User Manual

AWTK 自定义控件开发指南

用户手册

V1.0.2 Date:2025/10/31

类别	内容
关键词	AWTK、自定义控件
摘要	本文将介绍 AWTK 自定义控件的规范，以及通过 AWTK Designer 创建、导入并使用的自定义控件的步骤。

修订历史

版本	日期	原因
1.0.2	2023/05/26	• 添加如何通过继承完成自定义控件章节。
1.0.1	2022/10/24	• 完善导入自定义控件； • 介绍如何使用示例程序验证控件功能； • 完善自定义控件的初始属性和缺省样式。
1.0.0	2022/10/17	• first implement

目 录

文档导读

本文将介绍 AWTK 自定义控件的规范，以及通过 AWTK Designer（下面简称 Designer）创建、导入并使用的自定义控件的步骤，帮助用户深入了解并开发属于自己的自定义控件。

1. 简介

AWTK 中内置了丰富的基础控件（比如 button、label、edit 等）和扩展控件（gif、gauge、rich_text 等），开发者通过组合这些控件可以快速构建复杂的 GUI 界面，但当 AWTK 内置的这些控件无法满足更复杂的应用场景时，我们还可以通过开发自定义控件来实现这些功能。

AWTK 提供了非常便利的实现框架，只需定义好控件的结构类型并编写几个重载函数即可。本文将介绍 AWTK 自定义控件的规范，以及通过 AWTK Designer（下面简称 Designer）创建、导入并使用的自定义控件的步骤，帮助用户深入了解并开发属于自己的自定义控件。

注: AWTK Designer[1] 是专门用来制作 AWTK 应用程序 UI 界面的实用工具，只要通过拖拽和点击就可以完成复杂的界面设计。

1.1 自定义控件的目录结构

本文主要以显示数值的文本控件 number_label 为例，该控件可以在 Designer 中直接安装，如下图所示，也可以前往 JihuLab[2] 下载，比如此处下载 awtk-widget-number-label[3]。

图 1.1 number_label

下载完成后，可以打包资源并编译示例程序，查看自定义控件的示例效果，具体步骤可以参考控件目录下的 README.md 文档，自定义控件目录结构如下表所示：

目录/文件夹	说明	备注
bin	存放动态库和可执行文件	编译后生成
demos	存放示例程序的源代码	
design	存放示例程序的原始资源，可在 Designer 上编辑	
docs	存放相关说明文档	
idl	存放生成的 IDL 文件	
lib	存放静态库动态库	编译后生成
res	存放示例程序运行时需要的资源	打包资源后生成
scripts	存放打包资源的脚本	

[1] https://awtk.zlg.cn/awstudio/download.html
[2] https://jihulab.com/awtk
[3] https://jihulab.com/awtk/awtk-widget-number-label

续上表

目录/文件夹	说明	备注
src	存放自定义控件的源代码	
tests	存放单元测试源代码	
LICENSE	许可证	
project.json	项目描述文件，包含项目设置等信息	
README.md	说明文档	
SConstruct	SCons 编译脚本	

1.2 自定义控件的描述文件

在上一小节介绍的自定义控件目录中，project.json 文件包含了控件的基本描述信息，详见下表：

参数	说明
name	控件的类型名，全小写英文字母，单词之间用下划线连接
version	版本号
date	创建日期
team	开发团队名称
author	作者联系方式
desc	控件的功能描述
copyright	版权声明

以上信息均可在 Designer 创建自定义控件时填写，例如，awtk-widget-number-label/project.json 文件中包含的信息如下：

```
{
  "name": "number_label",
  "version": "1.0.0",
  "date": "2020-05-31",
  "team": "AWTK Develop Team",
  "author": "Li XianJing <xianjimli@hotmail.com>",
  "desc": "用于显示数值的文本控件。",
  "copyright": "Guangzhou ZHIYUAN Electronics Co.,Ltd.",
  ......
}
```

2. 使用自定义控件

2.1 安装自定义控件

Designer 的插件管理页面提供了一些推荐的自定义控件，用户可以直接安装到项目当中使用，安装完成后使用步骤与其他的内置控件一样，非常方便。

此处在插件管理页面安装 number_label 控件，安装成功后控件会被添加到”已安装”分组，并且 Designer 右下角会弹出安装成功提示框，点击”刷新”按钮后，就可以在控件列表的”自定义”分组中找到安装的 number_label 控件了，后续的使用步骤与其他内置的普通控件一样。

图 2.1　安装自定义控件

自定义控件会被安装在项目的 3rd 目录下，比如此处的 3rd/awtk-widget-number-label，并且 Designer 会自动完成以下工作：

1. 在项目的编译脚本（SConstruct）中引用自定义控件的动态链接库。
2. 在项目的 src/application.c 中添加自定义控件的注册代码。

2.2 卸载自定义控件

如果不想在当前的项目中继续使用某个自定义控件，可以在 Designer 插件管理页面的”已安装”分组中将其卸载，卸载成功后 Designer 会删除 3rd 目录下的本地文件并清除安装时自动添加的代码，并且右下角会弹出卸载成功提示框，点击”刷新”按钮后，控件列表的”自定义”分组就会清除该控件，如下图所示：

图 2.2　卸载自定义控件

注: 自定义控件被卸载后，原自定义控件将识别为虚拟的占位控件。如果后续没有另行注册同名的自定义控件，那么运行时会出现断言错误，关于占位控件的更多信息，请查阅本文第四章。

2.3　导入自定义控件

安装 Designer 中的提供的推荐自定义控件需要联网下载，如果电脑处于离线状态，本地有下载好的且符合规范的自定义控件，那么可以直接在 Designer 的插件管理页面中导入这些自定义控件，如下图所示，导入完成后，安装和使用的步骤与其他推荐的自定义控件一样。

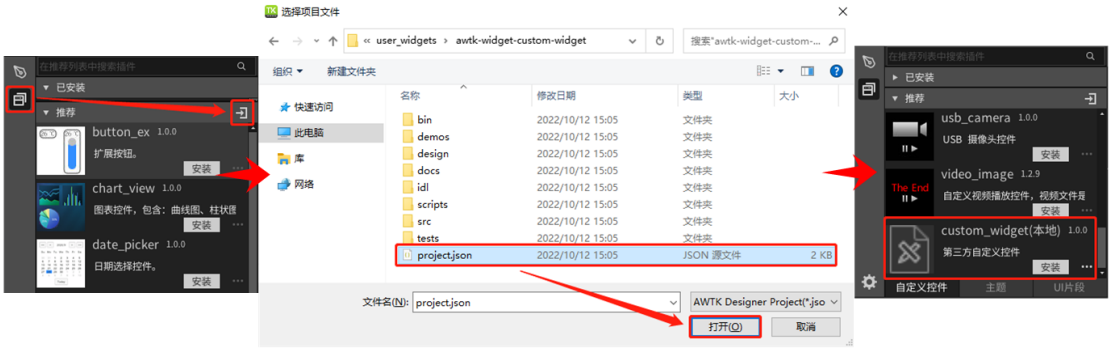

图 2.3　导入第三方自定义控件

注: 这些本地的自定义控件可以是 AWTK 团队开发的，也可以是第三方开发的，只要符合 AWTK 自定义控件规范[4] 就能正常导入并使用。

[4] https://github.com/zlgopen/awtk/blob/master/docs/custom_widget_rules.md

3. 开发自定义控件

3.1 创建自定义控件

开发自定义控件之前，需要参考 AWTK 自定义控件规范[5] 搭建自定义控件框架，为了节省开发时间，建议使用 Designer 新建自定义控件项目，新建完成后可以得到一个空白控件的框架，步骤如下图所示。

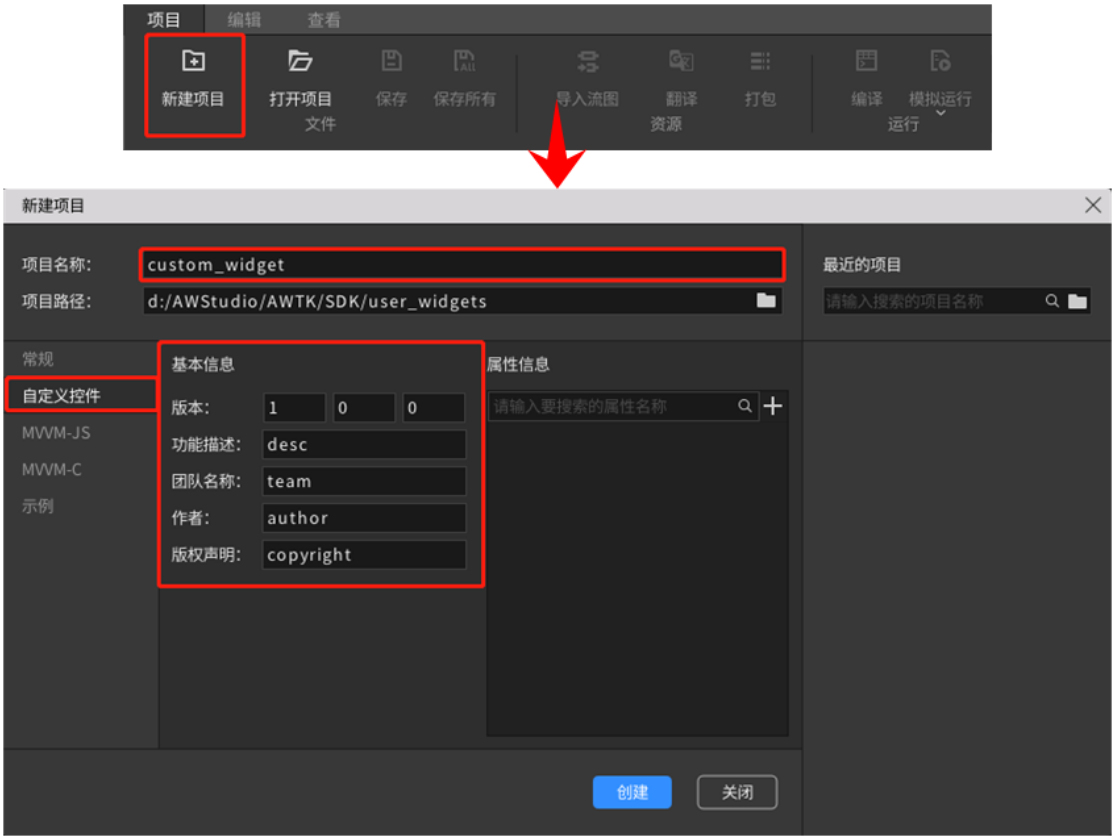

图 3.1 新建自定义控件项目

新建完成后，Designer 会自动打开该项目，并且可以在指定的项目路径中找到 `awtk-widget-xxx` 项目，例如此处的 `awtk-widget-custom-widget`，自定义控件项目的目录结构及其说明详见本文第一章。

3.2 实现自定义控件

使用 Designer 创建好的自定义控件只是一个空白的控件框架，无具体功能，需要我们遵循 AWTK 自定义控件规范[6] 完善相关的代码，才能真正实现自定义控件。下文会以 Designer 中内置推荐的 number_label 控件为例，介绍如何实现一个具体的自定义控件。

注: number_label 控件的获取方法详见本文第一章。

[5] https://github.com/zlgopen/awtk/blob/master/docs/custom_widget_rules.md
[6] https://github.com/zlgopen/awtk/blob/master/docs/custom_widget_rules.md

3.2.1 控件的虚表结构体

在编写控件的逻辑代码之前，首先需要了解 AWTK 中的控件虚表结构体（widget_vtable_t），该结构体中的成员主要描述控件类型，并完成控件的创建、绘制以及事件响应等等，详细的声明代码如下详见：widget.h[7]。

在实现自定义控件时，我们通常需要实现的 widget_vtable_t 结构体中的成员（属性/函数）详见下表：

属性/函数	类型	说明	触发时机
size	uint32_t	控件类型的大小	
type	const char*	控件类型的名称	
create	函数指针	控件的构造函数	创建控件
on_paint_self	函数指针	控件的绘制函数	每一帧都会调用该函数绘制控件
on_event	函数指针	控件的事件处理函数	控件接收到输入设备事件
set_prop	函数指针	设置控件属性	调用 widget_set_prop 设置属性
get_prop	函数指针	获取控件属性	调用 widget_get_prop 获取属性
on_destroy	函数指针	控件的销毁回调函数	销毁函数

注：除了以上列表中介绍的属性/函数外，控件虚表结构体（widget_vtable_t）中还有许多其他可重载的控件函数，具体请查看 widget.h[8]。

在代码中，可以直接使用 AWTK 提供的宏定义 TK_DECL_VTABLE 来创建控件的虚表结构体（widget_vtable_t），该宏定义的声明如下：

```
/* awtk/src/base/types_def.h */
#define TK_DECL_VTABLE(vt)                                                    \
  extern const widget_vtable_t g_##vt##_vtable;                               \
  const widget_vtable_t* vt##_get_widget_vtable(void) {return &g_##vt##_vtable;} \
  const widget_vtable_t g_##vt##_vtable
```

例如，在 number_label 控件中，widget_vtable_t 的定义代码如下，下文会详细介绍这些属性和函数的实现：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
TK_DECL_VTABLE(number_label) = {.size = sizeof(number_label_t),
                                .type = WIDGET_TYPE_NUMBER_LABEL,
                                ......
                                .create = number_label_create,
                                .on_paint_self = number_label_on_paint_self,
                                .set_prop = number_label_set_prop,
                                .get_prop = number_label_get_prop,
                                .on_event = number_label_on_event,
                                .on_destroy = number_label_on_destroy};
```

[7] https://github.com/zlgopen/awtk/blob/master/src/base/widget.h
[8] https://github.com/zlgopen/awtk/blob/master/src/base/widget.h

3.2.2 控件类型名称

控件类型名称要求使用小写的英文单词，多个单词之间用下划线连接。例如 number_label 控件的类型名称定义如下：

```
/* awtk-widget-number-label/src/number_label/number_label.h */
#define WIDGET_TYPE_NUMBER_LABEL "number_label"
```

并且该类型名称需要设置到控件虚表结构体（widget_vtable_t）中，让 AWTK 能够通过类型名称创建注册的控件，代码如下：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
TK_DECL_VTABLE(number_label) = {.type = WIDGET_TYPE_NUMBER_LABEL,
                                ......};
```

3.2.3 控件类型结构体

控件类型结构体定义时要求为：控件类型名称 + “_t”，并且必须以 widget_t 作为父类，例如 number_label 控件结构体的定义如下：

```
/* awtk-widget-number-label/src/number_label/number_label.h */
typedef struct _number_label_t {
  widget_t widget;
  ......
} number_label_t;
```

在控件类型结构体中可以定义控件特有的属性，属性名称要求使用小写的英文单词，多个单词之间用下划线连接，例如 number_label 控件的 format 属性定义如下：

```
/* awtk-widget-number-label/src/number_label/number_label.h */
typedef struct _number_label_t {
  widget_t widget;

  /**
   * @property {char*} format
   * @annotation ["set_prop","get_prop","readable","design","scriptable"]
   * 格式字符串。
   */
  char* format;
  ......
} number_label_t;
```

注：属性注释的格式以及详细含义请查阅：AWTK API 注释格式[9]。

定义好控件类型结构体后，需要将其大小设置到控件虚表结构体（widget_vtable_t）中，代码如下：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
TK_DECL_VTABLE(number_label) = {.size = sizeof(number_label_t),
                                ......};
```

[9] https://github.com/zlgopen/awtk/blob/master/docs/api_doc.md

注: AWTK 创建控件时，将使用控件虚表中的 size 属性 malloc 内存，如果该属性设置有误，程序运行中可能会出现越界访问，从而导致程序崩溃。

3.2.4 控件的构造函数

控件构造函数主要负责分配内存以及初始化控件属性，通常都需要先调用 widget_create 函数创建 widget_t 基类对象，并将控件虚表结构体（widget_vtable_t）赋给基类对象。例如 number_label 控件的构造函数代码如下：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
widget_t* number_label_create(widget_t* parent, xy_t x, xy_t y, wh_t w, wh_t h) {
  /* 创建对象并分配内存 */
  number_label_t* number_label =
      NUMBER_LABEL(widget_create(parent, TK_REF_VTABLE(number_label), x, y, w, h));

  /* 初始化控件属性 */
  number_label->format = tk_strdup(NUMBER_LABEL_DEFAULT_FORMAT);
  number_label->min = 0;
  number_label->max = 0;
  number_label->step = 1;
  number_label->readonly = FALSE;
  number_label->decimal_font_size_scale = 0.6;

  return (widget_t*)number_label;
}
```

3.2.5 控件的绘制函数

AWTK 程序启动后，会进入 GUI 主循环线程，每一循环一次（即绘制一帧）就会调用一次各个控件的绘制函数。通常我们可以调用 AWTK 提供的 canvas 和 vgcanvas 来绘制控件，具体的用法可以参考《AWTK 开发实践》中的画布章节。

例如，此处简单介绍 number_label 控件中的绘制函数：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
static ret_t number_label_paint_text(widget_t* widget, canvas_t* c, wstr_t* text) {
  /* 获取样式控件当前使用的样式属性：比如字体颜色、格式、字号、边距等等 */
  style_t* style = widget->astyle;
  color_t tc = style_get_color(style, STYLE_ID_TEXT_COLOR, trans);
  int32_t margin = style_get_int(style, STYLE_ID_MARGIN, 0);
  ......

  /* 设置画布的字体颜色和水平、垂直布局 */
  canvas_set_text_color(c, tc);
  canvas_set_text_align(c, align_h, align_v);

  /* 设置画布使用的字体并计算文本宽度 */
  canvas_set_font(c, font_name, font_size);
  int_part_width = canvas_measure_text(c, text->str, int_part_len);
```

```
  /* 其他画布相关操作 */
  ......

  /* 绘制文本 */
  canvas_draw_text(c, text->str + int_part_len, decimal_part_len, x, y);

  return RET_OK;
}

static ret_t number_label_on_paint_self(widget_t* widget, canvas_t* c) {
  /* 前置操作 */
  ......
  /* 绘制控件文本 */
  return number_label_paint_text(widget, c, text);
}
```

注：控件绘制函数主要就是依赖 AWTK 提供的 canvas 和 vgcanvas 实现，详情可以参考《AWTK 开发实践》，并且在 AWTK 的内部控件以及 Designer 推荐的自定义控件中都有大量的相关示例，均可参阅，此处便不再详细介绍了。

在某些控件的绘制中中，可能需要对绘制区域进行裁剪，此时会用到裁剪区的相关功能，AWTK 在绘制之前会先设置好裁剪区，在控件设置裁剪区时需要对两个裁剪区取交集，并且在绘制完成后恢复裁剪区，使用时可以参考以下代码进行实现：

```
static ret_t scroll_view_on_paint_children(widget_t* widget, canvas_t* c) {
  rect_t r;
  rect_t r_save;
  vgcanvas_t* vg = NULL;
  vg = canvas_get_vgcanvas(c);
  r = rect_init(c->ox, c->oy, widget->w, widget->h);

  /* 获取当前画布的裁剪区 */
  canvas_get_clip_rect(c, &r_save);
  /* 将当前裁剪区与控件本身所需的裁剪区取交集 */
  r = rect_intersect(&r, &r_save);
  /* 如果有使用 vgcanvas 也要进行 vgcanvas 裁剪区的设置*/
  if (vg != NULL) {
    vgcanvas_save(vg);
    /* 设置裁剪区 */
    vgcanvas_clip_rect(vg, (float_t)r.x, (float_t)r.y, (float_t)r.w, (float_t)r.h);
  }
  /* 设置裁剪区 */
  canvas_set_clip_rect(c, &r);

  /* 进行绘制 */
  /*...*/
  /* 结束绘制*/

  /* 恢复裁剪区 */
```

```
  canvas_set_clip_rect(c, &r_save);
  if (vg != NULL) {
    vgcanvas_clip_rect(vg, (float_t)r_save.x, (float_t)r_save.y, (float_t)r_save.w,
                       (float_t)r_save.h);
    vgcanvas_restore(vg);
  }
  return RET_OK;
}
```

3.2.6 控件的事件处理函数

如果想让控件在收到事件时做出相应的处理，那么就需要重载控件虚表结构体（widget_vtable_t）中 on_event 函数，此处以 number_label 控件为例，该函数的基本框架通常如下：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
ret_t number_label_on_event(widget_t* widget, event_t* e) {
  ret_t ret = RET_OK;
  number_label_t* number_label = NUMBER_LABEL(widget);
  /* 前置处理 */
  ......
  /* 根据不同的事件类型添加处理代码 */
  switch (e->type) {
    case /* 事件类型 */: {
      /* 事件处理代码 */
      break;
      }
    case EVT_KEY_DOWN: {
      key_event_t* evt = (key_event_t*)e;
      if (!(number_label->readonly)) {
       ......
      }
      break;
    }
    ......
  }

  return ret;
}
```

常用的控件事件类型详见下表，更多的请参考 events.h[10]。

事件类型	说明
EVT_POINTER_DOWN	指针按下事件
EVT_POINTER_MOVE	指针移动事件
EVT_POINTER_UP	指针抬起事件
EVT_KEY_DOWN	键按下事件

[10] https://github.com/zlgopen/awtk/blob/master/src/base/events.h

续上表

事件类型	说明
EVT_KEY_UP	键抬起事件
EVT_FOCUS	得到焦点事件
EVT_BLUR	失去焦点事件
等等…	

另外，我们需要注意 on_event 函数的返回值：

- 返回 RET_OK 表示事件处理完毕后继续向上传递，即控件的父控件会收到该事件。
- 返回 RET_STOP 表示事件处理完毕之后停止传递。

3.2.7 设置/获取控件的属性

控件虚表结构体（widget_vtable_t）中 set_prop 函数和 get_prop 函数分别用来设置和获取控件的属性，在调用 widget_set_prop() 函数和 widget_get_prop() 函数时，会回调到重载的控件函数中，此处以 number_label 控件为例，这两个重载函数的实现代码如下：

注: 在控件的 set_prop 函数和 get_prop 函数中，属性的值采用 value_t 类型保存，该类型的定义与用法请查阅：value.h[11]。

```
/* awtk-widget-number-label/src/number_label/number_label.c */
static ret_t number_label_set_prop(widget_t* widget, const char* name, const value_
↪t* v) {
  number_label_t* number_label = NUMBER_LABEL(widget);
  return_value_if_fail(widget != NULL && name != NULL && v != NULL, RET_BAD_
↪PARAMS);

  if (tk_str_eq(name, WIDGET_PROP_MIN)) {
    number_label->min = value_double(v);
    return RET_OK;
  } else if (tk_str_eq(name, WIDGET_PROP_MAX)) {
    number_label->max = value_double(v);
    return RET_OK;
  }
  ......
  return RET_NOT_FOUND;
}
```

```
/* awtk-widget-number-label/src/number_label/number_label.c */
static ret_t number_label_get_prop(widget_t* widget, const char* name, value_t* v)
↪{
  number_label_t* number_label = NUMBER_LABEL(widget);
  return_value_if_fail(widget != NULL && name != NULL && v != NULL, RET_BAD_
↪PARAMS);

  if (tk_str_eq(name, WIDGET_PROP_MIN)) {
    value_set_double(v, number_label->min);
    return RET_OK;
```

[11] https://github.com/zlgopen/awtk/blob/master/src/tkc/value.h

```
  } else if (tk_str_eq(name, WIDGET_PROP_MAX)) {
    value_set_double(v, number_label->max);
    return RET_OK;
  }
  ......
  return RET_NOT_FOUND;
}
```

3.2.8 控件的销毁回调函数

控件虚表结构体（widget_vtable_t）中 on_destroy 函数会在控件被销毁时回调执行，用于释放控件内部申请的内存或进行其他析构操作。

此处以 number_label 为例，该控件上保存了一个 malloc 出来 format 属性，在控件销毁时必须释放这块内存，代码如下：

```
/* awtk-widget-number-label/src/number_label/number_label.c */
static ret_t number_label_on_destroy(widget_t* widget) {
  number_label_t* number_label = NUMBER_LABEL(widget);
  return_value_if_fail(widget != NULL && number_label != NULL, RET_BAD_PARAMS);

  TKMEM_FREE(number_label->format);

  return RET_OK;
}
```

3.3 设置控件的初始属性和缺省样式

默认情况下，从 Designer 的控件列表的”自定义”分组中拖出一个控件，其初始属性将采用控件构造函数中值，并且没有缺省样式，这会导致控件的绘制函数无法获取对应的样式属性，比如文本颜色、文本大小、边距等等，画出来的控件就是全透明的。

如果需要指定控件的初始属性和缺省样式，可以在控件类型结构体（class）注释上补充如下格式的注释：

```
/**
...
 * ```xml
 * <!-- ui -->
 * 控件初始属性的 xml 描述（如果描述中包含子控件，会同时创建）
 * ```
...
 * ```xml
 * <!-- style -->
 * 控件默认样式的 xml 描述（如果描述中包含其它控件的样式，会同时添加到 default.xml
↪样式文件）
 * ```
...
 */
```

在 number_label 控件中，它的初始属性和缺省样式代码如下：

```
/* awtk-widget-number-label/src/number_label/number_label.h */
/**
 * @class number_label_t
 * @parent widget_t
 * @annotation ["scriptable","design","widget"]
 * 数值文本控件。
 *
 * 在 xml 中使用"number\_label"标签创建数值文本控件。如：
 *
 * ```xml
 * <!-- ui -->
 * <number_label x="c" y="50" w="24" h="100" value="40" format="%.4lf" decimal_
↪font_size_scale="0.5"/>
 * ```
 *
 * 可用通过 style 来设置控件的显示风格，如字体的大小和颜色等等。如：
 *
 * ```xml
 * <!-- style -->
 * <number_label>
 *   <style name="default" font_size="32">
 *     <normal text_color="black" />
 *   </style>
 *   <style name="green" font_name="led" font_size="32">
 *     <normal text_color="green" />
 *   </style>
 * </number_label>
 * ```
 */
typedef struct _number_label_t {
  ......
} number_label_t;
```

按照以上注释代码，在 Designer 中创建 number_label 控件时，其默认的 UI 代码如下：

```
<!-- UI 文件 -->
<number_label x="c" y="50" w="24" h="100" value="40" format="%.4lf" decimal_font_
↪size_scale="0.5"/>
```

注: 如果采用拖拽的方式创建控件，其 x、y 属性会被修改为鼠标抬起时的位置。

创建 number_label 控件后，注释中的缺省样式代码会被拷贝到项目的全局样式文件（default.xml），代码如下：

```
<!-- design/default/styles/default.xml -->
<number_label>
  <style name="default" font_size="32">
    <normal text_color="black" />
  </style>
```

```
  <style name="green" font_name="led" font_size="32">
    <normal text_color="green" />
  </style>
</number_label>
```

3.4 实现示例程序

开发完自定义控件之后，通常都需要在控件的示例程序中验证控件功能是否正常。使用 Designer 创建自定义控件项目时，会自动生成 demos 目录，该目录存放自定义控件的示例程序代码，用于展示控件效果并给用户提供简单的示例用法。

示例程序的实现方式与普通的 AWTK 应用程序一样，可以在界面上通过拖拽创建自定义控件，设置控件的属性和样式，编辑 demos 目录下的程序代码，打包资源并编译运行，如下图所示。

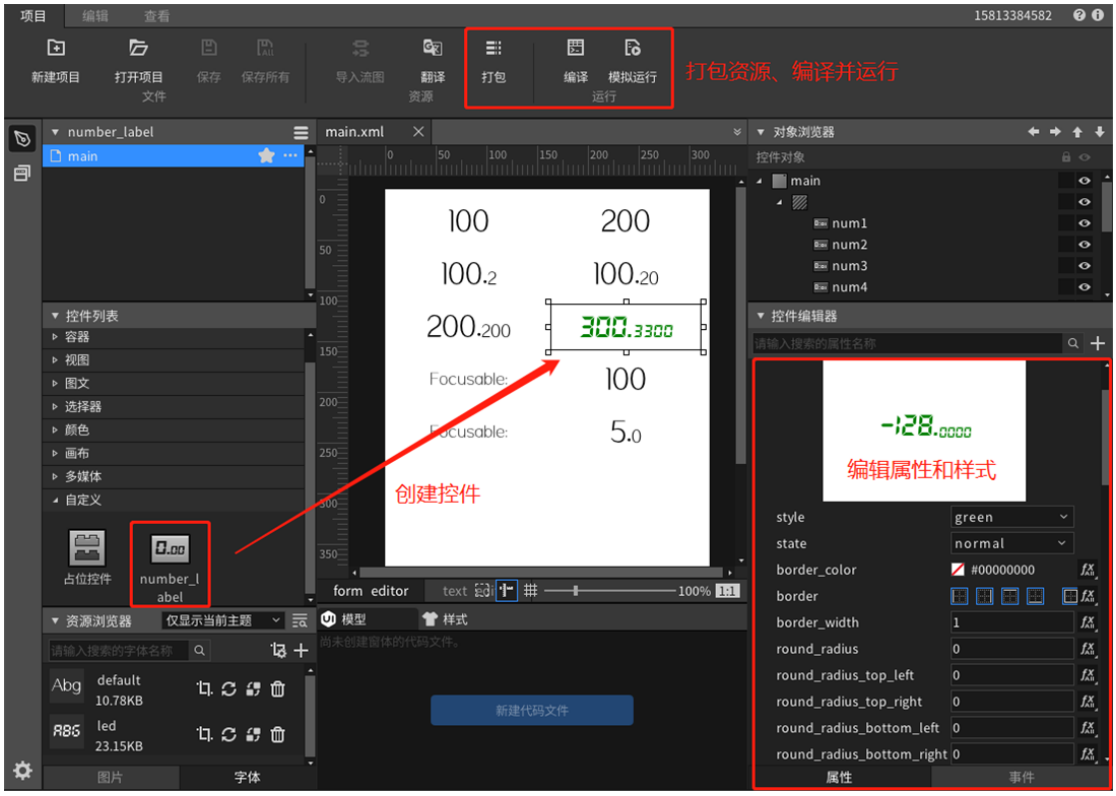

图 3.2　实现示例程序

3.5 完善单元测试

使用 Designer 创建自定义控件项目时，会自动生成 tests 目录，该目录存放自定义控件的单元测试代码，用于测试控件逻辑代码的准确性与稳定性，它们默认基于 GTest（Google Test）框架[12] 实现，具体的使用方法可参考 GTest 的 官方文档[13]。number_label 控件的单元测试代码可参阅：`tests/number_label_test.cc`，如无需测试可跳过本小节，此处不过多赘述。

[12] https://github.com/google/googletest
[13] http://code.google.com/p/googletest/w/list

3.6 自定义控件的相关图标

如果需要修改自定义控件在 Designer 中的图标，请将图标存放到指定位置，详见下文，这些图标如果不指定，则统一显示缺省图标。

3.6.1 插件图标

插件图标指 Designer 的”插件管理”页面上用于标识自定义控件或者描述其功能的图标，大小为 60*60 像素，默认为自定义控件项目的 docs/images/widget_preview.png 文件。

number_label 控件的效果如下图所示：

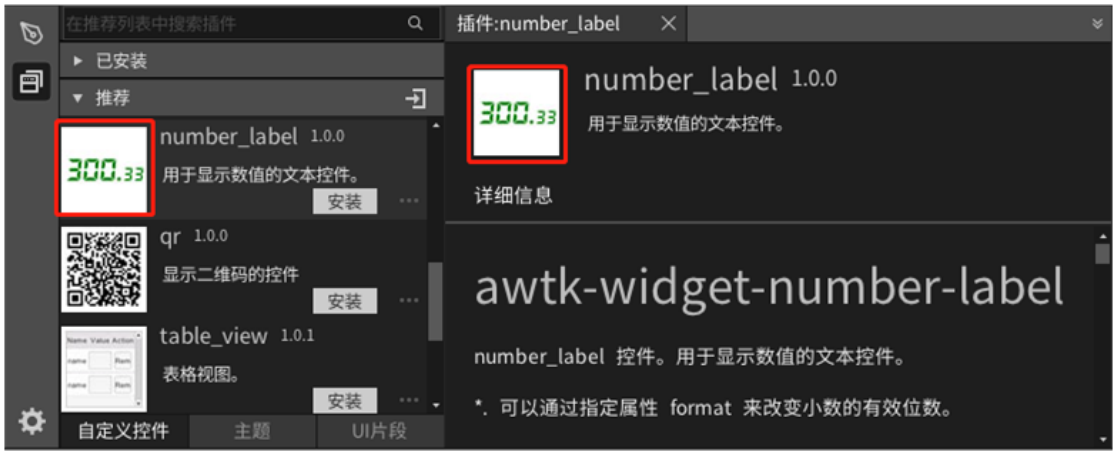

图 3.3 插件图标

3.6.2 控件列表上的图标

控件列表上的图标指 Designer 的控件列表上该控件显示的图标，大小为 48*48 像素，默认为自定义控件项目的 docs/images/widget_list.png 文件。

number_label 控件的效果如下图所示：

图 3.4 控件列表上的图标

注: 如果自定义控件库中包含多个控件，可以用“widget_list_”+ 控件类型名的形式，为控件单独指定图标，比如”widget_list_number_label.png”。

3.6.3 对象浏览器上的图标

对象浏览器上的图标指 Designer 的对象浏览器上该控件对象左侧显示的图标，大小为 16*16 像素，默认为自定义控件项目的 docs/images/widget_obj.png 文件。

number_label 控件的效果如下图所示：

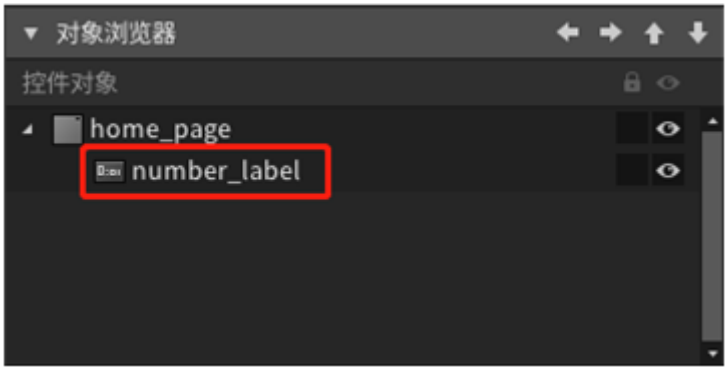

图 3.5 对象浏览器上的图标

注: 如果自定义控件库包含多个控件，可以用“widget_obj_”+控件类型名的形式，为控件单独指定图标，比如”widget_obj_number_label.png”。

3.7 注册自定义控件

在 AWTK 项目中，使用自定义控件之前，需要先注册自定义控件，如果是在 Designer 中导入并安装自定义控件，那么 Designer 会自动添加这些注册代码。这里我们仅简单介绍一下注册的方法。

例如，我们在一个新建的 AWTK 项目中安装 number_label 控件，自定义控件代码会被放在项目的 3rd 目录下，注册控件的步骤如下：

步骤一：在程序初始化时，将自定义控件类型注册到 AWTK 的控件工厂，代码如下：

```
/* app/src/application.c */
#include "../3rd/awtk-widget-number-label/src/number_label_register.h"

/**
 * 注册自定义控件
 */
static ret_t custom_widgets_register(void) {
  number_label_register();  /* 注册 number_label 控件 */
  return RET_OK;
}
......
/**
 * 初始化程序
 */
ret_t application_init(void) {
  custom_widgets_register();
  ......
  return navigator_to(APP_START_PAGE);
```

```
}
```

```
/* app/3rd/awtk-widget-number-label/src/number_label_register.c */
ret_t number_label_register(void) {
  /* 将 number_label 控件类型注册到 AWTK
↪的控件工厂（将控件类型与对应的构造函数绑定） */
  return widget_factory_register(widget_factory(), WIDGET_TYPE_NUMBER_LABEL,
↪number_label_create);
}
```

步骤二：在项目的编译脚本中添加自定义控件库，代码如下：

```
# app/SConstruct
......
# 设置自定义控件库的路径和名称
CUSTOM_WIDGET_LIBS = [{
    "root" : '3rd/awtk-widget-number-label', # 库路径
    'shared_libs': ['number_label'],         # 动态库名称
    'static_libs': []
}]

DEPENDS_LIBS = CUSTOM_WIDGET_LIBS + []

# 添加自定义控件库
helper = app.Helper(ARGUMENTS)
helper.set_deps(DEPENDS_LIBS)

app.prepare_depends_libs(ARGUMENTS, helper, DEPENDS_LIBS)
helper.call(DefaultEnvironment)
......
```

4. 使用继承实现自定义控件

在使用 AWTK 进行 UI 设计的时候，可能会出现某些控件的功能只能够满足一部分需求，这种时候就可以使用继承来创建一个自定义控件，本章将以 usb_camera[14] 为例讲解如何通过继承实现一个自定义控件。关于如何完整地开发一个自定义控件请参考开发自定义控件 章节，本章只介绍继承实现与普通实现的区别，不做具体的实现流程介绍。

4.1 完成自定义控件的定义

```
typedef struct _usb_camera_t {
  /* 继承的类 */
  mutable_image_t base;
  /* 子类自身特有属性定义 */
  ....

} usb_camera_t;
```

以上为 usb_camera 控件结构体定义的一部分，usb_camera 类继承自 mutable_image 类，所以在定义时需要将 mutable_image 类作为结构体的第一项。需要注意的是，只能进行单继承，不允许多继承。

4.2 完成自定义控件的虚表结构体定义

以 usb_camera 为例，它的虚表结构体定义如下所示，其中与普通自定义控件不同的是 get_parent_vt 属性，该属性在填写时请按照以下格式，将父类控件名称填写上。

```
TK_DECL_VTABLE(usb_camera) = {.size = sizeof(usb_camera_t),
                              .type = WIDGET_TYPE_USB_CAMERA,
                              .clone_properties = s_usb_camera_properties,
                              .persistent_properties = s_usb_camera_properties,
                              /* 获取父类虚表结构体 */
                              .get_parent_vt = TK_GET_PARENT_VTABLE(mutable_image),
                              .create = usb_camera_create,
                              .on_paint_self = usb_camera_on_paint_self,
                              .set_prop = usb_camera_set_prop,
                              .get_prop = usb_camera_get_prop,
                              .on_event = usb_camera_on_event,
                              .on_destroy = usb_camera_on_destroy};
```

[14] http://jibulab.com/awtk/awtk-widget-usb-camera

4.3 实现自定义控件的创建与销毁函数

以 usb_camera 控件的创建函数与销毁函数为例，在创建函数结束前调用父类的初始化函数，在销毁函数结束前调用父类的销毁函数。

```
/* 控件的创建函数 */
widget_t *usb_camera_create(widget_t *parent, xy_t x, xy_t y, wh_t w, wh_t h) {
  widget_t *widget =
      widget_create(parent, TK_REF_VTABLE(usb_camera), x, y, w, h);
  usb_camera_t *usb_camera = USB_CAMERA(widget);
  return_value_if_fail(usb_camera != NULL, NULL);
  /* 子类的初始化过程 */
  ......
  /* 父类的初始化函数 */
  mutable_image_init(widget);
  return widget;
}
/* 控件的销毁函数 */
static ret_t usb_camera_on_destroy(widget_t *widget) {
  usb_camera_t *usb_camera = USB_CAMERA(widget);
  return_value_if_fail(usb_camera != NULL, RET_BAD_PARAMS);
  /* 子类的销毁过程 */
  ......
  /* 父类的销毁函数 */
  return mutable_image_on_destroy(widget);
}
```

4.4 实现自定义控件的其他函数

关于其他自定义控件需要实现的函数，请参考开发自定义控件 章节，本章就不再赘述，在其他的函数里用户也可以按需使用父类的各种函数来简化子类的函数实现。

5. 使用占位控件

如果实现自定义控件时由于各种原因并没有遵循 AWTK 自定义控件规范[15]，比如直接在 AWTK 项目的 src 目录下添加控件的源文件，并手动调用 `widget_factory_register()` 函数注册控件。那么可以使用 Designer 中的占位控件来表示这些不规范的自定义控件，这样就可以直接在 Designer 中编辑控件了，操作步骤如下：

- 展开控件列表中的自定义分组，选中”占位控件”并拖曳到编辑区；
- 选中占位控件后，展开控件编辑器中控件的名称分组，修改 type 属性为目标控件类型；
- 如果要添加额外的属性，可以点击”+”按钮进行添加；
- 如果要添加额外的样式属性，可以点击样式分组右侧的按钮打开菜单，再点击”添加自定义样式属性”菜单项进行添加。

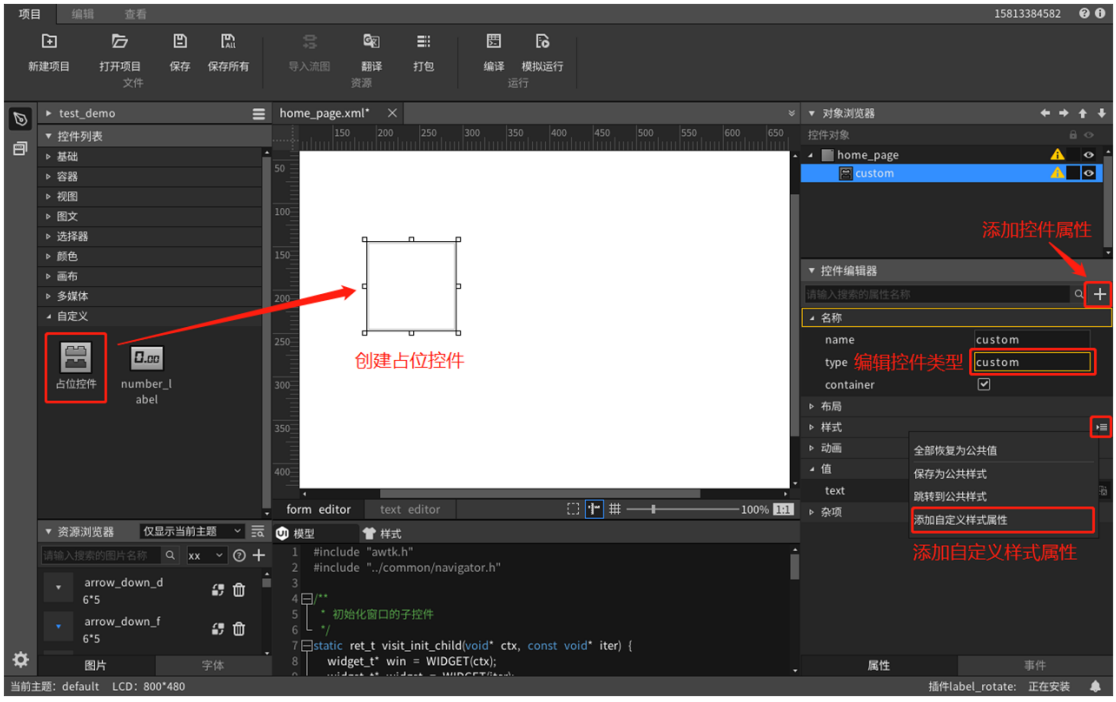

图 5.1　使用占位控件

由于占位控件并不是真实存在的，因此在编辑区或者预览时均无法看到具体效果。

注: 后续如果有同类型的控件安装到项目中，则对应的占位控件会自动更新为新安装的控件，此时可以看到控件的具体效果。

[15] https://github.com/zlgopen/awtk/blob/master/docs/custom_widget_rules.md

诚信共赢，持续学习，客户为先，专业专注，只做第一

广州致远电子股份有限公司

更多详情请访问 www.zlg.cn

欢迎拨打全国服务热线 400-888-4005