SigmaStar IDE
使用参考

REVISION HISTORY

Revision No.	Description	Date
{000001}	• {Initial release}	{05/06/2019}

TABLE OF CONTENTS

1. IDE 介绍

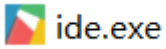
1.1. 使用须知

1. 在开始使用我们的屏之前，得先把开发工具安装好;
2. 打开工具，简单的熟悉一下开发环境，可以尝试着新建工程，浏览一下有哪些内容，不必深究里边的细节;
3. 新建的工程 UI 上面是没有东西的，我们可以先拖放个文本控件上去，然后编译、运行到我们的屏上看效果; **注意：如果你的是 Wi-Fi__33 版本的机器，务必配置好 ADB IP，才能正常下载。**

1.2. 初次使用工具

如果你已经成功安装了本工具，那么在你的桌面上，找到 

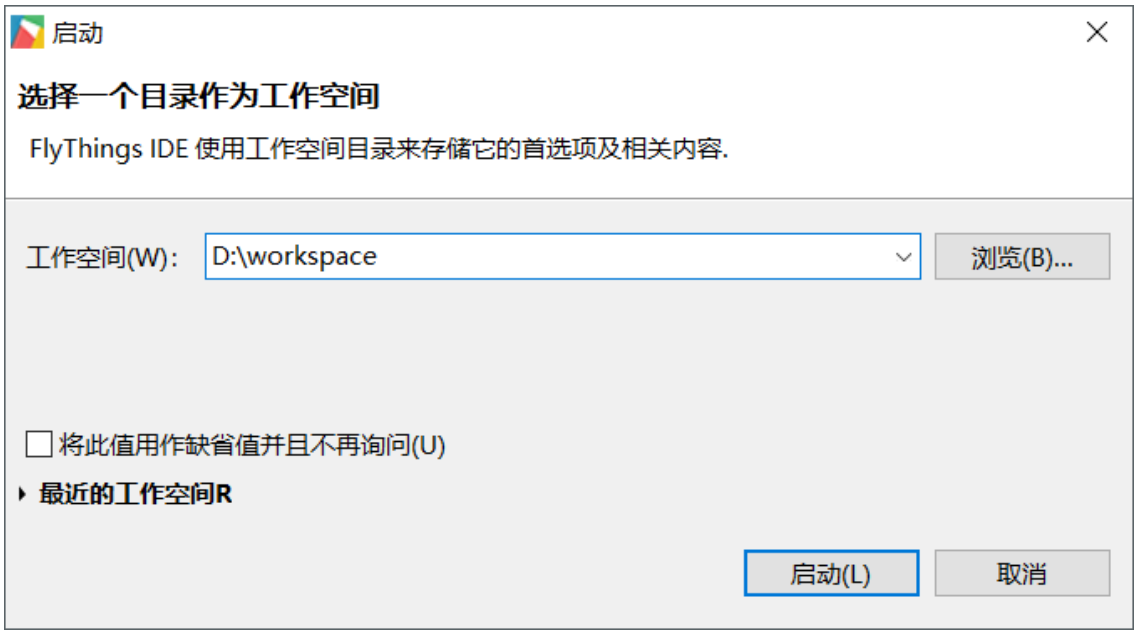
快捷方式，双击运行。

如果你删除了快捷方式，你还可以在安装目录中的 `bin` 文件夹下，找到 

直接运行。

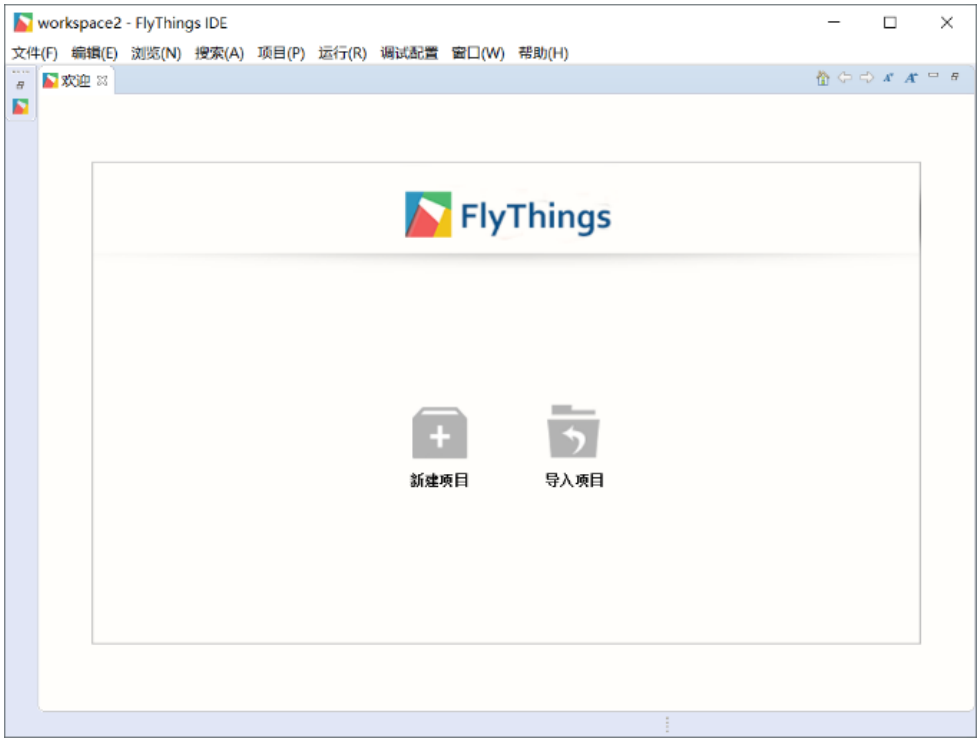
当你运行工具之后会弹出如下界面：

- **工作空间** 用于存储相关设置及历史纪录，你可以将它理解为一个容器，它可以同时管理多个项目，这样不用同时运行多个开发工具。



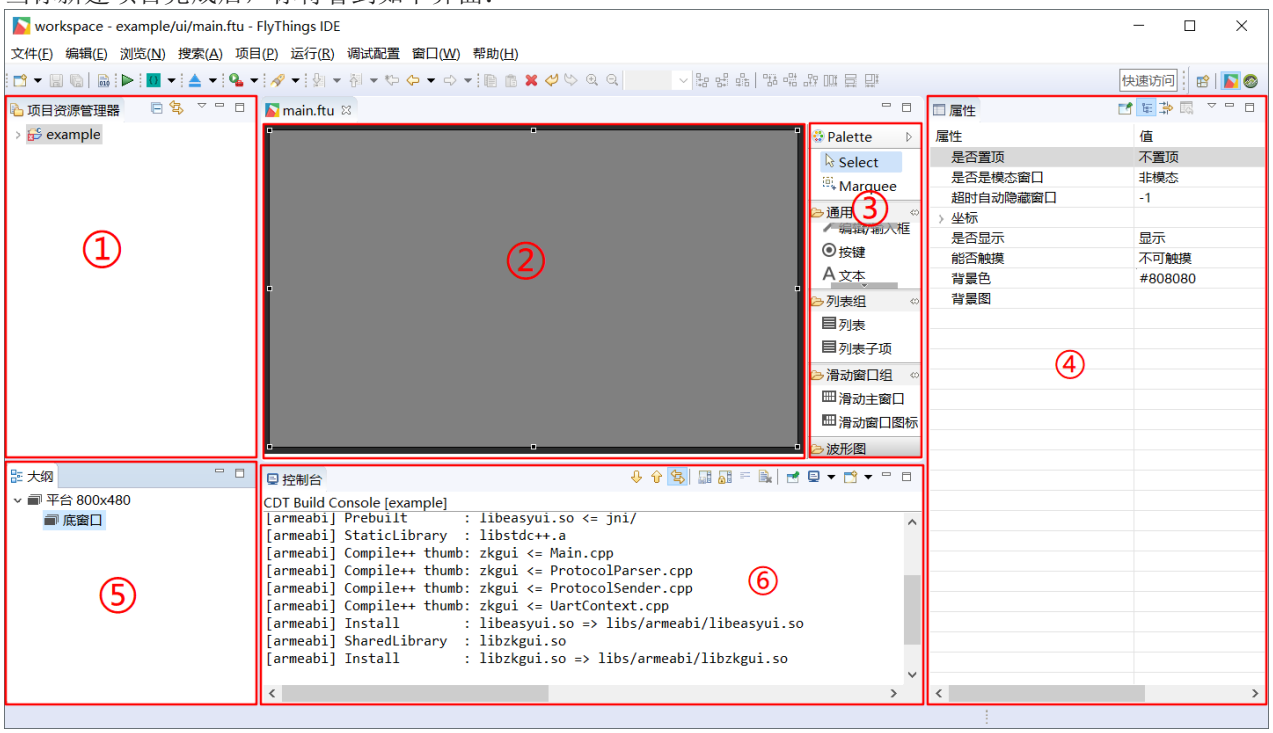
如果你第一次打开该软件，或者选择了一个新的工作空间，那么，你会看到下图这样的欢迎界面。 它提供了两个快捷功能： 新建项目 和 导入项目。

- **新建项目**
  它会逐步引导你如何新建一个项目。
- **导入项目**
  通过它，你可以将现有的项目导入到目前的工作空间，继续之前的开发。



1.3. IDE 工作区域使用介绍

当你新建项目完成后，你将看到如下界面：



编辑器大致分为六个区域。分别的作用如下：

① **区域 - 项目资源管理器** 它将项目文件夹内的资源文件、代码文件等以树形图的形式显示。你可以自由展开/收起，双击文件可以直接打开编辑。

② **区域 - UI 编辑框** 主要负责 UI 界面的编辑和即时预览，他是开发中主要的操作区域

③ **区域 - 控件画板** 它包含了所有内置的控件，你可以点击选择需要的的控件，将其拖拽到 **②区域** 即可完成控件的创建。

④ **区域 - 属性表** 当你在 **区域②** 中选择了某个控件后，它的所有属性将在这里以表格的形式显示，你可以在表格中自定义修改。

⑤ **区域 - 大纲视图** 它将已经创建的所有控件以树形图的形式展示；同样支持自由展开/收起； 可以清晰的了解控件之间的层级关系；并且可以直接拖拽某个节点，快速调整层级位置；双击节点可以快速显示/隐藏控件，这个在层级关系复杂后，非常好用。

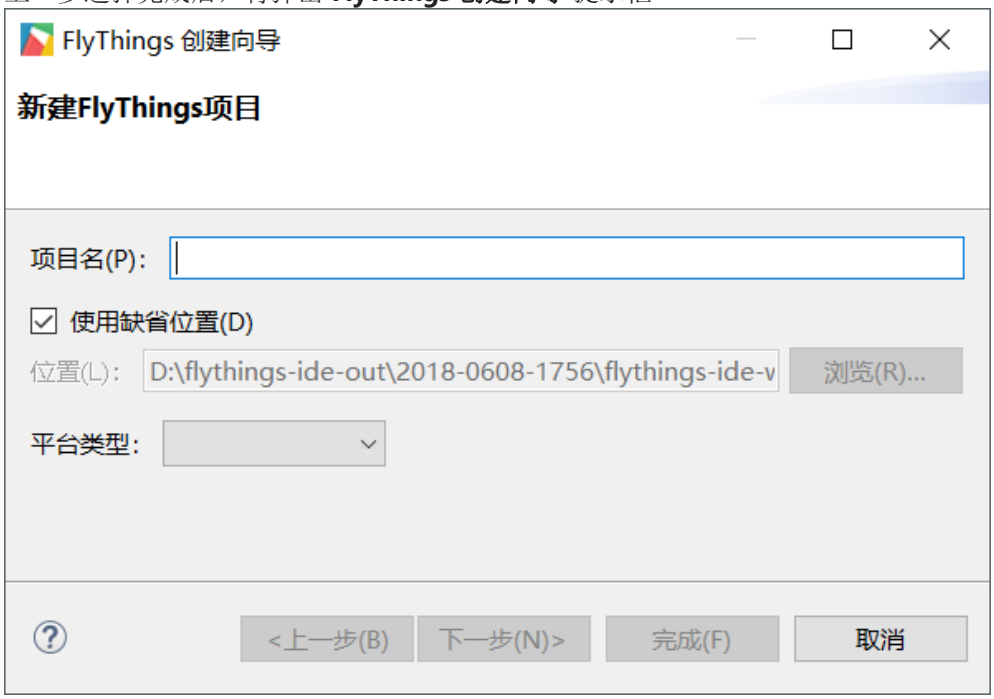
⑥ **区域 - 控制台** 编译代码时，这个位置将输出编译日志。如果编译失败，双击**错误提示内容**，可直接跳转到对应代码。

有了以上的基础后，现在，我们可以正式开始开发。

1.4. 创建和编译工程

新建一个项目具体步骤如下：

1. 在编辑器顶部的菜单栏中，依次选择 **文件 -> 新建 -> FlyThings 项目**
2. 上一步选择完成后，将弹出 **FlyThings 创建向导** 提示框



按要求填写新建项目相关的参数。 这些参数分别是:

1) **项目名**
   项目的名称；可以是字母、数字的组合，不能出现中文及空格。
2) **位置**
   项目的存储路径；同样不推荐路径中出现中文，防止编译异常。

3) **平台类型**

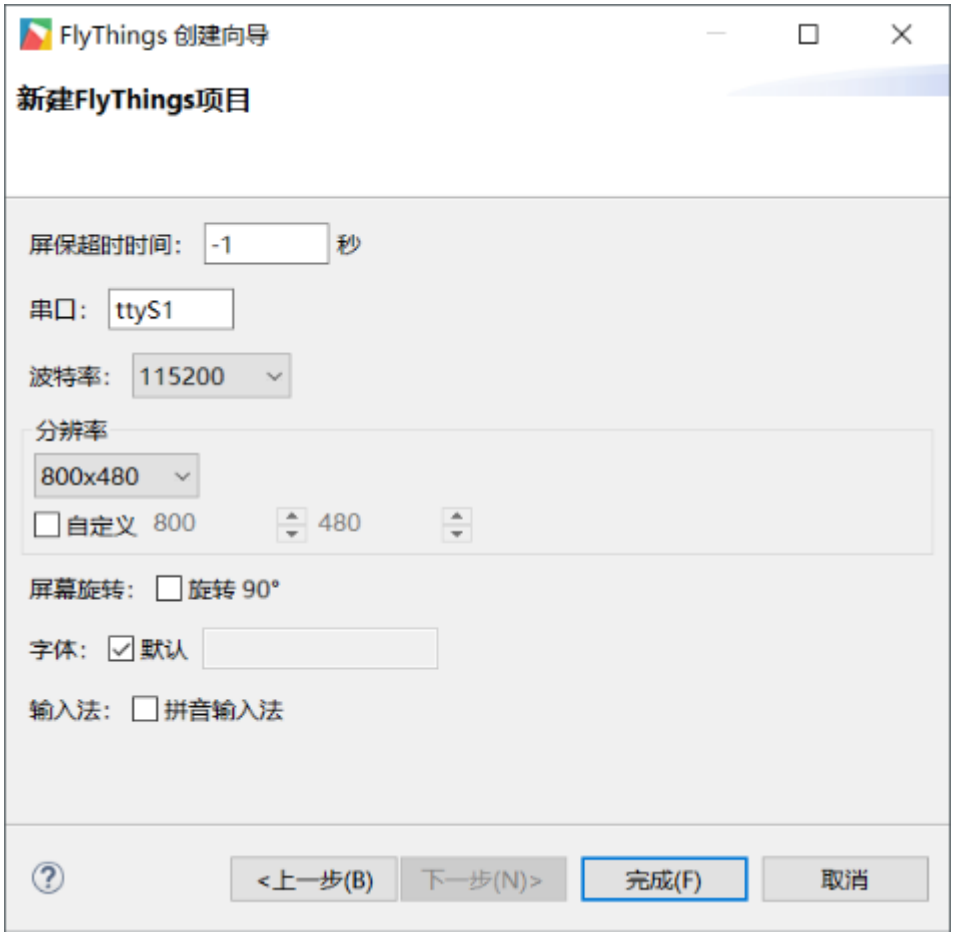
根据购买的串口屏，选择相应的平台，目前有

a. **Z11S**

b. **Z6S**

规范填写如上必须参数后，你可以直接选择 **完成**，来快速完成创建。但是现在，我们选择 **下一步** 来自定义更多的参数。

3. 点击下一步之后，我们将看到更多的参数定义



属性含义和作用：

屏保超时时间

系统提供屏保的功能。如果在指定的时间内，串口屏没有任何触摸操作，或者你没有通过代码重置屏保计时，那么，系统将自动进入屏保。

串口

指定通讯串口，一般情况下不需要修改。

波特率

指定通讯串口的波特率。

分辨率

以像素为单位，指定屏幕的宽高。

屏幕旋转

针对某些屏幕坐标轴方向不同，可勾选该选项，将显示内容旋转 90°，达到正常显示。

字体

支持自定义字体，如果你不满意默认字体，可取消默认，再选择你的字体文件。

输入法

如果你有中文输入的需求，可以勾选它， 配合编辑输入框控件，就可以解决中文输入了。

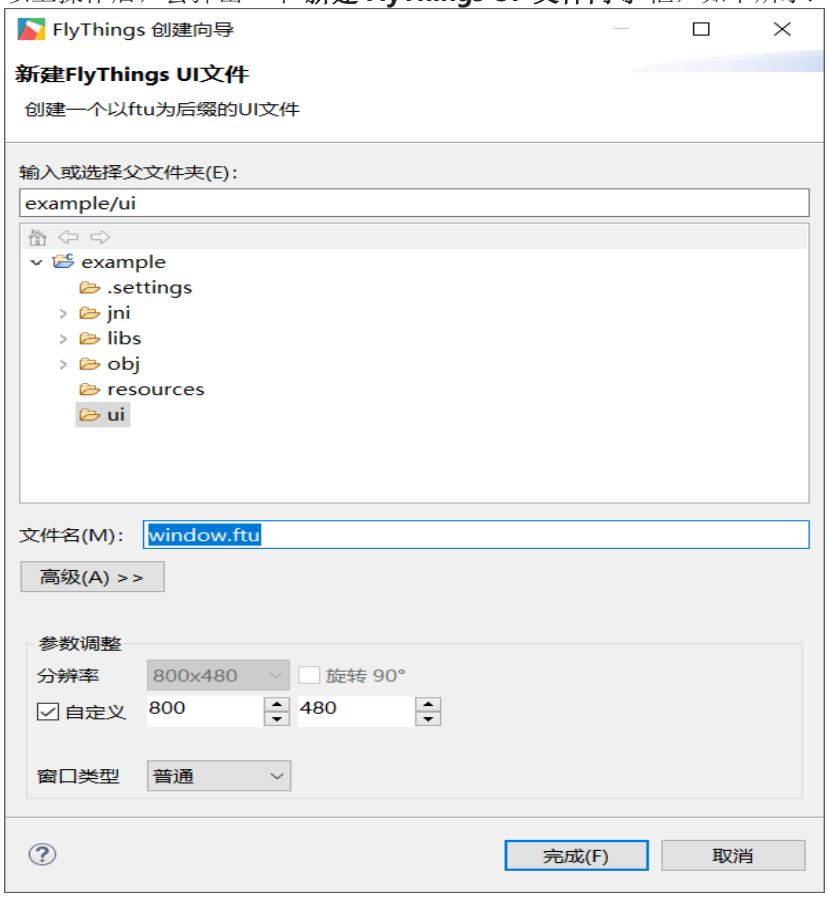
以上属性后续都可以再次修改，所以不必过分担心填写错误。属性都填写确认后，点击 **完成** 结束创建，创建过程会花费些许时间，耐心等待。

导入项目步骤如下：

1. 找到工具顶部的菜单栏，依次选择菜单 **文件** -> **导入**
2. 在弹出框中， 依次选择 **常规** -> **现有项目到工作空间中**， 再选择 **下一步。**
3. 在弹出框中选择 **浏览** 按钮，指定需要导入项目的文件夹。 点击 **确定**, 它会自动解析文件夹内包含的项目。
4. 如果项目文件没有损坏，你可以看到已经识别出来的项目，然后直接点击 **完成** ，导入的项目就会出现在项目资源管理器中，你可以继续查看/编辑它。

新建 UI 文件：

1. 在项目资源管理器中，展开需要创建 UI 文件的项目，选中项目下的 **ui** 文件夹，然后点击右键，在弹出菜单中，依次选择菜单 **新建** -> **FlyThings UI 文件.**
2. 以上操作后，会弹出一个 **新建 FlyThings UI 文件向导** 框，如下所示:



有三个参数需要指定：

文件名

你需要指定 UI 文件的文件名，文件名以字母、数字命名，以 **ftu** 为后缀名。

分辨率

你可以通过调整分辨率来控制 UI 界面的宽高，以像素为单位；

窗口类型

目前有四种窗口类型，分别为 **普通、状态栏、导航栏、屏保。**一般情况下我们保持 **普通** 选项就可以。

确定参数后，选择 **完成** ，创建过程结束。你可以在项目资源管理器中，项目的 ui 文件夹下看到新创建的 UI 文件。

编译项目，有以下三种方式：

1. 从项目资源管理器选择编译

   在项目资源管理器中，左键选中需要编译的项目名，然后右键，在弹出菜单中，选择编译该项目。

2. 通过工具栏选择编译

   在软件顶部的工具栏上有编译的快捷方式。 同样，先在项目资源管理器中，左键选中需要编译的项目名，然后再工具栏上找到 

该绿色三角符号按钮，点击它，即可编译选中的项目。

3. 通过快捷键编译

   使用快捷键 **Ctrl + Alt + Z** 快速完成编译。

在项目管理器中，选择项目名，右键点击后，在弹出菜单中选择**下载调试**即可将程序暂时下载到屏上运行。

项目通过编译后，就可以放到真机运行。支持下列**运行方式**：

1. 使用 WIFI 连接设备快速运行，仅支持带 WIFI 的机型
   1) 先进入设备的 WIFI 设置界面，将设备连接到与电脑相同的无线网络，也就是说，电脑和机器必须接入同一个 WIFI。
   2) 无线网络连接成功后，点击 WIFI 设置界面右上角的菜单，查看设备的 IP 地址。
   3) 这时，回到电脑上的开发工具，在菜单栏上，依次选择菜单 **调试配置** -> **ADB 配置**， 在弹出框中，ADB 连接方式选择 **WIFI**， 并填入设备的 IP 地址，应用保存。
   4) 完成连接配置后，再选择下载调试菜单项，它会暂时将项目代码同步到连接的设备中运行。

2. 使用 USB 连接设备快速运行

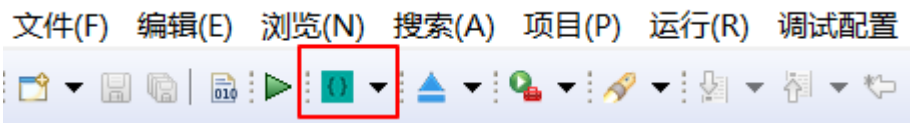
   对于不带 WIFI 功能的型号，几乎都支持 USB 线连接。 **注意：如果带有 WIFI 功能，USB 线连接是无效的。**
   1) 将设备与电脑通过 USB 线连接，如果电脑能将设备识别为 Android 设备，表示连接正常。 如果不能正常连接，电脑提示驱动问题，可尝试下载驱动。
   2) 当电脑正确识别设备后，回到电脑上的开发工具，在菜单栏，依次选择菜单 **调试配置** -> **ADB 配置**，在弹出框中，ADB 连接方式选择 **USB**，应用保存。
   3) 配置完成后，再选择下载调试菜单项，它会暂时将项目代码同步到连接的设备中运行。

3. 从 TF 卡启动，TF 卡仅支持 FAT32 格式

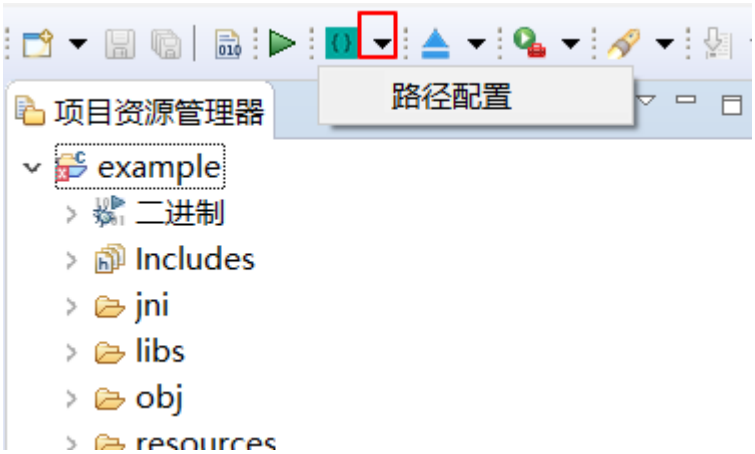
   如果由于其他原因，USB 和 WIFI 都不能正常使用、或者被占用，此可借助 TF 卡，从 TF 卡启动程序。

   首先我们要配置程序的输出目录。

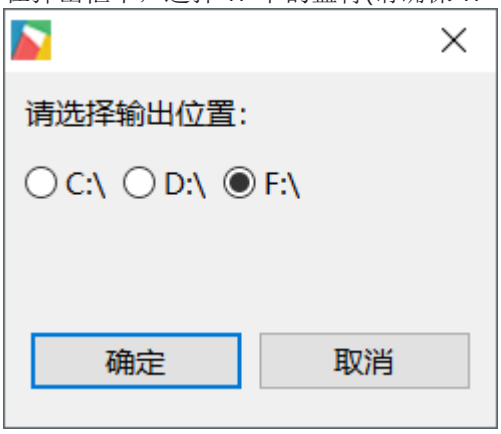
1） 找到工具栏上的这个按钮。



2） 点击旁边的黑色下拉箭头，在弹出的菜单中选择 **路径配置。**



3） 在弹出框中，选择 TF 卡的盘符(请确保 TF 卡能正常使用），点击确定。



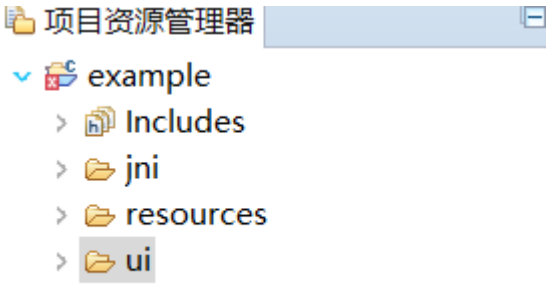
4） 在上面的步骤中，我们配置好了输出目录，现在点击下图中的按钮开始编译，它会将编译结果 打包输出到配置的盘符下。

5） 操作成功后， 将在配置的盘符下 生成 **EasyUI.cfg 、ui、lib、 font** 等目录和文件。

6） 将 TF 卡拔出，插入机器中，将机器重新上电，这时候，系统检测到 TF 卡里的文件，就会启动卡里的程序，而不是系统内的程序。

1.5. 项目代码结构介绍

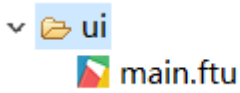
对于一个基本的项目，它的目录结构是这样的：



大致分为 **jni** 、**resources** 、 **ui** 三个文件夹。下面分别解释各个文件夹的作用。

1. ui 文件夹

   展开 **ui** 文件夹



可以看到默认包含了一个 **main.ftu** 文件。**ftu** 是 UI 文件的后缀名。每一个 **ftu** 文件对应一个应用界面。通常，一个应用包含多个界面，所以你需要在 **ui** 文件夹下创建多个 **ftu** 文件。你可以双击打开 **UI 文件** ，并对它进行编辑，并且可以即时预览效果。编辑结束后，你就可以开始 **“编译”** 。

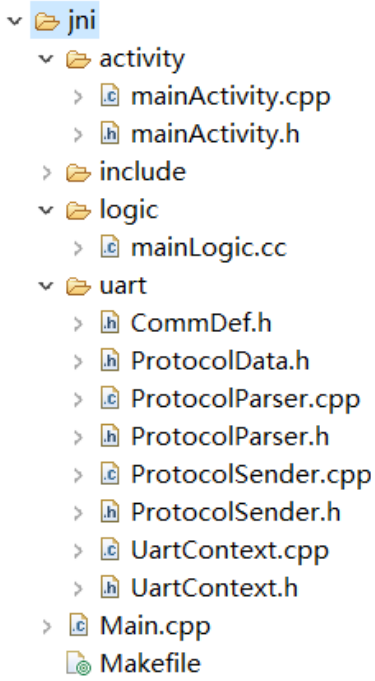
2. resource 文件夹

这个文件夹的内容就比较简单，主要用来存放项目的各种资源文件，包括 图片、字体、输入法配置文件等。如果你还有其他资源文件也可以添加到该文件夹，该文件夹会完全拷贝到机器中。 但是，由于机器自身存储空间的限制，不建议将大文件存放到该目录，更推荐你将较大的资源文件存放到 TF 卡中。

我们可以在代码中获取 resources 目录下的某个文件的绝对路径。

3. jni 文件夹

该文件夹主要为存放代码文件，她还包含了多个部分的代码。我们将 **jni** 文件夹展开

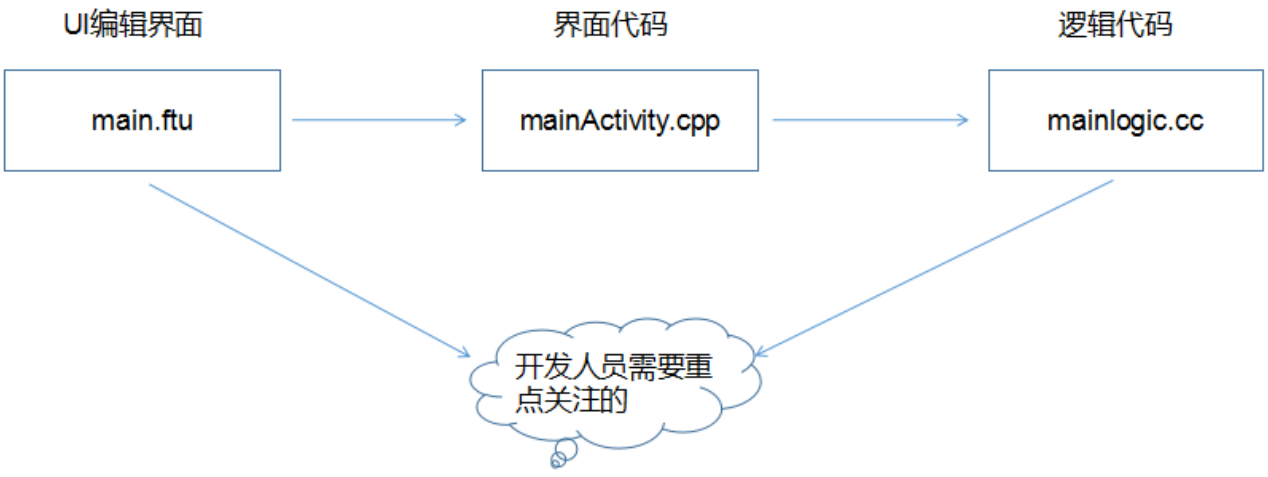


可以看到，它包含了 **activity** 、 **include** 、**logic** 、**uart** 、**Main.cpp**、 **Makefile** 共 6 个部分，每个部分作用如下：

1) activity 文件夹
   存放 UI 文件的基础类代码。每一个 UI 文件，经过编译后，都会生成相同前缀名的 **Activity** 类和 **Logic.cc** 文件。例如：ui 文件夹下有一个 **main.ftu**，那么经过编译后，会生成 **mainActivity.h**、**mainActivity.cpp** 以及 **mainLogic.cc**，**mainActivity** 类会存放在 **activity** 文件夹中，**mainLogic.cc** 文件会存放在 **logic** 文件夹中。
2) logic 子文件夹
   存放具体的逻辑代码。与上面的描述相同，每一个 UI 文件在编译后都会生成相对应前缀名的 **Logic.cc** 文件，**注意：我们的自定义代码，主要就是添加在这些 Logic.cc 文件中**。
3) include 子文件夹
   这里主要存放系统相关的头文件、所有控件相关的头文件。便于编译。
4) uart 子文件夹
   顾名思义，该文件夹存放串口操作相关的代码，包括读写串口，协议解析等。
5) Main.cpp
   整个应用的入口代码，包括选择开机的界面以及一些初始化，一般情况不需要修改该文件。
6) Makefile、Android.mk、Application.mk
   编译配置文件，包含了具体的源码编译过程，一般情况下不需要修改。

编译时，工具会遍历每个 UI 文件，读取 UI 文件中包含的控件。并且为这个控件声明指针变量，在代码中，通过这个指针，就可以操作对应的控件。 指针变量定义在同前缀名的 **Activity.cpp** 文件中。默认的 UI 文件编译生成的 **Logic.cc**。当我们在 UI 文件中添加控件后，再次编译时，工具会根据不同的控件生成不同的关联函数到对应的 **Logic.cc** 文件中。

ftu 文件与代码的对应关系：



在编译通过后，会在项目下生成 **libs** 目录，和 **obj** 目录，它们分别是编译的目标存放目录和编译的中间文件目录，这两个都可以自行清理或者直接删除都没有关系。

2. 控件介绍

2.1. 通用属性

控件的一些通用的属性及设置接口如下：

1. 控件 ID 值

ID 值为控件的唯一标识，每一个 ftu 文件里的控件 ID 值是不允许重名的，不同的 ftu 文件里的控件 ID 值允许重名；设置 ID 值后，编译完会在 activity 目录下对应的头文件中生成相应的宏定义：

```
28 /*TAG:Macro宏ID*/
29 #define ID_MAIN_Button1     20001
30 #define ID_MAIN_Window1     110001
31 /*TAG:Macro宏ID END*/
```

获取控件的 ID 值：

```
/**
 * 该接口定义于控件基类 ZKBase 中
 * 头文件位置： include/control/ZKBase.h
 *
 * 注意： 以下接口如未特殊说明，都表示定义在 ZKBase 类中
 * 所有控件直接或间接的继承了 ZKBase 类，所以，所有的控件都可以调用 ZKBase 类中 public 接口
 */
int getID() const;

/* 操作样例： 点击按钮控件，打印该控件 ID 值 */
static bool onButtonClick_Button1(ZKButton *pButton) {
    int id = pButton->getID();
    LOGD("onButtonClick_Button1 id %d\n", id);
    return false;
}
```

2. 控件位置

我们打开任一 ftu 文件，选中任一控件，在属性框中，我们可以看到**坐标**这一属性，该属性确定了该控件的显示位置：

坐标	
左	71
上	64
宽	327
高	174

其中**左上角的坐标值**是相对于父控件左上角位置：

通过代码设置和获取控件的位置：

```
/* 接口说明 */
// 设置位置
void setPosition(const LayoutPosition &position);
// 获取位置
const LayoutPosition& getPosition();


/* 操作样例 */
// 点击按钮控件，设置该按钮位置
static bool onButtonClick_Button1(ZKButton *pButton) {
    // 左：0，上：0，宽：100，高：200
    LayoutPosition pos(0, 0, 100, 200);
    pButton->setPosition(pos);
    return false;
}

// 点击按钮控件，获取该按钮位置
static bool onButtonClick_Button2(ZKButton *pButton) {
    // pos 的 mLeft、mTop、mWidth、mHeight 变量分别对应的就是坐标值
    LayoutPosition pos = pButton->getPosition();
    return false;
}
```

3. 背景色

背景色	#C0C0C0

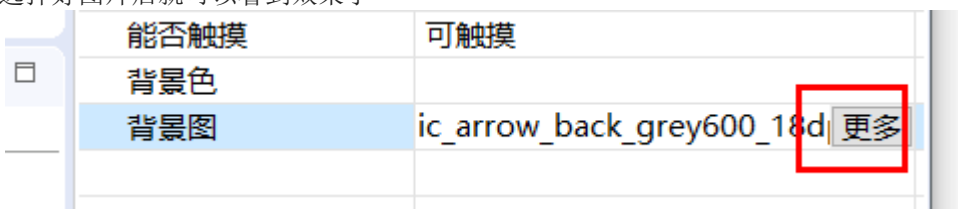
修改一下颜色即可看到效果
代码设置背景颜色：

```
/* color 为-1 时，背景设置为透明；其他颜色值为 0x RGB，颜色值不支持 alpha */
void setBackgroundColor(int color);


/* 操作样例： 点击按钮控件，设置背景颜色为红色 */
static bool onButtonClick_Button1(ZKButton *pButton) {
    pButton->setBackgroundColor(0xFF0000);
    return false;
}
```

4. 背景图

背景图	bg.png

选择好图片后就可以看到效果了



通过代码来设置背景图：

```
/**
 * pPicPath 参数可以有以下两种方式：
 * 1. 绝对路径，如："/mnt/extsd/pic/bg.png"
 * 2. 相对资源目录路径，只需把图片放到项目工程 resources 目录下，编译打包后，就可以使用了，如 resources 目录下有 bg.png 图片，只需设置"bg.png"即可
 */
void setBackgroundPic(const char *pPicPath);


/* 操作样例 */
mButton1Ptr->setBackgroundPic("/mnt/extsd/pic/bg.png"); // 设置绝对路径
mButton1Ptr->setBackgroundPic("bg.png");      // 设置 resources 目录下 bg.png 图片
```

5. 现实与隐藏

是否显示	显示

通过该属性，我们可以设置控件默认是显示还是隐藏状态；双击大纲视图中的控件可以快捷的修改该状态。

另外，我们还可以通过代码动态的设置控件显示和隐藏：

```
void setVisible(BOOL isVisible);
BOOL isVisible() const;

/* 操作样例 */
mButton1Ptr->setVisible(TRUE);  // 显示按钮控件
mButton1Ptr->setVisible(FALSE); // 隐藏按钮控件


/**
 * 窗口控件还可以使用下面的接口，功能一样
 * 头文件位置： include/window/ZKWindow.h
 */
void showWnd();  // 显示窗口
void hideWnd();  // 隐藏窗口
bool isWndShow() const;  // 窗口是否显示

/* 操作样例 */
mWindow1Ptr->showWnd();
mWindow1Ptr->hideWnd();
```

6. 控件状态

对于文本、按钮、列表子项它们有 5 种状态，这里我们需要讲解一下：正常显示状态、按下状态、选中状态、选中按下状态、无效状态，设置完后会影响到控件对应状态的背景颜色、文本颜色及显示图片。

背景颜色设置	
常显颜色	#FF8080
按下时颜色	#FF0000
选中时颜色	#0080FF
选中时按下的颜色	#8080FF
无效时颜色	#C0C0C0

颜色设置	
常显颜色	#FFFFFF
按下时颜色	#00FFFF
选中时颜色	#FF80C0
选中时按下的颜色	#FF80FF
无效时颜色	#808080

图片设置	
常显图片	off_normal.png
按下时图片	off_pressed.png
选中时图片	on_normal.png
选中时按下的图片	on_pressed.png
无效时图片	

按下状态不需要通过代码设置，触摸控件即为按下状态。
选中状态和无效状态的代码操作接口：

```
// 设置选中状态
void setSelected(BOOL isSelected);
BOOL isSelected() const;

/* 操作样例 */
mButton1Ptr->setSelected(TRUE);
 mButton1Ptr->setSelected(FALSE);

/**
 * 无效状态作用说明：控件设置为无效状态情况下，触摸控件没有作用，即不响应按下抬起事件
 */
// 设置无效状态
void setInvalid(BOOL isInvalid);
BOOL isInvalid() const;

/* 操作样例 */
mButton1Ptr->setInvalid(TRUE);
 mButton1Ptr->setInvalid(FALSE);
```

2.2. 控件集

支持下列控件：

1. 文本类 TextView
   支持显示文字，图片，配合定时器可以实现动画效果。

   示例：
   文本显示“Hello World”：

   ```
   mTextview1Ptr->setText("Hello World");
   ```

   文本设置数字和字符：

```
/* 接口定义见头文件: include/control/ZKTextView.h */
void setText(int text);   // 设置数字
void setText(char text); // 设置字符

/* 操作样例 */
mTextview1Ptr->setText(123); // Textview1 控件将显示"123"字符串
mTextview1Ptr->setText('c'); // Textview1 控件将显示'c'字符
```

设置文本颜色：

```
//将控件 Textview1 设置为无效状态；如果属性表中`无效时颜色`属性不为空，则将其设置为指定的颜色，否则无变化。
mTextview1Ptr->setInvalid(true);
//将控件 Textview1 设置为选中状态；如果属性表中`选中时颜色`属性不为空，则将其设置为指定的颜色，否则无变化。
mTextview1Ptr->setSelected(true);
//将控件 Textview1 设置为按下状态；如果属性表中`按下时颜色`属性不为空，则将其设置为指定的颜色，否则无变化。
mTextview1Ptr->setPressed(true);
//将控件 Textview1 设置为红色。
mTextview1Ptr->setTextColor(0xFF0000);
```

动态切换文本背景：

```
static void updateAnimation(){
        static int animationIndex = 0;
        char path[50] = {0};
        snprintf(path, sizeof(path), "animation/loading_%d.png", animationIndex);
        mTextviewAnimationPtr->setBackgroundPic(path);
        animationIndex = ++animationIndex % 60;
}
```

效果图：

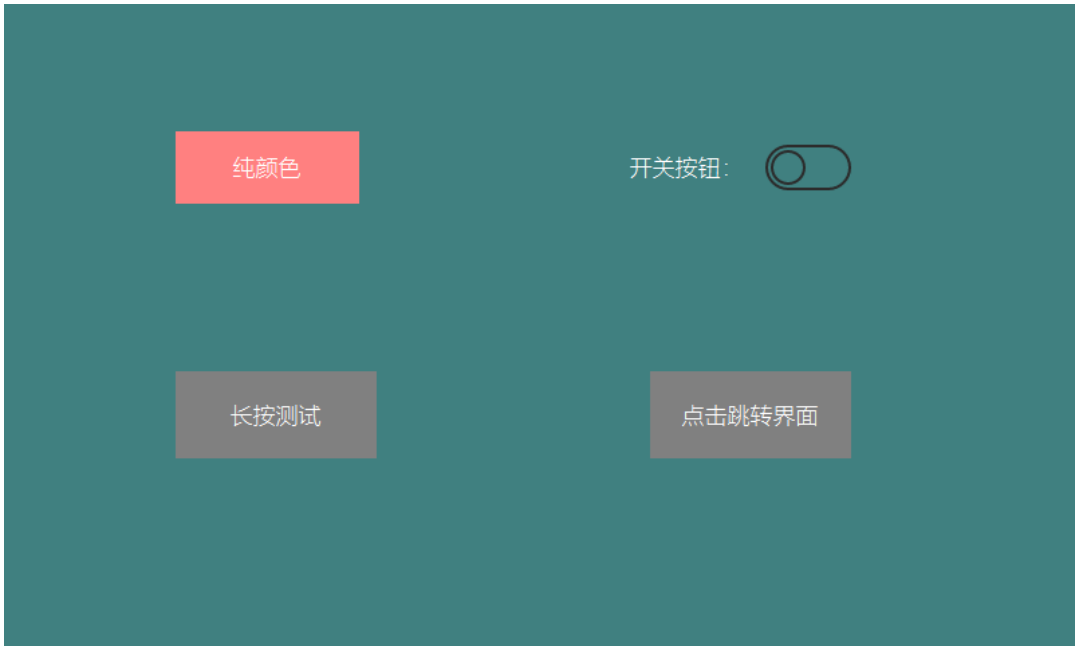
2. 按键类 Button
处理点击类事件。

示例：
按键事件：

```
static bool onButtonClick_Button1(ZKButton *pButton) {
        //LOGD(" ButtonClick Button1 !!!\n");
        return false;
}
```

在事件中处理相应业务逻辑。
设置按键文本和图片与文本控件相同。

效果图：



3. 滑块/进度条
显示和设置进度值。

滑块需要添加四张资源图片才能正常工作：
1） 背景图

2） 有效图

3） 滑块

4） 滑块按下

示例：
监听进度条变化：

```
static void onProgressChanged_XXXX(ZKSeekBar *pSeekBar, int progress) {
    //LOGD("XXXX 滑块的进度值变化为 %d !\n", progress);
}
```

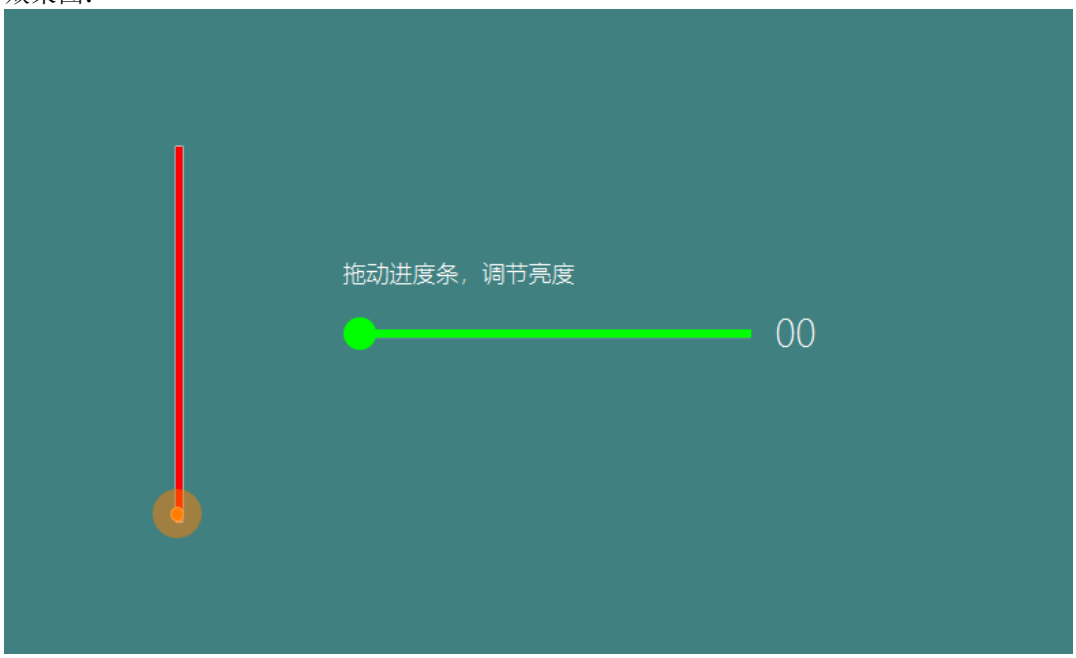
设置滑块进度：

```
//将滑块进度设置为 28
mSeekbarPtr->setProgress(28)
```

获取滑块进度值：

```
int progress = mSeekbarPtr->getProgress();
LOGD("当前滑块的进度值为 %s", progress);
```

效果图：



4. 指针仪表控件

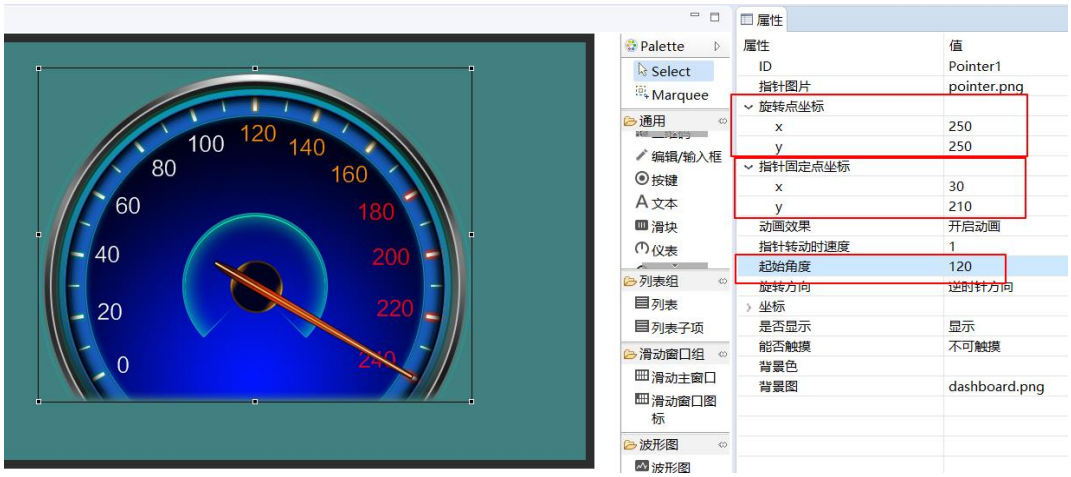
实现一个仪表，或者时钟转动类似的效果，用这个控件就可以轻松实现。用来处理图形旋转。

仪表控件需要添加背景图片和指针图片。
这类控件最常用的方法就是通过代码调整指针的旋转角度。 具体的函数是：

```
void setTargetAngle(float angle);
```

参数为旋转的目标角度。

效果图：



5. 列表控件

列表按键经常用于一个页面无法展示完成所有信息的时候使用，同时每个单元信息中存在一些一致的属性分类，如在显示 WIFI 信息，表格信息时会使用到此控件。

示例：

列表调用数据结构体：

```
*mDashbroadView1Ptr->setTargetAngle(120) 在控件mDashbroadVi
*
* 在Eclipse编辑器中 使用“alt + /”  快捷键可以打开智能提示
*/

typedef struct {
    const char* mainText;
    const char* subText;
    bool bOn;
} S_TEST_DATA;

static S_TEST_DATA sDataTestTab[] = {
    { "测试数据1", "testsub1", false },
    { "测试内容2", "testsub2", false },
    { "测试数据3", "testsub3", false },
    { "测试测试4", "testsub4", true },
    { "测试数据5", "testsub5", false },
    { "测试数据6", "testsub6", true },
    { "测试数据7", "testsub7", false },
    { "测试数据8", "testsub8", false },
    { "测试数据9", "testsub9", false },
    { "测试数据10", "testsub10", false },
    { "测试数据11", "testsub11", false }
};

/**
 * 注册定时器
 * 在此数组中添加即可
 */
```

列表相关的函数：

getListItemCount_List1： 获取列表的项数(即长度)

obtainListItemData_List1： 设置列表每一项的显示内容

onListItemClick_List1： 设置列表控件的点击事件

```
static int getListItemCount_List1(const ZKListView *pListView) {
    //LOGD(" getListItemCount_ List1  !!!\n");
    return sizeof(sDataTestTab) / sizeof(S_TEST_DATA);
}

static void obtainListItemData_List1(ZKListView *pListView,ZKListView::ZKListItem *pListItem, int index) {
    ZKListView::ZKListSubItem* psubText = pListItem->findSubItemByID(ID_MAIN_ListSub1);
    ZKListView::ZKListSubItem* psubButton = pListItem->findSubItemByID(ID_MAIN_ListSub2);

    psubText->setText(sDataTestTab[index].subText);
    pListItem->setText(sDataTestTab[index].mainText);
    psubButton->setSelected(sDataTestTab[index].bOn);
    //LOGD(" obtainListItemData_ List1  !!!\n");
}

static void onListItemClick_List1(ZKListView *pListView, int index, int id) {
    //LOGD(" onListItemClick_ List1  !!!\n");
    sDataTestTab[index].bOn = !sDataTestTab[index].bOn;
}
```

循环列表相关的函数：

```
static int getListItemCount_CycleList(const ZKListView *pListView) {
    //LOGD(" getListItemCount_ CycleList  !!!\n");
    return 50;
}

static void obtainListItemData_CycleList(ZKListView *pListView,ZKListView::ZKListItem *pListItem, int index) {
    //LOGD(" obtainListItemData_ CycleList  !!!\n");
    pListItem->setText(index + 1);
}

static void onListItemClick_CycleList(ZKListView *pListView, int index, int id) {
    //LOGD(" onListItemClick_ CycleList  !!!\n");
}
```

效果图：

6. 波形图

绘制曲线型或者折线型波形图。一个控件可添加多个波形。

示例：

设置波形线宽度，对应属性表上的 **线条宽度** 属性：

void setPenWidth(int index, int width)

设置波形颜色，对应属性表上的 **波形颜色** 属性：

void setPenColor(int index, ARGB color)

设置 x 轴缩放，对应属性表上的 **x 轴缩放** 属性：

void setXScale(int index, double xScale)

设置 y 轴缩放，对应属性表上的 **y 轴缩放** 属性：

void setYScale(int index, double yScale)

将 pPoints 数组中的 count 个点绘制到 第 index 个波形上：

```
typedef struct _MPPOINT
{
    float x;
    float y;
}MPPOINT;
```

*void setData(int index, const MPPOINT *pPoints, int count)*

将单个数据增加到波形上， `data` 为 y 值：

void addData(int index, float data)

清除波形图数据：

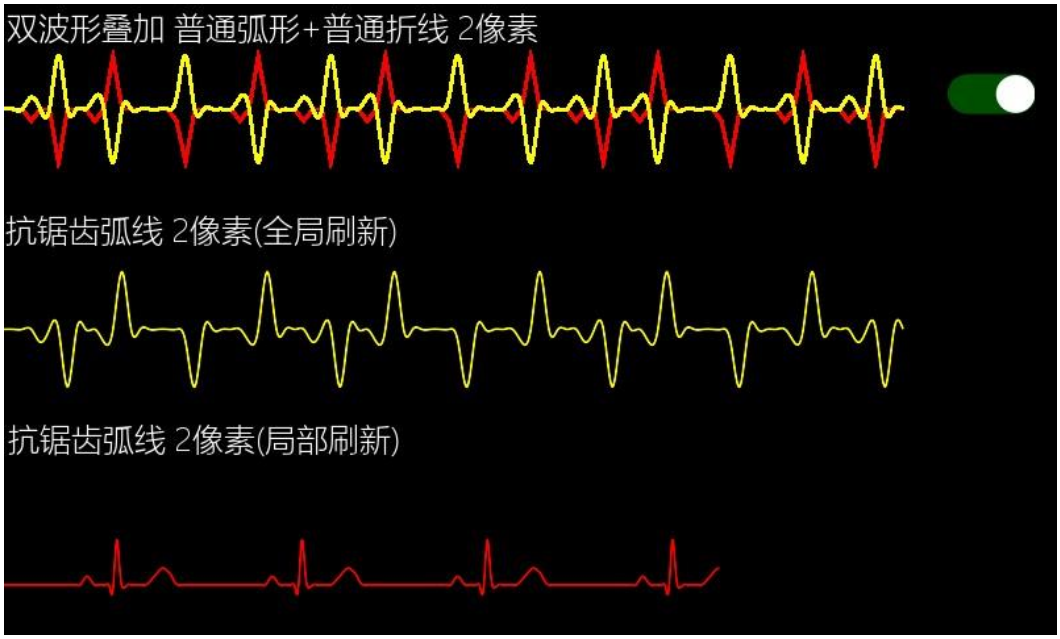
setData(index, NULL, 0)

如果采用 **void setData(int index, const MPPOINT *pPoints, int count)**方式绘制波形，通常需要自行将数组中的值按下标偏移：

```
static void movePoints(MPPOINT* p,int size){
   for(int i =0;i < size-1;i++){
        p[i].y = p[i+1].y;
   }
   p[size-1].y = 0;
}
```

一般情况下，添加定时器配合波形方便定时刷新

效果图：



7. 圆形进度条

某些情况下，我们需要显示一个 Loading 的加载动画。 那么圆形进度条这个控件就非常适合。

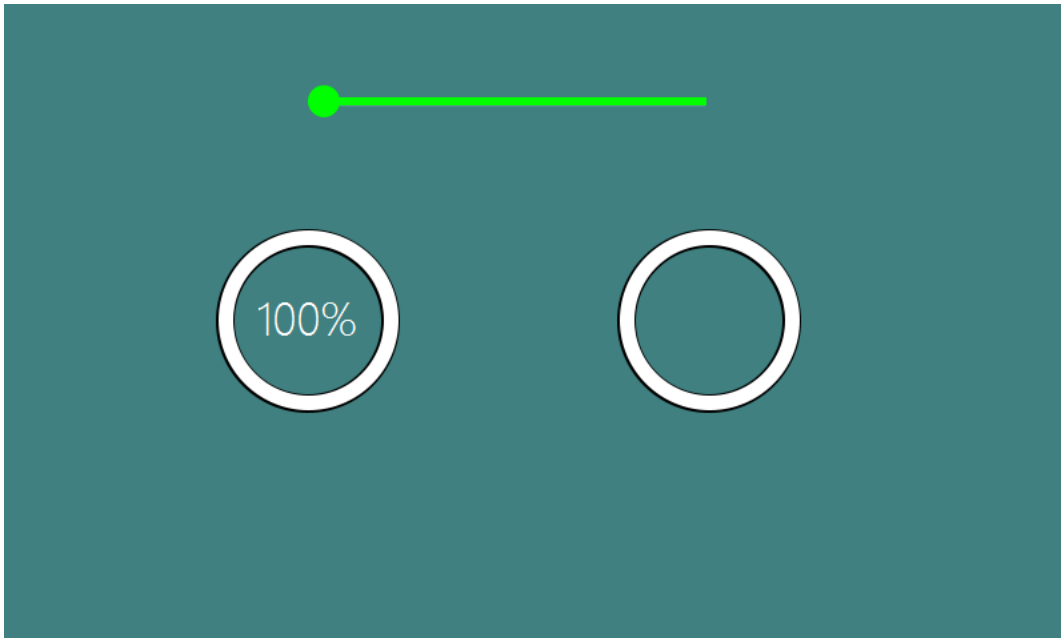
圆形进度条实质是显示当前进度对应的扇形区域，这个区域是对 **有效图** 的裁剪得来。举例：如果照上图属性设置， 最大值 100， 起始角度 0， 旋转方向为 顺时针方向，那么当我们设置 25 的进度时，仅显示右上角 90°的扇形区域。 如果进度值为 100，那么就显示全部的有效图。

注意：这个扇形的显示区域只针对**有效图**进行裁剪， **背景图**不会被裁剪。

示例：

```
//设置当前进度
void setProgress(int progress);
//得到当前进度值
int getProgress();

//设置进度最大值
void setMax(int max);
//得到进度最大值
int getMax();
```

效果图：



8. 二维码
根据设定内容生成二维码并显示。

示例：
设置二维码内容：
*bool loadQRCode(const char *pStr);*

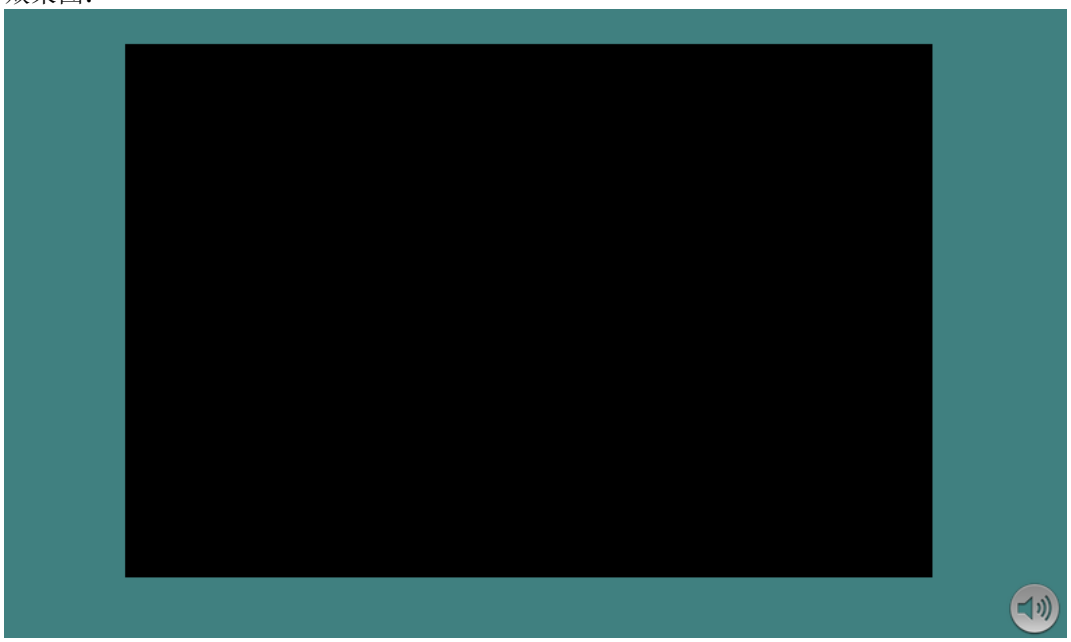
效果图：

9. 视频
该控件仅在带有多媒体多媒体功能的设备版本上可用。可循环播放指定目录的视频文件和由用户操作播放。循环播放需要用户添加视频配置文件，当控件设置为视频轮播类型时，会自动读取该配置文件。这个

配置文件需要位于 TF 卡根目录下，文件名是 **XXXX_video_list.txt**，XXXX 表示对应的 UI 文件前缀名。例如： 我在 **main.ftu** 中添加了一个视频控件，那么，对应的配置文件名为 **main_video_list.txt**
配置文件以行为单位， 每行为视频文件的绝对路径，如果视频文件也位于 TF 卡根目录，那么直接填写 **/mnt/extsd/** 加上 视频文件名即可。

注意：防止编码问题导致视频文件读取失败，请尽量使用英文命名视频文件。

效果图：



用户可根据需求添加 UI 和功能，如视频播放/暂停、拖到视频进度条等。

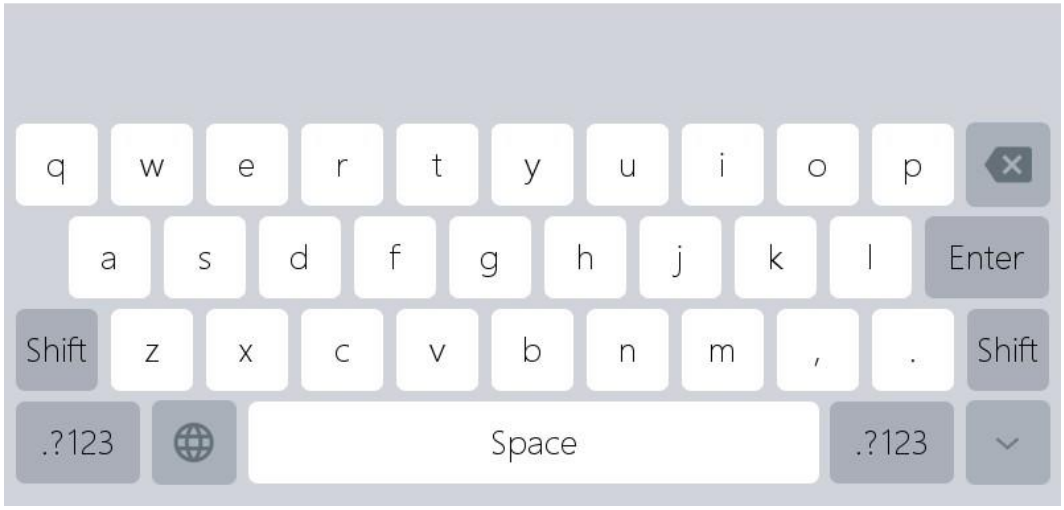
10. 文本数字输入框/中文输入法
    如果需要数字及中文输入，利用现有的 编辑/输入框 控件就可以快速实现。
    输入框相关属性：
    a. 是否为密码框输入
       如果选择是，当模拟键盘输入时，正在键入的字符会显示为指定的密码字符，否则无变化。
    b. 密码字符
       如果 是否为密码输入 选择是，正在输入的字符会显示为指定的密码字符，否则无变化。
    c. 文本类型
       全文本：表示可以输入中英文及数字，不受限制；
       仅数字：表示只能输入数字，其它受限。
    d. 提示文本
       当模拟键盘中内容为空时，会自动显示提示文本。

e. 提示文本颜色

当模拟键盘中内容为空时，会自动显示提示文本，该文本颜色为指定颜色。

效果图：

内置拼音输入法：



内置数字键盘输入：

11. 窗口容器

窗口实际是一个容器部件。可以包含所有的控件，也可以再次包含一个新的窗口。可以用于以下场景：

1) 显示隐藏一个控件组合
2) 当需要完成 tab 页面的时候可以通过多个窗口实现不同的窗口切换

3) 弹出对话框
4) 弹出悬浮框

示例：

```
//显示窗口
void showWnd();
//隐藏窗口
void hideWnd();
判断窗口是否显示
bool isWndShow();
```

效果图：

12. 滑动窗口

滑动窗口控件与手机主界面九宫格左右滑动的界面效果类似。由一个滑动主窗口和多个图标组成。添加了滑动窗口控件，那么在编译后，会自动生成关联函数。

示例：

翻页。参数 **bool isAnimatable** 表示是否开启翻页时的动画; 默认为 false，表示关闭动画。

```
// 切换到下一页
void turnToNextPage(bool isAnimatable = false);
// 切换到上一页
void turnToPrevPage(bool isAnimatable = false);
```

监听滑动窗口翻到了第几页

```
namespace { // 加个匿名作用域，防止多个源文件定义相同类名，运行时冲突

// 实现自己的监听接口
class MySlidePageChangeListener : public ZKSlideWindow::ISlidePageChangeListener {
public:
    virtual void onSlidePageChange(ZKSlideWindow *pSlideWindow, int page) {
```

```
        LOGD("page: %d\n", page);
    }
};

}

// 定义监听对象
static MySlidePageChangeListener sMySlidePageChangeListener;

static void onUI_init() {
    mSlidewindow1Ptr->setSlidePageChangeListener(&sMySlidePageChangeListener);
}
```

获取当前页

```
getCurrentPage()
```

效果图：

13. 画布

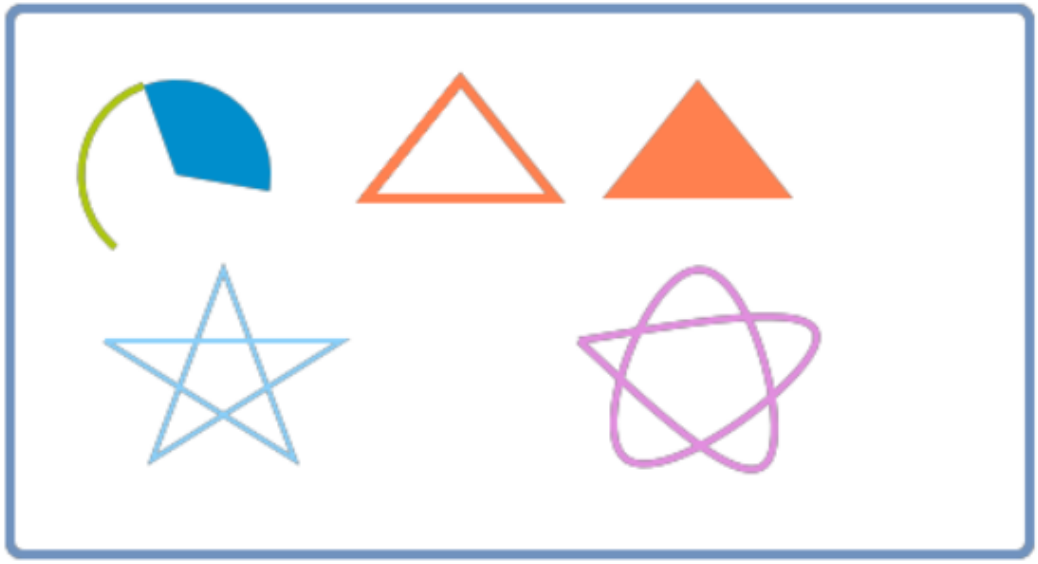
画布控件提供了简单几何图形绘制接口。 该控件几乎所有功能都需要代码实现。

示例：

```
static void onUI_init() {

    /**
     * 绘制一个圆角矩形边框
     */
    mPainter1Ptr->setLineWidth(4);
    mPainter1Ptr->setSourceColor(0x7092be);
    mPainter1Ptr->drawRect(10, 10, 430, 230, 5, 5);

    /**
```

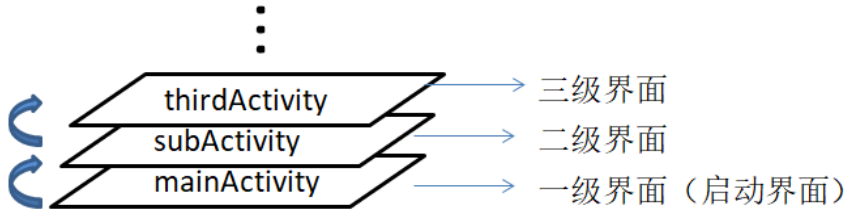
```
     * 绘制一段圆弧
     */
    mPainter1Ptr->setLineWidth(3);
    mPainter1Ptr->setSourceColor(0xadc70c);
    mPainter1Ptr->drawArc(80, 80, 40, 40, -20, -120);

    /**
     * 绘制一段扇形
     */
    mPainter1Ptr->setLineWidth(3);
    mPainter1Ptr->setSourceColor(0x008ecc);
    mPainter1Ptr->fillArc(80, 80, 40, 40, -20, 120);


    /**
     * 绘制三角形
     */
    mPainter1Ptr->setLineWidth(4);
    mPainter1Ptr->setSourceColor(0xff804f);
    mPainter1Ptr->drawTriangle(200, 40, 160, 90, 240, 90);//空心三角形
    mPainter1Ptr->fillTriangle(300, 40, 260, 90, 340, 90); //实心三角形

    /**
     * 绘制直线
     */
    MPPOINT points1[] = {
            {50 , 150},
            {150, 150},
            {70 , 200},
            {100, 120},
            {130, 200},
            {50 , 150}
    };
    /** 根据提供的多个点坐标依次连接成线 */
    mPainter1Ptr->setLineWidth(2);
    mPainter1Ptr->setSourceColor(0x88cffa);
    mPainter1Ptr->drawLines(points1, TABLESIZE(points1));

    /**
     * 绘制曲线
     */
    MPPOINT points2[] = {
            {250, 150},
            {350, 150},
            {270, 200},
            {300, 120},
            {330, 200},
            {250, 150}
    };
    mPainter1Ptr->setLineWidth(3);
    mPainter1Ptr->setSourceColor(0xe28ddf);
    /** 根据提供的多个点坐标连接为曲线 */
    mPainter1Ptr->drawCurve(points2, TABLESIZE(points2));
}
```

效果图：



3. 界面交互

界面的层级关系如下：

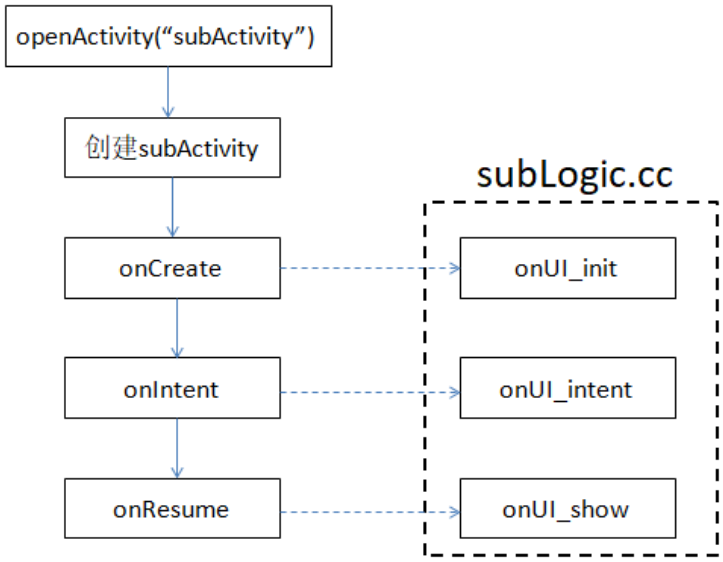


首先我们的应用起来后会先进入 **mainActivity** 对应的界面，即启动界面，之后通过 **openActivity** 方法打开了 **subActivity** 对应的界面，接着再进入 **thirdActivity** 对应的界面，就形成了上图所看到的层级效果了；后打开的界面在层级上层，它们间形成了栈的这样一种关系。

3.1. 打开界面时的活动流程

我们再来看看调用 **openActivity** 方法后，程序走了哪些流程，这里分两种情况介绍：

a. 界面栈中**不存在**即将要打开的界面；



我们先来看一下 **subLogic.cc** 中的 **onUI_init** 函数，只有界面栈中不存在该界面情况下，第一次打开时，会走这个函数，走到这里意味着所有控件指针也就初始化完成了，在这个函数里我们就可以开始对它们进行一些操作，如下：

```
static void onUI_init() {
  //Tips :添加 UI 初始化的显示代码到这里,如:mTextview1Ptr->setText("123");
  LOGD("sub onUI_init\n");
```

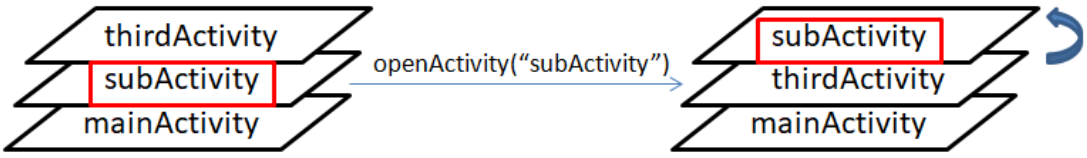
```
  mTextview1Ptr->setText("123");
}
```

界面打开时有数据传递过来，在 onUI_intent 回调函数中接收处理：

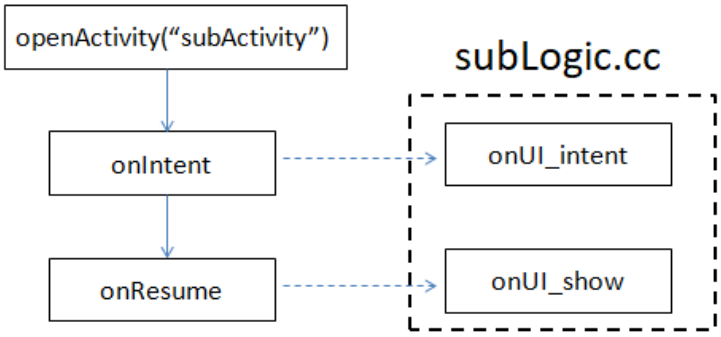
```
static void onUI_intent(const Intent *intentPtr) {
  LOGD("sub onUI_intent\n");
  // 判断不为空
  if (intentPtr) {
        // 键值解析
        std::string cmd = intentPtr->getExtra("cmd");          // "open"
        std::string value = intentPtr->getExtra("value"); // "ok"
        ......
  }
}
```

界面显示完成回调 **onUI_show** 函数；

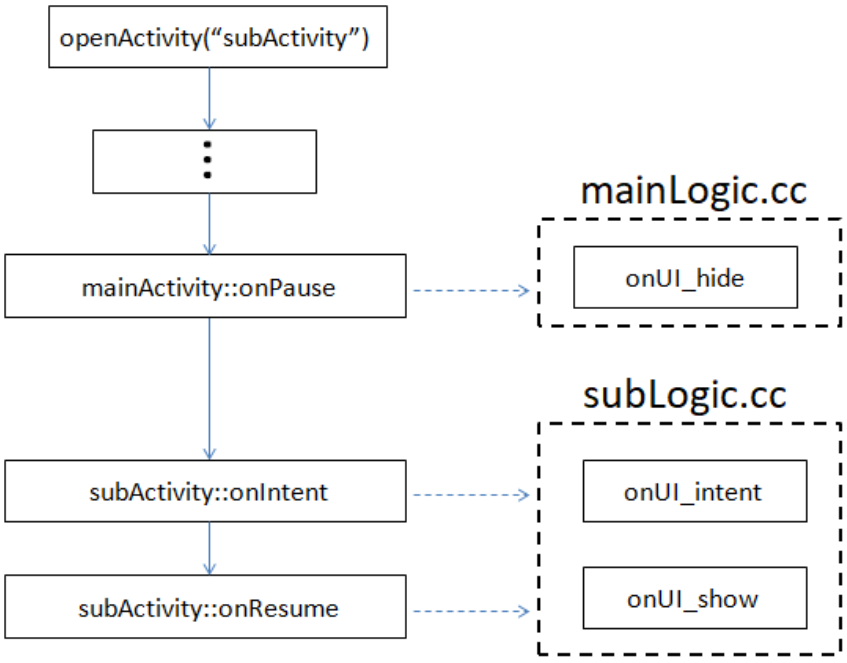
b. 界面栈中存在即将要打开的界面；



这种情况只是将界面栈中对应的界面移动到顶层，不走 **onUI_Init** 流程；



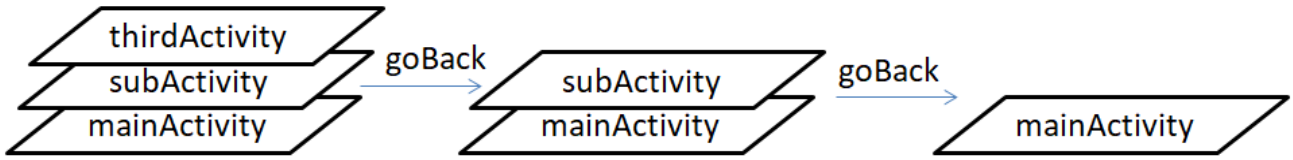
打开显示一个界面，意味着之前顶层的界面被隐藏掉了；假设在 **mainActivity** 界面打开了 **subActivity** 界面，它们的活动流程如下：



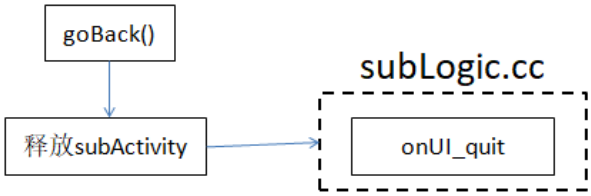
这里我们重点关注 **mainActivity** 界面**隐藏** ——> **subActivity** 界面**显示**流程；

3.2. 关闭界面时的活动流程

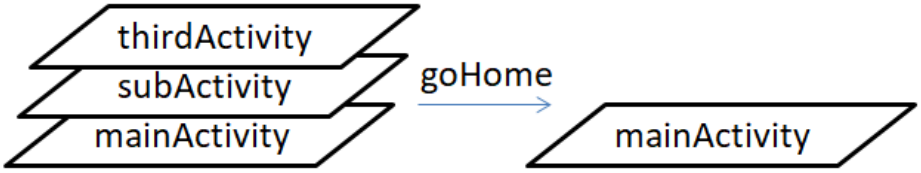
当我们调用 **goBack()**函数时，会将顶层的界面弹出，直到启动界面；



关闭界面时会回调 **onUI_quit** 函数，如果界面打开后有申请一些什么资源的，记得要在这里进行释放；



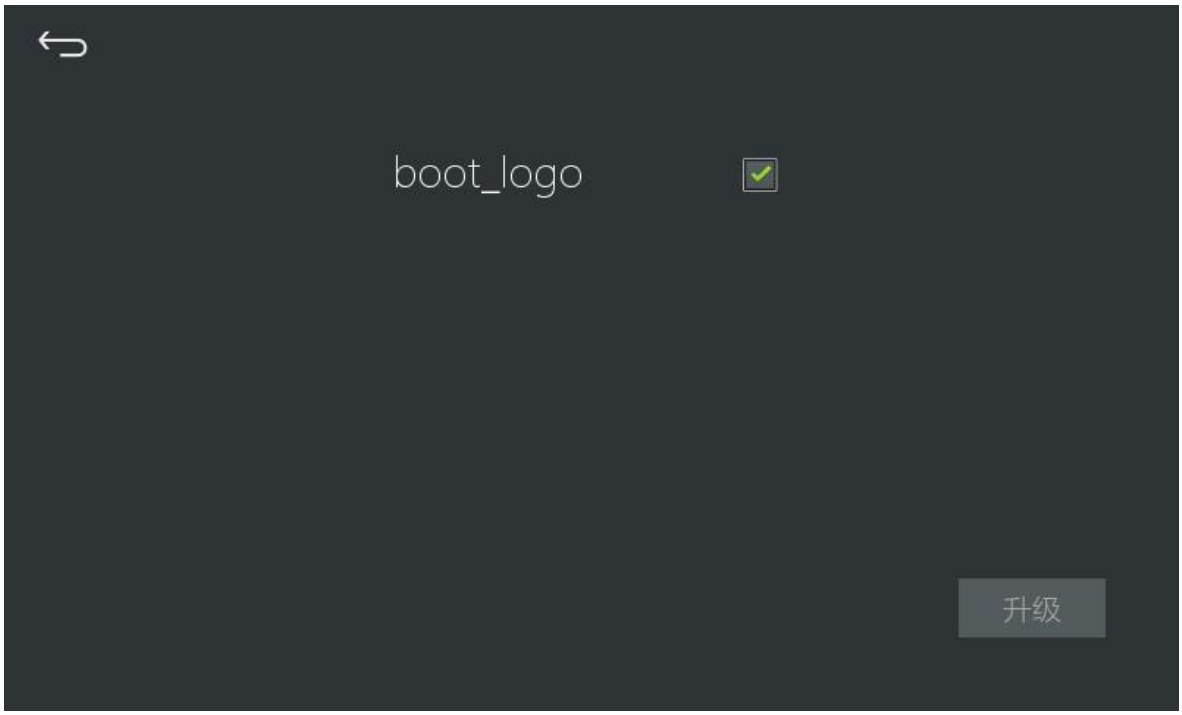
退出顶层的界面后，会将下一层界面显示出来，即会回调下一层界面的 **onUI_show** 接口；
当我们调用 **goHome()**函数时，会直接回退到启动界面，将其他界面都弹出；



当我们调用 **closeActivity("xxx")**函数时，可以移除除启动界面外任意界面；当移除的不是顶层的界面时，下一层的界面不会走流程；

3.3. 系统内置界面

除了开发人员自己定制的界面外，系统也内置了几个常用的界面，如插卡升级时出现的界面就属于内置界面之一；



另外，还有系统设置界面，打开方式：

```
EASYUICONTEXT->openActivity("ZKSettingActivity");
```

我们可以通过一个按钮点击跳转到该界面看一下效果（其他的几个内置界面都可以通过如下方式查看效果）。

```
static bool onButtonClick_Button1(ZKButton *pButton) {
    EASYUICONTEXT->openActivity("ZKSettingActivity");
    return false;
}
```

其中每一项点击进去后又是新的内置界面，打开网络设置：
EASYUICONTEXT->openActivity("NetSettingActivity");

打开 WIFI 设置：
EASYUICONTEXT->openActivity("WifiSettingActivity");

如果目标机器支持 wifi，打开右上角开关，界面上会显示搜索到的 wifi 信息；
打开热点设置界面：
EASYUICONTEXT->openActivity("SoftApSettingActivity");

回到刚刚的系统设置界面，我们再点击打开语言设置界面：

EASYUICONTEXT->openActivity("LanguageSettingActivity");

触摸校准界面：

EASYUICONTEXT->openActivity("TouchCalibrationActivity");

开发者选项界面：

EASYUICONTEXT->openActivity("DeveloperSettingActivity");

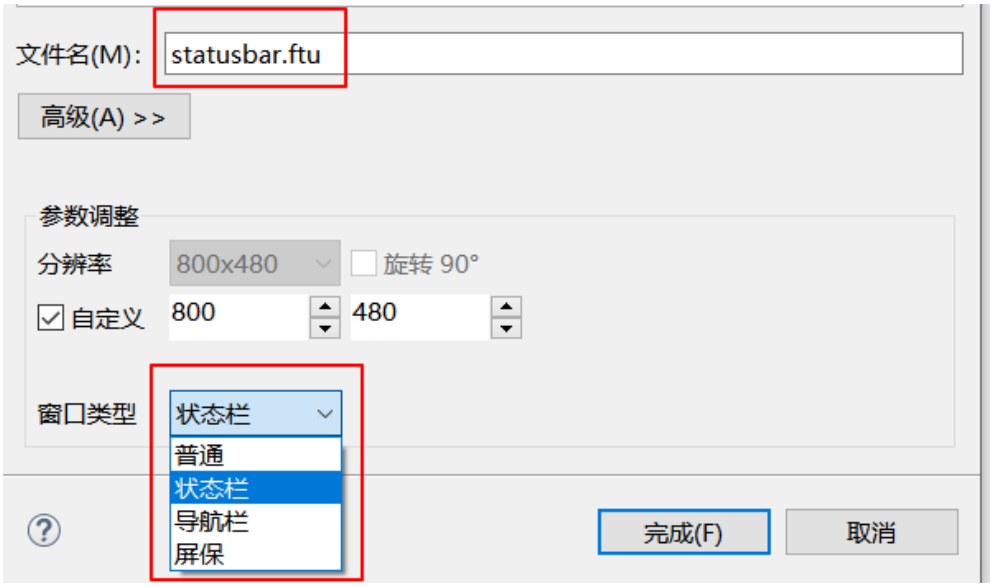
目前只有 ADB 的调试开关选项。

3.4. 系统应用

前面介绍的应用界面我们归类为普通窗口界面，一般情况下已经够用了，通过工具新建 UI 界面时，默认的窗口类型就是普通窗口：

如果某些场景需要一个悬浮在 UI 界面之上的显示区，那么普通窗口就不能胜任这份工作了，需要用到我们其他的几种类型的窗口了； 在 **窗口类型** 选项中，有三种特殊类型窗口选项，这三种特殊类型具有特殊的文件名，分别对应为：

a. 状态栏 **statusbar.ftu**
b. 导航栏 **navibar.ftu**
c. 屏保 **screensaver.ftu**



点击确定后，工具会帮我们自动生成相应的代码；这三种类型的窗口，对于控件的操作与普通窗口一样；

3.4.1 状态栏

这个状态栏跟 Android，iOS 手机的状态栏概念一致，是一个悬浮在 UI 界面之上的一个通用显示区。通常用于显示一些常见信息，或者放置返回键或 Home 键等等。如下效果：

系统提供了两个接口可以用于操作状态栏：

显示状态栏：
EASYUICONTEXT->showStatusBar();
隐藏状态栏：
EASYUICONTEXT->hideStatusBar();

3.4.2 导航栏

这个导航栏跟 Android 手机的导航栏概念一致，是一个悬浮在 UI 界面之上的一个通用操作或者显示区，一般在页面的底部。通常用于显示一些操作按键。导航栏实际上和状态栏没有什么差别。

显示导航栏：
EASYUICONTEXT->showNaviBar();
隐藏导航栏：
EASYUICONTEXT->hideNaviBar();

3.4.3 屏保应用

屏保应用是指当用户不再做系统交互的时候，时间超过某个指定时间长度。系统自动打开一个页面。 右键工程，选择 Properties 选项，在弹出的属性框里，我们可以对屏保超时时间进行设置，单位为秒，-1 表示不进屏保；

我们也可以通过代码进行一些设置，见 **jni/include/entry/EasyUIContext.h**：

添加头文件：

```
#include "entry/EasyUIContext.h"
```

设置屏保超时时间

```
//设置屏保超时时间为 5 秒
EASYUICONTEXT->setScreensaverTimeOut(5);
```

设置是否允许启用屏保

```
EASYUICONTEXT->setScreensaverEnable(false); //关闭屏保检测
EASYUICONTEXT->setScreensaverEnable(true); //恢复屏保检测
```

立即进入屏保

```
EASYUICONTEXT->screensaverOn();
```

立即退出屏保

```
EASYUICONTEXT->screensaverOff();
```

判断是否进入了屏保

```
EASYUICONTEXT->isScreensaverOn();
```

4. 定时器

在某些情况下，我们可能需要定时做一些操作。比如，间隔一定时间发送心跳包、定时查询数据刷新 UI 界面、或者做一些轮询的任务等等。如果你有以上的这些需求，那么定时器是一个方便的选择。

4.1. 定时器的使用

4.1.1 注册定时器

为了方便使用，我们以填充结构体的形式来添加定时器。
在 **Logic.cc** 文件中，默认会有这样一个结构体数组：

```
/**
 * 注册定时器
 * 在此数组中添加即可
 */
static S_ACTIVITY_TIMEER REGISTER_ACTIVITY_TIMER_TAB[] = {
 //{0,  6000}, //定时器 id=0, 时间间隔为 6 秒
 //{1,  1000},
 };
```

如果我们想要添加一个定时器时，只需要在这个数组中添加结构体即可。
这个结构体的定义如下：

```
typedef struct {
 int id; // 定时器 ID ，  不能重复
 int time; // 定时器时间间隔   单位/毫秒
}S_ACTIVITY_TIMEER;
```

4.1.2 添加定时器的逻辑代码

在数组中注册定时器后，当某一个定时器触发时，系统将调用对应 **Logic.cc** 文件中的 `void onUI_Timer(int id)`函数，针对这个定时器的所有操作代码，就是添加在这个函数中，函数具体定义如下：

```
static bool onUI_Timer(int id){
    //Tips:添加定时器响应的代码到这里,但是需要在本文件的       REGISTER_ACTIVITY_TIMER_TAB 数组中 注册
    //id 是定时器设置时候的标签,这里不要写耗时的操作，否则影响 UI 刷新,return:[true] 继续运行定时器;[false] 停止运行当前定时器
    return true;
 }
```

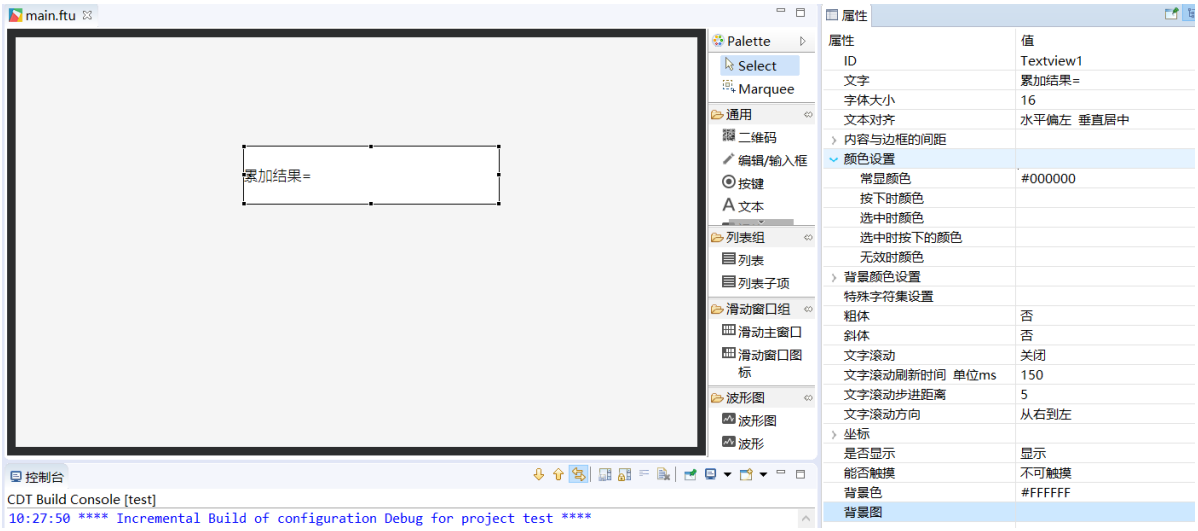
该函数同样是随 **Logic.cc** 文件默认生成。
注意函数的参数 **id** ，它与结构体数组中定义的 **id** 值相同，我们可以根据 **id** 值判断当前触发的是哪一个定时器，从而做一些针对性的操作。

注意：
每个界面的定时器都是独立的，不同界面定时器的 id 可以定义一样
注册的定时器，只要界面不销毁，都会一直在跑；
注册了无需手动停止，界面销毁了就会自动停止了。

4.2. 示例

需求： 有一个整形变量，每隔一秒钟，将变量累加 1，并且将最新结果显示到屏幕上。
具体实现过程如下：

1. 首先我们在 UI 文件中添加一个文本控件，用于显示累加的结果



2. 注册定时器， 在 **mainLogic.cc** 的定时器数组中，添加一个结构体， 定时器 id 为 1，时间间隔为 1 秒。注意时间单位为毫秒。

```
32
33 /**
34  * 注册定时器
35  * 在此数组中添加即可
36  */
37 static S_ACTIVITY_TIMEER REGISTER_ACTIVITY_TIMER_TAB[] = {
38
39     {1,  1000}, //定时器id=1 , 时间间隔1秒
40 };
41
```

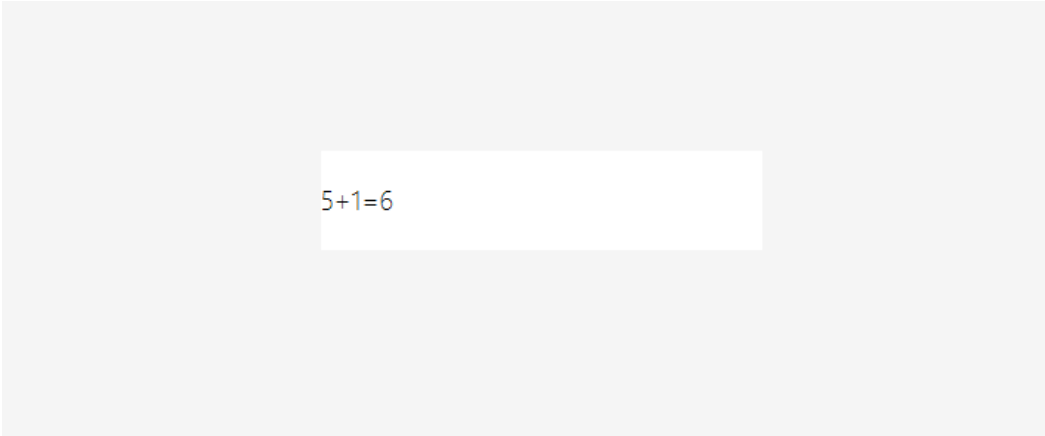
3. 在 **mainLogic.cc** 中，定义静态整型变量，初始化为 0

```
static int g_Count = 0;
```

4. 在 void onUI_Timer(int id)函数中，添加累加代码，并将其显示到文本控件中。

```
static bool onUI_Timer(int id){
    switch (id) {
    case 1:
        char buf[64] = {0};
        snprintf(buf, sizeof(buf), "%d + 1 = %d", g_Count, g_Count + 1);
        mTextview1Ptr->setText(buf);
        ++g_Count;
        break;
    }
    return true;
}
```

5. 编译运行



4.3. 任意开启停止定时器

我们可以在 REGISTER_ACTIVITY_TIMER_TAB 中添加预设的定时器，但这种方式不够灵活，无法任意开启/停止，下面介绍另一种添加定时器的方法。 Activity 类中有提供三个关于定时器的方法， 下面介绍如何使用。

```
/**
 * 注册定时器
 */
void registerUserTimer(int id, int time);
/**
 * 取消定时器
 */
void unregisterUserTimer(int id);
/**
 * 重置定时器
 */
void resetUserTimer(int id, int time);
```

1. 在 logic.cc 中，添加一个变量，标识该定时器是否已经注册。

```
/**
    * 是否注册了定时器
    */
```

```
static bool isRegistered = false;
#define TIMER_HANDLE    2
```

2. 我们再添加两个按键，在按键的点击事件中，我们分别添加 注册定时器、取消定时器的代码。

```
static bool onButtonClick_ButtonTimerOn(ZKButton *pButton) {
    //如果没有注册才进行注册定时器
    if (!isRegistered) {
        mActivityPtr->registerUserTimer(TIMER_HANDLE, 500);
        isRegistered = true;
    }
    return false;
}

static bool onButtonClick_ButtonTimerOff(ZKButton *pButton) {
    //如果已经注册了定时器，则取消注册
    if (isRegistered) {
        mActivityPtr->unregisterUserTimer(TIMER_HANDLE);
        isRegistered = false;
    }
    return false;
}
```

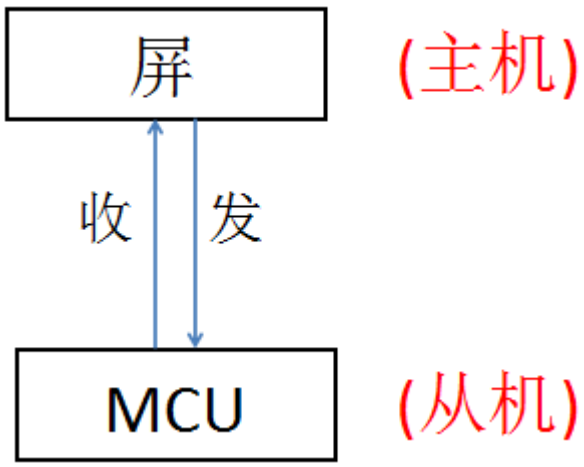
注意：

registerUserTimer、**unregisterUserTimer**、**resetUserTimer** 三个方法不能在 **onUI_Timer** 中调用，会造成死锁。

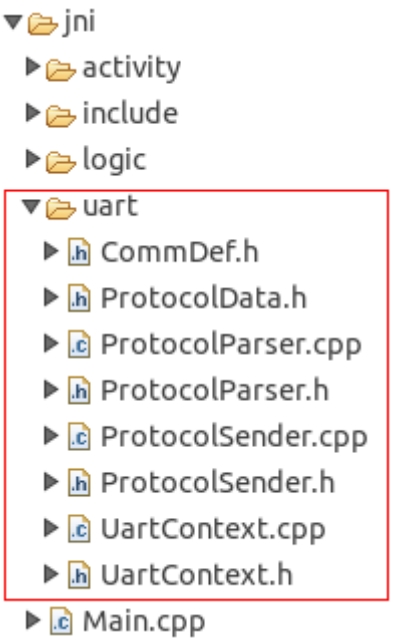
5. 串口通讯

5.1. 简介

下面这张图，是最最最简单的一个通讯模型；屏和 MCU 之间通过串口进行通信，它们之间只要定好协议，就可以进行交互了



这里有点需要注意的，传统的串口屏它们是作为从机端设备，通过 MCU 发相应的指令来控制它们；而我们的串口屏不一样，我们的屏是具有逻辑的，它可以自己实现交互，在这里作为主机端。
如果由自己从头来开发这部分通讯代码，那工作量将是巨大的；为了简化开发流程，使开发人员更多的关注业务逻辑的开发，我们的工具在新建工程时会自动生成串口通讯的代码：

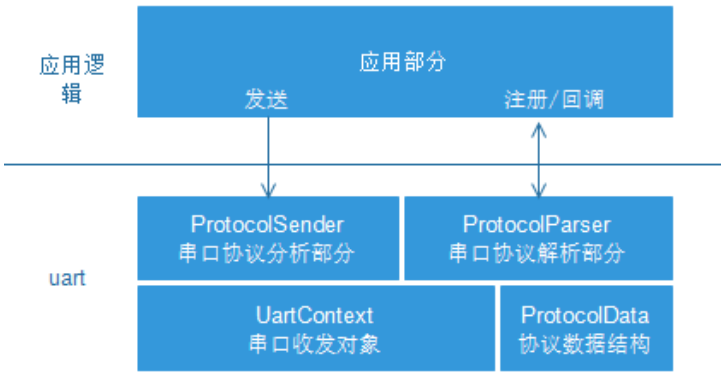


同时，我们也提供了协议数据与界面交互的回调接口：

开发人员更多的是关注数据在 UI 界面上的展示，而通讯部分则由我们的框架自动完成。 通讯框架中的协议解析部分需根据开发人员使用的通讯协议做相应的改动，接下来的通讯框架讲解这一章节中将会重点介绍原理及需要修改的部分。

5.2. 通讯框架讲解

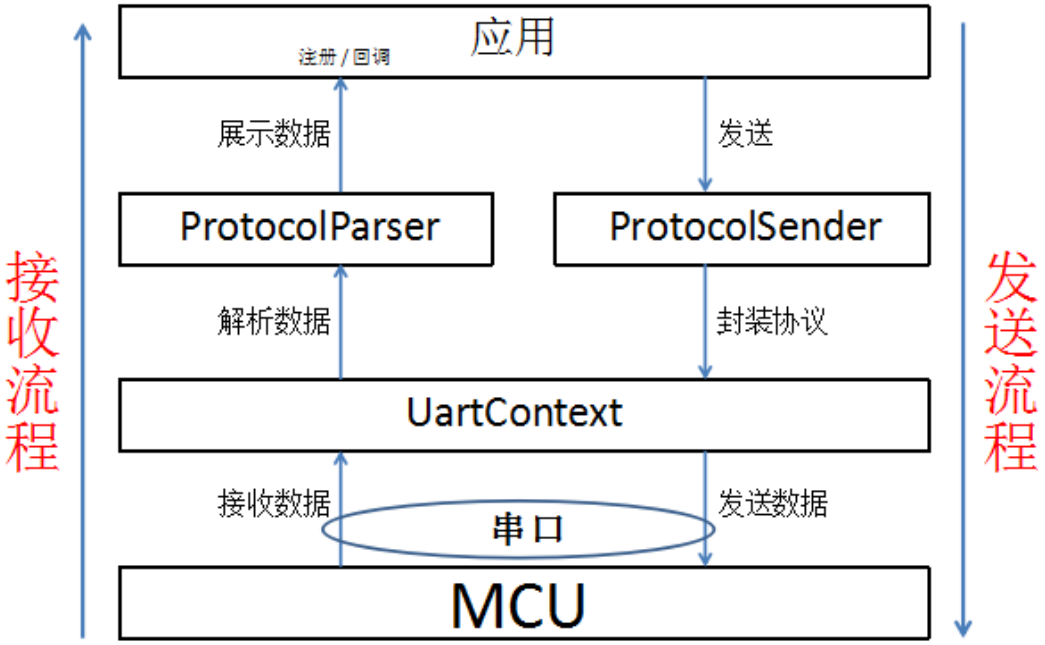
代码框架：



软件 APP 部分分为两层

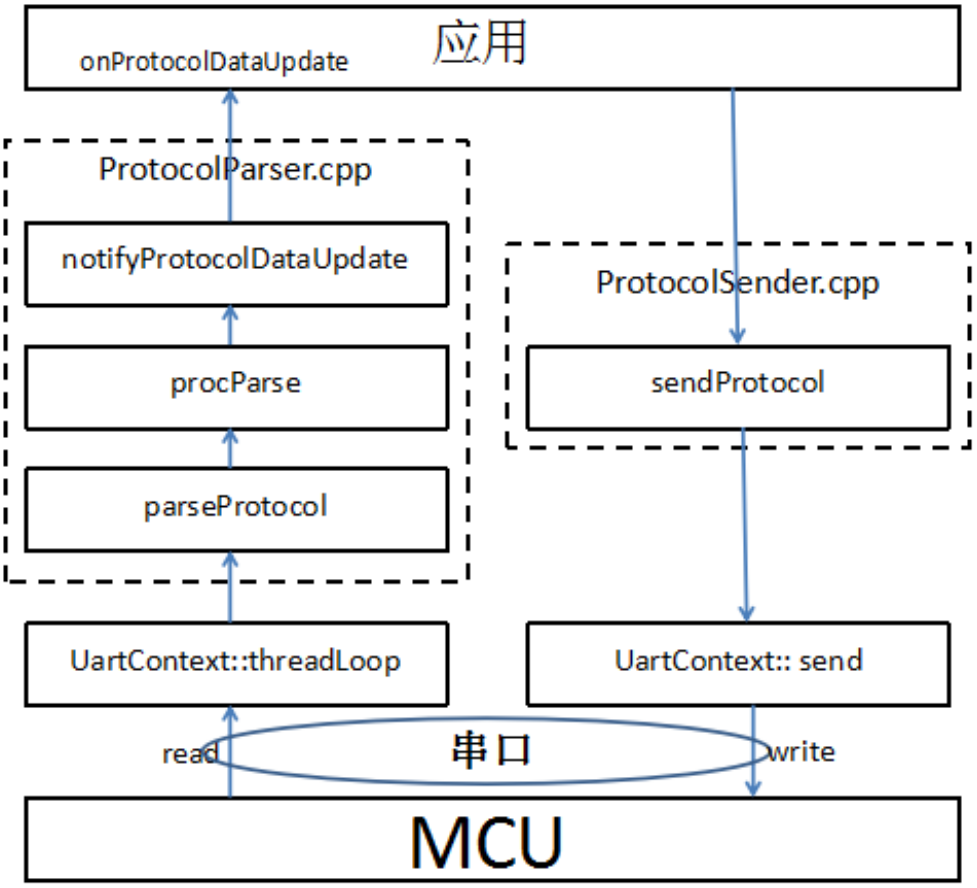
- uart 协议解析和封装的串口 HAL 层
  - UartContext：串口的实体控制层，提供串口的开关，发送，接收接口
  - ProtocolData：定义通讯的数据结构体，用于保存通讯协议转化出来的实际变量；
  - ProtocolSender：完成数据发送的封装；
  - ProtocolParser：完成数据的协议解析部分，然后将解析好的数据放到 ProtocolData 的数据结构中；同时管理了应用监听串口数据变化的回调接口；
- APP 应用接口层
  - 通过 ProtocolParser 提供的接口注册串口数据接收监听获取串口更新出来的 ProtocolData。
  - 通过 ProtocolSender 提供的接口往 MCU 发送指令信息

我们再细化一下这流程：



可以清楚的看到 **接收** 和 **发送** 两路流程一上一下，每一层的功能还是比较清晰的；

具体到代码对应的流程：



无论是接收还是发送流程，最终都是要经过 UartContext 对串口进行读写操作，这是一些标准化的流程，所以 UartContext 我们基本上是不用去修改的，也可以不用理会它是怎么实现的，当然，有兴趣的可以去看一下。

到此，我们对这个通讯模型有个大概的了解，之后我们再来看具体代码的实现。

1. 协议接收部分使用和修改方法

- 通讯协议格式修改

这里我们举一个比较常见的通讯协议例子:

协议头（2 字节）	命令（2 字节）	数据长度（1 字节）	数据（N）	校验（1 字节 可选)
0xFF55	Cmd	len	data	checksum

CommDef.h 文件中定义了同步帧头信息及最小数据包大小信息：

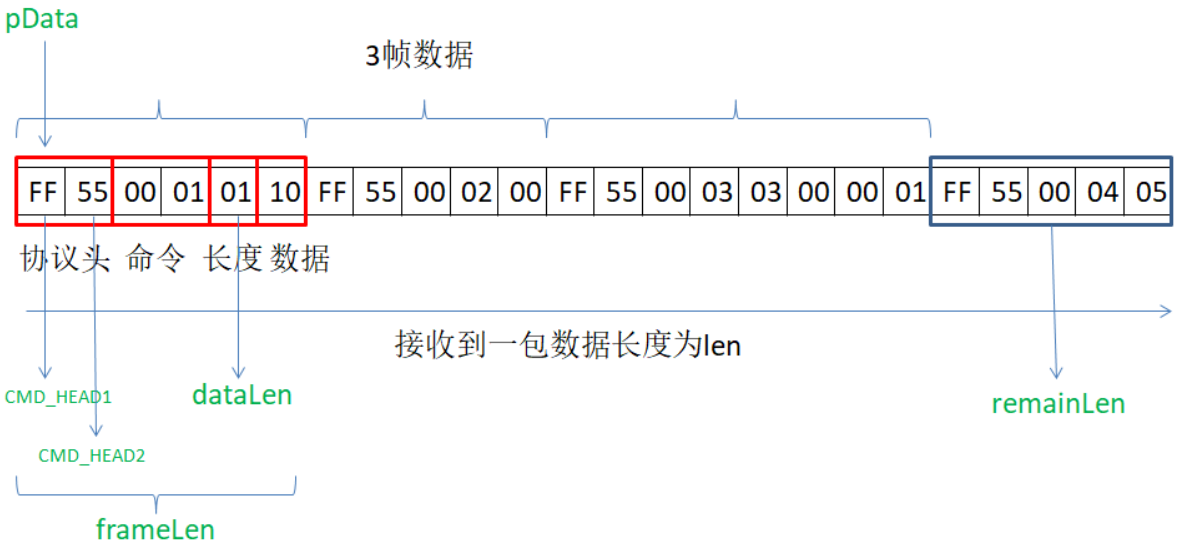
```
// 需要打印协议数据时，打开以下宏
//#define DEBUG_PRO_DATA

// 支持 checksum 校验，打开以下宏
//#define PRO_SUPPORT_CHECK_SUM

/* SynchFrame CmdID   DataLen Data CheckSum (可选) */
/*       2Byte   2Byte     1Byte      N Byte   1Byte */
// 有 CheckSum 情况下最小长度: 2 + 2 + 1 + 1 = 6
// 无 CheckSum 情况下最小长度: 2 + 2 + 1 = 5

#ifdef PRO_SUPPORT_CHECK_SUM
#define DATA_PACKAGE_MIN_LEN          6
#else
#define DATA_PACKAGE_MIN_LEN          5
#endif

// 同步帧头
#define CMD_HEAD1   0xFF
 #define CMD_HEAD2   0x55
```

ProtocolParser.cpp 文件，配置文件命令格式:

```
/**
 * 功能：解析协议
 * 参数：pData 协议数据，len 数据长度
 * 返回值：实际解析协议的长度
 */
int parseProtocol(const BYTE *pData, UINT len) {
  UINT remainLen = len;     // 剩余数据长度
  UINT dataLen;  // 数据包长度
  UINT frameLen;// 帧长度

  /**
   * 以下部分需要根据协议格式进行相应的修改，解析出每一帧的数据
   */
  while (remainLen >= DATA_PACKAGE_MIN_LEN) {
```

```
        // 找到一帧数据的数据头
        while ((remainLen >= 2) && ((pData[0] != CMD_HEAD1) || (pData[1] != CMD_HEAD2))) {
            pData++;
            remainLen--;
            continue;
        }

        if (remainLen < DATA_PACKAGE_MIN_LEN) {
            break;
        }

        dataLen = pData[4];
        frameLen = dataLen + DATA_PACKAGE_MIN_LEN;
        if (frameLen > remainLen) {
            // 数据内容不全
            break;
        }

        // 打印一帧数据，需要时在 CommDef.h 文件中打开 DEBUG_PRO_DATA 宏
#ifdef DEBUG_PRO_DATA
        for (int i = 0; i < frameLen; ++i) {
            LOGD("%x ", pData[i]);
        }
        LOGD("\n");
#endif

        // 支持 checksum 校验，需要时在 CommDef.h 文件中打开 PRO_SUPPORT_CHECK_SUM 宏
#ifdef PRO_SUPPORT_CHECK_SUM
        // 检测校验码
        if (getCheckSum(pData, frameLen - 1) == pData[frameLen - 1]) {
            // 解析一帧数据
            procParse(pData, frameLen);
        } else {
            LOGE("CheckSum error!!!!!!\n");
        }
#else
        // 解析一帧数据
        procParse(pData, frameLen);
#endif

        pData += frameLen;
        remainLen -= frameLen;
    }

    return len - remainLen;
}
```

上面的解析流程有点复杂，下面我们先给出一张图，再来分析可能会容易理解一些；一包数据可能包含 0 到多帧数据，下面这张图里，我们标出来有 3 帧数据，另外还有一帧数据不全，还少 5 个数据，不完整的那一帧数据将会拼接到下一包数据里



- 协议头需要修改：

```
// 1.修改协议头部分的定义，如果协议头长度有变化，则要注意修改协议头判断部分语句。
#define CMD_HEAD1   0xFF
#define CMD_HEAD2   0x55

// 2.协议头长度变化的时候需要修改这里。
while ((mDataBufLen >= 2) && ((pData[0] != CMD_HEAD1) || (pData[1] != CMD_HEAD2)))
```

- 协议长度的位置或者长度计算方式发生变化的修改：

```
// 这里的 pData[4] 代表的是第 5 个数据是长度的字节，如果变化了在这里修改一下。
dataLen = pData[4];
// 帧长度一般是数据长度加上头尾长度。如果协议中传的长度计算方式发生变化修改这个部分。
frameLen = dataLen + DATA_PACKAGE_MIN_LEN;
```

- 校验发生变化的情况

```
/**
 * 默认我们是关闭 checksum 校验的，如果需要支持 checksum 校验，在 CommDef.h 文件中打
PRO_SUPPORT_CHECK_SUM 宏
 * 当校验不一样的时候需要修改校验方法，
 * 1.校验内容变化修改这个位置
 *       if (getCheckSum(pData, frameLen - 1) == pData[frameLen - 1])
 * 2.校验计算公式变化修改 getCheckSum 函数里边的内容
 */

/**
 * 获取校验码
 */
BYTE getCheckSum(const BYTE *pData, int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
        sum += pData[i];
  }

  return (BYTE) (~sum + 1);
}
```

- 当完成一帧数据的接收后程序会调用 procParse 解析

```
// 支持 checksum 校验，需要时在 CommDef.h 文件中打开 PRO_SUPPORT_CHECK_SUM 宏
#ifdef PRO_SUPPORT_CHECK_SUM
 // 检测校验码
 if (getCheckSum(pData, frameLen - 1) == pData[frameLen - 1]) {
        // 解析一帧数据
        procParse(pData, frameLen);
 } else {
        LOGE("CheckSum error!!!!!!\n");
 }
#else
 // 解析一帧数据
 procParse(pData, frameLen);
#endif
```

2. 通讯协议数据对接 UI 控件

继续前面的协议框架我们进入到 procParse 的解析部分。 这里重点的代码是：ProtocolParser.cpp 打开文件然后找到 void procParse(const BYTE *pData, UINT len)

```
/*
 * 协议解析
 * 输入参数:
 *       pData: 一帧数据的起始地址
 *       len: 帧数据的长度
 */
void procParse(const BYTE *pData, UINT len) {
  /*
   * 解析 Cmd 值获取数据赋值到 sProtocolData 结构体中
      */
  switch (MAKEWORD(pData[2], pData[3])) {
  case CMDID_POWER:
        sProtocolData.power = pData[5];
        LOGD("power status:%d",sProtocolData.power);
        break;
  }
  notifyProtocolDataUpdate(sProtocolData);
}
```

以上 **MAKEWORD(pData[2], pData[3])** 在我们的协议例子中表示 Cmd 值； 当数据解析完成后通过 **notifyProtocolDataUpdate** 通知到页面 UI 更新，这个部分请参照后面的 UI 更新部分。

- 数据结构

上面的协议解析到了 sProtocolData 结构体中，sProtocolData 是一个静态的变量，用于保存 MCU（或者其他设备）串口发送过来的数据值。 这个数据结构在 ProtocolData.h 文件中。这里可以添加整个项目里面需要使用到的通讯变量。

```
typedef struct {
  // 可以在这里面添加协议的数据变量
  BYTE power;
} SProtocolData;
```

- UI 更新

UI 界面在工具生成 Activity.cpp 的时候就已经完成了 registerProtocolDataUpdateListener ，也就是说当数据更新的时候 logic 里面页面程序就会收到数据。

```
static void onProtocolDataUpdate(const SProtocolData &data) {
    // 串口数据回调接口
  if (mProtocolData.power != data.power) {
       mProtocolData.power = data.power;
  }

  if (mProtocolData.eRunMode != data.eRunMode) {
       mProtocolData.eRunMode = data.eRunMode;
       mbtn_autoPtr->setSelected(mProtocolData.eRunMode == E_RUN_MODE_MANUAL);
       if (mProtocolData.eRunMode != E_RUN_MODE_MANUAL) {
            mbtn_external_windPtr->setText(mProtocolData.externalWindSpeedLevel);
            mbtn_internal_windPtr->setText(mProtocolData.internalWindSpeedLevel);
       }
  }
    ...
}
```

在代码里面我们看到一个变量 mProtocolData，这是一个页面里面的 static 的变量。在 onUI_init()的时候会初始化。 如：

```
static SProtocolData mProtocolData;
static void onUI_init() {
  //Tips :添加 UI 初始化的显示代码到这里,如:mText1->setText("123");
  mProtocolData = getProtocolData(); // 初始化串口数据的结构体。
  // 开始初始化页面的 UI 显示
}
```

3. 串口数据发送

打开 **ProtocolSender.cpp** 当 APP 层需要发送数据给 MCU（或其他设备）的时候直接调用 **sendProtocol** 就可以了。 具体的协议封装由 **sendProtocol** 方法完成。用户可以根据自己的协议要求修改这个部分的代码。

```
/**
 * 需要根据协议格式进行拼接，以下只是个模板
 */
bool sendProtocol(const UINT16 cmdID, const BYTE *pData, BYTE len) {
  BYTE dataBuf[256];

  dataBuf[0] = CMD_HEAD1;
  dataBuf[1] = CMD_HEAD2;                  // 同步帧头

  dataBuf[2] = HIBYTE(cmdID);
  dataBuf[3] = LOBYTE(cmdID);          // 命令字节

  dataBuf[4] = len;

  UINT frameLen = 5;

  // 数据
  for (int i = 0; i < len; ++i) {
       dataBuf[frameLen] = pData[i];
```

```
        frameLen++;
    }

#ifdef PRO_SUPPORT_CHECK_SUM
    // 校验码
    dataBuf[frameLen] = getCheckSum(dataBuf, frameLen);
    frameLen++;
#endif

    return UARTCONTEXT->send(dataBuf, frameLen);
}
```

当界面上有个按键按下的时候可以操作：

```
BYTE mode[] = { 0x01, 0x02, 0x03, 0x04 };
sendProtocol(0x01, mode, 4);
```

5.3. 通讯案例实战

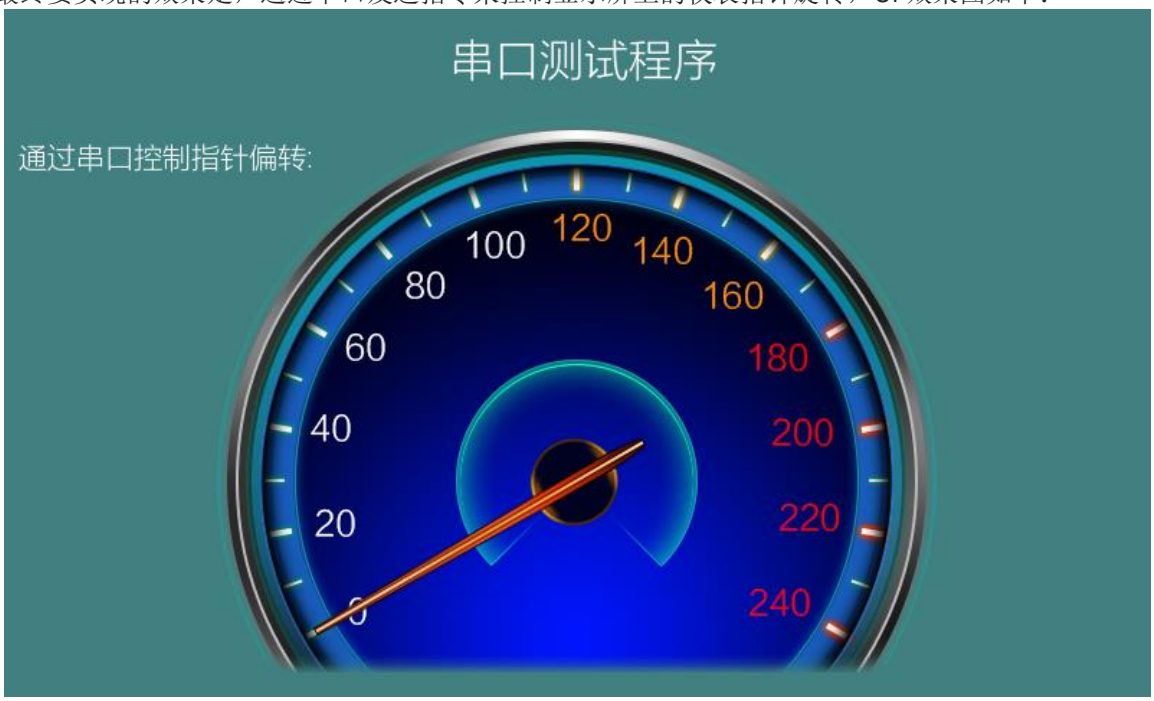
串口通讯主要有以下 4 点内容：

1. 接收数据
2. 解析数据
3. 展示数据
4. 发送数据

其中 解析数据 部分较为复杂，需要根据具体的通讯协议做相应的改动。

以前面的通讯协议为例，实现自己的一个简单的通讯程序：

我们最终要实现的效果是，通过串口发送指令来控制显示屏上的仪表指针旋转，UI 效果图如下：



我们只需要修改 **3 处** 地方就可以实现控制仪表指针旋转。

1）新增自己的协议指令 **CMDID_ANGLE** 对应的值为 **0x0001**

协议头（2 字节）	命令（2 字节）	数据长度（1 字节）	数据（N）	校验(1 字节 可选)
0xFF55	0x0001（见以下 CMDID_ANGLE）	1	angle	checksum

协议数据结构体里我们新增 1 变量，见 **ProtocolData.h :**

```
/******************** CmdID ************************/
#define CMDID_POWER                 0x0
#define CMDID_ANGLE                 0x1   // 新增 ID
/***************************************************/

typedef struct {
 BYTE power;
 BYTE angle;     // 新增变量，用于保存指针角度值
 } SProtocolData;
```

2) 由于我们使用的还是前面定义的协议格式，所以这里协议解析的部分我们不需要做任何改动，只需在 **procParse** 中处理对应的 CmdID 值即可:

```
/**
 * 解析每一帧数据
 */
static void procParse(const BYTE *pData, UINT len) {
 // CmdID
 switch (MAKEWORD(pData[3], pData[2])) {
 case CMDID_POWER:
        sProtocolData.power = pData[5];
        break;

 case CMDID_ANGLE: // 新增部分，保存角度值
        sProtocolData.angle = pData[5];
        break;
 }

 // 通知协议数据更新
 notifyProtocolDataUpdate(sProtocolData);
 }
```

3) 我们再来看界面接收到协议数据的回调接口，见 **logic/mainLogic.cc** :

```
static void onProtocolDataUpdate(const SProtocolData &data) {
 // 串口数据回调接口

 // 设置仪表指针旋转角度
 mPointer1Ptr->setTargetAngle(data.angle);
 }
```

完成以上流程后，接下来我们只需要通过 MCU 向屏发送相应的指令就可以看到仪表指针的旋转了；为了简单起见，我们这个程序里不做 checksum 校验，协议数据如下:

```
帧头        CmdID     数据长度     角度值
0xFF 0x55   0x00 0x01   0x01      angle
```

我们可以在 **CommDef.h** 文件中打开 DEBUG_PRO_DATA 宏，打印接收到的协议数据:

到此，串口的**接收数据** —> **解析数据** —> **展示数据** 就算完成了。
最后我们再来模拟一下串口**发送数据**：这里，我们给出的程序里，开启了一个定时器，2s 模拟一次数据发送：

```
static bool onUI_Timer(int id) {
 // 模拟发送串口数据
 BYTE data = rand() % 200;
 sendProtocol(CMDID_ANGLE, &data, 1);

 return true;
 }
```

以上代码其实就是模拟设置角度值，我们可以通过短接屏上通讯串口的 TX 和 RX，实现**自发自收**，也是可以看到仪表指针旋转的。

5.4. 串口配置

5.4.1 串口的选择

由于软件与硬件的设计兼容问题，导致软件串口号与硬件上的串口号标识可能存在不同的情况，具体对应关系如下：

- Z11 系列平台

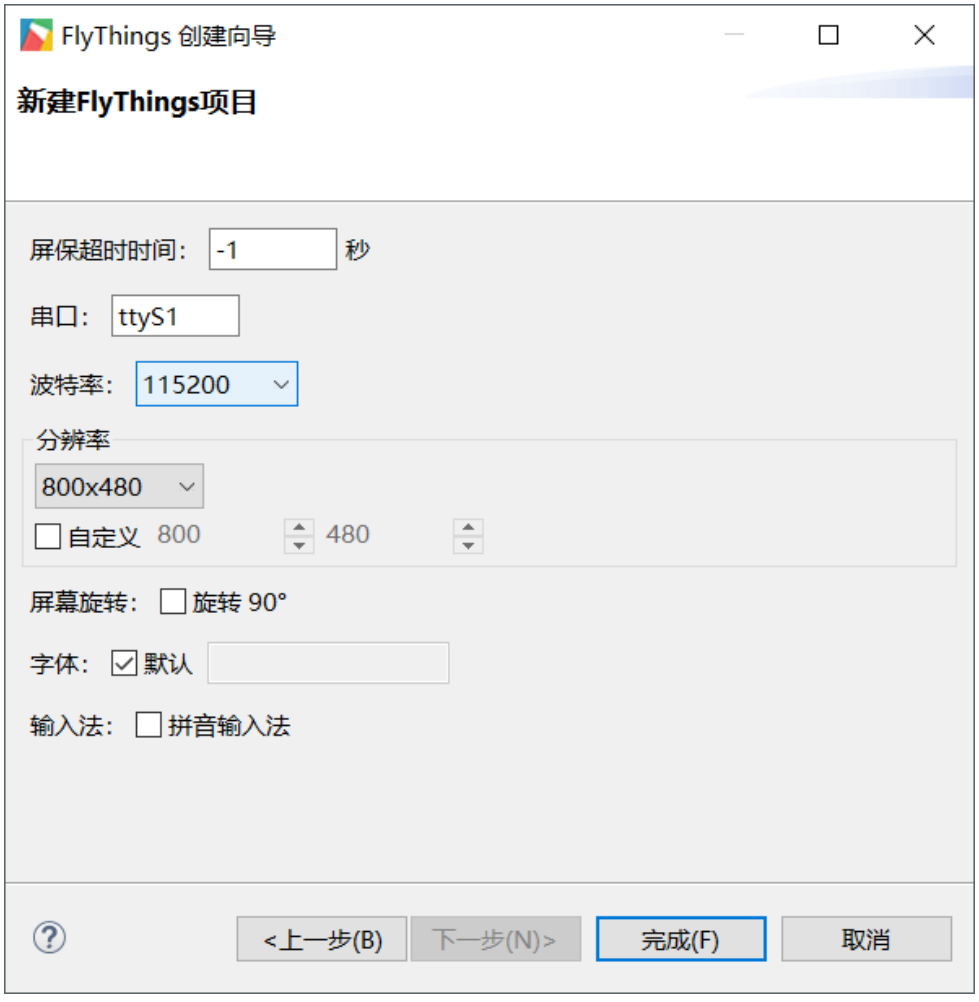
软件串口号	硬件串口号
ttyS0	UART1
ttyS1	UART2

- Z6 系列平台

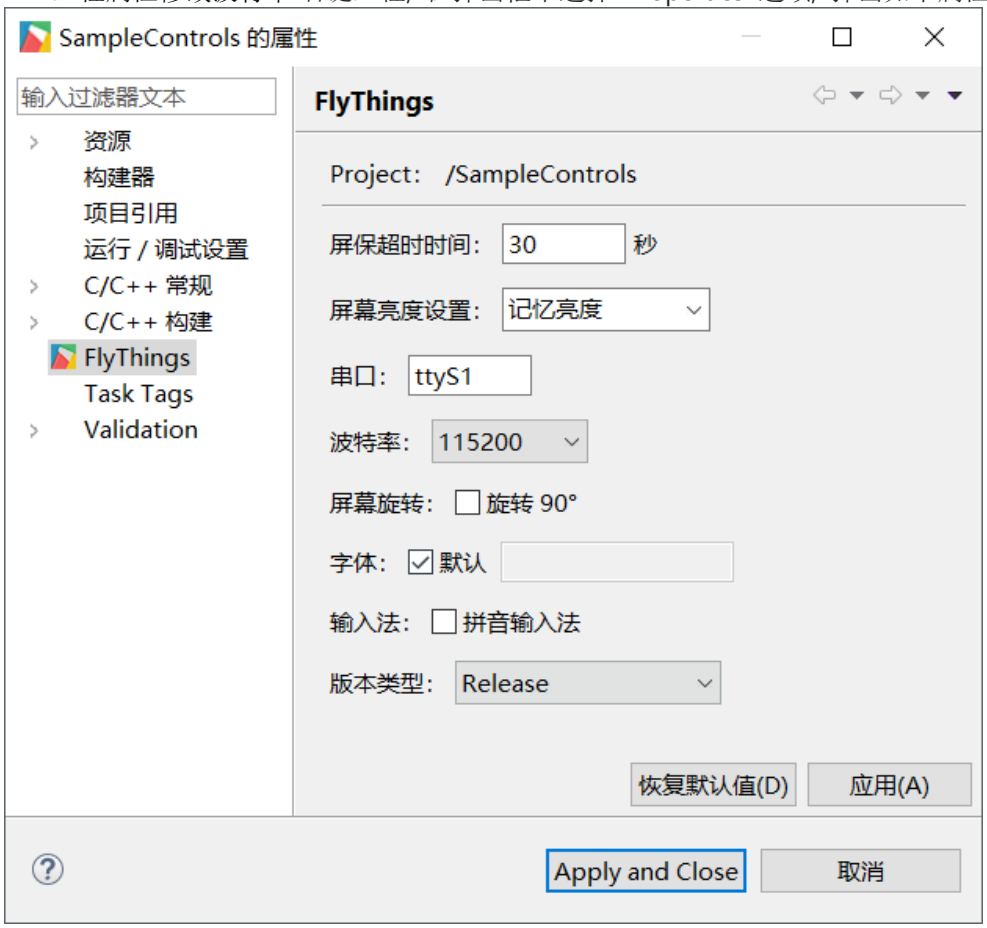
软件串口号	硬件串口号
ttyS0	UART0
ttyS1	UART1
ttyS2	UART2

5.4.2 串口波特率配置

- 新建工程时配置波特率



- 工程属性修改波特率 右键工程，在弹出框中选择 Properties 选项，弹出如下属性框



5.4.3 串口打开和关闭

打开源码路径 **jni/Main.cpp**，可以看到在程序初始化和销毁时分别有对串口打开和关闭的操作。

```
void onEasyUIInit(EasyUIContext *pContext) {
    LOGD("onInit\n");
    // 打开串口
    UARTCONTEXT->openUart(CONFIGMANAGER->getUartName().c_str(),
CONFIGMANAGER->getUartBaudRate());
}

void onEasyUIDeinit(EasyUIContext *pContext) {
    LOGD("onDestroy\n");
    // 关闭串口
    UARTCONTEXT->closeUart();
 }
```

5.5. 多串口配置

常规项目默认只支持一个串口，如果您要使用双串口，甚至多串口，可在常规项目中修改串口部分代码，以支持多串口。

修改部分说明：

- uart 部分代码经过修改，所以项目属性里的串口配置失效。
- 串口号及波特率请参照 **jni/uart/UartContext.cpp** 文件中的 **init()**函数修改。

```
void UartContext::init() {
    uart0 = new UartContext(UART_TTYS0);
    uart0->openUart("/dev/ttyS0", B9600);

    uart1 = new UartContext(UART_TTYS1);
    uart1->openUart("/dev/ttyS1", B9600);
}
```

- 发送数据到指定串口。

```
unsigned char buf[2] = {1, 1};
sendProtocolTo(UART_TTYS1, 1, buf, 2); //发送到 TTYS1 串口

unsigned char buf[2] = {0};
sendProtocolTo(UART_TTYS0, 1, buf, 2);//发送到 TTYS0 串口
```

- 接收串口数据的方式与常规项目相同。

  如果需要区分数据来自哪一个串口，建议在 **SProtocolData** 结构体中添加一个字段标识该帧来自哪一个串口。

  修改 **uart/ProtocolData.h**

```
typedef struct {
        BYTE power;
        int uart_from; //来自哪一个串口
} SProtocolData;
```

  修改 **uart/ProtocolParser.cpp**：

```
/**
    * 解析每一帧数据
    */
  static void procParse(int uart, const BYTE *pData, UINT len) {
        // CmdID
        switch (MAKEWORD(pData[3], pData[2])) {
        case CMDID_POWER:
            sProtocolData.power = pData[5];
            break;
        }

        sProtocolData.uart_from = uart; //标识该帧来自哪一个串口
        // 通知协议数据更新
        notifyProtocolDataUpdate(sProtocolData);
}
```

  然后在 **Logic.cc** 中，可以使用 **uart_from** 字段判断该数据来自哪一个串口。

```
static void onProtocolDataUpdate(const SProtocolData &data) {
    LOGD("onProtocol %d", data.uart_from);
```

```
    char buf[128] = {0};
    snprintf(buf, sizeof(buf), "收到串口 %d 的数据", data.uart_from);
    mTextview1Ptr->setText(buf);
}
```

6. 网络控制

6.1. WIFI 设置

WIFI 使用样例见 UuidSSDPlayer/stdc++/zk_full/jni/hotplugdetect/**wifidetect**/。

7. 多媒体

7.1. 视频播放

视频播放样例见 UuidSSDPlayer/**myplayer**/。

7.2. 音频播放

音频播放样例见 UuidSSDPlayer/**myplayer**/。

8. 系统操作

8.1. 数据存储

在某些应用场景中需要永久存储一些信息，如存储用户名称、密码或其他配置的一些信息，像这种数据内容比较少的情况，使用数据库去存储，操作起来会很繁琐，这里我们提供了一套简单的数据存储接口，以键-值对的方式存储，接口见 **storage/StoragePreferences.h**：

- 所需头文件

```
#include "storage/StoragePreferences.h"
```

- 主要接口

```
// 存储接口
    static bool putString(const std::string &key, const std::string &val);
    static bool putInt(const std::string &key, int val);
    static bool putBool(const std::string &key, bool val);
    static bool putFloat(const std::string &key, float val);

    // 删除指定键
    static bool remove(const std::string &key);
    // 清空存储数据
    static bool clear();

    // 获取接口，获取不到对应键值，返回 defVal 默认值
    static std::string getString(const std::string &key, const std::string &defVal);
    static int getInt(const std::string &key, int defVal);
    static bool getBool(const std::string &key, bool defVal);
    static float getFloat(const std::string &key, float defVal);
```

示例：

```
// 点击 Button1 存储用户名，"username":"zkswe"
static bool onButtonClick_Button1(ZKButton *pButton) {
    // 存储用户名
    StoragePreferences::putString("username", "zkswe");
    return false;
}

// 点击 Button2 获取用户名
static bool onButtonClick_Button2(ZKButton *pButton) {
    // 获取用户名
    std::string username = StoragePreferences::getString("username", "null");
    LOGD("username %s\n", username.c_str());
    return false;
}

// 点击 Button3 删除用户名
```

```
static bool onButtonClick_Button3(ZKButton *pButton) {
    // 删除用户名
    StoragePreferences::remove("username");
    return false;
}
```

8.2. 屏幕背光操作

所需头文件：

```
#include "utils/BrightnessHelper.h"
```

亮度调节：

- 调节背光亮度

  亮度范围是 0~100（注意：0 并不等于关屏）

  ```
  //将屏幕亮度调整为 80
  BRIGHTNESSHELPER->setBrightness(80);
  ```

- 获取当前亮度值

  ```
  BRIGHTNESSHELPER->getBrightness();
  ```

开关屏幕背光

- 关屏

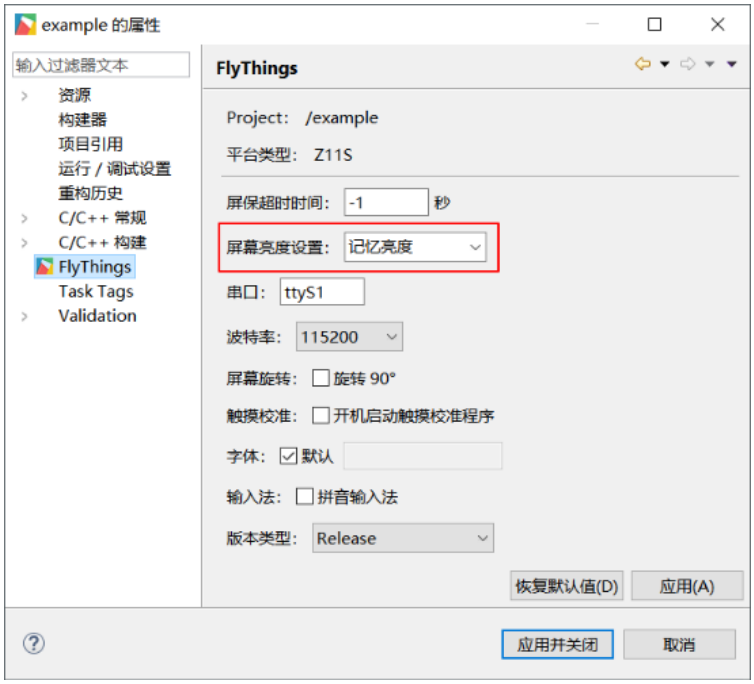
  ```
  BRIGHTNESSHELPER->screenOff();
  ```

- 开屏

  ```
  BRIGHTNESSHELPER->screenOn();
  ```

记忆亮度

系统开机起来默认是记忆最后调节的亮度值，如果想要修改为不记忆亮度或设置固定的亮度值，可以打开工程的属性进行修改：



8.3. 系统时间

- 所需头文件

```
#include "utils/TimeHelper.h"
```

tm 结构体各字段解释

```
struct tm {
    int tm_sec; /* 秒–取值区间为[0,59] */
  int tm_min; /* 分 - 取值区间为[0,59] */
  int tm_hour; /* 时 - 取值区间为[0,23] */
  int tm_mday; /* 一个月中的日期 - 取值区间为[1,31] */
  int tm_mon; /* 月份（从一月开始，0 代表一月） - 取值区间为[0,11] */
  int tm_year; /* 年份，其值从 1900 开始 */
    ...
}
```

- 获取当前日期

```
struct tm *t = TimeHelper::getDateTime();
```

显示时间代码样例

```
static void updateUI_time() {
    char timeStr[20];
    static bool bflash = false;
    struct tm *t = TimeHelper::getDateTime();

    sprintf(timeStr, "%02d:%02d:%02d", t->tm_hour,t->tm_min,t->tm_sec);
    mTextTimePtr->setText(timeStr); // 注意修改控件名称

    sprintf(timeStr, "%d 年%02d 月%02d 日", 1900 + t->tm_year, t->tm_mon + 1, t->tm_mday);
    mTextDatePtr->setText(timeStr); // 注意修改控件名称

    static const char *day[] = { "日", "一", "二", "三", "四", "五", "六" };
    sprintf(timeStr, "星期%s", day[t->tm_wday]);
    mTextWeekPtr->setText(timeStr); // 注意修改控件名称
}
```

- 设置时间代码样例

```
// 利用 tm 结构体设置时间
static void setSystemTime() {
  struct tm t;
  t.tm_year = 2017 - 1900;   //年
  t.tm_mon = 9 - 1;          //月
  t.tm_mday = 13;             //日
  t.tm_hour = 16;            //时
  t.tm_min = 0;              //分
  t.tm_sec = 0;              //秒
  TimeHelper::setDateTime(&t);
}
```

```
// 或者用字符串设置时间   date str format: 2017-09-13 16:00:00
TimeHelper::setDateTime("2017-09-13 16:00:00");
```

8.4. 获取设备唯一 ID

- 所需头文件

```
#include "security/SecurityManager.h"
```

- 读取设备 ID

```
// 设备 id 共 8 个字节
unsigned char devID[8];
// 成功返回 true，失败返回 false
bool ret = SECURITYMANAGER->getDevID(devID);
```

8.5. TF 卡拔插监听

通过注册监听接口，我们可以知道 TF 卡的拔插状态；这里我们首先需要实现自己的监听类：

```
#include "os/MountMonitor.h"

class MyMountListener : public MountMonitor::IMountListener {
public:
    virtual void notify(int what, int status, const char *msg) {
        switch (status) {
        case MountMonitor::E_MOUNT_STATUS_MOUNTED:  // 插入
            // msg 为挂载路径
            LOGD("mount path: %s\n", msg);
             mMountTextviewPtr->setText("TF 卡已插入");
            break;

        case MountMonitor::E_MOUNT_STATUS_REMOVE:    // 移除
            // msg 为卸载路径
            LOGD("remove path: %s\n", msg);
            mMountTextviewPtr->setText("TF 卡已移除");
            break;
        }
    }
};
```

定义监听对象：

```
static MyMountListener sMyMountListener;
```

注册监听：

```
MOUNTMONITOR->addMountListener(&sMyMountListener);
```

当我们不再需要监听时，需要移除监听：

```
MOUNTMONITOR->removeMountListener(&sMyMountListener);
```

8.6. 线程封装

标准的线程操作接口使用起来特别繁琐，而且也很容易出现问题；为此，我们框架对其进行了一些封装，见 **jni/include/system** 目录下头文件：

- Thread.h：线程类
- Mutex.h：锁类
- Condition.h：条件变量类

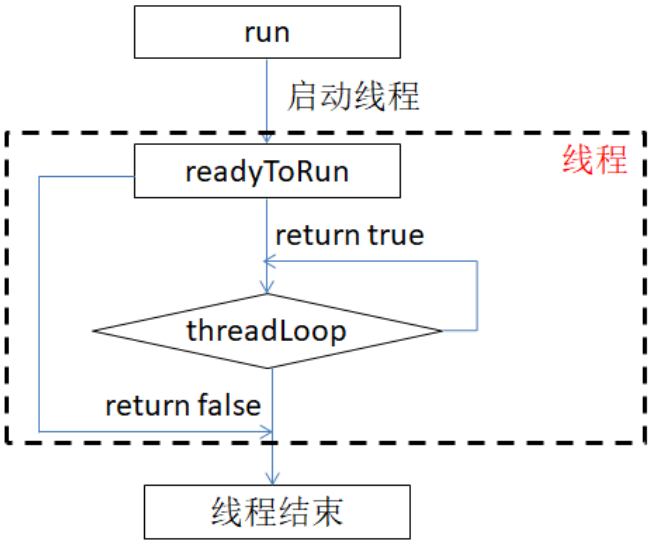
现在创建线程变得非常简单，只要继承 `Thread`，在 `threadLoop()`函数中添加自己的逻辑代码就可以了，另外还有一函数 `readyToRun()`线程创建完后会调一次，可以在该函数中做一些初始化的操作。

```
#include "system/Thread.h"

class TestThread : public Thread {
protected:
    virtual bool readyToRun() {
        // init something
        // return false 将退出线程
        return true;
    }
    virtual bool threadLoop() {
        // do something
        // return false 将退出线程
        return true;
    }
};
static TestThread sTestThread;
```

启动线程：

```
sTestThread.run();
```

完整流程如下：



除了以上两个函数返回 **false** 来退出线程外，还可以调用 **requestExit()** 或 **requestExitAndWait()** 请求退出线程，其中 **requestExit()** 调用完函数立即返回，并不代表线程也退出了，**requestExitAndWait()** 会等待线程退出才返回；调用 **requestExit()** 或 **requestExitAndWait()** 后 **Thread::exitPending()**将返回 **true** ，在 **threadLoop()** 可以通过检测 **exitPending()**判断是否需要退出线程。

定义好 **Mutex** 变量，使用如下代码就可以对相应的代码段加锁了：

是的

```
static Mutex sLock;
Mutex::Autolock _l(sLock);
```

举一个我们项目里面用到的例子，见 **jni/uart/ProtocolParser.cpp** ：

```
void registerProtocolDataUpdateListener(OnProtocolDataUpdateFun pListener) {
    Mutex::Autolock _l(sLock);
    LOGD("registerProtocolDataUpdateListener\n");
    if (pListener != NULL) {
        sProtocolDataUpdateListenerList.push_back(pListener);
    }
}
```

这样锁的作用域就是 **registerProtocolDataUpdateListener** 整个函数，函数返回后，自动解锁；其实以上锁的作用域就是变量 _l 的生命周期区间。

8.7. GPIO 操作

操作接口 **jni/include/utils/GpioHelper.h**

```
class GpioHelper {
public:
    // 返回值:    -1 失败，  1/0(高/低电平) 成功
    static int input(const char *pPin);
    // 返回值:    -1 失败，  0 成功
    static int output(const char *pPin, int val);
};
```

- Z11S 平台上

  目前只留了 3 组 io 口可以操作：

```
// 3 组 io 口定义
#define GPIO_PIN_B_02          "B_02"
#define GPIO_PIN_B_03          "B_03"
#define GPIO_PIN_E_20          "E_20"

#include "utils/GpioHelper.h"

// 读 B02 io 口状态
GpioHelper::input(GPIO_PIN_B_02);

// B02 io 口输出高电平
GpioHelper::output(GPIO_PIN_B_02, 1);
```

- SVPB 模组

有以下 12 组 io 口可以操作：

```
// 12 组 io 口定义
#define GPIO_PIN_7        "PIN7"
#define GPIO_PIN_8        "PIN8"
#define GPIO_PIN_9        "PIN9"
#define GPIO_PIN_10       "PIN10"
#define GPIO_PIN_11       "PIN11"
#define GPIO_PIN_12       "PIN12"
#define GPIO_PIN_13       "PIN13"
#define GPIO_PIN_14       "PIN14"
#define GPIO_PIN_23       "PIN23"
#define GPIO_PIN_24       "PIN24"
#define GPIO_PIN_26       "PIN26"
#define GPIO_PIN_27       "PIN27"

#include "utils/GpioHelper.h"

// 读 PIN_7 io 口状态
GpioHelper::input(GPIO_PIN_7);

// PIN_7 io 口输出高电平
GpioHelper::output(GPIO_PIN_7, 1);
```

8.8. SPI 操作

目前仅 **SV50PB 模组**支持该功能，操作接口 **jni/include/utils/SpiHelper.h** ，使用说明：

```
// 所需头文件
#include "utils/SpiHelper.h"

static void testSpi() {
    uint8_t tx[512], rx[512];

    /**
     * 定义变量
     *
     * 参数 1:  spi 总线号
     * 参数 2:  模式，可选值:  SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3
     * 参数 3:  spi 时钟频率，这里设置了 50M
     * 参数 4:  一个字有多少位，默认值:  8
     * 参数 5:  位顺序，true:  表示低位在前，false:  表示高位在前:  默认值:  false，高位在前
     */
    SpiHelper spi(1, SPI_MODE_0, 50*1000*1000, 8, false);

    memset(tx, 0, 512);
    memset(rx, 0, 512);

    tx[0] = 0x4B;

    /**
     * 单工写
     *
     * 参数 1:  写数据地址
     * 参数 2:  数据长度
```

```
     */
    if (!spi.write(tx, 5)) {
        LOGD("spi tx error!\n");
    }

    /**
     * 单工读
     *
     * 参数 1:  读数据地址
     * 参数 2:  数据长度
     */
    if (!spi.read(rx, 8)) {
        LOGD("spi rx error!\n");
    }

    for (int i = 0; i < 8; i++) {
        LOGD("spi[%d]=0x%x\n", i, rx[i]);
    }

    /**
     * 半双工传输
     *
     * 参数 1:  写数据地址
     * 参数 2:  写数据长度
     * 参数 3:  读数据地址
     * 参数 4:  读数据长度
     */
    if (!spi.halfduplexTransfer(tx, 5, rx, 8)) {
        LOGD("spi spi_halfduplex_transfer rx error!\n");
    }

    for (int i = 0; i < 8; i++) {
        LOGD("spi[%d]=0x%x\n", i, rx[i]);
    }
}
```

其他接口操作请参见头文件注释说明。

8.9. I2C 操作

目前仅 **SV50PB 模组**支持该功能，操作接口 jni/include/utils/I2CHelper.h ，使用说明：

```
// 所需头文件
#include "utils/I2CHelper.h"

#define CFG_L       0x47
#define CFG_H       0x80
#define VER_L       0x41
#define VER_H       0x81

static void testI2C() {
    uint8_t tx[512], rx[512];
    memset(tx, 0, 512);
    memset(rx, 0, 512);
```

```
    /**
     * 定义变量
     *
     * 参数 1:  i2c 总线号
     * 参数 2:  从机地址
     * 参数 3:  超时,  单位: ms
     * 参数 4:  重试次数
     */
    I2CHelper i2c(0, 0x5e, 1000, 5);

    tx[0] = CFG_H;
    tx[1] = CFG_L;

    /**
     * 单工写
     *
     * 参数 1:  写数据地址
     * 参数 2:  数据长度
     */
    if (!i2c.write(tx, 2)) {
        LOGD("i2c tx cfg error!\n");
    }

    /**
     * 单工读
     *
     * 参数 1:  读数据地址
     * 参数 2:  数据长度
     */
    if (!i2c.read(rx, 1)) {
        LOGD("i2c rx cfg error!\n");
    }

    LOGD("i2c reg[0x%x%x]=%x\n", CFG_H, CFG_L, rx[0]);
    memset(rx, 0, 512);

    /**
     * 半双工传输，即共用读写，中间无 stop 信号
     *
     * 参数 1:  写数据地址
     * 参数 2:  写数据长度
     * 参数 3:  读数据地址
     * 参数 4:  读数据长度
     */
    if (!i2c.transfer(tx, 2, rx, 1)) {
        LOGD("i2c i2c_transfer cfg error!\n");
    }

    LOGD("i2c reg[0x%x%x]=%x\n", CFG_H, CFG_L, rx[0]);

    tx[0] = VER_H;
    tx[1] = VER_L;
```

```
    if (!i2c.write(tx, 2)) {
        LOGD("i2c tx ver error!\n");
    }

    if (!i2c.read(rx, 1)) {
        LOGD("i2c rx ver error!\n");
    }

    LOGD("i2c reg[0x%x%x]=%x\n", VER_H, VER_L, rx[0]);
    memset(rx, 0, 512);
    if (!i2c.transfer(tx, 2, rx, 1)) {
        LOGD("twi i2c_transfer ver error!\n");
    }

    LOGD("i2c reg[0x%x%x]=%x\n", VER_H, VER_L, rx[0]);
}
```

其他接口操作请参见头文件注释说明.

8.10. ADC 操作

目前仅 **SV50PB 模组**支持该功能，操作接口 **jni/include/utils/AdcHelper.h** ，使用说明:

```
// 所需头文件
#include "utils/AdcHelper.h"

static void testAdc() {
  /**
   * 设置 adc 使能状态
   *
   * 参数： true 使能， false 禁止   默认是使能状态
   */
  AdcHelper::setEnable(true);

  for (int i = 0; i < 10; i++) {
      // 读取 adc 值
      int val = AdcHelper::getVal();
      LOGD("adc val = %d\n", val);
  }
}
```

9. 国际化

9.1. 多语言翻译

提供了多国语言翻译的功能，方便国际化。

9.1.1 如何添加翻译

1. 首先通过新建向导，创建翻译文件。
2. 创建完毕后，项目下会生成一个名为 **i18n** 的子文件夹，并且添加了默认的 tr 翻译文件。每个 tr 文件的文件名代表相应的语言 (请勿随意修改文件名)。
3. 双击打开 tr 文件，可以看出该内容为 xml 格式。每一个 string 标签表示一条翻译。**name** 属性用于给这条翻译取别名（建议统一为英文、数字、下划线组合命名），之后在代码中用别名代表该字符串。注意：同一配置文件内，别名不能重复。

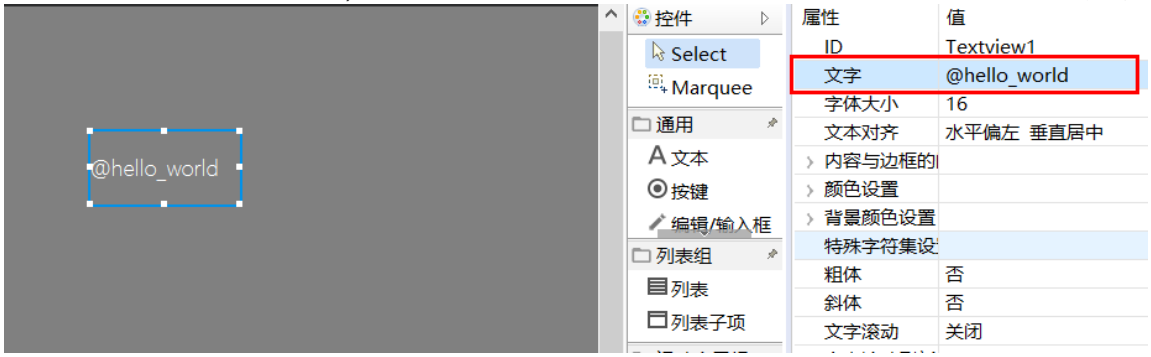
```
<?xml version="1.0" encoding="utf-8"?>
 <resources>
      <string name="hello_world">你好,世界!</string>
      <string name="hello_sir">你好，先生</string>
 </resources>
```

如果希望在字符串中换行，则用
转义，如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
      <string name="new_line_test">第一行&#x000A;第二行</string>
</resources>
```

如上所示，我们将 **“你好,世界!”** 这个字符串取别名为 **hello_world**、将 **你好，先生** 这个字符串取别名为 **hello_sir**，如果我们要添加更多的翻译，我们只需要按照示例添加 string 标签即可。

4. 既然是多国语言的翻译，我们就要为每个语言都添加相同 name 的标签。这样，当我们切换语言时，系统就会根据 name 属性，将内容替换。
5. 配置文件添加完毕后，我们就可以在 ui 文件以及代码中使用。
6. 打开 ui 文件，我们可以在文本属性里输入 **@hello_world** (我们用**@**符号后面跟上配置文件中的别名，来表示这个字符串需要翻译)。然后，我们下载运行，可以看到文本的内容已经自动翻译为 **你好，世界！**



7. 我们还可以在代码中对字符串进行翻译。以前我们设置字符串是用 **setText()** 这个成员方法，如果我们需要自动翻译，则需要使用 **setTextTr()** 成员方法。 例如：

```
/**
 * 当界面构造时触发
 */
static void onUI_init() {
 // setTextTr 参数为翻译配置文件中的 name 值，注意：这里传入的字符串前面不需要带@符号
 mTextview1Ptr->setTextTr("hello_world");
}
```

下载运行，我们可以看到内容已经替换为对应的字符串。

8. 我们还可以获取当前语言 name 对应的 value 值，然后做一些类似拼接的操作等等：

```
#include "manager/LanguageManager.h"

static bool onButtonClick_Button1(ZKButton *pButton) {
    //LOGD(" ButtonClick Button1 !!!\n");
    std::string hello = LANGUAGEMANAGER->getValue("hello");
    std::string world = LANGUAGEMANAGER->getValue("world");
    std::string ret = hello + " " + world;
    LOGD("ret: %s\n", ret.c_str());
    return false;
}
```

9.1.2 如何切换语言

1. 系统默认的语言为 **zh_CN** （中文简体）。
2. 你可以打开系统内置切换语言的界面选择语言。
   添加如下代码打开该界面：

```
EASYUICONTEXT->openActivity("LanguageSettingActivity");
```

   或者使用如下 API 自由切换语言：

```
EASYUICONTEXT->updateLocalesCode("zh_CN"); //设置为中文
EASYUICONTEXT->updateLocalesCode("en_US"); //设置为英文
EASYUICONTEXT->updateLocalesCode("ja_JP"); //设置为日语
```

9.1.3 字体要求

多语言翻译需要字体的支持。如果字体中不存在该文字，则会显示不正常，所以，字体中一定要包含多国语言的文字。

10. 升级和调试

10.1. ADB 调试

可以通过 USB 线 或者 WIFI 快速下载程序到机器中。具体步骤如下：

1. 首先确保电脑与机器成功连接，连接方式有两种：
   注意：如果您购买的是带有 WIFI 功能的版本。那么只能通过 WIFI 连接。USB 线不能使用；
   以太网版本优先使用 USB 线连接，如果 USB 连接不成功才使用 WIFI 连接（即网络连接）方式，如果都不能成功连接请联系我们。
   同理，如果您购买的是不带 WIFI 功能的版本，那么只能通过 USB 连接。
   USB 线连接成功可以看到

   1） 使用 USB 线连接电脑和机器。如果电脑能将机器识别为 Android 设备，表示连接正常。 如果不能正常连接，电脑提示驱动问题，可尝试下载更新驱动。
   2） 通过 WIFI 方式连接。（这种方式需要机器支持 WIFI 功能。）
   先进入机器的 WIFI 设置界面，将机器连接到与电脑相同的网络，也就是说，电脑和机器必须接入同一个 WIFI。（如果不同的网络会导致后续下载程序失败）。网络连接成功后，点击 WIFI 设置界面右上角菜单按钮查看机器的 IP 地址，然后，打开 IDE 开发工具，在菜单栏上，依次选择菜单 **调试配置 -> ADB IP 配置**， 将机器 IP 填入，选择确定。工具将尝试与机器连接，如果提示连接成功，则表示正常。如果提示失败，则需要检查 IP 是否正确？机器连接 WIFI 是否正常？
   WIFI 设置界面

2. 下载调试

完成上一步后，就可以直接下载程序了。在项目资源管理器中，选中项目名，右键，在弹出菜单中选择 **下载调试** 菜单， 选择后，它会先自动编译一次，编译成功后，再将程序下载到机器中，如果没有提示错误，那么你就可以看到机器程序已经得到了更新。

同样，在选中项目后，你还可以使用快捷键 **Ctrl + Alt + R** 下载调试。

如果电脑上连接有 Android 手机，可能会与机器造成冲突，导致下载失败。使用时，建议暂时断开 Android 手机连接。通过该方式运行程序，并不能将程序固化到设备中，如果您拔掉 TF 卡或者断电重启，程序将自动恢复。如果您希望固化程序到设备中，可以选择制作升级镜像，然后升级即可。

10.2. 查看打印日志

10.2.1 添加日志

添加日志所需头文件

```
#include "utils/Log.h"
```

打印统一调用 LOGD 或 LOGE 宏输出，使用方法与 C 语言的 `printf` 相同；默认生成的代码里就有调用的例子（默认被注释掉，需要时打开）：

```
static bool onButtonClick_Button1(ZKButton *pButton) {
        LOGD("onButtonClick_Button1\n");
        return true;
}
```

10.2.2 查看打印

我们有 2 种方式可以查看打印日志： **串口工具** 和 **ADB**。

1. **串口工具**

用 SecureCRT 工具或者其他串口工具连接板子，打开串口（波特率设置为 **115200**）。在终端上输入 **logcat** 后回车，点击屏上测试程序的按钮，会有如下打印输出。

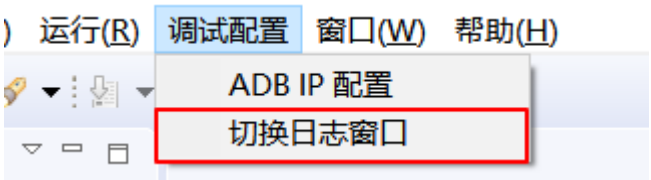
```
root@zkswe:/ # logcat
--------- beginning of /dev/log/main
D/zkgui   (  220): onButtonClick_Button1
```

如果打印过多，我们可以先输入 **logcat -c** 清除之前的打印，再在终端上输入 **logcat** 查看打印。

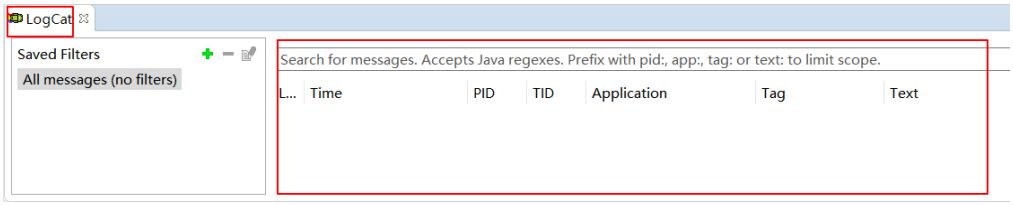
2. **ADB**

连接好 ADB 后，可以通过我们的工具查看程序的打印日志。 具体操作步骤如下：

1） 在菜单栏上 选择 **调试配置** -> **切换日志窗口** ，工具会切换到另一个界面。



2） 在新界面的左下角，选中 **LogCat**，如果连接正常，在右侧红框区域，你将看到机器的打印日志。



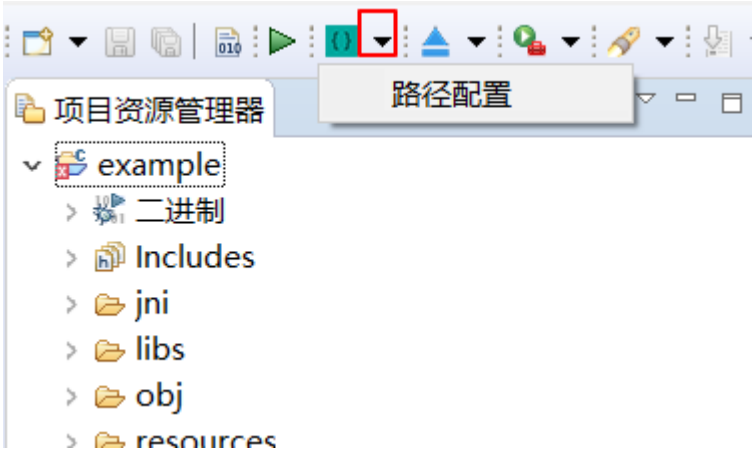
10.3. 从 TF 卡启动程序

当我们不能使用 ADB 下载程序时，还可以将程序下载到 TF 卡里，从 TF 卡来启动程序。
注意：TF 卡仅支持 FAT32 格式。

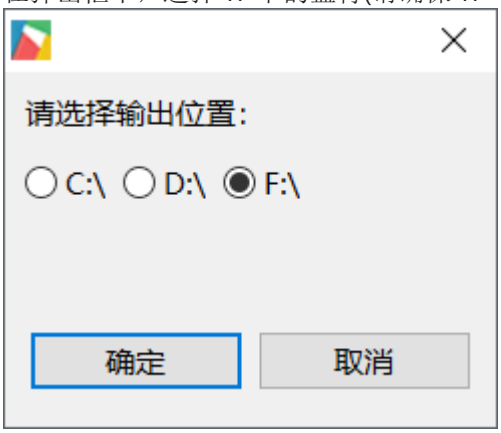
操作步骤：

1. 找到工具栏上的这个按钮

2. 点击旁边的黑色下拉箭头，在弹出的菜单中选择 **路径配置**



3. 在弹出框中，选择 TF 卡的盘符(请确保 TF 卡能正常使用），点击确定。



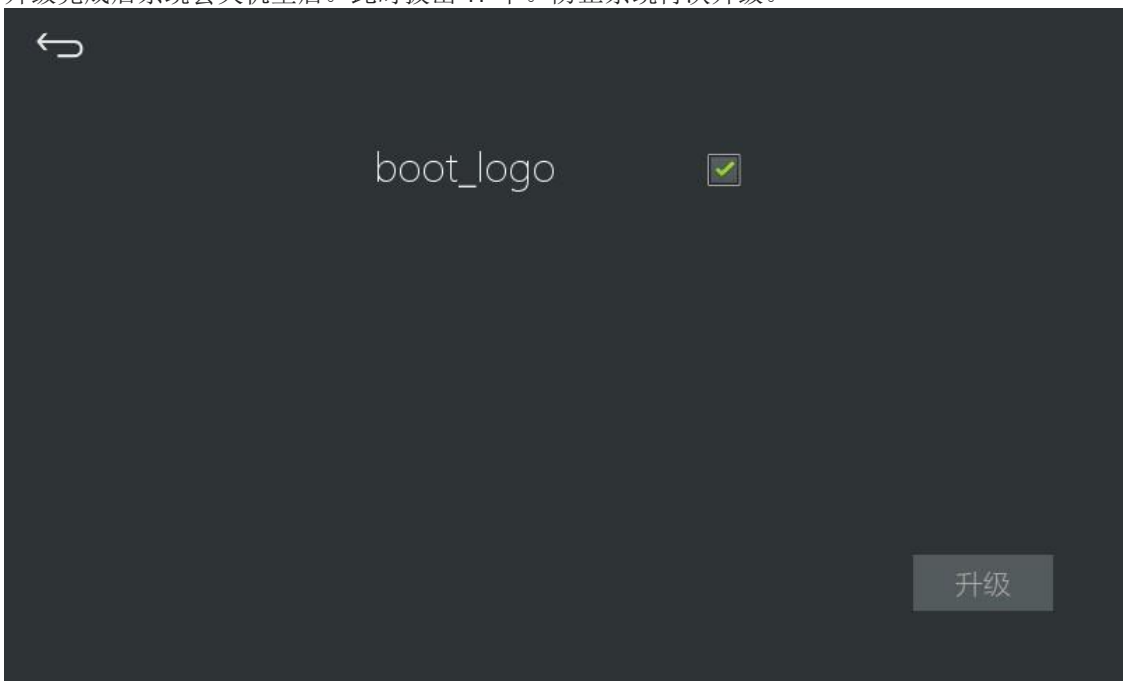
4. 在上面的步骤中，我们配置好了输出目录，现在点击下图中的按钮开始编译，它会将编译结果 打包输出到配置的盘符下。

5. 操作成功后， 将在配置的盘符下 生成 **EasyUI.cfg** 、**ui**、**lib**、 **font** 等目录和文件。
6. 将 TF 卡拔出，插入机器中，将机器重新上电，这时候，系统检测到 TF 卡里的文件，就会启动卡里的程序，而不是系统内的程序。

10.4. 升级开机 LOGO

升级开机 LOGO 步骤：

1. 首先准备一张开机 LOGO 图片，图片必须满足以下条件：
   1） 图片名称固定为 **boot_logo.JPG**，注意，文件后缀是大写的 **JPG**； 其他名称将无法识别；
   2） 图片大小不能超过 128KB；
   3） 图片分辨率必须与屏幕分辨率完全相同。
2. 将 **boot_logo.JPG** 拷贝到 TF 卡根目录下。
3. 将 TF 卡插入到机器中，然后设备会自动弹出升级提示。选择 **boot_logo** 然后点击升级即可。
4. 升级完成后系统会关机重启。此时拔出 TF 卡。防止系统再次升级。

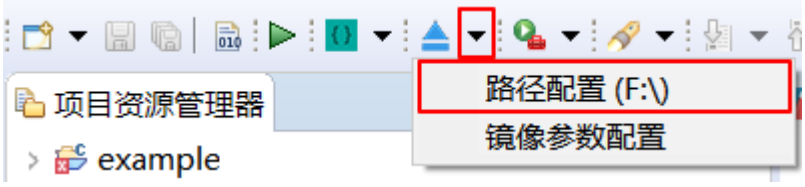


10.5. 制作升级镜像文件

升级镜像文件步骤：

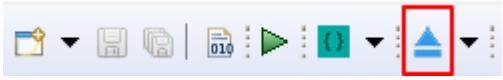
1. 找到工具栏上的这个按钮

2. 点击旁边的黑色下拉箭头，在弹出的菜单中选择 **路径配置**

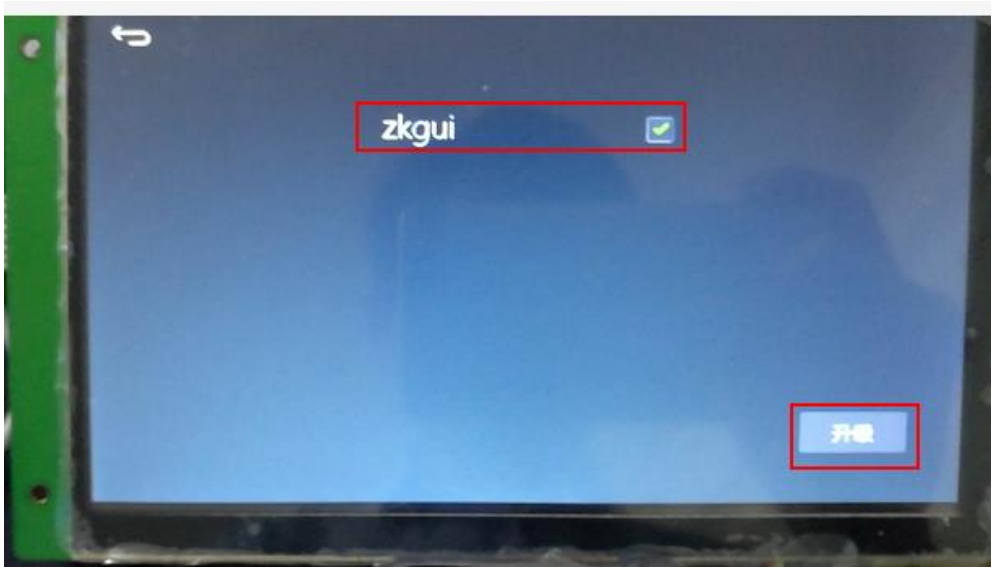


3. 在弹出框中，选择镜像文件的输出目录，点击确定。
4. 在上面的步骤中，我们配置好了输出目录，现在点击下图中的按钮开始编译，它会将编译结果 打包，并

生成 **update.img** 文件输出到配置的目录下。



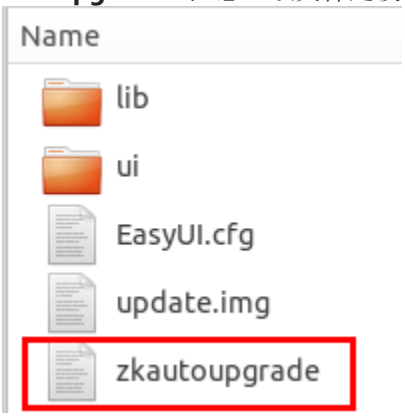
5. **update.img** 文件成功生成后，将其拷贝到 TF 卡里(**注意：使用前，请将 TF 卡格式化为 FAT32 格式**)，将 TF 卡插入机器中，机器重新上电，这时候，系统检测到 TF 卡里的文件，就会启动升级程序，在下图的界面中，勾选升级的项目，点击升级。升级完成后及时拔掉升级卡，防止重复升级。
   注意：TF 卡仅支持 FAT32 格式。



如果屏幕损坏或触摸不准情况下，导致不能通过点击按钮进行升级，那么这种情况下，我们可以通过自动升级这种方式来升级我们的系统。

10.6. 自动升级

在屏幕损坏或触摸不准情况下，想要对系统进行升级，我们可以在 TF 卡根目录下创建一个文件 **zkautoupgrade**（注意：该文件是没有后缀名的）



这样机器插卡后会自动勾选上升级项，默认 2s 后开始升级；如果需要控制其他时间后才开始升级，我们可以打开 zkautoupgrade 文件填相应的数字即可，单位为秒；升级完成后，系统重启，记得要拔出 TF 卡，防止再次自动升级。

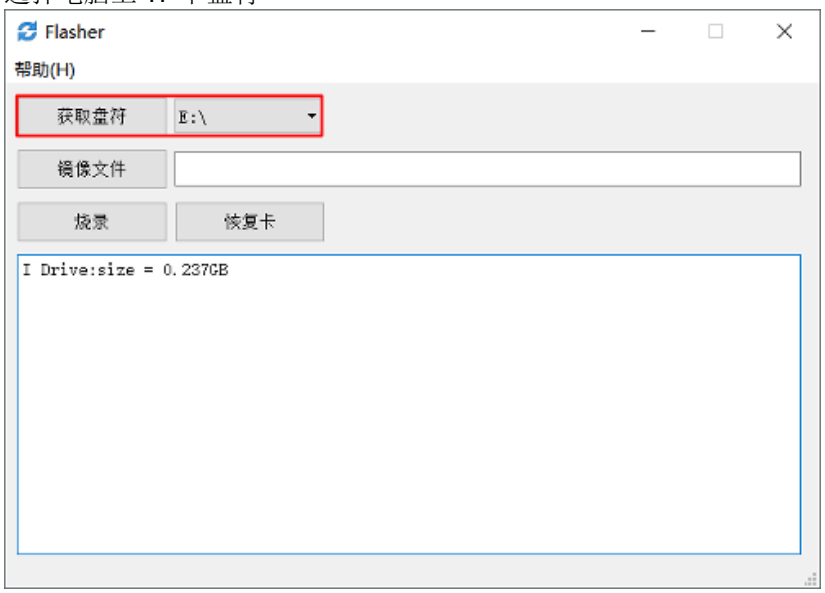
10.7. 制作刷机卡

注意：TF 卡的容量最大支持 16G，过大机器无法识别 TF 卡，无法升级。

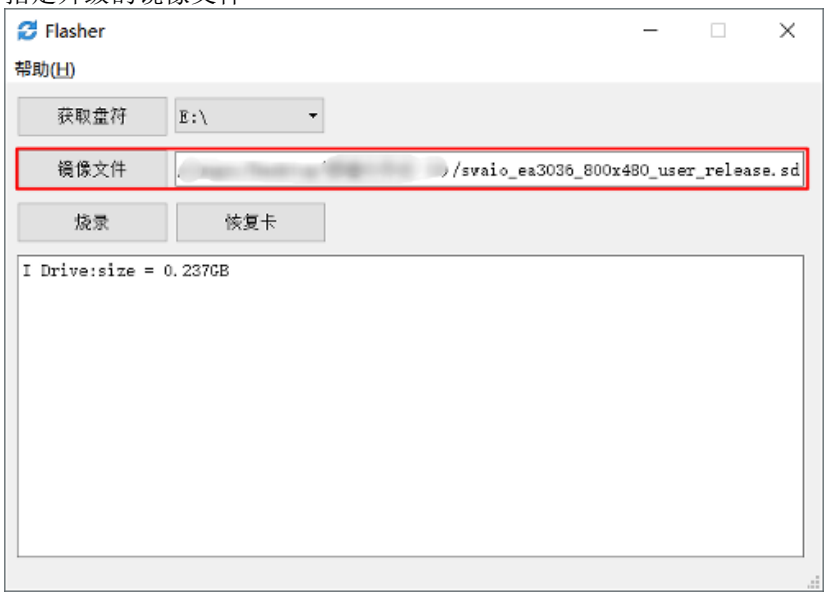
10.7.1 制作刷机卡步骤：

Flasher.zip

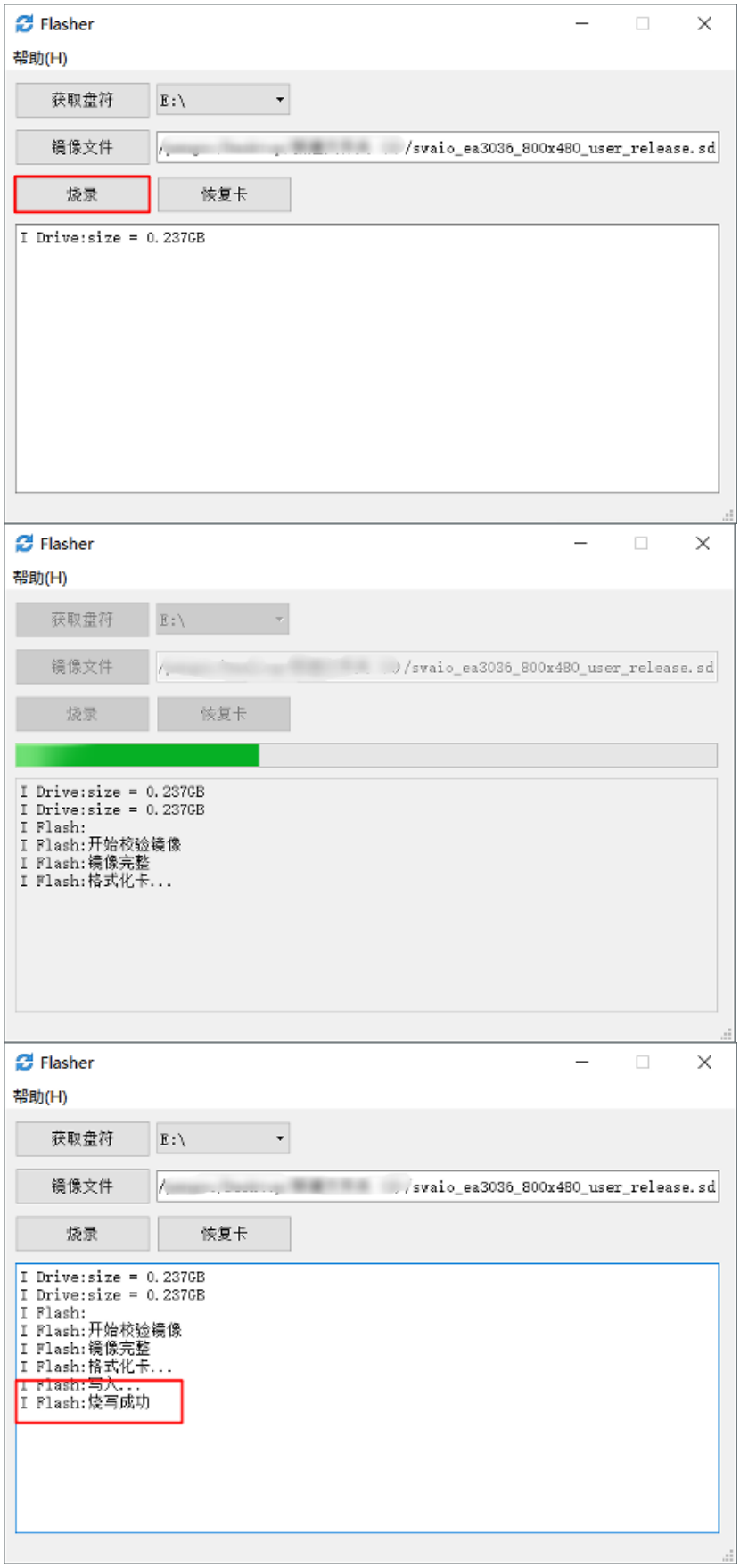
1. 获取电脑端刷机工具：
2. 选择电脑上 TF 卡盘符



3. 指定升级的镜像文件



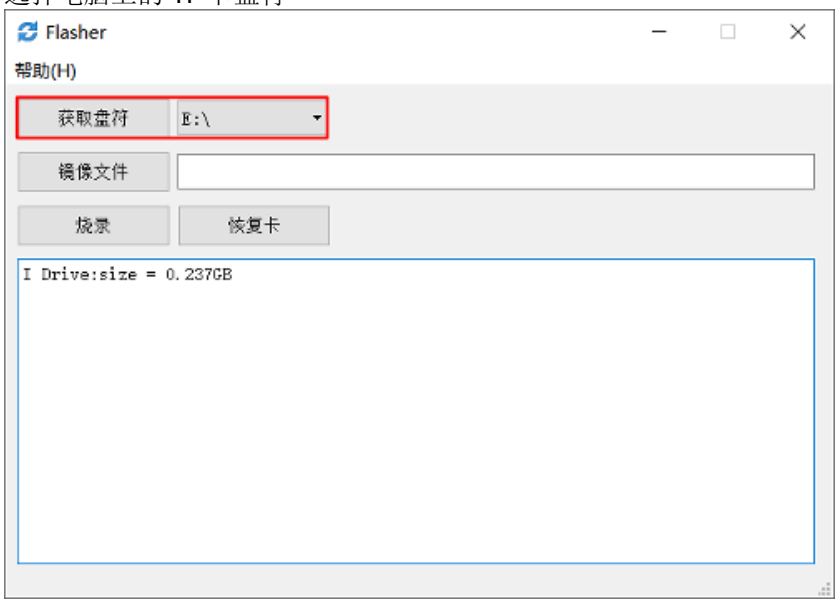
4. 点击烧录



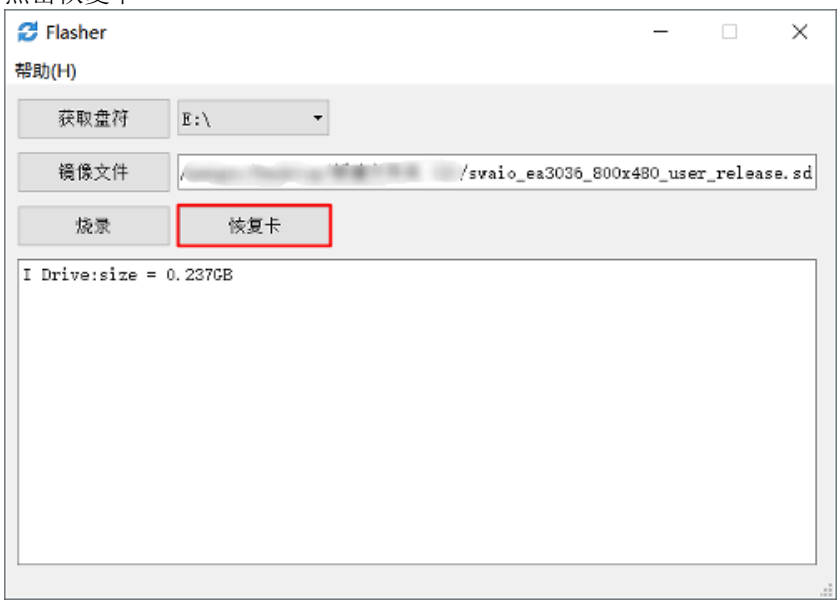
5. 少些成功后，拔出 TF 卡，擦汗如机器，再重新上电 3，即可进行升级
   注意：升级完成后，需要拔出 TF 卡，防止机器重启后反复升级。

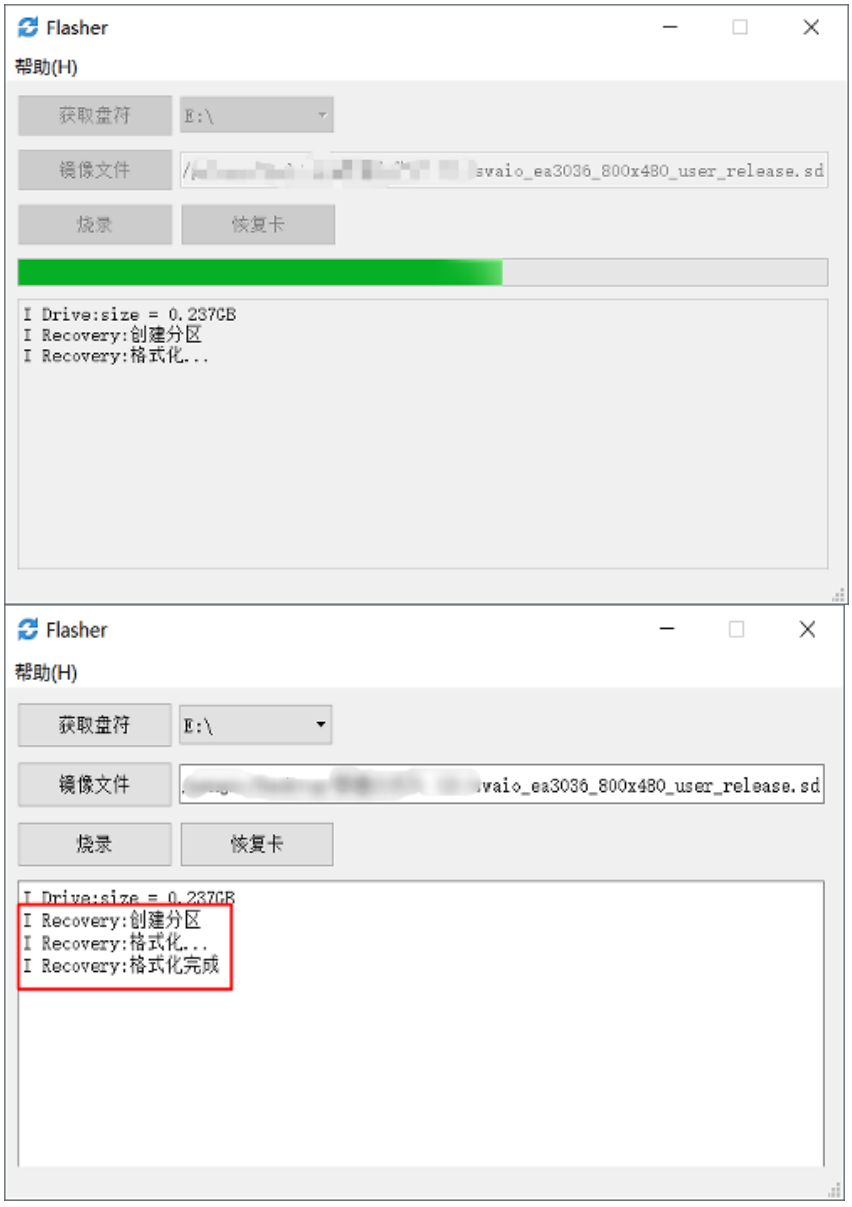
10.7.2 恢复卡步骤

1. 选择电脑上的 TF 卡盘符



2. 点击恢复卡





3. 格式化完成后，则卡恢复正常使用