**draw\_execute 详解**

**函数原型：**

**static void draw\_execute(lv\_draw\_vg\_lite\_unit\_t \* u)**

**结构体的解析：**

|  |
| --- |
| C struct \_lv\_draw\_vg\_lite\_unit\_t {  lv\_draw\_unit\_t base\_unit;  lv\_draw\_task\_t \* task\_act;  struct \_lv\_vg\_lite\_pending\_t \* image\_dsc\_pending;  lv\_cache\_t \* grad\_cache;  struct \_lv\_vg\_lite\_pending\_t \* grad\_pending;  uint16\_t flush\_count;  vg\_lite\_buffer\_t target\_buffer;  vg\_lite\_matrix\_t global\_matrix;  struct \_lv\_vg\_lite\_path\_t \* global\_path;  bool path\_in\_use; }; |

以 LV\_DRAW\_TASK\_TYPE\_IMAGE 为例

|  |
| --- |
| C  case LV\_DRAW\_TASK\_TYPE\_IMAGE:  lv\_draw\_vg\_lite\_img(draw\_unit, t->draw\_dsc, &t->area, false);  break; |

|  |
| --- |
| C struct \_lv\_draw\_unit\_t {  lv\_draw\_unit\_t \* next;  lv\_layer\_t \* target\_layer;  const lv\_area\_t \* clip\_area;  int32\_t (\*dispatch\_cb)(lv\_draw\_unit\_t \* draw\_unit, lv\_layer\_t \* layer);  int32\_t (\*evaluate\_cb)(lv\_draw\_unit\_t \* draw\_unit, lv\_draw\_task\_t \* task);  int32\_t (\*delete\_cb)(lv\_draw\_unit\_t \* draw\_unit); }; |

|  |
| --- |
| C struct \_lv\_draw\_task\_t {  lv\_draw\_task\_t \* next;  lv\_draw\_task\_type\_t type;  lv\_area\_t area;  lv\_area\_t \_real\_area;  lv\_area\_t clip\_area\_original;  lv\_area\_t clip\_area;  volatile int state;   void \* draw\_dsc;  uint8\_t preferred\_draw\_unit\_id;  uint8\_t preference\_score;  }; |



**函数调用流程解析**

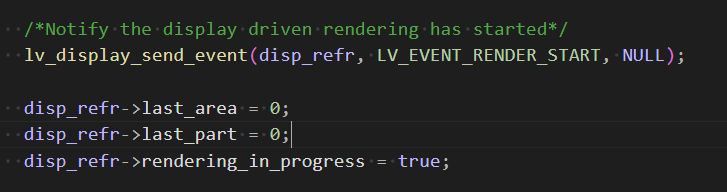
在miwear\_init.c 中，通过lvgl\_disp\_invalid\_cb函数注册lvgl\_disp\_poll\_cb，将lvgl\_disp\_poll\_cb注册到g\_refresh\_poll这个handle的poll\_cb中。

|  |
| --- |
| uv\_poll\_start(&g\_refresh\_poll, UV\_WRITABLE, lvgl\_disp\_poll\_cb); |

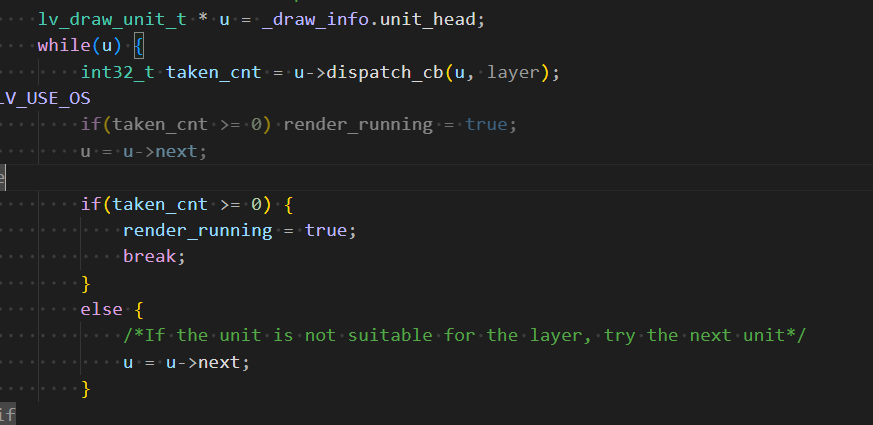
这里需要注意的是

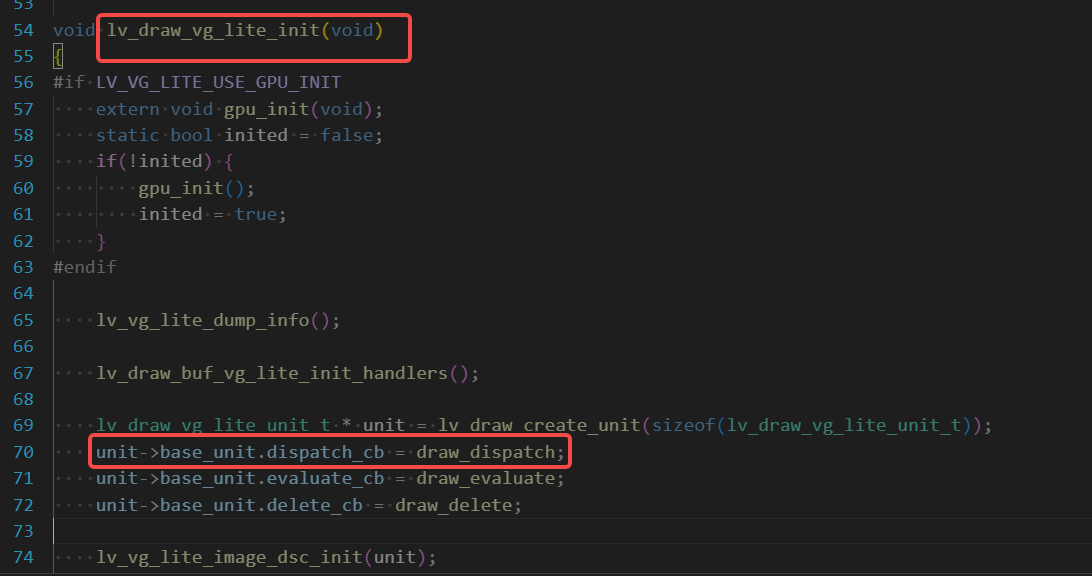
|  |
| --- |
| 使用 `uv\_poll\_t` 实例监听一个文件描述符（socket、文件等）的状态变化。在回调函数中，我们根据参数events判断文件描述符的状态变化类型，并输出相应的响应。在工作之前，先初始化 `uv\_poll\_t` 实例，然后调用 `uv\_poll\_start` 函数启动监听，最后通过 `uv\_run` 函数进入事件循环，等待文件描述符的状态变化事件。 |

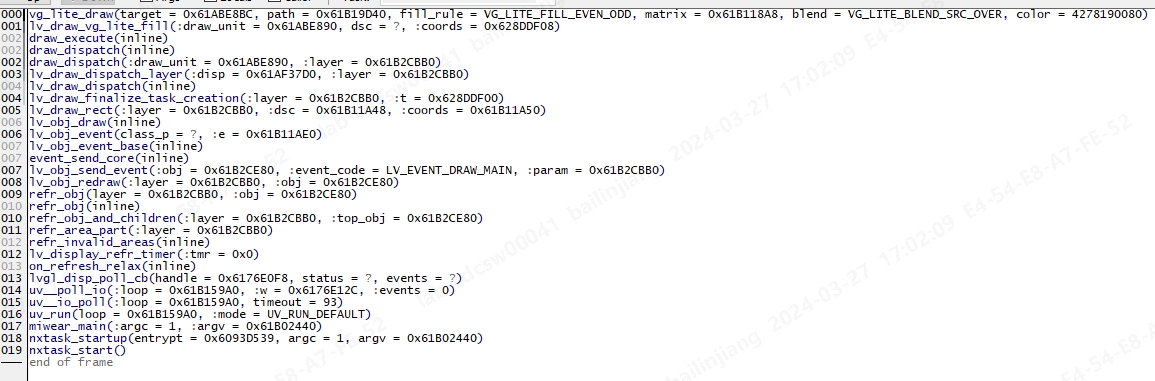
**lv\_global 这个全局变量的作用**



**lv\_draw\_dispatch\_layer函数的过程**



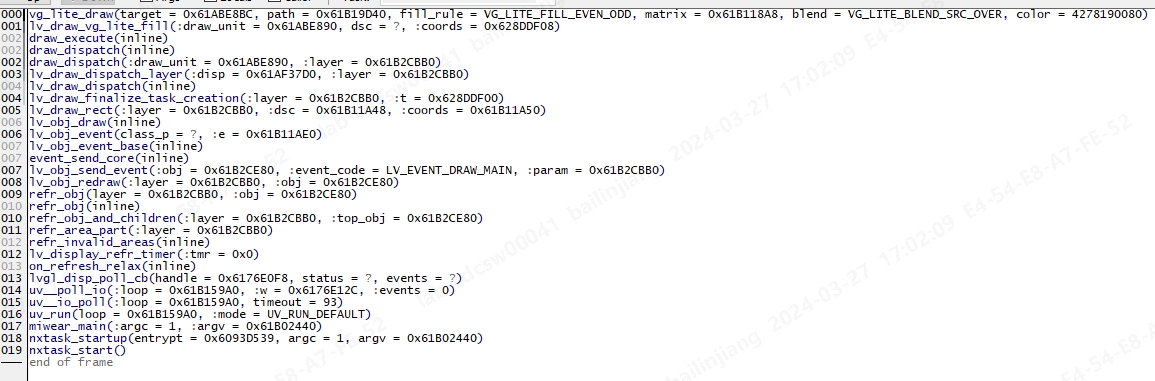






**lv\_draw\_finalize\_task\_creation**

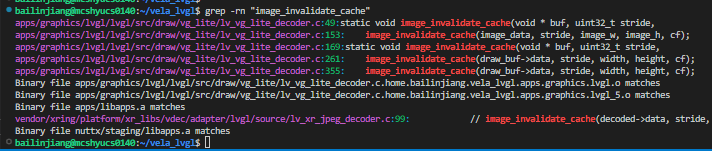
创建绘制任务(需要绘制的source)并尝试调度绘制单元(空闲的target)完成绘制任务，通过dispatch\_cb调用到具体的绘制单元完成绘制，M1项目会调用到vg\_lite这个目录下的draw\_dispatch，找到一个已经入队且独立的的task，进行绘制。然后调用draw\_execute开始执行绘制任务。



draw\_execute函数会从lvgl上层的绘制单元解析出lv\_draw\_unit\_t和lv\_draw\_task\_t，lv\_draw\_unit\_t里面有targetbuffer的信息，lv\_draw\_task\_t有source image的信息



发现的问题



|  |
| --- |
| 在image\_try\_self\_pre\_mul/decoder\_open\_variable/decoder\_open\_file三个函数有刷cache，这个意味着只是在首次读取文件数据的时候，刷过cache，在后续跟硬件GPU交互的时候，特别是中间处理过程，没有刷cache。 |

需要找到GPU往非framebuffer输出的地方，进行刷cache即可。lvgl渲染完成之后，刷cache