种特定状态，若状态不满足，程序就应立即转移至另一种状态来进行例外处理。一般而言，断言可以很方便地处理C语言风格的错误报告，即通过特殊的返回值报告错误。例如open系统调用：

int fd = open(...);

\_\_assert(fd >= 0, “Failed to open file”);

在这个示例中，断言了返回的文件描述符fd必须是一个非负数(POSIX标准规定，POSIX系统调用成功时返回值一律为非负数，而负数返回值则表示相应的错误码)。\_\_assert断言同样属于Throwable类的方法，必须继承该类才可以使用。

另外，注意\_\_assert，assert和static\_assert三个符号的区分。\_\_assert是ARTE机制提供的运行时的基于异常的断言机制；assert是一个宏，由C语言标准提供；static\_assert是C++关键字，是由C++编译器提供的编译时静态断言机制。

1. **Ciallo 2D Rendering Engine**

Ciallo是Cocoa的2D图形渲染引擎，同时负责了图形的显示工作。Ciallo是意大利语Ciao和英语Hello的合成词，二者都是问候语。~~这一词来自某著名会社(Yuzusoft)的Galgame作品《~~*~~Sabbat of the Witch~~*~~》中Inaba Meguru的口癖。~~

Ciallo大体将渲染分为两个部分，DDR和DIR。DDR即Device Depended Rendering(设备相关渲染)，这部分代码实现于具体的渲染硬件有关，Ciallo也实现了一个通用的CPU渲染器作为最坏情况下的备用选项。DIR即Device Independed Rendering(设备无关渲染)，DIR包含了从生成渲染树到布局元素的全部流程，它们的具体实现是与渲染硬件无关的。Ciallo的强大渲染能力来自Skia，Google开发的2D图形库。Skia在Android, Chromium等大型项目中被广泛使用。Ciallo的DDR部分本质上是对Skia进行了平台相关的进一步包装。

**3.1. Device Depended Rendering**

平台(Platform)是Ciallo中重要的概念，一个平台定义了将图像渲染到显示设备上的一系列方法。所有的平台实现都继承自GrBasePlatform抽象类(位于src/ciallo/DDR/GrBasePlatform.cc)，我们要求平台必须实现writableBuffer和submit两个方法，前者返回指向可写缓冲区的指针，高层类向该地址写入渲染结果(如果需要)；如果我们希望刚才写入的图像数据被显示到设备上，则调用submit方法。但要注意的是，如果使用GPU加速的渲染，则这两个方法不是必须的(它们甚至根本不会被调用)。因为在GPU加速的情况下，Vulkan直接将渲染好的纹理提交到设备上显示，没有任何复制的过程。平台也应该知道设备支持的像素格式，并将其转换为Ciallo的内部表示(GrColorFormat枚举)并将其传递给上层类(下文中的Compositor即我们所说的上层类)。

Ciallo目前实现了XCB平台，其它平台将在未来的版本中逐步实现，我们没有计划实现Win32平台。

平台最终为混成器(Compositor)服务，混成器负责将多个图像按照一定次序叠放后，根据它们的透明度值、叠放次序、位置等参数将它们合成为一张图像。图像的混合是一项开销很大的工作，即使在优化好的情况下，它也将占用大量的CPU时间(单线程)。因此混成器可以有不同的实现来支持一定程度上的硬件加速功能。目前Ciallo实现了三种混成器，分别是GrCpuCompositor, GrGpuCompositor和GrOpenCLCompositor(都在src/ciallo/DDR目录下)，它们分别对应CPU渲染，GPU渲染和OpenCL加速合成。

混成器接受来自RenderLayer的输入。RenderLayer是一个图层对象，同时也负责矢量图光栅化的功能。Compositor维护了所有存活的RenderLayer对象，从RenderLayer处取得图层并完成图像混合。Ciallo实现了三种RenderLayer，分别对应上述三种Compositor实现。每一个RenderLayer都有光栅化的能力，它接受一个SkPicture对象(或一张Skia支持的图像文件路径)，将SkPicture中的内容绘制到图层上，在用户调用update方法时图层被提交给混成器。RenderLayer通常并不表示一个单纯意义上的图层，它更多地表示了一个独立的光栅化资源。因此不宜创建过多的RenderLayer，除非某个绘图操作需要高性能、频繁的光栅化且希望被混成器独立合成(如Live2D, 视频等)。

使用CPU渲染时，RenderLayer使用Skia的位图光栅化功能，所有的绘制在CPU上完成，所有的图像缓冲区都在CPU内存上申请。混成器接受CPU内存指针(通过SkImage包装)，在CPU上混成后交给平台显示。在这种情况下，图像要经历三次拷贝：从图层上拷贝出图层快照交给混成器，混成器将图层拷贝到最终图像上，平台从混成器上拷贝图像到设备。

使用GPU渲染时，RenderLayer的光栅化全部在GPU上完成，所有的图像都以纹理的方式存储在GPU上。混成器接受GPU纹理的拷贝(不拷贝到CPU内存)，在GPU上完成混成。混成器不经过平台API，直接由Vulkan将图像提交到设备上显示。在这种情况下，图像没有在CPU内存上经过任何拷贝，但也要求平台必须给出在目标设备上直接创建Vulkan Surface的方法。

使用OpenCL加速合成时，RenderLayer仍然在CPU上完成光栅化，图像在CPU内存和OpenCL设备(多数情况下是GPU，也可能是FPGA, DSP甚至是CPU等OpenCL支持的并行计算设备)内存上都有存储。RenderLayer在CPU内存上完成光栅化，然后将图像拷贝到OpenCL设备上。混成器接受OpenCL设备内存对象，混成工作在OpenCL上以并行方式完成，然后混成器将图像拷贝回本地内存，提交给平台显示。相比CPU渲染，这种渲染方式内存的拷贝次数没有变化，但是拷贝的开销通常更大。CPU渲染时，内存在CPU内存上拷贝，地址空间不变；OpenCL加速合成时，内存需要从CPU拷贝到OpenCL设备，如果是独立GPU，则要经过PCI-E总线传输数据，这比在CPU本地直接传输开销更大。但是通常情况下OpenCL合成图像的速度比CPU快得多，这样平均下来使用OpenCL合成通常都比使用CPU渲染快，但或许仍远不及GPU渲染。

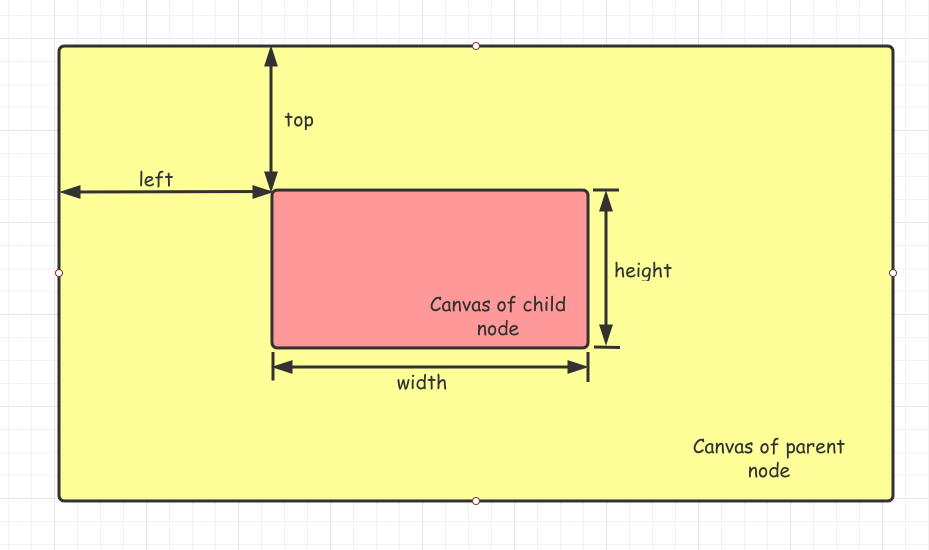
**3.2. Device Independed Rendering**

设备无关渲染(Device Independed Rendering, DIR)是建立在上述DDR驱动的基础上的。DIR部分进一步抽象了接口，整合了零散的接口，并加入了异步渲染机制。

**3.2.1. 渲染树和渲染结点**

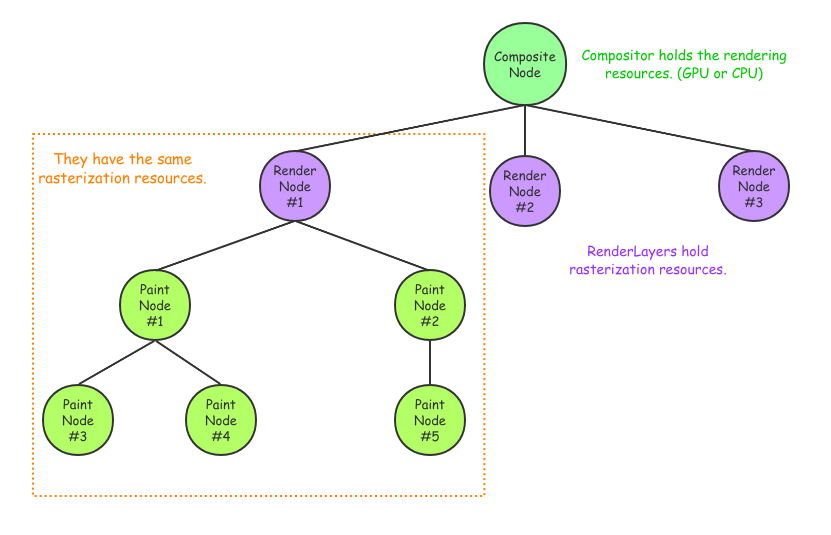
整个DIR模块基本围绕渲染树展开。渲染树(Rendering Tree)描述了图形元素之间的包含关系(但并不代表它表示了DOM元素之间的父子关系)。渲染树上的每一个结点都拥有一个特定尺寸的画布(Canvas)，画布是具有绘图能力的对象，每一个结点都具有在自身画布上绘制自身内容的能力(根结点和第一层结点较为特殊，下文将会介绍)，且记录了它自身在父级结点的画布上所占据的位置和尺寸。我们定义渲染树上任意非根、非第一层结点的结点为“平凡结点(Trivial Node)”，任意一个平凡结点都将把它子结点画布上的图像绘制到自身的画布上，这一过程也依赖于子结点在父结点画布上占据的位置和尺寸，我们称为平凡结点的布局信息(Layout Information)。

如图，top, left, width, height四个参数描述了子结点画布在父结点画布中的的布局信息，这些信息应由子结点拥有且可以动态更改。



每一个平凡结点上的画布都拥有自己的座标空间，并不依赖于它的布局信息(除了座标的最值受到画布尺寸的限制)。在绘制子结点的内容时，并不直接操作像素，而是记录子结点画布上的绘图指令，并对绘图指令进行座标变换、裁剪、滤镜、遮罩等操作。相比复制像素，记录绘图指令是十分高效的，并且所有的图元都由数学方式表示，不存在缩放时的失真问题。

渲染树的根结点和第一层节点较为特殊，根结点称为混成结点，它持有对混成器的引用，第一层结点称为渲染结点，它们持有对对应的RenderLayer的引用。N个渲染层结点将渲染树分成N棵子树，每棵子树在不同的RenderLayer被光栅化。下图展示了完整的渲染树结构。



**3.2.2. 基于任务的线程模型**

Ciallo支持异步渲染，这意味着主线程无需等待渲染完成，它可以在渲染线程渲染的同时去处理其它事情，也可以选择性地阻塞直到渲染完成。

为了实现异步渲染，Ciallo采取了基于任务队列的线程模型。它将进程分为工作线程Worker和主线程，并维护了一个命令队列。Worker一旦启动并完成初始化后，就等待阻塞等待直到命令队列中有新的命令出现。紧接着Worker从队列中取出命令并分配到合适的函数执行，处理完毕后继续等待下一个命令。每个命令有唯一对应的同步对象称为Fence，主线程在入队命令时enqueueCmd函数会返回一个Fence对象指针，调用Fence对象的wait方法可以使当前线程阻塞直到Fence对象对应的命令执行完毕；若在调用wait之前命令已经执行完毕，则直接返回。

Fence对象的生命周期是基于RAII引用计数自动管理的。在入队命令时Fence对象便被创建，当命令被真正提交到命令队列后，由Worker线程持有它的引用。命令执行完毕时，Worker线程不再持有它，若此时其它线程也没有任何对象持有该Fence对象，则它将立刻被释放；若此时仍有其它线程持有该Fence对象，则Fence对象会一直存活直到没有线程持有它。

Fence对象实现了主线程等待某一特定命令执行完毕，而Worker等待新命令入队则采用了eventfd机制实现。eventfd是Linux的众多线程间同步机制之一，它创建一个文件描述符，用户可以对该文件描述符读或写一个64位整形值。当eventfd以信号量模式工作时，内核维护一个64位整形的计数器，对文件描述符write将增加计数器(增量取决于用户传入write的参数)，read将使计数器变为0，当计数器值已经为0时，read系统调用会阻塞直到其它线程调用write使计数器为非0值。因此，某一时刻内核计数器的值就是当前的命令队列长度，当命令队列长度为0时Worker便会阻塞以主动让出CPU时间片。

每个Worker有自己的eventfd，Worker连同它的命令队列、eventfd一起被Thread对象管理。