lvgl 这篇文章的英文原版我是在 https://lvgl.io/上下载得到的。其实在一开始我本来不想翻译的，但是用到了 lvgl 时，就顺便翻译了，现在 lvgl 是 7.4.0 版本了吧，如果下方翻译的不好，请不要骂我，因为我翻译也是不容易的，我把下班的时间用来翻译了，一周上六天班，上班也是挺忙的，下班时从晚上 8 点开始着手翻译，周日也是花半天时间，花费了两个月时间，所以我希望这份文档对你们有帮助，其实在不影响工作的情况下，本人也是愿意花费这个时间。也做了不少中文化的工作。所以我是打算利用工作之余，边看边译，到读完这篇文档，也就有个中文版了，我不希望别人得到这个文档之后去变卖，其实大家一起学习是最好的。

本人很懒，我不过就是不想花费时间到手机上而已，这文档没有翻译附录，而且译完正文后也没有做过任何检查。所以如果有任何问题，请不要骂我。

Cai Xuefeng 一个爱玩嵌入式的小锋

Cai Xuefeng

第一章 LVGL 对象

1.1 对象的简介

在 LVGL 中，用户界面的基本构建块是对象，也称为小部件。例如，按钮，标签，图像，列表，图表或文本区域。在此处检查所有对象类型。

1.2 对象的属性

1.2.1 基本属性

在 LVGL 中所有对象类型都共享一些基本属性：

(1) 尺寸
(2) 父母
(3) 拖动启用
(4) 单击启用等
(5) 位置

您可以使用 lv_obj_set_...和 lv_obj_get_...功能设置/获取这些属性。例如：

```
/*设置基础对象属性*/
lv_obj_set_size(btn1, 100, 50);  /*按键大小*/
lv_obj_set_pos(btn1, 20,30);      /*按键位置*/
```


1.2.2 具体属性

对象类型也具有特殊的属性。例如，滑块具有

(1) 当前值
(2) 自定义样式
(3) 最小值、最高值

对于这些属性，每种对象类型都有唯一的 API 函数。例如一个滑块：

```
/*设置滑块的特殊属性*/
lv_slider_set_range(slider1, 0, 100);             /* 设置滑块的最小、最大值 */
lv_slider_set_value(slider1, 40, LV_ANIM_ON);    /* 设置滑块当下值的位置 */
lv_slider_set_action(slider1, my_action);        /* 设置滑块的回调函数 */
```

对象类型的 API 在其文档中进行了描述，但您也可以检查相应的头文件（例如 *lv_objx / lv_slider.h*）

1.3 工作机制

1.3.1 父类-子类的结构

父对象可以视为其子对象的容器。每个对象只有一个父对象（屏幕除外），但是一个父对象可以有无限多个子

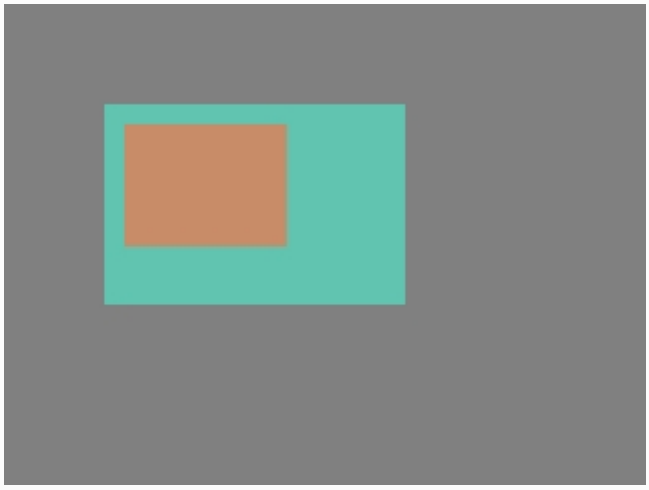
对象。父对象的类型没有限制，但是有典型的父对象（例如按钮）和典型的子对象（例如标签）。

1.3.2 一起移动

如果更改了父类对象的位置，则子类对象将与父类对象一起移动。因此，所有位置都相对于父类。（0; 0）坐标表示对象将独立于父对象的位置保留在父对象的左上角。

```
lv_obj_t * par = lv_obj_create(lv_scr_act(), NULL); /*创建一个父类对象在屏幕中*/
lv_obj_set_size(par, 100, 80);                      /*设置父类对象的位置*/

lv_obj_t * obj1 = lv_obj_create(par, NULL);     /*创建一个子类对象与父类对象中*/
lv_obj_set_pos(obj1, 10, 10);                   /*设置子类对象的位置*/
```

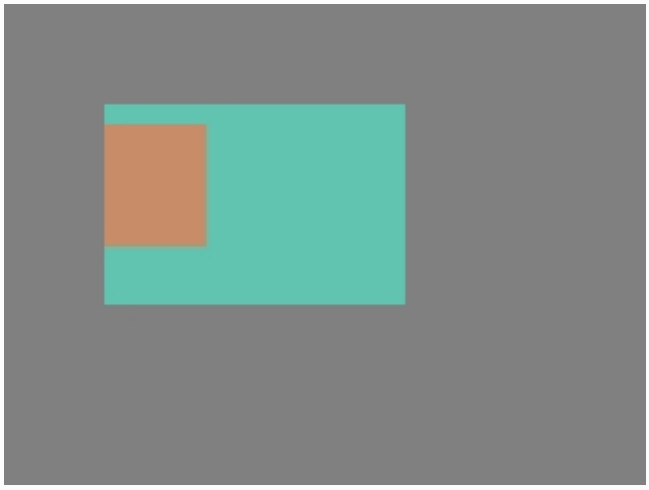

修改父类对象的位置:



```
lv_obj_set_pos(par, 50, 50);      /*移动父类对象，子类对象跟着移动.*/
```

1.3.3 仅在父类对象上可见

如果子类对象部分或全部不在其父级之外，则看不见外面的部分。



```
lv_obj_set_x(obj1, -30);/*Move the child a little bit of the parent*/
```

1.3.4 创建-删除对象

在 LVGL 中，可以在运行时动态创建和删除对象。这意味着仅当前创建的对象消耗 RAM。例如，如果需要图表，则可以在需要时创建它，并在不可见或不必要时将其删除。

每个对象类型都有自己的带有统一原型的**创建**功能。它需要两个参数：

(1) 指向父对象的指针。要创建屏幕，请以 NULL 作为父级。

(2) （可选）用于复制具有相同类型的对象的指针。该复制对象可以为 NULL，以避免复制操作。

所有对象均以 C 语言中的 lv_obj_t 指针作为句柄进行引用。以后可以使用该指针设置或获取对象的属性。

创建函数如下所示：


```
lv_obj_t * lv_ <type>_create(lv_obj_t * parent, lv_obj_t * copy);
```

所有对象类型都有一个通用的**删除**功能。它删除对象及其所有子对象。

```
void lv_obj_del(lv_obj_t * obj);
```

lv_obj_del 将立即删除该对象。如果由于任何原因无法立即删除对象，可以使用 lv_obj_del_async(obj)。这很有用，例如，如果您想在子 LV_EVENT_DELETE 信号中删除对象的父对象。

您可以使用以下方法删除对象的所有子对象（但不能删除对象本身）lv_obj_clean：

```
void lv_obj_clean(lv_obj_t * obj);
```

1.4 屏幕

1.4.1 创建屏幕

屏幕是没有父对象的特殊对象。因此，可以像这样创建它们：

```
lv_obj_t * scr1 = lv_obj_create(NULL, NULL);
```

可以使用任何对象类型创建屏幕。例如，创建墙纸的基础对象或图像。

1.4.2 获取活动屏幕

每个显示器上始终有一个活动屏幕。默认情况下，该库为每个显示创建并加载一个“基础对象”作为屏幕。

要获取当前活动的屏幕，请使用函数 lv_scr_act()来获取活动屏幕。

1.4.3 加载屏幕

要加载新屏幕，如以下函数原型
lv_scr_load(scr1)。

1.4.4 用动画加载屏幕

可以使用加载新屏幕的动画。

```
lv_scr_load_anim(scr,  transition_type,  time,  delay,  auto_del)
```

transition_type 存在以下转换类型，如表 1.4.4.1 所示：

参数	描述
LV_SCR_LOAD_ANIM_NONE	延时毫秒后立即切换
LV_SCR_LOAD_ANIM_OVER_LEFT/RIGHT/TOP/BOTTOM	将新屏幕移到给定方向上
LV_SCR_LOAD_ANIM_MOVE_LEFT/RIGHT/TOP/BOTTOM	将旧屏幕和新屏幕都移至给定方向
LV_SCR_LOAD_ANIM_FADE_ON	使新屏幕淡入旧屏幕

表 1.4.4.1 transition_type 存在以下转换类型描述

设置 auto_del 为 true 会在动画结束时自动删除旧屏幕。lv_scr_act()动画开始播放后，新屏幕将变为活动状态。

1.4.5 处理多个显示

屏幕在当前选择的默认显示上创建。在默认显示是最后的注册显示屏 lv_disp_drv_register，也可以使用显式地选择一个新的默认显示 lv_disp_set_default(disp)。

lv_scr_act()，lv_scr_load()并 lv_scr_load_anim()在默认屏幕上进行操作。

1.5 小部件

小部件可以包含多个部分。例如，按钮仅具有主要部分，而滑块则由背景，指示器和旋钮组成。

各部分的名称构造如下。例如。通常在将样式添加到对象时使用小部件。使用小部件可以将不同的样式分配给对象的不同小部件。

LV_ + <TYPE> _PART_ <NAME>LV_BTN_PART_MAINLV_SLIDER_PART_KNOB

1.6 状态

该对象可以处于以下状态的组合：

LV_STATE_DEFAULT 正常，已发布
LV_STATE_CHECKED 切换或选中
LV_STATE_FOCUSED 通过键盘或编码器聚焦或通过触摸板/鼠标单击
LV_STATE_EDITED 由编码器编辑
LV_STATE_HOVERED 鼠标悬停（现在不支持）
LV_STATE_PRESSED 已按下

LV_STATE_DISABLED 禁用或不活动

当用户按下，释放，聚焦等对象时，状态通常由库自动更改。但是，状态也可以手动更改。要完全覆盖当前状态，请使用。要设置或清除给定状态（但保持其他状态不变），在两种情况下都可以使用 ORed 状态值。例如。

lv_obj_set_state(obj, part, LV_STATE...)

lv_obj_add/clear_state(obj, part, LV_STATE_...)

lv_obj_set_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)

第二章 LVGL 的层数

2.1 创建顺序

默认情况下，LVGL 在背景上绘制旧对象，在前景上绘制新对象。

例如，假设我们向一个名为 button1 的父对象添加了一个按钮，然后向另一个名为 button2 的按钮添加了一个按钮。然后 button1（及其子对象）将在背景中，并且可以被 button2 及其子对象覆盖。


```c
/*创建一个屏幕*/
lv_obj_t * scr = lv_obj_create(NULL, NULL);
lv_scr_load(scr);           /*加载屏幕*/

/*创建两个按钮*/
lv_obj_t * btn1 = lv_btn_create(scr, NULL);         /*创建一个按钮 1 在屏幕中*/
lv_btn_set_fit(btn1, true, true);                   /*使能自动设置大小根据内容*/
lv_obj_set_pos(btn1, 60, 40);                   /*设置该按钮的位置 */

lv_obj_t * btn2 = lv_btn_create(scr, btn1);         /*复制第一个按钮*/
lv_obj_set_pos(btn2, 180, 80);                    /*设置该按钮的位置*/

/*添加标签与按钮中*/
lv_obj_t * label1 = lv_label_create(btn1, NULL);    /*创建一个标签于按钮 1*/
lv_label_set_text(label1, "Button 1");              /*设置标签内容*/

lv_obj_t * label2 = lv_label_create(btn2, NULL);    /*创建标签于按钮 2*/
lv_label_set_text(label2, "Button 2");              /*设置标签的内容*/

/*删除第一个按钮标签*/
lv_obj_del(label2);
```


2.2 成为前台

有几种方法可以将对象置于前台：

(1) 使用 lv_obj_set_top (obj,true)。如果 obj 或它的任何子对象被点击，那么 LVGL 将自动将该对象带到前台。它的工作原理类似于 PC 上的典型 GUI。当点击背景中的一个窗口时，它会自动来到前台

(2) 使用 lv_obj_move_foreground(obj)显式地告诉库将一个对象放到前台。类似地，使用 lv_obj_move_background(obj)移动到背景。

(3) 当 lv_obj_set_parent(obj, new_parent)被使用时，obj 将在 new_parent 的前台

2.3 顶层和系统层

LVGL 使用两个特殊的层，分别名为 layer_top 和 layer_sys。两者在显示器的所有屏幕上都是可见的和常见的。但是，它们不能在多个物理显示器之间共享。layer_top 总是位于默认屏幕的顶部(lv_scr_act())，而 layer_sys 位于 layer_top 的顶部

用户可以使用 layer_top 创建一些随处可见的内容。例如，一个菜单栏，一个弹出窗口等等。如果单击属性是启用的，那么 layer_top 将吸收所有用户单击，并作为一个模态

lv_obj_set_click (lv_layer_top(),true);

layer_sys 也用于类似的目的在 LVGL 上。例如，它将鼠标光标放在那里，以确保它总是可见的

第三章 LVGL 的事件

3.1 Even(事件)的简介

当发生用户可能感兴趣的事情时，例如在对象中，事件将在 LVGL 中触发。

（1）被点击

（2）被拖

（3）其值已更改，等等。

用户可以将回调函数分配给对象以查看这些事件。实际上，它看起来像这样:

```
lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
lv_obj_set_event_cb(btn, my_event_cb);   /*指定一个事件回调函数*/

...

static void my_event_cb(lv_obj_t * obj, lv_event_t event)
{
    switch(event) {
        case LV_EVENT_PRESSED:  /* 事件被按下 */
            printf("Pressed\n");
            break;

        case LV_EVENT_SHORT_CLICKED:/* 短点击 */
            printf("Short clicked\n");
            break;

        case LV_EVENT_CLICKED:  /* 点击 */
            printf("Clicked\n");
            break;

        case LV_EVENT_LONG_PRESSED: /* 长按 */
            printf("Long press\n");
            break;

        case LV_EVENT_LONG_PRESSED_REPEAT:/* 重复长按 */
            printf("Long press repeat\n");
            break;

        case LV_EVENT_RELEASED:  /* 释放 */
            printf("Released\n");
            break;
    }

      /*Etc.*/
}
```


注意：更多对象可以使用同一*事件回调*。

3.2 事件类型

3.2.1 一般事件

所有对象（例如 Buttons / Labels / Sliders 等）都将接收这些通用事件，而不管它们的类型如何。

3.2.2 相关的输入设备

当用户按下/释放对象时发送这些消息。它们不仅用于指针，还可以用于键盘，编码器和按钮输入设备。
LV_EVENT_PRESSED 该对象已被按下
LV_EVENT_PRESSING 对象被按下（在按下时连续发送）
LV_EVENT_PRESS_LOST 仍在按下输入设备，但不再在对象上
LV_EVENT_SHORT_CLICKED 在 LV_INDEV_LONG_PRESS_TIME 时间之前发布。如果拖动则不调用。
LV_EVENT_LONG_PRESSED 紧迫 LV_INDEV_LONG_PRESS_TIME 时间。如果拖动则不调用。
LV_EVENT_LONG_PRESSED_REPEAT LV_INDEV_LONG_PRESS_TIME 每 LV_INDEV_LONG_PRESS_REP_TIME 毫秒调用一次。如果拖动则不调用。
LV_EVENT_CLICKED 如果未拖动则调用释放（无论长按）
LV_EVENT_RELEASED 在每种情况下都被调用，即使对象已被拖动也被释放。如果在按下并从对象外部释放时从对象上滑出，则不会调用。在这种情况下，LV_EVENT_PRESS_LOST 发送。

3.2.3 相关指针

这些事件仅由类似指针的输入设备（例如鼠标或触摸板）发送
LV_EVENT_DRAG_BEGIN 开始拖动对象
LV_EVENT_DRAG_END 拖动完成（包括拖动）
LV_EVENT_DRAG_THROW_BEGIN 开始拖动拖动（拖动“动量”后释放）

3.2.4 键盘和编码器

这些事件由键盘和编码器输入设备发送。在[overview / indev]（输入设备）。
LV_EVENT_KEY 将密钥发送到对象。通常在按下它或在长按之后重复时。可以通过以下方式检索密钥
uint32_t * key = lv_event_get_data()
LV_EVENT_FOCUSED 对象集中在其组中
LV_EVENT_DEFOCUSED 对象在其组中散焦

3.2.5 一般事件

库发送的其他一般事件。
LV_EVENT_DELETE 正在删除对象。释放相关的用户分配数据。

3.2.6 特别事件

这些事件特定于特定的对象类型。

LV_EVENT_VALUE_CHANGED	对象值已更改（例如，对于 Slider）
LV_EVENT_INSERT	某物插入到对象中。（通常到文本区域）
LV_EVENT_APPLY	单击 “确定”，“应用”或类似的特定按钮。（通常来自键盘对象）
LV_EVENT_CANCEL	单击 “关闭”，“取消”或类似的特定按钮。（通常来自键盘对象）
LV_EVENT_REFRESH	查询以刷新对象。永远不会由库发送，但可以由用户发送。

3.3 自定义的数据

一些事件可能包含自定义数据。例如，LV_EVENT_VALUE_CHANGED 在某些情况下会告诉新值。有关更多信息。要在事件回调中获取自定义数据，请使用 lv_event_get_data()。

自定义数据的类型取决于发送对象，但如果是

然后单号是或 uint32_t *int32_t *

然后输入文字或 char * const char *

3.4 手动发送事件

3.4.1 任意事件

要将事件手动发送到对象，请使用。lv_event_send(obj, LV_EVENT_..., &custom_data)

例如，它可以通过模拟按钮按下来手动关闭消息框（尽管有更简单的方法）：

```
/*Simulate the press of the first button (indexes start from zero)*/
uint32_t btn_id = 0;
lv_event_send(mbox, LV_EVENT_VALUE_CHANGED, &btn_id);
```


3.4.2 刷新事件

LV_EVENT_REFRESH 这是一个特殊事件，因为它旨在供用户用来通知对象刷新自身。一些例子：

（1）通知标签根据一个或多个变量（例如当前时间）刷新其文本

（2）语言更改时刷新标签

（3）如果满足某些条件，则启用按钮（例如，输入正确的 PIN）

（4）如果超出限制，则将样式添加到对象/从对象删除样式，等等。

（5）处理类似情况的最简单方法是利用以下功能。

处理类似情况的最简单方法是使用以下函数。lv_event_send_refresh(obj)只是 lv_event_send(obj, LV_EVENT_REFRESH, NULL)。因此，它只向对象发送一个 LV_EVENT_REFRESH.

lv_event_send_refresh_recursive(obj)向一个对象及其所有子对象发送 LV_EVENT_REFRESH 事件。如果将 NULL 作为参数传递，则所有显示的所有对象都将刷新。

第四章 LVGL 的样式

4.1 样式简介

*样式*用于设置对象的外观。lvgl 中的样式受 CSS 启发很大。简而言之，概念如下：

样式是一个 `lv_style_t` 可以保存属性的变量，例如边框宽度，文本颜色等。与 `class` CSS 类似。

并非必须指定所有属性。未指定的属性将使用默认值。
可以将样式分配给对象以更改其外观。
样式可以被任意数量的对象使用。
样式可以级联，这意味着可以将多个样式分配给一个对象，并且每种样式可以具有不同的属性。例如，

`style_btn` 可能会导致默认的灰色按钮，并且 `style_btn_red` 只能添加一个 `background-color=red` 以覆盖背景色。

以后添加的样式具有更高的优先级。这意味着，如果以两种样式指定属性，则将使用后面添加的样式。
如果未在对象中指定，则某些属性（例如，文本颜色）可以从父级继承。
对象可以具有比“普通”样式更高优先级的局部样式。

与 CSS（伪类描述不同的状态，例如 `:hover`）不同，在 lvgl 中，将属性分配给给定的状态。（即“类”与状态无关，但是每个属性都有一个状态）
当对象更改状态时可以应用过渡。

4.2 样式的状态 Cai Xuefeng

对象可以处于以下状态：
LV_STATE_DEFAULT（0x00）：正常，已释放
LV_STATE_CHECKED（0x01）：切换或选中
LV_STATE_FOCUSED（0x02）：通过键盘或编码器聚焦或通过触摸板/鼠标单击
LV_STATE_EDITED （0x04）：由编码器编辑
LV_STATE_HOVERED（0x08）：鼠标悬停（现在不支持）
LV_STATE_PRESSED（0x10）：已按下
LV_STATE_DISABLED（0x20）：禁用或 void

例如，状态的组合也是可能的。`LV_STATE_FOCUSED | LV_STATE_PRESSED`

可以在每种状态和状态组合中定义样式属性。例如，为默认和按下状态设置不同的背景颜色。如果未在状态中定义属性，则将使用最佳匹配状态的属性。通常，它表示带有 `LV_STATE_DEFAULT` 状态的属性。如果即使对于默认状态也未设置该属性，则将使用默认值。（请参阅稍后）

但是“最佳匹配状态的属性”到底意味着什么？国家的优先级由其值显示（请参见上面的列表）。值越高，优先级越高。为了确定要使用哪个州的属性，我们举一个例子。让我们来看看背景色是这样定义的：

- `LV_STATE_DEFAULT`：白色
- `LV_STATE_PRESSED`：灰色
- `LV_STATE_FOCUSED`：红色

1. 默认情况下，对象处于默认状态，因此很简单：该属性在对象的当前状态中完美定义为白色
2. 按下对象时，有 2 个相关属性：默认为白色（默认与每个状态有关）和按下为灰色。按下状态的优先级为 0x10，高于默认状态的 0x00 优先级，因此将使用灰色。
3. 当物体聚焦时，会发生与按下状态相同的事情，并且将使用红色。(焦点状态的优先级高于默认状态)。
4. 聚焦并按下对象时，灰色和红色都可以使用，但是按下状态的优先级高于聚焦，因此将使用灰色。
5. 可以为设置例如玫瑰色。在这种情况下，此组合状态的优先级为 0x02 + 0x10 = 0x12，该优先级高于按下状态的优先级，因此将使用玫瑰色。LV_STATE_PRESSED | LV_STATE_FOCUSED
6. 选中对象后，没有属性可以设置此状态的背景色。因此，在缺少更好的选择的情况下，对象在默认状态的属性中仍为白色。

一些实用说明：

- 如果要为所有状态设置属性（例如红色背景色），只需将其设置为默认状态即可。如果对象找不到其当前状态的属性，它将回退到默认状态的属性。
- 使用 ORed 状态来描述复杂情况的属性。(例如，按+选中+集中)
- 对不同的状态使用不同的样式元素可能是一个好主意。例如，很难找到释放，按下，选中+按下，聚焦，聚焦+按下，聚焦+按下+选中等状态的背景颜色。相反，例如，将背景色用于按下和选中状态，并使用不同的边框颜色指示聚焦状态。
-

4.3 级联样式

不需要将所有属性设置为一种样式。可以向对象添加更多样式，然后让后来添加的样式修改或扩展其他样式的属性。例如，创建常规的灰色按钮样式，并为仅设置新的背景色的红色按钮创建新的样式。

当在 CSS 中列出所有使用的类时，这是相同的概念。`<div class=".btn .btn-red">`

较晚添加的样式优先于较早的样式。因此，在上面的灰色/红色按钮示例中，应首先添加常规按钮样式，然后再添加红色样式。但是，仍然考虑来自国家的优先级。因此，让我们研究以下情况：

- 基本的按钮样式定义了默认状态为深灰色和按下状态为浅灰色
- 红色按钮样式仅在默认状态下将背景色定义为红色

在这种情况下，释放按钮时（它处于默认状态）它将是红色的，因为在最后添加的样式（红色样式）中找到了完美的匹配。按下按钮时，浅灰色是更好的搭配，因为它完美地描述了当前状态，因此按钮将是浅灰色的。

4.4 继承

某些属性（通常与文本相关）可以从父对象的样式继承。仅当未以对象的样式（即使在默认状态下）设置给定属性时，才应用继承。在这种情况下，如果该属性是可继承的，则该属性的值也将在父级中搜索，直到一部分可以告诉该属性的值为止。父母会用自己的状态来说明价值。按下按钮后，文字颜色就从这里来，将使用按下的文字颜色。

4.5 小部件

对象可以具有可以具有自己样式的*小部件*。例如，页面包含四个部分：

- **背景**
- **可卷动**
- **滚动条**
- **边缘闪光灯**

有三种类型的对象部分的**主体**，**虚拟**和**现实**。

主要部分通常是对象的背景和最大部分。某些对象只有一个主要部分。例如，一个按钮只有一个背景。

虚拟小部件是实时绘制到主体小部件的其他小部件。它们后面没有“真实”对象。例如，页面的滚动条不是真实的对象，仅在绘制页面背景时才绘制。虚拟小部件始终具有与主要小部件相同的状态。如果可以继承该属性，则在转到父级之前还将考虑主体部分。

真实小部件是由主对象创建和管理的真实对象。例如，页面的可滚动部分是真实对象。实际小部件的状态可能与主要小部件的状态不同。

要查看对象具有哪些部分，请访问其文档页面。

4.6 初始化样式并设置/获取属性

样式存储在 `lv_style_t` 变量中。样式变量应为 `static`，全局或动态分配。换句话说，它们不能是函数中的局部变量，在函数存在时销毁。在使用样式之前，应使用进行初始化 `lv_style_init(&my_style)`。初始化后，可以设置样式属性。属性集函数如下所示： 例如，上面提到的示例如下所示：

```
lv_style_set_<property_name>(&style, <state>, <value>);
```



```
static lv_style_t style1;
lv_style_set_bg_color(&style1, LV_STATE_DEFAULT, LV_COLOR_WHITE);
lv_style_set_bg_color(&style1, LV_STATE_PRESSED, LV_COLOR_GRAY);
lv_style_set_bg_color(&style1, LV_STATE_FOCUSED, LV_COLOR_RED);
lv_style_set_bg_color(&style1, LV_STATE_FOCUSED | LV_STATE_PRESSED, lv_color_hex(0xf88));
```

可以使用复制样式。复制后，仍然可以自由添加属性。lv_style_copy(&style_destination, &style_source)

要删除属性，请使用：

```
lv_style_remove_prop(&style, LV_STYLE_BG_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS));
```

要从给定状态的样式中获取值，可以使用以下原型的功能：。将选择最匹配的属性，并返回其优先级。如果找不到该属性，则将返回。_lv_style_get_color/int/opa/ptr(&style, <prop>, <result buf>);-1

函数（`...color/int/opa/ptr`）的形式应根据的类型使用`<prop>`。

例如：

```
lv_color_t color;
int16_t res;
res = _lv_style_get_color(&style1,  LV_STYLE_BG_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS),
&color);
if(res >= 0) {
  //the bg_color is loaded into `color`
}
```

要重置样式（释放所有数据），请使用

```
lv_style_reset(&style);
```

4.7 管理样式列表

样式本身没有那么有用。应该将其分配给对象才能生效。对象的每个部分都存储一个*样式列表*，该列表是已分配样式的列表。

要将样式添加到对象，请使用 例如：lv_obj_add_style(obj, <part>, &style)

```
lv_obj_add_style(btn, LV_BTN_PART_MAIN, &btn);      /*Default button style*/
lv_obj_add_style(btn, LV_BTN_PART_MAIN, &btn_red);  /*Overwrite only a some colors to red*/
```

可以使用以下方式重置对象样式列表 lv_obj_reset_style_list(obj, <part>)

如果已经分配给对象的样式发生更改（即，其属性之一设置为新值），则应通过以下方式通知使用该样式的对象： lv_obj_refresh_style(obj)

要获得属性的最终值，包括级联，继承，局部样式和过渡（请参见下文），可以使用以下类似的 get 函数： 。这些函数使用对象的当前状态，如果没有更好的候选者，则返回默认值。例如：

lv_obj_get_style_<property_name>(obj, <part>)

```
lv_color_t color = lv_obj_get_style_bg_color(btn, LV_BTN_PART_MAIN);
```


4.8 本地风格

在对象的样式列表中，也可以存储所谓的局部属性。与 CSS 的概念相同。局部样式与普通样式相同，但是它仅属于给定的对象，不能与其他对象共享。要设置本地属性，请使用　例如以下功能：
lv_obj_set_style_local_<property_name>(obj, <part>, <state>, <value>);

```
lv_obj_set_style_local_bg_color(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, LV_COLOR_RED);
```

4.9 转换

默认情况下，当对象更改状态（例如，按下状态）时，会立即设置新状态下的新属性。但是，通过过渡，可以在状态改变时播放动画。例如，在按下按钮后，可以在 300 毫秒内将其背景色设置为所按下的颜色。

过渡的参数存储在样式中。可以设置

- 过渡时间
- 开始过渡之前的延迟
- 动画 path（也称为计时功能）
- 要设置动画的属性

可以为每个状态定义过渡属性。例如，将 500 ms 过渡时间设置为默认状态将意味着当对象进入默认状态时，将应用 500 ms 过渡时间。在按下状态下设置 100 ms 过渡时间将意味着在进入按下状态时 100 ms 过渡时间。因

此，此示例配置将导致快速进入印刷机状态而缓慢回到默认状态。

4.10 属性

样式中可以使用以下属性。

4.10.1 混合属性

- radius（lv_style_int_t）：设置背景的半径。0：无半径，LV_RADIUS_CIRCLE：最大半径。默认值：0。
- clip_corner（bool）：：true 可以将溢出的内容剪切到圆角（半径> 0）上。默认值：false。
- size（lv_style_int_t）：小部件内部元素的大小。是否使用此属性，请参见窗口小部件的文档。默认值：。LV_DPI / 20
- transform_width （lv_style_int_t）：使用此值使对象在两侧更宽。默认值：0。
- transform_height （lv_style_int_t）使用此值使对象在两侧都较高。默认值：0。
- transform_angle （lv_style_int_t）：旋转类似图像的对象。它的 uinit 为 0.1 度，对于 45 度使用 450。默认值：0。
- transform_zoom （lv_style_int_t）缩放类似图像的对象。LV_IMG_ZOOM_NONE 正常大小为 256（或），一半为 128，一半为 512，等等。默认值：LV_IMG_ZOOM_NONE。
- opa_scale（lv_style_int_t）：继承。按此比例缩小对象的所有不透明度值。由于继承了子对象，因此也会受到影响。默认值：LV_OPA_COVER。

4.10.2 填充和边距属性

*填充*可在边缘的内侧设置空间。意思是“我不要我的孩子们离我的身体太近，所以要保留这个空间”。*填充内部*设置了孩子之间的“差距”。 *边距*在边缘的外侧设置空间。意思是“我想要我周围的空间”。

如果启用了布局或 自动调整，则这些属性通常由 Container 对象使用。但是，其他小部件也使用它们来设置间距。有关详细信息，请参见小部件的文档。

- pad_top（lv_style_int_t）：在顶部设置填充。默认值：0。
- pad_bottom（lv_style_int_t）：在底部设置填充。默认值：0。
- pad_left（lv_style_int_t）：在左侧设置填充。默认值：0。
- pad_right（lv_style_int_t）：在右侧设置填充。默认值：0。
- pad_inner（lv_style_int_t）：设置子对象之间对象内部的填充。默认值：0。
- margin_top（lv_style_int_t）：在顶部设置边距。默认值：0。
- margin_bottom（lv_style_int_t）：在底部设置边距。默认值：0。
- margin_left（lv_style_int_t）：在左边设置边距。默认值：0。
- margin_right（lv_style_int_t）：在右边设置边距。默认值：0。

4.10.3 背景属性

背景是一个可以具有渐变和 `radius` 舍入的简单矩形。

- bg_color（lv_color_t）指定背景的颜色。默认值：LV_COLOR_WHITE。
- bg_opa（lv_opa_t）指定背景的不透明度。默认值：LV_OPA_TRANSP。
- bg_grad_color（lv_color_t）指定背景渐变的颜色。右侧或底部的颜色是。默认值：。

bg_grad_dir != LV_GRAD_DIR_NONELV_COLOR_WHITE

- bg_main_stop（uint8_t）：指定渐变应从何处开始。0：最左/最上位置，255：最右/最下位置。默认值：0。
- bg_grad_stop（uint8_t）：指定渐变应在何处停止。0：最左/最上位置，255：最右/最下位置。预设值：255。
- bg_grad_dir（lv_grad_dir_t）指定渐变的方向。可以 LV_GRAD_DIR_NONE/HOR/VER。默认值：LV_GRAD_DIR_NONE。
- bg_blend_mode（lv_blend_mode_t）：将混合模式设置为背景。可以 LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE)。默认值：LV_BLEND_MODE_NORMAL。


```c
#include "../../lv_examples.h"

/**
 * Using the background style properties
 */
void lv_ex_style_1(void)
{
    static lv_style_t style;
    lv_style_init(&style);
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);

    /*Make a gradient*/
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_bg_grad_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_bg_grad_dir(&style, LV_STATE_DEFAULT, LV_GRAD_DIR_VER);

    /*Shift the gradient to the bottom*/
    lv_style_set_bg_main_stop(&style, LV_STATE_DEFAULT, 128);
    lv_style_set_bg_grad_stop(&style, LV_STATE_DEFAULT, 192);


    /*Create an object with the new style*/
```

```
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.4 边框属性

边框绘制在*背景*上方。它具有 `radius` 舍入。

- border_color（lv_color_t）指定边框的颜色。默认值：LV_COLOR_BLACK。
- border_opa（lv_opa_t）指定边框的不透明度。默认值：LV_OPA_COVER。
- border_width（lv_style_int_t）：设置边框的宽度。默认值：0。
- border_side（lv_border_side_t）指定要绘制边框的哪一侧。可以LV_BORDER_SIDE_NONE/LEFT/RIGHT/TOP/BOTTOM/FULL。ORed 值也是可能的。默认值：LV_BORDER_SIDE_FULL。
- border_post（bool）：如果 true 在绘制完所有子级之后绘制边框。默认值：false。
- border_blend_mode（lv_blend_mode_t）：设置边框的混合模式。可以LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE)。默认值：LV_BLEND_MODE_NORMAL。


```
#include "../../lv_examples.h"

/**
 * Using the border style properties
 */
void lv_ex_style_2(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add border to the bottom+right*/
```

```
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_border_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);
    lv_style_set_border_side(&style, LV_STATE_DEFAULT, LV_BORDER_SIDE_BOTTOM | LV_BORDER_SIDE_RIGHT);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.5 轮廓属性

轮廓类似于*边框*，但绘制在对象外部。

- outline_color（lv_color_t）指定轮廓的颜色。默认值：LV_COLOR_BLACK。
- outline_opa（lv_opa_t）指定轮廓的不透明度。默认值：LV_OPA_COVER。
- outline_width（lv_style_int_t）：设置轮廓的宽度。默认值：0。
- outline_pad（lv_style_int_t）设置对象和轮廓之间的空间。默认值：0。
- outline_blend_mode（lv_blend_mode_t）：设置轮廓的混合模式。可以LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE)。默认值：LV_BLEND_MODE_NORMAL。

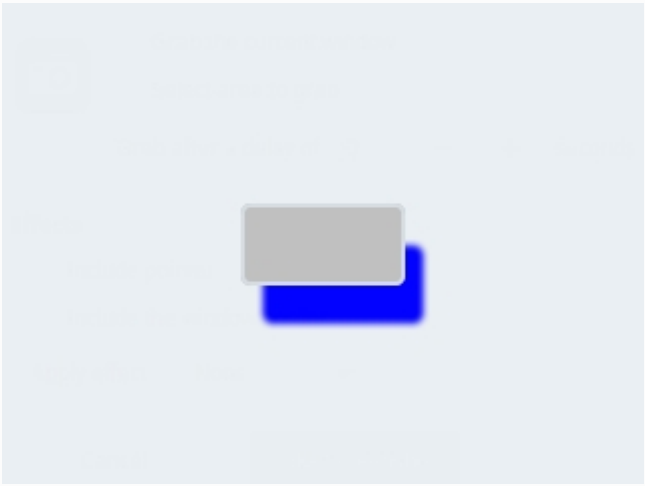

```
#include "../../lv_examples.h"

/**
 * Using the outline style properties
 */
void lv_ex_style_3(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
```

```
    /*Add outline*/
    lv_style_set_outline_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_outline_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_outline_pad(&style, LV_STATE_DEFAULT, 8);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.6 阴影属性

阴影是对象下方的模糊区域。

- shadow_color（lv_color_t）指定阴影的颜色。默认值：LV_COLOR_BLACK。
- shadow_opa（lv_opa_t）指定阴影的不透明度。默认值：LV_OPA_TRANSP。
- shadow_width（lv_style_int_t）：设置轮廓的宽度（模糊大小）。默认值：0。
- shadow_ofs_x（lv_style_int_t）：设置阴影的 X 偏移量。默认值：0。
- shadow_ofs_y（lv_style_int_t）：设置阴影的 Y 偏移量。默认值：0。
- shadow_spread（lv_style_int_t）：在每个方向上使阴影大于背景的值达到此值。默认值：0。
- shadow_blend_mode（lv_blend_mode_t）：设置阴影的混合模式。可以LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE）。默认值：LV_BLEND_MODE_NORMAL。



```
#include "../../lv_examples.h"

/**
 * Using the Shadow style properties
 */
void lv_ex_style_4(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
```

```
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add a shadow*/
    lv_style_set_shadow_width(&style, LV_STATE_DEFAULT, 8);
    lv_style_set_shadow_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_shadow_ofs_x(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_shadow_ofs_y(&style, LV_STATE_DEFAULT, 20);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.7 图案属性

图案是在背景中间绘制或重复以填充整个背景的图像（或符号）。

- pattern_image（）：指向变量的指针，图像文件或符号的 path。默认值：。

const void *lv_img_dsc_tNULL

- pattern_opa（lv_opa_t）：指定图案的不透明度。默认值：LV_OPA_COVER。
- pattern_recolor（lv_color_t）：将此颜色混合到图案图像中。如果是符号（文本），它将是文本颜色。默认值：LV_COLOR_BLACK。
- pattern_recolor_opa（lv_opa_t）：重着色的强度。默认值：（LV_OPA_TRANSP 不重新着色）。
- pattern_repeat（bool）：：true 图案将作为马赛克重复。false：将图案放置在背景中间。默认值：false。
- pattern_blend_mode（lv_blend_mode_t）：设置图案的混合模式。可以 LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE）。默认值：LV_BLEND_MODE_NORMAL。

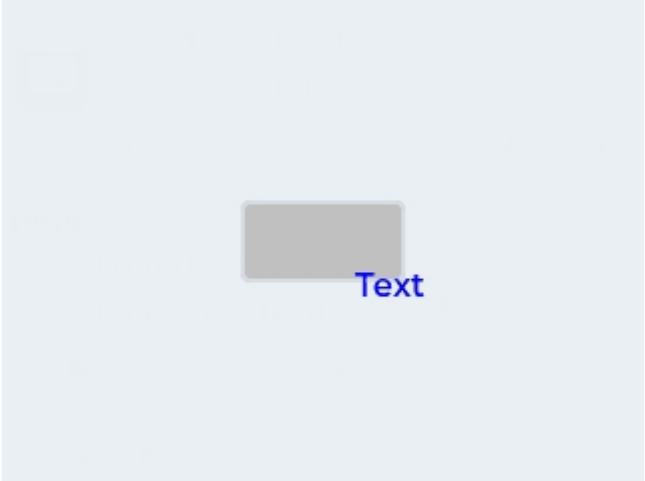
```
#include "../../lv_examples.h"

/**
 * Using the pattern style properties
 */
```

```c
void lv_ex_style_5(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add a repeating pattern*/
    lv_style_set_pattern_image(&style, LV_STATE_DEFAULT, LV_SYMBOL_OK);
    lv_style_set_pattern_recolor(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_pattern_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);
    lv_style_set_pattern_repeat(&style, LV_STATE_DEFAULT, true);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.8 数值属性

值是绘制到背景的任意文本。它可以是创建标签对象的轻量级替代。

- *value_str（）：指向要显示的文本的指针。仅保存指针！（不要将局部变量与 lv_style_set_value_str 一起使用，而应使用静态，全局或动态分配的数据）。默认值：。const char *NULL*
- *value_color（lv_color_t）：文本的颜色。默认值：LV_COLOR_BLACK。*
- *value_opa（lv_opa_t）：文本的不透明度。默认值：LV_OPA_COVER。*
- *value_font（）：指向文本字体的指针。默认值：。const lv_font_t *NULL*
- *value_letter_space（lv_style_int_t）：文本的字母空间。默认值：0。*
- *value_line_space（lv_style_int_t）：文本的行距。默认值：0。*
- *value_align（lv_align_t）：文本的对齐方式。可以 LV_ALIGN_...。默认值：LV_ALIGN_CENTER。*
- *value_ofs_x（lv_style_int_t）：与路线原始位置的 X 偏移量。默认值：0。*
- *value_ofs_y（lv_style_int_t）：从路线的原始位置偏移 Y。默认值：0。*
- *value_blend_mode（lv_blend_mode_t）：设置文本的混合模式。可以 LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE）。默认值：LV_BLEND_MODE_NORMAL。*




```c
#include "../../lv_examples.h"

/**
 * Using the value style properties
 */
void lv_ex_style_6(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add a value text properties*/
    lv_style_set_value_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_value_align(&style, LV_STATE_DEFAULT, LV_ALIGN_IN_BOTTOM_RIGHT);
    lv_style_set_value_ofs_x(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_value_ofs_y(&style, LV_STATE_DEFAULT, 10);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);

    /*Add a value text to the local style. This way every object can have different text*/
    lv_obj_set_style_local_value_str(obj, LV_OBJ_PART_MAIN, LV_STATE_DEFAULT, "Text");
}
```


4.10.9 文字属性

文本对象的属性。

- text_color（lv_color_t）：文本的颜色。默认值：LV_COLOR_BLACK。
- text_opa（lv_opa_t）：文本的不透明度。默认值：LV_OPA_COVER。

- text_font（）：指向文本字体的指针。默认值：。const lv_font_t *NULL
- text_letter_space（lv_style_int_t）：文本的字母空间。默认值：0。
- text_line_space（lv_style_int_t）：文本的行距。默认值：0。
- text_decor（lv_text_decor_t）：添加文字修饰。可以LV_TEXT_DECOR_NONE/UNDERLINE/STRIKETHROUGH。默认值：LV_TEXT_DECOR_NONE。
- text_sel_color（lv_color_t）：设置文本选择的背景色。默认值：LV_COLOR_BLACK
- text_blend_mode（lv_blend_mode_t）：设置文本的混合模式。可以LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE)。默认值：LV_BLEND_MODE_NORMAL。


```
#include "../../lv_examples.h"

/**
 * Using the text style properties
 */
void lv_ex_style_7(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);

    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_bottom(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 10);

    lv_style_set_text_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_text_letter_space(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_text_line_space(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_text_decor(&style, LV_STATE_DEFAULT, LV_TEXT_DECOR_UNDERLINE);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_label_create(lv_scr_act(), NULL);
```

```
    lv_obj_add_style(obj, LV_LABEL_PART_MAIN, &style);
    lv_label_set_text(obj, "Text of\n"
                           "a label");
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.10 线属性

线的属性。

- line_color（lv_color_t）：线条的颜色。默认值：LV_COLOR_BLACK
- line_opa（lv_opa_t）：直线的不透明度。默认值：LV_OPA_COVER
- line_width（lv_style_int_t）：线的宽度。默认值：0。
- line_dash_width（lv_style_int_t）：破折号的宽度。仅对水平或垂直线绘制虚线。0：禁用破折号。默认值：0。
- line_dash_gap（lv_style_int_t）：两条虚线之间的间隙。仅对水平或垂直线绘制虚线。0：禁用破折号。默认值：0。
- line_rounded（bool）：：true 绘制圆角的线尾。默认值：false。
- line_blend_mode（lv_blend_mode_t）：设置线条的混合模式。可以 LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE）。默认值：LV_BLEND_MODE_NORMAL。


```
#include "../../lv_examples.h"

/**
 * Using the line style properties
 */
void lv_ex_style_8(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    lv_style_set_line_color(&style, LV_STATE_DEFAULT, LV_COLOR_GRAY);
    lv_style_set_line_width(&style, LV_STATE_DEFAULT, 6);
    lv_style_set_line_rounded(&style, LV_STATE_DEFAULT, true);

    /*Create an object with the new style*/
```

```
    lv_obj_t * obj = lv_line_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_LINE_PART_MAIN, &style);

    static lv_point_t p[] = {{10, 30}, {30, 50}, {100, 0}};
    lv_line_set_points(obj, p, 3);

    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.11 图片属性

图像的属性。

- image_recolor（lv_color_t）：将此颜色混合到图案图像中。如果是符号（文本），它将是文本颜色。默认值：LV_COLOR_BLACK
- image_recolor_opa（lv_opa_t）：重新着色的强度。默认值：（LV_OPA_TRANSP 不重新着色）。默认值：LV_OPA_TRANSP
- image_opa（lv_opa_t）：图像的不透明度。默认值：LV_OPA_COVER
- image_blend_mode（lv_blend_mode_t）：设置图像的混合模式。可以 LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE）。默认值：LV_BLEND_MODE_NORMAL。

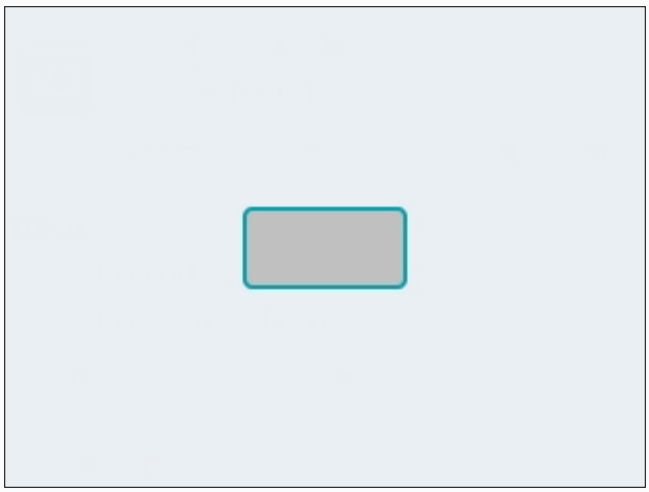

```
#include "../../lv_examples.h"

/**
 * Using the image style properties
 */
void lv_ex_style_9(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
```

```
    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_bottom(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 10);

    lv_style_set_image_recolor(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_image_recolor_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_img_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_IMG_PART_MAIN, &style);
    LV_IMG_DECLARE(img_cogwheel_argb);
    lv_img_set_src(obj, &img_cogwheel_argb);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

4.10.12 转换特性

用于描述状态更改动画的属性。

- transition_time（lv_style_int_t）：过渡时间。默认值：0。
- transition_delay（lv_style_int_t）：转换前的延迟。默认值：0。
- transition_prop_1（）：应在其上应用过渡的属性。将属性名称与大写字母一起使用，例如。默认值：0（无）。property nameLV_STYLE_LV_STYLE_BG_COLOR
- transition_prop_2（）：与 transition_1 相同，只是另一个属性。默认值：0（无）。property name
- transition_prop_3（）：与 transition_1 相同，只是另一个属性。默认值：0（无）。property name
- transition_prop_4（）：与 transition_1 相同，只是另一个属性。默认值：0（无）。property name
- transition_prop_5（）：与 transition_1 相同，只是另一个属性。默认值：0（无）。property name
- transition_prop_6（）：与 transition_1 相同，只是另一个属性。默认值：0（无）。property name
- transition_path（lv_anim_path_t）：过渡的动画 path。（需要为静态或全局变量，因为仅保存了其指针）。默认值：（lv_anim_path_def 线性 path）。



```
#include "../../lv_examples.h"

/**
```

```
 * Using the transitions style properties
 */
void lv_ex_style_10(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Set different background color in pressed state*/
    lv_style_set_bg_color(&style, LV_STATE_PRESSED, LV_COLOR_GRAY);

    /*Set different transition time in default and pressed state
     *fast press, slower revert to default*/
    lv_style_set_transition_time(&style, LV_STATE_DEFAULT, 500);
    lv_style_set_transition_time(&style, LV_STATE_PRESSED, 200);

    /*Small delay to make transition more visible*/
    lv_style_set_transition_delay(&style, LV_STATE_DEFAULT, 100);

    /*Add `bg_color` to transitioned properties*/
    lv_style_set_transition_prop_1(&style, LV_STATE_DEFAULT, LV_STYLE_BG_COLOR);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```


4.10.13 比例属性

鳞片状元素的辅助属性。体重秤具有正常区域和末端区域。顾名思义，结束区域是标度的结束，可以是临界值或 void 值。正常区域在结束区域之前。两个区域可能具有不同的属性。

- **scale_grad_color（lv_color_t）：在正常区域中，在比例尺线上对该颜色进行渐变。默认值：LV_COLOR_BLACK。**
- **scale_end_color（lv_color_t）：结束区域中刻度线的颜色。默认值：LV_COLOR_BLACK。**
- **scale_width（lv_style_int_t）：比例尺的宽度。默认值：。默认值：。LV_DPI / 8LV_DPI / 8**
- **scale_border_width（lv_style_int_t）：在标准区域的比例尺外侧绘制的边框的宽度。默认值：0。**
- **scale_end_border_width（lv_style_int_t）：在结束区域的刻度外侧上绘制边框的宽度。默认值：0。**
- **scale_end_line_width（lv_style_int_t）：结束区域中比例线的宽度。默认值：0。**


```c
#include "../../lv_examples.h"

/**
 * Using the scale style properties
 */
void lv_ex_style_11(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Set some paddings*/
    lv_style_set_pad_inner(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 5);

    lv_style_set_scale_end_color(&style, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_line_color(&style, LV_STATE_DEFAULT, LV_COLOR_WHITE);
    lv_style_set_scale_grad_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_line_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_scale_end_line_width(&style, LV_STATE_DEFAULT, 4);
    lv_style_set_scale_end_border_width(&style, LV_STATE_DEFAULT, 4);

    /*Gauge has a needle but for simplicity its style is not initialized here*/

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_gauge_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_GAUGE_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```


在小部件的文档中，您将看到诸如“小部件使用典型的背景属性”之类的句子。“典型背景”属性是：

- **背景**
- **边境**
- **大纲**
- **阴影**
- **模式**
- **值**

4.10.14 主题

主题是样式的集合。始终有一个活动主题，在创建对象时会自动应用其样式。它为 UI 提供了默认外观，可以通过添加其他样式来对其进行修改。

默认的主题被设定在 `lv_conf.h` 与 `LV_THEME_...` 定义。每个主题都具有以下属性

- **原色**
- **二次色**
- **小字体**
- **普通字体**
- **字幕字体**
- **标题字体**
- **标志（特定于给定主题）**

如何使用这些属性取决于主题。

有 3 个内置主题：

- empty **空：未添加默认样式**
- material **材质：令人印象深刻的现代主题-单声道：用于黑白显示的简单黑白主题**
- template **模板：一个非常简单的主题，可以将其复制以创建自定义主题**

4.10.15 扩展主题

内置主题可以通过自定义主题进行扩展。如果创建了自定义主题，则可以选择“基本主题”。基本主题的样式将添加到自定义主题之前。可以链接任何数量的主题。例如，材料主题->自定义主题->黑暗主题。

这是有关如何基于当前活动的内置主题创建自定义主题的示例。

```
    /*Get the current theme (e.g. material). It will be the base of the custom theme.*/
    lv_theme_t * base_theme = lv_theme_get_act();

    /*Initialize a custom theme*/
    static lv_theme_t custom_theme;                      /*Declare a theme*/
    lv_theme_copy(&custom_theme, )base_theme;             /*Initialize the custom theme from the base theme*/
    lv_theme_set_apply_cb(&custom_theme, custom_apply_cb);  /*Set a custom theme apply callback*/
    lv_theme_set_base(custom_theme, base_theme);           /*Set the base theme of the csutom theme*/

    /*Initialize styles for the new theme*/
    static lv_style_t style1;
    lv_style_init(&style1);
    lv_style_set_bg_color(&style1, LV_STATE_DEFAULT, custom_theme.color_primary);
```

```
...

/*Add a custom apply callback*/
static void custom_apply_cb(lv_theme_t * th, lv_obj_t * obj, lv_theme_style_t name)
{
    lv_style_list_t * list;

    switch(name) {
        case LV_THEME_BTN:
            list = lv_obj_get_style_list(obj, LV_BTN_PART_MAIN);
            _lv_style_list_add_style(list, &my_style);
            break;
    }
}
```

示例

造型按钮

代码：

```
#include "../../lv_examples.h"


/**
 * Create styles from scratch for buttons.
 */
void lv_ex_get_started_2(void)
{
    static lv_style_t style_btn;
    static lv_style_t style_btn_red;

    /*Create a simple button style*/
    lv_style_init(&style_btn);
    lv_style_set_radius(&style_btn, LV_STATE_DEFAULT, 10);
    lv_style_set_bg_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_COVER);
```

```
    lv_style_set_bg_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_GRAY);
    lv_style_set_bg_grad_dir(&style_btn, LV_STATE_DEFAULT, LV_GRAD_DIR_VER);

    /*Swap the colors in pressed state*/
    lv_style_set_bg_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_GRAY);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_SILVER);

    /*Add a border*/
    lv_style_set_border_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);
    lv_style_set_border_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_70);
    lv_style_set_border_width(&style_btn, LV_STATE_DEFAULT, 2);

    /*Different border color in focused state*/
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED, LV_COLOR_BLUE);
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED | LV_STATE_PRESSED, LV_COLOR_NAVY);

    /*Set the text style*/
    lv_style_set_text_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);

    /*Make the button smaller when pressed*/
    lv_style_set_transform_height(&style_btn, LV_STATE_PRESSED, -5);
    lv_style_set_transform_width(&style_btn, LV_STATE_PRESSED, -10);

    /*Add a transition to the size change*/
    static lv_anim_path_t path;
    lv_anim_path_init(&path);
    lv_anim_path_set_cb(&path, lv_anim_path_overshoot);

    lv_style_set_transition_prop_1(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_HEIGHT);
    lv_style_set_transition_prop_2(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_WIDTH);
    lv_style_set_transition_time(&style_btn, LV_STATE_DEFAULT, 300);
    lv_style_set_transition_path(&style_btn, LV_STATE_DEFAULT, &path);


    /*Create a red style. Change only some colors.*/
    lv_style_init(&style_btn_red);
    lv_style_set_bg_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_MAROON);
    lv_style_set_bg_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_MAROON);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_RED);
    lv_style_set_text_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_WHITE);

    /*Create buttons and use the new styles*/
    lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);     /*Add a button the current screen*/
    lv_obj_set_pos(btn, 10, 10);                            /*Set its position*/
    lv_obj_set_size(btn, 120, 50);                          /*Set its size*/
    lv_obj_reset_style_list(btn, LV_BTN_PART_MAIN);         /*Remove the styles coming from the theme*/
    lv_obj_add_style(btn, LV_BTN_PART_MAIN, &style_btn);
```

```
    lv_obj_t * label = lv_label_create(btn, NULL);          /*Add a label to the button*/
    lv_label_set_text(label, "Button");                     /*Set the labels text*/

    /*Create a new button*/
    lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), btn);
    lv_obj_set_pos(btn2, 10, 80);
    lv_obj_set_size(btn2, 120, 50);                          /*Set its size*/
    lv_obj_reset_style_list(btn2, LV_BTN_PART_MAIN);        /*Remove the styles coming from the
theme*/
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn);
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn_red);   /*Add the red style on top of the
current */
    lv_obj_set_style_local_radius(btn2, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, LV_RADIUS_CIRCLE); /*Add a
local style*/

    label = lv_label_create(btn2, NULL);         /*Add a label to the button*/
    lv_label_set_text(label, "Button 2");                    /*Set the labels text*/
}
```

Cai Xuefeng

第五章 输入设备

输入设备通常意味着：

- 指针式输入设备，如触摸板或鼠标
- 键盘，如普通键盘或简单的数字键盘
- 带有左/右转向和推入选项的编码器
- 外部硬件按钮，分配给屏幕上的特定点

重要

在进一步阅读之前，请阅读输入设备的[Porting]（/ porting / indev）部分

5.1 指针

指针输入设备可以具有光标。（通常是鼠标）

```
...
lv_indev_t * mouse_indev = lv_indev_drv_register(&indev_drv);

LV_IMG_DECLARE(mouse_cursor_icon);                      /*Declare the image file.*/
lv_obj_t * cursor_obj =  lv_img_create(lv_scr_act(), NULL); /*Create an image object for the cursor */
lv_img_set_src(cursor_obj, &mouse_cursor_icon);           /*Set the image source*/
lv_indev_set_cursor(mouse_indev, cursor_obj);             /*Connect the image  object to the driver*/
```

请注意，游标对象应具有。对于图像，默认情况下禁用*单击*。

```
lv_obj_set_click(cursor_obj, false)
```


5.2 键盘和编码器

您可以使用键盘或编码器完全控制用户界面，而无需触摸板或鼠标。它的作用类似于 PC 上的 *TAB* 键，可以在应用程序或网页中选择元素。

5.3 组

您想要通过键盘或编码器控制的对象需要添加到 *Group* *中*。在每个组中，只有一个集中的对象可以接收按下的键或编码器的动作。例如，如果将文本区域作为焦点，并且您在键盘上按了某个字母，则将发送键并将其插入到文本区域中。同样，如果将滑块聚焦，然后按向左或向右箭头，则滑块的值将被更改。

您需要将输入设备与组关联。一台输入设备只能将键发送到一组，但一组也可以从多个输入设备接收数据。

要创建组用途并将对象添加到组中，请使用。`lv_group_t * g = lv_group_create()lv_group_add_obj(g, obj)`

要将组与输入设备关联，请使用，其中是的返回值

`lv_indev_set_group(indev, g)indevlv_indev_drv_register()`

5.4 按键

有一些预定义的键具有特殊含义：

- **LV_KEY_NEXT** 专注于下一个对象
- **LV_KEY_PREV** 专注于上一个对象
- **LV_KEY_ENTER** 触发器 `LV_EVENT_PRESSED/CLICKED/LONG_PRESSED` 等事件
- **LV_KEY_UP** 增加值或向上移动
- **LV_KEY_DOWN** 减小值或向下移动
- **LV_KEY_RIGHT** 增加值或向右移动
- **LV_KEY_LEFT** 减小值或向左移动
- **LV_KEY_ESC** 关闭或退出（例如，关闭下拉列表）
- **LV_KEY_DEL** 删除（例如，“ 文本”区域中右侧的字符）
- **LV_KEY_BACKSPACE** 删除左侧的字符（例如，在文本区域中）
- **LV_KEY_HOME** 转到开头/顶部（例如，在“ 文本”区域中）
- **LV_KEY_END** 转到末尾（例如，在“ 文本”区域中）

最重要的特殊键 `LV_KEY_NEXT/PREV`， `LV_KEY_ENTER` 和 `LV_KEY_UP/DOWN/LEFT/RIGHT`。在 `read_cb` 功能中，应将一些键转换为这些特殊键，以便在组中导航并与所选对象进行交互。

通常，仅使用它就足够了， `LV_KEY_LEFT/RIGHT` 因为大多数对象可以用它们完全控制。

带编码器，你应该只使用 `LV_KEY_LEFT`， `LV_KEY_RIGHT` 和 `LV_KEY_ENTER`。

5.5 编辑和浏览模式

由于键盘有很多键，因此很容易在对象之间导航并使用键盘进行编辑。但是，编码器的“键”数量有限，因此很难使用默认选项进行导航。创建*导航*和*编辑*是为了避免编码器出现此问题。

在*导航*模式下，编码器 `LV_KEY_LEFT/RIGHT` 转换为 `LV_KEY_NEXT/PREV`。因此，将通过旋转编码器选择下一个或上一个对象。按 `LV_KEY_ENTER` 将更改为*编辑*模式。

在“ *编辑*”模式下， `LV_KEY_NEXT/PREV` 通常用于编辑对象。根据对象的类型，短按或长按可将其 `LV_KEY_ENTER` 更改回*导航*模式。通常，无法按下的对象（如 Slider）会在短按时离开“ *编辑*”模式。但是，对于具有短单击含义的对象（例如 Button），需要长按。

5.6 造型

如果通过触摸板单击对象或通过编码器或键盘将其聚焦，则转到 `LV_STATE_FOCUSED`。因此，将重点应用样式。

如果对象进入编辑模式，它将进入状态，因此将显示这些样式属性。 `LV_STATE_FOCUSED | LV_STATE_EDITED`

5.7 API

5.7.1 输入设备

功能

void lv_indev_init（ void ）

初始化显示输入设备子系统

void lv_indev_read_task（lv_task_t * *task* ）

定期调用以读取输入设备

参数

`task`：指向任务本身的指针

lv_indev_t * lv_indev_get_act（ void ）

获取当前处理的输入设备。也可以在动作功能中使用。

返回

指向当前正在处理的输入设备的指针；如果当前没有输入设备在处理，则为 NULL

lv_indev_type_t lv_indev_get_type（*const*lv_indev_t * *indev* ）

获取输入设备的类型

返回

lv_hal_indev_type_t （LV_INDEV_TYPE_...）中输入设备的类型

参数

`indev`：指向输入设备的指针

voidlv_indev_reset（lv_indev_t * *indev*，lv_obj_t * *obj* ）

重置一个或所有输入设备

参数

`indev`：指向输入设备以重置的指针，或者为 NULL 重置所有指针的指针

`obj`：指向触发复位的对象的指针。

voidlv_indev_reset_long_press（lv_indev_t * *indev* ）

重置输入设备的长按状态

参数

`indev_proc`：指向输入设备的指针

voidlv_indev_enable（lv_indev_t * *indev*，bool *en* ）

启用或禁用输入设备

参数

`indev`：指向输入设备的指针

`en`：true：启用；false：禁用

---

void lv_indev_set_cursor（lv_indev_t * indev，lv_obj_t * cur_obj ）

设置指针输入设备的光标（用于 LV_INPUT_TYPE_POINTER 和 LV_INPUT_TYPE_BUTTON）

参数

`indev`：指向输入设备的指针

`cur_obj`：指向要用作光标的对象的指针

---

void lv_indev_set_group（lv_indev_t * indev，lv_group_t * group ）

设置键盘输入设备的目标组（对于 LV_INDEV_TYPE_KEYPAD）

参数

`indev`：指向输入设备的指针

`group`：指向一个组

---

void lv_indev_set_button_points（lv_indev_t * indev，const lv_point_t 点 [] ）

设置 LV_INDEV_TYPE_BUTTON 的点数组。这些点将分配给按钮以按屏幕上的特定点

参数

`indev`：指向输入设备的指针

`group`：指向一个组

---

void lv_indev_get_point（const lv_indev_t * indev，lv_point_t * point ）

获取输入设备的最后一点（对于 LV_INDEV_TYPE_POINTER 和 LV_INDEV_TYPE_BUTTON）

参数

`indev`：指向输入设备的指针

`point`：指向存储结果的点的指针

---

lv_gesture_dir_t lv_indev_get_gesture_dir（const lv_indev_t * indev ）

直接获取当前手势

返回

当前手势直接

参数

`indev`：指向输入设备的指针

---

uint32_t lv_indev_get_key（const lv_indev_t * indev ）

获取输入设备的最后按下键（对于 LV_INDEV_TYPE_KEYPAD）

返回

最后按下的键（错误时为 0）

参数

`indev`：指向输入设备的指针

bool lv_indev_is_dragging（const lv_indev_t * indev ）

检查是否有输入设备拖动（对于 LV_INDEV_TYPE_POINTER 和 LV_INDEV_TYPE_BUTTON）

返回

true：拖动正在进行中

参数

`indev`：指向输入设备的指针

void lv_indev_get_vect（const lv_indev_t * indev，lv_point_t * point ）

获取输入设备的拖动向量（对于 LV_INDEV_TYPE_POINTER 和 LV_INDEV_TYPE_BUTTON）

参数

`indev`：指向输入设备的指针

`point`：指向存储向量的点的指针

lv_res_t lv_indev_finish_drag（lv_indev_t * indev ）

手动完成拖动。`LV_SIGNAL_DRAG_END` 并且 `LV_EVENT_DRAG_END` 将被发送。

返回

`LV_RES_INV` 如果要拖动的对象已删除。别的 `LV_RES_OK`。

参数

`indev`：指向输入设备的指针

void lv_indev_wait_release（lv_indev_t * indev ）

在下一个发行版之前什么都不做

参数

`indev`：指向输入设备的指针

lv_obj_t * lv_indev_get_obj_act（ void ）

获取 indev proc 函数中当前活动对象的指针。如果当前未处理任何对象或未使用组，则为 NULL。

返回

指向当前活动对象的指针

lv_obj_t * lv_indev_search_obj（lv_obj_t * obj，lv_point_t * point ）

按一点搜索最顶部，可点击的对象

返回

指向找到的对象的指针；如果没有合适的对象，则为 NULL

参数

`obj`：指向起始对象（通常是屏幕）的指针

`point`：指向搜索最高位子的点的指针

lv_task_t * lv_indev_get_read_task（lv_disp_t * *indev* ）

获取指向 indev 读取任务的指针，以使用 `lv_task_...` 函数修改其参数。

返回

指向 indev 读取刷新任务的指针。（错误时为 NULL）

参数

`indev`：指向输入设备的指针

5.72 组

typedef

*typedef*uint8_t lv_key_t

*typedef*void（* lv_group_style_mod_cb_t）（*struct* lv_group_t *，lv_style_t * ）

*typedef*void（* lv_group_focus_cb_t）（*struct* lv_group_t * ）

typedefstruct lv_group_tlv_group_t

组可以用于逻辑上保存对象，以便可以将它们单独聚焦。它们不是用于在屏幕上布置对象（`lv_cont` 为此尝试）。

*typedef*uint8_t lv_group_refocus_policy_t

枚举

enum [anonymous]

值：

*enumerator*LV_KEY_UP = 17

*enumerator*LV_KEY_DOWN = 18

*enumerator*LV_KEY_RIGHT = 19

*enumerator*LV_KEY_LEFT = 20

*enumerator*LV_KEY_ESC = 27

*enumerator*LV_KEY_DEL = 127

*enumerator*LV_KEY_BACKSPACE = 8

*enumerator*LV_KEY_ENTER = 10

*enumerator*LV_KEY_NEXT = 9

*enumerator*LV_KEY_PREV = 11

*enumerator*LV_KEY_HOME = 2

enumerator **LV_KEY_END = 3**

enum [anonymous]

值:

enumerator **LV_GROUP_REFOCUS_POLICY_NEXT = 0**

enumerator **LV_GROUP_REFOCUS_POLICY_PREV = 1**

功能

void **_lv_group_init**（ void ）

在里面。组模块

备注

内部函数，请勿直接调用。

lv_group_t * **lv_group_create**（ void ）

创建一个新的对象组

返回

指向新对象组的指针

void **lv_group_del**（lv_group_t * *group*）

删除组对象

参数

`group`：指向组的指针

void **lv_group_add_obj**（lv_group_t * *group*, lv_obj_t * *obj* ）

将对象添加到组

参数

`group`：指向组的指针

`obj`：指向要添加的对象的指针

void **lv_group_remove_obj**（lv_obj_t * *obj* ）

从组中删除对象

参数

`obj`：指向要删除的对象的指针

void **lv_group_remove_all_objs**（lv_group_t * *group*）

从组中删除所有对象

参数

`group`：指向组的指针

void **lv_group_focus_obj**（lv_obj_t * *obj* ）

专注于一个对象（散焦当前）

参数

`obj`：指向要关注的对象的指针

void lv_group_focus_next（lv_group_t * group）

聚焦组中的下一个对象（使当前焦点散焦）

> 参数

`group`：指向组的指针

void lv_group_focus_prev（lv_group_t * group）

将上一个对象聚焦在一个组中（使当前对象散焦）

> 参数

`group`：指向组的指针

void lv_group_focus_freeze（lv_group_t * group，bool en ）

不要让焦点从当前对象改变

> 参数

`group`：指向组的指针

`en`：真：冻结，假：释放冻结（正常模式）

lv_res_t lv_group_send_data（lv_group_t * group，uint32_t c ）

将控制字符发送到组的焦点对象

> 返回

组中聚焦对象的结果。

> 参数

`group`：指向组的指针

`c`：一个字符（使用 LV_KEY_ ..进行导航）

void lv_group_set_focus_cb（lv_group_t * group，lv_group_focus_cb_t focus_cb ）

为一个组设置一个函数，当一个新对象被聚焦时将被调用

> 参数

`group`：指向组的指针

`focus_cb`：回调函数；如果未使用，则为 NULL

void lv_group_set_refocus_policy（lv_group_t * group，lv_group_refocus_policy_t policy）

设置如果删除当前焦点的 obj，则焦点对准组中的下一个项目还是上一个项目。

> 参数

`group`：指向组的指针

`new`：重新调整政策枚举

void lv_group_set_editing（lv_group_t * group，bool edit）

手动设置当前模式（编辑或导航）。

> 参数

`group`：指向组的指针

`edit`：true：编辑模式；false：导航模式

void lv_group_set_click_focus（lv_group_t * group，bool en ）

设置 `click_focus` 属性。如果启用，则将聚焦对象，然后单击它。

> 参数

`group`：指向组的指针

`en`：true：启用 `click_focus`

void lv_group_set_wrap（lv_group_t * group，bool en ）

设置焦点下一个/上一个是否允许从 first-> last 或 last-> first 对象包装。

> 参数

`group`：指向组的指针

`en`：true：启用包装；false：禁止包装

lv_obj_t * lv_group_get_focused（const lv_group_t * group）

获取焦点对象；如果没有，则为 NULL

> 返回

指向聚焦对象的指针

> 参数

`group`：指向组的指针

lv_group_user_data_t * lv_group_get_user_data（lv_group_t * group ）

获取指向该组用户数据的指针

> 返回

指向用户数据的指针

> 参数

`group`：指向组的指针

lv_group_focus_cb_t lv_group_get_focus_cb（const lv_group_t * group）

获取组的焦点回调函数

> 返回

回调函数；如果未设置，则为 NULL

> 参数

`group`：指向组的指针

bool lv_group_get_editing（const lv_group_t * group）

获取当前模式（编辑或导航）。

返回

true：编辑模式；false：导航模式

参数

`group`：指向组的指针

bool lv_group_get_click_focus（const lv_group_t * group）

获取 click_focus 属性。

返回

true：`click_focus` 启用；假：禁用

参数

`group`：指向组的指针

bool lv_group_get_wrap（lv_group_t * group）

获取下一个/上一个焦点是否允许从 first-> last 或 last-> first 对象包装。

参数

`group`：指向组的指针

`en`：true：启用包装；false：禁止包装

struct _lv_group_t

#include <lv_group.h>

组可以用于逻辑上保存对象，以便可以将它们单独聚焦。它们不是用于在屏幕上布置对象（`lv_cont` 为此尝试）。

公众成员

lv_ll_t **obj_ll**

链接列表以将对象存储在组中

lv_obj_t ****obj_focus**

聚焦对象

lv_group_focus_cb_t**focus_cb**

聚焦新对象时调用的函数（可选）

lv_group_user_data_t **user_data**

uint8_t **frozen**

1：不能专注于新对象

uint8_t **editing**

1：编辑模式，0：导航模式

```
uint8_t click_focus
```

1：如果组中的某个对象被 indev 单击，则它将成为焦点

```
uint8_t refocus_policy
```

1：如果关注删除，则优先关注。0：如果关注删除，则关注下一个。

```
uint8_t wrap
```

1：焦点下一个/上一个可以包装在列表的末尾。0：焦点下一个/上一个在列表末尾停止。

5.8 显示

重要

LVGL 中*显示*的基本概念在[Porting]（/ porting / display）部分中进行了说明。因此，在进一步阅读之前，请先阅读[Porting]（/ porting / display）部分。

5.81 多种显示支持

在 LVGL 中，您可以有多个显示，每个显示都有自己的驱动程序和对象。唯一的限制是，每个显示器都必须具有相同的色深（如中所述 `LV_COLOR_DEPTH`）。如果在这方面显示有所不同，则可以在驱动程序中将渲染的图像转换为正确的格式 `flush_cb`。

创建更多的显示很容易：只需初始化更多的显示缓冲区并为每个显示注册另一个驱动程序。创建 UI 时，用于 `lv_disp_set_default(disp)` 告诉库在其上创建对象的显示。

为什么要多显示器支持？这里有些例子：

- 具有带有本地 UI 的“常规” TFT 显示屏，并根据需要在 VNC 上创建“虚拟”屏幕。（您需要添加您的 VNC 驱动程序）。
- 具有大型 TFT 显示屏和小型单色显示屏。
- 在大型仪器或技术中具有一些较小且简单的显示器。
- 有两个大型 TFT 显示屏：一个用于客户，一个用于店员。

5.8.2 仅使用一个显示器

使用更多显示器可能很有用，但在大多数情况下，并非必需。因此，如果仅注册一个显示器，则整个多显示器概念将被完全隐藏。默认情况下，最后创建的（唯一的）显示用作默认设置。

`lv_scr_act()`，`lv_scr_load(scr)`，`lv_layer_top()`，`lv_layer_sys()`，`LV_HOR_RES` 和 `LV_VER_RES` 始终施加的最后创建（默认）屏幕上。如果你传递 `NULL` 作为 `disp` 参数来显示相关的功能，通常会使用默认的显示。例如，`lv_disp_trig_activity(NULL)` 将在默认屏幕上触发用户活动。

5.8.3 视镜显示

要将显示器的图像镜像到另一个显示器，则无需使用多显示器支持。只需将接收 `drv.flush_cb` 到的缓冲区也转移到另一个显示器即可。

5.84 分割影像

您可以从较小的显示创建较大的显示。您可以如下创建它：

1. 将显示器的分辨率设置为大显示器的分辨率。
2. 在中 `drv.flush_cb`，截断并修改 `area` 每个显示的参数。
3. 将缓冲区的内容发送到带有截断区域的每个显示。

5.8.5 屏幕

每个显示器都有每组屏幕和屏幕上的对象。
确保不要混淆显示和屏幕：

- **显示**是绘制像素的物理硬件。
- **屏幕**是与特定显示器关联的高级根对象。一个显示器可以具有与其关联的多个屏幕，但反之则不然。

屏幕可以被认为是没有父级的最高级别的容器。屏幕的大小始终等于其显示，屏幕的大小始终为（0; 0）。因此，屏幕坐标不能更改，即 `lv_obj_set_pos()`， `lv_obj_set_size()` 或者屏幕上不能使用类似功能。

可以从任何对象类型创建屏幕，但是，两种最典型的类型是基础对象和图像（用于创建墙纸）。

要创建屏幕，请使用。可以是另一个屏幕来复制它。 `lv_obj_t * scr = lv_<type>_create(NULL, copy)copy`

要加载屏幕，请使用 `lv_scr_load(scr)`。要使用活动屏幕，请使用 `lv_scr_act()`。这些功能在默认显示屏上起作用。如果要指定要处理的显示，请使用 `lv_disp_get_scr_act(disp)` 和。屏幕也可以加载动画。在这里阅读更多。 `lv_disp_load_scr(disp, scr)`

可以使用删除屏幕 `lv_obj_del(scr)`，但是请确保您不删除当前加载的屏幕。

5.8.6 透明屏幕

通常，屏幕的不透明度是 `LV_OPA_COVER` 为其子级提供坚实的背景。如果不是这种情况（不透明度<100%），则显示器的背景颜色或图像将可见。有关更多详细信息，请参见显示背景部分。如果显示器的背景不透明性也不是，则 `LV_OPA_COVER` LVGL 不会绘制纯色背景。

此配置（透明屏幕和显示器）可用于创建 OSD 菜单，例如在其中将视频播放到下层，并在上层创建菜单。
为了处理透明显示器，LVGL 需要使用特殊（较慢）的颜色混合算法，因此需要使用

`LV_COLOR_SCREEN_TRANSP` n 启用此功能 `lv_conf.h`。由于此模式在像素的 Alpha 通道上运行，因此也是必需的。32 位颜色的 Alpha 通道在没有对象的情况下将为 0，在有实体对象的情况下将为 255。 `LV_COLOR_DEPTH = 32`

总而言之，要启用透明屏幕和显示以创建类似于 OSD 菜单的 UI:

- 启用 `LV_COLOR_SCREEN_TRANSP` 中 `lv_conf.h`
- 请务必使用 `LV_COLOR_DEPTH 32`

- 将屏幕的不透明度设置为 `LV_OPA_TRANSP` 例如

```
lv_obj_set_style_local_bg_opa(lv_scr_act(), LV_OBJMASK_PART_MAIN, LV_STATE_DEFAULT, LV_OPA_TRANSP)
```

- 将显示不透明度设置为 `LV_OPA_TRANSP` with `lv_disp_set_bg_opa(NULL, LV_OPA_TRANSP);`

5.8.7 显示器功能

1. 不活跃

在每个显示器上测量用户的不活动状态。每次使用输入设备（如果与显示器关联）都被视为一项活动。要获取自上一次活动以来经过的时间，请使用 `lv_disp_get_inactive_time(disp)`。如果 `NULL` 通过，则所有显示（**不是默认显示**）将返回整体最小的不活动时间。

您可以使用手动触发活动 `lv_disp_trig_activity(disp)`。如果 `disp` 为 `NULL`，则将使用默认屏幕（**并非所有显示**）。

5.8.8 背景

每个显示都有背景色，背景图像和背景不透明度属性。当当前屏幕是透明的或未定位为覆盖整个显示器时，它们将变为可见。

背景色是填充显示的一种简单颜色。可以用; `lv_disp_set_bg_color(disp, color)`

背景图像是文件的 path 或指向 `lv_img_dsc_t` 用作墙纸的变量（转换后的图像）的指针。可以设置; 如果设置了背景图像（未设置），则背景不会填充。`lv_disp_set_bg_color(disp, &my_img)NULLbg_color`

可以使用调整背景颜色或图像的不透明度。`lv_disp_set_bg_opa(disp, opa)`

`disp` 这些功能的参数可以将 `NULL` 其引用为默认显示。

5.8.9 色彩

颜色模块处理所有与颜色相关的功能，例如更改颜色深度，从十六进制代码创建颜色，在颜色深度之间转换，混合颜色等。

颜色模块定义了以下变量类型:

- **lv_color1_t** 存储单色。为了兼容性，它也具有 R，G，B 字段，但它们始终是相同的值（1 个字节）
- **lv_color8_t** 用于存储 8 位颜色（1 字节）的 R（3 位），G（3 位），B（2 位）分量的 struct。
- **lv_color16_t** 用于存储 16 位颜色（2 字节）的 R（5 位），G（6 位），B（5 位）分量的 struct。

- **lv_color32_t** 用于存储 24 位颜色（4 字节）的 R（8 位），G（8 位），B（8 位）分量的 struct。
- **lv_color_t** 等于 `lv_color1/8/16/24_t` 根据颜色深度设置
- **lv_color_int_t** `uint8_t`， `uint16_t` 或 `uint32_t` 根据颜色深度设置。用于从纯数字构建颜色阵列。
- **lv_opa_t** 一种简单的 `uint8_t` 类型，用于描述不透明度。

在 `lv_color_t`， `lv_color1_t`， `lv_color8_t`， `lv_color16_t` 和 `lv_color32_t` 类型已经得到了四个字段：

- **ch.red** 红色通道
- **ch.green** 绿色通道
- **ch.blue** 蓝色通道
- 全红+绿+蓝为一个数字

您可以在 *lv_conf.h* 中设置当前颜色深度，方法是将 `LV_COLOR_DEPTH` 定义设置为 1（单色），8、16 或 32。

5.8.10 转换颜色

您可以将一种颜色从当前颜色深度转换为另一种颜色。转换器函数以数字返回，因此您必须使用以下 `full` 字段：


```
lv_color_t c;
c.red   = 0x38;
c.green = 0x70;
c.blue  = 0xCC;

lv_color1_t c1;
c1.full = lv_color_to1(c);       /*Return 1 for light colors, 0 for dark colors*/

lv_color8_t c8;
c8.full = lv_color_to8(c);       /*Give a 8 bit number with the converted color*/

lv_color16_t c16;
c16.full = lv_color_to16(c); /*Give a 16 bit number with the converted color*/

lv_color32_t c24;
c32.full = lv_color_to32(c);     /*Give a 32 bit number with the converted color*/
```

5.8.11 交换 16 种颜色

您可以 `LV_COLOR_16_SWAP` 在 *lv_conf.h* 中设置以交换 *RGB565* 颜色的字节。如果通过 SPI 等面向字节的接口发送 16 位颜色，则很有用。

由于 16 位数字以 Little Endian 格式存储（低位地址的低位字节），因此接口将首先发送低位字节。但是，显示

通常首先需要较高的字节。字节顺序不匹配会导致色彩严重失真。

5.8.12 创建和混合颜色

您可以使用 LV_COLOR_MAKE 宏以当前颜色深度创建颜色。它使用 3 个参数（红色，绿色，蓝色）作为 8 位数字。例如，创建浅红色：。`my_color = COLOR_MAKE(0xFF,0x80,0x80)`

颜色也可以通过十六进制代码创建：或。

`my_color = lv_color_hex(0x288ACF)my_color = lv_folro_hex3(0x28C)`

可以使用混合两种颜色。比例可以是 0..255。0 表示全彩色 2，255 表示全彩色 1。

```
mixed_color = lv_color_mix(color1, color2, ratio)
```

也可以使用从 HSV 空间创建颜色。应该在 0..360 范围内，并且在 0..100 范围内。

`lv_color_hsv_to_rgb(hue, saturation, value)huesaturationvalue`

5.8.13 不透明度

为了描述不透明度，该 `lv_opa_t` 类型被创建为的包装 `uint8_t`。还介绍了一些定义：

- **LV_OPA_TRANSP** 值：0，表示不透明度使颜色完全透明
- **LV_OPA_10** 值：25，表示颜色仅覆盖一点
- **LV_OPA_20 ... OPA_80** 符合逻辑
- **LV_OPA_90** 值：229，表示几乎完全覆盖的颜色
- **LV_OPA_COVER** 值：255，表示颜色完全覆盖

您还可以将 `LV_OPA_*` 定义 `lv_color_mix()` 作为*比率使用*。

5.8.14 内置颜色

颜色模块定义了最基本的颜色，例如：

- #FFFFFF `LV_COLOR_WHITE`
- #000000 `LV_COLOR_BLACK`
- #808080 `LV_COLOR_GRAY`
- #c0c0c0 `LV_COLOR_SILVER`
- #ff0000 `LV_COLOR_RED`

- #800000 `LV_COLOR_MAROON`
- #00ff00 `LV_COLOR_LIME`
- #008000 `LV_COLOR_GREEN`
- #808000 `LV_COLOR_OLIVE`
- #0000ff `LV_COLOR_BLUE`
- #000080 `LV_COLOR_NAVY`
- #008080 `LV_COLOR_TEAL`
- #00ffff `LV_COLOR_CYAN`
- #00ffff `LV_COLOR_AQUA`
- #800080 `LV_COLOR_PURPLE`
- #ff00ff `LV_COLOR_MAGENTA`
- #ffa500 `LV_COLOR_ORANGE`
- #ffff00 `LV_COLOR_YELLOW`

以及 `LV_COLOR_WHITE` （全白）。

5.9API

5.9.1 显示

枚举

enum **lv_scr_load_anim_t**

值:

enumerator **LV_SCR_LOAD_ANIM_NONE**

enumerator **LV_SCR_LOAD_ANIM_OVER_LEFT**

enumerator **LV_SCR_LOAD_ANIM_OVER_RIGHT**

enumerator **LV_SCR_LOAD_ANIM_OVER_TOP**

enumerator **LV_SCR_LOAD_ANIM_OVER_BOTTOM**

enumerator **LV_SCR_LOAD_ANIM_MOVE_LEFT**

enumerator **LV_SCR_LOAD_ANIM_MOVE_RIGHT**

enumerator **LV_SCR_LOAD_ANIM_MOVE_TOP**

enumerator **LV_SCR_LOAD_ANIM_MOVE_BOTTOM**

enumerator **LV_SCR_LOAD_ANIM_FADE_ON**

功能

---

lv_obj_t * lv_disp_get_scr_act（lv_disp_t * *disp* ）

用指针返回活动屏幕

返回

指向活动屏幕对象的指针（由“ lv_scr_load（）”加载）

参数

`disp`：显示应该获得哪个活动屏幕的指针。（使用默认屏幕为 NULL）

---

lv_obj_t * lv_disp_get_scr_prev（lv_disp_t * *disp* ）

用指针返回上一屏幕。仅在屏幕转换期间使用。

返回

指向前一个屏幕对象的指针；如果现在不使用，则为 NULL

参数

`disp`：显示应该获得哪个先前屏幕的指针。（使用默认屏幕为 NULL）

---

void lv_disp_load_scr（lv_obj_t * *scr* ）

激活屏幕

参数

`scr`：指向屏幕的指针

---

lv_obj_t * lv_disp_get_layer_top（lv_disp_t * *disp* ）

返回顶层。（在每个屏幕上相同，并且在正常屏幕层之上）

返回

指向顶层对象的指针（透明屏幕大小 lv_obj）

参数

`disp`：显示应该获取顶层的指针。（使用默认屏幕为 NULL）

---

lv_obj_t * lv_disp_get_layer_sys（lv_disp_t * *disp* ）

随系统返回。层。（在每个屏幕上都相同，并且在正常屏幕和顶层之上）

返回

指向 sys 层对象的指针（透明屏幕大小 lv_obj）

参数

`disp`：显示哪个 sys 的指针。层应该得到。（使用默认屏幕为 NULL）

---

void lv_disp_assign_screen（lv_disp_t * *disp*，lv_obj_t * *scr* ）

将屏幕分配给显示器。

参数

`disp`：指向分配屏幕的显示器的指针

`scr`：指向要分配的屏幕对象的指针

void lv_disp_set_bg_color（lv_disp_t * *disp*，lv_color_t *color*）

设置显示器的背景色

参数

`disp`：指向显示器的指针

`color`：背景色

void lv_disp_set_bg_image（lv_disp_t * *disp*，const void* *img_src* ）

设置显示器的背景图像

参数

`disp`：指向显示器的指针

`img_src`：文件的 path 或指向 `lv_img_dsc_t` 变量的指针

void lv_disp_set_bg_opa（lv_disp_t * *disp*，lv_opa_t *opa* ）

背景的不透明度

参数

`disp`：指向显示器的指针

`opa`：不透明度（0..255）

void lv_scr_load_anim（lv_obj_t * *scr*，lv_scr_load_anim_t *anim_type*，uint32_t *time*，uint32_t *delay*，bool *auto_del* ）

切换动画画面

参数

`scr`：指向要加载的新屏幕的指针

`anim_type`：的动画类型 `lv_scr_load_anim_t`。例如 `LV_SCR_LOAD_ANIM_MOVE_LEFT`

`time`：动画时间

`delay`：转换前的延迟

`auto_del`：true：自动删除旧屏幕

uint32_t lv_disp_get_inactive_time（const lv_disp_t * *disp* ）

获取自显示器上一次用户活动（例如点击）以来经过的时间

返回

自上一次活动以来经过的滴答声（毫秒）

参数

`disp`：指向显示器的指针（NULL 表示总体上处于最小静止状态）

void lv_disp_trig_activity（lv_disp_t * *disp* ）

在显示屏上手动触发活动

参数

`disp`：指向显示的指针（使用默认显示为 NULL）

void lv_disp_clean_dcache（lv_disp_t * *disp* ）

清除与显示相关的所有 CPU 缓存。

参数

`disp`：指向显示的指针（使用默认显示为 NULL）

lv_task_t * _lv_disp_get_refr_task（lv_disp_t * *disp* ）

获取指向屏幕刷新任务的指针，以使用 `lv_task_...` 功能修改其参数。

返回

指向显示刷新任务的指针。（错误时为 NULL）

参数

`disp`：指向显示器的指针

lv_obj_t * lv_scr_act（ void ）

获取默认显示的活动屏幕

返回

指向活动屏幕的指针

lv_obj_t * lv_layer_top（ void ）

获取默认显示的顶层

返回

指向顶层的指针

lv_obj_t * lv_layer_sys（ void ）

获取默认显示的活动屏幕

返回

指向 sys 层的指针

void lv_scr_load（lv_obj_t * *scr* ）

lv_coord_t lv_dpx（ lv_coord_t *n* ）

5.9.2 色彩

typedef

*typedef*uint8_t lv_color_int_t

*typedef*lv_color1_t lv_color_t

枚举

enum [anonymous]

不透明度百分比。

值:

*enumerator*LV_OPA_TRANSP = 0

*enumerator*LV_OPA_0 = 0

*enumerator*LV_OPA_10 = 25

*enumerator*LV_OPA_20 = 51

*enumerator*LV_OPA_30 = 76

*enumerator*LV_OPA_40 = 102

*enumerator*LV_OPA_50 = 127

*enumerator*LV_OPA_60 = 153

*enumerator*LV_OPA_70 = 178

*enumerator*LV_OPA_80 = 204

*enumerator*LV_OPA_90 = 229

*enumerator*LV_OPA_100 = 255

*enumerator*LV_OPA_COVER = 255

功能

uint8_t lv_color_to1（lv_color_t *color*）

uint8_t lv_color_to8（lv_color_t *color*）

uint16_t lv_color_to16（lv_color_t *color*）

uint32_t lv_color_to32（lv_color_t *color*）

uint8_t lv_color_brightness（lv_color_t *color*）

获取颜色的亮度

返回

亮度[0..255]

参数

color：颜色

lv_color_t lv_color_make（ uint8_t *r*，uint8_t *g*，uint8_t *b* ）

lv_color_t lv_color_hex（ uint32_t *c* ）

lv_color_t lv_color_hex3（ uint32_t *c* ）

lv_color_t lv_color_lighten（lv_color_t *c*， lv_opa_t *lvl* ）

lv_color_t lv_color_darken（lv_color_t *c*， lv_opa_t *lvl* ）

lv_color_t lv_color_hsv_to_rgb（ uint16_t *h*， uint8_t *s*， uint8_t *v* ）

将 HSV 颜色转换为 RGB

返回

RGB 中指定的 RGB 颜色（深度为 LV_COLOR_DEPTH）

参数

h：色相[0..359]

s：饱和度[0..100]

v：值[0..100]

lv_color_hsv_t lv_color_rgb_to_hsv（ uint8_t *r8*， uint8_t *g8*， uint8_t *b8* ）

将 32 位 RGB 颜色转换为 HSV

返回

HSV 中给定的 RGB 颜色

参数

r8：8 位红色

g8：8 位绿色

b8：8 位蓝色

lv_color_hsv_t lv_color_to_hsv（lv_color_t *color*）

将颜色转换为 HSV

返回

HSV 中给定的颜色

参数

color：颜色

union lv_color1_t

公众成员

uint8_t blue

uint8_t green

uint8_t red

union lv_color1_t :: [anonymous]ch

uint8_t full

union lv_color8_t

公众成员

uint8_t blue

uint8_t green

uint8_t red

struct lv_color8_t :: [anonymous] ch

uint8_t full

union lv_color16_t

公众成员

uint16_t blue

uint16_t green

uint16_t red

uint16_t green_h

uint16_t green_l

struct lv_color16_t :: [anonymous] ch

uint16_t full

union lv_color32_t

公众成员

uint8_t blue

uint8_t green

uint8_t red

uint8_t alpha

struct lv_color32_t :: [anonymous] ch

uint32_t full

struct lv_color_hsv_t

公众成员

uint16_t h

uint8_t s

uint8_t v

5.10 字型

在 LVGL 中，字体是位图的集合以及渲染字母（字形）图像所需的其他信息。字体存储在 `lv_font_t` 变量中，可以在样式的 *text_font* 字段中设置。例如：

```
lv_style_set_text_font(&my_style, LV_STATE_DEFAULT, &lv_font_montserrat_28);  /*Set a larger font*/
```

字体具有 **bpp（每像素位数）**属性。它显示了多少位用于描述字体中的像素。为像素存储的值确定像素的不透明度。这样，如果 *bpp* 较高，则字母的边缘可以更平滑。可能的 *bpp* 值是 1、2、4 和 8（值越高表示质量越好）。

该 *BPP* 也影响所需的存储空间来存储字体。例如，*bpp = 4* 使字体比 *bpp = 1* 大近 4 倍。

5.11 Unicode 支持

LVGL 支持 **UTF-8** 编码的 Unicode 字符。需要配置您的编辑器，以将您的代码/文本保存为 UTF-8（通常是默认值），并确保在 *lv_conf.h* 中将 `LV_TXT_ENC` 其设置为。（这是默认值） `LV_TXT_ENC_UTF8`

测试一下试试:

```
lv_obj_t * label1 = lv_label_create(lv_scr_act(), NULL);
lv_label_set_text(label1, LV_SYMBOL_OK);
```

如果一切正常，则应显示一个✓字符。

5.12 内置字体

有几种不同大小的内置字体，可以通过 *LV_FONT_*在 lv_conf.h 中启用。...定义：

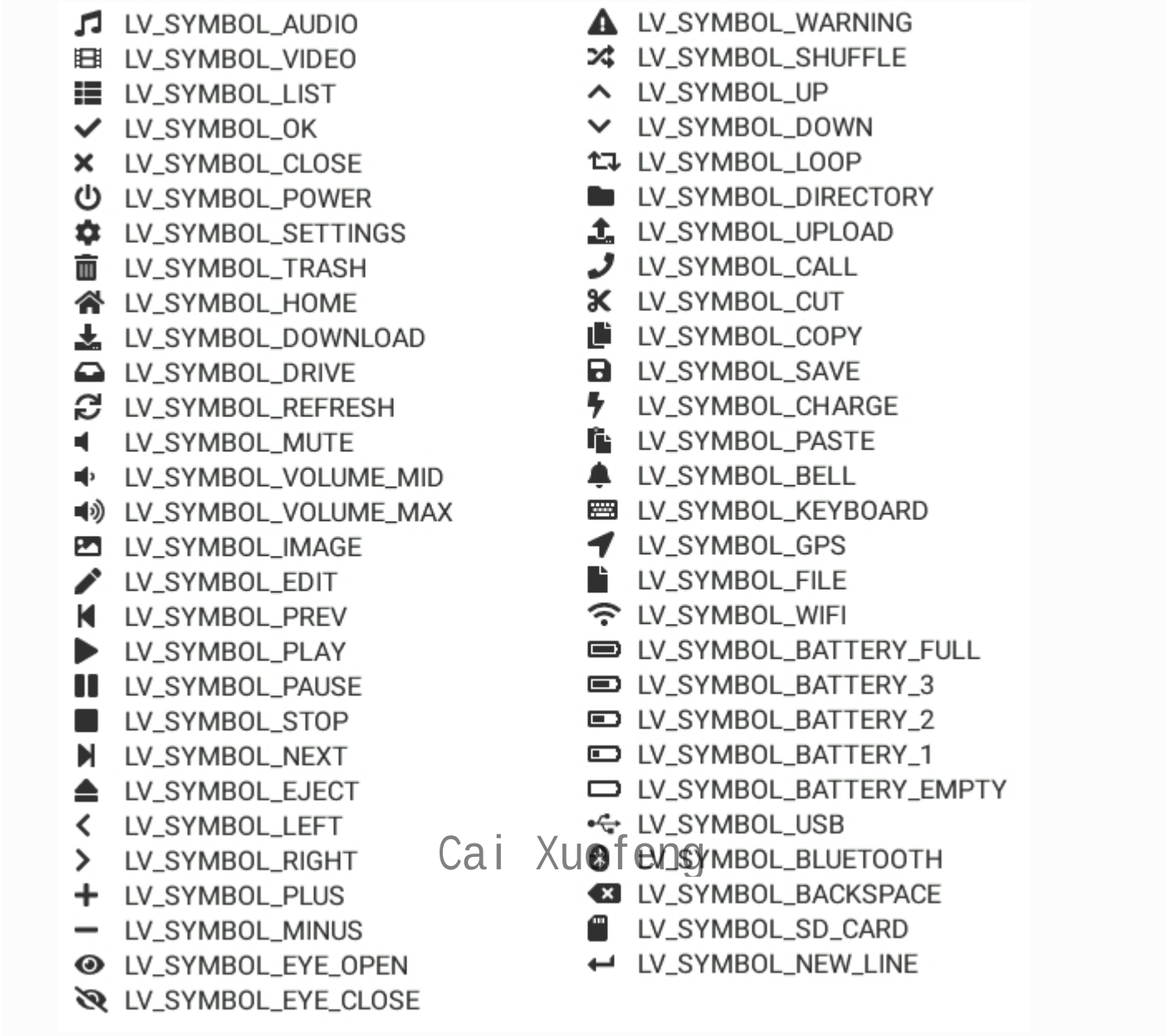
- `LV_FONT_MONTSERRAT_12` 12 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_14` 14 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_16` 16 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_18` 18 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_20` 20 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_22` 22 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_24` 24 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_26` 26 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_28` 28 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_30` 30 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_32` 32 px ASCII +内置符号

- `LV_FONT_MONTSERRAT_34` 34 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_36` 36 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_38` 38 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_40` 40 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_42` 42 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_44` 44 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_46` 46 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_48` 48 px ASCII +内置符号
- `LV_FONT_MONTSERRAT_12_SUBPX` 具有子像素渲染的 12 像素字体
- `LV_FONT_MONTSERRAT_28_COMPRESSED` 28 px 压缩字体和 3 bpp
- `LV_FONT_DEJAVU_16_PERSIAN_HEBREW` 16 px 希伯来语，阿拉伯语，Perisan 字母及其所有形式
- `LV_FONT_SIMSUN_16_CJK` 16 像素 1000 个最常见的 CJK 部首
- `LV_FONT_UNSCII_8` 8 像素像素完美字体

内置字体是**全局变量**，其名称类似于 `lv_font_montserrat_16` 16 px 高字体。要以某种样式使用它们，只需添加一个指向如上所示的字体变量的指针。

内置字体的 *bpp = 4*，包含 ASCII 字符并使用 Montserrat 字体。

除了 ASCII 范围，以下符号也从 FontAwesome 字体添加到内置字体中。



这些符号可以用作:

```
lv_label_set_text(my_label, LV_SYMBOL_OK);
```

或与字符串一起使用:

```
lv_label_set_text(my_label, LV_SYMBOL_OK "Apply");
```

或更多符号:

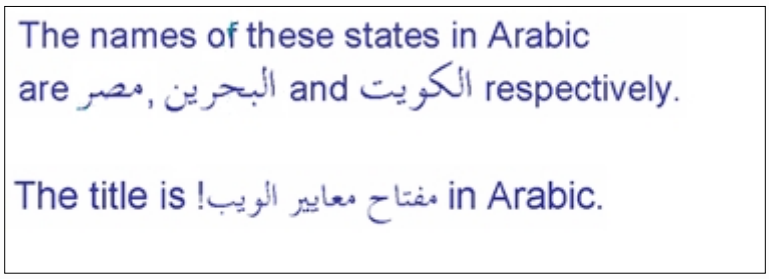
```
lv_label_set_text(my_label, LV_SYMBOL_OK LV_SYMBOL_WIFI LV_SYMBOL_PLAY);
```

5.13 特殊功能

5.13.1 双向支持

大多数语言使用从左到右（简称 LTR）的书写方向，但是某些语言（例如希伯来语，波斯语或阿拉伯语）使用从右到左（简称 RTL）的书写方向。

LVGL 不但支持 RTL 文本，而且还支持混合（又名双向，BiDi）文本渲染。一些例子:



可以 `LV_USE_BIDI` 在 *lv_conf.h* 中启用 BiDi 支持

所有文本都有一个基本方向（LTR 或 RTL），该方向确定一些渲染规则和文本的默认对齐方式（左或右）。但是，在 LVGL 中，基本方向不仅适用于标签。这是可以为每个对象设置的常规属性。如果未设置，则它将从父级继承。因此，设置屏幕的基本方向就足够了，每个对象都会继承它。

屏幕的默认基本方向可以 `LV_BIDI_BASE_DIR_DEF` 在 *lv_conf.h* 中设置，其他对象将从其父对象继承基本方向。

要设置对象的基本方向，请使用。可能的基本方向是: `lv_obj_set_base_dir(obj, base_dir)`

- `LV_BIDI_DIR_LTR`：从左到右基本方向
- `LV_BIDI_DIR_RTL`：从右到左基本方向
- `LV_BIDI_DIR_AUTO`：自动检测基准方向
- `LV_BIDI_DIR_INHERIT`：从父级继承基本方向（非屏幕对象的默认方向）

此列表总结了 RTL 基本方向对对象的影响:

- 默认情况下在右侧创建对象
- `lv_tabview`：从右到左显示标签
- `lv_checkbox`：显示右侧的框
- `lv_btnmatrix`：从右到左显示按钮
- `lv_list`：在右侧显示图标
- `lv_dropdown`：将选项向右对齐
- 在文本中 `lv_table`，`lv_btnmatrix`，`lv_keyboard`，`lv_tabview`，`lv_dropdown`，`lv_roller`，以正确显示的“BiDi 的处理”

5.13.2 阿拉伯语和波斯语支持

显示阿拉伯和波斯字符有一些特殊规则：字符的*形式*取决于它们在文本中的位置。如果隔离，开始，中间或结束位置，则需要使用同一字母的不同形式。除了这些连接规则外，还应考虑其他规则。

如果 `LV_USE_ARABIC_PERSIAN_CHARS` 已启用，LVGL 支持应用这些规则。

但是，存在一些限制：

- 仅支持显示文本（例如在标签上），文本输入（例如文本区域）不支持此功能
- 静态文本（即 const）不会被处理。例如，由设置的文本将被 `lv_label_set_text()` "阿拉伯语处理"，但 `lv_lable_set_text_static()` 不会。
- 文本获取函数（例如 `lv_label_get_text()`）将返回处理后的文本。

5.13.3 亚像素渲染

亚像素渲染意味着通过在红色，绿色和蓝色通道而不是像素级别上渲染将水平分辨率提高三倍。它利用了每个像素的物理颜色通道的位置。这样可以产生更高质量的字母抗锯齿。在这里学习更多。

亚像素渲染需要生成具有特殊设置的字体：

- 在网络转换器勾选 `Subpixel` 框
- 在命令行工具中使用 `--lcd` 标志。请注意，生成的字体大约需要 3 倍的内存。

仅当像素的颜色通道具有水平布局时，子像素渲染才有效。也就是说，R，G，B 通道彼此相邻而不位于彼此之上。颜色通道的顺序也需要与库设置匹配。默认情况下，LVGL 采用 `RGB` 顺序，但是可以通过在 *lv_conf.h* 中进行设置来交换*顺序*。 `LV_SUBPX_BGR 1`

5.13.4 压缩字体

字体的位图可以通过以下方式压缩

- 勾选 `Compressed` 在线转换器中的复选框
- 不将 `--no-compress` 标志传递给脱机转换器（默认情况下应用压缩）

较大的字体和较高的 bpp 压缩更有效。但是，渲染压缩字体的速度要慢 30%。因此，建议仅压缩用户界面的最大字体，因为

- 他们需要最多的记忆
- 他们可以更好地压缩
- 并且它们的使用频率可能不如中等大小的字体。（因此性能成本较小）

5.13.5 添加新字体

有几种方法可以向项目中添加新字体：

1. 最简单的方法是使用 Online 字体转换器。只需设置参数，单击"*转换*"按钮，将字体复制到您的项目中并使用它。**请务必仔细阅读该站点上提供的步骤，否则转换时会出错。**
2. 使用脱机字体转换器。（需要安装 Node.js）

3. 如果要创建类似内置字体（Roboto 字体和符号）但大小和/或范围不同的内容，则可以使用文件夹中的 `built_in_font_gen.py` 脚本 `lvgl/scripts/built_in_font`。（需要 Python 并 `lv_font_conv` 进行安装）

要在文件中声明字体，请使用 `LV_FONT_DECLARE(my_font_name)`。

为了使全局可用（如内置字体）的字体，将其添加到 `LV_FONT_CUSTOM_DECLARE` 在 *lv_conf.h*。

5.13.6 添加新符号

内置符号是从 FontAwesome 字体创建的。

1. 在 https://fontawesome.com 上搜索符号。例如 USB 符号。复制 `0xf287` 本例中的 Unicode ID 。
2. 打开在线字体转换器。添加添加 FontAwesome.woff。。
3. 设置名称，大小，BPP 等参数。您将使用此名称在代码中声明和使用字体。
4. 将符号的 Unicode ID 添加到范围字段。例如，USB 符号。可以列举更多符号。 `0xf287,`
5. 转换字体并将其复制到您的项目。确保编译字体的.c 文件。
6. 使用或简单声明字体。 `extern lv_font_t my_font_name;LV_FONT_DECLARE(my_font_name);`

使用符号

1. 将 Unicode 值转换为 UTF8。您可以在此站点上进行操作。对于 `0xf287` 该六角 *UTF-8 字节*是。 `EF 8A 87`
2. `define` 根据 UTF8 值创建一个： `#define MY_USB_SYMBOL "\xEF\x8A\x87"`
3. 创建一个标签并设置文本。例如。 `lv_label_set_text(label, MY_USB_SYMBOL)`

注意- 使用属性中定义的字体搜索此符号。要使用该符号，您可能需要更改它。例如 `lv_label_set_text(label, MY_USB_SYMBOL)style.text.font style.text.font = my_font_name`

5.14 在运行时加载字体

`lv_font_load` 可用于从文件加载字体。要加载的字体需要具有特殊的二进制格式。（不是 TTF 或 WOFF）。使用 lv_font_conv 与选项生成 LVGL 兼容的字体文件。 `--format bin`

请注意，要加载字体，需要启用 LVGL 的文件系统，并需要添加驱动程序。
例

```
lv_font_t * my_font;
my_font = lv_font_load(X/path/to/my_font.bin);
```

```
    /*Use the font*/

    /*Free the font if not required anymore*/
    lv_font_free(my_font);
```

5.14.1 添加新的字体引擎

LVGL 的字体界面设计得非常灵活。您不需要使用 LVGL 的内部字体引擎，但是可以添加自己的字体。例如，使用 FreeType 从 TTF 字体实时渲染字形，或者使用外部闪存存储字体的位图，并在库需要它们时读取它们。

可以在 lv_freetype 信息库中找到使用 FreeType 的传统。

为此，`lv_font_t` 需要创建一个自定义变量：

```
    /*Describe the properties of a font*/
    lv_font_t my_font;
    my_font.get_glyph_dsc = my_get_glyph_dsc_cb;       /*Set a callback to get info about gylphs*/
    my_font.get_glyph_bitmap = my_get_glyph_bitmap_cb;  /*Set a callback to get bitmap of a glyp*/
    my_font.line_height = height;                      /*The real line height where any text fits*/
    my_font.base_line = base_line;                     /*Base line measured from the top of line_height*/
    my_font.dsc = something_required;                  /*Store any implementation specific data here*/
    my_font.user_data = user_data;                     /*Optionally some extra user data*/

    ...

    /* Get info about glyph of `unicode_letter` in `font` font.
     * Store the result in `dsc_out`.
     * The next letter (`unicode_letter_next`) might be used to calculate the width required by this glyph
(kerning)
     */
    bool my_get_glyph_dsc_cb(const lv_font_t * font, lv_font_glyph_dsc_t * dsc_out, uint32_t
unicode_letter, uint32_t unicode_letter_next)
    {
        /*Your code here*/

        /* Store the result.
         * For example ...
         */
        dsc_out->adv_w = 12;       /*Horizontal space required by the glyph in [px]*/
        dsc_out->box_h = 8;        /*Height of the bitmap in [px]*/
        dsc_out->box_w = 6;        /*Width of the bitmap in [px]*/
        dsc_out->ofs_x = 0;        /*X offset of the bitmap in [pf]*/
        dsc_out->ofs_y = 3;        /*Y offset of the bitmap measured from the as line*/
        dsc_out->bpp   = 2;        /*Bits per pixel: 1/2/4/8*/

        return true;              /*true: glyph found; false: glyph was not found*/
    }

    /* Get the bitmap of `unicode_letter` from `font`. */
```

```
const uint8_t * my_get_glyph_bitmap_cb(const lv_font_t * font, uint32_t unicode_letter)
{
    /* Your code here */

    /* The bitmap should be a continuous bitstream where
     * each pixel is represented by `bpp` bits */

    return bitmap;    /*Or NULL if not found*/
}
```

第六章 图片

图像可以是存储位图本身和一些元数据的文件或变量。

6.1 储存图片

您可以将图像存储在两个位置

- 作为内部存储器（RAM 或 ROM）中的变量
- 作为文件

6.2 变数

内部存储在变量中的图像主要由 `lv_img_dsc_t` 具有以下字段的 struct 组成：

- 标头
  - *cf* 颜色格式。见下文
  - *w* 宽度（以像素为单位）（<= 2048）
  - *h* 高度（以像素为单位）（<= 2048）
  - *始终为零* 3 位需要始终为零
  - *保留*保留以备将来使用
- 指向存储图像本身的数组的数据指针
- **data_size** 的长度（`data` 以字节为单位）

这些通常作为 C 文件存储在项目中。它们像其他任何常量数据一样链接到生成的可执行文件中。

6.3 档案

要处理文件，您需要将*驱动器*添加到 LVGL。简而言之，*驱动器*是在 LVGL 中注册的用于文件操作的功能的集合（*打开*，*读取*，*关闭*等）。您可以向标准文件系统（SD 卡上的 FAT32）添加接口，也可以创建简单的文件系统以从 SPI 闪存读取数据。在每种情况下，*驱动器*都只是一种将数据读取和/或写入内存的抽象。请参阅文件系统部分以了解更多信息。

存储为文件的图像未链接到生成的可执行文件中，并且在绘制之前必须将其读取到 RAM 中。结果，它们不像可变图像那样对资源友好。但是，它们更容易替换，而无需重新编译主程序。

6.4 色彩格式

支持多种内置颜色格式：

- **LV_IMG_CF_TRUE_COLOR** 只需存储 RGB 颜色（使用 LVGL 配置的任何颜色深度）。
- **LV_IMG_CF_TRUE_COLOR_ALPHA** 像，`LV_IMG_CF_TRUE_COLOR` 但它还会为每个像素添加一个 alpha（透明）字节。

- **LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED** 类似于，`LV_IMG_CF_TRUE_COLOR` 但如果像素具有 `LV_COLOR_TRANSP`（在 *lv_conf.h* 中设置）颜色，则该像素将是透明的。
- **LV_IMG_CF_INDEXED_1 / 2/4 / 8BIT** 使用 **2、4、16** 或 256 色调色板，并以 1、2、4 或 8 位存储每个像素。

LV_IMG_CF_ALPHA_1 / 2/4 / 8BIT 仅将 Alpha 值存储在 1、2、4 或 8 位上。像素采用 `style.image.color` 和设置的不透明度的颜色。源图像必须是 Alpha 通道。这对于类似于字体的位图是理想的（整个图像是一种颜色，但是您希望能够更改它）。

`LV_IMG_CF_TRUE_COLOR` 图像的字节按以下顺序存储。

对于 32 位色深：

- 字节 0：蓝色
- 字节 1：绿色
- 字节 2：红色
- 字节 3：Alpha

对于 16 位色深：

- 字节 0：绿色 3 低位，蓝色 5 位
- 字节 1：红色 5 位，绿色 3 位
- 字节 2：Alpha 字节（仅适用于 LV_IMG_CF_TRUE_COLOR_ALPHA）

对于 8 位色深：

- 字节 0：红色 3 位，绿色 3 位，蓝色 2 位
- 字节 2：Alpha 字节（仅适用于 LV_IMG_CF_TRUE_COLOR_ALPHA）

您可以将图像以 *Raw* 格式存储，以表示它不是内置的颜色格式，并且需要使用外部 Imagedecoder 来解码图像。

- **LV_IMG_CF_RAW** 表示基本的原始图像（例如 PNG 或 JPG 图像）。
- **LV_IMG_CF_RAW_ALPHA** 表示图像具有 Alpha，并且为每个像素添加了 Alpha 字节。
- **LV_IMG_CF_RAW_CHROME_KEYED** 表示图像已 `LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED` 如上所述进行 chrome 键锁定。

6.5 添加和使用图像

您可以通过两种方式将图像添加到 LVGL：

- 使用在线转换器
- 手动创建图像

6.5.1 在线转换器

在线图像转换器可在这里找到：https：//lvgl.io/tools/imageconverter

通过在线转换器将图像添加到 LVGL 很容易。

1. 您需要首先选择 *BMP*，*PNG* 或 *JPG* 图像。
2. 给图像起一个将在 LVGL 中使用的名称。
3. 选择颜色格式。
4. 选择所需的图像类型。选择二进制文件将生成一个 `.bin` 文件，该文件必须单独存储并使用文件支持进行读取。选择一个变量将生成一个标准的 C 文件，该文件可以链接到您的项目中。
5. 点击*转换*按钮。转换完成后，您的浏览器将自动下载结果文件。

在转换器 C 数组（变量），对于所有的颜色深度（1,8，16 或 32）位图包含在 C 文件，但只有颜色深度匹配 `LV_COLOR_DEPTH` 在 *lv_conf.h* 实际上将被链接到所得到的可执行文件。

如果是二进制文件，则需要指定所需的颜色格式：

- RGB332 用于 8 位色深
- RGB565 用于 16 位色深
- RGB565 交换为 16 位颜色深度（交换了两个字节）
- RGB888 用于 32 位色深

6.5.2 手动创建图像

如果要在运行时生成图像，则可以制作图像变量以使用 LVGL 显示它。例如：

```
uint8_t my_img_data[] = {0x00, 0x01, 0x02, ...};

static lv_img_dsc_t my_img_dsc = {
    .header.always_zero = 0,
    .header.w = 80,
    .header.h = 60,
    .data_size = 80 * 60 * LV_COLOR_DEPTH / 8,
    .header.cf = LV_IMG_CF_TRUE_COLOR,          /*Set the color format*/
    .data = my_img_data,
};
```


如果是彩色格式，则 `LV_IMG_CF_TRUE_COLOR_ALPHA` 可以设置 `data_size` 为。

`80 * 60 * LV_IMG_PX_SIZE_ALPHA_BYTE`

在运行时创建和显示图像的另一个（可能更简单）的选项是使用 Canvas 对象。

6.5.3 使用图片

在 LVGL 中使用图像的最简单方法是使用 lv_img 对象显示它：

```
lv_obj_t * icon = lv_img_create(lv_scr_act(), NULL);

/*From variable*/
lv_img_set_src(icon, &my_icon_dsc);
```

```
/*From file*/
lv_img_set_src(icon, "S:my_icon.bin");
```

如果图像是使用在线转换器转换的，则应使用 `LV_IMG_DECLARE(my_icon_dsc)` 该图像在文件中声明要使用的位置。

6.6 图像 decoder

如您在“ 颜色格式”部分中所见，LVGL 支持多种内置图像格式。在许多情况下，这些都是您所需要的。LVGL 不直接支持通用图像格式，例如 PNG 或 JPG。

要处理非内置图像格式，您需要使用外部库，并通过*图像 decoder* 接口将它们附加到 LVGL 。

图像 decoder 包含 4 个回调：

- **info** 获取有关图像的一些基本信息（宽度，高度和颜色格式）。
- **打开**打开图像：存储已解码图像或将其设置 `NULL` 为指示可以逐行读取图像。
- 如果*打开*未完全打开图像，则**读取**该函数应从给定位置提供一些解码数据（最多 1 行）。
- **关闭**关闭打开的图像，释放分配的资源。

您可以添加任意数量的图像 decoder。当需要绘制图像时，该库将尝试所有已注册的图像 decoder，直到找到可以打开该图像的 decoder，即知道该格式。

的 `LV_IMG_CF_TRUE_COLOR_...`， `LV_IMG_INDEXED_...` 和 `LV_IMG_ALPHA_...` 格式（基本上，所有的非 `RAW` 格式）由内置 decoder 理解。

6.7 自定义图像格式

创建自定义图像的最简单方法是使用在线图像转换器和设置 `Raw`，或格式。它只会占用您上传的二进制文件的每个字节，并将其写为图像“位图”。然后，您需要连接一个图像 decoder，它将解析该位图并生成真实的可渲染位图。`Raw with alphaRaw with chrome keyed`

header.cf 将是 `LV_IMG_CF_RAW`， `LV_IMG_CF_RAW_ALPHA` 或 `LV_IMG_CF_RAW_CHROME_KEYED` 相应地。您应该根据需要选择正确的格式：完全不透明的图像，使用 Alpha 通道或使用色度键。

解码后，*原始*格式被库视为*真彩色*。换句话说，图像 decoder 必须根据[＃color-formats]（颜色格式）部分中描述的格式将 *Raw* 图像解码为 *True color*。

如果要创建自定义图像，则应使用 `LV_IMG_CF_USER_ENCODED_0..7` 颜色格式。然而，**库**可以绘制图像仅在*真彩色*格式（或*原始*，但最终它应该在*真彩色*格式）。因此， `LV_IMG_CF_USER_ENCODED_...` 库不知道这些格式，因此应从[＃color-formats]（颜色格式）部分将其解码为已知格式之一。例如，可以先将图像解码为非真彩色格式 `LV_IMG_INDEXED_4BITS`，然后调用内置的 decoder 函数将其转换为*真彩色*。

对于*用户编码*格式， `dsc->header.cf` 应根据新格式更改打开功能（）中的颜色格式。

6.8 注册图像 decoder

这是使 LVGL 与 PNG 图像配合使用的示例。

首先，您需要创建一个新的图像 decoder 并设置一些功能来打开/关闭 PNG 文件。它看起来应该像这样:

```
/*Create a new decoder and register functions */
lv_img_decoder_t * dec = lv_img_decoder_create();
lv_img_decoder_set_info_cb(dec, decoder_info);
lv_img_decoder_set_open_cb(dec, decoder_open);
lv_img_decoder_set_close_cb(dec, decoder_close);

/**
 * Get info about a PNG image
 * @param decoder pointer to the decoder where this function belongs
 * @param src can be file name or pointer to a C array
 * @param header store the info here
 * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
 */
static lv_res_t decoder_info(lv_img_decoder_t * decoder, const void * src, lv_img_header_t * header)
{
  /*Check whether the type `src` is known by the decoder*/
  if(is_png(src) == false) return LV_RES_INV;

  /* Read the PNG header and find `width` and `height` */
  ...

  header->cf = LV_IMG_CF_RAW_ALPHA;
  header->w = width;
  header->h = height;
}

/**
 * Open a PNG image and return the decided image
 * @param decoder pointer to the decoder where this function belongs
 * @param dsc pointer to a descriptor which describes this decoding session
 * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
 */
static lv_res_t decoder_open(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{

  /*Check whether the type `src` is known by the decoder*/
  if(is_png(src) == false) return LV_RES_INV;

  /*Decode and store the image. If `dsc->img_data` is `NULL`, the `read_line` function will be called to get the image data line-by-line*/
  dsc->img_data = my_png_decoder(src);

  /*Change the color format if required. For PNG usually 'Raw' is fine*/
  dsc->header.cf = LV_IMG_CF_...
```

```
    /*Call a built in decoder function if required. It's not required if`my_png_decoder` opened the
image in true color format.*/
    lv_res_t res = lv_img_decoder_built_in_open(decoder, dsc);

    return res;
}

/**
 * Decode `len` pixels starting from the given `x`, `y` coordinates and store them in `buf`.
 * Required only if the "open" function can't open the whole decoded pixel array. (dsc->img_data ==
NULL)
 * @param decoder pointer to the decoder the function associated with
 * @param dsc pointer to decoder descriptor
 * @param x start x coordinate
 * @param y start y coordinate
 * @param len number of pixels to decode
 * @param buf a buffer to store the decoded pixels
 * @return LV_RES_OK: ok; LV_RES_INV: failed
 */
lv_res_t decoder_built_in_read_line(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc, lv_coord_t
x,
                                    lv_coord_t y, lv_coord_t len, uint8_t * buf)
{
    /*With PNG it's usually not required*/

    /*Copy `len` pixels from `x` and `y` coordinates in True color format to `buf` */

}

/**
 * Free the allocated resources
 * @param decoder pointer to the decoder where this function belongs
 * @param dsc pointer to a descriptor which describes this decoding session
 */
static void decoder_close(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{
    /*Free all allocated data*/

    /*Call the built-in close function if the built-in open/read_line was used*/
    lv_img_decoder_built_in_close(decoder, dsc);

}
```


因此，总而言之：

- 在中 `decoder_info`，您应该收集有关图像的一些基本信息并将其存储在中 `header`。
- 在中 `decoder_open`，您应尝试打开指向的图像源 `dsc->src`。其类型已经在中。如果 decoder 不支持此格式/类型，请返回。但是，如果可以打开图像，则应在中设置指向已解码*真*

*彩色*图像的指针。如果格式已知，但是您不想在图像（例如没有内存）设置为调用以获取像素时进行解码。

`dsc->src_type == LV_IMG_SRC_FILE/VARIABLELV_RES_INVdsc->img_datadsc->img_data = NULLread_line`

- 在 `decoder_close` 您应该释放所有分配的资源。
- `decoder_read` 是可选的。解码整个图像需要额外的内存和一些计算开销。但是，如果可以解码图像的一行而不解码整个图像，则可以节省内存和时间。为了表明这一点，应该使用*行读取*功能，在打开功能中进行设置。`dsc->img_data = NULL`

6.9 手动使用图像 decoder

如果您尝试绘制原始图像（即使用 `lv_img` 对象），LVGL 将自动使用注册的图像 decoder，但是您也可以手动使用它们。创建一个 `lv_img_decoder_dsc_t` 变量来描述解码会话并调用 `lv_img_decoder_open()`。

```
lv_res_t res;
lv_img_decoder_dsc_t dsc;
res = lv_img_decoder_open(&dsc, &my_img_dsc, LV_COLOR_WHITE);

if(res == LV_RES_OK) {
  /*Do something with `dsc->img_data`*/
  lv_img_decoder_close(&dsc);
}
```


6.10 图像缓存

有时打开图像需要花费很多时间。连续解码 PNG 图像或从速度较慢的外部存储器加载图像会效率低下，并且对用户体验有害。

因此，LVGL 缓存给定数量的图像。缓存意味着一些图像将保持打开状态，因此 LVGL 可以快速访问它们，`dsc->img_data` 而无需再次对其进行解码。

当然，缓存图像会占用大量资源，因为它使用更多的 RAM（用于存储解码的图像）。LVGL 尝试尽可能地优化流程（请参见下文），但是您仍然需要评估这是否对您的平台有利。如果您有一个深层嵌入的目标，可以从相对较快的存储介质中解码小图像，则不建议使用图像缓存。

6.10.1 快取大小

高速缓存条目的数量可以 `LV_IMG_CACHE_DEF_SIZE` 在 *lv_conf.h* 中定义。默认值为 1，因此只有最近使用的图像将保持打开状态。

缓存的大小可以在运行时通过更改 `lv_img_cache_set_size(entry_num)`。

6.10.2 图片价值

当使用的图像多于缓存条目时，LVGL 无法缓存所有图像。而是，库将关闭其中一个缓存的图像（以释放空间）。

为了决定关闭哪个图像，LVGL 使用它先前对打开图像所花费的时间进行的测量。保存打开速度较慢的图像的高速缓存条目被认为更有价值，并尽可能长时间地保留在高速缓存中。

如果要或需要覆盖 LVGL 的测量值，可以在 decoder 打开功能中手动设置*打开*值的*时间*，以提供更高或更低的值。（将其保留以让 LVGL 对其进行设置。）`dsc->time_to_open = time_ms`

每个缓存条目都有一个*“生命”*值。每次通过缓存打开图像时，所有条目的*寿命*都会减少，从而使它们更旧。当使用缓存的图像时，其*寿命*会随着*打开*值的*时间*增加而增加，使其更生动。

如果缓存中没有更多空间，则始终关闭寿命最小的条目。

6.10.3 内存使用情况

请注意，缓存的图像可能会不断消耗内存。例如，如果缓存了 3 个 PNG 图像，则它们将在打开时消耗内存。因此，用户有责任确保有足够的 RAM 来缓存，甚至同时存储最大的图像。

6.10.4 清理缓存

假设您已将 PNG 图片加载到变量中并在对象中使用。如果图像已经被缓存，然后您更改基础 PNG 文件，则需要通知 LVGL 再次缓存图像。否则，没有简单的方法可以检测到基础文件已更改并且 LVGL 仍会绘制旧图像。

`lv_img_dsc_t my_pnglv_img`

为此，请使用 lv_img_cache_invalidate_src(&my_png)。如果 NULL 将其作为参数传递，则将清除整个缓存。

API

图像 decoder

typedef

***typedef*uint8_t lv_img_src_t**

***typedef*lv_res_t（* lv_img_decoder_info_f_t）（*struct* lv_img_decoder *decoder，*const*void* src，lv_img_header_t * header ）**

从图像获取信息并将其存储在 `header`

返回

LV_RES_OK：信息正确写入；LV_RES_INV：失败

参数

`src`：图像源。可以是指向 C 数组或文件名的指针（`lv_img_src_get_type` 用于确定类型）

`header`：在此处存储信息

typedef lv_res_t（* lv_img_decoder_open_f_t）（*struct* lv_img_decoder *decoder，*struct* lv_img_decoder_dsc * dsc ）

打开图像进行解码。准备它，以备日后阅读

> 参数

`decoder`：指向与 decoder 相关的功能的 decoder

`dsc`：指向 decoder 描述符的指针。`src`，`style` 已经在其中初始化。

typedef lv_res_t（* lv_img_decoder_read_line_f_t）（*struct* lv_img_decoder *decoder，*struct* lv_img_decoder_dsc * dsc，lv_coord_t x，lv_coord_t y，lv_coord_t len，uint8_t * buf ）

`len` 从给定的像素开始解码 `x`，进行 `y` 坐标处理并将其存储在中 `buf`。仅当“打开”功能无法返回整个解码像素数组时才需要。

> 返回

LV_RES_OK：好的；LV_RES_INV：失败

> 参数

`decoder`：指向与 decoder 相关的功能的 decoder

`dsc`：指向 decoder 描述符的指针

`x`：开始 x 坐标

`y`：开始 y 坐标

`len`：要解码的像素数

`buf`：用于存储解码像素的缓冲区

typedef void（* lv_img_decoder_close_f_t）（*struct* lv_img_decoder *decoder，*struct* lv_img_decoder_dsc * dsc ）

关闭待处理的解码。免费资源等

> 参数

`decoder`：指向与 decoder 相关的功能的 decoder

`dsc`：指向 decoder 描述符的指针

typedef struct lv_img_decoder lv_img_decoder_t

typedef struct lv_img_decoder_dsc lv_img_decoder_dsc_t

描述图像解码会话。存储有关解码的数据

> 枚举

枚举 [anonymous]

图片来源。

值：

***enumerator*LV_IMG_SRC_VARIABLE**

***enumerator*LV_IMG_SRC_FILE**

二进制/ C 变量

***enumerator*LV_IMG_SRC_SYMBOL**

文件系统中的文件

***enumerator*LV_IMG_SRC_UNKNOWN**

符号（lv_symbol_def.h）

功能

void_lv_img_decoder_init（ void ）

初始化图像 decoder 模块

lv_res_t lv_img_decoder_get_info（*const*char * *src*，lv_img_header_t * *标头*）

获取有关图像的信息。尝试尝试创建的图像 decoder。一旦能够获取信息，该信息将被使用。

返回

LV_RES_OK：成功；LV_RES_INV：无法获取有关图像的信息

参数

`src`：图像源。可以是 1）文件名：例如“ S：folder / img1.png”（驱动程序需要通过进行注册 `lv_fs_add_drv()`）2）变量：指向变量的指针 `lv_img_dsc_t` 3）符号：例如 `LV_SYMBOL_OK`

`header`：图像信息将存储在此处

lv_res_t lv_img_decoder_open（lv_img_decoder_dsc_t * *dsc*，*const*void* *src*，lv_color_t *color*）

打开图像。尝试尝试创建的图像 decoder。一旦能够打开 decoder 保存的图像 `dsc`

返回

LV_RES_OK：打开图像。`dsc->img_data` 并 `dsc->header` 设置。LV_RES_INV：没有注册的图像 decoder 能够打开图像。

参数

dsc：描述解码会话。只是指向 lv_img_decoder_dsc_t 变量的指针。

src：图像源。可以是 1）文件名：例如“ S：folder / img1.png”（驱动程序需要通过进行注册 lv_fs_add_drv()）2）变量：指向变量的指针 lv_img_dsc_t 3）符号：例如 LV_SYMBOL_OK

color：具有的图像颜色 LV_IMG_CF_ALPHA_...

lv_res_t lv_img_decoder_read_line（lv_img_decoder_dsc_t * *dsc*，lv_coord_t *x*，lv_coord_t *y*，lv_coord_t *len*，uint8_t * *buf*）

从打开的图像中读取一行

返回

LV_RES_OK：成功；LV_RES_INV：发生错误

参数

`dsc`：`lv_img_decoder_dsc_t`用于的指针`lv_img_decoder_open`

`x`：开始 X 坐标（从左开始）

`y`：开始 Y 坐标（从顶部开始）

`len`：要读取的像素数

`buf`：在此处存储数据

---

void lv_img_decoder_close（lv_img_decoder_dsc_t * *dsc* ）

关闭解码会话

参数

`dsc`：`lv_img_decoder_dsc_t`用于的指针`lv_img_decoder_open`

---

lv_img_decoder_t * lv_img_decoder_create（ void ）

创建一个新的图像 decoder

返回

指向新图像 decoder 的指针

---

void lv_img_decoder_delete（lv_img_decoder_t * *decoder*）

删除图像 decoder

参数

`decoder`：指向图像 decoder 的指针

---

void lv_img_decoder_set_info_cb（lv_img_decoder_t * *decoder*，lv_img_decoder_info_f_t *info_cb* ）

设置回调以获取有关图像的信息

参数

`decoder`：指向图像 decoder 的指针

`info_cb`：用于收集有关图像信息（填充`lv_img_header_t`struct）的函数

---

void lv_img_decoder_set_open_cb（lv_img_decoder_t * *decoder*，lv_img_decoder_open_f_t *open_cb* ）

设置回调以打开图像

参数

`decoder`：指向图像 decoder 的指针

`open_cb`：打开图像的功能

void lv_img_decoder_set_read_line_cb（lv_img_decoder_t * decoder, lv_img_decoder_read_line_f_t read_line_cb）

将回调设置为图像的解码行

> 参数

`decoder`：指向图像 decoder 的指针

`read_line_cb`：读取图像行的功能

void lv_img_decoder_set_close_cb（lv_img_decoder_t * decoder, lv_img_decoder_close_f_t close_cb）

设置回调以关闭解码会话。例如，关闭文件并释放其他资源。

> 参数

`decoder`：指向图像 decoder 的指针

`close_cb`：关闭解码会话的功能

lv_res_t lv_img_decoder_built_in_info（lv_img_decoder_t * decoder，const void* src, lv_img_header_t * 标头）

获取有关内置图像的信息

> 返回

LV_RES_OK：信息已成功存储在中 `header`；LV_RES_INV：未知格式或其他错误。

> 参数

`decoder`：此功能所属的 decoder

`src`：图片来源：指向 `lv_img_dsc_t` 变量，文件 path 或符号的指针

`header`：在此处存储图像数据

lv_res_t lv_img_decoder_built_in_open（lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc）

打开一个内置的图像

> 返回

LV_RES_OK：信息已成功存储在中 `header`；LV_RES_INV：未知格式或其他错误。

> 参数

`decoder`：此功能所属的 decoder

`dsc`：指向 decoder 描述符的指针。`src`，`style` 已经在其中初始化。

lv_res_t lv_img_decoder_built_in_read_line（lv_img_decoder_t * *decoder*, lv_img_decoder_dsc_t * *dsc*，lv_coord_t *x*，lv_coord_t *y*，lv_coord_t *len*，uint8_t * *buf*）

`len` 从给定的像素开始解码 `x`，进行 `y` 坐标处理并将其存储在中 `buf`。仅当“打开”功能无法返回整个解码像素数组时才需要。

返回

LV_RES_OK：好的；LV_RES_INV：失败

参数

`decoder`：指向与 decoder 相关的功能的 decoder

`dsc`：指向 decoder 描述符的指针

`x`：开始 x 坐标

`y`：开始 y 坐标

`len`：要解码的像素数

`buf`：用于存储解码像素的缓冲区

voidlv_img_decoder_built_in_close（lv_img_decoder_t * *decoder*, lv_img_decoder_dsc_t * *dsc*）

关闭待处理的解码。免费资源等

参数

`decoder`：指向与 decoder 相关的功能的 decoder

`dsc`：指向 decoder 描述符的指针

struct _lv_img_decoder

公众成员

lv_img_decoder_info_f_tinfo_cb

lv_img_decoder_open_f_topen_cb

lv_img_decoder_read_line_f_tread_line_cb

lv_img_decoder_close_f_tclose_cb

lv_img_decoder_user_data_t user_data

struct _lv_img_decoder_dsc

#include <lv_img_decoder.h>

描述图像解码会话。存储有关解码的数据

公众成员

lv_img_decoder_t *decoder

能够打开图像源的 decoder

const void*src

图像源。诸如“ S：my_img.png”的文件 path 或指向 `lv_img_dsc_t` 变量的指针

lv_color_tcolor

绘制图像的样式。

lv_img_src_tsrc_type

源类型：文件或变量。可根据 `open` 需要在功能中设置

lv_img_header_t header

有关打开的图像的信息：颜色格式，大小等。必须在 `open` 功能中设置

***const* uint8_t *img_data**

指向缓冲区的指针，在该缓冲区中以解码的纯格式存储图像的数据（像素）。必须在 `open` 功能中设置

uint32_t time_to_open

打开图像花了多少时间。[ms]如果未设置， `lv_img_cache` 将测量并设置打开时间

***const* 字符*error_msg**

无法打开图片时显示的文字而不是图片。可以在 `open` 函数中设置或设置为 NULL。

void*user_data

在此处存储任何自定义数据

图像缓存

Cai Xuefeng

功能

lv_img_cache_entry_t * _lv_img_cache_open（*constvoid* src，lv_color_t 颜色）

使用图像 decoder 界面打开图像并将其缓存。图像将保持打开状态，这意味着如果图像 decoder 打开回调分配了内存，则它将保留。如果打开了新图像并且该新图像在缓存中占据了该位置，则该图像将关闭。

返回

指向缓存条目的指针；如果可以打开图像，则为 NULL

参数

`src`：图像来源。文件的 path 或指向 `lv_img_dsc_t` 变量的指针

- `color`：具有的图像颜色 `LV_IMG_CF_ALPHA_...`

voidlv_img_cache_set_size（ uint16_t *new_slot_num* ）

设置要缓存的图像数。更多缓存的图像意味着同时打开更多的图像，这可能意味着更多的内存使用。例如，如果在 RAM 中打开 20 个 PNG 或 JPG 图像，则它们在缓存中打开时会消耗内存。

参数

new_entry_cnt：要缓存的图像数

voidlv_img_cache_invalidate_src（*constvoid* src ）

使缓存中的图像源 void。如果图像源已更新，因此需要再次缓存，则很有用。

参数

- src：文件的图像源 path 或 lv_img_dsc_t 变量的指针。

struct lv_img_cache_entry_t

#include <lv_img_cache.h>

从网络加载图像时，可能需要很长时间才能下载和解码图像。

为了避免重复这种情况，可以缓存重载图像。

公众成员

lv_img_decoder_dsc_t **dec_dsc**

图片信息

int32_t life

计算缓存条目的寿命。使用条目时添加 `time_tio_open` 到 `life`。在每个:: lv_img_cache_open 中将所有生命减一。如果 life == 0，则该条目可以重复使用

第七章 文件系统

LVGL 具有“文件系统”抽象模块，使您可以附加任何类型的文件系统。文件系统由驱动器号标识。例如，如果 SD 卡与字母相关联，则`'S'`可以访问文件，例如`"S:path/to/file.txt"`。

7.1 添加驱动程序

要添加驱动程序，`lv_fs_drv_t`需要像这样初始化：

```
    lv_fs_drv_t drv;
    lv_fs_drv_init(&drv);                     /*Basic initialization*/

    drv.letter = 'S';                         /*An uppercase letter to identify the drive */
    drv.file_size = sizeof(my_file_object);   /*Size required to store a file object*/
    drv.rddir_size = sizeof(my_dir_object);   /*Size required to store a directory object (used by
dir_open/close/read)*/
    drv.ready_cb = my_ready_cb;               /*Callback to tell if the drive is ready to use */
    drv.open_cb = my_open_cb;                 /*Callback to open a file */
    drv.close_cb = my_close_cb;               /*Callback to close a file */
    drv.read_cb = my_read_cb;                 /*Callback to read a file */
    drv.write_cb = my_write_cb;               /*Callback to write a file */
    drv.seek_cb = my_seek_cb;                 /*Callback to seek in a file (Move cursor) */
    drv.tell_cb = my_tell_cb;                 /*Callback to tell the cursor position  */
    drv.trunc_cb = my_trunc_cb;               /*Callback to delete a file */
    drv.size_cb = my_size_cb;                 /*Callback to tell a file's size */
    drv.rename_cb = my_rename_cb;             /*Callback to rename a file */

    drv.dir_open_cb = my_dir_open_cb;         /*Callback to open directory to read its content */
    drv.dir_read_cb = my_dir_read_cb;         /*Callback to read a directory's content */
    drv.dir_close_cb = my_dir_close_cb;       /*Callback to close a directory */

    drv.free_space_cb = my_free_space_cb;     /*Callback to tell free space on the drive */

    drv.user_data = my_user_data;             /*Any custom data if required*/

    lv_fs_drv_register(&drv);                 /*Finally register the drive*/
```


任何回调都可以`NULL`指示不支持该操作。

作为使用回调的示例，如果使用 LVGL：`lv_fs_open(&file, "S:/folder/file.txt", LV_FS_MODE_WR)`

1. 验证是否存在带字母的已注册驱动器`'S'`。
2. 检查它是否`open_cb`已实现（不是`NULL`）。
3. `open_cb`用`"folder/file.txt"`path 调用集合。

7.2 使用范例

下面的示例显示如何从文件读取：

```
lv_fs_file_t f;
lv_fs_res_t res;
res = lv_fs_open(&f, "S:folder/file.txt", LV_FS_MODE_RD);
if(res != LV_FS_RES_OK) my_error_handling();

uint32_t read_num;
uint8_t buf[8];
res = lv_fs_read(&f, buf, 8, &read_num);
if(res != LV_FS_RES_OK || read_num != 8) my_error_handling();

lv_fs_close(&f);
```

的模式 `lv_fs_open` 可以是 `LV_FS_MODE_WR` 打开或同时打开 `LV_FS_MODE_RD | LV_FS_MODE_WR`

本示例说明如何读取目录的内容。由驱动程序决定如何标记目录，但是 `'/'` 在目录名称前插入 a 可能是一个好习惯。

```
lv_fs_dir_t dir;
lv_fs_res_t res;
res = lv_fs_dir_open(&dir, "S:/folder");
if(res != LV_FS_RES_OK) my_error_handling();



char fn[256];
while(1) {
    res = lv_fs_dir_read(&dir, fn);
    if(res != LV_FS_RES_OK) {
        my_error_handling();
        break;
    }

    /*fn is empty, if not more files to read*/
    if(strlen(fn) == 0) {
        break;
    }

    printf("%s\n", fn);
}

lv_fs_dir_close(&dir);
```

7.3 使用图像驱动程序

图像对象也可以从文件中打开（除了存储在闪存中的变量）。
要初始化图像，需要以下回调：

- 打开

- 关
- 读
- 寻求
- 告诉

API

typedef

*typedef*uint8_t lv_fs_res_t

*typedef*uint8_t lv_fs_mode_t

typedefstruct lv_fs_drv_tlv_fs_drv_t

枚举

枚举[anonymous]

文件系统模块中的错误。

值：

*enumerator*LV_FS_RES_OK = 0

*enumerator*LV_FS_RES_HW_ERR

*enumerator*LV_FS_RES_FS_ERR

*enumerator*LV_FS_RES_NOT_EX

*enumerator*LV_FS_RES_FULL

*enumerator*LV_FS_RES_LOCKED

*enumerator*LV_FS_RES_DENIED

*enumerator*LV_FS_RES_BUSY

*enumerator*LV_FS_RES_TOUT

*enumerator*LV_FS_RES_NOT_IMP

*enumerator*LV_FS_RES_OUT_OF_MEM

*enumerator*LV_FS_RES_INV_PARAM

*enumerator*LV_FS_RES_UNKNOWN

枚举[anonymous]

文件系统模式。

值：

*enumerator*LV_FS_MODE_WR = 0x01

*enumerator*LV_FS_MODE_RD = 0x02

功能

void_lv_fs_init（ void ）

初始化文件系统界面

void lv_fs_drv_init（lv_fs_drv_t * *drv* ）

使用默认值初始化文件系统驱动程序。它通常用于在 ant（不是内存垃圾）字段中具有已知值。之后，您可以设置字段。

参数

`drv`：指向要初始化的驱动程序变量的指针

void lv_fs_drv_register（lv_fs_drv_t * *drv_p* ）

新增驱动器

参数

`drv_p`：指向 lv_fs_drv_tstruct 的指针，该 struct 由相应的函数指针初始化。数据将被复制，因此变量可以是本地的。

lv_fs_drv_t * lv_fs_get_drv（*字符信*）

从信中给司机一个指针

返回

指向驱动程序的指针；如果找不到，则为 NULL

参数

letter：驾驶员信

bool lv_fs_is_ready（*字符字母*）

测试驱动器是否准备就绪。如果 `ready` 函数未初始化，`true` 将返回。

返回

true：驱动器已准备就绪；false：驱动器未准备好

参数

letter：驱动器号

lv_fs_res_t lv_fs_open（lv_fs_file_t * *file_p*，*const*char * *path*，lv_fs_mode_t *模式*）

开启档案

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`path`：以驱动器号开头的文件的 path（例如 S：/folder/file.txt）

`mode`：读取：FS_MODE_RD，写入：FS_MODE_WR，两者：FS_MODE_RD | FS_MODE_WR

lv_fs_res_t lv_fs_close（lv_fs_file_t * *file_p* ）

关闭已打开的文件

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

---

lv_fs_res_t lv_fs_remove（*const char * path*）

删除档案

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`path`：要删除的文件的 path

---

lv_fs_res_t lv_fs_read（lv_fs_file_t * *file_p*，void * *buf*，uint32_t *btr*，uint32_t * *br*）

从文件读取

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`buf`：指向存储读取字节的缓冲区的指针

`btr`：要读取的字节

`br`：实际读取的字节数（读取的字节数）。如果未使用，则为 NULL。

---

lv_fs_res_t lv_fs_write（lv_fs_file_t * *file_p*，const void * *buf*，uint32_t *btw*，uint32_t * *bw*）

写入文件

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`buf`：指向要写入字节的缓冲区的指针

`btr`：要写入的字节

`br`：实际写入的字节数（已写入的字节数）。如果未使用，则为 NULL。

---

lv_fs_res_t lv_fs_seek（lv_fs_file_t * *file_p*，uint32_t *pos*）

设置文件中“光标”（读写指针）的位置

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`pos`：以字节索引表示的新位置（0：文件开始）

---

lv_fs_res_t lv_fs_tell（lv_fs_file_t * *file_p*，uint32_t * *pos* ）

给出读写指针的位置

返回

LV_FS_RES_OK 或'fs_res_t'中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`pos_p`：指针，用于存储读写指针的位置

---

lv_fs_res_t lv_fs_trunc（lv_fs_file_t * *file_p* ）

将文件大小截断为读写指针的当前位置

返回

LV_FS_RES_OK：没有错误，从 lv_fs_res_t 枚举中读取文件有任何错误

参数

`file_p`：指向“ ufs_file_t”变量的指针。（以 lv_fs_open 打开）

---

lv_fs_res_t lv_fs_size（lv_fs_file_t * *file_p*，uint32_t * *size* ）

给出文件的大小字节

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`file_p`：指向 lv_fs_file_t 变量的指针

`size`：指向存储大小的变量的指针

---

lv_fs_res_t lv_fs_rename（*const*char * *oldname*，*const*char * *newname* ）

重命名文件

返回

LV_FS_RES_OK 或'fs_res_t'中的任何错误

参数

oldname：文件的 path

newname：具有新名称的 path

---

lv_fs_res_t lv_fs_dir_open（lv_fs_dir_t * *rddir_p*，*const*char * *path* ）

初始化“ fs_dir_t”变量以进行目录读取

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`rddir_p`：指向'fs_read_dir_t'变量的指针

`path`：目录 path

---

lv_fs_res_t lv_fs_dir_read（lv_fs_dir_t * *rddir_p*，char * *fn* ）

从目录中读取下一个文件名。目录名称将以“ /”开头

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`rddir_p`：指向已初始化的'fs_rdir_t'变量的指针

`fn`：指向用于存储文件名的缓冲区的指针

---

lv_fs_res_t lv_fs_dir_close（lv_fs_dir_t * *rddir_p* ）

关闭目录阅读

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

`rddir_p`：指向已初始化的'fs_dir_t'变量的指针

---

lv_fs_res_t lv_fs_free_space（字符 *字母*，uint32_t 的* *total_p*，uint32_t 的* *free_p* ）

获得驱动程序的免费总大小（以 kB 为单位）

返回

LV_FS_RES_OK 或 lv_fs_res_t 枚举中的任何错误

参数

letter：驾驶员信

total_p：存储总大小的指针[kB]

free_p：用于存储可用大小的指针[kB]

---

char * lv_fs_get_letters（ char * *buf* ）

用现有驱动程序的字母填充缓冲区

返回

缓冲区

参数

`buf`：缓冲区以存储字母（在最后一个字母后添加'\ 0'）

const 字符* lv_fs_get_ext（*const* 字符* *fn* ）

返回文件名的扩展名

返回

指向起始扩展名的指针；如果没有扩展名，则为空字符串

参数

`fn`：带有文件名的字符串

char * lv_fs_up（ char * *path*）

提升一级

返回

截断的文件名

参数

`path`：指向文件名的指针

const 字符* lv_fs_get_last（*const*char * *path* ）

获取 path 的最后一个元素（例如，U：/ folder / file-> file）

返回

指向 path 中最后一个元素的开始的指针

参数

`buf`：缓冲区以存储字母（在最后一个字母后添加'\0'）

*struct*_lv_fs_drv_t

公众成员

char **letter**

uint16_t **file_size**

uint16_t **rddir_size**

bool（* **ready_cb**）（*struct* lv_fs_drv_t * drv ）

lv_fs_res_t（* **open_cb**）（*struct* lv_fs_drv_t * drv，void* file_p，*const*char * path，lv_fs_mode_t mode）

lv_fs_res_t（* **close_cb**）（*struct* lv_fs_drv_t * drv，void * file_p ）

lv_fs_res_t（* **remove_cb**）（*struct* lv_fs_drv_t * drv，*const* char * fn ）

lv_fs_res_t（* **read_cb**）（*struct* lv_fs_drv_t * drv，void* file_p，void* buf，uint32_t btr，uint32_t * br ）

lv_fs_res_t（* **write_cb**）（*struct* lv_fs_drv_t * drv，void* file_p，*const*void* buf，uint32_t btw，uint32_t * bw ）

lv_fs_res_t（* **seek_cb**）（*struct* lv_fs_drv_t * drv，void * file_p，uint32_t pos ）

lv_fs_res_t（* **tell_cb**）（*struct* lv_fs_drv_t * drv，void * file_p，uint32_t * pos_p ）

lv_fs_res_t（* **trunc_cb**）（*struct* lv_fs_drv_t * drv，void * file_p ）

lv_fs_res_t（* **size_cb**）（*struct* lv_fs_drv_t * drv，void * file_p，uint32_t * size_p ）

lv_fs_res_t (* rename_cb) (struct lv_fs_drv_t * drv, const char * oldname, const char * newname)

lv_fs_res_t (* free_space_cb) (struct lv_fs_drv_t * drv, uint32_t * total_p, uint32_t * free_p)

lv_fs_res_t (* dir_open_cb) (struct lv_fs_drv_t * drv, void * rddir_p, const char * path)

lv_fs_res_t (* dir_read_cb) (struct lv_fs_drv_t * drv, void * rddir_p, char * fn)

lv_fs_res_t (* dir_close_cb) (struct lv_fs_drv_t * drv, void * rddir_p)

lv_fs_drv_user_data_t user_data

自定义文件用户数据

struct lv_fs_file_t

公众成员

void* file_d

lv_fs_drv_t * drv

struct lv_fs_dir_t

公众成员

void* dir_d

lv_fs_drv_t * drv

7.4 动画制作

您可以使用动画在开始值和结束值之间自动更改变量的值。动画将通过定期调用带有相应 value 参数的“ animator”函数来发生。

该*动画*功能具有以下原型：

```
void func(void * var, lv_anim_var_t value);
```

该原型与 LVGL 的大多数设置功能兼容。例如或 lv_obj_set_x(obj, value)lv_obj_set_width(obj, value)

7.5 创建动画

要创建动画 `lv_anim_t`，必须初始化变量并使用 `lv_anim_set_...()` 功能对其进行配置。

```
/* INITIALIZE AN ANIMATION
 *----------------------*/

lv_anim_t a;
lv_anim_init(&a);

/* MANDATORY SETTINGS
 *------------------*/

/*Set the "animator" function*/
lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t) lv_obj_set_x);

/*Set the "animator" function*/
lv_anim_set_var(&a, obj);

/*Length of the animation [ms]*/
lv_anim_set_time(&a, duration);

/*Set start and end values. E.g. 0, 150*/
lv_anim_set_values(&a, start, end);

/* OPTIONAL SETTINGS
 *------------------*/

/*Time to wait before starting the animation [ms]*/
lv_anim_set_delay(&a, delay);

/*Set path (curve). Default is linear*/
lv_anim_set_path(&a, &path);

/*Set a callback to call when animation is ready.*/
lv_anim_set_ready_cb(&a, ready_cb);

/*Set a callback to call when animation is started (after delay).*/
```

```
    lv_anim_set_start_cb(&a, start_cb);

    /*Play the animation backward too with this duration. Default is 0 (disabled) [ms]*/
    lv_anim_set_playback_time(&a, wait_time);

    /*Delay before playback. Default is 0 (disabled) [ms]*/
    lv_anim_set_playback_delay(&a, wait_time);

    /*Number of repetitions. Default is 1.  LV_ANIM_REPEAT_INFINIT for infinite repetition*/
    lv_anim_set_repeat_count(&a, wait_time);

    /*Delay before repeat. Default is 0 (disabled) [ms]*/
    lv_anim_set_repeat_delay(&a, wait_time);

    /*true (default): apply the start vale immediately, false: apply start vale after delay when then
anim. really starts. */
    lv_anim_set_early_apply(&a, true/false);

    /* START THE ANIMATION
     *------------------*/
    lv_anim_start(&a);                          /*Start the animation*/
```

您可以同时在同一变量上应用**多个不同的动画**。例如，使用 `lv_obj_set_x` 和设置 x 和 y 坐标的动画 `lv_obj_set_y`。但是，只有一个动画可以存在给定的变量和函数对。因此 `lv_anim_start()` 将删除已经存在的可变功能动画。

7.6 动画 path

您可以确定**动画**的 **path**。在最简单的情况下，它是线性的，这意味着*开始*和*结束*之间的当前值 线性变化。甲 *path* 主要是其计算基于所述动画的当前状态中的下一个值集的函数。当前，有以下内置 path 功能：

- **lv_anim_path_linear** 线性动画
- **lv_anim_path_step** 最后一步更改
- **lv_anim_path_ease_in** 开头缓慢
- **lv_anim_path_ease_out** 最后慢
- **lv_anim_path_ease_in_out** 在开始和结束时也很慢
- **lv_anim_path_overshoot** 超出最终值
- **lv_anim_path_bounce** 从最终值反弹一点（就像撞墙一样）

可以这样初始化 path：

```
    lv_anim_path_t path;
    lv_anim_path_init(&path);
    lv_anim_path_set_cb(&path, lv_anim_path_overshoot);
    lv_anim_path_set_user_data(&path, &foo); /*Optional for custom functions*/

    /*Set the path in an animation*/
    lv_anim_set_path(&a, &path);
```

7.7 速度与时间

默认情况下，您可以设置动画时间。但是，在某些情况下，**动画速度**更加实用。

该函数以给定的速度计算从起始值到结束值所需的时间（以毫秒为单位）。速度以单位/秒为单位进行解释。例如， 将给出 5000 毫秒。例如，如果单位是像素，则 20 表示 20 px / sec 的速度。

lv_anim_speed_to_time(speed, start, end)lv_anim_speed_to_time(20,0,100)lv_obj_set_x

7.8 删除动画

您可以**删除动画**通过提供动画变量及其动画功能。`lv_anim_del(var, func)`

API

输入设备

typedef

---

*typedef*uint8_t **lv_anim_enable_t**

*typedef*lv_coord_t **lv_anim_value_t**

动画值的类型

---

*typedef*lv_anim_value_t（* **lv_anim_path_cb_t**）（*const*struct lv_anim_path_t *, *const*struct lv_anim_t * ）

在动画期间获取当前值

---

*typedef*struct lv_anim_path_t**lv_anim_path_t**

*typedef*void（* **lv_anim_exec_xcb_t**）（void*，lv_anim_value_t）

“动画师”功能的通用原型。第一个参数是要设置动画的变量。第二个参数是要设置的值。与函数兼容 in 表示其不是完全通用的原型，因为它没有作为第一个参数接收`lv_xxx_set_yyy(obj, value)x_xcb_tlv_anim_t *`

---

*typedef*void（* **lv_anim_custom_exec_cb_t**）（*struct* lv_anim_t *，lv_anim_value_t）

与第一个参数相同，`lv_anim_exec_xcb_t`但作为第一个参数接收。它更一致，但不太方便。可能由绑定生成器功能使用。`lv_anim_t *`

---

*typedef*void（* **lv_anim_ready_cb_t**）（*struct* lv_anim_t * ）

动画准备好时回叫

---

*typedef*void（* **lv_anim_start_cb_t**）（*struct* lv_anim_t * ）

动画真正开始播放时回叫（考虑`delay`）

---

*typedef*struct lv_anim_t**lv_anim_t**

描述动画

枚举

枚举[anonymous]

可用于指示在某种情况下是启用还是禁用动画

值：

*enumerator*LV_ANIM_OFF

*enumerator*LV_ANIM_ON

功能

void_lv_anim_core_init（ void ）

在里面。动画模块

voidlv_anim_init（lv_anim_t * *a* ）

初始化动画变量。例如：lv_anim_t a; lv_anim_init（&a）; lv_anim_set _...（&a）;

参数

`a`：指向 `lv_anim_t` 要初始化的变量的指针

voidlv_anim_set_var（lv_anim_t * *a*，void* *var* ）

设置变量进行动画处理

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`var`：指向要动画的变量的指针

voidlv_anim_set_exec_cb（lv_anim_t * *a*，lv_anim_exec_xcb_t *exec_cb* ）

设置动画功能 `var`

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`exec_cb`：可以在动画过程中执行的函数 LittelvGL 的内置函数可以使用。例如 lv_obj_set_x

voidlv_anim_set_time（lv_anim_t * *a*，uint32_t *持续时间*）

设置动画的持续时间

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`duration`：动画的持续时间（以毫秒为单位）

voidlv_anim_set_delay（lv_anim_t * *a*，uint32_t *delay* ）

设置开始动画之前的延迟

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`delay`：动画开始前的延迟（以毫秒为单位）

void lv_anim_set_values（lv_anim_t * a，lv_anim_value_t start，lv_anim_value_t end ）

设置动画的开始和结束值

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`start`：起始值

`end`：最终值

voidlv_anim_set_custom_exec_cb（lv_anim_t * a，lv_anim_custom_exec_cb_t exec_cb ）

类似 lv_anim_set_exec_cb 但 lv_anim_custom_exec_cb_t 接收作为第一个参数，而不是。当 LVGL 绑定到其他语言时，可以使用此函数，因为它与第一个参数更一致。动画变量可以存储在动画的

`lv_anim_t *void *lv_anim_t *user_sata`

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`exec_cb`：要执行的功能。

voidlv_anim_set_path（lv_anim_t * a，constlv_anim_path_t * path ）

设置动画的 path（曲线）。

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`path_cb`：一个函数，获取动画的当前值。内置功能以 `lv_anim_path_...`

voidlv_anim_set_start_cb（lv_anim_t * a，lv_anim_ready_cb_t start_cb ）

在动画真正开始时设置函数调用（考虑 `delay`）

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`start_cb`：动画开始时的函数调用

voidlv_anim_set_ready_cb（lv_anim_t * a，lv_anim_ready_cb_t ready_cb ）

动画准备好时设置函数调用

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`ready_cb`：动画准备好时的函数调用

void lv_anim_set_playback_time（lv_anim_t * a，uint16_t 时间）

准备好前进方向时播放动画

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`time`：播放动画的持续时间（以毫秒为单位）。0：禁用播放

void lv_anim_set_playback_delay（lv_anim_t * a，uint16_t 延迟）

准备好前进方向时播放动画

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`delay`：开始播放动画之前的延迟（以毫秒为单位）。

void lv_anim_set_repeat_count（lv_anim_t * a，uint16_t cnt ）

使动画重复。

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`cnt`：重复计数或 `LV_ANIM_REPEAT_INFINITE` 无限重复。0：禁用重复。

void lv_anim_set_repeat_delay（lv_anim_t * a，uint16_t 延迟）

在重复动画之前设置延迟。

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

`delay`：延迟（以毫秒为单位），然后重复动画。

void lv_anim_start（lv_anim_t * a ）

创建动画

参数

`a`：初始化的“ anim_t”变量。通话后不需要。

void lv_anim_path_init（lv_anim_path_t * path ）

初始化动画 path

参数

`path`：指向 path 的指针

void lv_anim_path_set_cb（lv_anim_path_t * path，lv_anim_path_cb_t cb ）

设置 path 的回调

参数

`path`：指向初始化 path 的指针

`cb`：回调

void lv_anim_path_set_user_data（lv_anim_path_t * *path*，void* *user_data* ）

设置 path 的用户数据

参数

`path`：指向初始化 path 的指针

`user_data`：指向用户数据的指针

int32_t lv_anim_get_delay（lv_anim_t * *a* ）

在开始动画之前获得延迟

返回

动画之前的延迟（以毫秒为单位）

参数

`a`：指向初始化 `lv_anim_t` 变量的指针

bool lv_anim_del（ void * *var*，lv_anim_exec_xcb_t *exec_cb* ）

使用给定的动画师功能删除变量的动画

返回

true：删除至少 1 个动画，false：不删除任何动画

参数

`var`：指向变量的指针

`exec_cb`：一个动画“ var”的函数指针，或者为 NULL 以忽略它并删除所有“ var”动画

lv_anim_t * lv_anim_get（ void * *var*，lv_anim_exec_xcb_t *exec_cb* ）

获取变量及其动画 `exec_cb`。

返回

指向动画的指针。

参数

`var`：指向变量的指针

`exec_cb`：为'var'设置动画的函数指针，或者为 NULL，以删除'var'的所有动画

bool lv_anim_custom_del（lv_anim_t * *a*，lv_anim_custom_exec_cb_t *exec_cb* ）

通过从获取动画变量来删除动画 `a`。仅带有的动画 `exec_cb` 将被删除。之所以存在此功能，是因为所有动画

都符合逻辑。函数 `lv_anim_t` 将第一个参数作为参数。它在 C 语言中不切实际，但可能会使 API 更加复杂，并使绑定更容易生成。

返回

true：删除至少 1 个动画，false：不删除任何动画

参数

`a`：指向动画的指针。

`exec_cb`：一个动画“ var”的函数指针，或者为 NULL 以忽略它并删除所有“ var”动画

uint16_t lv_anim_count_running（ void ）

获取当前正在运行的动画数

返回

运行动画的数量

uint16_t lv_anim_speed_to_time（ uint16_t *速度*，lv_anim_value_t *start*，lv_anim_value_t *end* ）

以给定的速度以及开始和结束值计算动画的时间

返回

具有给定参数的动画所需的时间[ms]

参数

`speed`：动画速度，单位/秒

`start`：动画的起始值

`end`：动画的最终值

voidlv_anim_refr_now（ void ）

手动刷新动画的状态。使动画在 `lv_task_handler` 无法暂时运行的阻塞过程中运行很有用。不应直接使用，因为它是在中调用的 `lv_refr_now()`。

lv_anim_value_t lv_anim_path_linear（*const*lv_anim_path_t * *path*，*const*lv_anim_t * *a* ）

应用线性特征计算动画的当前值

返回

要设置的当前值

参数

`a`：指向动画的指针

lv_anim_value_t lv_anim_path_ease_in（*const*lv_anim_path_t * *path*，*const*lv_anim_t * *a* ）

计算动画的当前值，减慢开始阶段的速度

返回

要设置的当前值

参数

a：指向动画的指针

---

lv_anim_value_t lv_anim_path_ease_out（const lv_anim_path_t * path，const lv_anim_t * a ）

计算动画的当前值，以减慢结束阶段的速度

返回

要设置的当前值

参数

a：指向动画的指针

---

lv_anim_value_t lv_anim_path_ease_in_out（const lv_anim_path_t * path，const lv_anim_t * a ）

计算应用“ S”特征（余弦）的动画的当前值

返回

要设置的当前值

参数

a：指向动画的指针

---

lv_anim_value_t lv_anim_path_overshoot（const lv_anim_path_t * path，const lv_anim_t * a ）

计算动画的当前值，最后带有超调

返回

要设置的当前值

参数

a：指向动画的指针

---

lv_anim_value_t lv_anim_path_bounce（const lv_anim_path_t * path，const lv_anim_t * a ）

计算 3 次跳动的动画的当前值

返回

要设置的当前值

参数

a：指向动画的指针

---

lv_anim_value_t lv_anim_path_step（const lv_anim_path_t * path，const lv_anim_t * a ）

计算应用阶跃特性的动画的当前值。（在动画末尾设置结束值）

返回

要设置的当前值

参数

a：指向动画的指针

变数

const lv_anim_path_t lv_anim_path_def

struct _lv_anim_path_t

公众成员

lv_anim_path_cb_t cb

void* user_data

struct _lv_anim_t

#include <lv_anim.h>

描述动画

公众成员

void* var

可变动画

lv_anim_exec_xcb_t exec_cb

执行动画的功能

lv_anim_start_cb_t start_cb

动画开始时调用它（考虑 `delay`）

lv_anim_ready_cb_t ready_cb

动画准备好后调用它

lv_anim_path_t path

描述动画的 path（曲线）

int32_t start

起始值

int32_t current

当前值

int32_t end

终值

int32_t time

动画时间（毫秒）

int32_t act_time

动画中的当前时间。设置为负数可以延迟。

uint32_t playback_delay

等待播放

uint32_t playback_time

播放动画的持续时间

uint32_t repeat_delay

等待，然后重复

uint16_t repeat_cnt

重复动画计数

uint8_t early_apply

1：即使存在也立即应用起始值 `delay`

lv_anim_user_data_t user_data

自定义用户数据

```
    uint32_t time_orig
    uint8_t playback_now
```

播放正在进行中

```
    uint32_t has_run
```

表示动画已在这一轮中运行

7.9 任务

LVGL 具有内置的任务系统。您可以注册一个函数以使其定期被调用。任务是在中处理和调用的

`lv_task_handler()`，需要每隔几毫秒定期调用一次。有关更多信息，请参见移植。

任务是非抢占式的，这意味着一个任务无法中断另一个任务。因此，您可以在任务中调用任何与 LVGL 相关的功能。

7.10 创建一个任务

要创建新任务，请使用。它将创建一个变量，以后可用于修改任务的参数。 也可以使用。它允许您创建新任务而无需指定任何参数。

`lv_task_create(task_cb, period_ms, LV_TASK_PRIO_OFF/LOWEST/LOW/MID/HIGH/HIGHEST, user_data)lv_task_t *lv_t`

`ask_create_basic()`

任务回调应具有原型。`void (*lv_task_cb_t)(lv_task_t *);`

例如：

```
void my_task(lv_task_t * task)
{
  /*Use the user_data*/
  uint32_t * user_data = task->user_data;
  printf("my_task called with user data: %d\n", *user_data);

  /*Do something with LVGL*/
  if(something_happened) {
    something_happened = false;
    lv_btn_create(lv_scr_act(), NULL);
  }
}

...

static uint32_t user_data = 10;
lv_task_t * task = lv_task_create(my_task, 500, LV_TASK_PRIO_MID, &user_data);
```


7.11 准备并重置

`lv_task_ready(task)` 使任务在的下一个调用上运行 `lv_task_handler()`。

`lv_task_reset(task)` 重置任务的期限。在定义的毫秒周期过后，它将再次调用。

7.12 设定参数

您可以稍后修改任务的一些参数:

- `lv_task_set_cb(task, new_cb)`
- `lv_task_set_period(task, new_period)`
- `lv_task_set_prio(task, new_priority)`

7.13 一键式任务

您可以通过调用使任务仅运行一次 `lv_task_once(task)`。首次调用该任务后，该任务将自动删除。

7.14 测量空闲时间

你可以在空闲时间百分比 `lv_task_handler` 用 `lv_task_get_idle()`。请注意，它仅衡量整个系统的空闲时间 `lv_task_handler`。如果您使用操作系统并调用 `lv_task_handler` 任务，可能会产生误导，因为它实际上无法衡量 OS 在空闲线程上花费的时间。

7.15 异步呼叫

在某些情况下，您无法立即执行操作。例如，您不能立即删除对象，因为其他对象仍在使用它，或者您不想立即阻止执行。对于这些情况，您可以使用，使在下次调用被调用。会在调用时传递给函数。请注意，仅保存了数据的指针，因此您需要确保在调用函数时变量将为“有效”。您可以使用*静态*，全局或动态分配的数据。

`lv_async_call(my_function, data_p)my_functionlv_task_handlerdata_p`

例如:

```
void my_screen_clean_up(void * scr)
{
  /*Free some resources related to `scr`*/

  /*Finally delete the screen*/
  lv_obj_del(scr);
}

...

/*Do somethings with the object on the current screen*/

/*Delete screen on next call of `lv_task_handler`. So not now.*/
lv_async_call(my_screen_clean_up, lv_scr_act());
```

```
/*The screen is still valid so you can do other things with it*/
```

如果您只想删除一个对象，而无需清理其中的任何内容 `my_screen_cleanup`，则可以使用 `lv_obj_del_async`，它将在下次调用时删除该对象 `lv_task_handler`。

API

typedef

***typedef*void（* lv_task_cb_t）（*struct* lv_task_t * ）**

任务执行这种类型的功能。

***typedef*uint8_t lv_task_prio_t**

***typedef*struct lv_task_t lv_task_t**

lv_task 的描述符

枚举

***枚举*[anonymous]**

lv_tasks 的可能优先级

值：

***enumerator*LV_TASK_PRIO_OFF = 0**

***enumerator*LV_TASK_PRIO_LOWEST**

***enumerator*LV_TASK_PRIO_LOW**

***enumerator*LV_TASK_PRIO_MID**

***enumerator*LV_TASK_PRIO_HIGH**

***enumerator*LV_TASK_PRIO_HIGHEST**

***enumerator*_LV_TASK_PRIO_NUM**

功能

void _lv_task_core_init（ void ）

初始化 lv_task 模块

lv_task_t * lv_task_create_basic（ void ）

创建一个“空”任务。它至少需要用 `lv_task_set_cb` 和初始化 `lv_task_set_period`

返回

指向创建任务的指针

lv_task_t * lv_task_create（lv_task_cb_t *task_xcb*，uint32_t 的*期间*，lv_task_prio_t *PRIO*，void* *的 user_data* ）

创建一个新的 lv_task

返回

指向新任务的指针

> 参数

`task_xcb`：本身就是任务的回调。它将被定期调用。（参数名称中的“ x”表示它不是完全通用的函数，因为它不遵循约定）`func_name(object, callback, ...)`

`period`：呼叫周期（以毫秒为单位）

`prio`：任务的优先级（LV_TASK_PRIO_OFF 表示任务已停止）

`user_data`：自定义参数

---

void lv_task_del（lv_task_t * *task* ）

删除一个 lv_task

> 参数

`task`：由 task 创建的指向 task_cb 的指针

---

void lv_task_set_cb（lv_task_t * *task*，lv_task_cb_t *task_cb* ）

设置回调任务（该函数定期调用）

> 参数

`task`：指向任务的指针

`task_cb`：定期调用的函数

---

void lv_task_set_prio（lv_task_t * *任务*，lv_task_prio_t *PRIO* ）

设置 lv_task 的新优先级

> 参数

`task`：指向 lv_task 的指针

`prio`：新的优先事项

---

void lv_task_set_period（lv_task_t * *task*，uint32_t *period* ）

设置 lv_task 的新期限

> 参数

`task`：指向 lv_task 的指针

`period`：新时期

---

void lv_task_ready（lv_task_t * *task* ）

准备一个 lv_task。它不会等待其期间。

> 参数

`task`：指向 lv_task 的指针。

void lv_task_set_repeat_count（lv_task_t * *task*，int32_t *repeat_count*）

设置任务重复的次数。

参数

`task`：指向 lv_task 的指针。

`repeat_count`：-1：无限 0：停止; n> 0：剩余时间

void lv_task_reset（lv_task_t * *task*）

重置 lv_task。毫秒后称为先前设置的时间段。

参数

`task`：指向 lv_task 的指针。

空 lv_task_enable（ bool *en*）

启用或禁用整个 lv_task 处理

参数

`en`：true：lv_task 处理正在运行，false：lv_task 处理已暂停

uint8_t lv_task_get_idle（ void ）

取得闲置百分比

返回

lv_task 空闲百分比

lv_task_t * lv_task_get_next（lv_task_t * *task*）

遍历任务

返回

下一个任务，如果没有更多任务，则为 NULL

参数

`task`：NULL 以开始迭代，或者返回上一个返回值以获取下一个任务

struct _lv_task_t

#include <lv_task.h>

lv_task 的描述符

公众成员

uint32_t **period**

任务应多久运行一次

uint32_t **last_run**

上一次任务执行

lv_task_cb_t**task_cb**

任务功能

void***user_data**

自定义用户数据

int32_t **repeat_count**

1：任务时间；-1：无限 0：停止; n> 0：剩余时间

```
    uint8_t prio
```

任务优先级

7.16 画画

使用 LVGL，您无需手动绘制任何内容。只需创建对象（如按钮和标签），移动并更改它们，LVGL 就会刷新并重新绘制所需的内容。

但是，对 LVGL 中的绘图方式有基本了解可能会很有用。

基本概念是不直接绘制到屏幕上，而是先绘制到内部缓冲区，然后在渲染准备好后将其复制到屏幕上。它具有两个主要优点：

1. **避免**在绘制 UI 层时出现**闪烁**。例如，当绘制 *背景+按钮+文本时*，每个“阶段”将在短时间内可见。
2. 修改 RAM 中的缓冲区并最终一次写入一个像素比直接在每个像素访问上读写显示器**更快**。（例如，通过具有 SPI 接口的显示控制器）。因此，它适用于多次重绘的像素（例如，背景+按钮+文本）。

7.17 缓冲类型

正如您可能在“ 移植”部分中了解到的，共有 3 种类型的缓冲区：

1. **一个缓冲区** -LVGL 将屏幕的内容绘制到缓冲区中，并将其发送到显示器。缓冲区可以小于屏幕。在这种情况下，较大的区域将被重画成多个部分。如果只有很小的区域发生变化（例如，按下按钮），则只会刷新那些区域。
2. **两个非屏幕大小的缓冲区** -具有两个缓冲区，LVGL 可以绘制到一个缓冲区中，而另一缓冲区的内容发送到背景中显示。应该使用 DMA 或其他硬件将数据传输到显示器，以让 CPU 同时绘图。这样，显示的渲染和刷新变得并行。如果缓冲区小于要刷新的区域，则 LVGL 将以类似于 *One 缓冲区的*块形式绘制显示内容。
3. **两个屏幕大小的缓冲区** -与 *两个非屏幕大小的缓冲区*相反，LVGL 将始终提供整个屏幕的内容，而不仅仅是块。这样，驱动程序可以简单地将帧缓冲区的地址更改为从 LVGL 接收的缓冲区。因此，当 MCU 具有 LCD / TFT 接口并且帧缓冲区只是 RAM 中的一个位置时，此方法最有效。

7.18 屏幕刷新机制

1. GUI 上发生某些事情，需要重绘。例如，已按下按钮，更改了图表或发生了动画等。
2. LVGL 将更改后的对象的旧区域和新区域保存到称为 *void 区域缓冲区的缓冲区中*。为了优化，在某些情况下，对象不会添加到缓冲区中：
   - 隐藏的对象未添加。
   - 不添加完全超出其父对象的对象。
   - 父区之外的区域将裁剪到父区。
   - 未添加其他屏幕上的对象。
3. 在每个 `LV_DISP_DEF_REFR_PERIOD` （在 *lv_conf.h 中*设置）中：
   - LVGL 检查 void 区域并加入相邻或相交的区域。

- 取得第一个连接区域，如果它小于*显示缓冲区*，则只需将区域的内容绘制到*显示缓冲区即可*。如果该区域不适合缓冲区，请向*显示缓冲区*绘制尽可能多的线。
- 绘制区域后，`flush_cb`从显示驱动程序调用以刷新显示。
- 如果该区域大于缓冲区，则也重新绘制其余部分。
- 对所有连接的区域执行相同的操作。

重绘区域时，库会搜索覆盖要重绘区域的最顶层对象，然后从该对象开始绘制。例如，如果按钮的标签已更改，则库将看到足以在文本下方绘制按钮，也不需要绘制背景。

关于绘制机制，缓冲区类型之间的差异如下：

1. **一个缓冲区** -LVGL 需要等待`lv_disp_flush_ready()`（在末尾调用`flush_cb`），然后才能开始重画下一部分。
2. **两个非屏幕大小的缓冲区** -当第一个缓冲区发送到 LVGL 时，LVGL 可以立即绘制到第二个缓冲区，`flush_cb`因为刷新应由后台的 DMA（或类似硬件）完成。
3. **两个屏幕大小的缓冲区** -调用后`flush_cb`，第一个缓冲区（如果显示为帧缓冲区）。它的内容被复制到第二个缓冲区，所有更改都绘制在其顶部。

7.19 遮罩

*遮罩*是 LVGL 绘图引擎的基本概念。要使用 LVGL，不需要了解这里描述的机制，但是您可能会对了解图纸在引擎盖下的工作方式感兴趣。

要学习遮罩，让我们先学习绘画的步骤：

1. 根据对象的样式（例如`lv_draw_rect_dsc_t`）创建绘制描述符。它告诉图形的参数，例如颜色，宽度，不透明度，字体，半径等。
2. 使用初始化的描述符和其他一些参数调用 draw 函数。它将原始形状渲染到当前绘制缓冲区。
3. 如果形状非常简单并且不需要遮罩，请转到＃5。否则创建所需的蒙版（例如，圆角矩形蒙版）
4. 将所有创建的蒙版应用到一行或几行。它使用创建的*遮罩*的“形状”在*遮罩缓冲区中*创建 0..255 值。例如，在根据掩码参数设置“线掩码”的情况下，将缓冲区的一侧保持原样（默认为 255），并将其余部分设置为 0 以指示应将另一侧除去。
5. 将图像或矩形混合到屏幕上。在混合蒙版（使某些像素透明或不透明），混合模式（加性，减性等）期间，将处理不透明性。
6. 从＃4 重复。

使用遮罩可创建几乎所有基本图元：

- **字母**从**字母**创建遮罩，并使用该遮罩绘制一个“字母色”矩形。
- 由 4 个“ ine masks”创建的**线条**，以遮盖**线条**的左，右，顶部和底部，以获得完美垂直的线条结尾
- **圆角矩形**实时为圆角矩形的每一行创建一个遮罩，并根据该遮罩绘制一个普通的填充矩形。

- **剪辑角**，以剪辑圆角上溢出的内容，同时应用圆角矩形蒙版。
- **矩形边框**与圆角矩形相同，但内部也被遮罩
- **圆弧绘制绘制**了一个圆边框，但是应用了一个圆弧蒙版。
- 将 Alpha 通道的 **ARGB 图像**分离为一个蒙版，并将该图像绘制为普通 RGB 图像。

如以上＃3 所述，在某些情况下，不需要面具：

- 一个单色的，不是圆角的矩形
- RGB 图像

LVGL 具有以下内置掩码类型，可以实时计算和应用：

- **LV_DRAW_MASK_TYPE_LINE** 删除线的一侧（顶部，底部，左侧或右侧）。

`lv_draw_line` 使用其中的 4 个。本质上，每条（倾斜）线通过形成一个矩形以 4 个线罩为边界。

- **LV_DRAW_MASK_TYPE_RADIUS** 删除也可以具有半径的矩形的内部或外部。也可以通过将半径设置为大值（`LV_RADIUS_CIRCLE`）来创建圆

- **LV_DRAW_MASK_TYPE_ANGLE** 删除圆形扇区。它用于 `lv_draw_arc` 删除“空”扇区。

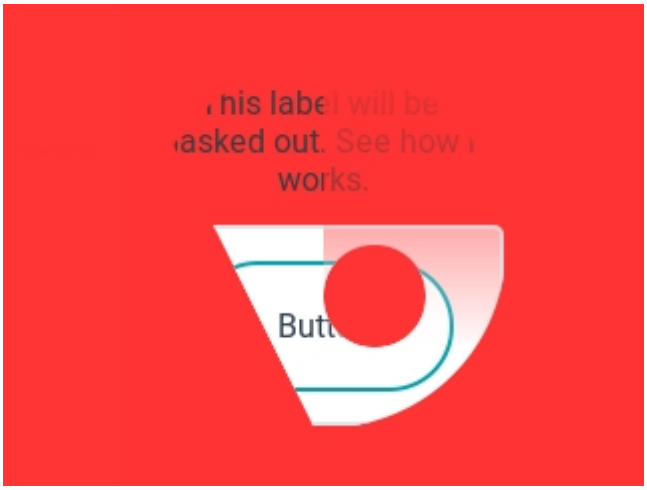
- **LV_DRAW_MASK_TYPE_FADE** 创建垂直淡入淡出（更改不透明度）
- **LV_DRAW_MASK_TYPE_MAP** 遮罩存储在数组中，并应用了必要的部分

在绘制过程中会自动创建和删除蒙版，但是 lv_objmask 允许用户添加蒙版。这是一个例子：

Cai Xuefeng

C

几个对象蒙版



源码：

```
#include "../../../lv_examples.h"
#if LV_USE_OBJMASK
void lv_ex_objmask_1(void)
{
    /*Set a very visible color for the screen to clearly see what happens*/
    lv_obj_set_style_local_bg_color(lv_scr_act(), LV_OBJ_PART_MAIN, LV_STATE_DEFAULT,␣
```

```
,!lv_color_hex3(0xf33));
lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
lv_obj_set_size(om, 200, 200);
lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
lv_obj_t * label = lv_label_create(om, NULL);
lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK);
lv_label_set_align(label, LV_LABEL_ALIGN_CENTER);
lv_obj_set_width(label, 180);
lv_label_set_text(label, "This label will be masked out. See how it works.");
lv_obj_align(label, NULL, LV_ALIGN_IN_TOP_MID, 0, 20);
lv_obj_t * cont = lv_cont_create(om, NULL);
lv_obj_set_size(cont, 180, 100);
lv_obj_set_drag(cont, true);
lv_obj_align(cont, NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -10);
lv_obj_t * btn = lv_btn_create(cont, NULL);
lv_obj_align(btn, NULL, LV_ALIGN_CENTER, 0, 0);
lv_obj_set_style_local_value_str(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, "Button
,!");
uint32_t t;
lv_refr_now(NULL);
t = lv_tick_get();
while(lv_tick_elaps(t) < 1000);
lv_area_t a;
lv_draw_mask_radius_param_t r1;
a.x1 = 10;
a.y1 = 10;
a.x2 = 190;
a.y2 = 190;
lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, false);
lv_objmask_add_mask(om, &r1);
lv_refr_now(NULL);
t = lv_tick_get();
while(lv_tick_elaps(t) < 1000);
a.x1 = 100;
a.y1 = 100;
a.x2 = 150;
a.y2 = 150;
lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, true);
lv_objmask_add_mask(om, &r1);
lv_refr_now(NULL);
```

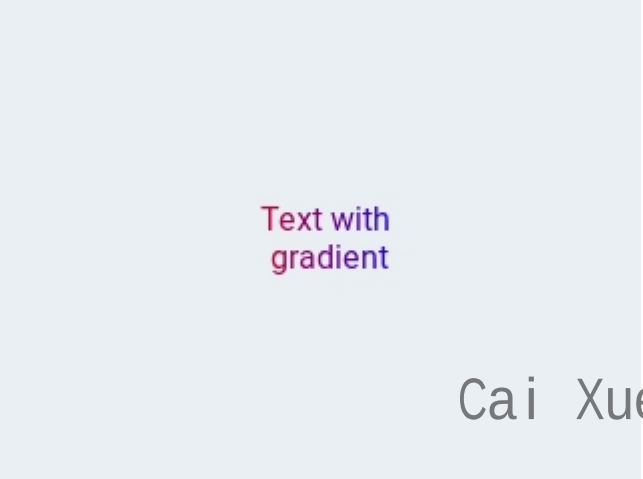

(continues on next page)

(continued from previous page)

```
t = lv_tick_get();
while(lv_tick_elaps(t) < 1000);
lv_draw_mask_line_param_t l1;
lv_draw_mask_line_points_init(&l1, 0, 0, 100, 200, LV_DRAW_MASK_LINE_SIDE_TOP);
lv_objmask_add_mask(om, &l1);
lv_refr_now(NULL);
```

```
        t = lv_tick_get();
        while(lv_tick_elaps(t) < 1000);
        lv_draw_mask_fade_param_t f1;
        a.x1 = 100;
        a.y1 = 0;
        a.x2 = 200;
        a.y2 = 200;
        lv_draw_mask_fade_init(&f1, &a, LV_OPA_TRANSP, 0, LV_OPA_COVER, 150);
        lv_objmask_add_mask(om, &f1);
}
#endif
```

文字遮罩



源码：

```
    #include "../../../lv_examples.h"
#if LV_USE_OBJMASK
#define MASK_WIDTH 100
#define MASK_HEIGHT 50

void lv_ex_objmask_2(void)
{
        /* Create the mask of a text by drawing it to a canvas*/
        static lv_opa_t mask_map[MASK_WIDTH * MASK_HEIGHT];
        /*Create a "8 bit alpha" canvas and clear it*/
        lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);
        lv_canvas_set_buffer(canvas, mask_map, MASK_WIDTH, MASK_HEIGHT, LV_IMG_CF_ALPHA_
        ,!8BIT);
        lv_canvas_fill_bg(canvas, LV_COLOR_BLACK, LV_OPA_TRANSP);
        /*Draw a label to the canvas. The result "image" will be used as mask*/
        lv_draw_label_dsc_t label_dsc;
        lv_draw_label_dsc_init(&label_dsc);
        label_dsc.color = LV_COLOR_WHITE;
        lv_canvas_draw_text(canvas, 5, 5, MASK_WIDTH, &label_dsc, "Text with gradient",␣
        ,!LV_LABEL_ALIGN_CENTER);
        /*The mask is reads the canvas is not required anymore*/
```

```
    lv_obj_del(canvas);
    /*Create an object mask which will use the created mask*/
    lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
    lv_obj_set_size(om, MASK_WIDTH, MASK_HEIGHT);
    lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
    /*Add the created mask map to the object mask*/
    lv_draw_mask_map_param_t m;
    lv_area_t a;
    a.x1 = 0;
    a.y1 = 0;
    a.x2 = MASK_WIDTH - 1;
    a.y2 = MASK_HEIGHT - 1;
    lv_draw_mask_map_init(&m, &a, mask_map);
    lv_objmask_add_mask(om, &m);
    /*Create a style with gradient*/
    static lv_style_t style_bg;
    lv_style_init(&style_bg);
    lv_style_set_bg_opa(&style_bg, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_bg_grad_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_bg_grad_dir(&style_bg, LV_STATE_DEFAULT, LV_GRAD_DIR_HOR);
    /* Create and object with the gradient style on the object mask.
    * The text will be masked from the gradient*/
    lv_obj_t * bg = lv_obj_create(om, NULL);
    lv_obj_reset_style_list(bg, LV_OBJ_PART_MAIN);
    lv_obj_add_style(bg, LV_OBJ_PART_MAIN, &style_bg);
    lv_obj_set_size(bg, MASK_WIDTH, MASK_HEIGHT);
}
#endif
```

第八章 基础对象（lv_obj）

8.1 总览

“基础对象”实现了屏幕上小部件的基本属性，例如：

- 坐标
- 父对象
- 孩子们
- 主要风格
- 诸如单击启用，拖动启用等属性。

在面向对象的思想中，它是继承 LVGL 中所有其他对象的基类。这尤其有助于减少代码重复。

Base 对象的功能也可以与其他小部件一起使用。例如

```
lv_obj_set_width(slider, 100)
```

Base 对象可以直接用作简单的小部件。然后就是矩形。

8.1.1 坐标

8.1.2 尺寸

可以使用和在单个轴上修改对象大小，或者可以同时修改两个轴的对象大小。

```
lv_obj_set_width(obj, new_width)
lv_obj_set_height(obj, new_height)
lv_obj_set_size(obj, new_width, new_height)
```

样式可以向对象添加边距。Margin 说“我想要我周围的空间”。设置宽度或高度减少边距或。更确切的方法是：。

```
lv_obj_set_width_margin(obj, new_width)
lv_obj_set_height_margin(obj, new_height)
new_width = left_margin + object_width + right_margin
```

要获取包括边距的宽度或高度，请使用 `lv_obj_get_width/height_margin(obj)` 。

样式也可以向对象添加填充。填充的意思是“我不要我的孩子们离我的身体太近，所以要保留这个空间”。设置通过 padding 或减小的宽度或高度。更精确的方式： 要使用 padding 减小宽度或高度，请使用。可以认为是“对象的有用大小”。

```
lv_obj_set_width_fit(obj, new_width)
lv_obj_set_height_fit(obj, new_height)
new_width = left_pad + object_width + right_pad
lv_obj_get_width/height_fit(obj)
```

当其他窗口小部件使用布局或自动调整时，边距和填充变得很重要。

8.1.3 位置

您可以设置 x 和 y 父坐标与相对而在同一时间，或两者。

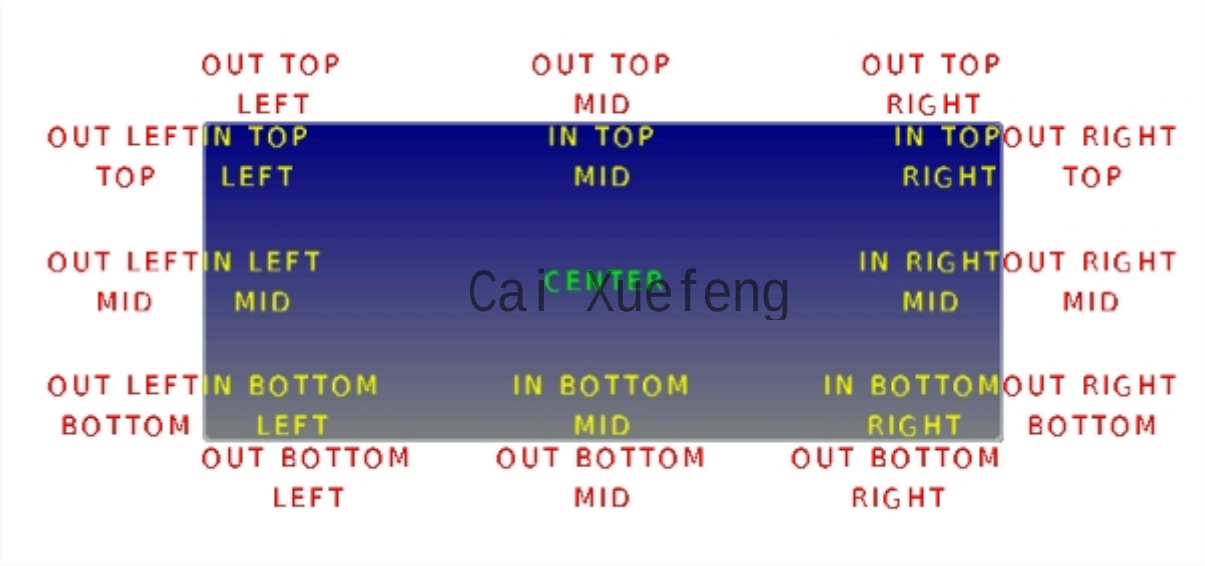
lv_obj_set_x(obj, new_x)
lv_obj_set_y(obj, new_y)
lv_obj_set_pos(obj, new_x, new_y)

8.1.4 对准

您可以使用将对象与另一个对齐。

lv_obj_align(obj, obj_ref, LV_ALIGN_..., x_ofs, y_ofs)

- `obj` 是要对齐的对象。
- `obj_ref` 是参考对象。 `obj` 将与之对齐。如果为，则将使用的父项。 `obj_ref = NULLobj`
- 第三个参数是对齐方式的*类型*。这些是可能的选项：



对齐类型的构建方式类似于 `LV_ALIGN_OUT_TOP_MID` 。

- 最后两个参数允许您在对齐对象后将其移动指定数量的像素。

例如，要在图像下方对齐文本：。或在其父级中间对齐文本：。
lv_obj_align(text, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 10)
lv_obj_align(text, NULL, LV_ALIGN_CENTER, 0, 0)

`lv_obj_align_origo` 的工作原理类似于， `lv_obj_align` 但它使对象的中心对齐。

例如，将按钮的中心对准图像的底部。lv_obj_align_origo(btn, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 0)

如果 `LV_USE_OBJ_REALIGN` 在 *lv_conf.h* 中启用了路线的参数，则会将其保存在对象中。然后，您只需调用即可重新对齐对象 `lv_obj_realign(obj)` 。等效于 `lv_obj_align` 使用相同的参数再次调用。

如果对齐发生在 `lv_obj_align_origo` ，则将在对象重新对齐时使用。

的 lv_obj_align_x/y 和 lv_obj_align_origo_x/y 功能可用于吨对准仅在一个轴上。

如果使用，则如果对象的大小在功能上发生变化，则对象将自动重新对齐。当将尺寸动画应用于对象并且需要保留原始位置时，这非常有用。

lv_obj_set_auto_realign(obj, true)
lv_obj_set_width/height/size()

请注意，屏幕的坐标无法更改。尝试在屏幕上使用这些功能将导致不确定的行为。

8.1.5 父类与子类

您可以使用设置对象的新父对象。要获取当前的父级，请使用。lv_obj_set_parent(obj, new_parent)

lv_obj_get_parent(obj)

要获取对象的子类，请使用（从最后到第一个）或（从第一个到最后）。要获得第一个孩子，请作为第二个参数传递，并使用返回值遍历这些孩子。如果没有更多的孩子，该函数将返回。例如：

lv_obj_get_child(obj, child_prev)

lv_obj_get_child_back(obj, child_prev)


```
lv_obj_t * child = lv_obj_get_child(parent, NULL);
while(child) {
    /*Do something with "child" */
    child = lv_obj_get_child(parent, child);
}
```

lv_obj_count_children(obj)告诉对象上子类的数量。lv_obj_count_children_recursive(obj)还告诉孩子的数量，但递归计算孩子的孩子数。

8.1.6 屏风

创建像这样的屏幕时，可以用加载它。该功能为您提供了指向当前屏幕的指针。

lv_obj_t * screen = lv_obj_create(NULL, NULL)

lv_scr_load(screen)lv_scr_act()

如果您有更多的显示，那么重要的是要知道这些功能在最后创建的或明确选择的（带有`lv_disp_set_default`）显示上起作用。

要获取对象的屏幕，请使用 lv_obj_get_screen(obj) 函数。

8.1.7 层数

有两个自动生成的层：

- 顶层
- 系统层

它们独立于屏幕，并且将显示在每个屏幕上。所述顶层是在屏幕上的每个对象的上方和系统层是上述顶层太。您可以将任何弹出窗口自由添加到顶层。但是，系统层仅限于系统级的内容（例如，鼠标光标将放在此处`lv_indev_set_cursor()`）。

的 `lv_layer_top()` 和 `lv_layer_sys()` 功能给出了一个指针到顶部或系统层。

你可以把一个对象的前景或将其与发送到后台 `lv_obj_move_foreground(obj)` 和`lv_obj_move_background(obj)`。

8.1.8 事件

要为对象设置事件回调，请使用，`lv_obj_set_event_cb(obj, event_cb)`

要将事件手动发送给对象，请使用 `lv_event_send(obj, LV_EVENT_..., data)`

Cai Xuefeng

8.2 小部件

小部件可以包含多个部分。例如，按钮仅具有主要部分，而滑块则由背景，指示器和旋钮组成。

各部分的名称构造如下。例如或。通常在将样式添加到对象时使用小部件。使用小部件可以将不同的样式分配给对象的不同小部件。LV_ + <TYPE> _PART_ <NAME>LV_BTN_PART_MAINLV_SLIDER_PART_KNOB

8.2.1 状态

该对象可以处于以下状态的组合：

- **LV_STATE_DEFAULT** 正常，已发布
- **LV_STATE_CHECKED** 切换或选中
- **LV_STATE_FOCUSED** 通过键盘或编码器聚焦或通过触摸板/鼠标单击
- **LV_STATE_EDITED** 由编码器编辑
- **LV_STATE_HOVERED** 鼠标悬停（现在不支持）
- **LV_STATE_PRESSED** 已按下
- **LV_STATE_DISABLED** 禁用或不活动

当用户按下，释放，聚焦等对象时，状态通常由库自动更改。但是，状态也可以手动更改。要完全覆盖当前状态，请使用。要设置或清除给定状态（但保持其他状态不变），在两种情况下都可以使用 ORed 状态值。例如。

lv_obj_set_state(obj, part, LV_STATE...)

lv_obj_add/clear_state(obj, part, LV_STATE_...)

lv_obj_set_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)

8.2.2 样式

要向对象添加样式，请使用功能。基础对象使用所有类似矩形的样式属性。

lv_obj_add_style(obj, part, &new_style)

要从对象中删除所有样式，请使用 lv_obj_reset_style_list(obj, part)

如果修改对象已经使用的样式，则可以刷新 lv_obj_refresh_style(obj) 每个对象使用的样式，或者使用给定样式 use 通知所有对象，以刷新受影响的对象 lv_obj_report_style_mod(&style)。如果参数 lv_obj_report_style_mod 为 NULL，则将通知所有对象。

8.2.3 属性

有一些属性可以通过以下方式启用/禁用：lv_obj_set_...(obj, true/false)

- **隐藏** -隐藏对象。它不会被绘制，输入设备会将其视为不存在。它的子级也将被隐藏。
- **点击** -允许您通过输入设备单击对象。如果禁用，则单击事件将传递到此事件后面的对象。（例如，默认情况下不可点击标签）
- **顶部** -如果启用，则单击此对象或其任何子级时，该对象将进入前台。
- **拖动** -启用拖动（通过输入设备移动）
- **drag_dir**-仅在特定方向上启用拖动。可以 `LV_DRAG_DIR_HOR/VER/ALL`。
- **drag_throw**-通过拖动启用“ throwing”，好像对象将具有动量
- **drag_parent**-如果启用，则在拖动过程中将移动对象的父对象。看起来就像拖动父级。递归检查，因此也可以传播给祖父母。
- **parent_event**-也将事件传播给父母。递归检查，因此也可以传播给祖父母。
- **opa_scale_enable**-启用不透明度缩放。请参见[＃opa-scale]（Opa 比例尺）部分。

8.2.4 保护

库中有一些自动发生的特定操作。为防止一种或多种此类行为，您可以保护对象免受它们侵害。存在以下保护：

- **LV_PROTECT_NONE** 没有保护
- **LV_PROTECT_POS** 防止自动定位（例如，容器中的布局）
- **LV_PROTECT_FOLLOW** 防止在自动排序（例如，容器布局）中遵循对象（进行“换行”）
- **LV_PROTECT_PARENT** 防止自动更改父项。（例如，Page 将在背景上创建的子代移动到可滚动页面）

- **LV_PROTECT_PRESS_LOST** 当按机滑出对象时，防止失去按机。（例如，如果按下某个按钮，则可以将其释放）
- **LV_PROTECT_CLICK_FOCUS** 如果对象在*组中*并且启用了单击焦点，则防止其自动聚焦。
- **LV_PROTECT_CHILD_CHG** 禁用子更改信号。库内部使用

该套/清除保护。您也可以使用保护类型的“*或*”值。`lv_obj_add/clear_protect(obj, LV_PROTECT_...)`

8.2.5 团体

一旦将一个对象添加到组中，可以使用获取该对象的当前组。
lv_group_add_obj(group, obj)
lv_obj_get_group(obj)

`lv_obj_is_focused(obj)`告诉对象当前是否集中在其组上。如果该对象未添加到组中，`false`将返回。

8.2.6 扩展点击区域

默认情况下，只能在对象的坐标上单击对象，但是可以使用扩展该区域。 描述可点击区域应在每个方向上超出默认设置的范围。lv_obj_set_ext_click_area(obj, left, right, top, bottom)left/right/top/bottom

需要在 *lv_conf.h* 中使用启用此功能 `LV_USE_EXT_CLICK_AREA`。可能的值为：

- **LV_EXT_CLICK_AREA_FULL** 将所有 4 个坐标存储为 `lv_coord_t`
- **LV_EXT_CLICK_AREA_TINY** 仅将水平和垂直坐标（使用左/右和上/下的较大值）存储为 `uint8_t`
- **LV_EXT_CLICK_AREA_OFF** 禁用此功能

8.3 事件

仅通用事件是按对象类型发送的。

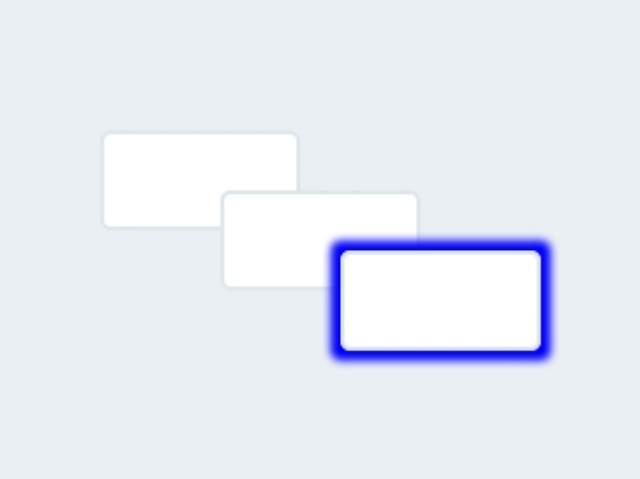
8.4 按键

对象类型不处理任何*键*。

例

C

具有自定义样式的基本对象



API

typedef

typedef uint8_t lv_design_mode_t

typedef uint8_t lv_design_res_t

typedef lv_design_res_t（* lv_design_cb_t）（*struct* _lv_obj_t * obj，*const*lv_area_t * clip_area，lv_design_mode_t 模式）

设计回调用于在屏幕上绘制对象。它接受对象，遮罩区域以及绘制对象的方式。

typedef uint8_t lv_event_t

发送到对象的事件类型。

typedef void（* lv_event_cb_t）（*struct* _lv_obj_t * obj，lv_event_t event ）

事件回调。事件用于通知用户对该对象采取的某些操作。有关详细信息，请参见 lv_event_t。

typedef uint8_t lv_signal_t

typedef lv_res_t（* lv_signal_cb_t）（*struct* _lv_obj_t * obj，lv_signal_t sign，void* param ）

typedef uint8_t lv_protect_t

typedef uint8_t lv_state_t

typedef struct _lv_obj_tlv_obj_t

typedef uint8_t lv_obj_part_t

枚举

Enum [anonymous]

设计模式

值：

enumerator **LV_DESIGN_DRAW_MAIN**

绘制对象的主要部分

enumerator **LV_DESIGN_DRAW_POST**

在对象上画画

enumerator **LV_DESIGN_COVER_CHK**

检查对象是否完全覆盖“ mask_p”区域

Enum [anonymous]

设计结果

值：

enumerator **LV_DESIGN_RES_OK**

准备好

enumerator **LV_DESIGN_RES_COVER**

`LV_DESIGN_COVER_CHK` 如果区域完全覆盖，则返回

enumerator **LV_DESIGN_RES_NOT_COVER**

`LV_DESIGN_COVER_CHK` 如果未覆盖区域，则返回

enumerator **LV_DESIGN_RES_MASKED**

`LV_DESIGN_COVER_CHK` 如果区域被遮盖，则返回（孩子也不会遮盖）

Enum [anonymous]

值：

enumerator **LV_EVENT_PRESSED**

该对象已被按下

enumerator **LV_EVENT_PRESSING**

按下对象（按下时连续调用）

enumerator **LV_EVENT_PRESS_LOST**

用户仍在按，但将光标/手指滑离对象

enumerator **LV_EVENT_SHORT_CLICKED**

用户在短时间内按下对象，然后释放它。如果拖动则不调用。

enumerator **LV_EVENT_LONG_PRESSED**

对象已被按下至少 `LV_INDEV_LONG_PRESS_TIME`。如果拖动则不调用。

enumerator **LV_EVENT_LONG_PRESSED_REPEAT**

`LV_INDEV_LONG_PRESS_TIME` 每 `LV_INDEV_LONG_PRESS_REP_TIME` 毫秒调用一次。如果拖动则不调用。

enumerator **LV_EVENT_CLICKED**

如果没有拖动则调用释放（无论长按）

enumerator **LV_EVENT_RELEASED**

在对象释放后的每种情况下调用

enumerator **LV_EVENT_DRAG_BEGIN**

enumerator **LV_EVENT_DRAG_END**

enumerator **LV_EVENT_DRAG_THROW_BEGIN**

enumerator **LV_EVENT_GESTURE**

对象已被手势

enumerator **LV_EVENT_KEY**

enumerator **LV_EVENT_FOCUSED**

enumerator **LV_EVENT_DEFOCUSED**

enumerator **LV_EVENT_LEAVE**

enumerator **LV_EVENT_VALUE_CHANGED**

对象的值已更改（即，滑块已移动）

enumerator **LV_EVENT_INSERT**

enumerator **LV_EVENT_REFRESH**

enumerator **LV_EVENT_APPLY**

单击“确定”，“应用”或类似的特定按钮

enumerator **LV_EVENT_CANCEL**

单击了“关闭”，“取消”或类似的特定按钮

enumerator **LV_EVENT_DELETE**

对象被删除

enumerator **_LV_EVENT_LAST**

活动数量

Enum [anonymous]

信号供对象本身使用或扩展对象的功能。应用程序应使用 lv_obj_set_event_cb 来通知对象上发生的事件。

值：

enumerator **LV_SIGNAL_CLEANUP**

对象被删除

enumerator **LV_SIGNAL_CHILD_CHG**

子项已删除/添加

enumerator **LV_SIGNAL_COORD_CHG**

对象坐标/大小已更改

enumerator **LV_SIGNAL_PARENT_SIZE_CHG**

父母的大小已更改

enumerator **LV_SIGNAL_STYLE_CHG**

对象的样式已更改

enumerator **LV_SIGNAL_BASE_DIR_CHG**

基本目录已更改

enumerator **LV_SIGNAL_REFR_EXT_DRAW_PAD**

对象的额外填充已更改

enumerator **LV_SIGNAL_GET_TYPE**

LVGL 需要检索对象的类型

enumerator **LV_SIGNAL_GET_STYLE**

获取对象的样式

enumerator **LV_SIGNAL_GET_STATE_DSC**

获取对象的状态

enumerator **LV_SIGNAL_HIT_TEST**

先进的命中测试

enumerator **LV_SIGNAL_PRESSED**

该对象已被按下

enumerator **LV_SIGNAL_PRESSING**

按下对象（按下时连续调用）

enumerator **LV_SIGNAL_PRESS_LOST**

用户仍在按，但将光标/手指滑离对象

enumerator **LV_SIGNAL_RELEASED**

用户在短时间内按下对象，然后释放它。如果拖动则不调用。

enumerator **LV_SIGNAL_LONG_PRESS**

对象已被按下至少 `LV_INDEV_LONG_PRESS_TIME`。如果拖动则不调用。

enumerator **LV_SIGNAL_LONG_PRESS_REP**

`LV_INDEV_LONG_PRESS_TIME` 每 `LV_INDEV_LONG_PRESS_REP_TIME` 毫秒调用一次。如果拖动则不调用。

enumerator **LV_SIGNAL_DRAG_BEGIN**

enumerator **LV_SIGNAL_DRAG_THROW_BEGIN**

enumerator **LV_SIGNAL_DRAG_END**

enumerator **LV_SIGNAL_GESTURE**

对象已被手势

enumerator **LV_SIGNAL_LEAVE**

通过输入设备单击或选择另一个对象

enumerator **LV_SIGNAL_FOCUS**

enumerator **LV_SIGNAL_DEFOCUS**

enumerator **LV_SIGNAL_CONTROL**

enumerator **LV_SIGNAL_GET_EDITABLE**

enum **[anonymous]**

值：

enumerator **LV_PROTECT_NONE = 0x00**

enumerator **LV_PROTECT_CHILD_CHG = 0x01**

禁用儿童更换信号。被库使用

enumerator **LV_PROTECT_PARENT = 0x02**

防止自动更改父级（例如，在 lv_page 中）

enumerator **LV_PROTECT_POS = 0x04**

防止自动定位（例如，在 lv_cont 布局中）

enumerator **LV_PROTECT_FOLLOW = 0x08**

防止在自动排序中遵循对象（例如，在 lv_cont PRETTY 布局中）

enumerator **LV_PROTECT_PRESS_LOST = 0x10**

如果 `indev` 按的是该对象，但在按时却扫出，请勿搜索其他对象。

enumerator **LV_PROTECT_CLICK_FOCUS = 0x20**

单击以防止聚焦对象

Enum [anonymous]

值：

enumerator **LV_STATE_DEFAULT = 0x00**

enumerator **LV_STATE_CHECKED = 0x01**

enumerator **LV_STATE_FOCUSED = 0x02**

enumerator **LV_STATE_EDITED = 0x04**

enumerator **LV_STATE_HOVERED = 0x08**

enumerator **LV_STATE_PRESSED = 0x10**

enumerator **LV_STATE_DISABLED = 0x20**

enum [anonymous]

值：

enumerator **LV_OBJ_PART_MAIN**

enumerator **_LV_OBJ_PART_VIRTUAL_LAST = _LV_OBJ_PART_VIRTUAL_FIRST**

enumerator **_LV_OBJ_PART_REAL_LAST = _LV_OBJ_PART_REAL_FIRST**

enumerator **LV_OBJ_PART_ALL = 0xFF**

功能

voidlv_init（ void ）

在里面。“ lv”库。

voidlv_deinit（ void ）

取消初始化“ lv”库当前仅在不使用自定义分配器或启用 GC 时实现。

lv_obj_t * lv_obj_create（lv_obj_t * *parent*，*const*lv_obj_t * *copy*）

创建一个基本对象

返回

指向新对象的指针

参数

- `parent`：指向父对象的指针。如果为 NULL，则将创建一个屏幕
- `copy`：指向基础对象的指针，如果不为 NULL，则将从其复制新对象

lv_res_t lv_obj_del（lv_obj_t * *obj* ）

删除“ obj”及其所有子项

返回

LV_RES_INV，因为对象已删除

参数

- `obj`：指向要删除的对象的指针

voidlv_obj_del_anim_ready_cb（lv_anim_t * *a* ）

动画就绪回调中易于使用的函数，可在动画就绪时删除对象

参数

- `a`：指向动画的指针

voidlv_obj_del_async（*struct* _lv_obj_t * *obj* ）

辅助函数，用于异步删除对象。在无法直接在 `LV_EVENT_DELETE` 处理程序（即父级）中删除对象的情况下很有用。

看到

lv_async_call

参数

- `obj`：要删除的对象

voidlv_obj_clean（lv_obj_t * *obj* ）

删除对象的所有子对象

参数

- `obj`：指向对象的指针

voidlv_obj_invalidate_area（*const*lv_obj_t * *obj*，*const*lv_area_t * *area*）

将对象的区域标记为 void。此区域将由“ lv_refr_task”重绘

参数

- `obj`：指向对象的指针
- `area`：要重绘的区域

void lv_obj_invalidate（const lv_obj_t * obj ）

将对象标记为 void，因此将通过“ lv_refr_task”重绘其当前位置

参数

- `obj`：指向对象的指针

bool lv_obj_area_is_visible（const lv_obj_t * obj，lv_area_t * area ）

判断对象的某个区域现在是否可见（甚至部分可见）

返回

true：可见；false：不可见（隐藏，在父级之外，在其他屏幕上等）

参数

- `obj`：指向对象的指针
- `area`：检查。该区域的可见部分将写回到这里。

bool lv_obj_is_visible（const lv_obj_t * obj ）

告诉对象现在是否可见（甚至部分可见）

返回

true：可见；false：不可见（隐藏，在父级之外，在其他屏幕上等）

参数

- `obj`：指向对象的指针

void lv_obj_set_parent（lv_obj_t * obj，lv_obj_t * parent）

为对象设置一个新的父对象。其相对位置将相同。

参数

- `obj`：指向对象的指针。不能是屏幕。
- `parent`：指向新父对象的指针。（不能为 NULL）

void lv_obj_move_foreground（lv_obj_t * obj ）

移动并反对前景

参数

- `obj`：指向对象的指针

void lv_obj_move_background（lv_obj_t * obj ）

移动并反对背景

参数

- `obj`：指向对象的指针

void lv_obj_set_pos（lv_obj_t * obj，lv_coord_t x，lv_coord_t y ）

设置对象的相对位置（相对于父对象）

参数

- `obj`：指向对象的指针
- `x`：距父级左侧的新距离
- `y`：距离父级顶部的新距离

void lv_obj_set_x（lv_obj_t * obj，lv_coord_t x ）

设置对象的 x 坐标

参数

- `obj`：指向对象的指针
- `x`：从父级到左侧的新距离

void lv_obj_set_y（lv_obj_t * obj，lv_coord_t y ）

设置对象的 y 坐标

参数

- `obj`：指向对象的指针
- `y`：距离父级顶部的新距离

void lv_obj_set_size（lv_obj_t * obj，lv_coord_t w，lv_coord_t h ）

设置对象的大小

参数

- `obj`：指向对象的指针
- `w`：新宽度
- `h`：新高度

void lv_obj_set_width（lv_obj_t * obj，lv_coord_t w ）

设置对象的宽度

参数

- `obj`：指向对象的指针
- `w`：新宽度

void lv_obj_set_height（lv_obj_t * obj，lv_coord_t h ）

设定物件的高度

参数

- `obj`：指向对象的指针
- `h`：新高度

void lv_obj_set_width_fit（lv_obj_t * obj，lv_coord_t w ）

设置通过左右填充减少的宽度。

参数

- `obj`：指向对象的指针
- `w`：没有填充的宽度

void lv_obj_set_height_fit（lv_obj_t * obj，lv_coord_t h ）

设置通过顶部和底部填充减小的高度。

参数

- `obj`：指向对象的指针
- `h`：不带衬垫的高度

void lv_obj_set_width_margin（lv_obj_t * obj，lv_coord_t w ）

通过考虑左边距和右边距来设置对象的宽度。物体的宽度为 `obj_w = w - margin_left - margin_right`

参数

- `obj`：指向对象的指针
- `w`：包括边距在内的新高度

void lv_obj_set_height_margin（lv_obj_t * obj，lv_coord_t h ）

通过考虑顶部和底部边距来设置对象的高度。物体高度将为 `obj_h = h - margin_top - margin_bottom`

参数

- `obj`：指向对象的指针
- `h`：包括边距在内的新高度

void lv_obj_align（lv_obj_t * obj，const lv_obj_t * base，lv_align_t align，lv_coord_t x_ofs，lv_coord_t y_ofs ）

将一个对象与另一个对象对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。
- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `x_ofs`：对齐后的 x 坐标偏移
- `y_ofs`：对齐后的 y 坐标偏移

void lv_obj_align_x（lv_obj_t * obj，const lv_obj_t * base，lv_align_t align，lv_coord_t x_ofs ）

将一个对象与另一个对象水平对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。
- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `x_ofs`：对齐后的 x 坐标偏移

void lv_obj_align_y（lv_obj_t * obj，const lv_obj_t * base，lv_align_t align，lv_coord_t y_ofs ）

将一个对象与另一个对象垂直对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。
- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `y_ofs`：对齐后的 y 坐标偏移

void lv_obj_align_mid（lv_obj_t * *obj*，const lv_obj_t * *base*，lv_align_t *align*，lv_coord_t *x_ofs*，lv_coord_t *y_ofs* ）

将一个对象与另一个对象对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。
- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `x_ofs`：对齐后的 x 坐标偏移
- `y_ofs`：对齐后的 y 坐标偏移

void lv_obj_align_mid_x（lv_obj_t * *obj*，const lv_obj_t * *base*，lv_align_t *align*，lv_coord_t *x_ofs* ）

将对象的中点与另一个对象水平对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。
- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `x_ofs`：对齐后的 x 坐标偏移

void lv_obj_align_mid_y（lv_obj_t * *obj*，const lv_obj_t * *base*，lv_align_t *align*，lv_coord_t *y_ofs* ）

将对象的中点与另一个对象垂直对齐。

参数

- `obj`：指向要对齐的对象的指针
- `base`：指向对象的指针（如果为 NULL，则使用父对象）。'obj'将与其对齐。

- `align`：对齐类型（请参见“ lv_align_t”枚举）
- `y_ofs`：对齐后的 y 坐标偏移

void lv_obj_realign（lv_obj_t * *obj* ）

根据最后一个 `lv_obj_align` 参数重新对齐对象。

参数

- `obj`：指向对象的指针

void lv_obj_set_auto_realign（lv_obj_t * *obj*，bool *en* ）

当对象的大小根据最后一个 `lv_obj_align` 参数更改时，启用对象的自动重新对齐。

参数

- `obj`：指向对象的指针
- `en`：true：启用自动重新对齐；false：禁用自动重新对齐

void lv_obj_set_ext_click_area（lv_obj_t * *OBJ*，lv_coord_t *left*，lv_coord_t *right*，lv_coord_t *top*，lv_coord_t *bottom*）

设置扩展的可点击区域的大小

参数

- `obj`：指向对象的指针
- `left`：可扩展点击位于左侧[px]
- `right`：可扩展点击位于右侧[px]
- `top`：扩展可点击位于顶部[px]
- `bottom`：可扩展点击位于底部[px]

void lv_obj_add_style（lv_obj_t * *obj*，uint8_t *part*，lv_style_t * *style* ）

将新样式添加到对象的样式列表。

参数

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`

- `style`：指向要添加的样式的指针（仅会保存其指针）

void lv_obj_remove_style（lv_obj_t * *obj*，uint8_t *part*，lv_style_t * *style* ）

从对象的样式列表中删除样式。

参数

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`
- `style`：指向要删除的样式的指针

void lv_obj_clean_style_list（lv_obj_t * *obj*，uint8_t *part* ）

将样式重置为默认（空）状态。释放所有已使用的内存，并取消待处理的相关转换。通常在`LV_SIGN_CLEAN_UP 中使用。

参数

- `obj`：指向对象的指针
- `part`：对象的应重置样式列表的部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`

void lv_obj_reset_style_list（lv_obj_t * *obj*，uint8_t *part* ）

将样式重置为默认（空）状态。释放所有已使用的内存，并取消待处理的相关转换。还通知对象有关样式更改的信息。

参数

- `obj`：指向对象的指针
- `part`：对象的应重置样式列表的部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`

void lv_obj_refresh_style（lv_obj_t * *obj*，uint8_t *part*，lv_style_property_t *prop* ）

通知对象（及其子对象）其样式已修改

参数

- `obj`：指向对象的指针
- `prop`： `LV_STYLE_PROP_ALL` 或 `LV_STYLE_...` 财产。它用于优化需要刷新的内容。

void lv_obj_report_style_mod（ lv_style_t * *style*）

修改样式时通知所有对象

参数

- `style`：指向样式的指针。仅具有这种样式的对象将被通知（NULL 通知所有对象）

void _lv_obj_set_style_local_color（lv_obj_t * *OBJ*，uint8_t *type*，lv_style_property_t *prop*，lv_color_t *color*）

在给定状态下设置对象的一部分的局部样式属性。

注意

不能直接使用。使用 specifc 属性获取函数。例如:: `lv_obj_style_get_border_opa()`

注意

出于性能原因，不检查属性是否确实具有颜色类型

参数

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`

- `prop`：样式属性与状态进行或运算。例如 `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`：要设置的值

void _lv_obj_set_style_local_int（lv_obj_t * *OBJ*，uint8_t *type*，lv_style_property_t *prop*，lv_style_int_t *value*）

在给定状态下设置对象的一部分的局部样式属性。

注意

不应直接使用。请改用特定的属性 get 函数。例如： `lv_obj_style_get_border_opa()`

注意

出于性能原因，不检查该属性是否确实具有整数类型

参数

- `obj`：指向对象的指针

- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`
- `prop`：样式属性与状态进行或运算。例如 `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`：要设置的值

void _lv_obj_set_style_local_opa（lv_obj_t * *obj*，uint8_t *type*，lv_style_property_t *prop*，lv_opa_t *opa* ）

在给定状态下设置对象的一部分的局部样式属性。

> **注意**
>
> 不应直接使用。请改用特定的属性 get 函数。例如： `lv_obj_style_get_border_opa()`

> **注意**
>
> 出于性能原因，不检查属性是否确实具有不透明度类型

> **参数**

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`
- `prop`：样式属性与状态进行或运算。例如 `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`：要设置的值

void _lv_obj_set_style_local_ptr（lv_obj_t * *obj*，uint8_t *type*，lv_style_property_t *prop*，*const*void* *值*）

在给定状态下设置对象的一部分的局部样式属性。

> **注意**
>
> 不应直接使用。请改用特定的属性 get 函数。例如： `lv_obj_style_get_border_opa()`

> **注意**
>
> 出于性能原因，不检查该属性是否确实具有指针类型

> **参数**

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`
- `prop`：样式属性与状态进行或运算。例如 `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`：要设置的值

bool lv_obj_remove_style_local_prop（lv_obj_t * *obj*，uint8_t *part*，lv_style_property_t *prop* ）

从具有给定状态的对象的一部分中删除局部样式属性。

> **注意**
>
> 不应直接使用。请改用特定的属性删除功能。例如：`lv_obj_style_remove_border_opa()`

> **返回**
>
> true：已找到并删除了该属性；false：找不到该属性

> **参数**

- `obj`：指向对象的指针
- `part`：对象的应删除样式属性的部分。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_PART_MAIN`， `LV_SLIDER_PART_KNOB`
- `prop`：样式属性与状态进行或运算。例如 `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`

void _lv_obj_disable_style_caching（lv_obj_t * *obj*，bool *dis* ）

启用/禁用对象的样式提示

> **参数**

- `obj`：指向对象的指针
- `dis`：true：禁用；false：启用（重新启用）

void lv_obj_set_hidden（lv_obj_t * *obj*，bool *en* ）

隐藏一个对象。它不会显示和单击。

参数

- `obj`：指向对象的指针
- `en`：true：隐藏对象

void lv_obj_set_adv_hittest（lv_obj_t * *obj*，bool *en* ）

设置是否在对象上启用高级命中测试

参数

- `obj`：指向对象的指针
- `en`：true：启用高级命中测试

void lv_obj_set_click（lv_obj_t * *obj*，bool *en* ）

启用或禁用对象单击

参数

- `obj`：指向对象的指针
- `en`：true：使对象可点击

void lv_obj_set_top（lv_obj_t * *obj*，bool *en* ）

如果单击该对象或其任何子对象，则启用该选项可将其置于最前

参数

- `obj`：指向对象的指针
- `en`：true：启用自动顶部功能

void lv_obj_set_drag（lv_obj_t * *obj*，bool *en* ）

启用对象拖动

参数

- `obj`：指向对象的指针
- `en`：true：使对象可拖动

void lv_obj_set_drag_dir（lv_obj_t * *obj*，lv_drag_dir_t *drag_dir* ）

设置可以拖动对象的方向

参数

- `obj`：指向对象的指针

- `drag_dir`：允许的拖动方向的按位或

void lv_obj_set_drag_throw（lv_obj_t * *obj*，bool *en* ）

拖动后启用对象投掷

参数

- `obj`：指向对象的指针
- `en`：true：启用拖动

void lv_obj_set_drag_parent（lv_obj_t * *obj*，bool *en* ）

启用使用父项进行拖动相关的操作。如果尝试拖动对象，则父级将被移动

参数

- `obj`：指向对象的指针
- `en`：true：为对象启用“拖动父对象”

void lv_obj_set_focus_parent（lv_obj_t * *obj*，bool *en* ）

启用以将父级用于焦点状态。聚焦对象时，父级将获得状态（仅可见）

参数

- `obj`：指向对象的指针
- `en`：true：为对象启用“焦点父对象”

void lv_obj_set_gesture_parent（lv_obj_t * *obj*，bool *en* ）

启用将父项用于手势相关的操作。如果尝试手势对象，则将父对象移动

参数

- `obj`：指向对象的指针
- `en`：true：为对象启用“手势父级”

void lv_obj_set_parent_event（lv_obj_t * *obj*，bool *en* ）

也将事件传播给父级

参数

- `obj`：指向对象的指针
- `en`：true：启用事件传播

void lv_obj_set_base_dir（lv_obj_t * *obj*，lv_bidi_dir_t *dir* ）

设置对象的基本方向

参数

- `obj`：指向对象的指针
- `dir`：新的基本方向。`LV_BIDI_DIR_LTR/RTL/AUTO/INHERIT`

void lv_obj_add_protect（lv_obj_t * obj，uint8_t prot ）

在保护字段中设置一个或多个位

参数

- `obj`：指向对象的指针
- `prot`：来自“或”的值 `lv_protect_t`

void lv_obj_clear_protect（lv_obj_t * obj，uint8_t prot ）

清除保护字段中的一个或多个位

参数

- `obj`：指向对象的指针
- `prot`：来自“或”的值 `lv_protect_t`

void lv_obj_set_state（lv_obj_t * obj，lv_state_t state）

设置对象的状态（完全覆盖）。如果在样式中指定，则将从前一个状态开始到当前状态的过渡动画

参数

- `obj`：指向对象的指针
- `state`：新状态

void lv_obj_add_state（lv_obj_t * obj，lv_state_t state）

将一个或多个给定状态添加到对象。其他状态位将保持不变。如果在样式中指定，则将从前一个状态开始到当前状态的过渡动画

参数

- `obj`：指向对象的指针
- `state`：要添加的状态位。例如 `LV_STATE_PRESSED | LV_STATE_FOCUSED`

void lv_obj_clear_state（lv_obj_t * obj，lv_state_t state）

删除对象的给定状态。其他状态位将保持不变。如果在样式中指定，则将从前一个状态开始到当前状态的过渡动画

参数

- `obj`：指向对象的指针
- `state`：要删除的状态位。例如 `LV_STATE_PRESSED | LV_STATE_FOCUSED`

void lv_obj_finish_transitions（lv_obj_t * obj，uint8_t part ）

在对象的一部分上完成所有挂起的过渡

参数

- `obj`：指向对象的指针
- `part`：对象的一部分，例如 `LV_BRN_PART_MAIN` 或 `LV_OBJ_PART_ALL` 所有部分

void lv_obj_set_event_cb（lv_obj_t * obj，lv_event_cb_t event_cb ）

为对象设置事件处理函数。用户用于对对象发生的事件做出反应。

参数

- `obj`：指向对象的指针
- `event_cb`：新事件功能

lv_res_t lv_event_send（lv_obj_t * obj，lv_event_t event，const void* data）

向对象发送事件

返回

LV_RES_OK：`obj` 在事件中没有被删除；LV_RES_INV：`obj` 在事件中被删除

参数

- `obj`：指向对象的指针
- `event`：来自的事件类型 `lv_event_t`。
- `data`：取决于对象类型和事件的任意数据。（通常 `NULL`）

lv_res_t lv_event_send_refresh（lv_obj_t * obj ）

将 LV_EVENT_REFRESH 事件发送到对象

返回

LV_RES_OK：成功，LV_RES_INV：由于该事件，对象变为 void（例如，删除）。

参数

- `obj`：指向对象。（不能为 NULL）

void lv_event_send_refresh_recursive（lv_obj_t * obj ）

将 LV_EVENT_REFRESH 事件发送给对象及其所有子对象

参数

- `obj`：指向一个对象的指针或 NULL 以刷新所有显示的所有对象

lv_res_t lv_event_send_func（lv_event_cb_t event_xcb，lv_obj_t * obj，lv_event_t event，const void* data）

使用对象，事件和数据调用事件函数。

返回

LV_RES_OK：`obj` 在事件中没有被删除；LV_RES_INV：`obj` 在事件中被删除

参数

- `event_xcb`：事件回调函数。如果 `NULL` `LV_RES_OK` 返回则不执行任何操作。（参数名称中的“ x”表示它不是完全通用的函数，因为它不遵循约定）

```
func_name(object, callback, ...)
```


- `obj`：指向与事件关联的对象的指针（可以 `NULL` 简单地调用 `event_cb` ）
- `event`： 一个事件
- `data`：指向自定义数据的指针

const void* lv_event_get_data（void）

获取 `data` 当前事件的参数

返回

该 `data` 参数

void lv_obj_set_signal_cb（lv_obj_t * obj，lv_signal_cb_t signal_cb ）

设置对象的信号功能。由库内部使用。始终在新信号中调用前一个信号功能。

参数

- `obj`：指向对象的指针
- `signal_cb`：新信号功能

lv_res_t lv_signal_send（lv_obj_t * *obj*，lv_signal_t *signal*，void * *param* ）

向对象发送事件

返回

LV_RES_OK 或 LV_RES_INV

参数

- `obj`：指向对象的指针
- `event`：来自的事件类型 `lv_event_t`。

voidlv_obj_set_design_cb（lv_obj_t * *obj*，lv_design_cb_t *design_cb* ）

为对象设置新的设计功能

参数

- `obj`：指向对象的指针
- `design_cb`：新的设计功能

void* lv_obj_allocate_ext_attr（lv_obj_t * *obj*，uint16_t *ext_size* ）

分配一个新的分机。对象的数据

返回

指向分配的 ext 的指针

参数

- `obj`：指向对象的指针
- `ext_size`：新分机的大小。数据

voidlv_obj_refresh_ext_draw_pad（lv_obj_t * *obj* ）

向对象发送“ LV_SIGNAL_REFR_EXT_SIZE”信号以刷新扩展的绘制区域。`lv_obj_invalidate(obj)`此功能后，需要手动使对象 void。

参数

- `obj`：指向对象的指针

lv_obj_t * lv_obj_get_screen（*const*lv_obj_t * *obj* ）

返回对象的屏幕

返回

指向屏幕的指针

参数

- `obj`：指向对象的指针

---

lv_disp_t * lv_obj_get_disp（const lv_obj_t * obj ）

获取对象的显示

返回

指针对象的显示

---

lv_obj_t * lv_obj_get_parent（const lv_obj_t * obj ）

返回对象的父对象

返回

指向“ obj”的父对象的指针

参数

- `obj`：指向对象的指针

---

lv_obj_t * lv_obj_get_child（const lv_obj_t * obj，const lv_obj_t * child ）

遍历对象的子项（从“最新的，最后创建的”开始）

返回

'act_child'之后的子级；如果没有更多子级，则为 NULL

参数

- `obj`：指向对象的指针
- `child`：第一次调用时为 NULL，以获取下一个孩子，以后再返回上一个返回值

---

lv_obj_t * lv_obj_get_child_back（const lv_obj_t * obj，const lv_obj_t * child ）

遍历对象的子项（从“最早的”开始，首先创建）

返回

'act_child'之后的子级；如果没有更多子级，则为 NULL

参数

- `obj`：指向对象的指针
- `child`：第一次调用时为 NULL，以获取下一个孩子，以后再返回上一个返回值

---

uint16_t lv_obj_count_children（const lv_obj_t * obj ）

计算对象的子代（仅直接在“ obj”上的子代）

返回

'obj'的子代号

参数

- `obj`：指向对象的指针

---

uint16_t lv_obj_count_children_recursive（*const*lv_obj_t * *obj* ）

递归计算对象的子代

返回

'obj'的子代号

参数

- `obj`：指向对象的指针

---

void lv_obj_get_coords（*const*lv_obj_t * *obj*，lv_area_t * *cords_p* ）

将对象的坐标复制到区域

参数

- `obj`：指向对象的指针
- `cords_p`：指向存储坐标的区域的指针

---

void lv_obj_get_inner_coords（*const*lv_obj_t * *obj*，lv_area_t * *coords_p* ）

减少由 `lv_obj_get_coords()` 对象的图形可用区域重试的区域。（没有边框的大小或其他额外的图形元素）

参数

- `coords_p`：将结果区域存储在此处

---

lv_coord_t lv_obj_get_x（*const*lv_obj_t * *obj* ）

获取对象的 x 坐标

返回

“ obj”与其父级左侧的距离

参数

- `obj`：指向对象的指针

---

lv_coord_t lv_obj_get_y（*const*lv_obj_t * *obj* ）

获取对象的 y 坐标

返回

“ obj”与其父对象顶部的距离

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_width（const lv_obj_t * obj ）

获取对象的宽度

返回

宽度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_height（const lv_obj_t * obj ）

获取对象的高度

返回

高度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_width_fit（const lv_obj_t * obj ）

通过左右填充减少宽度。

返回

仍然适合容器的宽度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_height_fit（const lv_obj_t * obj ）

通过顶部底部填充减少高度。

返回

仍然适合容器的高度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_height_margin（lv_obj_t * *obj* ）

通过考虑顶部和底部边距来获取对象的高度。返回的高度将是

```
obj_h + margin_top + margin_bottom
```

返回

包括您的边距在内的高度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_width_margin（lv_obj_t * *obj* ）

通过考虑左边距和右边距来获得对象的宽度。返回的宽度将是

```
obj_w + margin_left + margin_right
```

返回

包括您的边距在内的高度

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_width_grid（lv_obj_t * *obj*，uint8_t *div*，uint8_t *span* ）

划分对象的宽度并获得给定列数的宽度。考虑填充。

返回

根据给定参数的宽度

参数

- `obj`：指向对象的指针
- `div`：表示 false 设有多少列。如果设置为 1，则宽度将设置为父级的宽度。如果 2 仅为父级宽度的一半，则为父级的内部填充；如果为 3，则只有第三个父级宽度。
- `span`：合并了多少列

lv_coord_t lv_obj_get_height_grid（lv_obj_t * *obj*，uint8_t *div*，uint8_t *span* ）

划分对象的高度并获得给定列数的宽度。考虑填充。

返回

根据给定参数的高度

参数

- `obj`：指向对象的指针
- `div`：表示 false 设有多少行。如果为 1，则将高度设置为父级的高度。如果 2 为仅父级一半的高度-父级的内部填充如果为 3，仅第三级父级的高度-2 *父级的内部填充
- `span`：合并了多少行

bool lv_obj_get_auto_realign（const lv_obj_t * obj ）

获取对象的自动重新对齐属性。

返回

true：启用自动重新对齐；false：禁用自动重新对齐

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_ext_click_pad_left（const lv_obj_t * obj ）

获取扩展的可点击区域的左侧填充

返回

扩展的左填充

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_ext_click_pad_right（const lv_obj_t * obj ）

获取扩展的可点击区域的正确填充

返回

扩展的右填充

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_ext_click_pad_top（const lv_obj_t * obj ）

获取扩展的可点击区域的顶部填充

返回

扩展的顶部填充

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_ext_click_pad_bottom（const lv_obj_t * obj ）

获取扩展的可点击区域的底部填充

返回

扩展的底部填充

参数

- `obj`：指向对象的指针

lv_coord_t lv_obj_get_ext_draw_pad（const lv_obj_t * obj ）

获取对象的扩展尺寸属性

返回

扩展尺寸属性

参数

- `obj`：指向对象的指针

lv_style_list_t * lv_obj_get_style_list（const lv_obj_t * obj，uint8_t part）

获取对象小部件的样式列表。

返回

指向样式列表的指针。（可以 `NULL`）

参数

- `obj`：指向对象的指针。
- `part`：part 应该获取样式列表的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`

lv_style_int_t _lv_obj_get_style_int（const lv_obj_t * obj，uint8_t part，lv_style_property_t prop ）

获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内

返回

当前状态下给定小部件的属性值。如果找不到该属性，则将返回默认值。

注意

不应直接使用。请改用特定的属性 get 函数。例如：`lv_obj_style_get_border_width()`

注意

出于性能原因，不检查该属性是否确实具有整数类型

参数

- `obj`：指向对象的指针
- `part`：应获取样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`
- `prop`：要获取的属性。例如 `LV_STYLE_BORDER_WIDTH`。对象的状态将在内部添加

---

lv_color_t _lv_obj_get_style_color（*const*lv_obj_t * *obj*，uint8_t *part*，lv_style_property_t *prop* ）

获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内

返回

当前状态下给定小部件的属性值。如果找不到该属性，则将返回默认值。

注意

不应直接使用。请改用特定的属性 get 函数。例如：`lv_obj_style_get_border_color()`

注意

出于性能原因，不检查属性是否确实具有颜色类型

参数

- `obj`：指向对象的指针
- `part`：应获取样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`
- `prop`：要获取的属性。例如 `LV_STYLE_BORDER_COLOR`。对象的状态将在内部添加

---

lv_opa_t _lv_obj_get_style_opa（*const*lv_obj_t * *obj*，uint8_t *part*，lv_style_property_t *prop* ）

获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内

返回

当前状态下给定小部件的属性值。如果找不到该属性，则将返回默认值。

注意

不应直接使用。请改用特定的属性 get 函数。例如：`lv_obj_style_get_border_opa()`

> 注意

出于性能原因，不检查属性是否确实具有不透明度类型

> 参数

- `obj`：指向对象的指针
- `part`：应获取样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`
- `prop`：要获取的属性。例如 `LV_STYLE_BORDER_OPA`。对象的状态将在内部添加

*const*void* _lv_obj_get_style_ptr（*const*lv_obj_t * *obj*，uint8_t *part*，lv_style_property_t *prop* ）

获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内

> 返回

当前状态下给定小部件的属性值。如果找不到该属性，则将返回默认值。

> 注意

不应直接使用。请改用特定的属性 get 函数。例如：`lv_obj_style_get_border_opa()`

> 注意

出于性能原因，不检查该属性是否确实具有指针类型

> 参数

- `obj`：指向对象的指针
- `part`：应获取样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`
- `prop`：要获取的属性。例如 `LV_STYLE_TEXT_FONT`。对象的状态将在内部添加

lv_style_t * lv_obj_get_local_style（lv_obj_t * *obj*，uint8_t *part*）

获取对象部分的局部样式。

> 返回

指向本地样式的指针（如果存在）`NULL`。

参数

- `obj`：指向对象的指针
- `part`：应设置样式属性的对象部分。例如 `LV_OBJ_PART_MAIN`，`LV_BTN_PART_MAIN`，`LV_SLIDER_PART_KNOB`

bool lv_obj_get_hidden（const lv_obj_t * obj）

获取对象的隐藏属性

返回

true：对象被隐藏

参数

- `obj`：指向对象的指针

bool lv_obj_get_adv_hittest（const lv_obj_t * obj）

获取是否在对象上启用了高级命中测试

返回

true：启用高级匹配测试

参数

- `obj`：指向对象的指针

bool lv_obj_get_click（const lv_obj_t * obj）

获取对象的点击启用属性

返回

true：可点击对象

参数

- `obj`：指向对象的指针

bool lv_obj_get_top（const lv_obj_t * obj）

获取对象的顶部启用属性

返回

true：启用自动顶部功能

参数

- `obj`：指向对象的指针

bool lv_obj_get_drag（const lv_obj_t * obj ）

获取对象的拖动启用属性

返回

true：对象可拖动

参数

- `obj`：指向对象的指针

lv_drag_dir_t lv_obj_get_drag_dir（const lv_obj_t * obj ）

获取可以拖动对象的方向

返回

可以将对象拖入允许方向的按位或

参数

- `obj`：指向对象的指针

bool lv_obj_get_drag_throw（const lv_obj_t * obj ）

获取对象的拖动启动属性

返回

true：启用拖动

参数

- `obj`：指向对象的指针

bool lv_obj_get_drag_parent（const lv_obj_t * obj ）

获取对象的拖动父级属性

返回

true：启用拖动父级

参数

- `obj`：指向对象的指针

bool lv_obj_get_focus_parent（const lv_obj_t * obj ）

获取对象的焦点父级属性

返回

true：启用焦点父级

参数

- `obj`：指向对象的指针

`bool lv_obj_get_parent_event（const lv_obj_t * obj ）`

获取对象的拖动父级属性

返回

true：启用拖动父级

参数

- `obj`：指向对象的指针

`bool lv_obj_get_gesture_parent（const lv_obj_t * obj ）`

获取对象的手势父级属性

返回

true：启用手势父级

参数

- `obj`：指向对象的指针

`lv_bidi_dir_t lv_obj_get_base_dir（const lv_obj_t * obj ）`

`uint8_t lv_obj_get_protect（const lv_obj_t * obj ）`

获取对象的保护字段

返回

保护字段（的“或”值 `lv_protect_t`）

参数

- `obj`：指向对象的指针

`bool lv_obj_is_protected（const lv_obj_t * obj，uint8_t prot ）`

检查给定保护位字段的至少一位是否已设置

返回

false：未设置任何给定位，true：至少设置了一位

参数

- `obj`：指向对象的指针
- `prot`：保护要测试的位（的“或”值 `lv_protect_t`）

lv_state_t lv_obj_get_state（const lv_obj_t * obj，uint8_t 部分）

lv_signal_cb_t lv_obj_get_signal_cb（const lv_obj_t * obj ）

获取对象的信号功能

返回

信号功能

参数

- `obj`：指向对象的指针

lv_design_cb_t lv_obj_get_design_cb（const lv_obj_t * obj ）

获取对象的设计功能

返回

设计功能

参数

- `obj`：指向对象的指针

lv_event_cb_t lv_obj_get_event_cb（const lv_obj_t * obj ）

获取对象的事件函数 Cai Xuefeng

返回

事件功能

参数

- `obj`：指向对象的指针

bool lv_obj_is_point_on_coords（lv_obj_t * obj，const lv_point_t * point）

检查给定的屏幕空间点是否在对象的坐标上。

该方法主要用于高级命中测试算法，以检查该点是否在对象内（作为优化）。

参数

- `obj`：要检查的对象
- `point`：屏幕空间点

bool lv_obj_hittest（lv_obj_t * obj，lv_point_t * point ）

对在屏幕空间中特定位置的对象进行命中测试。

返回

如果对象被认为在该点之下，则返回 true

参数

- `obj`：要进行命中测试
- `point`：屏幕空间点

---

void* lv_obj_get_ext_attr（const lv_obj_t * obj ）

获取 ext 指针

返回

ext 指针而不是动态版本将其用作 ext-> data1，而不是 da（ext）-> data1

参数

- `obj`：指向对象的指针

---

void lv_obj_get_type（const lv_obj_t * obj，lv_obj_type_t * buf ）

获取对象及其祖先类型。将其名称以 `type_buf` 当前类型开头。例如 buf.type [0] =“ lv_btn”，buf.type [1] =“ lv_cont”，buf.type [2] =“ lv_obj”

参数

- `obj`：指向应获取类型的对象的指针
- `buf`：指向用于 `lv_obj_type_t` 存储类型的缓冲区的指针

---

lv_obj_user_data_t lv_obj_get_user_data（const lv_obj_t * obj ）

获取对象的用户数据

返回

用户数据

参数

- `obj`：指向对象的指针

---

lv_obj_user_data_t * lv_obj_get_user_data_ptr（const lv_obj_t * obj ）

获取指向对象的用户数据的指针

返回

指向用户数据的指针

参数

- `obj`：指向对象的指针

void lv_obj_set_user_data（lv_obj_t * obj，lv_obj_user_data_t data）

设置对象的用户数据。数据将被复制。

参数

- `obj`：指向对象的指针
- `data`： 用户数据

void* lv_obj_get_group（const lv_obj_t * obj ）

获取对象组

返回

指向对象组的指针

参数

- `obj`：指向对象的指针

bool lv_obj_is_focused（const lv_obj_t * obj ）

判断对象是否是组的聚焦对象。

返回

true：对象已聚焦，false：对象未聚焦或不在组中

参数

- `obj`：指向对象的指针

lv_obj_t * lv_obj_get_focused_obj（const lv_obj_t * obj ）

通过考虑获得真正专注的对象 `focus_parent`。

返回

真正聚焦的对象

参数

- `obj`：起始对象

lv_res_t lv_obj_handle_get_type_signal（lv_obj_type_t * buf，const char* name）

用于信号回调中以处理 `LV_SIGNAL_GET_TYPE` 信号

返回

LV_RES_OK

参数

- `buf`：的指针 `lv_obj_type_t`。（`param`在信号回调中）
- `name`：对象的名称。例如“ lv_btn”。（仅保存指针）

void lv_obj_init_draw_rect_dsc（lv_obj_t * *obj*，uint8_t *类型*，lv_draw_rect_dsc_t * *draw_dsc* ）

根据对象的样式初始化矩形描述符

注意

仅设置相关字段。例如，是否将不评估其他边框属性。`border width == 0`

参数

- `obj`：指向对象的指针
- `type`：样式类型。例如 `LV_OBJ_PART_MAIN`， `LV_BTN_SLIDER_KOB`
- `draw_dsc`：描述符的初始化

void lv_obj_init_draw_label_dsc（lv_obj_t * *obj*，uint8_t *type*，lv_draw_label_dsc_t * *draw_dsc* ）

void lv_obj_init_draw_img_dsc（lv_obj_t * *obj*，uint8_t *part*，lv_draw_img_dsc_t * *draw_dsc* ）

void lv_obj_init_draw_line_dsc（lv_obj_t * *obj*，uint8_t *part*，lv_draw_line_dsc_t * *draw_dsc* ）

lv_coord_t lv_obj_get_draw_rect_ext_pad_size（lv_obj_t * *obj*，uint8_t *part* ）

获取所需的额外大小（围绕对象部分）以绘制阴影，轮廓，值等。

参数

- `obj`：指向对象的指针
- `part`：对象的一部分

void lv_obj_fade_in（lv_obj_t * *obj*，uint32_t *time*，uint32_t *delay*）

使用 `opa_scale` 动画淡入（从透明到完全覆盖）对象及其所有子对象。

参数

- `obj`：淡入的对象
- `time`：动画的持续时间[ms]
- `delay`：等待动画开始播放[ms]

void lv_obj_fade_out（lv_obj_t * obj，uint32_t time，uint32_t delay）

使用 `opa_scale` 动画淡出（从完全覆盖到透明）对象及其所有子对象。

参数

- `obj`：淡入的对象
- `time`：动画的持续时间[ms]
- `delay`：等待动画开始播放[ms]

bool lv_debug_check_obj_type（const lv_obj_t * obj，const char * obj_type ）

检查是否有给定类型的对象

返回

true：有效

参数

- `obj`：指向对象的指针
- `obj_type`：对象的类型。（例如“ lv_btn”）

bool lv_debug_check_obj_valid（const lv_obj_t * obj ）

检查是否还有任何对象“处于活动状态”以及层次 struct 的一部分

返回

true：有效

参数

- `obj`：指向对象的指针
- `obj_type`：对象的类型。（例如“ lv_btn”）

struct lv_realign_t

公众成员

```
const struct _lv_obj_t *base
lv_coord_t xofs
lv_coord_t yofs
lv_align_t align
uint8_t auto_realign
uint8_t mid_align
```

1：origo（对象的中心）与 `lv_obj_align_origo`

---

struct **_lv_obj_t**

公众成员

***struct* _lv_obj_t *parent**

指向父对象的指针

lv_ll_t child_ll

链接列表以存储子对象

lv_area_t coords

对象的坐标（x1，y1，x2，y2）

lv_event_cb_tevent_cb

事件回调函数

lv_signal_cb_tsignal_cb

对象类型特定的信号功能

lv_design_cb_tdesign_cb

对象类型特定的设计功能

void*ext_attr

对象类型特定的扩展数据

lv_style_list_t style_list

uint8_t ext_click_pad_hor

水平方向额外的点击填充

uint8_t ext_click_pad_ver

垂直方向上的额外单击填充

lv_area_t ext_click_pad

额外的单击填充区域。

lv_coord_t ext_draw_pad

在每个方向上扩大尺寸以进行绘制。

uint8_t click

1：可以被输入设备按下

uint8_t drag

1：启用拖动

uint8_t drag_throw

1：启用拖动拖动

uint8_t drag_parent

1：将改为拖动父级

uint8_t hidden

1：对象被隐藏

uint8_t top

1：如果单击对象或其子对象，则转到前景

uint8_t parent_event

1：也将对象的事件发送给父对象。

uint8_t adv_hittest

1：使用高级匹配测试（速度较慢）

uint8_t gesture_parent

1：父母将改为手势

uint8_t focus_parent

1：父母会专注

lv_drag_dir_t drag_dir

可以向哪个方向拖动对象

lv_bidi_dir_t base_dir

与该对象有关的文本的基本方向

void*group_p

uint8_t protect

可以防止自动发生的动作。来自“或”值 `lv_protect_t`

lv_state_t state

lv_realign_t realign

有关最后一次调用 lv_obj_align 的信息。

lv_obj_user_data_t user_data

对象的自定义用户数据。

struct lv_obj_type_t

#include <lv_obj.h>

由使用 `lv_obj_get_type()`。对象及其祖先类型存储在此处

公众成员

const 字符* **type**[**LV_MAX_ANCESTOR_NUM**]

[0]：实际类型，[1]：祖先，[2]＃1 的祖先... [x]：“ lv_obj”

struct lv_hit_test_info_t

公众成员

lv_point_t ***point**

bool **result**

struct lv_get_style_info_t

公众成员

uint8_t **part**

lv_style_list_t ***result**

struct lv_get_state_info_t

公众成员

uint8_t **part**

lv_state_t **result**

第九章 弧（lv_arc）

9.1 总览

弧由背景弧和前景弧组成。两者都可以具有起始角度和终止角度以及厚度。

9.2 小部件和样式

弧的主要部分称为 `LV_ARC_PART_MAIN`。它使用典型的背景样式属性绘制背景，并使用*线型*属性绘制圆弧。圆弧的大小和位置将遵守*填充*样式的属性。

`LV_ARC_PART_INDIC` 是虚拟小部件，它使用*线型*属性绘制了另一个弧。它的填充值是相对于背景弧线解释的。指示器圆弧的半径将根据最大填充值进行修改。

`LV_ARC_PART_KNOB` 是虚拟小部件，它绘制在弧形指示器的末端。它使用所有背景属性和填充值。使用零填充时，旋钮的大小与指示器的宽度相同。较大的填充使其较大，较小的填充使其较小。

9.3 用法

9.3.1 角度

要设置背景角度，请使用或功能。零度位于对象的右中间（3 点钟），并且度数沿顺时针方向增加。角度应在[0; 360]范围内。

`lv_arc_set_bg_angles(arc, start_angle, end_angle)`

`lv_arc_set_bg_start/end_angle(arc, start_angle)`

同样，功能或设置指示器圆弧的角度。

`lv_arc_set_angles(arc, start_angle, end_angle)`

`lv_arc_set_start/end_angle(arc, start_angle)`

9.3.2 回转

可以向偏移 0 度的位置。`lv_arc_set_rotation(arc, deg)`

9.3.3 范围和值

除了手动设置角度外，弧还可以具有范围和值。要设置范围，请使用。使用范围和值，指示器的角度将在背景角度之间映射。

`lv_arc_set_range(arc, min, max)`

`lv_arc_set_value(arc, value)`

9.3.4 类型

弧可以具有不同的“类型”。仅使用由设置的值 `lv_arc_set_value` 。存在以下类型：

- `LV_ARC_TYPE_NORMAL` 指示器弧顺时针绘制（最小电流）
- `LV_ARC_TYPE_REVERSE` 指示器弧沿逆时针方向绘制（最大电流）
- `LV_ARC_TYPE_SYMMETRIC` 从中间点到当前值的指示弧线。

9.3.5 事件

除了通用事件外，弧还发送以下特殊事件：

- 当按下/拖动弧以设置新值时发送 **LV_EVENT_VALUE_CHANGED**。

了解有关事件的更多信息。

9.4 按键

该对象对象类型不处理任何按*鍵*。

9.5 实例

C

简单弧

码

弧形装载机

码

API

typedef

typedef uint8_t lv_arc_type_t

typedef uint8_t lv_arc_part_t

枚举

Enum [anonymous]

值：

enumerator LV_ARC_TYPE_NORMAL

enumerator LV_ARC_TYPE_SYMMETRIC

enumerator LV_ARC_TYPE_REVERSE

enum [anonymous]

值：

enumerator LV_ARC_PART_BG= LV_OBJ_PART_MAIN

enumerator LV_ARC_PART_INDIC

enumerator LV_ARC_PART_KNOB

enumerator _LV_ARC_PART_VIRTUAL_LAST

enumerator _LV_ARC_PART_REAL_LAST= _LV_OBJ_PART_REAL_LAST

功能

lv_obj_t * lv_arc_create（lv_obj_t * *par*，const lv_obj_t * *copy* ）

创建弧对象

返回

指向创建的弧的指针

参数

- `par`：指向对象的指针，它将是新弧的父对象
- `copy`：指向弧对象的指针，如果不为NULL，则将从其复制新对象

void lv_arc_set_start_angle（lv_obj_t * *arc*，uint16_t *start* ）

设置圆弧的起始角度。0 度：右，90 底等。

参数

- `arc`：指向弧对象的指针
- `start`：起始角度

void lv_arc_set_end_angle（lv_obj_t * *arc*，uint16_t *end* ）

设置圆弧的起始角度。0 度：右，90 底等。

参数

- `arc`：指向弧对象的指针
- `end`：结束角度

void lv_arc_set_angles（lv_obj_t * *arc*，uint16_t *start*，uint16_t *end* ）

设置开始和结束角度

参数

- `arc`：指向弧对象的指针
- `start`：起始角度
- `end`：结束角度

void lv_arc_set_bg_start_angle（lv_obj_t * arc，uint16_t start ）

设置弧形背景的起始角度。0 度：右，90 底等。

参数

- `arc`：指向弧对象的指针
- `start`：起始角度

void lv_arc_set_bg_end_angle（lv_obj_t * arc，uint16_t end ）

设置弧形背景的起始角度。0 度：右，90 底等。

参数

- `arc`：指向弧对象的指针
- `end`：结束角度

void lv_arc_set_bg_angles（lv_obj_t * arc，uint16_t start，uint16_t end ）

设置弧形背景的开始和结束角度

参数

- `arc`：指向弧对象的指针
- `start`：起始角度
- `end`：结束角度

void lv_arc_set_rotation（lv_obj_t * arc，uint16_t rotation_angle ）

设置整个圆弧的旋转

参数

- `arc`：指向弧对象的指针
- `rotation_angle`：旋转角度

void lv_arc_set_type（lv_obj_t * arc，lv_arc_type_t type ）

设置圆弧的类型。

参数

- `arc`：指向弧对象的指针
- `type`：圆弧型

void lv_arc_set_value（lv_obj_t * arc，int16_t value）

在圆弧上设置一个新值

参数

- `arc`：指向弧对象的指针
- `value`：新价值

void lv_arc_set_range（lv_obj_t * arc，int16_t min，int16_t max ）

设置圆弧的最小值和最大值

参数

- `arc`：指向弧对象的指针
- `min`：最小值
- `max`：最大值

void lv_arc_set_chg_rate（lv_obj_t * arc，uint16_t threshold）

设置圆弧旋钮增量位置的阈值。

参数

- `arc`：指向弧对象的指针
- `threshold`：增量阈值

void lv_arc_set_adjustable（lv_obj_t * arc，bool adjustable）

设置圆弧是否可调。

参数

- `arc`：指向弧对象的指针
- `adjustable`：圆弧是否具有可以拖动的旋钮

uint16_t lv_arc_get_angle_start（lv_obj_t * arc ）

获取圆弧的起始角度。

返回

起始角度[0..360]

参数

- `arc`：指向弧对象的指针

uint16_t lv_arc_get_angle_end（lv_obj_t * arc ）

获取圆弧的末端角度。

返回

端角[0..360]

参数

- `arc`：指向弧对象的指针

uint16_t lv_arc_get_bg_angle_start（lv_obj_t * arc ）

获取弧形背景的起始角度。

返回

起始角度[0..360]

参数

- `arc`：指向弧对象的指针

uint16_t lv_arc_get_bg_angle_end（lv_obj_t * arc ）

获取弧形背景的终止角度。

返回

端角[0..360]

参数

- `arc`：指向弧对象的指针

lv_arc_type_t lv_arc_get_type（const lv_obj_t *arc）

获取圆弧是否为类型。

返回

弧形

参数

- `arc`：指向弧对象的指针

int16_t lv_arc_get_value（const lv_obj_t * arc）

获取圆弧的值

返回

弧的值

参数

- `arc`：指向弧对象的指针

`int16_t lv_arc_get_min_value(const lv_obj_t * arc)`

获得圆弧的最小值

返回

圆弧的最小值

参数

- `arc`：指向弧对象的指针

`int16_t lv_arc_get_max_value(const lv_obj_t * arc)`

获取圆弧的最大值

返回

圆弧的最大值

参数

- `arc`：指向弧对象的指针

`bool lv_arc_is_dragged(const lv_obj_t * arc)`

给出弧线是否被拖动

返回

真：拖动进行中 false：不拖动

参数

- `arc`：指向弧对象的指针

`bool lv_arc_get_adjustable(lv_obj_t * arc )`

获取圆弧是否可调。

返回

圆弧是否具有可以拖动的旋钮

参数

- `arc`：指向弧对象的指针

struct lv_arc_ext_t

公众成员

```
uint16_t rotation_angle
uint16_t arc_angle_start
uint16_t arc_angle_end
uint16_t bg_angle_start
uint16_t bg_angle_end
lv_style_list_t style_arc
lv_style_list_t style_knob
int16_t cur_value
int16_t min_value
int16_t max_value
uint16_t dragging
uint16_t type
uint16_t adjustable
uint16_t chg_rate
uint32_t last_tick
int16_t last_angle
```

第十章 进度条（lv_bar）

10.1 总览

条对象上有一个背景和一个指示器。指示器的宽度根据条的当前值设置。
如果对象的宽度小于其高度，则可以创建垂直条。
不仅可以结束，还可以设置条的起始值，从而改变指示器的起始位置。

10.2 小部件和样式

进度条的主要部分被称为 `LV_BAR_PART_BG`，它使用典型的背景样式属性。

`LV_BAR_PART_INDIC` 是一个虚拟部件，还使用了所有典型的背景属性。默认情况下，指示器的最大尺寸与背景的尺寸相同，但是在其中设置正的填充值 `LV_BAR_PART_BG` 将使指示器变小。（负值会使它变大）如果在指标上使用了*值*样式属性，则将根据指标的当前大小来计算对齐方式。例如，中心对齐的值始终显示在指示器的中间，而不管其当前大小如何。

10.3 用法

10.3.1 值和范围

可以通过设置新值。该值在一个范围内（最小值和最大值）进行解释，可以使用进行修改。默认范围是1..100。

`lv_bar_set_value(bar, new_value, LV_ANIM_ON/OFF)`

`lv_bar_set_range(bar, min, max)`

`lv_bar_set_value` 可以根据上一个参数（`LV_ANIM_ON/OFF`）设置是否带有动画的新值。动画的时间可以通过调整。时间以毫秒为单位。`lv_bar_set_anim_time(bar, 100)`

也可以使用设置条的起始值 `lv_bar_set_start_value(bar, new_value, LV_ANIM_ON/OFF)`

10.3.2 模式

如果启用，则可以将条形对称地绘制为零（从零开始，从左到右绘制）。
`lv_bar_set_type(bar, LV_BAR_TYPE_SYMMETRICAL)`

10.4 事件

仅通用事件是按对象类型发送的。

10.5 按键

对象类型不处理任何按*键*。

例

C

简单吧

API

typedef

typedef uint8_t lv_bar_type_t

typedef uint8_t lv_bar_part_t

枚举

Enum [anonymous]

值:

enumerator LV_BAR_TYPE_NORMAL

enumerator LV_BAR_TYPE_SYMMETRICAL

enumerator LV_BAR_TYPE_CUSTOM

enum [anonymous]

进度条小部件

值：

enumerator **LV_BAR_PART_BG**

enumerator **LV_BAR_PART_INDIC**

进度条背景风格。

enumerator **_LV_BAR_PART_VIRTUAL_LAST**

条形填充区域样式。

功能

lv_obj_t * **lv_bar_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个条对象

返回

指向创建的条的指针

参数

- `par`：指向对象的指针，它将是新栏的父对象
- `copy`：指向条对象的指针，如果不为 NULL，则将从其复制新对象

void **lv_bar_set_value**（lv_obj_t * *bar*，int16_t *value*，lv_anim_enable_t *anim* ）

在栏上设置新值

参数

- `bar`：指向条对象的指针
- `value`：新价值
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void **lv_bar_set_start_value**（lv_obj_t * *bar*，int16_t *start_value*，lv_anim_enable_t *anim* ）

在栏上设置新的起始值

参数

- `bar`：指向条对象的指针
- `value`：新的起始值

- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_bar_set_range（lv_obj_t * *bar*，int16_t *min*，int16_t *max*）

设置条的最小值和最大值

参数

- `bar`：指向 bar 对象的指针
- `min`：最小值
- `max`：最大值

void lv_bar_set_type（lv_obj_t * *bar*，lv_bar_type_t *type*）

设置栏的类型。

参数

- `bar`：指向条对象的指针
- `type`：条形

void lv_bar_set_anim_time（lv_obj_t * *bar*，uint16_t *anim_time*）

设置栏的动画时间

参数

- `bar`：指向条对象的指针
- `anim_time`：动画时间（以毫秒为单位）。

int16_t lv_bar_get_value（*const* lv_obj_t * *bar*）

获取进度条的价值

返回

进度条的价值

参数

- `bar`：指向条对象的指针

int16_t lv_bar_get_start_value（*const* lv_obj_t * *bar*）

获取条形图的起始值

返回

条的起始值

参数

- `bar`：指向条对象的指针

`int16_t lv_bar_get_min_value（const lv_obj_t * bar ）`

获取条的最小值

返回

进度条的最小值

参数

- `bar`：指向条对象的指针

`int16_t lv_bar_get_max_value（const lv_obj_t * bar ）`

获取条形的最大值

返回

进度条的最大值

参数

- `bar`：指向条对象的指针

`lv_bar_type_t lv_bar_get_type（lv_obj_t * bar ）`

获取栏的类型。

返回

条型

参数

- `bar`：指向条对象的指针

`uint16_t lv_bar_get_anim_time（const lv_obj_t * bar ）`

获取栏的动画时间

返回

动画时间（以毫秒为单位）。

参数

- `bar`：指向条对象的指针

`struct lv_bar_anim_t`

公众成员

```
lv_obj_t *bar
lv_anim_value_t anim_start
lv_anim_value_t anim_end
lv_anim_value_t anim_state
```

struct lv_bar_ext_t

```
#include <lv_bar.h>
```

条数据

公众成员

```
int16_t cur_value
int16_t min_value
int16_t max_value
int16_t start_value
lv_area_t indic_area
lv_anim_value_t anim_time
lv_bar_anim_t cur_value_anim
lv_bar_anim_t start_value_anim
uint8_t type
lv_style_list_t style_indic
```

第十一章 按钮（lv_btn）

11.1 总览

按钮是简单的矩形对象。它们源自容器，因此也可以提供布局和配合。此外，可以启用它以在单击时自动进入检查状态。

11.2 小部件和样式

这些按钮只有一个主要样式 `LV_BTN_PART_MAIN`，可以使用以下组中的所有属性：

- 背景
- 边境
- 大纲
- 阴影
- 值
- 模式
- 过渡

启用*布局*或*适合*时，它还将使用 *padding* 属性。

11.3 用法 Cai Xuefeng

11.3.1 状态

为了简化按钮的使用，可以使用来获取按钮的状态 `lv_btn_get_state(btn)`。它返回以下值之一：

- **LV_BTN_STATE_RELEASED**
- **LV_BTN_STATE_PRESSED**
- **LV_BTN_STATE_CHECKED_RELEASED**
- **LV_BTN_STATE_CHECKED_PRESSED**
- **LV_BTN_STATE_DISABLED**
- **LV_BTN_STATE_CHECKED_DISABLED**

使用按钮可以手动更改状态。`lv_btn_get_state(btn, LV_BTN_STATE_...)`

如果需要状态的更精确描述（例如，重点关注），`lv_obj_get_state(btn)` 则可以使用一般性描述。

11.3.2 可检查

您可以配置按钮的*切换按钮*用。在这种情况下，单击时，按钮将自动进入状态，或再次单击时将返回状态。

`lv_btn_set_checkable(btn, true)LV_STATE_CHECKED`

11.3.3 布局和适合

与 Containers 相似，按钮也具有 layout 和 fit 属性。

- `lv_btn_set_layout(btn, LV_LAYOUT_...)` 设置布局。默认值为 `LV_LAYOUT_CENTER`。因此，如果您添加标签，则标签将自动与中间对齐，并且不能与一起移动 `lv_obj_set_pos()`。您可以使用禁用布局。`lv_btn_set_layout(btn, LV_LAYOUT_OFF)`
- `lv_btn_set_fit/fit2/fit4(btn, LV_FIT_..)` 可以根据子代，父代和适合类型自动设置按钮的宽度和/或高度。

11.4 事件

除了通用事件，以下特殊事件还通过按钮发送：

- **LV_EVENT_VALUE_CHANGED**-切换按钮时发送。

11.5 按键

以下按键由按钮处理：

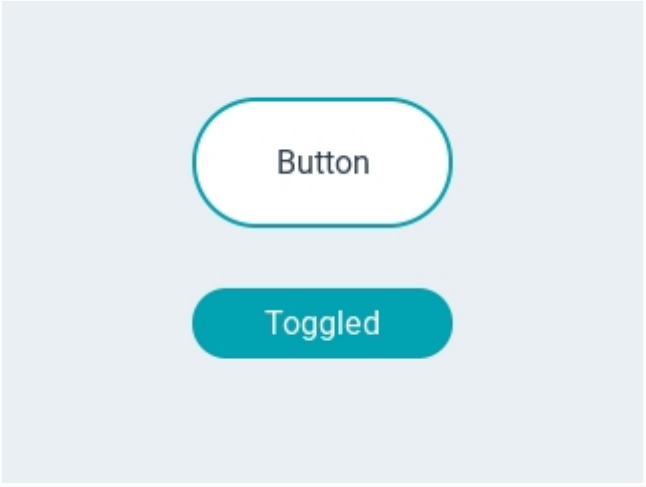
- **LV_KEY_RIGHT / UP**-如果启用了切换，则进入切换状态。
- **LV_KEY_LEFT / DOWN**-如果启用了切换，则进入非切换状态。

请注意，的状态 `LV_KEY_ENTER` 已转换为 `LV_EVENT_PRESSED/PRESSING/RELEASED` etc。

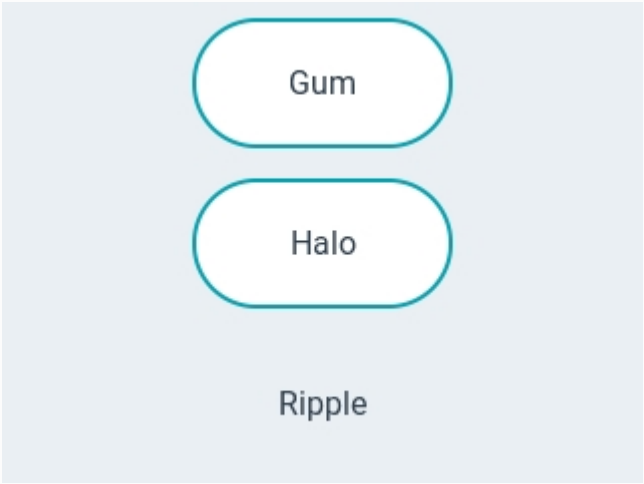
例

C

简单按钮



码



API

typedef

typedef uint8_t **lv_btn_state_t**

typedef uint8_t **lv_btn_part_t**

枚举

Enum [anonymous]

按钮的可能状态。它不仅可以用于按钮，还可以用于其他类似按钮的对象

值:

enumerator **LV_BTN_STATE_RELEASED**

enumerator **LV_BTN_STATE_PRESSED**

enumerator **LV_BTN_STATE_DISABLED**

enumerator **LV_BTN_STATE_CHECKED_RELEASED**

enumerator **LV_BTN_STATE_CHECKED_PRESSED**

enumerator **LV_BTN_STATE_CHECKED_DISABLED**

enumerator **_LV_BTN_STATE_LAST**

enum [anonymous]

款式

值:

enumerator **LV_BTN_PART_MAIN**= LV_OBJ_PART_MAIN

enumerator **_LV_BTN_PART_VIRTUAL_LAST**

enumerator **_LV_BTN_PART_REAL_LAST=** _LV_OBJ_PART_REAL_LAST

功能

---

lv_obj_t * **lv_btn_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个按钮对象

返回

指向创建的按钮的指针

参数

- `par`：指向对象的指针，它将是新按钮的父对象
- `copy`：指向按钮对象的指针，如果不为 NULL，则将从其复制新对象

---

voidlv_btn_set_checkable（lv_obj_t * *btn*，bool *tgl* ）

启用切换状态。释放时，按钮将从/切换到切换状态。

参数

- `btn`：指向按钮对象的指针
- `tgl`：true：启用切换状态，false：禁用

---

voidlv_btn_set_state（lv_obj_t * *btn*，lv_btn_state_t *state*）

设置按钮的状态

参数

- `btn`：指向按钮对象的指针
- `state`：按钮的新状态（来自 lv_btn_state_t 枚举）

---

voidlv_btn_toggle（lv_obj_t * *btn* ）

切换按钮的状态（ON-> OFF，OFF-> ON）

参数

- `btn`：指向按钮对象的指针

---

voidlv_btn_set_layout（lv_obj_t * *btn*，lv_layout_t *layout*）

在按钮上设置布局

参数

- `btn`：指向按钮对象的指针
- `layout`：来自“ lv_cont_layout_t”的布局

void **lv_btn_set_fit4**（lv_obj_t * *BTN*，lv_fit_t *left*，lv_fit_t *right*，lv_fit_t *top*，lv_fit_t *bottom*）

分别在所有四个方向上设置适合策略。它告诉您如何自动更改按钮的大小。

参数

- `btn`：指向按钮对象的指针
- `left`：从左适合政策 `lv_fit_t`
- `right`：合适的政策来自 `lv_fit_t`
- `top`：最适合的政策，来自 `lv_fit_t`
- `bottom`：自下而上的政策 `lv_fit_t`

void **lv_btn_set_fit2**（lv_obj_t * *btn*，lv_fit_t *hor*，lv_fit_t *ver* ）

分别水平和垂直设置适合策略。它告诉您如何自动更改按钮的大小。

参数

- `btn`：指向按钮对象的指针
- `hor`：来自的水平拟合政策 `lv_fit_t`
- `ver`：垂直适合政策，来自 `lv_fit_t`

void **lv_btn_set_fit**（lv_obj_t * *btn*，lv_fit_t *fit* ）

一次在所有 4 个方向上设置拟合策略。它告诉您如何自动更改按钮的大小。

参数

- `btn`：指向按钮对象的指针
- `fit`：符合政策 `lv_fit_t`

lv_btn_state_t **lv_btn_get_state**（*const* lv_obj_t * *btn* ）

获取按钮的当前状态

返回

按钮的状态（来自 lv_btn_state_t 枚举）如果按钮处于禁用状态，`LV_BTN_STATE_DISABLED` 则将其与其他按钮状态进行“或”运算。

参数

- `btn`：指向按钮对象的指针

bool lv_btn_get_checkable（const lv_obj_t * btn ）

获取按钮的切换启用属性

返回

true：启用检查，false：停用

参数

- `btn`：指向按钮对象的指针

lv_layout_t lv_btn_get_layout（const lv_obj_t * btn ）

获取按钮的布局

返回

来自“ lv_cont_layout_t”的布局

参数

- `btn`：指向按钮对象的指针

lv_fit_t lv_btn_get_fit_left（const lv_obj_t * btn ）

获取左合身模式

返回

的元素 `lv_fit_t`

参数

- `btn`：指向按钮对象的指针

lv_fit_t lv_btn_get_fit_right（const lv_obj_t * btn ）

获得合适的健身模式

返回

的元素 `lv_fit_t`

参数

- `btn`：指向按钮对象的指针

lv_fit_t lv_btn_get_fit_top（const lv_obj_t * btn ）

获取最适合的模式

返回

的元素 `lv_fit_t`

参数

- `btn`：指向按钮对象的指针

lv_fit_t lv_btn_get_fit_bottom（const lv_obj_t * btn ）

获取最合适的模式

返回

的元素 `lv_fit_t`

参数

- `btn`：指向按钮对象的指针

struct lv_btn_ext_t

#include <lv_btn.h>

按钮的扩展数据

公众成员

lv_cont_ext_t cont

分机号 祖先

uint8_t checkable

1：启用切换

第十二章 按钮矩阵（lv_btnmatrix）

12.1 总览

Button Matrix 对象可以在行和列中显示**多个按钮**。
希望使用按钮矩阵而不是容器和单个按钮对象的主要原因是：

- 对于基于网格的按钮布局，按钮矩阵更易于使用。
- 每个按钮矩阵消耗的内存少得多。

12.2 小部件和样式

Button 矩阵的主要部分称为 `LV_BTNMATRIX_PART_BG` 。它使用典型的背景样式属性绘制背景。

`LV_BTNMATRIX_PART_BTN` 是虚拟部分，它指的是按钮矩阵上的按钮。它还使用所有典型的背景属性。

背景中的顶部/底部/左侧/右侧填充值用于在侧面保留一些空间。在按钮之间应用内部填充。

12.3 用法

12.3.1 按钮的文字

每个按钮上都有一个文本。要指定它们，需要使用称为 map 的描述符字符串数组。可以使用设置地图。地图的声明应类似于。请注意，最后一个元素必须为空字符串！ lv_btnmatrix_set_map(btnm, my_map)

```
const char * map[] = {"btn1", "btn2", "btn3", ""}
```

"\n"在地图中使用以使换行。例如。每行按钮的宽度自动计算。 `{"btn1", "btn2", "\n", "btn3", ""}`

12.3.2 控制按钮

的宽度按钮可以相对于与同一行被设置为其他按钮 中的线例如具有两个按钮：btnA，宽度= 1 和，btnB 宽度 = 2，btnA 将有 33%的宽度和 btnB 将有 66%的宽度。这类似于该属性在 CSS 中的工作方式。

```
lv_btnmatrix_set_btn_width(btnm, btn_id, width)flex-grow
```

除了宽度，每个按钮都可以使用以下参数进行自定义：

- **LV_BTNMATRIX_CTRL_HIDDEN-**将按钮隐藏（隐藏的按钮仍占据布局中的空间，它们不可见或不可单击）
- **LV_BTNMATRIX_CTRL_NO_REPEAT-**长按按钮时禁用重复
- **LV_BTNMATRIX_CTRL_DISABLED-**禁用按钮
- **LV_BTNMATRIX_CTRL_CHECKABLE-**启用按钮切换

- **LV_BTNMATRIX_CTRL_CHECK_STATE-**设置切换状态
- **LV_BTNMATRIX_CTRL_CLICK_TRIG-**如果为 0，按钮将在按下时起作用，如果为 1，则在释放时起作用

设置或清除按钮的控件属性，分别使用和 。可以或编辑更多的值

```
lv_btnmatrix_set_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)
```

```
lv_btnmatrix_clear_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)LV_BTNM_CTRL_...
```

为按钮矩阵的所有按钮设置/清除相同的控件属性，请使用和 。

```
lv_btnmatrix_set_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)
```

```
lv_btnmatrix_clear_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)
```

使用为按钮矩阵设置控制图（类似于文本的图）。的元素应为。元素的数量应等于按钮的数量（不包括换行符）。

```
lv_btnmatrix_set_ctrl_map(btnm, ctrl_map)
```

```
ctrl_mapctrl_map[0] = width | LV_BTNM_CTRL_NO_REPEAT |  LV_BTNM_CTRL_TGL_ENABLE
```

12.3.3 一次检查

可以启用“一次检查”功能，一次只允许检查（切换）一个按钮。`lv_btnmatrix_set_one_check(btnm, true)`

12.3.4 重新着色

在文本上的按钮可以重新着色类似的重新着色功能标签的对象。要启用它，请使用。之后，带有文字的按钮将变为红色。

```
lv_btnmatrix_set_recolor(btnm, true)#FF0000 Red#
```

12.3.5 对齐按钮的文字

要使按钮上的文本对齐，请使用。按钮矩阵中的所有文本项都将符合设置时的对齐方式。

```
lv_btnmatrix_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)
```

12.3.6 记录

Button 矩阵对象的权重非常轻，因为按钮不是在飞行中虚拟绘制的。这样，一个按钮仅使用 8 个额外的字

节，而不是普通 Button 对象的∽100-150 字节大小（加上其容器的大小和每个按钮的标签）。

此设置的缺点是，将各个按钮的样式设置为与其他按钮不同的功能受到限制（除了切换功能之外）。如果您需要该功能，则使用单个按钮很有可能是一种更好的方法。

12.4 事件

除了通用事件外，按钮矩阵还发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED-**在按下/释放按钮时或在长按之后重复时发送。事件数据设置为按下/释放按钮的 ID。

12.5 按键

以下*按键*由按钮处理：

- **LV_KEY_RIGHT / UP / LEFT / RIGHT-**在按钮之间导航以选择一个
- **LV_KEY_ENTER-**按下/释放所选按钮

例

C

Cai Xuefeng

简单按钮矩阵

API

typedef

```
typedef uint16_t lv_btnmatrix_ctrl_t
```

typedef uint8_t lv_btnmatrix_part_t

枚举

Enum [anonymous]

键入以存储按钮控制位（禁用，隐藏等）。前 3 位用于存储宽度

值：

enumerator LV_BTNMATRIX_CTRL_HIDDEN = 0x0008

隐藏按钮

enumerator LV_BTNMATRIX_CTRL_NO_REPEAT = 0x0010

不要重复按此按钮。

enumerator LV_BTNMATRIX_CTRL_DISABLED = 0x0020

禁用此按钮。

enumerator LV_BTNMATRIX_CTRL_CHECKABLE = 0x0040

*可以*切换按钮。

enumerator LV_BTNMATRIX_CTRL_CHECK_STATE = 0x0080

当前已切换按钮（例如，选中）

enumerator LV_BTNMATRIX_CTRL_CLICK_TRIG = 0x0100

1：在 CLICK 上发送 LV_EVENT_SELECTED，0：在 PRESS 上发送 LV_EVENT_SELECTED

Enum [anonymous]

值：

enumerator LV_BTNMATRIX_PART_BG

enumerator LV_BTNMATRIX_PART_BTN

功能

LV_EXPORT_CONST_INT（ LV_BTNMATRIX_BTN_NONE ）

lv_obj_t * lv_btnmatrix_create（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个按钮矩阵对象

返回

指向创建的按钮矩阵的指针

参数

- `par`：指向对象的指针，它将是新按钮矩阵的父对象
- `copy`：指向按钮矩阵对象的指针，如果不为 NULL，则将从其复制新对象

void lv_btnmatrix_set_map（lv_obj_t * *btnm*，const char * *map* [] ）

设置新地图。将根据地图创建/删除按钮。按钮矩阵保留了对地图的引用，因此在矩阵有效期内不得释放字符串数组。

参数

- `btnm`：指向按钮矩阵对象的指针
- `map`：指针的字符串数组。最后一个字符串必须是：“”。使用“ \ n”进行换行。

void lv_btnmatrix_set_ctrl_map（lv_obj_t * *btnm*，const lv_btnmatrix_ctrl_t *ctrl_map* [] ）

设置按钮矩阵的按钮控制图（隐藏，禁用等）。控制图数组将被复制，因此在此函数返回后可以将其释放。

参数

- `btnm`：指向按钮矩阵对象的指针
- `ctrl_map`：指向 `lv_btn_ctrl_t` 控制字节数组的指针。数组的长度和元素的位置必须与单个按钮的数量和顺序匹配（即，不包括换行符）。地图的元素应如下所示：


```
ctrl_map[0] = width | LV_BTNMATRIX_CTRL_NO_REPEAT | LV_BTNMATRIX_CTRL_TGL_ENABLE
```

void lv_btnmatrix_set_focused_btn（lv_obj_t * *btnm*，uint16_t *id* ）

设置焦点按钮，即在视觉上突出显示它。

参数

- `btnm`：指向按钮矩阵对象的指针
- `id`：要聚焦的按钮的索引（`LV_BTNMATRIX_BTN_NONE` 以除去焦点）

void lv_btnmatrix_set_recolor（const lv_obj_t * *btnm*，bool *en* ）

启用按钮文本的重新着色

参数

- `btnm`：指向按钮矩阵对象的指针
- `en`：true：启用重新着色；false：禁用

void lv_btnmatrix_set_btn_ctrl（const lv_obj_t * *btnm*，uint16_t *btn_id*，lv_btnmatrix_ctrl_t *ctrl* ）

设置按钮矩阵的按钮属性

参数

- `btnm`：指向按钮矩阵对象的指针
- `btn_id`：基于 0 的按钮的索引进行修改。（不计算新行）

void lv_btnmatrix_clear_btn_ctrl（*const* lv_obj_t * *btnm*，uint16_t *btn_id*，lv_btnmatrix_ctrl_t *ctrl*）

清除按钮矩阵中按钮的属性

参数

- `btnm`：指向按钮矩阵对象的指针
- `btn_id`：基于 0 的按钮的索引进行修改。（不计算新行）

void lv_btnmatrix_set_btn_ctrl_all（lv_obj_t * *btnm*，lv_btnmatrix_ctrl_t *ctrl*）

设置按钮矩阵中所有按钮的属性

参数

- `btnm`：指向按钮矩阵对象的指针
- `ctrl`：要从中设置的属性 `lv_btnmatrix_ctrl_t`。值可以进行或运算。

void lv_btnmatrix_clear_btn_ctrl_all（lv_obj_t * *btnm*，lv_btnmatrix_ctrl_t *ctrl*）

清除按钮矩阵中所有按钮的属性

参数

- `btnm`：指向按钮矩阵对象的指针
- `ctrl`：要从中设置的属性 `lv_btnmatrix_ctrl_t`。值可以进行或运算。
- `en`：true：设置属性；false：清除属性

void lv_btnmatrix_set_btn_width（lv_obj_t * *btnm*，uint16_t *btn_id*，uint8_t *width*）

设置单个按钮的相对宽度。该方法将导致矩阵再生并且是相对昂贵的操作。建议使用来指定初始宽度，`lv_btnmatrix_set_ctrl_map` 并且此方法仅用于动态更改。

参数

- `btnm`：指向按钮矩阵对象的指针
- `btn_id`：基于 0 的按钮的索引进行修改。

- `width`：相对宽度（与同一行中的按钮相比）。[1..7]

void lv_btnmatrix_set_one_check（lv_obj_t * btnm，bool one_chk）

使按钮矩阵像选择器小部件一样（一次只能切换一个按钮）。`Checkable` 必须在要使用 `lv_btnmatrix_set_ctrl` 或选择的按钮上启用 `lv_btnmatrix_set_btn_ctrl_all`。

参数

- `btnm`：按钮矩阵对象
- `one_chk`：是否启用“一次检查”模式

void lv_btnmatrix_set_align（lv_obj_t * btnm，lv_label_align_t align）

设置地图文字的对齐方式（向左，向右或居中）

参数

- `btnm`：指向 btnmatrix 对象的指针
- `align`：LV_LABEL_ALIGN_LEFT，LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER

const 字符** lv_btnmatrix_get_map_array（const lv_obj_t * btnm）

获取按钮矩阵的当前图

返回

当前地图

参数

- `btnm`：指向按钮矩阵对象的指针

bool lv_btnmatrix_get_recolor（const lv_obj_t * btnm）

检查按钮的文本是否可以使用重新着色

返回

true：启用文本重新着色；false：禁用

参数

- `btnm`：指向按钮矩阵对象的指针

uint16_t lv_btnmatrix_get_active_btn（const lv_obj_t * btnm）

获取用户最后按下的按钮的索引（按下，释放等）。在 `event_cb` 获取按钮的文本，检查是否隐藏等方面很有用。

返回

最后释放按钮的索引（LV_BTNMATRIX_BTN_NONE：如果未设置）

参数

- `btnm`：指向按钮矩阵对象的指针

---

const 字符* lv_btnmatrix_get_active_btn_text（*const* lv_obj_t * *btnm* ）

获取用户最后一次“激活”按钮的文本（按下，释放等） `event_cb`

返回

最后释放的按钮的文本（NULL：如果未设置）

参数

- `btnm`：指向按钮矩阵对象的指针

---

uint16_t lv_btnmatrix_get_focused_btn（*const* lv_obj_t * *btnm* ）

获取焦点按钮的索引。

返回

焦点按钮的索引（LV_BTNMATRIX_BTN_NONE：如果未设置）

参数

- `btnm`：指向按钮矩阵对象的指针

---

const 字符* lv_btnmatrix_get_btn_text（*const* lv_obj_t * *btnm*，uint16_t *btn_id* ）

获取按钮的文字

返回

btn_index`按钮的文本

参数

- `btnm`：指向按钮矩阵对象的指针
- `btn_id`：索引一个不计算换行符的按钮。（lv_btnmatrix_get_pressed / released 的返回值）

---

bool lv_btnmatrix_get_btn_ctrl（lv_obj_t * *btnm*，uint16_t *btn_id*，lv_btnmatrix_ctrl_t *ctrl* ）

获取按钮矩阵的按钮是启用还是禁用控件值

返回

true：禁用长按重复；否：长按重复启用

参数

- `btnm`：指向按钮矩阵对象的指针
- `btn_id`：索引一个不计算换行符的按钮。（例如 lv_btnmatrix_get_pressed / released 的返回值）
- `ctrl`：要检查的控制值（可以使用 ORed 值）

---

bool lv_btnmatrix_get_one_check（*const* lv_obj_t * *btnm* ）

查找是否启用了“一次切换”模式。

返回

是否启用“一次切换”模式

参数

- `btnm`：按钮矩阵对象

---

lv_label_align_t lv_btnmatrix_get_align（*const* lv_obj_t * *btnm* ）

获取 align 属性

返回

LV_LABEL_ALIGN_LEFT，LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER

参数

- `btnm`：指向 btnmatrix 对象的指针

***struct* lv_btnmatrix_ext_t**

公众成员

const char**map_p

lv_area_t *button_areas

lv_btnmatrix_ctrl_t *ctrl_bits

lv_style_list_t style_btn

uint16_t btn_cnt

uint16_t btn_id_pr

uint16_t btn_id_focused

uint16_t btn_id_act

uint8_t recolor

```
    uint8_t one_check
    uint8_t align
```

Cai Xuefeng

第十三章 日历（lv_calendar）

13.1 总览

Calendar 对象是经典日历，可以：

- 突出显示当天
- 突出显示任何用户定义的日期
- 显示日期名称
- 单击按钮进入下一个/上一个月
- 突出显示点击的日子

13.2 小部件和样式

日历的主要部分称为 `LV_CALENDAR_PART_BG`。它使用典型的背景样式属性绘制背景。

除以下虚拟部分外：

- `LV_CALENDAR_PART_HEADER` 显示当前年和月名称的上部区域。它还具有用于移动下一个/上个月的按钮。它使用典型的背景属性和填充来与背景（顶部，左侧，右侧）和日期名称（底部）保持一定距离。
- `LV_CALENDAR_PART_DAY_NAMES` 在标题下方显示日期名称。它使用文本样式属性填充来与背景（左，右），标题（上）和日期（下）保持一定距离。
- `LV_CALENDAR_PART_DATES` 显示从 1..28 / 29/30/31 开始的日期数字（取决于当前月份）。根据本部分中定义的状态来绘制状态的不同“状态”：
  - 正常日期：以 `LV_STATE_DEFAULT` 样式绘制
  - 印刷日期：`LV_STATE_PRESSED` 风格鲜明
  - 今天：以 `LV_STATE_FOCUSED` 风格绘制
  - 突出显示的日期：以 `LV_STATE_CHECKED` 样式绘制

13.3 用法

13.3.1 总览

设置和获取的日历日期，该 `lv_calendar_date_t` 类型用于这是一个 struct `year`， `month` 和 `day` 领域。

13.3.2 当前日期

要设置当前日期（今天），请使用功能。`lv_calendar_set_today_date(calendar, &today_date)`

13.3.3 显示日期

要设置显示的日期，请使用；`lv_calendar_set_shown_date(calendar, &shown_date)`

13.3.4 重点日子

高亮显示的日期列表应存储在由 lv_calendar_date_t 加载的数组中。仅保存数组指针，因此数组应为静态或全局变量。`lv_calendar_set_highlighted_dates(calendar, &highlighted_dates)`

13.3.5 日子名称

日子的名字可以用看起来像的地方调整

```
lv_calendar_set_day_names(calendar, day_names)day_namesconst char * day_names[7] = {"Su", "Mo", ...};
```


13.3.6 月份名称

类似于 `day_names`，月份名称可以用设置。

```
lv_calendar_set_month_names(calendar, month_names_array)
```

13.4 事件

除常规事件外，日历**还会**发送以下特殊事件： 当前月份更改时，**还会**发送 **LV_EVENT_VALUE_CHANGED**。

在*与输入设备相关的*事件中，`lv_calendar_get_pressed_date(calendar)`告诉当前正在按下的 `NULL` 日期，或者如果未按下任何日期则返回。

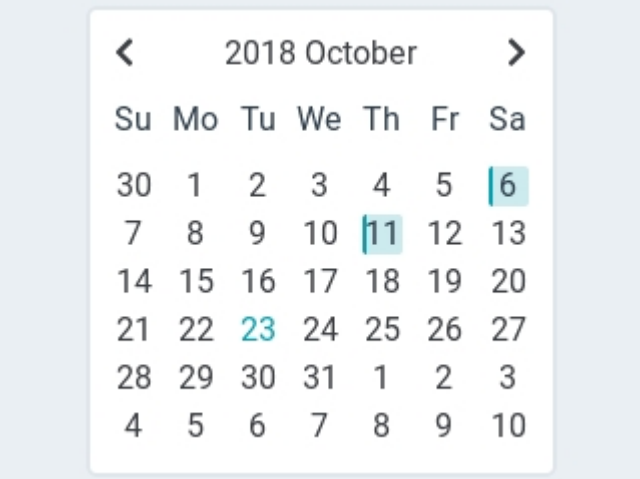
13.5 按键

对象类型不处理任何*键*。

例

C

选择日期的日历



API

typedef

typedef uint8_t `lv_calendar_part_t`

枚举

Enum [anonymous]

日历小部件

值:

enumerator `LV_CALENDAR_PART_BG`

背景和“正常”日期数字样式

enumerator `LV_CALENDAR_PART_HEADER`

enumerator `LV_CALENDAR_PART_DAY_NAMES`

日历标题样式

enumerator `LV_CALENDAR_PART_DATE`

日名样式

功能

lv_obj_t * lv_calendar_create（lv_obj_t * par，const lv_obj_t * copy）

创建日历对象

返回

指向创建的日历的指针

参数

- `par`：指向对象的指针，它将是新日历的父对象
- `copy`：指向日历对象的指针，如果不为 NULL，则将从其复制新对象

void lv_calendar_set_today_date（lv_obj_t * calendar，lv_calendar_date_t * today）

设定今天的日期

参数

- `calendar`：指向日历对象的指针
- `today`：指向 `lv_calendar_date_t` 包含今天的日期的变量的指针。该值将被保存，也可以是局部变量。

void lv_calendar_set_showed_date（lv_obj_t * calendar，lv_calendar_date_t * showed）

设置当前显示

参数

- `calendar`：指向日历对象的指针
- `showed`：指向 `lv_calendar_date_t` 包含要显示日期的变量的指针。该值将被保存，也可以是局部变量。

void lv_calendar_set_highlighted_dates（lv_obj_t * calendar，
lv_calendar_date_t highlighted [ ]，uint16_t date_num ）

设置突出显示的日期

参数

- `calendar`：指向日历对象的指针
- `highlighted`：指向 `lv_calendar_date_t` 包含日期的数组的指针。只有一个指针会被保存！不能是本地阵列。
- `date_num`：数组中的日期数

void lv_calendar_set_day_names（lv_obj_t * calendar，const char ** day_names ）

设置日子的名称

参数

- `calendar`：指向日历对象的指针
- `day_names`：指向具有名称的数组的指针。例如，将仅保存指针，因此该变量不能是局部变量，以后将被销毁。`const char * days[7] = {"Sun", "Mon", ...}`

void lv_calendar_set_month_names（lv_obj_t * calendar，const char ** month_names）

设置月份名称

参数

- `calendar`：指向日历对象的指针
- `month_names`：指向具有名称的数组的指针。例如，将仅保存指针，因此该变量不能是局部变量，以后将被销毁。`const char * days[12] = {"Jan", "Feb", ...}`

lv_calendar_date_t * lv_calendar_get_today_date（const lv_obj_t * calendar）

获取今天的日期

返回

返回指向 `lv_calendar_date_t` 包含今天的日期的变量的指针。

参数

- `calendar`：指向日历对象的指针

lv_calendar_date_t * lv_calendar_get_showed_date（const lv_obj_t * calendar）

获取当前显示

返回

指向 `lv_calendar_date_t` 包含日期的变量的指针正在显示。

参数

- `calendar`：指向日历对象的指针

lv_calendar_date_t * lv_calendar_get_pressed_date（const lv_obj_t * calendar）

获取按日期。

返回

指向 `lv_calendar_date_t` 包含按下日期的变量的指针。`NULL` 如果未按日期（例如标题）

参数

- `calendar`：指向日历对象的指针

---

lv_calendar_date_t * **lv_calendar_get_highlighted_dates**（*const* lv_obj_t * *calendar*）

获取突出显示的日期

返回

指向 `lv_calendar_date_t` 包含日期的数组的指针。

参数

- `calendar`：指向日历对象的指针

---

uint16_t **lv_calendar_get_highlighted_dates_num**（*const* lv_obj_t * *calendar*）

获取突出显示的日期数

返回

突出显示的天数

参数

- `calendar`：指向日历对象的指针

---

const char** **lv_calendar_get_day_names**（*const* lv_obj_t * *calendar*）

获取日子的名字

返回

指向日期名称数组的指针

参数

- `calendar`：指向日历对象的指针

---

const char** **lv_calendar_get_month_names**（*const* lv_obj_t * *calendar*）

获取月份名称

返回

指向月份名称数组的指针

参数

- `calendar`：指向日历对象的指针

---

struct **lv_calendar_date_t**

#include <lv_calendar.h>

表示日历对象上的日期（与平台无关）。

公众成员

```
uint16_t year
int8_t month
int8_t day
```

struct lv_calendar_ext_t

公众成员

```
lv_calendar_date_t today
lv_calendar_date_t showed_date
lv_calendar_date_t *highlighted_dates
int8_t btn_pressing
uint16_t highlighted_dates_num
lv_calendar_date_t pressed_date
const char** day_names
const char** month_names
lv_style_list_t style_header
lv_style_list_t style_day_names
lv_style_list_t style_date_nums
```

第十四章 画布（lv_canvas）

14.1 总览

画布继承自图像，用户可以在其中绘制任何内容。可以使用 lvgl 的绘图引擎在此处绘制矩形，文本，图像，圆弧。除了一些“效果”，还可以应用，例如旋转，缩放和模糊。

14.2 小部件和样式

画布的主要部分称为 `LV_CANVAS_PART_MAIN`，只有 *image_recolor* 属性用于为 `LV_IMG_CF_ALPHA_1/2/4/8BIT` 图像赋予颜色。

14.3 用法

14.3.1 缓冲

画布需要一个缓冲区来存储绘制的图像。要将缓冲区分配给画布，请使用。静态缓冲区（不仅是局部变量）在哪里， 用于保存画布的图像。例如，。这有助于确定具有不同颜色格式的缓冲区的大小。


```
lv_canvas_set_buffer(canvas, buffer, width, height, LV_IMG_CF_...)bufferstatic lv_color_t buffer[LV_CANVAS_BUF_SIZE_TRUE_COLOR(width, height)]LV_CANVAS_BUF_SIZE_...
```

画布支持所有内置颜色格式，例如 `LV_IMG_CF_TRUE_COLOR` 或 `LV_IMG_CF_INDEXED_2BIT`。请参阅“ 颜色格式”部分中的完整列表。

14.3.2 调色板

对于 `LV_IMG_CF_INDEXED_...` 颜色格式，必须使用初始化调色板 。它将 *index = 3 的*像素设置为红色。 `lv_canvas_set_palette(canvas, 3, LV_COLOR_RED)`

14.3.3 画画

要在画布上设置像素，请使用。使用或时，需要将颜色的索引或 Alpha 值作为颜色传递。例如`lv_canvas_set_px(canvas, x, y, LV_COLOR_RED)LV_IMG_CF_INDEXED_...LV_IMG_CF_ALPHA_...lv_color_t c; c.full = 3;`

`lv_canvas_fill_bg(canvas, LV_COLOR_BLUE, LV_OPA_50)`将整个画布填充为具有 50%不透明度的蓝色。请注意，如果当前的颜色格式不支持颜色（例如`LV_IMG_CF_ALPHA_2BIT`），则将忽略颜色。同样，如果不支持不透明度（例如`LV_IMG_CF_TRUE_COLOR`），它将被忽略。

可以使用将像素数组复制到画布。缓冲区和画布的颜色格式需要匹配。`lv_canvas_copy_buf(canvas, buffer_to_copy, x, y, width, height)`

画一些东西到画布上使用

- `lv_canvas_draw_rect(canvas, x, y, width, heigth, &draw_dsc)`
- `lv_canvas_draw_text(canvas, x, y, max_width, &draw_dsc, txt, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`
- `lv_canvas_draw_img(canvas, x, y, &img_src, &draw_dsc)`
- `lv_canvas_draw_line(canvas, point_array, point_cnt, &draw_dsc)`
- `lv_canvas_draw_polygon(canvas, points_array, point_cnt, &draw_dsc)`
- `lv_canvas_draw_arc(canvas, x, y, radius, start_angle, end_angle, &draw_dsc)`

`draw_dsc`是一个`lv_draw_rect/label/img/line_dsc_t`变量，应首先使用`lv_draw_rect/label/img/line_dsc_init()`函数进行初始化，然后使用所需的颜色和其他值对其进行修改。

绘制功能可以绘制为任何颜色格式。例如，可以在`LV_IMG_VF_ALPHA_8BIT`画布上绘制文本，然后将结果图像用作 lv_objmask 中的蒙版。

14.3.4 转变

`lv_canvas_transform()`可用于旋转和/或缩放图像的图像并将结果存储在画布上。该函数需要以下参数：

- `canvas` 指向画布对象以存储转换结果的指针。
- `img pointer` 转换为图像描述符。也可以是其他画布的图像描述符（`lv_canvas_get_img()`）。
- `angle` 旋转角度（0..3600），0.1 度分辨率
- `zoom` 缩放系数（256 不缩放，512 倍大，128 倍大）
- `offset_x` 偏移量 X 告诉将结果数据放在目标画布上的位置
- `offset_y` 偏移量 X 告诉将结果数据放在目标画布上的位置
- `pivot_x` 旋转的 X 轴。相对于源画布。设置为绕中心旋转 `source width / 2`
- `pivot_y` 旋转的枢轴 Y。相对于源画布。设置为绕中心旋转 `source height / 2`
- `antialias` true：在转换过程中应用抗锯齿。看起来更好，但速度较慢。

请注意，画布无法自身旋转。您需要源和目标画布或图像。

14.3.5 模糊

画布的给定区域可以用水平和垂直模糊。 是模糊的半径（值越大表示毛刺强度越大）。是应该应用模糊的区域（相对于画布进行解释）。

`lv_canvas_blur_hor(canvas, &area, r)lv_canvas_blur_ver(canvas, &area, r)rarea`

14.4 事件

默认情况下，画布的单击被禁用（由 Image 继承），因此不生成任何事件。

如果启用了单击（），则仅通用事件按对象类型发送。`lv_obj_set_click(canvas, true)`

14.5 按键

对象类型不处理任何 *键*。

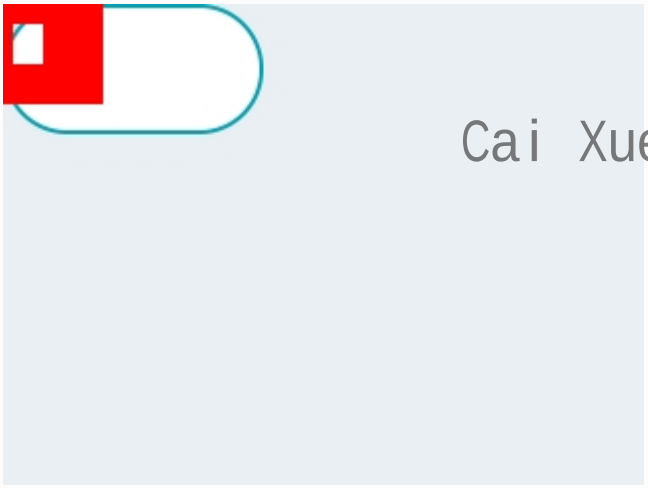
例

C

在画布上绘图并旋转

码

带色度键的透明画布



API

typedef

typedef uint8_t `lv_canvas_part_t`

枚举

Enum [anonymous]

值:

enumerator `LV_CANVAS_PART_MAIN`

功能

lv_obj_t * lv_canvas_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个画布对象

返回

指向创建的画布的指针

参数

- `par`：指向对象的指针，它将是新画布的父对象
- `copy`：指向画布对象的指针，如果不为 NULL，则将从其复制新对象

void lv_canvas_set_buffer（lv_obj_t * *canvas*，void * *buf*，lv_coord_t *w*，lv_coord_t *h*，lv_img_cf_t *cf* ）

为画布设置缓冲区。

参数

- `buf`：画布内容所在的缓冲区。所需的大小为（lv_img_color_format_get_px_size（cf）* w）/ 8 * h）可以使用它分配 `lv_mem_alloc()` 或可以是静态分配的数组（例如，静态 lv_color_t buf [100 * 50]），也可以是 RAM 中的地址或外部地址 SRAM
- `canvas`：指向画布对象的指针

- `w`：画布的宽度
- `h`：画布的高度
- `cf`：颜色格式。 `LV_IMG_CF_...`

voidlv_canvas_set_px（lv_obj_t * *canvas*，lv_coord_t *x*，lv_coord_t *y*，lv_color_t *c* ）

设置画布上像素的颜色

参数

- `canvas`：
- `x`：要设置点的 x 坐标
- `y`：要设置点的 x 坐标
- `c`：点的颜色

voidlv_canvas_set_palette（lv_obj_t * *canvas*，uint8_t *id*，lv_color_t *c* ）

使用索引格式设置画布的调色板颜色。仅对 `LV_IMG_CF_INDEXED1/2/4/8`

参数

- `canvas`：指向画布对象的指针
- `id`：要设置的调色板颜色：
  - 为 `LV_IMG_CF_INDEXED1`：0..1
  - 为 `LV_IMG_CF_INDEXED2`：0..3
  - 为 `LV_IMG_CF_INDEXED4`：0..15
  - 为 `LV_IMG_CF_INDEXED8`：0..255
- `c`：要设置的颜色

lv_color_t lv_canvas_get_px（lv_obj_t * *canvas*，lv_coord_t *x*，lv_coord_t *y* ）

获取画布上像素的颜色

返回

点的颜色

参数

- `canvas`：
- `x`：要设置点的 x 坐标
- `y`：要设置点的 x 坐标

lv_img_dsc_t * lv_canvas_get_img（lv_obj_t * *canvas*）

获取画布的图像作为指向 `lv_img_dsc_t` 变量的指针。

返回

指向图像描述符的指针。

参数

- `canvas`：指向画布对象的指针

voidlv_canvas_copy_buf（lv_obj_t * *canvas*，*const*void* *to_copy*，lv_coord_t *x*，lv_coord_t *y*，lv_coord_t *w*，lv_coord_t *h* ）

将缓冲区复制到画布

参数

- `canvas`：指向画布对象的指针

- `to_copy`：要复制的缓冲区。颜色格式必须与画布的缓冲区颜色格式匹配
- `x`：目标位置的左侧
- `y`：目标位置的顶部
- `w`：要复制的缓冲区的宽度
- `h`：要复制的缓冲区的高度

void lv_canvas_transform（lv_obj_t * *canvas*，lv_img_dsc_t * *img*，int16_t *angle*，uint16_t *zoom*，lv_coord_t *offset_x*，lv_coord_t *offset_y*，*int32_tivot_x*，*int32_tivot_y*，bool *antialias* ）

转换并成像并将结果存储在画布上。

参数

- `canvas`：指向画布对象的指针，以存储转换结果。
- `img`：指向要转换的图像描述符的指针。也可以是其他画布的图像描述符（`lv_canvas_get_img()`）。
- `angle`：旋转角度（0..3600），0.1 度分辨率
- `zoom`：变焦倍数（256 无变焦）；
- `offset_x`：偏移 X 以告知将结果数据放在目标画布上的位置
- `offset_y`：偏移 X 以告知将结果数据放在目标画布上的位置
- `pivot_x`：旋转 X 轴。相对于源画布设置为围绕中心旋转 `source width / 2`
- `pivot_y`：旋转的枢轴 Y。相对于源画布设置为围绕中心旋转 `source height / 2`
- `antialias`：在转换过程中应用抗锯齿。看起来更好，但速度较慢。

voidlv_canvas_blur_hor（lv_obj_t * *canvas*，*const*lv_area_t * *area*，uint16_t *r* ）

在画布上应用水平模糊

参数

- `canvas`：指向画布对象的指针
- `area`：要模糊的区域。如果 `NULL` 整个画布会模糊。
- `r`：模糊半径

void lv_canvas_blur_ver（lv_obj_t * canvas，const lv_area_t * area，uint16_t r ）

在画布上应用垂直模糊

参数

- canvas：指向画布对象的指针
- area：要模糊的区域。如果 NULL 整个画布会模糊。
- r：模糊半径

void lv_canvas_fill_bg（lv_obj_t * canvas，lv_color_t color，lv_opa_t opa ）

用颜色填充画布

参数

- canvas：指向画布的指针
- color：背景色
- opa：所需的不透明度

void lv_canvas_draw_rect（lv_obj_t * canvas，lv_coord_t x，lv_coord_t y，lv_coord_t w，lv_coord_t h，const lv_draw_rect_dsc_t * rect_dsc ）

在画布上画一个矩形

参数

- canvas：指向画布对象的指针
- x：矩形的左坐标
- y：矩形的顶部坐标
- w：矩形的宽度
- h：矩形的高度
- rect_dsc：矩形的描述符

void lv_canvas_draw_text（lv_obj_t * canvas，lv_coord_t x，lv_coord_t y，lv_coord_t max_w，lv_draw_label_dsc_t * label_draw_dsc，const char * txt，lv_label_align_t align ）

在画布上绘制文本。

参数

- canvas：指向画布对象的指针

- `x`：文本的左坐标
- `y`：文本的顶部坐标
- `max_w`：文字的最大宽度。文本将被包装以适合此大小
- `label_draw_dsc`：指向有效标签描述符的指针 `lv_draw_label_dsc_t`
- `txt`：要显示的文字
- `align`：对齐文字（`LV_LABEL_ALIGN_LEFT/RIGHT/CENTER`）

void lv_canvas_draw_img（lv_obj_t * *canvas*，lv_coord_t *x*，lv_coord_t *y*，*const* void* *src*，*const* lv_draw_img_dsc_t * *img_draw_dsc* ）

在画布上绘制图像

参数

- `canvas`：指向画布对象的指针
- `x`：图像的左坐标
- `y`：图像的最高坐标
- `src`：图像源。可以是 `lv_img_dsc_t` 变量的指针或图像的 path。
- `img_draw_dsc`：指向有效标签描述符的指针 `lv_draw_img_dsc_t`

void lv_canvas_draw_line（lv_obj_t * *canvas*，*const* lv_point_t *points* []，uint32_t *point_cnt*，*const* lv_draw_line_dsc_t * *line_draw_dsc* ）

在画布上画一条线

参数

- `canvas`：指向画布对象的指针
- `points`：线的点
- `point_cnt`：点数
- `line_draw_dsc`：指向初始化 `lv_draw_line_dsc_t` 变量的指针

void lv_canvas_draw_polygon（lv_obj_t * *canvas*，*const* lv_point_t *points* [ ]，uint32_t *point_cnt*，*const* lv_draw_rect_dsc_t * *poly_draw_dsc* ）

在画布上绘制多边形

参数

- `canvas`：指向画布对象的指针
- `points`：多边形的点
- `point_cnt`：点数
- `poly_draw_dsc`：指向初始化 `lv_draw_rect_dsc_t` 变量的指针

void lv_canvas_draw_arc（lv_obj_t * *canvas*，lv_coord_t *x*，lv_coord_t *y*，lv_coord_t *r*，int32_t *start_angle*，int32_t *end_angle*，*const*lv_draw_line_dsc_t * *arc_draw_dsc* ）

在画布上画圆弧

参数

- `canvas`：指向画布对象的指针
- `x`：弧线的 origo x
- `y`：弧线的起源
- `r`：圆弧半径
- `start_angle`：起始角度（度）
- `end_angle`：结束角度（度）
- `arc_draw_dsc`：指向初始化 `lv_draw_line_dsc_t` 变量的指针

struct lv_canvas_ext_t

公众成员

lv_img_ext_t **img**

lv_img_dsc_t **dsc**

第十五章 复选框（lv_cb）

15.1 总览

Checkbox 对象是从 Button 背景构建的，Button 背景还包含 Button *项目 sign* 和 Label，以实现经典的复选框。

15.2 小部件和样式

复选框的主要部分称为 `LV_CHECKBOX_PART_BG`。它是“项目 sign”及其旁边的文本的容器。背景使用所有典型的背景样式属性。

项目 sign 是真正的 lv_obj 对象，可以使用引用 `LV_CHECKBOX_PART_BULLET`。项目 sign 会自动继承背景的状态。因此，背景被按下时，子弹也会进入按下状态。项目 sign 还使用所有典型的背景样式属性。

标签没有专用部分。由于*文本*样式属性始终是继承的，因此可以在背景样式中设置其样式。

15.3 用法

15.3.1 文本

可以通过功能修改文本。它将动态分配文本。`lv_checkbox_set_text(cb, "New text")`

要设置静态文本，请使用。这样，将仅存储的指针，并且在存在复选框的情况下不应释放该指针。`lv_checkbox_set_static_text(cb, txt)txt`

15.3.2 选中/取消选中

您可以通过手动选中/取消选中复选框。设置将选中该复选框，而将取消选中该复选框。`lv_checkbox_set_checked(cb, true/false)truefalse`

15.3.3 禁用复选框

要禁用复选框，请使用。`lv_checkbox_set_disabled(cb, true)`

15.3.4 获取/设置复选框状态

您可以使用 `lv_checkbox_get_state(cb)` 返回当前状态的函数来获取 Checkbox 的当前状态。您可以使用设置复选框的当前状态。枚举定义的可用状态为：

`lv_checkbox_set_state(cb, state)lv_btn_state_t`

- **LV_BTN_STATE_RELEASED**
- **LV_BTN_STATE_PRESSED**
- **LV_BTN_STATE_DISABLED**
- **LV_BTN_STATE_CHECKED_RELEASED**
- **LV_BTN_STATE_CHECKED_PRESSED**
- **LV_BTN_STATE_CHECKED_DISABLED**

15.4 事件

除了常规事件外，复选框还发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED**-切换复选框时发送。

请注意，与通用输入设备相关的事件（如 `LV_EVENT_PRESSED` ）也以非活动状态发送。您需要检查状态 `lv_cb_is_inactive(cb)` 以忽略非活动复选框中的事件。

了解有关事件的更多信息。

15.5 按键

以下*按键*由“按钮”处理：

- **LV_KEY_RIGHT / UP**-如果启用了切换，则进入切换状态
- **LV_KEY_LEFT / DOWN**-如果启用了切换，则进入非切换状态

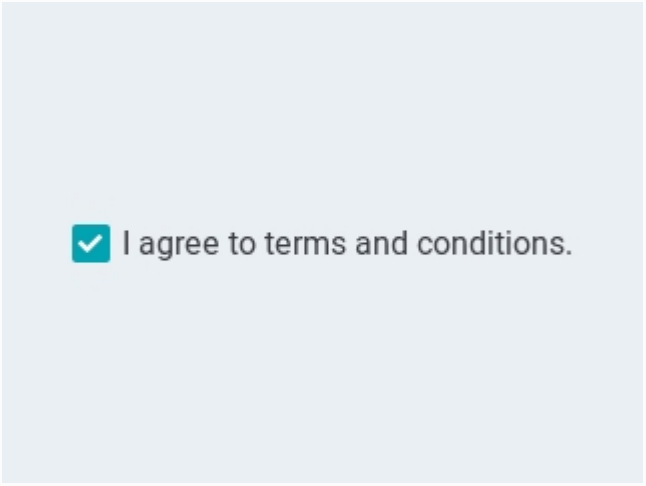
请注意，照常将的状态 `LV_KEY_ENTER` 转换为 `LV_EVENT_PRESSED/PRESSING/RELEASED` etc。

了解更多有关按键的信息。

例

C

简单复选框



API

typedef

typedef uint8_t **lv_checkbox_style_t**

枚举

Enum [anonymous]

复选框样式。

值：

enumerator **LV_CHECKBOX_PART_BG**= LV_BTN_PART_MAIN

对象背景样式。

enumerator **_LV_CHECKBOX_PART_VIRTUAL_LAST**

enumerator **LV_CHECKBOX_PART_BULLET**= _LV_BTN_PART_REAL_LAST

包装盒样式（已发布）。

enumerator **_LV_CHECKBOX_PART_REAL_LAST**

功能

lv_obj_t * **lv_checkbox_create**（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建一个复选框对象

返回

指向创建的复选框的指针

参数

- `par`：指向对象的指针，它将是新复选框的父对象
- `copy`：指向复选框对象的指针，如果不为 NULL，则将从其复制新对象

void lv_checkbox_set_text（lv_obj_t * cb，const char* txt ）

设置复选框的文本。`txt` 将被复制，并且在此函数返回后可能会被释放。

参数

- `cb`：指向复选框的指针
- `txt`：复选框的文本。使用 NULL 刷新当前文本。

void lv_checkbox_set_text_static（lv_obj_t * cb，const char* txt ）

设置复选框的文本。`txt` 在此复选框有效期内不得释放。

参数

- `cb`：指向复选框的指针
- `txt`：复选框的文本。使用 NULL 刷新当前文本。

void lv_checkbox_set_checked（lv_obj_t * cb，bool checked）

设置复选框的状态

参数

- `cb`：指向复选框对象的指针
- `checked`：true：选中复选框；false：取消选中

void lv_checkbox_set_disabled（lv_obj_t * cb ）

使复选框处于非活动状态（禁用）

参数

- `cb`：指向复选框对象的指针

void lv_checkbox_set_state（lv_obj_t * cb，lv_btn_state_t state）

设置复选框的状态

参数

- `cb`：指向复选框对象的指针
- `state`：复选框的新状态（来自 lv_btn_state_t 枚举）

const 字符* lv_checkbox_get_text（const lv_obj_t * cb ）

获取复选框的文本

返回

指向复选框文本的指针

参数

- cb：指向复选框对象的指针

bool lv_checkbox_is_checked（const lv_obj_t * cb ）

获取复选框的当前状态

返回

true：选中；false：未检查

参数

- cb：指向复选框对象的指针

bool lv_checkbox_is_inactive（const lv_obj_t * cb ）

获取复选框是否处于非活动状态。

返回

true：不活动；false：未激活

参数

- cb：指向复选框对象的指针

lv_btn_state_t lv_checkbox_get_state（const lv_obj_t * cb ）

获取复选框的当前状态

返回

复选框的状态（来自 lv_btn_state_t 枚举）

参数

- cb：指向复选框对象的指针

struct lv_checkbox_ext_t

公众成员

lv_btn_ext_t bg_btn

lv_obj_t *bullet

lv_obj_t *label

Cai Xuefeng

第十六章 图表（lv_chart）

16.1 总览

图表是可视化数据点的基本对象。它们支持*折线*图（将点与线连接和/或在其上绘制点）和*柱形图*。

图表还支持分隔线，2 y 轴，刻度线和刻度线文本。

16.2 小部件和样式

图表的主要部分被调用 `LV_CHART_PART_BG`，它使用所有典型的背景属性。该*文本*样式属性确定轴文本的风格和*线条*属性决定了蜱虫的风格。 *填充*值在侧面增加了一些空间，因此使*序列区域*更小。填充也可用于为轴文本和刻度线留出空间。

该系列的背景称为 `LV_CHART_PART_SERIES_BG`，并放置在主要背景上。在此部分上绘制了分隔线和系列数据。除典型的背景样式属性外，分割线还使用*线型*属性。所述*填充*值告诉该部分与轴的文本之间的空间。

该系列的样式可以通过引用 `LV_CHART_PART_SERIES`。对于列类型，使用以下属性：

- *radius*：条的半径
- *padding_inner*：相同 x 坐标的列之间的间隔

如果是线型，则使用以下属性：

- *线属性*来描述线
- 点的*大小*半径
- *bg_opa*：线条下方区域的整体不透明度
- *bg_main_stop*：的%*bg_opa* 在顶部以创建一个 alpha 褪色（0：在顶部透明，255：*bg_opa* 在顶部）
- *bg_grad_stop*：底部 *bg_opa* *的百分比*以创建 alpha 渐变（0：底部透明，255：*bg_opa* 顶部）
- *bg_drag_dir*：应该 `LV_GRAD_DIR_VER` 允许通过 *bg_main_stop* 和 *bg_grad_stop* 进行 Alpha 淡入

16.3 用法

16.3.1 数据系列

您可以通过将任意数量的系列添加到图表。如果不使用外部数组，它将为包含所选数据点和数组的 struct 的 struct 分配数据，如果分配了外部数组，则将与该系列关联的任何内部点都释放，而将系列点分配给该外部数组。`lv_chart_add_series(chart, color)lv_chart_series_tcolor`

16.3.2 系列类型

存在以下**数据显示类型**：

- **LV_CHART_TYPE_NONE-**不显示任何数据。可用于隐藏系列。
- **LV_CHART_TYPE_LINE-**在两点之间画线。
- **LV_CHART_TYPE_COLUMN-**绘制列。

您可以使用指定显示类型。可以对类型进行“或”运算（如）。

`lv_chart_set_type(chart, LV_CHART_TYPE_...)LV_CHART_TYPE_LINE`

16.3.3 修改数据

您有几个选项可以设置系列数据:Cai Xuefeng

1. 在数组之类的中手动设置值，并使用刷新图表。

`ser1->points[3] = 7lv_chart_refresh(chart)`

2. 使用 id 是您要更新的点的索引。`lv_chart_set_point_id(chart, ser, value, id)`
3. 使用。`lv_chart_set_next(chart, ser, value)`
4. 将所有点初始化为给定值：。`lv_chart_init_points(chart, ser, value)`
5. 使用设置数组中的所有点。`lv_chart_set_points(chart, ser, value_array)`

使用 `LV_CHART_POINT_DEF` 的值使库跳过绘制点，列或线段。

16.3.4 覆盖系列的默认起点

如果希望绘图从默认点（序列的点[0]）以外的其他点开始，则可以使用以下函数设置替代索引，其中 id 是要从其开始绘图的新索引位置。`lv_chart_set_x_start_point(chart, ser, id)`

16.3.5 设置外部数据源

您可以通过向外部数据源分配以下功能来使图表系列更新： 其中 array 是 lv_coord_t 的外部数组，具有 point_cnt 元素。注意：您应该在外部数据源更新后调用，以更新图表。

```
lv_chart_set_ext_array(chart, ser, array, point_cnt )lv_chart_refresh(chart)
```

16.3.6 获取当前图表信息

有四个功能可获取有关图表的信息：

1. `lv_chart_get_type(chart)` 返回当前图表类型。
2. `lv_chart_get_point_count(chart)` 返回当前图表点数。
3. `lv_chart_get_x_start_point(ser)` 返回指定系列的当前绘图索引。
4. `lv_chart_get_point_id(chart, ser, id)` 返回指定系列的特定索引处的数据值。

16.3.7 更新方式

`lv_chart_set_next` 可以根据*更新模式*以两种方式*运行*：

- **LV_CHART_UPDATE_MODE_SHIFT-**将旧数据向左移动，然后向右添加新数据。
- **LV_CHART_UPDATE_MODE_CIRCULAR-**循环添加新数据（如 ECG 图）。

可以使用更改更新模式。`lv_chart_set_update_mode(chart, LV_CHART_UPDATE_MODE_...)`

16.3.8 点数

序列中的点数可以通过修改。默认值为 10。注意：当将外部缓冲区分配给序列时，这也会影响处理的点数。`lv_chart_set_point_count(chart, point_num)`

16.3.9 垂直范围

您可以使用来指定 y 方向上的最小值和最大值。点的值将按比例缩放。默认范围是：0..100。

```
lv_chart_set_range(chart, y_min, y_max)
```

16.3.10 分割线

水平和垂直分隔线的数量可以通过修改。默认设置为 3 条水平分割线和 5 条垂直分割线。

```
lv_chart_set_div_line_count(chart, hdiv_num, vdiv_num)
```

16.3.11 刻度线和标签

刻度和标签可以添加到轴上。

`lv_chart_set_x_tick_text(chart, list_of_values, num_tick_marks, LV_CHART_AXIS_...)`在 x 轴上设置刻度和文本。`list_of_values`是一个带有`'\n'`终止文本（期望最后一个）且带有用于刻度的文本的字符串。例如。可以。如果设置为 if ，则告诉两个标签之间的刻度数。如果是，则它指定总的滴答数。

```
const char * list_of_values = "first\nsec\nthird"list_of_valuesNULLlist_of_valuesnum_tick_markslist_of_va
```

```
luesNULL
```

*主刻度线*绘制在放置文本的位置，*次刻度线*绘制在其他位置。设置 x 轴上刻度线的长度。


```
lv_chart_set_x_tick_length(chart, major_tick_len, minor_tick_len)
```

y 轴也存在相同的功能：`lv_chart_set_y_tick_text`和`lv_chart_set_y_tick_length`。

16.4 事件

仅通用事件是按对象类型发送的。

了解有关事件的更多信息。

16.5 按键

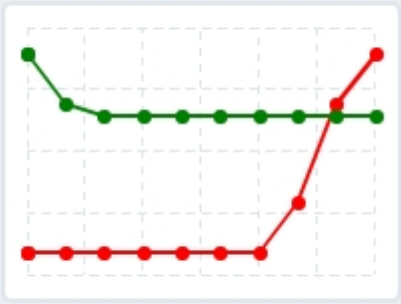
对象类型不处理任何*键*。

了解更多有关按键的信息。

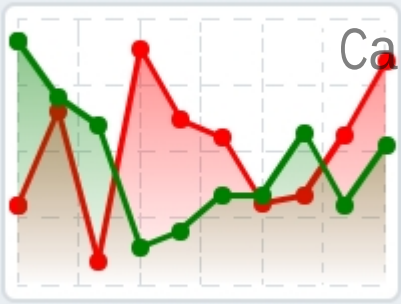
例

C

折线图



码



API

typedef

typedef uint8_t **lv_chart_type_t**

typedef uint8_t **lv_chart_update_mode_t**

typedef uint8_t **lv_chart_axis_t**

typedef uint8_t **lv_chart_axis_options_t**

枚举

Enum **[anonymous]**

图表类型

值：

enumerator **LV_CHART_TYPE_NONE** = 0x00

不要画系列

enumerator **LV_CHART_TYPE_LINE** = 0x01

用线连接点

enumerator **LV_CHART_TYPE_COLUMN** = 0x02

绘制列

enumerator **LV_CHART_TYPE_SCATTER** = 0x03

X / Y 图表，点和/或线

Enum [anonymous]

图表更新模式 `lv_chart_set_next`

值：

enumerator **LV_CHART_UPDATE_MODE_SHIFT**

向左移动旧数据，向右添加新数据

enumerator **LV_CHART_UPDATE_MODE_CIRCULAR**

循环添加新数据

Enum [anonymous]

值：

enumerator **LV_CHART_AXIS_PRIMARY_Y**

enumerator **LV_CHART_AXIS_SECONDARY_Y**

enumerator **_LV_CHART_AXIS_LAST**

enum [anonymous]

轴数据

值：

enumerator **LV_CHART_AXIS_SKIP_LAST_TICK** = 0x00

不要画最后的刻度

enumerator **LV_CHART_AXIS_DRAW_LAST_TICK** = 0x01

画最后一个刻度

enumerator **LV_CHART_AXIS_INVERSE_LABELS_ORDER** = 0x02

以倒序绘制刻度标签

Enum [anonymous]

值：

enumerator **LV_CHART_PART_BG**= LV_OBJ_PART_MAIN

enumerator **LV_CHART_PART_SERIES_BG**= _LV_OBJ_PART_VIRTUAL_LAST

enumerator **LV_CHART_PART_SERIES**

功能

LV_EXPORT_CONST_INT（LV_CHART_POINT_DEF）

LV_EXPORT_CONST_INT（LV_CHART_TICK_LENGTH_AUTO）

lv_obj_t * **lv_chart_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建图表背景对象

返回

指向创建的图表背景的指针

参数

- `par`：指向对象的指针，它将是新图表背景的父对象
- `copy`：指向图表背景对象的指针，如果不为 NULL，则将从其复制新对象

---

lv_chart_series_t * **lv_chart_add_series**（lv_obj_t * *chart*，lv_color_t *color*）

分配数据系列并将其添加到图表

返回

指向已分配数据系列的指针

参数

- `chart`：指向图表对象的指针
- `color`：数据系列的颜色

---

voidlv_chart_clear_series（lv_obj_t * *chart*，lv_chart_series_t * *series*）

明确系列的要点

参数

- `chart`：指向图表对象的指针
- `series`：指向要清除的图表系列的指针

void lv_chart_set_div_line_count（lv_obj_t * *chart*，uint8_t *hdiv*，uint8_t *vdiv* ）

设置水平和垂直分割线的数量

参数

- `chart`：指向图形背景对象的指针
- `hdiv`：水平分割线数
- `vdiv`：垂直分割线数

void lv_chart_set_y_range（lv_obj_t * *chart*，lv_chart_axis_t *axis*，lv_coord_t *ymin*，lv_coord_t *ymax* ）

在轴上设置最小和最大 y 值

参数

- `chart`：指向图形背景对象的指针
- `axis`：`LV_CHART_AXIS_PRIMARY_Y` 或 `LV_CHART_AXIS_SECONDARY_Y`
- `ymin`：y 最小值
- `ymax`：y 最大值

void lv_chart_set_type（lv_obj_t * *chart*，lv_chart_type_t *type*）

设置图表的新类型

参数

- `chart`：指向图表对象的指针
- `type`：图表的新类型（来自“ lv_chart_type_t”枚举）

void lv_chart_set_point_count（lv_obj_t * *chart*，uint16_t *point_cnt* ）

设置图表上数据线上的点数

参数

- `chart`：指向图表对象的指针 r
- `point_cnt`：数据线上的新点数

void lv_chart_init_points（lv_obj_t * *chart*，lv_chart_series_t * *ser*，lv_coord_t *y* ）

用一个值初始化所有数据点

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `y`：所有点的新值

void lv_chart_set_points（lv_obj_t * *chart*，lv_chart_series_t * *ser*，lv_coord_t *y_array* [] ）

设置数组中的点的值

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `y_array`：'lv_coord_t'点的数组（带有'points count'元素）

void lv_chart_set_next（lv_obj_t * *chart*，lv_chart_series_t * *ser*，lv_coord_t *y* ）

向右移动所有数据并在数据线上设置最右边的数据

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `y`：最正确数据的新价值

void lv_chart_set_update_mode（lv_obj_t * *chart*，lv_chart_update_mode_t *update_mode* ）

设置图表对象的更新模式。

参数

- `chart`：指向图表对象的指针
- `update`：模式

void lv_chart_set_x_tick_length（lv_obj_t * *chart*，uint8_t *major_tick_len*，uint8_t *minor_tick_len* ）

设置 x 轴上刻度线的长度

参数

- `chart`：指向图表的指针
- `major_tick_len`：主要刻度线的长度或 `LV_CHART_TICK_LENGTH_AUTO` 自动设置（添加标签的位置）

- `minor_tick_len`：次刻度的长度，`LV_CHART_TICK_LENGTH_AUTO`可自动设置（不添加标签）

void lv_chart_set_y_tick_length（lv_obj_t * chart，uint8_t major_tick_len，uint8_t minor_tick_len ）

设置 y 轴上刻度线的长度

参数

- `chart`：指向图表的指针
- `major_tick_len`：主要刻度线的长度或`LV_CHART_TICK_LENGTH_AUTO`自动设置（添加标签的位置）
- `minor_tick_len`：次刻度的长度，`LV_CHART_TICK_LENGTH_AUTO`可自动设置（不添加标签）

void lv_chart_set_secondary_y_tick_length（lv_obj_t * chart，uint8_t major_tick_len，uint8_t minor_tick_len ）

设置次要 y 轴上刻度线的长度

参数

- `chart`：指向图表的指针

- `major_tick_len`：主要刻度线的长度或`LV_CHART_TICK_LENGTH_AUTO`自动设置（添加标签的位置）
- `minor_tick_len`：次刻度的长度，`LV_CHART_TICK_LENGTH_AUTO`可自动设置（不添加标签）

void lv_chart_set_x_tick_texts（lv_obj_t * chart，const char * list_of_values，uint8_t num_tick_marks，lv_chart_axis_options_t options ）

设置图表的 X 轴刻度计数和标签

参数

- `chart`：指向图表对象的指针
- `list_of_values`：字符串值列表，以终止

  ，除了最后一个

- `num_tick_marks`：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数
- `options`：额外的选择

void lv_chart_set_secondary_y_tick_texts（lv_obj_t * chart，const char * list_of_values，uint8_t num_tick_marks，lv_chart_axis_options_t options ）

设置次要 Y 轴刻度计数和图表标签

参数

- `chart`：指向图表对象的指针
- `list_of_values`：字符串值列表，以终止

，除了最后一个

- `num_tick_marks`：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数
- `options`：额外的选择

void lv_chart_set_y_tick_texts（lv_obj_t * chart，const char * list_of_values，uint8_t num_tick_marks，lv_chart_axis_options_t options ）

设置图表的 Y 轴刻度计数和标签

参数

- `chart`：指向图表对象的指针

- `list_of_values`：字符串值列表，以终止

，除了最后一个

- `num_tick_marks`：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数
- `options`：额外的选择

void lv_chart_set_x_start_point（lv_obj_t * chart，lv_chart_series_t * ser，uint16_t id ）

设置数据数组中 x 轴起点的索引

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `id`：数据数组中 x 点的索引

void lv_chart_set_ext_array（lv_obj_t * chart，lv_chart_series_t * ser，lv_coord_t array []，uint16_t point_cnt ）

设置要用于图表的外部数据点阵列注意：用户有责任确保 point_cnt 与外部阵列大小匹配。

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `array`：图表的外部点数组

void **lv_chart_set_point_id**（lv_obj_t * *chart*，lv_chart_series_t * *ser*，lv_coord_t *value*，uint16_t *id* ）

直接基于索引在图表系列中设置单个点值

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `value`：分配给数组点的值
- `id`：数组中 x 点的索引

void **lv_chart_set_series_axis**（lv_obj_t * *chart*，lv_chart_series_t * *ser*，lv_chart_axis_t *axis*）

设置系列的 Y 轴

参数

- `chart`：指向图表对象的指针
- `ser`：指向系列的指针
- `axis`：`LV_CHART_AXIS_PRIMARY_Y` 或 `LV_CHART_AXIS_SECONDARY_Y`

lv_chart_type_t **lv_chart_get_type**（*const* lv_obj_t * *chart*）

获取图表类型

返回

图表类型（来自“ lv_chart_t”枚举）

参数

- `chart`：指向图表对象的指针

uint16_t **lv_chart_get_point_count**（*const* lv_obj_t * *chart*）

获取图表上每条数据线的数据点编号

返回

每条数据线上的点号

参数

- `chart`：指向图表对象的指针

---

uint16_t **lv_chart_get_x_start_point**（lv_chart_series_t * *ser*）

获取数据数组中 x 轴起点的当前索引

返回

数据数组中当前 x 起点的索引

参数

- `ser`：指向“图表”上的数据系列的指针

---

lv_coord_t **lv_chart_get_point_id**（lv_obj_t * *chart*，lv_chart_series_t * *ser*，uint16_t *id*）

直接基于索引获取图表系列中的单个点值

返回

索引 ID 处数组点的值

参数

- `chart`：指向图表对象的指针
- `ser`：指向“图表”上的数据系列的指针
- `id`：数组中 x 点的索引

---

lv_chart_axis_t **lv_chart_get_series_axis**（lv_obj_t * *chart*，lv_chart_series_t * *ser*）

获取系列的 Y 轴

返回

`LV_CHART_AXIS_PRIMARY_Y` 要么 `LV_CHART_AXIS_SECONDARY_Y`

参数

- `chart`：指向图表对象的指针
- `ser`：指向系列的指针

---

void**lv_chart_refresh**（lv_obj_t * *chart*）

如果其数据线已更改，则刷新图表

参数

- `chart`：指向图表对象的指针

struct lv_chart_series_t

公众成员

lv_coord_t *points

lv_color_tcolor

uint16_t start_point

uint8_t ext_buf_assigned

lv_chart_axis_ty_axis

struct lv_chart_axis_cfg_t

公众成员

const 字符*list_of_values

lv_chart_axis_options_toptions

uint8_t num_tick_marks

uint8_t major_tick_len

uint8_t minor_tick_len

struct lv_chart_ext_t

公众成员

lv_ll_t series_ll

lv_coord_t ymin[_LV_CHART_AXIS_LAST]

lv_coord_t ymax[_LV_CHART_AXIS_LAST]

uint8_t hdiv_cnt

uint8_t vdiv_cnt

uint16_t point_cnt

lv_style_list_t style_series_bg

lv_style_list_t style_series

lv_chart_type_ttype

lv_chart_axis_cfg_ty_axis

lv_chart_axis_cfg_tx_axis

lv_chart_axis_cfg_tsecondary_y_axis

uint8_t update_mode

第十七章 容器（lv_cont）

17.1 总览

容器本质上是具有布局和自动调整大小功能的**基本对象**。

17.2 小部件和样式

容器只有一种主要样式 `LV_CONT_PART_MAIN`，可以使用所有典型的免费属性和填充来自动调整布局大小。

17.3 用法

17.3.1 布局

您可以在容器上应用布局以自动订购其子代。布局间距来自样式的 `pad` 属性。可能的布局选项：

- **LV_LAYOUT_OFF-**不要对齐子代。
- **LV_LAYOUT_CENTER-**将子项与列中的中心对齐，并 `padd_inner` 在它们之间保持**间距**。
- **LV_LAYOUT_COLUMN_LEFT-**在左对齐的列中对齐子级。请 `padd_left` 在左边，空间 `pad_top` 空间的顶部和 `pad_inner` 孩子之间的空间。
- **LV_LAYOUT_COLUMN_MID-**在中心列中对齐子代。 `padd_top` 在顶部和 `pad_inner` 孩子之间保持空间。
- **LV_LAYOUT_COLUMN_RIGHT-**在右对齐的列中对齐子代。保持 `padd_right` 右边的 `pad_top` 空间，顶部的 `pad_inner` 空间和孩子之间的空间。
- **LV_LAYOUT_ROW_TOP-**在顶部对齐的行中对齐子级。请 `pad_left` 在左边，空间 `pad_top` 空间的顶部和 `pad_inner` 孩子之间的空间。
- **LV_LAYOUT_ROW_MID-**在居中的行中对齐子级。 `pad_left` 在左边和 `pad_inner` 孩子之间保持空间。

- **LV_LAYOUT_ROW_BOTTOM-**在底部对齐的行中对齐子级。请 `pad_left` 在左边，空间 `pad_bottom` 空间的底部和 `pad_inner` 孩子之间的空间。
- **LV_LAYOUT_PRETTY_TOP -**将作为连续多的对象可能（至少 `pad_inner` 空间和 `pad_left/right` 空间两侧）。在孩子之间的每一行中平均分配空间。如果这是连续不同身高的孩子，请对齐其上边缘。
- **LV_LAYOUT_PRETTY_MID-**与 **LV_LAYOUT_PRETTY_MID** 相同，`LV_LAYOUT_PRETTY_TOP` 但是如果此处的孩子连续排成不同的高度，则对齐他们的中线。
- **LV_LAYOUT_PRETTY_BOTTOM-**与 **LV_LAYOUT_PRETTY_BOTTOM** 相同，`LV_LAYOUT_PRETTY_TOP` 但是如果这是连续高度不同的子项，请对齐其底线。
- **LV_LAYOUT_GRID-**类似于 `LV_LAYOUT_PRETTY` 但不能平均划分水平空间，只是让它们之间的 `pad_left/right` 边缘和 `pad_inner` 空间分开。

17.3.2 自动调整

容器具有自动适应功能，可以根据其子代和/或父代自动更改容器的大小。存在以下选项：

- **LV_FIT_NONE-**不要自动更改大小。
- **LV_FIT_TIGHT-**将容器收缩包装在其所有子容器周围，同时 `pad_top/bottom/left/right` 在边缘保留空间。
- **LV_FIT_PARENT-**将大小设置为父项的大小减去 `pad_top/bottom/left/right`（来自父项的样式）空间。
- **LV_FIT_MAX -**使用 `LV_FIT_PARENT` 而不是父小，`LV_FIT_TIGHT` 时大。它将确保该容器至少是其父容器的大小。

要为所有方向设置自动适合模式，请使用。要在水平和垂直方向上使用不同的自动拟合，请使用。要在所有四个方向上使用不同的自动拟合，请使用。

```
lv_cont_set_fit(cont, LV_FIT_...)lv_cont_set_fit2(cont, hor_fit_type, ver_fit_type)lv_cont_set_fit4(cont, left_fit_type, right_fit_type, top_fit_type, bottom_fit_type)
```

17.4 事件

仅通用事件是按对象类型发送的。

17.5 按键

对象类型不处理任何 *键*。

例

C

自动装配的容器

API

typedef

typedef uint8_t **lv_layout_t**

typedef uint8_t **lv_fit_t**

枚举

Enum [anonymous]

容器布局选项

值：

enumerator **LV_LAYOUT_OFF** = 0

没有布局

enumerator **LV_LAYOUT_CENTER**

中心对象

enumerator **LV_LAYOUT_COLUMN_LEFT**

库伦：

- 将物体彼此下方放置
- 保持 `pad_top` 顶部空间
- 保持对象 `pad_inner` 之间的空间列左对齐

enumerator **LV_LAYOUT_COLUMN_MID**

列中间对齐

enumerator **LV_LAYOUT_COLUMN_RIGHT**

列右对齐

enumerator **LV_LAYOUT_ROW_TOP**

行：

- 将对象彼此相邻放置
- `pad_left` 在左侧保留空间
- 保持 `pad_inner` 物体之间的空间
- 如果应用布局的对象具有该行将从正确的应用空间开始行顶部对齐

`base_dir == LV_BIDI_DIR_RTLpad.right`

enumerator **LV_LAYOUT_ROW_MID**

行中间对齐

enumerator **LV_LAYOUT_ROW_BOTTOM**

行底对齐

enumerator **LV_LAYOUT_PRETTY_TOP**

漂亮：

- 将对象彼此相邻放置
- 如果没有更多空间，则开始新行
- 尊重 `pad_left` 并 `pad_right` 确定连续的可用空间
- `pad_inner` 在同一行中的对象之间保持空间
- `pad_inner` 在行之间保持对象之间的空间
- 平均划分剩余的水平空间行顶部对齐

enumerator **LV_LAYOUT_PRETTY_MID**

行中间对齐

enumerator **LV_LAYOUT_PRETTY_BOTTOM**

行底对齐

enumerator **LV_LAYOUT_GRID**

格

- 将对象彼此相邻放置
- 如果没有更多空间，则开始新行
- 尊重 `pad_left` 并 `pad_right` 确定连续的可用空间
- `pad_inner` 在同一行中的对象之间保持空间
- `pad_inner` 在行之间保持对象之间的空间
- 与不同的是 `PRETTY`，请 `GRID` 始终 `pad_inner` 在对象之间保持水平空间，以免将剩余的水平空间均分。

enumerator **_LV_LAYOUT_LAST**

enum **[anonymous]**

如何调整儿童周围容器的大小。

值：

enumerator **LV_FIT_NONE**

不要自动更改大小

enumerator **LV_FIT_TIGHT**

收缩包裹孩子

enumerator **LV_FIT_PARENT**

使尺寸与父母的边缘对齐

enumerator **LV_FIT_MAX**

首先将尺寸与父对象的边缘对齐，但是如果没有对象，则将其放大

enumerator **_LV_FIT_LAST**

enum **[anonymous]**

值：

enumerator **LV_CONT_PART_MAIN**= LV_OBJ_PART_MAIN

enumerator **_LV_CONT_PART_VIRTUAL_LAST**= _LV_OBJ_PART_VIRTUAL_LAST

enumerator **_LV_CONT_PART_REAL_LAST**= _LV_OBJ_PART_REAL_LAST

功能

lv_obj_t * lv_cont_create（lv_obj_t * par，const lv_obj_t * copy）

创建一个容器对象

返回

指向创建的容器的指针

参数

- `par`：指向对象的指针，它将是新容器的父对象
- `copy`：指向容器对象的指针，如果不为 NULL，则将从其复制新对象

void lv_cont_set_layout（lv_obj_t * cont，lv_layout_t layout）

在容器上设置布局

参数

- `cont`：指向容器对象的指针
- `layout`：来自“ lv_cont_layout_t”的布局

void lv_cont_set_fit4（lv_obj_t * cont，lv_fit_t left，lv_fit_t right，lv_fit_t top，lv_fit_t bottom）

分别在所有四个方向上设置适合策略。它告诉如何自动更改容器的大小。

参数

- `cont`：指向容器对象的指针
- `left`：从左适合政策 `lv_fit_t`
- `right`：合适的政策来自 `lv_fit_t`
- `top`：最适合的政策，来自 `lv_fit_t`
- `bottom`：自下而上的政策 `lv_fit_t`

void lv_cont_set_fit2（lv_obj_t * cont，lv_fit_t hor，lv_fit_t ver ）

分别水平和垂直设置适合策略。它告诉您如何自动更改容器的大小。

参数

- `cont`：指向容器对象的指针
- `hor`：来自的水平拟合政策 `lv_fit_t`
- `ver`：垂直适合政策，来自 `lv_fit_t`

void lv_cont_set_fit（lv_obj_t * cont，lv_fit_t fit ）

一次在所有 4 个方向上设置拟合策略。它告诉您如何自动更改容器的大小。

参数

- `cont`：指向容器对象的指针
- `fit`：符合政策 `lv_fit_t`

lv_layout_t lv_cont_get_layout（const lv_obj_t * cont）

获取容器的布局

返回

来自“ lv_cont_layout_t”的布局

参数

- `cont`：指向容器对象的指针

lv_fit_t lv_cont_get_fit_left（const lv_obj_t * cont）

获取容器的左拟合模式

返回

的元素 `lv_fit_t`

参数

- `cont`：指向容器对象的指针

lv_fit_t lv_cont_get_fit_right（const lv_obj_t * cont）

获取正确的容器安装方式

返回

的元素 `lv_fit_t`

参数

- `cont`：指向容器对象的指针

lv_fit_t lv_cont_get_fit_top（const lv_obj_t * cont）

获取容器的最适合模式

返回

的元素 `lv_fit_t`

参数

- `cont`：指向容器对象的指针

---

lv_fit_t lv_cont_get_fit_bottom（*const* lv_obj_t * *cont*）

获取容器的底部拟合模式

返回

的元素 `lv_fit_t`

参数

- `cont`：指向容器对象的指针

---

struct lv_cont_ext_t

公众成员

```
lv_layout_t layout
lv_fit_t fit_left
lv_fit_t fit_right
lv_fit_t fit_top
lv_fit_t fit_bottom
```

第十八章 颜色选择器（lv_cpicker）

18.1 总览

顾名思义，*拾色器*允许选择颜色。可以依次选择颜色的色相，饱和度和值。

小部件具有两种形式：圆形（圆盘）和矩形。

在这两种形式中，长按对象，颜色选择器将更改为颜色的下一个参数（色相，饱和度或值）。此外，双击将重置当前参数。

18.2 小部件和样式

拾色器的主要部分称为 `LV_CPICKER_PART_BG`。以圆形形式，它使用 *scale_width* 设置圆的宽度，并使用 *pad_inner* 在圆和内部预览圆之间填充。在矩形模式下，*半径*可以用于在矩形上应用半径。

该对象具有称为的虚拟部分 `LV_CPICKER_PART_KNOB`，它是在当前值上绘制的矩形（或圆形）。它使用所有矩形（如样式属性和填充）使其大于圆形或矩形背景的宽度。

18.3 用法

18.3.1 类型

可以通过以下方式更改颜色选择器的类型 `lv_cpicker_set_type(cpicker, LV_CPICKER_TYPE_RECT/DISC)`

18.3.2 设定颜色

该 colro 可以手动设置或一次性使用或

```
lv_cpicker_set_hue/saturation/value(cpicker, x)lv_cpicker_set_hsv(cpicker, hsv)lv_cpicker_set_color(cpick
er, rgb)
```

18.3.3 色彩模式

可以使用选择当前的色彩模式。

```
lv_cpicker_set_color_mode(cpicker, LV_CPICKER_COLOR_MODE_HUE/SATURATION/VALUE)
```

使用以下方法固定颜色（不要长按更改） `lv_cpicker_set_color_mode_fixed(cpicker, true)`

18.3.4 旋钮颜色

`lv_cpicker_set_knob_colored(cpicker, true)` 使旋钮自动将所选颜色显示为背景色。

18.4 事件

仅通用事件是按对象类型发送的。

18.5 按键

- **LV_KEY_UP**，**LV_KEY_RIGHT** 将当前参数的值增加 1
- **LV_KEY_DOWN**，**LV_KEY_LEFT** 将当前参数减 1
- **LV_KEY_ENTER** 长按将显示下一个模式。通过双击将重置当前参数。

例

C

Cai Xuefeng

光盘颜色选择器

API

typedef

typedef uint8_t lv_cpicker_type_t

typedef uint8_t lv_cpicker_color_mode_t

枚举

Enum [anonymous]

值：

enumerator LV_CPICKER_TYPE_RECT

enumerator LV_CPICKER_TYPE_DISC

enum [anonymous]

值：

enumerator LV_CPICKER_COLOR_MODE_HUE

enumerator LV_CPICKER_COLOR_MODE_SATURATION

enumerator LV_CPICKER_COLOR_MODE_VALUE

enum [anonymous]

值：

enumerator LV_CPICKER_PART_MAIN= LV_OBJ_PART_MAIN

enumerator LV_CPICKER_PART_KNOB= _LV_OBJ_PART_VIRTUAL_LAST

enumerator _LV_CPICKER_PART_VIRTUAL_LAST

enumerator _LV_CPICKER_PART_REAL_LAST= _LV_OBJ_PART_REAL_LAST

功能

lv_obj_t * lv_cpicker_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个拾色器对象

返回

指向创建的颜色选择器的指针

参数

- `par`：指向对象的指针，它将是新 colorpicker 的父对象
- `copy`：指向颜色选择器对象的指针，如果不为 NULL，则将从其复制新对象

void lv_cpicker_set_type（lv_obj_t * *cpicker*，lv_cpicker_type_t *type*）

为颜色选择器设置新类型

参数

- `cpicker`：指向颜色选择器对象的指针
- `type`：新型的颜色选择器（来自“ lv_cpicker_type_t”枚举）

bool lv_cpicker_set_hue（lv_obj_t * *cpicker*，uint16_t *hue*）

设置颜色选择器的当前色相。

返回

如果更改，则为 true，否则为 false

参数

- `cpicker`：指向 colorpicker 对象的指针
- `hue`：当前选择的色相[0..360]

bool lv_cpicker_set_saturation（lv_obj_t * *cpicker*，uint8_t *saturation*）

设置颜色选择器的当前饱和度。

返回

如果更改，则为 true，否则为 false

参数

- `cpicker`：指向 colorpicker 对象的指针
- `saturation`：当前选择的饱和度[0..100]

bool lv_cpicker_set_value（lv_obj_t * *cpicker*，uint8_t *val* ）

设置颜色选择器的当前值。

返回

如果更改，则为 true，否则为 false

参数

- `cpicker`：指向 colorpicker 对象的指针
- `val`：当前选择的值[0..100]

bool lv_cpicker_set_hsv（lv_obj_t * *cpicker*，lv_color_hsv_t *hsv* ）

设置颜色选择器的当前 hsv。

返回

如果更改，则为 true，否则为 false

参数

- `cpicker`：指向 colorpicker 对象的指针
- `hsv`：当前选择的 hsv

bool **lv_cpicker_set_color**（lv_obj_t * *cpicker*，lv_color_t *color*）

设置颜色选择器的当前颜色。

返回

如果更改，则为 true，否则为 false

参数

- `cpicker`：指向 colorpicker 对象的指针
- `color`：当前选择的颜色

void **lv_cpicker_set_color_mode**（lv_obj_t * *cpicker*，lv_cpicker_color_mode_t *mode*）

设置当前的色彩模式。

参数

- `cpicker`：指向 colorpicker 对象的指针
- `mode`：色彩模式（色相/饱和度/色度）

void **lv_cpicker_set_color_mode_fixed**（lv_obj_t * *cpicker*，bool *fixed*）

长时间按中心更改颜色模式时设置

参数

- `cpicker`：指向 colorpicker 对象的指针
- `fixed`：长按时无法更改色彩模式

void **lv_cpicker_set_knob_colored**（lv_obj_t * *cpicker*，bool *en* ）

使旋钮上色为当前颜色

参数

- `cpicker`：指向 colorpicker 对象的指针
- `en`：true：为旋钮着色；false：不给旋钮上色

lv_cpicker_color_mode_t **lv_cpicker_get_color_mode**（lv_obj_t * *cpicker* ）

获取当前的颜色模式。

返回

色彩模式（色相/饱和度/色度）

参数

- `cpicker`：指向 colorpicker 对象的指针

`bool` **lv_cpicker_get_color_mode_fixed**（*lv_obj_t * cpicker*）

长按中心时获取颜色模式是否更改

返回

长按无法更改模式

参数

- `cpicker`：指向 colorpicker 对象的指针

`uint16_t` **lv_cpicker_get_hue**（*lv_obj_t * cpicker*）

获取颜色选择器的当前色相。

返回

当前选择的色相

参数

- `cpicker`：指向 colorpicker 对象的指针

`uint8_t` **lv_cpicker_get_saturation**（*lv_obj_t * cpicker*）

获取选色器的当前饱和度。

返回

当前选择的饱和度

参数

- `cpicker`：指向 colorpicker 对象的指针

`uint8_t` **lv_cpicker_get_value**（*lv_obj_t * cpicker*）

获取颜色选择器的当前色相。

返回

当前选择值

参数

- `cpicker`：指向 colorpicker 对象的指针

`lv_color_hsv_t` **lv_cpicker_get_hsv**（*lv_obj_t * cpicker*）

获取当前选择的彩色选择器的 hsv。

返回

当前选择的 hsv

参数

- `cpicker`：指向 colorpicker 对象的指针

---

lv_color_t lv_cpicker_get_color（lv_obj_t * *cpicker*）

获取颜色选择器的当前选定颜色。

返回

当前选择的颜色

参数

- `cpicker`：指向 colorpicker 对象的指针

---

boollv_cpicker_get_knob_colored（lv_obj_t * *cpicker*）

旋钮是否着色为当前颜色

返回

true：为旋钮着色；false：不给旋钮上色

参数

- `cpicker`：指向颜色选择器对象的指针

---

struct lv_cpicker_ext_t

公众成员

lv_color_hsv_thsv
lv_style_list_t style_list
lv_point_t pos
uint8_t colored
struct lv_cpicker_ext_t :: [anonymous]knob
uint32_t last_click_time
uint32_t last_change_time
lv_point_t last_press_point
lv_cpicker_color_mode_tcolor_mode
uint8_t color_mode_fixed
lv_cpicker_type_ttype

十九章 下拉列表（lv_dropdown）

19.1 总览

下拉列表允许用户从列表中选择一个值。

下拉列表默认情况下处于关闭状态，并显示单个值或预定义的文本。激活后（通过单击下拉列表），将创建一个列表，用户可以从中选择一个选项。当用户选择新值时，该列表将被删除。

19.2 小部件和样式

调用下拉列表的主要部分，`LV_DROPDOWN_PART_MAIN`它是一个简单的 lv_obj 对象。它使用所有典型的背景属性。*按下*，*聚焦*，*编辑*等阶梯也照常应用。

单击主对象时创建的列表是 Page。它的背景部分可以被引用，`LV_DROPDOWN_PART_LIST`并为矩形本身使用所有典型的背景属性，并为选项使用文本属性。要调整选项之间的*间距*，*请*使用 *text_line_space* 样式属性。填充值可用于在边缘上留出一些空间。

页面的可滚动部分被隐藏，其样式始终为空（透明，无填充）。

滚动条可以被引用`LV_DROPDOWN_PART_SCRLBAR`并使用所有典型的背景属性。

可以`LV_DROPDOWN_PART_SELECTED`使用所有典型的背景属性引用并使用所选的选项。它将以其默认状态在所选选项上绘制一个矩形，并在按下状态下在被按下的选项上绘制一个矩形。

19.3 用法

19.3.1 设定选项

这些选项作为字符串通过传递到下拉列表。选项之间应用分隔。例如：。该字符串将保存在下拉列表中，因此也可以保存在本地变量中。

```
lv_dropdown_set_options(dropdown, options)\n"First\nSecond\nThird"
```

该函数插入一个新选项以建立索引。`lv_dropdown_add_option(dropdown, "New option", pos)pos`

为了节省内存，这些选项也可以使用静态（常量）字符串进行设置。在这种情况下，当下拉列表存在且不能使用时，选项字符串应处于活动状态

`lv_dropdown_set_static_options(dropdown, options)lv_dropdown_add_option`

您可以使用手动选择一个选项，其中 *id* 是选项的索引。`lv_dropdown_set_selected(dropdown, id)`

19.3.2 获取选择的选项

使用获取当前选择的选项 `lv_dropdown_get_selected(dropdown)`。它将返回所选选项的*索引*。

`lv_dropdown_get_selected_str(dropdown, buf, buf_size)`将所选选项的名称复制到 `buf`。

19.3.3 方向

该列表可以在任何一侧创建。默认值 `LV_DROPDOWN_DOWN` 可以通过功能进行修改。

`lv_dropdown_set_dir(dropdown, LV_DROPDOWN_DIR_LEFT/RIGHT/UP/DOWN)`

如果列表垂直于屏幕之外，它将与边缘对齐。

19.3.4 sign

可以使用以下命令将 sign（通常是箭头）添加到下拉列表中：

`lv_dropdown_set_symbol(dropdown, LV_SYMBOL_...)`

如果下拉列表的方向是 `LV_DROPDOWN_DIR_LEFT`，sign 将显示在左侧，否则显示在右侧。

19.3.5 最大高度

下拉列表的最大高度可以通过设置。默认情况下，它设置为 3/4 垂直分辨率。

`lv_dropdown_set_max_height(dropdown, height)`

19.3.6 显示所选

主要部分可以显示所选选项或静态文本。可以用来控制。

```
lv_dropdown_set_show_selected(sropdown, true/false)
```

可以使用设置静态文本。仅保存文本指针。`lv_dropdown_set_text(dropdown, "Text")`

19.3.7 动画时间

下拉列表的打开/关闭动画时间由调整。动画时间为零表示没有动画。

```
lv_dropdown_set_anim_time(ddlist, anim_time)
```

19.3.8 手动打开/关闭

要手动打开或关闭下拉列表，可以使用该功能。`lv_dropdown_open/close(dropdown, LV_ANIM_ON/OFF)`

19.4 事件

除通用事件外，下拉列表还发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED**-选择新选项时发送。

19.5 按键

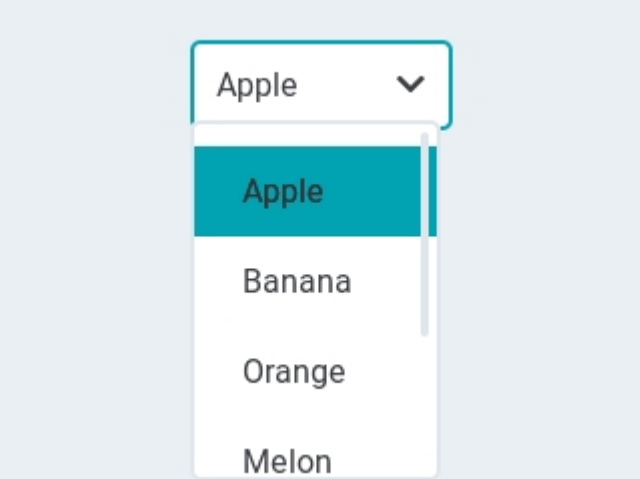
以下*按键*由按钮处理：

- **LV_KEY_RIGHT / DOWN**-选择下一个选项。
- **LV_KEY_LEFT / UP**-选择上一个选项。
- **LY_KEY_ENTER**-应用选定的选项（发送 `LV_EVENT_VALUE_CHANGED` 事件并关闭下拉列表）。

例

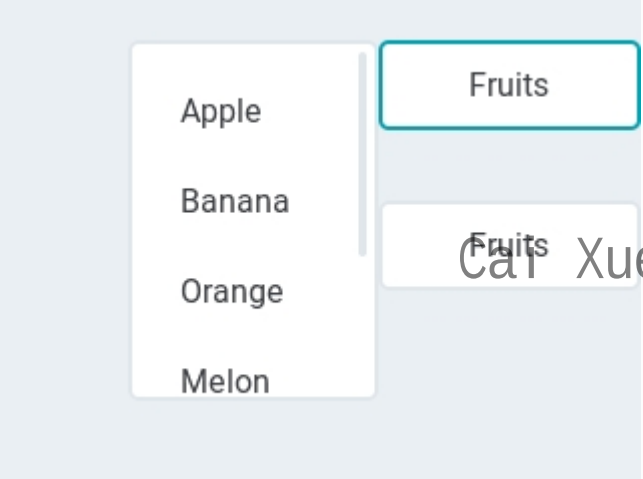
C

简单的下拉列表



码

删除“上”列表



API

typedef

typedef uint8_t `lv_dropdown_dir_t`

typedef uint8_t `lv_dropdown_part_t`

枚举

Enum [anonymous]

值:

enumerator `LV_DROPDOWN_DIR_DOWN`

enumerator `LV_DROPDOWN_DIR_UP`

enumerator `LV_DROPDOWN_DIR_LEFT`

enumerator `LV_DROPDOWN_DIR_RIGHT`

enum [anonymous]

值：

enumerator LV_DROPDOWN_PART_MAIN= LV_OBJ_PART_MAIN

enumerator LV_DROPDOWN_PART_LIST= _LV_OBJ_PART_REAL_LAST

enumerator LV_DROPDOWN_PART_SCROLLBAR

enumerator LV_DROPDOWN_PART_SELECTED

功能

lv_obj_t * lv_dropdown_create（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建一个下拉列表对象

返回

指向创建的下拉列表的指针

参数

- `par`：指向对象的指针，它将是新下拉列表的父对象
- `copy`：指向下拉列表对象的指针，如果不为 NULL，则将从其复制新对象

void lv_dropdown_set_text（lv_obj_t * *ddlist*，*const* char* *txt* ）

设置 ddlist 的文本（如果显示在按钮上）`show_selected = false`

参数

- `ddlist`：指向下拉列表对象的指针
- `txt`：将文本作为字符串（仅保存其指针）

void lv_dropdown_clear_options（lv_obj_t * *ddlist* ）

清除下拉列表中的所有选项。静态或动态。

参数

- `ddlist`：下拉列表对象的指针

void lv_dropdown_set_options（lv_obj_t * *ddlist*，*const* char * *options*）

在字符串的下拉列表中设置选项

参数

- `ddlist`：下拉列表对象的指针
- `options`：带有'

'分开的选项。例如“ One \ nTwo \ nThree”可以在调用此函数后销毁选项字符串

void lv_dropdown_set_options_static（lv_obj_t * ddlist，const char * options）

在字符串的下拉列表中设置选项

参数

- `ddlist`：下拉列表对象的指针
- `options`：带有'的静态字符串

'分开的选项。例如“一个\ n 两个\ n 三个”

void lv_dropdown_add_option（lv_obj_t * ddlist，const char * options，uint32_t pos ）

将选项从字符串添加到下拉列表中。仅适用于动态选项。

参数

- `ddlist`：下拉列表对象的指针
- `option`：没有'

'。例如“四个”

- `pos`：插入位置，从 0 开始索引，LV_DROPDOWN_POS_LAST =字符串的结尾

void lv_dropdown_set_selected（lv_obj_t * ddlist，uint16_t sel_opt ）

设置所选选项

参数

- `ddlist`：下拉列表对象的指针
- `sel_opt`：所选选项的编号（0 ...选项编号-1）；

void lv_dropdown_set_dir（lv_obj_t * ddlist，lv_dropdown_dir_t dir ）

设置下拉列表的方向

参数

- `ddlist`：指向下拉列表对象的指针
- `dir`：LV_DROPDOWN_DIR_LEF / RIGHT / TOP / BOTTOM

void lv_dropdown_set_max_height（lv_obj_t * ddlist，lv_coord_t h ）

设置下拉列表的最大高度

参数

- `ddlist`：指向下拉列表的指针
- `h`：最大高度

void lv_dropdown_set_symbol（lv_obj_t * ddlist，const char* sign）

设置箭头或其他 sign，以在关闭下拉列表时显示。

参数

- `ddlist`：下拉列表对象的指针
- `symbol`：类似 `LV_SYMBOL_DOWN` 或 NULL 的文字不会绘制图标

void lv_dropdown_set_show_selected（lv_obj_t * ddlist，bool show ）

设置 ddlist 是否突出显示最后选择的选项并显示其文本

参数

- `ddlist`：指向下拉列表对象的指针
- `show`：真 false

const 字符* lv_dropdown_get_text（lv_obj_t * ddlist ）

获取 ddlist 的文本（如果显示在按钮上）`show_selected = false`

返回

文字字串

参数

- `ddlist`：指向下拉列表对象的指针

const 字符* lv_dropdown_get_options（const lv_obj_t * ddlist ）

获取下拉列表的选项

返回

以'分隔的选项

'-s（例如“ Option1 \ nOption2 \ nOption3”）

参数

- `ddlist`：下拉列表对象的指针

uint16_t lv_dropdown_get_selected（const lv_obj_t * ddlist ）

获取选择的选项

返回

所选选项的 ID（0 ...选项编号-1）；

参数

- `ddlist`：下拉列表对象的指针

uint16_t lv_dropdown_get_option_cnt（const lv_obj_t * ddlist ）

获取选项总数

返回

列表中的选项总数

参数

- `ddlist`：下拉列表对象的指针

void lv_dropdown_get_selected_str（const lv_obj_t * ddlist，char * buf，uint32_t buf_size ）

获取当前选择的选项作为字符串

参数

- `ddlist`：指向 ddlist 对象的指针
- `buf`：指向存储字符串的数组的指针
- `buf_size`：`buf` 以字节为单位的大小。0：忽略它。

lv_coord_t lv_dropdown_get_max_height（const lv_obj_t * ddlist ）

获取固定高度值。

返回

打开 ddlist 时的高度（0：自动调整大小）

参数

- `ddlist`：指向下拉列表对象的指针

const 字符* lv_dropdown_get_symbol（lv_obj_t * ddlist ）

关闭下拉列表时获取绘制 sign

返回

sign 或 NULL（如果未启用）

参数

- `ddlist`：下拉列表对象的指针

lv_dropdown_dir_t lv_dropdown_get_dir（const lv_obj_t * ddlist ）

关闭下拉列表时获取绘制 sign

返回

sign 或 NULL（如果未启用）

参数

- `ddlist`：下拉列表对象的指针

bool lv_dropdown_get_show_selected（lv_obj_t * ddlist ）

获取 ddlist 是否突出显示最后选择的选项并显示其文本

返回

真 false

参数

- `ddlist`：指向下拉列表对象的指针

void lv_dropdown_open（lv_obj_t * ddlist ）

打开带有或不带有动画的下拉列表

参数

- `ddlist`：下拉列表对象的指针

void lv_dropdown_close（lv_obj_t * ddlist ）

关闭（折叠）下拉列表

参数

- `ddlist`：下拉列表对象的指针
- `anim_en`：LV_ANIM_ON：使用动画；LV_ANOM_OFF：不使用动画

struct lv_dropdown_ext_t

公众成员

```
lv_obj_t *page
const char*text
const char*symbol
char*options
lv_style_list_t style_selected
```

```
    lv_style_list_t style_page
    lv_style_list_t style_scrlbar
    lv_coord_t max_height
    uint16_t option_cnt
    uint16_t sel_opt_id
    uint16_t sel_opt_id_orig
    uint16_t pr_opt_id
    lv_dropdown_dir_t dir
    uint8_t show_selected
    uint8_t static_txt
```

第二十章 量规（lv_gauge）

20.1 总览

量规是一种带有刻度标签和一根或多根针的仪表。

20.2 小部件和样式

量规的主要部分称为 `LV_GAUGE_PART_MAIN`。它使用典型的背景样式属性绘制背景，并使用*线*和*比例*样式属性绘制“较小”比例线。它还使用 *text* 属性设置比例标签的样式。*pad_inner* 用于设置刻度线和刻度标签之间的空间。

`LV_GAUGE_PART_MAJOR` 是一个虚拟小部件，它使用 *line* 和 *scale* 样式属性描述了主要的比例尺线（添加了标签）。

`LV_GAUGE_PART_NEEDLE` 也是虚拟小部件，它通过*线型*属性来描述针。的*大小*和典型的背景属性用于描述在所述针（多个）的枢转点的矩形（或圆形）。 *pad_inner* 用于使针比刻度线的外半径小。

20.3 用法

20.3.1 设定值和针

量规可以显示多于一根针。使用该功能设置针数和每根针具有颜色的阵列。数组必须是静态或全局变量，因为仅存储其指针。`lv_gauge_set_needle_count(gauge, needle_num, color_array)`

您可以用来设置针的值。`lv_gauge_set_value(gauge, needle_id, value)`

20.3.2 规模

您可以使用该功能调整刻度角度以及刻度线和标签的数量。默认设置为 220 度，6 个比例标签和 21 条线。`lv_gauge_set_scale(gauge, angle, line_num, label_cnt)`

量规的刻度可以偏移。可以用调节。`lv_gauge_set_angle_offset(gauge, angle)`

20.3.3 范围

量规的范围可以通过指定。默认范围是 0..100。`lv_gauge_set_range(gauge, min, max)`

20.3.4 针图

图像也可以用作针。图像应指向右侧（如`==>`）。要设置图像，请使用。和旋转中心从左上角偏移。图像将以（样式属性）强度重新着色为针的颜色。

`lv_gauge_set_needle_img(gauge1, &img, pivot_x, pivot_y)pivot_xpivot_yimage_recolor_opa`

20.3.5 临界值

要设置临界值，请使用。此值之后，比例尺颜色将更改为 *scale_end_color*。默认临界值为 80。`lv_gauge_set_critical_value(gauge, value)`

20.4 事件

仅通用事件是按对象类型发送的。Cai Xuefeng

20.5 按键

对象类型不处理任何 *键*。

例

C

简易量规

API

typedef

typedef void（* lv_gauge_format_cb_t）（lv_obj_t *表号，char * buf，int bufsize，int32_t 值）

typedef uint8_t lv_gauge_style_t

枚举

Enum [anonymous]

值：

> *enumerator* LV_GAUGE_PART_MAIN= LV_LINEMETER_PART_MAIN
>
> *enumerator* LV_GAUGE_PART_MAJOR= _LV_LINEMETER_PART_VIRTUAL_LAST
>
> *enumerator* LV_GAUGE_PART_NEEDLE
>
> *enumerator* _LV_GAUGE_PART_VIRTUAL_LAST= _LV_LINEMETER_PART_VIRTUAL_LAST
>
> *enumerator* _LV_GAUGE_PART_REAL_LAST= _LV_LINEMETER_PART_REAL_LAST

功能

lv_obj_t * lv_gauge_create（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建仪表对象

> 返回
>
> 指向创建的量规的指针
>
> > 参数
> >
> > - `par`：指向对象的指针，它将是新量规的父对象
> > - `copy`：指向量规对象的指针，如果不为 NULL，则将从其复制新对象

void lv_gauge_set_needle_count（lv_obj_t * *量规*，*uint8_t needle_cnt*，*const*lv_color_t *colors* [ ] ）

设定针数

参数

- `gauge`：量规对象的指针
- `needle_cnt`：新针数
- `colors`：针的颜色数组（带有'num'元素）

---

void lv_gauge_set_value（lv_obj_t * *gauge*，uint8_t *needle_id*，int32_t * *value*）

设定指针的值

参数

- `gauge`：指向量规的指针
- `needle_id`：指针的 id
- `value`：新值

---

void lv_gauge_set_range（lv_obj_t * *gauge*，int32_t *min*，int32_t *max* ）

设置量规的最小值和最大值

参数

- `gauge`：指向量规对象的指针
- `min`：最小值
- `max`：最大值

---

void lv_gauge_set_critical_value（lv_obj_t * *gauge*，int32_t *value*）

在秤上设置一个临界值。此值之后，将绘制“ line.color”比例线

参数

- `gauge`：指向量规对象的指针
- `value`：临界值

---

void lv_gauge_set_scale（lv_obj_t * *gauge*，uint16_t *angle*，uint8_t *line_cnt*，uint8_t *label_cnt* ）

设置量规的刻度设置

参数

- `gauge`：指向量规对象的指针

- `angle`：比例尺角度（0..360）
- `line_cnt`：刻度线数。要在标签之间获得给定的“细分”行：

```
line_cnt = (sub_div + 1) * (label_cnt - 1) + 1
```

- `label_cnt`：刻度标签计数。

void lv_gauge_set_angle_offset（lv_obj_t * gauge，uint16_t angle）

为旋转量规的角度设置一个偏移量。

参数

- `gauge`：指向线表对象的指针
- `angle`：角度偏移（0..360），顺时针旋转

void lv_gauge_set_needle_img（lv_obj_t * gauge，const void* img，lv_coord_t pivot_x，lv_coord_t pivot_y）

设置图像以显示为针。针图像应水平并且指向右侧（`--->`）。

参数

- `gauge`：指向量规对象的指针
- `img_src`：指向 `lv_img_dsc_t` 变量或图像 path 的指针（不是 `lv_img` 对象）
- `pivot_x`：图像旋转中心的 X 坐标
- `pivot_y`：图像旋转中心的 Y 坐标

空隙 lv_gauge_set_formatter_cb（lv_obj_t * gauge，lv_gauge_format_cb_t format_cb）

分配一个功能以格式化量规值

参数

- `gauge`：指向量规对象的指针
- `format_cb`：指向 lv_gauge_format_cb_t 函数的指针

int32_t lv_gauge_get_value（const lv_obj_t * gauge，uint8_t needle）

获得针的价值

返回

针的值[min，max]

参数

- `gauge`：量规对象的指针
- `needle`：指针的 id

`uint8_t lv_gauge_get_needle_count（const lv_obj_t *  gauge）`

获取量规上的指针数

返回

指针数

参数

- `gauge`：量规的指针

`int32_t lv_gauge_get_min_value（const lv_obj_t * lmeter ）`

获取量规的最小值

返回

量规的最小值

参数

- `gauge`：指向量规对象的指针

`int32_t lv_gauge_get_max_value（const lv_obj_t * lmeter ）`

获取量规的最大值

返回

量规的最大值

参数

- `gauge`：指向量规对象的指针

`int32_t lv_gauge_get_critical_value（const lv_obj_t *  gauge）`

获得规模的临界值。

返回

临界值

参数

- `gauge`：指向量规对象的指针

`uint8_t lv_gauge_get_label_count（const lv_obj_t *  gauge）`

设置标签数（也可以设置粗线）

返回

标签数量

参数

- `gauge`：指向量规对象的指针

uint16_t lv_gauge_get_line_count（const lv_obj_t * gauge）

获取量规的刻度号

返回

刻度单位数

参数

- `gauge`：指向量规对象的指针

uint16_t lv_gauge_get_scale_angle（const lv_obj_t * gauge）

获取量规的刻度角

返回

刻度角

参数

- `gauge`：指向量规对象的指针

uint16_t lv_gauge_get_angle_offset（lv_obj_t * gauge）

获取量规的偏移量。

返回

角度偏移（0..360）

参数

- `gauge`：指向量规对象的指针

const void* lv_gauge_get_needle_img（lv_obj_t * gauge）

获取图像以显示为针。

返回

指向 `lv_img_dsc_t` 变量或图像（不是 `lv_img` 对象）path 的指针。 `NULL` 如果不使用。

参数

- `gauge`：指向量规对象的指针

lv_coord_t lv_gauge_get_needle_img_pivot_x（lv_obj_t * gauge）

获取针图像旋转中心的 X 坐标

返回

图像旋转中心的 X 坐标

参数

- `gauge`：指向量规对象的指针

lv_coord_t lv_gauge_get_needle_img_pivot_y（lv_obj_t * gauge）

获取针图像旋转中心的 Y 坐标

返回

图像旋转中心的 X 坐标

参数

- `gauge`：指向量规对象的指针

struct lv_gauge_ext_t

公众成员

```
lv_linemeter_ext_t lmeter
int32_t *values
const lv_color_t *needle_colors
const void *needle_img
lv_point_t needle_img_pivot
lv_style_list_t style_needle
lv_style_list_t style_strong
uint8_t needle_count
uint8_t label_count
lv_gauge_format_cb_t format_cb
```

第二十一章 图片（lv_img）

21.1 总览

*图像*是从 Flash（作为数组）或从外部作为文件显示的基本对象。图像也可以显示 sign（`LV_SYMBOL_...`）。

使用图像 decoder 界面，也可以支持自定义图像格式。

21.2 小部件和样式

图像只有一个主要部分 `LV_IMG_PART_MAIN`，它使用典型的背景样式属性绘制背景矩形和*图像*属性。填充值用于使背景实际变大。（它不会更改图像的实际大小，但仅在绘图期间应用大小修改）

21.3 用法

21.3.1 图片来源

为了提供最大的灵活性，图像的来源可以是：

- 代码中的变量（带有像素的 C 数组）。
- 外部存储的文件（例如 SD 卡上的文件）。
- 带 sign 的文字。

要设置图像的来源，请使用。`lv_img_set_src(img, src)`

要从 PNG，JPG 或 BMP 图像生成**像素阵列**，请使用在线图像转换器工具并使用其指针设置转换后的图像： 要使变量在 C 文件中可见，您需要使用进行声明。

```
lv_img_set_src(img1, &converted_img_var);LV_IMG_DECLARE(converted_img_var)
```

要使用**外部文件**，您还需要使用在线转换器工具转换图像文件，但是现在您应该选择二进制输出格式。您还需要使用 LVGL 的文件系统模块，并为基本文件操作注册具有某些功能的驱动程序。进入文件系统以了解更多信息。要设置来自文件的图像，请使用。

```
lv_img_set_src(img, "S:folder1/my_img.bin")
```

您可以类似于 Labels 来设置 **sign**。在这种情况下，图像将根据样式中指定的*字体*呈现为文本。它可以使用轻量级的单色“字母”代替实际图像。您可以将 sign 设置为。

```
lv_img_set_src(img1, LV_SYMBOL_OK)
```

21.3.2 标签为图片

图像和标签有时用于传达相同的内容。例如，描述按钮的作用。因此，图像和标签可以互换。为了处理这些图像，甚至可以通过将其 `LV_SYMBOL_DUMMY` 用作文本前缀来显示文本。例如，。

```
lv_img_set_src(img, LV_SYMBOL_DUMMY "Some text")
```

21.3.3 透明度

内部（可变）和外部图像支持 2 种透明度处理方法：

- **Chrome 抠像** -具有 `LV_COLOR_TRANSP`（*lv_conf.h*）颜色的像素将是透明的。
- **Alpha 字节** -Alpha 字节添加到每个像素。

21.3.4 调色板和 Alpha 指数

除了*本色*（RGB）颜色格式外，还支持以下格式：

- 已建立**索引** -图像具有调色板。
- **Alpha 索引** -仅存储 Alpha 值。

可以在字体转换器中选择这些选项。要了解有关颜色格式的更多信息，请阅读图像部分。

23.3.5 重新着色

根据像素的亮度，可以在运行时将图像重新着色为任何颜色。在不存储同一图像的更多版本的情况下，显示图像的不同状态（选中，void，按下等）非常有用。可以通过 `img.intense` 在 `LV_OPA_TRANSP`（不重新着色，值：0）和 `LV_OPA_COVER`（完全重新着色，值：255）之间设置样式来启用此功能。默认值为 `LV_OPA_TRANSP` 禁用此功能。

21.3.6 自动尺寸

如果该功能启用，则可以自动将图像对象的大小设置为图像源的宽度和高度。如果启用了*自动调整大小*，则在设置新文件时，对象大小将自动更改。以后，您可以手动修改大小。该*自动大小*默认情况下，如果图像是不是一个屏幕中启用。`lv_img_set_auto_size(image, true)`

21.3.7 镶嵌

如果对象大小在任何方向上都大于图像大小，则图像将像马赛克一样重复。仅从非常狭窄的源中创建大图像是一项非常有用的功能。例如，您可以制作具有特殊渐变的 *300 x 1* 图像，然后使用镶嵌功能将其设置为墙纸。

21.3.8 抵消

使用和，您可以向显示的图像添加一些偏移。如果对象尺寸小于图像源尺寸，则很有用。使用 offset 参数，可以通过对 x 或 y 偏移量进行动画处理来创建纹理图集或“运行中的图像”效果。

```
lv_img_set_offset_x(img, x_ofs)lv_img_set_offset_y(img, y_ofs)
```

21.3.9 转变

使用图像将被缩放。设置为或禁用缩放。较大的值将放大图像（例如，两倍大小），较小的值将缩小图像（例如，一半大小）。分数尺度也适用。例如扩大 10%。

```
lv_img_set_zoom(img, factor)factor256LV_IMG_ZOOM_NONE512128281
```

要旋转图像，请使用。角度精度为 0.1 度，因此对于 45.8°设置 458。

```
lv_img_set_angle(img, angle)
```

默认情况下，旋转的枢轴点是图像的中心。可以用更改。是左上角。

```
lv_img_set_pivot(img, pivot_x, pivot_y)0;0
```

可以使用调整转换的质量。启用抗锯齿功能后，转换的质量更高，但速度较慢。

```
lv_img_set_antialias(img, true/false)
```

转换需要整个图像可用。因此 `LV_IMG_CF_INDEXED_...`， `LV_IMG_CF_ALPHA_...`可以转换索引图像（），仅 Alpha 图像（）或文件中的图像。换句话说，转换仅适用于以 C 数组存储的真彩色图像，或者自定义图像 decoder 返回整个图像。

注意，图像对象的真实坐标在变换期间不会改变。那 `lv_obj_get_width/height/x/y()` 将返回原始的非缩放坐标。

21.3.10 旋转

图像可以旋转

21.4 事件

仅通用事件是按对象类型发送的。

21.5 按键

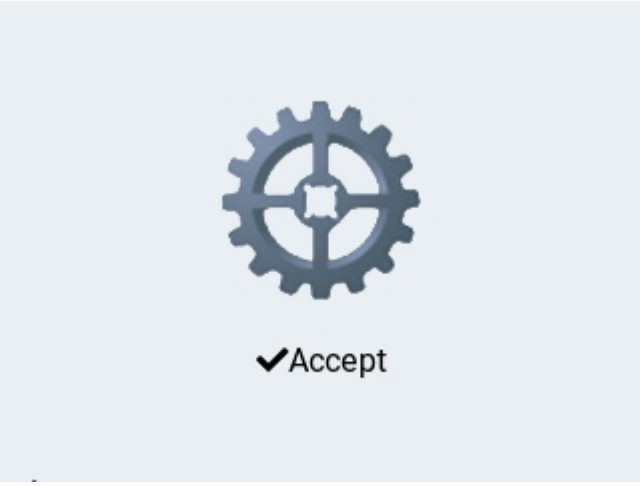
对象类型不处理任何*键*。

例

C

Cai Xuefeng

图片来自变量和 sign



码

图像重新着色

API

typedef

typedef uint8_t **lv_img_part_t**

枚举

Enum [anonymous]

值：

enumerator **LV_IMG_PART_MAIN**

功能

lv_obj_t * **lv_img_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个图像对象

返回

指向创建的图像的指针

参数

- `par`：指向对象的指针，它将是新按钮的父对象
- `copy`：指向图像对象的指针，如果不为 NULL，则将从其复制新对象

voidlv_img_set_src（lv_obj_t * *img*，*const*void* *src_img* ）

设置像素图以按图像显示

参数

- `img`：指向图像对象的指针
- `data`：图像数据

void lv_img_set_auto_size（lv_obj_t * *IMG*，bool *autosize_en* ）

启用自动尺寸功能。如果启用，则对象大小将与图片大小相同。

参数

- `img`：指向图像的指针
- `en`：true：启用自动调整大小，false：禁用自动调整大小

void lv_img_set_offset_x（lv_obj_t * *img*，lv_coord_t *x* ）

设置图像源的偏移量。因此将以新的原点显示图像。

参数

- `img`：指向图像的指针
- `x`：沿 x 轴的新偏移量。

void lv_img_set_offset_y（lv_obj_t * *img*，lv_coord_t *y* ）

设置图像源的偏移量。因此将以新的原点显示图像。

参数

- `img`：指向图像的指针
- `y`：沿 y 轴的新偏移量。

void lv_img_set_pivot（lv_obj_t * *img*，lv_coord_t *pivot_x* ，lv_coord_t *pivot_y* ）

设置图像的旋转中心。图像将围绕此点旋转

参数

- `img`：指向图像对象的指针
- `pivot_x`：图像的旋转中心 x
- `pivot_y`：图像的旋转中心 y

void lv_img_set_angle（lv_obj_t * *img*，int16_t *angle* ）

设置图像的旋转角度。图像将围绕由 `lv_img_set_pivot()`

参数

- `img`：指向图像对象的指针
- `angle`：旋转角度，以度为单位，分辨率为 0.1 度（0..3600：顺时针）

void lv_img_set_zoom（lv_obj_t * img，uint16_t zoom ）

设置图像的变焦倍数。

参数

- `img`：指向图像对象的指针
- `zoom`：缩放系数。
  - 256 或 LV_ZOOM_IMG_NONE 无变焦
  - <256：按比例缩小
  - > 256 放大
  - 一半大小
  - 512 倍

void lv_img_set_antialias（lv_obj_t * img，bool antialias）

启用/禁用转换的抗锯齿功能（旋转，缩放）

参数

- `img`：指向图像对象的指针
- `antialias`：true：抗锯齿；false：不抗锯齿

const void* lv_img_get_src（lv_obj_t * img ）

获取图像的来源

返回

图像源（sign，文件名或 C 数组）

参数

- `img`：指向图像对象的指针

const 字符* lv_img_get_file_name（const lv_obj_t * img ）

获取图像文件集的名称

返回

文档名称

参数

- `img`：指向图像的指针

bool lv_img_get_auto_size（const lv_obj_t * img ）

获取自动尺寸启用属性

返回

true：启用自动调整大小，false：禁用自动调整大小

参数

- `img`：指向图像的指针

lv_coord_t lv_img_get_offset_x（lv_obj_t * *img* ）

获取 img 对象的 offset.x 属性。

返回

offset.x 值。

参数

- `img`：指向图像的指针

lv_coord_t lv_img_get_offset_y（lv_obj_t * *img* ）

获取 img 对象的 offset.y 属性。

返回

offset.y 值。

参数

- `img`：指向图像的指针

uint16_t lv_img_get_angle（lv_obj_t * *img* ）

获取图像的旋转角度。

返回

旋转角度（度）（0..359）

参数

- `img`：指向图像对象的指针

voidlv_img_get_pivot（lv_obj_t * *img*，lv_point_t * *center* ）

获取图像的旋转中心。

参数

- `img`：指向图像对象的指针
- `center`：图像的旋转中心

uint16_t lv_img_get_zoom（lv_obj_t * *img* ）

获取图像的缩放系数。

返回

缩放系数（256：无缩放）

参数

- `img`：指向图像对象的指针

bool lv_img_get_antialias（lv_obj_t * *img* ）

获取转换（旋转，缩放）是否抗锯齿

返回

true：抗锯齿；false：不抗锯齿

参数

- `img`：指向图像对象的指针

struct lv_img_ext_t

公众成员

const void* **src**
lv_point_t **offset**
lv_coord_t **w**
lv_coord_t **h**
uint16_t **angle**
lv_point_t **pivot**
uint16_t **zoom**
uint8_t **src_type**
uint8_t **auto_size**
uint8_t **cf**
uint8_t **antialias**

第二十二章 图片按钮（lv_imgbtn）

22.1 总览

图像按钮与简单的“按钮”对象非常相似。唯一的区别是，它在每种状态下显示用户定义的图像，而不是绘制矩形。在阅读本节之前，请阅读“ 按钮”一节以更好地理解。

22.2 小部件和样式

“图像”按钮对象只有一个主要部分，`LV_IMG_BTN_PART_MAIN`从那里可以使用所有*图像*样式属性。可以使用 *image_recolor* 和 *image_recolor_opa* 属性在每种状态下为图像重新着色。例如，如果按下该按钮可使图像变暗。

22.3 用法

22.3.1 图片来源

要将图像设置为某种状态，请使用。除了“图像”按钮不支持“sign”外，图像源的工作方式与“ 图像”对象中所述的相同。`lv_imgbtn_set_src(imgbtn, LV_BTN_STATE_..., &img_src)`

如果`LV_IMGBTN_TILED`在 *lv_conf.h* 中启用，则变为可用。使用平铺功能，将重复*中间*图像以填充对象的宽度。因此，您可以使用设置图像按钮的宽度。但是，如果没有此选项，则宽度将始终与图像源的宽度相同。

```
lv_imgbtn_set_src_tiled(imgbtn, LV_BTN_STATE_..., &img_src_left, &img_src_mid, &img_src_right)LV_IMGBTN_T
```

```
ILEDlv_obj_set_width()
```

22.3.2 按钮功能

同样正常的按钮，和也的作品。

```
lv_imgbtn_set_checkable(imgbtn, true/false)lv_imgbtn_toggle(imgbtn)lv_imgbtn_set_state(imgbtn, LV_BTN_STA
```

```
TE_...)
```

22.4 事件

除了一般事件外，按钮还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED-**切换按钮时发送。

请注意，与通用输入设备相关的事件（如 `LV_EVENT_PRESSED` ）也以非活动状态发送。您需要检查状态 `lv_btn_get_state(btn)` 以忽略非活动按钮中的事件。

了解有关事件的更多信息。

22.5 按键

以下*按键*由按钮处理：

- **LV_KEY_RIGHT / UP-**如果启用了切换，则进入切换状态。
- **LV_KEY_LEFT / DOWN-**如果启用了切换，则进入非切换状态。

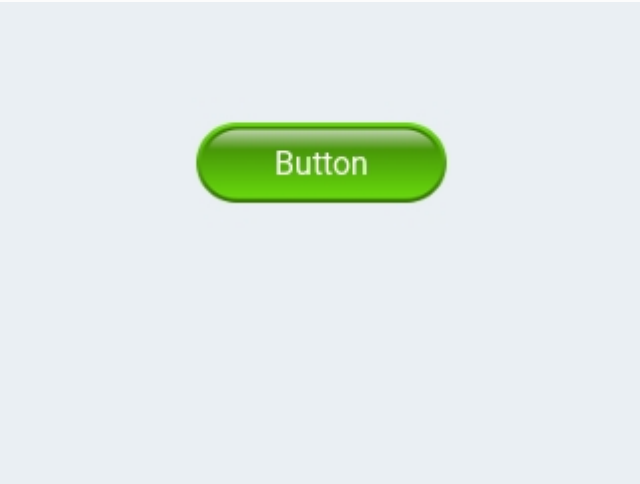
请注意，照常将的状态 `LV_KEY_ENTER` 转换为 `LV_EVENT_PRESSED/PRESSING/RELEASED` etc。

了解更多有关按键的信息。

例 Cai Xuefeng

C

简单图像按钮



API

typedef

typedef uint8_t lv_imgbtn_part_t

枚举

Enum [anonymous]

值：

enumerator LV_IMGBTN_PART_MAIN= LV_BTN_PART_MAIN

功能

lv_obj_t * lv_imgbtn_create（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建图像按钮对象

返回

指向创建的图像按钮的指针

参数

- `par`：指向对象的指针，它将是新图像按钮的父对象
- `copy`：指向图像按钮对象的指针，如果不为 NULL，则将从其复制新对象

voidlv_imgbtn_set_src（lv_obj_t * *imgbtn*，lv_btn_state_t *state*，*const*void* *src* ）

为图像按钮的状态设置图像

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：针对哪个状态设置了新图像（来自 `lv_btn_state_t`）
- `src`：指向图像源的指针（C 数组或文件 path）

voidlv_imgbtn_set_src_tiled（lv_obj_t * *imgbtn*，lv_btn_state_t *state*，*const*void* *src_left*，*const*void* *src_mid*，*const*void* *src_right* ）

为图像按钮的状态设置图像

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：针对哪个状态设置了新图像（来自 `lv_btn_state_t`）
- `src_left`：指向按钮左侧图像源的指针（C 数组或文件 path）

- `src_mid`：指向按钮中间的图像源的指针（理想情况下为 1px 宽）（C 数组或文件 path）
- `src_right`：指向按钮右侧图像源的指针（C 数组或文件 path）

void lv_imgbtn_set_checkable（lv_obj_t * imgbtn，bool tgl ）

启用切换状态。释放时，按钮将从/切换到切换状态。

参数

- `imgbtn`：指向图像按钮对象的指针
- `tgl`：true：启用切换状态，false：禁用

void lv_imgbtn_set_state（lv_obj_t * imgbtn，lv_btn_state_t state）

设置图像按钮的状态

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：按钮的新状态（来自 lv_btn_state_t 枚举）

void lv_imgbtn_toggle（lv_obj_t * imgbtn）

切换图像按钮的状态（ON-> OFF，OFF-> ON）

参数

- `imgbtn`：指向图像按钮对象的指针

const void* lv_imgbtn_get_src（lv_obj_t * imgbtn，lv_btn_state_t state）

获取给定状态的图像

返回

指向图像源的指针（C 数组或文件 path）

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：从中获取图像的状态 `lv_btn_state_t`

const void* lv_imgbtn_get_src_left（lv_obj_t * imgbtn，lv_btn_state_t state）

获取给定状态下的左侧图像

返回

指向左侧图像源的指针（C 数组或文件 path）

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：从中获取图像的状态 `lv_btn_state_t`

const void* lv_imgbtn_get_src_middle（lv_obj_t * imgbtn，lv_btn_state_t state）

获取给定状态的中间图像

返回

指向中间图像源的指针（C 数组或文件 path）

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：从中获取图像的状态 `lv_btn_state_t`

const void* lv_imgbtn_get_src_right（lv_obj_t * imgbtn，lv_btn_state_t state）

在给定状态下获取正确的图像

返回

指向左侧图像源的指针（C 数组或文件 path）

参数

- `imgbtn`：指向图像按钮对象的指针
- `state`：从中获取图像的状态 `lv_btn_state_t`

lv_btn_state_t lv_imgbtn_get_state（const lv_obj_t * imgbtn ）

获取图像按钮的当前状态

返回

按钮的状态（来自 lv_btn_state_t 枚举）

参数

- `imgbtn`：指向图像按钮对象的指针

bool lv_imgbtn_get_checkable（const lv_obj_t * imgbtn ）

获取图像按钮的切换启用属性

返回

true：启用切换，false：禁用

参数

- `imgbtn`：指向图像按钮对象的指针

struct **lv_imgbtn_ext_t**

公众成员

lv_btn_ext_t **btn**

*const*void* **img_src_mid**[_LV_BTN_STATE_LAST]

*const*void* **img_src_left**[_LV_BTN_STATE_LAST]

*const*void* **img_src_right**[_LV_BTN_STATE_LAST]

lv_img_cf_t **act_cf**

uint8_t **tiled**

第二十三章 键盘（lv_keyboard）

23.1 总览

Keyboard 对象是一个特殊的 Button 矩阵，具有预定义的按键映射和其他功能，以实现虚拟键盘来编写文本。

23.2 小部件和样式

类似于按钮 maticals，键盘包括 2 部分：

- `LV_KEYBOARD_PART_BG` 这是主要部分，并使用了所有典型的背景属性
- `LV_KEYBOARD_PART_BTN` 这是按钮的虚拟部分。它还使用所有典型的背景属性和文本属性。

23.3 用法

23.3.1 模式

键盘具有以下模式：

- **LV_KEYBOARD_MODE_TEXT_LOWER-**显示小写字母
- **LV_KEYBOARD_MODE_TEXT_UPPER-**显示大写字母
- **LV_KEYBOARD_MODE_TEXT_SPECIAL-**显示特殊字符
- **LV_KEYBOARD_MODE_NUM-**显示数字，+ / -号和小数点。

该 `TEXT` 模式布局包含按钮来改变模式。

要手动设置模式，请使用。默认为更多 。

```
lv_keyboard_set_mode(kb, mode)LV_KEYBOARD_MODE_TEXT_UPPER
```

23.3.2 分配文本区域

您可以为键盘分配一个文本区域，以自动将单击的字符放在此处。要分配文本区域，请使用。

```
lv_keyboard_set_textarea(kb, ta)
```

可以使用键盘**来管理**分配的文本区域的**光标**：分配了键盘后，上一个文本区域的光标将被隐藏，并且将显示新的文本区域。当通过“*确定*”或“*关闭*”按钮*关闭*键盘时，光标也将被隐藏。光标管理器功能由启用。默认为不管理。`lv_keyboard_set_cursor_manage(kb, true)`

23.3.3 新按键图

您可以使用和为键盘指定新的地图（布局）。了解有关 Button 矩阵对象的更多信息。请记住，使用以下关键字将具有与原始地图相同的效果：

`lv_keyboard_set_map(kb, map)lv_keyboard_set_ctrl_map(kb, ctrl_map)`

- *LV_SYMBOL_OK*-申请。
- *LV_SYMBOL_CLOSE*-关闭。
- *LV_SYMBOL_BACKSPACE*-左侧删除。
- *LV_SYMBOL_LEFT*-向左移动光标。
- *LV_SYMBOL_RIGHT*-向右移动光标。
- *“ABC”*-加载大写地图。
- *“abc”*-加载小写字母映射。
- *“输入”*-新行。

23.4 事件

除了通用事件外，键盘还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED**-按下/释放按钮时或在长按后重复时发送。事件数据设置为按下/释放按钮的 ID。
- **LV_EVENT_APPLY** -的*确定*按钮被点击。
- **LV_EVENT_CANCEL** -该*关闭*按钮被点击。

键盘具有一个名为的**默认事件处理程序**回调 `lv_keyboard_def_event_cb`。它处理按钮的按下，地图更改，分配的文本区域等。您可以将其完全替换为自定义事件处理程序，但是，您可以 `lv_keyboard_def_event_cb` 在事件处理程序的开头进行调用以处理与以前相同的操作。

23.5 按键

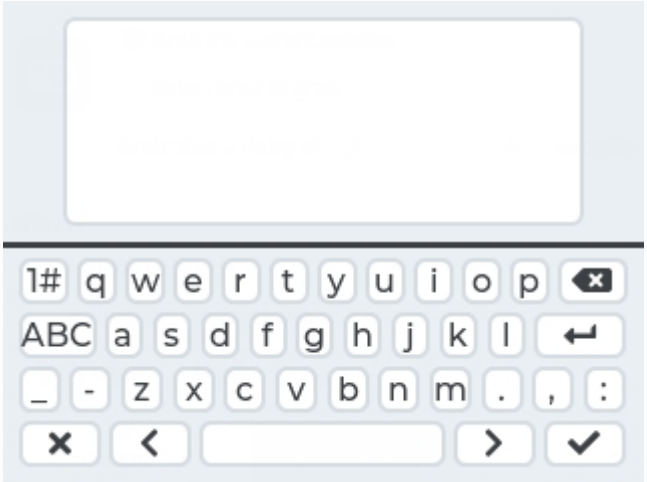
以下*按键*由按钮处理：

- **LV_KEY_RIGHT / UP / LEFT / RIGHT**-在按钮之间导航并选择一个。
- **LV_KEY_ENTER**-按下/释放所选按钮。

例子

C

带文字区域的键盘



API

typedef

typedef uint8_t lv_keyboard_mode_t

typedef uint8_t lv_keyboard_style_t

枚举

Enum [anonymous]

当前键盘模式。

值：

enumerator LV_KEYBOARD_MODE_TEXT_LOWER

enumerator LV_KEYBOARD_MODE_TEXT_UPPER

enumerator LV_KEYBOARD_MODE_SPECIAL

enumerator LV_KEYBOARD_MODE_NUM

enum [anonymous]

值：

enumerator LV_KEYBOARD_PART_BG

enumerator LV_KEYBOARD_PART_BTN

功能

lv_obj_t * lv_keyboard_create（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建键盘对象

返回

指向创建的键盘的指针

参数

- `par`：指向对象的指针，它将是新键盘的父对象
- `copy`：指向键盘对象的指针，如果不为 NULL，则将从其复制新对象

void lv_keyboard_set_textarea（lv_obj_t * kb，lv_obj_t * ta ）

为键盘分配一个文本区域。按下的字符将放在此处。

参数

- `kb`：指向 Keyboard 对象的指针
- `ta`：指向要在此处写入文本区域对象的指针

void lv_keyboard_set_mode（lv_obj_t * kb，lv_keyboard_mode_t mode）

设置新的模式（文本或数字地图）

参数

- `kb`：指向 Keyboard 对象的指针
- `mode`：“ lv_keyboard_mode_t”中的模式

void lv_keyboard_set_cursor_manage（lv_obj_t * kb，bool en ）

自动隐藏或显示当前文本区域的光标

参数

- `kb`：指向 Keyboard 对象的指针
- `en`：true：在当前文本区域显示光标，false：隐藏光标

void lv_keyboard_set_map（lv_obj_t * kb，lv_keyboard_mode_t mode，const char * map [ ] ）

为键盘设置新地图

参数

- `kb`：指向 Keyboard 对象的指针
- `mode`：键盘映射以更改“ lv_keyboard_mode_t”

- `map`：指向用来描述地图的字符串数组的指针。有关更多信息，请参见“ lv_btnmatrix_set_map（）”。

---

void lv_keyboard_set_ctrl_map（lv_obj_t * *kb*，lv_keyboard_mode_t *mode*，const lv_btnmatrix_ctrl_t *ctrl_map* [ ] ）

设置键盘的按钮控制图（隐藏，禁用等）。控制图数组将被复制，因此在此函数返回后可以将其释放。

参数

- `kb`：指向键盘对象的指针
- `mode`：键盘 Ctrl 映射以更改'lv_keyboard_mode_t'
- `ctrl_map`：指向 `lv_btn_ctrl_t` 控制字节数组的指针。请参阅：`lv_btnmatrix_set_ctrl_map` 有关更多详细信息。

---

lv_obj_t * lv_keyboard_get_textarea（const lv_obj_t * *kb* ）

为键盘分配一个文本区域。按下的字符将放在此处。

返回

指向分配的文本区域对象的指针

参数

- `kb`：指向 Keyboard 对象的指针

---

lv_keyboard_mode_t lv_keyboard_get_mode（const lv_obj_t * *kb* ）

设置新的模式（文本或数字地图）

返回

来自“ lv_keyboard_mode_t”的当前模式

参数

- `kb`：指向 Keyboard 对象的指针

---

bool lv_keyboard_get_cursor_manage（const lv_obj_t * *kb* ）

获取当前的光标管理模式。

返回

true：在当前文本区域显示光标，false：隐藏光标

参数

- `kb`：指向 Keyboard 对象的指针

const 字符** lv_keyboard_get_map_array（*const*lv_obj_t * *kb* ）

获取键盘的当前地图

> 返回

当前地图

> 参数

- `kb`：指向键盘对象的指针

void lv_keyboard_def_event_cb（lv_obj_t * *kb*，lv_event_t *event*）

默认键盘事件，用于将字符添加到“文本”区域并更改地图。如果将自定义 `event_cb` 添加到键盘，则会从该键盘调用此函数来处理按钮的单击

> 参数

- `kb`：指向键盘的指针
- `event`：触发事件

struct lv_keyboard_ext_t

公众成员


```
lv_btnmatrix_ext_t btnm
lv_obj_t *ta
lv_keyboard_mode_t mode
uint8_t cursor_mng
```

第二十四章 标签（lv_label）

24.1 总览

标签是用于显示文本的基本对象类型。

24.2 小部件和样式

标签只有一个主要部分，称为 `LV_LABEL_PART_MAIN`。它使用所有典型的背景属性和文本属性。填充值可用于使文本的区域在相关方向上变小。

24.3 用法

24.3.1 设定文字

您可以使用运行时在标签上设置文本。它将动态分配一个缓冲区，并将提供的字符串复制到该缓冲区中。因此，在该函数返回后，您无需将传递的文本保留在范围内。

```
lv_label_set_text(label, "New text")lv_label_set_text
```


带有 **printf 格式**可以用来设置文本。`lv_label_set_text_fmt(label, "Value: %d", 15)`

标签是能够从显示文本**的静态字符缓冲区**是`\0`封端的。为此，请使用。在这种情况下，文本不会存储在动态内存中，而是直接使用给定的缓冲区。这意味着数组不能是在函数退出时超出范围的局部变量。常量字符串可以安全地使用（除非与一起使用，否则它会就地修改缓冲区），因为它们存储在 ROM 存储器中，该存储器始终可以访问。

```
lv_label_set_static_text(label, "Text")lv_label_set_static_textLV_LABEL_LONG_DOT
```

您也可以将**原始数组**用作标签文本。数组不必`\0`终止。在这种情况下，文本将与一样保存到动态存储器中`lv_label_set_text`。要设置原始字符数组，请使用函数。

```
lv_label_set_array_text(label, char_array, size)
```

24.3.2 越线

换行符由标签对象自动处理。您可以用来`\n`换行。例如：`"line1\nline2\n\nline4"`

24.3.3 长模式

默认情况下，标签对象的宽度会自动扩展为文本大小。否则，可以根据几种长模式策略来操纵文本：

- **LV_LABEL_LONG_EXPAND**-将对象大小扩展为文本大小（默认）
- **LV_LABEL_LONG_BREAK**-保持对象的宽度，折断（包装）太长的线并扩大对象的高度
- **LV_LABEL_LONG_DOT**-保持对象大小，打断文本并在最后一行写点（**使用时不支持**`lv_label_set_static_text`）
- **LV_LABEL_LONG_SROLL**-保持大小并来回滚动标签
- **LV_LABEL_LONG_SROLL_CIRC**-保持大小并循环滚动标签
- **LV_LABEL_LONG_CROP**-保持大小并裁剪文本

您可以使用指定长模式`lv_label_set_long_mode(label, LV_LABEL_LONG_...)`

重要的是要注意，当创建标签并设置其文本时，标签的大小已扩展为文本大小。此外，默认情况下`LV_LABEL_LONG_EXPAND`，*长模式*`lv_obj_set_width/height/size()`void。

因此，您需要更改*长模式*，首先设置新的*长模式*，然后使用设置大小`lv_obj_set_width/height/size()`。

另一个重要的注意事项是`LV_LABEL_LONG_DOT`**就地操纵文本缓冲区**，以添加/删除点。使用`lv_label_set_text`或时`lv_label_set_array_text`，将分配一个单独的缓冲区，并且该实现细节未被注意。事实并非如此`lv_label_set_static_text`！**如果打算使用，传递给的缓冲区**`lv_label_set_static_text`**必须是可写的**`LV_LABEL_LONG_DOT`。

24.3.4 文字对齐

文本的行可以通过左右对齐。请注意，它将仅对齐线，而不对齐标签对象本身。`lv_label_set_align(label, LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)`

标签本身不支持垂直对齐；您应该将标签放在更大的容器中，然后将整个标签对象对齐。

24.3.5 文字重新着色

在文本中，您可以使用命令来重新着色部分文本。例如：。可以按功能为每个标签分别启用此功能。`"Write a #ff0000 red# word"lv_label_set_recolor()`

请注意，重新着色只能在一行中进行。因此，`\n`不应在重新着色的文本中使用它或将其`LV_LABEL_LONG_BREAK`换行，否则新行中的文本将不会重新着色。

24.3.6 很长的文字

Lvgl 通过保存一些额外的数据（∽12 个字节）以加快绘图速度，可以有效地处理很长的字符（> 40k 个字符）。要启用此功能，请在 *lv_conf.h 中进行*设置。`LV_LABEL_LONG_TXT_HINT 1`

24.3.7 sign

标签可以在字母旁边显示 sign（或单独显示）。阅读字体部分以了解有关 sign 的更多信息。

24.4 事件

仅通用事件是按对象类型发送的。

24.5 按键

对象类型不处理任何*键*。

例

C

标签重新着色和滚动


Re-color words of a
label and wrap long
text automatically.
It is a circularly scro

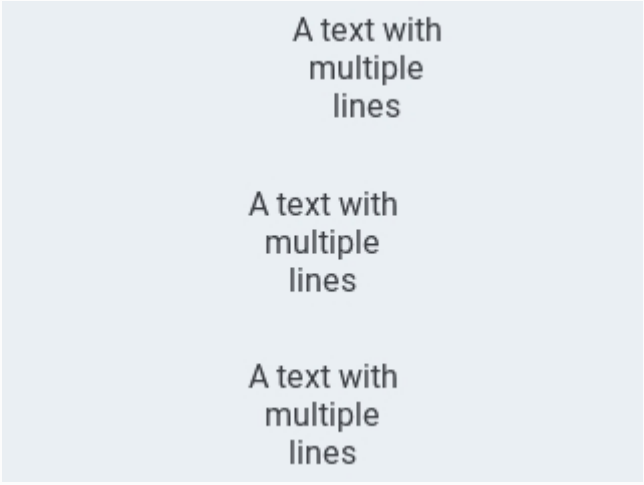
码

文字阴影


A simple method to create
shadows on text
It even works with
newlines and spaces.
Cai Xuefeng

码

对齐标签


A text with
multiple
lines
A text with
multiple
lines
A text with
multiple
lines

API

typedef

typedef uint8_t **lv_label_long_mode_t**

typedef uint8_t **lv_label_align_t**

typedef uint8_t **lv_label_part_t**

枚举

Enum [anonymous]

长模式行为。在' lv_label_ext_t '中使用

值:

enumerator **LV_LABEL_LONG_EXPAND**

将对象大小扩展为文本大小

enumerator **LV_LABEL_LONG_BREAK**

保持对象的宽度，断开太长的线并扩大对象的高度

enumerator **LV_LABEL_LONG_DOT**

如果文字太长，请保持大小并在最后写点

enumerator **LV_LABEL_LONG_SROLL**

保持大小并来回滚动文本

enumerator **LV_LABEL_LONG_SROLL_CIRC**

保持大小并循环滚动文本

enumerator **LV_LABEL_LONG_CROP**

保持大小并裁剪文本

Enum [anonymous]

标签对齐政策

值:

enumerator **LV_LABEL_ALIGN_LEFT**

文字向左对齐

enumerator **LV_LABEL_ALIGN_CENTER**

文字居中对齐

enumerator **LV_LABEL_ALIGN_RIGHT**

文字右对齐

enumerator **LV_LABEL_ALIGN_AUTO**

根据文本的方向使用左或右（LTR / RTL）

***Enum** [anonymous]*

标签样式

值：

enumerator **LV_LABEL_PART_MAIN**

功能

LV_EXPORT_CONST_INT（LV_LABEL_DOT_NUM）

LV_EXPORT_CONST_INT（LV_LABEL_POS_LAST）

LV_EXPORT_CONST_INT（LV_LABEL_TEXT_SEL_OFF）

lv_obj_t * **lv_label_create**（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建标签对象

返回

指向创建的按钮的指针

参数

- `par`：指向对象的指针，它将是新标签的父对象
- `copy`：指向按钮对象的指针，如果不为 NULL，则将从其复制新对象

void lv_label_set_text（lv_obj_t * *label*，*const* char* *text*）

设置标签的新文本。标签将分配内存以存储文本。

参数

- `label`：指向标签对象的指针
- `text`：'\ 0'终止的字符串。使用 NULL 刷新当前文本。

void lv_label_set_text_fmt（lv_obj_t * *label*，*const* char * *fmt*，... ）

设置标签的新格式的文本。标签将分配内存以存储文本。

参数

- `label`：指向标签对象的指针

- `fmt`：`printf`-类似格式

void lv_label_set_text_static（lv_obj_t * label，const char* text）

设置一个静态文本。标签将不会保存该标签，因此标签存在时，“文本”变量必须为“有效”。

参数

- `label`：指向标签对象的指针
- `text`：指向文本的指针。使用 NULL 刷新当前文本。

void lv_label_set_long_mode（lv_obj_t * label，lv_label_long_mode_t long_mode ）

使用更长的文本然后设置对象大小来设置标签的行为

参数

- `label`：指向标签对象的指针
- `long_mode`：来自“ lv_label_long_mode”枚举的新模式。在此功能之后，应在 LV_LONG_BREAK / LONG / ROLL 中设置标签的大小

void lv_label_set_align（lv_obj_t * label，lv_label_align_t align ）

设置标签的对齐方式（左或右）

参数

- `label`：指向标签对象的指针
- `align`：'LV_LABEL_ALIGN_LEFT'或'LV_LABEL_ALIGN_LEFT'

void lv_label_set_recolor（lv_obj_t * label，bool en ）

通过内联命令启用重新着色

参数

- `label`：指向标签对象的指针
- `en`：true：启用重新着色，false：禁用

void lv_label_set_anim_speed（lv_obj_t * label，uint16_t anim_speed ）

在 LV_LABEL_LONG_SROLL / SCROLL_CIRC 模式下设置标签的动画速度

参数

- `label`：指向标签对象的指针
- `anim_speed`：动画速度（以像素/秒为单位）

void lv_label_set_text_sel_start（lv_obj_t * label，uint32_t index）

设置选择开始索引。

参数

- label：指向标签对象的指针。
- index：要设置的索引。LV_LABEL_TXT_SEL_OFF什么也没选择。

void lv_label_set_text_sel_end（lv_obj_t * label，uint32_t index）

设置选择结束索引。

参数

- label：指向标签对象的指针。
- index：要设置的索引。LV_LABEL_TXT_SEL_OFF什么也没选择。

字符* lv_label_get_text（const lv_obj_t * label）

获取标签文本

返回

标签文字

参数

- label：指向标签对象的指针

lv_label_long_mode_t lv_label_get_long_mode（const lv_obj_t * label）

获取标签的长模式

返回

长模式

参数

- label：指向标签对象的指针

lv_label_align_t lv_label_get_align（const lv_obj_t * label）

获取 align 属性

返回

LV_LABEL_ALIGN_LEFT 或 LV_LABEL_ALIGN_CENTER

参数

- label：指向标签对象的指针

bool lv_label_get_recolor（const lv_obj_t * label ）

获取重新着色属性

返回

true：启用重新着色，false：禁用

参数

- `label`：指向标签对象的指针

uint16_t lv_label_get_anim_speed（const lv_obj_t * label）

在 LV_LABEL_LONG_ROLL 和 SCROLL 模式下获取标签的动画速度

返回

动画速度，以 px / sec 为单位

参数

- `label`：指向标签对象的指针

void lv_label_get_letter_pos（const lv_obj_t * label，uint32_t index，lv_point_t * pos ）

获取字母的相对 x 和 y 坐标

参数

- `label`：指向标签对象的指针
- `index`：字母的索引[0 ...文本长度]。以字符索引表示，而不是以字节索引表示（与 UTF-8 不同）
- `pos`：将结果存储在此处（例如，索引= 0 给出 0; 0 坐标）

uint32_t lv_label_get_letter_on（const lv_obj_t * label，lv_point_t * pos ）

获取标签相对点上的字母索引

返回

'pos_p'点上字母的索引（例如 0; 0 为 0。字母）以字符索引而不是字节索引表示（与 UTF-8 不同）

参数

- `label`：指向标签对象的指针
- `pos`：指向标签上具有坐标的指针

bool lv_label_is_char_under_pos（const lv_obj_t * label，lv_point_t * pos ）

检查是否在一个点下绘制了一个字符。

返回

是否在该点下绘制了字符

参数

- `label`：标签对象
- `pos`：指向要检查的字符

uint32_t lv_label_get_text_sel_start（*const*lv_obj_t * *label*）

获取选择开始索引。

返回

选择开始索引。`LV_LABEL_TXT_SEL_OFF`如果未选择任何内容。

参数

- `label`：指向标签对象的指针。

uint32_t lv_label_get_text_sel_end（*const*lv_obj_t * *label*）

获取选择结束索引。

返回

选择结束索引。`LV_LABEL_TXT_SEL_OFF`如果未选择任何内容。

参数

- `label`：指向标签对象的指针。

lv_style_list_t * lv_label_get_style（lv_obj_t * *label*，uint8_t *type*）

voidlv_label_ins_text（lv_obj_t * *label*，uint32_t *pos*，*const* char* *txt* ）

在标签上插入文本。标签文本不能是静态的。

参数

- `label`：指向标签对象的指针
- `pos`：要插入的字符索引。以字符索引而不是字节索引（在 UTF-8 中不同）表示

0：在第一个字符之前。LV_LABEL_POS_LAST：在最后一个字符之后。

- `txt`：指向要插入的文本的指针

voidlv_label_cut_text（lv_obj_t * *label*，uint32_t *pos*，uint32_t *cnt* ）

从标签中删除字符。标签文本不能是静态的。

参数

- `label`：指向标签对象的指针
- `pos`：要插入的字符索引。以字符索引而不是字节索引（在 UTF-8 中不同）表示 0：在第一个字符之前。
- `cnt`：要剪切的字符数

struct lv_label_ext_t

#include <lv_label.h>

标签数据

公众成员

Char *text
Char *tmp_ptr
Char tmp[LV_LABEL_DOT_NUM+ 1]
union lv_label_ext_t :: [anonymous]dot
uint32_t dot_end
uint16_t anim_speed
lv_point_t offset
lv_draw_label_hint_t hint
uint32_t sel_start
uint32_t sel_end
lv_label_long_mode_tlong_mode
uint8_t static_txt
uint8_t align
uint8_t recolor
uint8_t expand
uint8_t dot_tmp_alloc

第二十五章 LED（lv_led）

25.1 总览

LED 是矩形（或圆形）的对象。它的亮度可以调节。亮度降低时，LED 的颜色会变暗。

25.2 小部件和样式

LED 只有一个主要部分，称为 LED `LV_LED_PART_MAIN`，它使用了所有典型的背景样式属性。

25.3 用法

25.3.1 亮度

您可以使用设置亮度。亮度应介于 0（最暗）和 255（最亮）之间。

```
lv_led_set_bright(led, bright)
```

Cai Xuefeng

25.3.2 切换

使用 `lv_led_on(led)` 和 `lv_led_off(led)` 将亮度设置为预定义的 ON 或 OFF 值。在 `lv_led_toggle(led)` ON 和 OFF 状态之间切换。

25.4 事件

仅通用事件是按对象类型发送的。

25.5 按键

对象类型不处理任何*键*。

例

C

定制风格的 LED

API

typedef

typedef uint8_t lv_led_part_t

枚举

enum [anonymous]

值：

enumerator LV_LED_PART_MAIN= LV_OBJ_PART_MAIN

功能

lv_obj_t * lv_led_create（lv_obj_t * *par*，*const* lv_obj_t * *copy* ）

创建一个 led 对象

返回

指向创建的 LED 的指针

参数

- `par`：指向对象的指针，它将是新 led 的父对象
- `copy`：指向 led 对象的指针，如果不为 NULL，则将从其复制新对象

voidlv_led_set_bright（lv_obj_t * *led*，uint8_t *bright*）

设置 LED 对象的亮度

参数

- `led`：指向 LED 对象的指针

- `bright`：LV_LED_BRIGHT_MIN（最大暗）... LV_LED_BRIGHT_MAX（最大亮）

void lv_led_on（lv_obj_t * *led*）

LED 灯亮

参数

- `led`：指向 LED 对象的指针

void lv_led_off（lv_obj_t * *led*）

熄灭 LED

参数

- `led`：指向 LED 对象的指针

void lv_led_toggle（lv_obj_t * *led*）

切换 LED 的状态

参数

- `led`：指向 LED 对象的指针

uint8_t lv_led_get_bright（*const* lv_obj_t * *led*）

获取 LEd 对象的亮度

返回

亮 0（最大暗）... 255（最大亮）

参数

- `led`：指向 LED 对象的指针

struct lv_led_ext_t

公众成员

uint8_t bright

第二十六章 线（lv_line）

26.1 总览

Line 对象能够在一组点之间绘制直线。

26.2 小部件和样式

线只有一个主要部分，称为 `LV_LABEL_PART_MAIN`。它使用所有*线型*属性。

26.3 用法

26.3.1 设置点

这些点必须存储在 `lv_point_t` 数组中，并通过函数传递给对象。

```
lv_line_set_points(lines, point_array, point_cnt)
```


26.3.2 自动尺寸

可以根据其点自动设置线对象的大小。可以通过该功能启用。如果启用，则在设置点后，将根据点之间的最大 x 和 y 坐标更改对象的宽度和高度。该*自动调整大小*默认情况下启用。

```
lv_line_set_auto_size(line, true)
```

26.3.3 倒 y

通过默认，*y == 0* 点位于对象的顶部。在某些情况下，它可能是直觉的，因此可以使用反转 y 坐标。在这种情况下， *y == 0* 将是对象的底部。该 *Ÿ 反转*默认情况下禁用。

```
lv_line_set_y_invert(line, true)
```

26.4 事件

仅通用事件是按对象类型发送的。

26.5 按键

对象类型不处理任何*键*。

例

C

简单线

API

Cai Xuefeng

typedef

```
typedef uint8_t lv_line_style_t
```

枚举

```
enum [anonymous]
```

值：

enumerator **LV_LINE_PART_MAIN**

功能

```
lv_obj_t * lv_line_create（lv_obj_t * par，const lv_obj_t * copy）
```

创建线对象

返回

指向创建的行的指针

参数

- `par`：指向对象的指针，它将是新行的父对象

void lv_line_set_points（lv_obj_t * line，const lv_point_t point_a [ ]，uint16_t point_num ）

设置点数组。线对象将连接这些点。

参数

- `line`：指向线对象的指针
- `point_a`：点数组。仅保存地址，因此数组不能是将被破坏的局部变量
- `point_num`：'point_a'中的点数

void lv_line_set_auto_size（lv_obj_t * line，bool en ）

启用（或禁用）自动调整大小选项。对象的大小将适合其要点。（将宽度设置为 x max，将高度设置为 y max）

参数

- `line`：指向线对象的指针
- `en`：true：启用自动调整大小，false：禁用自动调整大小

void lv_line_set_y_invert（lv_obj_t * line，bool en ）

启用（或禁用）y 坐标反转。如果启用，则将从对象的高度减去 y，因此 y = 0 坐标将在底部。

参数

- `line`：指向线对象的指针
- `en`：true：启用 y 反转，false：禁用 y 反转

bool lv_line_get_auto_size（const lv_obj_t * line）

获取自动尺寸属性

返回

true：启用自动调整大小，false：禁用

参数

- `line`：指向线对象的指针

bool lv_line_get_y_invert（const lv_obj_t * line）

获取 y 反转属性

返回

true：启用 y 反转，false：禁用

参数

- `line`: 指向线对象的指针

---

struct **lv_line_ext_t**

公众成员

const lv_point_t ***point_array**

uint16_t **point_num**

uint8_t **auto_size**

uint8_t **y_inv**

第二十七章 列表（lv_list）

27.1 总览

列表是从背景页面和其上的按钮构建的。按钮包含一个可选的类似图标的图像（也可以是一个 sign）和一个 Label。当列表足够长时，可以滚动它。

27.2 小部件和样式

列表与页面具有相同的部分

- `LV_LIST_PART_BG`
- `LV_LIST_PART_SCRL`
- `LV_LIST_PART_SCRLBAR`
- `LV_LIST_PART_EDGE_FLASH`

有关详细信息，请参见页面文档。

列表上的按钮被视为普通按钮，它们只有一个主要部分称为 `LV_BTN_PART_MAIN`。

27.3 用法

27.3.1 添加按钮

您可以使用或使用 symbol 添加新的列表元素（按钮）。如果您不想添加图像，请用作图像源。该函数返回指向创建的按钮的指针，以允许进行进一步的配置。

```
lv_list_add_btn(list, &icon_img, "Text")lv_list_add_btn(list, SYMBOL_EDIT, "Edit text")NULL
```

根据对象的宽度，将按钮的宽度设置为最大。按钮的高度会根据内容自动调整。（*内容高度 + padding_top + padding_bottom*）。

使用 `LV_LABEL_LONG_SROLL_CIRC` 长模式创建标签，以自动循环滚动长标签。

`lv_list_get_btn_label(list_btn)` 并 `lv_list_get_btn_img(list_btn)` 可以用来获取标签和列表按钮的图像。可以直接使用等文本 `lv_list_get_btn_text(list_btn)`。

27.3.2 删除按钮

要删除列表元素，只需使用 `lv_obj_del(btn)` 返回值 `lv_list_add_btn()`。

要清除列表（删除所有按钮），请使用 `lv_list_clean(list)`

27.3.3 手动导航

您可以使用 `lv_list_up(list)` 和在列表中手动导航 `lv_list_down(list)`。

您可以使用直接将焦点放在按钮上。 `lv_list_focus(btn, LV_ANIM_ON/OFF)`

上/下/焦点移动的**动画时间**可以通过以下方式设置：。动画时间为零表示不是动画。
`lv_list_set_anim_time(list, anim_time)`

27.3.4 布局

默认情况下，列表是垂直的。要获取水平列表，请使用。
`lv_list_set_layout(list, LV_LAYOUT_ROW_MID)`

27.3.5 边缘闪光灯

当列表到达最高或最低位置时，可以显示类似圆圈的效果。 启用此功能。
`lv_list_set_edge_flash(list, true)`

27.3.6 滚动传播

如果列表是在其他可滚动元素（如 Page）上创建的，并且列表无法进一步滚动，则滚动可以传播到父级。这样，滚动将在父级上继续。可以启用 `lv_list_set_scroll_propagation(list, true)`

27.4 事件

仅通用事件是按对象类型发送的。

了解有关事件的更多信息。

27.5 按键

The following *Keys* are processed by the Lists:

- **LV_KEY_RIGHT/DOWN** Select the next button
- **LV_KEY_LEFT/UP** Select the previous button

Note that, as usual, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

The Selected buttons are in `LV_BTN_STATE_PR/TG_PR` state.

To manually select a button use `lv_list_set_btn_selected(list, btn)`. When the list is defocused and focused again it will restore the last selected button.

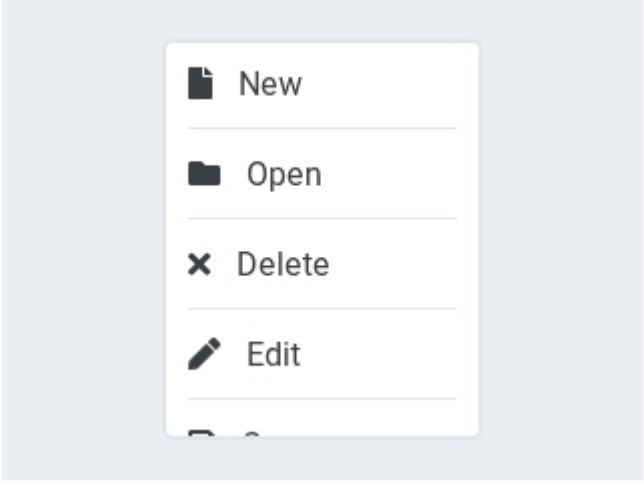
Learn more about Keys.

Cai Xuefeng

示例

C

Simple List



API

Typedefs

typedef uint8_t **lv_list_style_t**

Enums

enum [anonymous]

List styles.

Values:

enumerator **LV_LIST_PART_BG** = LV_PAGE_PART_BG

List background style

enumerator **LV_LIST_PART_SCROLLBAR** = LV_PAGE_PART_SCROLLBAR

List scrollbar style.

enumerator **LV_LIST_PART_EDGE_FLASH** = LV_PAGE_PART_EDGE_FLASH

List edge flash style.

enumerator **_LV_LIST_PART_VIRTUAL_LAST** = _LV_PAGE_PART_VIRTUAL_LAST

enumerator **LV_LIST_PART_SCROLLABLE** = LV_PAGE_PART_SCROLLABLE

List scrollable area style.

enumerator **_LV_LIST_PART_REAL_LAST** = _LV_PAGE_PART_REAL_LAST

Functions

lv_obj_t ***lv_list_create**(lv_obj_t ***par**, *const* lv_obj_t ***copy**)

Create a list objects

Return

pointer to the created list

Parameters

- `par`: pointer to an object, it will be the parent of the new list
- `copy`: pointer to a list object, if not NULL then the new object will be copied from it

void **lv_list_clean**(lv_obj_t ***list**)

Delete all children of the scrl object, without deleting scrl child.

Parameters

- `list`: pointer to an object

lv_obj_t *lv_list_add_btn(lv_obj_t *list, const void *img_src, const char *txt)

Add a list element to the list

Return

pointer to the new list element which can be customized (a button)

Parameters

- `list`: pointer to list object
- `img_fn`: file name of an image before the text (NULL if unused)
- `txt`: text of the list element (NULL if unused)

bool lv_list_remove(const lv_obj_t *list, uint16_t index)

Remove the index of the button in the list

Return

true: successfully deleted

Parameters

- `list`: pointer to a list object
- `index`: pointer to a the button's index in the list, index must be 0 <= index < lv_list_ext_t.size

void lv_list_focus_btn(lv_obj_t *list, lv_obj_t *btn)

Make a button selected

Parameters

- `list`: pointer to a list object
- `btn`: pointer to a button to select NULL to not select any buttons

void lv_list_set_scrollbar_mode(lv_obj_t *list, lv_scrollbar_mode_t mode)

Set the scroll bar mode of a list

Parameters

- `list`: pointer to a list object
- `sb_mode`: the new mode from 'lv_page_sb_mode_t' enum

void lv_list_set_scroll_propagation(lv_obj_t *list, bool en)

Enable the scroll propagation feature. If enabled then the List will move its parent if there is no more space to scroll.

Parameters

- `list`: pointer to a List
- `en`: true or false to enable/disable scroll propagation

---

void `lv_list_set_edge_flash`(lv_obj_t *_list_, bool _en_)

Enable the edge flash effect. (Show an arc when the an edge is reached)

Parameters

- `list`: pointer to a List
- `en`: true or false to enable/disable end flash

---

void `lv_list_set_anim_time`(lv_obj_t *_list_, uint16_t _anim_time_)

Set scroll animation duration on 'list_up()' 'list_down()' 'list_focus()'

Parameters

- `list`: pointer to a list object
- `anim_time`: duration of animation [ms]

---

void `lv_list_set_layout`(lv_obj_t *_list_, lv_layout_t _layout_)

Set layout of a list

Parameters

- `list`: pointer to a list object
- `layout`: which layout should be used

***const* char *`lv_list_get_btn_text`(*const* lv_obj_t *_btn_)**

Get the text of a list element

Return

pointer to the text

Parameters

- `btn`: pointer to list element

---

lv_obj_t *`lv_list_get_btn_label`(*const* lv_obj_t *_btn_)

Get the label object from a list element

Return

pointer to the label from the list element or NULL if not found

Parameters

- `btn`: pointer to a list element (button)

---

lv_obj_t *lv_list_get_btn_img(const lv_obj_t *btn)

Get the image object from a list element

Return

pointer to the image from the list element or NULL if not found

Parameters

- `btn`: pointer to a list element (button)

---

lv_obj_t *lv_list_get_prev_btn(const lv_obj_t *list, lv_obj_t *prev_btn)

Get the next button from list. (Starts from the bottom button)

Return

pointer to the next button or NULL when no more buttons

Parameters

- `list`: pointer to a list object
- `prev_btn`: pointer to button. Search the next after it.

---

lv_obj_t *lv_list_get_next_btn(const lv_obj_t *list, lv_obj_t *prev_btn)

Get the previous button from list. (Starts from the top button)

Return

pointer to the previous button or NULL when no more buttons

Parameters

- `list`: pointer to a list object
- `prev_btn`: pointer to button. Search the previous before it.

---

int32_t lv_list_get_btn_index(const lv_obj_t *list, const lv_obj_t *btn)

Get the index of the button in the list

Return

the index of the button in the list, or -1 of the button not in this list

Parameters

- `list`: pointer to a list object. If NULL, assumes btn is part of a list.
- `btn`: pointer to a list element (button)

---

uint16_t **lv_list_get_size**(*const* lv_obj_t **list*)

Get the number of buttons in the list

Return

the number of buttons in the list

Parameters

- `list`: pointer to a list object

---

lv_obj_t ***lv_list_get_btn_selected**(*const* lv_obj_t **list*)

Get the currently selected button. Can be used while navigating in the list with a keypad.

Return

pointer to the selected button

Parameters

- `list`: pointer to a list object

---

lv_layout_t**lv_list_get_layout**(lv_obj_t **list*)

Get layout of a list

Return

layout of the list object

Parameters

- `list`: pointer to a list object

---

lv_scrollbar_mode_t**lv_list_get_scrollbar_mode**(*const* lv_obj_t **list*)

Get the scroll bar mode of a list

Return

scrollbar mode from 'lv_scrollbar_mode_t' enum

Parameters

- `list`: pointer to a list object

bool lv_list_get_scroll_propagation(lv_obj_t *list)

Get the scroll propagation property

Return

true or false

Parameters

- `list`: pointer to a List

bool lv_list_get_edge_flash(lv_obj_t *list)

Get the scroll propagation property

Return

true or false

Parameters

- `list`: pointer to a List

uint16_t lv_list_get_anim_time(const lv_obj_t *list)

Get scroll animation duration

Return

duration of animation [ms]

Parameters

- `list`: pointer to a list object

void lv_list_up(const lv_obj_t *list)

Move the list elements up by one

Parameters

- `list`: pointer a to list object

void lv_list_down(const lv_obj_t *list)

Move the list elements down by one

Parameters

- `list`: pointer to a list object

void lv_list_focus(const lv_obj_t * btn，lv_anim_enable_t anim ）

专注于列表按钮。这样可以确保该按钮在列表中可见。

参数

- `btn`：指向要聚焦的列表按钮的指针
- `anim`：LV_ANOM_ON：带有动画滚动，LV_ANIM_OFF：不带有动画

struct **lv_list_ext_t**

公众成员

```
lv_page_ext_t page
lv_obj_t *last_sel_btn
lv_obj_t *act_sel_btn
```

第二十八章 仪表（lv_lmeter）

28.1 总览

线表对象由一些绘制比例的径向线组成。设置线表的值将按比例更改比例线的颜色。

28.2 小部件和样式

线表只有一个主要部分，称为 `LV_LINEMETER_PART_MAIN`。它使用所有典型的背景属性绘制矩形或圆形背景，并使用 *line* 和 *scale* 属性绘制比例线。活动行（与当前值的较小值相关）从 *line_color* 变为 *scale_grad_color*。最后一行（当前值之后）设置为 *scale_end_color* 颜色。

28.3 用法

28.3.1 设定值

当比例的比例部分设置为新值时，将重新着色。`lv_linemeter_set_value(linemeter, new_value)`

28.3.2 范围和角度

该功能设置线表的范围。`lv_linemeter_set_range(linemeter, min, max)`

您可以设置规模的角度和线路的数量：。默认角度为 240，默认行号为 31。

`lv_linemeter_set_scale(linemeter, angle, line_num)`

28.3.3 角度偏移

默认情况下，刻度角相对于 y 轴对称地解释。这导致“站立”线表。与 `lv_linemeter_set_angle_offset` 一个偏移可以被添加的规模角度。例如，它可以用于将四分之一线表放在角落或半线表到右侧或左侧。

28.3.4 镜子

默认情况下，线表的线路是顺时针激活的。可以使用更改。

```
lv_linemeter_set_mirror(linemeter, true/false)
```

28.4 事件

仅通用事件是按对象类型发送的。

了解有关事件的更多信息。

28.5 按键

对象类型不处理任何*键*。

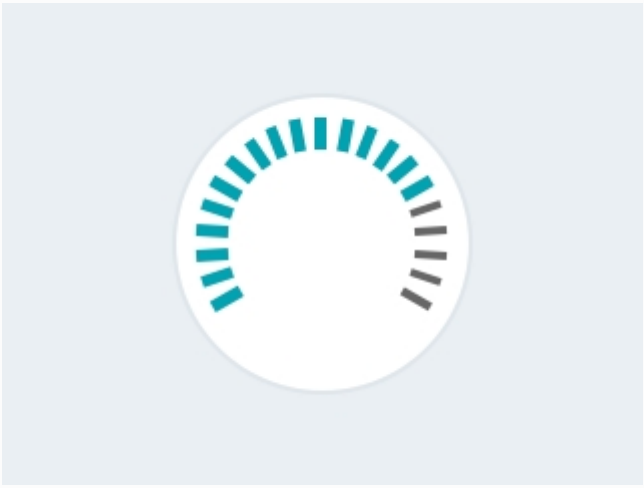
了解更多有关按键的信息。

例

C

Cai Xuefeng

简单仪表



API

typedef

```
typedef uint8_t lv_linemeter_part_t
```

枚举

enum [anonymous]

值：

enumerator **LV_LINEMETER_PART_MAIN**

enumerator **_LV_LINEMETER_PART_VIRTUAL_LAST**

enumerator **_LV_LINEMETER_PART_REAL_LAST**= **_LV_OBJ_PART_REAL_LAST**

功能

lv_obj_t * **lv_linemeter_create**（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建一个线表对象

返回

指向创建的线表的指针

参数

- `par`：指向对象的指针，它将是新线表的父级
- `copy`：指向线表对象的指针，如果不为 NULL，则将从其复制新对象

void**lv_linemeter_set_value**（lv_obj_t * *lmeter*，int32_t *value*）

在线路表上设置一个新值 Cai Xuefeng

参数

- `lmeter`：指向线表对象的指针
- `value`：新价值

void**lv_linemeter_set_range**（lv_obj_t * *lmeter*，int32_t *min*，int32_t *max* ）

设置线表的最小值和最大值

参数

- `lmeter`：指向线表对象的指针
- `min`：最小值
- `max`：最大值

void **lv_linemeter_set_scale**（lv_obj_t * *lmeter*，uint16_t *angle*，uint16_t *line_cnt* ）

设置线表的刻度设置

参数

- `lmeter`：指向线表对象的指针

- `angle`：比例尺角度（0..360）
- `line_cnt`：行数

void lv_linemeter_set_angle_offset（lv_obj_t * *lmeter*，uint16_t *angle*）

为线表的角度设置一个偏移量以旋转它。

参数

- `lmeter`：指向线表对象的指针
- `angle`：角度偏移（0..360），顺时针旋转

void lv_linemeter_set_mirror（lv_obj_t * *lmeter*，bool *mirror*）

设置仪表增长的方向，顺时针或逆时针（镜像）

参数

- `lmeter`：指向线表对象的指针
- `mirror`：镜子设置

int32_t lv_linemeter_get_value（const lv_obj_t * *lmeter* ）

获取线表的值

返回

线表的值

参数

- `lmeter`：指向线表对象的指针

int32_t lv_linemeter_get_min_value（const lv_obj_t * *lmeter* ）

获得线表的最小值

返回

线表的最小值

参数

- `lmeter`：指向线表对象的指针

int32_t lv_linemeter_get_max_value（const lv_obj_t * *lmeter* ）

获取线表的最大值

返回

线表的最大值

参数

- `lmeter`：指向线表对象的指针

---

uint16_t lv_linemeter_get_line_count（*const*lv_obj_t * *lmeter* ）

获取线表的刻度号

返回

刻度单位数

参数

- `lmeter`：指向线表对象的指针

---

uint16_t lv_linemeter_get_scale_angle（*const*lv_obj_t * *lmeter* ）

获取线规的刻度角

返回

刻度角

参数

- `lmeter`：指向线表对象的指针

---

uint16_t lv_linemeter_get_angle_offset（lv_obj_t * *lmeter* ）

获取线表的偏移量。

返回

角度偏移（0..360）

参数

- `lmeter`：指向线表对象的指针

---

void lv_linemeter_draw_scale（lv_obj_t * *lmeter*，*const*lv_area_t * *clip_area*，uint8_t *part* ）

bool lv_linemeter_get_mirror（lv_obj_t * *lmeter* ）

获取线表的镜像设置

返回

镜像（对或错）

参数

- `lmeter`：指向线表对象的指针

***struct* lv_linemeter_ext_t**

公众成员

```
    uint16_t scale_angle
    uint16_t angle_ofs
    uint16_t line_cnt
    int32_t cur_value
    int32_t min_value
    int32_t max_value
    uint8_t mirrored
```

Cai Xuefeng

第二十九章 消息框（lv_msdbox）

29.1 总览

消息框充当弹出窗口。它们是由背景 Container，Label 和 Button 的 Button 矩阵构建的。

文本将自动分成多行（具有 `LV_LABEL_LONG_MODE_BREAK`），高度将自动设置为包含文本和按钮（`LV_FIT_TIGHT` 垂直适合），

29.2 小部件和样式

消息框的主要部分被调用 `LV_MSGBOX_PART_MAIN`，它使用所有典型的背景样式属性。使用填充会增加侧面的空间。*pad_inner* 将在文本和按钮之间添加空格。该*标签*样式属性影响文本的样式。

按钮部分与 Button 矩阵的情况相同：

- `LV_MSGBOX_PART_BTN_BG` 按钮的背景
- `LV_MSGBOX_PART_BTN` 按钮

29.3 用法

29.3.1 设定文字

要设置文本，请使用功能。不仅将保存文本指针，而且文本也可以位于局部变量中。

```
lv_msgbox_set_text(msgbox, "My text")
```

29.3.2 添加按钮

要添加按钮，请使用该功能。需要像这样指定按钮的文本。有关更多信息，请访问 Button 矩阵文档。`lv_msgbox_add_btns(msgbox, btn_str)const char * btn_str[] = {"Apply", "Close", ""}`

仅在 `lv_msgbox_add_btns()` 首次调用时才会创建按钮矩阵。

29.3.3 自动关闭

带有该消息的框可以在几毫秒后自动关闭带有动画。该功能停止启动的自动关闭。

`lv_msgbox_start_auto_close(mbox, delay)delaylv_mbox_stop_auto_close(mbox)`

关闭动画的持续时间可以通过设置。`lv_mbox_set_anim_time(mbox, anim_time)`

29.4 事件

除了“ 常规”事件外，“消息”框还会发送以下特殊事件：

- 单击该按钮时发送 **LV_EVENT_VALUE_CHANGED**。事件数据设置为单击按钮的 ID。

消息框具有一个默认的事件回调，当单击按钮时，该事件回调将自行关闭。

29.5 按键值

以下*按键*由按钮处理：

- **LV_KEY_RIGHT / DOWN** 选择下一个按钮
- **LV_KEY_LEFT / TOP** 选择上一个按钮
- **LV_KEY_ENTER** 单击选定的按钮

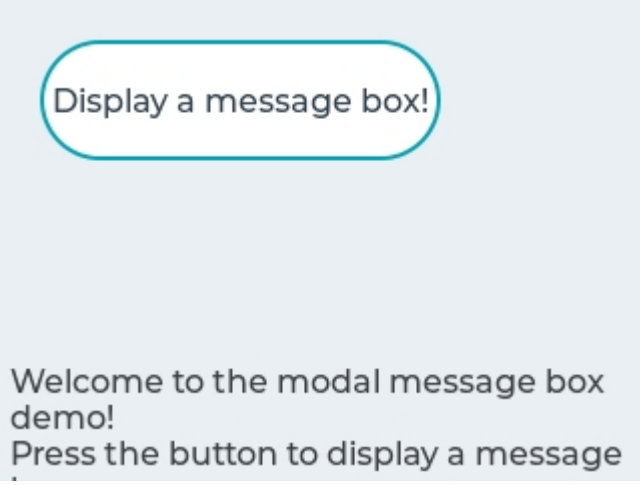
例

C

简单消息框

码

模态



API

typedef

typedef uint8_t **lv_msgbox_style_t**

枚举

enum [anonymous]

消息框样式。

值：

enumerator **LV_MSGBOX_PART_BG**= LV_CONT_PART_MAIN

enumerator **LV_MSGBOX_PART_BTN_BG**= _LV_CONT_PART_REAL_LAST

enumerator **LV_MSGBOX_PART_BTN**

功能

lv_obj_t * **lv_msgbox_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个消息框对象

返回

指向创建的消息框的指针

参数

- `par`：指向对象的指针，它将是新消息框的父对象
- `copy`：指向消息框对象的指针，如果不为 NULL，则将从其复制新对象

void**lv_msgbox_add_btns**（lv_obj_t * *mbox*，*const*char * *btn_mapaction* [ ] ）

将按钮添加到消息框

参数

- `mbox`：指向消息框对象的指针
- `btn_map`：按钮描述符（按钮矩阵图）。例如 const char * txt [] = {“ ok”，“ close”，“”}（不能是局部变量）

void lv_msgbox_set_text（lv_obj_t * *mbox*，*const* char* *txt* ）

设置消息框的文本

参数

- `mbox`：指向消息框的指针
- `txt`：以'\ 0'结尾的字符串，它将是消息框的文本

void lv_msgbox_set_anim_time（lv_obj_t * *mbox*，uint16_t *anim_time* ）

设置动画时长

参数

- `mbox`：指向消息框对象的指针
- `anim_time`：动画长度（以毫秒为单位）（0：无动画）

void lv_msgbox_start_auto_close（lv_obj_t * *mbox*，uint16_t *delay* ）

在给定时间后自动删除消息框

参数

- `mbox`：指向消息框对象的指针
- `delay`：删除消息框之前等待的时间（以毫秒为单位）

void lv_msgbox_stop_auto_close（lv_obj_t * *mbox* ）

停止自动。关闭消息框

参数

- `mbox`：指向消息框对象的指针

void lv_msgbox_set_recolor（lv_obj_t * *mbox*，bool *en* ）

设置是否启用重新着色。必须在之后调用 `lv_msgbox_add_btns`。

参数

- `mbox`：指向消息框对象的指针
- `en`：是否启用重新着色

const 字符* lv_msgbox_get_text（*const* lv_obj_t * *mbox* ）

获取消息框的文本

返回

指向消息框文本的指针

参数

- `mbox`：指向消息框对象的指针

uint16_t lv_msgbox_get_active_btn（lv_obj_t * *mbox* ）

获取用户最后按下的按钮的索引（按下，释放等）`event_cb`。

返回

最后释放按钮的索引（LV_BTNMATRIX_BTN_NONE：如果未设置）

参数

- `mbox`：指向消息框对象的指针

const 字符* lv_msgbox_get_active_btn_text（lv_obj_t * *mbox* ）

获取用户最后按下的按钮的文本（按下，释放等）`event_cb`。

返回

最后释放的按钮的文本（NULL：如果未设置）

参数

- `mbox`：指向消息框对象的指针

uint16_t lv_msgbox_get_anim_time（*const* lv_obj_t * *mbox* ）

获取动画持续时间（接近动画时间）

返回

动画长度（以毫秒为单位）（0：无动画）

参数

- `mbox`：指向消息框对象的指针

bool lv_msgbox_get_recolor（*const* lv_obj_t * *mbox* ）

获取是否启用了重新着色

返回

是否启用重新着色

参数

- `mbox`：指向消息框对象的指针

---

lv_obj_t * **lv_msgbox_get_btnmatrix**（lv_obj_t * ***mbox***）

获取消息框按钮矩阵

返回

指向按钮矩阵对象的指针

备注

除非 `lv_msgbox_add_btns` 已被调用，否则返回值将为 NULL

参数

- `mbox`：指向消息框对象的指针

---

struct **lv_msgbox_ext_t**

公众成员

lv_cont_ext_t **bg**

lv_obj_t ***text**

lv_obj_t ***btnm**

uint16_t **anim_time**

第三十一章 对象遮罩（lv_objmask）

31.1 总览

绘制其子级时，*对象蒙版*能够向图形添加一些蒙版。

31.2 小部件和样式

对象蒙版只有一个主要部分 `LV_OBJMASK_PART_BG`，它使用典型的背景样式属性。

31.3 用法

31.3.1 加面膜

在向*对象蒙版*添加蒙版之前，应初始化*蒙版*：

```
lv_draw_mask_<type>_param_t mask_param;
lv_draw_mask_<type>_init(&mask_param, ...);
lv_objmask_mask_t * mask_p = lv_objmask_add_mask(objmask, &mask_param);
```


Lvgl 支持以下掩码类型：

- **线**剪辑**线**的左上/右下的像素。可以从两个点或一个点和一个角度初始化：
- **角度**使像素仅在给定的开始角度和结束角度之间
- **半径**将像素仅保留在可以具有半径的矩形内（也可以是圆形）。可以反转以将像素保持在矩形之外。
- 垂直**淡入**淡出（根据其 y 位置更改像素的不透明度）
- **map** 使用 alpha 蒙版（字节数组）描述像素的不透明度。

遮罩中的坐标是相对于对象的。也就是说，如果对象移动，则蒙版也会随之移动。

有关 mask *init* 函数的详细信息，请参见下面的 API 文档。

31.3.2 更新遮罩

可以使用来更新现有的掩码，其中的返回值为。

`lv_objmask_update_mask(objmask, mask_p, new_param)mask_plv_objmask_add_mask`

31.3.3 取下面膜

口罩可以用 `lv_objmask_remove_mask(objmask, mask_p)`

31.4 事件

仅通用事件是按对象类型发送的。

31.5 按键

对象类型不处理任何*键*。

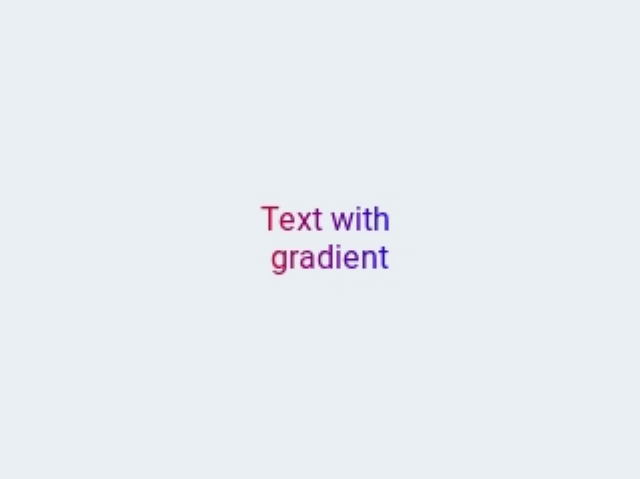
例

C

几个对象蒙版

码

文字遮罩



API

typedef

typedef uint8_t lv_objmask_part_t

枚举

enum [anonymous]

值：

enumerator LV_OBJMASK_PART_MAIN

功能

lv_obj_t * lv_objmask_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建对象遮罩对象

返回

指向创建的对象掩码的指针

参数

- par：指向对象的指针，它将是新对象蒙版的父对象
- copy：指向对象掩码对象的指针，如果不为 NULL，则将从其复制新对象

lv_objmask_mask_t * lv_objmask_add_mask（lv_obj_t * *objmask*，void * *param*）

添加面膜

返回

指向添加的蒙版的指针

参数

- `objmask`：指向对象遮罩对象的指针
- `param`：初始化的 mask 参数

void lv_objmask_update_mask（lv_obj_t * *objmask*，lv_objmask_mask_t * *mask*，void * *param* ）

更新已创建的蒙版

参数

- `objmask`：指向对象遮罩对象的指针
- `mask`：指向创建的蒙版的指针（由返回 `lv_objmask_add_mask`）
- `param`：初始化的 mask 参数（由初始化 `lv_draw_mask_line/angle/.../_init`）

voidlv_objmask_remove_mask（lv_obj_t * *objmask*，lv_objmask_mask_t * *mask* ）

取下口罩

参数

- `objmask`：指向对象遮罩对象的指针
- `mask`：指向创建的遮罩的指针（由返回 `lv_objmask_add_mask`）如果 `NULL` 通过，则将删除所有遮罩。

struct lv_objmask_mask_t

公众成员

void*param

struct lv_objmask_ext_t

公众成员

lv_cont_ext_tcont

lv_ll_t mask_ll

typedef

typedef uint8_t lv_draw_mask_res_t

typedef uint8_t lv_draw_mask_type_t

typedef lv_draw_mask_res_t（* lv_draw_mask_xcb_t）（ lv_opa_t * mask_buf，lv_coord_t abs_x，lv_coord_t abs_y，lv_coord_t len，void * p ）

每种掩码类型的通用回调类型。由库内部使用。

typedef uint8_t lv_draw_mask_line_side_t

typedef struct _lv_draw_mask_map_param_tlv_draw_mask_map_param_t

typedef _lv_draw_mask_saved_t lv_draw_mask_saved_arr_t [_LV_MASK_MAX_NUM]

枚举

enum [anonymous]

值：

enumerator LV_DRAW_MASK_RES_TRANSP

enumerator LV_DRAW_MASK_RES_FULL_COVER

enumerator LV_DRAW_MASK_RES_CHANGED

enumerator LV_DRAW_MASK_RES_UNKNOWN

enum [anonymous]

值：

enumerator LV_DRAW_MASK_TYPE_LINE

enumerator LV_DRAW_MASK_TYPE_ANGLE

enumerator LV_DRAW_MASK_TYPE_RADIUS

enumerator LV_DRAW_MASK_TYPE_FADE

enumerator LV_DRAW_MASK_TYPE_MAP

enum [anonymous]

值：

enumerator LV_DRAW_MASK_LINE_SIDE_LEFT = 0

enumerator LV_DRAW_MASK_LINE_SIDE_RIGHT

enumerator LV_DRAW_MASK_LINE_SIDE_TOP

enumerator LV_DRAW_MASK_LINE_SIDE_BOTTOM

功能

int16_t lv_draw_mask_add（ void * *param*，void * *custom_id* ）

添加一个绘图蒙版。涂完之后（直到取下面罩）之后绘制的所有内容都会受到面罩的影响。

返回

一个整数，即掩码的 ID。可用于中 `lv_draw_mask_remove_id`。

参数

- `param`：初始化的 mask 参数。仅保存指针。
- `custom_id`：用于标识掩码的自定义指针。用于中 `lv_draw_mask_remove_custom`。

void* lv_draw_mask_remove_id（ int16_t *id* ）

删除具有给定 ID 的面罩

返回

移除的遮罩的参数。如果更多的遮罩具有 `custom_id` ID，则将返回最后一个遮罩的参数

参数

- `id`：掩码的 ID。归还者 `lv_draw_mask_add`

---

void* lv_draw_mask_remove_custom（void* *custom_id* ）

删除所有具有给定自定义 ID 的蒙版

返回

返回删除的遮罩的参数。如果更多的遮罩具有 `custom_id` ID，则将返回最后一个遮罩的参数

参数

- `custom_id`：用于 `lv_draw_mask_add`

---

voidlv_draw_mask_line_points_init（lv_draw_mask_line_param_t * *param*，lv_coord_t *p1x*，lv_coord_t *p1y*，lv_coord_t *p2x*，lv_coord_t *p2y*，lv_draw_mask_line_side_t *side*）

从两点初始化线罩。

参数

- `param`：指向 `lv_draw_mask_param_t` 初始化的指针
- `p1x`：直线第一个点的 X 坐标
- `p1y`：直线第一个点的 Y 坐标
- `p2x`：直线第二点的 X 坐标
- `p2y`：直线第二点的 y 坐标
- `side`：和 `lv_draw_mask_line_side_t` 描述要保留哪一边的元素。随着

`LV_DRAW_MASK_LINE_SIDE_LEFT/RIGHT` 与水平线的所有像素都保持着

`LV_DRAW_MASK_LINE_SIDE_TOP/BOTTOM` 和垂直线的所有像素都保持

---

voidlv_draw_mask_line_angle_init（lv_draw_mask_line_param_t * *param*，lv_coord_t *p1x*，lv_coord_t *py*，int16_t *angle*，lv_draw_mask_line_side_t *side*）

从一个点和一个角度初始化线罩。

参数

- `param`：指向 `lv_draw_mask_param_t` 初始化的指针
- `px`：线点的 X 坐标
- `py`：线点的 X 坐标
- `angle`：右 0 度，底部：90
- `side`：和 `lv_draw_mask_line_side_t` 描述要保留哪一边的元素。随着

`LV_DRAW_MASK_LINE_SIDE_LEFT/RIGHT` 与水平线的所有像素都保持着

`LV_DRAW_MASK_LINE_SIDE_TOP/BOTTOM` 和垂直线的所有像素都保持

void lv_draw_mask_angle_init（lv_draw_mask_angle_param_t * *param*，lv_coord_t *vertex_x*，lv_coord_t *vertex_y*，lv_coord_t *start_angle*，lv_coord_t *end_angle* ）

初始化一个角度遮罩。

参数

- `param`：指向 `lv_draw_mask_param_t` 初始化的指针
- `vertex_x`：角顶点的 X 坐标（绝对坐标）
- `vertex_y`：角顶点的 Y 坐标（绝对坐标）
- `start_angle`：起始角度（度）。右侧 0 度，底部 90 度
- `end_angle`：结束角度

void lv_draw_mask_radius_init（lv_draw_mask_radius_param_t * *param*，const lv_area_t * *rect*，lv_coord_t *radius*，bool *inv* ）

初始化淡入淡出蒙版。

参数

- `param`：指向 a 的参数指针 `lv_draw_mask_param_t` 进行初始化
- `rect`：要影响的矩形的坐标（绝对坐标）
- `radius`：矩形的半径
- `inv`：true：将像素保留在矩形内；将像素保持在矩形之外

void lv_draw_mask_fade_init（lv_draw_mask_fade_param_t * *PARAM*，const lv_area_t * *coords*，lv_opa_t *opa_top*，lv_coord_t *y_top*，lv_opa_t *opa_bottom*，lv_coord_t *y_bottom* ）

初始化淡入淡出蒙版。

参数

- `param`：指向 `lv_draw_mask_param_t` 初始化的指针
- `coords`：要影响的区域的坐标（绝对坐标）
- `opa_top`：顶部不透明
- `y_top`：从哪个坐标开始变为不透明 `opa_bottom`
- `opa_bottom`：底部的不透明度
- `y_bottom`：到达哪个坐标 `opa_bottom`。

void lv_draw_mask_map_init(lv_draw_mask_map_param_t * param, const lv_area_t * coords, const lv_opa_t * map)

初始化地图遮罩。

参数

- `param`：指向 `lv_draw_mask_param_t` 初始化的指针
- `coords`：地图坐标（绝对坐标）
- `map`：带有掩码值的字节数组

struct lv_draw_mask_common_dsc_t

公众成员

lv_draw_mask_xcb_t cb
lv_draw_mask_type_t type

struct lv_draw_mask_line_param_t

公众成员

lv_draw_mask_common_dsc_t dsc
lv_point_t p1
lv_point_t p2
lv_draw_mask_line_side_t side
struct lv_draw_mask_line_param_t :: [anonymous] cfg
lv_point_t origo
int32_t xy_steep
int32_t yx_steep
int32_t steep

int32_t spx

uint8_t flat

uint8_t inv

struct lv_draw_mask_angle_param_t

公众成员

lv_draw_mask_common_dsc_t dsc

lv_point_t vertex_p

lv_coord_t start_angle

lv_coord_t end_angle

struct lv_draw_mask_angle_param_t :: [anonymous] cfg

lv_draw_mask_line_param_t start_line

lv_draw_mask_line_param_t end_line

uint16_t delta_deg

struct lv_draw_mask_radius_param_t

公众成员

lv_draw_mask_common_dsc_t dsc

lv_area_t rect

lv_coord_t radius

uint8_t outer

struct lv_draw_mask_radius_param_t :: [anonymous] cfg

int32_t y_prev

lv_sqrt_res_t y_prev_x

struct lv_draw_mask_fade_param_t

公众成员

lv_draw_mask_common_dsc_t dsc

lv_area_t coords

lv_coord_t y_top

lv_coord_t y_bottom

lv_opa_t opa_top

lv_opa_t opa_bottom

struct lv_draw_mask_fade_param_t :: [anonymous] cfg

struct _lv_draw_mask_map_param_t

公众成员

lv_draw_mask_common_dsc_t dsc

lv_area_t coords

const lv_opa_t *map

struct _lv_draw_mask_map_param_t :: [anonymous]cfg

struct _lv_draw_mask_saved_t

公众成员

void*param

void*custom_id

第三十二章 滚子（lv_roller）

32.1 总览

Roller 允许您通过滚动简单地从多个选项中选择一个选项。

32.2 小部件和样式

压路机的主要部分称为 `LV_ROLLER_PART_BG`。它是一个矩形，并使用所有典型的背景属性。Roller 标签的样式继承自背景的*文本*样式属性。要调整选项之间的*间距，请*使用 *text_line_space* 样式属性。该*填充*样式属性设置两侧的空间。

中间的选定选项可通过 `LV_ROLLER_PART_SELECTED` 虚拟小部件引用。除了典型的背景属性外，它还使用*文本*属性来更改所选区域中文本的外观。

32.3 用法

32.3.1 设定选项 Cai Xuefeng

这些选项作为字符串通过传递给 Roller 。选项之间应用分隔。例如：。

```
lv_roller_set_options(roller, options, LV_ROLLER_MODE_NORMAL/INFINITE)\n"First\nSecond\nThird"
```

`LV_ROLLER_MODE_INIFINITE` 使滚筒呈圆形。

您可以使用手动选择一个选项，其中 *id* 是选项的索引。

```
lv_roller_set_selected(roller, id, LV_ANIM_ON/OFF)
```

32.3.2 获取选择的选项

获取当前使用的选项 `lv_roller_get_selected(roller)` 将返回所选选项的*索引*。

`lv_roller_get_selected_str(roller, buf, buf_size)` 将所选选项的名称复制到 `buf`。

32.3.3 对齐选项

要水平对齐标签，请使用。`lv_roller_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`

32.3.4 可见行

可见行数可以通过 `lv_roller_set_visible_row_count(roller, num)`

32.3.5 动画时间

当滚轴滚动且未完全停在某个选项上时，它将自动滚动到最近的有效选项。滚动动画的时间可以通过更改。动画时间为零表示没有动画。`lv_roller_set_anim_time(roller, anim_time)`

32.4 事件

此外，通用事件以及以下特殊事件是通过下拉列表发送的：

- 选择新选项时发送 **LV_EVENT_VALUE_CHANGED**

32.5 按键

以下*按键*由按钮处理：

- **LV_KEY_RIGHT / DOWN** 选择下一个选项
- **LV_KEY_LEFT / UP** 选择上一个选项
- **LY_KEY_ENTER** 应用选定的选项（发送 `LV_EVENT_VALUE_CHANGED` 事件）

例

C

简易滚筒

API

typedef

typedef uint8_t **lv_roller_mode_t**

typedef uint8_t **lv_roller_part_t**

枚举

enum [anonymous]

滚轮模式。 Cai Xuefeng

值:

enumerator **LV_ROLLER_MODE_NORMAL**

普通模式（滚轴在选项末尾结束）。

enumerator **LV_ROLLER_MODE_INIFINITE**

无限模式（滚子可以永远滚动）。

enum [anonymous]

值:

enumerator **LV_ROLLER_PART_BG**= LV_PAGE_PART_BG

enumerator **LV_ROLLER_PART_SELECTED**= _LV_PAGE_PART_VIRTUAL_LAST

enumerator **_LV_ROLLER_PART_VIRTUAL_LAST**

功能

lv_obj_t * **lv_roller_create**（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个滚子对象

返回

指向创建的滚子的指针

参数

- `par`：指向对象的指针，它将是新滚子的父对象
- `copy`：指向滚子对象的指针，如果不为 NULL，则将从其复制新对象

void lv_roller_set_options（lv_obj_t * *roller*，const char * *options*，lv_roller_mode_t *mode*）

在滚筒上设置选项

参数

- `roller`：指向滚轮对象的指针
- `options`：带有'

'分开的选项。例如“一个\ n 两个\ n 三个”

- `mode`：`LV_ROLLER_MODE_NORMAL` 或 `LV_ROLLER_MODE_INFINITE`

void lv_roller_set_align（lv_obj_t * *roller*，lv_label_align_t *align*）

设置滚筒选项的对齐方式（左，右或中心[默认]）

参数

- `roller`：-指向滚轮对象的指针
- `align`：-lv_label_align_t 值之一（左，右，中心）

void lv_roller_set_selected（lv_obj_t * *roller*，uint16_t *sel_opt*，lv_anim_enable_t *anim* ）

设置所选选项

参数

- `roller`：指向滚轮对象的指针
- `sel_opt`：所选选项的编号（0 ...选项编号-1）；
- `anim`：LV_ANOM_ON：设置动画；LV_ANIM_OFF 立即设置

void lv_roller_set_visible_row_count（lv_obj_t * *roller*，uint8_t *row_cnt* ）

设置高度以显示给定的行数（选项）

参数

- `roller`：指向滚轮对象的指针

- `row_cnt`：所需的可见行数

void lv_roller_set_auto_fit（lv_obj_t * roller，bool auto_fit）

允许根据其内容自动设置滚筒宽度。

参数

- `roller`：指向滚轮对象的指针
- `auto_fit`：true：启用自动调整

void lv_roller_set_anim_time（lv_obj_t * roller，uint16_t anim_time）

设置打开/关闭动画时间。

参数

- `roller`：指向滚轮对象的指针
- `anim_time`：打开/关闭动画时间[ms]

uint16_t lv_roller_get_selected（const lv_obj_t * roller）

获取所选选项的 ID

返回

所选选项的 ID（0 ...选项编号-1）；

参数

- `roller`：指向滚轮对象的指针

uint16_t lv_roller_get_option_cnt（const lv_obj_t * roller）

获取选项总数

返回

列表中的选项总数

参数

- `roller`：指向滚轮对象的指针

void lv_roller_get_selected_str（const lv_obj_t * roller，char * buf，uint32_t buf_size）

获取当前选择的选项作为字符串

参数

- `roller`：指向滚轮对象的指针

- `buf`：指向存储字符串的数组的指针
- `buf_size`：`buf`以字节为单位的大小。0：忽略它。

lv_label_align_t lv_roller_get_align（const lv_obj_t * roller）

获取 align 属性。_create 之后的默认对齐方式是 LV_LABEL_ALIGN_CENTER

返回

LV_LABEL_ALIGN_LEFT，LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER

参数

- `roller`：指向滚轮对象的指针

bool lv_roller_get_auto_fit（lv_obj_t * roller）

获取是否启用了自动调整选项。

返回

true：启用自动调整

参数

- `roller`：指向滚轮对象的指针

const 字符* lv_roller_get_options（const lv_obj_t * roller）

获取压路机的选项

返回

以'分隔的选项

'-s（例如“ Option1 \ nOption2 \ nOption3”）

参数

- `roller`：指向滚轮对象的指针

uint16_t lv_roller_get_anim_time（const lv_obj_t * roller）

获取打开/关闭动画时间。

返回

打开/关闭动画时间[ms]

参数

- `roller`：滚轮指针

struct lv_roller_ext_t

公众成员

lv_page_ext_t page

lv_style_list_t style_sel

uint16_t option_cnt

uint16_t sel_opt_id

uint16_t sel_opt_id_ori

lv_roller_mode_t mode

uint8_t auto_fit

第三十三章滑块（lv_slider）

33.1 总览

Slider 对象看起来像一个带有旋钮的 Bar。可以拖动旋钮以设置一个值。滑块也可以是垂直或水平的。

33.2 小部件和样式

滑块的主要部分被称为 `LV_SLIDER_PART_BG`，它使用典型的背景样式属性。

`LV_SLIDER_PART_INDIC` 是一个虚拟部件，还使用了所有典型的背景属性。默认情况下，指标的最大大小与背景的大小相同，但是在其中设置正的填充值 `LV_SLIDER_PART_BG` 将使指标变小。（负值会使它变大）如果在指标上使用了*值*样式属性，则将根据指标的当前大小来计算对齐方式。例如，中心对齐的值始终显示在指示器的中间，而不管其当前大小如何。

`LV_SLIDER_PART_KNOB` 是一个虚拟部件，使用所有典型的背景属性来描述旋钮。类似于*指示器*的*值*的文本也对准旋钮的当前位置和尺寸。默认情况下，旋钮是正方形（带有半径），其边长等于滑块的较小边。可以使用*填充*值使旋钮变大。填充值也可以是不对称的。

33.3 用法

33.3.1 值和范围

要设置初始值，请使用。 设置动画时间（以毫秒为单位）。

```
lv_slider_set_value(slider, new_value, LV_ANIM_ON/OFF)lv_slider_set_anim_time(slider, anim_time)
```

要指定范围（最小，最大值），可以使用。`lv_slider_set_range(slider, min , max)`

33.3.2 对称范围

除了普通类型外，滑块还可以配置为两种其他类型：

- `LV_SLIDER_TYPE_NORMAL` 普通型

- `LV_SLIDER_TYPE_SYMMETRICAL` 将指标对称地绘制为零（从零开始，从左到右）
- `LV_SLIDER_TYPE_RANGE` 允许为左（起始）值使用附加旋钮。（可用于 `lv_slider_set/get_left_value()`）

类型可以用 `lv_slider_set_type(slider, LV_SLIDER_TYPE_...)`

33.3.3 仅旋钮模式

通常，可以通过拖动旋钮或单击滑块来调整滑块。在后一种情况下，旋钮移动到所单击的点，并且滑块值相应地变化。在某些情况下，希望将滑块设置为仅在拖动旋钮时做出反应。

通过调用启用此功能。`lv_obj_set_adv_hittest(slider, true);`

33.4 事件

除了通用事件之外，滑块还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 在使用键拖动或更改滑块时发送。拖动滑块（仅在释放滑块时）会连续发送事件。使用 `lv_slider_is_dragged` 以决定是否为滑块被拖动或刚刚发布。

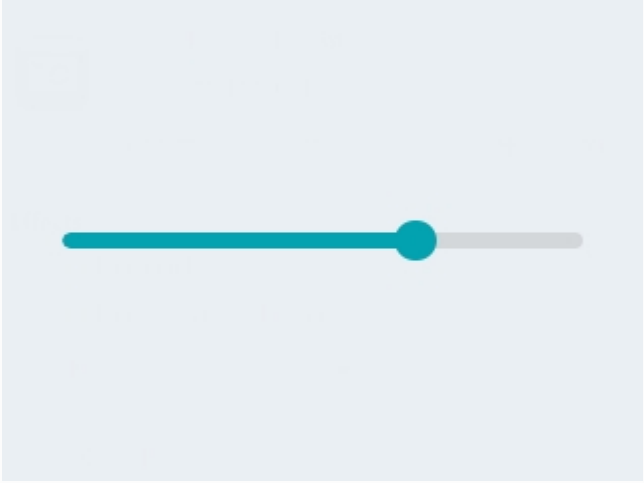
33.5 按键

- **LV_KEY_UP**，**LV_KEY_RIGHT** 将滑块的值增加 1
- **LV_KEY_DOWN**，**LV_KEY_LEFT** 将滑块的值减 1

例

C

带有 custo mstyle 的滑块



码

用滑块设定值

API

typedef

typedef uint8_t `lv_slider_type_t`

枚举

enum [anonymous]

值:

enumerator `LV_SLIDER_TYPE_NORMAL`

enumerator `LV_SLIDER_TYPE_SYMMETRICAL`

enumerator `LV_SLIDER_TYPE_RANGE`

enum [anonymous]

内置样式的滑块

值：

enumerator **LV_SLIDER_PART_BG**

enumerator **LV_SLIDER_PART_INDIC**

滑块背景样式。

enumerator **LV_SLIDER_PART_KNOB**

滑块指示器（填充区域）样式。

功能

lv_obj_t * lv_slider_create（lv_obj_t *　par，const lv_obj_t *　copy）

创建一个滑块对象

返回

指向创建的滑块的指针

参数

- `par`：指向对象的指针，它将是新滑块的父对象
- `copy`：指向滑块对象的指针，如果不为 NULL，则将从其复制新对象

void lv_slider_set_value（lv_obj_t *　*slider*，int16_t　*value*，lv_anim_enable_t *anim* ）

在滑块上设置一个新值

参数

- `slider`：指向滑块对象的指针
- `value`：新价值
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_slider_set_left_value（lv_obj_t *　*slider*，int16_t *left_value*，lv_anim_enable_t *anim* ）

为滑块的左旋钮设置一个新值

参数

- `slider`：指向滑块对象的指针
- `left_value`：新价值
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_slider_set_range（lv_obj_t * slider，int16_t min，int16_t max ）

设置条的最小值和最大值

参数

- `slider`：指向滑块对象的指针
- `min`：最小值
- `max`：最大值

void lv_slider_set_anim_time（lv_obj_t * slider，uint16_t anim_time ）

设置滑块的动画时间

参数

- `slider`：指向条对象的指针
- `anim_time`：动画时间（以毫秒为单位）。

void lv_slider_set_type（lv_obj_t * slider，lv_slider_type_t type）

使滑块对称为零。指标将从零开始增长，而不是最小位置。

参数

- `slider`：指向滑块对象的指针
- `en`：true：启用禁用对称行为；false：禁用

int16_t lv_slider_get_value（const lv_obj_t * slider）

获取滑块主旋钮的值

返回

滑块主旋钮的值

参数

- `slider`：指向滑块对象的指针

int16_t lv_slider_get_left_value（const lv_obj_t * slider）

获取滑块左旋钮的值

返回

滑块左旋钮的值

参数

- `slider`：指向滑块对象的指针

`int16_t lv_slider_get_min_value（const lv_obj_t * slider）`

获取滑块的最小值

返回

滑块的最小值

参数

- `slider`：指向滑块对象的指针

`int16_t lv_slider_get_max_value（const lv_obj_t * slider）`

获取滑块的最大值

返回

滑块的最大值

参数

- `slider`：指向滑块对象的指针

`bool lv_slider_is_dragged（const lv_obj_t *  slider）`

给出滑块是否被拖动

返回

真：拖动进行中 false：不拖动

参数

- `slider`：指向滑块对象的指针

`uint16_t lv_slider_get_anim_time（lv_obj_t * slider）`

获取滑块的动画时间

返回

动画时间（以毫秒为单位）。

参数

- `slider`：指向滑块对象的指针

`lv_slider_type_t lv_slider_get_type（lv_obj_t *  slider）`

获取滑块是否对称。

返回

true：启用对称；false：禁用

参数

- `slider`：指向条对象的指针

struct **lv_slider_ext_t**

公众成员

lv_bar_ext_t **bar**
lv_style_list_t **style_knob**
lv_area_t **left_knob_area**
lv_area_t **right_knob_area**
int16_t ***value_to_set**
uint8_t **dragging**

第三十四章 Spinbox（lv_spinbox）

34.1 总览

Spinbox 包含一个数字文本，可通过*按键*或 API 函数增加或减少数字。Spinbox 的下面是修改后的 Text 区域。

34.2 小部件和样式

Spinbox 的主要部分被称为 `LV_SPINBOX_PART_BG` 使用所有典型背景样式属性的矩形背景。它还使用其*文本*样式属性描述标签的样式。

`LV_SPINBOX_PART_CURSOR` 是描述光标的虚拟部分。阅读文本区域文档以获取详细说明。

34.2.1 设定格式

`lv_spinbox_set_digit_format(spinbox, digit_count, separator_position)` 设置数字的格式。

`digit_count` 设置位数。前导零被添加以填充左侧的空间。`separator_position` 设置小数点前的位数。`0` 表示没有小数点。

`lv_spinbox_set_padding_left(spinbox, cnt)cnt` 在 sign 之间最左边的数字之间添加“空格”字符。

34.2.2 值和范围

`lv_spinbox_set_range(spinbox, min, max)` 设置 Spinbox 的范围。

`lv_spinbox_set_value(spinbox, num)` 手动设置 Spinbox 的值。

`lv_spinbox_increment(spinbox)` 并 `lv_spinbox_decrement(spinbox)` 增加/减少 Spinbox 的值。

`lv_spinbox_set_step(spinbox, step)` 设置增量减量。

34.3 事件

除通用事件外，下拉列表还发送以下特殊事件：

- 值更改时发送 **LV_EVENT_VALUE_CHANGED**。（该值设置为的事件数据 `int32_t`）
- **LV_EVENT_INSERT** 由祖先文本区域发送，但不应使用。

34.4 按键

以下*按键*由按钮处理：

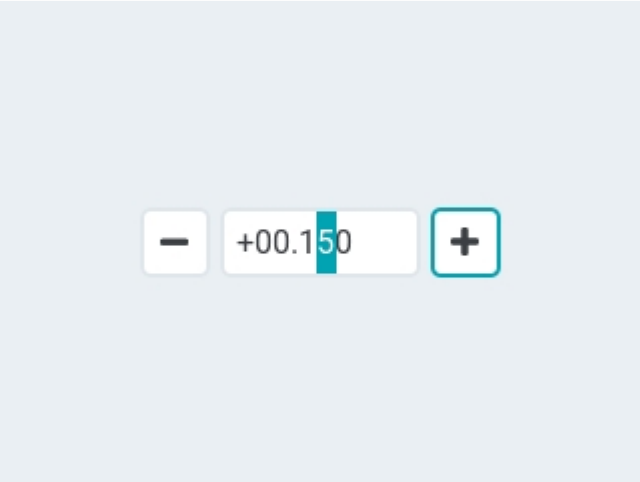
- **LV_KEY_LEFT / RIGHT** 使用*键盘*向左/向右移动光标。使用*编码器*递减/递增所选数字。
- **LY_KEY_ENTER** 应用选定的选项（发送 `LV_EVENT_VALUE_CHANGED` 事件并关闭下拉列表）
- **LV_KEY_ENTER** 使用*编码器*获得了净数字。跳到最后一个之后的第一个。

例

C

Cai Xuefeng

简单旋转框



API

typedef

```
typedef uint8_t lv_spinbox_part_t
```

枚举

enum [anonymous]

值：

enumerator **LV_SPINBOX_PART_BG**= LV_TEXTAREA_PART_BG

enumerator **LV_SPINBOX_PART_CURSOR**= LV_TEXTAREA_PART_CURSOR

enumerator **_LV_SPINBOX_PART_VIRTUAL_LAST**= _LV_TEXTAREA_PART_VIRTUAL_LAST

enumerator **_LV_SPINBOX_PART_REAL_LAST**= _LV_TEXTAREA_PART_REAL_LAST

功能

lv_obj_t * **lv_spinbox_create**（lv_obj_t * *par*，*const*lv_obj_t * *copy*）

创建一个 Spinbox 对象

返回

指向创建的旋转框的指针

参数

- `par`：指向对象的指针，它将是新旋转框的父对象
- `copy`：指向 Spinbox 对象的指针，如果不为 NULL，则将从其复制新对象

空隙 **lv_spinbox_set_rollover**（lv_obj_t * *spinbox*，bool*b* ）

设置旋转框翻转功能

参数

- `spinbox`：指向 Spinbox 的指针
- `b`：true 或 false 启用或禁用（默认）

空隙 **lv_spinbox_set_value**（lv_obj_t * *spinbox*，int32_t *i*）

设置旋转框值

参数

- `spinbox`：指向 Spinbox 的指针
- `i`：要设置的值

void**lv_spinbox_set_digit_format**（lv_obj_t * *spinbox*，uint8_t *DIGIT_COUNT*，uint8_t *separator_position* ）

设置 Spinbox 数字格式（数字计数和十进制格式）

参数

- `spinbox`：指向 Spinbox 的指针

- `digit_count`：不包括小数点分隔符和 sign 的位数
- `separator_position`：小数点前的位数。如果为 0，则不显示小数点

空隙 lv_spinbox_set_step（lv_obj_t * *spinbox*，uint32_t step）

设置旋转框步骤

参数

- `spinbox`：指向 Spinbox 的指针
- `step`：递增/递减的步骤

voidlv_spinbox_set_range（lv_obj_t * spinbox，int32_t *range_min*，int32_t *range_max* ）

设置旋转框值范围

参数

- `spinbox`：指向 Spinbox 的指针
- `range_min`：最大值（含）
- `range_max`：最小值（含）

空隙 lv_spinbox_set_padding_left（lv_obj_t * *spinbox*，uint8_t *cb*）

以数字计数设置 Spinbox 左填充（在 sign 和第一个数字之间添加）

参数

- `spinbox`：指向 Spinbox 的指针
- `cb`：在值更改事件上调用回调函数

boollv_spinbox_get_rollover（lv_obj_t * *spinbox*）

获取 Spinbox 翻转功能状态

参数

- `spinbox`：指向 Spinbox 的指针

int32_t lv_spinbox_get_value（lv_obj_t * *spinbox*）

获取 Spinbox 数字值（用户必须根据其数字格式转换为 float）

返回

数值框的整数值

参数

- `spinbox`：指向 Spinbox 的指针

void lv_spinbox_step_next（lv_obj_t * spinbox）

通过除以 10 选择下一个较低的数字进行编辑

参数

- `spinbox`：指向 Spinbox 的指针

void lv_spinbox_step_prev（lv_obj_t * spinbox）

通过将步数乘以 10 选择下一个较高的数字进行编辑

参数

- `spinbox`：指向 Spinbox 的指针

void lv_spinbox_increment（lv_obj_t * spinbox）

将 Spinbox 值增加一级

参数

- `spinbox`：指向 Spinbox 的指针

void lv_spinbox_decrement（lv_obj_t * spinbox）

将 Spinbox 值减 1

参数

- `spinbox`：指向 Spinbox 的指针

struct lv_spinbox_ext_t

公众成员

```
lv_textarea_ext_t ta
int32_t value
int32_t range_max
int32_t range_min
int32_t step
uint8_t rollover
uint16_t digit_count
uint16_t dec_point_pos
uint16_t digit_padding_left
```

第三十五章 微调器（lv_spinner）

35.1 总览

Spinner 对象是边界上的旋转弧。

35.2 小部件和样式

微调器使用以下部分：

- `LV_SPINNER_PART_BG`： 主要部分
- `LV_SPINNER_PART_INDIC`：旋转弧（虚拟部分）

小部件和样式的工作原理与 Arc 相同。阅读其文档以获取详细说明。

35.3 用法

35.3.1 弧长

Cai Xuefeng

圆弧的长度可以通过调节。`lv_spinner_set_arc_length(spinner, deg)`

35.3.2 纺丝速度

旋转速度可以通过调节。`lv_spinner_set_spin_time(preload, time_ms)`

35.3.3 旋转类型

您可以从更多旋转类型中进行选择：

- **LV_SPINNER_TYPE_SPINNING_ARC** 旋转弧线，在顶部放慢速度
- **LV_SPINNER_TYPE_FILLSPIN_ARC** 旋转弧线，在顶部放慢速度，但也可以拉伸弧线
- **LV_SPINNER_TYPE_CONSTANT_ARC** 以恒定速度旋转弧

如果他们使用一个应用 `lv_spinner_set_type(preload, LV_SPINNER_TYPE_...)`

35.3.4 旋转方向

旋转方向可以用改变。`lv_spinner_set_dir(preload, LV_SPINNER_DIR_FORWARD/BACKWARD)`

35.4 事件

仅通用事件是按对象类型发送的。

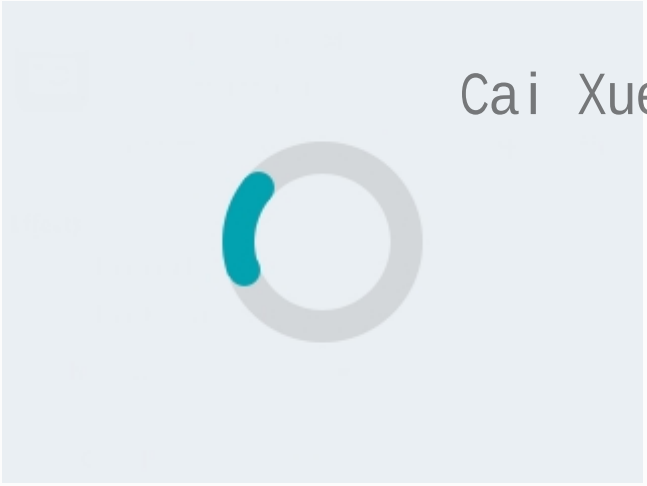
35.5 按键

对象类型不处理任何 *键*。

例

C

简单的微调器



API

typedef

```
typedef uint8_t lv_spinner_type_t
typedef uint8_t lv_spinner_dir_t
typedef uint8_t lv_spinner_style_t
```

枚举

```
enum [anonymous]
```

微调器的类型。

值：

enumerator LV_SPINNER_TYPE_SPINNING_ARC

enumerator LV_SPINNER_TYPE_FILLSPIN_ARC

enumerator LV_SPINNER_TYPE_CONSTANT_ARC

enum [anonymous]

旋转器应旋转的方向。

值：

enumerator LV_SPINNER_DIR_FORWARD

enumerator LV_SPINNER_DIR_BACKWARD

enum [anonymous]

值：

enumerator LV_SPINNER_PART_BG= LV_ARC_PART_BG

enumerator LV_SPINNER_PART_INDIC= LV_ARC_PART_INDIC

enumerator _LV_SPINNER_PART_VIRTUAL_LAST

enumerator _LV_SPINNER_PART_REAL_LAST= _LV_ARC_PART_REAL_LAST

功能

lv_obj_t * lv_spinner_create（lv_obj_t *　*par*，const lv_obj_t * *copy*）

创建微调器对象

返回

指向创建的微调器的指针

参数

- `par`：指向对象的指针，它将是新微调器的父对象
- `copy`：指向微调器对象的指针，如果不为 NULL，则将从其复制新对象

void lv_spinner_set_arc_length（lv_obj_t * *spinner*，lv_anim_value_t *deg* ）

设置旋转弧的长度（以度为单位）

参数

- `spinner`：指向微调器对象的指针
- `deg`：弧长

void lv_spinner_set_spin_time（lv_obj_t *　*spinner*，uint16_t　*time*）

设置弧的旋转时间

参数

- `spinner`：指向微调器对象的指针
- `time`：一轮时间（以毫秒为单位）

void **lv_spinner_set_type**（lv_obj_t * *spinner*，lv_spinner_type_t *type*）

设置微调器的动画类型。

参数

- `spinner`：指向微调器对象的指针
- `type`：微调器的动画类型

void **lv_spinner_set_dir**（lv_obj_t * *spinner*，lv_spinner_dir_t *direction* ）

设置微调器的动画方向

参数

- `spinner`：指向微调器对象的指针
- `direction`：微调器的动画方向

lv_anim_value_t **lv_spinner_get_arc_length**（const lv_obj_t * *spinner*）

获取微调器的弧长[度]

参数

- `spinner`：指向微调器对象的指针

uint16_t **lv_spinner_get_spin_time**（const lv_obj_t * *spinner*）

获取弧的旋转时间

参数

- `spinner`：指向微调框对象的指针[毫秒]

lv_spinner_type_t **lv_spinner_get_type**（lv_obj_t * *spinner* ）

获取微调器的动画类型。

返回

动画类型

参数

- `spinner`：指向微调器对象的指针

lv_spinner_dir_t lv_spinner_get_dir（lv_obj_t * *spinner* ）

获取微调器的动画方向

返回

动画方向

参数

- `spinner`：指向微调器对象的指针

void lv_spinner_anim_cb（ void * *ptr*，lv_anim_value_t *val* ）

Animator 函数（exec_cb）旋转微调器的弧线。

参数

- `ptr`：指向微调器的指针
- `val`：当前期望值[0..360]

struct lv_spinner_ext_t

公众成员

lv_arc_ext_t arc

lv_anim_value_t arc_length

uint16_t time

lv_spinner_type_t anim_type

lv_spinner_dir_t anim_dir

第三十六章 开关（lv_switch）

36.1 总览

开关可用于打开/关闭某物。它看起来像一个小滑块。

36.2 小部件和样式

交换机使用以下部分：

- `LV_SWITCH_PART_BG`： 主要部分
- `LV_SWITCH_PART_INDIC`：指标（虚拟部分）
- `LV_SWITCH_PART_KNOB`：旋钮（虚拟部分）

小部件和样式的工作原理与 Slider 相同。阅读其文档以获取详细说明。

＃＃用法

36.2.1 变更状态 Cai Xuefeng

的开关的状态可以通过点击它或者通过改变，或功能

`lv_switch_on(switch, LV_ANIM_ON/OFF)lv_switch_off(switch, LV_ANIM_ON/OFF)lv_switch_toggle(switch, LV_ANOM`

`_ON/OFF)`

36.2.2 动画时间

开关更改状态时的动画时间可以通过进行调整。`lv_switch_set_anim_time(switch, anim_time)`

36.3 事件

除通用事件外，交换机还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 当开关更改状态时发送。

36.4 按键

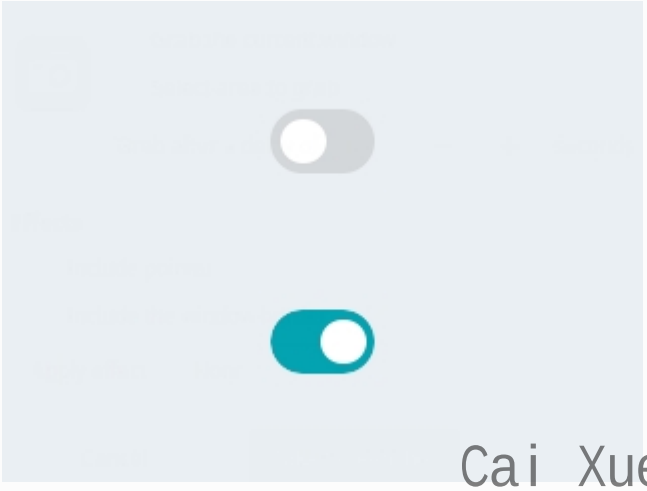
- **LV_KEY_UP**，**LV_KEY_RIGHT** 打开滑块
- **LV_KEY_DOWN**，**LV_KEY_LEFT** 关闭滑块

了解更多有关按键的信息。

例

C

简单开关



API

typedef

typedef uint8_t lv_switch_part_t

枚举

enum [anonymous]

开关小部件。

值:

enumerator LV_SWITCH_PART_BG= LV_BAR_PART_BG

切换背景。

enumerator LV_SWITCH_PART_INDIC= LV_BAR_PART_INDIC

切换填充区域。

enumerator LV_SWITCH_PART_KNOB= _LV_BAR_PART_VIRTUAL_LAST

开关旋钮。

enumerator **_LV_SWITCH_PART_VIRTUAL_LAST**

功能

lv_obj_t * lv_switch_create（lv_obj_t *par，const lv_obj_t * copy）

创建一个开关对象

返回

指向创建的开关的指针

参数

- `par`：指向对象的指针，它将是新开关的父对象
- `copy`：指向切换对象的指针，如果不为 NULL，则将从其复制新对象

void lv_switch_on（lv_obj_t * sw，lv_anim_enable_t anim ）

打开开关

参数

- `sw`：指向切换对象的指针
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_switch_off（lv_obj_t * sw，lv_anim_enable_t anim ）

关闭开关

参数

- `sw`：指向切换对象的指针
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

bool lv_switch_toggle（lv_obj_t * sw，lv_anim_enable_t anim ）

切换开关的位置

返回

开关的结果状态。

参数

- `sw`：指向切换对象的指针
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void **lv_switch_set_anim_time**（lv_obj_t * *sw*，uint16_t *anim_time* ）

设置开关的动画时间

返回

指向样式的样式指针

参数

- `sw`：指向切换对象的指针
- `anim_time`：动画时间

bool **lv_switch_get_state**（*const* lv_obj_t * *sw* ）

获取开关状态

返回

false：关闭；true：开启

参数

- `sw`：指向切换对象的指针

uint16_t **lv_switch_get_anim_time**（*const* lv_obj_t * *sw* ）

获取开关的动画时间

返回

指向样式的样式指针

参数

- `sw`：指向切换对象的指针

struct **lv_switch_ext_t**

公众成员

lv_bar_ext_t **bar**

lv_style_list_t **style_knob**

uint8_t **state**

第三十七章 表格（lv_table）

37.1 总览

像往常一样，表是从包含文本的行，列和单元格构建的。

由于仅存储文本，因此 Table 对象的权重非常轻。没有为单元创建任何实际对象，但它们是动态绘制的。

37.2 小部件和样式

该表的主要部分称为 `LV_TABLE_PART_BG`。它是一个类似于背景的矩形，并使用所有典型的背景样式属性。

对于单元，有 4 个虚拟部分。每个单元格都具有类型（1、2、3 或 4），该类型指示要在其上应用哪个部分的样式。单元格部分为：

- `LV_TABLE_PART_CELL1`
- `LV_TABLE_PART_CELL2`
- `LV_TABLE_PART_CELL3`
- `LV_TABLE_PART_CELL4`

单元格还使用所有典型的背景样式属性。如果`\n`单元格内容中有一个换行符（），则在换行符之后将使用*线条*样式属性绘制一条水平分隔线。

单元格中的文本样式是从单元格部分或背景部分继承的。

37.3 用法

37.3.1 行和列

要设置行数和列数，请使用和

`lv_table_set_row_cnt(table, row_cnt)lv_table_set_col_cnt(table, col_cnt)`

37.3.2 宽度和高度

列的宽度可以用设置。Table 对象的总宽度将设置为列宽的总和。

```
lv_table_set_col_width(table, col_id, width)
```

高度是根据单元格样式（字体，填充等）和行数自动计算得出的。

37.3.3 设定储存格值

单元格只能存储文本，因此在将数字显示在表格中之前，需要将数字转换为文本。

`lv_table_set_cell_value(table, row, col, "Content")`。文本由表保存，因此它甚至可以是局部变量。

可以在。之类的文本中使用换行符`"Value\n60.3"`。

37.3.4 对齐

单元格中的文本对齐方式可以使用进行单独调整。

```
lv_table_set_cell_align(table, row, col, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)
```


37.3.5 电池类型

您可以使用 4 种不同的单元格类型。每个都有自己的风格。

单元格类型可用于添加不同的样式，例如：

- 表头
- 第一栏
- 突出显示一个单元格
- 等等

可以选择的类型可以是 1、2、3 或 4。`lv_table_set_cell_type(table, row, col, type)` `type`

37.3.6 合并单元格

单元格可以与合并。要合并更多相邻的单元格，请对每个单元格应用此功能。

```
lv_table_set_cell_merge_right(table, col, row, true)
```

37.3.7 裁剪文字

默认情况下，文字会自动换行以适合单元格的宽度，并且单元格的高度会自动设置。要禁用此功能并保持启用状态不变。`lv_table_set_cell_crop(table, row, col, true)`

37.3.8 滚动

使表格可滚动放置在页面上

37.4 事件

仅通用事件是按对象类型发送的。

37.5 按键

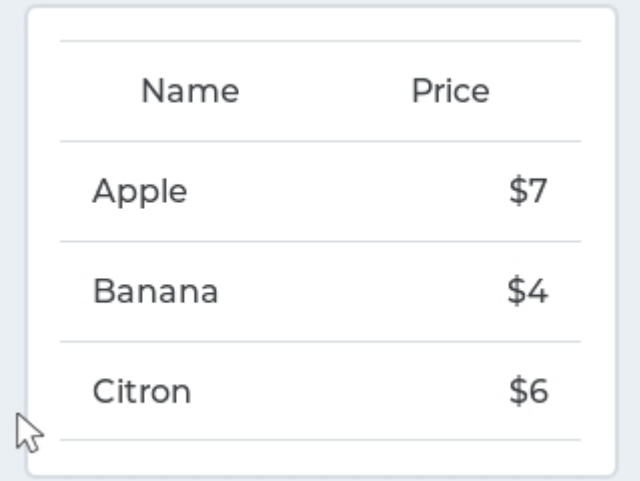
对象类型不处理任何*键*。

例

Cai Xuefeng

C

简单表



API

枚举

enum [anonymous]

值：

enumerator **LV_TABLE_PART_BG**

enumerator **LV_TABLE_PART_CELL1**

enumerator **LV_TABLE_PART_CELL2**

enumerator **LV_TABLE_PART_CELL3**

enumerator **LV_TABLE_PART_CELL4**

功能

lv_obj_t * **lv_table_create**（lv_obj_t * *par*，*const*lv_obj_t * *复制*）

创建一个表对象

返回

指向已创建表的指针

参数

- `par`：指向对象的指针，它将是新表的父对象
- `copy`：指向表对象的指针，如果不为 NULL，则将从其复制新对象

voidlv_table_set_cell_value（lv_obj_t * *table*，uint16_t *row*，uint16_t *col*，*const* char* *txt* ）

设置单元格的值。

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID
- `txt`：要在单元格中显示的文本。它将被复制并保存，因此在此函数调用之后不需要此变量。

voidlv_table_set_row_cnt（lv_obj_t * *table*，uint16_t *row_cnt* ）

设置行数

参数

- `table`：指向 Table 对象的表指针
- `row_cnt`：行数

void lv_table_set_col_cnt（lv_obj_t * table，uint16_t col_cnt）

设置列数

参数

- `table`：指向 Table 对象的表指针
- `col_cnt`： 列数。必须为<LV_TABLE_COL_MAX

void lv_table_set_col_width（lv_obj_t * table，uint16_t col_id，lv_coord_t w）

设置列宽

参数

- `table`：指向 Table 对象的表指针
- `col_id`：列[0 .. LV_TABLE_COL_MAX -1]的 ID
- `w`：列的宽度

void lv_table_set_cell_align（lv_obj_t * table，uint16_t row，uint16_t col，lv_label_align_t align）

设置文本在单元格中的对齐方式

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID
- `align`：LV_LABEL_ALIGN_LEFT 或 LV_LABEL_ALIGN_CENTER 或 LV_LABEL_ALIGN_RIGHT

void lv_table_set_cell_type（lv_obj_t * table，uint16_t row，uint16_t col，uint8_t type）

设置单元格的类型。

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID
- `type`：1,2,3 或 4。将相应选择单元格样式。

void **lv_table_set_cell_crop**（lv_obj_t * *table*，uint16_t *row*，uint16_t *col*，bool *crop*）

设置细胞裁剪。（请勿根据其内容调整单元格的高度）

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID
- `crop`：true：裁剪细胞含量；false：将单元格高度设置为内容。

void **lv_table_set_cell_merge_right**（lv_obj_t * *table*，uint16_t *row*，uint16_t *col*，bool *en* ）

与正确的邻居合并一个单元格。右边的单元格的值将不会显示。

参数

- `table`：指向 Table 对象的表指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID
- `en`：true：合并权；false：不正确合并

const char * **lv_table_get_cell_value**（lv_obj_t * *table*，uint16_t *row*，uint16_t *col* ）

获取单元格的值。

返回

单元格中的文字

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID

uint16_t **lv_table_get_row_cnt**（lv_obj_t * *table*）

获取行数。

返回

行数。

参数

- `table`：指向 Table 对象的表指针

---

uint16_t **lv_table_get_col_cnt**（lv_obj_t * *table*）

获取列数。

返回

列数。

参数

- `table`：指向 Table 对象的表指针

---

lv_coord_t **lv_table_get_col_width**（lv_obj_t * *table*，uint16_t *col_id* ）

获取列的宽度

返回

列宽

参数

- `table`：指向 Table 对象的表指针
- `col_id`：列[0 .. LV_TABLE_COL_MAX -1]的 ID

---

lv_label_align_t **lv_table_get_cell_align**（lv_obj_t * *table*，uint16_t *row*，uint16_t *col* ）

获取单元格的文本对齐方式

返回

LV_LABEL_ALIGN_LEFT（错误时默认）或 LV_LABEL_ALIGN_CENTER 或 LV_LABEL_ALIGN_RIGHT

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID

---

lv_label_align_t **lv_table_get_cell_type**（lv_obj_t * *table*，uint16_t *row*，uint16_t *col* ）

获取单元格的类型

返回

1,2,3 或 4

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID

lv_label_align_t lv_table_get_cell_crop（lv_obj_t * *table*，uint16_t *row*，uint16_t *col* ）

获取单元格的裁剪属性

返回

true：启用文本裁剪；false：禁用

参数

- `table`：指向 Table 对象的指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID

boollv_table_get_cell_merge_right（lv_obj_t * *table*，uint16_t *row*，uint16_t *col* ）

获取单元格合并属性。

返回

true：合并权；false：不正确合并

参数

- `table`：指向 Table 对象的表指针
- `row`：第[0 .. row_cnt -1]行的 ID
- `col`：列[0 .. col_cnt -1]的 ID

lv_res_t lv_table_get_pressed_cell（lv_obj_t * *table*，uint16_t * *row*，uint16_t * *col* ）

获取最后按下或被按下的单元格

返回

LV_RES_OK：找到一个有效的已按下的单元格，LV_RES_INV：未按下任何有效的单元格

参数

- `table`：指向表对象的指针
- `row`：指向变量以存储按下的行的指针

- `col`：指向存储按下的列的变量的指针

union lv_table_cell_format_t

#include <lv_table.h>

内部表格单元格格式 struct 。

请改用 `lv_table` API。

公众成员

uint8_t **align**

uint8_t **right_merge**

uint8_t **type**

uint8_t **crop**

struct lv_table_cell_format_t :: **[anonymous]s**

uint8_t **format_byte**

struct lv_table_ext_t

公众成员

uint16_t **col_cnt**

uint16_t **row_cnt**

char****cell_data**

lv_coord_t ***row_h**

lv_style_list_t **cell_style[LV_TABLE_CELL_STYLE_CNT]**

lv_coord_t **col_w[LV_TABLE_COL_MAX]**

uint8_t **cell_types**

第三十八章 Tabview（lv_tabview）

38.1 总览

选项卡视图对象可用于组织选项卡中的内容。

38.2 小部件和样式

Tab 视图对象包含几个部分。主要是 `LV_TABVIEW_PART_BG`。它是一个矩形容器，用于容纳 Tab 视图的其他部分。

在背景上创建了两个重要的真实部分：

- `LV_TABVIEW_PART_BG_SCRL`：这是 Page 的可滚动部分。它使选项卡的内容彼此相邻。页面的背景始终是透明的，不能从外部访问。
- `LV_TABVIEW_PART_TAB_BG`：选项卡按钮，它是一个 Button 矩阵。单击一个按钮将滚动 `LV_TABVIEW_PART_BG_SCRL` 到相关选项卡的内容。可以通过访问选项卡按钮 `LV_TABVIEW_PART_TAB_BTN`。选择选项卡时，按钮处于选中状态，可以使用设置样式 `LV_STATE_CHECKED`。选项卡的按钮矩阵的高度是根据字体高度加上背景和按钮样式的填充来计算的。

列出的所有部分均支持典型的背景样式属性和填充。

`LV_TABVIEW_PART_TAB_BG` 还有一个额外的实际部分，即指标，称为 `LV_TABVIEW_PART_INDIC`。它是当前选定选项卡下的一个类似矩形的细对象。当选项卡视图设置为另一个选项卡的动画时，指示符也将设置为动画。使用典型的背景样式属性可以是样式。该尺寸样式属性将设置其厚度。

添加新选项卡后，将为其创建一个页面，并将 `LV_TABVIEW_PART_BG_SCRL` 新按钮添加到“ `LV_TABVIEW_PART_TAB_BG` 按钮”矩阵中。创建的页面可以用作普通页面，并且具有通常的页面部分。

38.3 用法

38.3.1 添加标签

可以使用添加新标签。它将返回指向可以创建选项卡内容的 Page 对象的指针。

```
lv_tabview_add_tab(tabview, "Tab name")
```

38.3.2 变更标签

要选择一个新标签，您可以：

- 在按钮矩阵部分上单击它
- 滑动
- 使用功能 `lv_tabview_set_tab_act(tabview, id, LV_ANIM_ON/OFF)`

38.3.3 更改标签的名称

要 `id` 在运行时更改选项卡的名称（基础按钮矩阵的显示文本），可以使用该功能。

```
lv_tabview_set_tab_name(tabview, id, name)
```

38.3.4 Tab 按钮的位置

默认情况下，选项卡选择器按钮位于“选项卡”视图的顶部。可以用

```
lv_tabview_set_btns_pos(tabview, LV_TABVIEW_TAB_POS_TOP/BOTTOM/LEFT/RIGHT/NONE)
```

`LV_TABVIEW_TAB_POS_NONE` 将隐藏标签。

请注意，添加标签后，您无法将标签的位置从顶部或底部更改为左侧或右侧。

38.3.5 动画时间

动画时间用调整。加载新选项卡时使用。`lv_tabview_set_anim_time(tabview, anim_time_ms)`

38.3.6 滚动传播

由于选项卡的内容对象是一个 Page，因此它可以接收来自其他类似 Page 的对象的滚动传播。例如，如果在选项卡的内容上创建了一个文本区域，并且滚动了该文本区域，但到达末尾，则滚动可以传播到内容页面。可以使用启用它。`lv_page/textarea_set_scroll_propagation(obj, true)`

默认情况下，选项卡的内容页面已启用滚动传播，因此，当它们水平滚动时，滚动会传播到`LV_TABVIEW_PART_BG_SCRL`该页面，这样页面将被滚动。

可以使用禁用手动滑动。`lv_page_set_scroll_propagation(tab_page, false)`

38.4 事件

除了通用事件之外，滑块还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 通过滑动或单击选项卡按钮选择新选项卡时发送

38.5 按键

Tabview 处理以下*键*：

- **LV_KEY_RIGHT / LEFT** 选择一个标签
- **LV_KEY_ENTER** 更改为所选标签

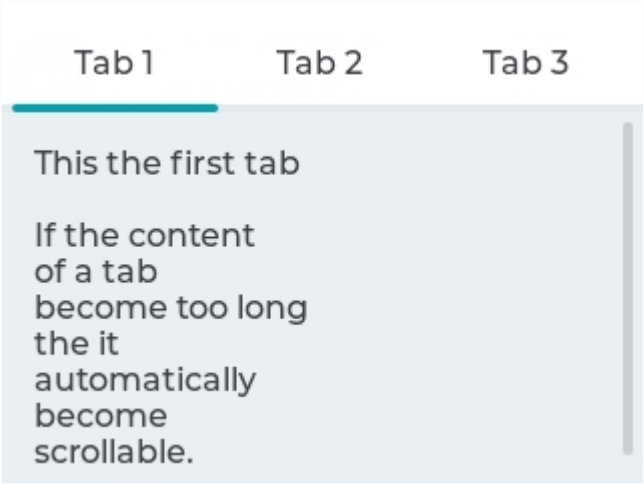
了解更多有关按键的信息。

Cai Xuefeng

例

C

简单的 Tabview



API

typedef

typedef uint8_t lv_tabview_btns_pos_t

typedef uint8_t lv_tabview_part_t

枚举

enum [anonymous]

tabview 按钮的位置。

值：

enumerator LV_TABVIEW_TAB_POS_NONE

enumerator LV_TABVIEW_TAB_POS_TOP

enumerator LV_TABVIEW_TAB_POS_BOTTOM

enumerator LV_TABVIEW_TAB_POS_LEFT

enumerator LV_TABVIEW_TAB_POS_RIGHT

enum [anonymous]

值：

enumerator LV_TABVIEW_PART_BG= LV_OBJ_PART_MAIN

enumerator _LV_TABVIEW_PART_VIRTUAL_LAST= _LV_OBJ_PART_VIRTUAL_LAST

enumerator LV_TABVIEW_PART_BG_SCRLLABLE= _LV_OBJ_PART_REAL_LAST

enumerator LV_TABVIEW_PART_TAB_BG

enumerator LV_TABVIEW_PART_TAB_BTN

enumerator LV_TABVIEW_PART_INDIC

enumerator _LV_TABVIEW_PART_REAL_LAST

功能

lv_obj_t * lv_tabview_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个标签视图对象

返回

指向创建的选项卡的指针

参数

- `par`：指向对象的指针，它将是新选项卡的父对象
- `copy`：指向选项卡对象的指针，如果不为 NULL，则将从其复制新对象

lv_obj_t * lv_tabview_add_tab（lv_obj_t * *tabview*，const char* *name*）

添加具有给定名称的新标签页

返回

指向创建的页面对象（lv_page）的指针。您可以在这里创建内容

参数

- `tabview`：指向“选项卡视图”对象的指针，该对象将在何处添加新选项卡
- `name`：选项卡按钮上的文本

void lv_tabview_clean_tab（lv_obj_t * *tab*）

删除由创建的选项卡的所有子项 `lv_tabview_add_tab`。

参数

- `tab`：指向标签的指针

void lv_tabview_set_tab_act（lv_obj_t * *tabview*，uint16_t *id*，lv_anim_enable_t *anim* ）

设置新标签

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：要加载的标签页的索引
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_tabview_set_tab_name（lv_obj_t * *tabview*，uint16_t *id*，char * *name* ）

设置标签的名称。

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：应该设置名称的标签的索引
- `name`：新标签页名称

void lv_tabview_set_anim_time（lv_obj_t * *tabview*，uint16_t *anim_time* ）

设置新标签页加载时标签页视图的动画时间

参数

- `tabview`：指向 Tab 视图对象的指针
- `anim_time`：动画时间（以毫秒为单位）

void lv_tabview_set_btns_pos（lv_obj_t * *tabview*，lv_tabview_btns_pos_t *btns_pos* ）

设置选项卡选择按钮的位置

参数

- `tabview`：指向选项卡视图对象的指针
- `btns_pos`：哪个按钮位置

---

uint16_t lv_tabview_get_tab_act（*const* lv_obj_t * *tabview* ）

获取当前活动选项卡的索引

返回

活动标签索引

参数

- `tabview`：指向 Tab 视图对象的指针

---

uint16_t lv_tabview_get_tab_count（*const* lv_obj_t * *tabview* ）

获取标签数

返回

标签数

参数

- `tabview`：指向 Tab 视图对象的指针

---

lv_obj_t * lv_tabview_get_tab（*const* lv_obj_t * *tabview*，uint16_t *id* ）

获取选项卡的页面（内容区域）

返回

指向页面（lv_page）对象的指针

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：选项卡的索引（> = 0）

---

uint16_t lv_tabview_get_anim_time（*const* lv_obj_t * *tabview* ）

加载新选项卡时获取选项卡视图的动画时间

返回

动画时间（以毫秒为单位）

参数

- `tabview`：指向 Tab 视图对象的指针

lv_tabview_btns_pos_t lv_tabview_get_btns_pos（*const* lv_obj_t * *tabview* ）

获取选项卡选择按钮的位置

参数

- `tabview`：指向 ab 视图对象的指针

struct lv_tabview_ext_t

公众成员

```
lv_obj_t *btns
lv_obj_t *indic
lv_obj_t *content
const 字符**tab_name_ptr
lv_point_t point_last
uint16_t tab_cur
uint16_t tab_cnt
uint16_t anim_time
lv_tabview_btns_pos_tbtns_pos
```

第三十九章 Tabview（lv_tabview）

39.1 总览

选项卡视图对象可用于组织选项卡中的内容。

39.2 小部件和样式

Tab 视图对象包含几个部分。主要是 `LV_TABVIEW_PART_BG`。它是一个矩形容器，用于容纳 Tab 视图的其他部分。

在背景上创建了两个重要的真实部分：

- `LV_TABVIEW_PART_BG_SCRL`：这是 Page 的可滚动部分。它使选项卡的内容彼此相邻。页面的背景始终是透明的，不能从外部访问。
- `LV_TABVIEW_PART_TAB_BG`：选项卡按钮，它是一个 Button 矩阵。单击一个按钮将滚动 `LV_TABVIEW_PART_BG_SCRL` 到相关选项卡的内容。可以通过访问选项卡按钮 `LV_TABVIEW_PART_TAB_BTN`。选择选项卡时，按钮处于选中状态，可以使用设置样式 `LV_STATE_CHECKED`。选项卡的按钮矩阵的高度是根据字体高度加上背景和按钮样式的填充来计算的。

列出的所有部分均支持典型的背景样式属性和填充。

`LV_TABVIEW_PART_TAB_BG` 还有一个额外的实际部分，即指标，称为 `LV_TABVIEW_PART_INDIC`。它是当前选定选项卡下的一个类似矩形的细对象。当选项卡视图设置为另一个选项卡的动画时，指示符也将设置为动画。使用典型的背景样式属性可以是样式。该尺寸样式属性将设置其厚度。

添加新选项卡后，将为其创建一个页面，并将 `LV_TABVIEW_PART_BG_SCRL` 新按钮添加到“ `LV_TABVIEW_PART_TAB_BG` 按钮”矩阵中。创建的页面可以用作普通页面，并且具有通常的页面部分。

39.3 用法

39.3.1 添加标签

可以使用添加新标签。它将返回指向可以创建选项卡内容的 Page 对象的指针。

```
lv_tabview_add_tab(tabview, "Tab name")
```

39.3.2 变更标签

要选择一个新标签，您可以：

- 在按钮矩阵部分上单击它
- 滑动
- 使用功能 `lv_tabview_set_tab_act(tabview, id, LV_ANIM_ON/OFF)`

39.3.3 更改标签的名称

要 `id` 在运行时更改选项卡的名称（基础按钮矩阵的显示文本），可以使用该功能。

```
lv_tabview_set_tab_name(tabview, id, name)
```

39.3.4 Tab 按钮的位置

默认情况下，选项卡选择器按钮位于“选项卡”视图的顶部。可以用

```
lv_tabview_set_btns_pos(tabview, LV_TABVIEW_TAB_POS_TOP/BOTTOM/LEFT/RIGHT/NONE)
```

`LV_TABVIEW_TAB_POS_NONE` 将隐藏标签。

请注意，添加标签后，您无法将标签的位置从顶部或底部更改为左侧或右侧。

39.3.5 动画时间

动画时间用调整。加载新选项卡时使用。`lv_tabview_set_anim_time(tabview, anim_time_ms)`

39.3.6 滚动传播

由于选项卡的内容对象是一个 Page，因此它可以接收来自其他类似 Page 的对象的滚动传播。例如，如果在选项卡的内容上创建了一个文本区域，并且滚动了该文本区域，但到达末尾，则滚动可以传播到内容页面。可以使用启用它。`lv_page/textarea_set_scroll_propagation(obj, true)`

默认情况下，选项卡的内容页面已启用滚动传播，因此，当它们水平滚动时，滚动会传播到`LV_TABVIEW_PART_BG_SCRL`该页面，这样页面将被滚动。

可以使用禁用手动滑动。`lv_page_set_scroll_propagation(tab_page, false)`

39.4 事件

除了通用事件之外，滑块还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 通过滑动或单击选项卡按钮选择新选项卡时发送

39.5 按键

Tabview 处理以下*键*：

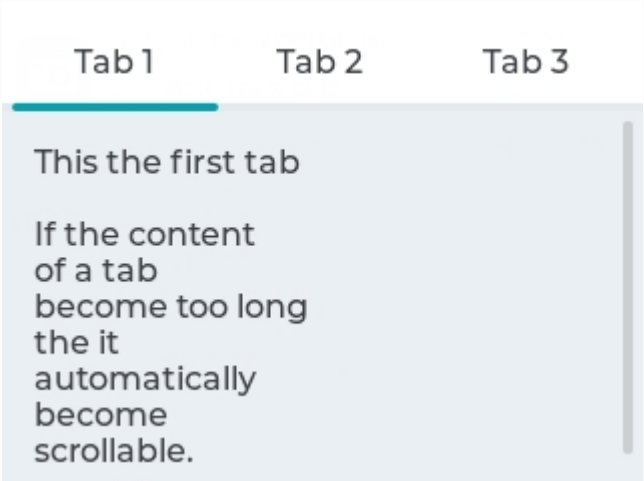
- **LV_KEY_RIGHT / LEFT** 选择一个标签
- **LV_KEY_ENTER** 更改为所选标签

例

C

Cai Xuefeng

简单的 Tabview



API

typedef

typedef uint8_t **lv_tabview_btns_pos_t**

typedef uint8_t lv_tabview_part_t

枚举

enum [anonymous]

tabview 按钮的位置。

值：

enumerator LV_TABVIEW_TAB_POS_NONE

enumerator LV_TABVIEW_TAB_POS_TOP

enumerator LV_TABVIEW_TAB_POS_BOTTOM

enumerator LV_TABVIEW_TAB_POS_LEFT

enumerator LV_TABVIEW_TAB_POS_RIGHT

enum [anonymous]

值：

enumerator LV_TABVIEW_PART_BG= LV_OBJ_PART_MAIN

enumerator _LV_TABVIEW_PART_VIRTUAL_LAST= _LV_OBJ_PART_VIRTUAL_LAST

enumerator LV_TABVIEW_PART_BG_SCRLLABLE= _LV_OBJ_PART_REAL_LAST

enumerator LV_TABVIEW_PART_TAB_BG

enumerator LV_TABVIEW_PART_TAB_BTN

enumerator LV_TABVIEW_PART_INDIC

enumerator _LV_TABVIEW_PART_REAL_LAST

功能

lv_obj_t * lv_tabview_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个标签视图对象

返回

指向创建的选项卡的指针

参数

- `par`：指向对象的指针，它将是新选项卡的父对象
- `copy`：指向选项卡对象的指针，如果不为 NULL，则将从其复制新对象

lv_obj_t * lv_tabview_add_tab（lv_obj_t * *tabview*，const char * *name*）

添加具有给定名称的新标签页

返回

指向创建的页面对象（lv_page）的指针。您可以在这里创建内容

参数

- `tabview`：指向“选项卡视图”对象的指针，该对象将在何处添加新选项卡
- `name`：选项卡按钮上的文本

void lv_tabview_clean_tab（lv_obj_t * *tab*）

删除由创建的选项卡的所有子项 `lv_tabview_add_tab`。

参数

- `tab`：指向标签的指针

void lv_tabview_set_tab_act（lv_obj_t * *tabview*，uint16_t *id*，lv_anim_enable_t *anim* ）

设置新标签

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：要加载的标签页的索引
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_tabview_set_tab_name（lv_obj_t * *tabview*，uint16_t *id*，char * *name* ）

设置标签的名称。

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：应该设置名称的标签的索引
- `name`：新标签页名称

void lv_tabview_set_anim_time（lv_obj_t * *tabview*，uint16_t *anim_time* ）

设置新标签页加载时标签页视图的动画时间

参数

- `tabview`：指向 Tab 视图对象的指针
- `anim_time`：动画时间（以毫秒为单位）

void lv_tabview_set_btns_pos（lv_obj_t * *tabview*，lv_tabview_btns_pos_t *btns_pos* ）

设置选项卡选择按钮的位置

参数

- `tabview`：指向选项卡视图对象的指针
- `btns_pos`：哪个按钮位置

`uint16_t lv_tabview_get_tab_act（const lv_obj_t * tabview ）`

获取当前活动选项卡的索引

返回

活动标签索引

参数

- `tabview`：指向 Tab 视图对象的指针

`uint16_t lv_tabview_get_tab_count（const lv_obj_t * tabview ）`

获取标签数

返回

标签数

参数

- `tabview`：指向 Tab 视图对象的指针

`lv_obj_t * lv_tabview_get_tab（const lv_obj_t * tabview，uint16_t id ）`

获取选项卡的页面（内容区域）

返回

指向页面（lv_page）对象的指针

参数

- `tabview`：指向 Tab 视图对象的指针
- `id`：选项卡的索引（> = 0）

`uint16_t lv_tabview_get_anim_time（const lv_obj_t * tabview ）`

加载新选项卡时获取选项卡视图的动画时间

返回

动画时间（以毫秒为单位）

参数

- `tabview`：指向 Tab 视图对象的指针

`lv_tabview_btns_pos_t lv_tabview_get_btns_pos（const lv_obj_t * tabview ）`

获取选项卡选择按钮的位置

参数

- `tabview`：指向 ab 视图对象的指针

struct **lv_tabview_ext_t**

公众成员

lv_obj_t ***btns**

lv_obj_t ***indic**

lv_obj_t ***content**

const 字符****tab_name_ptr**

lv_point_t **point_last**

uint16_t **tab_cur**

uint16_t **tab_cnt**

uint16_t **anim_time**

lv_tabview_btns_pos_t**btns_pos**

第四十章 平铺视图（lv_tileview）

40.1 总览

Tileview 是一个容器对象，其中的元素（称为 *tile*）可以以网格形式排列。通过滑动，用户可以在图块之间导航。

如果 Tileview 是屏幕尺寸的，它将提供您可能已经在智能手表上看到的用户界面。

40.2 小部件和样式

Tileview 与 Page 的部分相同。期望，`LV_PAGE_PART_SCRL`因为它不能被引用，并且始终透明。请参阅页面的详细文档。

40.3 用法

40.3.1 有效职位

磁贴不必在每个元素都存在的地方形成完整的网格。网格中可以有孔，但必须是连续的，即不能有空的行或列。

随着有效的位置也可设置。仅可以滚动到该位置。该指数手段左上方图块。例如，给出“ L”形的平铺视图。它指示其中没有图块，因此用户无法在那里滚动。

```
lv_tileview_set_valid_positions(tileview, valid_pos_array, array_len)0,0lv_point_t valid_pos_array[] = {{
```

```
0,0}, {0,1}, {1,1}, {{LV_COORD_MIN, LV_COORD_MIN}}{1,1}
```

换句话说，`valid_pos_array`告诉人们瓷砖在哪里。可以即时更改它以禁用特定图块上的某些位置。例如，可能有一个 2x2 网格，其中添加了所有图块，但第一行（y = 0）作为“主行”，第二行（y = 1）包含其上方图块的选项。false 设水平滚动只能在主行中进行，而在第二行中的选项之间则不可能进行。在这种情况下，`valid_pos_array`选择新的主磁贴时需要更改：

- 对于第一个主磁贴：禁用选项磁贴`{0,0}, {0,1}, {1,0}{1,1}`
- 用于第二个主磁贴以禁用选项磁贴`{0,0}, {1,0}, {1,1}{0,1}`

40.3.2 设置瓷砖

要设置当前可见的图块，请使用。`lv_tileview_set_tile_act(tileview, x_id, y_id, LV_ANIM_ON/OFF)`

40.3.3 添加元素

要添加元素，只需在 Tileview 上创建一个对象并将其手动定位到所需位置即可。

`lv_tileview_add_element(tielview, element)`应该用来使 Tileview 滚动（拖动）其元素之一。例如，如果图块上有一个按钮，则需要将该按钮显式添加到 Tileview 中，以使用户也可以使用该按钮滚动 Tileview。

40.3.4 滚动传播

页面状对象（如 List）的滚动传播功能在这里可以很好地使用。例如，可以有一个完整的列表，当列表到达最顶部或最底部时，用户将改为滚动图块视图。

40.3.5 动画时间

使用可以调整 Tileview 的动画时间。`lv_tileview_set_anim_time(tileview, anim_time)`

在以下情况下应用动画

- 选择一个新的图块 `lv_tileview_set_tile_act`
- 当前磁贴稍微滚动然后释放（还原原始标题）
- 当前磁贴滚动超过一半大小，然后释放（移至下一个磁贴）

40.3.6 边缘闪光灯

当滚动到达的图块视图碰到 void 位置或图块视图的末尾时，可以添加“边缘闪光”效果。

使用启用此功能。`lv_tileview_set_edge_flash(tileview, true)`

40.4 事件

除了通用事件之外，滑块还会发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 当加载了带有滚动或的新图块时发送

`lv_tileview_set_act`。将事件数据设置为新图块的索引 `valid_pos_array`（类型为）`uint32_t *`

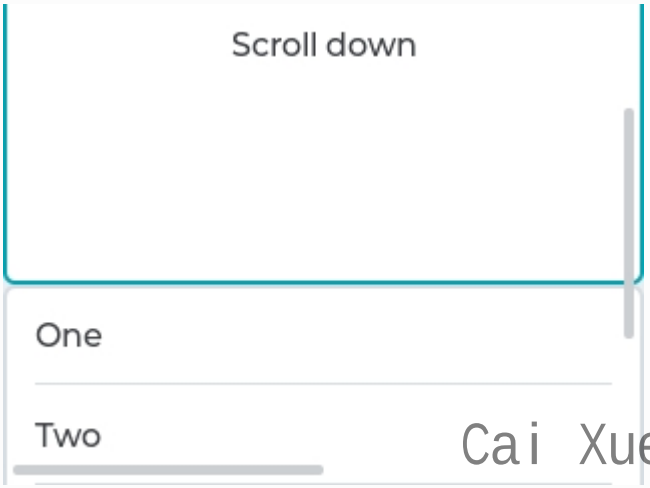
40.5 按键

- **LV_KEY_UP，LV_KEY_RIGHT** 将滑块的值增加 1
- **LV_KEY_DOWN，LV_KEY_LEFT** 将滑块的值减 1

例

C

包含内容的 **Tileview**



API

枚举

enum [anonymous]

值：

enumerator LV_TILEVIEW_PART_BG= LV_PAGE_PART_BG

enumerator LV_TILEVIEW_PART_SCROLLBAR= LV_PAGE_PART_SCROLLBAR

enumerator LV_TILEVIEW_PART_EDGE_FLASH= LV_PAGE_PART_EDGE_FLASH

enumerator _LV_TILEVIEW_PART_VIRTUAL_LAST= _LV_PAGE_PART_VIRTUAL_LAST

enumerator _LV_TILEVIEW_PART_REAL_LAST= _LV_PAGE_PART_REAL_LAST

功能

lv_obj_t * lv_tileview_create（lv_obj_t * *par*，*const* lv_obj_t * *copy*）

创建一个 tileview 对象

返回

指向创建的 tileview 的指针

参数

- `par`：指向对象的指针，它将是新 tileView 的父对象
- `copy`：指向 tileview 对象的指针，如果不为 NULL，则将从其复制新对象

void lv_tileview_add_element（lv_obj_t * tileview，lv_obj_t * element）

在 tileview 上注册一个对象。注册对象将能够滑动 tileview

参数

- `tileview`：指向 Tileview 对象的指针
- `element`：指向对象的指针

void lv_tileview_set_valid_positions（lv_obj_t * tileview，const lv_point_t valid_pos []，uint16_t valid_pos_cnt ）

设置有效位置的索引。只能在这些位置滚动。

参数

- `tileview`：指向 Tileview 对象的指针
- `valid_pos`：数组宽度索引。例如。仅保存了指针，因此不能成为局部变量。


```
lv_point_t p[] = {{0,0}, {1,0}, {1,1}
```

- `valid_pos_cnt`：`valid_pos` 数组中元素的数量

void lv_tileview_set_tile_act（lv_obj_t * tileview，lv_coord_t x，lv_coord_t y，lv_anim_enable_t anim ）

设置要显示的图块

参数

- `tileview`：指向 tileview 对象的指针
- `x`：列 ID（0、1、2 ...）
- `y`：行号（0，1，2 ...）
- `anim`：LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：立即更改值

void lv_tileview_set_edge_flash（lv_obj_t * tileview，bool en ）

启用边缘闪光效果。（到达边缘时显示弧线）

参数

- `tileview`：指向 Tileview 的指针
- `en`：true 或 false 启用/禁用端闪

void **lv_tileview_set_anim_time**（lv_obj_t * *tileview*，uint16_t *anim_time* ）

设置图块视图的动画时间

参数

- `tileview`：指向页面对象的指针
- `anim_time`：动画时间（以毫秒为单位）

void **lv_tileview_get_tile_act**（lv_obj_t * *tileview*，lv_coord_t * *x*，lv_coord_t * *y* ）

获取要显示的图块

参数

- `tileview`：指向 tileview 对象的指针
- `x`：列 ID（0、1、2 ...）
- `y`：行号（0，1，2 ...）

bool **lv_tileview_get_edge_flash**（lv_obj_t * *tileview* ）

获取滚动传播属性

返回

对或错

参数

- `tileview`：指向 Tileview 的指针

uint16_t **lv_tileview_get_anim_time**（lv_obj_t * *tileview* ）

获取图块视图的动画时间

返回

动画时间（以毫秒为单位）

参数

- `tileview`：指向页面对象的指针

struct **lv_tileview_ext_t**

公众成员

```
lv_page_ext_tpage
const lv_point_t *valid_pos
uint16_t valid_pos_cnt
uint16_t anim_time
lv_point_t act_id
uint8_t drag_top_en
uint8_t drag_bottom_en
uint8_t drag_left_en
uint8_t drag_right_en
```

第四十一章 窗口（lv_win）

41.1 总览

窗口是类似容器的对象，由带有标题和按钮的标题以及内容区域构建。

41.2 小部件和样式

主要部分 `LV_WIN_PART_BG` 包含另外两个实际部分：

1. `LV_WIN_PART_HEADER`：顶部的标题容器，带有标题和控制按钮
2. `LV_WIN_PART_CONTENT_SCRL` 页眉下方内容的页面可滚动部分。

除此之外，`LV_WIN_PART_CONTENT_SCRL` 还有一个名为的滚动条部分 `LV_WIN_PART_CONTENT_SCRL`。阅读 Page 的文档以获取有关滚动条的更多详细信息。

所有部分均支持典型的背景属性。标题使用标题部分的 *Text* 属性。

控制按钮的高度为：*标头高度-标头 padding_top-标头 padding_bottom*。

41.2.1 标题

标题上有一个标题，可以通过以下方式修改。`lv_win_set_title(win, "New title")`

41.2.2 控制按钮

可以使用以下命令将控制按钮添加到窗口标题的右侧：要在窗口标题的左侧添加按钮，请改用。第二个参数是图像源，因此它可以是 sign，指向变量的指针或文件的 path。

```
lv_win_add_btn_right(win, LV_SYMBOL_CLOSE)lv_win_add_btn_left(win, LV_SYMBOL_CLOSE)lv_img_dsc_t
```

可以使用来设置按钮的宽度。如果按钮将是方形的。`lv_win_set_btn_width(win, w)w == 0`

`lv_win_close_event_cb` 可以用作关闭窗口的事件回调。

41.2.3 滚动条

滚动条的行为可以通过设置。有关详细信息，请参见页面。

`lv_win_set_scrlbar_mode(win, LV_SCRLBAR_MODE_...)`

41.2.4 手动滚动和聚焦

要直接滚动窗口，可以使用或。`lv_win_scroll_hor(win, dist_px)lv_win_scroll_ver(win, dist_px)`

要使 Window 在其上显示一个对象，请使用。`lv_win_focus(win, child, LV_ANIM_ON/OFF)`

滚动和焦点动画的时间可以通过 `lv_win_set_anim_time(win, anim_time_ms)`

41.2.5 布局

要设置内容的布局，请使用。有关详细信息，请参见容器。

`lv_win_set_layout(win, LV_LAYOUT_...)` Cai Xuefeng

41.3 事件

仅通用事件是按对象类型发送的。

41.4 按键

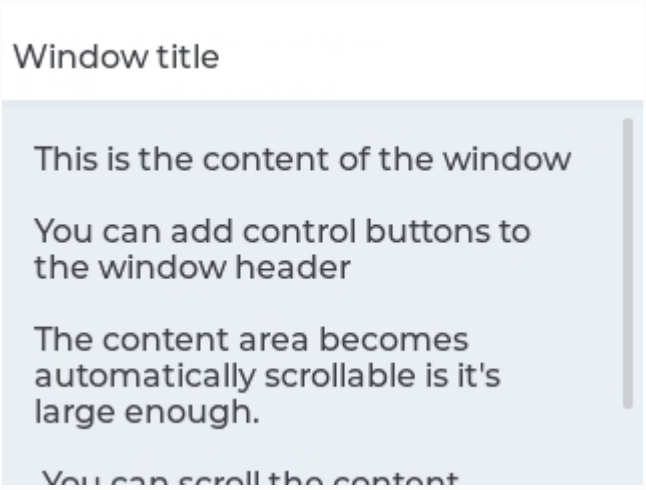
页面处理以下*键*:

- **LV_KEY_RIGHT / LEFT / UP / DOWN** 滚动页面

例

C

简单的窗口



API

枚举

enum [anonymous]

窗户小部件。

值：

enumerator LV_WIN_PART_BG= LV_OBJ_PART_MAIN

窗口对象背景样式。

enumerator _LV_WIN_PART_VIRTUAL_LAST

enumerator LV_WIN_PART_HEADER= _LV_OBJ_PART_REAL_LAST

窗口标题栏背景样式。

enumerator LV_WIN_PART_CONTENT_SCROLLABLE

窗口内容样式。

enumerator LV_WIN_PART_SCROLLBAR

窗口滚动条样式。

enumerator _LV_WIN_PART_REAL_LAST

功能

lv_obj_t * lv_win_create（lv_obj_t * *par*，const lv_obj_t * *copy*）

创建一个窗口对象

返回

指向创建的窗口的指针

参数

- `par`：指向对象的指针，它将是新窗口的父对象
- `copy`：指向窗口对象的指针，如果不为 NULL，则将从其复制新对象

---

void lv_win_clean（lv_obj_t * *win* ）

删除 scrl 对象的所有子级，而不删除 scrl 子级。

参数

- `win`：指向对象的指针

---

lv_obj_t * lv_win_add_btn_right（lv_obj_t * *win*，const void* *img_src* ）

在窗口标题右侧添加控制按钮

返回

指向创建的按钮对象的指针

参数

- `win`：指向窗口对象的指针
- `img_src`：图片来源（“ lv_img_t”变量，文件或 sign 的 path）

---

lv_obj_t * lv_win_add_btn_left（lv_obj_t * *win*，const void* *img_src* ）

在窗口标题的左侧添加控制按钮

返回

指向创建的按钮对象的指针

参数

- `win`：指向窗口对象的指针
- `img_src`：图片来源（“ lv_img_t”变量，文件或 sign 的 path）

---

void lv_win_close_event_cb（lv_obj_t * *btn*，lv_event_t • *evet*）

可以分配给窗口控制按钮以关闭窗口

参数

- `btn`：指向寡妇头上的控制按钮的指针
- `evet`：事件类型

---

void lv_win_set_title（lv_obj_t * *win*，const char* *title*）

设置窗口标题

参数

- `win`：指向窗口对象的指针
- `title`：新标题的字符串

void lv_win_set_header_height（lv_obj_t * *win*，lv_coord_t *size* ）

设置窗口的控制按钮大小

返回

控制按钮尺寸

参数

- `win`：指向窗口对象的指针

void lv_win_set_btn_width（lv_obj_t * *win*，lv_coord_t *width* ）

设置页眉上的控制按钮的宽度

参数

- `win`：指向窗口对象的指针
- `width`：控制按钮的宽度。0：自动使它们成正方形。

void lv_win_set_content_size（lv_obj_t * *win*，lv_coord_t *w*，lv_coord_t *h* ）

设置内容区域的大小。

参数

- `win`：指向窗口对象的指针
- `w`：宽度
- `h`：高度（窗口将随着标题的高度变高）

void lv_win_set_layout（lv_obj_t * *win*，lv_layout_t　*layout*）

设置窗口的布局

参数

- `win`：指向窗口对象的指针
- `layout`：来自“ lv_layout_t”的布局

void lv_win_set_scrollbar_mode（lv_obj_t * *win*，lv_scrollbar_mode_t *sb_mode* ）

设置窗口的滚动条模式

参数

- `win`：指向窗口对象的指针
- `sb_mode`：来自“ lv_scrollbar_mode_t”的新滚动条模式

void lv_win_set_anim_time（lv_obj_t * win，uint16_t anim_time ）

将焦点动画持续时间设置为 `lv_win_focus()`

参数

- `win`：指向窗口对象的指针
- `anim_time`：动画的持续时间[ms]

void lv_win_set_drag（lv_obj_t * win，bool en ）

设置窗口的拖动状态。如果设置为“ true”，则可以像在 PC 上一样拖动窗口。

参数

- `win`：指向窗口对象的指针
- `en`：是否启用拖动

const 字符* lv_win_get_title（const lv_obj_t * win ）

获取窗口标题

返回

窗口的标题字符串

参数

- `win`：指向窗口对象的指针

lv_obj_t * lv_win_get_content（const lv_obj_t * win ）

获取窗口（`lv_page`）的内容持有者对象以允许其他自定义

返回

窗口内容所在的 Page 对象

参数

- `win`：指向窗口对象的指针

lv_coord_t lv_win_get_header_height（const lv_obj_t * win ）

获取标题高度

返回

标题高度

参数

- `win`：指向窗口对象的指针

---

lv_coord_t **lv_win_get_btn_width**（lv_obj_t * *win* ）

获取标题上控制按钮的宽度

返回

控制按钮的宽度。0：正方形。

参数

- `win`：指向窗口对象的指针

---

lv_obj_t * **lv_win_get_from_btn**（*const* lv_obj_t * *ctrl_btn* ）

从其控制按钮之一获取寡妇的指针。在仅知道按钮的控制按钮的操作中很有用。

返回

指向“ ctrl_btn”窗口的指针

参数

- `ctrl_btn`：指向窗口控制按钮的指针

---

lv_layout_t **lv_win_get_layout**（lv_obj_t * *win* ）

获取窗口的布局

返回

窗口的布局（来自“ lv_layout_t”）

参数

- `win`：指向窗口对象的指针

---

lv_scrollbar_mode_t **lv_win_get_sb_mode**（lv_obj_t * *win* ）

获取窗口的滚动条模式

返回

窗口的滚动条模式（来自“ lv_sb_mode_t”）

参数

- `win`：指向窗口对象的指针

uint16_t lv_win_get_anim_time（const lv_obj_t * win ）

获取焦点动画的持续时间

返回

动画持续时间[ms]

参数

- `win`：指向窗口对象的指针

lv_coord_t lv_win_get_width（lv_obj_t * win ）

获取窗口内容区域的宽度（页面可滚动）

返回

内容区域的宽度

参数

- `win`：指向窗口对象的指针

bool lv_win_get_drag（const lv_obj_t * win ）

获取窗口的拖动状态。如果设置为“ true”，则可以像在 PC 上一样拖动窗口。

返回

窗口是否可拖动

参数

- `win`：指向窗口对象的指针

void lv_win_focus（lv_obj_t * win，lv_obj_t * obj，lv_anim_enable_t anim_en ）

专注于一个对象。它确保对象将在窗口中可见。

参数

- `win`：指向窗口对象的指针
- `obj`：指向要聚焦的对象的指针（必须在窗口中）
- `anim_en`：LV_ANIM_ON 带有动画焦点；LV_ANIM_OFF 不含动画的焦点

void lv_win_scroll_hor（lv_obj_t * win，lv_coord_t dist ）

水平滚动窗口

参数

- `win`：指向窗口对象的指针

- `dist`：滚动距离（<0：向右滚动；> 0 向左滚动）

void lv_win_scroll_ver（lv_obj_t * *win*，lv_coord_t *dist* ）

垂直滚动窗口

参数

- `win`：指向窗口对象的指针
- `dist`：滚动距离（<0：向下滚动；> 0 向上滚动）

struct lv_win_ext_t

公众成员

```
lv_obj_t *page
lv_obj_t *header
字符*title_txt
lv_coord_t btn_w
```